

TMS320F28004x Microcontroller Workshop

Workshop Guide and Lab Manual



Kenneth W. Schachter
Revision 1.0
July 2019

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 © 2019 Texas Instruments Incorporated

Revision History

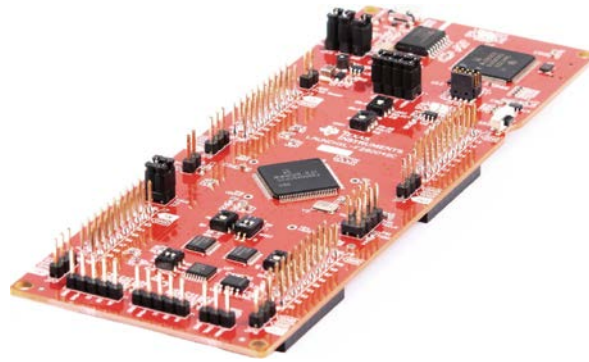
July 2019 – Revision 1.0

Mailing Address


Texas Instruments
C2000 Training Technical
13905 University Boulevard
Sugar Land, TX 77479

TMS320F28004x Microcontroller Workshop

TMS320F28004x Microcontroller Workshop



Texas Instruments
C2000 Technical Training

 TEXAS INSTRUMENTS C2000 is a trademark of Texas Instruments. Copyright © 2019 Texas Instruments. All rights reserved.

Workshop Outline

Workshop Outline

1. Architecture Overview
2. Programming Development Environment
 - Lab: Linker command file
3. Peripheral Register Programming
4. Reset and Interrupts
5. System Initialization
 - Lab: Watchdog and interrupts
6. Analog Subsystem
 - Lab: Build a data acquisition system
7. Control Peripherals
 - Lab: Generate and graph a PWM waveform
8. Direct Memory Access (DMA)
 - Lab: Use DMA to buffer ADC results
9. Control Law Accelerator (CLA)
 - Lab: Use CLA to filter PWM waveform
10. System Design
 - Lab: Run the code from flash memory
11. Communications (SCI echoback from C2000Ware)
12. Support Resources

Required Workshop Materials

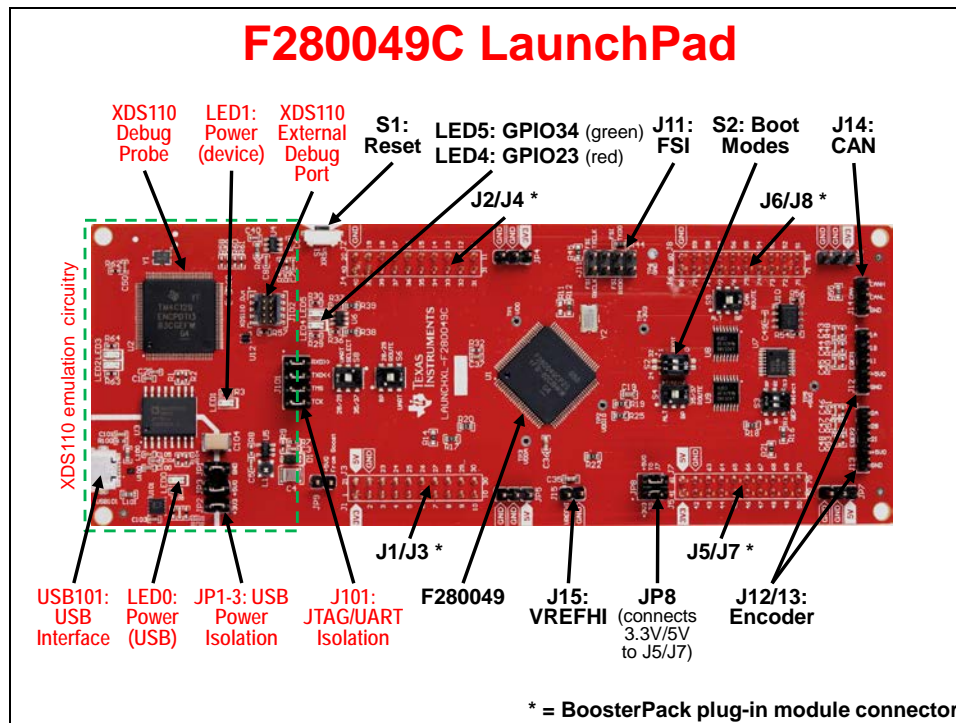
Required Workshop Materials

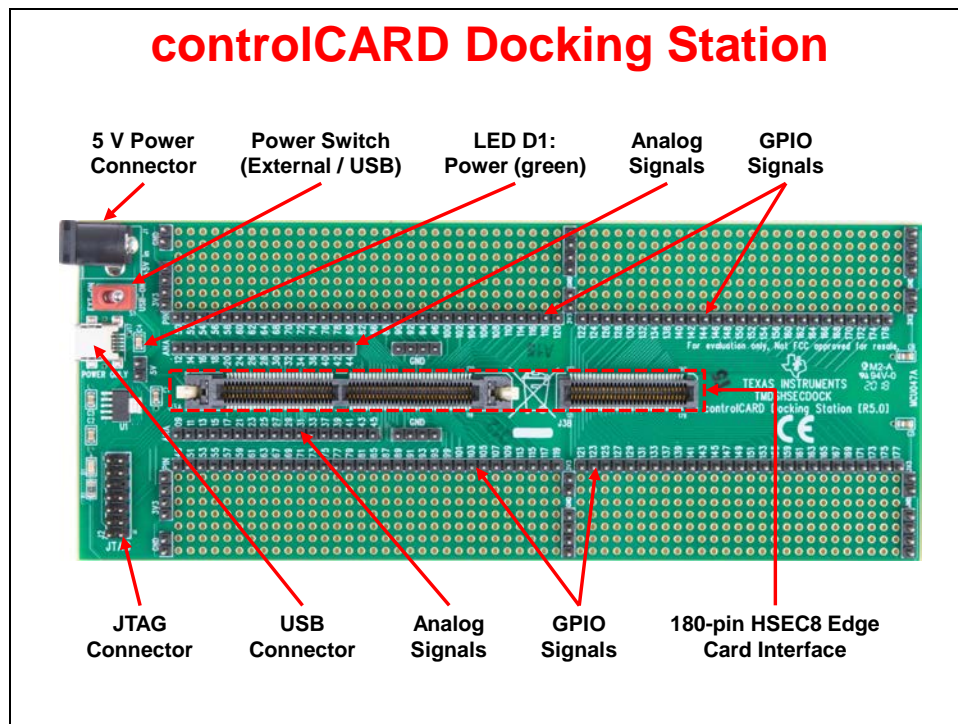
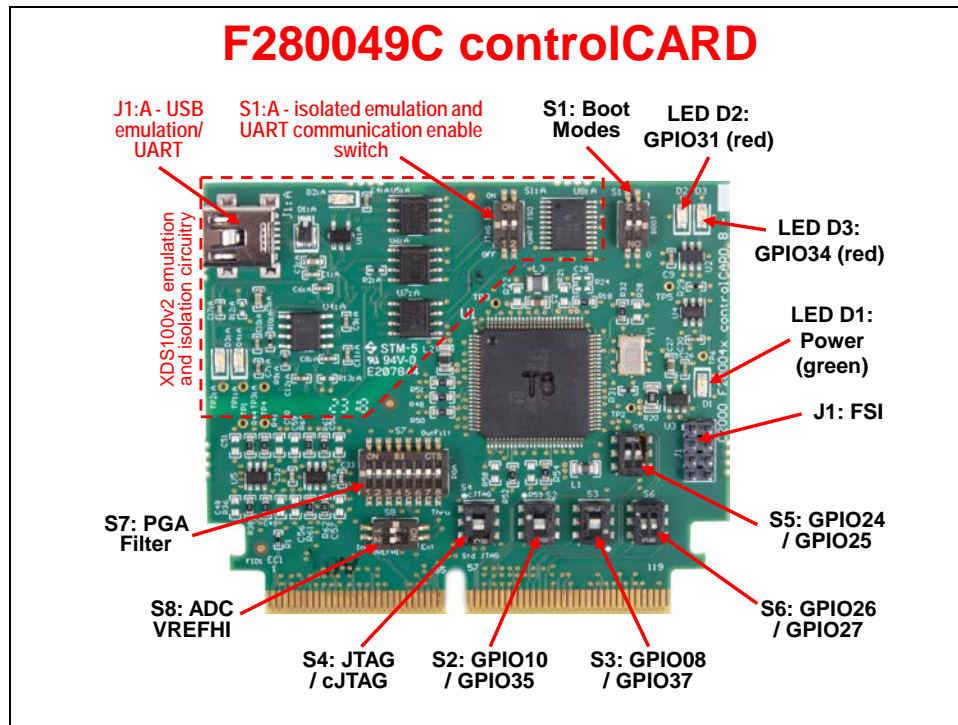
- ◆ <http://training.ti.com/c2000-f28004x-microcontroller-workshop>
- ◆ **F280049C LaunchPad** (LAUNCHXL-F280049C)
- ◆ **Install Code Composer Studio v9.0.1**
- ◆ **Run the workshop installer**

F28004x Microcontroller Workshop-1.0-Setup.exe

- ◆ Lab Files / Solution Files
- ◆ Workshop Manual

Development Tools





TMDSHSECDOCK is a baseboard that provides header pin access to key signals on compatible HSEC180-based controlCARDS. A breadboard area is available for rapid prototyping. Board power can be provided by either a USB cable or a 5V barrel power supply.

Architecture Overview

Introduction

This architectural overview introduces the basic architecture of the C2000™ family of microcontrollers from Texas Instruments. The F28004x series provides high performance processing for a variety of system control applications. The C2000 processors are ideal for applications combining digital signal processing, microcontroller processing, efficient C code execution, and operating system tasks.

Unless otherwise noted, the terms C28x and F28004x refer to the TMS320F28004x family of devices throughout the remainder of this workshop manual. For specific details and differences between device family members, please refer to the device data sheet, user's guides, and the technical reference manual.

Module Objectives

When this module is complete, you should have a basic understanding of the F28004x architecture and how all of its components work together to create a high-end, uniprocessor control system.

Module Objectives

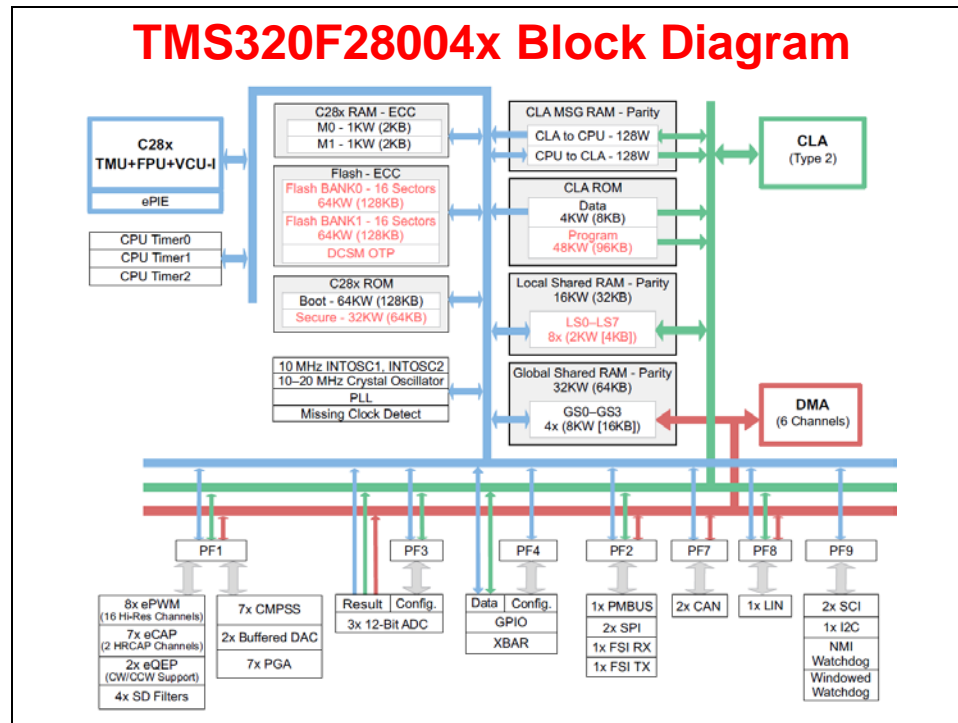
- ◆ **Review the F28004x block diagram and device features**
- ◆ **Describe the F28004x bus structure and memory map**
- ◆ **Identify the various memory blocks on the F28004x**
- ◆ **Identify the peripherals available on the F28004x**

Chapter Topics

Architecture Overview	1-1
<i>Introduction to the TMS320F28004x.....</i>	<i>1-3</i>
C28x Internal Bussing	1-4
C28x CPU + FPU + VCU + TMU and CLA	1-5
Special Instructions	1-6
CPU Pipeline	1-7
C28x CPU + FPU + VCU + TMU Pipeline	1-8
Memory	1-9
Memory Map	1-9
Dual Code Security Module (DCSM)	1-10
Peripherals	1-10
Fast Interrupt Response Manager	1-11
Math Accelerators	1-12
Viterbi / Complex Math Unit (VCU-II)	1-12
Trigonometric Math Unit (TMU).....	1-13
Configurable Logic Block (CLB).....	1-14
On-Chip Safety Features	1-15
Summary.....	1-16

Introduction to the TMS320F28004x

The TMS320F004x are device members of the C2000 microcontroller (MCU) product family. These devices are most commonly used within embedded control applications. Even though the topics presented in this workshop are based on the TMS320F28004x device family, most all of the topics are fully applicable to other C2000 MCU product family members. The F28004x MCU utilizes the TI 32-bit C28x CPU architecture. The MCU has access to a set of highly integrated analog and control peripherals, which provides a complete solution for demanding real-time high-performance signal processing applications, such as digital power, industrial drives, inverters, and motor control.



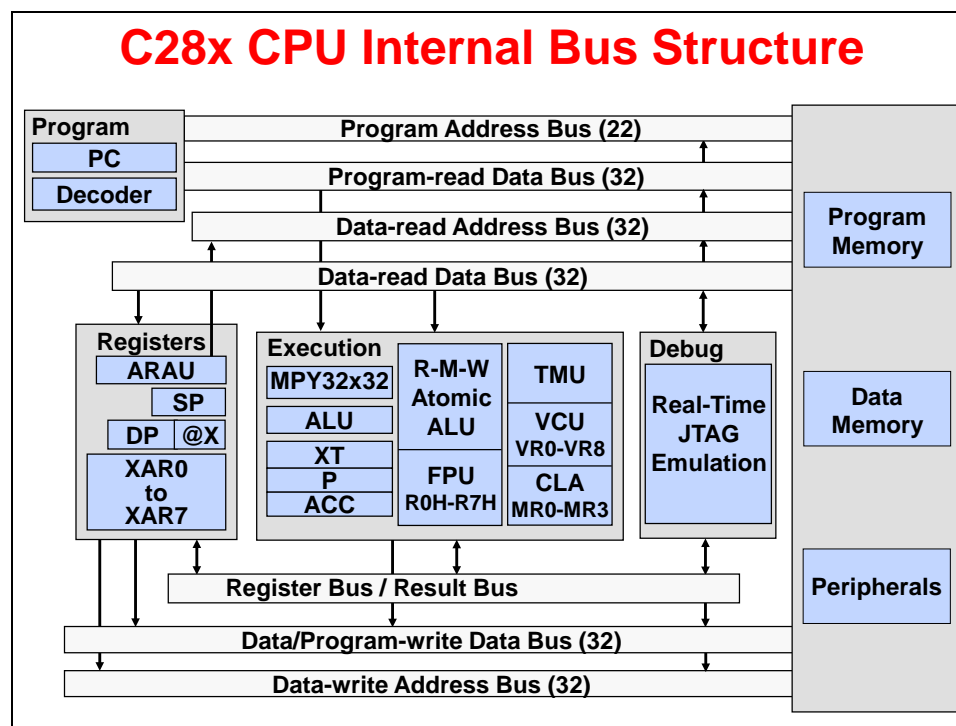
The above block diagram represents an overview of the device features; however refer to the data sheet for details about a specific device family member. The F28004x CPU is based on the TI 32-bit C28x fixed-point accumulator-based architecture and it is capable of operating at a clock frequency of up to 100 MHz. The CPU is tightly coupled with a Floating-Point Unit (FPU) which enables support for hardware IEEE-754 single-precision floating-point format operations. Also, a tightly coupled Trigonometric Math Unit (TMU) extends the capability of the CPU to efficiently execute trigonometric and arithmetic operations commonly found in control system applications. Similar to the FPU, the TMU provides hardware support for IEEE-754 single-precision floating-point operations which accelerate trigonometric math functions. A Viterbi, Complex Math, and CRC Unit (VCU) further extends the capabilities of the CPU for supporting various communication-based algorithms and is very useful for general-purpose signal processing applications, such as filtering and spectral analysis.

The Control Law Accelerator (CLA) is an independent 32-bit floating-point math hardware accelerator which executes real-time control algorithms in parallel with the main C28x CPU, effectively doubling the computational performance. With direct access to the various control and communication peripherals, the CLA minimizes latency, enables a fast trigger response, and avoids CPU overhead. Also, with direct access to the ADC results registers, the CLA is able to read the result on the same cycle that the ADC sample conversion is completed, providing “just-in-time” reading, which reduces the sample to output delay.

C28x Internal Bussing

As with many high performance microcontrollers, multiple busses are used to move data between the memory blocks, peripherals, and the CPU. The C28x memory bus architecture consists of six buses (three address and three data):

- 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)

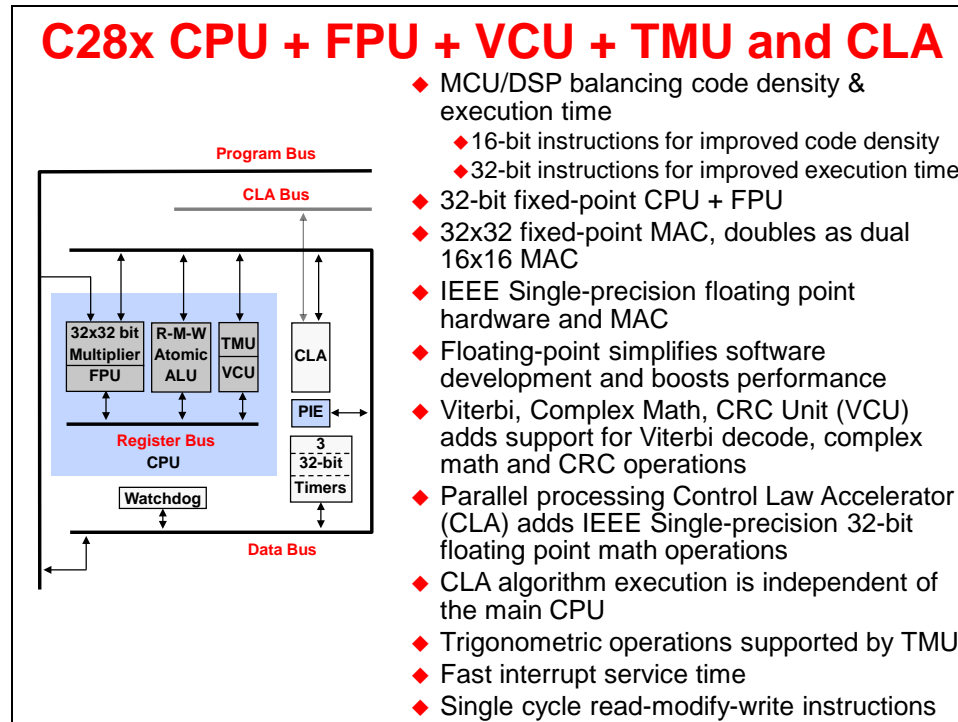


The 32-bit-wide data busses provide single cycle 32-bit operations. This multiple bus architecture (Harvard Bus Architecture) enables the C28x to fetch an instruction, read a data value and write a data value in a single cycle. All peripherals and memory blocks are attached to the memory bus with prioritized memory accesses.

C28x CPU + FPU + VCU + TMU and CLA

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

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

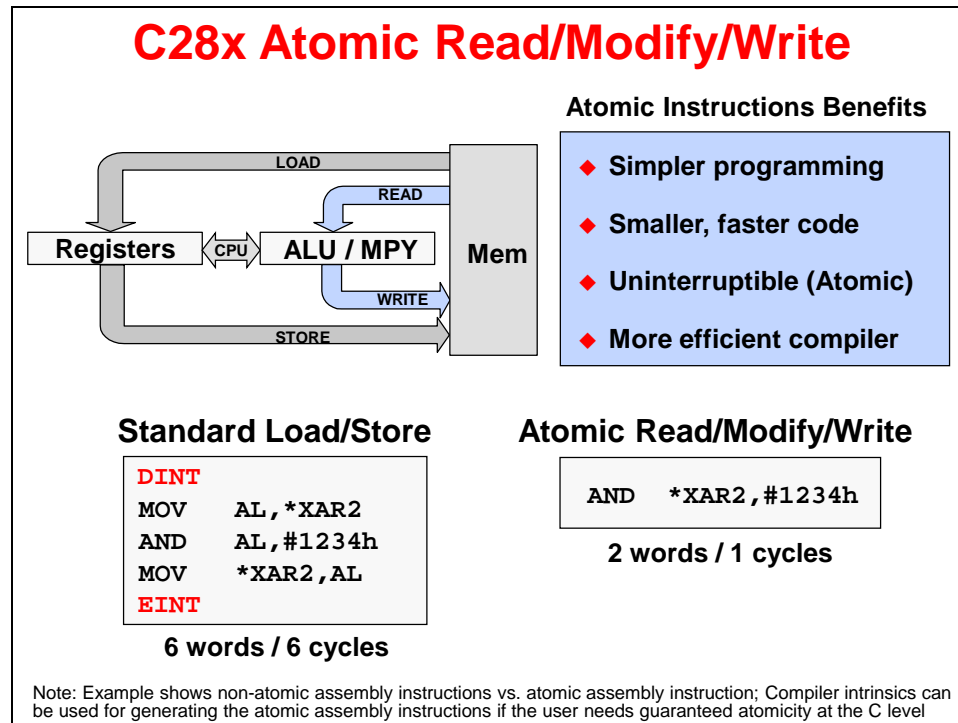


The C28x design supports an efficient C engine with hardware that allows the C compiler to generate compact code. Multiple busses and an internal register bus allow an efficient and flexible way to operate on the data. The architecture is also supported by powerful addressing modes, which allow the compiler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

The C28x is as efficient in DSP math tasks as it is in system control tasks. This efficiency removes the need for a second processor in many systems. The 32 x 32-bit multiply-accumulate (MAC) capabilities can also support 64-bit processing, enable the C28x to efficiently handle higher numerical resolution calculations 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). The devices also feature floating-point units.

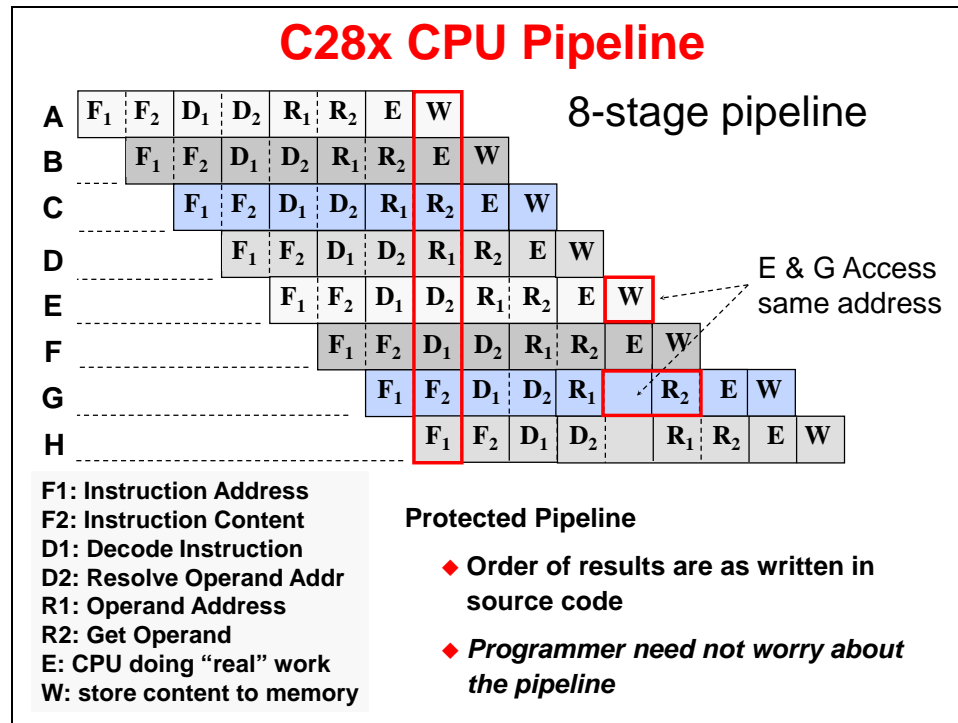
The addition of the Floating-Point Unit (FPU) to the fixed-point CPU core enables support for hardware IEEE-754 single-precision floating-point format operations. The FPU adds an extended set of floating-point registers and instructions to the standard C28x architecture, providing seamless integration of floating-point hardware into the CPU.

Special Instructions



Atomic instructions are a group of small common instructions which 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.

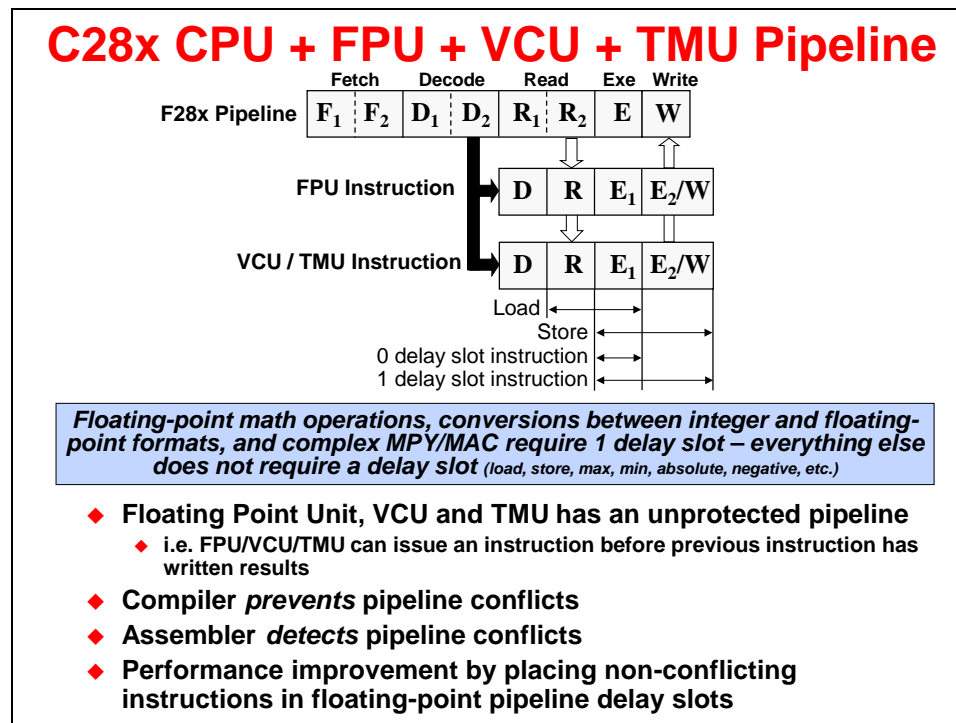
CPU Pipeline



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

This pipelining also enables the C28x CPU 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. With the 8-stage pipeline most operations can be performed in a single cycle.

C28x CPU + FPU + VCU + TMU Pipeline



Floating-point unit (FPU), VCU and TMU 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/TMU pipeline are:

Math	MPYF32, ADDF32, SUBF32, MACF32, VCOMPY	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

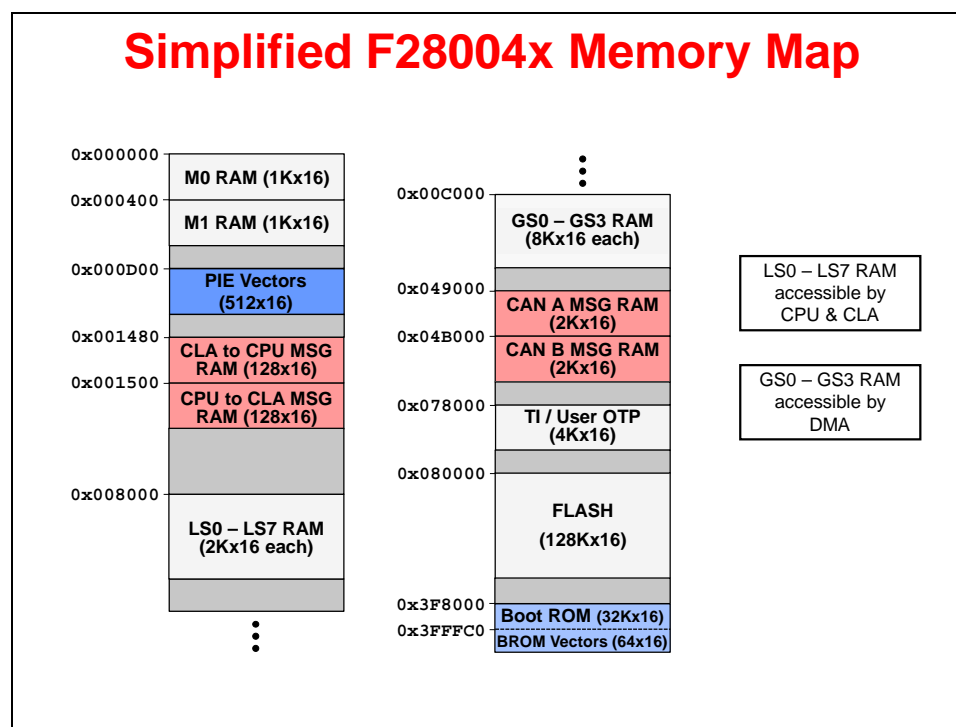
* Note: MOV32 between FPU and CPU registers is a special case.

Memory

The F28004x utilizes a memory map where the unified memory blocks can be accessed in either program space, data space, or both spaces. This type of memory map lends itself well for supporting high-level programming languages. The memory map structure consists of RAM blocks dedicated to the CPU, RAM blocks accessible by the CPU and CLA, RAM blocks accessible by the DMA module, message RAM blocks between the CPU and CLA, CAN message RAM blocks, flash, and one-time programmable (OTP) memory. The Boot ROM is factory programmed with boot software routines and standard tables used in math related algorithms.

Memory Map

The C28x CPU core contains no memory, but can access on-chip and off-chip memory. The C28x uses 32-bit data addresses and 22-bit program addresses. This allows for a total address reach of 4G words (1 word = 16-bits) in data memory and 4M words in program memory.



There are two dedicated RAM blocks (M0 and M1) which are tightly coupled with the CPU, and only the CPU has access to them. The PIE Vectors are a special memory area containing the vectors for the peripheral interrupts. The eight local shared memory blocks, LS0 through LS7, are accessible by the CPU and CLA. The four global shared memory blocks, GS0 through GS3, are accessible by the CPU and DMA.

There are two types of message RAM blocks: CLA message RAM blocks and CAN message RAM blocks. The CLA message RAM blocks are used to share data between the CPU and CLA. The CAN message RAM blocks contain message objects and parity bits for the message objects (CAN message RAM can only be accessed in debug mode).

The user OTP is a one-time, programmable, memory block which contains device specific calibration data for the ADC, internal oscillators, and buffered DACs, in addition to settings used

by the flash state machine for erase and program operations. Additionally, it contains locations for programming security settings, such as passwords for selectively securing memory blocks, configuring the standalone boot process, as well as selecting the boot-mode pins in case the factory-default pins cannot be used. This information is programmed into the dual code security module (DCSM). The flash memory is primarily used to store program code, but can also be used to store static data. The boot ROM and boot ROM vectors are located at the bottom of the memory map.

Dual Code Security Module (DCSM)

Dual Code Security Module

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

Z1_CSMPSWD0	Z2_CSMPSWD0
Z1_CSMPSWD1	Z2_CSMPSWD1
Z1_CSMPSWD2	Z2_CSMPSWD2
Z1_CSMPSWD3	Z2_CSMPSWD3

- ◆ **Various on-chip memory resources can be assigned to either zone 1 or zone 2**
- ◆ **Each zone has its own password**
- ◆ **128-bit user defined password is stored in OTP**
- ◆ **128-bits = $2^{128} = 3.4 \times 10^{38}$ possible passwords**
- ◆ **To try 1 password every 8 cycles at 100 MHz, it would take at least 8.6×10^{23} years to try all possible combinations!**

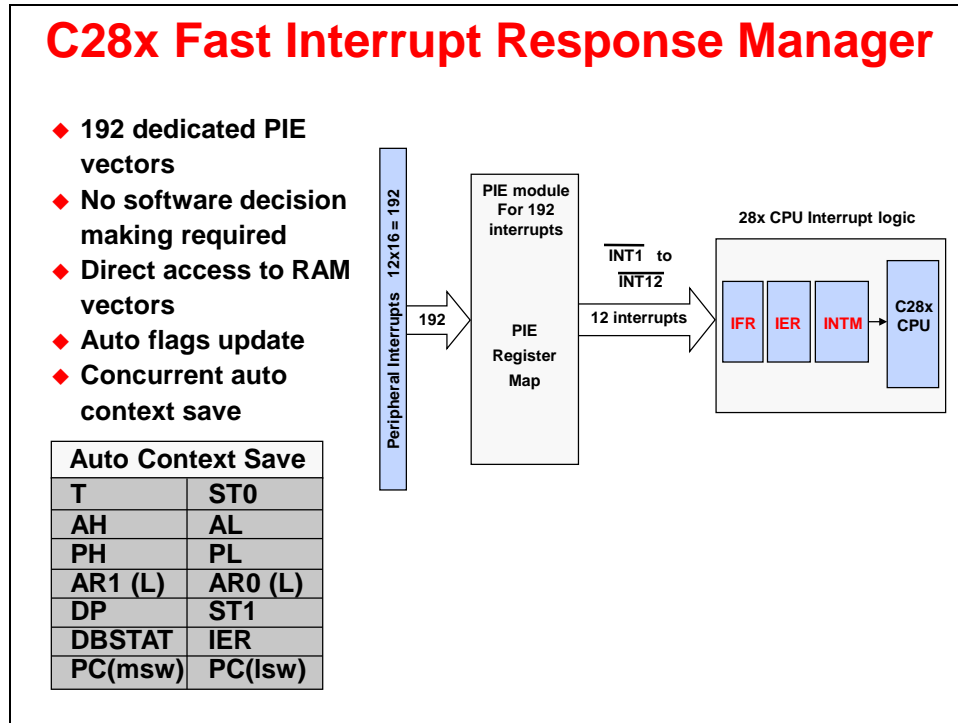
Peripherals

The F28004x is available with a variety of built in peripherals optimized to support control applications. See the data sheet for specific availability.

- ePWM
- eCAP
- eQEP
- CMPSS
- ADC
- DAC
- PGA
- Watchdog
- DMA
- CLA
- SDFM
- SPI
- SCI
- I2C
- LIN
- CAN
- FSI
- PMBUS

Fast Interrupt Response Manager

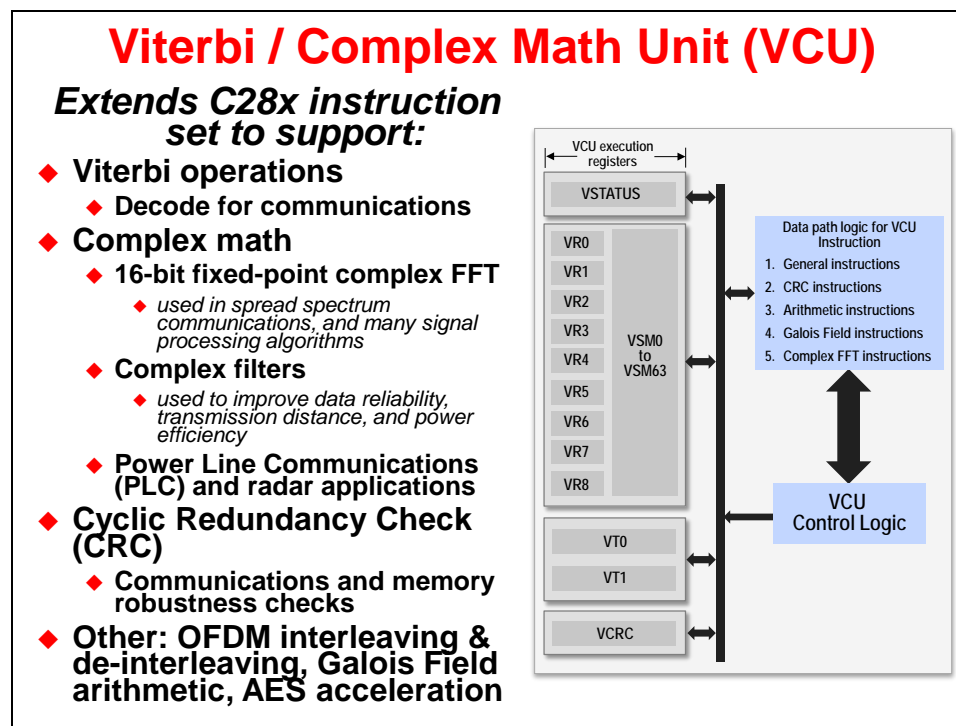
The fast interrupt response manager is capable of automatically performing a context save of critical registers. This results in the ability of servicing many asynchronous events with minimal latency. The F28004x context saves and restores 14 registers with a zero cycle penalty. This feature helps reduce the interrupt service routine overhead.



By incorporating the very fast interrupt response manager with the peripheral interrupt expansion (PIE) block, it is possible to allow up to 192 interrupt vectors to be processed by the CPU. More details about this will be covered in the reset and interrupts, and system initialization modules.

Math Accelerators

Viterbi / Complex Math Unit (VCU-II)

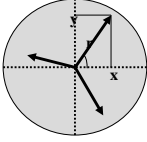
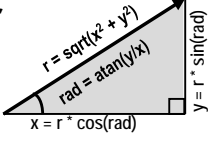


The Viterbi, Complex Math, and CRC Unit (VCU) adds an extended set of registers and instructions to the standard C28x architecture for supporting various communications-based algorithms, such as power line communications (PLC) standards PRIME and G3. These algorithms typically require Viterbi decoding, complex Fast Fourier Transform (FFT), complex filters, and cyclical redundancy check (CRC). By utilizing the VCU a significant performance benefit is realized over a software implementation. It performs fixed-point operations using the existing instruction set format, pipeline, and memory bus architecture. Additionally, the VCU is very useful for general-purpose signal processing applications such as filtering and spectral analysis.

Trigonometric Math Unit (TMU)

Trigonometric Math Unit (TMU)

Adds instructions to FPU for calculating common trigonometric operations

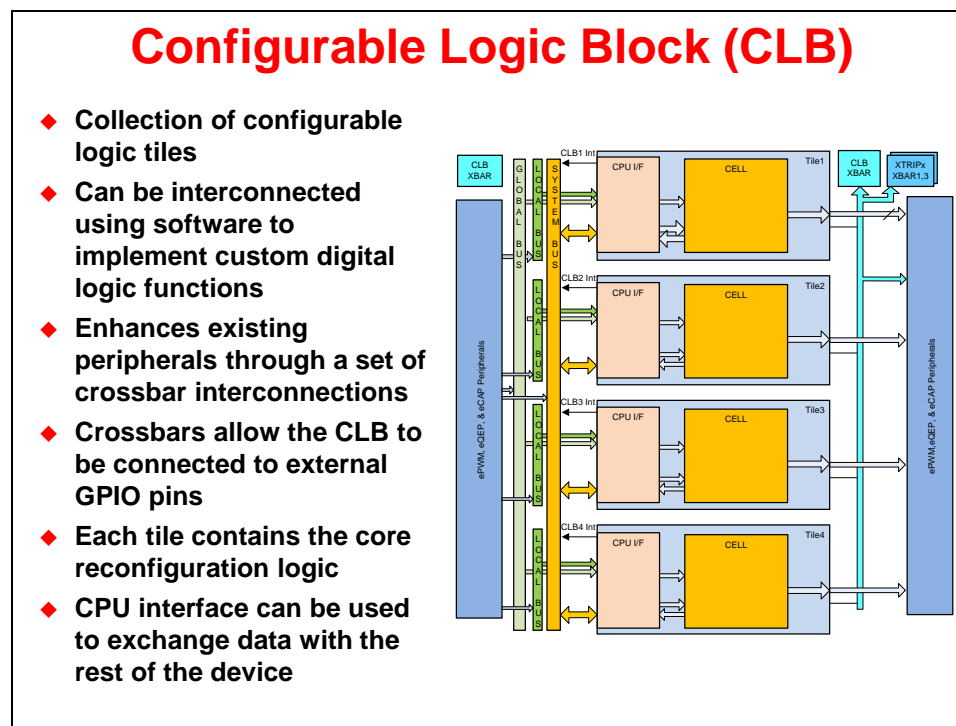



Operation	Instruction		Exe Cycles	Result Latency	FPU Cycles w/o TMU
$Z = Y/X$	DIVF32	Rz, Ry, Rx	1	5	-24
$Y = \text{sqrt}(X)$	SQRTF32	Ry, Rx	1	5	-26
$Y = \sin(X/2\pi)$	SINPUF32	Ry, Rx	1	4	-33
$Y = \cos(X/2\pi)$	COSPUF32	Ry, Rx	1	4	-33
$Y = \text{atan}(X)/2\pi$	ATANPUF32	Ry, Rx	1	4	-53
Instruction To	QUADF32	Rw, Rz, Ry, Rx	3	11	-90
Support ATAN2	ATANPUF32	Ra, Rz			
Calculation	ADDF32	Rb, Ra, Rw			
$Y = X * 2\pi$	MPY2PIF32	Ry, Rx	1	2	-4
$Y = X * 1/2\pi$	DIV2PIF32	Ry, Rx	1	2	-4

- ◆ **Supported by natural C and C-intrinsics**
- ◆ **Significant performance impact on algorithms such as:**
 - Park / Inverse Park
 - dq0 Transform & Inverse dq0
 - Space Vector GEN
 - FFT Magnitude & Phase Calculations

The Trigonometric Math Unit (TMU) is an extension of the FPU and the C28x instruction set, and it efficiently executes trigonometric and arithmetic operations commonly found in control system applications. Similar to the FPU, the TMU provides hardware support for IEEE-754 single-precision floating-point operations that are specifically focused on trigonometric math functions. Seamless code integration is accomplished by built-in compiler support that automatically generates TMU instructions where applicable. This dramatically increases the performance of trigonometric functions, which would otherwise be very cycle intensive. It uses the same pipeline, memory bus architecture, and FPU registers as the FPU, thereby removing any special requirements for interrupt context save or restore.

Configurable Logic Block (CLB)



The Configurable Logic Block (CLB) is configured by software and it allows for the implementation of custom digital logic functions. Enhancements to the existing peripherals are enabled through a set of crossbar interconnections. This provides a high level of connectivity to existing control peripherals such as the enhanced pulse width modulators (ePWM), the enhanced capture modules (eCAP), and the enhanced quadrature encoder pulse modules (eQEP). The crossbars also allow the CLB to be connected to external GPIO pins. Therefore, the CLB can be configured to interact with device peripherals to perform small logical functions such as simple PWM generators, or to implement custom serial data exchange protocols.

The CLB subsystem contains a number of identical tiles. There are four tiles in the F28004x CLB subsystems. Each tile contains combinational and sequential logic blocks, as well as other dedicated hardware. The figure above shows the structure of the CLB subsystem in the F28004x device.

On-Chip Safety Features

On-Chip Safety Features

- ◆ **Memory Protection**
 - ◆ ECC and parity enabled RAMs, shared RAMs protection
 - ◆ ECC enabled flash memory
- ◆ **Clock Checks**
 - ◆ Missing clock detection logic
 - ◆ PLLSLIP detection
 - ◆ NMIWDs
 - ◆ Windowed watchdog
- ◆ **Write Register Protection**
 - ◆ LOCK protection on system configuration registers
 - ◆ EALLOW protection
 - ◆ PIE vector address validity check
- ◆ **Annunciation**
 - ◆ Single error pin for external signaling of error

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)
- ◆ Trigonometric math unit (TMU)
- ◆ Atomic read-modify-write instructions
- ◆ Fast interrupt response manager
- ◆ 128Kw on-chip flash memory
- ◆ Dual code security module (DCSM)
- ◆ Control peripherals
- ◆ Analog peripherals
- ◆ Direct memory access (DMA)
- ◆ 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**

Chapter Topics

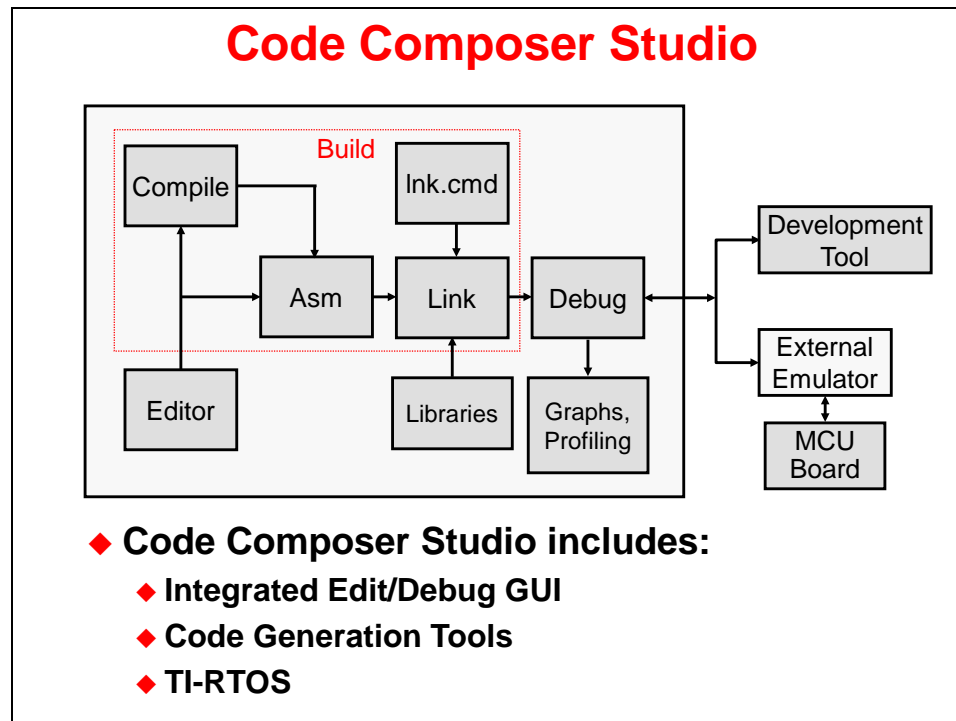
Programming Development Environment.....	2-1
<i>Code Composer Studio.....</i>	<i>2-3</i>
Software Development and COFF Concepts.....	2-3
Code Composer Studio.....	2-4
Edit and Debug Perspective (CCSv9).....	2-5
Target Configuration	2-6
CCSv9 Project.....	2-7
Creating a New CCSv9 Project.....	2-8
CCSv9 Build Options – Compiler / Linker	2-9
CCS 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 2: Linker Command File.....</i>	<i>2-18</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.

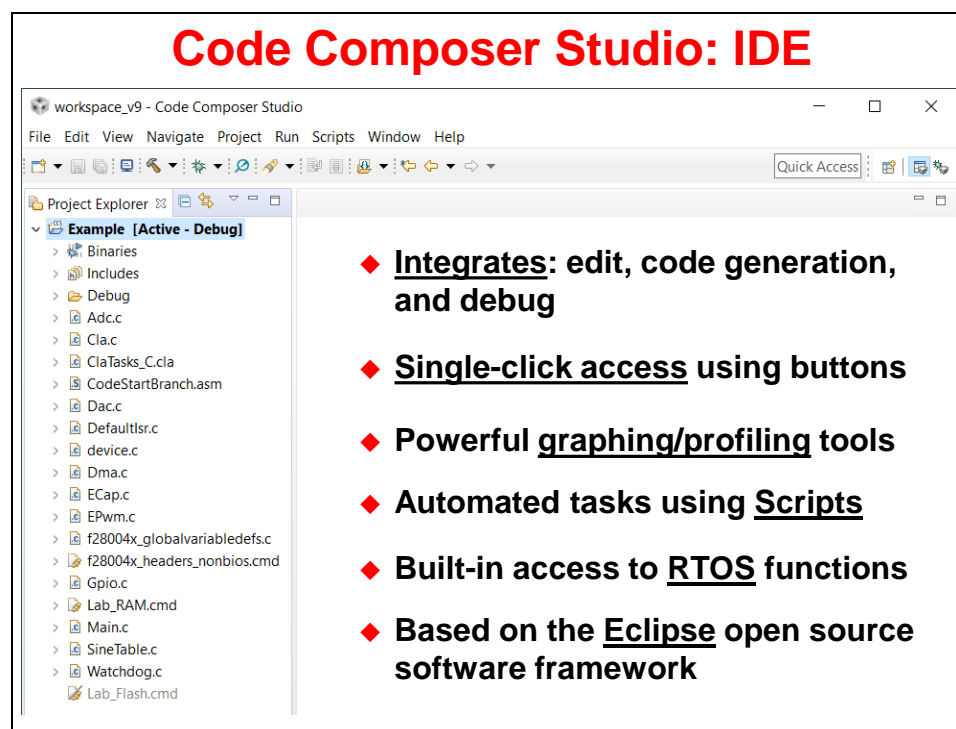
The high level of modularity and portability resulting from this system simplifies the processes of verification, debug and maintenance. The process of COFF development is presented in greater detail in the following paragraphs.

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™ (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.

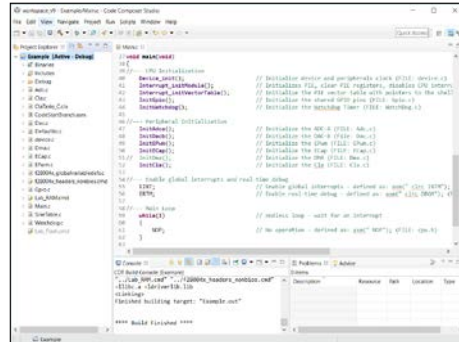
CCS is based on the Eclipse open source software framework. The Eclipse software framework was originally developed as an open framework for creating development tools. Eclipse offers an excellent software framework for building software development environments and it is becoming a standard framework 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 (CCSv9)

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.

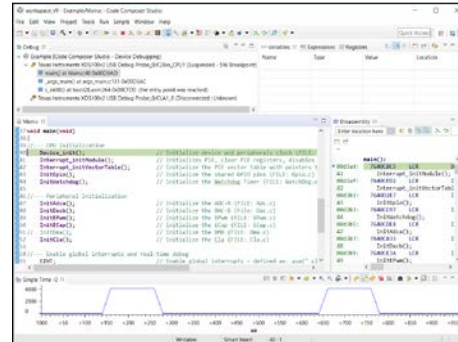
Edit and Debug Perspective (CCSv9)

- ◆ 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

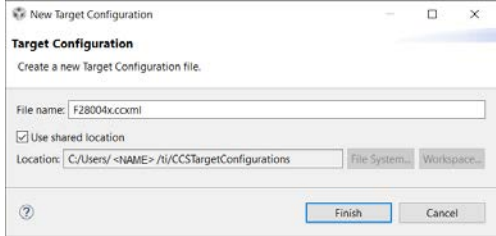
- ◆ Displays views used for debugging
 - ◆ Menus and toolbars associated with debugging, watch and memory windows, graphs, etc.

By default, 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.

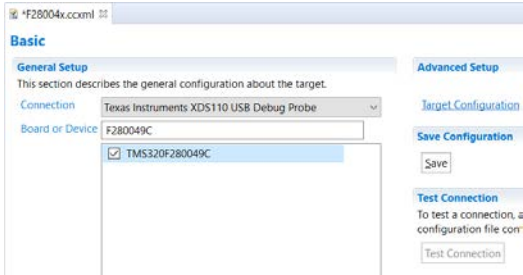
Target Configuration

A Target Configuration defines how CCS connects 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**

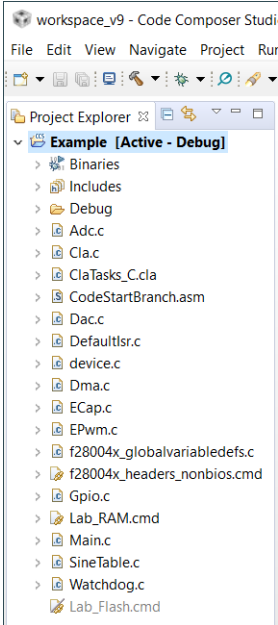


◆ **Select connection type**
◆ **Select device**
◆ **Save configuration**

CCSv9 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.

CCSv9 Project



The screenshot shows the Project Explorer window in Code Composer Studio. The project is named 'Example [Active - Debug]'. The file tree includes folders for Binaries, Includes, and Debug. Source files include Adc.c, Cla.c, ClaTasks_C.cla, CodeStartBranch.asm, Dac.c, DefaultIsr.c, device.c, Dma.c, ECap.c, EPwm.c, f28004x_globalvariabledefs.c, f28004x_headers_nonbios.cmd, Gpio.c, Lab_RAM.cmd, Main.c, SineTable.c, Watchdog.c, and Lab_Flash.cmd.

Project files contain:

- ◆ **List of files:**
 - ◆ Source (C, assembly)
 - ◆ Libraries
 - ◆ Linker command files
 - ◆ TI-RTOS configuration file
- ◆ **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 TI-RTOS, as well as build configurations.

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

File → New → CCS Project

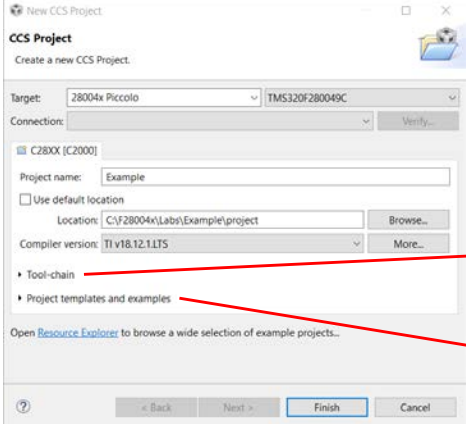
Along with the main Project menu, you can also manage open projects using the right-click popup menu. Either of these menus allows you to modify a project, such as add files to a project, or open the properties of a project to set the build options.

Creating a New CCSv9 Project

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

Creating a New CCSv9 Project

1. Project Name, Location, and Device



◆ **File → New → CCS Project**

2. Tool-chain

- Tool-chain
 - Output type: Executable
 - Output format: legacy COFF
 - Device endianness: little
 - Linker command file: **none**
 - Runtime support library: <automatic>

3. Project templates and examples

- Project templates and examples
 - type filter text
 - Empty Projects
 - Empty Project
 - Empty Project (with main.c)
 - Empty Assembly-only Project
 - Empty RTSC Project

Creates an empty project initialized for the selected device.

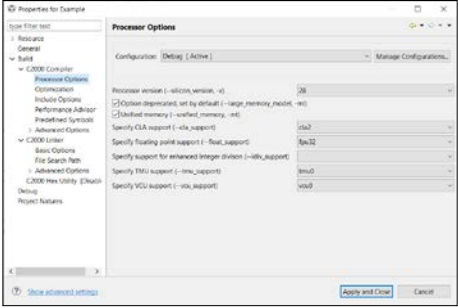
After a project is created, the build options are configured.

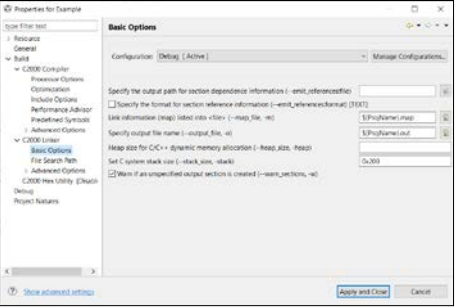
CCSv9 Build Options – Compiler / Linker

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.

CCSv9 Build Options – Compiler / Linker





◆ **Compiler**

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

◆ **Linker**

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

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

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

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

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





CCS Debug Environment

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



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.	T1 Debug Toolbar Target Menu Debug View Context Menu
	Terminate All	Terminates all active debug sessions.	Target Menu Debug View Toolbar

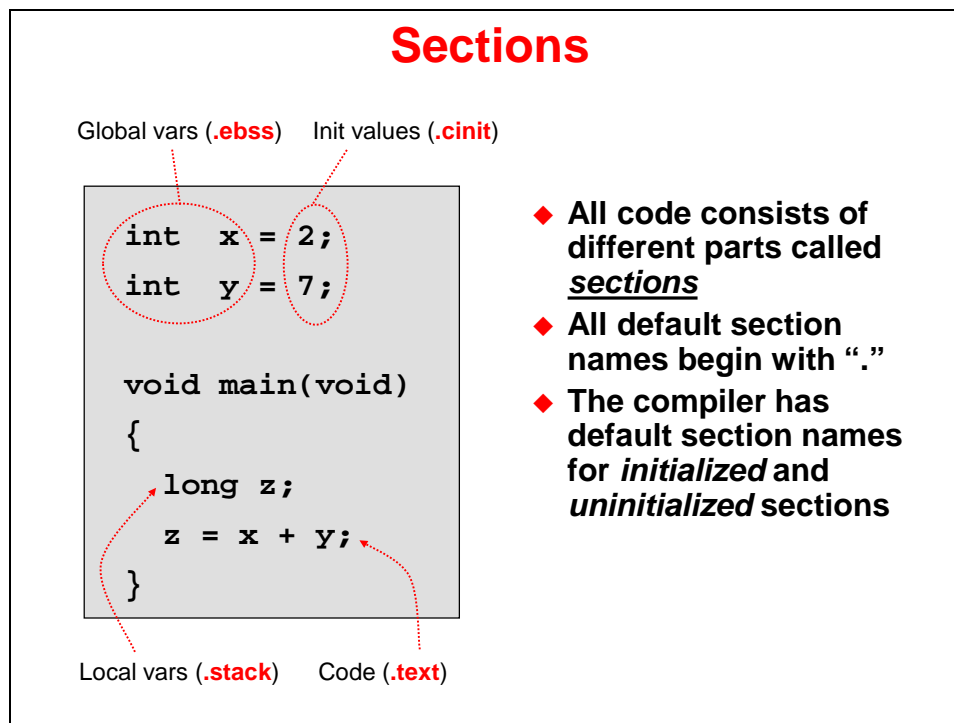
Program execution

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`.



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)

The 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.

Compiler Section Names		
Initialized Sections		
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
Uninitialized Sections		
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.

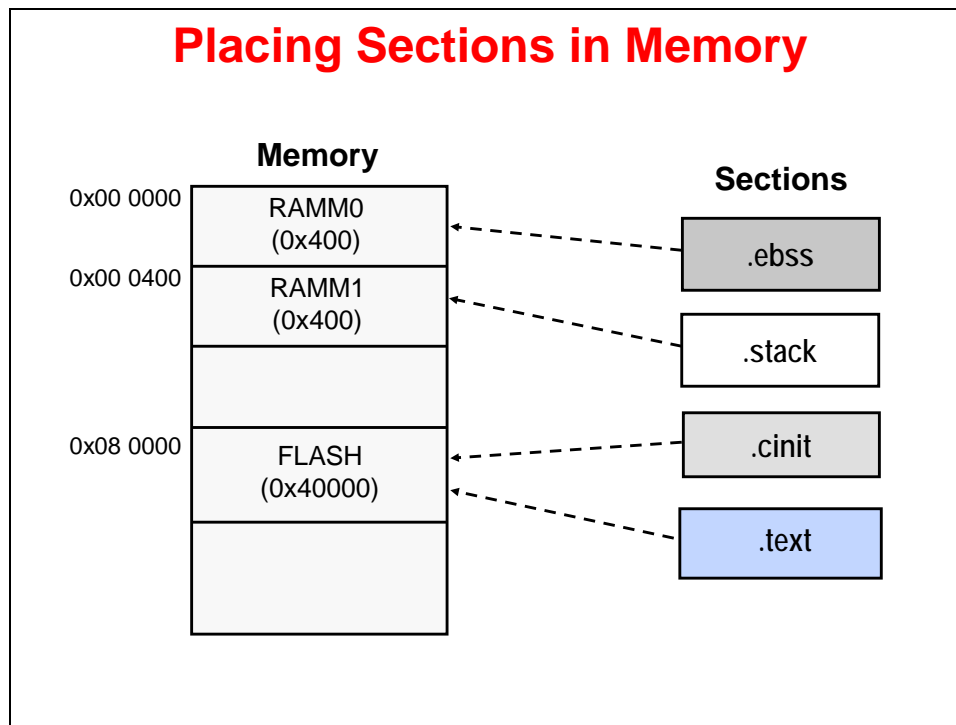
Constants (.cinit – initialized data)

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.

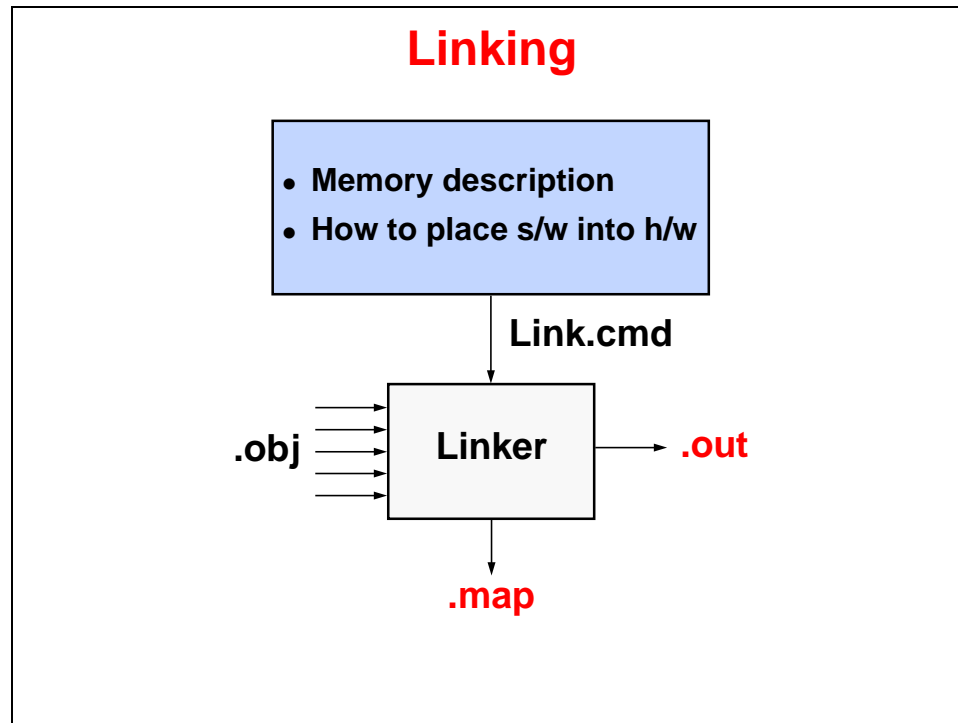


Linking code is a three step process:

1. Defining the various regions of memory (on-chip RAM 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 256Kw FLASH starting at memory location 0x080000, it would read:

```
MEMORY
{
  FLASH:  origin = 0x080000 , length = 0x040000
}
```

Each memory segment is defined using the above format. If you added RAMM0 and RAMM1, it would look like:

```
MEMORY
{
  RAMM0:  origin = 0x000000 , length = 0x0400
  RAMM1:  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 = 0x080000, length = 0x40000

    PAGE 1:          /* Data Memory */
    RAMM0:          origin = 0x000000, length = 0x400
    RAMM1:          origin = 0x000400, length = 0x400
}
SECTIONS
{
    .text:>          FLASH          PAGE = 0
    .ebss:>          RAMM0          PAGE = 1
    .cinit:>         FLASH          PAGE = 0
    .stack:>         RAMM1          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:>  RAMM0          PAGE 1
    .cinit:> FLASH          PAGE 0
    .stack:> RAMM1          PAGE 1
}
    
```

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

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

Summary: Linker Command File

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

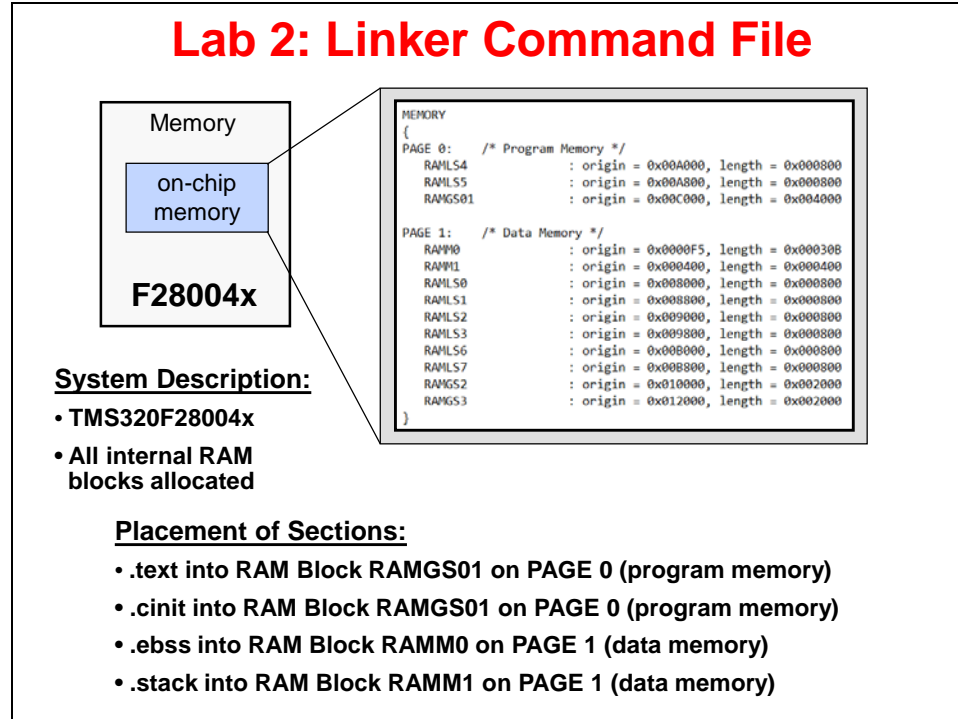
Linker Command File Summary

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

Lab 2: Linker Command File

➤ Objective

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



➤ Initial Hardware Set Up

Note: The lab exercises in this workshop have been developed and targeted for the F280049C LaunchPad. Optionally, the F280049C Experimenter Kit can be used. Refer to Appendix A for additional information on using the F280049C Experimenter Kit with this workshop.

- **F280049C LaunchPad:**

Using the supplied USB cable – plug the USB Standard Type A connector into the computer USB port and the USB Micro Type B connector into the LaunchPad. This will power the LaunchPad 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 *C2000Ware* 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 `Launch`.

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 C/C++ projects. A CCS Debug perspective view will automatically be enabled when the debug session is started. This perspective is used for debugging C/C++ projects.

Setup Target Configuration

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

`File` → `New` → `Target Configuration File`

In the file name field type **F28004x.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 XDS110 USB Debug Probe”. In the “Board or Device” box type **TMS320F280049C** to filter the options. In the box below, check the box to select “TMS320F280049C”.

The LaunchPad XDS110 USB Debug Probe is only wired to support 2-pin cJTAG mode. Under Advanced Setup click “Target Configuration” and highlight “Texas Instruments XDS110 USB Debug Probe_0”. Under Connection Properties set the JTAG/SWD/cJTAG Mode to “cJTAG (1149.7) 2-pin advanced modes”.

Click `Save` to save the configuration, then close the “F28004x.ccxml” setup window by clicking the `x` on the tab.

5. To view the target configurations, click:

`View` → `Target Configurations`

and click the sign (`+` or `>`) to the left of “User Defined”. Notice that the F28004x.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 or click: Project → 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 “28004x Piccolo”. In the pull-down list immediately to the right, choose the “TMS320F280049C”.

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

7. The next 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:\F28004x\Labs\Lab2\project

Click Select Folder.

8. Next, open the “Tool-chain” 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. If the workbench is empty, reset the perspective view by clicking:

Window → Perspective → Reset Perspective...

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 are looking in C:\F28004x\Labs\Lab2\source. 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 sign (+ or >) 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 0x200.

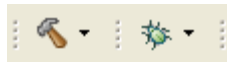
15. Under “C2000 Compiler” select the “Processor Options”. Notice the large memory model and unified memory boxes are checked. Next, notice the “Specify CLA support” is set to `cla2`, the “Specify floating point support” is set to `fpu32`, the “Specify TMU support” is set to `TMU0`, and the “Specify VCU support” is set to `vcu0`. Select `Apply` and `Close` 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 `RAMLS4`, `RAMLS5` and `RAMGS01` 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 sign (+’ or >’) 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 red circle with a white “x” inside of it.
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 CCS. 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 `<enter>`. 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 “16-Bit Hex – TI Style” 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 Code Composer Studio 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 Register Programming

Introduction

This module starts with exploring different types of programming models; which include the traditional `#define` macro approach, the bit field structure header files approach, and the driver library approach. In this workshop, the C2000 Peripheral Driver Library, or Driverlib, will be used. Driverlib is a set of low-level drivers for configuring memory-mapped peripheral registers. The Driverlib provides a more readable and portable approach to peripheral register programming than the other programming model methods.

The Driverlib is written in C and all source code can be found within C2000Ware. It provides drivers for all peripherals, as well as drivers for configuring various memory mapped device settings. In this module, you will learn how to use the Driverlib to facilitate programming the peripherals and the device.

Module Objectives

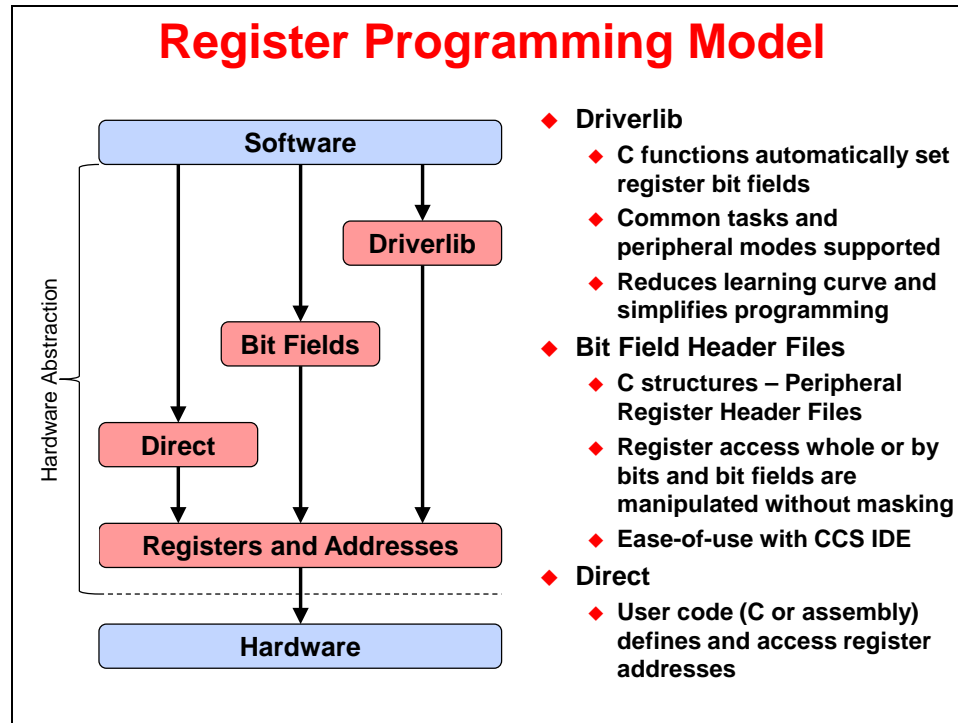
Module Objectives

- ◆ Review register programming model
- ◆ Understand the usage of the F28004x Driverlib and associated files
- ◆ Program an application using Driverlib
- ◆ Discuss Driverlib optimization

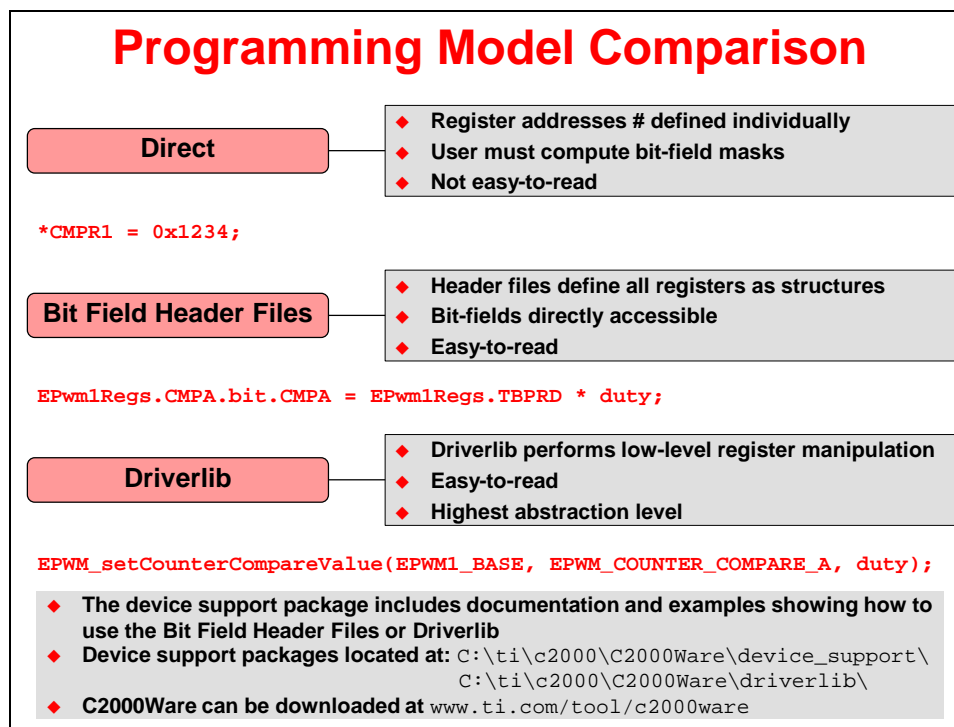
Chapter Topics

Peripheral Register Programming.....	3-1
<i>Register Programming Model</i>	<i>3-3</i>
<i>Driver Library (Driverlib).....</i>	<i>3-5</i>
Construction of a Driverlib Function.....	3-6
Driverlib Optimization	3-7
Driverlib API Functions and Examples.....	3-8
Content Assist.....	3-9
Driverlib Documentation.....	3-9
Driverlib Summary.....	3-10
Lab File Directory Structure	3-10

Register Programming Model



The various levels of the programming model provide different degrees of abstraction. The highest level is DriverLib which are C functions that automatically set the bit fields. This gives you a minimum amount of flexibility in exchange for a reduced learning curve and simplified programming. The bit field header files are C structures that allow registers to be access whole or by bits and bit fields, and modified without masking. This provides a balance between ease of use and flexibility when working with Code Composer Studio. Direct access to the registers is the lowest level where the user code, in C or assembly, defines and access register addresses.



The above slide provides a comparison of each programming model, from the lowest level to the highest level of abstraction. With direct access to the registers, the register addresses are #defined individually and the user must compute the bit-field mask. The bit field header files define all registers as structures and the bit fields are directly accessible. DriverLib performs low-level register manipulation and provides the highest level of abstraction. This workshop makes use of the Driverlib, which provides flexibility and makes it easy to program the device. Device support packages can be downloaded from www.ti.com.

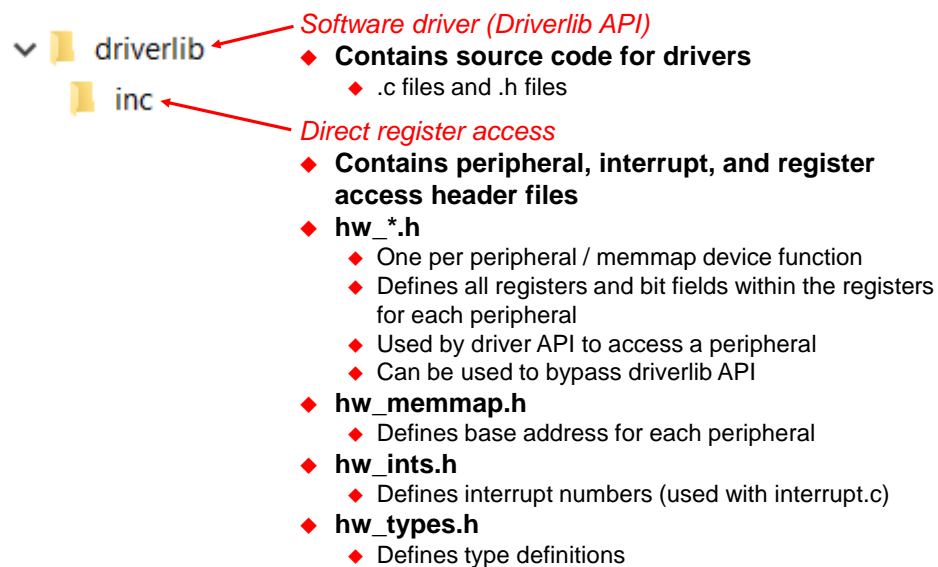
Driver Library (Driverlib)

Driver Library (Driverlib)

- ◆ **Driverlib “APIs” provide many advantages and benefits:**
 - ◆ Require less detailed knowledge of the hardware
 - ◆ Produce code that is easy-to-write and easy-to-read
 - ◆ Generally require less development time
 - ◆ Provide portability across other C2000 devices
 - ◆ Optimize well; remove overhead and speed up code execution
- ◆ **If needed, “direct register access” can be used to create custom Driverlib functions**
 - ◆ Requires detailed knowledge of:
 - ◆ Operation of each register and bit field
 - ◆ Interactions and sequencing required for proper peripheral operation
 - ◆ Can result in smaller and more efficient code
- ◆ **Both APIs and direct register access can be used independently or combined**

The Driver Library (Driverlib) is a set of drivers for accessing the peripherals and device configuration registers. While Driverlib is not drivers in the pure operating system sense (it does not have a common interface and it does not connect into a global device driver), they do provide a software layer to facilitate a slightly higher level of programming.

Driverlib File Structure



Construction of a Driverlib Function

Construction of a Driverlib Function

◆ **Driverlib API functions are built on top of the direct register access model**

- ◆ Useful to understand for debugging or when needing to directly access a register or bit field
- ◆ Uses a similar approach to traditional #define

◆ **Naming convention used in header files macros:**

- Values that end in **_BASE** are module instance base addresses
- Values that contain an **_O_** are register address offsets
- Values that end in **_M** are mask for multi-bit field register
- Values that end in **_S** are the number of bits to shift

hw_types.h contains the follow macros:

- **HWREG(x)** are 32-bit accesses
- **HWREGH(x)** are 16-bit accesses (or upper/lower 32-bit word)
- **HWREGB(x)** are 8-bit accesses
- **HWREGBP(x)** are used with byte peripherals

where x is the address to be accessed

With the direct register access model, the peripherals are programmed by writing values directly to the peripheral's registers. A set of macros is provided to simplify this process. These macros are stored in several header files contained in the `\inc` directory.

Driverlib Function Example

user.c – user source file

```
// Configure EPWM clock prescaler to TBCLK = EPWMCLK
EPWM_setClockPrescaler(EPWM2_BASE, EPWM_CLOCK_DIVIDER_1, EPWM_HSCLOCK_DIVIDER_1);
```

hw_memmap.h

```
#define EPWM2_BASE 0x00004100U // EPWM2
```

epwm.h

Contains typedef enum values for:
EPWM_CLOCK_DIVIDER (_1 = 0)
EPWM_HSCLOCK_DIVIDER (_1 = 0)

epwm.h – EPWM Driver

```
static inline void EPWM_setClockPrescaler(uint32_t base,
                                           EPWM_ClockDivider prescaler,
                                           EPWM_HSClockDivider highSpeedPrescaler)
{
    ASSERT(EPWM_isBaseValid(base));

    // write to CLKDIV and HSPCLKDIV bit
    HWREGH(base + EPWM_O_TBCTL) =
        ((HWREGH(base + EPWM_O_TBCTL) &
         ~(EPWM_TBCTL_CLKDIV_M | EPWM_TBCTL_HSPCLKDIV_M)) |
         (((uint16_t)prescaler << EPWM_TBCTL_CLKDIV_S) |
          ((uint16_t)highSpeedPrescaler << EPWM_TBCTL_HSPCLKDIV_S)));
}
```

hw_epwm.h

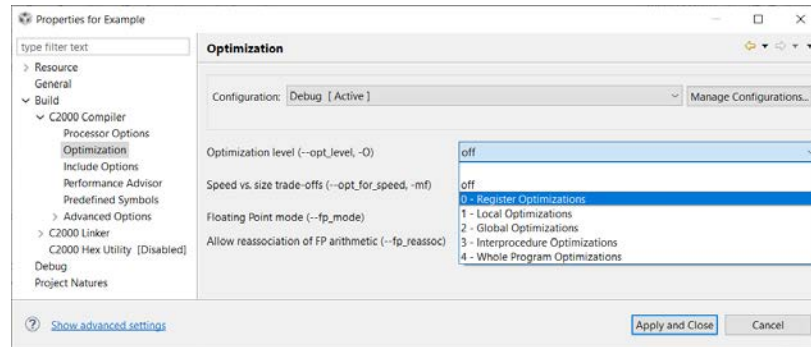
Contains #define for all:
EPWM_xxx values in epwm.h

Note: CCS 'F3' will open declaration

Driverlib Optimization

Driverlib Optimization

- ◆ In general, software abstraction can come at the cost of performance
- ◆ However, Driverlib's low-level abstraction and optimization-conscious design makes it efficient



- ◆ Most functions have been declared as inline functions
 - ◆ Allows the compiler to treat functions like macros when optimizer is turned on
 - ◆ Removes the overhead of function calls and speeds up code execution
- ◆ Use compiler option `--opt_level` set to 0 or higher

The optimization options are selected in the CCS project by right-clicking on the project in the project explorer window and then clicking 'Properties'. In the properties window, the optimization settings are located under: Build → C2000 Compiler → Optimization

Optimization Example

◆ Read ADC conversion results

- ◆ Optimization level of -O2; **Enable inlining** (`--enable_inlining`)
- ◆ Single MOV instruction is generated for each function call
- ◆ Results: 3 words of code / 3 cycles to execute

C-Source Code	Generated Assembly Instruction
<code>tmp[0]=ADC_readResult(ADCARERESULT_BASE, ADC_SOC_NUMBER0);</code>	<code>MOV *-SP[3], *(0:0x0b00)</code>
<code>tmp[1]=ADC_readResult(ADCARERESULT_BASE, ADC_SOC_NUMBER1);</code>	<code>MOV *-SP[2], *(0:0x0b01)</code>
<code>tmp[2]=ADC_readResult(ADCARERESULT_BASE, ADC_SOC_NUMBER2);</code>	<code>MOV *-SP[1], *(0:0x0b02)</code>

- ◆ When compiled with:
 - ◆ Optimization level of -O2; **Disable inlining** (`--disable_inlining`)
- ◆ Results:
 - ◆ 22 words of code
 - ◆ 4 words for `ADC_readResult()` and 18 words for the calling functions
 - ◆ 53 cycles to execute

Driverlib API Functions and Examples

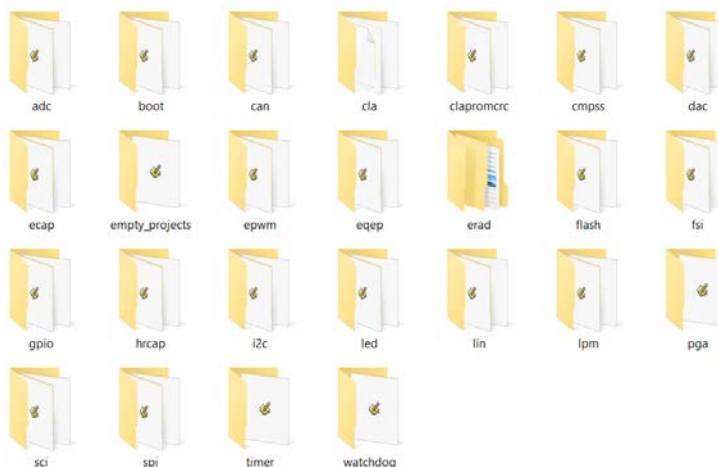
Driverlib API Functions

ADC	DMA	Interrupt
ASysCtl	ECAP	LIN
CAN	EPWM	MemCfg
CLA	EQEP	PGA
CLAPROMCRC	Flash	PMBus
CMPSS	FSI	SCI
CPU Timer	GPIO	SDFM
DAC	HRCAP	SPI
DCC	HRPWM	SysCtrl
DCSM	I2C	X-BAR

See the F28004x Peripheral Driver Library User's Guide for details

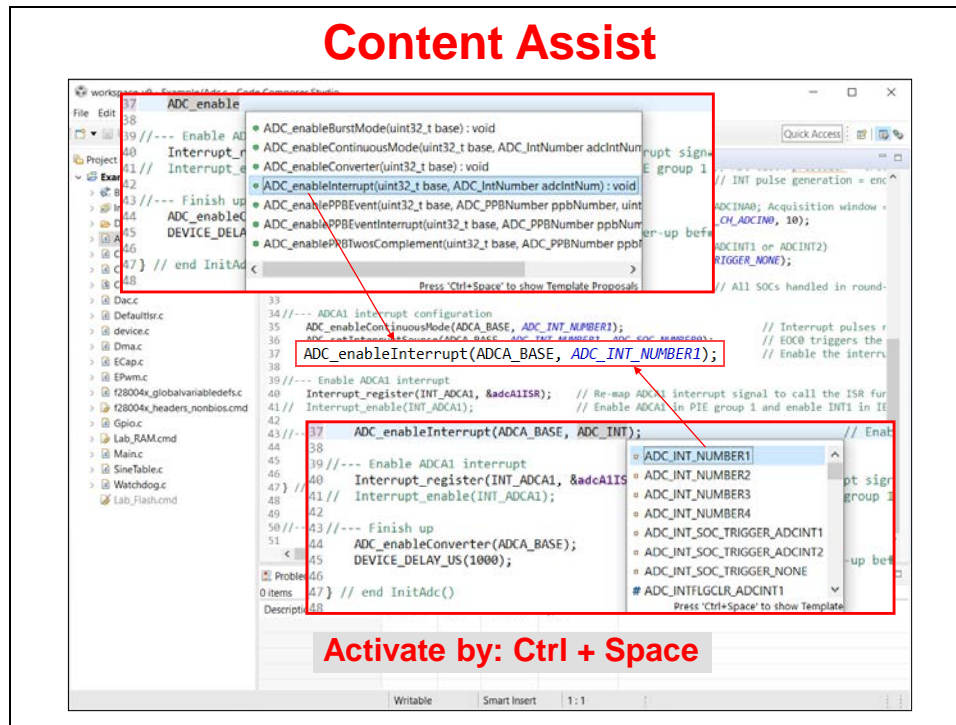
Driverlib Examples

- ◆ Example projects are helpful for getting started



C:\ti\c2000\C2000Ware_<version>\driverlib\F28004x\examples

Content Assist



The Content Assist feature can be used to offer suggestions for completing function and parameter names. Also, hover over the function to view its description.

Driverlib Documentation

Driverlib Documentation
Available in .pdf or .html formats

TEXAS INSTRUMENTS

F28004x Peripheral Driver Library 1.04.00.00

USER'S GUIDE

Copyright © 2018 Texas Instruments Incorporated.

TEXAS INSTRUMENTS Technology for Innovators™

Main Page Modules Data Structures Files Related Pages

Texas Instruments F28004x Peripheral Driver Library

The F28004x Peripheral Driver Library is a set of drivers for accessing the peripherals found on the F28004x microcontrollers. While they are not drivers in the pure operating system sense (that is, they do not have a common interface and do not connect into a global device driver infrastructure), they do provide a software layer to facilitate a slightly higher level of programming than direct register accesses.

The capabilities and organization of the drivers are governed by the following design goals:

- They are written entirely in C except where absolutely not possible.
- Where possible, computations that can be performed at compile time are done there instead of at run time.
- They are intended to make code more portable across other C2000 devices.
- Code written with these APIs will be more readable than code written using many direct register accesses.

Some consequences of this are that the drivers are not necessarily as efficient as they could be (from a code size and/or execution speed point of view). While the most efficient piece of code for operating a peripheral would be written in assembly and custom tailored to the specific requirements of the application, further size optimizations of the drivers would make them more difficult to understand.

For many applications, the drivers can be used as is. But in some cases, the drivers will have to be enhanced or rewritten in order to meet the functionality, memory, or processing requirements of the application. If so, the existing driver can be used as a reference on how to operate the peripheral.

Copyright © 2018, Texas Instruments Incorporated

C:\ti\c2000\C2000Ware_<version>\device_support\f28004x\docs

Driverlib Summary

Driverlib Summary

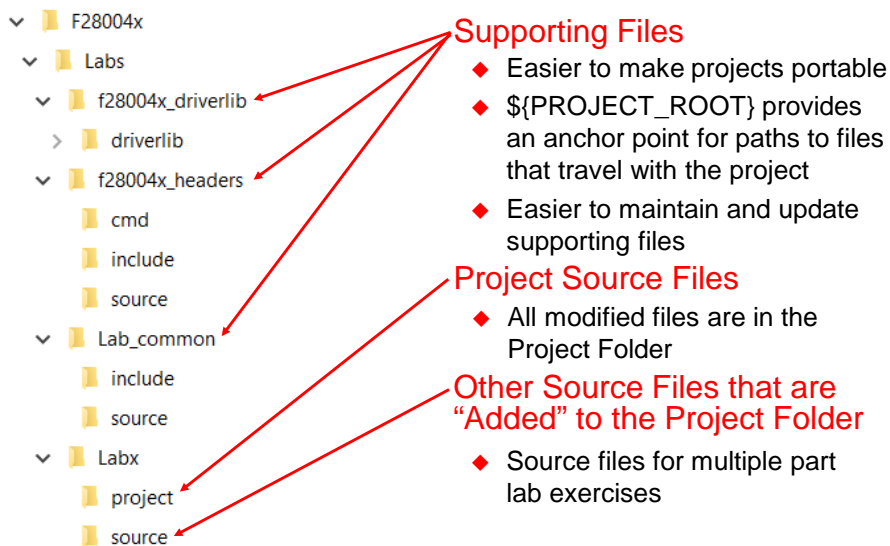
- ◆ Easier code development
- ◆ Easy to use
- ◆ Has been written to be optimized well
- ◆ CCS – hover over function to view description
- ◆ Compatible with Bit Field Header Files
- ◆ TI has already done all the work!
 - ◆ Use the correct Driverlib package for your device:

F28004x	F2807x	F2837xS	F2837xD
---------	--------	---------	---------

Go to <http://www.ti.com> and enter “C2000Ware” in the keyword search box

Lab File Directory Structure

Lab File Directory Structure



Note: CCSv9 will automatically add ALL files contained in the folder where the project is created

Reset and Interrupts

Introduction

This module describes the device reset and interrupt process, and explains how the Peripheral Interrupt Expansion (PIE) is used to service the peripheral interrupts.

Module Objectives

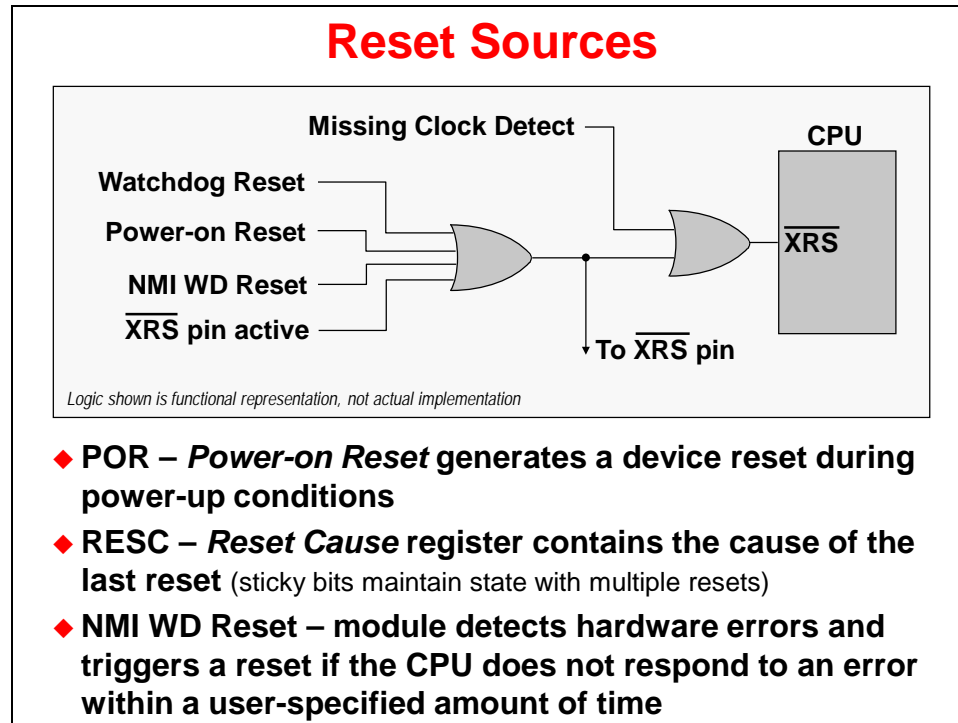
Module Objectives

- ◆ **Reset Sources**
- ◆ **Enhanced Boot Modes**
- ◆ **Peripheral Reset**
- ◆ **Interrupt Source and Interrupt Structure**
- ◆ **Peripheral Interrupt Expansion**
- ◆ **Initialize Interrupt Module**
- ◆ **Event Sequence of an Interrupt**

Chapter Topics

Reset and Interrupts	4-1
<i>Reset and Boot Process</i>	<i>4-3</i>
Reset - Bootloader	4-5
Emulation Boot Mode.....	4-6
Stand-Alone Boot Mode	4-7
Boot Mode Definition	4-8
Reset Code Flow – Summary	4-9
Emulation Boot Mode using Code Composer Studio GEL	4-9
Getting to main()	4-10
Peripheral Software Reset Registers.....	4-11
<i>Interrupts</i>	<i>4-12</i>
Interrupt Processing	4-13
Interrupt Enable Register (IER).....	4-14
Interrupt Global Mask Bit (INTM)	4-14
Peripheral Interrupt Expansion (PIE)	4-15
PIE Block Initialization.....	4-18
Interrupt Signal Flow – Summary.....	4-20
Interrupt Response and Latency.....	4-21

Reset and Boot Process



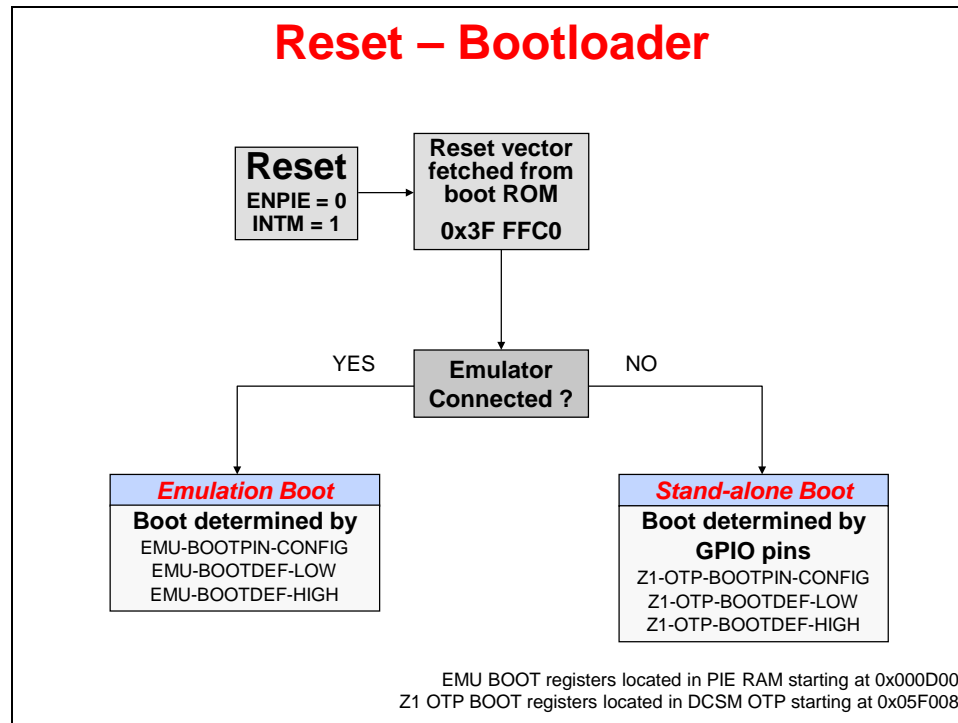
The device has various reset sources, which include an external reset pin, watchdog timer reset, power-on reset which generates a device reset during power-up conditions, NMI reset, and a missing clock detect reset. A reset cause register (RESC) is available and can be read to determine the cause of the reset. The external reset pin is the main chip-level reset for the device, and it resets the device to the default state. The power-on reset (POR) circuit is used to create a clean reset throughout the device during power-up, while suppressing glitches on the input/output pins.

Enhanced Boot Modes

- ◆ The enhanced boot modes provide for the ability to move, reduce, or eliminate boot mode select pins
- ◆ **BOOTPIN-CONFIG register selects boot pins to be used**
 - ◆ Emulation Boot Mode: **EMU-BOOTPIN-CONFIG** register
 - ◆ Stand-Alone Boot Mode: **Z1-OTP-BOOTPIN-CONFIG** register
- ◆ **BOOTDEF register determines boot mode option and assignment of peripheral GPIO pins or flash/RAM entry point**
 - ◆ Emulation Boot Mode: **EMU-BOOTDEF-LOW/HIGH** register
 - ◆ Stand-Alone Boot Mode: **Z1-OTP-BOOTDEF-LOW/HIGH** register

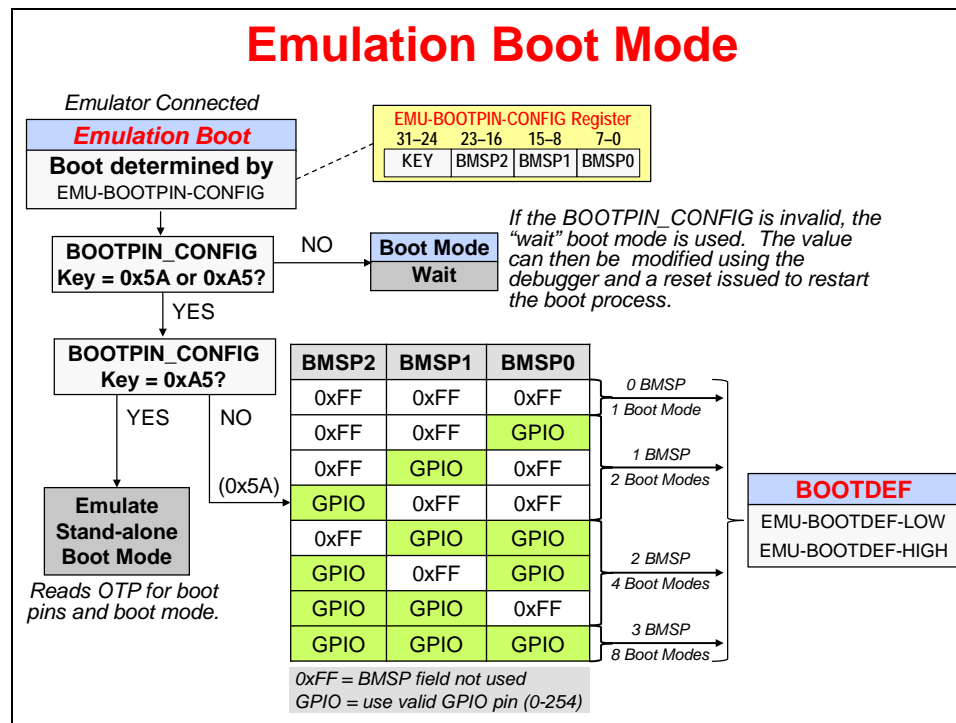
When the MCU is powered-on, and each time the MCU is reset, the internal bootloader software located in the boot ROM is executed. The boot ROM contains bootloading routines and execution entry points into specific on-chip memory blocks. This initial software program is used to load an application to the device RAM through the various bootable peripherals, or it can be configured to start an application located in flash. The F28004x is extremely flexible in its ability to use alternate, reduce, or completely eliminate boot mode selection pins by programming a BOOTPIN_CONFIG register.

Reset - Bootloader



After the MCU is powered-up or reset, the peripheral interrupt expansion block, also known as the PIE block, and the master interrupt switch INTM are disabled. This prevents any interrupts during the boot process. The program counter is set to 0x3FFFC0, where the reset vector is fetched. Execution then continues in the boot ROM at the code section named InitBoot. If the emulator is connected, then the boot process follows the Emulation Boot mode flow. In Emulation Boot mode, the boot is determined by the EMU-BOOTPIN-CONFIG and EMU-BOOTDEF-LOW/HIGH registers located in the PIE RAM. If the emulator is not connected, the boot process follows the Stand-alone Boot mode flow. In Stand-alone Boot mode, the boot is determined by two GPIO pins or the Z1-OTP-BOOTPIN-CONFIG and Z1-OTP-BOOTDEF-HIGH/LOW registers located in the DCSM OTP.

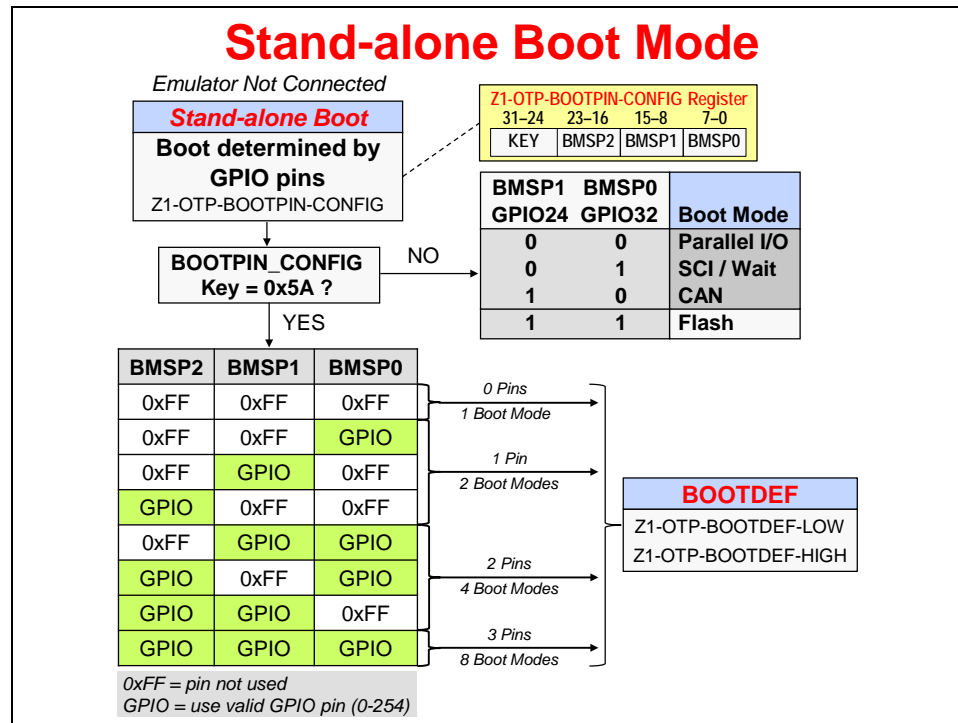
Emulation Boot Mode



In Emulation Boot mode, first the KEY value located in the EMU-BOOTPIN-CONFIG register (bit fields 31-24) is checked for a value of 0x5A or 0xA5. If the KEY value is not 0x5A or 0xA5, the "wait" boot mode is entered. The KEY value and the Boot Mode Selection Pin values (BMSP2-0, bit fields 23-0) can then be modified using the debugger and a reset is issued to restart the boot process. This is the typical sequence followed during device power-up with the emulator connected, allowing the user to control the boot process using the debugger.

Once the EMU-BOOTPIN-CONFIG register is configured and a reset is issued, the KEY value is checked again. If the KEY value is set to 0xA5 the Stand-alone Boot mode is emulated and the Z1-OTP-BOOTPIN-CONFIG register is read for the boot pins and boot mode. Otherwise, the KEY value is set to 0x5A and the boot mode is determined by the BMSP bit field values in the EMU-BOOTPIN-CONFIG register and the EMU-BOOTDEF-LOW/HIGH registers. The EMU-BOOTPIN-CONFIG register contains three BMSP bit fields. If the BMSP bit field is set to 0xFF, then the bit field is not used. Therefore, the boot modes can be set by zero, one, two, or three BMSP bit fields. This provides one, two, four, or eight boot mode options, respectively. Details about the BOOTDEF options will be discussed after the Stand-alone Boot mode is covered.

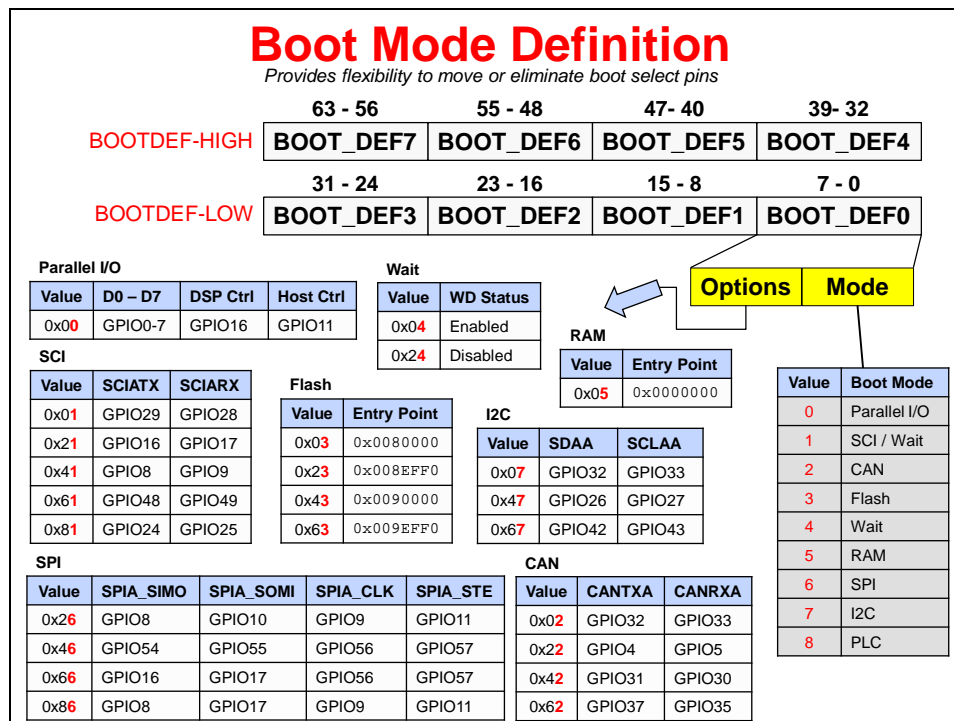
Stand-Alone Boot Mode



In Stand-alone Boot mode, if the KEY value located in the Z1-OTP-BOOTPIN-CONFIG register (bit fields 31-24) is not 0x5A, the boot mode is determined by the default GPIO24 and GPIO32 pins. These two pins provide four boot options – Parallel I/O, SCI/Wait, CAN or Flash. If the KEY value is 0x5A the boot mode is determined by the BMSP bit field values in the Z1-OTP-BOOTPIN-CONFIG register and the OTP-BOOTDEF-LOW/HIGH registers. The Z1-OTP-BOOTPIN-CONFIG register contains three BMSP bit fields. If the BMSP bit field is set to 0xFF, then the GPIO pin is not used. Therefore, the boot modes can be set by zero, one, two, or three GPIO pins. This provides one, two, four, or eight boot mode options, respectively.

Boot Mode Definition

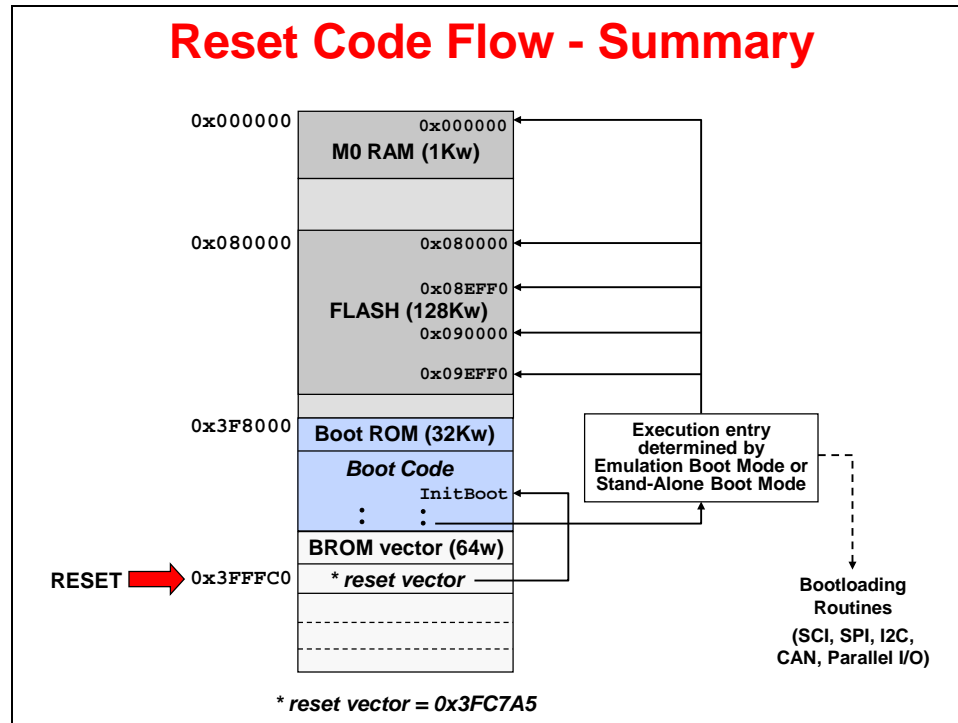
The BOOTDEF options described here applies to both the EMU-BOOTDEF-LOW/HIGH registers used in Emulation Boot mode and the Z1-OTP-BOOTDEF-LOW/HIGH registers used in Stand-alone Boot mode. The BOOTDEF-LOW/HIGH registers consist of eight separate bit fields named BOOT_DEF0 through BOOT-DEF7. These bit fields correspond to the one, two, four, or eight boot mode options that are selected by the zero, one, two, or three BMSP bit fields/GPIO pins, respectively in the BOOTPIN_CONFIG register. Therefore, if zero BMSP bit fields/GPIO pins are selected, then only the BOOT_DEF0 bit field in the BOOTDEF-LOW/HIGH registers is used. Likewise, if three BMSP bit fields/GPIO pins are selected, then BOOT_DEF0 through BOOT_DEF7 in the BOOTDEF-LOW/HIGH registers is used.



The value in the BOOT_DEF bit fields determines which peripheral is used for bootloading or the entry point that is used for code execution. In the BOOT_DEF bit field the lower bits define the boot mode used and the upper bits define the options for that bit mode. Utilizing this type of booting technique provides flexibility for selecting multiple boot modes, as well as reducing the number of boot mode pins.

Reset Code Flow – Summary

In summary, the reset code flow is as follows. After reset, the program counter is set to 0x3FFFC0, where the flow is vectored to the Init_Boot code in the Boot ROM. The Init_Boot code defines the execution entry based on emulation boot mode or stand-alone boot mode. The entry point can be executing boot-loading routines, entry to the flash, or M0 RAM.



Emulation Boot Mode using Code Composer Studio GEL

The CCS GEL file is used to setup the boot modes for the device during debug. By default the GEL file provides functions to set the device for “Boot to SARAM” and “Boot to FLASH”. The GEL file can be modified to include other boot mode options, if desired.

```

/*****
/* EMU Boot Mode - Set Boot Mode During Debug */
/*****
menuitem "EMU Boot Mode Select"
hotmenu EMU_BOOT_SARAM()
{
    *(unsigned long *)0xD00 = 0x5AFFFFFF;
    *0xD04 = 0x0005;
}
hotmenu EMU_BOOT_FLASH()
{
    *(unsigned long *)0xD00 = 0x5AFFFFFF;
    *0xD04 = 0x0003;
}

```

To access the GEL file use: Tools → GEL Files

Getting to main()

After reset how do we get to main()?

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

CodeStartBranch.asm

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

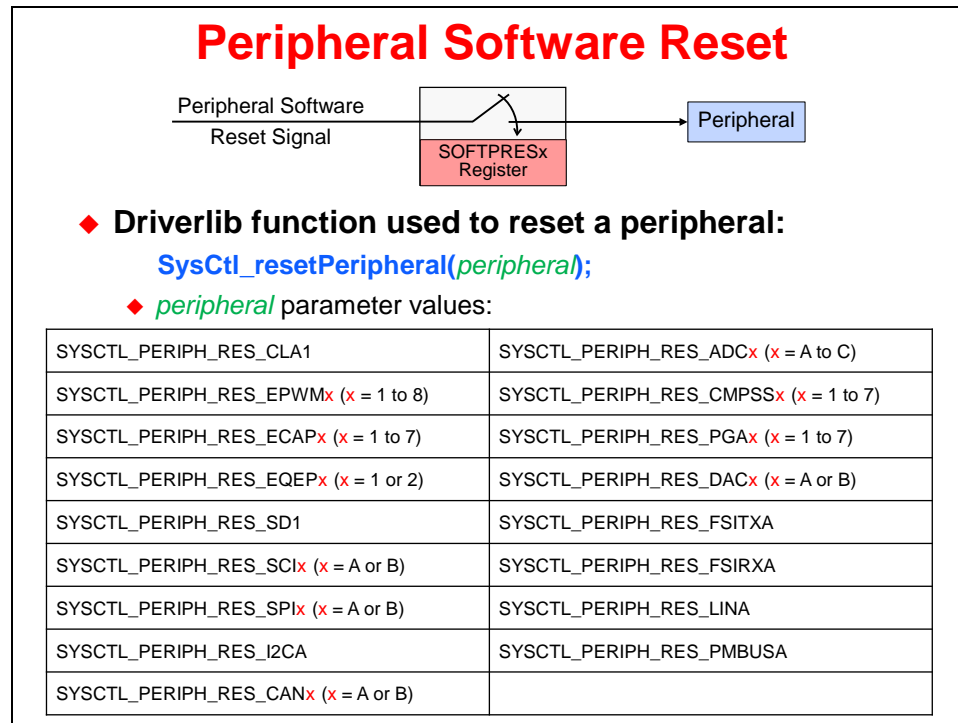
Linker.cmd

```
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 RAMM0; to run out of Flash, the "codestart" section would be linked to the entry point of the Flash memory block

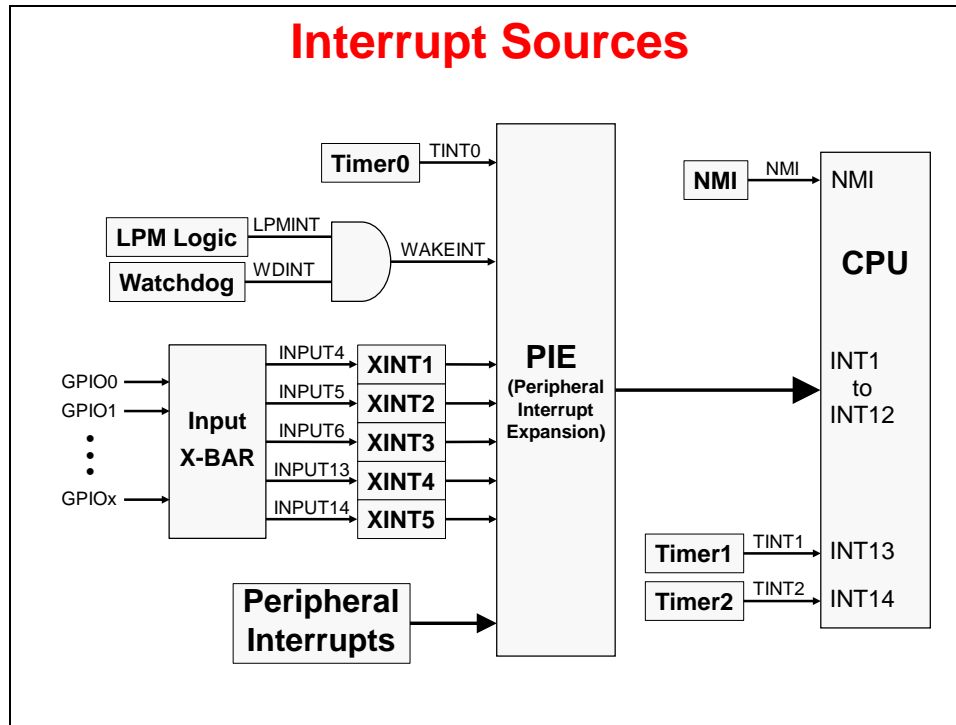
After reset how do we get to main? When the bootloader process is completed, a branch to the compiler runtime support library is located at the code entry point. This branch to `_c_int00` is executed, then the compiler environment is set up, and finally `main` is called.

Peripheral Software Reset Registers



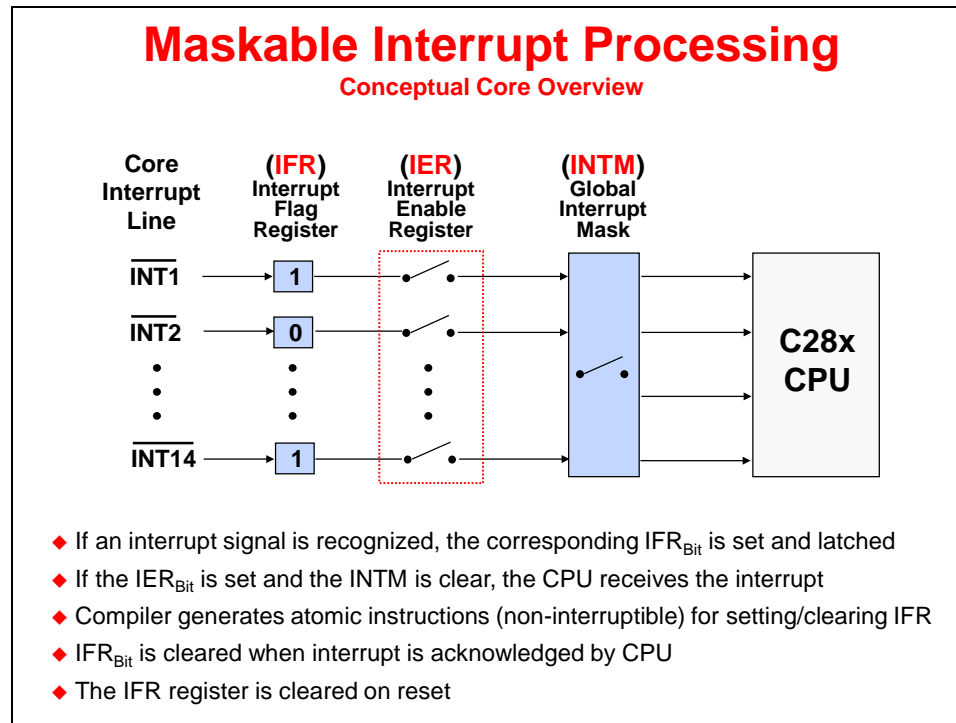
The peripheral software reset register (SOFTPRESx) contains the reset bit for each peripheral. The Driverlib functions are used to reset a peripheral, as shown above.

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 five external interrupt lines, which are mapped through the Input X-BAR, and the external reset pin. The CPU core has 14 interrupt lines. The Peripheral Interrupt Expansion block, known as the PIE block, is connected to the core interrupt lines 1 through 12 and is used to expand the CPU core interrupt capability, allowing up to 192 possible interrupt sources.

Interrupt Processing



By using a series of flag and enable registers, the CPU can be configured to service one interrupt while others remain pending, or perhaps disabled when servicing certain critical tasks. When an interrupt signal occurs on a core line, the interrupt flag register (IFR) for that core line is set. If the appropriate interrupt enable register (IER) is enabled for that core line, and the interrupt global mask (INTM) is enabled, the interrupt signal will propagate to the core. Once the interrupt service routine (ISR) starts processing the interrupt, the INTM bit is disabled to prevent nested interrupts. The IFR is then cleared and ready for the next interrupt signal. When the interrupt servicing is completed, the INTM bit is automatically enabled, allowing the next interrupt to be serviced. Notice that when the INTM bit is '0', the "switch" is closed and enabled. When the bit is '1', the "switch" is open and disabled. The IER is managed by enabling and disabling Driverlib parameter values. The INTM bit in the status register is managed by using a Driverlib function or in-line assembly instructions (macro).

Interrupt Enable Register (IER)

Interrupt Enable Register (IER)

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

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

- ◆ **Driverlib function used to modify IER:**
 - `Interrupt_enableInCPU(cpuInterrupt);`
 - `Interrupt_disableInCPU(cpuInterrupt);`
- ◆ *cpuInterrupt* parameter is a logical OR of the values:
 - ◆ INTERRUPT_CPU_INT x
 - ◆ where x is the interrupt number between 1 and 14
 - ◆ INTERRUPT_CPU_DLOGINT
 - ◆ INTERRUPT_CPU_RTOSINT
- ◆ **IER register is cleared on reset**

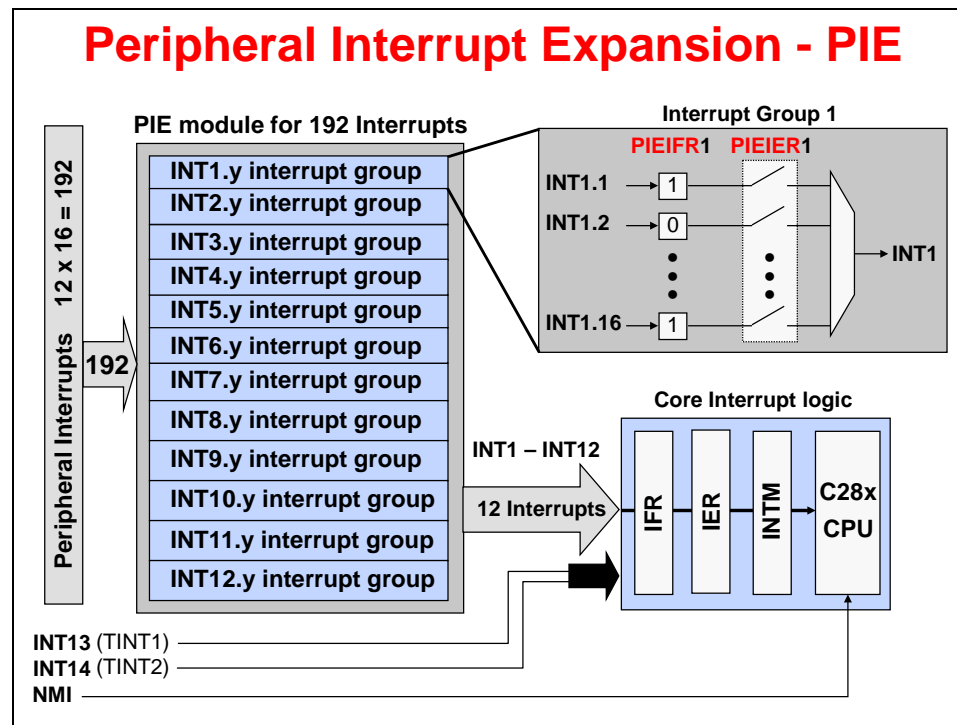
Interrupt Global Mask Bit (INTM)

Interrupt Global Mask Bit

ST1		Bit 0 INTM
-----	--	----------------------

- ◆ **INTM is used to globally enable/disable interrupts:**
 - ◆ Enable: INTM = 0
 - ◆ Disable: INTM = 1 (reset value)
- ◆ **Driverlib function used to modify INTM:**
 - `Interrupt_enableMaster();`
 - `Interrupt_disableMaster();`
- ◆ **Alternatively the following macros can be used:**
 - `EINT;` //defined as - `asm(" clrc INTM");`
 - `DINT;` //defined as - `asm(" setc INTM");`

Peripheral Interrupt Expansion (PIE)



The C28x CPU core has a total of fourteen interrupt lines, of which two interrupt lines are directly connected to CPU Timers 1 and 2 (on INT13 and INT14, respectively) and the remaining twelve interrupt lines (INT1 through INT12) are used to service the peripheral interrupts. A Peripheral Interrupt Expansion (PIE) module multiplexes up to sixteen peripheral interrupts into each of the twelve CPU interrupt lines, further expanding support for up to 192 peripheral interrupt signals. The PIE module also expands the interrupt vector table, allowing each unique interrupt signal to have its own interrupt service routine (ISR), permitting the CPU to support a large number of peripherals.

The PIE module has an individual flag and enable bit for each peripheral interrupt signal. Each of the sixteen peripheral interrupt signals that are multiplexed into a single CPU interrupt line is referred to as a “group”, so the PIE module consists of 12 groups. Each PIE group has a 16-bit flag register (PIEIFRx), a 16-bit enable register (PIEIERx), and a bit field in the PIE acknowledge register (PIEACK) which acts as a common interrupt mask for the entire group. For a peripheral interrupt to propagate to the CPU, the appropriate PIEIFR must be set, the PIEIER enabled, the CPU IFR set, the IER enabled, and the INTM enabled. Note that some peripherals can have multiple events trigger the same interrupt signal, and the cause of the interrupt can be determined by reading the peripheral’s status register.

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 sixteen interrupt sources. Each group has a PIE interrupt enable register and a PIE interrupt flag register. Note that interrupt lines 13, 14, and NMI bypass the PIE block.

F28004x PIE Assignment Table - Lower

	INTx.8	INTx.7	INTx.6	INTx.5	INTx.4	INTx.3	INTx.2	INTx.1
INT1	WAKE (WDOG)	TIMER0		XINT2	XINT1	ADCC1	ADCB1	ADCA1
INT2	EPWM8_TZ	EPWM7_TZ	EPWM6_TZ	EPWM5_TZ	EPWM4_TZ	EPWM3_TZ	EPWM2_TZ	EPWM1_TZ
INT3	EPWM8	EPWM7	EPWM6	EPWM5	EPWM4	EPWM3	EPWM2	EPWM1
INT4		ECAP7	ECAP6	ECAP5	ECAP4	ECAP3	ECAP2	ECAP1
INT5							EQEP2	EQEP1
INT6					SPIB_TX	SPIB_RX	SPIA_TX	SPIA_RX
INT7			DMA_CH6	DMA_CH5	DMA_CH4	DMA_CH3	DMA_CH2	DMA_CH1
INT8							I2CA_FIFO	I2CA
INT9	CANB_1	CANB_0	CANA_1	CANA_0	SCIB_TX	SCIB_RX	SCIA_TX	SCIA_RX
INT10	ADCB4	ADCB3	ADCB2	ADCB_EVT	ADCA4	ADCA3	ADCA2	ADCA_EVT
INT11	CLA1_8	CLA1_7	CLA1_6	CLA1_5	CLA1_4	CLA1_3	CLA1_2	CLA1_1
INT12	FPU_UND_ERFLOW	FPU_OV_ERFLOW				XINT5	XINT4	XINT3

Note: above label names proceed with **INT_** and #defines are located in **driverlib/inc/hw_ints.h**

The PIE assignment table maps each peripheral interrupt to the unique vector location for that interrupt service routine. Notice the interrupt numbers on the left represent the twelve core group interrupt lines and the interrupt numbers across the top represent the lower eight of the sixteen peripheral interrupts within the core group interrupt line. The next figure shows the upper eight of the sixteen peripheral interrupts within the core group interrupt line.

F28004x PIE Assignment Table - Upper

	INTx.16	INTx.15	INTx.14	INTx.13	INTx.12	INTx.11	INTx.10	INTx.9
INT1								
INT2								
INT3								
INT4		ECAP7_2	ECAP6_2					
INT5	SDFM1_DR4	SDFM1_DR3	SDFM1_DR2	SDFM1_DR1				SDFM1
INT6								
INT7	DCC	CLA1PR_OMCRC	FSIRXA_INT2	FSIRXA_INT1	FSITXA_INT2	FSITXA_INT1		
INT8				PMBUSA			LINA_1	LINA_0
INT9								
INT10					ADCC4	ADCC3	ADCC2	ADCC_EVT
INT11								
INT12	CLA_UND_ERFLOW	CLA_OV_ERFLOW		SYS_PLL_SLIP	RAM_ACC_VIOL	FLASH_CO_RR_ERR	RAM_CO_RR_ERR	

Note: above label names proceed with **INT_** and #defines are located in **driverlib/inc/hw_ints.h**

PIEIER and PIEACK Registers

PIEIER_x register (x = 1 to 12)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
INTx.16	INTx.15	INTx.14	INTx.13	INTx.12	INTx.11	INTx.10	INTx.9	INTx.8	INTx.7	INTx.6	INTx.5	INTx.4	INTx.3	INTx.2	INTx.1

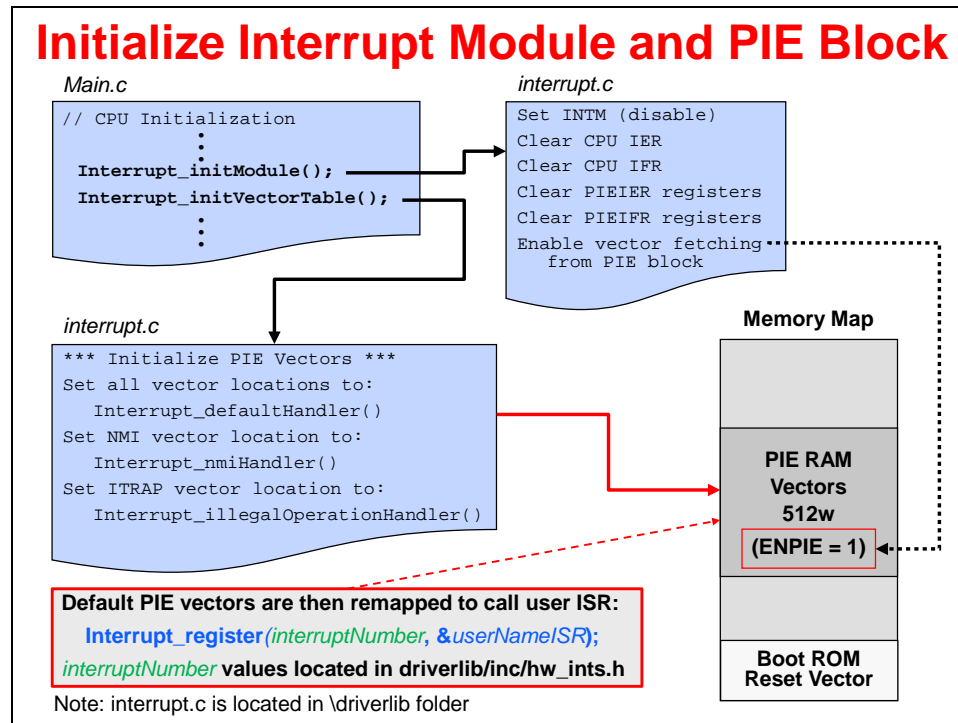
PIE Interrupt Acknowledge register (PIEACK)

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

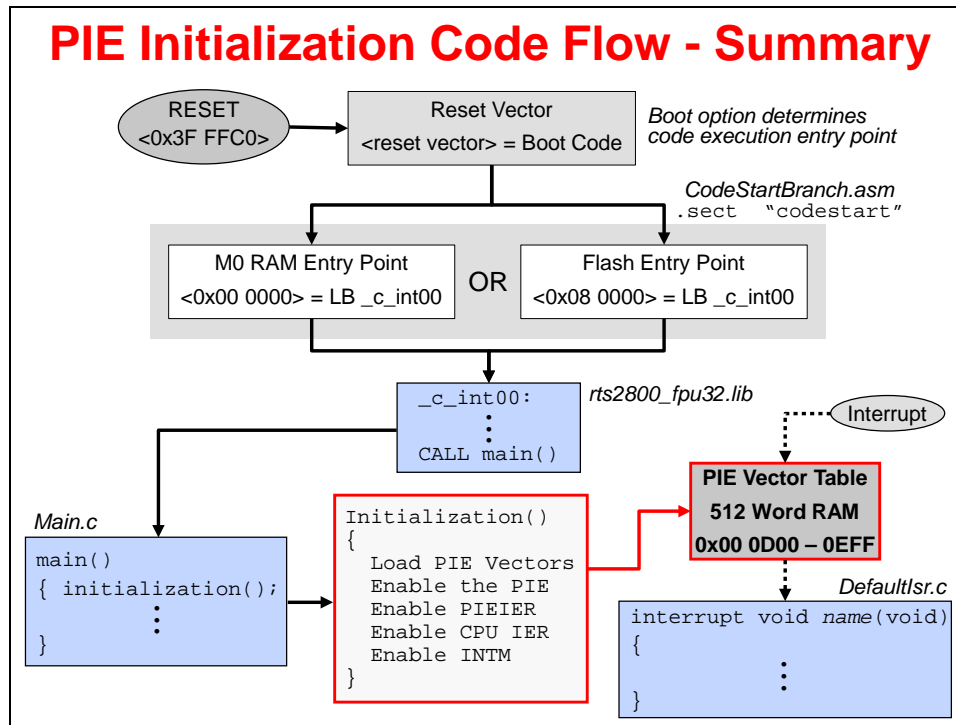
- ◆ **NOTE:** These Driverlib functions modify **BOTH** the PIEIER and core IER registers:
 - `Interrupt_enable(interruptNumber);`
 - `Interrupt_disable(interruptNumber);`
 - ◆ `interruptNumber` values are supplied in driverlib/inc/hw_ints.h
- ◆ **Driverlib function used to acknowledge PIE group:**
 - `Interrupt_clearACKGroup(group);`
 - ◆ `group` parameter is a logical OR of the values:
 - ◆ `INTERRUPT_ACK_GROUPx`
 - ◆ where `x` is the interrupt number between 1 and 12
 - ◆ Acknowledges group and clears any interrupt flag within group
 - ◆ Required to receive further interrupts in PIE group (done in ISR)

Similar to the core interrupt process, the PIE module has an individual flag and enable bit for each peripheral interrupt signal. Each PIE group has a 16-bit flag register, a 16-bit enable register, and a bit field in the PIE acknowledge register which acts as a common interrupt mask for the entire group. An enable PIE bit in the PIECTRL register is used to activate the PIE module. Note that when using the Driverlib function to enable and disable interrupts, both the PIEIER and CPU core IER registers are modified.

PIE Block Initialization



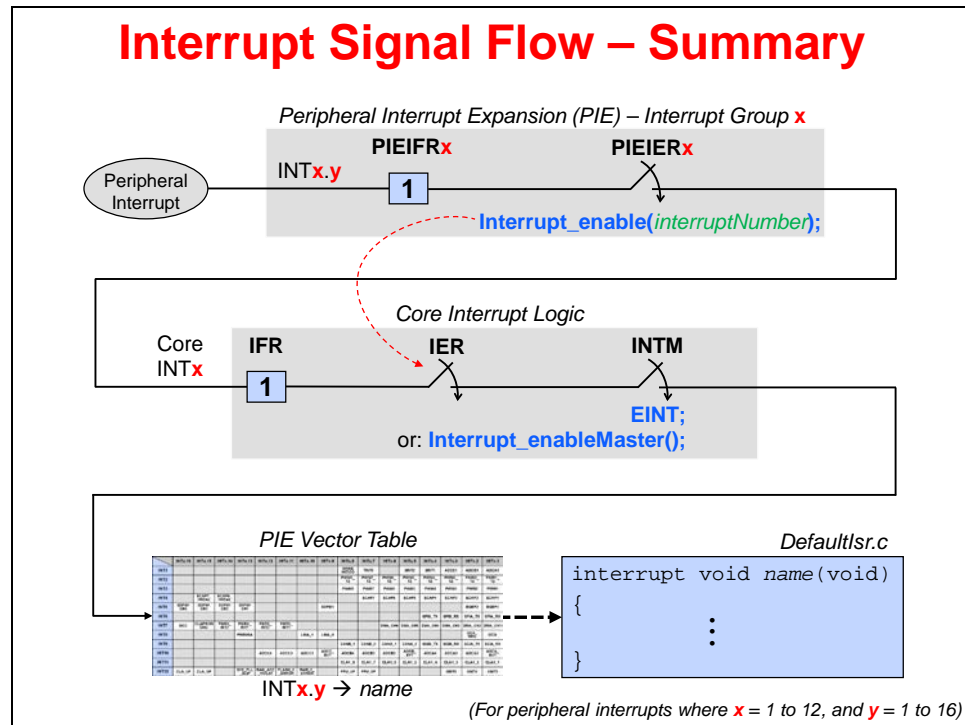
Two separate functions are called to initialize the interrupt module and PIE block. During processor initialization the interrupt vectors, as mapped in the PIE interrupt assignment table, are copied to the PIE RAM and then the PIE module is enabled by setting ENPIE to '1'. When the CPU receives an interrupt, the vector address of the ISR is fetched from the PIE RAM, and the interrupt with the highest priority that is both flagged and enabled is executed. Priority is determined by the location within the interrupt vector table. The lowest numbered interrupt has the highest priority when multiple interrupts are pending.



In summary, the PIE initialization code flow is as follows. After the device is reset and execution of the boot code is completed, the selected boot option determines the code entry point. In this figure, two different entry points are shown. The one on the left is for memory block M0 RAM, and the one on the right is for flash.

In either case, the CodeStartBranch.asm file has a Long Branch instruction to the entry point of the runtime support library. After the runtime support library completes execution, main is called. In main, the two functions are called to initialize the interrupt process and enable the PIE module. When the CPU receives an interrupt, the vector address of the ISR is fetched from the PIE RAM, and the interrupt with the highest priority that is both flagged and enabled is executed. Priority is determined by the location within the interrupt vector table.

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

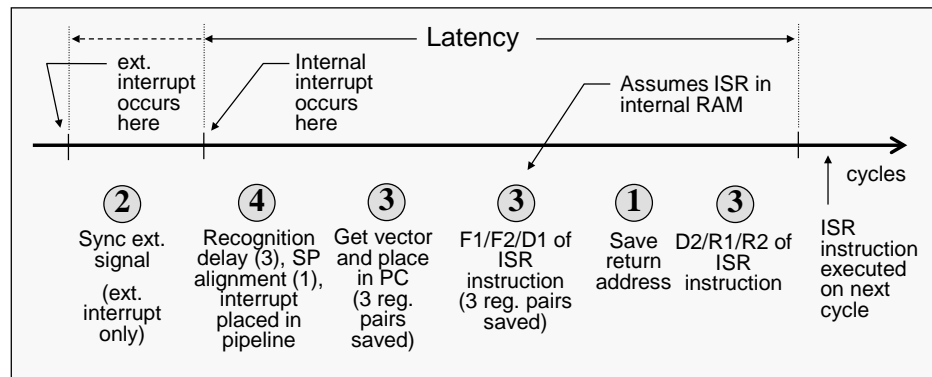
Interrupt Response - Hardware Sequence

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 covers the operation of the OSC/PLL-based clock module and watchdog timer. Also, the general-purpose digital I/O, external interrupts, low power modes and the register protection will be covered.

Module Objectives

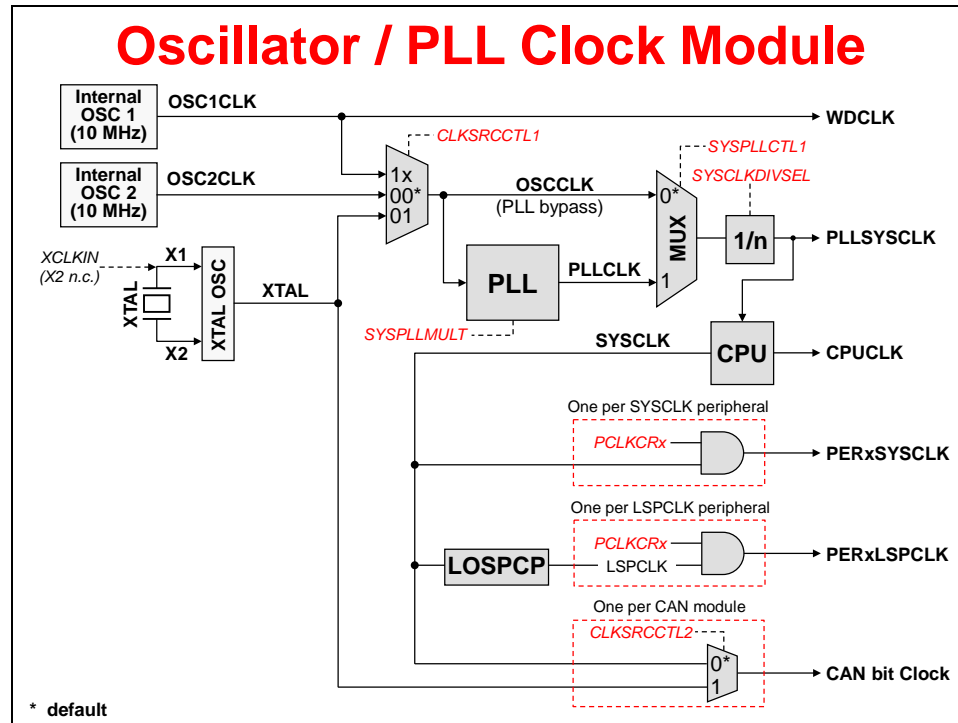
Module Objectives

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

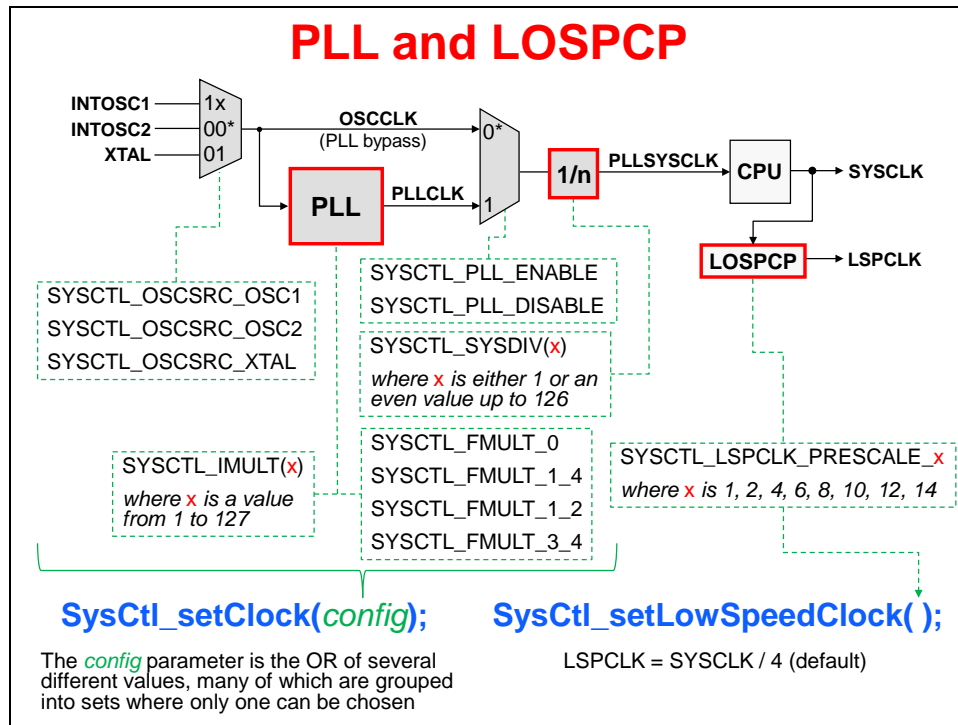
Chapter Topics

System Initialization	5-1
<i>Oscillator/PLL Clock Module</i>	5-3
Initializing Clock Modules.....	5-5
<i>Watchdog Timer</i>	5-6
<i>General Purpose Digital I/O</i>	5-10
Configuring GPIO Pins.....	5-11
GPIO Input X-Bar.....	5-14
GPIO Output X-Bar.....	5-15
<i>External Interrupts</i>	5-17
<i>Low Power Modes</i>	5-18
<i>Register Protection</i>	5-20
<i>Lab 5: System Initialization</i>	5-22

Oscillator/PLL Clock Module



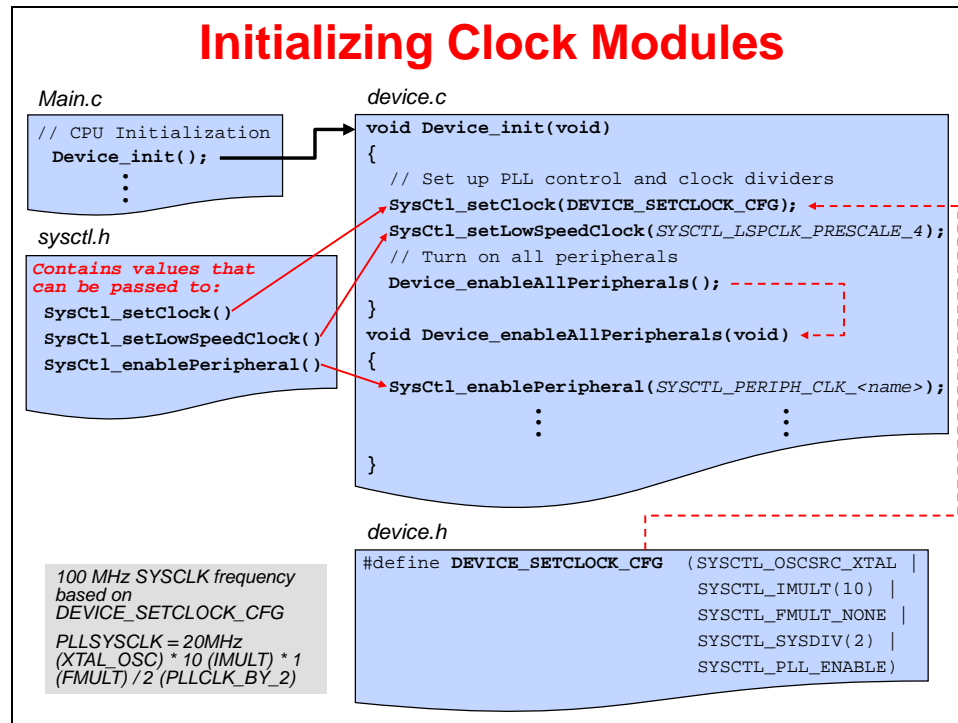
The device clock signals are derived from one of four clock sources: Internal Oscillator 1 (INTOSC1), Internal Oscillator 2 (INTOSC2), External Oscillator (XTAL), and single-ended 3.3V external clock (XCLKIN). At power-up, the device is clocked from the on-chip 10 MHz oscillator INTOSC2. INTOSC2 is the primary internal clock source, and is the default system clock at reset. The device also includes a redundant on-chip 10 MHz oscillator INTOSC1. INTOSC1 is a backup clock source, which normally only clocks the watchdog timers and missing clock detection circuit. Additionally, the device includes dedicated X1 and X2 pins for supporting an external clock source such as an external oscillator, crystal, or resonator.



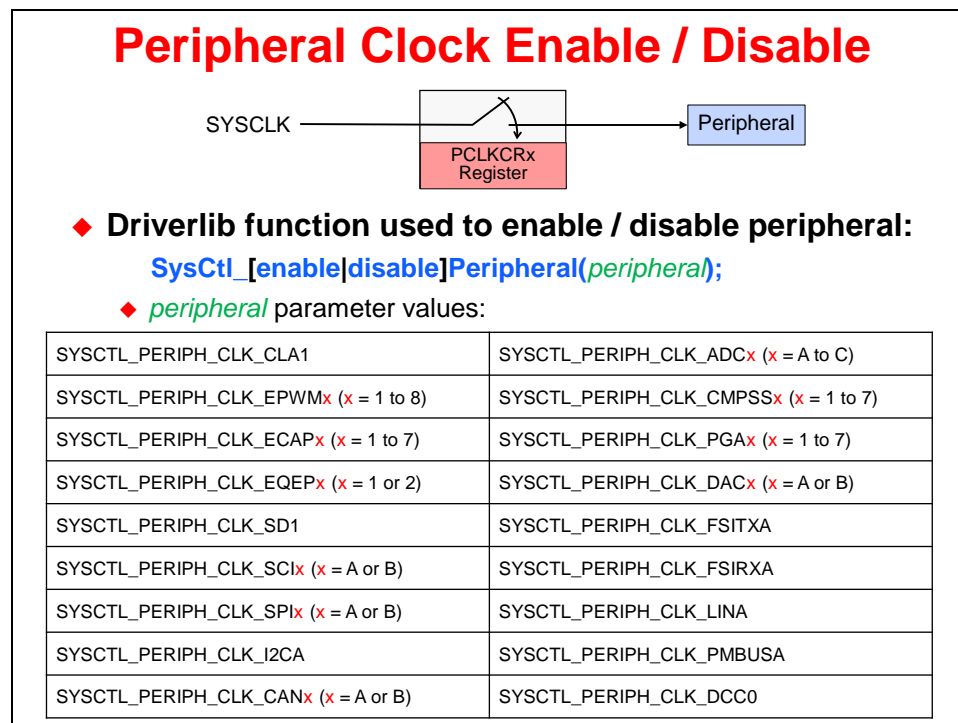
The clock sources can be multiplied using the PLL and divided down to produce the desired clock frequencies for a specific application. A clock source can be fed directly into the CPU or multiplied using the PLL. The PLL provides the capability to use the internal 10 MHz oscillator and run the device at the full 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 7-bit integer and 2-bit fractional ratio control to select different CPU clock rates. The C28x CPU provides a SYSCLK 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 SYSCLK and use their own clock prescalers for operation.

Initializing Clock Modules



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



Watchdog Timer

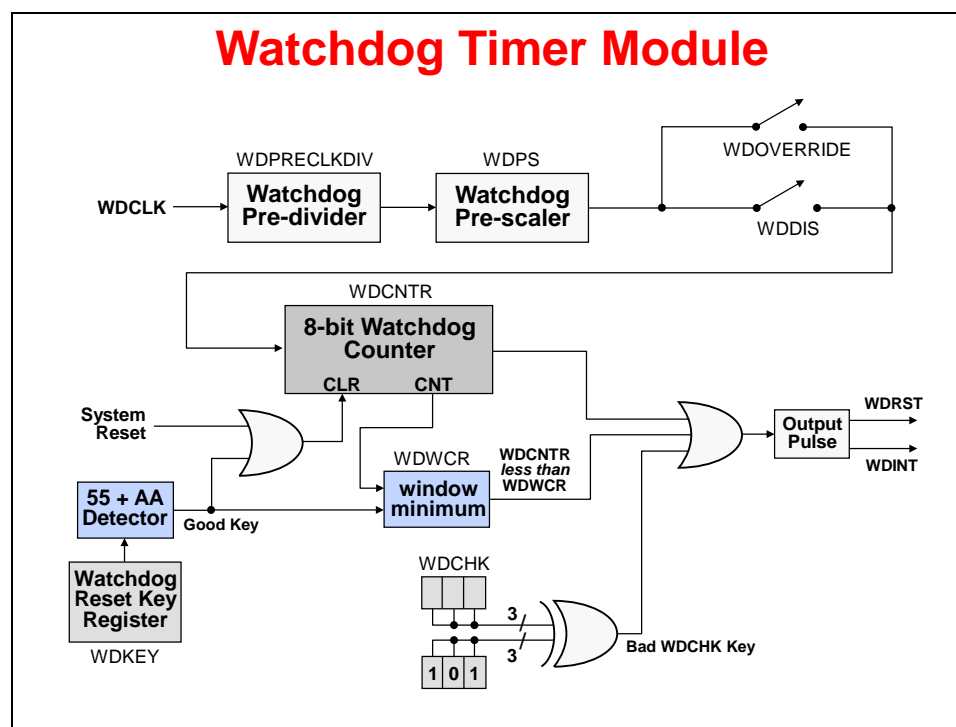
The watchdog timer is a safety feature which resets the device if the program runs away or gets trapped in an unintended infinite loop. The watchdog counter runs independent of the CPU. If the counter overflows, a user-selectable reset or interrupt is triggered. During runtime the correct key values in the proper sequence must be written to the watchdog key register in order to reset the counter before it overflows.

Watchdog Timer

- ◆ **Resets the device if the CPU crashes**
 - ◆ **Watchdog counter runs independently 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 set the PWM outputs to a high-impedance state, which will turn off the power converters in a properly designed system.

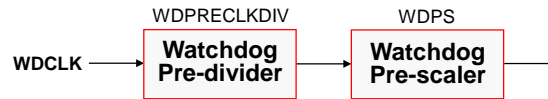
The watchdog timer starts running immediately after system power-up/reset, and must be dealt with by software soon after. Specifically, the watchdog must be serviced or disabled within 13.11 milliseconds (using a 10 MHz watchdog clock) after any reset before a watchdog initiated reset will occur. This translates into 131,072 watchdog clock 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 the pre-divider and then pre-scaled, if desired for slower watchdog time periods. A watchdog disable switch allows the watchdog to be enabled and disabled. Also a watchdog override switch provides an additional safety mechanism to insure the watchdog cannot be disabled. Once set, the only means to disable the watchdog is by a system reset.

During initialization, a value '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. The watchdog also contains an optional "windowing" feature that requires a minimum delay between counter resets.

Watchdog Pre-divider and Pre-scaler



```
SysCtl_setWatchdogPredivider(SYSCTL_WD_PREDIV_x);
```

where **x** is 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, or 4096
default = 512 (providing backwards compatibility)

```
SysCtl_setWatchdogPrescaler(SYSCTL_WD_PRESCALE_x);
```

where **x** is 1, 2, 4, 8, 16, 32, or 64
default = 1

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

Watchdog Driverlib Functions

- ◆ WDDIS – disable / enable
 - ◆ Functions only if WDOVERRIDE is not cleared
 - `SysCtl_disableWatchdog();`
 - `SysCtl_enableWatchdog();`
- ◆ WDOVERRIDE (clear only to protect – reset device to disable)
 - `SysCtl_clearWatchdogOverride();`
- ◆ Watchdog Mode – reset / interrupt
 - `SysCtl_setWatchdogMode(mode);`
 - ◆ mode parameter values:
 - SYSCTL_WD_MODE_RESET
 - SYSCTL_WD_MODE_INTERRUPT
- ◆ Watchdog Minimum Window
 - `SysCtl_setWatchdogWindowValue(value);`
 - ◆ value parameter sets a minimum delay between counter resets (0 = disabled)

Resetting the Watchdog

◆ Driverlib functions:

- ◆ `SysCtl_serviceWatchdog();` // writes 0x55 followed by 0xAA
// to WDKEY (watchdog reset)
- ◆ `SysCtl_enableWatchdogReset();` // writes 0x55 to WDKEY
- ◆ `SysCtl_resetWatchdog();` // writes 0xAA to WDKEY

◆ WDKEY write values:

0x55 - counter enabled for reset on next 0xAA write

0xAA - counter set to zero if reset enabled

◆ Writing any other value has no effect

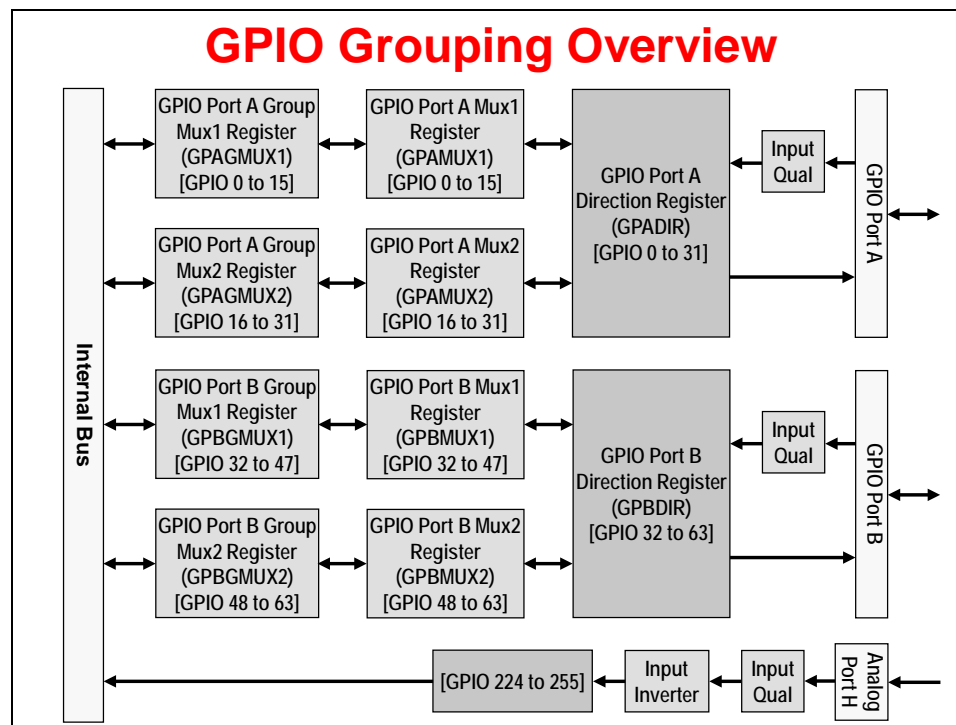
◆ Watchdog should not be serviced solely in an ISR

- ◆ If main code crashes, but interrupt continues to execute, the watchdog will not catch the crash
- ◆ Could put the 0x55 WDKEY in the main code, and the 0xAA WDKEY in an ISR; this catches main code crashes and also ISR crashes

WDKEY Write Results

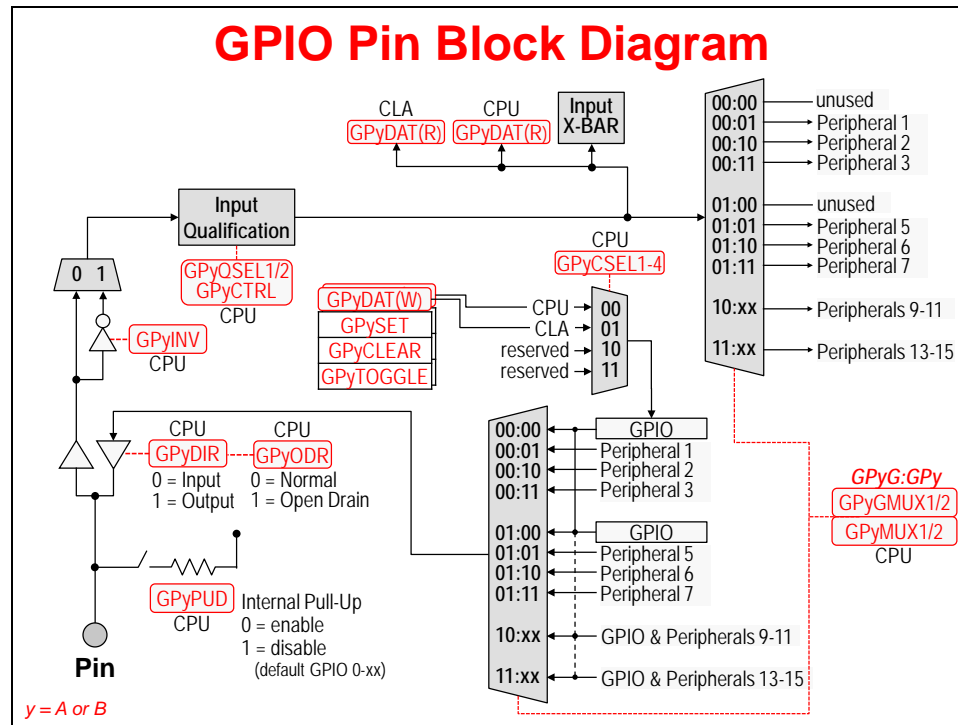
Sequential Step	Value Written to WDKEY	Result
1	0xAA	No action
2	0xAA	No action
3	0x55	WD counter enabled for reset on next AAh write
4	0x55	WD counter enabled for reset on next AAh write
5	0x55	WD counter enabled for reset on next AAh write
6	0xAA	WD counter is reset
7	0xAA	No action
8	0x55	WD counter enabled for reset on next AAh write
9	0xAA	WD counter is reset
10	0x55	WD counter enabled for reset on next AAh write
11	0x23	No effect; WD counter not reset on next AAh write
12	0xAA	No action due to previous invalid value
13	0x55	WD counter enabled for reset on next AAh write
14	0xAA	WD counter is reset

General Purpose Digital I/O



The F28004x device incorporates a multiplexing scheme to enable each I/O pin to be configured as a GPIO pin or one of several peripheral I/O signals. Sharing a pin across multiple functions maximizes application flexibility while minimizing package size and cost. A GPIO Group multiplexer and four GPIO Index multiplexers provide a double layer of multiplexing to allow up to twelve independent peripheral signals and a digital I/O function to share a single pin. Each output pin can be controlled by either a peripheral or either the CPU or CLA. By default, all of the pins are configured as GPIO, and when configured as a signal input pin, a qualification sampling period can be specified to remove unwanted noise. Optionally, each pin has an internal pullup resistor that can be enabled in order to keep the input pin in a known state when no external signal is driving the pin. The GPIO pins are grouped into two ports (Port A and Port B), and each port has 32 pins. For a GPIO, each port has a series of registers that are used to control the value on the pins, and within these registers each bit corresponds to one GPIO pin. Additionally, Analog Port H is an input only which has input qualification capability.

If the pin is configured as GPIO, a direction (DIR) register is used to specify the pin as either an input or output. By default, all GPIO pins are inputs. The current state of a GPIO pin corresponds to a bit value in a data (DAT) register, regardless if the pin is configured as GPIO or a peripheral function. Writing to the DAT register bit field clears or sets the corresponding output latch, and if the pin is configured as an output the pin will be driven either low or high. The state of various GPIO output pins on the same port can be easily modified using the SET, CLEAR, and TOGGLE registers. The advantage of using these registers is a single instruction can be used to modify only the pins specified without disturbing the other pins. This also eliminates any timing issues that may occur when writing directly to the data registers.



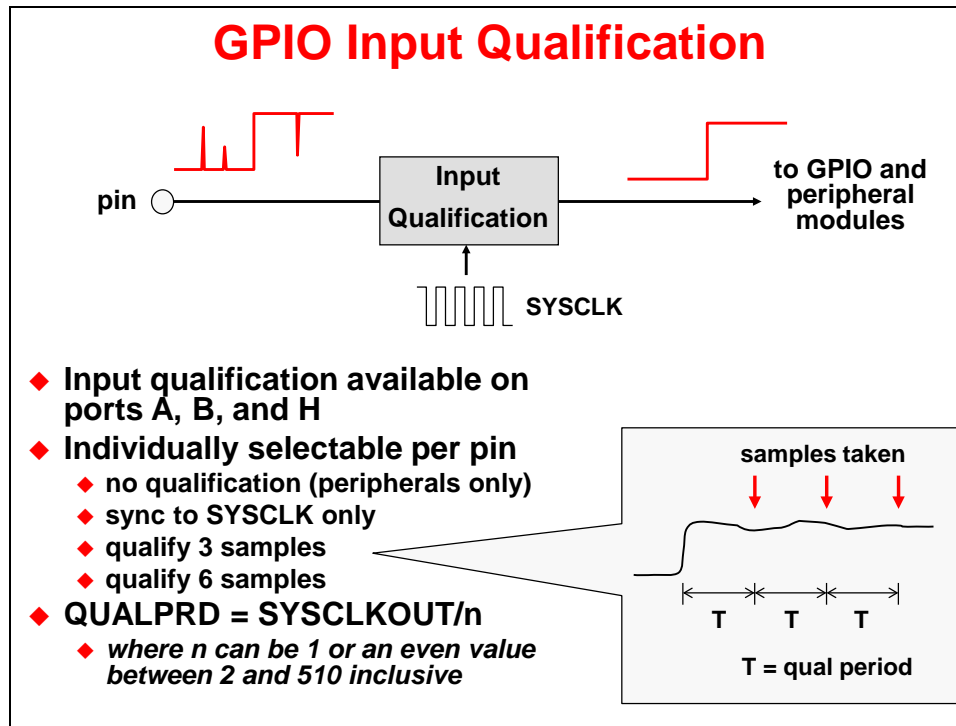
Configuring GPIO Pins

Configuring GPIO Pins using Driverlib

- ◆ **Configure peripheral multiplexing**
`GPIO_setPinConfig(pinConfig);`
 - ◆ *pinConfig* is defined in `pin_map.h` (GPIO_#_value)
- ◆ **Configure pin properties**
`GPIO_setPadConfig(pin, pinType);`
 - ◆ *pin* is the GPIO pin number
 - ◆ *pinType* can be the following values:
 - ◆ GPIO_PIN_TYPE_STD
 - ◆ GPIO_PIN_TYPE_PULLUP
 - ◆ GPIO_PIN_TYPE_OD
 - ◆ GPIO_PIN_TYPE_INVERT
 - ◆ INVERT may be OR-ed with STD or PULLUP
- ◆ **Set direction for pins configured as GPIO**
`GPIO_setDirectionMode(pin, pinIO);`
 - ◆ *pin* is the GPIO pin number
 - ◆ *pinIO* can be following values:
 - ◆ GPIO_DIR_MODE_IN
 - ◆ GPIO_DIR_MODE_OUT

The input qualification scheme is very flexible, and the type of input qualification can be configured for each GPIO pin individually. In the case of a GPIO input pin, the qualification can be specified as only synchronize to SYSCLOCKOUT or qualification by a sampling window. For pins

that are configured as peripheral inputs, the input can also be asynchronous in addition to synchronized to SYSCLKOUT or qualified by a sampling window.



Input Qualification Driverlib Functions

◆ Qualification Mode

`GPIO_setQualificationMode(pin, qualification);`

- ◆ *pin* is the GPIO pin number
- ◆ *qualification* values are:
 - ◆ GPIO_QUAL_SYNC
 - ◆ GPIO_QUAL_3SAMPLE
 - ◆ GPIO_QUAL_6SAMPLE
 - ◆ GPIO_QUAL_ASYNC

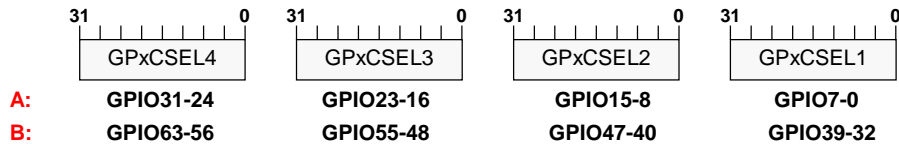
◆ Qualification Period

`GPIO_setQualificationPeriod(pin, divider);`

- ◆ *pin* is the GPIO pin number
- ◆ *divider* is the value by which the frequency of SYSCLKOUT is divided and it can be 1 or an even value between 2 and 510 inclusive

GPIO Core Select

- ◆ Selects which core's GPIODAT/SET/CLEAR/TOGGLE registers are used to control a pin
- ◆ Each pin individually controlled



- ◆ Driverlib function used to select core:

`GPIO_setMasterCore(pin, core);`

- ◆ `pin` is the GPIO pin number
- ◆ `core` parameter values:
 - ◆ `GPIO_CORE_CPU1`
 - ◆ `GPIO_CORE_CPU1_CLA1`

Driverlib GPIO Data Control Functions

◆ Pin Functions

`GPIO_readPin(pin);`

`GPIO_writePin(pin, outVal);`

`GPIO_togglePin(pin);`

- ◆ `pin` is the GPIO pin number
- ◆ `outVal` parameter is the value written to the pin

◆ Port Functions

`GPIO_readPortData(port);`

`GPIO_writePortData(port, outVal);`

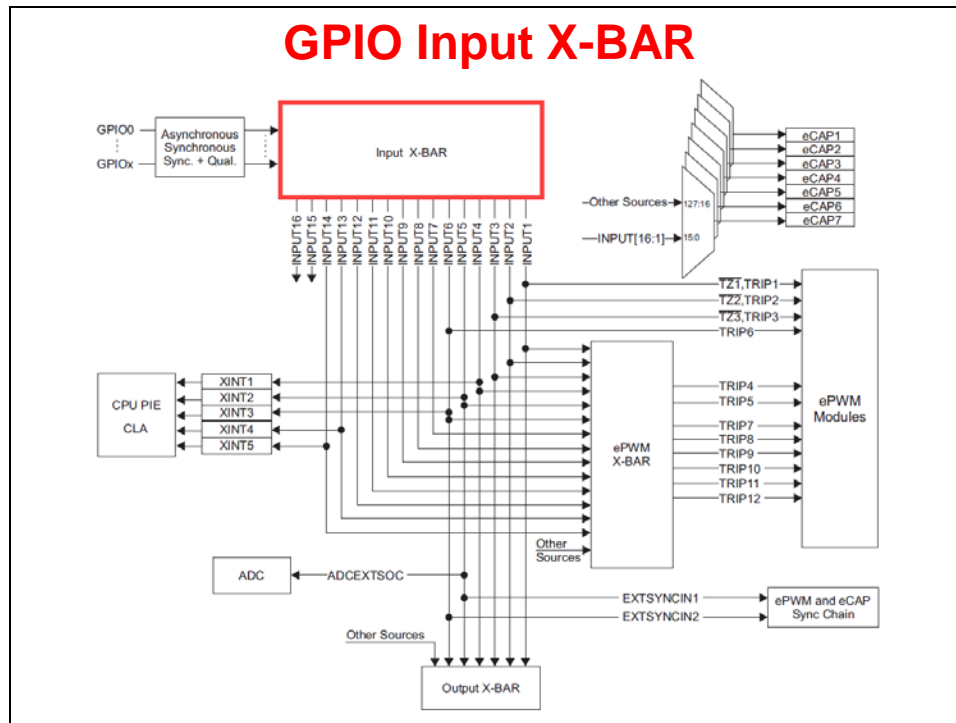
`GPIO_setPortPins(port, pinMask);`

`GPIO_clearPortPins(port, pinMask);`

`GPIO_togglePortPins(port, pinMask);`

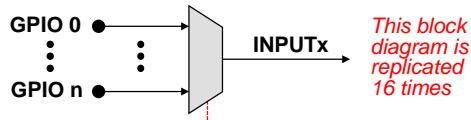
- ◆ `port` is the GPIO port: `GPIO_PORT_x` where `x` is the port letter
- ◆ `outVal` parameter is bit-packed value (32 pins) written to the port
- ◆ `pinMask` parameter is a bit-packed value (32 pins) masking the port

GPIO Input X-Bar



The Input X-BAR is used to route external GPIO signals into the device. It has access to every GPIO pin, where each signal can be routed to any or multiple destinations which include the ADCs, eCAPs, ePWMs, Output X-BAR, and external interrupts. This provides additional flexibility above the multiplexing scheme used by the GPIO structure. Since the GPIO does not affect the Input X-BAR, it is possible to route the output of one peripheral to another, such as measuring the output of an ePWM with an eCAP for frequency testing.

GPIO Input X-BAR Architecture

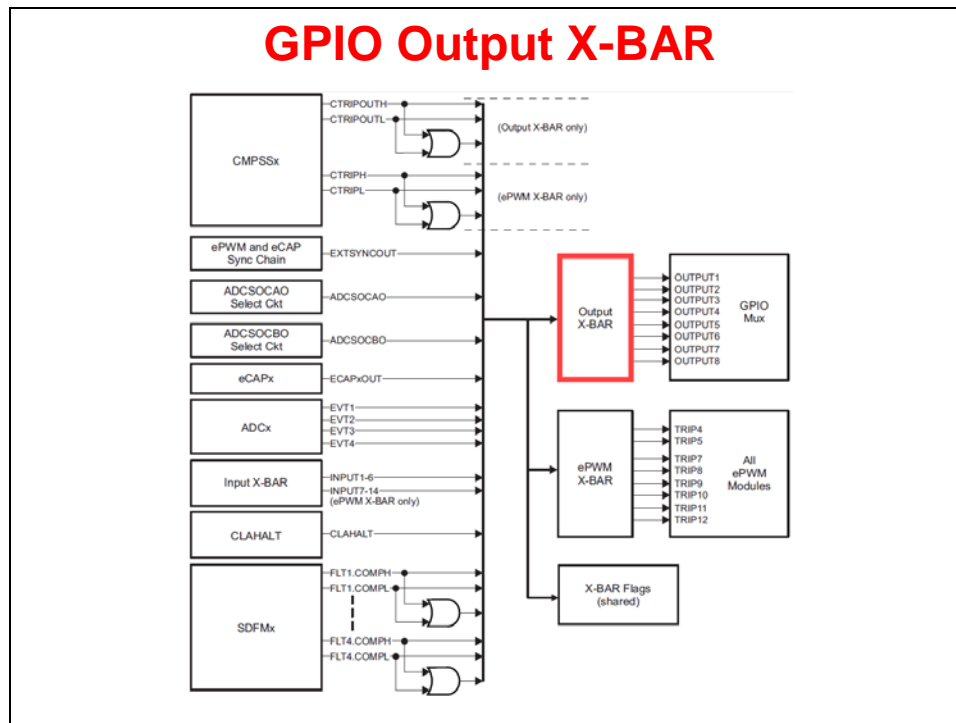


`XBAR_setInputPin(input, pin);`

<i>input</i>	Destinations (<i>pin</i> is the GPIO pin number)
XBAR_INPUT1	eCAPx, ePWM X-BAR, ePWM[TZ1, TRIP1], Output X-BAR
XBAR_INPUT2	eCAPx, ePWM X-BAR, ePWM[TZ2, TRIP2], Output X-BAR
XBAR_INPUT3	eCAPx, ePWM X-BAR, ePWM[TZ3, TRIP3], Output X-BAR
XBAR_INPUT4	eCAPx, ePWM X-BAR, XINT1, Output X-BAR
XBAR_INPUT5	eCAPx, ePWM X-BAR, XINT2, ADCEXTSOC, EXTSYNCIN1, Output X-BAR
XBAR_INPUT6	eCAPx, ePWM X-BAR, XINT3, ePWM[TRIP6], EXTSYNCIN2, Output X-BAR
XBAR_INPUT7	eCAPx, ePWM X-BAR
XBAR_INPUT8	eCAPx, ePWM X-BAR
XBAR_INPUT9	eCAPx, ePWM X-BAR
XBAR_INPUT10	eCAPx, ePWM X-BAR
XBAR_INPUT11	eCAPx, ePWM X-BAR
XBAR_INPUT12	eCAPx, ePWM X-BAR
XBAR_INPUT13	eCAPx, ePWM X-BAR, XINT4
XBAR_INPUT14	eCAPx, ePWM X-BAR, XINT5
XBAR_INPUT15	eCAPx
XBAR_INPUT16	eCAPx

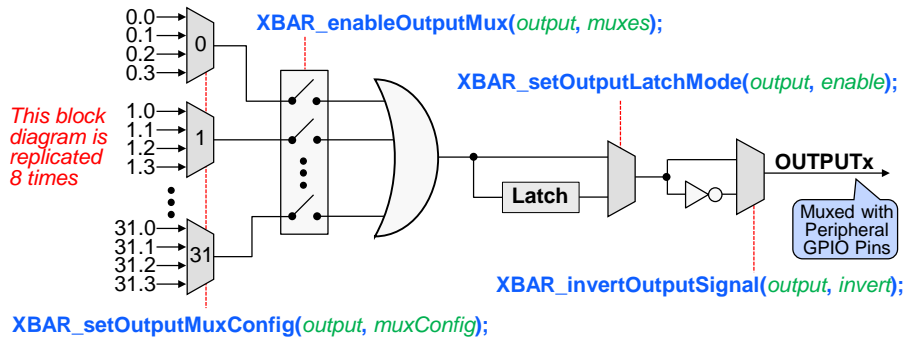
GPIO Output X-Bar

GPIO Output X-BAR



The Output X-BAR is used to route various internal signals out of the device. It contains eight outputs that are routed to the GPIO structure, where each output has one or multiple assigned pin positions, which are labeled as OUTPUTXBARx. Additionally, the Output X-BAR can select a single signal or logically OR up to 32 signals.

GPIO Output X-BAR Architecture



MUX	0	1	2	3
0	CMPSS1.CTRIPOUTH	CMPSS1.CTRIPOUTH_OR_CTRIPOUTL	ADCAEVT1	ECAP1OUT
1	CMPSS1.CTRIPOUTL	INPUTXBAR1	ADCCEV11	
2	CMPSS2.CTRIPOUTH	CMPSS2.CTRIPOUTH_OR_CTRIPOUTL	ADCAEVT2	ECAP2OUT
3	CMPSS2.CTRIPOUTL	INPUTXBAR2	ADCCEV12	
4	CMPSS3.CTRIPOUTH	CMPSS3.CTRIPOUTH_OR_CTRIPOUTL	ADCAEVT3	ECAP3OUT
5	CMPSS3.CTRIPOUTL	INPUTXBAR3	ADCCEV13	
6	CMPSS4.CTRIPOUTH	CMPSS4.CTRIPOUTH_OR_CTRIPOUTL	ADCAEVT4	ECAP4OUT
7	CMPSS4.CTRIPOUTL	INPUTXBAR4	ADCCEV14	
8	CMPSS5.CTRIPOUTH	CMPSS5.CTRIPOUTH_OR_CTRIPOUTL	ADCBVT1	ECAP5OUT
9	CMPSS5.CTRIPOUTL	INPUTXBAR5		
10	CMPSS6.CTRIPOUTH	CMPSS6.CTRIPOUTH_OR_CTRIPOUTL	ADCBVT2	ECAP6OUT
11	CMPSS6.CTRIPOUTL	INPUTXBAR6		
12	CMPSS7.CTRIPOUTH	CMPSS7.CTRIPOUTH_OR_CTRIPOUTL	ADCBVT3	ECAP7OUT
13	CMPSS7.CTRIPOUTL	ADCSOCAD		
14			ADCBVT4	EXTSYNCOUT
15		ADCSOCB0		

MUX	0	1	2	3
16	SD1FLT1.COMPH	SD1FLT1.COMPH_OR_COMPL		
17	SD1FLT1.COMPL			CLAHALT
18	SD1FLT2.COMPH	SD1FLT2.COMPH_OR_COMPL		
19	SD1FLT2.COMPL			
20	SD1FLT3.COMPH	SD1FLT3.COMPH_OR_COMPL		
21	SD1FLT3.COMPL			
22	SD1FLT4.COMPH	SD1FLT4.COMPH_OR_COMPL		
23	SD1FLT4.COMPL			
24				
25				
26				
27				
28				
29				
30				
31				

External Interrupts

External Interrupts

- ◆ 5 external interrupt signals
 - ◆ XINT1, XINT2, XINT3, XINT4 and XINT5
- ◆ Each external interrupt can be mapped to any of the GPIO pins via the X-BAR Input architecture
 - ◆ XINT1-5 are sources for Input X-BAR signals 4, 5, 6, 13, and 14 respectively
- ◆ XINT1, XINT2, and XINT3 also have a free-running 16-bit counter which measures the elapsed time between interrupts
 - ◆ Counter resets to zero each time the interrupt occurs

- ◆ Driverlib function used to read counter value:

```
GPIO_getInterruptCounter(extIntNum);
```

- ◆ *extIntNum* parameter is: GPIO_INT_XINT x ($x = 1, 2, \text{ or } 3$)

Configuring External Interrupts

- ◆ Configuring external interrupts is a multi-step process:
 - ◆ Select GPIO pin, set polarity, and enable interrupt

```
GPIO_setInterruptPin(pin, extIntNum);
```

```
GPIO_setInterruptType(extIntNum, intType);
```

```
GPIO_[enable|disable]Interrupt(extIntNum);
```

- ◆ *pin* is the GPIO pin number
- ◆ *extIntNum* parameter specifies the external interrupt
 - ◆ GPIO_INT_XINT1
 - ◆ GPIO_INT_XINT2
 - ◆ GPIO_INT_XINT3
 - ◆ GPIO_INT_XINT4
 - ◆ GPIO_INT_XINT5
- ◆ *intType* parameter specifies the type of interrupt trigger
 - ◆ GPIO_INT_TYPE_FALLING_EDGE
 - ◆ GPIO_INT_TYPE_RISING_EDGE
 - ◆ GPIO_INT_TYPE_BOTH_EDGES

Low Power Modes

Low Power Modes

Low Power Mode	CPU Logic Clock	Peripheral Logic Clock	Watchdog Clock	PLL	INTOSC 1/2	XTAL
Normal Run	on	on	on	on	on	on
IDLE	off	on	on	on	on	on
HALT	off	off	off	off	on	on

- ◆ **HALT**
 - ◆ INTOSC – not automatically powered down; software configurable
 - ◆ XTAL – can be powered down by software at any time
- ◆ **STANDBY is not supported but can be emulated – see device Technical Reference Manual**
- ◆ **See device data sheet for each low power mode power consumption**

Low Power Mode Exit

Low Power Mode \ Exit Interrupt	Reset	GPIO 0 - 63 Signal	Watchdog Interrupt	Any Enabled Interrupt
IDLE	yes	no	yes	yes
HALT	yes	yes	no	no

Low Power Mode Driverlib Functions

- ◆ **Configuring low power mode**

 - `SysCtl_enterIdleMode();` //enter IDLE mode

 - `SysCtl_enterHaltMode();` //enter HALT mode

- ◆ **Set pin to wake up device from HALT mode**

 - `SysCtl_enableLPMWakeupPin(pin);`

 - `SysCtl_disableLPMWakeupPin(pin);`

 - ◆ *pin* is the GPIO pin number (numerical value 0-63)

- ◆ **Run watchdog while in HALT mode**

 - `SysCtl_enableWatchdogInHalt();`

 - `SysCtl_disableWatchdogInHalt();`

Register Protection

LOCK Protection Registers

- ◆ “LOCK” registers protects several system configuration registers from spurious CPU writes
- ◆ Once the LOCK Driverlib functions are set the respective locked registers can no longer be modified by software:

ASysCtl_lockTemperatureSensor	ASysCtl_lockCMPHNmux	PGA_lockRegisters
ASysCtl_lockANAREF	ASysCtl_lockCMPLNmux	SysCtl_lockAccessControlRegs
ASysCtl_lockVMON	ASysCtl_lockVREG	SysCtl_lockSyncSelect
ASysCtl_lockDCDC	DAC_lockRegister	XBAR_lockInput
ASysCtl_lockPGAADCINmux	EPWM_lockRegisters	XBAR_lockOutput
ASysCtl_lockCMPHPmux	HRPWM_lockRegisters	XBAR_lockEPWM
ASysCtl_lockCMPLPmux	MemCfg_commitConfig	

- ◆ The following Driverlib functions can be locked/unlocked:

FSI_lockTxCtrl	GPIO_lockPortConfig	MemCfg_lockConfig
FSI_lockRxCtrl	GPIO_unlockPortConfig	MemCfg_unlockConfig
SysCtl_lockExtADCSOCSelect		

A series of “lock” registers can be used to protect several system configuration settings from spurious CPU writes. After the lock registers bits are set, the respective locked registers can no longer be modified. However, some registers have lock/unlock capability.

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)

- ◆ ***Driverlib functions automatically take care of EALLOW and EDIS protection***
- ◆ **The following registers are protected:**

Device Configuration & Emulation

Flash

Code Security Module

PIE Vector Table

DMA, CLA, SD, EMIF, X-Bar (some registers)

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

ePWM, CMPSS, ADC, DAC (some registers)

GPIO (control registers only)

System Control

See device data sheet and Technical Reference Manual for detailed listings

Lab 5: System Initialization

➤ Objective

The objective of this lab exercise 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 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 interrupt process will be tested by using the watchdog to generate an interrupt. This lab will make use of the F28004x Driver Library (Driverlib) to simplify the programming of the device. 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 “TMS320F280049C”, and leaving the “Connection” box blank). Name the project **Lab5**. Uncheck the “Use default location” box. Using the “Browse...” button navigate to: C:\F28004x\Labs\Lab5\project then click Select Folder. 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:\F28004x\Labs\Lab5\source:

CodeStartBranch.asm	Lab_5_6_7.cmd
DefaultIsr_5.c	Main_5.c
device.c	Watchdog_5.c
Gpio.c	

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 Driverlib files and common lab header files. Under “C2000 Compiler” select “Include Options”. In the include search path 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 (*one at a time*):

```
${PROJECT_ROOT}/../../../../f28004x_driverlib/driverlib
${PROJECT_ROOT}/../../../../f28004x_driverlib/driverlib/inc
${PROJECT_ROOT}/../../../../Lab_common/include
```

Click OK to include each search path.

4. Next, we need to setup the file search path for Driverlib. Under “C2000 Linker” select “File Search Path”. The file search path box will open and in the include library file section (“Include library file or command file as input”) click the Add icon. Then in the “Add file path” window type:

driverlib.lib

and click OK. Then in the library search path section ("Add <dir> to library search path") click the Add icon. In the "Add directory path" window type:

```
${PROJECT_ROOT}/../../f28004x_driverlib/driverlib/ccs/Debug
```

and click OK.

- Now, we need to setup the predefined symbols. Under "C2000 Compiler" select "Predefined Symbols". In the predefined name box that opens ("Pre-define NAME") click the Add icon. Then in the "Enter Value" window type **_LAUNCHXL_F280049C**. This name is used in the project to conditionally include #defines for pin numbers and other GPIO configuration code specific to the LaunchPad (rather than the controlCARD). This conditional code is located in the `device.h` file. Click OK to include the name. Finally, click Apply and Close to save and close the Properties window.

Memory Configuration

- 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 "RAM" branches to address `0x000000` upon bootloader completion.

Notice that the linker command file `Lab_5_6_7.cmd` has a memory block named `BEGIN_M0: origin = 0x000000, length = 0x0002`, in program memory. The existing parts of memory blocks `BOOT_RSVD` and `RAMM0` in data memory has been modified to avoid any overlaps with this memory block.

- 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. Close the `Lab_5_6_7.cmd` linker command file.

System Initialization

- Open and inspect `main_5.c`. Notice the `Device_init()` function call to `device.c` for initializing the device.
- Open `Watchdog_5.c` and edit the file to configure the watchdog for generating a reset. Also, edit the file to disable the watchdog. Make the modifications to the file at the appropriate locations in the code. Save your work.
- Open and inspect `Gpio.c`. Notice the `Driverlib` functions that are being used to configure the GPIO pins. Also, notice the input X-BAR configuration. This file will be used in the remaining lab exercises.

Build and Load

- Click the "Build" button and watch the tools run in the Console window. Check for errors in the Problems window.
- Click the "Debug" button (green bug). The CCS Debug perspective view should open, the program will load automatically, and you should now be at the start of `main()`.



13. After CCS loaded the program in the previous step, it set the program counter (PC) to point to `_c_int00`. It then ran through the C-environment initialization routine in the `rts2800_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 “RAMM0” at address `0x000000`. Set the bootloader mode using the menu bar by clicking:

Scripts → EMU Boot Mode Select → `EMU_BOOT_RAM`

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_RAM`.

Run the Code – Watchdog Reset Disabled

14. 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.
15. 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. (Alternatively, 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 (or become trapped as explained in the watchdog hardware reset below).
16. 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”.

Run the Code – Watchdog Reset Enabled

17. Open the Project Explorer window in the CCS Debug perspective view by selecting `View → Project Explorer`. In `Watchdog_5.c` add the `Driverlib` function to enable the watchdog. This will enable the watchdog to function and cause a reset. Save the file.
18. Build the project by clicking `Project → Build Project`. Select `Yes` to “Reload the program automatically”.

Alternatively, you can add the “Build” button to the tool bar in the CCS Debug perspective (if it is not already there) so that it will be available for future use. Click `Window → Perspective → Customize Perspective...` and then select the `Tool Bar Visibility` tab. Check the `Code Composer Studio Project Build` box. This will automatically select the “Build” button in the `Tool Bar Visibility` tab. Click `OK`.

19. Again, 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`.
20. This time we will have the watchdog issue a reset that will toggle the `XRSn` pin (i.e. perform a hardware reset). Now 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 RAMM0 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 RAMM0, and execution continued until the breakpoint was hit in `main()`.

Configure 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.

21. In `Main_5.c` notice the two function calls to `interrupt.c` for initializing the PIE registers and PIE vectors:

```
Interrupt_initModule();
Interrupt_initVectorTable();
```

22. Modify `main()` to enable global interrupts at the appropriate location in the code.
23. In `Watchdog_5.c` add the Driverlib function to cause the watchdog to generate an interrupt rather than a reset.
24. Using the “PIE Interrupt Assignment Table” shown in the previous module find the location for the watchdog interrupt “`INT_WAKE`” and fill in the following information:

PIE group #: _____ # within group: _____

This will be used in the next step.

25. Next modify `Watchdog_5.c` at the appropriate locations in the code as follows:
- Add the Driverlib function to re-map the watchdog interrupt signal to call the ISR function. (Hint: `#define` name in `driverlib/inc/hw_ints.h` and label name in `DefaultIsr_5.c`)
 - Add the Driverlib function to enable the appropriate PIEIER and core IER
26. Save all changes to the files.
27. Inspect `DefaultIsr_5.c`. This file contains interrupt service routines. The ISR for WAKE interrupt 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.

Build and Load

28. Build the project by clicking `Project` → `Build Project`, or by clicking on the “Build” button (if it has been added to the tool bar). Select `Yes` to “Reload the program automatically”.

Run the Code – Watchdog Interrupt

29. Place the cursor in the “main loop” section, right click the mouse key and select `Run To Line`.
30. 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 `wakeISR()`.

Terminate Debug Session and Close Project

31. Terminate the active debug session using the `Terminate` button. This will close the debugger and return Code Composer Studio to the CCS Edit perspective view.
32. 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. During this lab exercise, the watchdog was actually re-enabled (or disabled again) in the file `Watchdog_5.c`.

Analog Subsystem

Introduction

The Analog Subsystem consists of the Analog-to-Digital Converter (ADC), Comparator Subsystem (CMPSS), Programmable Gain Amplifier (PGA), Digital-to-Analog Converter (DAC), and the Analog Subsystem Interconnect. This module will explain the operation of each subsystem. The lab exercise will use the ADC to perform data acquisition.

Module Objectives

Module Objectives

- ◆ Understand the operation of the:
 - ◆ Analog-to-Digital Converter (ADC)
 - ◆ Comparator Subsystem (CMPSS)
 - ◆ Programmable Gain Amplifier (PGA)
 - ◆ Digital-to-Analog Converter (DAC)
 - ◆ Analog Subsystem Interconnect
- ◆ Use the ADC to perform data acquisition

Analog Subsystem:

- Three 12-Bit Analog-to-Digital Converters (ADCs)
 - 3.45 MSPS each (up to 10.35 MSPS per system)
 - Selectable internal reference of 2.5v or 3.3v
 - Ratiometric external reference set by VREFHI/VREFLO
- Seven Comparator Subsystems (CMPSS)
 - Each contains:
 - Two analog comparators
 - Two programmable 12-bit reference DACs
 - One ramp generator and Two digital glitch filter
- Seven Programmable Gain Amplifiers (PGAs)
 - Each features:
 - Four programmable gain modes: 3x, 6x, 12x, 24x
 - Programmable output filtering
- Two 12-bit Buffered Digital-to-Analog Converter Outputs (DACs)
 - Selectable reference voltage

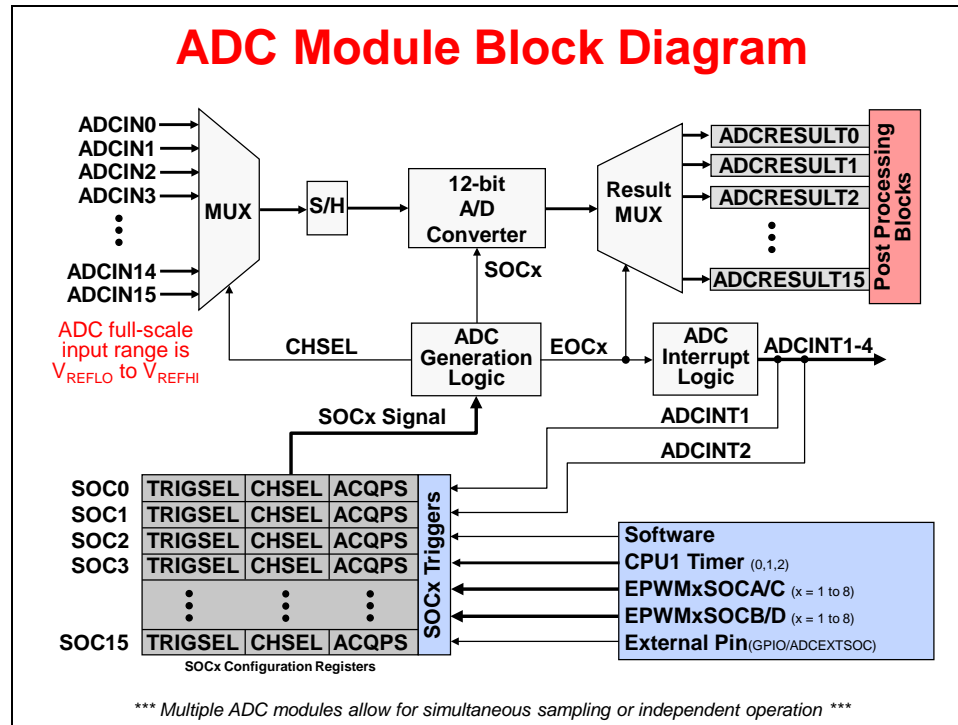
Chapter Topics

Analog Subsystem	6-1
<i>Analog-to-Digital Converter (ADC)</i>	6-3
ADC Module Block Diagram	6-3
ADC Triggering	6-5
ADC Conversion Priority	6-7
Post Processing Block	6-10
ADC Clocking Flow	6-12
ADC Timing	6-13
ADC Conversion Result Registers	6-13
Signed Input Voltages	6-14
Built-In ADC Calibration	6-14
Analog Subsystem External Reference	6-15
<i>Comparator Subsystem (CMPSS)</i>	6-16
Comparator Subsystem Block Diagram.....	6-17
<i>Programmable Gain Amplifier (PGA)</i>	6-18
PGA Block Diagram	6-18
<i>Digital-to-Analog Converter (DAC)</i>	6-19
Buffered DAC Block Diagram.....	6-20
<i>Analog Subsystem Interconnect</i>	6-21
Analog Group Connections	6-22
Analog Group Connection – Example.....	6-23
<i>Lab 6: Analog-to-Digital Converter</i>	6-25

Analog-to-Digital Converter (ADC)

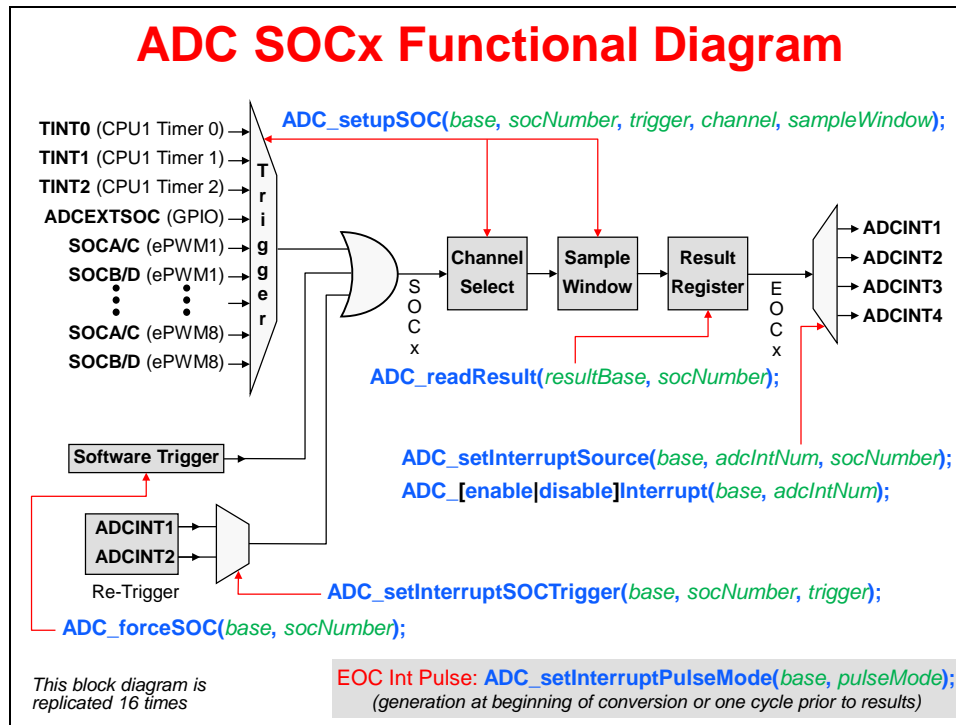
The F28004x includes three independent high-performance ADC modules. Each ADC module has a single sample-and-hold (S/H) circuit and using multiple ADC modules enables simultaneous sampling or independent operation (sequential sampling). The ADC module is implemented using a successive approximation (SAR) type ADC with a resolution of 12-bits and a performance of 3.45 MSPS, yielding up to 10.35 MSPS for the device.

ADC Module Block Diagram



The ADC triggering and conversion sequencing is managed by a series of start-of-conversion (SOCx) configuration registers. Each SOCx register configures a single channel conversion, where the SOCx register specifies the trigger source that starts the conversion, the channel to convert, and the acquisition sample window duration. Multiple SOCx registers can be configured for the same trigger, channel, and/or acquisition window. Configuring multiple SOCx registers to use the same trigger will cause that trigger to perform a sequence of conversions, and configuring multiple SOCx registers for the same trigger and channel can be used to oversample the signal.

The various trigger sources that can be used to start an ADC conversion include the General-Purpose Timers, the ePWM modules, an external pin, and by software. Also, the flag setting of either ADCINT1 or ADCINT2 can be configured as a trigger source which can be used for continuous conversion operation. The ADC interrupt logic can generate up to four interrupts. The results for SOC 0 through 15 appear in result registers 0 through 15, respectively.



The figure above is a conceptual view highlighting a single ADC start-of-conversion functional flow from triggering to interrupt generation. This figure is replicated 16 times and the Driverlib functions highlight the sections that they modify.

ADC SOC Driverlib Function

- ◆ **Configure a start-of-conversion (SOC)**

`ADC_setupSOC(base, socNumber, trigger, channel, sampleWindow);`

- ◆ *base* is the ADC base address: ADC_x_BASE (x = A to C)
- ◆ *socNumber* values are:
 - ◆ ADC_SOC_NUMBER_x (x = 0 to 15)
- ◆ *trigger* values are:
 - ◆ ADC_TRIGGER_SW_ONLY
 - ◆ ADC_TRIGGER_CPU1_TINT_x (x = 0 to 2)
 - ◆ ADC_TRIGGER_GPIO
 - ◆ ADC_TRIGGER_EPWM_x_SOCA (x = 1 to 8)
 - ◆ ADC_TRIGGER_EPWM_x_SOCB (x = 1 to 8)
- ◆ *channel* values are:
 - ◆ ADC_CH_ADCIN_x (x = 0 to 15)
- ◆ *sampleWindow* parameter is the acquisition window duration in SYSCLK cycles: value between 1 and 512 cycles inclusive (Note: see data sheet for minimum value – the F28004x has a minimum value of 8 cycles)

ADC Driverlib Functions

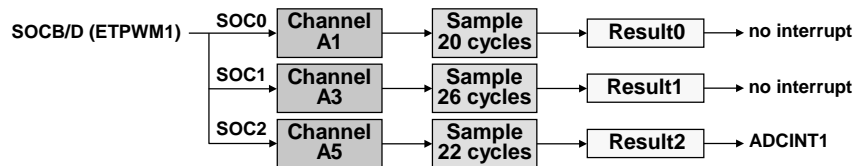
- ◆ Configure an interrupt start-of-conversion trigger
`ADC_setInterruptSOCTrigger(base, socNumber, trigger);`
- ◆ Set EOC source for an ADC interrupt
`ADC_setInterruptSource(base, adcIntNum, socNumber);`
`ADC_[enable|disable]Interrupt(base, adcIntNum);`
- ◆ Force an SOC conversion (software trigger)
`ADC_forceSOC(base, socNumber);`
- ◆ Configure ADC EOC interrupt pulse generation
`ADC_setInterruptPulseMode(base, pulseMode);`
- ◆ Read ADC result register
`ADC_readResult(resultBase, socNumber);`

- ◆ *base* is the ADC base address: ADC_x_BASE (x = A to C)
- ◆ *socNumber* is: ADC_SOC_NUMBER_x (x = 0 to 15)
- ◆ *Trigger* value is:
 - ◆ ADC_INT_SOC_TRIGGER_NONE
 - ◆ ADC_INT_SOC_TRIGGER_ADCINT_x (x = 1 or 2)
- ◆ *adcIntNum* value is: ADC_INT_NUMBER_x (x = 1 to 4)
- ◆ *pulseMode* value is: ADC_PULSE_END_OF__x (x = ACQ_WIN or CONV)
- ◆ *resultBase* value is: ADC_xRESULT_BASE (x = A to C)

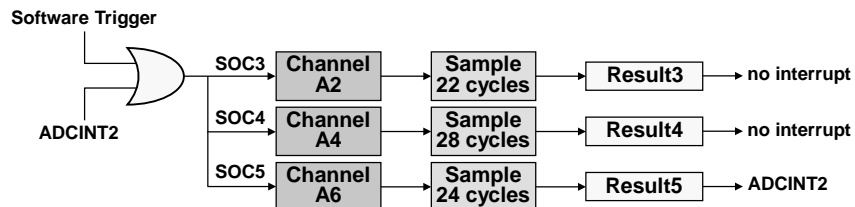
ADC Triggering

Example – ADC Triggering

Sample A1 → A3 → A5 when ePWM1 SOCB/D is generated and then generate ADCINT1:



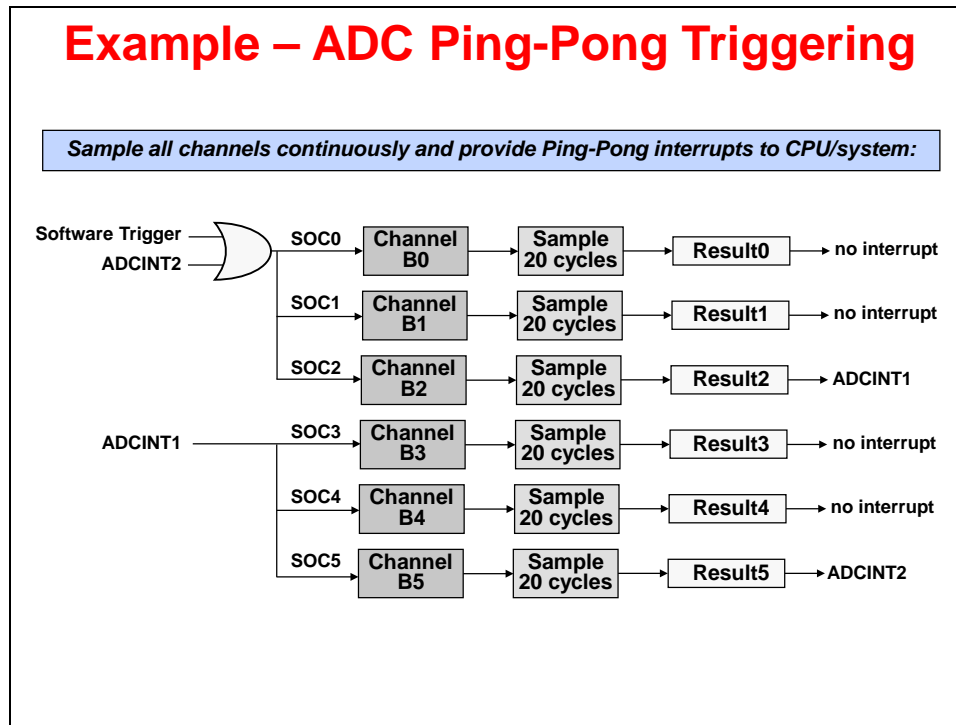
Sample A2 → A4 → A6 continuously and generate ADCINT2:



Note: setting ADCINT2 flag does not need to generate an interrupt

The top example in the figure above shows channels A1, A3, and A5 being converted with a trigger from EPWM1. After A5 is converted, ADCINT1 is generated. The bottom example shows channels A2, A4, and A6 being converted initially by a software trigger. Then, after A6 is

converted, ADCINT2 is generated and also fed back as a trigger to start the process again.



The ADC ping-pong triggering example in the figure above shows channels B0 through B5 being converted, triggered initially by software. After channel B2 is converted, ADCINT1 is generated, which also triggers channel B3. After channel B5 is converted, ADCINT2 is generated and is also fed back to start the process again from the beginning. Additionally, ADCINT1 and ADCINT2 are being used to manage the ping-pong interrupts for the interrupt service routines.

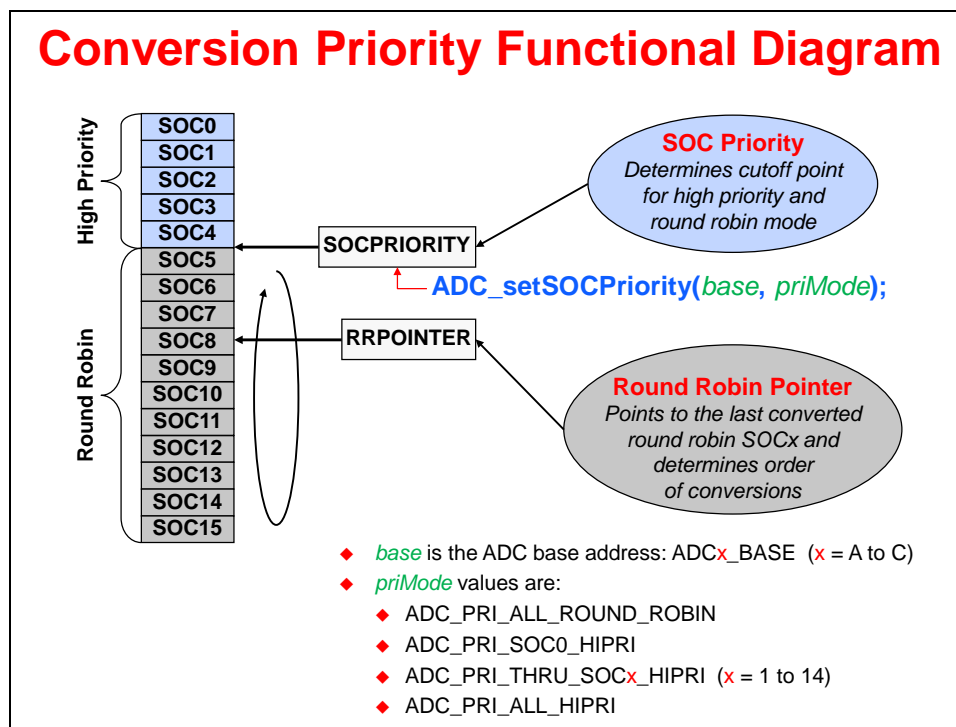
ADC Conversion Priority

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
 - ◆ **Round Robin Burst Mode**
 - ◆ Allows a single trigger to convert one or more SOC's in the round robin wheel
 - ◆ Uses BURSTTRIG instead of TRIGSEL for all round robin SOC's (not high priority)

When multiple triggers are received at the same time, the ADC conversion priority determines the order in which they are converted. Three different priority modes are supported. The default priority mode is round robin, where no start-of-conversion has an inherently higher priority over another, and the priority depends upon a round robin pointer. The round robin pointer operates in a circular fashion, constantly wrapping around to the beginning. In high priority mode, one or more than one start-of-conversion is assigned as high priority. The high priority start-of-conversion can then interrupt the round robin wheel, and after it has been converted the wheel will continue where it was interrupted. High priority mode is assigned first to the lower number start-of-conversion and then in increasing numerical order. If two high priority start-of-conversion triggers occur at the same time, the lower number will take precedence. Burst mode allows a single trigger to convert one or more than one start-of-conversion sequentially at a time. This mode uses a separate Burst Control register to select the burst size and trigger source.

Conversion Priority Functional Diagram



In this conversion priority functional diagram, the Start-of-Conversion Priority Control Register contains two bit fields. The Start-of-Conversion Priority bit fields determine the cutoff point between high priority and round robin mode, whereas the Round-Robin Pointer bit fields contains the last converted round robin start-of-conversion which determines the order of conversions.

Round Robin Priority Example

SOC PRIORITY configured as 0;
RRPOINTER configured as 15;
SOC0 is highest RR priority

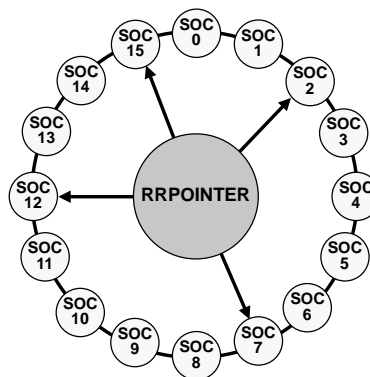
SOC7 trigger received

SOC7 is converted;
RRPOINTER now points to SOC7;
SOC8 is now highest RR priority

SOC2 & SOC12 triggers received simultaneously

SOC12 is converted;
RRPOINTER points to SOC12;
SOC13 is now highest RR priority

SOC2 is converted;
RRPOINTER points to SOC2;
SOC3 is now highest RR priority



High Priority Example

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

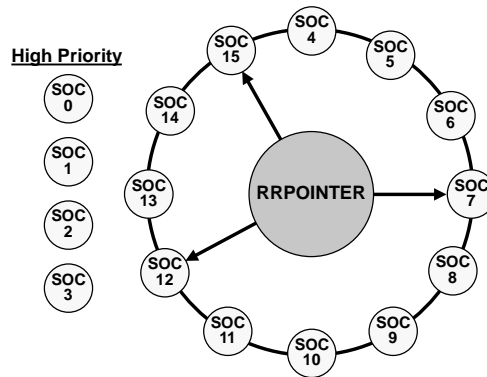
SOC 7 trigger received

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

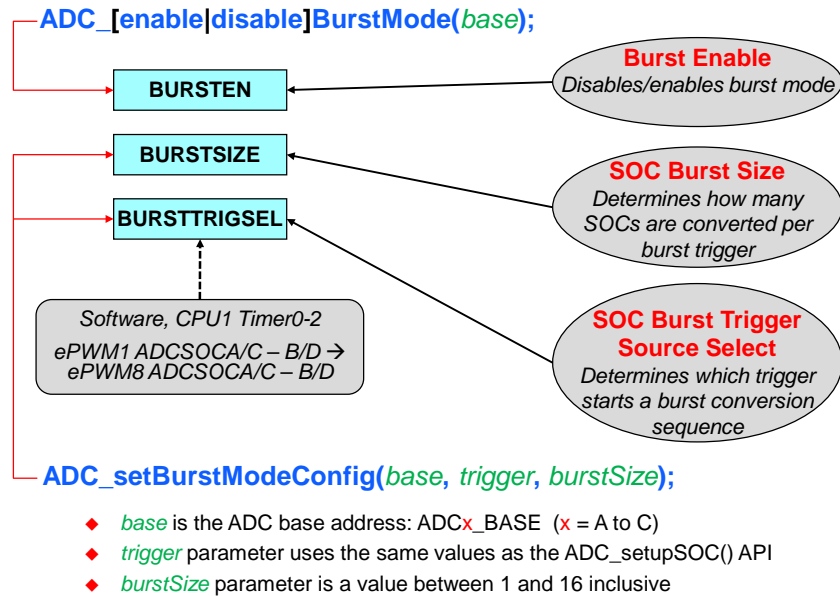
SOC 2 & SOC 12 triggers received simultaneously

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



Round Robin Burst Mode Diagram



The Round-Robin Burst mode utilizes an ADC Burst Control register to enable the burst mode, determine the burst size, and select the burst trigger source. This register is modified using the two Driverlib functions shown in the figure.

Round Robin Burst Mode with High Priority Example

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

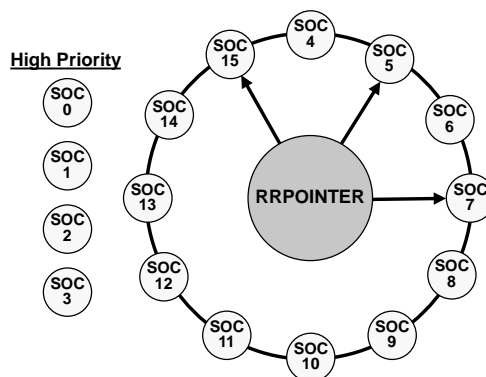
BURSTTRIG trigger received

SOC 4 & SOC 5 is converted;
RR POINTER points to SOC 5;
SOC 6 is now highest RR priority

BURSTTRIG & SOC 1 triggers received simultaneously

SOC 1 is converted;
RR POINTER stays pointing to SOC 5

SOC 6 & SOC 7 is converted;
RR POINTER points to SOC 7;
SOC 8 is now highest RR priority

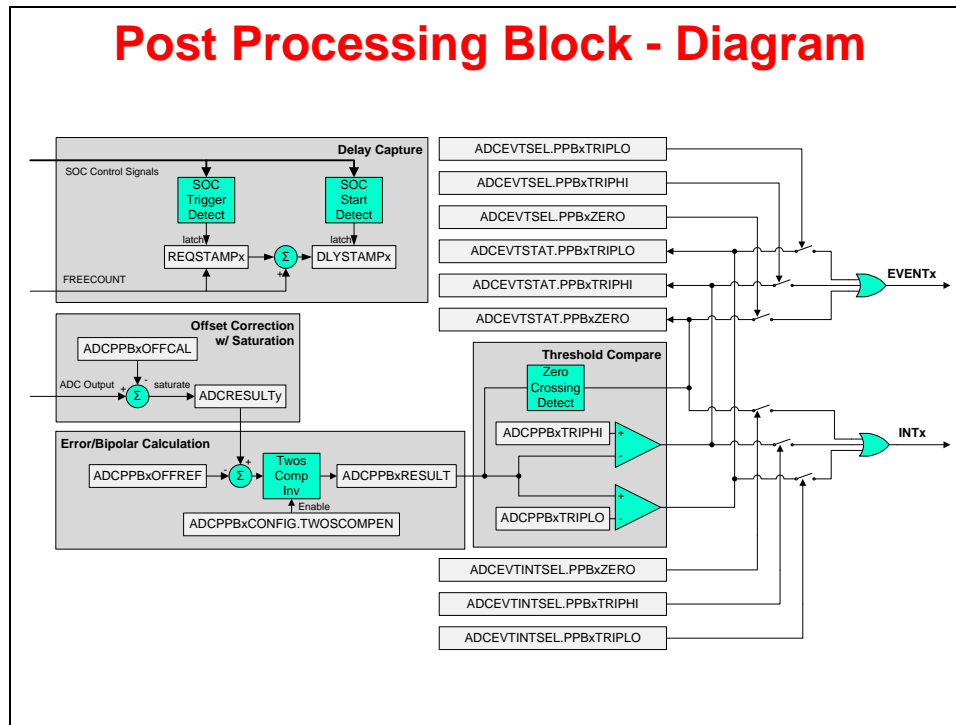


Note: `enableBurstMode` and `burstSize = 2`

Post Processing Block

Purpose of the Post Processing Block

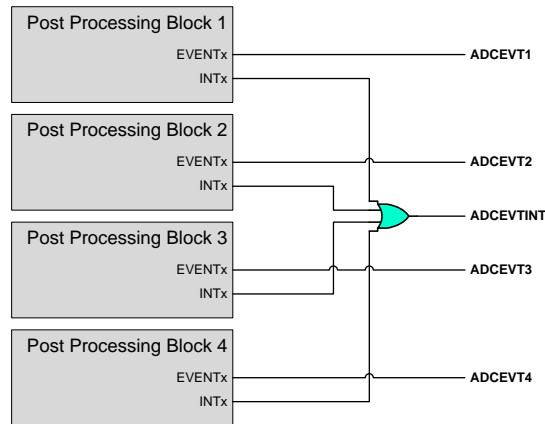
- ◆ **Offset Correction**
 - ◆ *Remove an offset associated with an ADCIN channel possibly caused by external sensors and signal sources*
 - ◆ Zero-overhead; saving cycles
- ◆ **Error from Set-point Calculation**
 - ◆ *Subtract out a reference value which can be used to automatically calculate an error from a set-point or expected value*
 - ◆ Reduces the sample to output latency and software overhead
- ◆ **Limit and Zero-Crossing Detection**
 - ◆ *Automatically perform a check against a high/low limit or zero-crossing and can generate a trip to the ePWM and/or an interrupt*
 - ◆ Decreases the sample to ePWM latency and reduces software overhead; trip the ePWM based on an out of range ADC conversion without CPU intervention
- ◆ **Trigger-to-Sample Delay Capture**
 - ◆ *Capable of recording the delay between when the SOC is triggered and when it begins to be sampled*
 - ◆ Allows software techniques to reduce the delay error



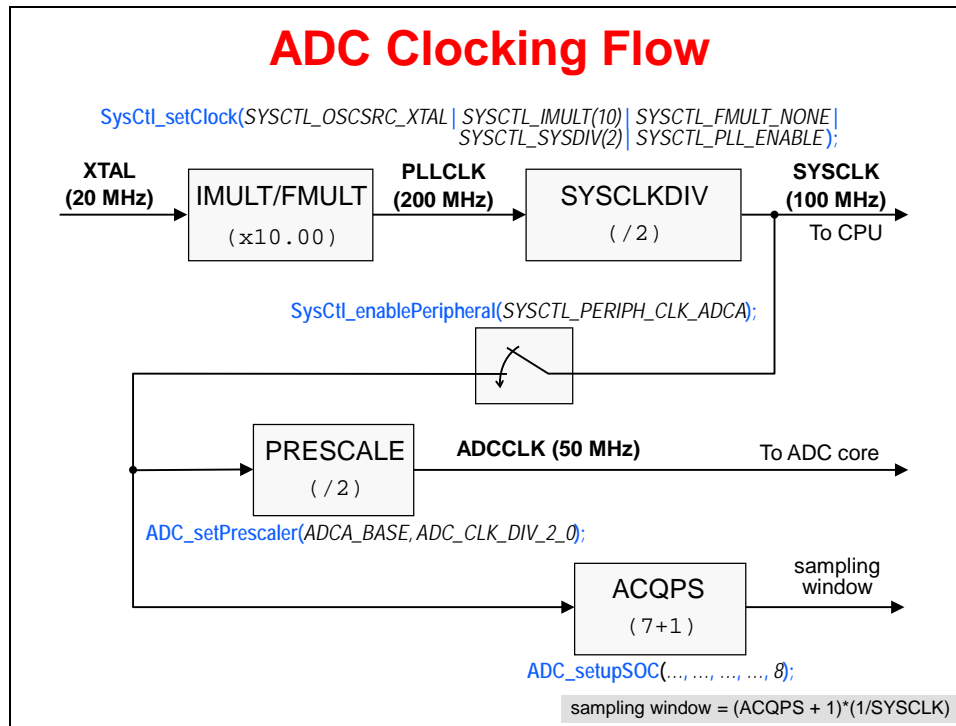
To further enhance the capabilities of the ADC, each ADC module incorporates four post-processing blocks (PPB), and each PPB can be linked to any of the ADC result registers. The PPBs can be used for offset correction, calculating an error from a set-point, detecting a limit and zero-crossing, and capturing a trigger-to-sample delay. Offset correction can simultaneously remove an offset associated with an ADCIN channel that was possibly caused by external sensors or signal sources with zero-overhead, thereby saving processor cycles. Error calculation can automatically subtract out a computed error from a set-point or expected result register value, reducing the sample to output latency and software overhead. Limit and zero-crossing detection automatically performs a check against a high/low limit or zero-crossing and can generate a trip to the ePWM and/or generate an interrupt. This lowers the sample to ePWM latency and reduces software overhead. Also, it can trip the ePWM based on an out-of-range ADC conversion without any CPU intervention which is useful for safety conscious applications. Sample delay capture records the delay between when the SOCx is triggered and when it begins to be sampled. This can enable software techniques to be used for reducing the delay error.

Post Processing Block Interrupt Event

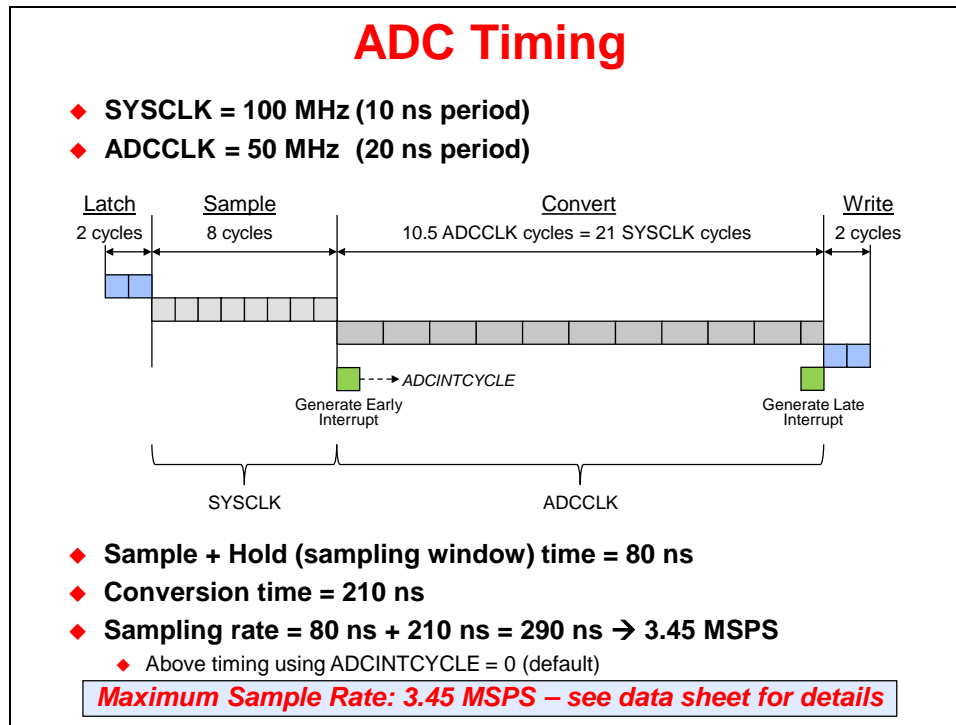
- ◆ Each ADC module contains four Post Processing Blocks
- ◆ Each Post Processing Block can be associated with any of the 16 ADCRESULTx registers



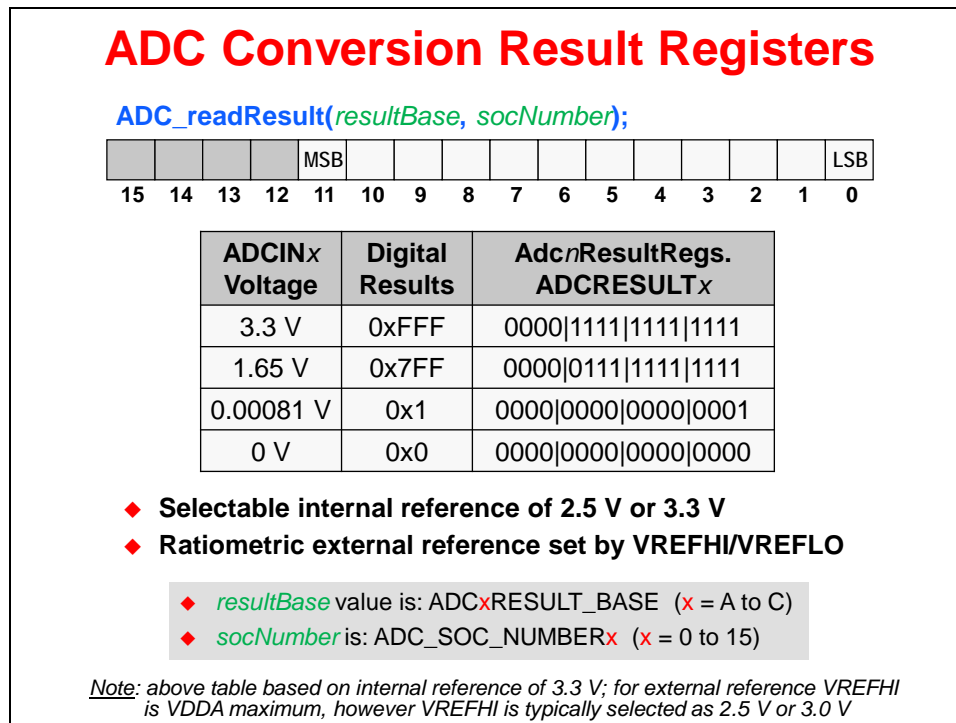
ADC Clocking Flow



ADC Timing



ADC Conversion Result Registers

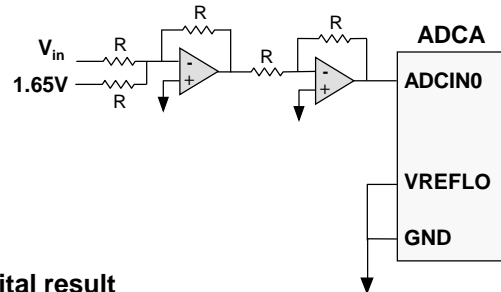


Signed Input Voltages

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 "Lab.h"
#define offset 0x07FF
void main(void)
{
    int16_t value;           // signed

    value = ADC_readResult(resultBase, socNumber) - offset;
}
```

Built-In ADC Calibration

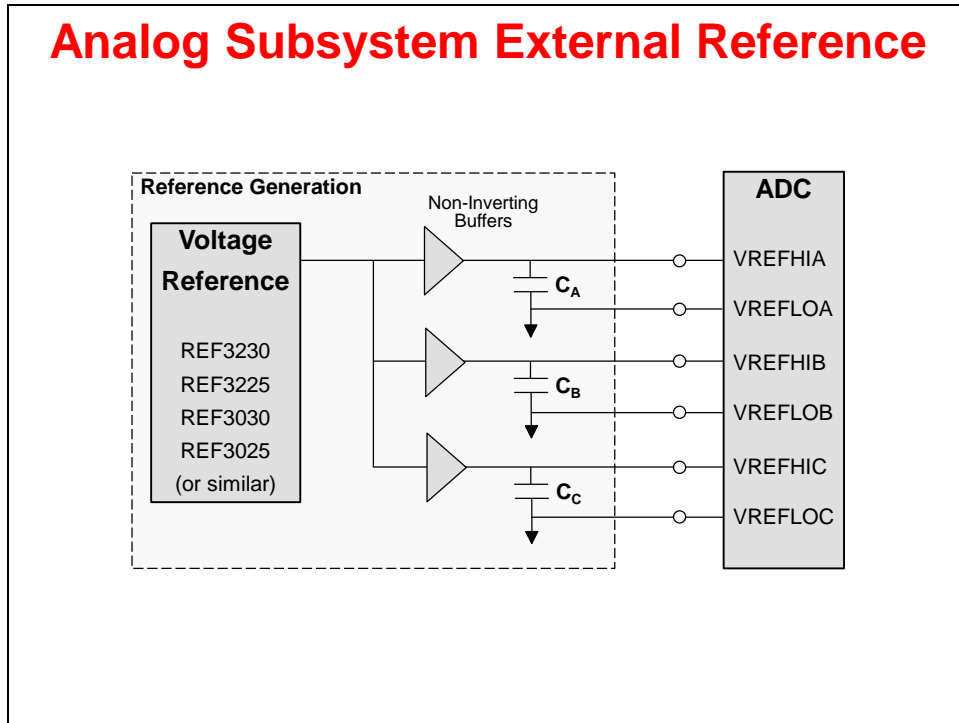
Built-In ADC Calibration

- ◆ TI reserved OTP contains device specific calibration data for the ADC, internal oscillators and buffered DAC
- ◆ 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 specifications 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))0x00070280
void main(void)
{
    (*Device_cal)();           // call Device_cal()
}
```

Note: Device_cal address is located in the bootrom code, which can be found at -
 C:\ti\c2000\C2000Ware_<version>\libraries\boot_rom\28004x\revB\rom_sources\F28004x_ROM\bootROM\include\bootrom.h

Analog Subsystem External Reference



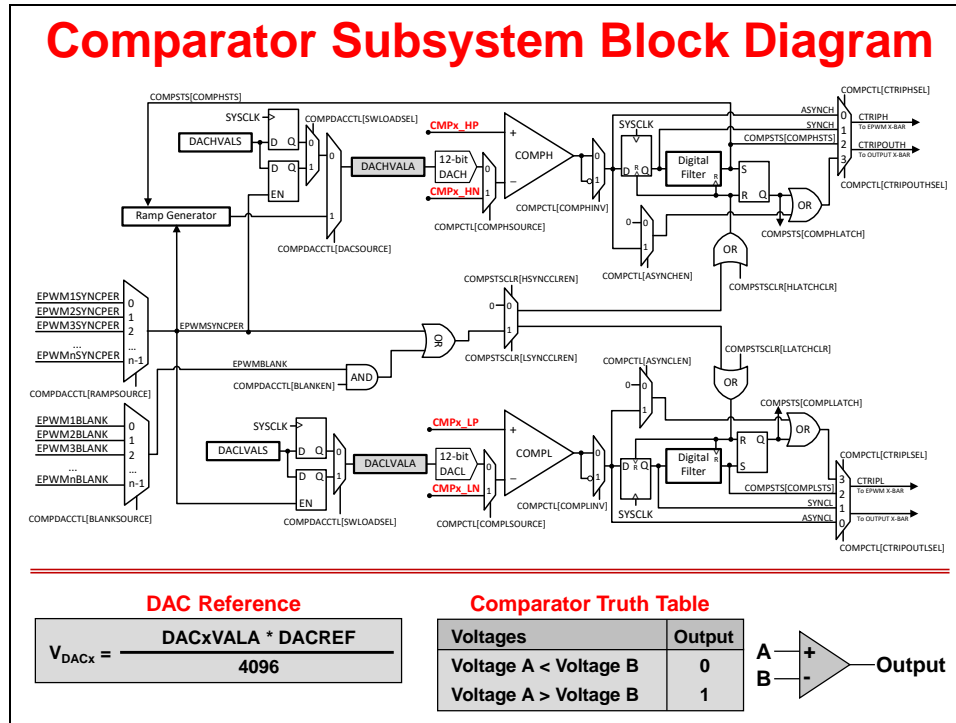
Comparator Subsystem (CMPSS)

Comparator Subsystem

- ◆ **Each CMPSS consists of:**
 - ◆ Two analog comparators
 - ◆ Two programmable reference 12-bit DACs
 - ◆ Two digital filters and one ramp generator
- ◆ **Each comparator generates a digital output**
 - ◆ Indicates if voltage on positive input is greater than the voltage on the negative input
 - ◆ Positive input can be driven from an external pin or PGA
 - ◆ Negative input can be driven by an external pin or 12-bit DAC
- ◆ **Each comparator output can be digitally filtered to remove spurious trip signals (majority vote)**
- ◆ **Ramp generator used for peak current mode control**
- ◆ **Ability to synchronize with EPWMSYNCO event, SYSCCLK, and a clear signal with EPWMBLANK**
- ◆ **DAC reference voltage can be either VDDA or VDAC**

The F28004x includes independent Comparator Subsystem (CMPSS) modules that are useful for supporting applications such as peak current mode control, switched-mode power, power factor correction, and voltage trip monitoring. The Comparator Subsystem modules have the ability to synchronize with a PWMSYNC event.

Comparator Subsystem Block Diagram



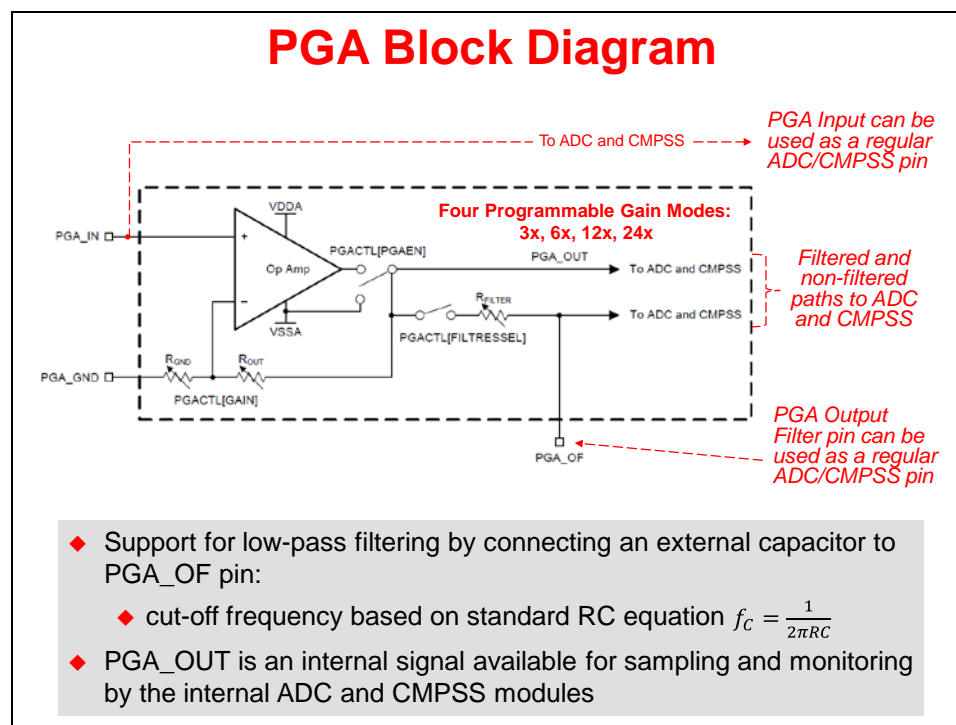
Each CMPSS module is designed around a pair of analog comparators which generates a digital output indicating if the voltage on the positive input is greater than the voltage on the negative input. The comparator positive and negative input signals are independently selectable by using the analog subsystem interconnect scheme. The positive input to the comparator is always driven from an external pin. The negative input can be driven by either an external pin or an internal programmable 12-bit digital-to-analog (DAC) as a reference voltage. Values written to the DAC can take effect immediately or be synchronized with ePWM events. A falling-ramp generator is optionally available to the control the internal DAC reference value for one comparator in the module. Each comparator output is fed through a programmable digital filter that can remove spurious trip signals. Also included is PWM blanking capability to clear-and-reset existing or imminent trip conditions near the EPWM cycle boundaries. The output of the CMPSS generates trip signals to the ePWM event trigger submodule and GPIO structure.

Programmable Gain Amplifier (PGA)

Programmable Gain Amplifier (PGA)

- ◆ Amplifies small input signals to increase the dynamic range of the downstream ADC and CMPSS modules
- ◆ Reduces cost and design effort over external standalone amplifiers
 - ◆ On-chip integration ensures compatible with ADC and CMPSS
 - ◆ Internally powered by VDDA and VSSA
- ◆ Adaptable to various performance needs
 - ◆ Software selectable gain and filter settings
 - ◆ Four programmable gain modes: 3x, 6x, 12x, 24x
 - ◆ Embedded series resistors for RC filtering
- ◆ Hardware based analog offset and gain trimming reduces offset and gain error
 - ◆ Instead of software post-processing

PGA Block Diagram



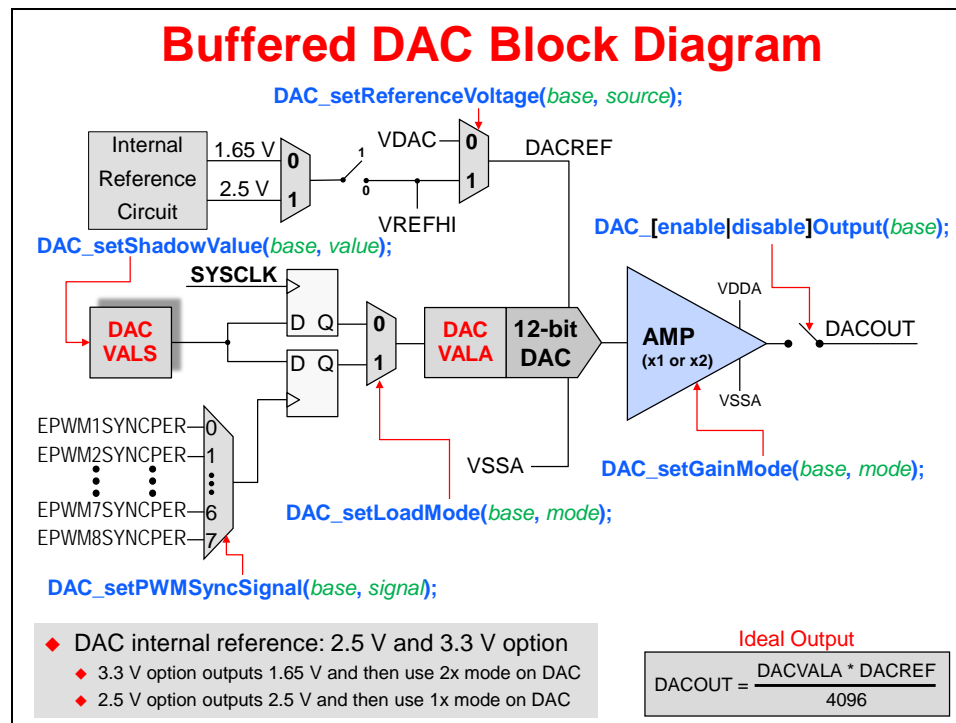
Digital-to-Analog Converter (DAC)

Digital-to-Analog Converter

- ◆ 12-bit DAC provides a programmable reference output voltage
- ◆ Analog output buffer is capable of driving an external load
- ◆ Selectable reference voltage
- ◆ Can be used as a general-purpose DAC for generating a DC voltage and AC waveforms (e.g. sine, square, triangle, etc.)
- ◆ Ability to be synchronized with EPWMSYNCPER events

The buffered 12-bit DAC module can be used to provide a programmable reference output voltage and it includes an analog output buffer that is capable of driving an external load. Values written to the DAC can take effect immediately or be synchronized with ePWM events.

Buffered DAC Block Diagram



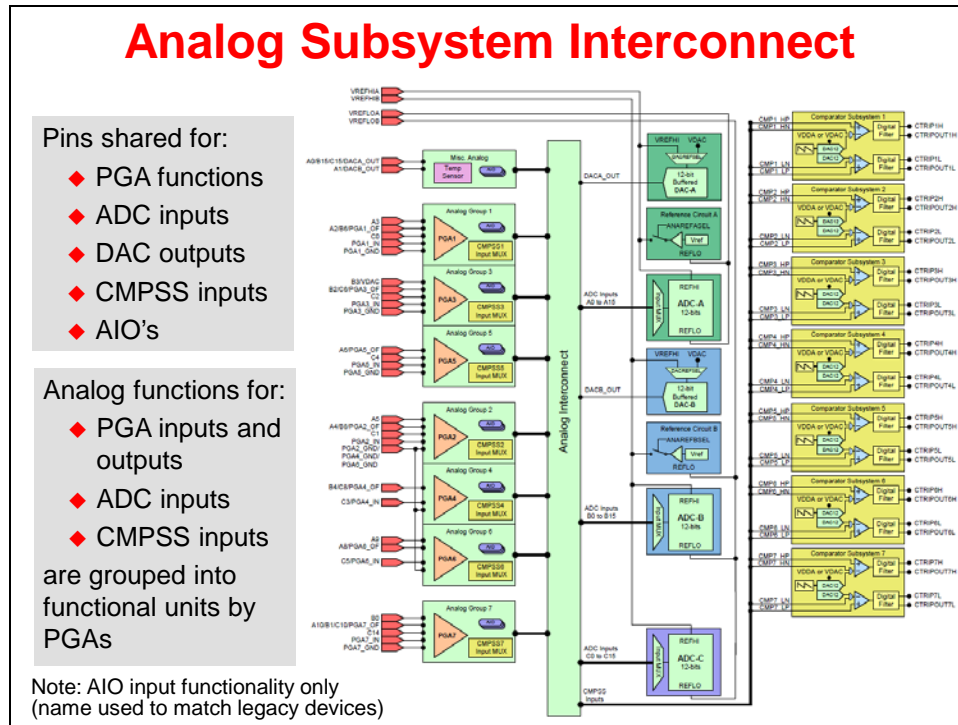
Two sets of DACVAL registers are present in the buffered DAC module: DACVALA and DACVALS. DACVALA is a read-only register that actively controls the DAC value. DACVALS is a writable shadow register that loads into DACVALA either immediately or synchronized with the next PWMSYNC event. The ideal output of the internal DAC can be calculated as shown in the equation above.

DAC Driverlib Functions

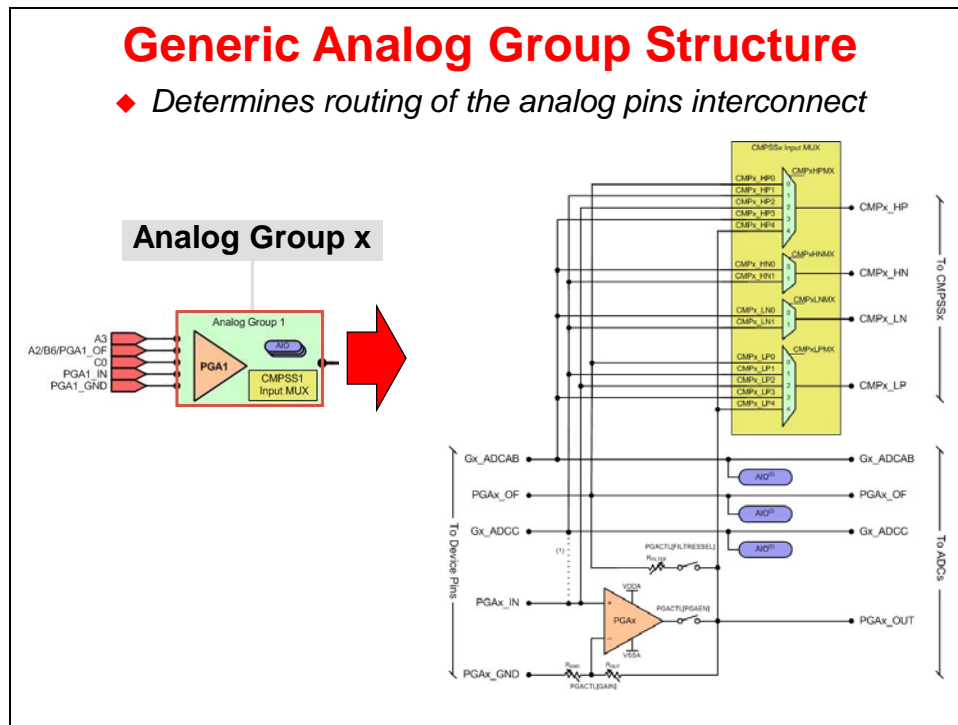
- ◆ Set the DAC reference voltage
`DAC_setReferenceVoltage(base, source);`
- ◆ Set the DAC gain mode
`DAC_setGainMode(base, mode);`
- ◆ Set the DAC load mode
`DAC_setLoadMode(base, mode);`
- ◆ Set DAC shadow value
`DAC_setShadowValue(base, value);`
- ◆ Enable / disable DAC output
`DAC_[enable|disable]Output(base);`
- ◆ Set DAC PWMSYNC signal
`DAC_setPWMSyncSignal(base, signal);`

- ◆ *base* is the ADC base address: DAC_x_BASE (x = A or B)
- ◆ *source* value is: DAC_REF_VDAC or DAC_REF_ADC_VREFHI
- ◆ *mode* (gain) value is: DAC_GAIN_ONE or DAC_GAIN_TWO
- ◆ *mode* (load) value is: DAC_LOAD_SYSCLK or DAC_LOAD_PWMSYNC
- ◆ *value* is the 12-bit code to be loaded into the active value register
- ◆ *signal* is the selected PWM signal (e.g. 2 selects PWM sync signal 2)

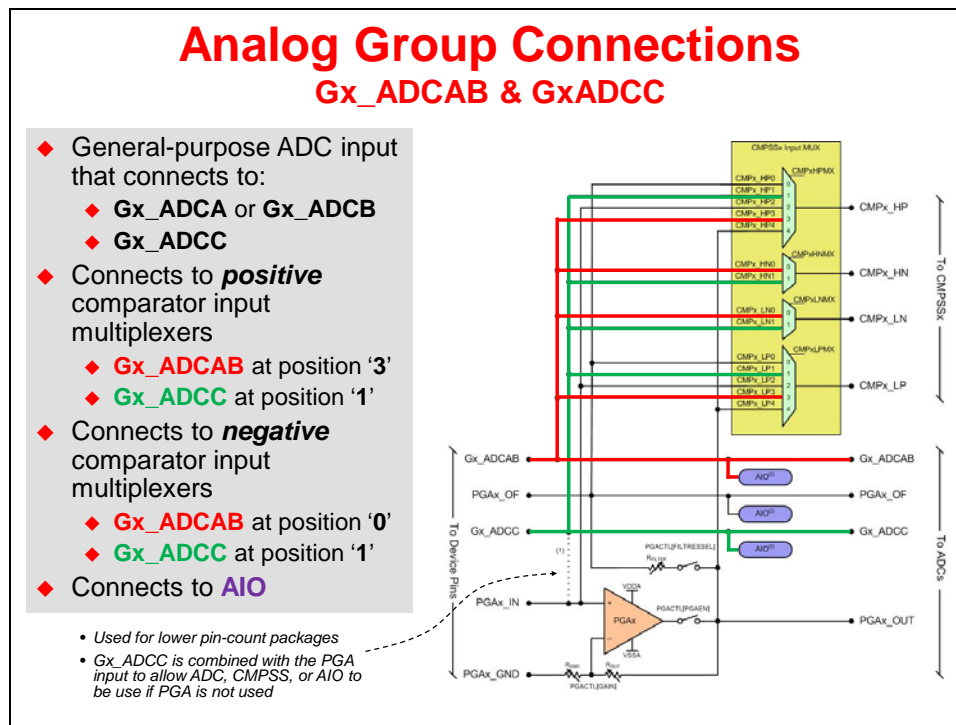
Analog Subsystem Interconnect



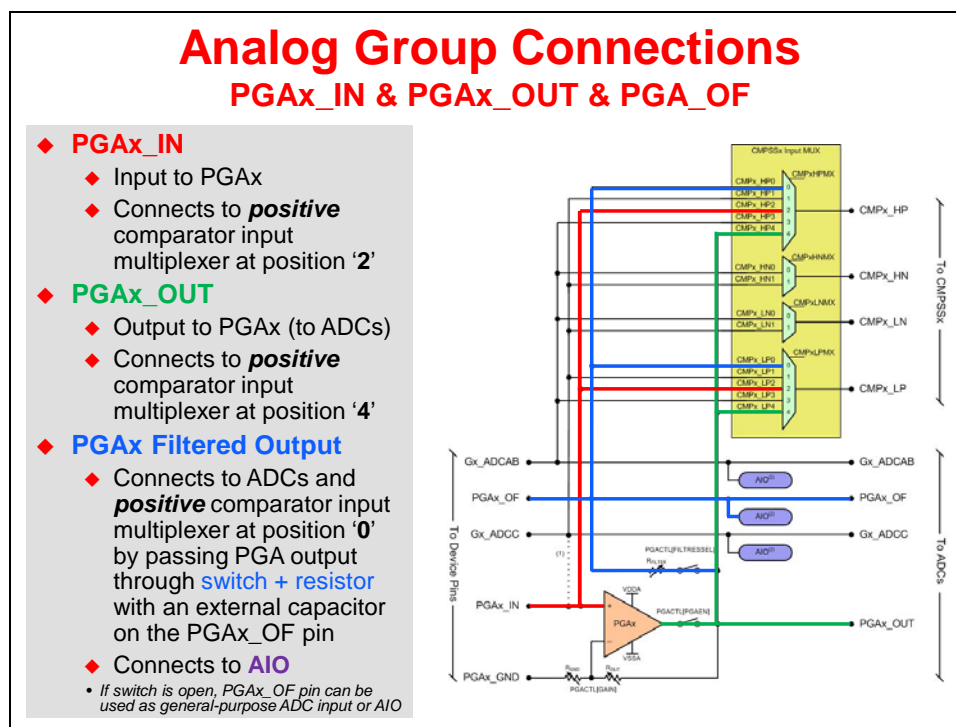
The Analog Subsystem Interconnect enables a very flexible pin usage, allowing for smaller device packages. The DAC outputs, comparator subsystem inputs, PGA functions, and digital inputs are multiplexed with the ADC inputs. This type of interconnect permits a single pin to route a signal to multiple analog modules. The figure below is the generic analog group structure



Analog Group Connections



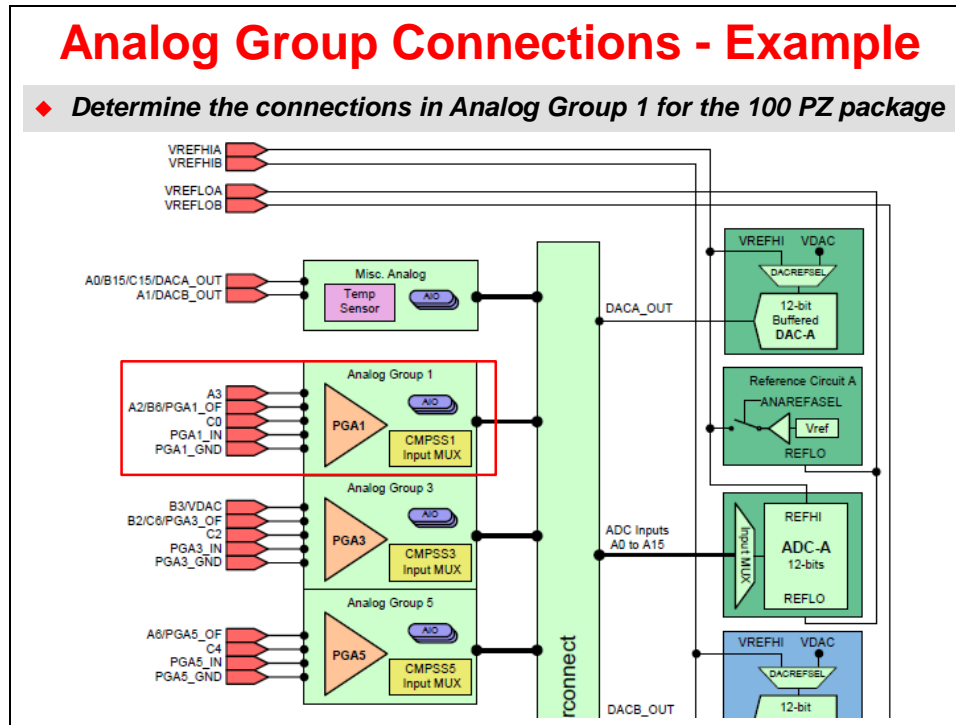
The general-purpose ADC input pins, shown by the red and green lines, connects to the ADCs and the input multiplexers which feed the positive and negative comparator subsystem inputs. Also, the ADC input pins connect as inputs to the AIOs.



The PGA input pin, shown by the red line, connects to the PGA and the input multiplexers which feed the positive comparator subsystem inputs. The PGA output, shown by the green line, connects to the ADCs and the input multiplexers which feed the positive comparator subsystem inputs. The PGA filtered output, shown by the blue line, connects to the ADCs and the input multiplexers which feed the positive comparator subsystem inputs, in addition to being an input to the AIO.

Analog Group Connection – Example

In this example, we will determine the connections in Analog Group 1 for the 100 PZ package.



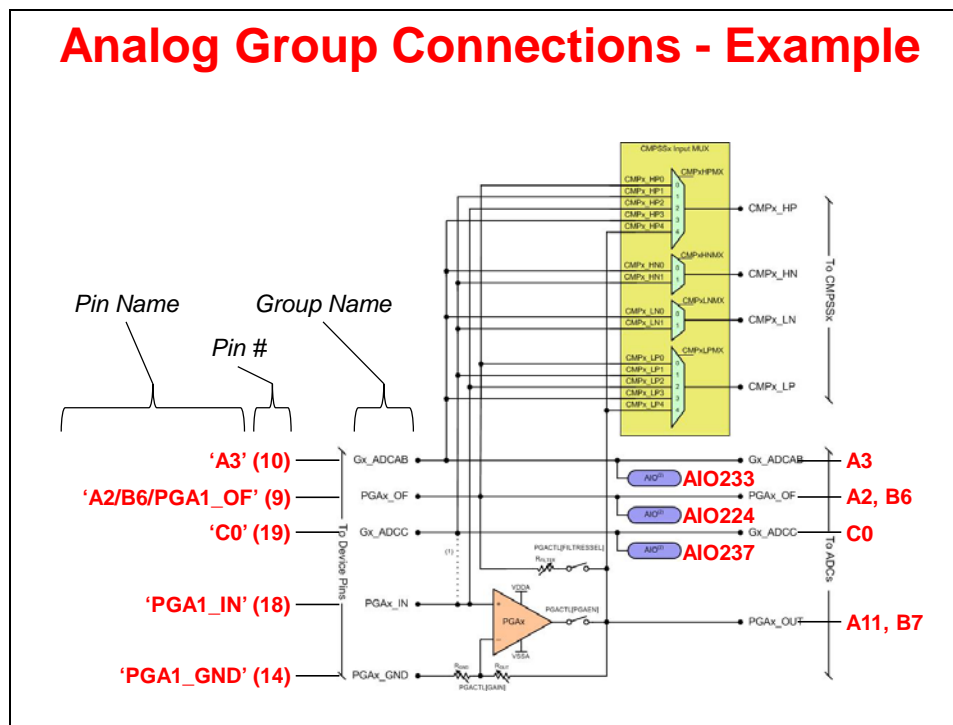
Using the Analog Pins and Internal Connection table, notice that group name G1_ADCAB is connected to pin 10 and has a pin name as A3. This signal is always connected to ADCA and it is multiplexed with the comparator subsystem inputs. It is also connected to AIO233. The remaining pin numbers, pin names, and connections are determined the same way.

Analog Pins and Internal Connections

PIN NAME	GROUP NAME	PACKAGE			ALWAYS CONNECTED (NO MUX)				COMPARATOR SUBSYSTEM (MUX)				AIO INPUT	
		100 PZ	64 PM	56 RSH	ADCA	ADCB	ADCC	PGA	DAC	HIGH POSITIVE	HIGH NEGATIVE	LOW POSITIVE		LOW NEGATIVE
VREFHA	-	25												
VREFHB	-	24	16	14										
VREFHC	-	27					A13							
VREFLOA	-	26	17	15			B13							
VREFLOC	-							C13						
Analog Group 1														
A3	G1_ADCAB	10			A3					CMP1_HP3	CMP1_HN0	CMP1_LP3	CMP1_LN0	AIO233
A2/B6/PGA1_OF	PGA1_OF	9	9	8	A2	B6		PGA1_OF		CMP1_HP0	CMP1_HN0	CMP1_LP0	CMP1_LN0	AIO224
C0	G1_ADCC	19	12	10			C0			CMP1_HP1	CMP1_HN1	CMP1_LP1	CMP1_LN1	AIO237
PGA1_IN	PGA1_IN	18						PGA1_IN						
PGA1_GND	PGA1_GND	14	10	9				PGA1_GND						
PGA1_OUT ⁽¹⁾	PGA1_OUT ⁽¹⁾				A11	B7		PGA1_OUT		CMP1_HP4		CMP1_LP4		
Analog Group 2														
B5	G2_ADCAB	35			A5					CMP2_HP3	CMP2_HN0	CMP2_LP3	CMP2_LN0	AIO234
B4/C8/PGA2_OF	PGA2_OF	36	23	21	A4	B8		PGA2_OF		CMP2_HP0		CMP2_LP0		AIO225

For this example, the complete Analog Group 1 connections are shown. Again, notice that pin name 'A3' is connected to pin 10 as an input to ADCAIN3, and it is multiplexed with the comparator subsystem inputs. Also, it is connected as an input to AIO233. The other remaining connections can be mapped back to the Analog Pins and Internal Connection table.

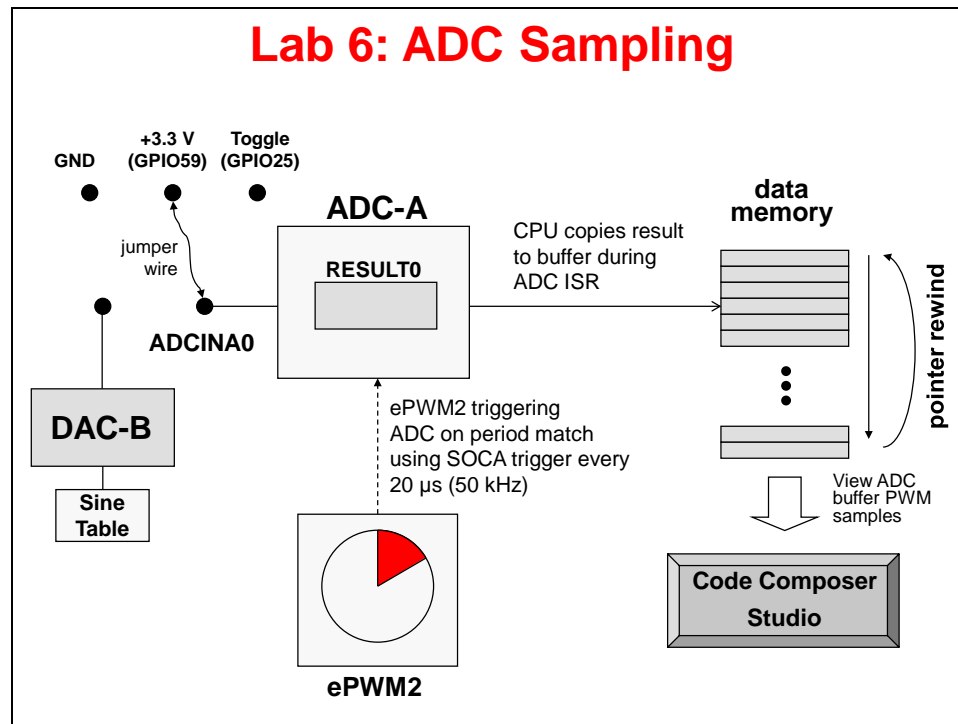
Analog Group Connections - Example



Lab 6: Analog-to-Digital Converter

➤ Objective

The objective of this lab exercise is to become familiar with the programming and operation of the on-chip analog-to-digital converter (ADC). The microcontroller (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. In the second part of this lab exercise, the digital-to-analog converter (DAC) will be explored.

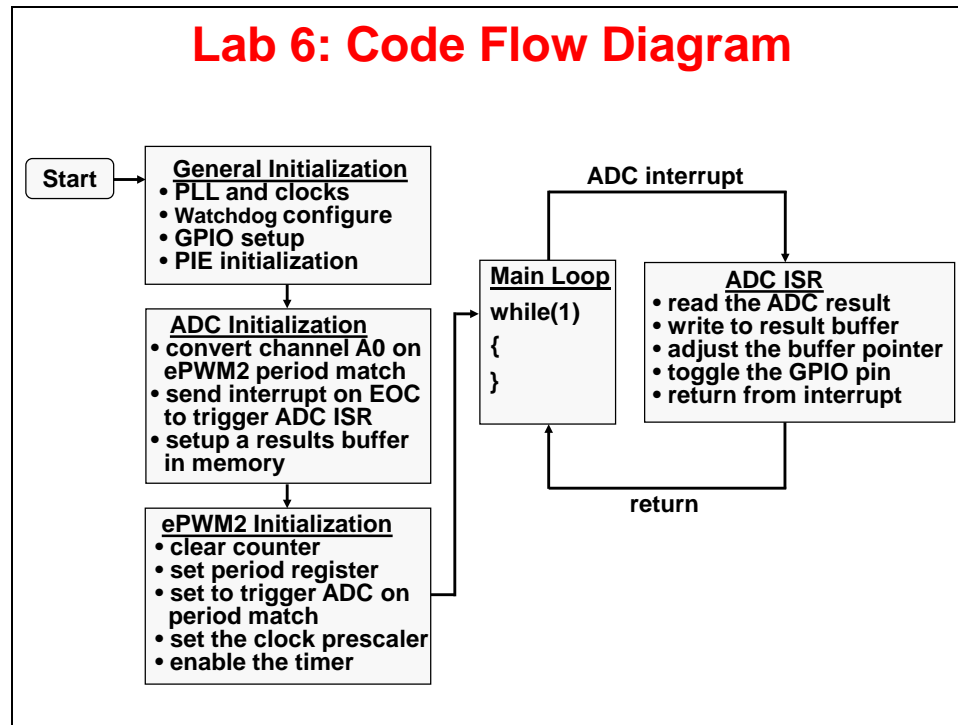


Recall that there are three basic ways to initiate an ADC start of conversion (SOC):

1. Using software
 - a. SOC_x (where x = 0 to 15) causes a software initiated conversion [ADC_TRIGGER_SW_ONLY]
2. Automatically triggered on user selectable conditions
 - a. CPU Timer 0/1/2 interrupt [ADC_TRIGGER_CPU1_TINT_x]
 - b. ePWM_xSOCA / ePWM_xSOCB (x = 1 to 8) [ADC_TRIGGER_EPWM_x_SOCA/B]
 - 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/D)
 - c. ADC interrupt ADCINT1 or ADCINT2
 - triggers SOC_x selected by the ADC Interrupt Trigger SOC [ADC_INT_SOC_TRIGGER_NONE or ADC_INT_SOC_TRIGGER_ADCINT_x]
3. Externally triggered using a pin
 - a. SOC_x trigger by ADCEXTSOC via INPUT5 X-BAR GPIO pin [ADC_TRIGGER_GPIO]

One or more of these methods may be applicable to a particular application. In this lab exercise, 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 (GPIO25) high and low in the ADC interrupt service routine. This pin will be connected to the ADC input pin, and sampled. The ADC ISR will also toggle LED5 on the LaunchPad as a visual indication that the ISR is running. 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 ADCA channel 0 (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
- GPIO25 pin is also toggled in the ADC ISR
- ADC ISR will also toggle the LaunchPad LED5 as a visual indication that it is running

➤ Procedure

Open the Project

1. A project named `Lab6` has been created for this lab exercise. Open the project by clicking on `Project` → `Import CCS Projects`. The “Import CCS Eclipse Projects” window will open. Click `Browse...` next to the “Select search-directory” box. Navigate to: `C:\F28004x\Labs\Lab6\project` and click `Select Folder`. Then click `Finish` to import the project. All build options have been configured the same as the previous lab exercise. The files used in this lab exercise are:

<code>Adc_6.c</code>	<code>Gpio.c</code>
<code>CodeStartBranch.asm</code>	<code>Lab_5_6_7.cmd</code>
<code>Dac.c</code>	<code>Main_6.c</code>
<code>DefaultIsr_6.c</code>	<code>SineTable.c</code>
<code>device.c</code>	<code>Watchdog.c</code>
<code>EPwm_6.c</code>	

Note: The `Dac.c` and `SineTable.c` files are used to generate a sine waveform in the second part of this lab exercise.

Setup ADC Initialization and Enable Core/PIE Interrupts

2. In `Main_6.c` add code to call the `InitAdca()`, `InitDacb()`, 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. The `InitDacb()` function will be used in the second part of this lab exercise.
3. Edit `Adc_6.c` to configure SOC0 in the ADC as follows:
 - SOC0 converts input ADCINA0
 - SOC0 has a 8 SYSCLK cycle acquisition window
 - SOC0 is triggered by the ePWM2 SOCA
 - SOC0 triggers ADCINT1 on end-of-conversion
 - All SOCs run round-robin
4. Using the “PIE Interrupt Assignment Table” find the location for the ADC interrupt “INT_ADCA1” 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_6.c` to do the following:
 - Add the Driverlib function to re-map the ADC interrupt signal to call the ISR function. (Hint: #define name in `driverlib/inc/hw_ints.h` and label name in `DefaultIsr_6.c`)
 - Add the Driverlib function to enable the appropriate PIEIER and core IER
6. Save all changes to the files.
7. Inspect `DefaultIsr_6.c`. This file contains the ADC interrupt service routine.

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

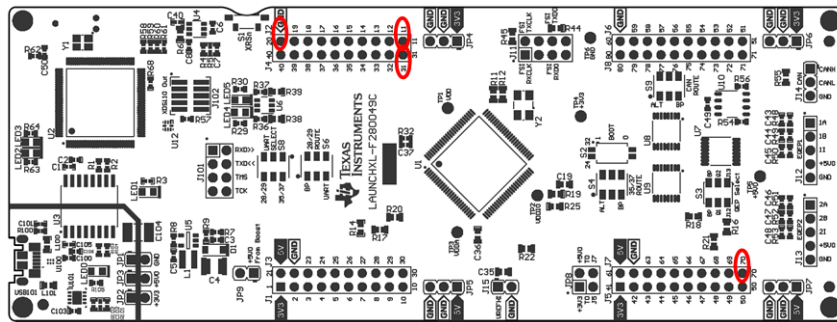
Run the Code

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

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. Then `<enter>` to view the contents of the ADC result buffer.

Note: *Exercise care when connecting any jumper wires to the LaunchPad header pins since the power to the USB connector is on!*


Refer to the following diagram for the location of the pins that will need to be connected:



11. Using a jumper wire, connect the ADCINA0 (pin #70) to “GND” (pin #20) on the LaunchPad. Then run the code again, and halt it after a few seconds. Verify that the ADC results buffer contains the expected value of `~0x0000`.
12. Adjust the jumper wire to connect the ADCINA0 (pin #70) to “+3.3V” (pin #11; GPIO-59) on the LaunchPad. (Note: pin # GPIO-59 has been set to “1” in `Gpio.c`). Then run the code again, and halt it after a few seconds. Verify that the ADC results buffer contains the expected value of `~0x0FFF`.
13. Adjust the jumper wire to connect the ADCINA0 (pin #70) to GPIO25 (pin #31) on the LaunchPad. Then run the code again, and halt it after a few seconds. Examine the contents of the ADC results buffer (the contents should be alternating `~0x0000` and `~0x0FFF` values). Are the contents what you expected?
14. Open and setup a graph to plot a 50-point window of the ADC results buffer. Click: `Tools` → `Graph` → `Single Time` and set the following values:

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

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

15. Recall that the code toggled the GPIO25 pin alternately high and low. (Also, the ADC ISR is toggling the LED5 on the LaunchPad as a visual indication that the ISR is running). If you had an oscilloscope available to display GPIO25, 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 GPIO25 pin at a 50 kHz rate. Therefore, a complete cycle (toggle high, then toggle low) occurs at half this rate, or 25 kHz. We therefore expect the period of the waveform to be 40 μ s. Confirm this by measuring the period of the 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 real-time 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 jumper wire between ADCINA0 (pin #70) and GPIO25 (pin #31) 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 sign (+’ or >’) 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 `Cancel` to close the window.

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. To enable and run real-time emulation mode, click the “Enable Silicon Real-time Mode” toolbar button . A window may open and if prompted select `Yes` to the “Do you want to enable realtime mode?” question. This will force the debug enable mask bit (DBGM) in status register ST1 to ‘0’, which will allow the memory and register values to be passed to the host processor for updating (i.e. debug events are enabled). Hereafter, `Resume` and `Suspend` are used to run and halt real-time debugging. In the remaining lab exercises we will run and halt the code in real-time emulation mode.
21. Run the code (real-time mode).
22. **Carefully** remove and replace the jumper wire from GPIO25 (pin #31). Are the values updating in the Memory Browser and Single Time graph as expected?
23. Halt the code.
24. So far, we have seen data flowing from the MCU to the debugger in real-time. 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 GPIO25 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, halt the CPU.

Setup DAC to Generate a Sine Waveform

Next, we will configure DACB to generate a fixed frequency sine wave. This signal will appear on an analog output pin of the device (ADCINA1). Then using the jumper wire we will connect the DACB output to the ADCA input (ADCINA0) and display the sine wave in a graph window.

25. Notice the following code lines in the ADCA1 ISR in `DefaultIsr_6.c`:

```
//--- Write to DAC-B to create input to ADC-A0
if(SINE_ENABLE == 1)
{
    DacOutput = DacOffset + ((QuadratureTable[iQuadratureTable++] ^ 0x8000) >> 5);
}
else
{
    DacOutput = DacOffset;
}
if(iQuadratureTable > (SINE_PTS - 1)) // Wrap the index
{
    iQuadratureTable = 0;
}
DAC_setShadowValue(DACB_BASE, DacOutput);
```

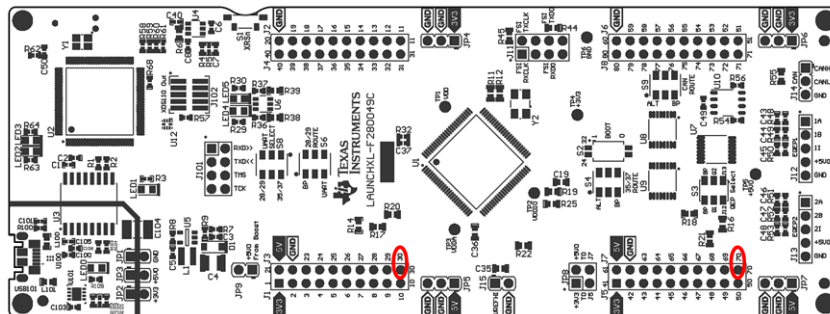
The variable `DacOffset` allows the user to adjust the DC output of DACB from the Expressions window in CCS. The variable `Sine_Enable` is a switch which adds a fixed frequency sine wave to the DAC offset. The sine wave is generated using a 25-point look-up table contained in the `SineTable.c` file. We will plot the sine wave in a graph window while manually adjusting the offset.

26. Open and inspect `SineTable.c`. (If needed, open the Project Explorer window in the CCS Debug perspective view by clicking `View` → `Project Explorer`). The file consists of an array of 25 signed integer points which represent four quadrants of sinusoidal data. The 25 points are a complete cycle. In the source code we need to sequentially access each of the 25 points in the array, converting each one from signed 16-bit to un-signed 12-bit format before writing it to the DACVALS register of DACB.

27. Add the following variables to the Expressions window:

- `SINE_ENABLE`
- `DacOffset`

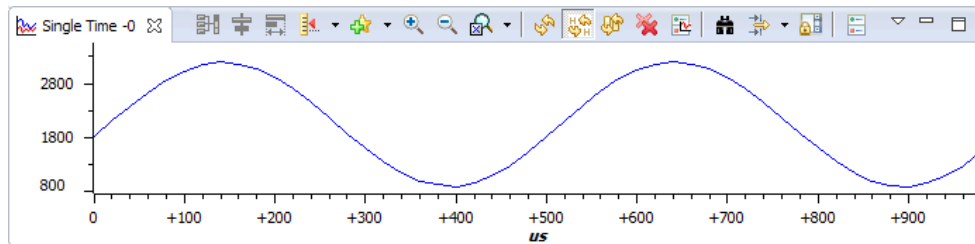
28. Adjust the jumper wire to connect the ADCINA0 (pin #70) to DACB (pin #30) on the LaunchPad. Refer to the following diagram for the pins that need to be connected.



29. Run the code (real-time mode).

30. At this point, the graph should be displaying a DC signal near zero. Click on the `DacOffset` variable in the Expressions window and change the value to 800. This changes the DC output of the DAC which is applied to the ADC input. The level of the graph display should be about 800 and this should be reflected in the value shown in the memory buffer (note: 800 decimal = 0x320 hex).

31. Enable the sine generator by changing the variable `SINE_ENABLE` in the Expressions window to 1.
32. You should now see sinusoidal data in the graph window.



33. Try removing and re-connecting the jumper wire to show this is real data is running in real-time emulation mode. Also, you can try changing the DC offset variable to move the input waveform to a different average value (the maximum distortion free offset is about 2000).
34. Halt the code.

Terminate Debug Session and Close Project

35. Terminate the active debug session using the `Terminate` button. This will close the debugger and return Code Composer Studio to the CCS Edit perspective view.
36. Next, close the project by right-clicking on `Lab6` in the Project Explorer window and select `Close Project`.

End of Exercise

Control Peripherals

Introduction

The C2000 high-performance control peripherals are an integral component for all digital control systems. This module starts with a review of pulse width modulation (PWM) and then explains how the ePWM is configured for generating PWM waveforms. Also, the use of the eCAP and the eQEP will be discussed. Additionally, an overview of the Sigma Delta Filter Module 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)**
- ◆ **Describe the purpose of the Sigma Delta Filter Module (SDFM)**

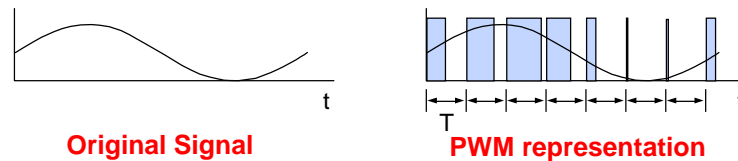
Chapter Topics

Control Peripherals	7-1
<i>PWM Review</i>	7-3
<i>ePWM</i>	7-5
ePWM Time-Base Sub-Module	7-7
ePWM Compare Sub-Module	7-10
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	7-23
ePWM Trip-Zone and Digital Compare Sub-Modules	7-25
ePWM Event-Trigger Sub-Module	7-31
High Resolution PWM (HRPWM).....	7-33
<i>eCAP</i>	7-35
<i>eQEP</i>	7-39
<i>Sigma Delta Filter Module (SDFM)</i>	7-42
<i>Lab 7: Control Peripherals</i>	7-44

PWM Review

What is Pulse Width Modulation?

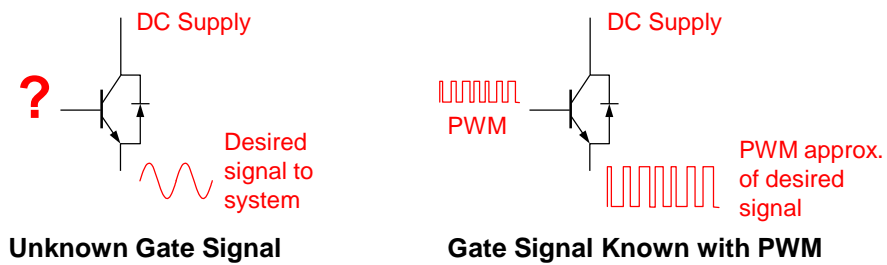
- ◆ PWM is a scheme to represent a signal as a sequence of pulses
 - ◆ fixed carrier frequency
 - ◆ fixed pulse amplitude
 - ◆ pulse width proportional to instantaneous signal amplitude
 - ◆ PWM energy \approx original signal energy



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.

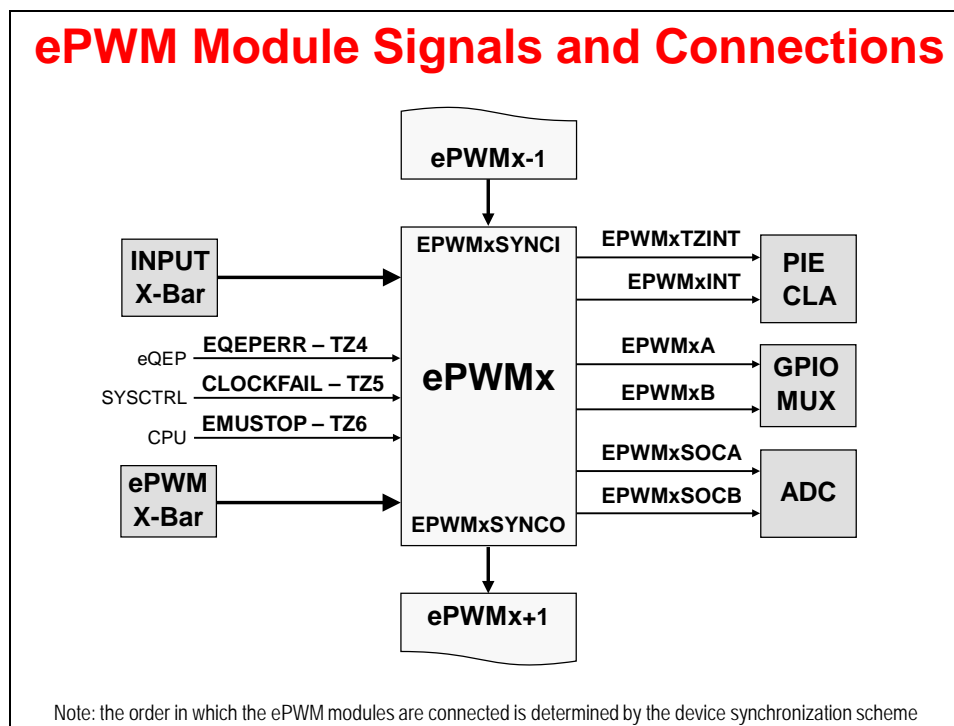
Why use PWM with Power Switching Devices?

- ◆ 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

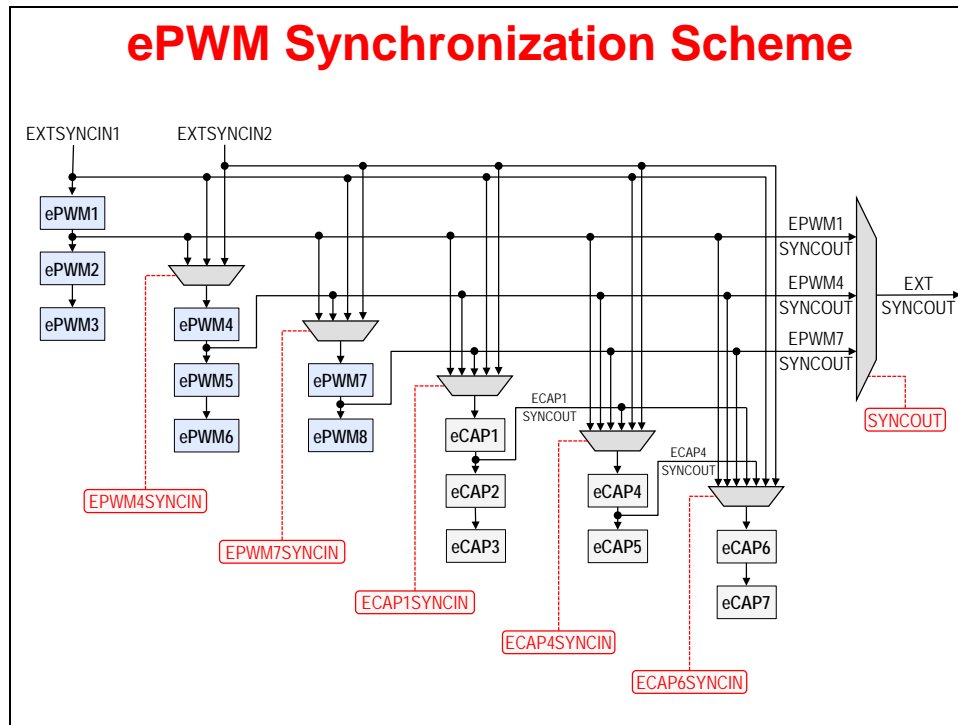


Power switching devices can be difficult to control when operating in the proportional region, but are easy to control in the saturation and cutoff regions. Since PWM is a digital signal by nature and easy for an MCU to generate, it is ideal for use with power switching devices. Essentially, PWM performs a DAC function, where the duty cycle is equivalent to the DAC analog amplitude value.

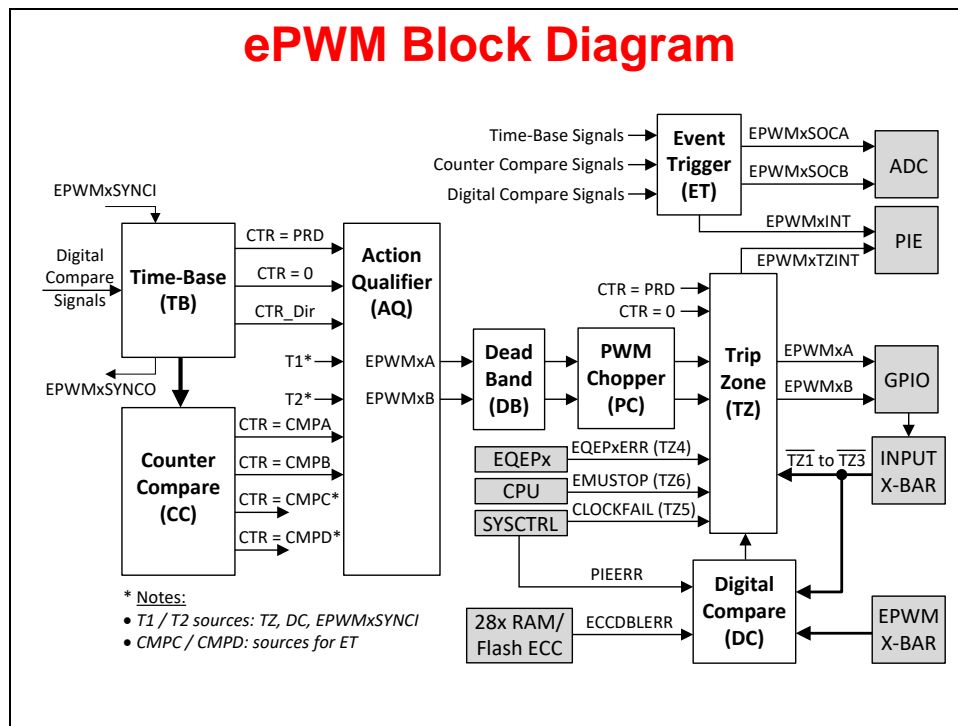
ePWM



The ePWM modules are highly programmable, extremely flexible, and easy to use, while being capable of generating complex pulse width waveforms with minimal CPU overhead or intervention. Each ePWM module is identical with two PWM outputs, EPWMxA and EPWMxB, and multiple modules can be synchronized to operate together as required by the system application design. The generated PWM waveforms are available as outputs on the GPIO pins. Additionally, the EPWM module can generate ADC start of conversion signals and generate interrupts to the PIE block. External trip zone signals can trip the output, as well as generate interrupts. The outputs of the comparators are used as inputs to the ePWM X-Bar. Next, the internal details of the ePWM module will be covered.

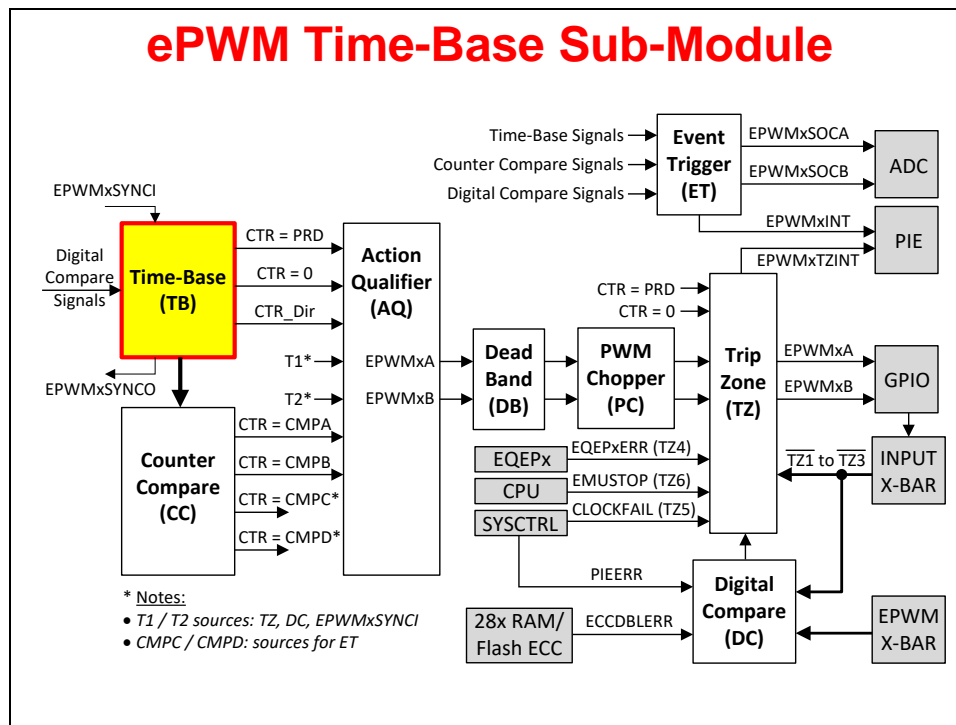


Various ePWM modules (and eCAP units) can be grouped together for synchronization.

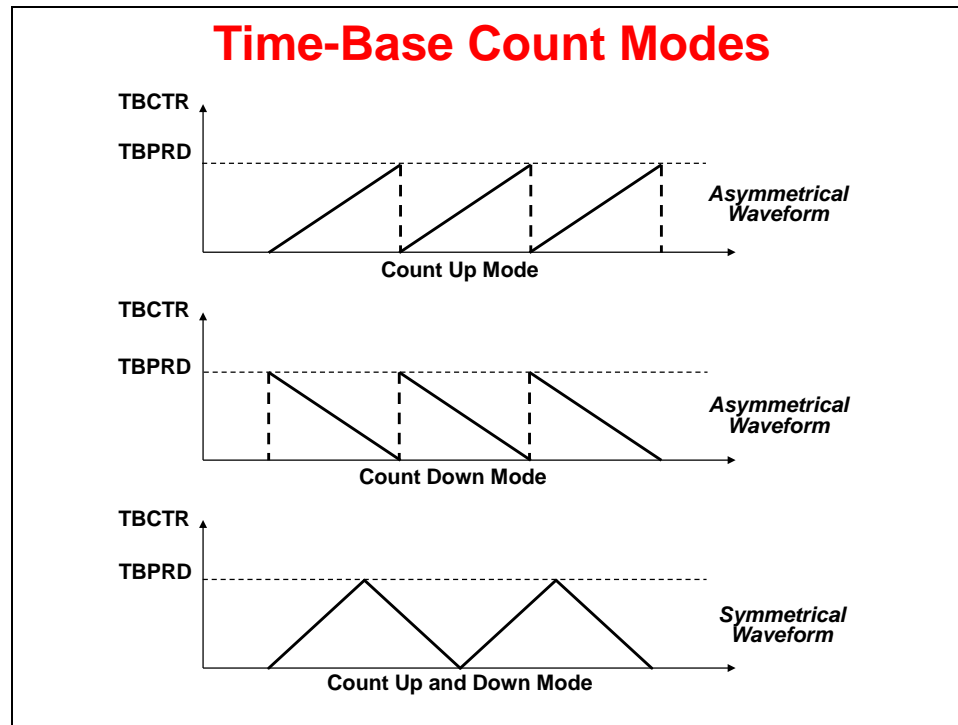


The ePWM module consists of eight submodules: time-base, counter-compare, action-qualifier, dead-band generator, PWM chopper, trip-zone, digital-compare, and event-trigger.

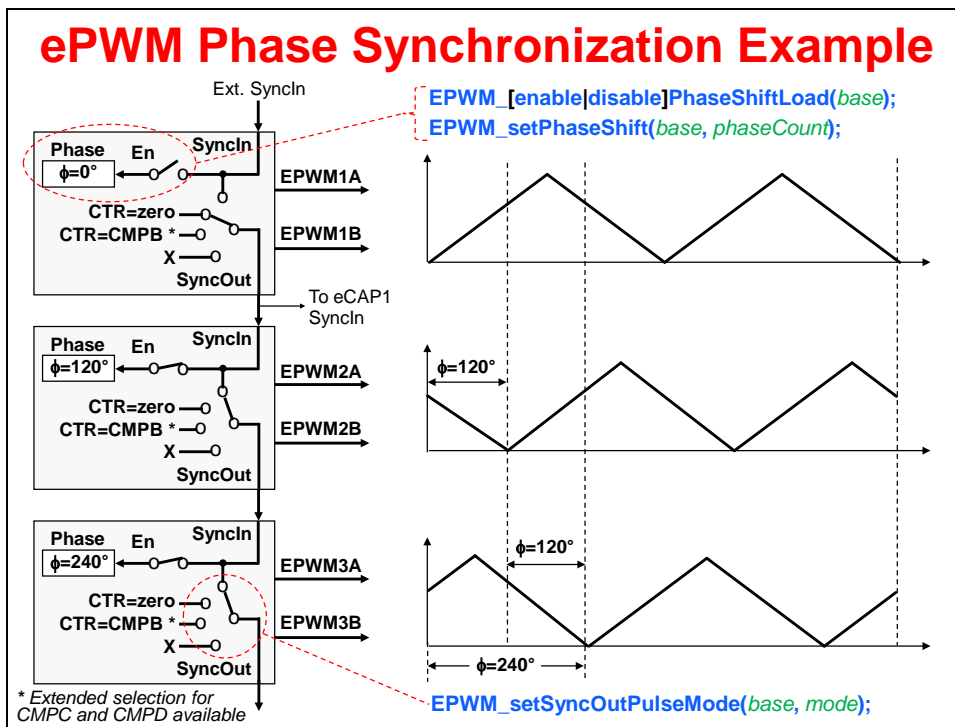
ePWM Time-Base Sub-Module



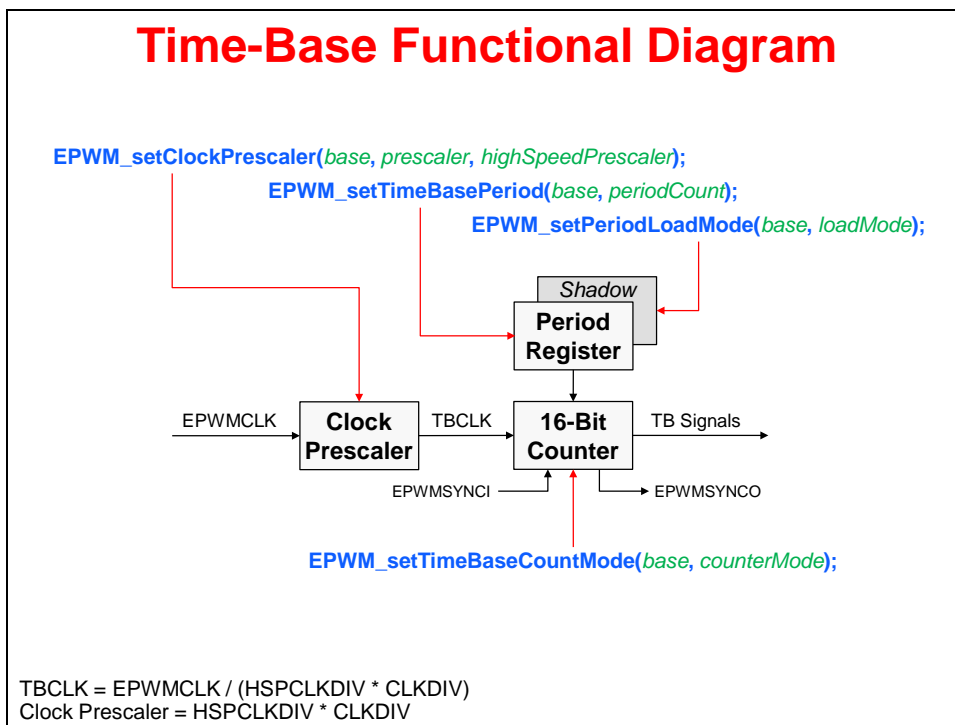
The time-base submodule consists of a dedicated 16-bit counter, along with built-in synchronization logic to allow multiple ePWM modules to work together as a single system. A clock pre-scaler divides the EPWM clock to the counter and a period register is used to control the frequency and period of the generated waveform. The period register has a shadow register, which acts like a buffer to allow the register updates to be synchronized with the counter, thus avoiding corruption or spurious operation from the register being modified asynchronously by the software.



The time-base counter operates in three modes: up-count, down-count, and up-down-count. In up-count mode the time-base counter starts counting from zero and increments until it reaches the period register value, then the time-base counter resets to zero and the count sequence starts again. Likewise, in down-count mode the time-base counter starts counting from the period register value and decrements until it reaches zero, then the time-base counter is loaded with the period value and the count sequence starts again. In up-down-count mode the time-base counter starts counting from zero and increments until it reaches the period register value, then the time-base counter decrements until it reaches zero and the count sequence repeats. The up-count and down-count modes are used to generate asymmetrical waveforms, and the up-down-count mode is used to generate symmetrical waveforms.



Synchronization allows multiple ePWM modules to work together as a single system. The synchronization is based on a synch-in signal, time-base counter equals zero, or time-base counter equals compare B register, which can also be extended to compare C and compare D. Additionally, the waveform can be phase-shifted.

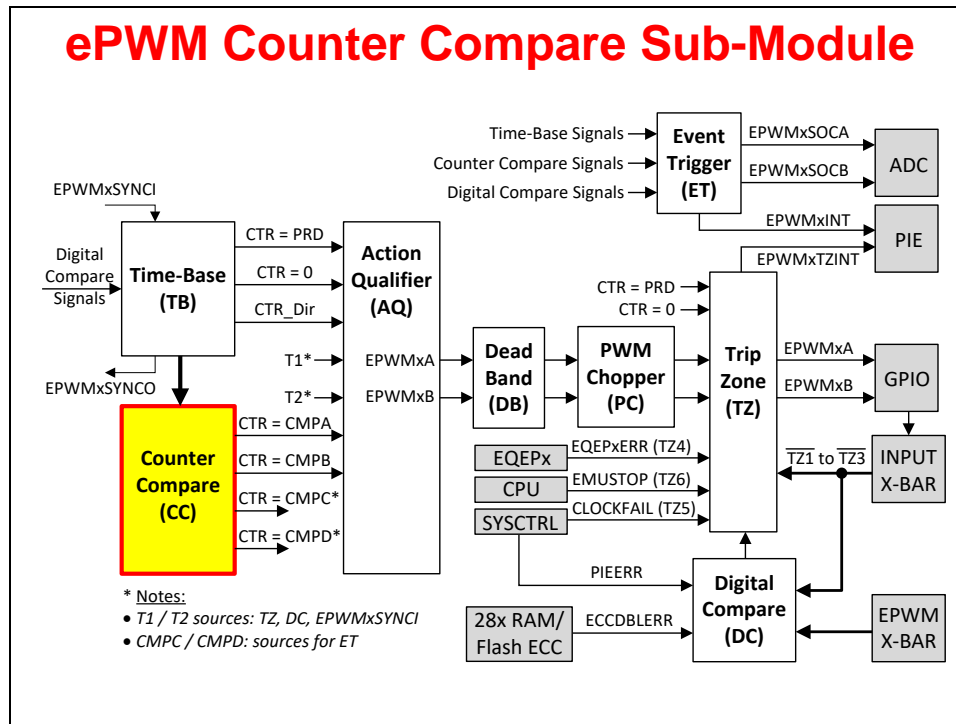


Time-Base Driverlib Functions

- ◆ **Set Time-Base clock (HSPCLKDIV and CLKDIV)**
`EPWM_setClockPrescaler(base, prescaler, highSpeedPrescaler);`
 - ◆ **Set Time-Base count mode**
`EPWM_setTimeBaseCounterMode(base, counterMode);`
 - ◆ **Set Time-Base period value and period load mode**
`EPWM_setTimeBasePeriod(base, periodCount);`
`EPWM_setPeriodLoadMode(base, loadMode);`
 - ◆ **Set EPWMSYNC out pulse event**
`EPWM_setSyncOutPulseMode(base, mode);`
- ◆ *base* is the ePWM base address: EPWM_x_BASE (x = 1 to 8)
 - ◆ *prescaler* value is: EPWM_CLOCK_DIVIDER_x (x = 2ⁿ where n is 0 to 7)
 - ◆ *highSpeedPrescaler* value is: EPWM_HSCLOCK_DIVIDER_x (x = 1 or an even value between 2 and 14 inclusive)
 - ◆ *counterMode* value is: EPWM_COUNTER_MODE_x (x = UP, DOWN, UP_DOWN, or STOP_FREEZE)
 - ◆ *periodCount* can have a maximum value of 0xFFFF
 - ◆ *loadMode* value is: EPWM_PERIOD_x (x = SHADOW_LOAD or DIRECT_LOAD)
 - ◆ *mode* value is: EPWM_SYNC_OUT_PULSE_ON_x (x = SOFTWARE, COUNTER_ZERO, COUNTER_COMPARE_y (y = B, C, or D), or EPWM_SYNC_OUT_PULSE_DISABLED)

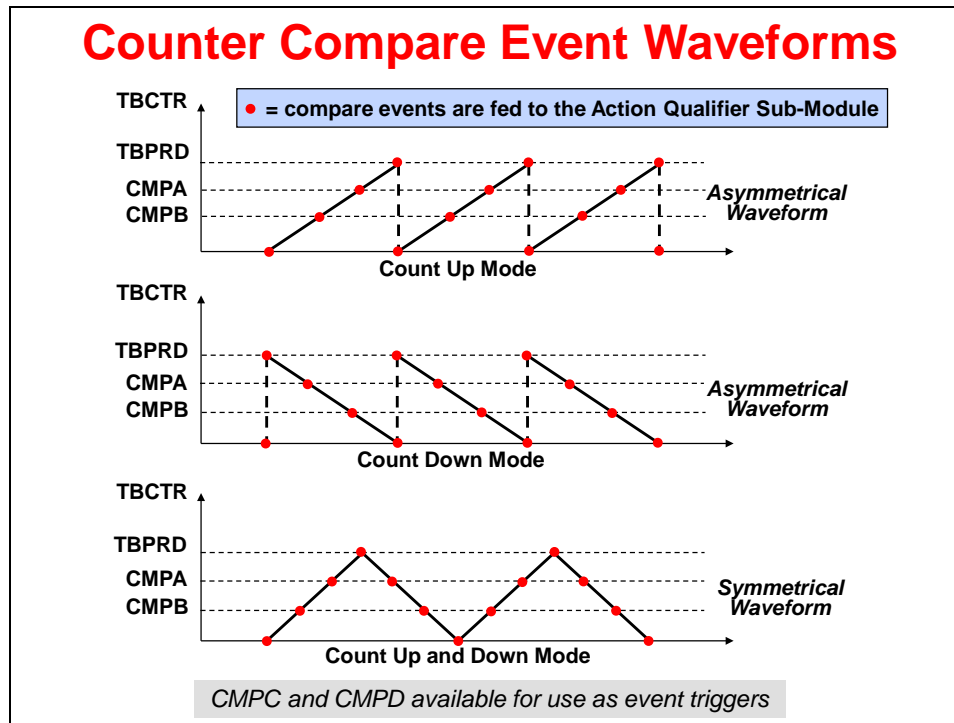
ePWM Compare Sub-Module

ePWM Counter Compare Sub-Module



The counter-compare submodule continuously compares the time-base count value to four counter compare registers (CMPA, CMPB, CMPC, and CMPD) and generates four independent compare events (i.e. time-base counter equals a compare register value) which are fed to the

action-qualifier and event-trigger submodules. The counter compare registers are shadowed to prevent corruption or glitches during the active PWM cycle. Typically CMPA and CMPB are used to control the duty cycle of the generated PWM waveform, and all four compare registers can be used to start an ADC conversion or generate an ePWM interrupt. For the up-count and down-count modes, a counter match occurs only once per cycle, however for the up-down-count mode a counter match occurs twice per cycle since there is a match on the up count and down count.

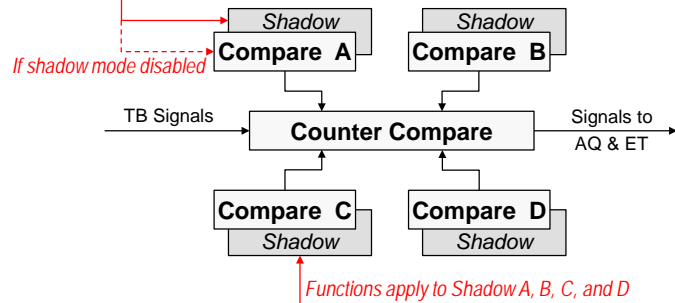


The above ePWM Compare Event Waveform diagram shows the compare matches which are fed into the action qualifier. Notice that with the count up and count down mode, there are matches on the up-count and down-count.

Counter Compare Functional Diagram

`EPWM_setCounterCompareValue(base, compModule, compCount);`

Function applies to Compare A, B, C, and D



`EPWM_setCounterCompareShadowLoadMode(base, compModule, loadMode);`

Shadow mode – the compare register is double buffered

`EPWM_disableCounterCompareShadowLoadMode(base, compModule);`

Immediate mode – the shadow register is not used

Counter Compare Driverlib Functions

◆ Set Counter Compare value

`EPWM_setCounterCompareValue(base, compModule, compCount);`

◆ Enable and set Counter Compare shadow load mode

`EPWM_setCounterCompareShadowLoadMode(base, compModule, loadMode);`

◆ Disable Counter Compare shadow load mode

`EPWM_disableCounterCompareShadowLoadMode(base, compModule);`

◆ *base* is the ePWM base address: EPWM_x_BASE (*x* = 1 to 8)

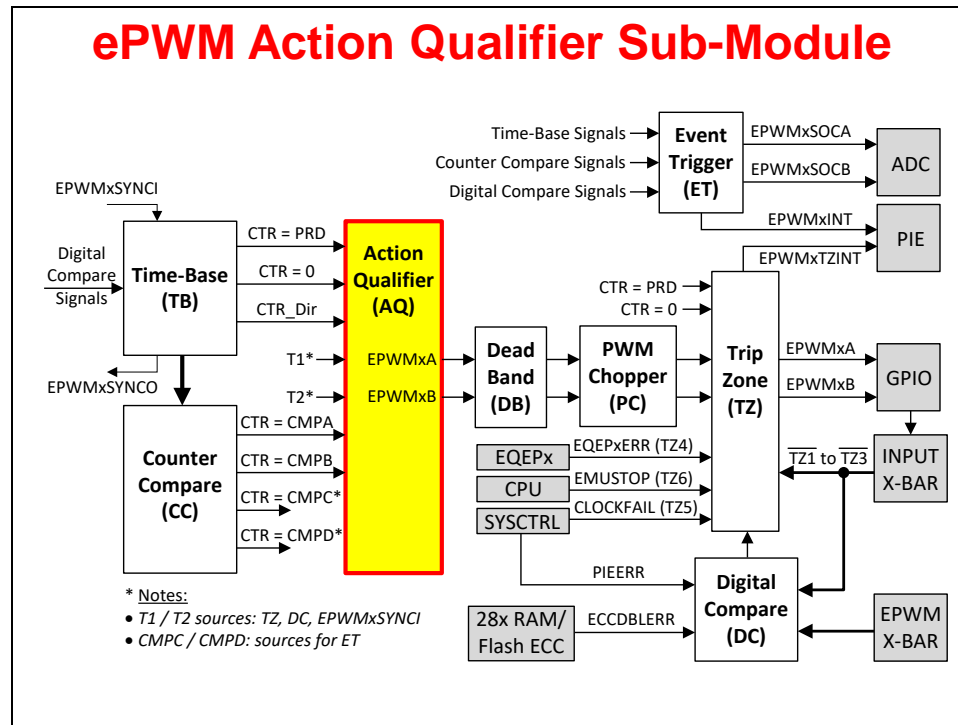
◆ *compModule* value is: EPWM_COUNTER_COMPARE_*x* (*x* = A, B, C, or D)

◆ *compCount* can have a maximum value of 0xFFFF

◆ *loadMode* value is:

- ◆ EPWM_COMP_LOAD_ON_CNTR_*x* (*x* = ZERO, PERIOD, or ZERO_PERIOD)
- ◆ EPWM_COMP_LOAD_FREEZE
- ◆ EPWM_COMP_LOAD_ON_SYNC_CNTR_*x* (*x* = ZERO, PERIOD, or ZERO_PERIOD)
- ◆ EPWM_COMP_LOAD_ON_SYNC_ONLY

ePWM Action Qualifier Sub-Module



The action-qualifier submodule is the key element in the ePWM module which is responsible for constructing and generating the switched PWM waveforms. It utilizes match events from the time-base and counter-compare submodules for performing actions on the EPWMxA and EPWMxB output pins. These first three submodules are the main blocks which are used for generating a basic PWM waveform.

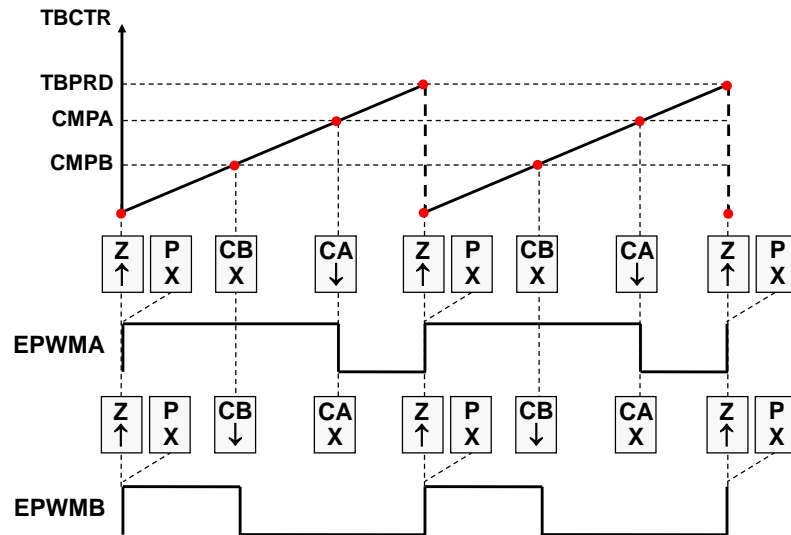
ePWM Action Qualifier Actions							
for EPWMA and EPWMB							
S/W Force	Time-Base Counter equals:				Trigger Events:		EPWM Output Actions
	Zero	CMPA	CMPB	TBPRD	T1	T2	
SW X	Z X	CA X	CB X	P X	T1 X	T2 X	Do Nothing
SW ↓	Z ↓	CA ↓	CB ↓	P ↓	T1 ↓	T2 ↓	Clear Low
SW ↑	Z ↑	CA ↑	CB ↑	P ↑	T1 ↑	T2 ↑	Set High
SW T	Z T	CA T	CB T	P T	T1 T	T2 T	Toggle

Tx Event Sources = DCAEVT1, DCAEVT2, DCBEVT1, DCBEVT2, TZ1, TZ2, TZ3, EPWMxSYNCIN

The Action Qualifier actions are setting the pin high, clearing the pin low, toggling the pin, or doing nothing to the pin, based independently on count-up and count-down time-base match event. The match events are when the time-base counter equals the period register value, the time-base counter is zero, the time-base counter equals CMPA, the time-base counter equals CMPB, or a Trigger event (T1 and T2) based on a comparator, trip, or sync signal. Note that zero and period actions are fixed in time, whereas CMPA and CMPB actions are movable in time by programming their respective registers. Actions are configured independently for each output using shadowed registers, and any or all events can be configured to generate actions on either output. Also, the output pins can be forced to any action using software.

Count Up Asymmetric Waveform

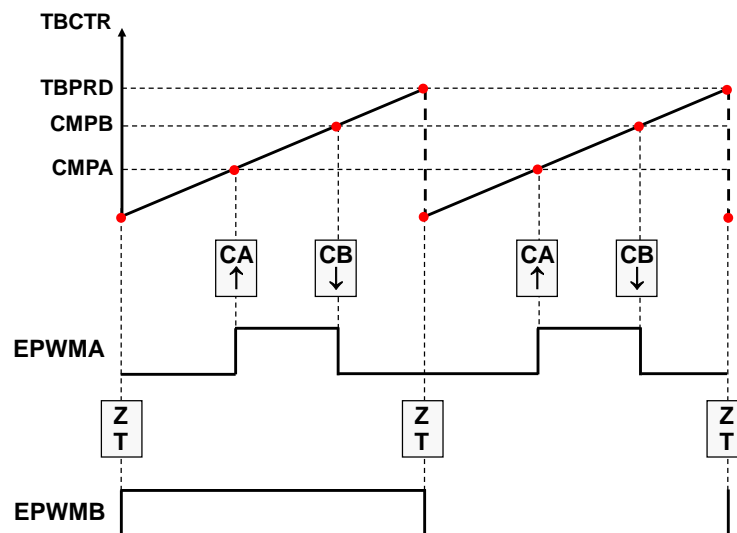
with Independent Modulation on EPWMA / B



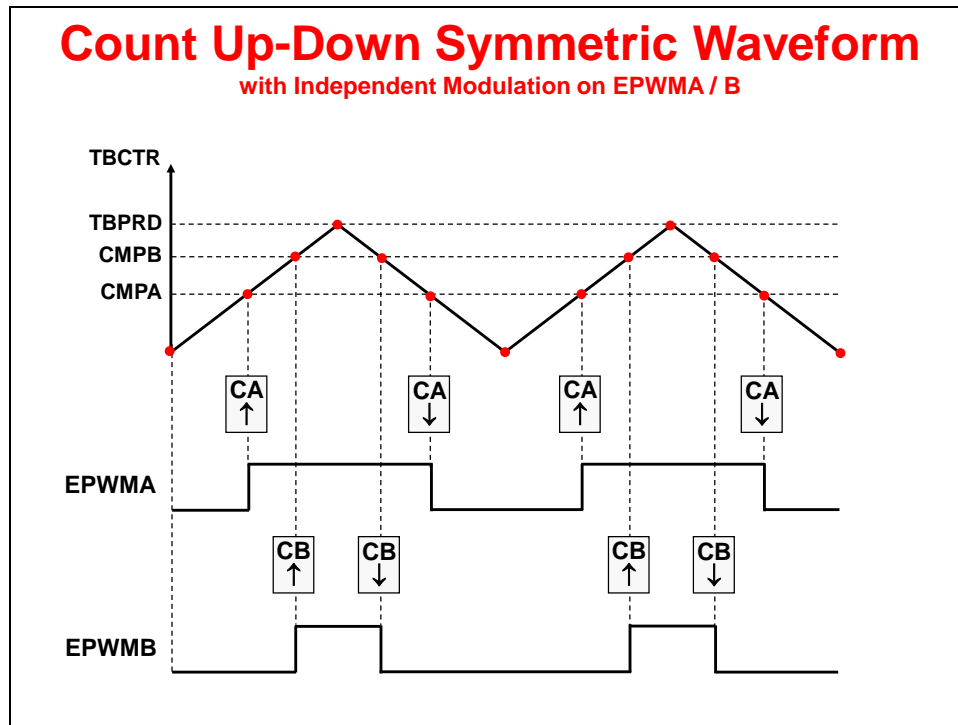
The next few figures show how the setting of the action qualifier with the compare matches are used to modulate the output pins. Notice that the output pins for EPWMA and EPWMB are completely independent. In the example above, the EPWMA output is being set high on the zero match and cleared low on the compare A match. The EPWMB output is being set high on the zero match and cleared low on the compare B match.

Count Up Asymmetric Waveform

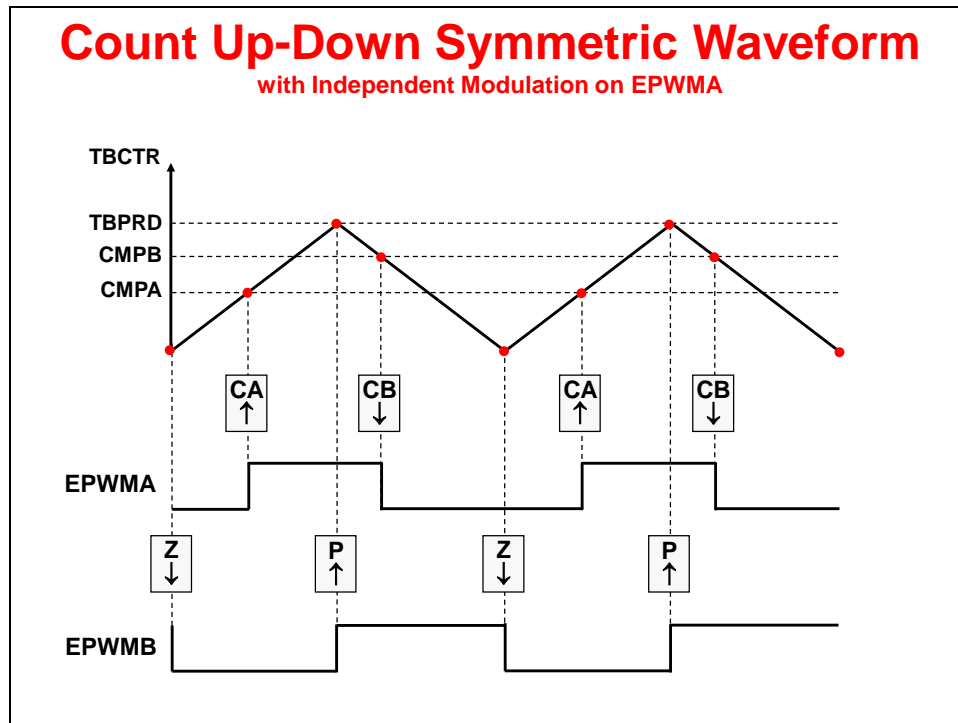
with Independent Modulation on EPWMA



In the example above, the EPWMA output is being set high on the compare A match and being cleared low on the compare B match, while the EPWMB output is being toggled on the zero match.



In the example above, there are different output actions on the up-count and down-count using a single compare register. The EPWMA and EPWMB outputs are being set high on the compare A and B up-count matches and cleared low on the compare A and B down-count matches.



And finally in the example above, again using different output actions on the up-count and down-count, the EPWMA output is being set high on the compare A up-count match and being cleared low on the compare B down-count match. The EPWMB output is being cleared low on the zero match and being set high on the period match.

Action Qualifier Driverlib Functions

- ◆ **Configure Action Qualifier output on ePWMA or ePWMB**

```
EPWM_setActionQualifierAction(base, epwmOutput, output,  
                               event);
```

- ◆ **Set Action Qualifier trigger source for event T1 or T2**

```
EPWM_setActionQualifierT1TriggerSource(base, trigger);
```

```
EPWM_setActionQualifierT2TriggerSource(base, trigger);
```

- ◆ *base* is the ePWM base address: EPWM x _BASE ($x = 1$ to 8)
- ◆ *epwmOutput* value is: EPWM_AQ_OUTPUT_ x ($x = A$ or B)
- ◆ *output* value is: EPWM_AQ_OUTPUT_ x ($x = NO_CHANGE, LOW, HIGH, or TOGGLE$)
- ◆ *event* value is:
 - ◆ EPWM_AQ_OUTPUT_ON_TIMEBASE_ x ($x = ZERO, PERIOD, UP_CMPA, DOWN_CMPA, UP_CMPB, or DOWN_CMPB$)
 - ◆ EPWM_AQ_OUTPUT_ON_T1_ x ($x = COUNT_UP or COUNT_DOWN$)
 - ◆ EPWM_AQ_OUTPUT_ON_T2_ x ($x = COUNT_UP or COUNT_DOWN$)
- ◆ *trigger* value is: EPWM_AQ_TRIGGER_EVENT_TRIG_ x ($x = DCA_1, DCA_2, DCB_1, DCB_2, TZ_1, TZ_2, TZ_3, or EPWM_SYNCIN$)

Action Qualifier Driverlib Functions

- ◆ **Enable and set Action Qualifier shadow load mode**

```
EPWM_setActionQualifierShadowLoadMode(base, aqModule,  
                                         loadMode);
```

- ◆ **Disable Action Qualifier shadow load mode**

```
EPWM_disableActionQualifierShadowLoadMode(base,  
                                             aqModule);
```

- ◆ *base* is the ePWM base address: EPWM x _BASE ($x = 1$ to 8)
- ◆ *aqModule* value is: EPWM_ACTION_QUALIFIER_ x ($x = A$ or B)
- ◆ *loadMode* value is:
 - ◆ EPWM_AQ_LOAD_ON_CNTR_ x ($x = ZERO, PERIOD, or ZERO_PERIOD$)
 - ◆ EPWM_AQ_LOAD_FREEZE
 - ◆ EPWM_AQ_LOAD_ON_SYNC_CNTR_ x ($x = ZERO, PERIOD, or ZERO_PERIOD$)
 - ◆ EPWM_AQ_LOAD_ON_SYNC_ONLY

Action Qualifier Driverlib Functions

- ◆ Trigger continuous software forced output on ePWM (A or B)
`EPWM_setActionQualifierContSWForceAction(base, epwmOutput, output);`
 - ◆ Set continuous software force shadow reload mode
`EPWM_setActionQualifierContSWForceShadowMode(base, mode);`
 - ◆ Set one time software forced action output and force action
`EPWM_setActionQualifierSWAction(base, epwmOutput, output*);`
`EPWM_forceActionQualifierSWAction(base, epwmOutput);`
- ◆ *base* is the ePWM base address: EPWM_x_BASE (x = 1 to 8)
- ◆ *epwmOutput* value is: EPWM_AQ_OUTPUT_x (x = A or B)
- ◆ *output* value is: EPWM_AQ_x (x = SW_DISABLED, OUTPUT_LOW, or OUTPUT_HIGH)
- ◆ *mode* value is:
- ◆ EPWM_AQ_SW_SH_LOAD_ON_x (x = CNTR_ZERO, CNTR_PERIOD, or CNTR_ZERO_PERIOD)
 - ◆ EPWM_AQ_SW_IMMEDIATE_LOAD
- ◆ *output** value is: EPWM_AQ_OUTPUT_x (x = NO_CHANGE, LOW, HIGH, or TOGGLE)

Asymmetric and Symmetric Waveform Generation using the ePWM

PWM switching frequency:

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

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

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

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

PWM resolution:

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

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

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

Note that in the symmetric case, you could actually update the compare value for both the rising and falling PWM edges. This would effectively give you $2 * 500 = 1000$ 'Period' values, and hence the same resolution as the asymmetric case.

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

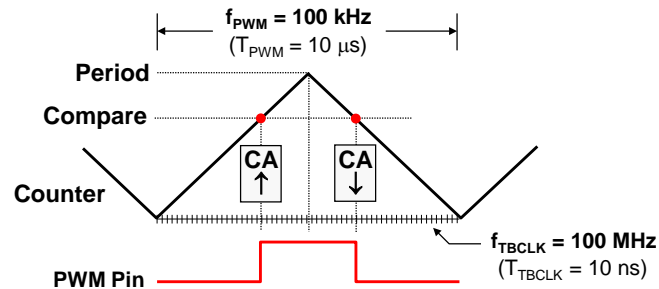
Symmetric PWM: $\text{TxCMPR} = (100\% - \text{duty cycle}) * \text{TxPR}$

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 100 kHz, 25% duty symmetric PWM from a 100 MHz time base clock

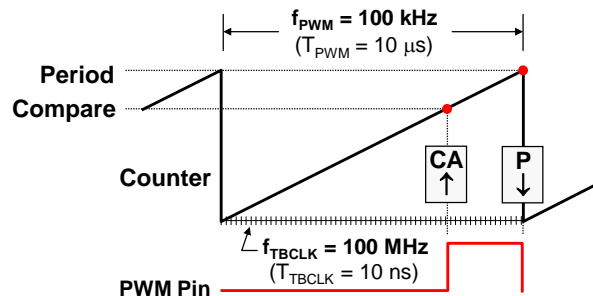


$$TBPRD = \frac{1}{2} \cdot \frac{f_{TBCLK}}{f_{PWM}} = \frac{1}{2} \cdot \frac{100 \text{ MHz}}{100 \text{ kHz}} = 500$$

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

Asymmetric PWM Computation Example

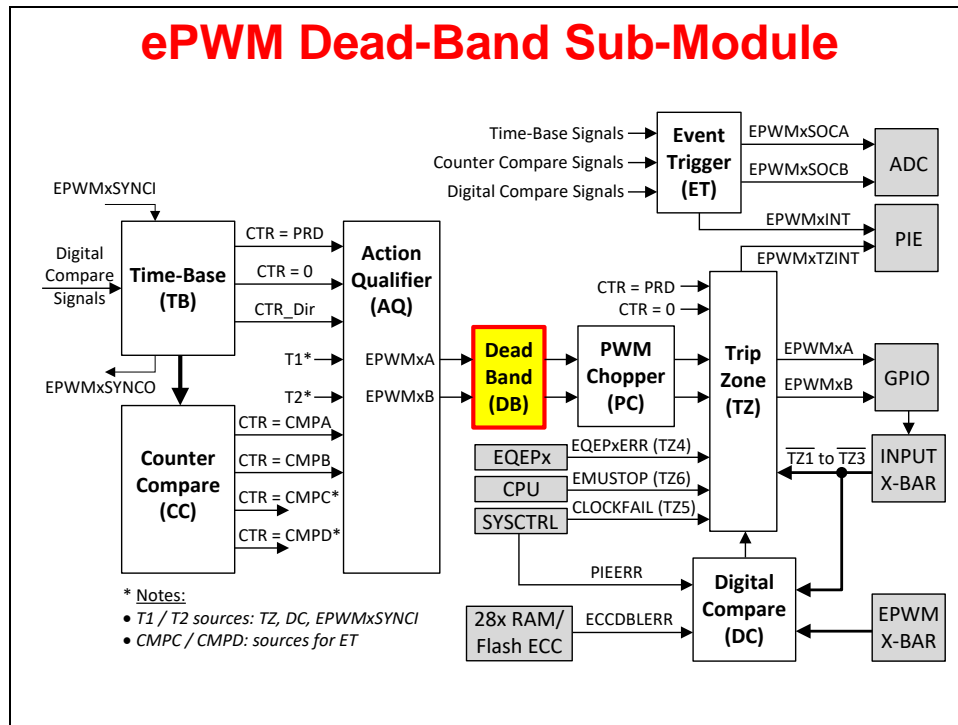
- ◆ Determine TBPRD and CMPA for 100 kHz, 25% duty asymmetric PWM from a 100 MHz time base clock



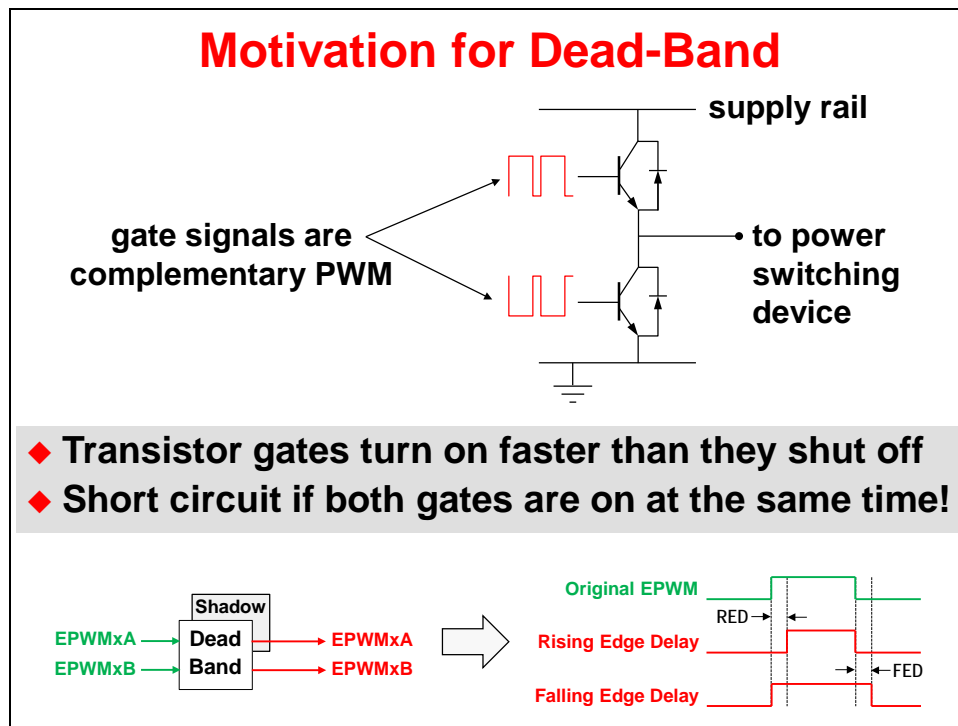
$$TBPRD = \frac{f_{TBCLK}}{f_{PWM}} - 1 = \frac{100 \text{ MHz}}{100 \text{ kHz}} - 1 = 999$$

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

ePWM Dead-Band Sub-Module

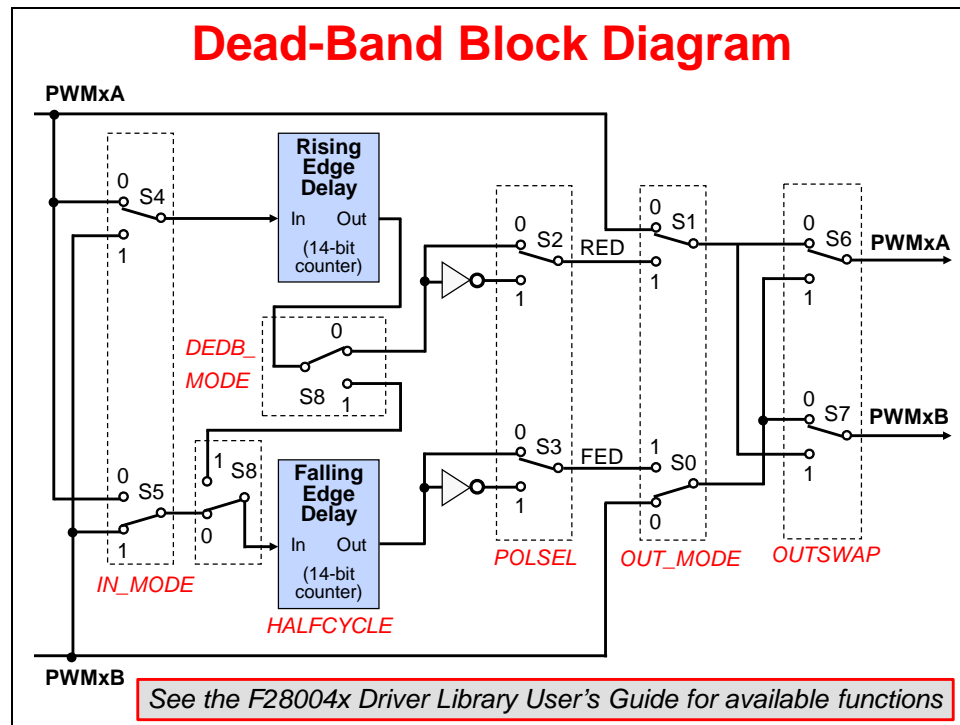


The dead-band sub-module provides a means to delay the switching of a gate signal, thereby allowing time for gates to turn off and preventing a short circuit. This sub-module supports independently programmable rising-edge and falling-edge delays with various options for generating the appropriate signal outputs on EPWMxA and EPWMxB.

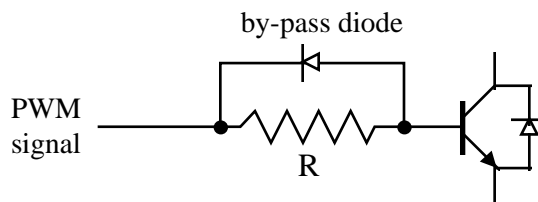


To explain further, power-switching devices turn on faster than they shut off. This issue would momentarily provide a path from supply rail to ground, giving us a short circuit. The dead-band sub-module alleviates this issue.

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 complementary 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.



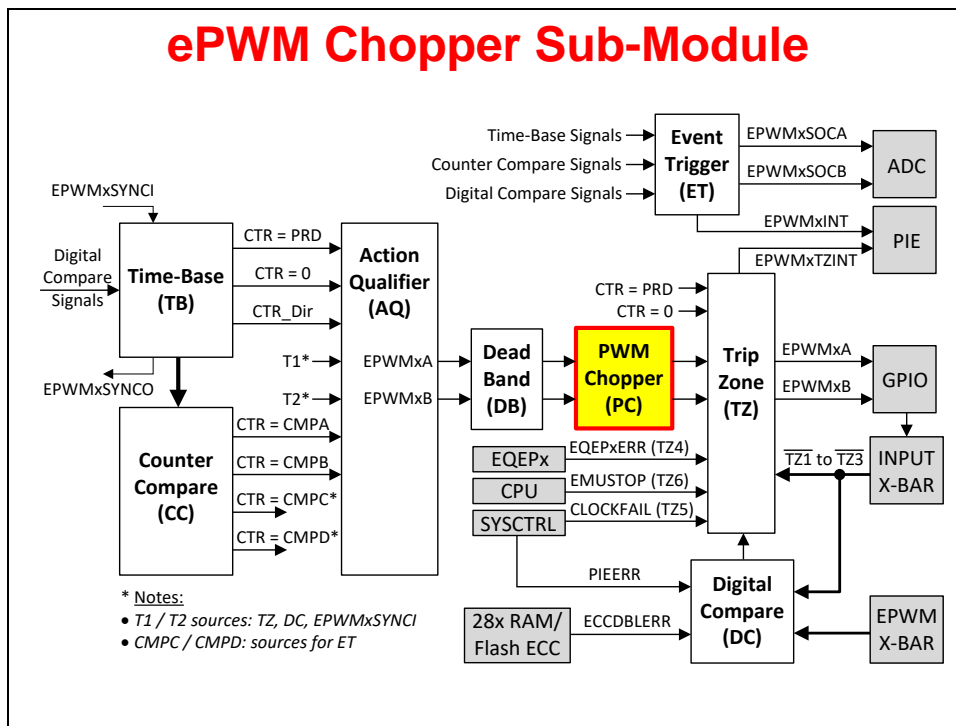
Shoot-through control via power circuit modification

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 complementary 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 Chopper Sub-Module

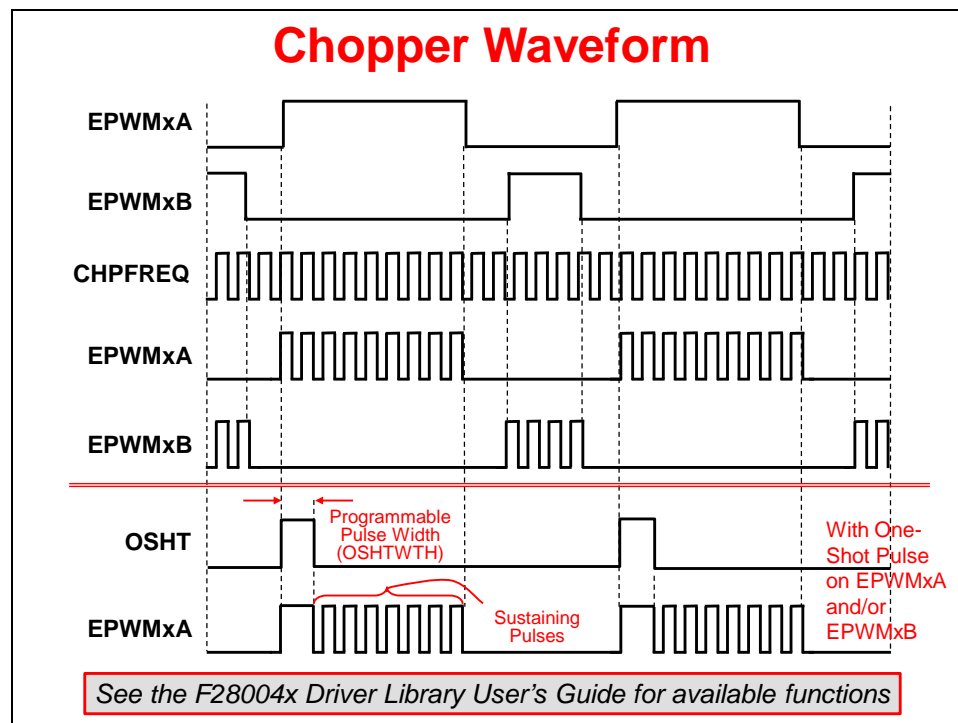


The PWM chopper submodule is used with pulse transformer-based gate drives to control the power switching devices. This submodule modulates a high-frequency carrier signal with the PWM waveform that is generated by the action-qualifier and dead-band submodules. Programmable options are available to support the magnetic properties and characteristics of the transformer and associated circuitry.

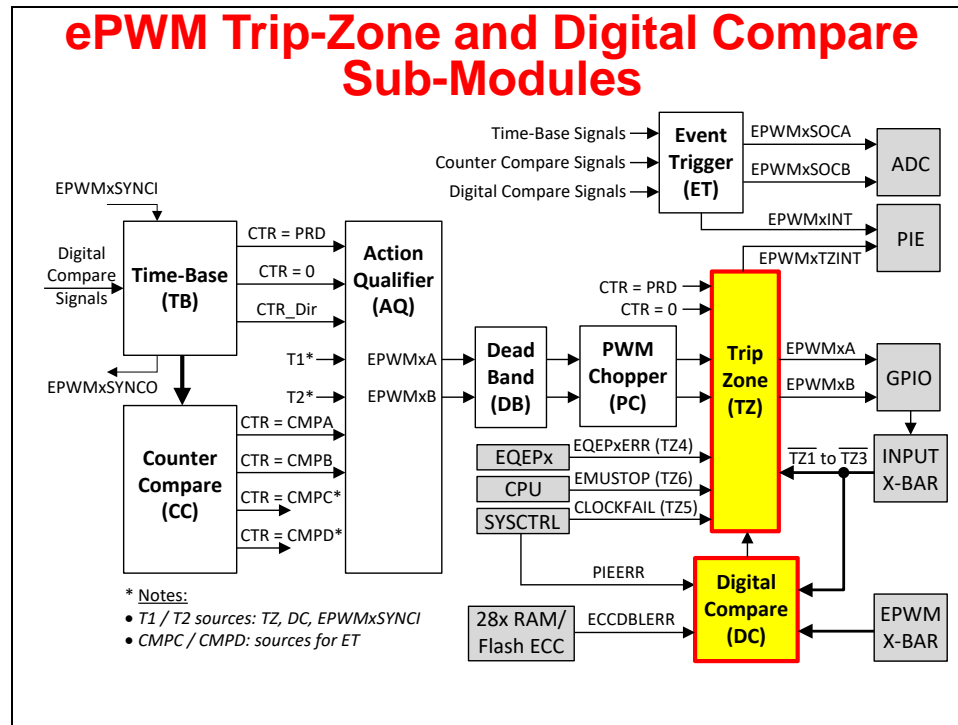
Purpose of the PWM Chopper

- ◆ 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

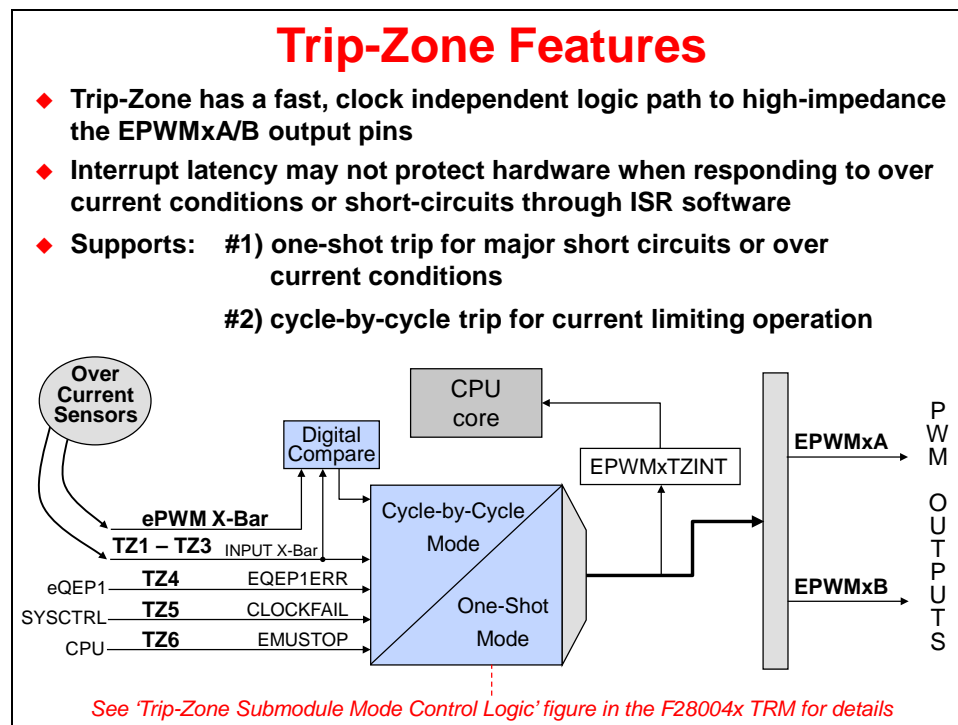
Shown in the figure below, a high-frequency carrier signal is ANDed with the ePWM outputs. Also, this circuit provides an option to include a larger, one-shot pulse width before the sustaining pulses.



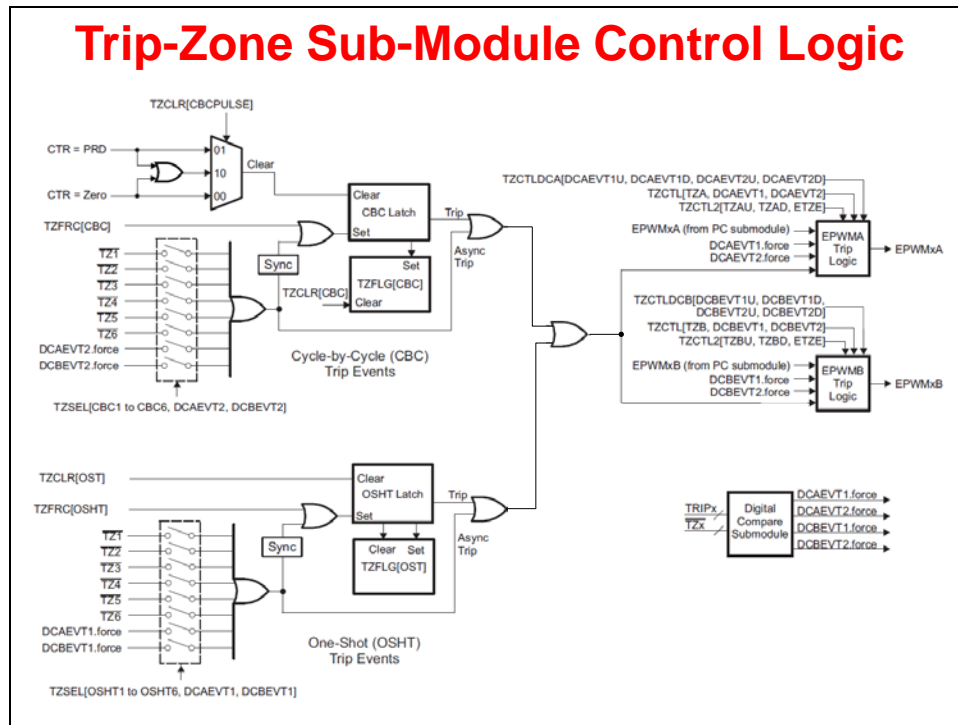
ePWM Trip-Zone and Digital Compare Sub-Modules



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.



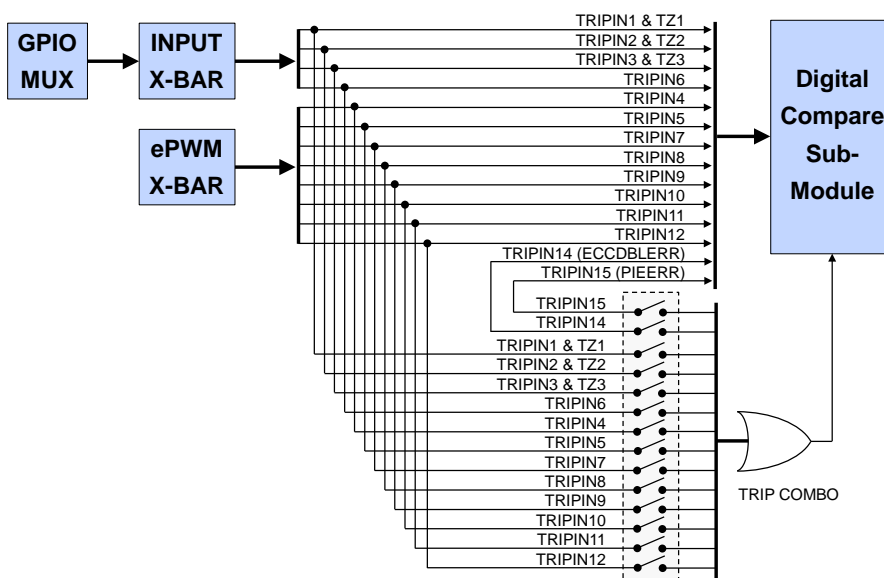
The trip-zone submodule utilizes a fast clock independent logic mechanism to quickly handle fault conditions by forcing the EPWMxA and EPWMxB outputs to a safe state, such as high, low, or high-impedance, thus avoiding any interrupt latency that may not protect the hardware when responding to over current conditions or short circuits through ISR software. It supports one-shot trips for major short circuits or over current conditions, and cycle-by-cycle trips for current limiting operation. The trip-zone signals can be generated externally from any GPIO pin which is mapped through the Input X-Bar (TZ1 – TZ3), internally from an eQEP error signal (TZ4), system clock failure (TZ5), or from an emulation stop output from the CPU (TZ6). Additionally, numerous trip-zone source signals can be generated from the digital-compare subsystem.



Trip-Zone Driverlib Functions

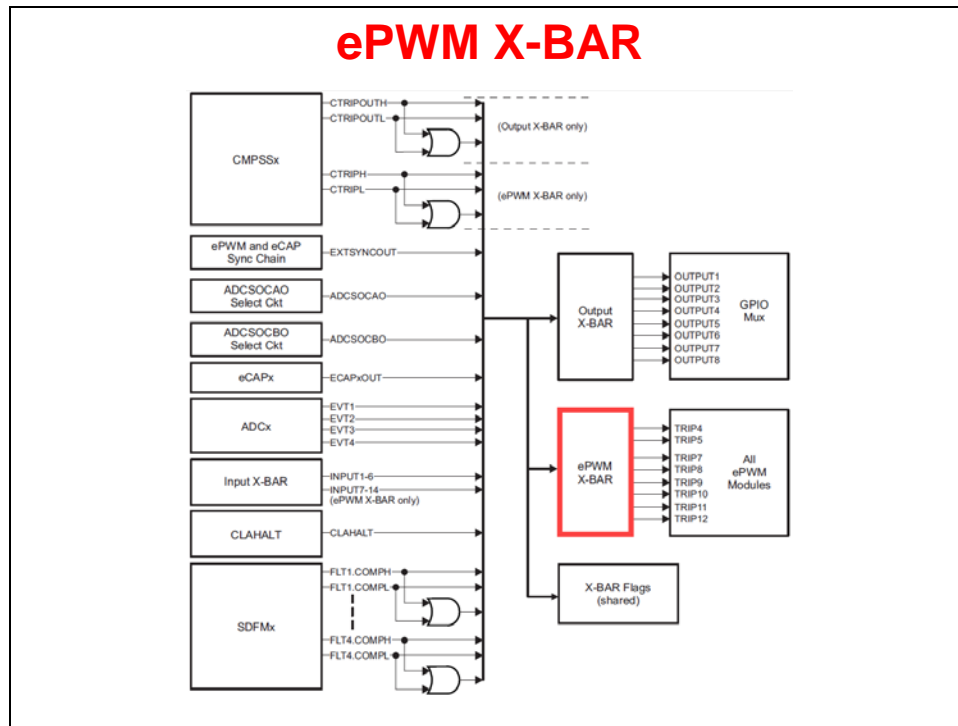
- ◆ Enable / disable Trip-Zone signals
`EPWM_[enable|disable]TripZoneSignals(base, tzSignal);`
 - ◆ Set Trip-Zone Action
`EPWM_setTripZoneAction(base, tzEvent, tzAction);`
 - ◆ Enable / disable Trip-Zone interrupts
`EPWM_[enable|disable]TripZoneInterrupt(base, tzInterrupt);`
- ◆ *base* is the ePWM base address: EPWM_x_BASE (x = 1 to 8)
- ◆ *tzSignal* is logical OR values of:
- ◆ EPWM_TZ_SIGNAL_CBC_x (x = 1 to 6)
 - ◆ EPWM_TZ_SIGNAL_x (x = DCAEVT2 or DCBEVT2) <CBC>
 - ◆ EPWM_TZ_SIGNAL_OSHT_x (x = 1 to 6)
 - ◆ EPWM_TZ_SIGNAL_x (x = DCAEVT1 or DCBEVT1) <OSHT>
- ◆ *tzEvent* value is: EPWM_TZ_ACTION_EVENT_x (x = TZA, TZB, DCAEVT1, DCAEVT2, DCBEVT1, or DCBEVT2)
- ◆ *tzAction* value is: EPWM_TZ_ACTION_x (x = HIGH_Z, HIGH, LOW, or DISABLE)
- ◆ *tzInterrupt* is logical OR values of: EPWM_TZ_INTERRUPT_x (x = CBC, OST, DCAEVT1, DCAEVT2, DCBEVT1, or DCBEVT2)

Digital Compare Trip Inputs

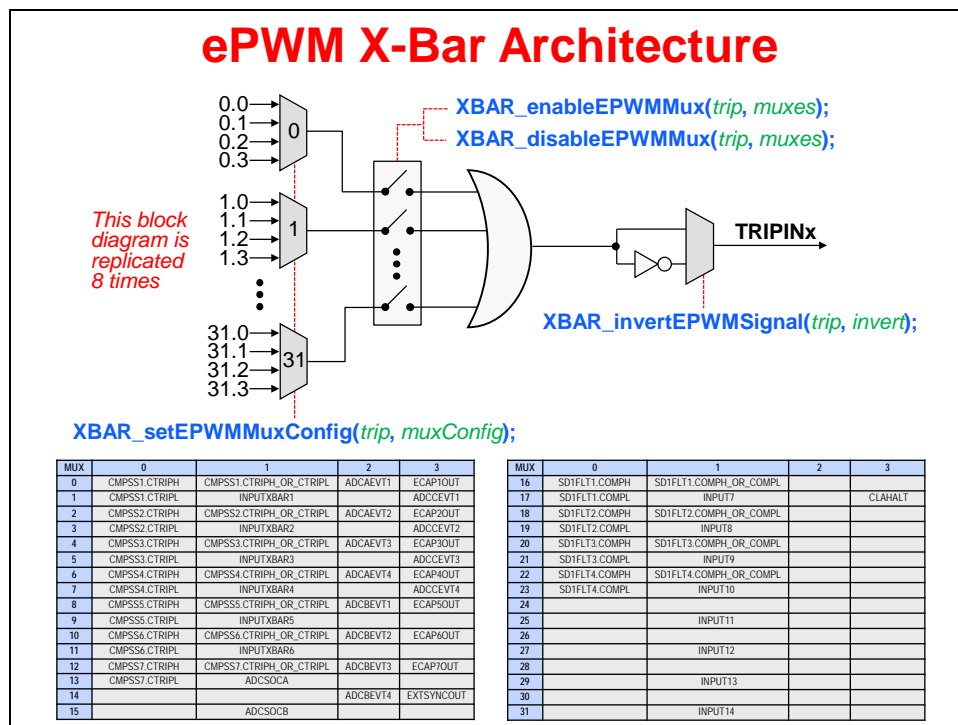


The digital compare submodules receive their trip signals from the Input X-BAR and ePWM X-BAR.

The ePWM X-BAR is used to route various internal and external signals to the ePWM modules. Eight trip signals from the ePWM X-BAR are routed to all of the ePWM modules.



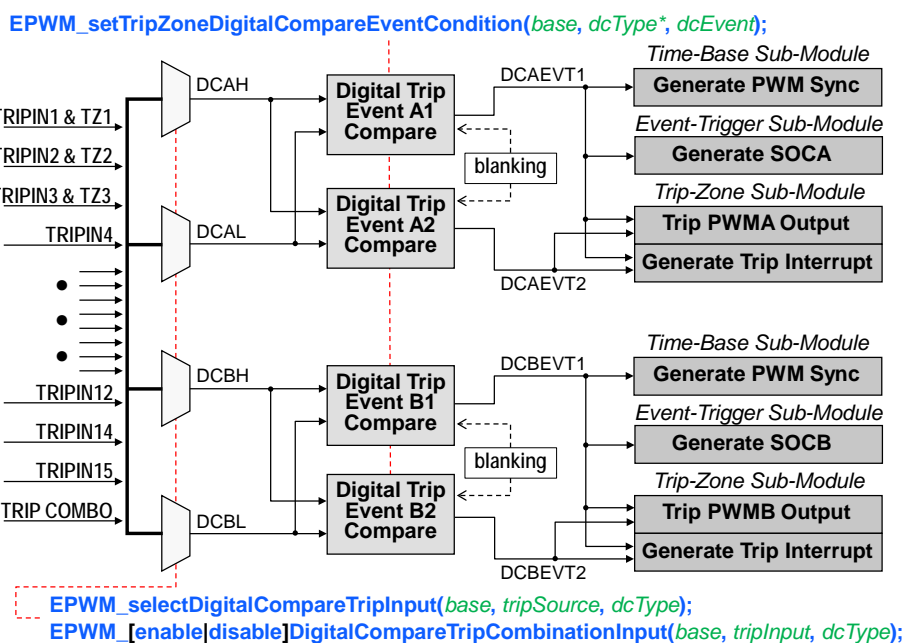
The ePWM X-BAR architecture block diagram shown below is replicated 8 times. The ePWM X-BAR can select a single signal or logically OR up to 32 signals. The table in the figure defines the various trip sources that can be multiplexed to the trip-zone and digital compare submodules.



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
- ◆ Digital compare module inputs are:
 - ◆ Input X-Bar
 - ◆ ePWM X-Bar
 - ◆ Trip-zone input pins
- ◆ A compare event is generated when one or more of its selected inputs are either high or low
- ◆ Optional 'Blanking' can be used to temporarily disable the compare action in alignment with PWM switching to eliminate noise effects

Digital Compare Sub-Module Signals



The digital-compare subsystem compares signals external to the ePWM module, such as a signal from the CMPSS analog comparators, to directly generate PWM events or actions which are then used by the trip-zone, time-base, and event-trigger submodules. These 'compare' events can trip the ePWM module, generate a trip interrupt, sync the ePWM module, or generate an ADC start of

conversion. A compare event is generated when one or more of its selected inputs are either high or low. The signals can originate from any external GPIO pin which is mapped through the Input X-Bar and from various internal peripherals which are mapped through the ePWM X-Bar. Additionally, an optional 'blanking' function 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:

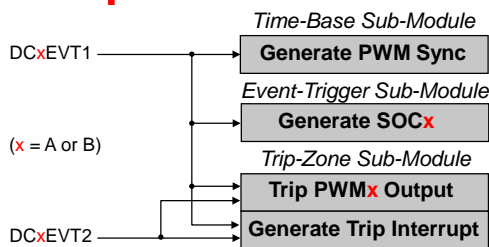
i.	DCyH → low	DCyL → don't care
ii.	DCyH → high	DCyL → don't care
iii.	DCyL → low	DCyH → don't care
iv.	DCyL → high	DCyH → don't care
v.	DCyL → high	DCyH → low

```
EPWM_setTripZoneDigitalCompareEventCondition(base, dcType, dcEvent);
```

Digital Compare Driverlib Functions

- ◆ Select Digital Compare trip inputs
`EPWM_selectDigitalCompareTripInput(base, tripSource, dcType);`
 - ◆ Enable / disable Digital Compare trip combination inputs
`EPWM_[enable|disable]DigitalCompareTripCombinationInput(base, tripInput, dcType);`
 - ◆ Set Digital Compare conditions which cause Trip-Zone events
`EPWM_setTripZoneDigitalCompareEventCondition(base, dcType*, dcEvent);`
- ◆ *base* is the ePWM base address: EPWMx_BASE (x = 1 to 8)
 - ◆ *tripSource* value is:
 - ◆ EPWM_DC_TRIP_TRIPINx (x = 1 to 12, 14, 15)
 - ◆ EPWM_DC_TRIP_COMBINATION - selects trip signals enabled by the EPWM_enableDigitalCompareTripCombinationInput() function
 - ◆ *dcType* value is: EPWM_DC_TYPE_x (x = DCAH, DCAL, DCBH, or DCBL)
 - ◆ *tripInput* value is: EPWM_DC_COMBINATIONAL_TRIPINx (x = 1 to 12, 14, 15)
 - ◆ *dcType** value is: EPWM_TZ_DC_OUTPUT_x (x = A1, A2, B1, or B2)
 - ◆ *dcEvent* value is: EPWM_TZ_EVENT_x (x = DC_DISABLED, DCyH_LOW, DCyH_HIGH, DCyL_LOW, DCyL_HIGH, or DCyL_HIGH_DCyH_LOW)
→ where y in DCyH/DCyL represents DCAH/DCAL or DCBH/DCBL

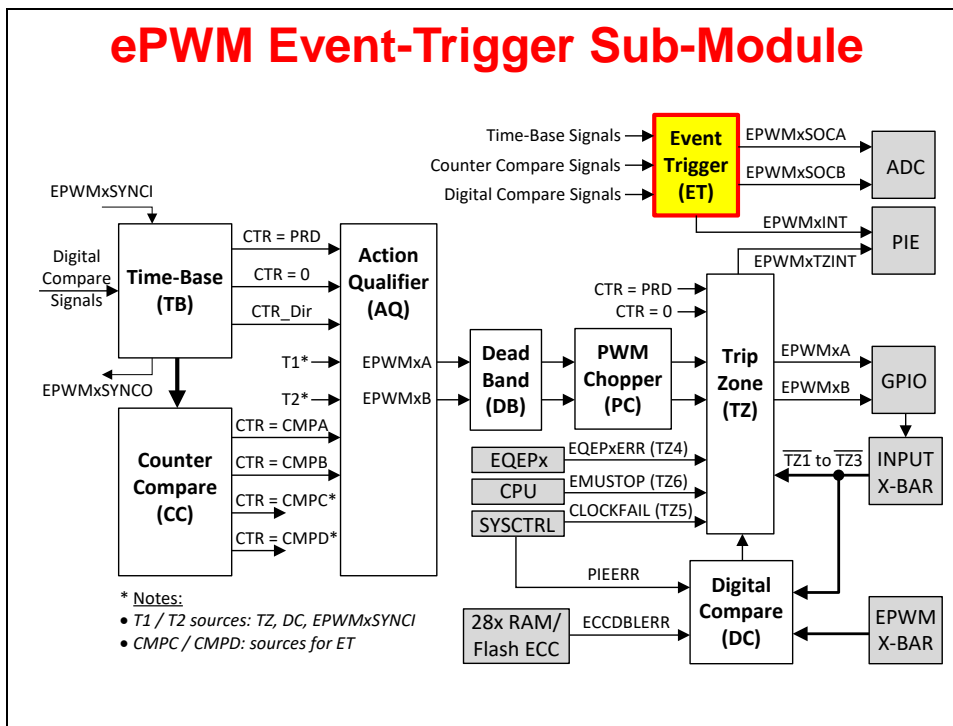
Digital Compare Driverlib Functions



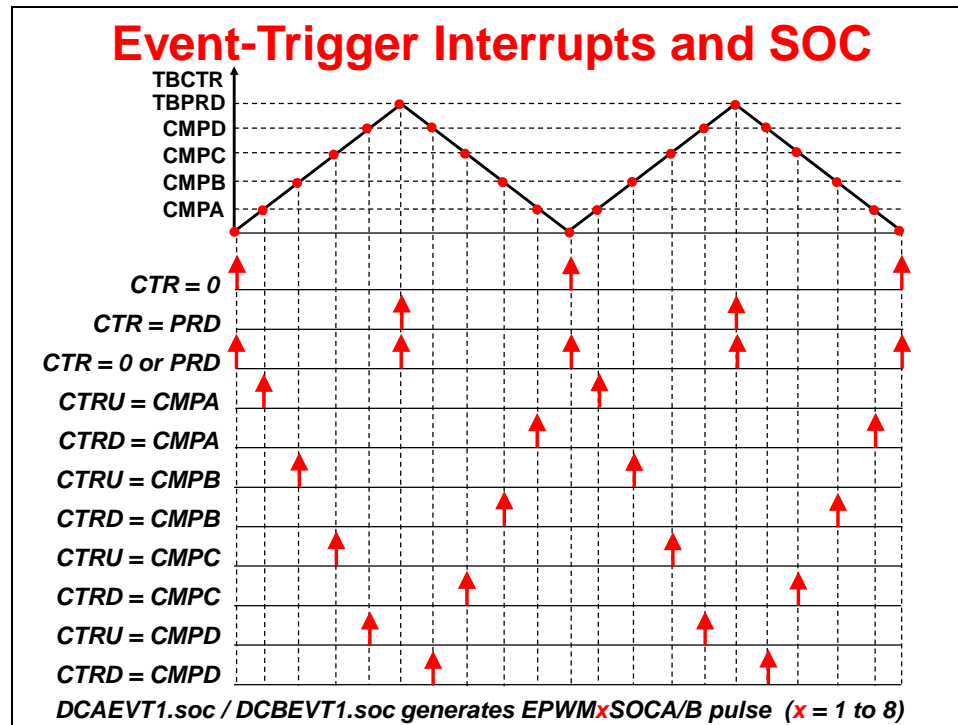
- ◆ Enable / disable Digital Compare Sync Event
`EPWM_[enable|disable]DigitalCompareSyncEvent(base, dcModule);`
 - ◆ Enable / disable Digital Compare ADC trigger
`EPWM_[enable|disable]DigitalCompareADCTrigger(base, dcModule);`
 - ◆ Enable / disable Trip-Zone signals and Trip-Zone interrupts
 - ◆ See Trip-Zone Driverlib Functions:
 - `EPWM_[enable|disable]TripZoneSignals(base, tzSignal);`
 - `EPWM_[enable|disable]TripZoneInterrupt(base, tzInterrupt);`
- ◆ *base* is the ePWM base address: EPWM_x_BASE (x = 1 to 8)
 ◆ *dcModule* value is: EPWM_DC_MODULE_x (x = A or B)

ePWM Event-Trigger Sub-Module

ePWM Event-Trigger Sub-Module



The event-trigger submodule manages the events generated by the time-base, counter-compare, and digital-compare submodules for generating an interrupt to the CPU and/or a start of conversion pulse to the ADC when a selected event occurs.



These event triggers can occur when the time-base counter equals zero, period, zero or period, and the up or down count match of a compare register. Recall that the digital-compare subsystem can also generate an ADC start of conversion based on one or more compare events. Notice counter up and down are independent and separate.

Event-Trigger Driverlib Functions

- ◆ Enable / disable Event-Trigger ADC SOC
`EPWM_[enable|disable]ADCTrigger(base, adcSOCType);`
 - ◆ Set Event-Trigger ADC SOC source
`EPWM_setADCTriggerSource(base, adcSOCType, socSource);`
 - ◆ Set Event-Trigger ADC SOC prescale
`EPWM_setADCTriggerEventPrescale(base, adcSOCType, preScaleCount);`
- ◆ *base* is the ePWM base address: EPWMx_BASE (x = 1 to 8)
 - ◆ *adcSOCType* value is: EPWM_SOC_x (x = A or B)
 - ◆ *socSource* value is:
 - ◆ EPWM_SOC_DCxEVT1 (x = A or B)
 - ◆ EPWM_SOC_TBCTR_x (x = ZERO, PERIOD, ZERO_OR_PERIOD)
 - ◆ EPWM_SOC_TBCTR_U_CMPx (x = A, B, C, or D)
 - ◆ EPWM_SOC_TBCTR_D_CMPx (x = A, B, C, or D)
 - ◆ *preScaleCount* value is: 1 to 15 (0 disables the prescale)

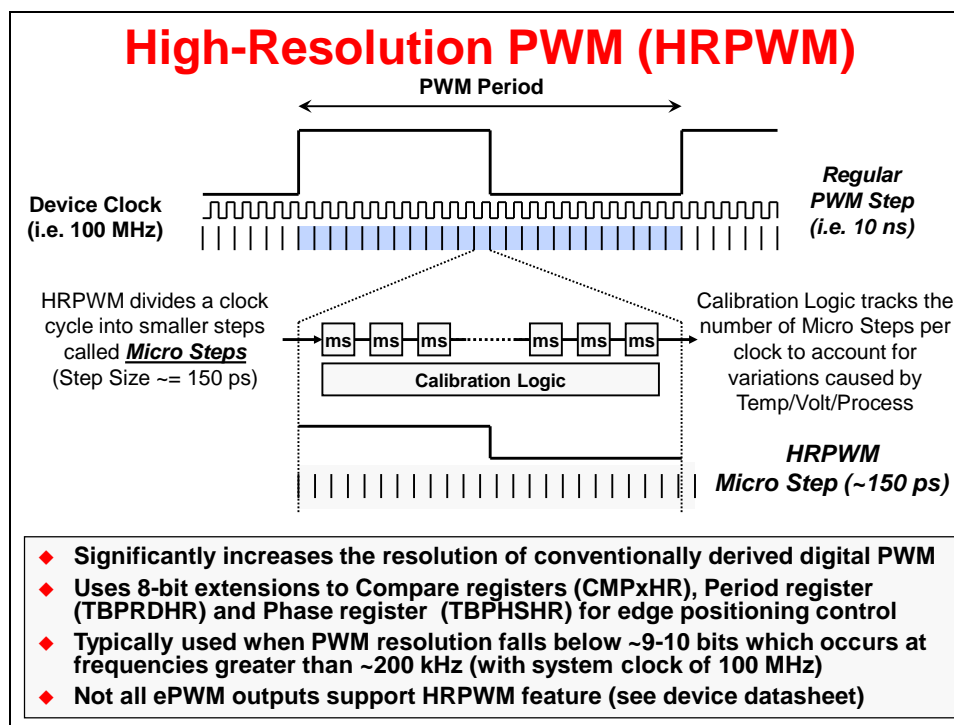
The event-trigger submodule also incorporates pre-scaling logic to issue an interrupt request or ADC start of conversion at every event or up to every fifteenth event.

Event-Trigger Driverlib Functions

- ◆ Enable / disable Event-Trigger interrupt
`EPWM_[enable|disable]Interrupt(base);`
- ◆ Set Event-Trigger interrupt source
`EPWM_setInterruptSource(base, interruptSource);`
- ◆ Set Event-Trigger interrupt event counts
`EPWM_setInterruptEventCount(base, eventCount);`

- ◆ *base* is the ePWM base address: EPWM_x_BASE (x = 1 to 8)
- ◆ *interruptSource* value is:
 - ◆ EPWM_INT_TBCTR_x (x = ZERO, PERIOD, ZERO_OR_PERIOD)
 - ◆ EPWM_INT_TBCTR_U_CMPx (x = A, B, C, or D)
 - ◆ EPWM_INT_TBCTR_D_CMPx (x = A, B, C, or D)
- ◆ *eventCount* determines the number of events that have to occur before an interrupt is issued; (value is: 1 to 15)

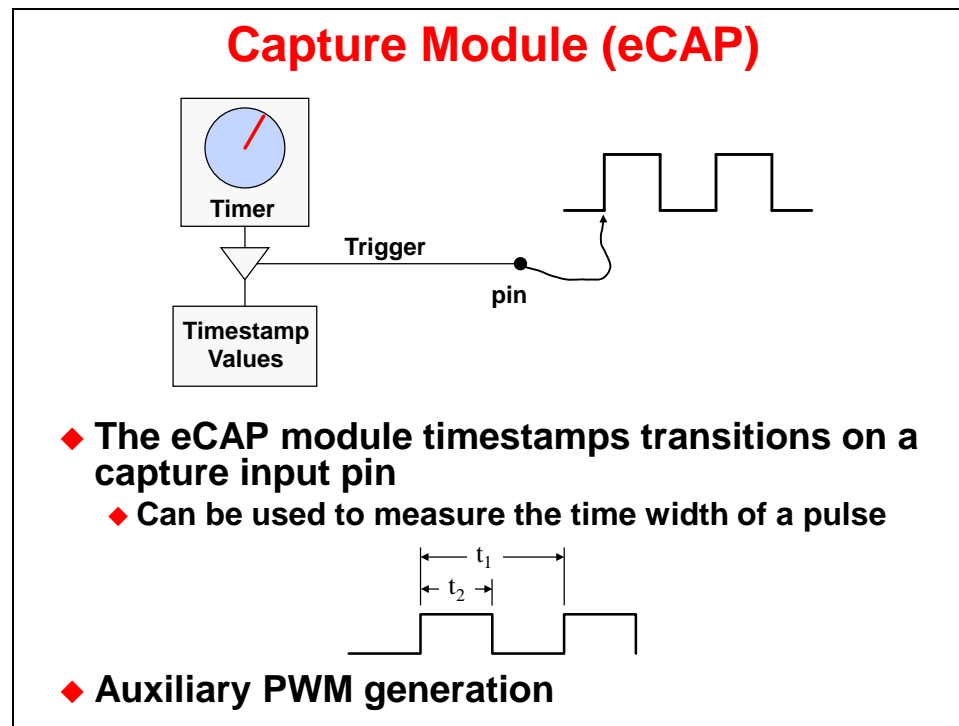
High Resolution PWM (HRPWM)



The ePWM module is capable of significantly increase its time resolution capabilities over the standard conventionally derived digital PWM. This is accomplished by adding 8-bit extensions to

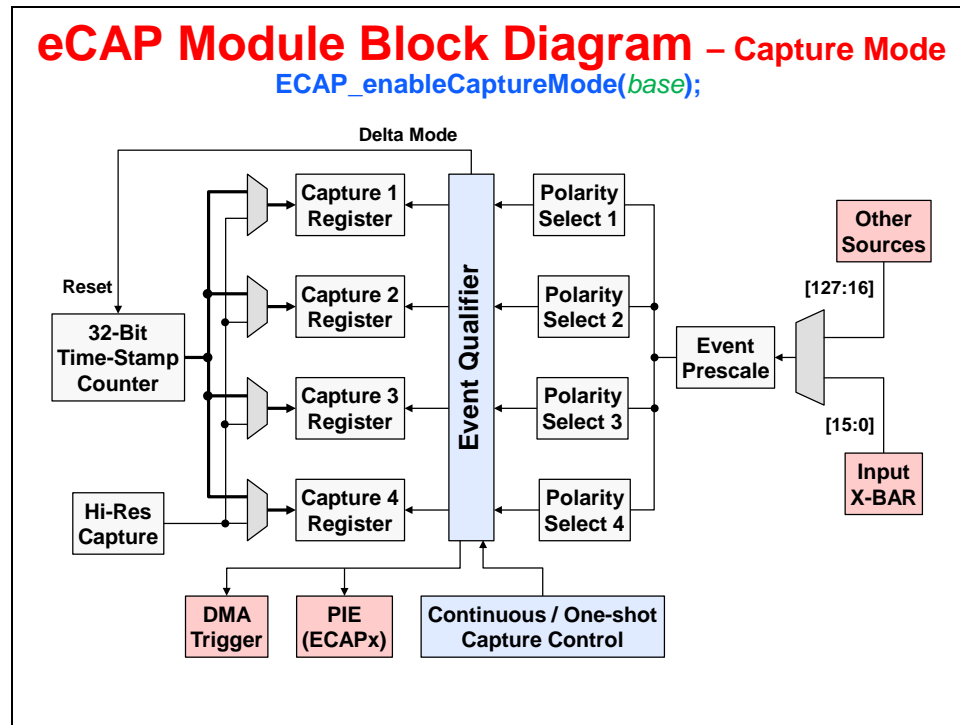
the counter compare register (CMPxHR), period register (TBPRDHR), and phase register (TBPHSHR), providing a finer time granularity for edge positioning control. This is known as high-resolution PWM (HRPWM) and it is based on micro edge positioner (MEP) technology. The MEP logic is capable of positioning an edge very finely by sub-dividing one coarse system clock of the conventional PWM generator with time step accuracy on the order of 150 picoseconds. A self-checking software diagnostics mode is used to determine if the MEP logic is running optimally, under all operating conditions such as for variations caused by temperature, voltage, and process. HRPWM is typically used when the PWM resolution falls below approximately 9 or 10 bits which occurs at frequencies greater than approximately 200 kHz with an EPWMCLK of 100 MHz.

eCAP



The capture units allow time-based logging of external signal transitions. It is used to accurately time external events by timestamping transitions on the capture input pin. It can be used to measure the speed of a rotating machine, determine the elapsed time between pulses, calculate the period and duty cycle of a pulse train signal, and decode current/voltage measurements derived from duty cycle encoded current/voltage sensors.

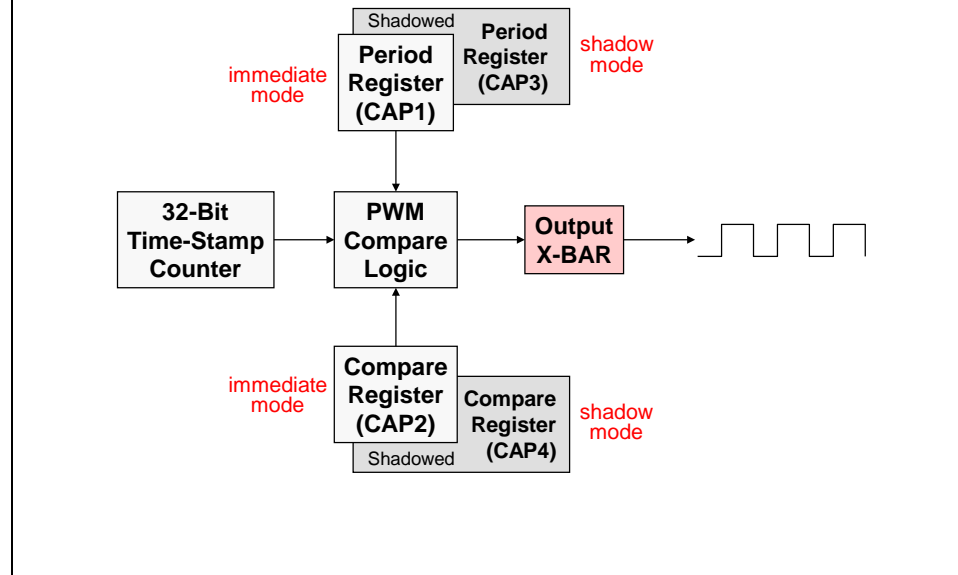
Capture units can be configured to trigger an A/D conversion that is synchronized with an external event. There are several potential advantages to using the capture for this function over the ADCEXTSOC pin associated with the ADC module. First, the ADCEXTSOC pin is level triggered, and therefore only 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 ADCEXTSOC 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 ADCEXTSOC 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.



The eCAP module captures signal transitions on a dedicated input pin and sequentially loads a 32-bit time-base counter value in up to four 32-bit time-stamp capture registers (CAP1 – CAP4). By using a 32-bit counter, rollover is minimized. Independent edge polarity can be configured as rising or falling edge, and the module can be run in either one-shot mode for up to four time-stamp events or continuous mode to capture up to four time-stamp events operating as a circular buffer. The capture input pin is routed through the Input X-Bar, allowing any GPIO pin on the device to be used as the input. Also, the input capture signal can be pre-scaled and interrupts can be generated on any of the four capture events. The time-base counter can be run in either absolute or difference (delta) time-stamp mode. In absolute mode the counter runs continuously, whereas in difference mode the counter resets on each capture

eCAP Module Block Diagram – APWM Mode

```
ECAP_enableAPWMMode(base);
```



If the module is not used in capture mode, the eCAP module can be configured to operate as a single channel asymmetrical PWM module (i.e. time-base counter operates in count-up mode).

eCAP Driverlib Function

- ◆ Enable capture or APWM mode


```
ECAP_enableCaptureMode(base);
```

```
ECAP_enableAPWMMode(base);
```
- ◆ Select eCAP input signal


```
ECAP_selectECAPInput(base, input);
```
- ◆ Stop / start time stamp counter

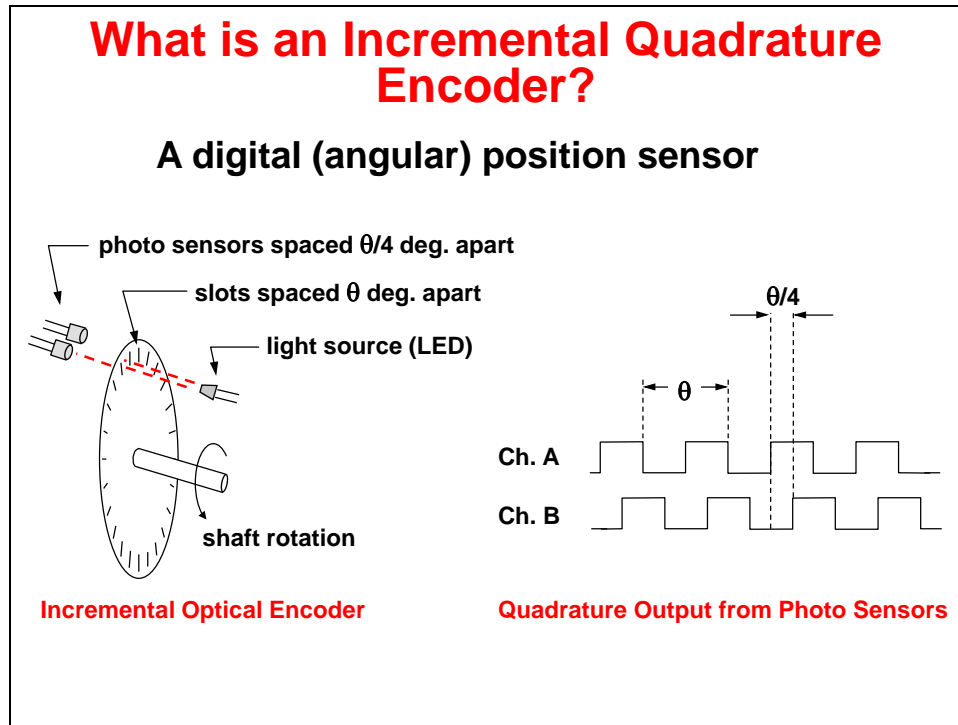

```
ECAP_[stop|start]Counter(base);
```
- ◆ *base* is the eCAP base address: ECAP_x_BASE (*x* = 1 to 7)
- ◆ *input* value is:
 - ◆ ECAP_INPUT_INPUTXBAR_x (*x* = 1 to 16)
 - ◆ ECAP_INPUT_CAN_x_INT0 (*x* = A or B)
 - ◆ ECAP_INPUT_ECAP_DELAY_CLOCK
 - ◆ ECAP_INPUT_OUTPUTXBAR_x (*x* = 1 to 8)
 - ◆ ECAP_INPUT_ADC_y_EVENT_x (*x* = 1 to 4) (*y* = A, B, or C)
 - ◆ ECAP_INPUT_SDFM1_FLT_x_COMPARE_y (*x* = 1 to 4) (*y* = LOW, HIGH, or HIGH_OR_LOW)
 - ◆ ECAP_INPUT_CMPSS_x_CTRIP_y (*x* = 1 to 7) (*y* = LOW, HIGH, or HIGH_OR_LOW)

eCAP Driverlib Function

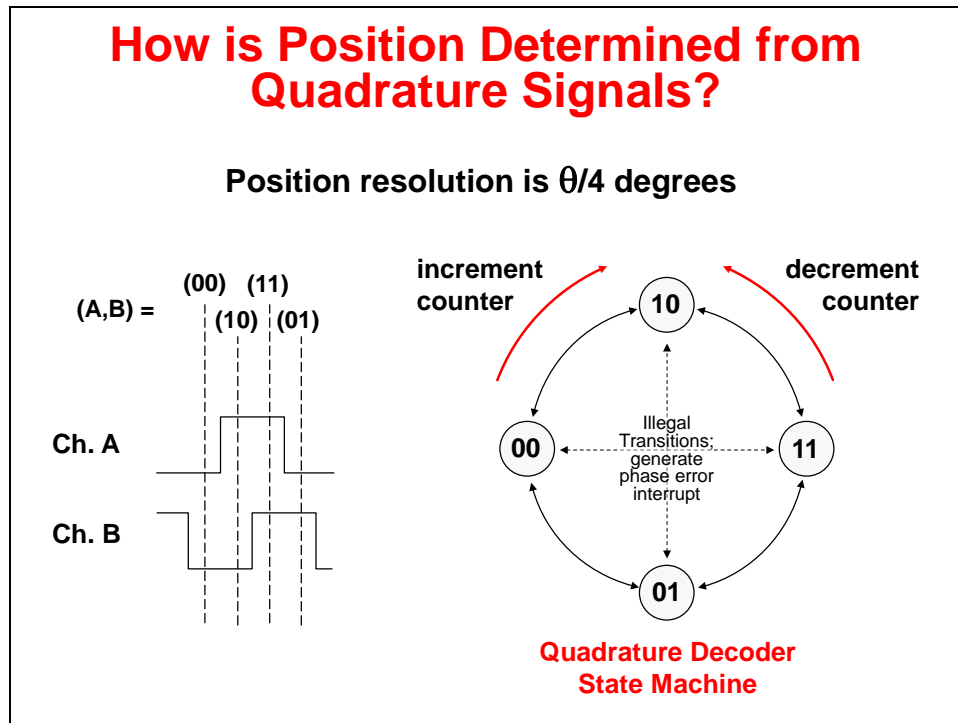
- ◆ **Enable / disable load of time stamp on capture event (CAP1 –CAP4)**
`ECAP_[enable|disable]TimeStampCapture(base);`
 - ◆ **Set capture mode (continuous/wrap or one-shot/stop mode)**
`ECAP_setCaptureMode(base, mode, event);`
 - ◆ **Sets the capture event polarity (rising or falling edge)**
`ECAP_setEventPolarity(base, event, polarity);`
 - ◆ **Enable / disable counter reset on event (delta / absolute time stamp)**
`ECAP_[enable|disable]CounterResetOnEvent(base, event);`
 - ◆ **Set and enable / disable capture event interrupt source**
`ECAP_[enable|disable]Interrupt(base, intFlags);`
 - ◆ **Set eCAP input event filter prescaler counter**
`ECAP_setEventPrescaler(base, preScalerValue);`
- ◆ *base* is the eCAP base address: ECAP_x_BASE (x = 1 to 7)
 - ◆ *mode* value is: ECAP_CONTINUOUS_CAPTURE_MODE or ECAP_ONE_SHOT_CAPTURE_MODE
 - ◆ *event* value is: ECAP_EVENT_x (x = 1 to 4)
 - ◆ *polarity* value is: ECAP_EVNT_RISING_EDGE or ECAP_EVNT_FALLING_EDGE
 - ◆ *intFlags* value is: ECAP_ISR_SOURCE_CAPTURE_x (x = EVENT_1, EVENT_2, EVENT_3, EVENT_4, OVERFLOW, PERIOD, COMPARE)
 - ◆ *preScalerValue* value is: 1, 2, ...31 (divide is 2x - i.e. 5 is /10; 1 = no prescale)

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

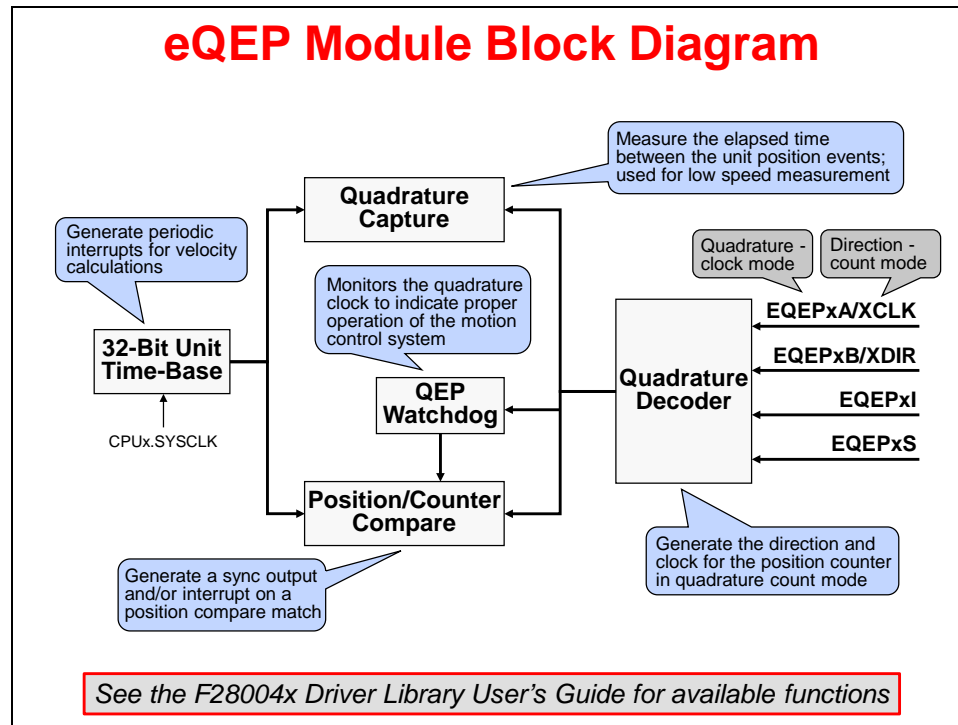
eQEP



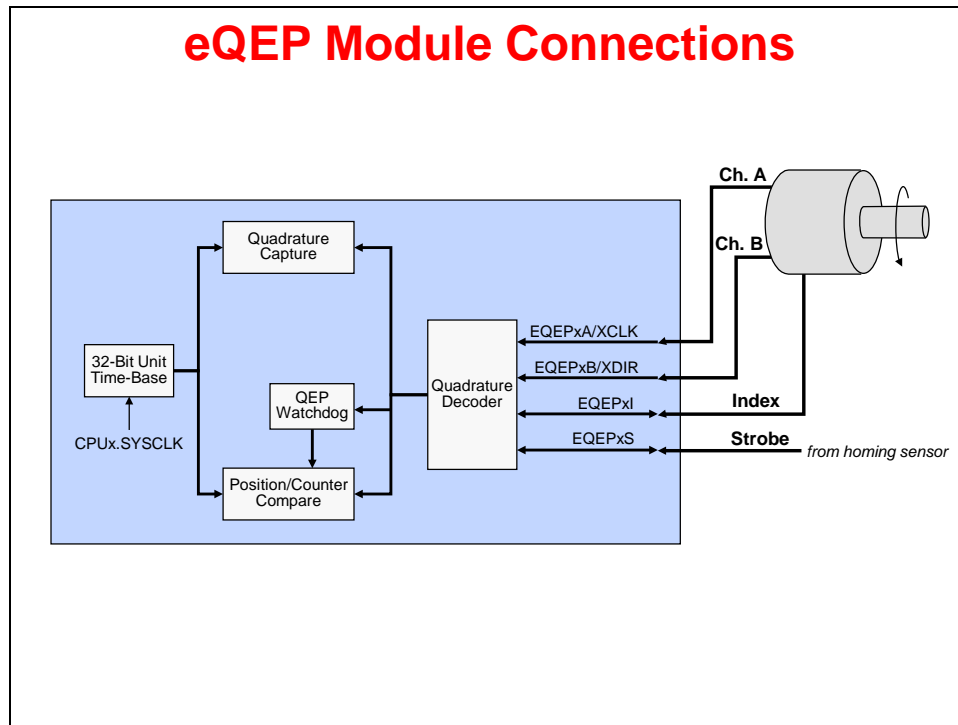
The eQEP module interfaces with a linear or rotary incremental encoder for determining position, direction, and speed information from a rotating machine that is typically found in high-performance motion and position-control systems.



A quadrature decoder state machine is used to determine position from two quadrature signals.

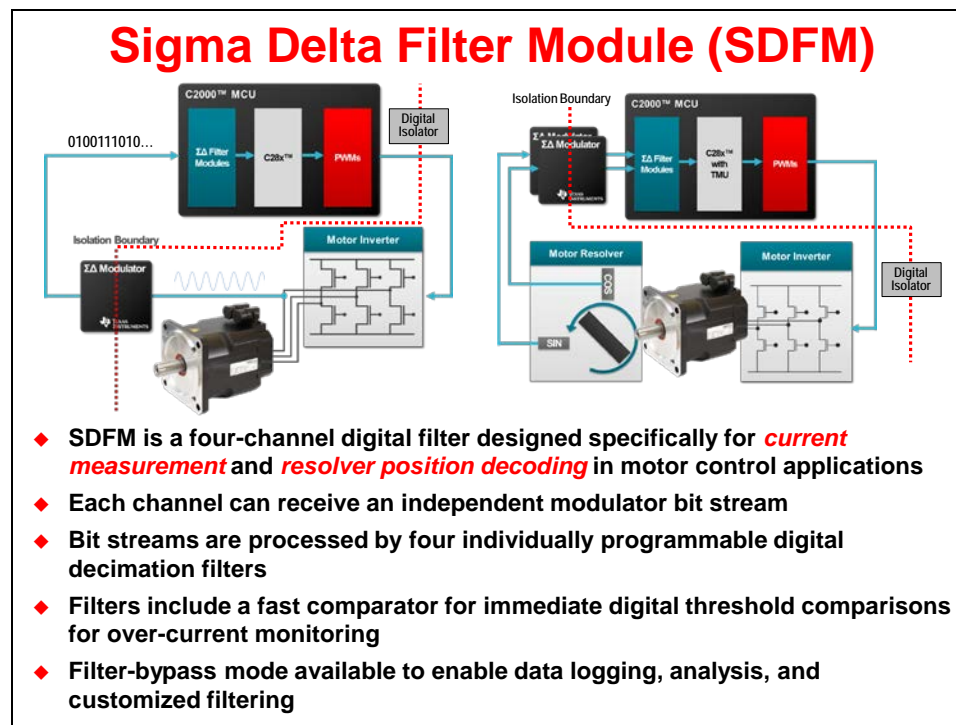


The inputs include two pins (QEPA and QEPB) for quadrature-clock mode or direction-count mode, an index pin (QEPI), and a strobe pin (QEPS). These pins are configured using the GPIO multiplexer and need to be enabled for synchronous input. In quadrature-clock mode, two square wave signals from a position encoder are inputs to QEPA and QEPB which are 90 electrical degrees out of phase. This phase relationship is used to determine the direction of rotation. If the position encoder provides direction and clock outputs, instead of quadrature outputs, then direction-count mode can be used. QEPA input will provide the clock signal and QEPB input will have the direction information. The QEPI index signal occurs once per revolution and can be used to indicate an absolute start position from which position information is incrementally encoded using quadrature pulses. The QEPS strobe signal can be connected to a sensor or limit switch to indicate that a defined position has been reached.

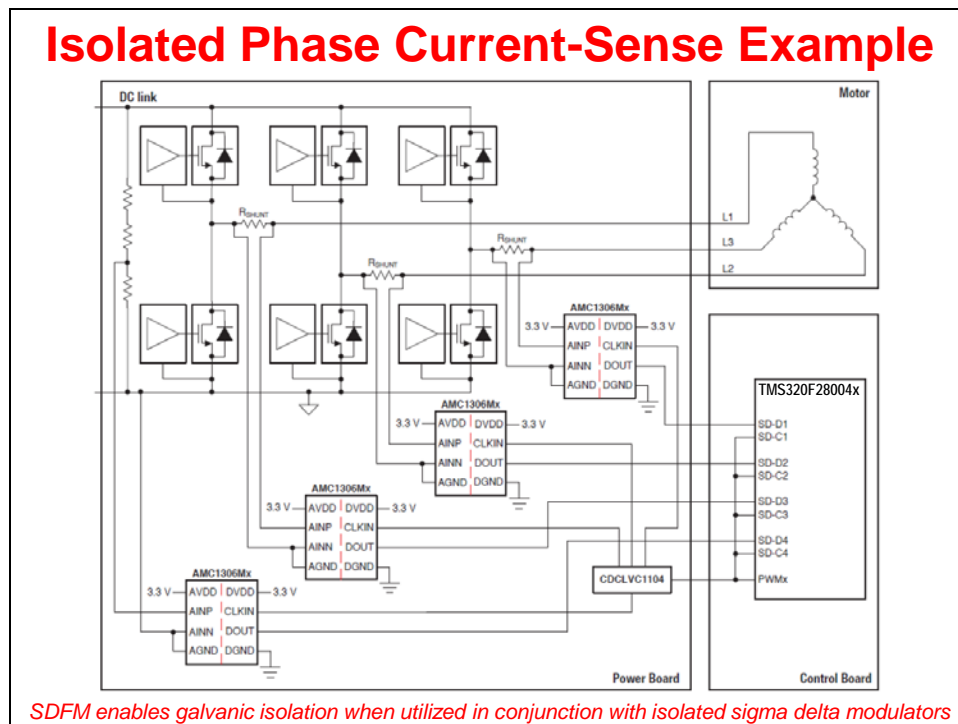
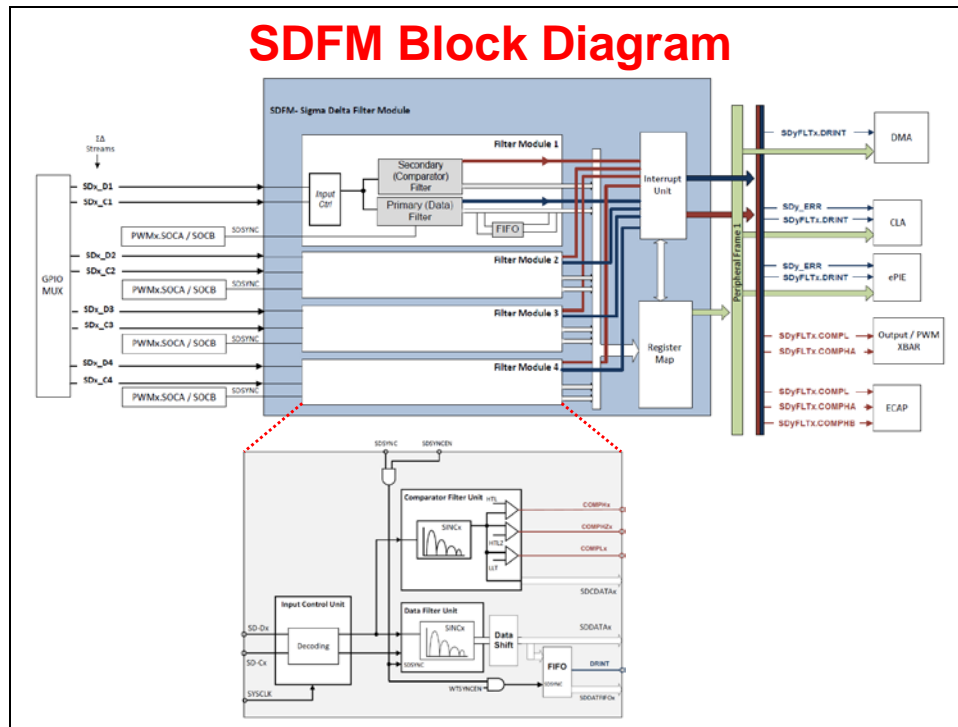


The above figure shows a summary of the connections to the eQEP module.

Sigma Delta Filter Module (SDFM)



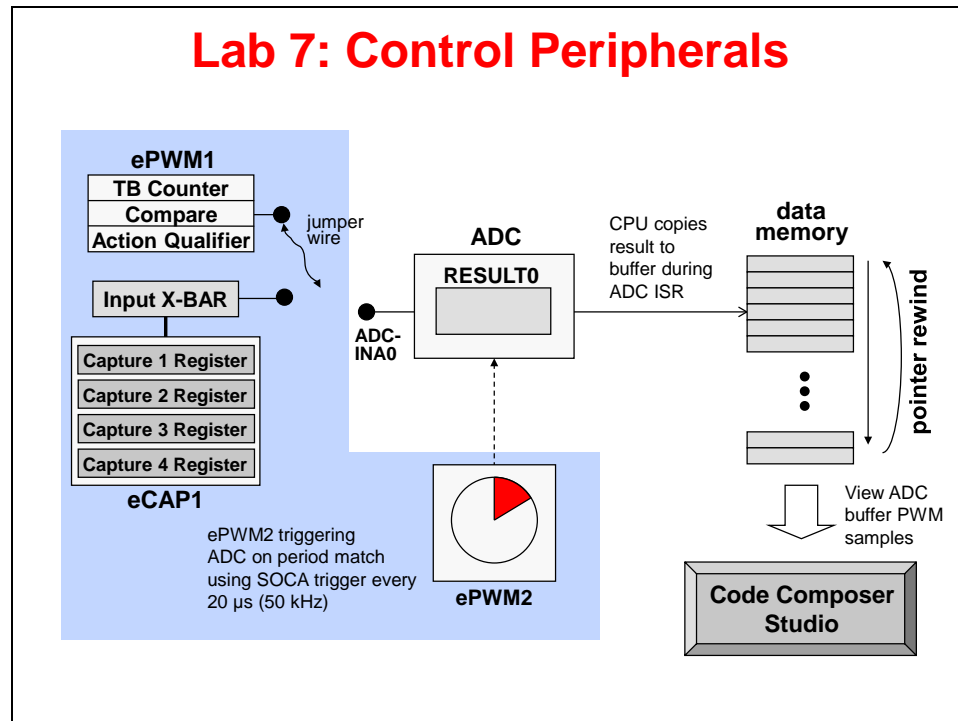
The SDFM is a four-channel digital filter designed specifically for current measurement and resolver position decoding in motor control applications. Each channel can receive an independent delta-sigma modulator bit stream which is processed by four individually programmable digital decimation filters. The filters include a fast comparator for immediate digital threshold comparisons for over-current and under-current monitoring. Also, a filter-bypass mode is available to enable data logging, analysis, and customized filtering. The SDFM pins are configured using the GPIO multiplexer. A key benefit of the SDFM is it enables a simple, cost-effective, and safe high-voltage isolation boundary.



Lab 7: Control Peripherals

➤ Objective

The objective of this lab exercise 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 symmetrical 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 exercise. Open the project by clicking on **Project** → **Import CCS Projects**. The "Import CCS Eclipse Projects" window will open. Click **Browse...** next to the "Select search-directory" box. Navigate to: `C:\F28004x\Labs\Lab7\project` and click **Select Folder**. Then click **Finish** to import the project. All build options have been configured the same as the previous lab exercise. The files used in this lab exercise are:

Adc.c	EPwm_7.c
CodeStartBranch.asm	Gpio.c
Dac.c	Lab_5_6_7.cmd
DefaultIsr_7.c	Main_7.c
device.c	SineTable.c
ECap_7.c	Watchdog.c

Note: The `ECap_7.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.

Generate PWM Waveform

- Open and inspect `Gpio.c`. Notice the `Driverlib` functions used to configure ePWM1A as an output on the GPIO pin.
- Edit `EPwm_7.c` to configure ePWM1A as described in the objective for this lab exercise (i.e. generate a 2 kHz, 25% duty cycle symmetrical PWM waveform):
 - Set the timebase period and counter compare values by using the `#define` global variable names in the beginning of `Lab.h`, which is located in the Project Explorer window in the includes folder under `/Lab_common/include`
 - Set the action qualifier to generate the specified waveform
 - Enable the timebase count mode to generate a symmetrical PWM waveform

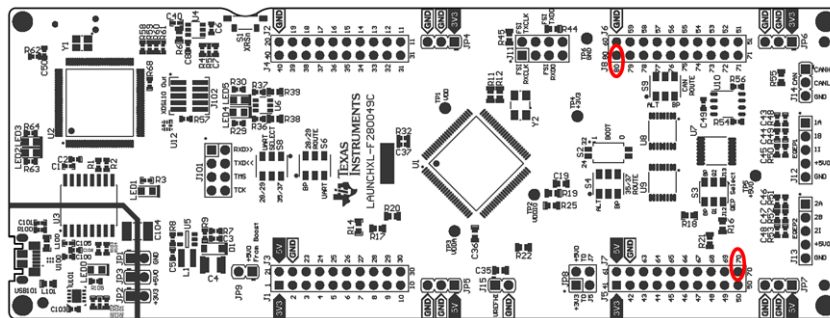
Note that the deadband, PWM chopper, and all trip zone and DC compare actions have been disabled. Save your work.

Build and Load

- Click the “Build” button and watch the tools run in the `Console` window. Check for errors in the Problems window.
- Click the “Debug” button (green bug). The 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_RAM` using the Scripts menu.

Run the Code – PWM Waveform

- Using a jumper wire, connect the PWM1A (pin #80) to ADCINA0 (pin #70) on the LaunchPad. Refer to the following diagram for the pins that need to be connected.



- 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.

8. Run the code (real-time mode). 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	μ s

Select OK to save the graph options.

10. The graphical display should show the generated 2 kHz, 25% duty cycle symmetric PWM waveform. The period of a 2 kHz signal is 500 μ s. You can confirm this by measuring the period of the waveform using the “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. Halt the code.

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. Add (copy) `ECap_7.c` to the project from `C:\F28004x\Labs\Lab7\source`.

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. In `Gpio.c` and notice the Driverlib functions for configuring GPIO24 as the input. Next, notice the Driverlib function setting GPIO24 as the signal source for Input X-BAR INPUT7. The GPIO24 pin via INPUT7 will be routed as the input to eCAP1.

17. Open `DefaultIsr_7.c` and locate the eCAP1 interrupt service routine (`ecap1ISR`). 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. Open and edit `ECap_7.c` to:

- Set the event polarity to capturing the rising and falling edges of the PWM waveform in order to calculate the PWM duty and PWM period
- Enable eCAP interrupt after three (3) capture events

Also, notice the Driverlib function that is used to select Input X-BAR INPUT7 as the source for eCAP1. In `Gpio.c` the GPIO24 pin has been configured as the source for INPUT7.

19. Using the "PIE Interrupt Assignment Table" find the location for the eCAP1 interrupt "`INT_ECAP1`" 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.c` to do the following:

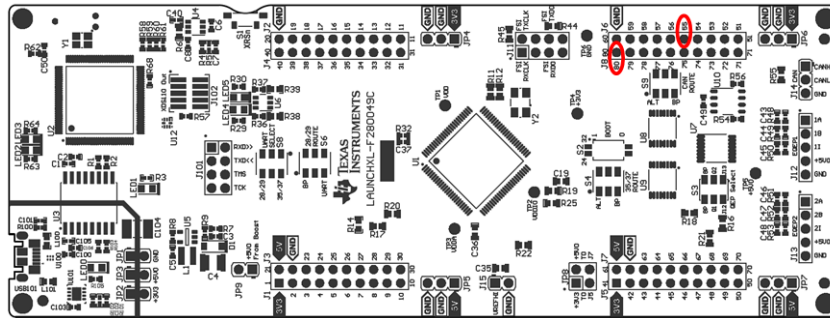
- Add the Driverlib function to re-map the ECAP1 interrupt signal to call the ISR function. (Hint: `#define` name in `driverlib/inc/hw_ints.h` and label name in `DefaultIsr_7.c`)
- Add the Driverlib function to enable the appropriate PIEIER and core IER

Build and Load

21. Save all changes to the files and build the project by clicking `Project` → `Build Project`, or by clicking on the “Build” button if you have added it to the tool bar. Select `Yes` to “Reload the program automatically”.

Run the Code – Pulse Width Measurement

22. Using a jumper wire, connect the ePWM1A (pin #80) to eCAP1 (pin #55, feed from the Input X-BAR using GPIO24) on the LaunchPad. Refer to the following diagram for the pins that need to be connected.



23. 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. Scroll the window up, if needed.

24. Set the memory browser properties format to “32-Bit UnSigned Int”. We will be running our code in real-time mode, and we will need to have the memory browser continuously refresh.

25. Run the code (real-time mode). Notice the values for `PwmDuty` and `PwmPeriod`.

26. Halt the code.

Questions:

- How do the captured values for `PwmDuty` and `PwmPeriod` relate to the compare register and time-base period 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?

Internal Pulse Width Measurement Using Input X-BAR

27. Modify `Gpio.c` to use GPIO0 as the signal source to Input X-BAR INPUT7 rather than GPIO24. (Hint: you only need to modify the Input X-BAR Driverlib function). Recall that in `Gpio.c` ePWM1A has been configured as an output on the GPIO0 pin. This modification will internally route the PWM output as the input to the eCAP1. Therefore, remove the jumper wire since it is not needed.
28. Save all changes and build the project. Select `Yes` to “Reload the program automatically”.
29. Run the code (real-time mode) and verify that the results are the same.
30. Halt the code.

Modulate the PWM Waveform

Next, we will 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 #80) to ADCINA0 (pin #70), and the Single Time graph is displayed. The graph must be enabled for continuous refresh.

31. Run the code (real-time mode).
32. Open the Registers window by clicking: `View` → `Registers`
33. In the Registers window scroll down and expand “EPwm1Regs”. Then scroll down and expand “CMPA”. In the Value field for “CMPA” right-click and set the Number Format to Decimal. The Registers window must be enabled for continuous refresh.
34. Change the “CMPA” 18750 value (within a range of 2500 and 22500). Notice the effect on the PWM waveform in the graph. Also, notice the value for `PwmDuty` changes in the Memory Browser window.

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.

35. In `DefaultIsr_7.c` notice the code in the ADCA1 interrupt service routine used to modulate the PWM1A output between 10% and 90% duty cycle
36. In `Main_7.c` add “PWM_MODULATE” to the Expressions window. Simply highlight `PWM_MODULATE` with the mouse, right click and select “Add Watch Expression...” and then select `OK`. The global variable `PWM_MODULATE` should now be in the Expressions window with a value of “0”.
37. With the code still running in real-time mode, change the “PWM_MODULATE” from “0” to “1” and observe the PWM waveform in the graph. The value for `PwmDuty` will update continuously in the Memory Browser window. Also, in the Registers window notice the CMPA value being continuously updated.
38. Halt the code.

Terminate Debug Session and Close Project

39. Terminate the active debug session using the `Terminate` button. This will close the debugger and return Code Composer Studio to the CCS Edit perspective view.

40. Next, close the project by right-clicking on `Lab7` in the Project Explorer window and select `Close Project`.

End of Exercise

Direct Memory Access

Introduction

This module explains the operation of the direct memory access (DMA) controller. The DMA has six channels with independent PIE interrupts.

Module Objectives

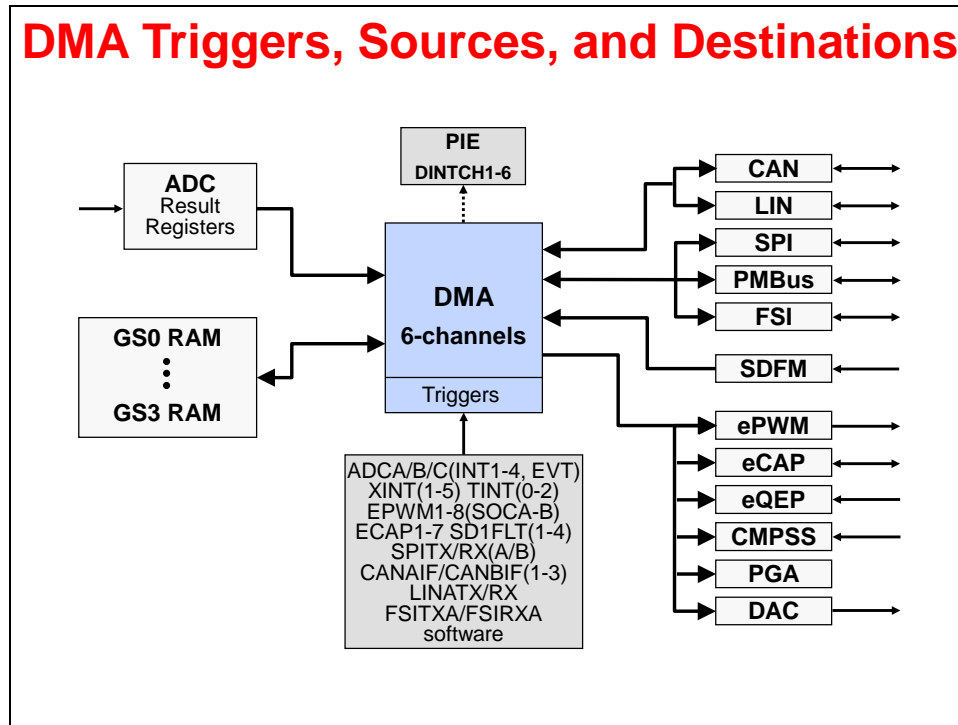
Module Objectives

- ◆ 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*

Chapter Topics

Direct Memory Access.....	8-1
<i>Direct Memory Access (DMA).....</i>	<i>8-3</i>
Basic Operation.....	8-4
DMA Examples	8-6
Channel Priority Modes.....	8-9
DMA Throughput.....	8-10
DMA Driverlib Functions	8-11
<i>Lab 8: Servicing the ADC with DMA</i>	<i>8-13</i>

Direct Memory Access (DMA)



The DMA module provides a hardware method of transferring data between peripherals and/or memory without intervention from the CPU, effectively freeing up the CPU for other functions. Using the DMA is ideal when an application requires moving large amounts of data from an off-chip peripheral to on-chip memory, or from a peripheral such as the ADC result register to a memory RAM block, or between two peripherals. Additionally, the DMA is capable of rearranging the data for optimal CPU processing such as binning and “ping-pong” buffering.

Specifically, the DMA can read data from the ADC result registers, transfer to or from memory blocks G0 through G3, transfer to or from the various peripherals, and also modify registers in the ePWM. A DMA transfer is started by a peripheral or software trigger. There are six independent DMA channels, where each channel can be configured individually and each DMA channel has its own unique PIE interrupt for CPU servicing. All six DMA channels operate the same way, except channel 1 can be configured at a higher priority over the other five channels. At its most basic level the DMA is a state machine consisting of two nested loops and tightly coupled address control logic which gives the DMA the capability to rearrange the blocks of data during the transfer for post processing. When a DMA transfers is completed, the DMA can generate an interrupt.

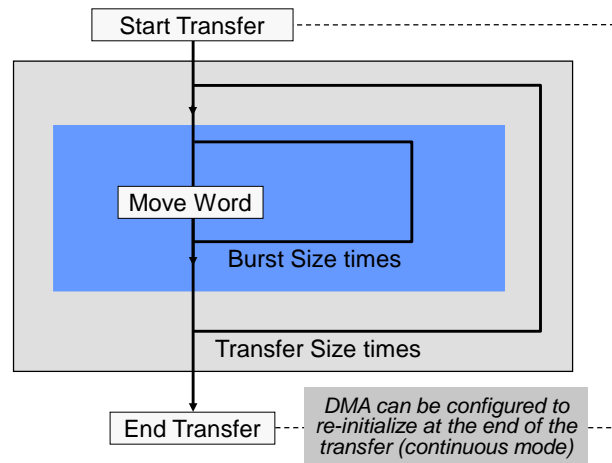
Basic Operation

DMA Definitions

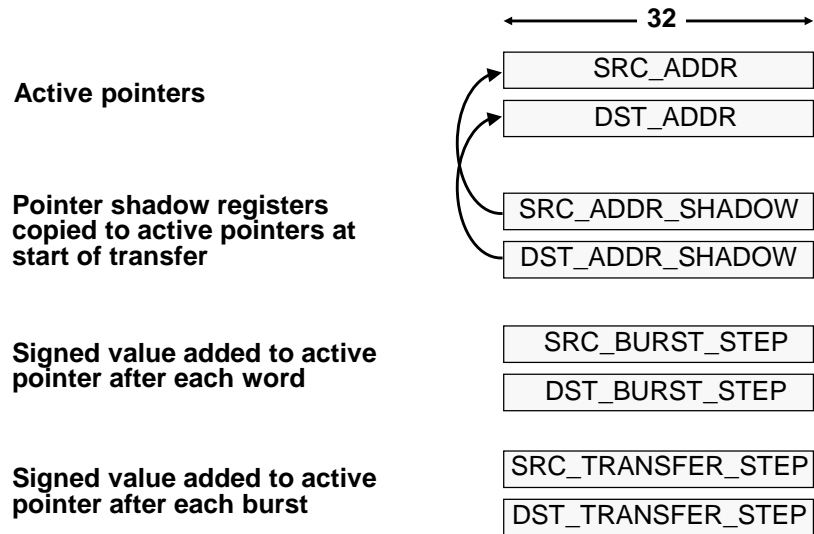
- ◆ **Word**
 - ◆ 16 or 32 bits
 - ◆ Word size is configurable per DMA channel
- ◆ **Burst**
 - ◆ Consists of multiple words
 - ◆ Smallest amount of data transferred at one time
- ◆ **Burst Size**
 - ◆ Number of words per burst
 - ◆ Specified by BURST_SIZE register
 - ◆ 5-bit 'N-1' value (maximum of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

Simplified State Machine Operation

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

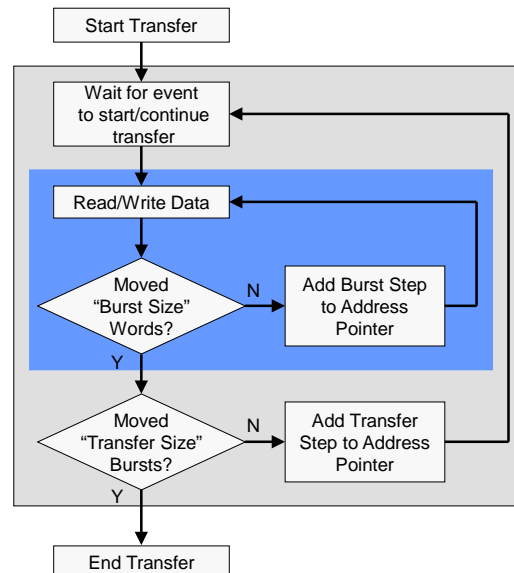


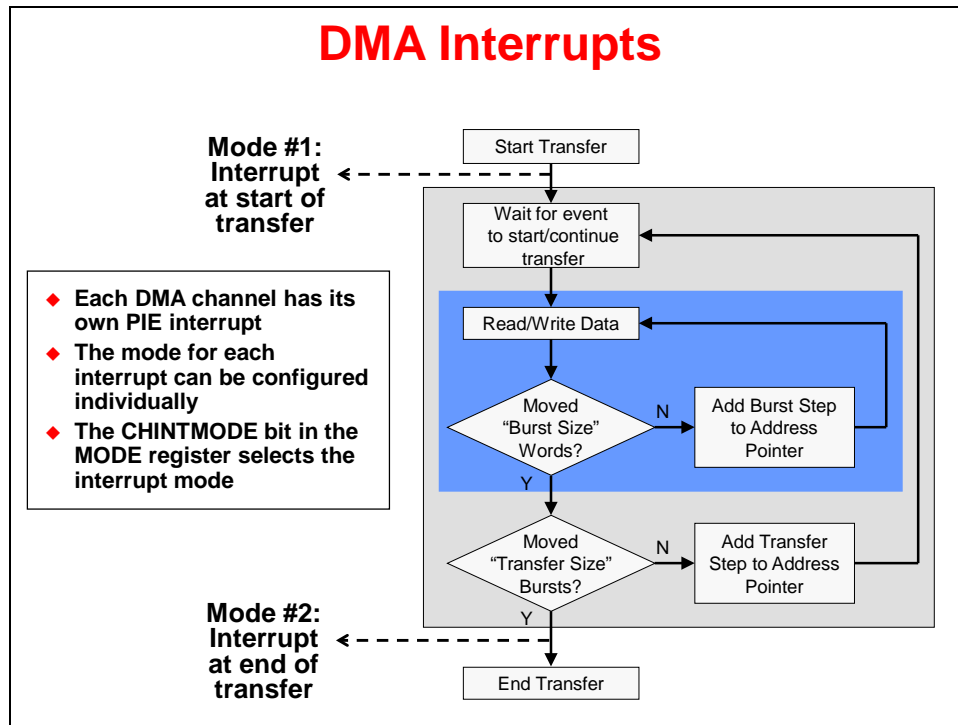
Basic Address Control Registers



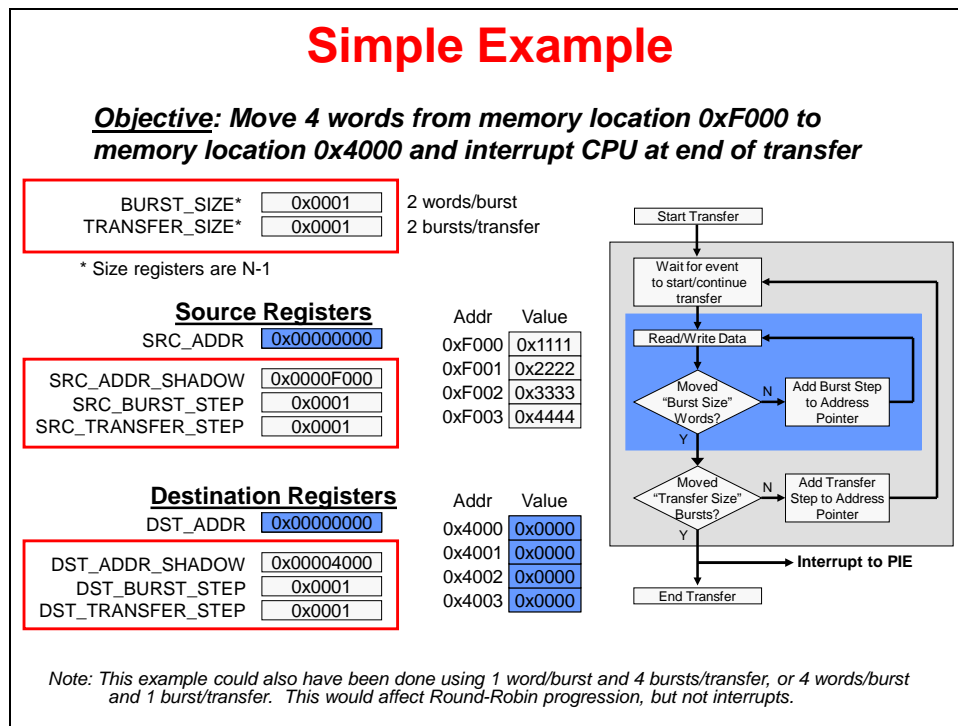
Simplified State Machine Example

3 words/burst
2 bursts/transfer



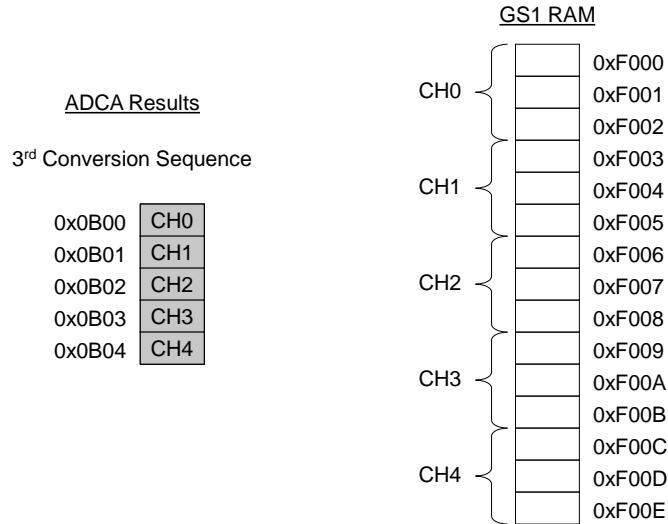


DMA Examples



Data Binning Example

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



Data Binning Example Register Setup

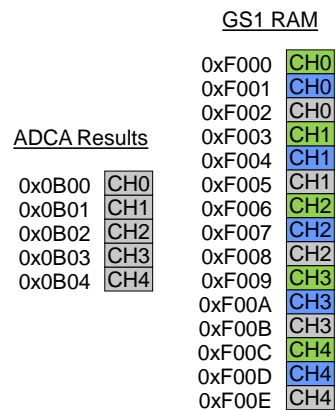
Objective: Bin 3 samples of 5 ADC channels, then interrupt the CPU

ADC Registers:

SOC0 – SOC4 configured to CH0 – CH4, respectively,
ADCA 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	0xFFFFC	(-4)
DST_ADDR_SHADOW	0x0000F000	starting address**
DST_BURST_STEP	0x0003	
DST_TRANSFER_STEP	0xFFFF5	(-11)



* Size registers are N-1

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

Ping-Pong Buffer Example

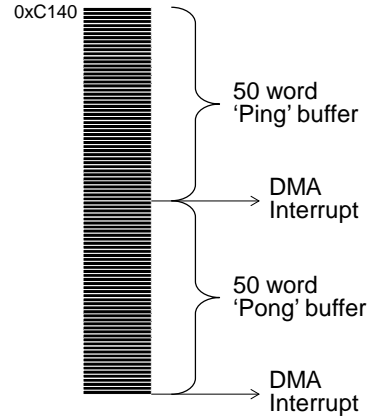
Objective: Buffer ADC ch. 0 ping-pong style, 50 samples per buffer

ADCA Result Register

0x0B00 **ADCRESULT0**

SOC0 configured to ADCINA0
with 1 conversion per trigger

GS0 RAM



Ping-Pong Example Register Setup

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

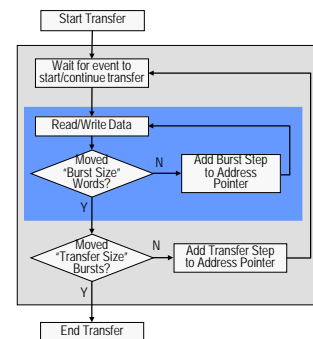
ADC Registers:

Convert ADCA 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	0x0000B00	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.

Channel 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

```

    graph TD
      Start[Start Transfer] --> Wait[Wait for event to start/continue transfer]
      Wait --> Read[Read/Write Data]
      Read --> Burst{Moved "Burst Size" Words?}
      Burst -- N --> AddBurst[Add Burst Step to Address Pointer]
      AddBurst --> Read
      Burst -- Y --> Transfer{Moved "Transfer Size" Bursts?}
      Transfer -- N --> AddTransfer[Add Transfer Step to Address Pointer]
      AddTransfer --> Read
      Transfer -- Y --> End[End Transfer]
  
```

DMA Throughput

DMA Throughput

- ◆ 4 cycles/word
- ◆ 1 cycle delay to start each burst
- ◆ 1 cycle delay returning from CH1 high priority interrupt
- ◆ 32-bit transfer doubles throughput

Example: 128 16-bit words from ADC to RAM

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

Example: 64 32-bit words from ADC to RAM

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

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 Driverlib Functions

DMA Driverlib Functions

- ◆ Initialize the DMA controller (*hard reset*)
`DMA_initController();`
- ◆ Set DMA channel priority mode (*round-robin or CH1 high priority*)
`DMA_setPriorityMode(ch1IsHighPri);`
- ◆ Configures the DMA channel trigger and mode
`DMA_configMode(base, trigger, config);`
- ◆ Enable /disable peripheral trigger for DMA transfer
`DMA_[enable|disable]Trigger(base);`
- ◆ Start / stop DMA channel (*'start' – wait for first trigger event*)
`DMA_[start|stop]Channel(base);`

- ◆ `ch1IsHighPri` value is 'false' for round-robin or 'true' for CH1 high priority
- ◆ `base` is the DMA channel base address: `DMA_CHx_BASE` ($x = 1$ to 6)
- ◆ `trigger` value is located in `dma.h` – *see table on next slide for values*
- ◆ `config` value is the logical OR of:
 - ◆ `DMA_CFG_ONESHOT_x` ($x = \text{DISABLE or ENABLE}$)
 - ◆ `DMA_CFG_CONTINUOUS_x` ($x = \text{DISABLE or ENABLE}$)
 - ◆ `DMA_CFG_SIZE_xBIT` ($x = 16$ or 32)

Peripheral Interrupt Trigger Sources

`DMA_configMode(base, trigger, config);`

DMA_TRIGGER_SOFTWARE	DMA_TRIGGER_XINT2	DMA_TRIGGER_EPWM7SOCB	DMA_TRIGGER_SPIATX
DMA_TRIGGER_ADCA1	DMA_TRIGGER_XINT3	DMA_TRIGGER_EPWM8SOCA	DMA_TRIGGER_SPIARX
DMA_TRIGGER_ADCA2	DMA_TRIGGER_XINT4	DMA_TRIGGER_EPWM8SOCB	DMA_TRIGGER_SPIBTX
DMA_TRIGGER_ADCA3	DMA_TRIGGER_XINT5	DMA_TRIGGER_TINT0	DMA_TRIGGER_SPIBRX
DMA_TRIGGER_ADCA4	DMA_TRIGGER_EPWM1SOCA	DMA_TRIGGER_TINT1	DMA_TRIGGER_LINATX
DMA_TRIGGER_ADCAEVT	DMA_TRIGGER_EPWM1SOCB	DMA_TRIGGER_TINT2	DMA_TRIGGER_LINARX
DMA_TRIGGER_ADCB1	DMA_TRIGGER_EPWM2SOCA	DMA_TRIGGER_ECAP1	DMA_TRIGGER_FSITXA
DMA_TRIGGER_ADCB2	DMA_TRIGGER_EPWM2SOCB	DMA_TRIGGER_ECAP2	DMA_TRIGGER_FSIRXA
DMA_TRIGGER_ADCB3	DMA_TRIGGER_EPWM3SOCA	DMA_TRIGGER_ECAP3	DMA_TRIGGER_CANAIF1
DMA_TRIGGER_ADCB4	DMA_TRIGGER_EPWM3SOCB	DMA_TRIGGER_ECAP4	DMA_TRIGGER_CANAIF2
DMA_TRIGGER_ADCBEVT	DMA_TRIGGER_EPWM4SOCA	DMA_TRIGGER_ECAP5	DMA_TRIGGER_CANAIF3
DMA_TRIGGER_ADCC1	DMA_TRIGGER_EPWM4SOCB	DMA_TRIGGER_ECAP6	DMA_TRIGGER_CANBIF1
DMA_TRIGGER_ADCC2	DMA_TRIGGER_EPWM5SOCA	DMA_TRIGGER_ECAP7	DMA_TRIGGER_CANBIF2
DMA_TRIGGER_ADCC3	DMA_TRIGGER_EPWM5SOCB	DMA_TRIGGER_SDFM1FLT1	DMA_TRIGGER_CANBIF3
DMA_TRIGGER_ADCC4	DMA_TRIGGER_EPWM6SOCA	DMA_TRIGGER_SDFM1FLT2	
DMA_TRIGGER_ADCCEVT	DMA_TRIGGER_EPWM6SOCB	DMA_TRIGGER_SDFM1FLT3	
DMA_TRIGGER_XINT1	DMA_TRIGGER_EPWM7SOCA	DMA_TRIGGER_SDFM1FLT4	

DMA Driverlib Functions

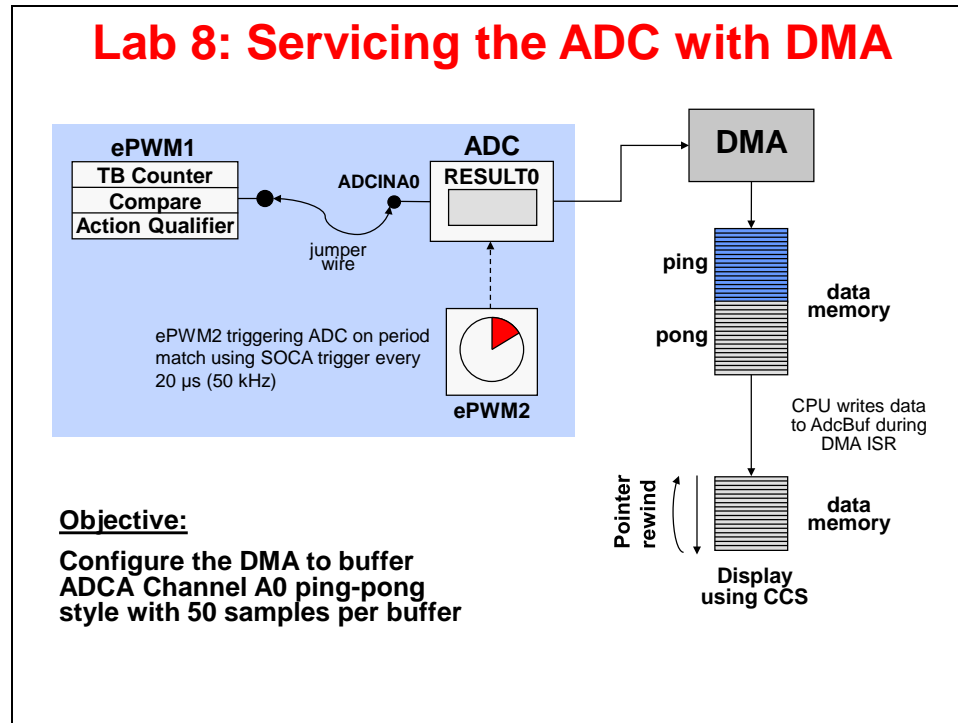
- ◆ **Configure the source and destination addresses of the DMA channel**
`DMA_configAddresses(base, const void * destAddr, const void * srcAddr);`
- ◆ **Configures the burst size and the address step size**
`DMA_configBurst (base, size, srcStep, destStep);`
- ◆ **Configures the transfer size and the address step size**
`DMA_configTransfer (base, transferSize, srcStep, destStep);`
- ◆ **Set the channel interrupt mode**
`DMA_setInterruptMode(base, mode);`
- ◆ **Enable / disable DMA channel CPU interrupt**
`DMA_[enable|disable]Interrupt (base);`

- ◆ *base* is the DMA channel base address: DMA_CHx_BASE (x = 1 to 6)
- ◆ *destAddr* is a pointer to the destination address
- ◆ *srcAddr* is a pointer to the source address
- ◆ *size* value is the number of words per burst (range from 1 to 32 words)
- ◆ *srcStep* and *destStep* value is the step size (signed values from -4096 to 4095)
- ◆ *transferSize* value is the number of bursts per transfer (max value of 65536)
- ◆ *mode* value is: DMA_INT_AT_BEGINNING or DMA_INT_AT_END

Lab 8: Servicing the ADC with DMA

➤ Objective

The objective of this lab exercise is to become familiar with operation of the DMA. In the previous lab exercise, the CPU was used to store the ADC conversion result in the memory buffer during the ADC ISR. In this lab exercise 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 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 Lab8 has been created for this lab exercise. Open the project by clicking on `Project` → `Import CCS Projects`. The "Import CCS Eclipse Projects" window will open. Click `Browse...` next to the "Select search-directory" box. Navigate to: `C:\F28004x\Labs\Lab8\project` and click `Select Folder`. Then click `Finish` to import the project. All build options have been configured the same as the previous lab exercise. The files used in this lab exercise are:

Adc.c	EPwm.c
CodeStartBranch.asm	Gpio.c
Dac.c	Lab_8.cmd
DefaultIsr_8.c	Main_8.c
device.c	SineTable.c
Dma_8.c	Watchdog.c
ECap.c	

Inspect Lab_8.cmd

2. Open and inspect `Lab_8.cmd`. Notice that a section called “`dmaMemBufs`” is being linked to `RAMGS2`. This section links the destination buffer for the DMA transfer to a DMA accessible memory space. Close the inspected file.

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_8.c` to implement the DMA operation as described in the objective for this lab exercise:
 - Enable the peripheral interrupt trigger for channel 1 DMA transfer
 - Generate an interrupt at the beginning of a new transfer
 - Enable the DMA channel CPU interrupt

Note: the DMA has been configured for an ADC interrupt “ADCA1” to trigger the start of a DMA CH1 transfer. Additionally, the DMA is set for 16-bit data transfers with one burst per trigger and auto re-initialization at the end of the transfer. At the end of the code the channel is enabled to run.

4. Open `Main_8.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 read the ADC result register 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 read the ADC result register in the DMA ISR.

5. Edit `Adc.c` to *comment out* the code used to enable the ADCA1 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 “`INT_DMA_CH1`” 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_8.c` to do the following:

- Add the Driverlib function to re-map the DMA_CH1 interrupt signal to call the ISR function. (Hint: #define name in `driverlib/inc/hw_ints.h` and label name in `DefaultIsr_8.c`)
 - Add the Driverlib function to enable the appropriate PIEIER and core IER
8. Inspect `DefaultIsr_8.c` and notice that this file contains the DMA interrupt service routine which implements the ping-pong style buffer. Save 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_RAM` 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 #80) to ADCINA0 (pin #70) is in place on the LaunchPad.

11. Run the code (real-time mode). Open and watch the memory browser update. Verify that the ADC result buffer contains updated values.
12. 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.

13. The graphical display should show the generated 2 kHz, 25% duty cycle symmetric PWM waveform. Notice that the results match the previous lab exercise.
14. Halt the code.

Terminate Debug Session and Close Project

15. Terminate the active debug session using the `Terminate` button. This will close the debugger and return Code Composer Studio to the CCS Edit perspective view.

16. Next, close the project by right-clicking on Lab8 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. It executes algorithms independent of the CPU. This extends the capabilities of the C28x CPU by adding parallel processing. The CLA has direct access to the ADC result registers. Additionally, the CLA has access to all ePWM, high-resolution PWM, eCAP, eQEP, CMPSS, DAC, SDFM, PGA, SPI, LIN, FSI, PMBUS, CLB and GPIO data 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. The CLA responds to peripheral interrupts independently of the CPU. Utilizing the CLA for time-critical tasks frees up the CPU to perform other system, diagnostics, and communication functions concurrently. Additionally, the CLA has the capability of running a background task.

Module Objectives

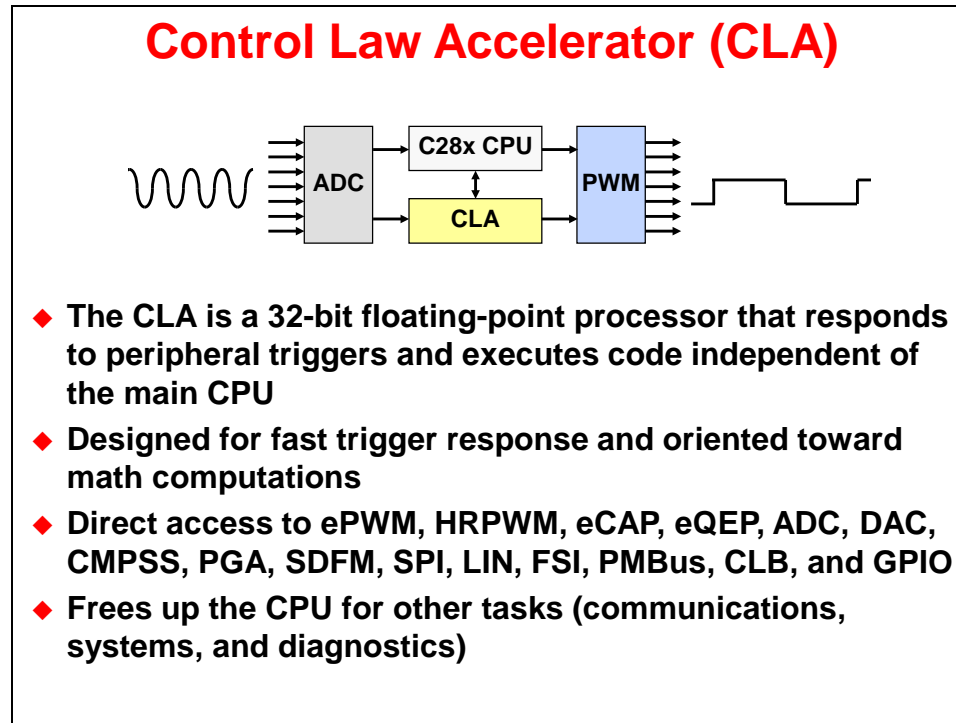
Module Objectives

- ◆ Explain the purpose and operation of the Control Law Accelerator (CLA)
- ◆ Describe the CLA initialization procedure
- ◆ Discuss the CLA registers, Driverlib functions, and programming flow

Chapter Topics

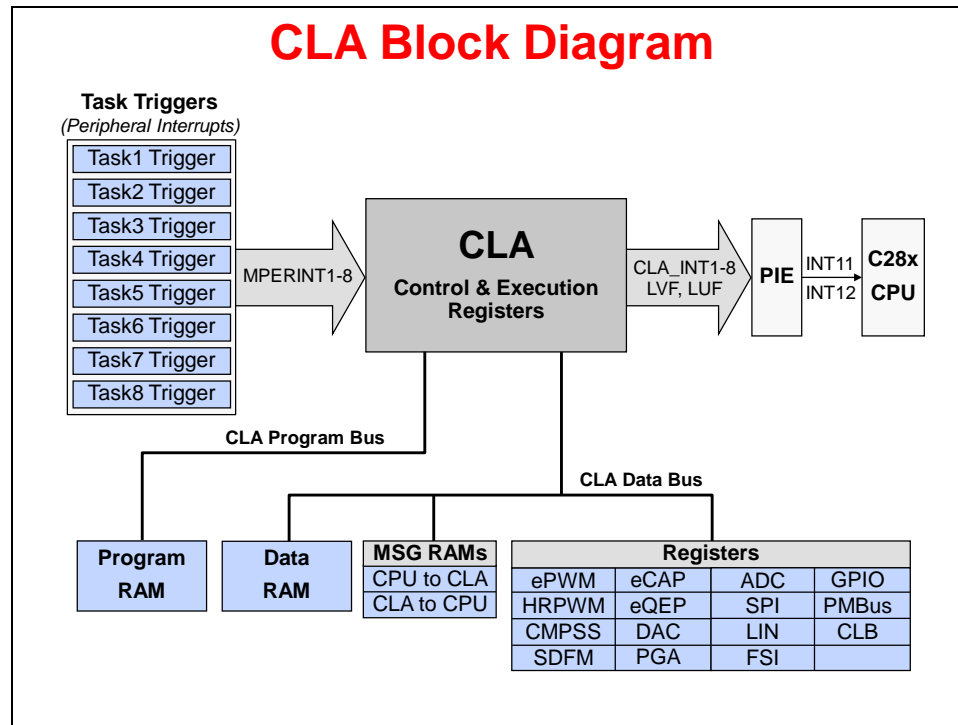
Control Law Accelerator.....	9-1
<i>Control Law Accelerator (CLA)</i>	9-3
CLA Block Diagram	9-4
CLA Tasks.....	9-4
CLA Memory and Register Access	9-5
CLA Control and Execution Registers	9-6
Task Trigger	9-6
Software Trigger.....	9-7
Background Task	9-8
Memory Configuration.....	9-9
Task Vector	9-10
CLA Initialization	9-10
Enabling CLA Support in CCS	9-11
CLA Task C Programming	9-11
C2000Ware – CLA Software Support	9-13
CLA Compiler Scratchpad Memory Area.....	9-13
CLA Initialization Code Example.....	9-14
CLA Task C Code Example	9-14
CLA Code Debugging	9-15
<i>Lab 9: CLA Floating-Point FIR Filter</i>	9-16

Control Law Accelerator (CLA)

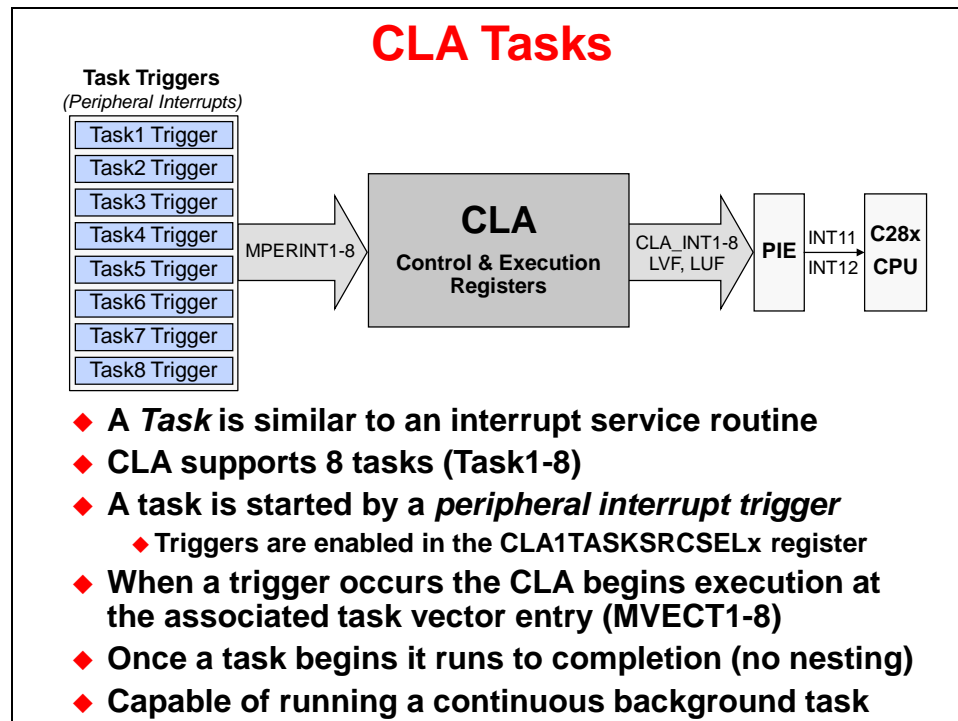


The CLA is an independent 32-bit floating-point math hardware accelerator which executes real-time control algorithms in parallel with the main C28x CPU, effectively doubling the computational performance. The CLA responds directly to peripheral triggers, which can free up the C28x CPU for other tasks, such as communications and diagnostics. With direct access to the various control and communication peripherals, the CLA minimizes latency, enables a fast trigger response, and avoids CPU overhead. Also, with direct access to the ADC results registers, the CLA is able to read the result on the same cycle that the ADC sample conversion is completed, providing “just-in-time” reading, which reduces the sample to output delay.

CLA Block Diagram



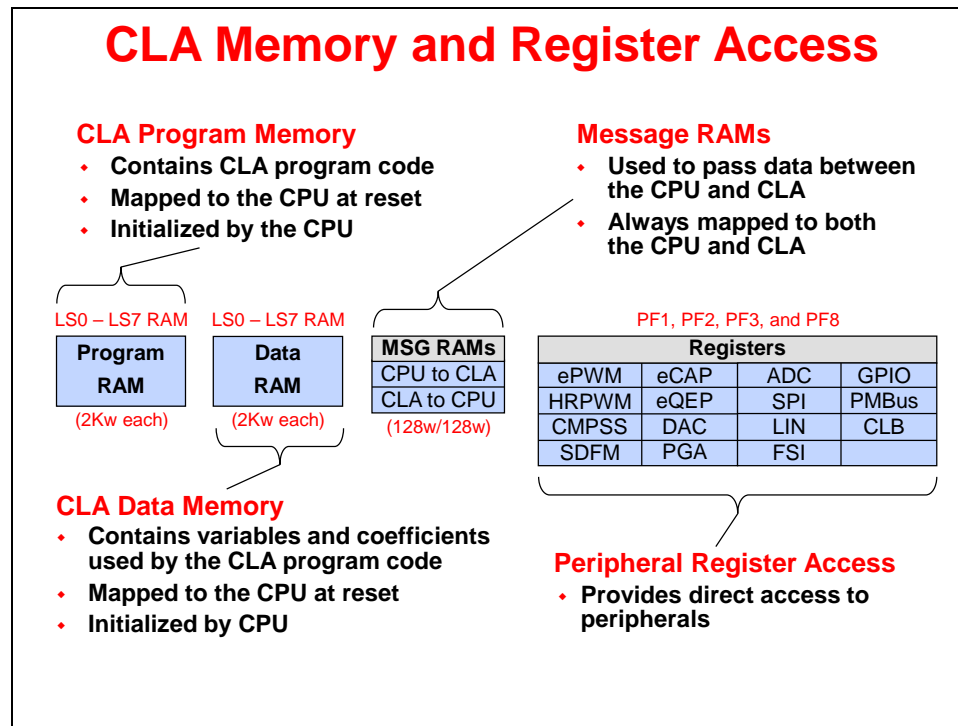
CLA Tasks



Programming the CLA consists of initialization code, which is performed by the CPU, and tasks. A task is similar to an interrupt service routine, and once started it runs to completion. Each task

is capable of being triggered by a variety of peripherals without CPU intervention, which makes the CLA very efficient since it does not use interrupts for hardware synchronization, nor must the CLA do any context switching. Unlike the traditional interrupt-based scheme, the CLA approach becomes deterministic. The CLA supports eight independent tasks and each is mapped back to an event trigger. Also, the CLA is capable of running a continuous background task. Since the CLA is a software programmable accelerator, it is very flexible and can be modified for different applications.

CLA Memory and Register Access



CLA Data Memory

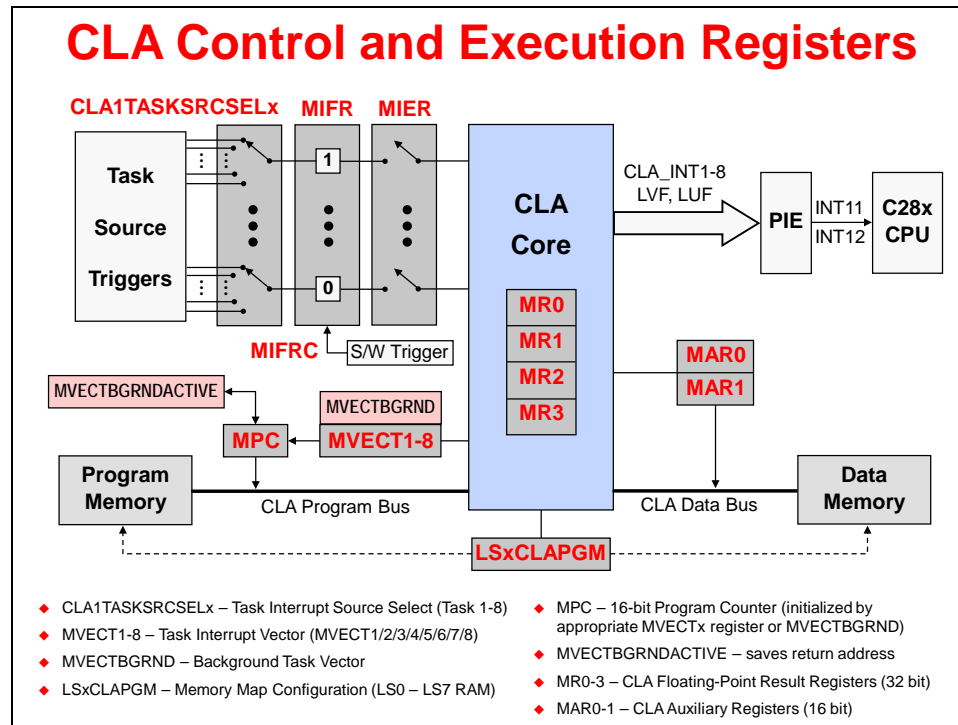
- Contains variables and coefficients used by the CLA program code
- Mapped to the CPU at reset
- Initialized by CPU

Peripheral Register Access

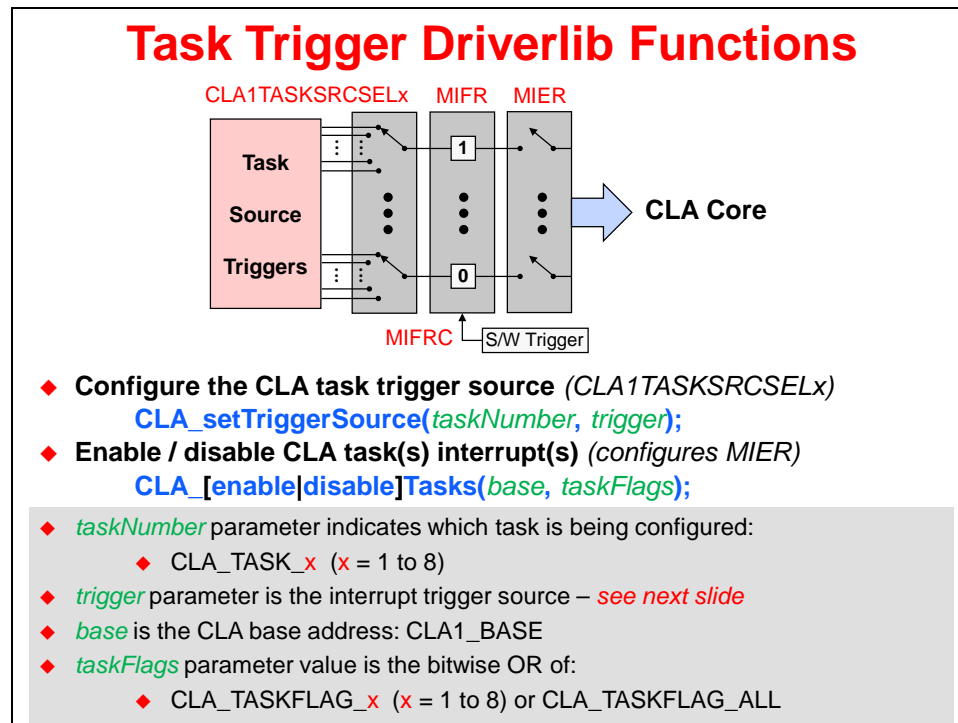
- Provides direct access to peripherals

The CLA has access to the LSx RAM blocks and each memory block can be configured to be either dedicated to the CPU or shared between the CPU and CLA. After reset the memory block is mapped to the CPU, where it can be initialized by the CPU before being shared with the CLA. Once it is shared between the CPU and CLA it then can be configured to be either program memory or data memory. When configured as program memory it contains the CLA program code, and when configured as data memory it contains the variable and coefficients that are used by the CLA program code. Additionally, dedicated message RAMs are used to pass data between the CPU and CLA, and CLA and CPU.

CLA Control and Execution Registers



Task Trigger



Task Interrupt Trigger Sources

`CLA_setTriggerSource(taskNumber, trigger);`

CLA_TRIGGER_SOFTWARE	CLA_TRIGGER_ADCAEVT	CLA_TRIGGER_TINT1	CLA_TRIGGER_SDFM1DRINT2
CLA_TRIGGER_ADCA1	CLA_TRIGGER_XINT1	CLA_TRIGGER_TINT2	CLA_TRIGGER_SDFM1DRINT3
CLA_TRIGGER_ADCA2	CLA_TRIGGER_XINT2	CLA_TRIGGER_ECAP1INT	CLA_TRIGGER_SDFM1DRINT4
CLA_TRIGGER_ADCA3	CLA_TRIGGER_XINT3	CLA_TRIGGER_ECAP2INT	CLA_TRIGGER_PMBUSAINT
CLA_TRIGGER_ADCA4	CLA_TRIGGER_XINT4	CLA_TRIGGER_ECAP3INT	CLA_TRIGGER_SPITXAINT
CLA_TRIGGER_ADCAEVT	CLA_TRIGGER_XINT5	CLA_TRIGGER_ECAP4INT	CLA_TRIGGER_SPIRXAINT
CLA_TRIGGER_ADCB1	CLA_TRIGGER_EPWM1INT	CLA_TRIGGER_ECAP5INT	CLA_TRIGGER_SPITXBINT
CLA_TRIGGER_ADCB2	CLA_TRIGGER_EPWM2INT	CLA_TRIGGER_ECAP6INT	CLA_TRIGGER_SPIRXBINT
CLA_TRIGGER_ADCB3	CLA_TRIGGER_EPWM3INT	CLA_TRIGGER_ECAP7INT	CLA_TRIGGER_LINAIN1
CLA_TRIGGER_ADCB4	CLA_TRIGGER_EPWM4INT	CLA_TRIGGER_EQEP1INT	CLA_TRIGGER_LINAIN0
CLA_TRIGGER_ADCBEVT	CLA_TRIGGER_EPWM5INT	CLA_TRIGGER_EQEP2INT	CLA_TRIGGER_CLA1PROMCRC
CLA_TRIGGER_ADCC1	CLA_TRIGGER_EPWM6INT	CLA_TRIGGER_ECAP6INT2	CLA_TRIGGER_FSITXAINT1
CLA_TRIGGER_ADCC2	CLA_TRIGGER_EPWM7INT	CLA_TRIGGER_ECAP7INT2	CLA_TRIGGER_FSITXAINT2
CLA_TRIGGER_ADCC3	CLA_TRIGGER_EPWM8INT	CLA_TRIGGER_SDFM1INT	CLA_TRIGGER_FSIRXAINT1
CLA_TRIGGER_ADCC4	CLA_TRIGGER_TINT0	CLA_TRIGGER_SDFM1DRINT1	CLA_TRIGGER_FSIRXAINT2

- ◆ Select 'CLA_TRIGGER_SOFTWARE' if task is unused or software triggered (default value)

Software Trigger

Software Triggering a Task

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

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

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

`CLA_forceTask(base, taskFlags);`

- ◆ *base* is the CLA base address: CLA1_BASE
- ◆ *taskFlags* value is the bitwise OR of the tasks:
CLA_TASKFLAG_x (x = 1 to 8) or CLA_TASKFLAG_ALL

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

`CLA_[enable|disable]IACK(base);`

- ◆ Then trigger the task with the assembly instruction:
`asm(" IACK #<Task>");`
- ◆ For example, to trigger TASK4:
`asm(" IACK #0x0008");`
- ◆ More efficient – function does not require EALLOW

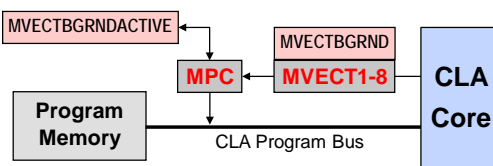
Background Task

Background Task

- ◆ Option to run 8 tasks or 7 tasks and 1 background task
 - ◆ Task 8 can be set to be the background task
 - ◆ While Tasks 1-7 service peripheral triggers in the foreground
 - ◆ Runs continuously until disabled or device/soft reset
 - ◆ Can be triggered by a peripheral or software
 - ◆ Tasks 1 - 7 can interrupt background task in priority order (Task1 is highest, Task7 is lowest)
 - ◆ Can make portions of background task uninterruptible, if needed
- ◆ Background task useful for continuous functions such as communications and clean-up routines

Background Task Registers

- ◆ **MVECTBGRND** register contains the background task vector
- ◆ Branch return address is saved to **MVECTBGRNDACTIVE** register
 - ◆ Address gets popped to the MPC when execution returns



- ◆ Enable / disable background task
`CLA_[enable|disable]BackgroundTask(base);`
- ◆ Start the background task (provided there are no other pending tasks)
`CLA_startBackgroundTask(base);`
- ◆ Enable / disable background task hardware trigger
`CLA_[enable|disable]HardwareTrigger(base);`
 - ◆ Trigger source for the background task selected by `CLA1TASKSRCSELx`
- ◆ `base` is the CLA base address: `CLA1_BASE`

Background Task Interrupts

- ◆ By default background tasks are interruptible
 - ◆ Highest priority pending task executes first
 - ◆ When task completes, and there are no other pending interrupt, execution returns to the background task
- ◆ Sections of background task can be made uninterruptible
 - ◆ Using compiler intrinsics:
 - ◆ `__disable_interrupts();` // MSETC BGINTM
 - ◆ `__enable_interrupts();` // MCLR C BGINTM

```

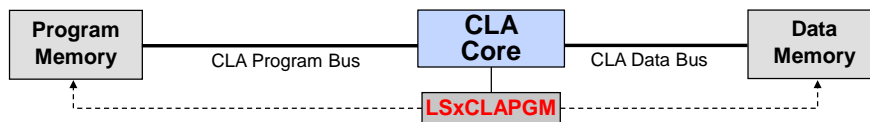
//-----
// Task 8 - Background Task
//-----
__attribute__((interrupt("background"))) void Cla1Task8 ( void )
{
    // Code below is interruptible
    // CODE GOES HERE...
    // Make this portion un-interruptible
    __disable_interrupts();
    // CODE GOES HERE...
    __enable_interrupts();
    // Code below is interruptible
    // CODE GOES HERE...
}

```

Specifies that this is a "background" task instead of a regular interrupt

Memory Configuration

Memory Config Driverlib Functions



- ◆ Set the LSx memory RAM configuration (CPU only or CPU & CLA)


```
MemCfg_setLSRAMMasterSel(ramSection, masterSel);
```
- ◆ Set the CLA memory RAM configuration type (LSx = Data or Program)

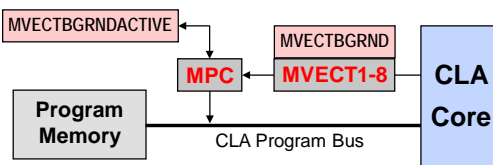

```
MemCfg_setCLAMemType(ramSections, claMemType);
```

- ◆ *ramSection* parameter value is:
 - ◆ MEMCFG_SECT_LSx (x = 0 to 7)
- ◆ *masterSel* value is RAM section dedicated to the CPU or shared between the CPU and the CLA:
 - ◆ MEMCFG_LSRAMMASTER_CPU_ONLY
 - ◆ MEMCFG_LSRAMMASTER_CPU_CLA1
- ◆ *ramSections* parameter value is an OR of:
 - ◆ MEMCFG_SECT_LSx (x = 0 to 7)
- ◆ *claMemType* value is RAM section is configured as CLA data memory or CLA program memory:
 - ◆ MEMCFG_CLA_MEM_DATA or MEMCFG_CLA_MEM_PROGRAM

Task Vector

Task Vector Driverlib Functions

- ◆ Task interrupt vector registers (MVECT1 to MVECT8) contain the start address for each task



- ◆ Map CLA task interrupt vector ($MVECT_x$)
`CLA_mapTaskVector(base, clIntVect, claTaskAddr);`
- ◆ Map CLA background task interrupt vector ($MVECTBGRND$)
`CLA_mapBackgroundTaskVector(base, claTaskAddr);`

- ◆ *base* is the CLA base address: CLA1_BASE
- ◆ *clIntVect* parameter is the CLA interrupt vector value:
 - ◆ CLA_MVECT_x ($x = 1$ to 8)
- ◆ *claTaskAddr* is the start address of the task code

CLA Initialization

CLA Initialization

Performed by the CPU during software initialization

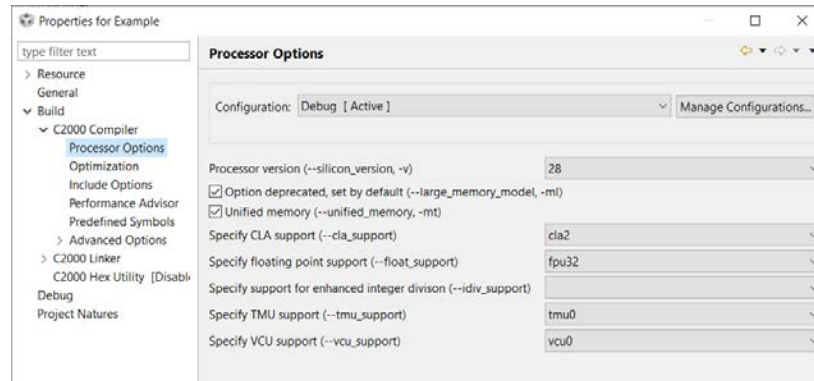
1. Copy CLA task code from flash to CLA program RAM
2. Initialize CLA data RAMs, as needed
 - ◆ Populate with data coefficients, constants, etc.
3. Configure the CLA registers
 - ◆ Enable the CLA clock – `SysCtl_enablePeripheral(SYSCTL_PERIPH_CLK_CLA1);`
 - ◆ Populate the CLA task interrupt vectors (MVECT1-8 registers)
 - ◆ Select the desired task interrupt sources (CLA1TASKSRCSELx registers)
 - ◆ If desired, enable IACK instruction to start tasks using software
 - ◆ Map CLA program RAM and data RAMs to CLA space
4. Configure desired CLA task completion interrupts in the PIE
5. Enable CLA task triggers in the MIER register
6. Initialize the desired peripherals to trigger the CLA tasks

Data can be passed between the CLA and CPU via message RAMs or allocated CLA Data RAM

Enabling CLA Support in CCS

Enabling CLA Support in CCS

- ◆ Set the “Specify CLA support” project option to ‘cla2’
- ◆ 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 C Programming

CLA Task C Programming Language Implementation

- ◆ Supports C only (no C++ or GCC extension support)
- ◆ Different data type sizes than C28x CPU and FPU

TYPE	CPU and FPU	CLA
char	16 bit	16 bit
short	16 bit	16 bit
int	16 bit	32 bit
long	32 bit	32 bit
long long	64 bit	32 bit
float	32 bit	32 bit
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
 - ◆ 16-bit values mostly used to read/write 16-bit peripheral registers
 - ◆ There is no SW or HW support for 64-bit integer or floating point

CLA Task C Language Restrictions (1 of 2)

- ◆ **No initialization support for global and static local variables**

```
int16_t x;           // valid
int16_t x=5;        // not valid
```

- ◆ **Initialized global variables should be declared in a .c file instead of the .cla file**

```
.c file:           .cla file:
int16_t x=5;       extern int16_t x;
```

- ◆ **For initialized static variables, easiest solution is to use an initialized global variable instead**
- ◆ **No recursive function calls**
- ◆ **No function pointers**

CLA Task C Language Restrictions (2 of 2)

- ◆ **No support for certain fundamental math operations**

- ◆ **integer division:** $z = x/y;$
- ◆ **modulus (remainder):** $z = x\%y;$
- ◆ **unsigned 32-bit integer compares**

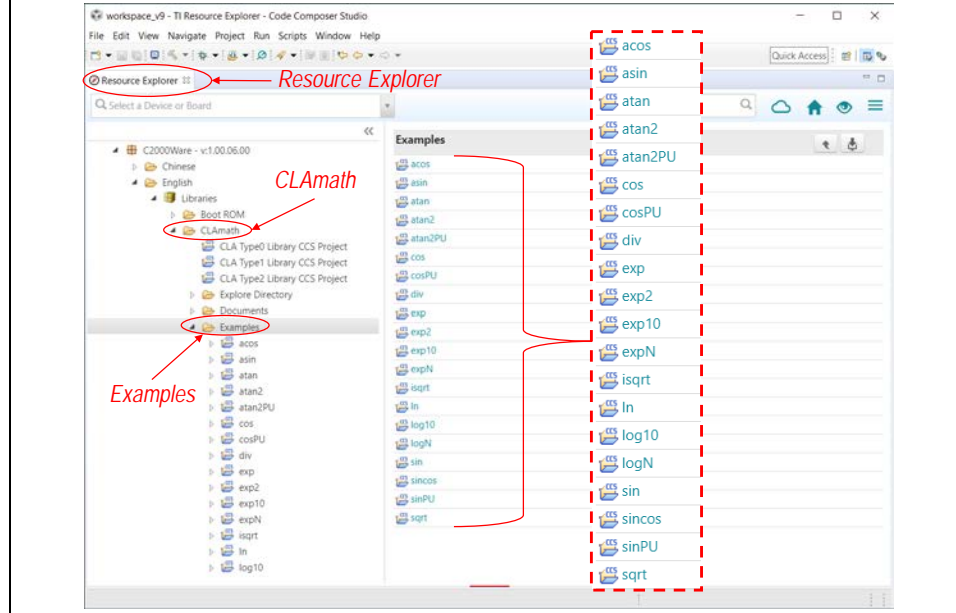
```
uint32_t i;  if(i < 10) {...}    // not valid
int32_t i;   if(i < 10) {...}    // valid
uint16_t i;  if(i < 10) {...}    // valid
int16_t i;   if(i < 10) {...}    // valid
float32_t x; if(x < 10) {...}    // valid
```

- ◆ **No standard C math library functions, but TI provides some function examples (next slide)**

C2000Ware – CLA Software Support

C2000Ware™ - CLA Software Support

- ◆ TI provides some examples of floating-point math CLA functions



CLA Compiler Scratchpad Memory Area

CLA Compiler Scratchpad Memory Area

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

Lab.cmd

```
MEMORY
{
    :
}

SECTIONS
{
    :
    /** CLA Compiler Required Sections **/
    .scratchpad : > RAMLS0, PAGE = 1
    :
}
```

CLA Initialization Code Example

CLA Initialization Code Example

Lab.h

```

#include "driverlib.h" // cla.h
#include "f28004x_device.h"
:
extern interrupt void Cla1Task1();
extern interrupt void Cla1Task2();
:
extern interrupt void Cla1Task8();

```

Cla.c

```

#include "Lab.h"

// Initialize CLA task interrupt vectors
CLA_mapTaskVector(CLA1_BASE, CLA_MVECT_1, (uint16_t)&Cla1Task1);
CLA_mapTaskVector(CLA1_BASE, CLA_MVECT_2, (uint16_t)&Cla1Task2);
:           :           :           :
CLA_mapTaskVector(CLA1_BASE, CLA_MVECT_7, (uint16_t)&Cla1Task7);
CLA_mapTaskVector(CLA1_BASE, CLA_MVECT_8, (uint16_t)&Cla1Task8);

```

- ◆ Defines data types and special registers specific to the CLA
- ◆ Defines register bit field structures
- ◆ CLA task prototypes are prefixed with the 'interrupt' keyword
- ◆ CLA task symbols are visible to all C28x CPU and CLA code

CLA Task C Code Example

CLA Task C Code Example

ClaTasks_C.cla

```

#include "F28004x_device.h"
#include "Lab.h"
;-----
interrupt void Cla1Task1 (void)
{
:
__mdebugstop1();
:
xDelay[0] = (float32_t)AdcaResultRegs.ADCRESULT0;
Y = coeffs[4] * xDelay[4];
xDelay[4] = xDelay[3];
:
xDelay[1] = xDelay[0];
Y = Y + coeffs[0] * xDelay[0];
ClaFilteredOutput = (uint16_t)Y;
}
;-----
interrupt void Cla1Task2 (void)
{
:
}
;-----

```

- ◆ .cla extension causes the c2000 compiler to invoke the CLA compiler
- ◆ Bit Field peripheral address definitions
- ◆ 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 Code Debugging

CLA Code Debugging

- ◆ The CLA and CPU are debugged from the same JTAG port
- ◆ You can halt, single-step, and run the CLA independent of the CPU

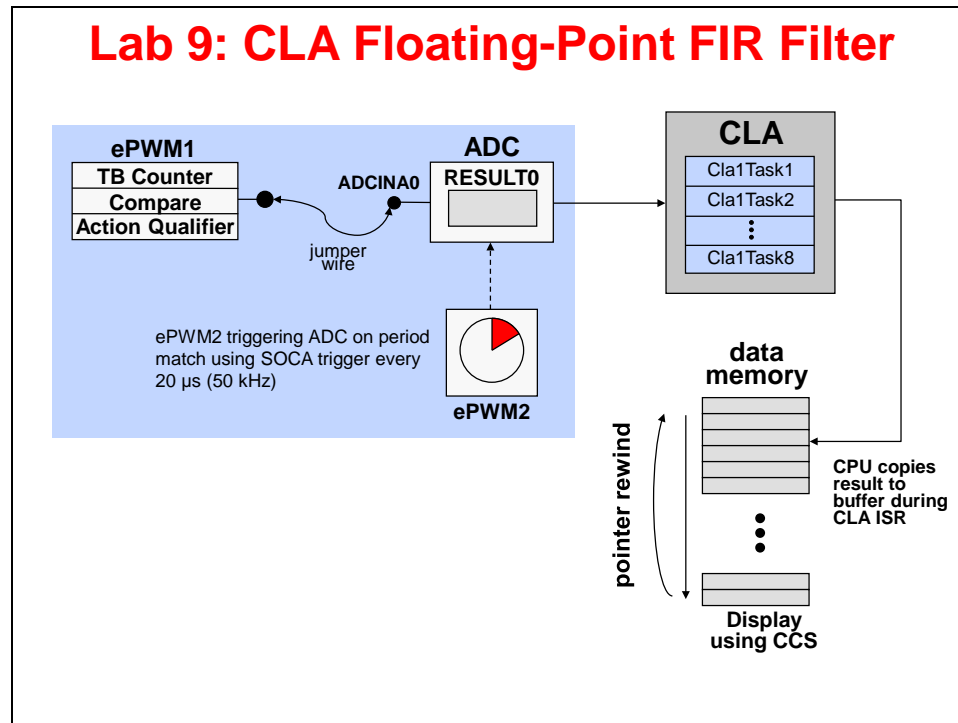
- 1. Insert a breakpoint in the CLA code**
 - ◆ Insert a MDEBUGSTOP1 instruction(s) in the code where desired then rebuild/reload
 - ◆ In C code, can use `asm(" MDEBUGSTOP1")`
 - ◆ When the debugger is not connected, the MDEBUGSTOP1 acts like an MNOP
- 2. Connect to the CLA target in CCS**
 - ◆ This enables CLA breakpoints
- 3. Run the CPU target**
 - ◆ CLA task will trigger (via peripheral interrupt or software)
 - ◆ CLA executes instructions until MDEBUGSTOP1 is hit
- 4. Load the code symbols into the CLA context in CCS**
 - ◆ This allows source-level debug
 - ◆ Needs to be done only once per debug session unless the .out file changes
- 5. Debug the CLA code**
 - ◆ Can single-step the code, or run to the next MDEBUGSTOP1 or to the end of the task
 - ◆ If another task is pending, it will start at the end of the previous task
- 6. Disconnect the CLA target to disable CLA breakpoints, if desired**

Note: when using the legacy MDEBUGSTOP instruction, a CLA single step executes one pipeline cycle, whereas a CPU single step executes one instruction (and flushes the pipeline); see TRM for details

Lab 9: CLA Floating-Point FIR Filter

➤ Objective

The objective of this lab exercise is to become familiar with operation and programming of the CLA. In this lab exercise, the ePWM1A generated 2 kHz, 25% duty cycle symmetric 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 exercise 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.

➤ Procedure

Open the Project

1. A project named `Lab9` has been created for this lab exercise. Open the project by clicking on `Project` → `Import CCS Projects`. The “Import CCS Eclipse Projects” window will open. Click `Browse...` next to the “Select search-directory” box. Navigate to: `C:\F28004x\Labs\Lab9\project` and click `Select Folder`. Then click `Finish` to import the project. All build options have been configured the same as the previous lab exercise. The files used in this lab exercise are:

Adc.c	EPwm.c
Cla_9.c	f28004x_globalvariabledefs.c
ClaTasks_C.cla	f28004x_headers_nonbios.cmd
CodeStartBranch.asm	Gpio.c
Dac.c	Lab_9.cmd
DefaultIsr_9_10.c	Main_9.c
device.c	SineTable.c
Dma.c	Watchdog.c
ECap.c	

Project Build Options and Enabling CLA Support in CCS

- In this lab exercise the *Bit Field Header Files* are used for reading the ADC result register in the CLA task file (ClaTasks_C.cla). We need to setup the include search path to include the bit field header files. Open the build options by right-clicking on Lab9 in the Project Explorer window and select “Properties”. Under “C2000 Compiler” select “Include Options”. In the include search path box that opens (“Add dir to #include search path”) click the Add icon. Then in the “Add directory path” window type:

```
#{PROJECT_ROOT}/../../../../f28004x_headers/include
```

Click OK to include the search path.

Note: from the bit field header files, f28004x_globalvariabledefs.c and f28004x_headers_nonbios.cmd have already been added to the project.

- Next, we will confirm that CLA support has been enabled. Under “C2000 Compiler” select “Processor Options” and notice the “Specify CLA support” is set to cla2. This is needed to compile and assemble CLA code. Click Apply and Close to save and close the Properties window.

Inspect Lab_9.cmd

- Open and inspect Lab_9.cmd. Notice that a section called “Cla1Prog” is being linked to RAMLS4. This section links the CLA program tasks to the CPU memory space. Two other sections called “Cla1Data1” and “Cla1Data2” are being linked to RAMLS1 and RAMLS2, respectively, for the CLA data. These memory spaces will be mapped to the CLA memory space during initialization. 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 exercise 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.

- The CLA C compiler uses a section called .scratchpad for storing local and compiler generated temporary variables. This scratchpad memory area is allocated using the linker command file. Notice .scratchpad is being linked to RAMLS0. Close the Lab_9.cmd linker command file.

Setup CLA Initialization

During the CLA initialization, the CPU memory block RAMLS4 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 ADCAINT1 interrupt trigger. The next section will setup the PIE interrupt for the CLA.

6. 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.
7. Edit `Clas_9.c` to implement the CLA operation as described in the objective for this lab exercise:
 - Set Task 1 peripheral interrupt trigger source to ADCA1
 - Set Task 8 peripheral interrupt trigger source to SOFTWARE
 - Disable the Background Task
 - Enable the use of the IACK instruction to trigger a task
 - Enable CLA Task 8 interrupt for one-time initialization routine (clear delay buffer)
 - Enable CLA Task 1 interrupt

Note: the CLA has been configured for `RAMLS0`, `RAMLS1`, `RAMLS2`, and `RAMLS4` memory blocks to be shared between the CPU and CLA. The `RAMLS4` memory block is mapped to CLA program memory space, and the `RAMLS0`, `RAMLS1` and `RAMLS2` memory blocks are mapped to CLA data memory space. Also, the `RAMLS0` memory block is used for the CLA C compiler scratchpad. Notice that CLA Task 8 interrupt is disabled after the one-time initialization routine (clear delay buffer) is completed.

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

```
InitCla();
```

at the desired spot in `main()`.

9. In `Main_9.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 Control Peripherals lab exercise (i.e. ePWM lab), the ADC generated an interrupt to the CPU, and the CPU read the ADC result register 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 read the ADC result register 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.

10. Remember that in `Adc.c` we *commented out* the code used to enable the ADCA1 interrupt in PIE group 1. This is no longer being used. The CLA interrupt will be used instead.
11. Using the “PIE Interrupt Assignment Table” find the location for the CLA Task 1 interrupt “`INT_CLA1_1`” and fill in the following information:

PIE group #: _____ # within group: _____

This information will be used in the next step.

12. Modify the end of `Cla_9.c` to do the following:
 - Add the Driverlib function to re-map the CLA1_1 interrupt signal to call the ISR function. (Hint: `#define` name in `driverlib/inc/hw_ints.h` and label name in `DefaultIsr_9_10.c`)
 - Add the Driverlib function to enable the appropriate PIEIER and core IER
13. Open and inspect `DefaultIsr_9_10.c`. Notice that this file contains the CLA interrupt service routine. Save all modified files.

Build and Load

14. Click the “Build” button and watch the tools run in the Console window. Check for errors in the Problems window.
15. 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_RAM` using the Scripts menu.

Run the Code – Test the CLA Operation

Note: For the next step, check to be sure that the jumper wire connecting PWM1A (pin #80) to ADCINA0 (pin #70) is in place on the LaunchPad.

16. Run the code (real-time mode). Open and watch the memory browser window update. Verify that the ADC result buffer contains updated values.
17. 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	μs

18. 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.
19. Halt the code.

Terminate Debug Session and Close Project

20. Terminate the active debug session using the `Terminate` button. This will close the debugger and return Code Composer Studio to the CCS Edit perspective view.
21. Next, close the project by right-clicking on `Lab9` in the Project Explorer window and select `Close Project`.

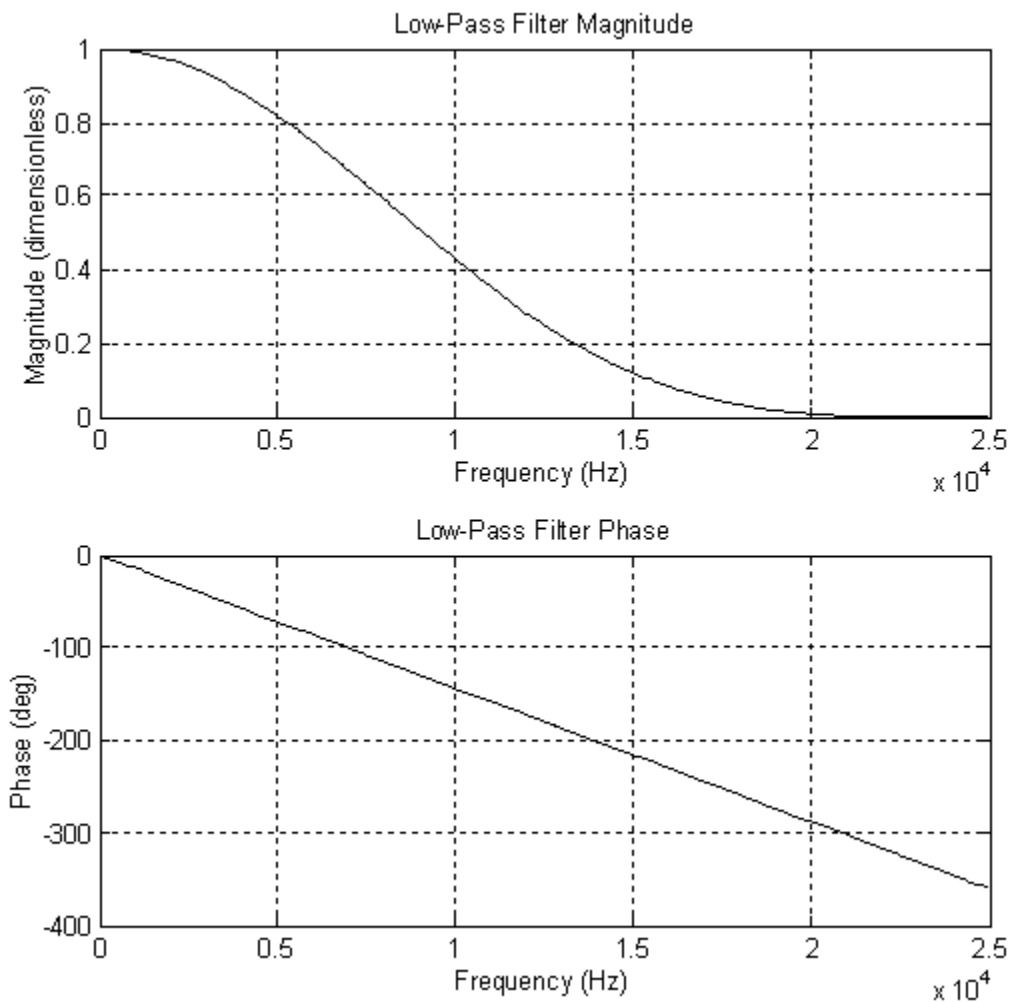
End of Exercise

Lab 9 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



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

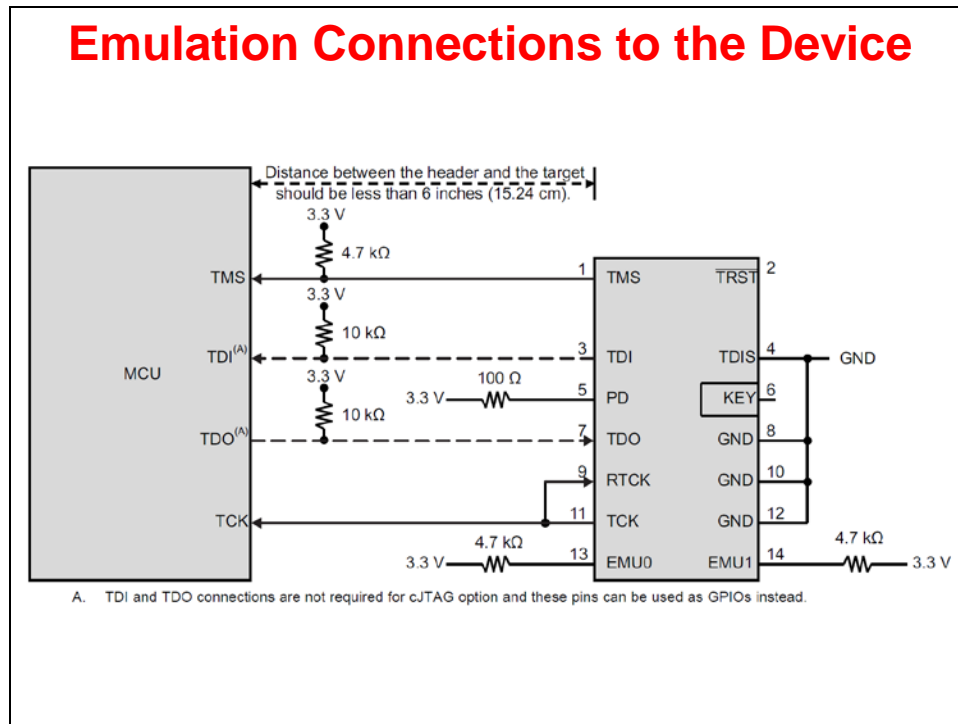
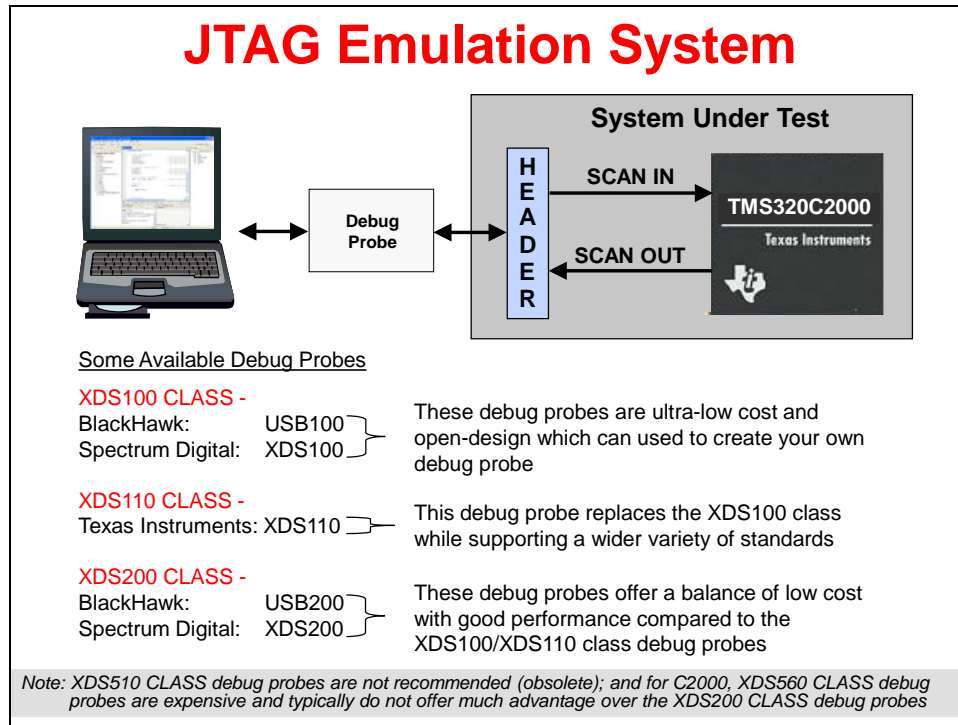
Module Objectives

- ◆ **JTAG Emulation**
- ◆ **Analysis and Diagnostic Capabilities**
- ◆ **Flash Configuration and Memory Performance**
- ◆ **Flash Programming**
- ◆ **Dual Code Security Module (DCSM)**

Chapter Topics

System Design	10-1
<i>Emulation and Analysis Block.....</i>	<i>10-3</i>
<i>Analysis and Diagnostic Capabilities</i>	<i>10-5</i>
<i>Flash Configuration and Memory Performance</i>	<i>10-7</i>
<i>Flash Programming.....</i>	<i>10-11</i>
<i>Dual Code Security Module (DCSM)</i>	<i>10-13</i>
<i>Lab 10: Programming the Flash.....</i>	<i>10-17</i>

Emulation and Analysis Block



Emulation Mode Driverlib Functions

Selects the behavior of the peripheral during emulation control

Driverlib Function	Options
CLAPROMCRC_setEmulationMode()	CLAPROMCRC_MODE_SOFT CLAPROMCRC_MODE_FREE
CPUTimer_setEmulationMode()	CPUTIMER_EMULATIONMODE_STOPAFTERNEXTDECREMENT CPUTIMER_EMULATIONMODE_STOPATZERO CPUTIMER_EMULATIONMODE_RUNFREE
DMA_setEmulationMode()	DMA_EMULATION_STOP DMA_EMULATION_FREE_RUN
ECAP_setEmulationMode()	ECAP_EMULATION_STOP ECAP_EMULATION_RUN_TO_ZERO ECAP_EMULATION_FREE_RUN
EPWM_setEmulationMode()	EPWM_EMULATION_STOP_AFTER_NEXT_TB EPWM_EMULATION_STOP_AFTER_FULL_CYCLE EPWM_EMULATION_FREE_RUN
EQEP_setEmulationMode()	EQEP_EMULATIONMODE_STOPIMMEDIATELY EQEP_EMULATIONMODE_STOPATROLLOVER EQEP_EMULATIONMODE_RUNFREE
I2C_setEmulationMode()	I2C_EMULATION_STOP_SCL_LOW I2C_EMULATION_FREE_RUN
SPI_setEmulationMode()	SPI_EMULATION_STOP_MIDWAY SPI_EMULATION_STOP_AFTER_TRANSMIT SPI_EMULATION_FREE_RUN

Note: see the F28004x Driverlib User's Guide for detailed usage

Analysis and Diagnostic Capabilities

Analysis and Diagnostic Capabilities

- ◆ **C28x CPU has two hardware analysis units:**
 - ◆ Address and Data Comparison Units (ACU/DCU)
 - ◆ ACU – counts events or monitors address buses
 - ◆ DCU – monitors address and data buses
 - ◆ ACU and DCU can be configured as analysis breakpoints or watchpoints; in addition, ACU can be configured as a benchmark counter or event counter
- ◆ **Embedded real-time analysis and diagnostic (ERAD) module:**
 - ◆ Enhances the debug and system analysis capabilities
 - ◆ ERAD module is implemented external to the C28x CPU core
 - ◆ ERAD module consists of Enhanced Bus Comparator Units (EBC) and Benchmark System Event Counter Units (BSEC)
 - ◆ EBC – used to generate hardware breakpoints, hardware watchpoints and other output events
 - ◆ BSEC – used to analyze and profile the system

C28x CPU Hardware Analysis Units

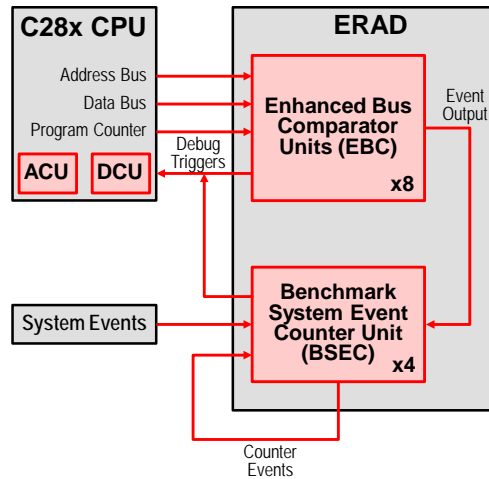
- ◆ The C28x CPU two hardware analysis units can be configured to provide any one of the following advanced debug features:

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

ERAD Module

ERAD expands device debug and system analysis capabilities to 10 hardware breakpoints and 10 hardware watchpoints

- ◆ **8 Enhanced Bus Comparator Units**
 - ◆ Similar to ACU/DCU
 - ◆ Inputs: address bus, program counter, data bus
 - ◆ Comparison mode
 - ◆ (>, >=, <, <=)
 - ◆ Output: debug triggers, event output
- ◆ **4 Benchmark System Event Counter Units**
 - ◆ Similar to count feature in ACU
 - ◆ Inputs: events from EBC, system events
 - ◆ Outputs: debug events
- ◆ ERAD can be used by the application or debugger



Flash Configuration and Memory Performance

Basic Flash Operation

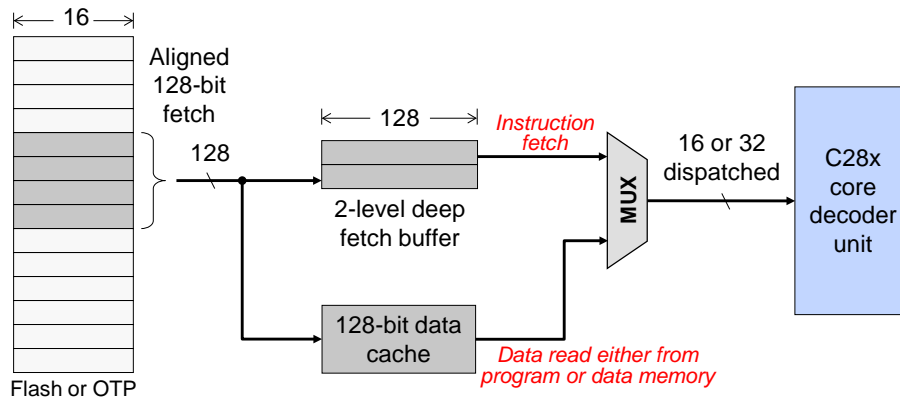
- ◆ RWAIT bit-field in the FRDCNTL register specifies the number of random accesses wait states
- ◆ OTP reads are hardwired for 10 wait states (RWAIT has no effect)
- ◆ Must specify the number of SYSCLK cycle wait-states; *Reset defaults are maximum value (15)*
- ◆ Flash/OTP reads returned after (RWAIT + 1 SYSCLK cycles)
- ◆ Flash configuration code should not be run from the flash memory

- ◆ Set the number of wait states for a flash read access
`Flash_setWaitstates(ctrlBase, waitstates);`
- ◆ *ctrlBase* is base address of the flash control registers: FLASH0CTRL_BASE
- ◆ *waitstates* value is a number between 0 and 15

*** Refer to the F28004x data sheet for value details ***

For 100 MHz, RWAIT = 4

Speeding Up Execution in Flash / OTP



- ◆ Enable prefetch mechanism:
`Flash_enablePrefetch(ctrlBase);`
- ◆ Enable data cache:
`Flash_enableCache(ctrlBase);`
- ◆ *ctrlBase* is base address of the flash control registers: FLASH0CTRL_BASE

Code Execution Performance

- ◆ Assume 100 MHz SYSCLKOUT and single-cycle execution for each instruction

Internal RAM: 100 MIPS

Fetch up to 32 bits every cycle → 1 instruction/cycle

Flash: 100 MIPS

Assume RWAIT=4, prefetch buffer enabled

Fetch 128 bits every 4 cycles:

(128 bits) / (32-bits per instruction worst-case) → 4 instructions/4 cycles

PC discontinuities will degrade this

Benchmarking in control applications has shown actual performance of about 90% efficiency, yielding approximately 90 MIPS

Data Access Performance

- ◆ Assume 100 MHz SYSCLKOUT

Memory	16-bit access (words/cycle)	32-bit access (words/cycle)	Notes
Internal RAM	1	1	
Flash 'sequential' access	0.73 (8 words/11 cycles)	0.57 (4 words/7 cycles)	Assumes RWAIT = 4, flash data cache enabled, all 128 bits in cache are used
Flash random access	0.25 (1 word/4 cycles)	0.25 (1 word/4 cycles)	Assumes RWAIT = 4

- ◆ Internal RAM has best data performance – put time critical data here
- ◆ Flash performance often sufficient for constants and tables
- ◆ Note that the flash instruction fetch pipeline will also stall during a flash data access
- ◆ For best flash performance, arrange data so that all 128 bits in a cache line are utilized (e.g. sequential access)

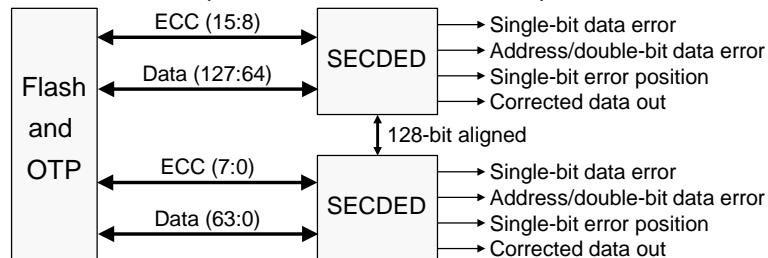
Flash / OTP Power Modes

- ◆ Power configuration settings save power by putting Flash/OTP to 'Sleep' or 'Standby' mode; flash will automatically enter 'Active' mode if a Flash/OTP access is made
- ◆ At reset Flash/OTP is in sleep mode
- ◆ Operates in three power modes:
 - ◆ Sleep (lowest power)
 - ◆ Standby (shorter transition time to active)
 - ◆ Active (highest power)
- ◆ After an access is made, Flash/OTP can automatically power down to 'Standby' or 'Sleep' (active grace period set in user programmable counters)

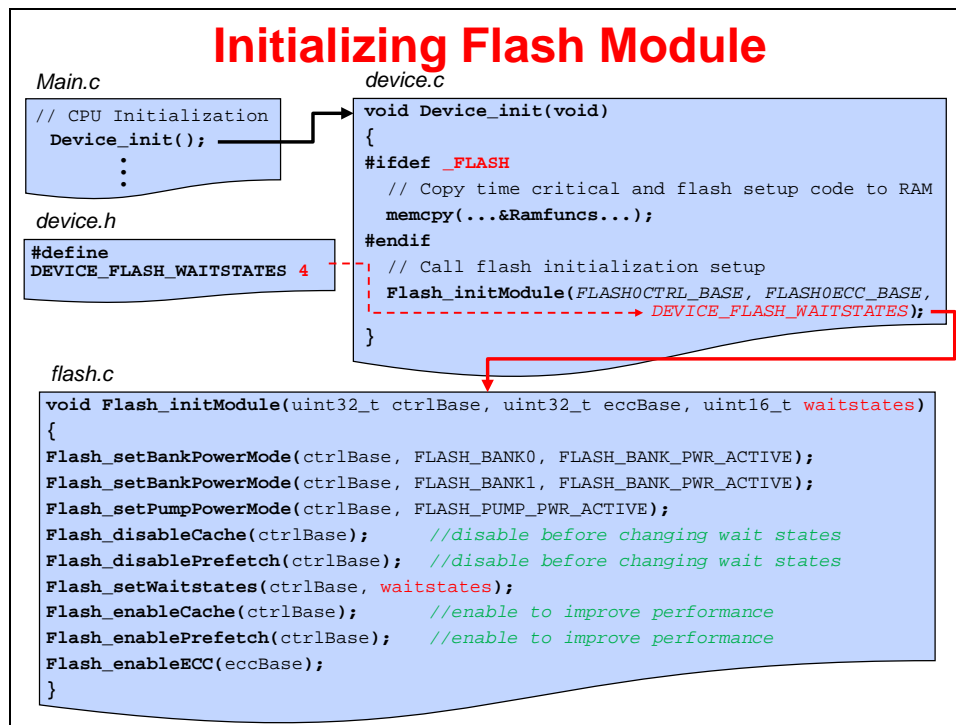
- ◆ Set fallback power mode for flash bank:
`Flash_setBankPowerMode(ctrlBase, bank, powerMode);`
- ◆ Set fallback power mode for charge pump:
`Flash_setPumpPowerMode(ctrlBase, powerMode);`
- ◆ `ctrlBase` is base address of the flash control registers: FLASH0CTRL_BASE
- ◆ `bank` parameter is: FLASH_BANK0 or FLASH_BANK1
- ◆ `powerMode` value for:
 - ◆ Bank – FLASH_BANK_PWR_x (x = SLEEP, STANDBY, or ACTIVE)
 - ◆ Pump – FLASH_PUMP_PWR_x (x = SLEEP or ACTIVE)

Error Correction Code (ECC) Protection

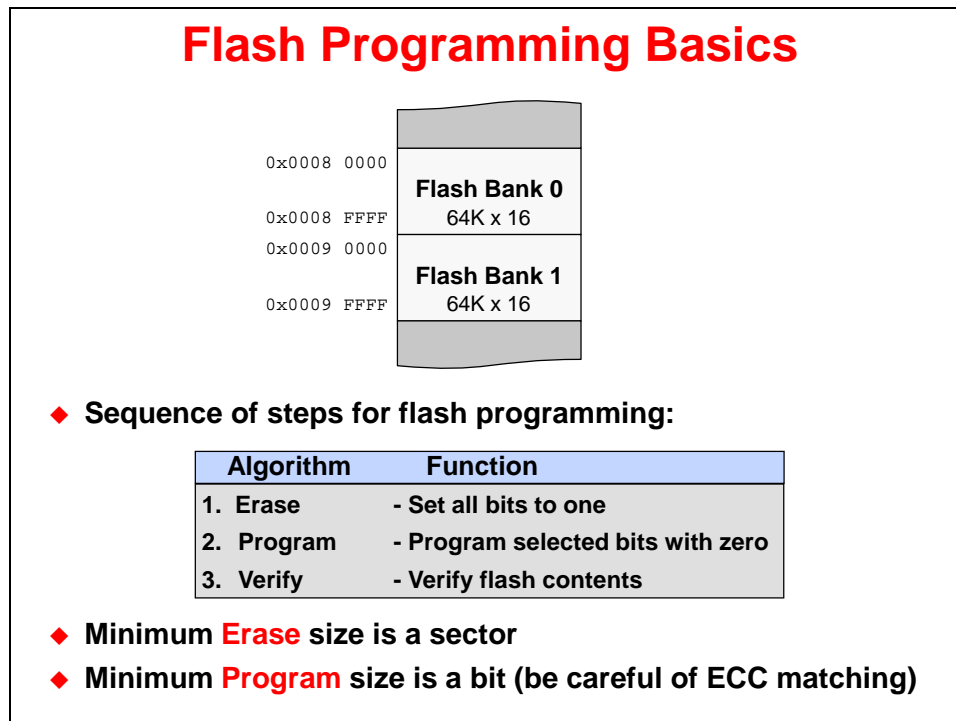
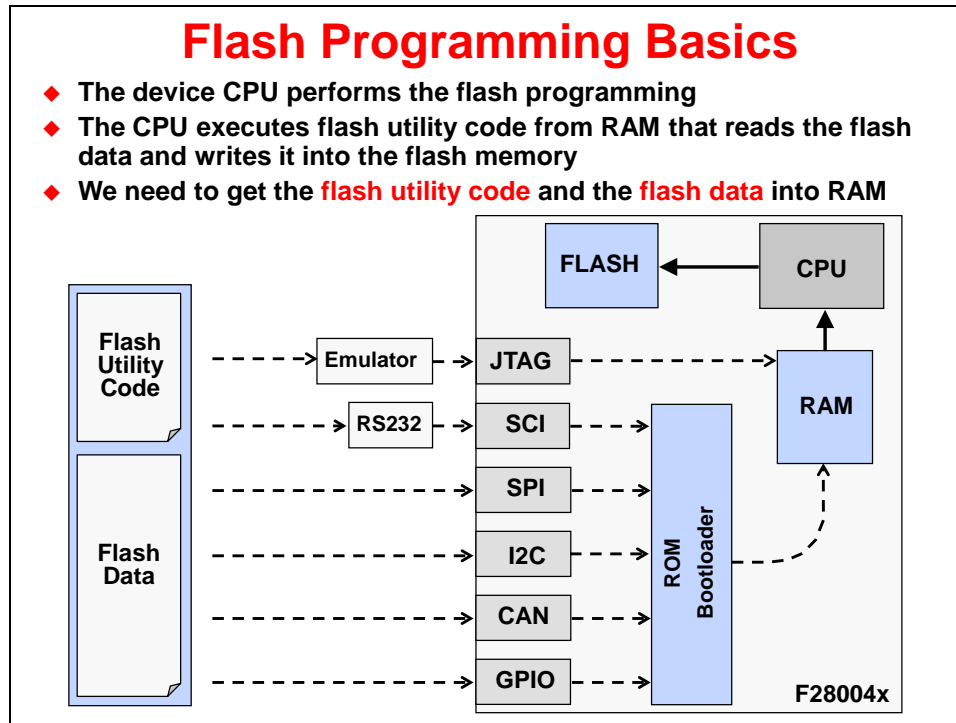
- ◆ Provides capability to screen out Flash/OTP memory faults (enabled at reset)
- ◆ Single error correction and double error detection (SECEDED)
- ◆ For every 64-bits of Flash/OTP, 8 ECC check bits are calculated and programmed into ECC memory
- ◆ ECC check bits are programmed along with Flash/OTP data
- ◆ During an instruction fetch or data read operation the 64-bit data/8-bit ECC are processed by the SECEDED to determine one of three conditions:
 - ◆ No error occurred
 - ◆ A correctable error (single bit data error) occurred
 - ◆ A non-correctable error (double bit data error or address error) occurred



- ◆ Enable ECC protection: `Flash_enableECC(eccBase);`
- ◆ `ctrlBase` is base address of the ECC registers: FLASH0ECC_BASE



Flash Programming



Flash Programming Utilities

- ◆ **JTAG Emulator Based**
 - ◆ CCS on-chip Flash programmer (Tools → On-Chip Flash)
 - ◆ CCS UniFlash (TI universal Flash utility)
 - ◆ BlackHawk Flash utilities (requires Blackhawk emulator)
 - ◆ Elprotronic FlashPro2000
- ◆ **SCI Serial Port Bootloader Based**
 - ◆ CodeSkin C2Prog
 - ◆ Elprotronic FlashPro2000
- ◆ **Production Test/Programming Equipment Based**
 - ◆ BP Microsystems 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>)

Dual Code Security Module (DCSM)

Dual Code Security Module (DCSM)

- ◆ DCSM offers protection for two zones – zone 1 & zone 2
- ◆ Each zone has its own dedicated secure OTP
 - ◆ Contains security configurations for each zone
- ◆ The following on-chip memory can be secured:

- ◆ Flash – each sector individually
 - ◆ LS0-7 RAM – each block individually
- ◆ Data reads and writes from secured memory are only allowed for code running from secured memory
- ◆ All other data read/write accesses are blocked:
 - JTAG emulator/debugger, ROM bootloader, code running in external memory or unsecured internal memory

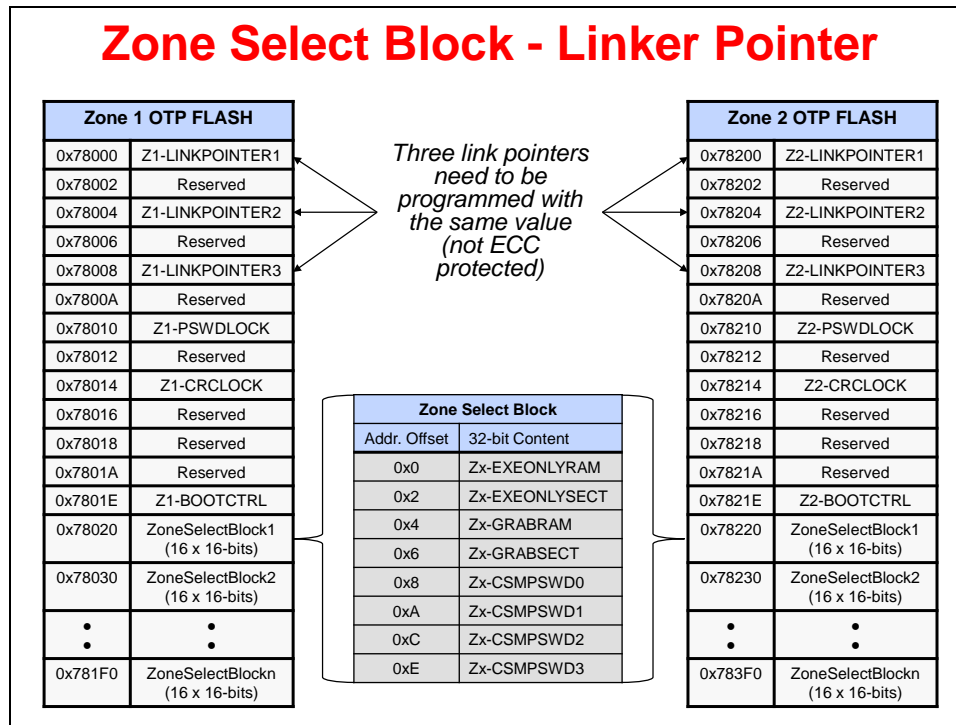
Zone Selection

- ◆ Each securable on-chip memory resource can be allocated to either zone 1 (Z1), zone 2 (Z2), or as non-secure

- ◆ **DcsmZ1Regs.Z1_GRABSECTR** register:
 - ◆ Allocates individual flash sectors to zone 1 or non-secure
 - ◆ **DcsmZ2Regs.Z2_GRABSECTR** register:
 - ◆ Allocates individual flash sectors to zone 2 or non-secure
 - ◆ **DcsmZ1Regs.Z1_GRABRAMR** register:
 - ◆ Allocates LS0-7 to zone 1 or non-secure
 - ◆ **DcsmZ2Regs.Z2_GRABRAMR** register:
 - ◆ Allocates LS0-7 to zone 2 or non-secure

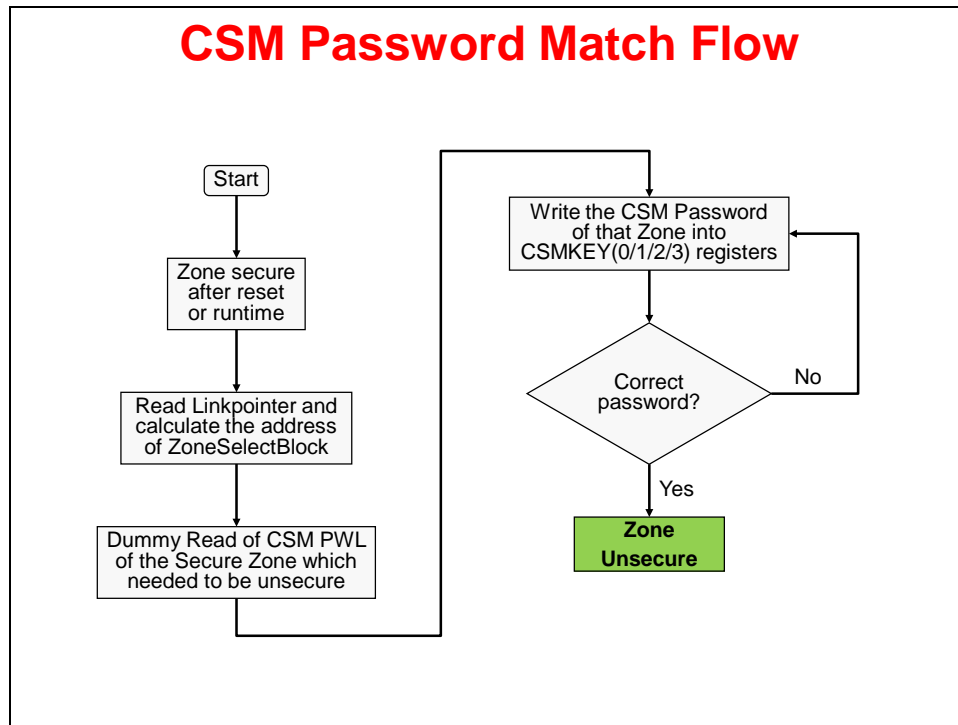
Technical Reference Manual contains a table to resolve mapping conflicts

Zone Select Block - Linker Pointer



Secure and Unsecure the CSM

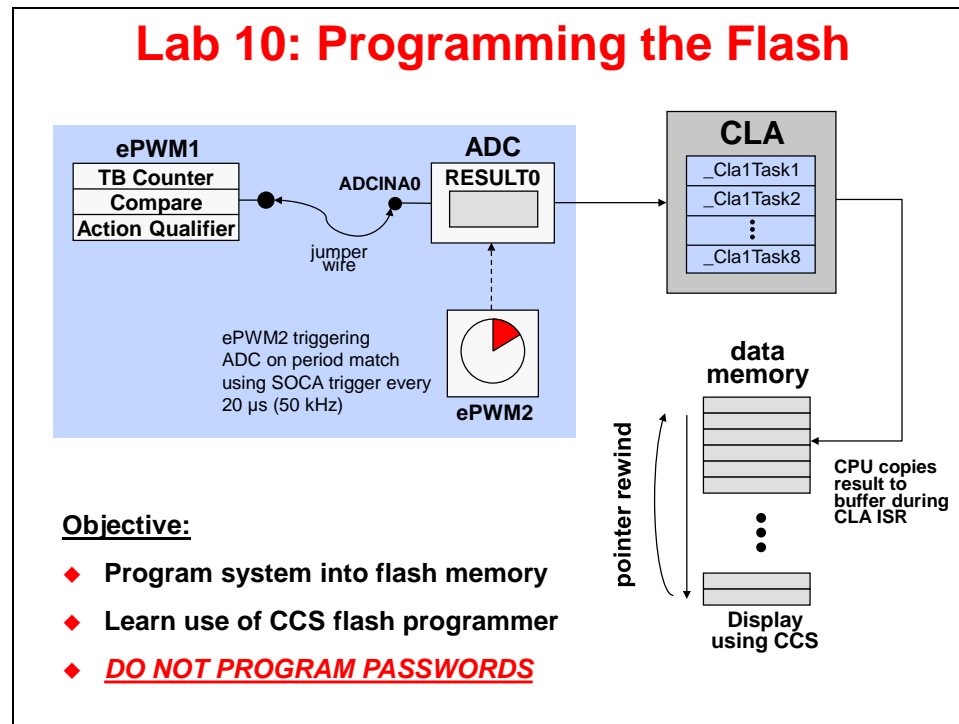
- ◆ The CSM is always secured after reset
- ◆ To unsecure the CSM:
 - ◆ Perform a dummy read of each CSMPSWD(0,1,2,3) register (*passwords in the OTP*)
 - ◆ Write the correct password to each CSMKEY(0,1,2,3) register
- ◆ The boot ROM code will automatically unlock the device as part of the initialization sequence for devices that do not have passwords programmed
 - ◆ See *Technical Reference Manual* for details



Lab 10: Programming the Flash

➤ Objective

The objective of this lab exercise is to program and execute code from the on-chip flash memory. The TMS320F280049C 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 exercise, the steps required to properly configure the software for execution from internal flash memory will be covered.



➤ Procedure

Open the Project

1. A project named Lab10 has been created for this lab exercise. Open the project by clicking on Project → Import CCS Projects. The "Import CCS Eclipse Projects" window will open. Click Browse... next to the "Select search-directory" box. Navigate to: C:\F28004x\Labs\Lab10\project and click Select Folder. Then click Finish to import the project. All build options have been configured the same as the previous lab exercise. The files used in this lab exercise are:

Adc.c	EPwm.c
Cla.c	f28004x_globalvariabledefs.c
ClaTasks_C.cla	f28004x_headers_nonbios.cmd
CodeStartBranch.asm	Gpio.c
Dac.c	Lab_10.cmd
DefaultIsr_9_10.c	Main_10.c
device.c	SineTable.c
Dma.c	Watchdog.c
ECap.c	

Project Build Options

- We need to setup the predefined symbols for programming the flash. Open the build options by right-clicking on Lab10 in the Project Explorer window and select "Properties". Under "C2000 Compiler" select "Predefined Symbols". In the predefined name box that opens ("Pre-define NAME") click the Add icon. Then in the "Enter Value" window type **_FLASH**. This name is used in the project to conditionally include code specific to initializing the flash module. This conditional code is located in the `device.c` file. Click OK to include the name. Then click Apply and Close to save and close the Properties window.

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 embedded system means that no debug probe (emulator) is available to initialize the device RAM. Therefore, all initialized sections must be linked to the on-chip flash memory.

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

- Open and inspect the linker command file `Lab_10.cmd`. Notice that the first flash sector has been divided into two blocks named `BEGIN_FLASH` and `FLASH_BANK0_SEC0`. The `FLASH_BANK0_SEC0` flash sector origin and length has been modified to avoid conflicts with the other flash sector spaces. The remaining flash sectors have been combined into a single block named `FLASH_BANK0_SEC1_15`. Additionally, a second bank is available named `FLASH_BANK1_SEC0_15`. See the reference slide at the end of this lab exercise for further details showing the address origins and lengths of the various flash sectors used.
- Edit `Lab_10.cmd` to link the following compiler sections to on-chip flash memory block `FLASH_BANK0_SEC1_15`:

Compiler Sections:

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

Initializing the Interrupt Vectors from Flash to RAM

The interrupt vectors must be located in on-chip flash memory and at power-up needs to be loaded to the PIE RAM as part of the device initialization procedure. The code that performs this

process is part of Driverlib. In `main()`, the call to `Interrupt_initModule()` enables vector fetching from PIE block, and the call to `Interrupt_initVectorTable()` loads the vector table. In Driverlib, both functions are part of `interrupt.c`.

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 code that performs this process is part of Driverlib. In `main()`, the call to `Device_init()` copies time critical and flash setup code to RAM and then calls `Flash_initModule()` to initialize the flash. In Driverlib, these functions are part of `device.c` and `flash.c`.

5. In the Driverlib, `flash.c` uses the C compiler `CODE_SECTION` pragma to place the `Flash_initModule()` function into a linkable section named `".TI.ramfunc"`.
6. The `".TI.ramfunc"` section will be linked using the user linker command file `Lab_10.cmd`. In `Lab_10.cmd` the `".TI.ramfunc"` will load to flash (load address) but will run from RAMLS5 (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 RAMLS5 memory we are linking `".TI.ramfunc"` to, we are specifying `"PAGE = 0"` (which is program memory).

Dual Code Security Module and Passwords

The DCSM module provides protection against unwanted copying (i.e. pirating!) of your code from flash, OTP, and LS0-7 RAM blocks. The DCSM uses a 128-bit password made up of 4 individual 32-bit words. They are located in the OTP. During this lab exercise, device default passwords will be used – therefore only dummy reads of the password locations are needed to unsecure the DCSM. **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. Again, **DO NOT CHANGE THE DEVICE PASSWORD VALUES.**

Executing from Flash after Reset

The F280049C 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 `0x080000` in the flash. An instruction that branches to the beginning of your program needs to be placed at this address. Note that `BEGIN_FLASH` begins at address `0x080000`. 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`.

7. 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`.

8. In the earlier lab exercises, the section “codestart” was directed to the memory named BEGIN_M0. Edit `Lab_10.cmd` so that the section “codestart” will be directed to BEGIN_FLASH. Save your work.

On power up the reset vector will be fetched and the ROM bootloader will begin execution. If the debug probe (emulator) is connected, the device will be in emulation boot mode and will use the EMU-BOOT registers values in the PIE RAM to determine the boot mode. This mode was utilized in the previous lab exercises. In this lab exercise, we will be disconnecting the debug probe and running in stand-alone boot mode (but do not disconnect the emulator yet!). The bootloader will read the Z1-OTP-BOOT registers values from their locations in the OTP. The behavior when these values have not been programmed (i.e., KEY not 0x5A) 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 RAMLS4 for both load and run purposes. At runtime, all the code did was map the RAMLS4 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 RAMLS4. The C-compiler runtime support library contains a memory copy function called `memcpy()` which will be used to perform the copy. After the copy is performed, the RAMLS4 block will then be mapped to CLA program memory space as was done in the earlier lab.

9. In `Lab_10.cmd` notice that the named section “Cla1Prog” will now load to flash (load address) but will run from RAMLS4 (run address). The linker will also be used to generate symbols for the load start, load size, and run start addresses.
10. Open `Cla.c` and notice that the memory copy function `memcpy()` is being used to copy the CLA program code from flash to RAMLS4 using the symbols generated by the linker. Just after the copy the `Driverlib MemCfg_setCLAMemType()` function is used to configure the RAMLS4 block as CLA program memory space. Close the opened files.

Build – Lab.out


11. 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.

12. Program the flash memory by clicking the “Debug” button (green bug). The CCS Debug perspective view will open and the flash memory will be programmed. *(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. Then the program will load automatically, and you should now be at the start of `main()`.

Running the Code – Using CCS

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

Run → Reset → CPU Reset

The program counter should now be at address 0x3FC7A5 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.

14. Under Scripts on the menu bar click:

EMU Boot Mode Select → EMU_BOOT_FLASH

This has the debugger load values into EMU-BOOT registers so that the bootloader will jump to "Flash" at address 0x080000.

15. Next click:

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.

16. You can now run the CPU, and you should observe the LED5 on the LaunchPad blinking.
17. Halt the CPU.
18. Try resetting the CPU, select the EMU_BOOT_FLASH boot mode, and then hitting run (without doing the Go Main procedure). The LED should be blinking again.
19. Halt the CPU.

Terminate Debug Session and Close Project

20. Terminate the active debug session using the `Terminate` button. This will close the debugger and return Code Composer Studio to the CCS Edit perspective view.
21. Next, close the project by right-clicking on `Lab10` in the Project Explorer window and select `Close Project`.

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

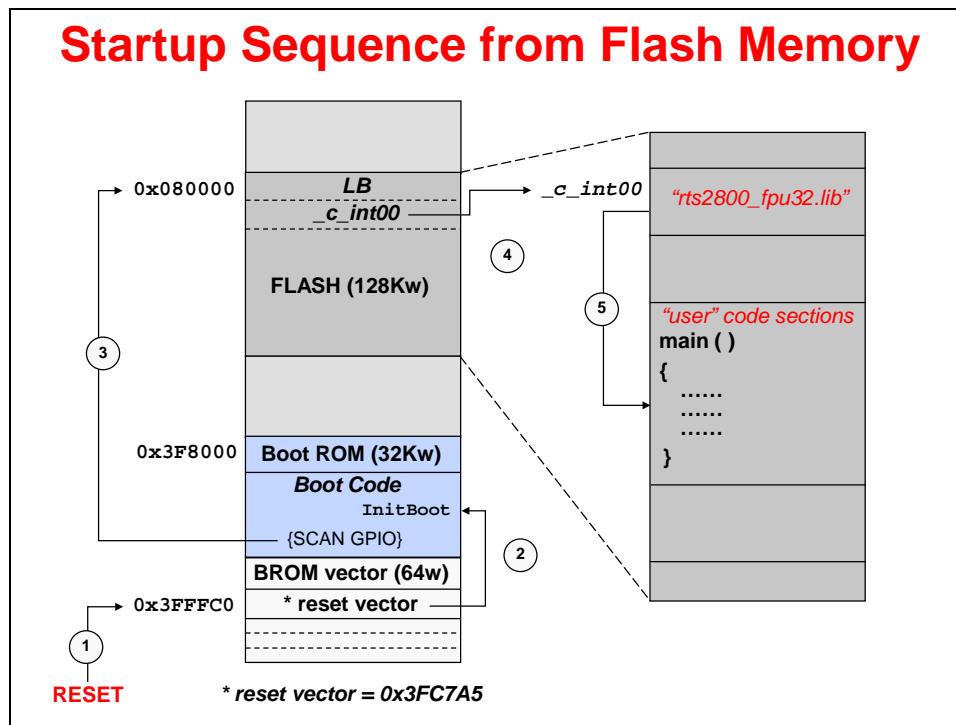
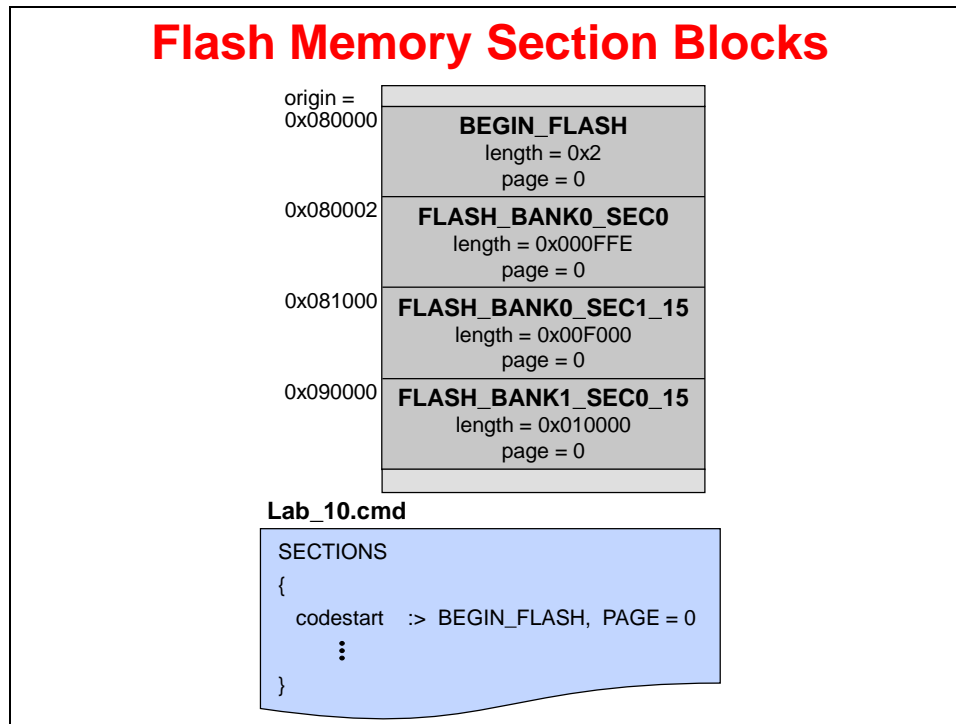
Recall that if the device is in stand-alone boot mode, the state of GPIO24 and GPIO32 pins are used to determine the boot mode. On the LaunchPad switch SW2 controls the boot options for the F280049C device. Check that switch SW2 positions 1 and 2 are set to the default “1” (printed on the board; both switches are towards the MCU). This will configure the device (in stand-alone boot mode) to boot mode flash. Since the Z1-OTP-BOOT registers have not been programmed, the default will be boot from flash. Details of the switch positions can be found in the LaunchPad User’s Guide.

22. Close Code Composer Studio.
23. Disconnect the USB cable from the LaunchPad (i.e. remove power from the LaunchPad).

24. Re-connect the USB cable to the LaunchPad (i.e. power the LaunchPad). The LED should be blinking, showing that the code is now running from flash memory.

End of Exercise

Lab 10 Reference: Programming the Flash



Introduction

The TMS320F28004x contains features that allow for several methods of communication and data exchange between the MCU 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 the Technical Reference Manual, 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)**
- ◆ **Local Interconnect Network (LIN)**
- ◆ **Inter-Integrated Circuit (I2C)**
- ◆ **Controller Area Network (CAN)**
- ◆ **Power Management Bus (PMBus)**
- ◆ **Fast Serial Interface (FSI)**

The F28004x MCU includes numerous communications peripherals that extend the connectivity of the device. These communications peripherals include Serial Peripheral Interface (SPI), Serial Communication Interface (SCI), Local Interconnect Network (LIN), Inter-Integrated Circuit (I2C), Controller Area Network (CAN), Power Management Bus (PMBus), and Fast Serial Interface (FSI).

Chapter Topics

Communications	11-1
<i>Communications Techniques</i>	11-3
<i>Serial Peripheral Interface (SPI)</i>	11-4
SPI Summary	11-6
<i>Serial Communications Interface (SCI)</i>	11-7
Multiprocessor Wake-Up Modes	11-9
SCI Summary	11-11
<i>Local Interconnect Network (LIN)</i>	11-12
LIN Message Frame and Data Timing	11-13
LIN Summary	11-14
<i>Inter-Integrated Circuit (I2C)</i>	11-15
I2C Operating Modes and Data Formats	11-16
I2C Summary	11-17
<i>Controller Area Network (CAN)</i>	11-18
CAN Bus and Node	11-19
Principles of Operation	11-20
Message Format and Block Diagram	11-21
CAN Summary	11-22
<i>Power Management Bus (PMBus)</i>	11-23
Conceptual Block Diagram and Connections	11-23
PMBus Summary	11-24
<i>Fast Serial Interface (FSI)</i>	11-25
CPU Interface and Connection	11-26
FSI Summary	11-27
<i>Lab 11: C2000Ware SCI Echoback Example</i>	11-28

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 module 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 C2000 device family has both synchronous and asynchronous serial ports. The features and operation will be described in this module.

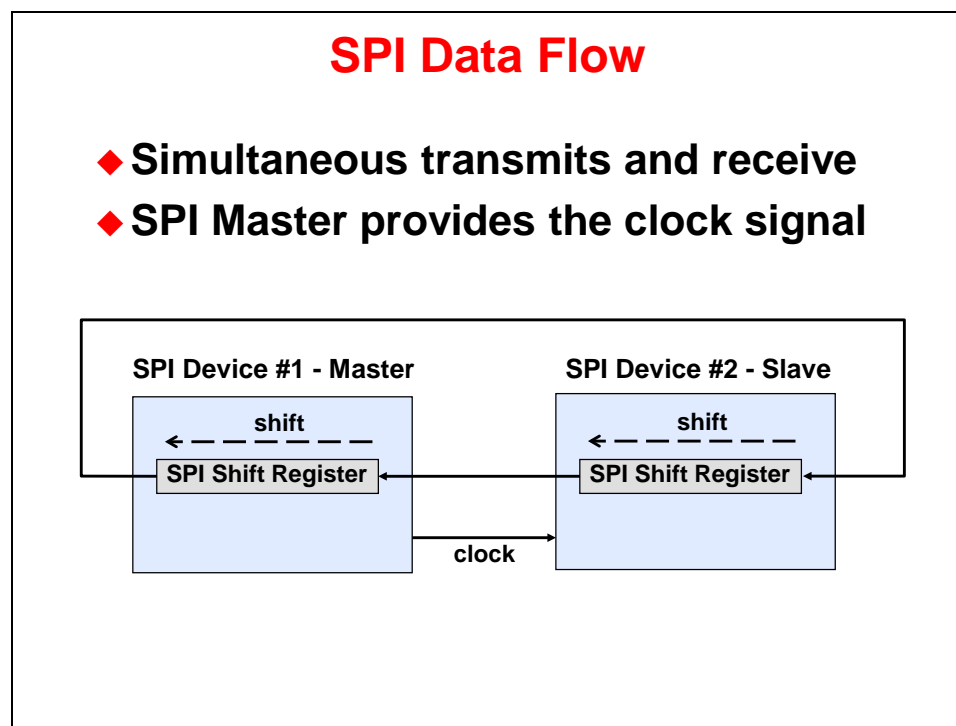
Serial Peripheral Interface (SPI)

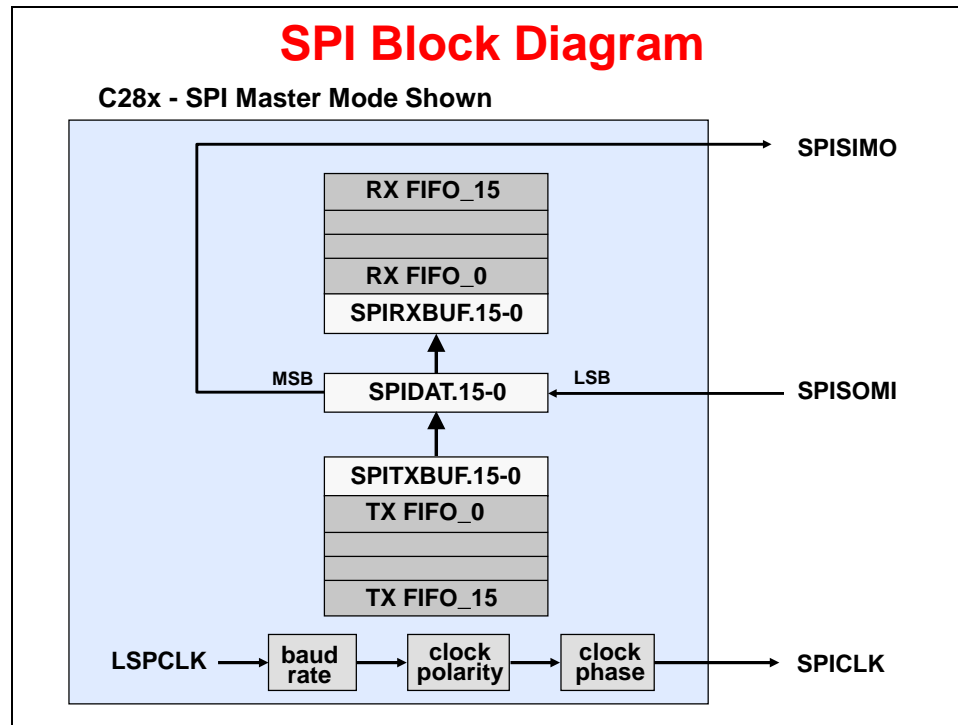
The SPI is a high-speed synchronous serial port that shifts a programmable length serial bit stream into and out of the device at a programmable bit-transfer rate. It is typically used for communications between processors and external peripherals, and it has a 16-level deep receive and transmit FIFO for reducing servicing overhead. 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, SLAVE sends 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

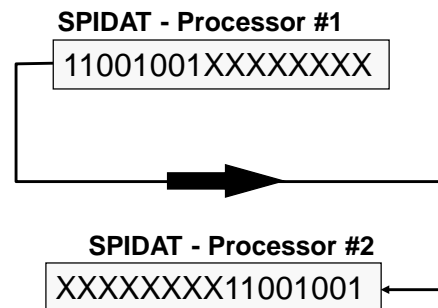
1. Slave writes data to be sent to its shift register (SPIDAT)
2. Master writes data to be sent to its shift register (SPIDAT or SPITXBUF)
3. Completing Step 2 automatically starts SPICLK signal of the Master
4. MSB of the Master's shift register (SPIDAT) is shifted out, and LSB of the Slave's shift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

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

SPI Data Character Justification

- ◆ Programmable data length of 1 to 16 bits
- ◆ Transmitted data of less than 16 bits must be left justified
 - ◆ MSB transmitted first
- ◆ Received data of less than 16 bits are right justified
- ◆ User software must mask-off unused MSB's



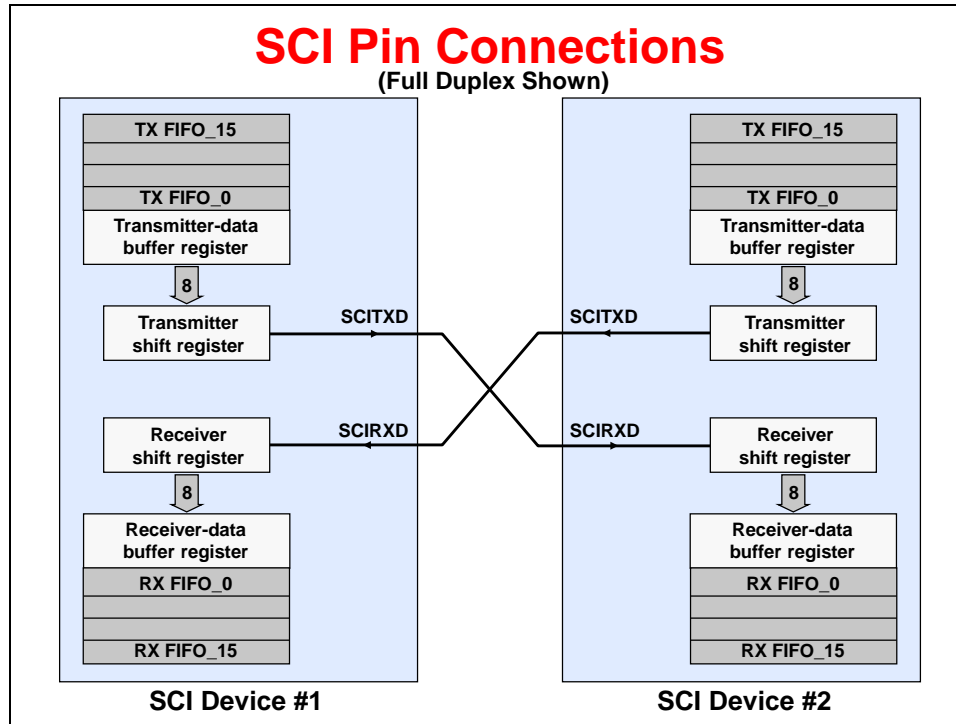
SPI Summary

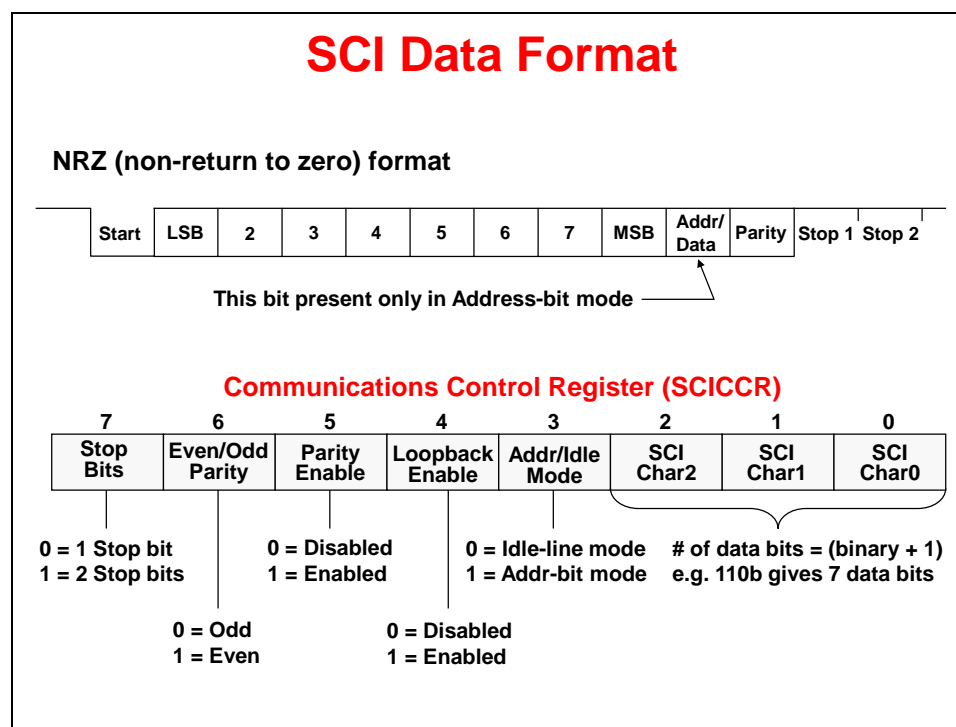
SPI Summary

- ◆ 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 is a two-wire asynchronous serial port (also known as a UART) that supports communications between the processor and other asynchronous peripherals that use the standard non-return-to-zero (NRZ) format. A receiver and transmitter 16-level deep FIFO is used to reduce servicing overhead. 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.





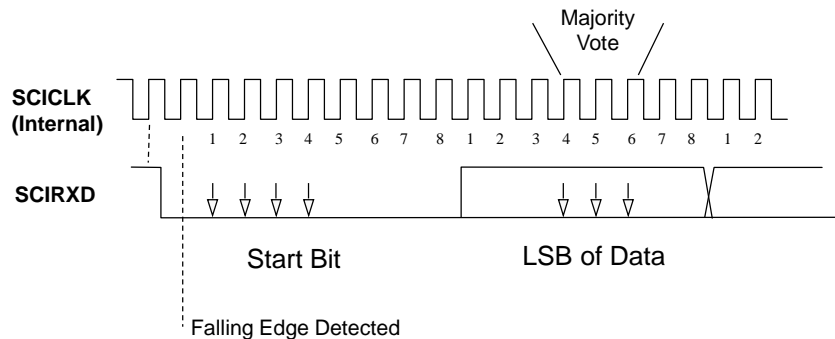
The basic unit of data is called a **character** and is 1 to 8 bits in length. Each character of data is formatted with a start bit, 1 or 2 stop bits, an optional parity bit, and an optional address/data bit. A character of data along with its formatting bits is called a **frame**. Frames are organized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



Note: 8 SCICLK periods per data bit

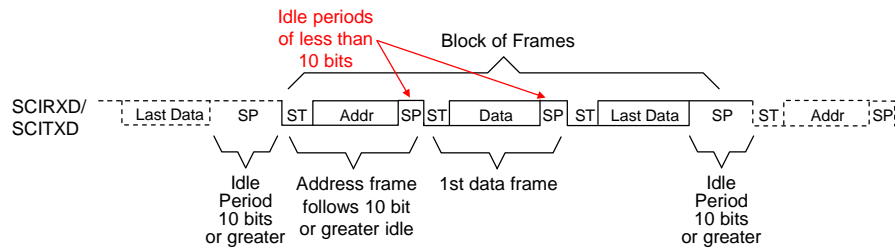
Multiprocessor Wake-Up Modes

Multiprocessor Wake-Up Modes

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

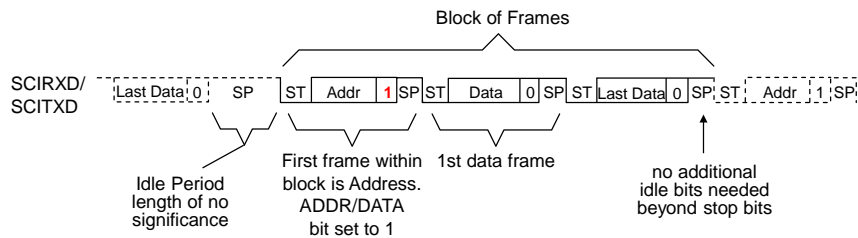
Idle-Line Wake-Up Mode

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



Address-Bit Wake-Up Mode

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



The SCI interrupt logic generates interrupt flags when it receives or transmits a complete character as determined by the SCI character length. This provides a convenient and efficient way of timing and controlling the operation of the SCI transmitter and receiver. The interrupt flag for the transmitter is TXRDY (SCICTL2.7), and for the receiver RXRDY (SCIRXST.6). TXRDY is

set when a character is transferred to TXSHF and SCITXBUF is ready to receive the next character. In addition, when both the SCIBUF and TXSHF registers are empty, the TX EMPTY flag (SCICTL2.6) is set. When a new character has been received and shifted into SCIRXBUF, the RXRDY flag is set. In addition, the BRKDT flag is set if a break condition occurs. A break condition is where the SCIRXD line remains continuously low for at least ten bits, beginning after a missing stop bit. Each of the above flags can be polled by the CPU to control SCI operations, or interrupts associated with the flags can be enabled by setting the RX/BK INT ENA (SCICTL2.1) and/or the TX INT ENA (SCICTL2.0) bits active high.

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

SCI Summary

SCI Summary

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

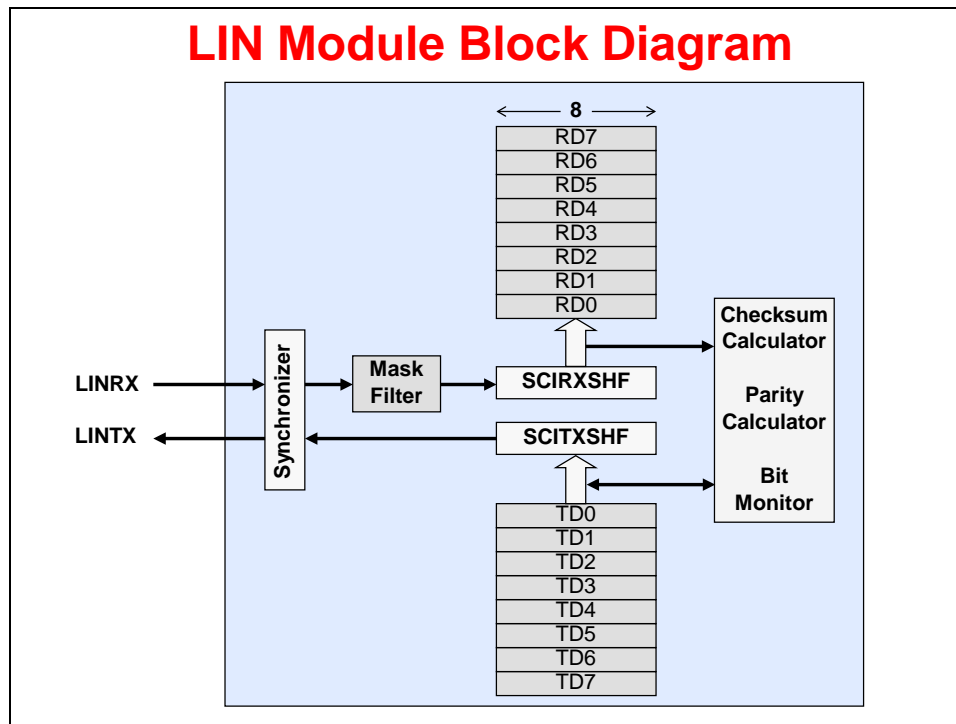
Local Interconnect Network (LIN)

Local Interconnect Network (LIN)

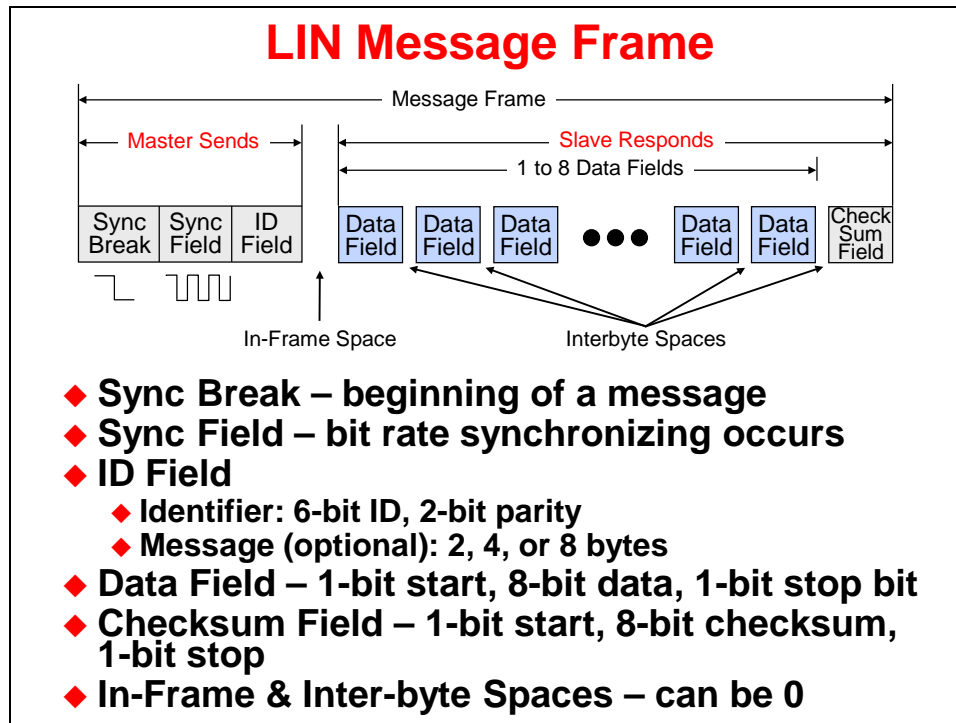
- ◆ **A broadcast serial network**
 - ◆ One master, up to sixteen addressable slaves
 - ◆ Serial link layer similar to UART (e.g., start, data, stop bits)
 - ◆ Single wire (plus ground)
 - ◆ 12V bus (originally designed for automotive apps)
 - ◆ No bus arbitration or collision detection
 - ◆ Master initiates all communication
 - ◆ A single slave responds
 - ◆ Configurable Baud Rate up to 20 Kbits/s
- ◆ **C2000 LIN module**
 - ◆ Compliant with the LIN spec 2.1
 - ◆ Can be used as an SCI (UART), if desired

The LIN standard is based on the SCI (UART) serial data link format. The communication concept is single-master/multiple-slave with a message identification for multi-cast transmission between any network nodes.

LIN Module Block Diagram

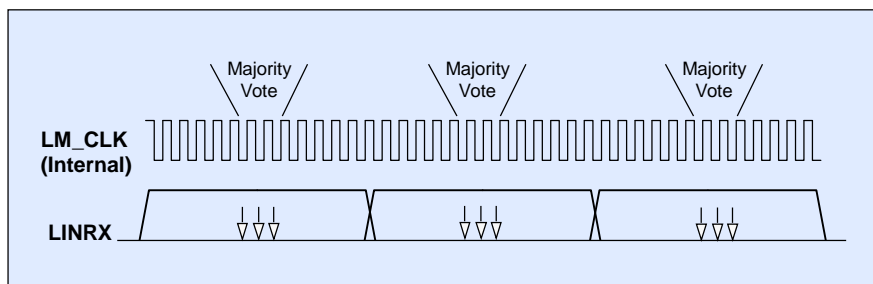


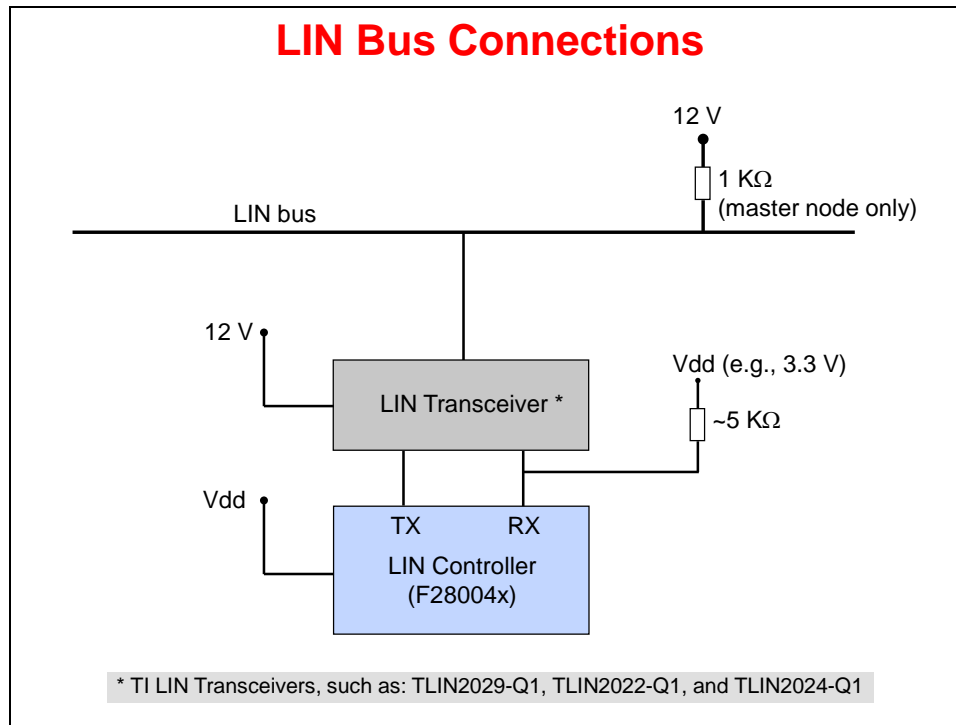
LIN Message Frame and Data Timing



LIN Data Timing

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





LIN Summary

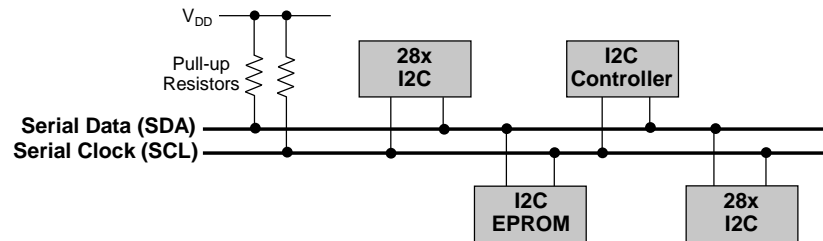
LIN Summary

- ◆ Functionally compatible with standalone SCI of C28x devices
- ◆ Identification masks for filtering
- ◆ Automatic master header generation
- ◆ 2^{31} programmable transmission rates with 7 fractional bits
- ◆ Automatic wakeup support
- ◆ Error detection (bit, bus, no response, checksum, synchronization, parity)
- ◆ Multi-buffered receive/transmit units

Inter-Integrated Circuit (I2C)

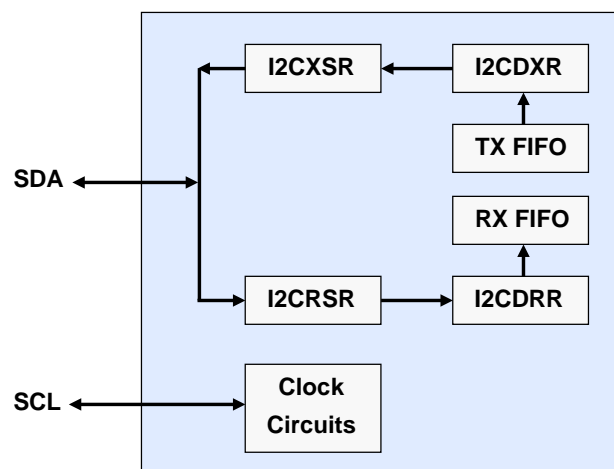
Inter-Integrated Circuit (I2C)

- ◆ NXP Semiconductors 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)



The I2C provides an interface between devices that are compliant I2C-bus specification version 2.1 and connect using an I2C-bus. External components attached to the 2-wire serial bus can transmit or receive 1 to 8-bit data to or from the device through the I2C module.

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

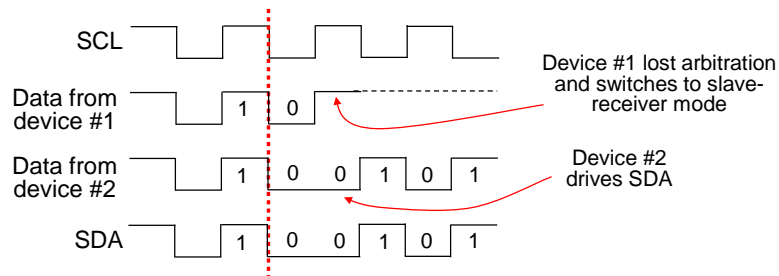
n = 1 to 8 bits

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

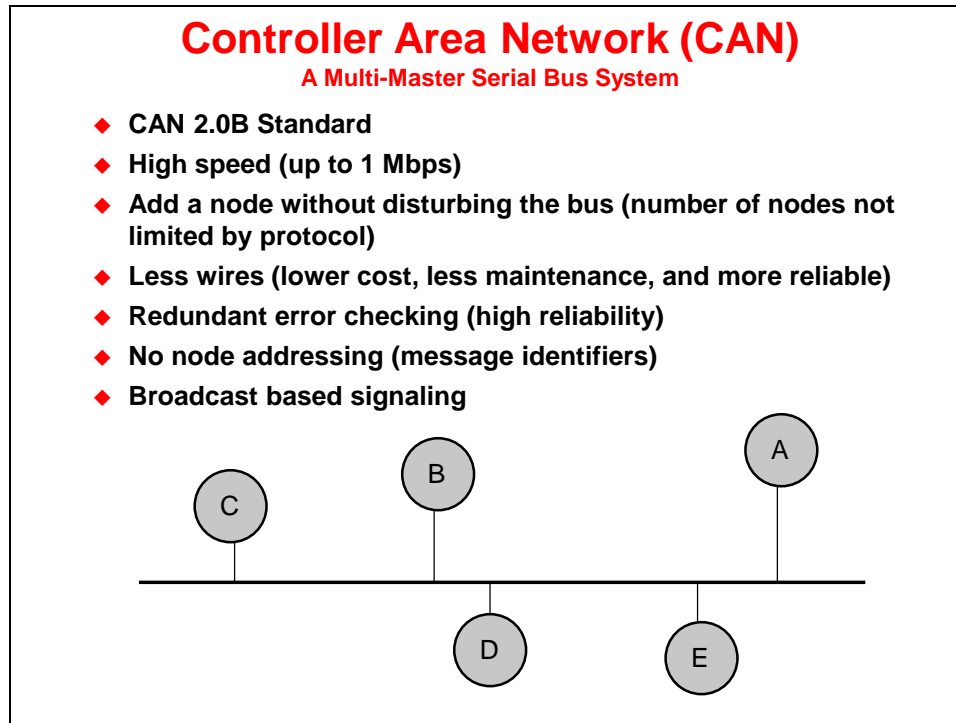


I2C Summary

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**

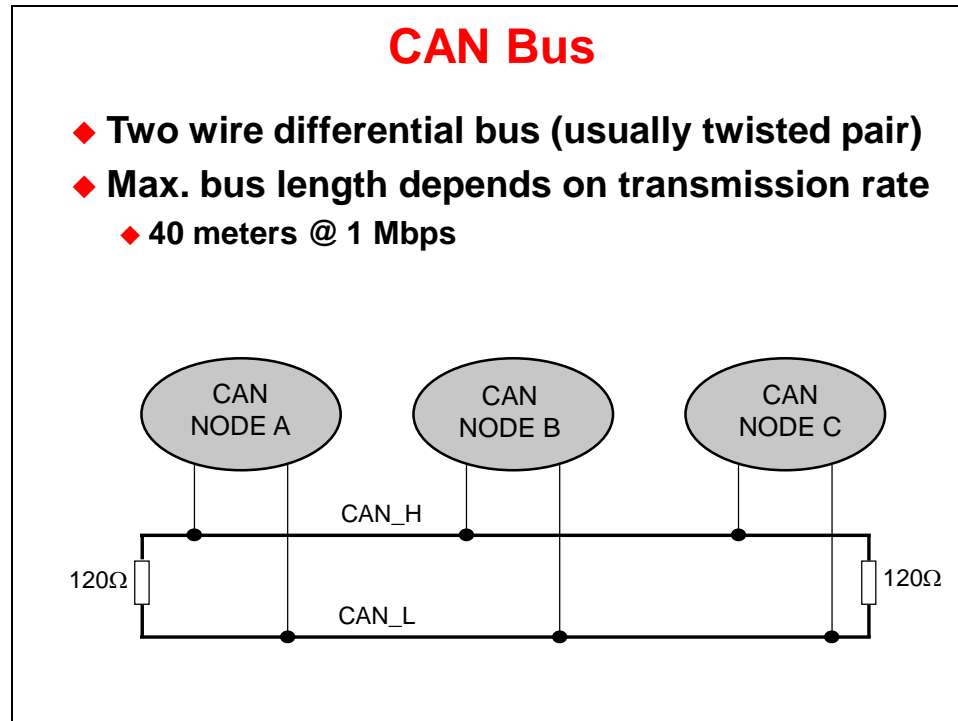
Controller Area Network (CAN)



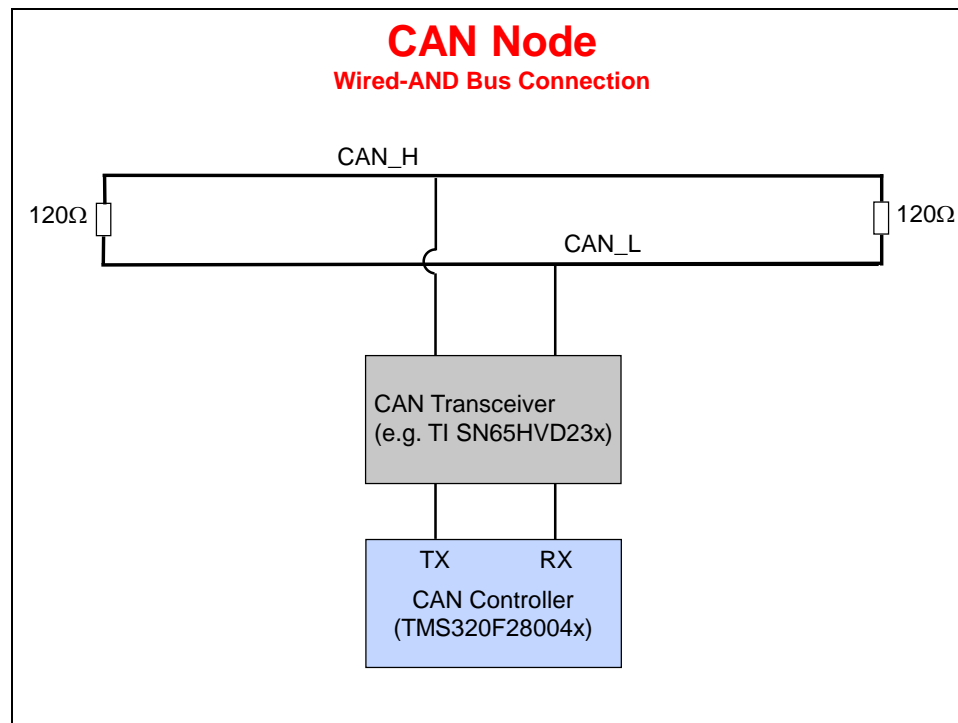
The CAN module is a serial communications protocol that efficiently supports distributed real-time control with a high level of security. It supports bit-rates up to 1 Mbit/s and is compliant with the CAN 2.0B protocol specification.

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 can be up to 1 Mbit/second.



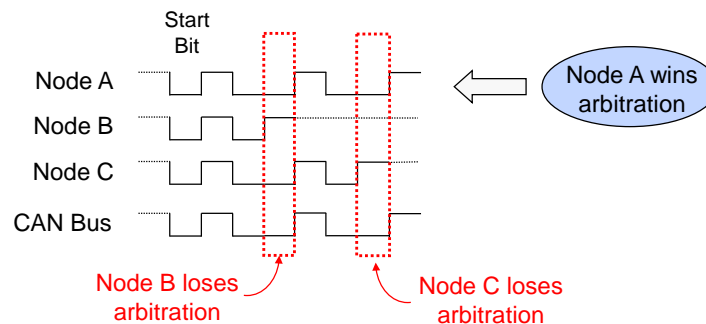
Principles of Operation

Principles of Operation

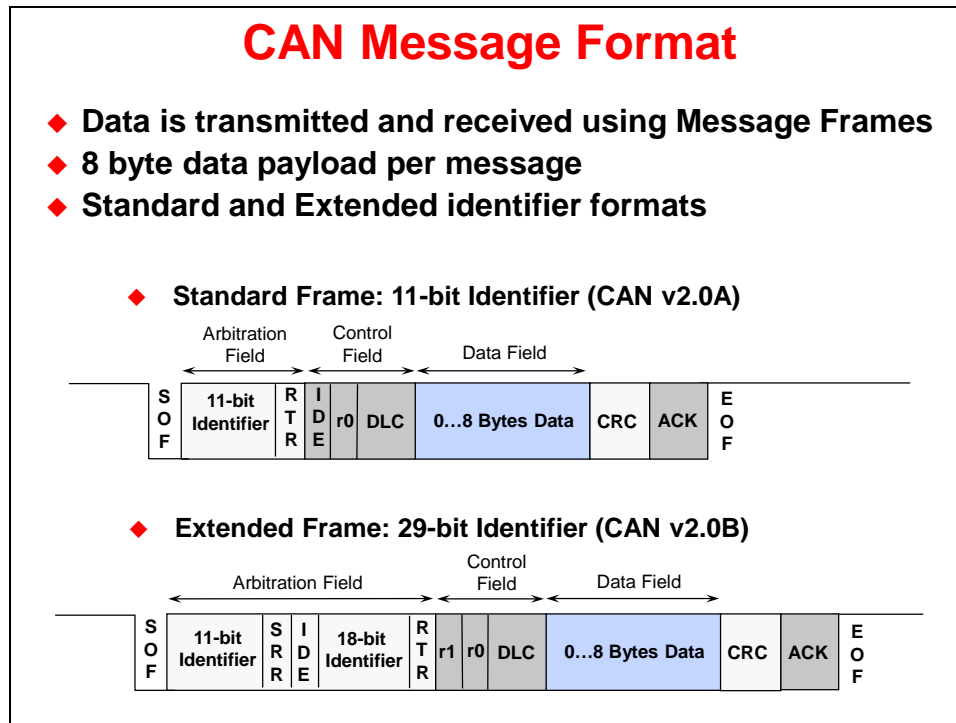
- ◆ Data messages transmitted are identifier based, not address based
- ◆ Content of message is labeled by an identifier that is unique throughout the network
 - ◆ (e.g. rpm, temperature, position, pressure, etc.)
- ◆ All nodes on network receive the message and each performs an acceptance test on the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

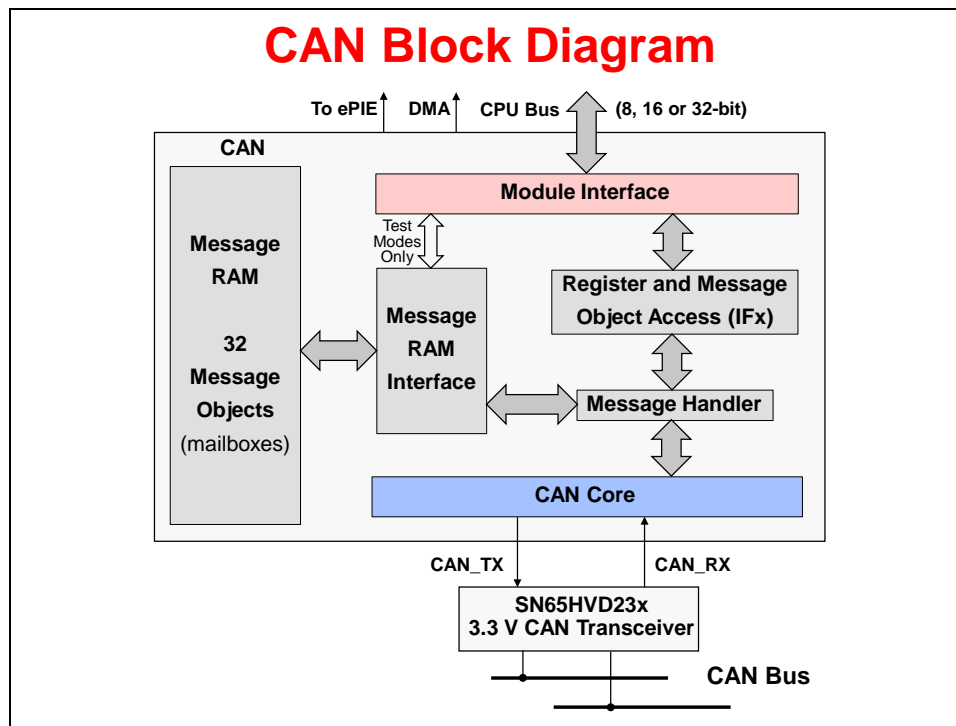
- ◆ Bus arbitration resolved via arbitration with wired-AND bus connections
 - ◆ Dominate state (logic 0, bus is high)
 - ◆ Recessive state (logic 1, bus is low)



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:

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

CAN Summary

CAN Summary

- ◆ **Fully compliant with CAN standard v2.0B**
- ◆ **Supports data rates up to 1 Mbps**
- ◆ **Thirty-two message objects**
 - ◆ **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**
- ◆ **Two interrupt lines**
- ◆ **Self-test mode**

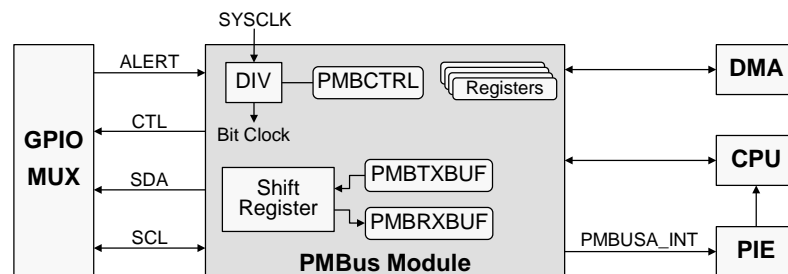
Power Management Bus (PMBus)

Power Management Bus (PMBus)

- ◆ Provides an interface between the MCU and devices compliant with the SMI Forum PMBus Specification:
 - ◆ Part I version 1.0 and Part II version 1.1
 - ◆ Enables a standard 2-wire communications protocol between power supply components
- ◆ Based on SMBus and supports I2C mode
 - ◆ uses a similar physical layer to I2C
- ◆ Support for master and slave modes
- ◆ Support for two speeds:
 - ◆ Standard Mode: Up to 100 kHz
 - ◆ Fast Mode: 400 kHz
- ◆ Four-byte transmit and receive buffers
- ◆ Packet error checking (PEC)

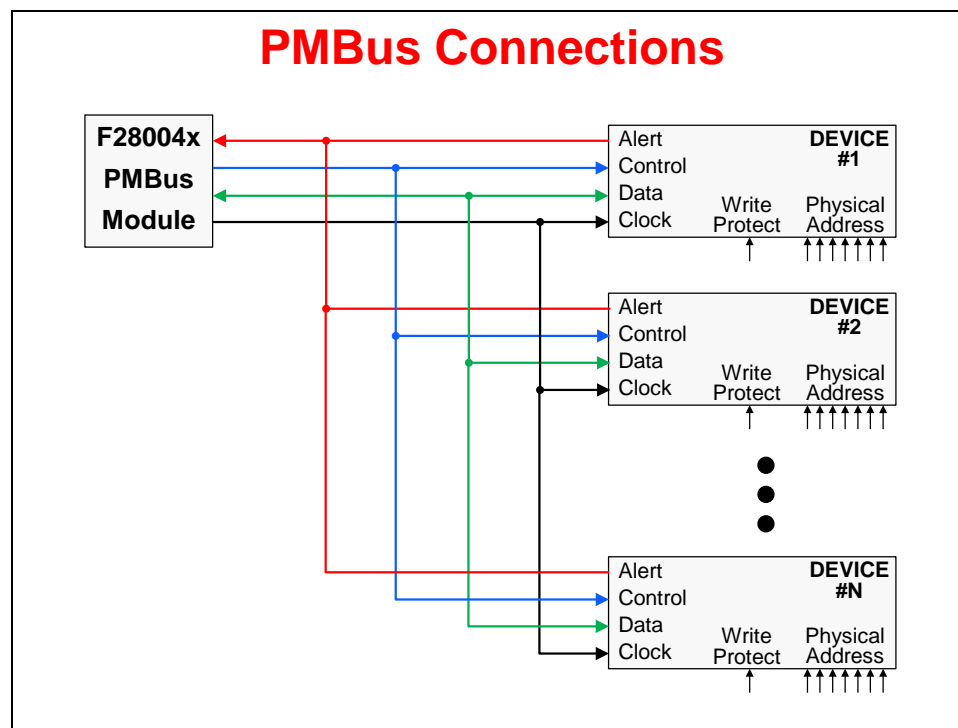
Conceptual Block Diagram and Connections

Conceptual Block Diagram

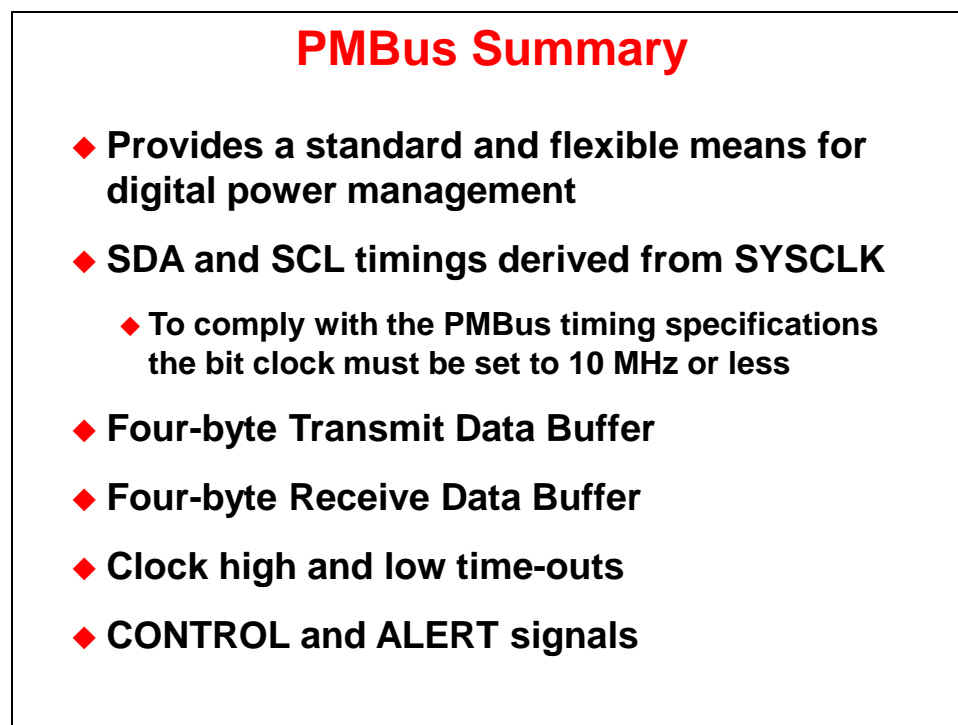


- ◆ **SCL is the bus clock**
 - ◆ Normally controlled by the master; can be held low by a slave to delay a transaction (to allow more time for processing)
- ◆ **SDA is the bidirectional data line**
- ◆ **CONTROL is a slave input that can trigger an interrupt**
 - ◆ Can be used to tell a slave device to shut down
- ◆ **ALERT is a slave output/master input**
 - ◆ Allows a slave to request attention from the master

The PMBus module provides an interface for communicating between the microcontroller and other devices that are compliant with the System Management Interface (SMI) specification. PMBus is an open-standard digital power management protocol that enables communication between components of a power system.



PMBus Summary



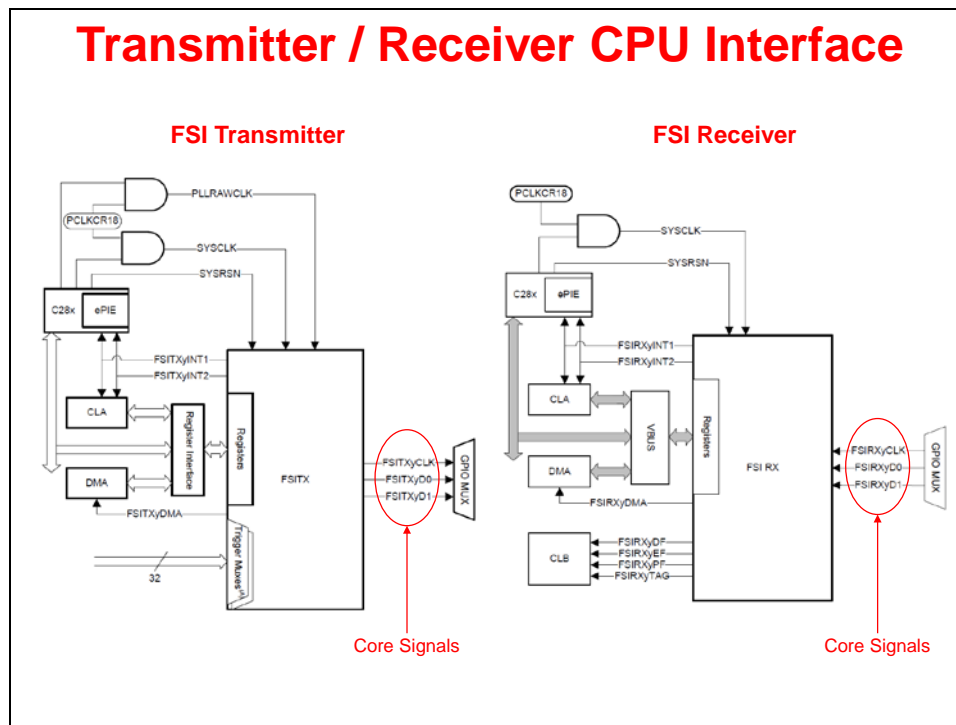
Fast Serial Interface (FSI)

Fast Serial Interface (FSI)

- ◆ **Ensure reliable high-speed serial communication across an isolation barrier**
 - ◆ Provides galvanic isolation where different circuits do not have common power and ground connections
- ◆ **Consists of independent transmitter (FSITX) and receiver (FSIRX) cores**
 - ◆ Each cores is configured and operated independently
- ◆ **Point-to-point (single master/single slave)**
- ◆ **Fast transfer: 50 MHz**
- ◆ **Dual data rate (100 Mbps @ 50 MHz clock)**
- ◆ **Single or dual data lines**
- ◆ **Programmable data length**
- ◆ **Hardware- or software-calculated CRC**
- ◆ **Frame error detection**
- ◆ **Two interrupts per FSI core**

The FSI module is a highly reliable high-speed serial communication peripheral capable of operating at dual data rate providing 100 Mbps transfer using a 50 MHz clock. The FSI consists of independent transmitter and receiver cores that are configured and operated independently. FSI is a point-to-point communication protocol operating in a single-master/single-slave configuration. With this high-speed data rate and low channel count, the FSI can be used to increase the amount of information transmitted and reduce the costs to communicate over an isolation barrier.

CPU Interface and Connection



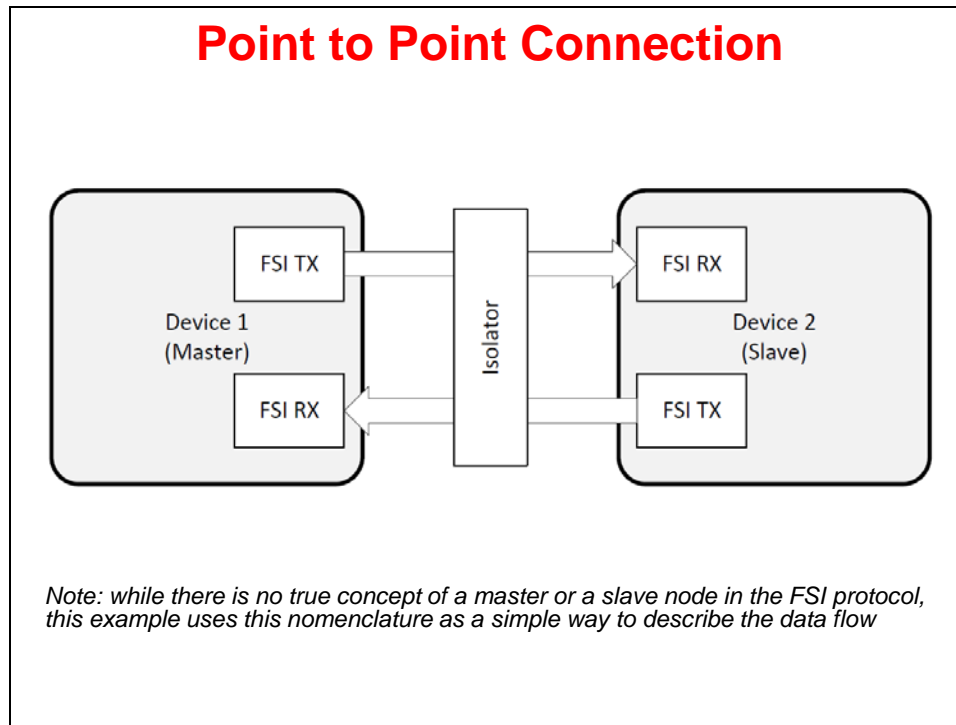
Transmitter / Receiver Core Signals

Transmitter Core Signal

- ◆ **TXCLK – Transmit clock**
- ◆ **TXD0 – Primary data output line for transmission**
 - ◆ For multi-lane transmission:
 - ◆ Contains all the even numbered bits of the data and CRC bytes
 - ◆ Other frame fields will be transmitted in full
- ◆ **TXD1 – Additional data output line for transmission**
 - ◆ Configured for multi-lane transmission:
 - ◆ Contain all the odd numbered bits of the data and CRC bytes
 - ◆ Applies only to the data words and the CRC bytes

Receiver Core Signal

- ◆ **RXCLK – Receive clock**
 - ◆ Connected to the TXCLK of the transmitting FSI module
- ◆ **RXD0 – Primary data input line for reception**
 - ◆ Connected to the TXD0 of the transmitting FSI module
- ◆ **RXD1 – Additional data input line for reception**
 - ◆ Connected to the TXD1 of the transmitting FSI module if multi-lane transmission is used



FSI Summary

FSI Summary

- ◆ **Highly reliable high-speed serial peripheral for communicating over an isolation barrier**
- ◆ **High-speed data rate and low channel count**
 - ◆ **Increases the amount of information transmitted and reduce the costs**
- ◆ **Separate transmit and receive modules**
- ◆ **Point-to-point communication protocol**
 - ◆ **FSI transmitter core communicates directly to a single FSI receiver core**
- ◆ **Skew compensation for signal delay due to isolation**
- ◆ **Line break detection**

Lab 11: C2000Ware SCI Echoback Example

➤ Objective

The objective of this lab exercise is to learn how to import and run a project from C2000Ware. In the previous lab exercises, a local copy of the required C2000Ware files was included with the lab files. This provided portability and made the workshop files self-contained and independent of other support files or resources. It is assumed that Code Composer Studio (CCS) is already installed; however, the installation of C2000Ware will be required and included in the procedure directions.

For this lab exercise, the F28004x Driver Library (Driverlib) SCI echoback example will be used. The SCI echoback example receives and returns data through the SCI-A port. The CCS terminal feature will be used to view the data from the SCI and to send characters to the SCI. Each character that is received by the SCI port is sent back to the host. The program will print out a greeting and then ask you to enter a character which it will echo back to the terminal. Also, a watch variable 'loop count' is included to count the number of characters sent.

The SCI-A port on the F280049C LaunchPad will communicate to the host PC using the USB connection. The XDS110 debug probe enumerates as both a debugger and a virtual COM port. By default, SCI-A is mapped to the virtual COM port of the XDS110 debug probe using GPIO28 and GPIO29. (Alternatively, SCI-A can be connected to a host PC using an external connection via a transceiver and cable).

➤ Procedure

Download and Install C2000Ware

1. Download C2000Ware from <http://www.ti.com/tool/c2000ware>. (Also, C2000Ware can be downloaded using the CCS Resource Explorer).
2. Install C2000Ware using the default location (i.e. C:\ti\c2000\).

Import the Project

3. Import 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:\ti\c2000\C2000Ware_<version>\driverlib\f28004x\examples\sci
```

and click **Select Folder**.

4. In the "Discovered projects" window that opens select **sci_ex1_echoback** and then click **Finish** to import the project.

Modify the Target Configuration File

The F28004x Driverlib example projects include a target configuration file within each project. This target configuration has been setup to use the XDS100v2 standard JTAG mode. Recall that the F280049C LaunchPad XDS110 USB Debug Probe is only wired to support 2-pin cJTAG mode. Therefore, before using the LaunchPad with these examples, the target configuration file needs to be reconfigured.

5. The `sci_ex1_echoback` project should now appear in the Project Explorer window. Expand the project and open the 'targetConfigs' folder. Rename the file name from `TMS320F280049M.ccxml` to `TMS320F280049C.ccxml` (i.e. change from "M" to "C"). Next, double-left click on the `TMS320F280049C.ccxml` file.
6. In the window that opens, select the emulator using the "Connection" pull-down list and choose "Texas Instruments XDS110 USB Debug Probe". In the "Board or Device" box type **TMS320F280049C** to filter the options. In the box below, check the box to select "TMS320F280049C".
7. Under Advanced Setup click "Target Configuration" and highlight "Texas Instruments XDS110 USB Debug Probe_0". Under Connection Properties set the JTAG/SWD/cJTAG Mode to "cJTAG (1149.7) 2-pin advanced modes".

Click `Save` to save the configuration, then close the "TMS320F280049C.ccxml" setup window by clicking the `X` on the tab.

Inspect `sci_ex1_echoback.c`

8. Open and inspect `sci_ex1_echoback.c`. Notice that code lines 139 through 148 use the Driverlib functions to configure SCI-A.

```

139  SCI_setConfig(SCIA_BASE, DEVICE_LSPCLK_FREQ, 9600, (SCI_CONFIG_WLEN_8 |
140                                                     SCI_CONFIG_STOP_ONE |
141                                                     SCI_CONFIG_PAR_NONE));
142  SCI_resetChannels(SCIA_BASE);
143  SCI_resetRxFIFO(SCIA_BASE);
144  SCI_resetTxFIFO(SCIA_BASE);
145  SCI_clearInterruptStatus(SCIA_BASE, SCI_INT_TXFF | SCI_INT_RXFF);
146  SCI_enableFIFO(SCIA_BASE);
147  SCI_enableModule(SCIA_BASE);
148  SCI_performSoftwareReset(SCIA_BASE);

```

Next, code lines 161 through 169 displays the greeting on the terminal and waits for a character to be entered. Code lines 174 and 179 through 181 reads a character and writes it back to the terminal. Finally, code line 186 increments the loop counter.

```


161  msg = "\r\n\r\nHello World!\0";
162  SCI_writeCharArray(SCIA_BASE, (uint16_t*)msg, 17);
163  msg = "\r\nYou will enter a character, and the DSP will echo it back!\n\0";
164  SCI_writeCharArray(SCIA_BASE, (uint16_t*)msg, 62);
165
166  for(;;)
167  {
168      msg = "\r\nEnter a character: \0";
169      SCI_writeCharArray(SCIA_BASE, (uint16_t*)msg, 22);
170
171      //
172      // Read a character from the FIFO.
173      //
174      receivedChar = SCI_readCharBlockingFIFO(SCIA_BASE);
175
176      //
177      // Echo back the character.
178      //
179      msg = " You sent: \0";
180      SCI_writeCharArray(SCIA_BASE, (uint16_t*)msg, 13);
181      SCI_writeCharBlockingFIFO(SCIA_BASE, receivedChar);
182
183      //
184      // Increment the loop count variable.
185      //
186      loopCounter++;
187  }

```

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_RAM` using the Scripts menu.

Configure a CCS Terminal

11. To view the terminal information on your PC, we first need to determine the COM port associated with the F280049C LaunchPad. To do this in Windows 10, right click on ‘This PC’ (or ‘My Computer’ in earlier versions of Windows) and select Properties. In the dialog box that appears, click on Device Manager (or the Hardware tab and open Device Manager). Alternatively, you can open the Windows Search and type “Device Manager”. Then look for an entry under Ports (COM & LPT) titled “User UART (COMX)” or “USB Serial Port (COMX)”, where X is a number. Remember this number for the next step where we will open and configure a serial terminal.
12. Open a CCS terminal by clicking the Terminal button  or clicking:

View → Terminal

13. A Launch Terminal window will open. Configure the terminal with the following settings:
 - Choose terminal: Serial Terminal
 - Serial Port: (from above COM port number X)
 - Baud rate: 9600
 - Data size: 8
 - Parity: None
 - Stop bits: 1
 - Encoding: Default (ISO-8859-1)

Click `OK` to launch the terminal window.

Run the Code

14. In `sci_ex1_echoback.c` towards the top of the code highlight the global variable ‘loopCounter’ with the mouse, right click and select “Add Watch Expression...” and then select `OK`. The global variable `loopCounter` should now be in the Expressions window with a value of “0”. This global variable will be used to count the number of characters sent.
15. Enable the Expressions window for continuous refresh.
16. Run the code (using the `Resume` button on the toolbar). The program will print out a greeting and then ask you to enter a character which it will echo back to the terminal.

```
Hello World!  
You will enter a character, and the DSP will echo it back!  
Enter a character:
```

17. Enter a character and the character will echo back to the terminal. Also, the `loopCounter` will increment with each character sent.
18. Halt the code (using the `Suspend` button on the toolbar).

Terminate Debug Session and Close Project

19. Terminate the active debug session using the `Terminate` button. This will close the debugger and return Code Composer Studio to the CCS Edit perspective view.
20. Next, close the project by right-clicking on `sci_ex1_echoback` in the Project Explorer window and select `Close Project`.

End of Exercise

Support Resources

Introduction

This module contains various references to support the development process.

Module Objectives

Module Objectives

- ◆ **C2000 MCU Device Workshops Website**
- ◆ **Documentation Resources**
- ◆ **C2000Ware™**
- ◆ **TI Development Tools**
- ◆ **Additional Resources**
 - ◆ **Product Information Center**
 - ◆ **On-line support**

Chapter Topics

Support Resources	12-1
<i>TI Support Resources</i>	<i>12-3</i>
C2000 MCU Device Workshops Website	12-3
Documentation Resources	12-4
C2000Ware™	12-4
C2000 Experimenter's Kit	12-5
F28335 Peripheral Explorer Kit	12-6
C2000 LaunchPad Evaluation Kit	12-7
C2000 controlCARD Application Kits	12-8
XDS100/110/200 Class JTAG Debug Probes	12-9
Product Information Resources	12-10

TI Support Resources

C2000 MCU Device Workshops Website

C2000 MCU Device Workshops Website

TI Training & Videos >
Applications & designs
Live training
Products
Tools & software
Feedback
Global

C2000™ MCU Device Workshops Email

The C2000 Microcontroller (MCU) Workshops have been developed to help engineers gain a full understanding and complete working knowledge of the C2000 MCU family. Learning is accomplished through a detailed workshop manual and by performing the hands-on lab exercises. Each workshop starts with the basic concepts and progresses to more advanced topics in a logical flow. The topics and lab exercises build on the previous one completed, running a common theme throughout the workshop. All workshops guide the user through an architectural overview, the programming development environment, system initialization, peripheral configuration, and programming an application into flash memory using the Code Composer Studio on-chip flash programmer. The workshops are ideal for both beginner and advanced users.

Table of contents

1. C2000 MCU Device Workshops

1. C2000 MCU Device Workshops

Below are the available C2000 MCU workshops:

#	🔒	Title	Duration	Overview
1.1		C2000™ F28004x Microcontroller Workshop		The C2000 TMS320F28004x Microcontroller Workshop is a hands-on technical course based on the TMS320F280049C device.
1.2		C2000™ F2837xD Microcontroller Workshop		The C2000 TMS320F2837xD Microcontroller Workshop is a hands-on technical course based on the TMS320F28379D device.
1.3		C2000™ F2806x Microcontroller Workshop		The C2000 TMS320F2806x Microcontroller Workshop is a hands-on technical course based on the TMS320F28069 device.
1.4		C2000™ F2803x Microcontroller Workshop		The C2000 TMS320F2803x Microcontroller Workshop is a hands-on technical course based on the TMS320F28035 device.
1.5		C2000™ F2802x Microcontroller Workshop		The C2000 TMS320F2802x Microcontroller Workshop is a hands-on technical course based on the TMS320F28027 device.
1.6		C2000™ F2833x Microcontroller Workshop		The C2000 TMS320F2833x Microcontroller Workshop is a hands-on technical course based on the TMS320F28335 device.
1.7		C2000™ F280x Microcontroller Workshop		The C2000 TMS320F280x Microcontroller Workshop is a hands-on technical course based on the TMS320F2806 device.
1.8		C2000™ F281x Microcontroller Workshop		The C2000 TMS320F281x Microcontroller Workshop is a hands-on technical course based on the TMS320F2812 device.
1.9		C2000™ F28M35x Microcontroller Workshop		The C2000 F28M35x Microcontroller Workshop is a hands-on technical course based on the F28M35H52C1 device.
1.10		C2000™ LF240x Microcontroller Workshop		The C2000 TMS320LF240x Microcontroller Workshop is a hands-on technical course based on the TMS320LF2407A device.

Additional information

● Learn more about C2000 MCUs

<https://training.ti.com/c2000-mcu-device-workshops>

At the TI Training Portal you will find all of the materials for the C2000 Microcontroller Workshops, which include support for the following device families:

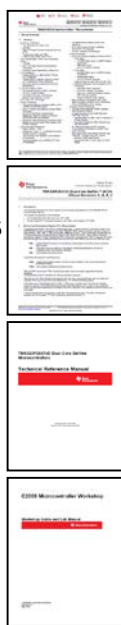
- TMS320F28004x
- TMS320F2837xD
- TMS320F2806x
- TMS320F2803x
- TMS320F2802x
- TMS320F2833x
- TMS320F280x
- TMS320F281x
- TMS320LF240x
- F28M35x

Documentation Resources

Documentation Resources

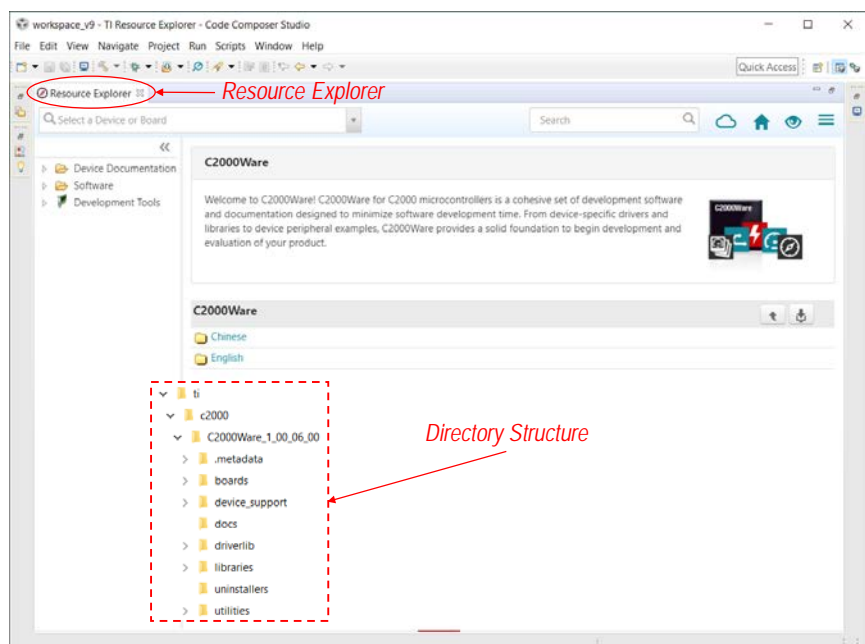
- ◆ **Data Sheet**
 - ◆ Contains device electrical characteristics and timing specifications
 - ◆ Key document for hardware engineers
- ◆ **Silicon Errata**
 - ◆ Contains deviations from original specifications
 - ◆ Includes silicon revision history
- ◆ **Technical Reference Manual (TRM)**
 - ◆ Contains architectural descriptions and register/bit definitions
 - ◆ Key document for firmware engineers
- ◆ **Workshop Materials**
 - ◆ Hands-on device training materials
 - ◆ For hardware and software engineers

Documentation resources can be found at www.ti.com/c2000



C2000Ware™

C2000Ware™



The screenshot shows the TI Resource Explorer window with the following structure:


- ti
 - c2000
 - C2000Ware_1.00.06.00
 - .metadata
 - boards
 - device_support
 - docs
 - driverlib
 - libraries
 - uninstallers
 - utilities

C2000Ware for C2000 microcontrollers is a cohesive set of software infrastructure, tools, and documentation that is designed to minimize system development time. It includes device-specific

drivers and support software, as well as system application examples. C2000Ware provides the needed resources for development and evaluation. It can be downloaded from the TI website.

C2000 Experimenter's Kit

C2000 Experimenter Kit




- ◆ **Part Number:**
 - ◆ TMDSDOCK280049C
 - ◆ TMDSDOCK28379D
 - ◆ TMDSDOCK28069
 - ◆ TMDSDOCK28035
 - ◆ TMDSDOCK28027
 - ◆ TMDSDOCK28335
 - ◆ TMDSDOCK2808
 - ◆ TMDSDOCKKH52C1
- JTAG debug probe required for:*
 - ◆ TMDSDOCK28343
 - ◆ TMDSDOCK28346-168
- ◆ **Experimenter Kits include**
 - ◆ controlCARD
 - ◆ USB docking station
 - ◆ C2000 applications software with example code and full hardware details available in C2000Ware
 - ◆ Code Composer Studio (download)
- ◆ **Docking station features**
 - ◆ Access to controlCARD signals
 - ◆ Breadboard areas
 - ◆ On-board USB JTAG debug probe
 - ◆ *JTAG debug probe not required*
- ◆ **controlCARDS are also available separately**
- ◆ **Available through TI authorized distributors and the TI store**

The C2000 Experimenter Kits is a tool for device exploration and initial prototyping. 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. The docking station provides access to all of the controlCARD signals with two prototyping breadboard areas and header pins, allowing for development of custom solutions. Most have on-board USB JTAG emulation capabilities and no external debug probe 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 debug probe.

F28335 Peripheral Explorer Kit

F28335 Peripheral Explorer Kit



- ◆ **Experimenter Kit includes**
 - ◆ F28335 controlCARD
 - ◆ Peripheral Explorer baseboard
 - ◆ C2000 applications software with example code and full hardware details available in C2000Ware
 - ◆ Code Composer Studio (download)
- ◆ **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)
- ◆ **On-board USB JTAG debug probe**
 - ◆ *JTAG debug probe not required*
- ◆ **Available through TI authorized distributors and the TI store**

TMDSPREX28335

The C2000 Peripheral Explorer Kit is a learning tool for new C2000 developers and university students. The kit includes a peripheral explorer board and a controlCARD with the TMS320F28335 microcontroller. The board includes many hardware-based peripheral components for interacting with the various peripherals common to C2000 microcontrollers, such as the ADC, PWMs, eCAP, I2C, CAN, SPI and McBSP. A teaching ROM is provided containing presentation slides, a learning textbook, and laboratory exercises with solutions.

C2000 LaunchPad Evaluation Kit

C2000 LaunchPad Evaluation Kit



- ◆ **Part Number:**
 - ◆ LAUNCHXL-F28027
 - ◆ LAUNCHXL-F28027F
 - ◆ LAUNCHXL-F28069M
 - ◆ LAUNCHXL-F28379D
 - ◆ LAUNCHXL-F280049C
- ◆ **Low-cost evaluation kit**
 - ◆ F28027 and F28379D standard versions
 - ◆ F28027F version with InstaSPIN-FOC
 - ◆ F28069M version with InstaSPIN-MOTION
- ◆ **Various BoosterPacks available**
- ◆ **On-board JTAG debug probe**
 - ◆ *JTAG debug probe not required*
- ◆ **Access to LaunchPad signals**
- ◆ **C2000 applications software with example code and full hardware details in available in C2000Ware**
- ◆ **Code Composer Studio** (download)
- ◆ **Available through TI authorized distributors and the TI store**

The C2000 LaunchPads are low-cost, powerful evaluation platforms which are used to develop real-time control systems based on C2000 microcontrollers. Various LaunchPads are available and developers can find a LaunchPad with the required performance and feature mix for any application. The C2000 BoosterPacks expand the power of the LaunchPads with application-specific plug-in boards, allowing developers to design full solutions using a LaunchPad and BoosterPack combination.

C2000 controlCARD Application Kits

C2000 controlCARD Application Kits



- ◆ **Developer's Kit for – *Motor Control, Digital Power, etc. applications***
- ◆ **Kits includes**
 - ◆ controlCARD and application specific baseboard
 - ◆ Code Composer Studio (download)
- ◆ **Software download includes**
 - ◆ Complete schematics, BOM, gerber files, and source code for board and all software
 - ◆ Quickstart demonstration GUI for quick and easy access to all board features
 - ◆ Fully documented software specific to each kit and application
- ◆ **See www.ti.com/c2000 for other kits and more details**
- ◆ **Available through TI authorized distributors and the TI store**

The C2000 Application Kits demonstrate the full capabilities of the C2000 microcontroller in a specific application. The kits are complete evaluation and development tools where the user can modify both the hardware and software to best fit their needs. Each kit uses a device specific controlCARD and a specific application board. All kits are completely open source with full documentation and are supplied with complete schematics, bill of materials, board design details, and software. Visit the TI website for a complete list of available Application Kits.

XDS100/110/200 Class JTAG Debug Probes

XDS100 / XDS110 / XDS200 Class JTAG Debug Probes



◆ **Blackhawk**
◆ **USB100v2**



◆ **Texas Instruments**
◆ **XDS110**



◆ **Spectrum Digital**
◆ **XDS100v2**



◆ **Blackhawk**
◆ **USB200**

www.blackhawk-dsp.com



◆ **Spectrum Digital**
◆ **XDS200**

www.spectrumdigital.com

The JTAG debug probes are used during development to program and communicate with the C20000 microcontroller. While almost all C2000 development tools include emulation capabilities, after you have developed your own target board an external debug probe will be needed. Debug probes are available with different features and at different price points. Shown here are popular debug probes from various manufacturers.

Product Information Resources

For More Information . . .

- ◆ **USA – Product Information Center (PIC)**
 - ◆ Phone: +1-703-344-7012
- ◆ **TI E2E Community**
 - ◆ <http://e2e.ti.com>
- ◆ **Embedded Processor Wiki**
 - ◆ <http://processors.wiki.ti.com>
- ◆ **TI Training**
 - ◆ <http://training.ti.com>
- ◆ **TI store**
 - ◆ <http://store.ti.com>
- ◆ **TI website**
 - ◆ <http://www.ti.com>

For more information and support, 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 – F280049C Experimenter Kit

Overview

This appendix provides a quick reference and mapping of the header pins used on the F280049C LaunchPad and F280049C Experimenter Kit. This allows either development board to be used with the workshop.

Chapter Topics

Appendix A – F280049C Experimenter Kit	A-1
<i>F280049C Experimenter Kit.....</i>	<i>A-3</i>
Initial Hardware Setup	A-3
Docking Station and LaunchPad Pin Mapping.....	A-3
controlCARD and LaunchPad LED Mapping	A-3
Docking Station and LaunchPad Pin Locations	A-4
Stand-Alone Operation (No Emulator)	A-4

F280049C Experimenter Kit

Initial Hardware Setup

- **F280049C Experimenter Kit:**

Insert the F280049C controlCARD into the Docking Station connector slot. Using the two (2) supplied USB cables – plug the USB Standard Type A connectors into the computer USB ports and plug the USB Mini-B connectors as follows:

- J1:A on the controlCARD (left side) – isolated XDS100v2 JTAG emulation
- J17 on the Docking Station – board power

On the Docking Station move switch S1 to the “USB-ON” position. This will power the Docking Station and controlCARD using the power supplied by the computer USB port. Additionally, the other computer USB port will power the on-board isolated JTAG emulator and provide the JTAG communication link between the device and Code Composer Studio.

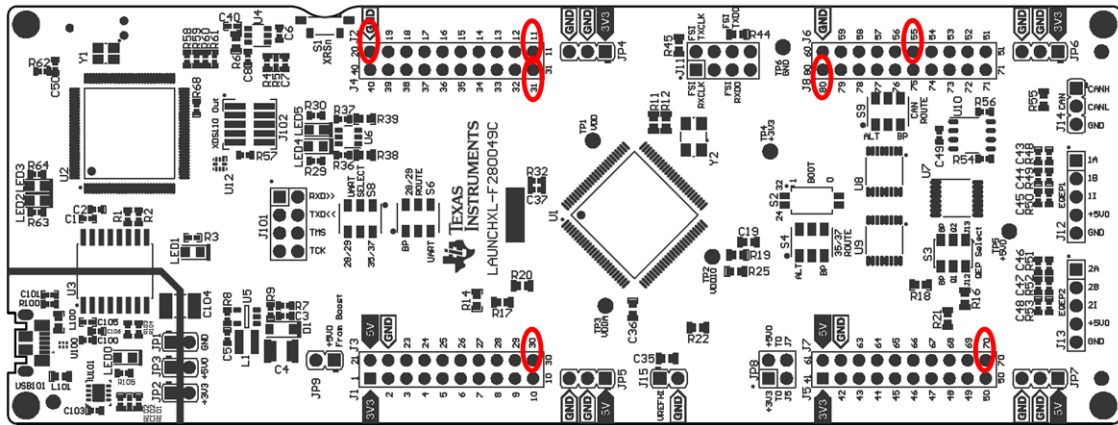
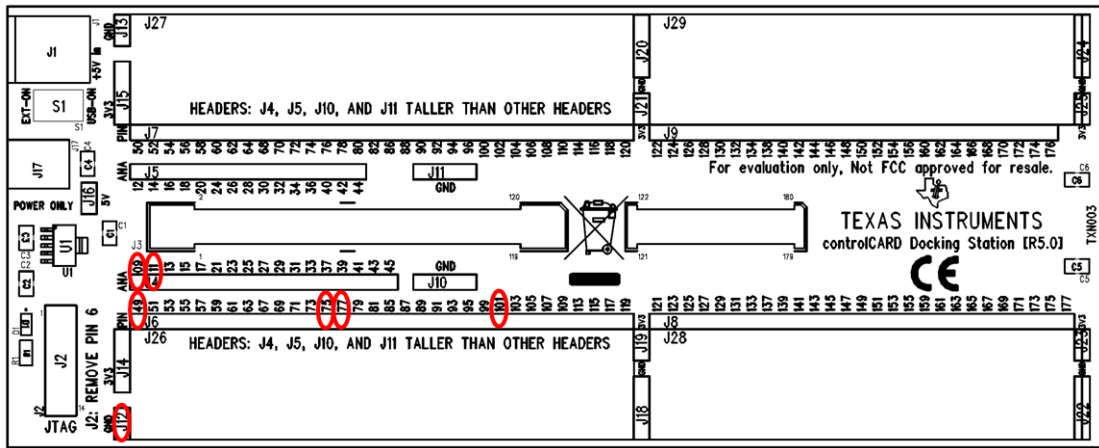
Docking Station and LaunchPad Pin Mapping

Function	Docking Station	LaunchPad
ADCINA0	ANA header, Pin # 09	Pin # 70
GND	GND	Pin # 20 (GND)
GPIO59 ('1')	Pin # 101	Pin # 11
GPIO25 ('T')	Pin # 77	Pin # 31
DACOUTB	ANA header, Pin # 11	Pin # 30
PWM1A	Pin # 49	Pin # 80
ECAP1 (via Input X-bar)	Pin # 75 (GPIO24)	Pin # 55 (GPIO24)

controlCARD and LaunchPad LED Mapping

Function	controlCARD	LaunchPad
LED – Power	LED D1 (green)	LED1 (red)
LED – GPIO31 / GPIO23	LED D2 (red)	LED4 (red)
LED – GPIO34	LED D3 (red)	LED5 (green)

Docking Station and LaunchPad Pin Locations



Stand-Alone Operation (No Emulator)

When the device is in stand-alone boot mode, the state of GPIO24 and GPIO32 pins are used to determine the boot mode. On the controlCARD boot mode selection switch S1 controls the boot options for the F280049C device. This switch is installed with 180 degree rotation. Check that switch S1 positions 2 and 1 are set to the default “1 – up” position (both switches up). This will configure the device (in stand-alone boot mode) to boot from flash. Details of the switch positions can be found in the controlCARD User’s Guide.