TMS320F2837xD Microcontroller Workshop

Workshop Guide and Lab Manual

TEXAS INSTRUMENTS

Kenneth W. Schachter

Revision 2.0
January 2018

Important Notice

Revision History

February 2015 – Revision 1.0

May 2015 – Revision 1.1

January 2018 – Revision 2.0

Mailing Address

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

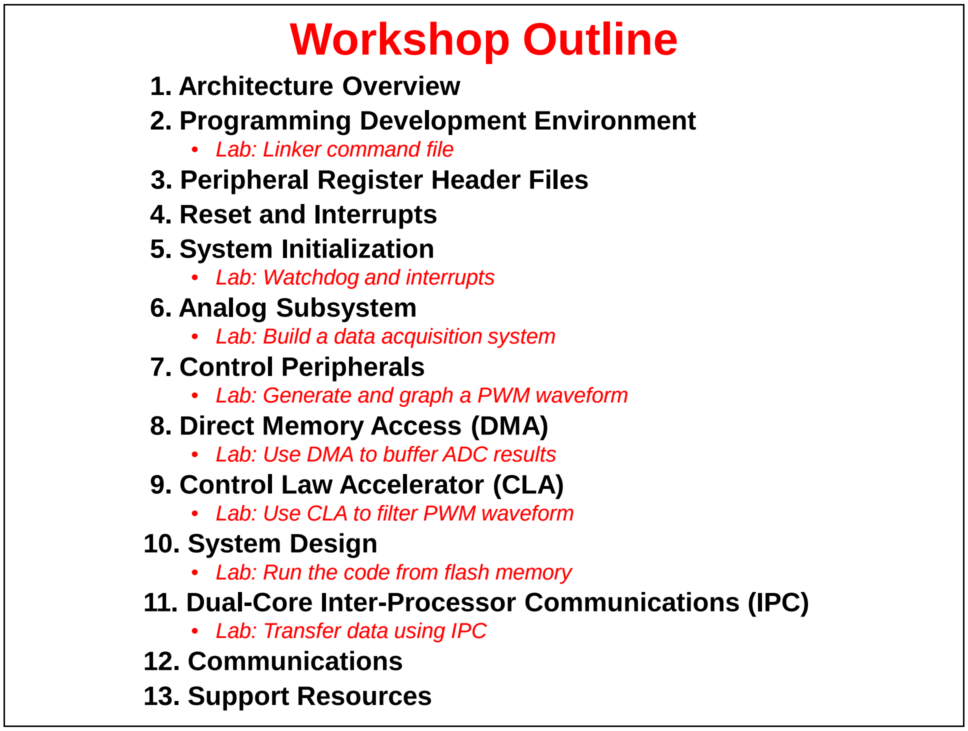
TMS320F2837xD Microcontroller Workshop

TMS320F2837xD Microcontroller Workshop

Texas Instruments
C2000 Technical Training

C2000 is trademarks of Texas Instruments.

Workshop Outline



Workshop Outline

1. Architecture Overview
2. Programming Development Environment
   - *Lab: Linker command file*
3. Peripheral Register Header Files
4. Reset and Interrupts
5. System Initialization
   - *Lab: Watchdog and interrupts*
6. Analog 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. Dual-Core Inter-Processor Communications (IPC)
    - *Lab: Transfer data using IPC*
12. Communications
13. Support Resources

Required Workshop Materials

Required Workshop Materials

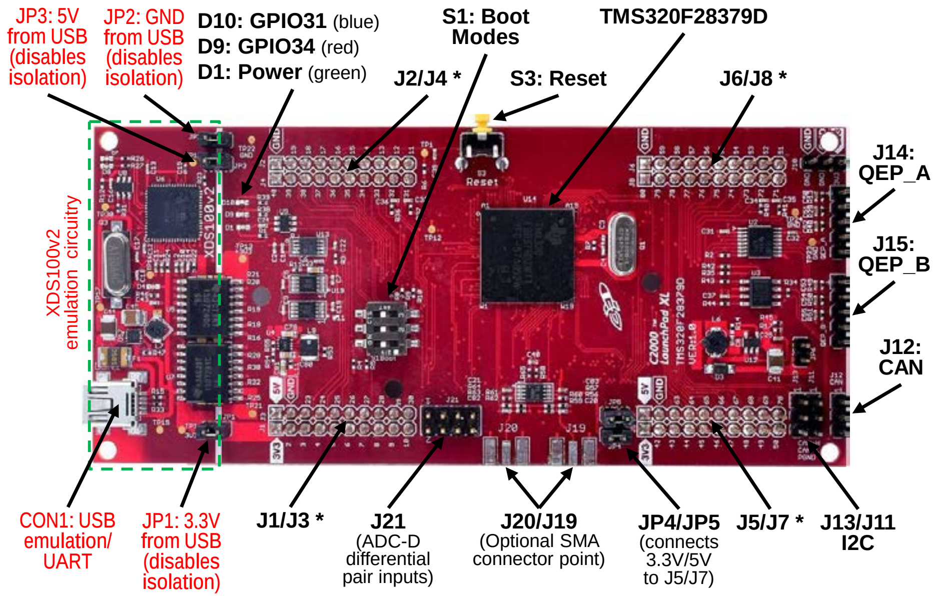
- **http://processors.wiki.ti.com/index.php/C2000_Multi-Day_Workshop**
- **F28379D LaunchPad** (LAUNCHXL-F28379D)
- **Install Code Composer Studio v7.3.0**
- **Run the workshop installer**

  F2837xD Microcontroller Workshop-2.0-Setup.exe

  - **Lab Files / Solution Files**
  - **Workshop Manual**

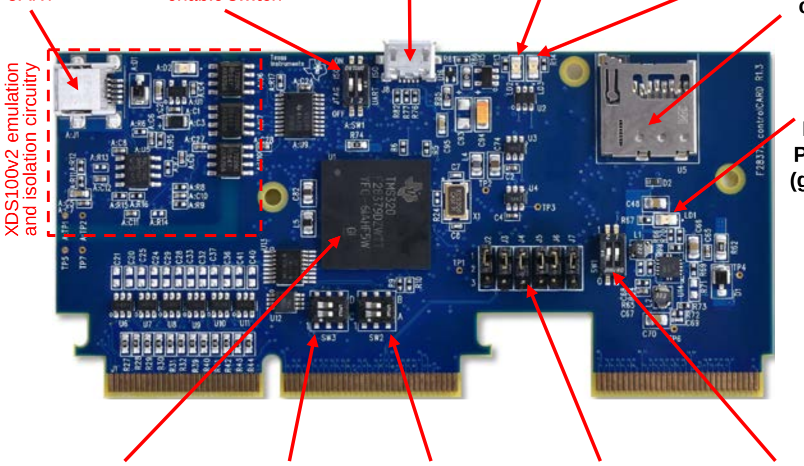
Development Tools

F28379D LaunchPad


JP3: 5V from USB (disables isolation)
JP2: GND from USB (disables isolation)
D10: GPIO31 (blue)
D9: GPIO34 (red)
D1: Power (green)
S1: Boot Modes
TMS320F28379D
J2/J4 *
S3: Reset
J6/J8 *
J14: QEP_A
J15: QEP_B
J12: CAN
XDS100v2 emulation circuitry
CON1: USB emulation/ UART
JP1: 3.3V from USB (disables isolation)
J1/J3 *
J21 (ADC-D differential pair inputs)
J20/J19 (Optional SMA connector point)
JP4/JP5 (connects 3.3V/5V to J5/J7)
J5/J7 *
J13/J11 I2C

*** = BoosterPack plug-in module connector** Note: F28379D – 337 pin package

F28379D controlCARD



A:J1 - USB emulation/ UART

A:SW1 - isolated emulation and UART communication enable switch

J8: Host/ Device

LED LD2: GPIO31 (red)

LED LD3: GPIO34 (red)

U5: µSD card

LED LD1: Power (green)

XDS100v2 emulation and isolation circuitry

TMS320F28379D

SW3: ADC VREFHI ADC C & D

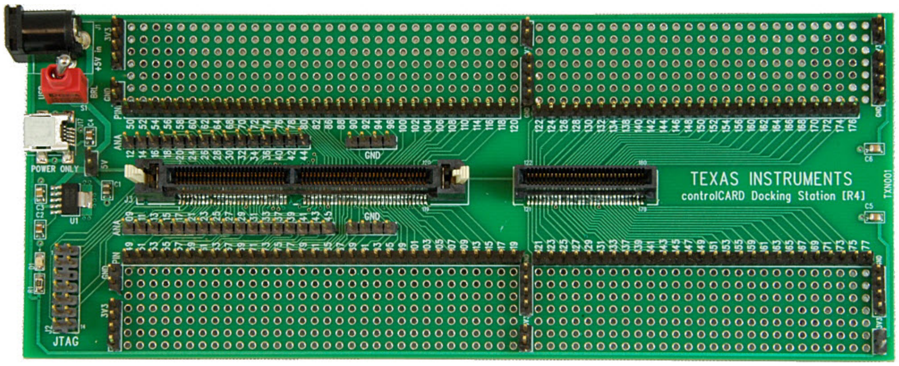
SW2: ADC VREFHI ADC A & B

J2 - J7: USB PHY connection enable jumpers

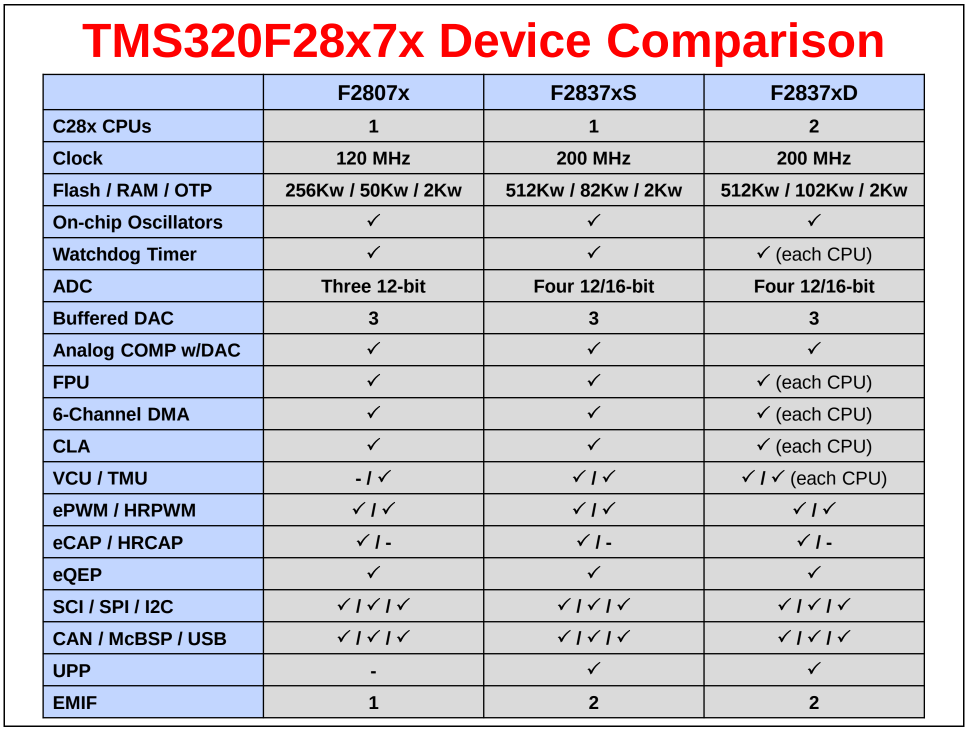
SW1: Boot Modes

Note: F28379D – 337 BGA

controlCARD Docking Station



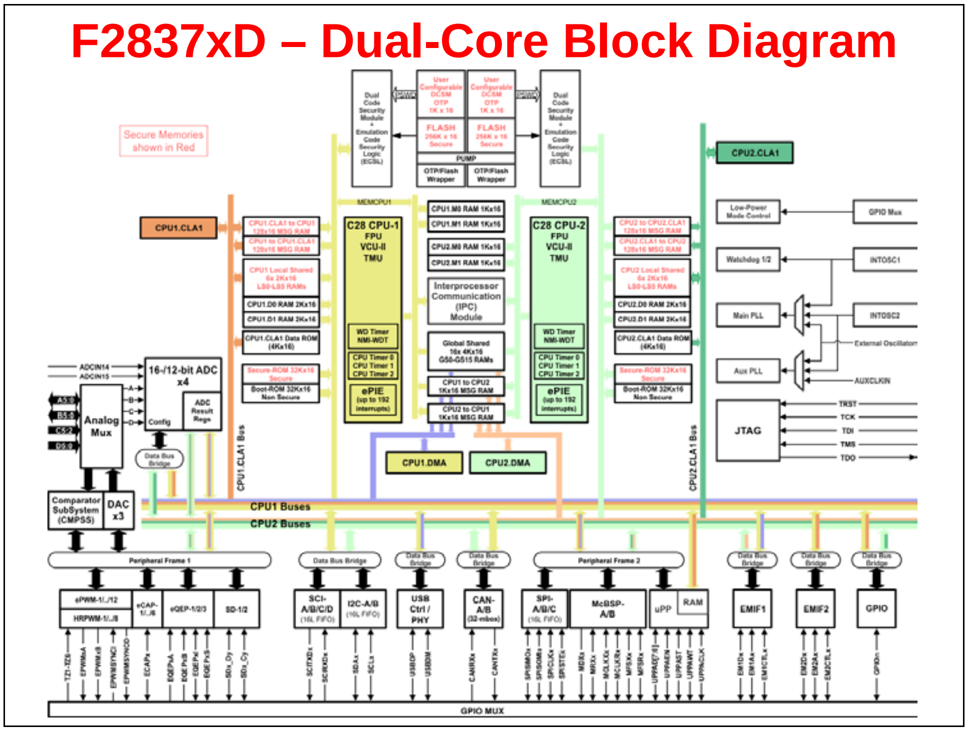
TMS320F28x7x Device Comparison



TMS320F28x7x Device Comparison

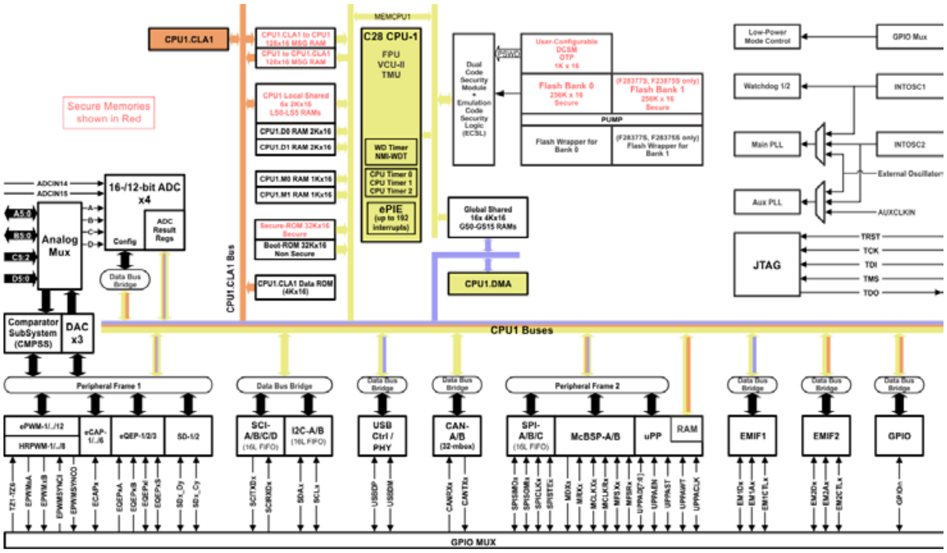
	F2807x	F2837xS	F2837xD
C28x CPUs	1	1	2
Clock	120 MHz	200 MHz	200 MHz
Flash / RAM / OTP	256Kw / 50Kw / 2Kw	512Kw / 82Kw / 2Kw	512Kw / 102Kw / 2Kw
On-chip Oscillators	✓	✓	✓
Watchdog Timer	✓	✓	✓ (each CPU)
ADC	Three 12-bit	Four 12/16-bit	Four 12/16-bit
Buffered DAC	3	3	3
Analog COMP w/DAC	✓	✓	✓
FPU	✓	✓	✓ (each CPU)
6-Channel DMA	✓	✓	✓ (each CPU)
CLA	✓	✓	✓ (each CPU)
VCU / TMU	- / ✓	✓ / ✓	✓ / ✓ (each CPU)
ePWM / HRPWM	✓ / ✓	✓ / ✓	✓ / ✓
eCAP / HRCAP	✓ / -	✓ / -	✓ / -
eQEP	✓	✓	✓
SCI / SPI / I2C	✓ / ✓ / ✓	✓ / ✓ / ✓	✓ / ✓ / ✓
CAN / McBSP / USB	✓ / ✓ / ✓	✓ / ✓ / ✓	✓ / ✓ / ✓
UPP	-	✓	✓
EMIF	1	2	2

TMS320F28x7x Block Diagrams

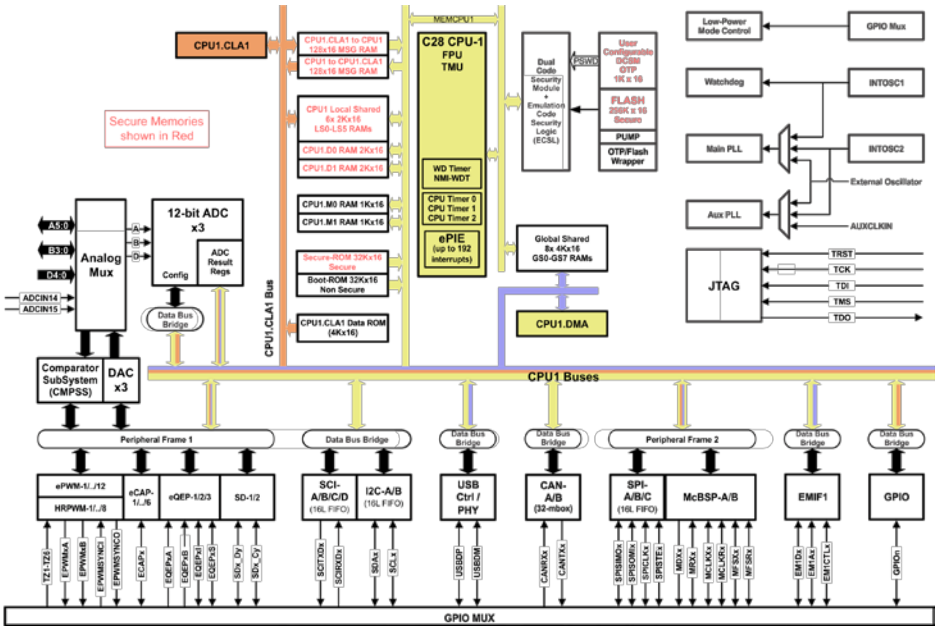


F2837xD – Dual-Core Block Diagram

F2837xS – Single-Core Block Diagram



F2807x – Block Diagram



Architecture Overview

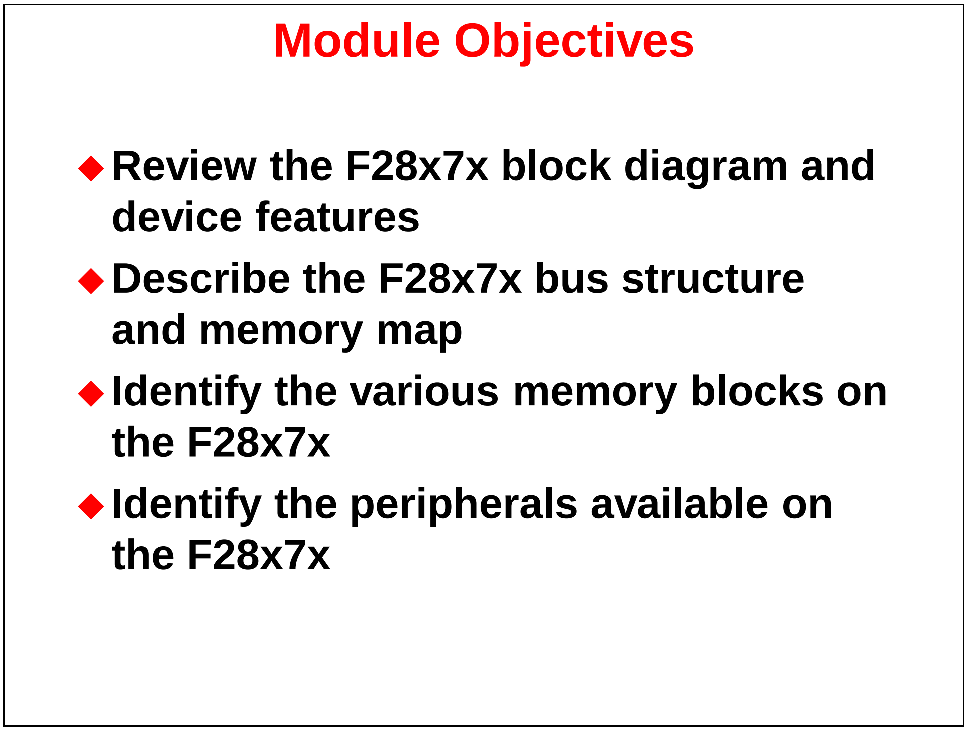
Introduction

This architectural overview introduces the basic architecture of the C2000™ family of microcontrollers from Texas Instruments. The F28x7x series adds a new level of high performance processing ability. The C2000™ is ideal for applications combining digital signal processing, microcontroller processing, efficient C code execution, and operating system tasks.

Unless otherwise noted, the terms C28x and F28x7x refer to TMS320F28x7x devices throughout the remainder of these notes. For specific details and differences please refer to the device data sheet, user's guide, and technical reference manual.

Module Objectives

When this module is complete, you should have a basic understanding of the F28x7x architecture and how all of its components work together to create a high-end, uniprocessor control system.



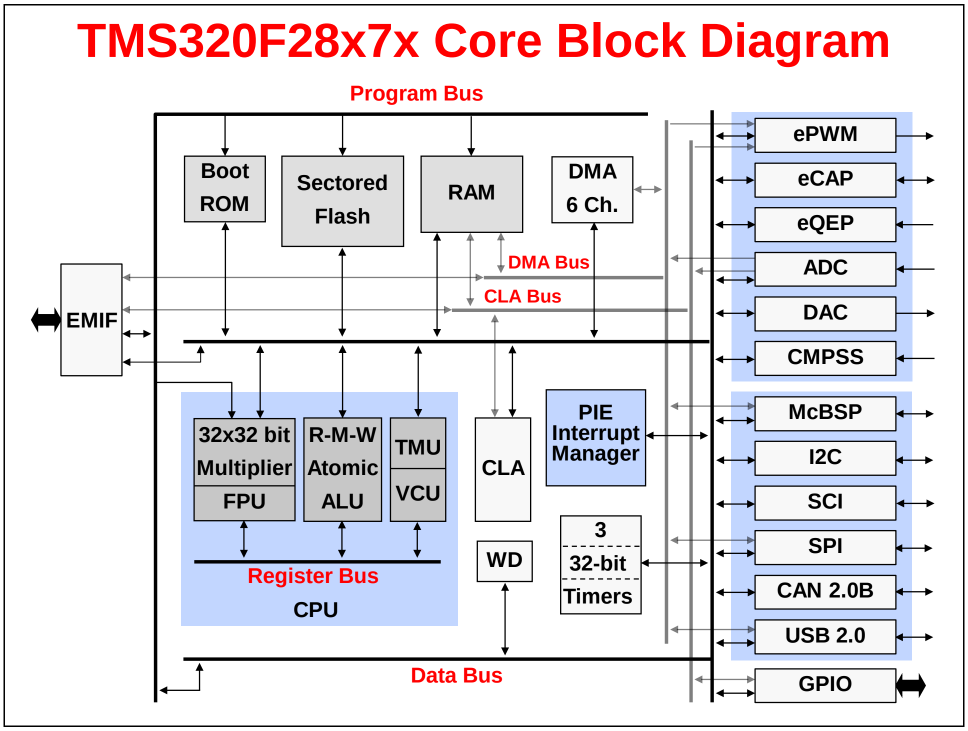
Module Objectives

- ◆ **Review the F28x7x block diagram and device features**
- ◆ **Describe the F28x7x bus structure and memory map**
- ◆ **Identify the various memory blocks on the F28x7x**
- ◆ **Identify the peripherals available on the F28x7x**

Chapter Topics

Introduction to the TMS320F28x7x

The TMS320F37xD, TMS320F37xS, and TMS320F07x, collectively referred to as the TMS320F28x7x or F28x7x, 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 TMS320F2837xD dual-core device series, most all of the topics are fully applicable to the TMS320F2837xS and TMS320F2807x single-core device series. The F2837xD dual-core MCU design is based on the TI 32-bit C28x CPU architecture. Each core is identical with access to its own local RAM and flash memory, as well as globally shared RAM memory. Sharing information between the two CPU cores is accomplished with an Inter-Processor Communications (IPC) module. Additionally, each core shares access to a common set of highly integrated analog and control peripherals, providing a complete solution for demanding real-time high-performance signal processing applications, such as digital power, industrial drives, inverters, and motor control.



TMS320F28x7x Core Block Diagram

The above block diagram represents an overview of all device features and is not specific to any one device. The F28x7x device is designed around a multibus architecture, also known as a modified Harvard architecture. This can be seen in the block diagram by the separate program bus and data bus, along with the link between the two buses. This type of architecture greatly enhances the performance of the device.

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

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

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

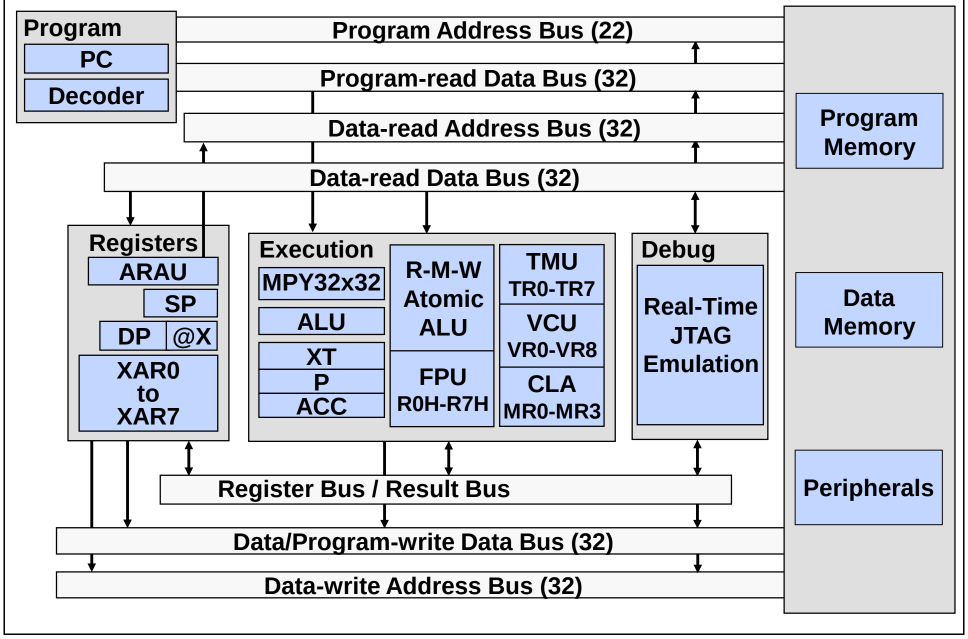
The PIE block, or Peripheral Interrupt Expansion block, manages the interrupts from the peripherals. In the bottom right corner is the general-purpose I/O. The CPU has a watchdog module and three 32-bit general-purpose timers are available. Also, the device features an external memory interface, as shown on the left side.

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)

C28x CPU Internal Bus Structure

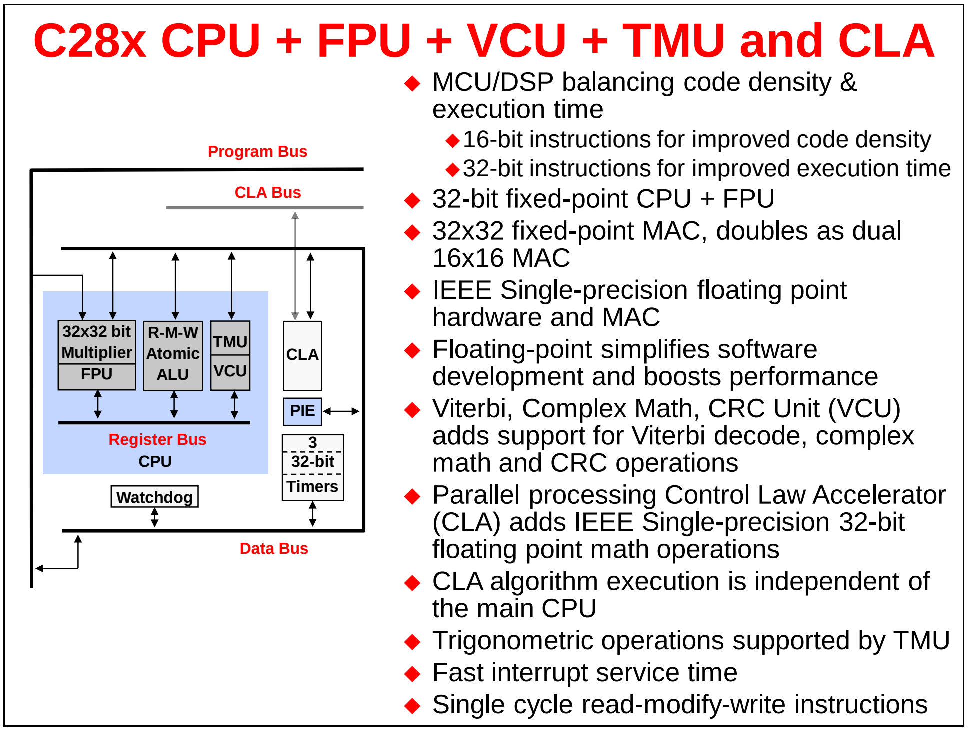


The 32-bit-wide data busses provide single cycle 32-bit operations. This multiple bus architecture (Harvard Bus Architecture) enables the C28x to fetch an instruction, read a data value and write a data value in a single cycle. All peripherals and memory blocks are attached to the memory bus with prioritized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



C28x CPU + FPU + VCU + TMU and CLA

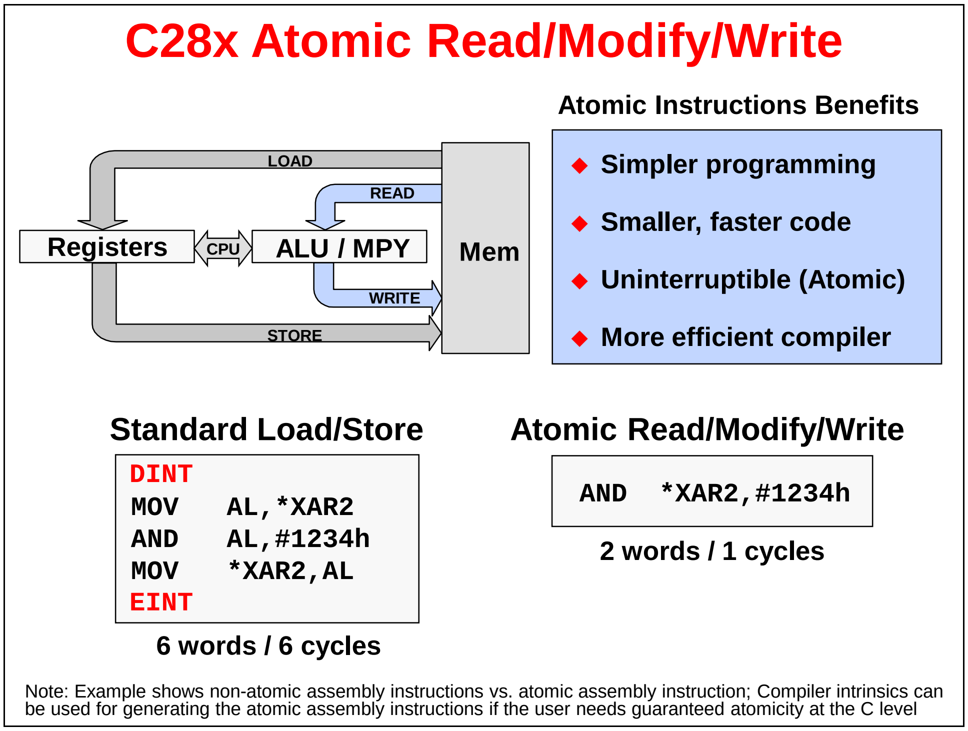
- MCU/DSP balancing code density & execution time
  - 16-bit instructions for improved code density
  - 32-bit instructions for improved execution time
- 32-bit fixed-point CPU + FPU
- 32x32 fixed-point MAC, doubles as dual 16x16 MAC
- IEEE Single-precision floating point hardware and MAC
- Floating-point simplifies software development and boosts performance
- Viterbi, Complex Math, CRC Unit (VCU) adds support for Viterbi decode, complex math and CRC operations
- Parallel processing Control Law Accelerator (CLA) adds IEEE Single-precision 32-bit floating point math operations
- CLA algorithm execution is independent of the main CPU
- Trigonometric operations supported by TMU
- Fast interrupt service time
- Single cycle read-modify-write instructions

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, though the F2837xD is a dual-core device for even higher performance. 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



C28x Atomic Read/Modify/Write

Atomic Instructions Benefits

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

Standard Load/Store

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

6 words / 6 cycles

Atomic Read/Modify/Write

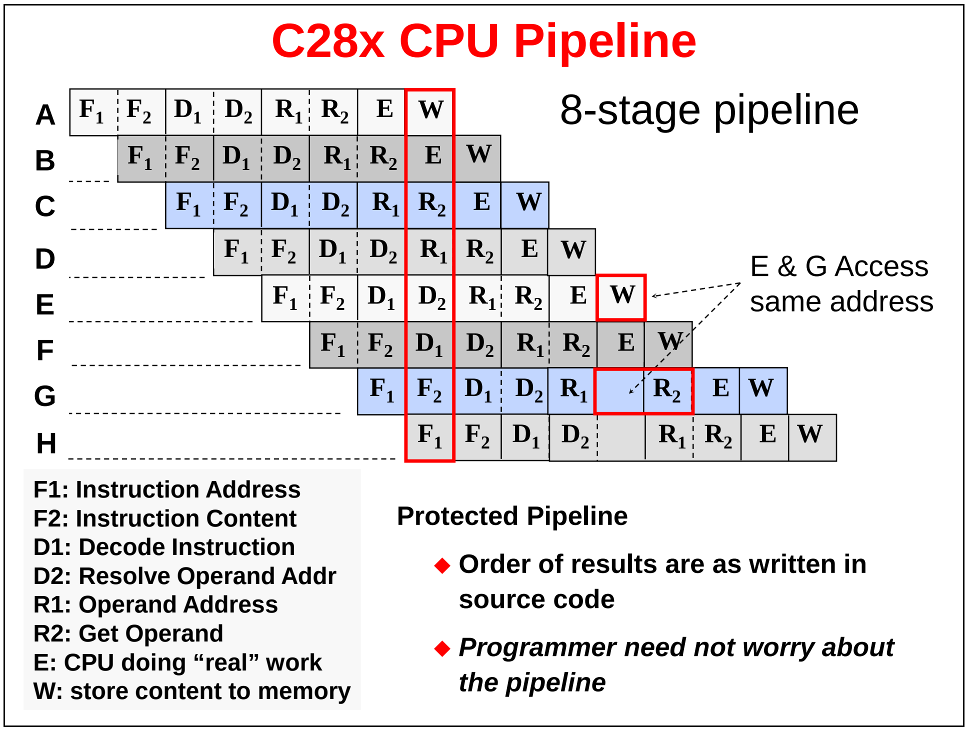
```
AND   *XAR2,#1234h
```

2 words / 1 cycles

Note: Example shows non-atomic assembly instructions vs. atomic assembly instruction; Compiler intrinsics can be used for generating the atomic assembly instructions if the user needs guaranteed atomicity at the C level

Atomic instructions are a group of small common instructions which are non-interuptable. The atomic ALU capability supports instructions and code that manages tasks and processes. These instructions usually execute several cycles faster than traditional coding.

CPU Pipeline



C28x CPU Pipeline

8-stage pipeline

A	F_1	F_2	D_1	D_2	R_1	R_2	E	W								
B		F_1	F_2	D_1	D_2	R_1	R_2	E	W							
C			F_1	F_2	D_1	D_2	R_1	R_2	E	W						
D				F_1	F_2	D_1	D_2	R_1	R_2	E	W					
E					F_1	F_2	D_1	D_2	R_1	R_2	E	W				
F						F_1	F_2	D_1	D_2	R_1	R_2	E	W			
G							F_1	F_2	D_1	D_2	R_1		R_2	E	W	
H								F_1	F_2	D_1	D_2		R_1	R_2	E	W

E & G Access same address

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

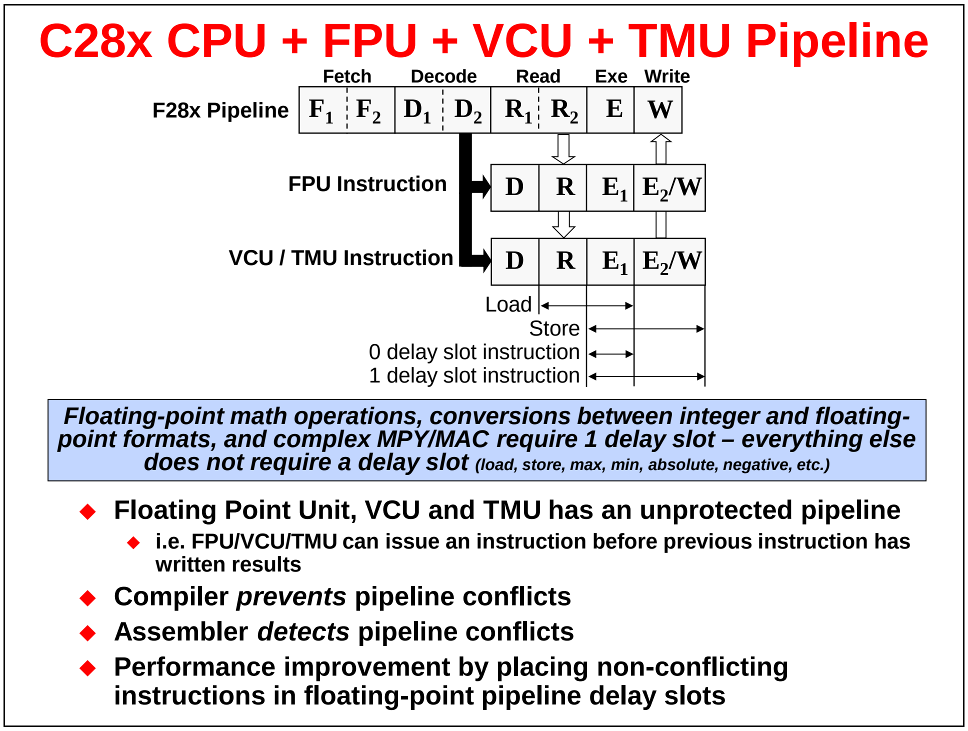
Protected Pipeline

- **Order of results are as written in source code**
- ***Programmer need not worry about the pipeline***

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

This pipelining also enables the C28x 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



C28x CPU + FPU + VCU + TMU Pipeline

***Floating-point math operations, conversions between integer and floating-point formats, and complex MPY/MAC require 1 delay slot – everything else does not require a delay slot** (load, store, max, min, absolute, negative, etc.)*

- **Floating Point Unit, VCU and TMU has an unprotected pipeline**
  - **i.e. FPU/VCU/TMU can issue an instruction before previous instruction has written results**
- **Compiler *prevents* pipeline conflicts**
- **Assembler *detects* pipeline conflicts**
- **Performance improvement by placing non-conflicting instructions in floating-point pipeline delay slots**

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.

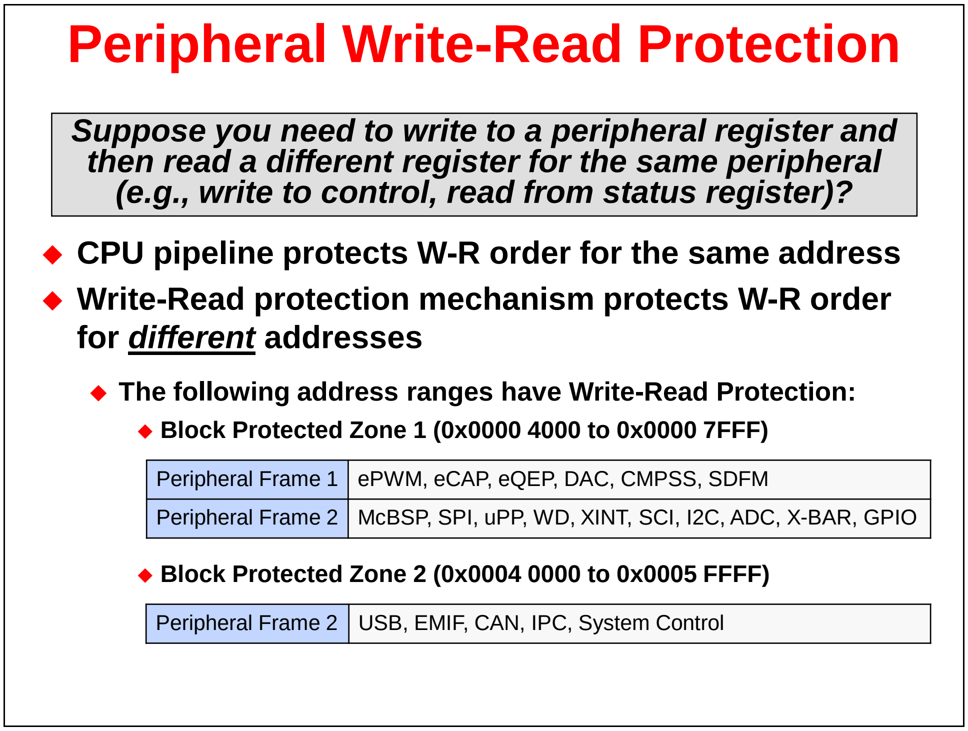
In the user's guide, instructions requiring delay slots have a 'p' after their cycle count. The 2p stands for 2 pipelined cycles. A new instruction can be started on each cycle. The result is valid only 2 instructions later.

Three general guideslines for the FPU/VCU/TMU pipeline are:

Math	MPYF32, ADDF32, SUBF32, MACF32, VCMPY	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.

Peripheral Write-Read Protection



Peripheral Write-Read Protection

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

- **CPU pipeline protects W-R order for the same address**
- **Write-Read protection mechanism protects W-R order for *different* addresses**
  - **The following address ranges have Write-Read Protection:**
    - **Block Protected Zone 1 (0x0000 4000 to 0x0000 7FFF)**

Peripheral Frame 1	ePWM, eCAP, eQEP, DAC, CMPSS, SDFM
Peripheral Frame 2	McBSP, SPI, uPP, WD, XINT, SCI, I2C, ADC, X-BAR, GPIO

- **Block Protected Zone 2 (0x0004 0000 to 0x0005 FFFF)**

Peripheral Frame 2	USB, EMIF, CAN, IPC, System Control

The peripheral write-read protection is a mechanism to protect the write-read order for peripherals at different addresses. This works similar to the CPU pipeline protection of write-read order for the same address.

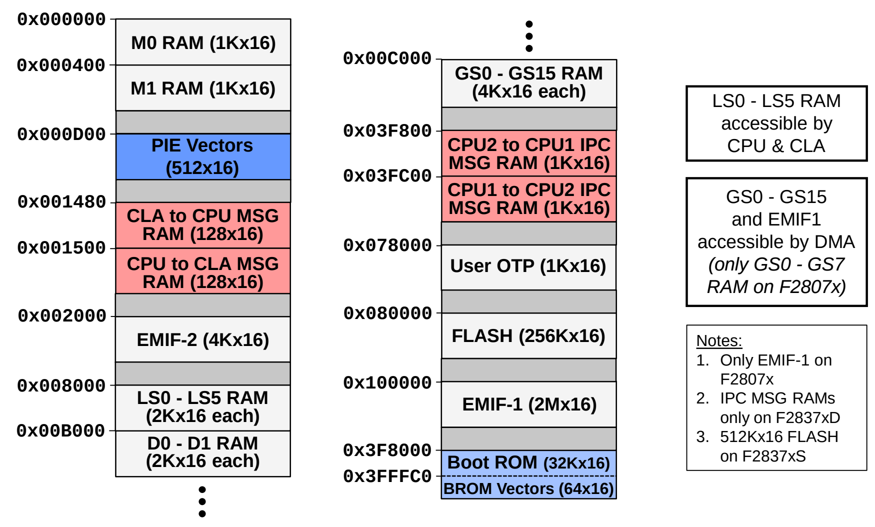
Memory

The F28x7x MCU 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 structure consisting of dedicated RAM blocks, shared local RAM blocks, shared global RAM blocks, message RAM blocks, Flash, and one-time programmable (OTP) memory. The Boot 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.

Simplified F28x7x Memory Map



0x000000 M0 RAM (1Kx16)
0x000400 M1 RAM (1Kx16)
0x000D00 PIE Vectors (512x16)
0x001480 CLA to CPU MSG RAM (128x16)
0x001500 CPU to CLA MSG RAM (128x16)
0x002000 EMIF-2 (4Kx16)
0x008000 LS0 - LS5 RAM (2Kx16 each)
0x00B000 D0 - D1 RAM (2Kx16 each)

0x00C000 GS0 - GS15 RAM (4Kx16 each)
0x03F800 CPU2 to CPU1 IPC MSG RAM (1Kx16)
0x03FC00 CPU1 to CPU2 IPC MSG RAM (1Kx16)
0x078000 User OTP (1Kx16)
0x080000 FLASH (256Kx16)
0x100000 EMIF-1 (2Mx16)
0x3F8000 Boot ROM (32Kx16)
0x3FFFC0 BROM Vectors (64x16)

LS0 - LS5 RAM accessible by CPU & CLA

GS0 - GS15 and EMIF1 accessible by DMA *(only GS0 - GS7 RAM on F2807x)*

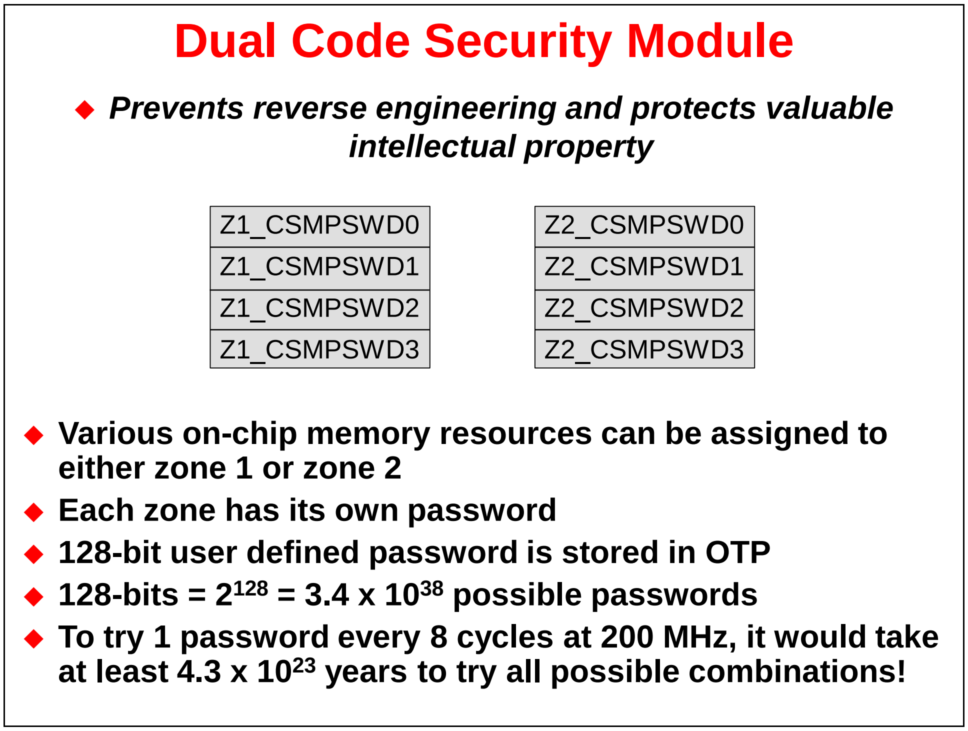
Notes:
1. Only EMIF-1 on F2807x
2. IPC MSG RAMs only on F2837xD
3. 512Kx16 FLASH on F2837xS

There are four dedicated RAM block (M0, M1, D0, and D1) 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 six local shared memory blocks, LS0 through LS5, are accessible by its CPU and CLA. Global shared memory blocks GS0 through GS15 on the F2837x and through GS7 on the F2807x are accessible by CPU and DMA.

There are two types of message RAM blocks: CPU message RAM blocks and CLA message RAM blocks. The CPU message RAM blocks are used to share data between CPU1 subsystem and CPU2 subsystem in a dual-core device via inter-processor communications. The CLA message RAM blocks are used to share date between the CPU and CLA.

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. Notice that the external memory interface is assigned a region within the memory map. 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 x 10^{38} possible passwords**
- **To try 1 password every 8 cycles at 200 MHz, it would take at least 4.3 x 10^{23} years to try all possible combinations!**

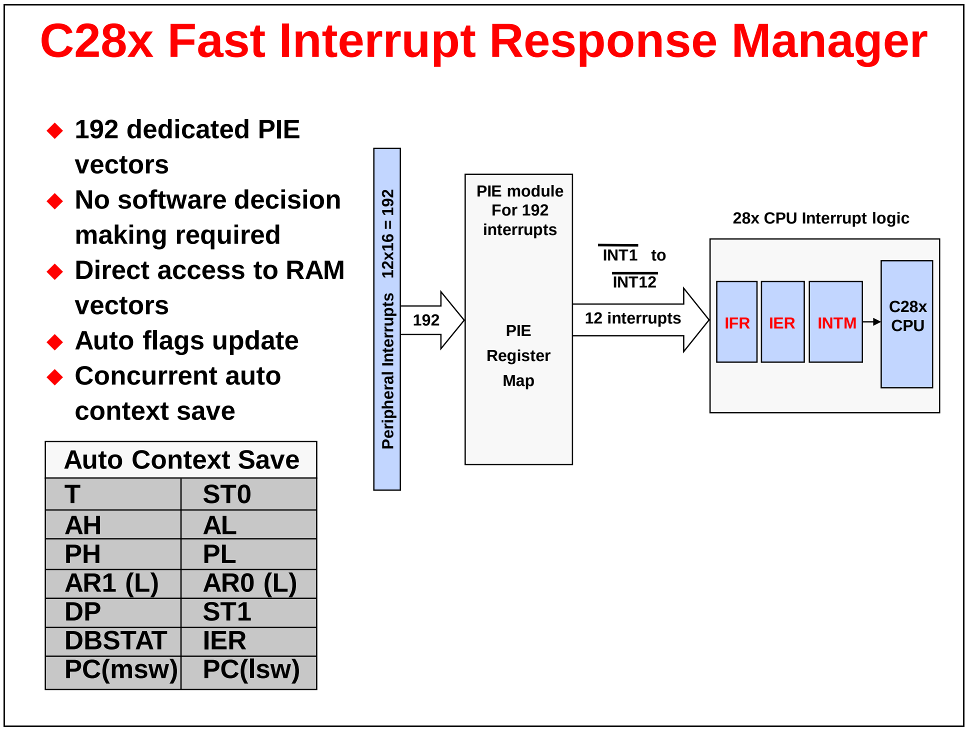
Peripherals

The F28x7x is available with a variety of built in peripherals optimized to support control applications. These peripherals vary depending on which F28x7x device selected.

- ePWM
- eCAP
- eQEP
- CMPSS
- ADC
- DAC
- Watchdog Timer
- DMA
- CLA
- SDFM
- SPI
- SCI
- I2C
- McBSP
- CAN
- USB
- GPIO
- EMIF

Fast Interrupt Response Manager

The fast interrupt response manage is capable of automatically performing context save of critical registers. This results in the ability of servicing many asynchronous events with minimal latency. The F28x7x implements a zero cycle penalty to do 14 registers context saved and restored during an interrupt. This feature helps reduces the interrupt service routine overheads.



C28x Fast Interrupt Response Manager

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

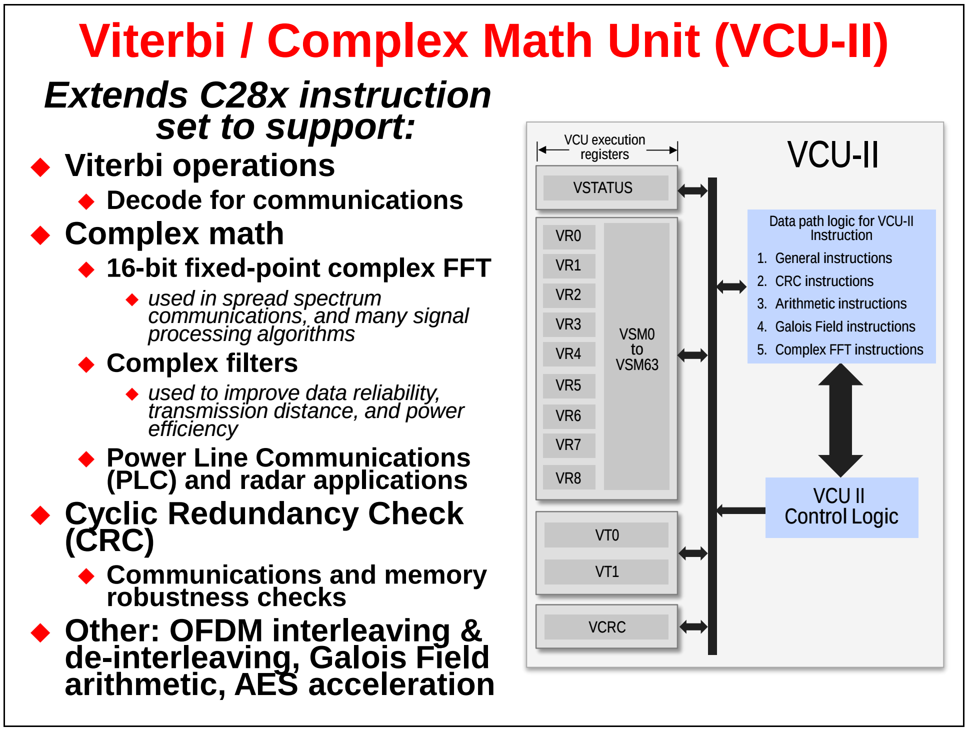
Auto Context Save	
T	ST0
AH	AL
PH	PL
AR1 (L)	AR0 (L)
DP	ST1
DBSTAT	IER
PC(msw)	PC(lsw)

Peripheral Interrupts 12x16 = 192
192
PIE module For 192 interrupts
PIE Register Map
$\overline{\text{INT1}}$ to $\overline{\text{INT12}}$
12 interrupts
28x CPU Interrupt logic
IFR
IER
INTM
C28x CPU

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, interrupts, and system initialization modules.

Math Accelerators

Viterbi / Complex Math Unit (VCU-II)



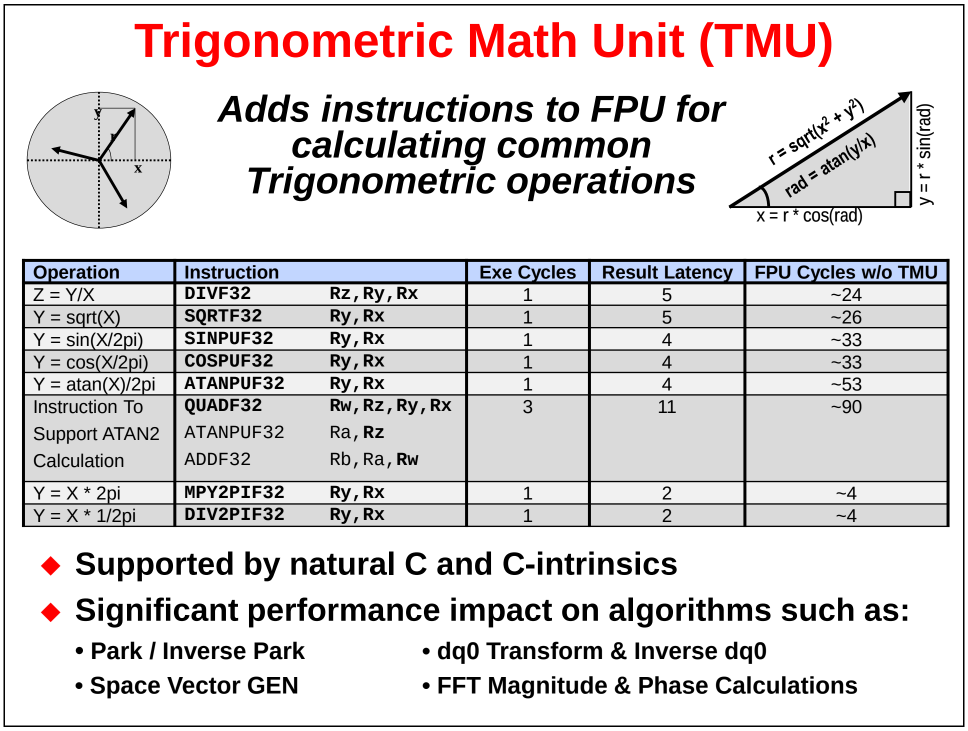
Viterbi / Complex Math Unit (VCU-II)

Extends C28x instruction set to support:

- **Viterbi operations**
  - **Decode for communications**
- **Complex math**
  - **16-bit fixed-point complex FFT**
    - *used in spread spectrum communications, and many signal processing algorithms*
  - **Complex filters**
    - *used to improve data reliability, transmission distance, and power efficiency*
  - **Power Line Communications (PLC) and radar applications**
- **Cyclic Redundancy Check (CRC)**
  - **Communications and memory robustness checks**
- **Other: OFDM interleaving & de-interleaving, Galois Field arithmetic, AES acceleration**

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

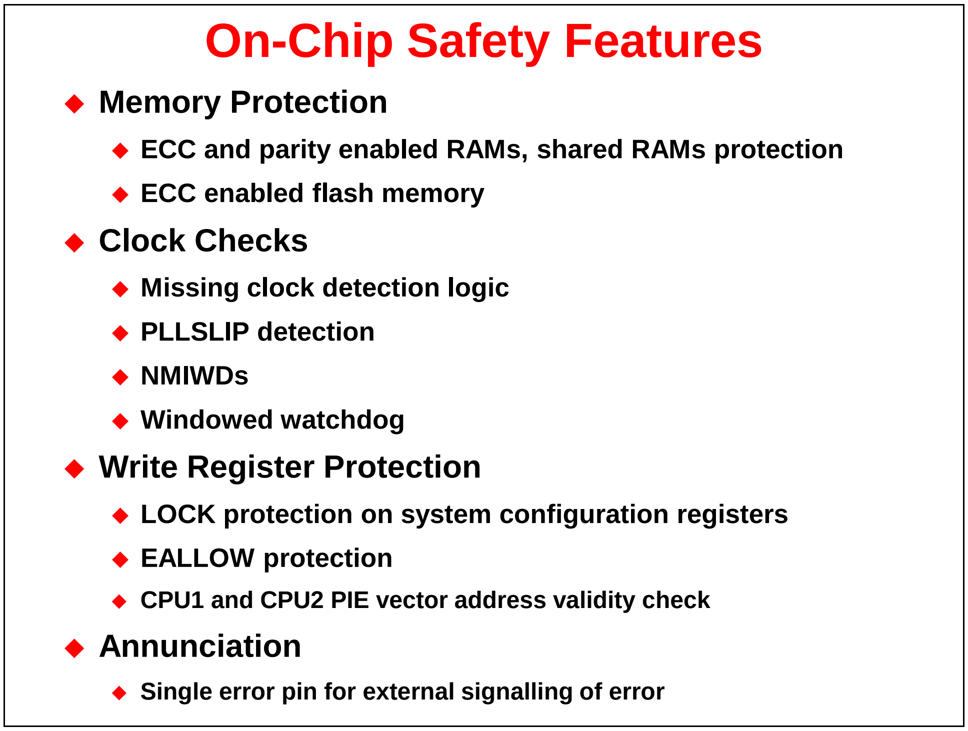
y
r
x
r = sqrt(x^2 + y^2)
rad = atan(y/x)
x = r * cos(rad)
y = r * sin(rad)

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

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

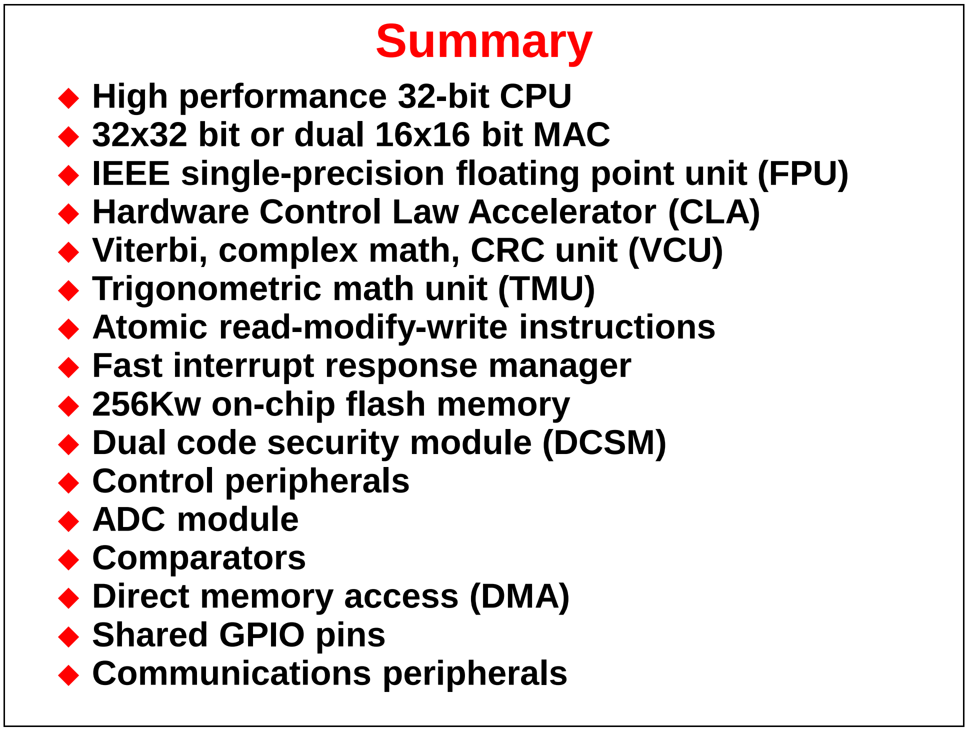
On-Chip Safety Features



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
  - CPU1 and CPU2 PIE vector address validity check
- Annunciation
  - Single error pin for external signalling of error

Summary



Summary

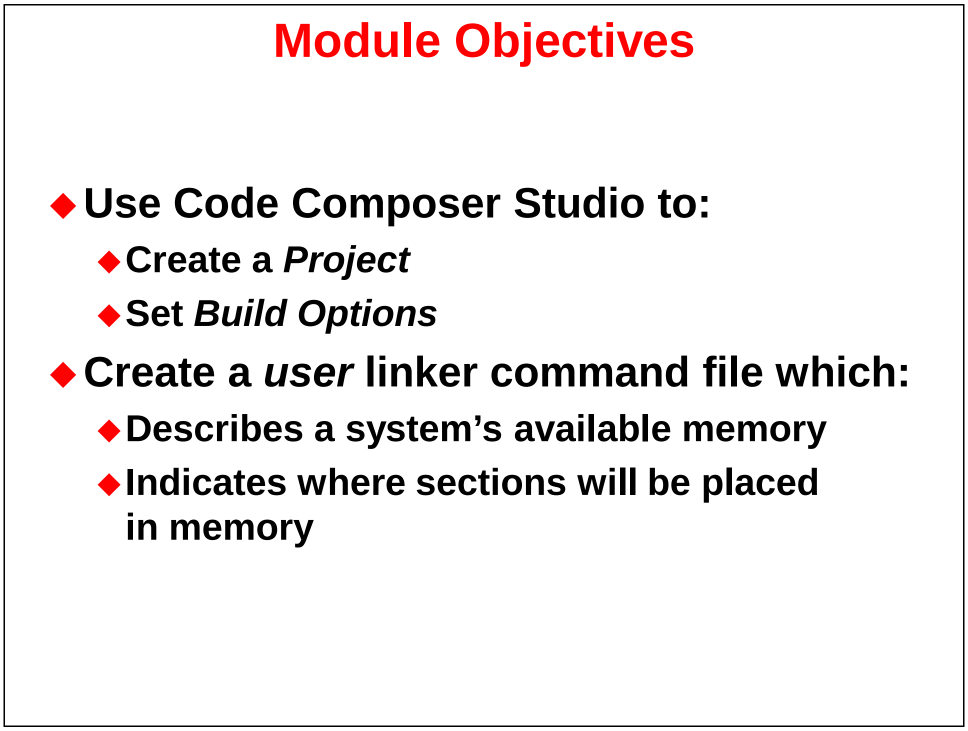
- High performance 32-bit CPU
- 32x32 bit or dual 16x16 bit MAC
- IEEE single-precision floating point unit (FPU)
- Hardware Control Law Accelerator (CLA)
- Viterbi, complex math, CRC unit (VCU)
- Trigonometric math unit (TMU)
- Atomic read-modify-write instructions
- Fast interrupt response manager
- 256Kw on-chip flash memory
- Dual code security module (DCSM)
- Control peripherals
- ADC module
- Comparators
- 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

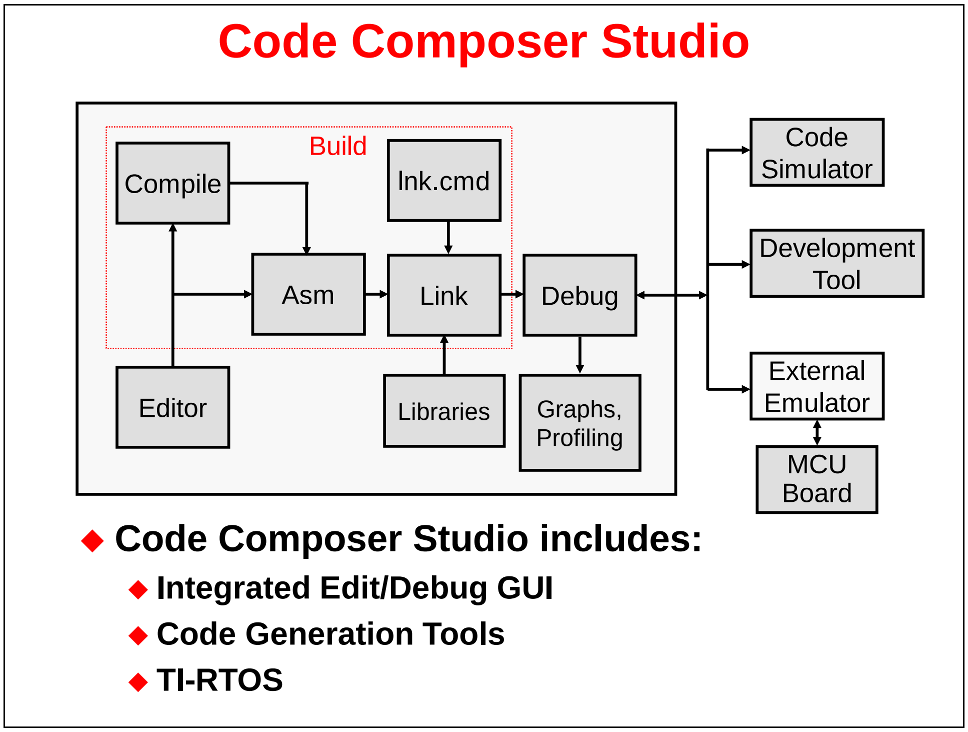
Chapter Topics

Code Composer Studio

Software Development and COFF Concepts

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

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



Code Composer Studio

- **Code Composer Studio includes:**
  - **Integrated Edit/Debug GUI**
  - **Code Generation Tools**
  - **TI-RTOS**

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

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

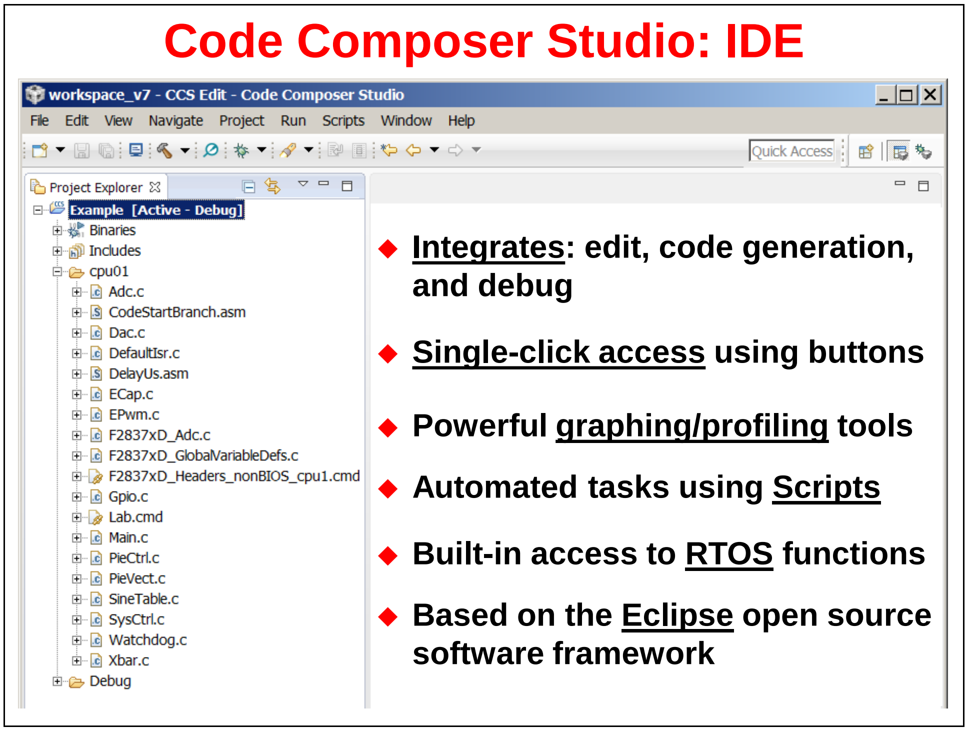
The high level of modularity and portability resulting from this system simplifies the processes of verification, debug and maintenance. The process of COFF development is presented in greater detail in the following paragraphs.

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

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

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

Code Composer Studio



Code Composer Studio: IDE

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

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

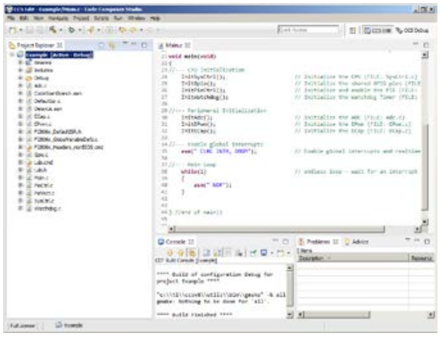
CCS is based on the Eclipse open source software framework. The Eclipse software framework was originally developed as an open framework for creating development tools. Eclipse offers an excellent software framework for building software development environments and it is becoming a standard framework used by many embedded software vendors. CCS combines the advantages of the Eclipse software framework with advanced embedded debug capabilities from TI resulting in a compelling feature-rich development environment for embedded developers. CCS supports running on both Windows and Linux PCs. Note that not all features or devices are supported on Linux.

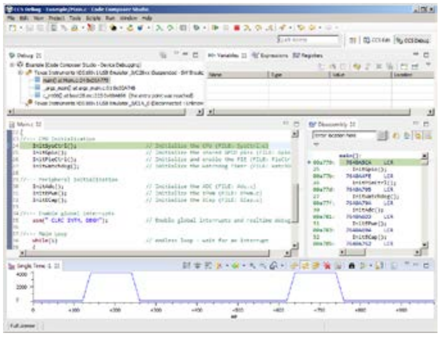
Edit and Debug Perspective (CCSv7)

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 (CCSv7)

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





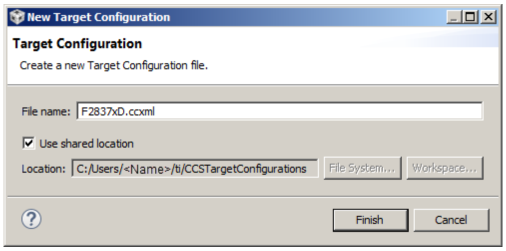
- **Edit Perspective**
  - **Displays views used during code development**
    - **C/C++ project, editor, etc.**
- **Debug Perspective**
  - **Displays views used for debugging**
    - **Menus and toolbars associated with debugging, watch and memory windows, graphs, etc.**

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

Target Configuration

A Target Configuration 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



Target Configuration

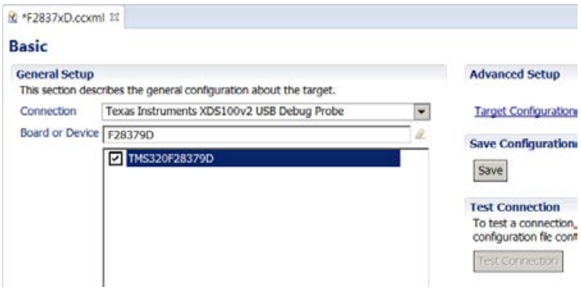
Create a new Target Configuration file.

File name: F2837xD.ccxml

☑ Use shared location

Location: C:/Users/<Name>/ti/CCSTargetConfigurations

- *File → New → Target Configuration File*



This section describes the general configuration about the target.

Connection: Texas Instruments XDS100v2 USB Debug Probe

Board or Device: F28379D

☑ TMS320F28379D

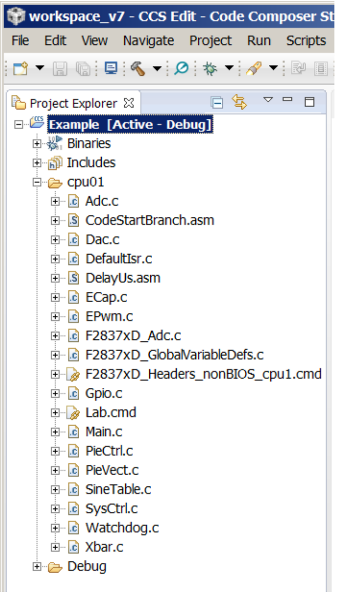
To test a connection, configuration file cont

- Select connection type
- Select device
- Save configuration

CCSv7 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.

CCSv7 Project


workspace_v7 - CCS Edit - Code Composer St
File Edit View Navigate Project Run Scripts
Project Explorer

Example [Active - Debug]
- Binaries
- Includes
- cpu01
  - Adc.c
  - CodeStartBranch.asm
  - Dac.c
  - DefaultIsr.c
  - DelayUs.asm
  - ECap.c
  - EPwm.c
  - F2837xD_Adc.c
  - F2837xD_GlobalVariableDefs.c
  - F2837xD_Headers_nonBIOS_cpu1.cmd
  - Gpio.c
  - Lab.cmd
  - Main.c
  - PieCtrl.c
  - PieVect.c
  - SineTable.c
  - SysCtrl.c
  - Watchdog.c
  - Xbar.c
- Debug

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.

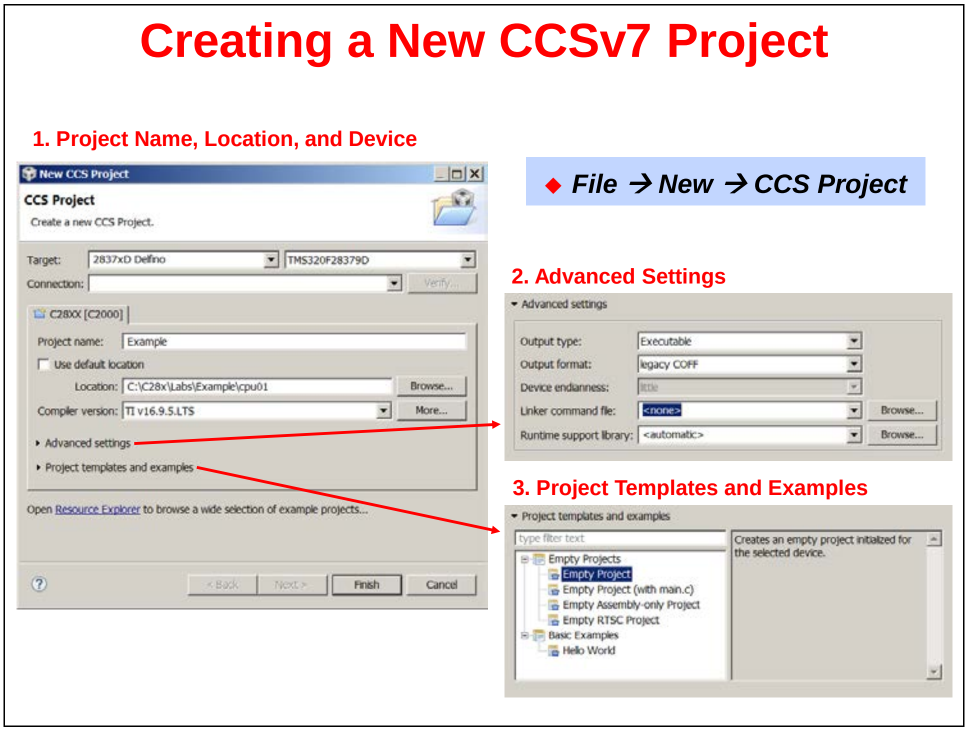
To create a new project, you need to select the following menu items:

`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 CCSv7 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 CCSv7 Project

◆ *File → New → CCS Project*

1. Project Name, Location, and Device

New CCS Project

CCS Project

Create a new CCS Project.

Target: 2837xD Delfino | TMS320F28379D

Connection:

C28XX [C2000]

Project name: Example

□ Use default location

Location: C:\C28x\Labs\Example\cpu01

Compiler version: TI v16.9.5.LTS

Open Resource Explorer to browse a wide selection of example projects...

2. Advanced Settings

Advanced settings

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

Creates an empty project initialized for the selected device.

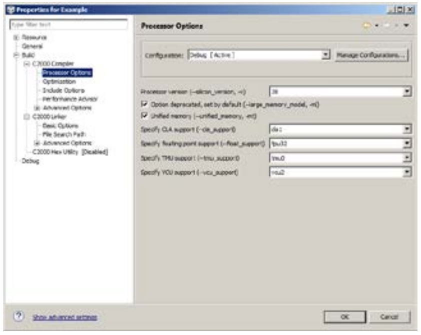
After a project is created, the build options are configured.

CCSv7 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.

CCSv7 Build Options – Compiler / Linker



Configuration: Debug [Active]

Processor version (--silicon_version, -v): 28

☑ Option deprecated, set by default (--large_memory_model, -ml)

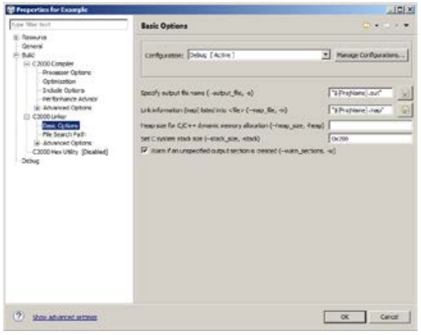
☑ Unified memory (--unified_memory, -mt)

Specify CLA support (--cla_support): cla1

Specify floating point support (--float_support): fpu32

Specify TMU support (--tmu_support): tmu0

Specify VCU support (--vcu_support): vcu2



Configuration: Debug [Active]

Specify output file name (--output_file, -o): "${ProjName}.out"

Link information (map) listed into <file> (--map_file, -m): "${ProjName}.map"

Heap size for C/C++ dynamic memory allocation (--heap_size, -heap):

Set C system stack size (--stack_size, -stack): 0x200

☑ Warn if an unspecified output section is created (--warn_sections, -w)

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

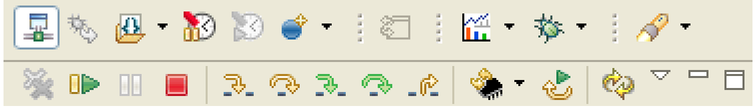
There are many linker options but these four handle all of the basic needs.

- `-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 CCS:



The common debugging and program execution descriptions are shown below:

Start debugging

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

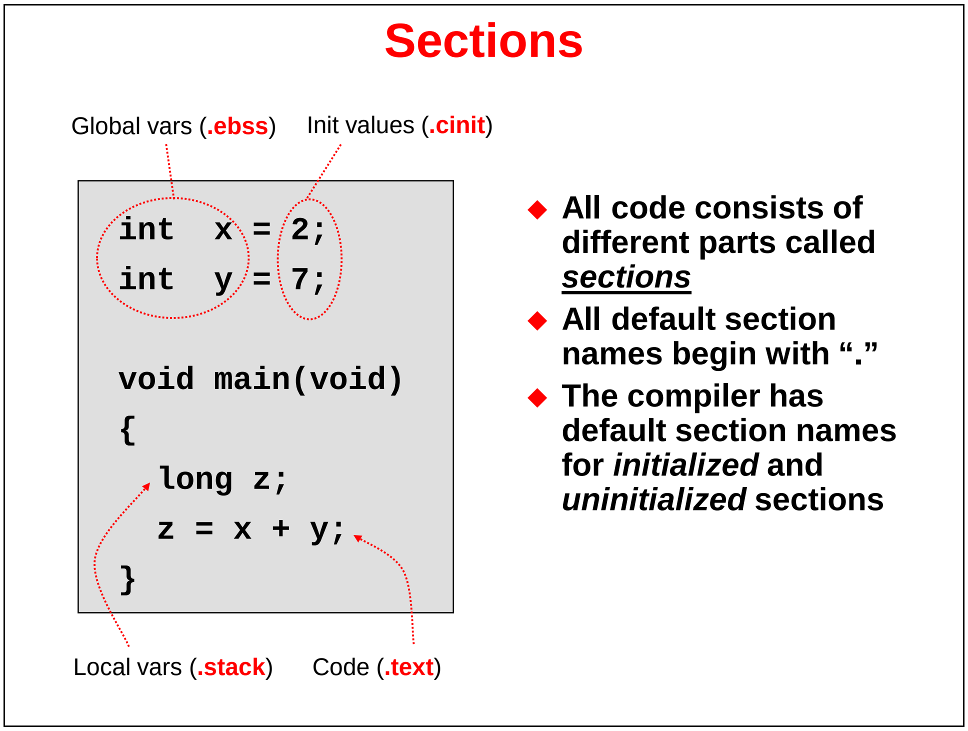
Program execution

Image	Name	Description	Availability
⏸	Halt	Halts the selected target. The rest of the debug views will update automatically with most recent target data.	Target Menu Debug View Toolbar
▶	Run	Resumes the execution of the currently loaded program from the current PC location. Execution continues until a breakpoint is encountered.	Target Menu Debug View Toolbar
⇨I	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
m	Go to Main	Runs the programs until the beginning of function main in 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 .txt.



Sections

Global vars (**.ebss**) Init values (**.cinit**)

```
int  x = 2;
int  y = 7;

void main(void)
{
  long z;
  z = x + y;
}
```

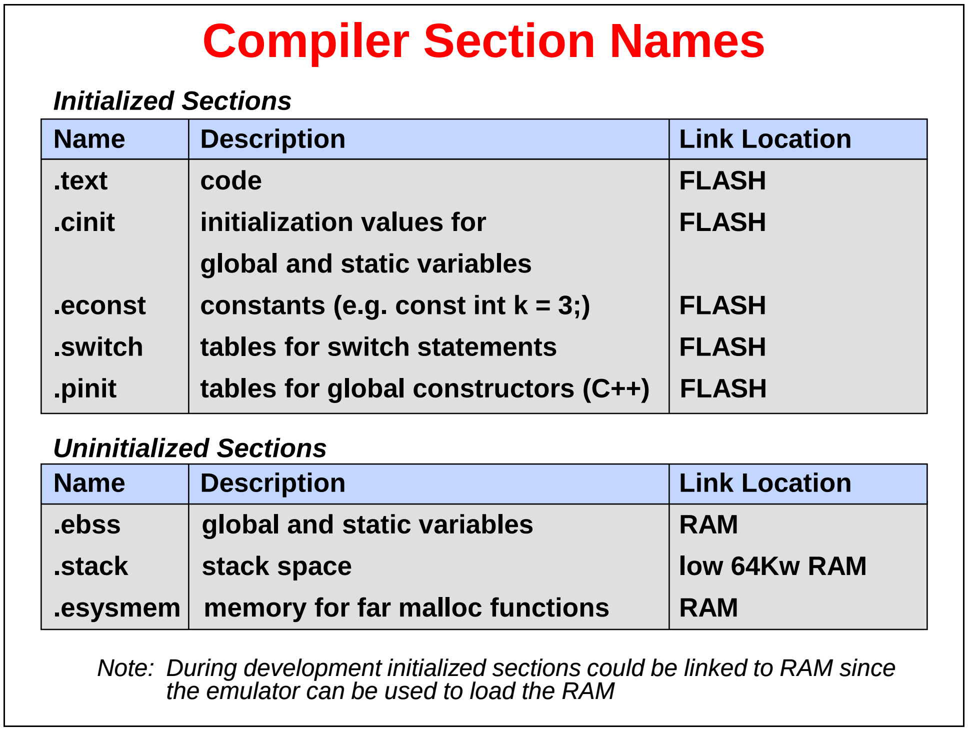
Local vars (**.stack**) Code (**.text**)

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

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

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

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

Uninitialized Sections

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

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

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

Program Code (.text)

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

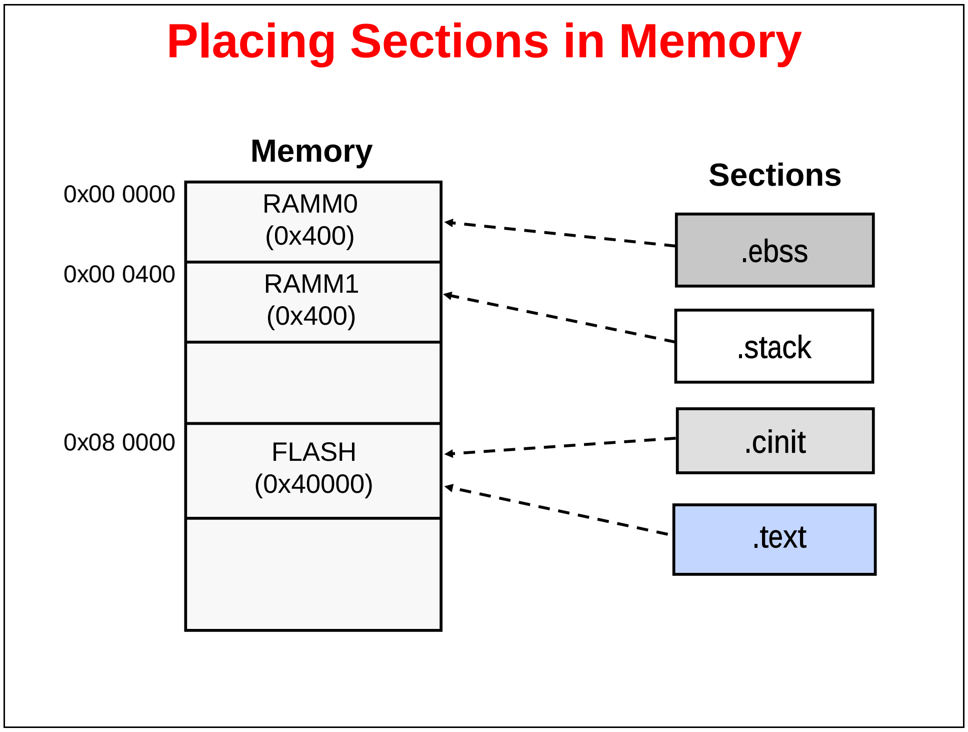
Constants (.cinit – initialized data)

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

Variables (.ebss – uninitialized data)

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

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 .txt, need to be placed into flash.



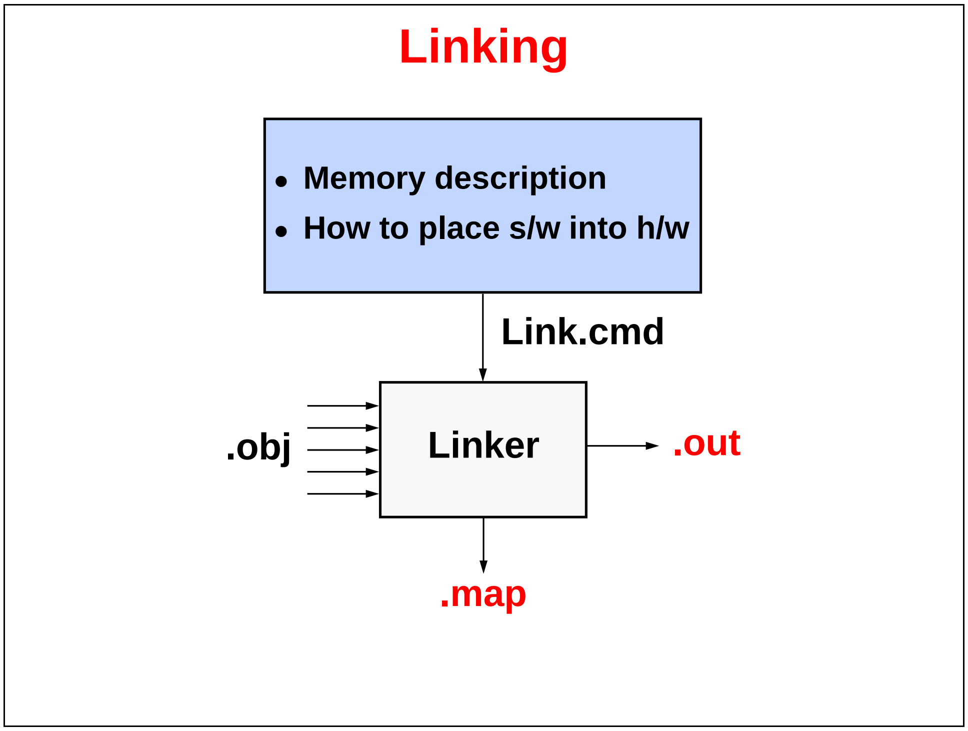
Placing Sections in Memory

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.



Linking

- Memory description
- How to place s/w into h/w

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:

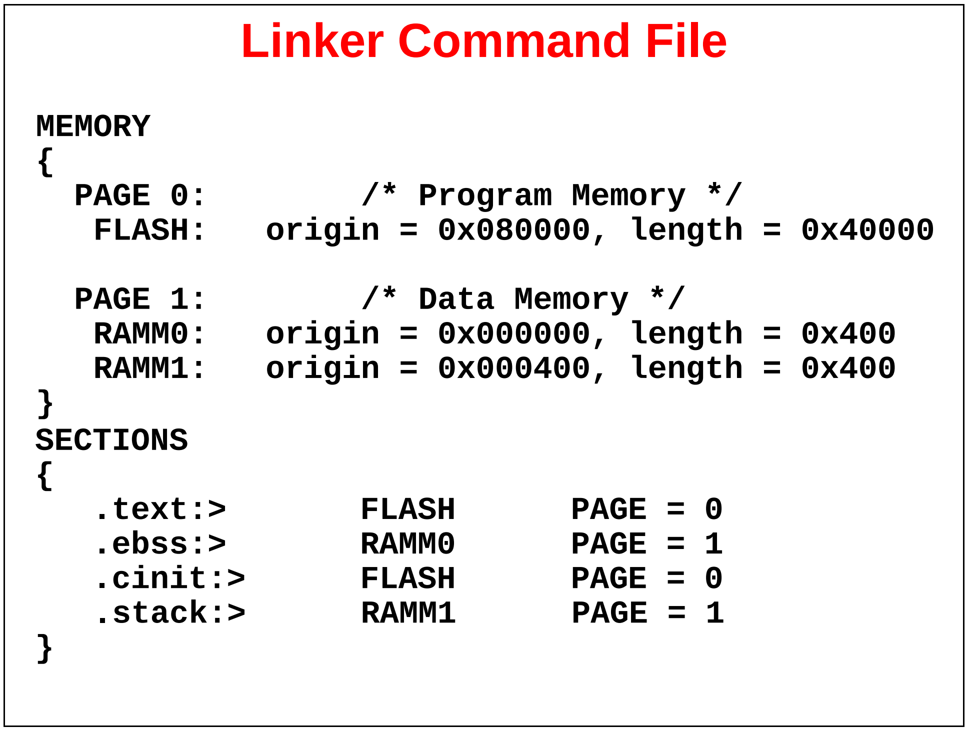
```
MEMORY
{
   FLASH:  origin = 0x080000 , length = 0x040000
}
```

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:

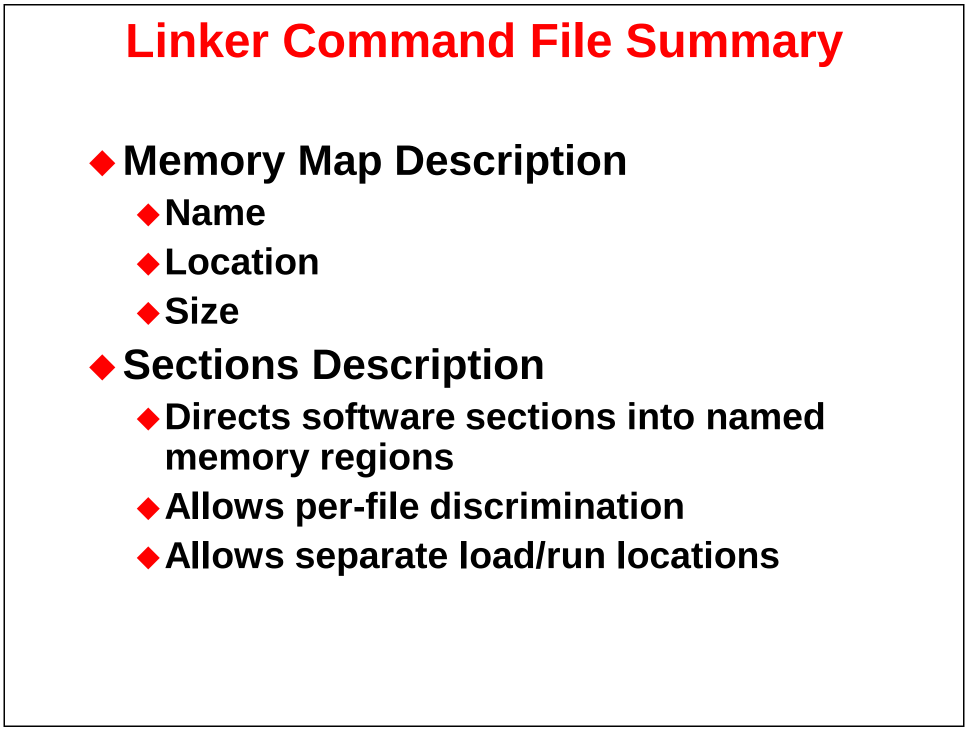
```
SECTIONS
{
   .text:>  FLASH       PAGE 0
   .ebss:>  RAMM0       PAGE 1
   .cinit:> FLASH       PAGE 0
   .stack:> RAMM1       PAGE 1
}
```

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:

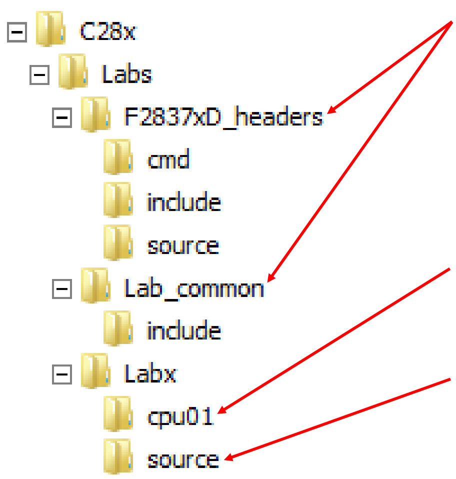


Linker Command File Summary

- **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 File Directory Structure

Lab File Directory Structure



Supporting Files

- Easier to make projects portable
- ${PROJECT_ROOT} provides an anchor point for paths to files that travel with the project
- Easier to maintain and update supporting files

Project Source Files

- All modified files are in the Project Folder

Other Source Files that are "Added" to the Project Folder

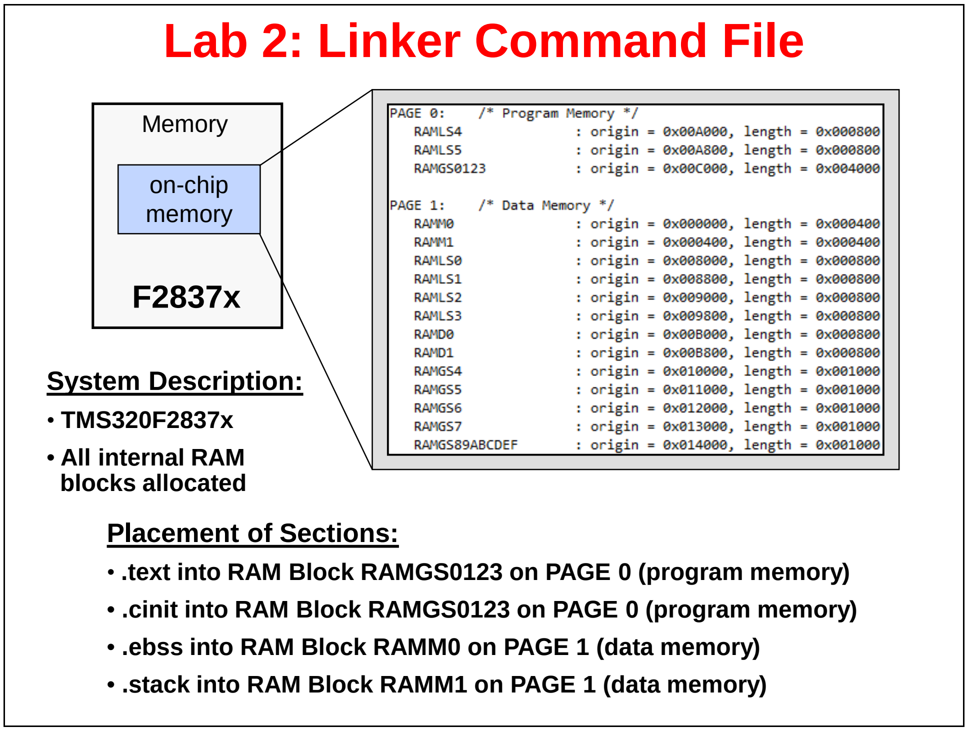
- Source files for multiple part lab exercises

Note: CCSv7 will automatically add ALL files contained in the folder where the project is created

Lab 2: Linker Command File

➢ Objective

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



Lab 2: Linker Command File

```
PAGE 0:    /* Program Memory */
   RAMLS4             : origin = 0x00A000, length = 0x000800
   RAMLS5             : origin = 0x00A800, length = 0x000800
   RAMGS0123          : origin = 0x00C000, length = 0x004000

PAGE 1:    /* Data Memory */
   RAMM0              : origin = 0x000000, length = 0x000400
   RAMM1              : origin = 0x000400, length = 0x000400
   RAMLS0             : origin = 0x008000, length = 0x000800
   RAMLS1             : origin = 0x008800, length = 0x000800
   RAMLS2             : origin = 0x009000, length = 0x000800
   RAMLS3             : origin = 0x009800, length = 0x000800
   RAMD0              : origin = 0x00B000, length = 0x000800
   RAMD1              : origin = 0x00B800, length = 0x000800
   RAMGS4             : origin = 0x010000, length = 0x001000
   RAMGS5             : origin = 0x011000, length = 0x001000
   RAMGS6             : origin = 0x012000, length = 0x001000
   RAMGS7             : origin = 0x013000, length = 0x001000
   RAMGS89ABCDEF      : origin = 0x014000, length = 0x001000
```

System Description:

- **TMS320F2837x**
- **All internal RAM blocks allocated**

Placement of Sections:

- **.text into RAM Block RAMGS0123 on PAGE 0 (program memory)**
- **.cinit into RAM Block RAMGS0123 on PAGE 0 (program memory)**
- **.ebss into RAM Block RAMM0 on PAGE 1 (data memory)**
- **.stack into RAM Block RAMM1 on PAGE 1 (data memory)**

➢ Initial Hardware Set Up

Note: The lab exercises in this workshop have been developed and targeted for the F28379D LaunchPad. Optionally, the F28379D Experimenter Kit can be used. Other F2807x or F2837xS development tool kits may be used and might require some minor modifications to the lab code and/or lab directions; however the Inter-Processor Communications lab exercise will require either the F28379D LaunchPad or the F28379D Experimenter Kit. Refer to Appendix A for additional information about the F28379D Experimenter Kit.

- **F28379D LaunchPad:**

Using the supplied USB cable – plug the USB Standard Type A connector into the computer USB port and the USB Mini 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.

At the beginning of the workshop, boot mode switch S1 position 3 must be set to "1 – ON". This will configure the device for emulation boot mode.

➢ 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 `OK`.

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

2. The first time CCS opens an introduction page appears. Close the page by clicking the `X` on the "Getting Started" tab. You should now have an empty workbench. The term "workbench" refers to the desktop development environment. Maximize CCS to fill your screen.

   The workbench will open in the CCS Edit perspective view. Notice the "CCS Edit" icon in the upper right-hand corner. A perspective defines the initial layout views of the workbench windows, toolbars, and menus which are appropriate for a specific type of task (i.e. code development or debugging). This minimizes clutter to the user interface. The CCS Edit perspective is used to create or build 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 **F2837xD.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 XDS100v2 USB Debug Probe". In the "Board or Device" box type **TMS320F28379D** to filter the options. In the box below, check the box to select "TMS320F28379D". Click `Save` to save the configuration, then close the "F2837xD.ccxml" setup window by clicking the `X` on the tab.

5. To view the target configurations, click:

   `View → Target Configurations`

   and click the plus sign (+) to the left of "User Defined". Notice that the F2837xD.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 "2837xD Delfino". In the pull-down list immediately to the right, choose the "TMS320F28379D".

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:\C28x\Labs\Lab2\cpu01`

   Click `OK`.

8. Next, open the "Advanced setting" section and set the "Linker command file" to "<none>". We will be using our own linker command file rather than the one supplied by CCS. Leave the "Runtime Support Library" set to "<automatic>". This will automatically select the "rts2800_fpu32.lib" runtime support library for floating-point devices.

9. Then, open the "Project templates and examples" section and select the "Empty Project" template. Click `Finish`.

10. A new project has now been created. Notice the Project Explorer window contains `Lab2`. 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're looking in `C:\C28x\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 plus sign (+) to the left of `Lab2` and notice that the files are listed.

Project Build Options

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

    `Project → Properties`

14. A "Properties" window will open and in the section on the left under "Build" be sure that the "C2000 Compiler" and "C2000 Linker" options are visible. Next, under "C2000 Linker" select the "`Basic Options`". Notice that .out and .map files are being specified. The .out file is the executable code that will be loaded into the MCU. The .map file will contain a linker report showing memory usage and section addresses in memory. Also notice the stack size is set to 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

`cla1`, 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 `vcu2`. Select `OK` to close the Properties window.

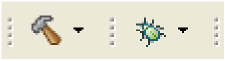
Linker Command File – Lab2.cmd

16. Open and inspect `Lab2.cmd` by double clicking on the filename in the Project Explorer window. Notice that the `Memory{}` declaration describes the system memory shown on the “Lab2: Linker Command File” slide in the objective section of this lab exercise. Memory blocks RAMLS4, RAMLS5 and RAMGS0123 have been placed in program memory on page 0, and the other memory blocks have been placed in data memory on page 1.

17. In the `Sections{}` area notice that the sections defined on the slide have been “linked” into the appropriate memories. Also, notice that a section called .reset has been allocated. The .reset section is part of the rts2800_fpu32.lib and is not needed. By putting the TYPE = DSECT modifier after its allocation the linker will ignore this section and not allocate it. Close the inspected file.

Build and Load the Project

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

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

19. Click the “`Build`” button and watch the tools run in the Console window. Check for errors in the Problems window (we have deliberately put an error in Lab2.c). When you get an error, you will see the error message in the Problems window. Expand the error by clicking on the plus sign (+) to the left of the “Errors”. Then simply double-click the error message. The editor will automatically open to the source file containing the error, with the code line highlighted with a 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`

    A Launching Debug Session window will open. Select only CPU1 to load the program on (i.e. ***uncheck*** CPU2), and then click `OK`.

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

Debug Environment Windows

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

23. Open a "Memory Browser" to view the global variable "`z`".

    Click: `View` → `Memory Browser` on the menu bar.

    Type **`&z`** into the address field, select "`Data`" memory page, and then <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

Peripherial Registers Header Files

Introduction

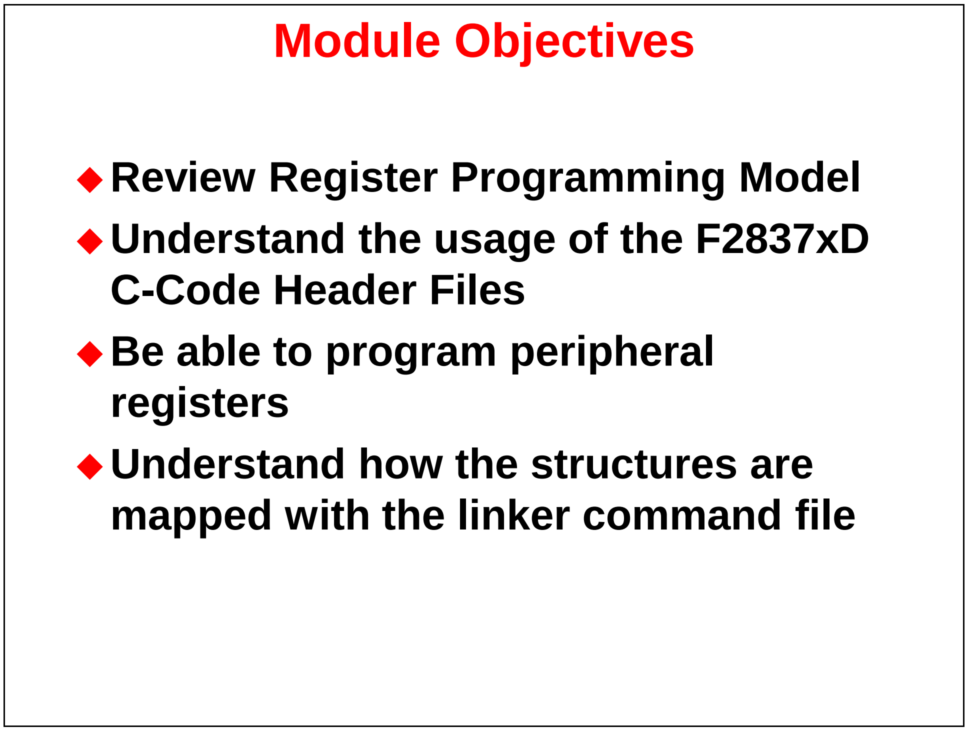
The purpose of the F2837xD C-code header files is to simplify the programming of the many peripherals on the F28x device. Typically, to program a peripheral the programmer needs to write the appropriate values to the different fields within a control register. In its simplest form, the process consists of writing a hex value (or masking a bit field) to the correct address in memory. But, since this can be a burdensome and repetitive task, the C-code header files were created to make this a less complicated task.

The F2837xD C-code header files are part of a library consisting of C functions, macros, peripheral structures, and variable definitions. Together, this set of files is known as the 'header files.'

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

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

Module Objectives

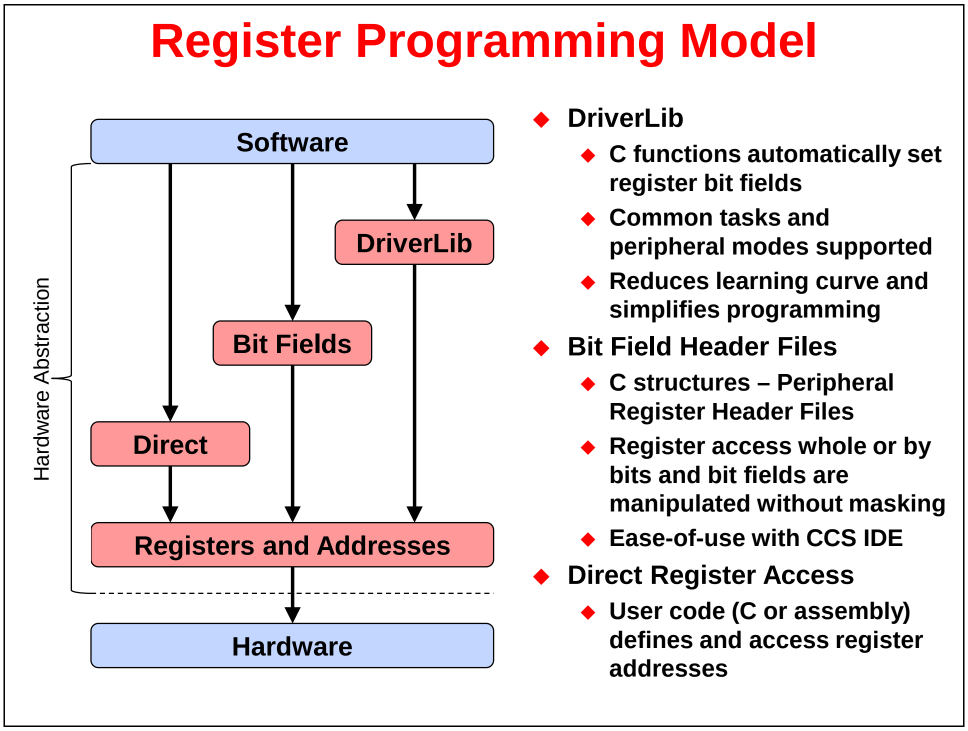


Module Objectives

- Review Register Programming Model
- Understand the usage of the F2837xD C-Code Header Files
- Be able to program peripheral registers
- Understand how the structures are mapped with the linker command file

Chapter Topics

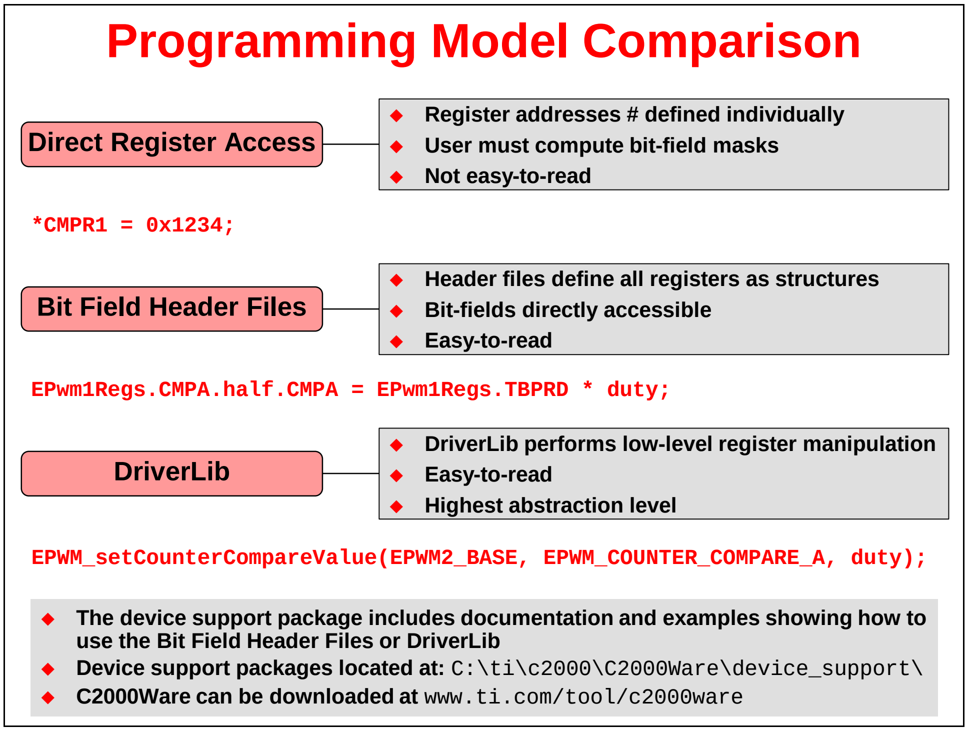
Register Programming Model



Register Programming Model

- DriverLib
  - C functions automatically set register bit fields
  - Common tasks and peripheral modes supported
  - Reduces learning curve and simplifies programming
- Bit Field Header Files
  - C structures – Peripheral Register Header Files
  - Register access whole or by bits and bit fields are manipulated without masking
  - Ease-of-use with CCS IDE
- Direct Register Access
  - User code (C or assembly) defines and access register addresses

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 the least 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 nice balance between ease of use and flexibility when working with Code Composer Studio. Direct register access is the lowest level where the user code, in C or assembly, defines and access register addresses.



Programming Model Comparison

Direct Register Access

- **Register addresses # defined individually**
- **User must compute bit-field masks**
- **Not easy-to-read**

```
*CMPR1 = 0x1234;
```

Bit Field Header Files

- **Header files define all registers as structures**
- **Bit-fields directly accessible**
- **Easy-to-read**

```
EPwm1Regs.CMPA.half.CMPA = EPwm1Regs.TBPRD * duty;
```

DriverLib

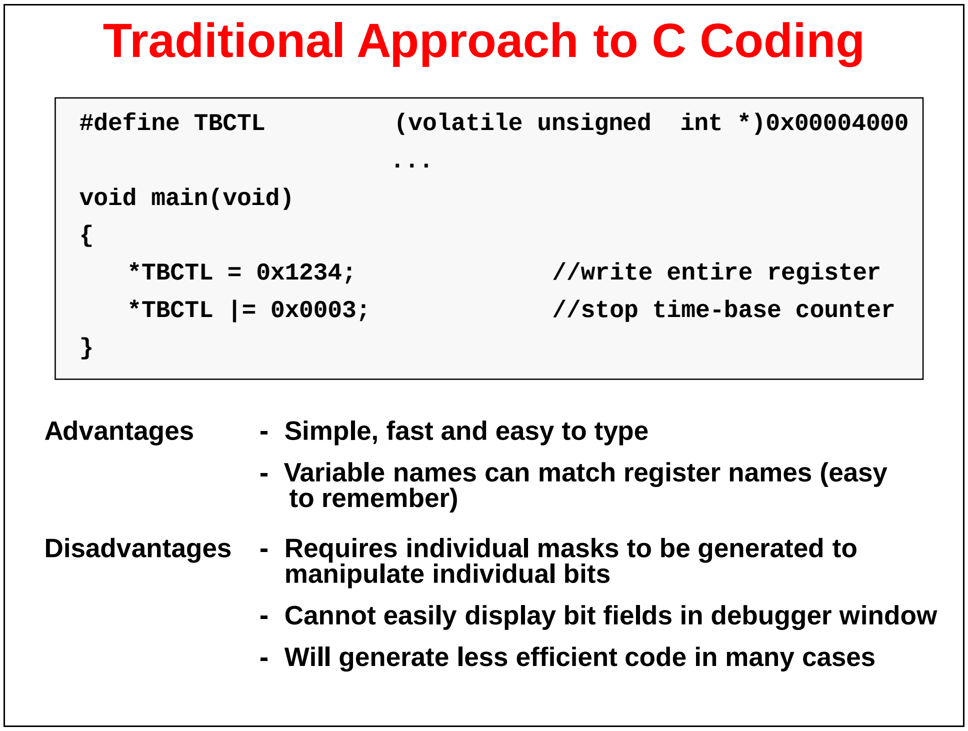
- **DriverLib performs low-level register manipulation**
- **Easy-to-read**
- **Highest abstraction level**

```
EPWM_setCounterCompareValue(EPWM2_BASE, EPWM_COUNTER_COMPARE_A, duty);
```

- **The device support package includes documentation and examples showing how to use the Bit Field Header Files or DriverLib**
- **Device support packages located at:** `C:\ti\c2000\C2000Ware\device_support\`
- **C2000Ware can be downloaded at** `www.ti.com/tool/c2000ware`

The above slide provides a comparison of each programming model, from the lowest level to the highest level of abstraction. With direct register access, 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 bit field header files, which provides a balance between ease of use and flexibility. Device support packages can be downloaded from www.ti.com.

Traditional and Structure Approach to C Coding



Traditional Approach to C Coding

```
#define TBCTL        (volatile unsigned  int *)0x00004000
                     ...
void main(void)
{
   *TBCTL = 0x1234;             //write entire register
   *TBCTL |= 0x0003;            //stop time-base counter
}
```

Advantages - **Simple, fast and easy to type**

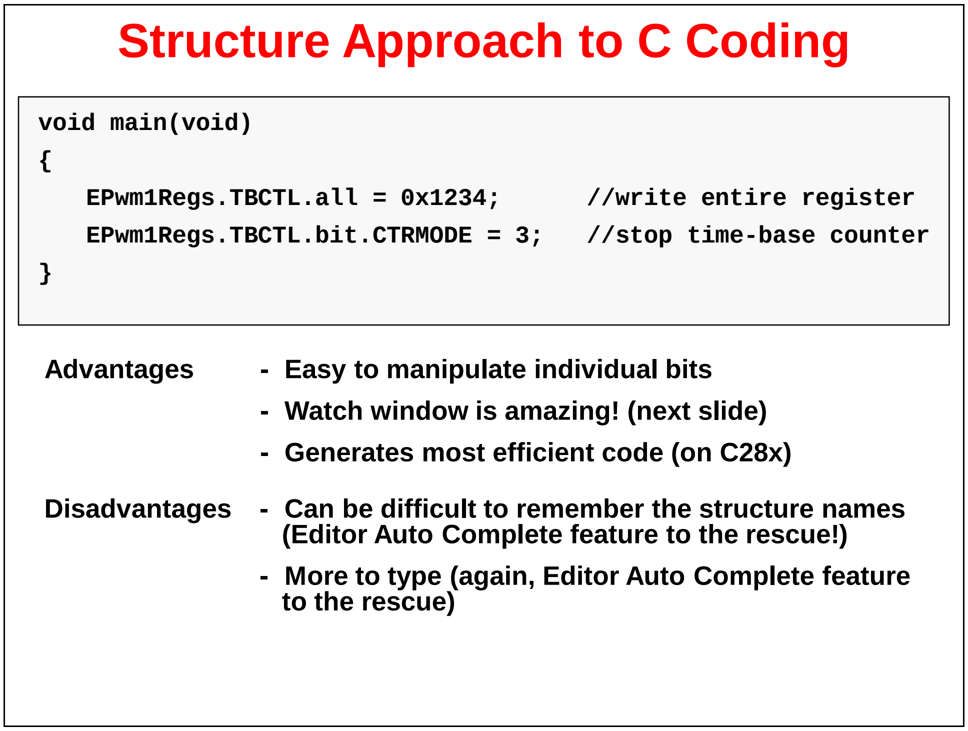
- **Variable names can match register names (easy to remember)**

Disadvantages - **Requires individual masks to be generated to manipulate individual bits**

- **Cannot easily display bit fields in debugger window**
- **Will generate less efficient code in many cases**

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

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



Structure Approach to C Coding

```
void main(void)
{
    EPwm1Regs.TBCTL.all = 0x1234;      //write entire register
    EPwm1Regs.TBCTL.bit.CTRMODE = 3;   //stop time-base counter
}
```

Advantages

- **Easy to manipulate individual bits**
- **Watch window is amazing! (next slide)**
- **Generates most efficient code (on C28x)**

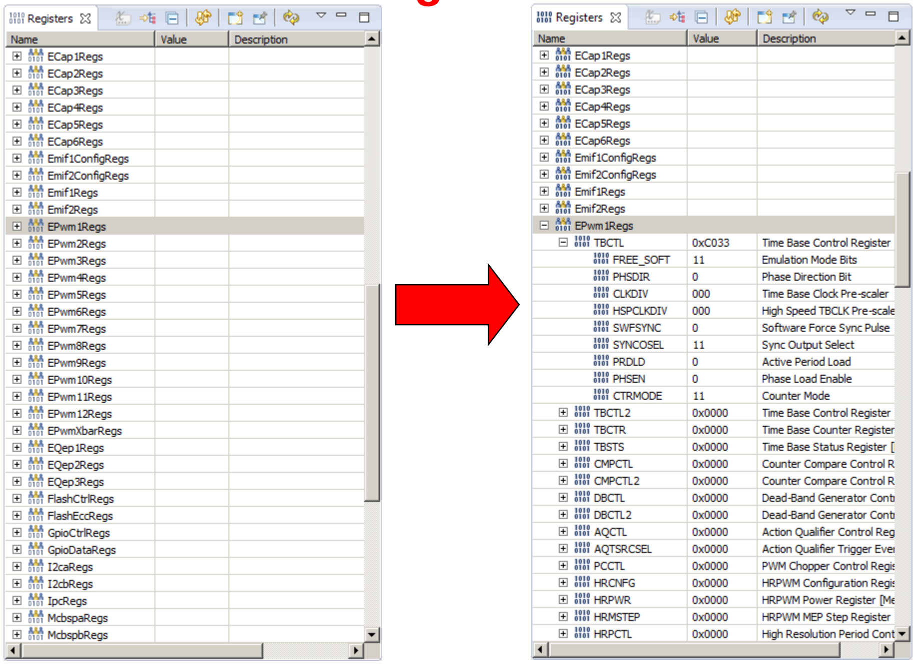
Disadvantages

- **Can be difficult to remember the structure names (Editor Auto Complete feature to the rescue!)**
- **More to type (again, Editor Auto Complete feature to the rescue)**

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

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

Built-in Register Window



Registers

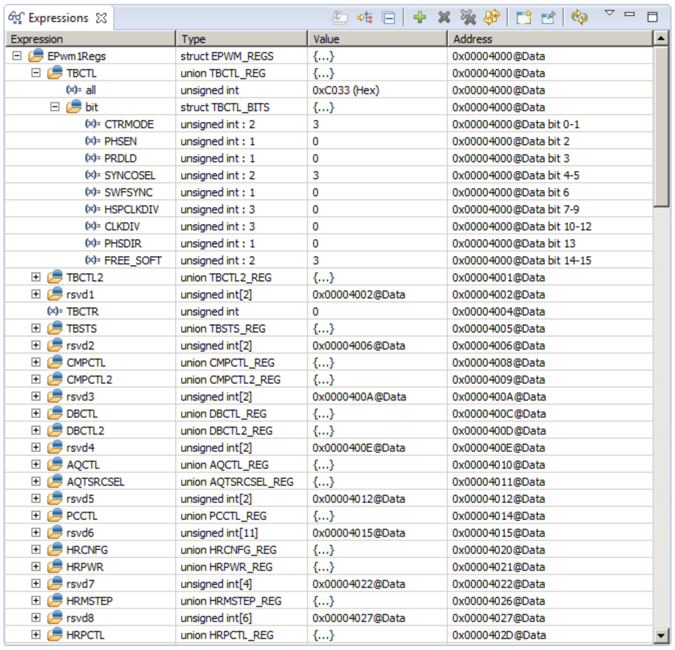
Name	Value	Description
ECap1Regs		
ECap2Regs		
ECap3Regs		
ECap4Regs		
ECap5Regs		
ECap6Regs		
Emif1ConfigRegs		
Emif2ConfigRegs		
Emif1Regs		
Emif2Regs		
EPwm1Regs		
EPwm2Regs		
EPwm3Regs		
EPwm4Regs		
EPwm5Regs		
EPwm6Regs		
EPwm7Regs		
EPwm8Regs		
EPwm9Regs		
EPwm10Regs		
EPwm11Regs		
EPwm12Regs		
EPwmXbarRegs		
EQep1Regs		
EQep2Regs		
EQep3Regs		
FlashCtrlRegs		
FlashEccRegs		
GpioCtrlRegs		
GpioDataRegs		
I2caRegs		
I2cbRegs		
IpcRegs		
McbspaRegs		
McbspbRegs		

Registers

Name	Value	Description
ECap1Regs		
ECap2Regs		
ECap3Regs		
ECap4Regs		
ECap5Regs		
ECap6Regs		
Emif1ConfigRegs		
Emif2ConfigRegs		
Emif1Regs		
Emif2Regs		
EPwm1Regs		
TBCTL	0xC033	Time Base Control Register
FREE_SOFT	11	Emulation Mode Bits
PHSDIR	0	Phase Direction Bit
CLKDIV	000	Time Base Clock Pre-scaler
HSPCLKDIV	000	High Speed TBCLK Pre-scale
SWFSYNC	0	Software Force Sync Pulse
SYNCOSEL	11	Sync Output Select
PRDLD	0	Active Period Load
PHSEN	0	Phase Load Enable
CTRMODE	11	Counter Mode
TBCTL2	0x0000	Time Base Control Register
TBCTR	0x0000	Time Base Counter Register
TBSTS	0x0000	Time Base Status Register [
CMPCTL	0x0000	Counter Compare Control R
CMPCTL2	0x0000	Counter Compare Control R
DBCTL	0x0000	Dead-Band Generator Conti
DBCTL2	0x0000	Dead-Band Generator Conti
AQCTL	0x0000	Action Qualifier Control Reg
AQTSRCSEL	0x0000	Action Qualifier Trigger Evei
PCCTL	0x0000	PWM Chopper Control Regis
HRCNFG	0x0000	HRPWM Configuration Regis
HRPWR	0x0000	HRPWM Power Register [Me
HRMSTEP	0x0000	HRPWM MEP Step Register
HRPCTL	0x0000	High Resolution Period Cont

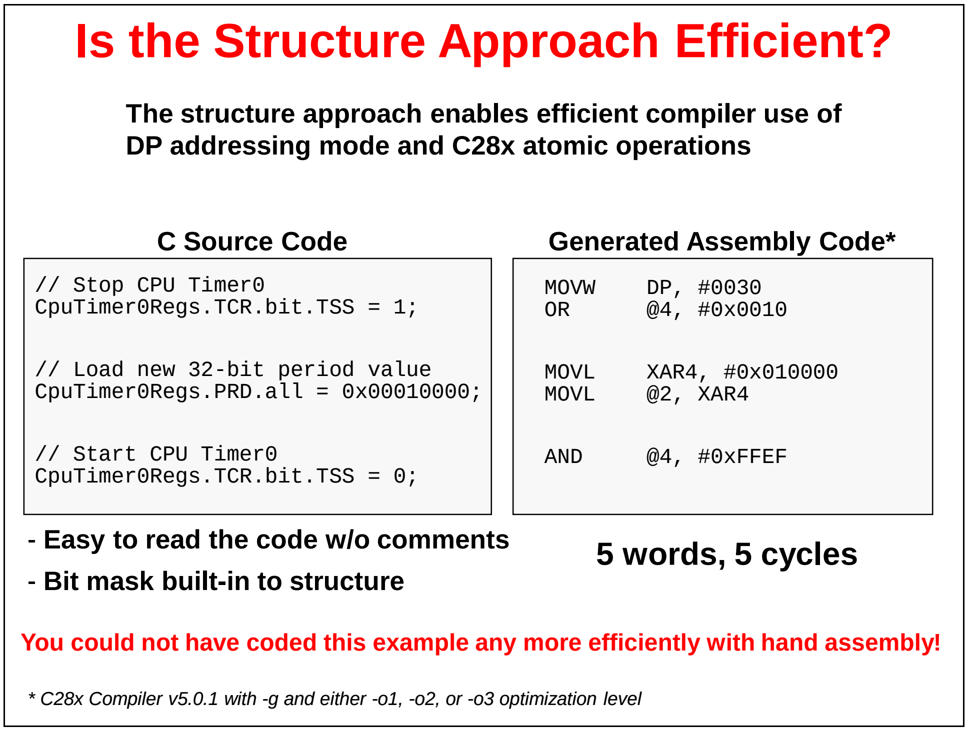
Register values can be viewed using the built-in Register Window. Also, the peripheral can be added to the expression window. In addition to viewing register values, individual bit fields can be modified. There is no need to refer to the reference guide to identify the bit field settings.

Expressions Window using Structures



Expressions

Expression	Type	Value	Address
EPwm1Regs	struct EPWM_REGS	{...}	0x00004000@Data
TBCTL	union TBCTL_REG	{...}	0x00004000@Data
all	unsigned int	0xC033 (Hex)	0x00004000@Data
bit	struct TBCTL_BITS	{...}	0x00004000@Data
CTRMODE	unsigned int : 2	3	0x00004000@Data bit 0-1
PHSEN	unsigned int : 1	0	0x00004000@Data bit 2
PRDLD	unsigned int : 1	0	0x00004000@Data bit 3
SYNCOSEL	unsigned int : 2	3	0x00004000@Data bit 4-5
SWFSYNC	unsigned int : 1	0	0x00004000@Data bit 6
HSPCLKDIV	unsigned int : 3	0	0x00004000@Data bit 7-9
CLKDIV	unsigned int : 3	0	0x00004000@Data bit 10-12
PHSDIR	unsigned int : 1	0	0x00004000@Data bit 13
FREE_SOFT	unsigned int : 2	3	0x00004000@Data bit 14-15
TBCTL2	union TBCTL2_REG	{...}	0x00004001@Data
rsvd1	unsigned int[2]	0x00004002@Data	0x00004002@Data
TBCTR	unsigned int	0	0x00004004@Data
TBSTS	union TBSTS_REG	{...}	0x00004005@Data
rsvd2	unsigned int[2]	0x00004006@Data	0x00004006@Data
CMPCTL	union CMPCTL_REG	{...}	0x00004008@Data
CMPCTL2	union CMPCTL2_REG	{...}	0x00004009@Data
rsvd3	unsigned int[2]	0x0000400A@Data	0x0000400A@Data
DBCTL	union DBCTL_REG	{...}	0x0000400C@Data
DBCTL2	union DBCTL2_REG	{...}	0x0000400D@Data
rsvd4	unsigned int[2]	0x0000400E@Data	0x0000400E@Data
AQCTL	union AQCTL_REG	{...}	0x00004010@Data
AQTSRCSEL	union AQTSRCSEL_REG	{...}	0x00004011@Data
rsvd5	unsigned int[2]	0x00004012@Data	0x00004012@Data
PCCTL	union PCCTL_REG	{...}	0x00004014@Data
rsvd6	unsigned int[11]	0x00004015@Data	0x00004015@Data
HRCNFG	union HRCNFG_REG	{...}	0x00004020@Data
HRPWR	union HRPWR_REG	{...}	0x00004021@Data
rsvd7	unsigned int[4]	0x00004022@Data	0x00004022@Data
HRMSTEP	union HRMSTEP_REG	{...}	0x00004026@Data
rsvd8	unsigned int[6]	0x00004027@Data	0x00004027@Data
HRPCTL	union HRPCTL_REG	{...}	0x0000402D@Data



Is the Structure Approach Efficient?

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

C Source Code

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


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


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

Generated Assembly Code*

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

MOVL    XAR4, #0x010000
MOVL    @2, XAR4

AND     @4, #0xFFEF
```

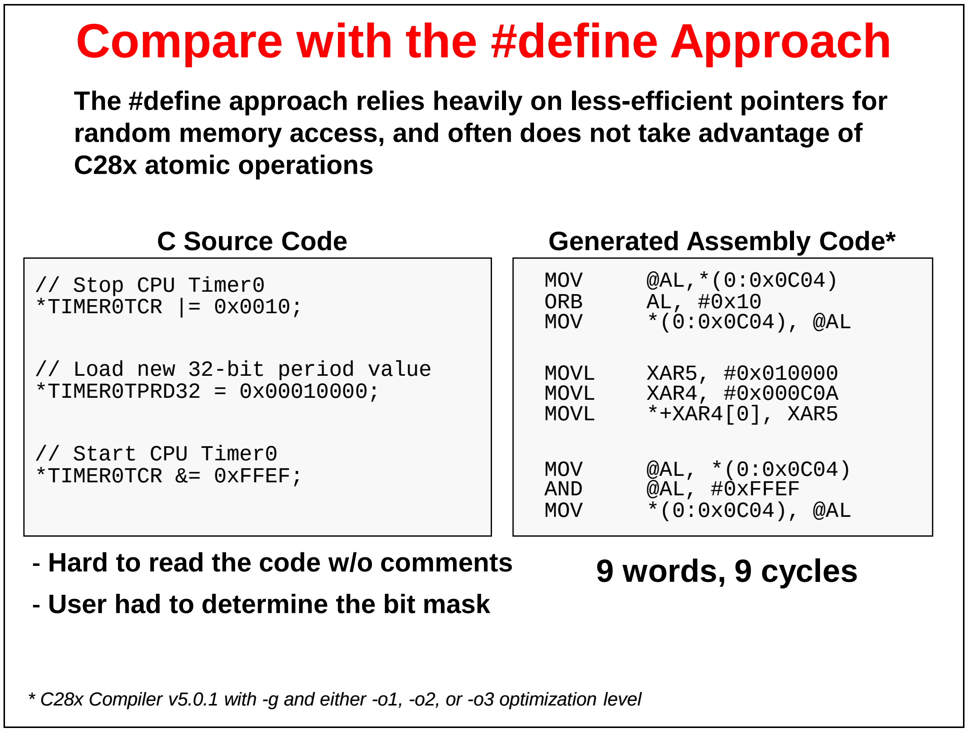
- Easy to read the code w/o comments

- Bit mask built-in to structure

5 words, 5 cycles

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

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



Compare with the #define Approach

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

C Source Code

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


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


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

Generated Assembly Code*

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

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

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

- Hard to read the code w/o comments

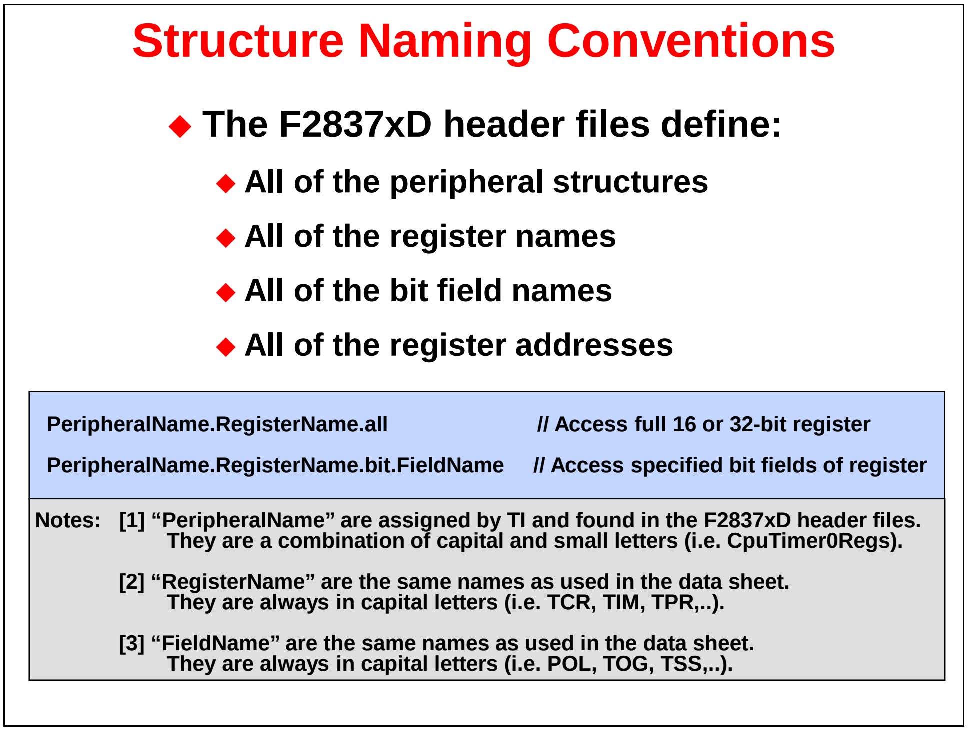
- User had to determine the bit mask

9 words, 9 cycles

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

Naming Conventions

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



Structure Naming Conventions

- **The F2837xD header files define:**
  - **All of the peripheral structures**
  - **All of the register names**
  - **All of the bit field names**
  - **All of the register addresses**

```
PeripheralName.RegisterName.all              // Access full 16 or 32-bit register
PeripheralName.RegisterName.bit.FieldName    // Access specified bit fields of register
```

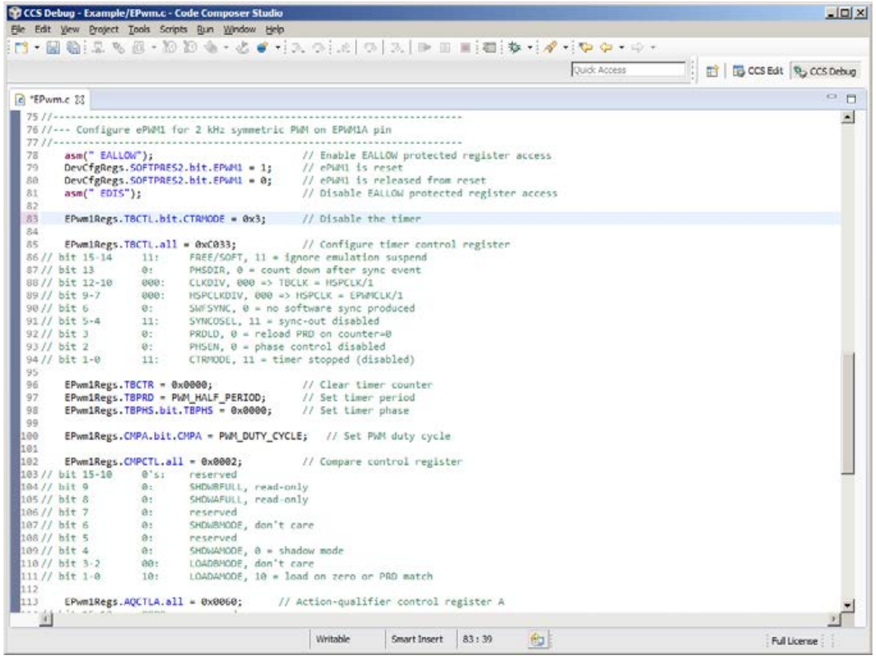
Notes: [1] "PeripheralName" are assigned by TI and found in the F2837xD header files. They are a combination of capital and small letters (i.e. CpuTimer0Regs).

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

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

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

Editor Auto Complete to the Rescue!

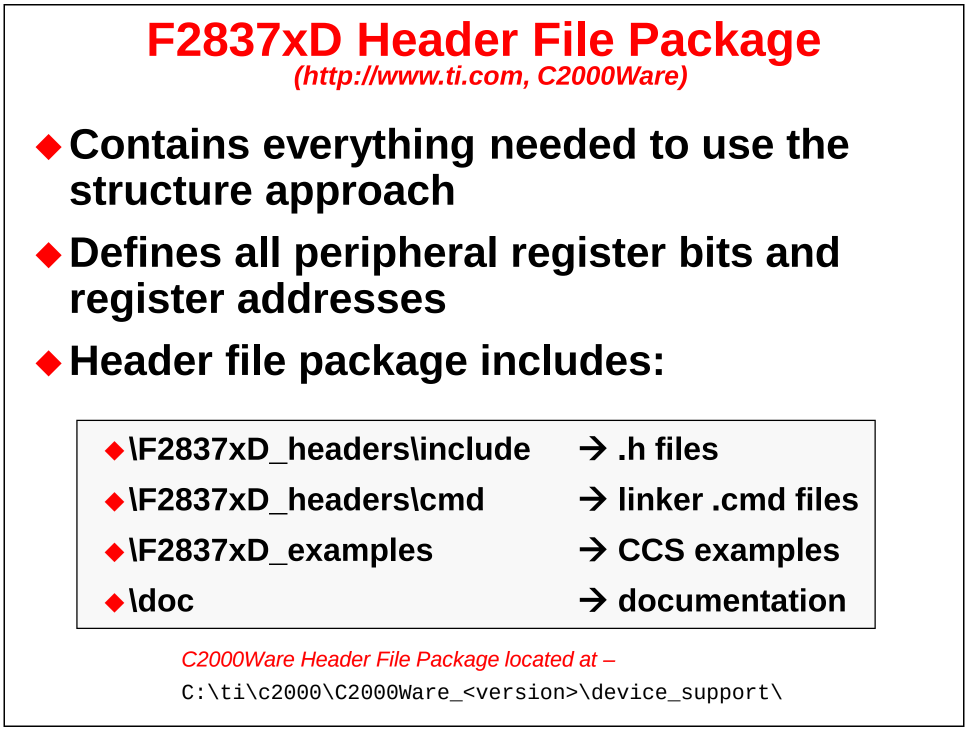

CCS Debug - Example/EPwm.c - Code Composer Studio
File Edit View Project Tools Scripts Run Window Help
Quick Access
CCS Edit
CCS Debug
*EPwm.c
Writable
Smart Insert
83 : 39
Full License

```
//---------------------------------------------------------------------
//--- Configure ePWM1 for 2 kHz symmetric PWM on EPWM1A pin
//---------------------------------------------------------------------
    asm(" EALLOW");                          // Enable EALLOW protected register access
    DevCfgRegs.SOFTPRES2.bit.EPWM1 = 1;      // ePWM1 is reset
    DevCfgRegs.SOFTPRES2.bit.EPWM1 = 0;      // ePWM1 is released from reset
    asm(" EDIS");                            // Disable EALLOW protected register access

    EPwm1Regs.TBCTL.bit.CTRMODE = 0x3;       // Disable the timer

    EPwm1Regs.TBCTL.all = 0xC033;            // Configure timer control register
// bit 15-14     11:     FREE/SOFT, 11 = ignore emulation suspend
// bit 13        0:      PHSDIR, 0 = count down after sync event
// bit 12-10     000:    CLKDIV, 000 => TBCLK = HSPCLK/1
// bit 9-7       000:    HSPCLKDIV, 000 => HSPCLK = EPWMCLK/1
// bit 6         0:      SWFSYNC, 0 = no software sync produced
// bit 5-4       11:     SYNCOSEL, 11 = sync-out disabled
// bit 3         0:      PRDLD, 0 = reload PRD on counter=0
// bit 2         0:      PHSEN, 0 = phase control disabled
// bit 1-0       11:     CTRMODE, 11 = timer stopped (disabled)

    EPwm1Regs.TBCTR = 0x0000;                // Clear timer counter
    EPwm1Regs.TBPRD = PWM_HALF_PERIOD;       // Set timer period
    EPwm1Regs.TBPHS.bit.TBPHS = 0x0000;      // Set timer phase

    EPwm1Regs.CMPA.bit.CMPA = PWM_DUTY_CYCLE;   // Set PWM duty cycle

    EPwm1Regs.CMPCTL.all = 0x0002;           // Compare control register
// bit 15-10     0's:    reserved
// bit 9         0:      SHDWBFULL, read-only
// bit 8         0:      SHDWAFULL, read-only
// bit 7         0:      reserved
// bit 6         0:      SHDWBMODE, don't care
// bit 5         0:      reserved
// bit 4         0:      SHDWAMODE, 0 = shadow mode
// bit 3-2       00:     LOADBMODE, don't care
// bit 1-0       10:     LOADAMODE, 10 = load on zero or PRD match

    EPwm1Regs.AQCTLA.all = 0x0060;        // Action-qualifier control register A
```

The editor auto complete feature works as follows. First, you type EPwm1Regs. Then, when you type a "." a window opens up, allowing you to select a control register. In this example TBCTL is selected. Then, when you type the "." a window opens up, allowing you to select "all" or "bit". In this example "bit" is selected. Then, when you type the "." a window opens up, allowing you to select a bit field. In this example CTRMODE is selected. And now, the structure is completed.

F2837xD C-Code Header Files

The F2837xD header file package contains everything needed to use the structure approach. It defines all the peripheral register bits and register addresses. The header file package includes the header files, linker command files, code examples, and documentation. The header file package is available from C2000Ware.



F2837xD Header File Package

(http://www.ti.com, C2000Ware)

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

◆ \F2837xD_headers\include	→ .h files
◆ \F2837xD_headers\cmd	→ linker .cmd files
◆ \F2837xD_examples	→ CCS examples
◆ \doc	→ documentation

C2000Ware Header File Package located at –

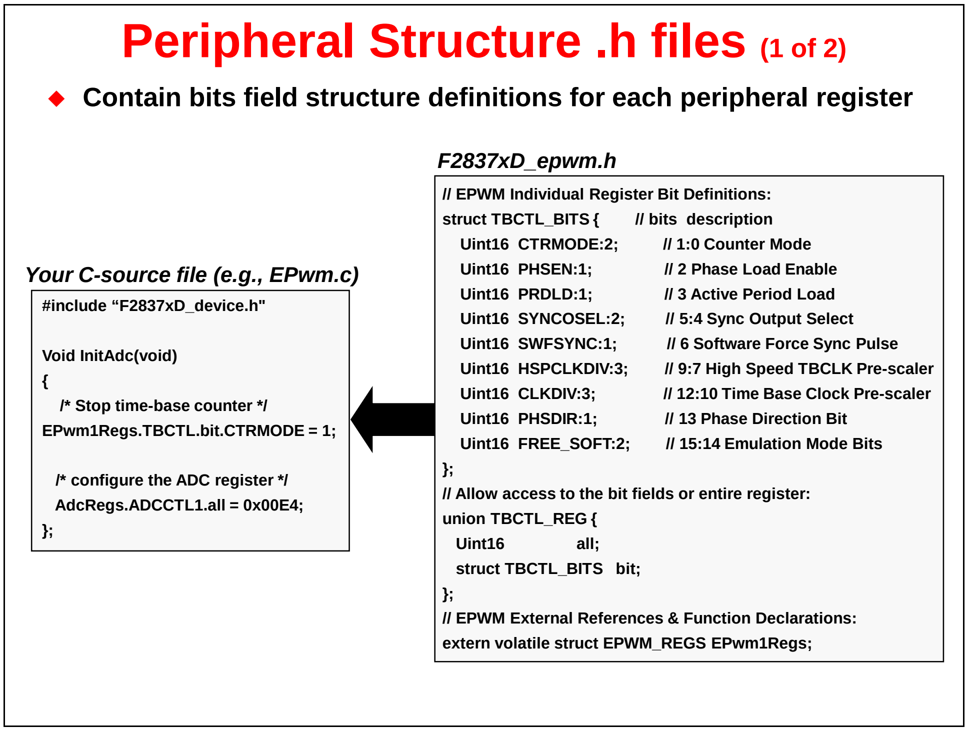
`C:\ti\c2000\C2000Ware_<version>\device_support\`

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

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

Peripheral Structure .h File

The F2837xD_Device.h header file is the main include file. By including this file in the .c source code, all of the peripheral specific .h header files are automatically included. Of course, each specific .h header file can be included individually in an application that does not use all the header files, or you can comment out the ones you do not need. (Also includes typedef statements).



Peripheral Structure .h files (1 of 2)

- Contain bits field structure definitions for each peripheral register

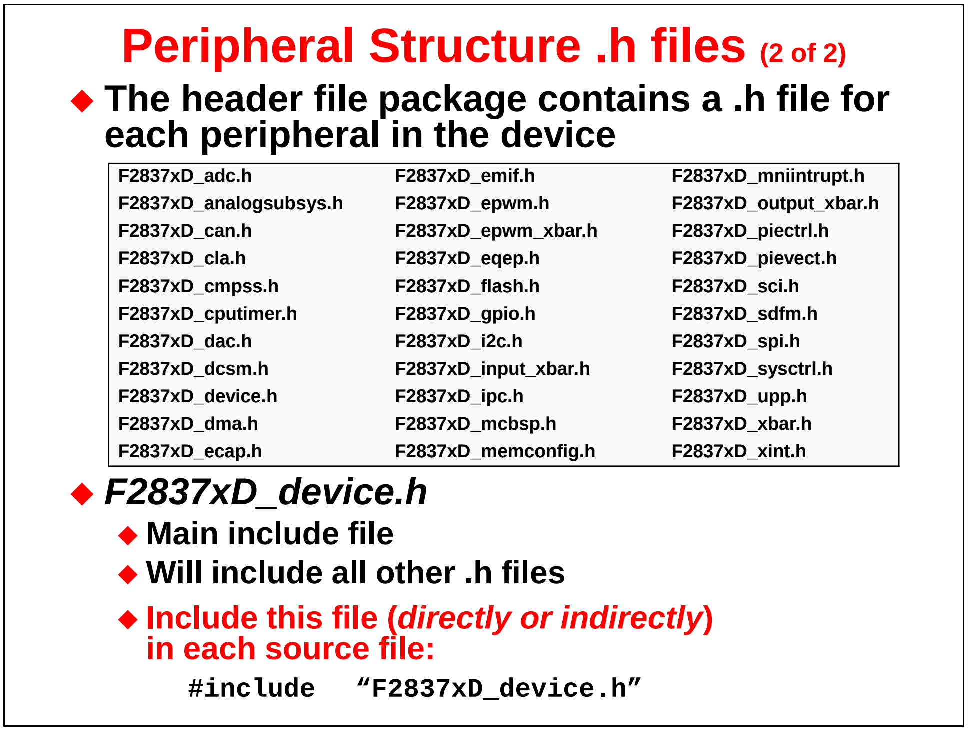
Your C-source file (e.g., EPwm.c)

```
#include “F2837xD_device.h"

Void InitAdc(void)
{
   /* Stop time-base counter */
EPwm1Regs.TBCTL.bit.CTRMODE = 1;

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

F2837xD_epwm.h

```
// EPWM Individual Register Bit Definitions:
struct TBCTL_BITS {        // bits  description
   Uint16  CTRMODE:2;        // 1:0 Counter Mode
   Uint16  PHSEN:1;          // 2 Phase Load Enable
   Uint16  PRDLD:1;          // 3 Active Period Load
   Uint16  SYNCOSEL:2;       // 5:4 Sync Output Select
   Uint16  SWFSYNC:1;        // 6 Software Force Sync Pulse
   Uint16  HSPCLKDIV:3;      // 9:7 High Speed TBCLK Pre-scaler
   Uint16  CLKDIV:3;         // 12:10 Time Base Clock Pre-scaler
   Uint16  PHSDIR:1;         // 13 Phase Direction Bit
   Uint16  FREE_SOFT:2;      // 15:14 Emulation Mode Bits
};
// Allow access to the bit fields or entire register:
union TBCTL_REG {
  Uint16              all;
  struct TBCTL_BITS   bit;
};
// EPWM External References & Function Declarations:
extern volatile struct EPWM_REGS EPwm1Regs;
```

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



Peripheral Structure .h files (2 of 2)

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

F2837xD_adc.h
F2837xD_analogsubsys.h
F2837xD_can.h
F2837xD_cla.h
F2837xD_cmpss.h
F2837xD_cputimer.h
F2837xD_dac.h
F2837xD_dcsm.h
F2837xD_device.h
F2837xD_dma.h
F2837xD_ecap.h
F2837xD_emif.h
F2837xD_epwm.h
F2837xD_epwm_xbar.h
F2837xD_eqep.h
F2837xD_flash.h
F2837xD_gpio.h
F2837xD_i2c.h
F2837xD_input_xbar.h
F2837xD_ipc.h
F2837xD_mcbsp.h
F2837xD_memconfig.h
F2837xD_mniintrupt.h
F2837xD_output_xbar.h
F2837xD_piectrl.h
F2837xD_pievect.h
F2837xD_sci.h
F2837xD_sdfm.h
F2837xD_spi.h
F2837xD_sysctrl.h
F2837xD_upp.h
F2837xD_xbar.h
F2837xD_xint.h

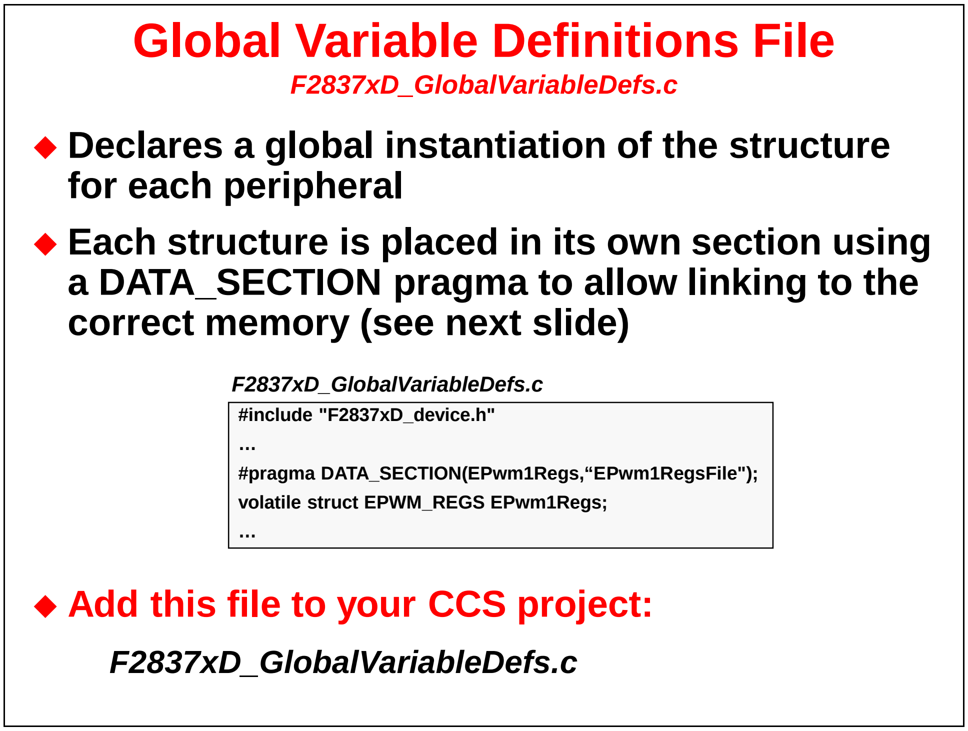
- ***F2837xD_device.h***
  - Main include file
  - Will include all other .h files
  - Include this file (*directly or indirectly*) in each source file:

```
#include   “F2837xD_device.h”
```

The header file package contains a .h file for each peripheral in the device. The F2837xD_Device.h file is the main include file. It will include all of the other .h files. There are three steps needed to use the header files. The first step is to include this file directly or indirectly in each source files.

Global Variable Definitions File

With F2837xD_GlobalVariableDefs.c included in the project all the needed variable definitions are globally defined.



Global Variable Definitions File

F2837xD_GlobalVariableDefs.c

- Declares a global instantiation of the structure for each peripheral
- Each structure is placed in its own section using a DATA_SECTION pragma to allow linking to the correct memory (see next slide)

F2837xD_GlobalVariableDefs.c

```
#include "F2837xD_device.h"
…
#pragma DATA_SECTION(EPwm1Regs,“EPwm1RegsFile");
volatile struct EPWM_REGS EPwm1Regs;
…
```

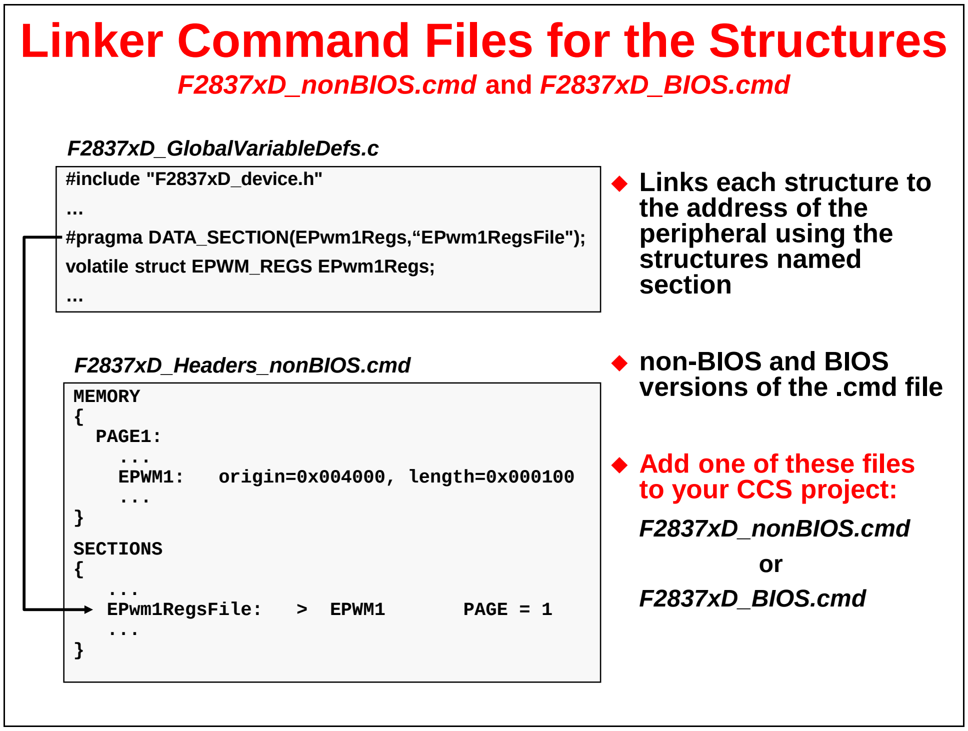
- Add this file to your CCS project:

F2837xD_GlobalVariableDefs.c

The global variable definition file declares a global instantiation of the structure for each peripheral. Each structure is placed in its own section using a DATA_SECTION pragma to allow linking to the correct memory. The second step for using the header files is to add F2837xD_GlobalVariableDefs.c file to your project.

Mapping Structures to Memory

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



Linker Command Files for the Structures

F2837xD_nonBIOS.cmd* and *F2837xD_BIOS.cmd

F2837xD_GlobalVariableDefs.c

```
#include "F2837xD_device.h"
...
#pragma DATA_SECTION(EPwm1Regs,“EPwm1RegsFile");
volatile struct EPWM_REGS EPwm1Regs;
...
```

F2837xD_Headers_nonBIOS.cmd

```
MEMORY
{
  PAGE1:
    ...
    EPWM1:   origin=0x004000, length=0x000100
    ...
}
SECTIONS
{
   ...
   EPwm1RegsFile:   >  EPWM1       PAGE = 1
   ...
}
```

- **Links each structure to the address of the peripheral using the structures named section**
- **non-BIOS and BIOS versions of the .cmd file**
- **Add one of these files to your CCS project:**

F2837xD_nonBIOS.cmd

or

F2837xD_BIOS.cmd

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

Linker Command File

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

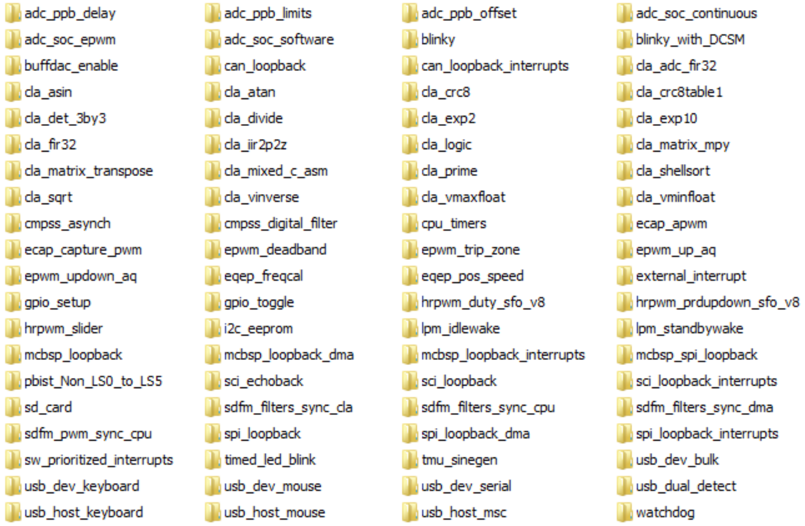
The user can modify their own linker command file, or use a pre-configured linker command file such as F28075.cmd. This file has the peripheral memory regions defined and tied to the individual peripheral.

Peripheral Specific Routines

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

Peripheral Specific Examples

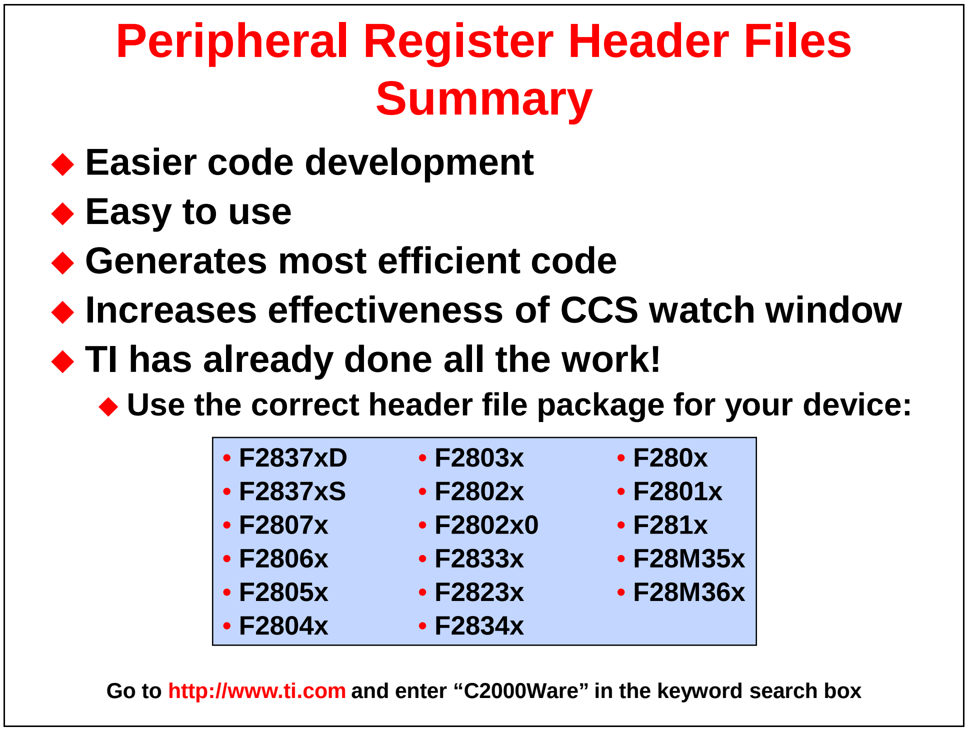
- Example projects for each peripheral
- Helpful to get you started



adc_ppb_delay
adc_ppb_limits
adc_ppb_offset
adc_soc_continuous
adc_soc_epwm
adc_soc_software
blinky
blinky_with_DCSM
buffdac_enable
can_loopback
can_loopback_interrupts
cla_adc_fir32
cla_asin
cla_atan
cla_crc8
cla_crc8table1
cla_det_3by3
cla_divide
cla_exp2
cla_exp10
cla_fir32
cla_iir2p2z
cla_logic
cla_matrix_mpy
cla_matrix_transpose
cla_mixed_c_asm
cla_prime
cla_shellsort
cla_sqrt
cla_vinverse
cla_vmaxfloat
cla_vminfloat
cmpss_asynch
cmpss_digital_filter
cpu_timers
ecap_apwm
ecap_capture_pwm
epwm_deadband
epwm_trip_zone
epwm_up_aq
epwm_updown_aq
eqep_freqcal
eqep_pos_speed
external_interrupt
gpio_setup
gpio_toggle
hrpwm_duty_sfo_v8
hrpwm_prdupdown_sfo_v8
hrpwm_slider
i2c_eeprom
lpm_idlewake
lpm_standbywake
mcbsp_loopback
mcbsp_loopback_dma
mcbsp_loopback_interrupts
mcbsp_spi_loopback
pbist_Non_LS0_to_LS5
sci_echoback
sci_loopback
sci_loopback_interrupts
sd_card
sdfm_filters_sync_cla
sdfm_filters_sync_cpu
sdfm_filters_sync_dma
sdfm_pwm_sync_cpu
spi_loopback
spi_loopback_dma
spi_loopback_interrupts
sw_prioritized_interrupts
timed_led_blink
tmu_sinegen
usb_dev_bulk
usb_dev_keyboard
usb_dev_mouse
usb_dev_serial
usb_dual_detect
usb_host_keyboard
usb_host_mouse
usb_host_msc
watchdog

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

Summary



Peripheral Register Header Files Summary

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

- F2837xD
- F2837xS
- F2807x
- F2806x
- F2805x
- F2804x
- F2803x
- F2802x
- F2802x0
- F2833x
- F2823x
- F2834x
- F280x
- F2801x
- F281x
- F28M35x
- F28M36x

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

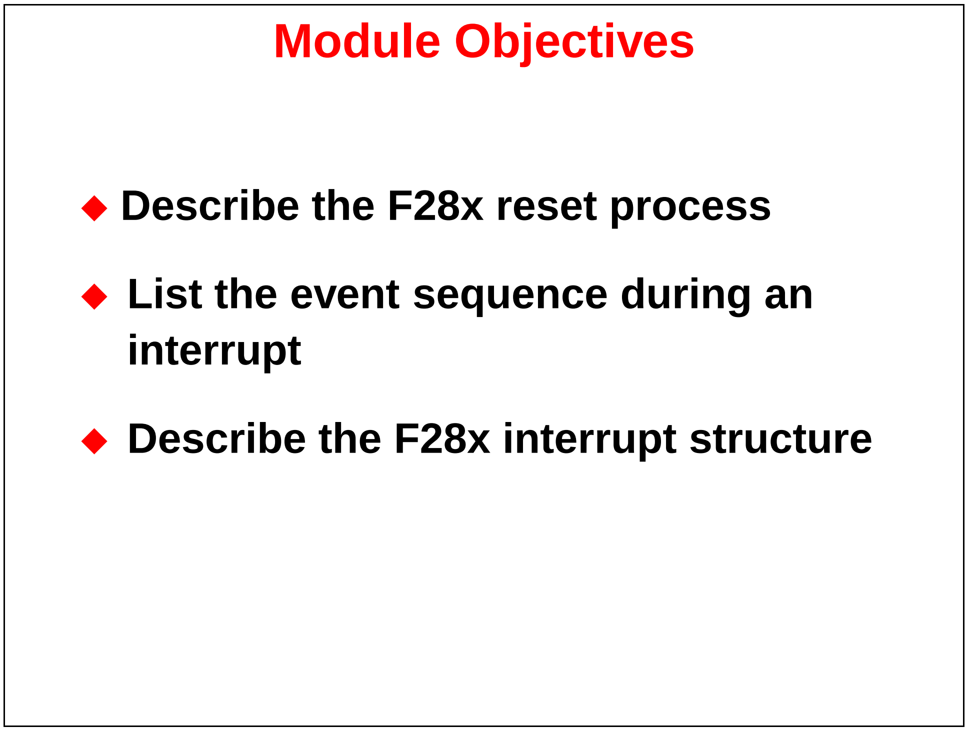
In summary, the peripheral register header files allow for easier code development, they are easy to use, generates the most efficient code, works great with Code Composer Studio, and TI has already done the work for you. Just make sure to use the correct header file package for your device.

Reset and Interrupts

Introduction

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

Module Objectives

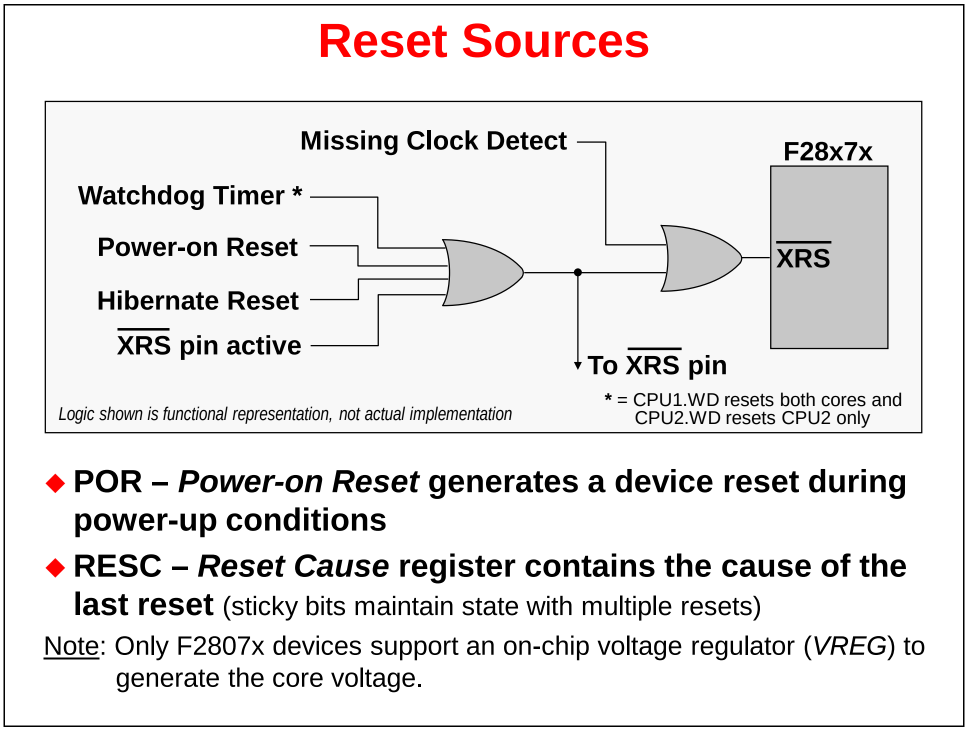


Module Objectives

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

Chapter Topics

Reset and Boot Process



Reset Sources

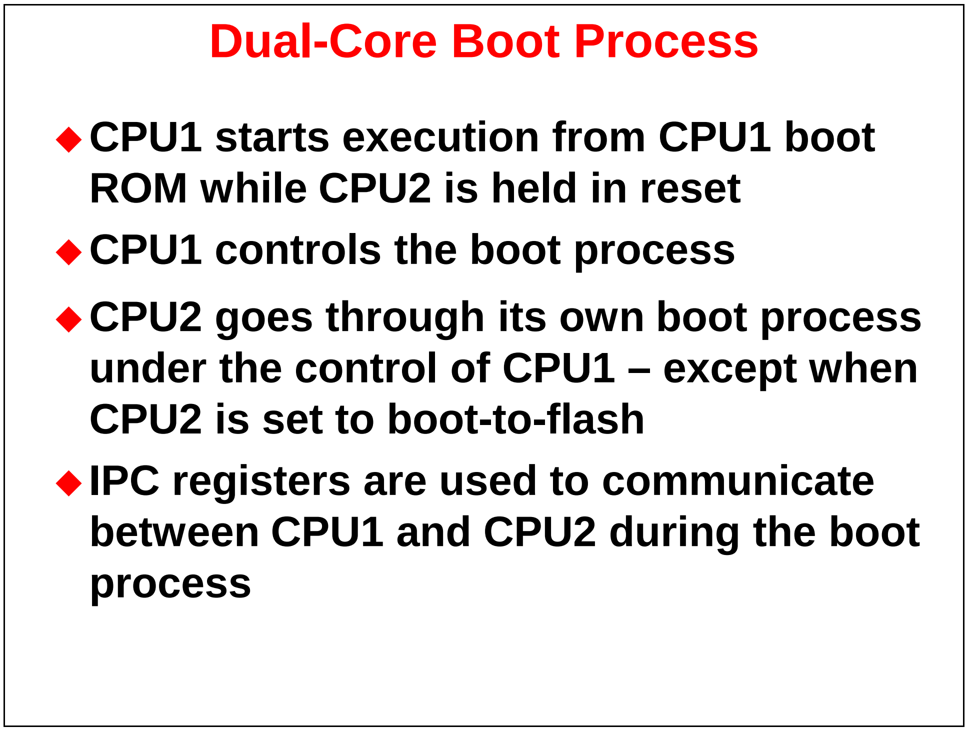
Logic shown is functional representation, not actual implementation

***** = CPU1.WD resets both cores and CPU2.WD resets CPU2 only

- **POR – *Power-on Reset* generates a device reset during power-up conditions**
- **RESC – *Reset Cause* register contains the cause of the last reset** (sticky bits maintain state with multiple resets)

Note: Only F2807x devices support an on-chip voltage regulator (*VREG*) to generate the core voltage.

The device has various reset sources, but in general resets on CPU1 will reset the entire device and resets on CPU2 will reset only the CPU2 subsystem. The reset sources include an external reset pin, watchdog timer reset, power-on reset which generates a device reset during power-up conditions, Hibernate reset, as well as a missing clock detect reset. A reset cause register (RESC) is available for each CPU subsystem which 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 both CPU subsystems to their 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. Note, only the F2807x devices support an on-chip voltage regulator to generate the core voltage.

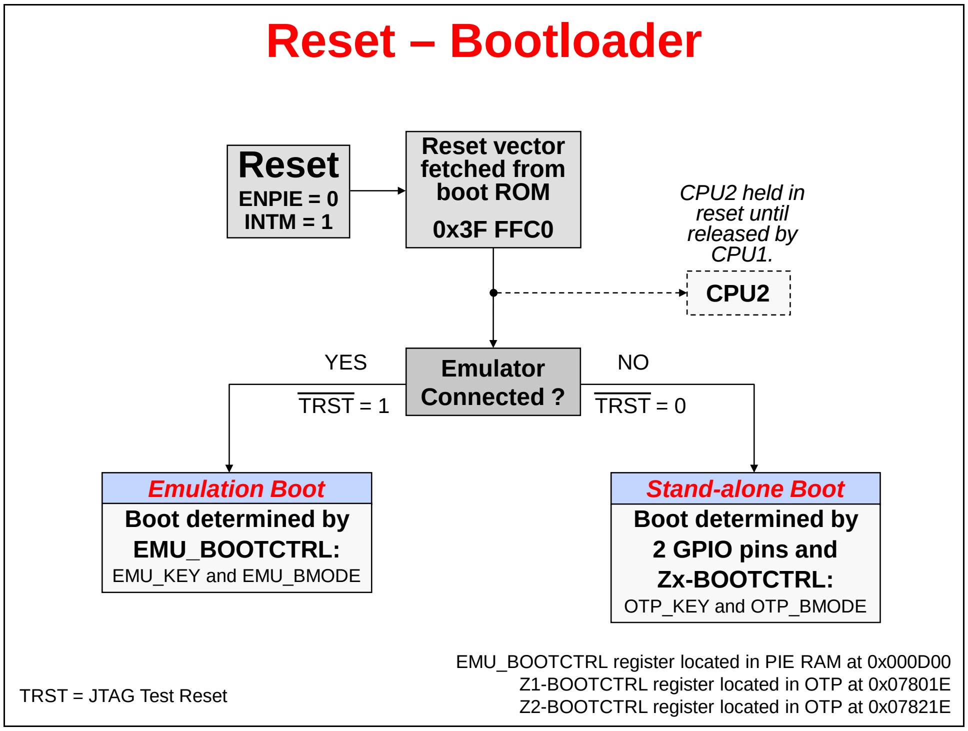


Dual-Core Boot Process

- CPU1 starts execution from CPU1 boot ROM while CPU2 is held in reset
- CPU1 controls the boot process
- CPU2 goes through its own boot process under the control of CPU1 – except when CPU2 is set to boot-to-flash
- IPC registers are used to communicate between CPU1 and CPU2 during the boot process

During the F2837xD dual-core microcontroller booting, CPU1 controls the boot process and starts execution from the CPU1 boot ROM while CPU2 is held in reset. CPU2 goes through its own boot process under the control of CPU1, except when CPU2 is set to boot-to-flash. The IPC registers are used to communicate between CPU1 and CPU2 during the boot process. Additionally, the boot ROM contains the necessary boot loading routines to support peripheral boot loading.

Reset - Bootloader



Reset – Bootloader

Reset
ENPIE = 0
INTM = 1
Reset vector fetched from boot ROM
0x3F FFC0
CPU2 held in reset until released by CPU1.
CPU2
YES
$\overline{\text{TRST}}$ = 1
Emulator Connected ?
NO
$\overline{\text{TRST}}$ = 0
Emulation Boot
Boot determined by EMU_BOOTCTRL:
EMU_KEY and EMU_BMODE
Stand-alone Boot
Boot determined by 2 GPIO pins and Zx-BOOTCTRL:
OTP_KEY and OTP_BMODE

TRST = JTAG Test Reset

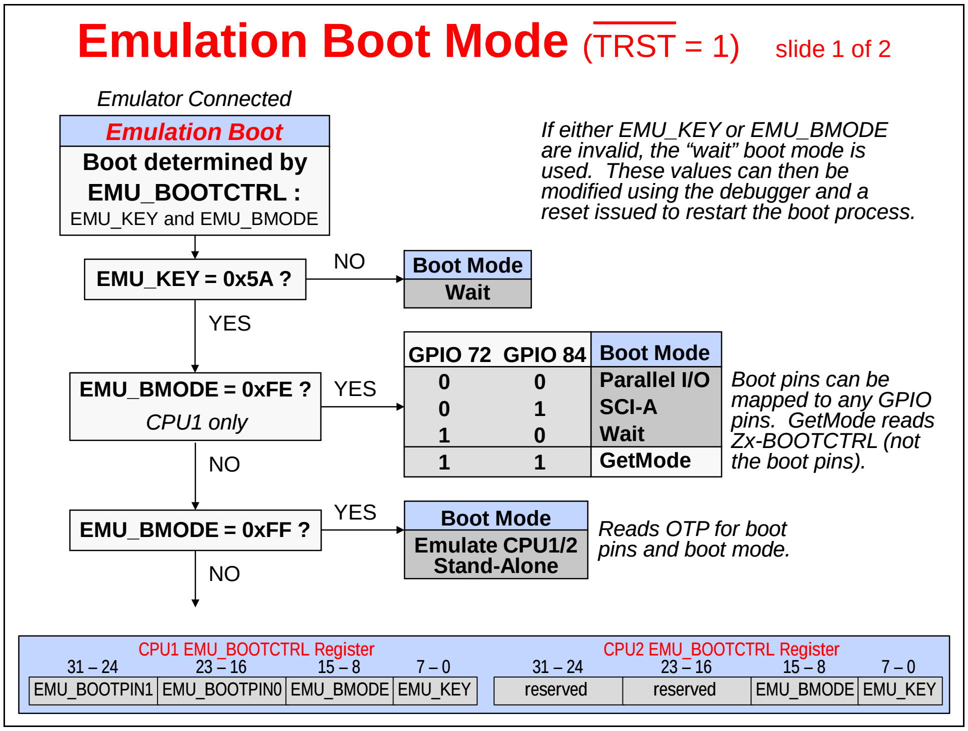
EMU_BOOTCTRL register located in PIE RAM at 0x000D00
Z1-BOOTCTRL register located in OTP at 0x07801E
Z2-BOOTCTRL register located in OTP at 0x07821E

When the device is 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. In the boot code the JTAG Test Reset line (TRST line) is checked to determine if the emulator is connected.

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_BOOTCTRL register located in the PIE RAM. Specific details about the boot flow are then determined by the EMU_KEY and EMU_BMODE bit fields in the EMU_BOOTCTRL register.

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 and the Z1-BOOTCTRL and Z2-BOOTCTRL registers located in the OTP. Specific details about the boot flow are then determined by the OTP_KEY and OTP_BMODE bit fields in the Z1-BOOTCTRL and Z2-BOOTCTRL registers.

Emulation Boot Mode



Emulation Boot Mode ($\overline{\text{TRST}}$ = 1) slide 1 of 2

Emulator Connected

Emulation Boot

Boot determined by EMU_BOOTCTRL :
EMU_KEY and EMU_BMODE

If either EMU_KEY or EMU_BMODE are invalid, the "wait" boot mode is used. These values can then be modified using the debugger and a reset issued to restart the boot process.

EMU_KEY = 0x5A ? — NO → **Boot Mode: Wait**

YES

EMU_BMODE = 0xFE ? *CPU1 only* — YES →

GPIO 72	GPIO 84	Boot Mode
0	0	Parallel I/O
0	1	SCI-A
1	0	Wait
1	1	GetMode

Boot pins can be mapped to any GPIO pins. GetMode reads Zx-BOOTCTRL (not the boot pins).

NO

EMU_BMODE = 0xFF ? — YES → **Boot Mode: Emulate CPU1/2 Stand-Alone**

Reads OTP for boot pins and boot mode.

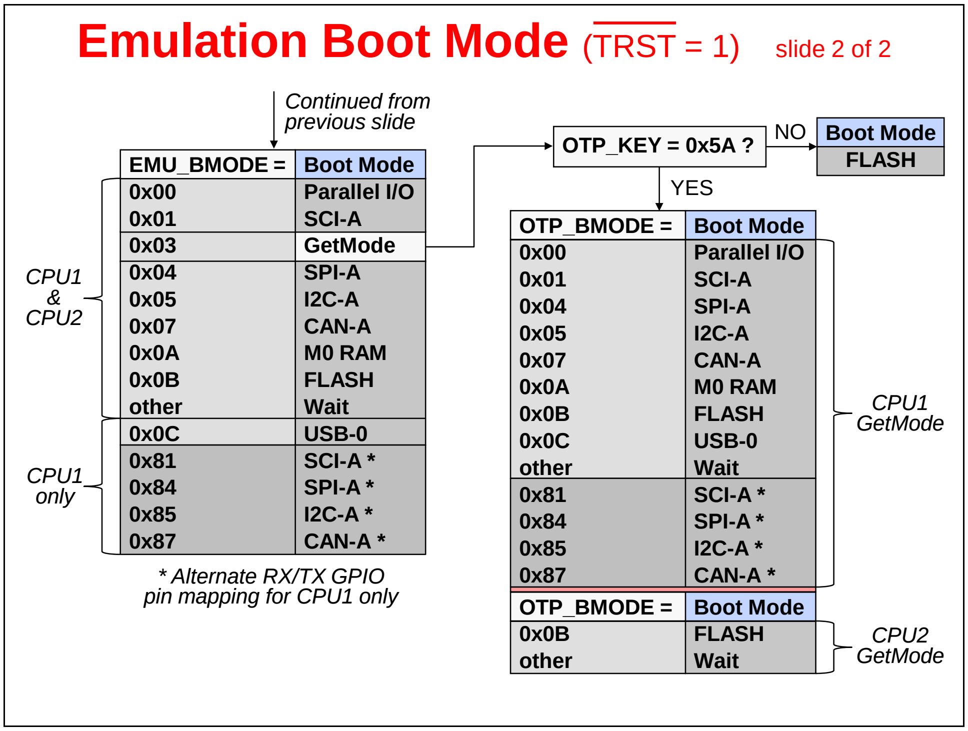
NO

CPU1 EMU_BOOTCTRL Register

31 – 24	23 – 16	15 – 8	7 – 0
EMU_BOOTPIN1	EMU_BOOTPIN0	EMU_BMODE	EMU_KEY

CPU2 EMU_BOOTCTRL Register

31 – 24	23 – 16	15 – 8	7 – 0
reserved	reserved	EMU_BMODE	EMU_KEY



Emulation Boot Mode ($\overline{\text{TRST}}$ = 1) slide 2 of 2

Continued from previous slide

CPU1 & CPU2

EMU_BMODE =	Boot Mode
0x00	Parallel I/O
0x01	SCI-A
0x03	GetMode
0x04	SPI-A
0x05	I2C-A
0x07	CAN-A
0x0A	M0 RAM
0x0B	FLASH
other	Wait
0x0C	USB-0
0x81	SCI-A *
0x84	SPI-A *
0x85	I2C-A *
0x87	CAN-A *

CPU1 only

** Alternate RX/TX GPIO pin mapping for CPU1 only*

GetMode → **OTP_KEY = 0x5A ?** — NO → **Boot Mode: FLASH**

YES

CPU1 GetMode

OTP_BMODE =	Boot Mode
0x00	Parallel I/O
0x01	SCI-A
0x04	SPI-A
0x05	I2C-A
0x07	CAN-A
0x0A	M0 RAM
0x0B	FLASH
0x0C	USB-0
other	Wait
0x81	SCI-A *
0x84	SPI-A *
0x85	I2C-A *
0x87	CAN-A *

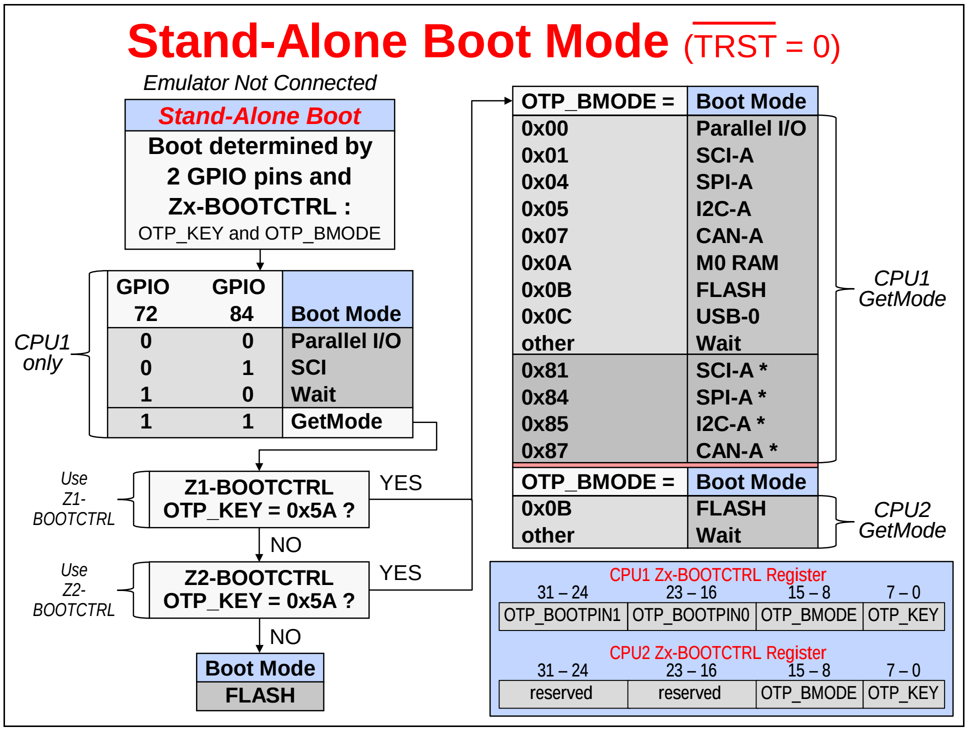
CPU2 GetMode

OTP_BMODE =	Boot Mode
0x0B	FLASH
other	Wait

In Emulation Boot mode, first the EMU_KEY bit fields are checked for a value of 0x5A. If either EMU_KEY or EMU_BMODE bit fields are invalid, the "Wait" boot mode is entered. These bit field values can then be modified using the debugger and then 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_KEY bit fields are set to 0x5A, then the EMU_BMODE bit field values determines the boot mode. The various Emulation Boot modes supported are Parallel I/O, SCI, SPI, I2C, CAN, M0 RAM, FLASH, USB, and Wait. The GetMode and when EMU_BMODE bit fields have a value of 0xFE or 0xFF are used to emulate the Stand-alone Boot mode.

Stand-Alone Boot Mode



Stand-Alone Boot Mode ($\overline{\text{TRST}}$ = 0)

Emulator Not Connected

Stand-Alone Boot

Boot determined by 2 GPIO pins and Zx-BOOTCTRL :

OTP_KEY and OTP_BMODE

CPU1 only

GPIO 72	GPIO 84	Boot Mode
0	0	Parallel I/O
0	1	SCI
1	0	Wait
1	1	GetMode

Use Z1-BOOTCTRL

Z1-BOOTCTRL OTP_KEY = 0x5A ? YES / NO

Use Z2-BOOTCTRL

Z2-BOOTCTRL OTP_KEY = 0x5A ? YES / NO

Boot Mode
FLASH

CPU1 GetMode

OTP_BMODE =	Boot Mode
0x00	Parallel I/O
0x01	SCI-A
0x04	SPI-A
0x05	I2C-A
0x07	CAN-A
0x0A	M0 RAM
0x0B	FLASH
0x0C	USB-0
other	Wait
0x81	SCI-A *
0x84	SPI-A *
0x85	I2C-A *
0x87	CAN-A *

CPU2 GetMode

OTP_BMODE =	Boot Mode
0x0B	FLASH
other	Wait

CPU1 Zx-BOOTCTRL Register

31 – 24	23 – 16	15 – 8	7 – 0
OTP_BOOTPIN1	OTP_BOOTPIN0	OTP_BMODE	OTP_KEY

CPU2 Zx-BOOTCTRL Register

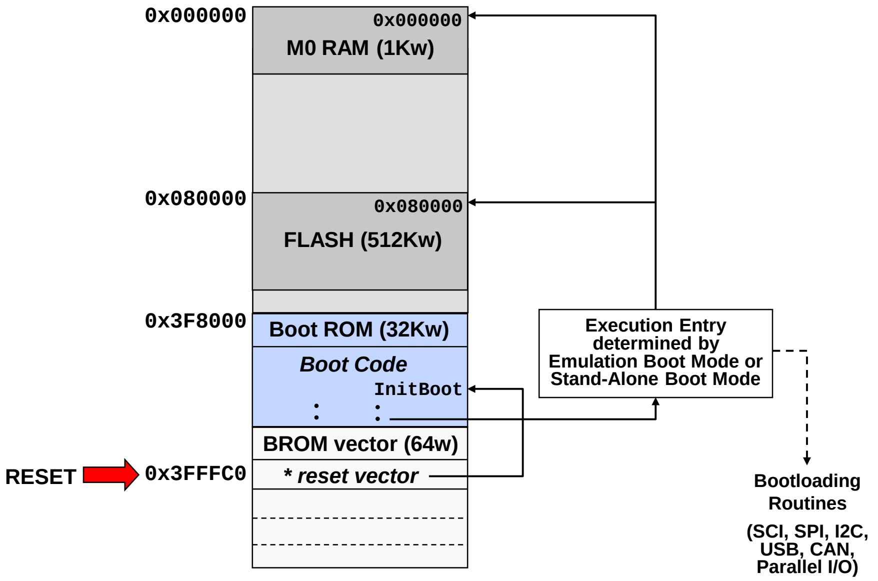
31 – 24	23 – 16	15 – 8	7 – 0
reserved	reserved	OTP_BMODE	OTP_KEY

In Stand-alone boot mode, first GPIO pins 72 and 84 are checked to determine if the boot mode is Parallel I/O, SCI, Wait, or GetMode. These pin can be remapped to any GPIO pins, if needed, and the default “unconnected” pins set the boot mode to GetMode. In GetMode the OTP_KEY bit fields in the Z1-BOOTCTRL and Z2-BOOTCTRL registers are checked for a value of 0x5A. An un-programmed device will have these locations set as 1’s, and the flash boot mode is entered, as expected for the default mode. If the OTP_KEY bit fields in either Z1-BOOTCTRL or Z2-BOOTCTRL registers has a value of 0x5A, then the OTP_BMODE bit field values in the registers determines the boot mode. The various Stand-alone Boot modes supported are Parallel I/O, SCI, SPI, I2C, CAN, M0 RAM, FLASH, USB, and Wait.

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.

Reset Code Flow - Summary



*** reset vector = 0x3FF16A for CPU1; 0x3FEC52 for CPU2***

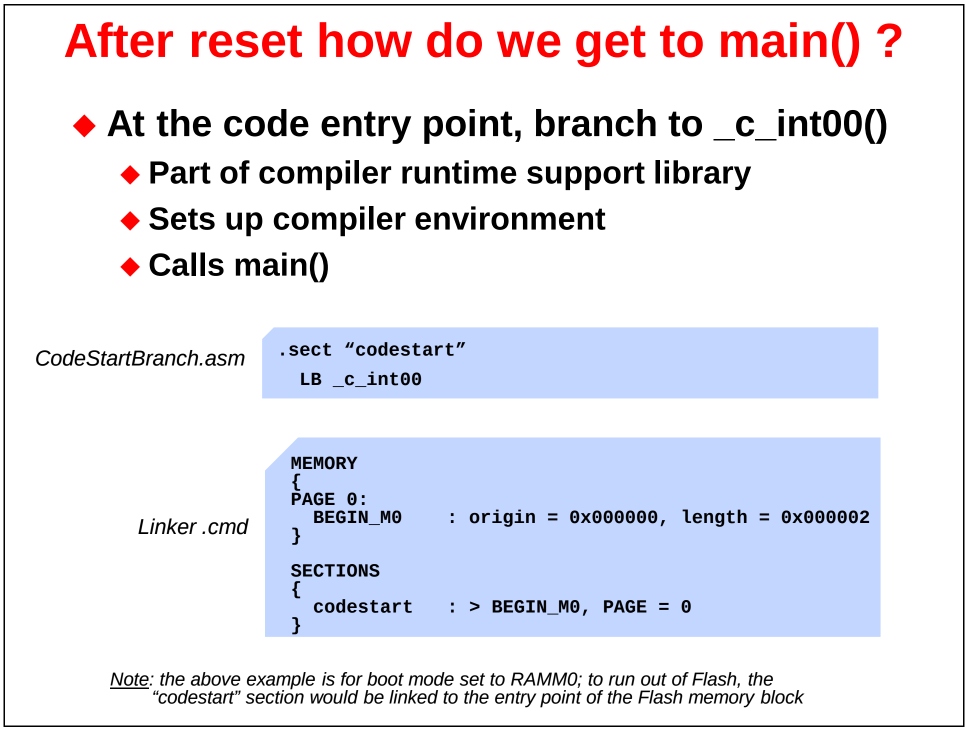
Emulation Boot Mode using Code Composer Studio GEL

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”. It 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()
{
    *0xD00 = 0x0A5A;
}
hotmenu EMU_BOOT_FLASH()
{
    *0xD00 = 0x0B5A;
}
```

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 runtime support library
  - Sets up compiler environment
  - Calls main()

CodeStartBranch.asm

```
.sect “codestart”
  LB _c_int00
```

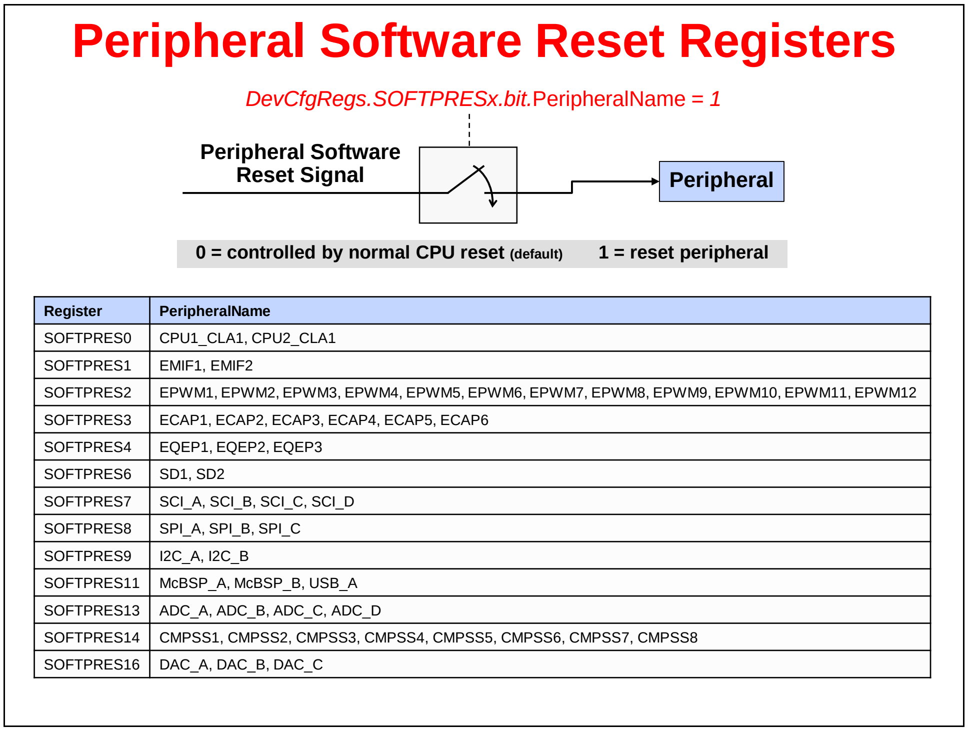
Linker .cmd

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

SECTIONS
{
  codestart   : > BEGIN_M0, PAGE = 0
}
```

Note: the above example is for boot mode set to 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



Peripheral Software Reset Registers

*DevCfgRegs.SOFTPRESx.bit.*PeripheralName = *1*

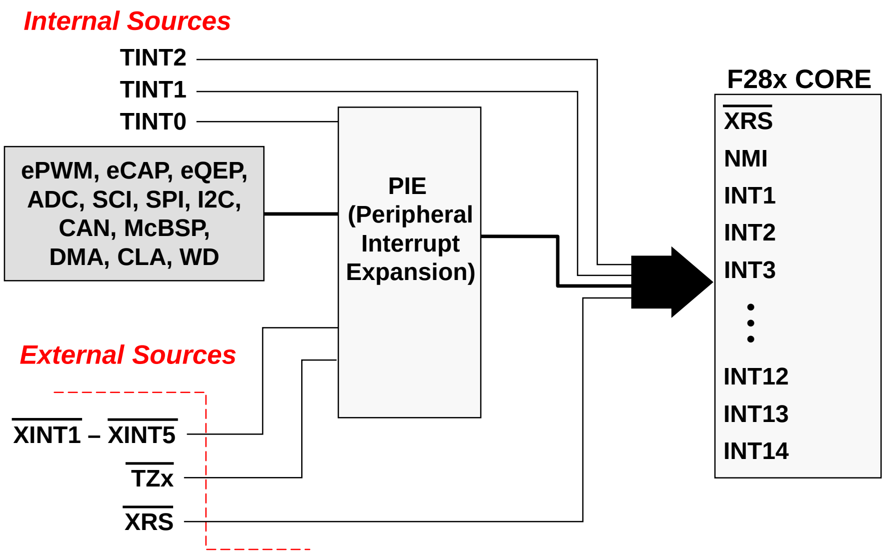
0 = controlled by normal CPU reset (default) **1 = reset peripheral**

Register	PeripheralName
SOFTPRES0	CPU1_CLA1, CPU2_CLA1
SOFTPRES1	EMIF1, EMIF2
SOFTPRES2	EPWM1, EPWM2, EPWM3, EPWM4, EPWM5, EPWM6, EPWM7, EPWM8, EPWM9, EPWM10, EPWM11, EPWM12
SOFTPRES3	ECAP1, ECAP2, ECAP3, ECAP4, ECAP5, ECAP6
SOFTPRES4	EQEP1, EQEP2, EQEP3
SOFTPRES6	SD1, SD2
SOFTPRES7	SCI_A, SCI_B, SCI_C, SCI_D
SOFTPRES8	SPI_A, SPI_B, SPI_C
SOFTPRES9	I2C_A, I2C_B
SOFTPRES11	McBSP_A, McBSP_B, USB_A
SOFTPRES13	ADC_A, ADC_B, ADC_C, ADC_D
SOFTPRES14	CMPSS1, CMPSS2, CMPSS3, CMPSS4, CMPSS5, CMPSS6, CMPSS7, CMPSS8
SOFTPRES16	DAC_A, DAC_B, DAC_C

The peripheral software reset register contains the reset bit for each peripheral.

Interrupts

Interrupt Sources



Internal Sources

TINT2

TINT1

TINT0

ePWM, eCAP, eQEP, ADC, SCI, SPI, I2C, CAN, McBSP, DMA, CLA, WD

PIE (Peripheral Interrupt Expansion)

External Sources

$\overline{\text{XINT1}}$ – $\overline{\text{XINT5}}$

$\overline{\text{TZx}}$

$\overline{\text{XRS}}$

F28x CORE

$\overline{\text{XRS}}$

NMI

INT1

INT2

INT3

⋮

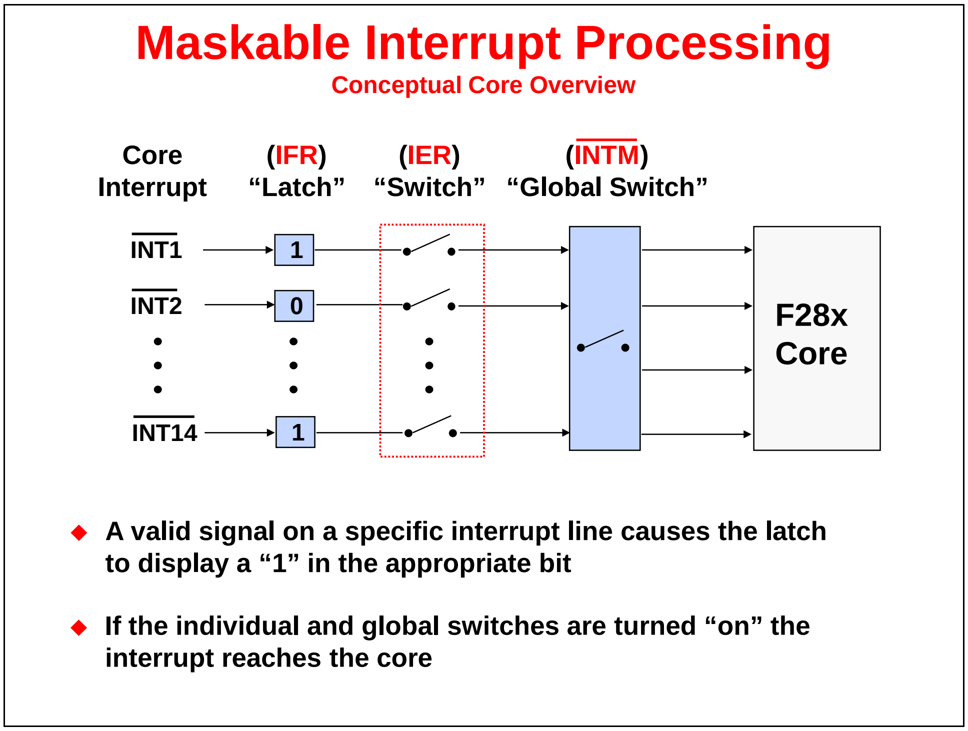
INT12

INT13

INT14

The internal interrupt sources include the general purpose timers 0, 1, and 2, and all of the peripherals on the device. External interrupt sources include the three external interrupt lines, the trip zones, and the external reset pin. The core has 14 interrupt lines. 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 core interrupt capability, allowing up to 192 possible interrupt sources.

Interrupt Processing



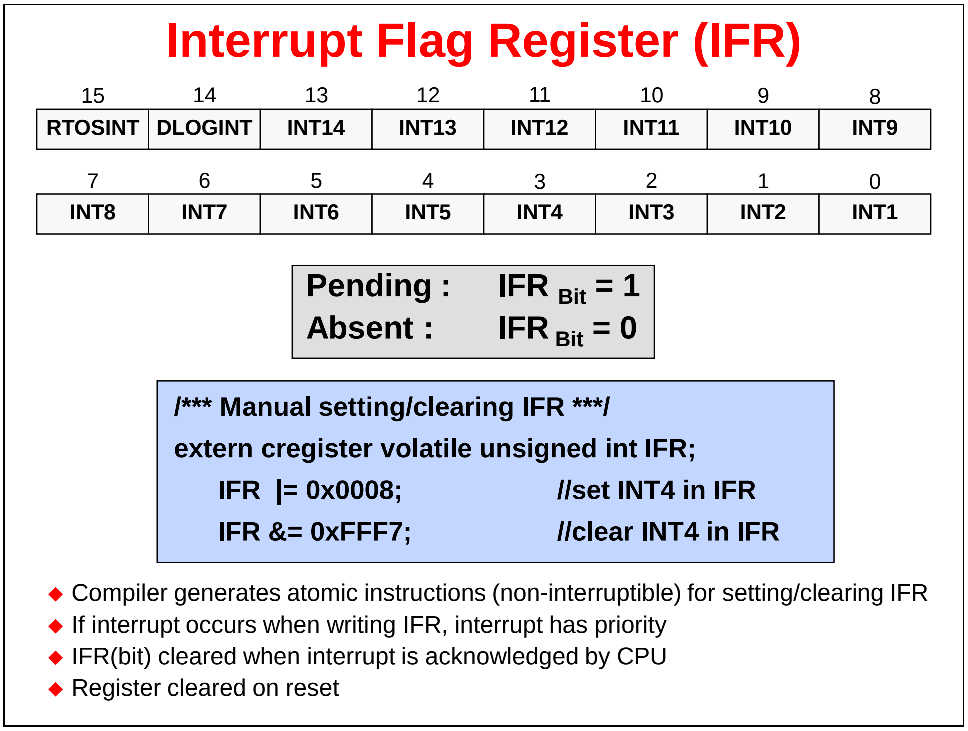
Maskable Interrupt Processing

Conceptual Core Overview

- A valid signal on a specific interrupt line causes the latch to display a “1” in the appropriate bit
- If the individual and global switches are turned “on” the interrupt reaches the core

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 ORing and ANDing mask values. The INTM bit in the status register is managed by using in-line assembly instructions.

Interrupt Flag Register (IFR)



Interrupt Flag Register (IFR)

15	14	13	12	11	10	9	8
RTOSINT	DLOGINT	INT14	INT13	INT12	INT11	INT10	INT9

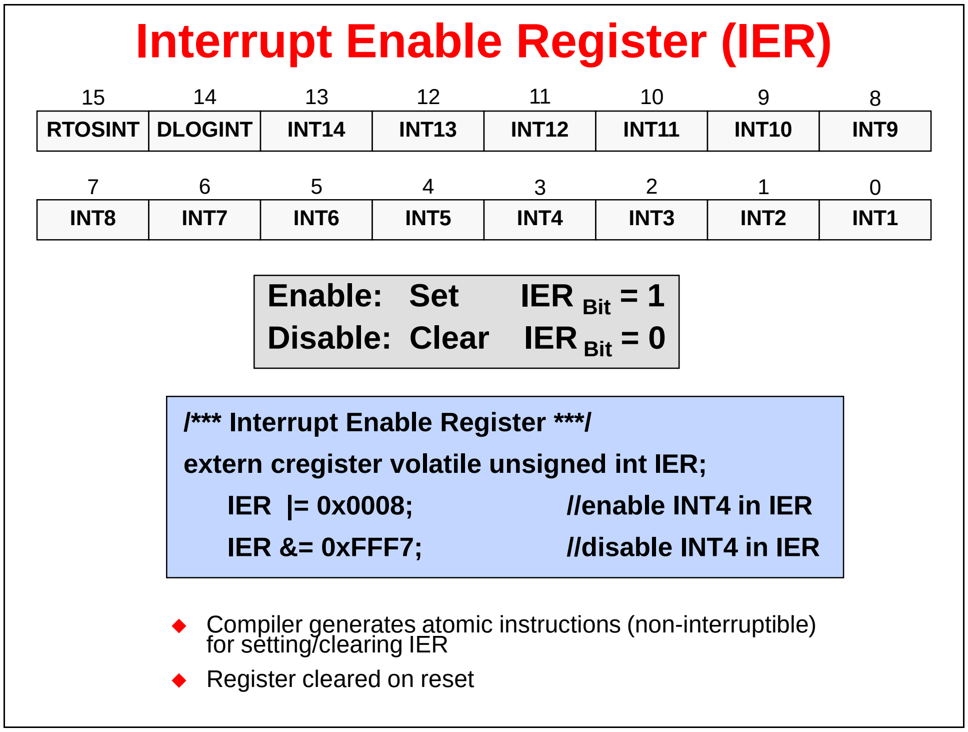
7	6	5	4	3	2	1	0
INT8	INT7	INT6	INT5	INT4	INT3	INT2	INT1

Pending : IFR $_{Bit}$ = 1
Absent : IFR $_{Bit}$ = 0

```
/*** Manual setting/clearing IFR ***/
extern cregister volatile unsigned int IFR;
    IFR  |= 0x0008;             //set INT4 in IFR
    IFR &= 0xFFF7;              //clear INT4 in IFR
```

- Compiler generates atomic instructions (non-interruptible) for setting/clearing IFR
- If interrupt occurs when writing IFR, interrupt has priority
- IFR(bit) cleared when interrupt is acknowledged by CPU
- Register cleared on reset

Interrupt Enable Register (IER)



Interrupt Enable Register (IER)

15	14	13	12	11	10	9	8
RTOSINT	DLOGINT	INT14	INT13	INT12	INT11	INT10	INT9

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

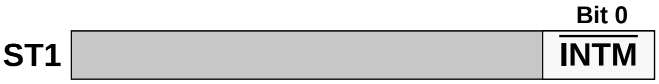
Enable: Set IER $_{Bit}$ = 1
Disable: Clear IER $_{Bit}$ = 0

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

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

Interrupt Global Mask Bit (INTM)

Interrupt Global Mask Bit

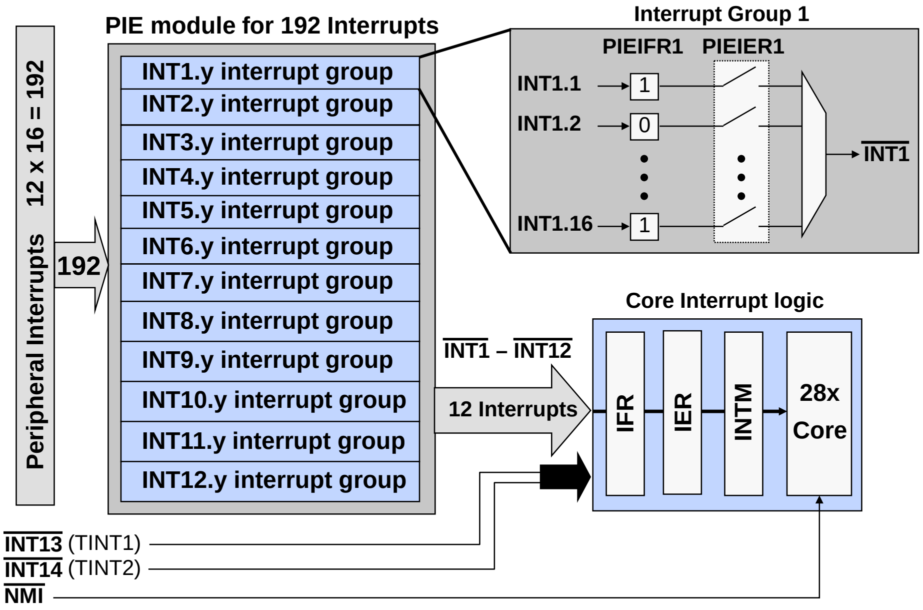


- INTM used to globally enable/disable interrupts:
  - Enable: $\overline{\text{INTM}}$ = 0
  - Disable: $\overline{\text{INTM}}$ = 1 (reset value)
- INTM modified from assembly code only:

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

Peripheral Interrupt Expansion (PIE)

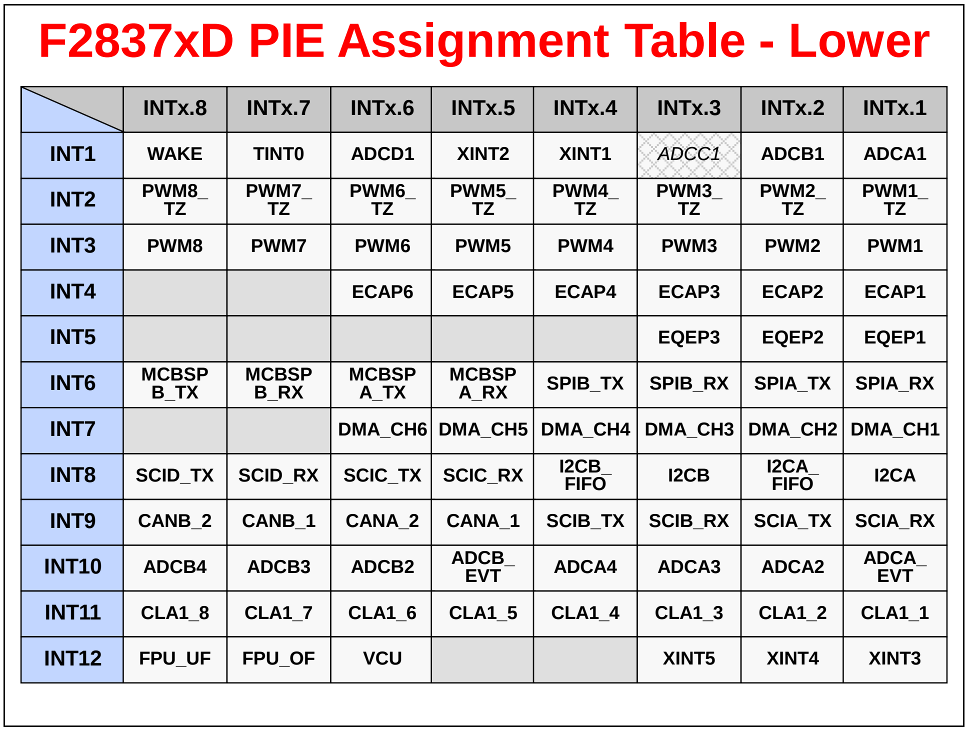
Peripheral Interrupt Expansion - PIE



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.

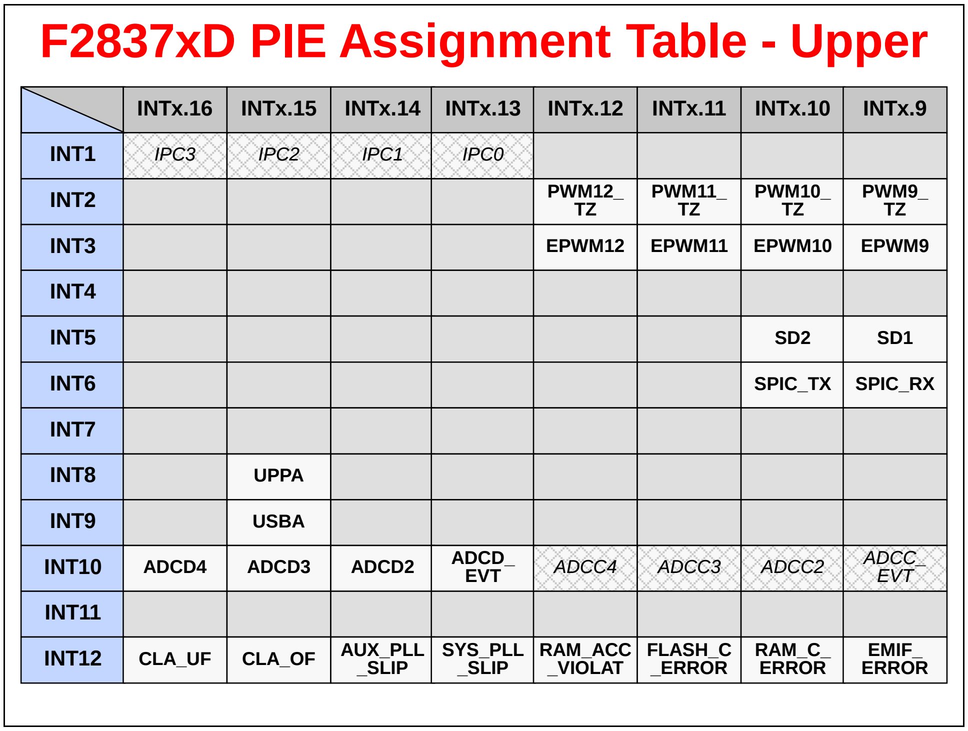
We have already discussed the interrupt process in the core. Now we need to look at the peripheral interrupt expansion block. This block is connected to the core interrupt lines 1 through 12. The PIE block consists of 12 groups. Within each group, there are sixteen interrupt sources. Each group has a PIE interrupt enable register and a PIE interrupt flag register. Note that interrupt lines 13, 14, and NMI bypass the PIE block.



F2837xD PIE Assignment Table - Lower

	INTx.8	INTx.7	INTx.6	INTx.5	INTx.4	INTx.3	INTx.2	INTx.1
INT1	WAKE	TINT0	ADCD1	XINT2	XINT1	*ADCC1*	ADCB1	ADCA1
INT2	PWM8_TZ	PWM7_TZ	PWM6_TZ	PWM5_TZ	PWM4_TZ	PWM3_TZ	PWM2_TZ	PWM1_TZ
INT3	PWM8	PWM7	PWM6	PWM5	PWM4	PWM3	PWM2	PWM1
INT4			ECAP6	ECAP5	ECAP4	ECAP3	ECAP2	ECAP1
INT5						EQEP3	EQEP2	EQEP1
INT6	MCBSP B_TX	MCBSP B_RX	MCBSP A_TX	MCBSP A_RX	SPIB_TX	SPIB_RX	SPIA_TX	SPIA_RX
INT7			DMA_CH6	DMA_CH5	DMA_CH4	DMA_CH3	DMA_CH2	DMA_CH1
INT8	SCID_TX	SCID_RX	SCIC_TX	SCIC_RX	I2CB_FIFO	I2CB	I2CA_FIFO	I2CA
INT9	CANB_2	CANB_1	CANA_2	CANA_1	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_UF	FPU_OF	VCU			XINT5	XINT4	XINT3

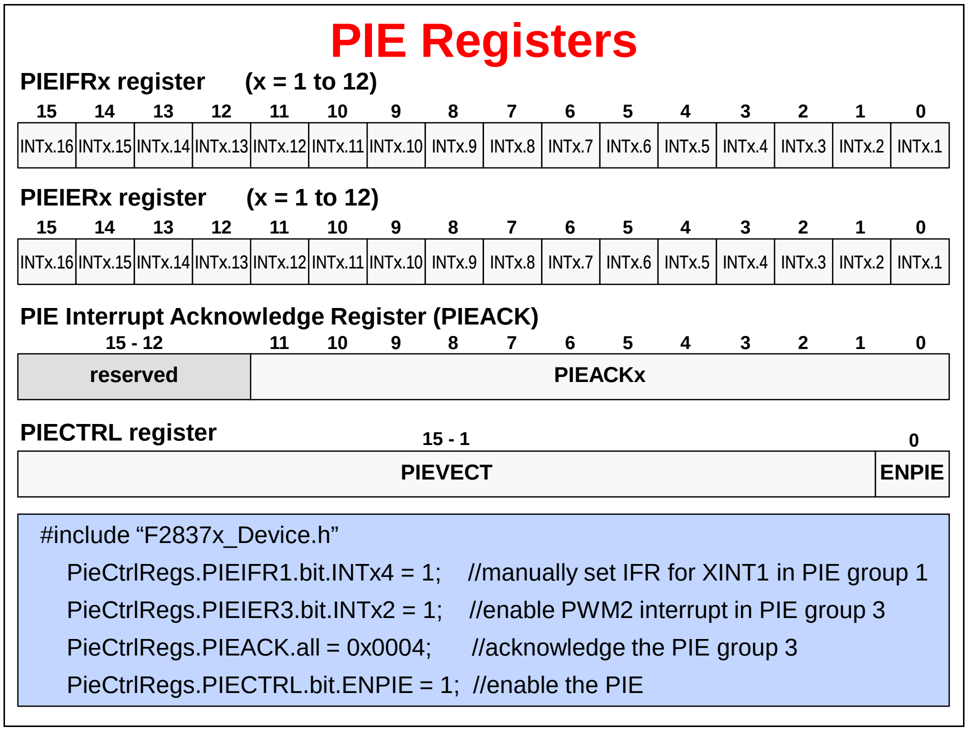
The PIE assignment table maps each peripheral interrupt to the unique vector location for that interrupt service routine. Notice the interrupt numbers on the left represent the twelve core group interrupt lines and the interrupt numbers across the top represent the lower eight of the sixteen peripheral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



F2837xD PIE Assignment Table - Upper

	INTx.16	INTx.15	INTx.14	INTx.13	INTx.12	INTx.11	INTx.10	INTx.9
INT1	*IPC3*	*IPC2*	*IPC1*	*IPC0*				
INT2					PWM12_TZ	PWM11_TZ	PWM10_TZ	PWM9_TZ
INT3					EPWM12	EPWM11	EPWM10	EPWM9
INT4								
INT5							SD2	SD1
INT6							SPIC_TX	SPIC_RX
INT7								
INT8		UPPA						
INT9		USBA						
INT10	ADCD4	ADCD3	ADCD2	ADCD_EVT	*ADCC4*	*ADCC3*	*ADCC2*	*ADCC_EVT*
INT11								
INT12	CLA_UF	CLA_OF	AUX_PLL_SLIP	SYS_PLL_SLIP	RAM_ACC_VIOLAT	FLASH_C_ERROR	RAM_C_ERROR	EMIF_ERROR

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. The enable PIE bit in the PIECTRL register is used to activate the PIE module.



PIE Registers

PIEIFRx 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

PIEIERx 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	PIEACKx											

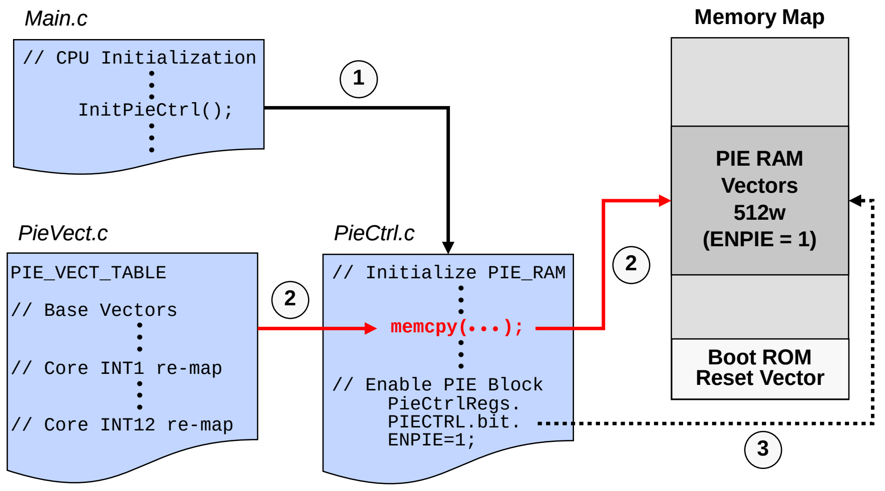
PIECTRL register

15 - 1	0
PIEVECT	ENPIE

```
#include “F2837x_Device.h”
    PieCtrlRegs.PIEIFR1.bit.INTx4 = 1;    //manually set IFR for XINT1 in PIE group 1
    PieCtrlRegs.PIEIER3.bit.INTx2 = 1;    //enable PWM2 interrupt in PIE group 3
    PieCtrlRegs.PIEACK.all = 0x0004;      //acknowledge the PIE group 3
    PieCtrlRegs.PIECTRL.bit.ENPIE = 1;  //enable the PIE
```

PIE Block Initialization

PIE Block Initialization

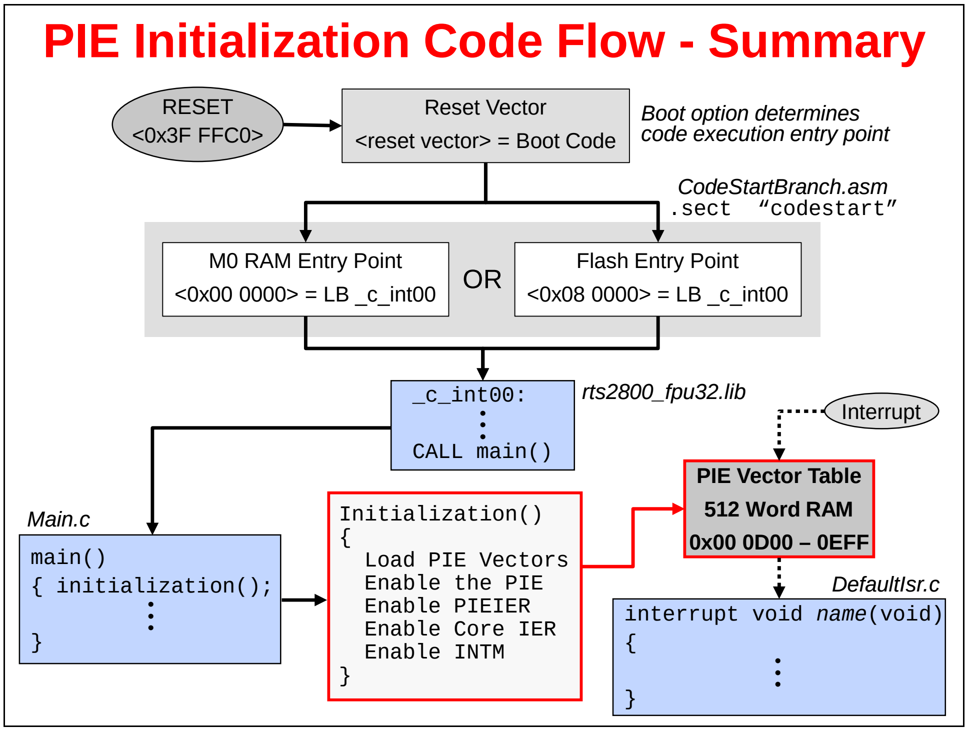


```
// CPU Initialization
      :
   InitPieCtrl();
      :
```

```
PIE_VECT_TABLE
// Base Vectors
      :
// Core INT1 re-map
      :
// Core INT12 re-map
```

```
// Initialize PIE_RAM
      :
     memcpy(...);
      :
// Enable PIE Block
     PieCtrlRegs.
     PIECTRL.bit.
     ENPIE=1;
```

The interrupt vector table, as mapped in the PIE interrupt assignment table, is located in the PieVect.c file. During processor initialization a function call to PieCtrl.c file is used to copy the interrupt vector table 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.



PIE Initialization Code Flow - Summary

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

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

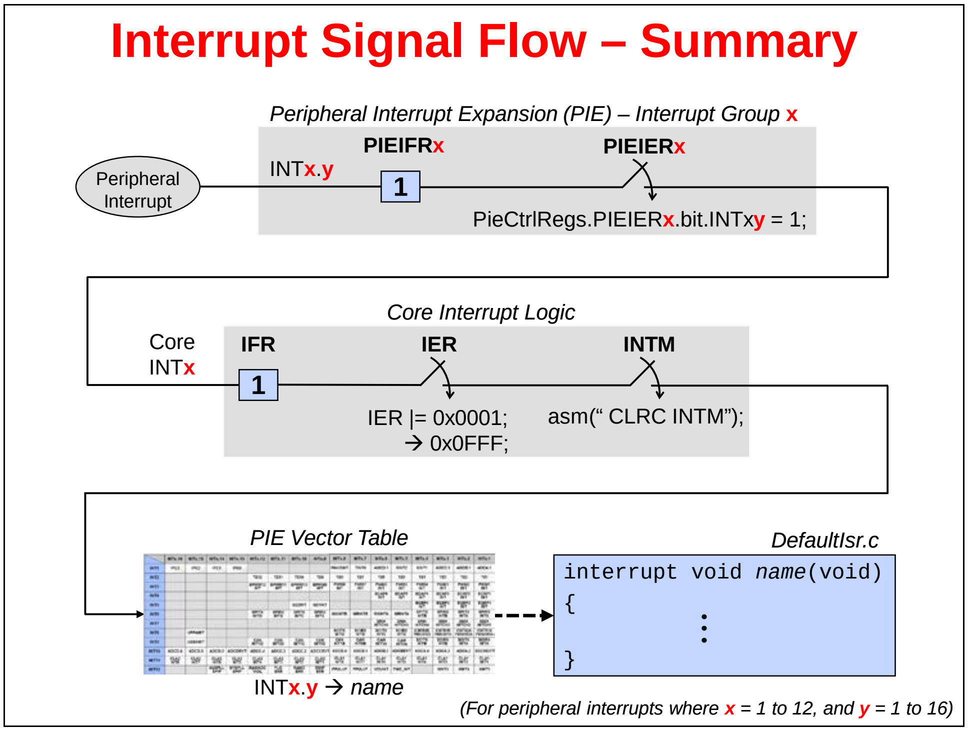
```
Initialization()
{
  Load PIE Vectors
  Enable the PIE
  Enable PIEIER
  Enable Core IER
  Enable INTM
}
```

```
interrupt void name(void)
{
        :
}
```

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, a function is 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



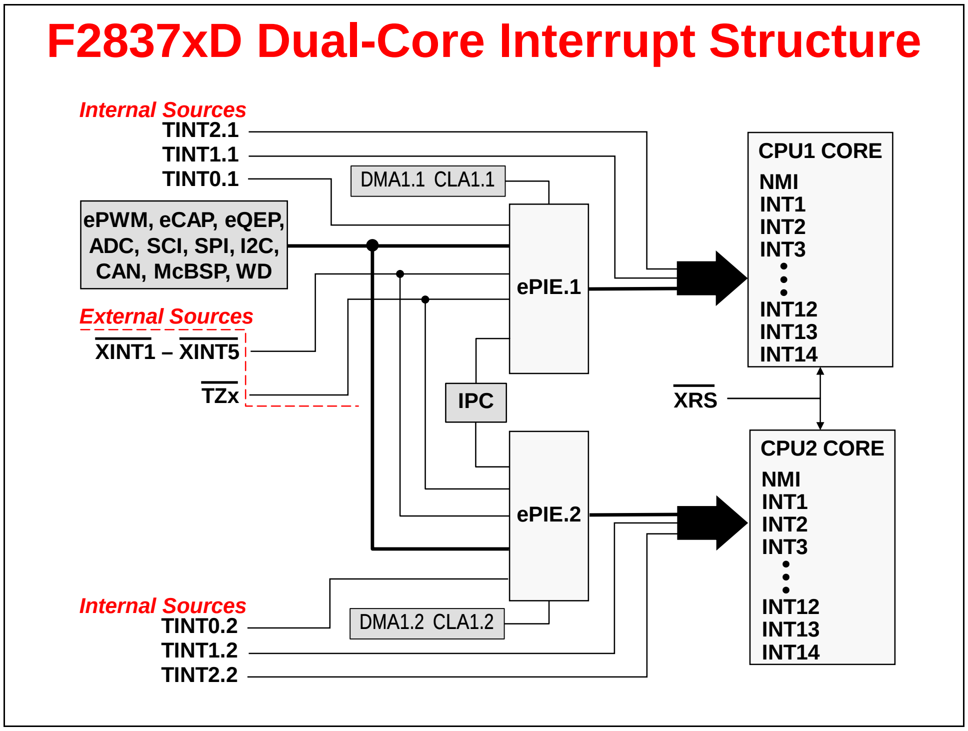
Interrupt Signal Flow – Summary

```
interrupt void name(void)
{
        ⋮
}
```

(For peripheral interrupts where x = 1 to 12, and y = 1 to 16)

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.

F2837xD Dual-Core Interrupt Structure



F2837xD Dual-Core Interrupt Structure

Internal Sources
TINT2.1
TINT1.1
TINT0.1
DMA1.1 CLA1.1
ePWM, eCAP, eQEP, ADC, SCI, SPI, I2C, CAN, McBSP, WD
External Sources
XINT1 – XINT5
TZx
ePIE.1
IPC
ePIE.2
XRS
CPU1 CORE
NMI
INT1
INT2
INT3
INT12
INT13
INT14
CPU2 CORE
NMI
INT1
INT2
INT3
INT12
INT13
INT14
Internal Sources
TINT0.2
DMA1.2 CLA1.2
TINT1.2
TINT2.2

Each C28x CPU core in the F2837xD device has its own PIE module, and each PIE module is configured independently. Some interrupt signals are sourced from shared peripherals that can be owned by either CPU, and these interrupt signals are sent to both CPU PIE modules regardless of which CPU owns the peripheral. Therefore, if enabled a peripheral owned by one CPU can cause an interrupt on the other CPU.

Interrupt Response and Latency

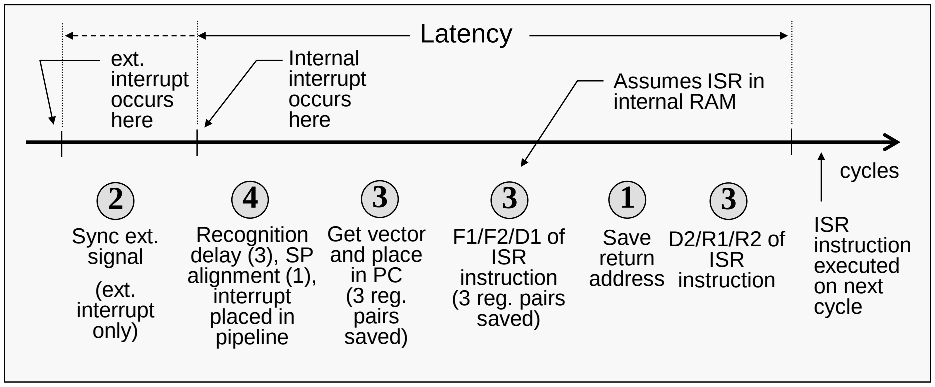
Interrupt Response - Hardware Sequence

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

Note: some actions occur simultaneously, none are interruptible

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

Interrupt Latency



Latency

ext. interrupt occurs here

Internal interrupt occurs here

Assumes ISR in internal RAM

cycles

② Sync ext. signal (ext. interrupt only)

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

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

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

① Save return address

③ D2/R1/R2 of ISR instruction

ISR instruction executed on next cycle

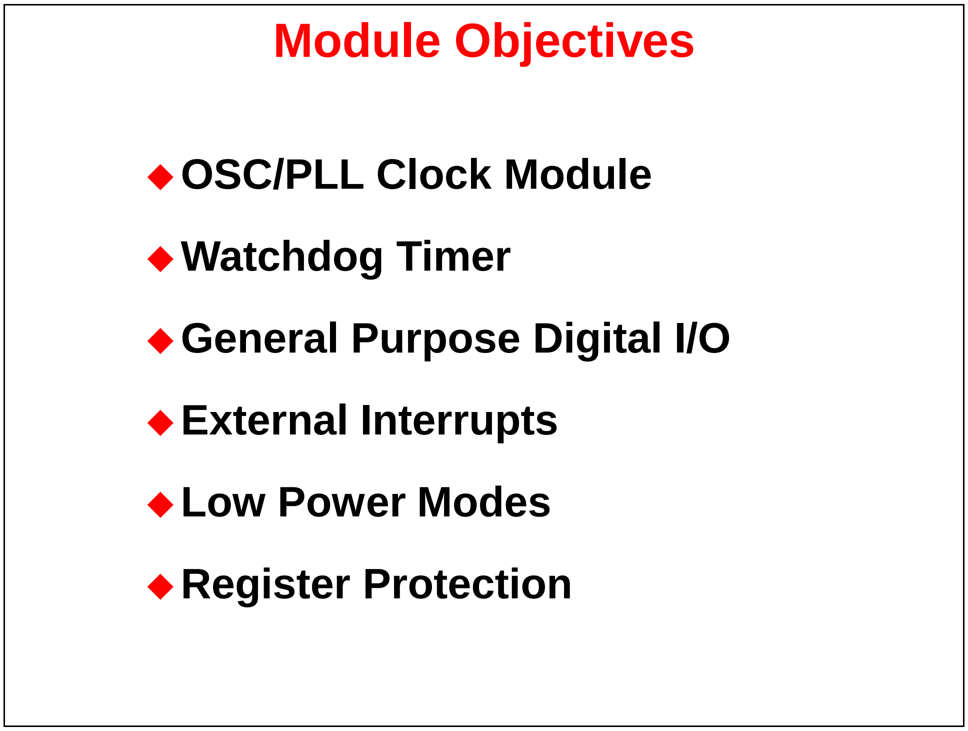
- **Minimum latency (to when real work occurs in the ISR):**
  - **Internal interrupts: 14 cycles**
  - **External interrupts: 16 cycles**
- **Maximum latency: Depends on wait states, INTM, etc.**

System Initialization

Introduction

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

Module Objectives



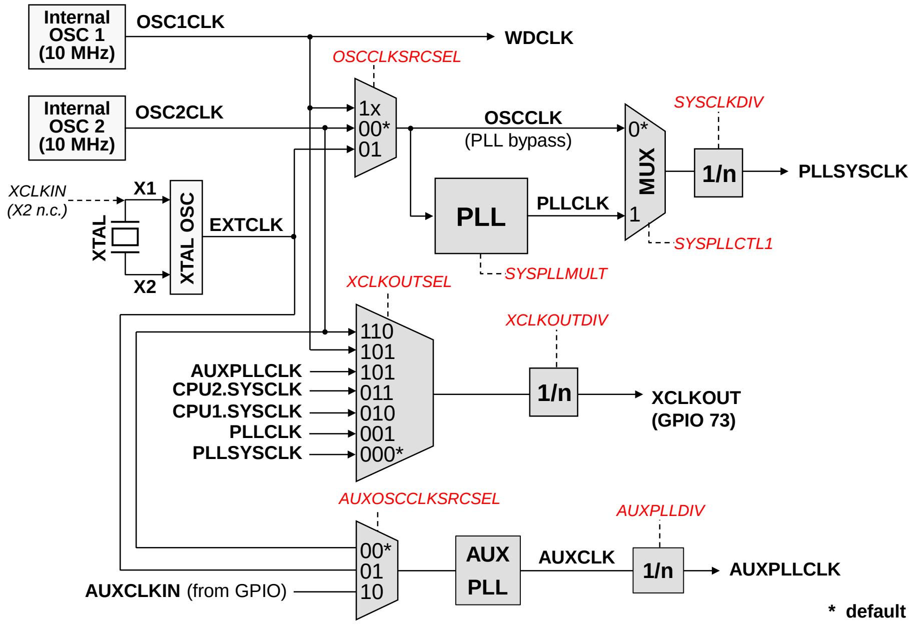
Module Objectives

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

Chapter Topics

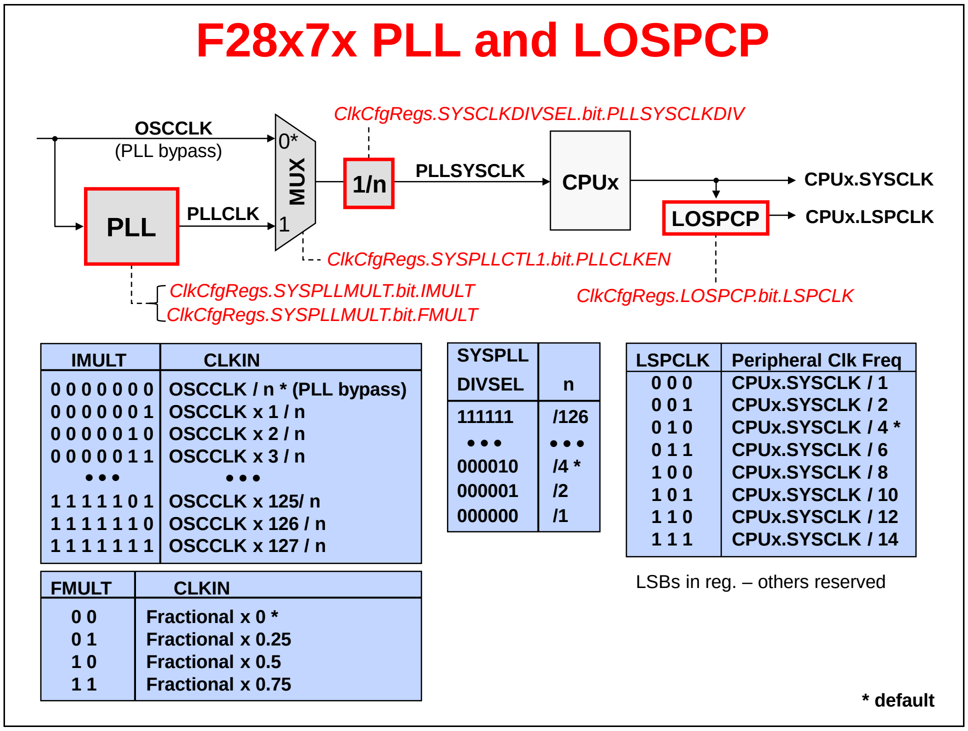
Oscillator/PLL Clock Module

F28x7x 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 Auxiliary Clock Input (AUXCLKIN). At power-up, the device is clocked from the on-chip 10 MHz oscillator INTOSC2. INTSOC2 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 AUXCLKIN is used as the bit clock source for the USB and CAN to generate the precise frequency requirements.



F28x7x PLL and LOSPCP

OSCCLK
(PLL bypass)
PLL
PLLCLK
0*
MUX
1
ClkCfgRegs.SYSCLKDIVSEL.bit.PLLSYSCLKDIV
1/n
PLLSYSCLK
CPUx
CPUx.SYSCLK
LOSPCP
CPUx.LSPCLK
ClkCfgRegs.SYSPLLCTL1.bit.PLLCLKEN
ClkCfgRegs.SYSPLLMULT.bit.IMULT
ClkCfgRegs.SYSPLLMULT.bit.FMULT
ClkCfgRegs.LOSPCP.bit.LSPCLK

IMULT	CLKIN
0 0 0 0 0 0 0	OSCCLK / n * (PLL bypass)
0 0 0 0 0 0 1	OSCCLK x 1 / n
0 0 0 0 0 1 0	OSCCLK x 2 / n
0 0 0 0 0 1 1	OSCCLK x 3 / n
• • •	• • •
1 1 1 1 1 0 1	OSCCLK x 125/ n
1 1 1 1 1 1 0	OSCCLK x 126 / n
1 1 1 1 1 1 1	OSCCLK x 127 / n

SYSPLL DIVSEL	n
111111	/126
• • •	• • •
000010	/4 *
000001	/2
000000	/1

LSPCLK	Peripheral Clk Freq
0 0 0	CPUx.SYSCLK / 1
0 0 1	CPUx.SYSCLK / 2
0 1 0	CPUx.SYSCLK / 4 *
0 1 1	CPUx.SYSCLK / 6
1 0 0	CPUx.SYSCLK / 8
1 0 1	CPUx.SYSCLK / 10
1 1 0	CPUx.SYSCLK / 12
1 1 1	CPUx.SYSCLK / 14

LSBs in reg. – others reserved

FMULT	CLKIN
0 0	Fractional x 0 *
0 1	Fractional x 0.25
1 0	Fractional x 0.5
1 1	Fractional x 0.75

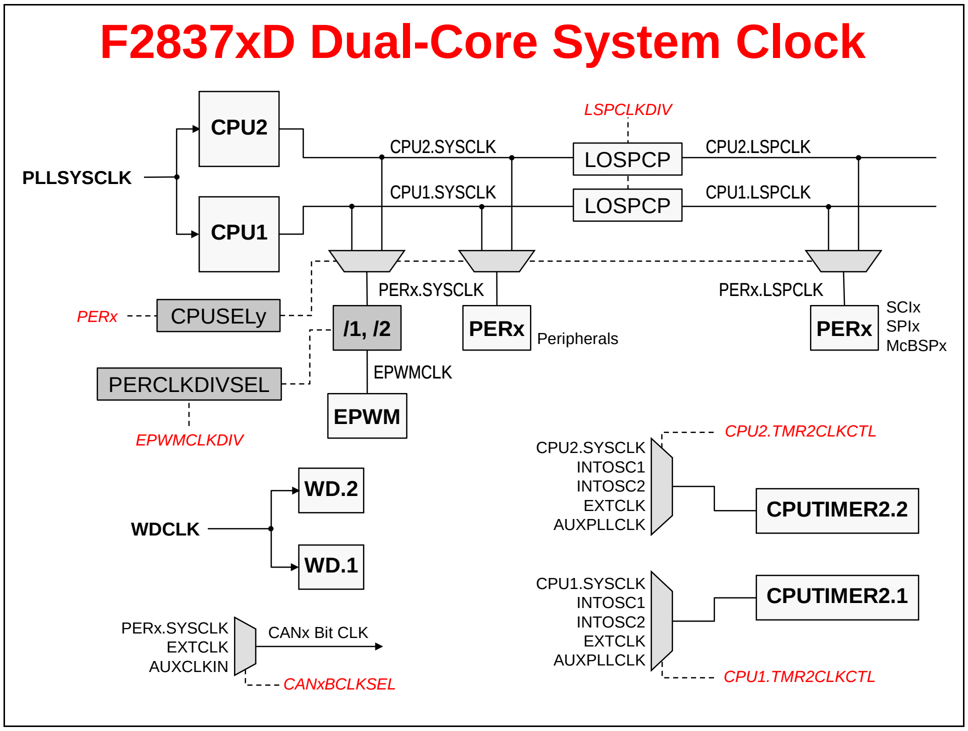
*** default**

The clock sources can be multiplied using the PLL and divided down to produce the desired clock frequencies for a specific application. By default, the CPU1 subsystem owns the PLL clock configuration, however a clock control semaphore is available for the CPU2 subsystem to access the clock configuration registers.

A clock source can be fed directly into the core or multiplied using the PLL. The PLL gives us the capability to use the internal 10 MHz oscillator 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.

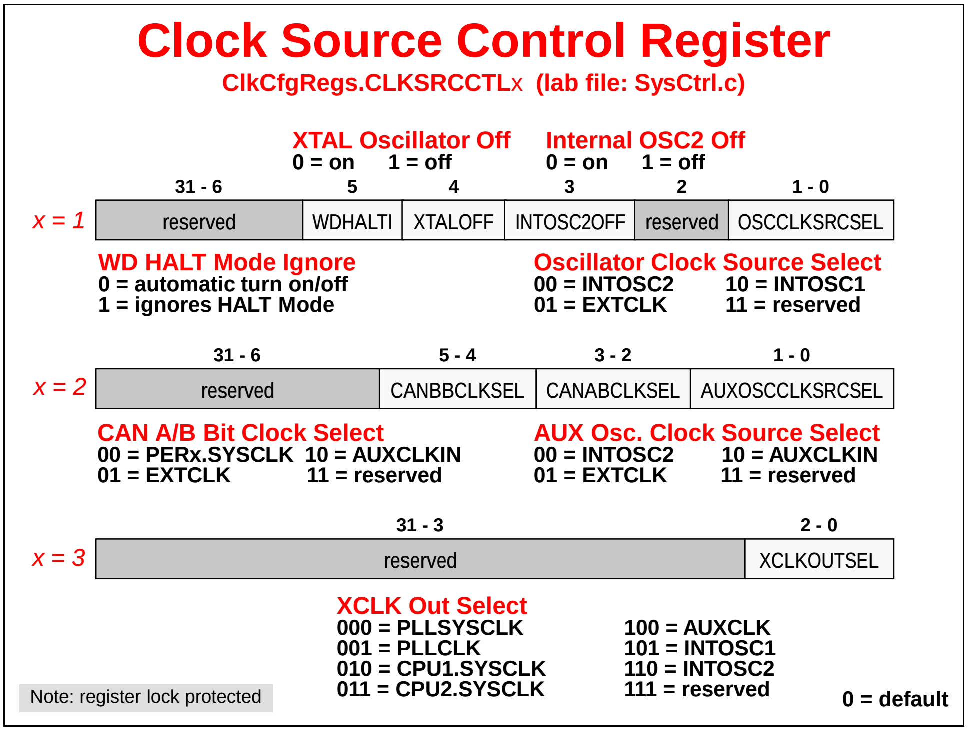
The PLL has a 7-bit integer and 2-bit fractional ratio control to select different CPU clock rates. The C28x CPU provides a SYSCLK clock signal. This signal is prescaled to provide a clock source for some of the on-chip communication peripherals through the low-speed peripheral clock prescaler. Other peripherals are clocked by SYSCLK and use their own clock prescalers for operation.

F2837xD Dual-Core System Clock



F2837xD Dual-Core System Clock

The PLL system clock is fed to both the CPU1 and CPU2 subsystems. By default, all peripherals are assigned to the CPU1 subsystem. Using the CPU selection register, each individual peripheral can be assigned to either the CPU1 or CPU2 subsystem. The clock for the EPWM modules are limited to 100 MHz, and by using the peripheral clock divider selection register, this clock can be divided down to meet this specification.



Clock Source Control Register

ClkCfgRegs.CLKSRCCTLx (lab file: SysCtrl.c)

XTAL Oscillator Off
0 = on 1 = off

Internal OSC2 Off
0 = on 1 = off

	31 - 6	5	4	3	2	1 - 0
x = 1	reserved	WDHALTI	XTALOFF	INTOSC2OFF	reserved	OSCCLKSRCSEL

WD HALT Mode Ignore
0 = automatic turn on/off
1 = ignores HALT Mode

Oscillator Clock Source Select
00 = INTOSC2 10 = INTOSC1
01 = EXTCLK 11 = reserved

	31 - 6	5 - 4	3 - 2	1 - 0
x = 2	reserved	CANBBCLKSEL	CANABCLKSEL	AUXOSCCLKSRCSEL

CAN A/B Bit Clock Select
00 = PERx.SYSCLK 10 = AUXCLKIN
01 = EXTCLK 11 = reserved

AUX Osc. Clock Source Select
00 = INTOSC2 10 = AUXCLKIN
01 = EXTCLK 11 = reserved

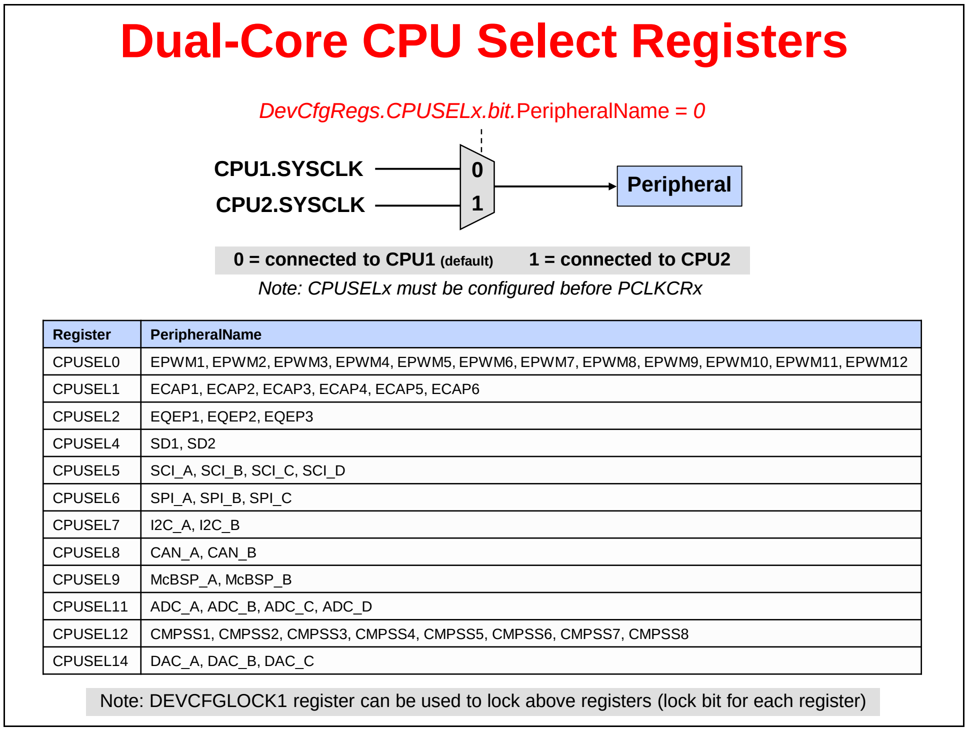
	31 - 3	2 - 0
x = 3	reserved	XCLKOUTSEL

XCLK Out Select
000 = PLLSYSCLK
001 = PLLCLK
010 = CPU1.SYSCLK
011 = CPU2.SYSCLK

100 = AUXCLK
101 = INTOSC1
110 = INTOSC2
111 = reserved

Note: register lock protected

0 = default



Dual-Core CPU Select Registers

*DevCfgRegs.CPUSELx.bit.*PeripheralName = *0*

CPU1.SYSCLK

CPU2.SYSCLK

0

1

Peripheral

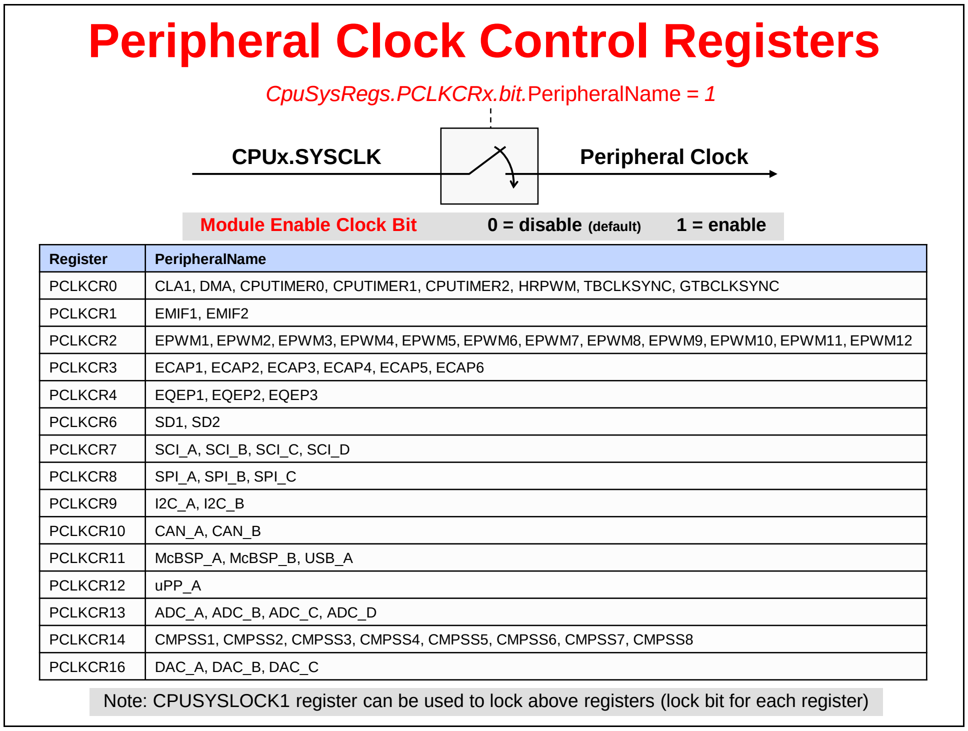
0 = connected to CPU1 (default) **1 = connected to CPU2**

Note: CPUSELx must be configured before PCLKCRx

Register	PeripheralName
CPUSEL0	EPWM1, EPWM2, EPWM3, EPWM4, EPWM5, EPWM6, EPWM7, EPWM8, EPWM9, EPWM10, EPWM11, EPWM12
CPUSEL1	ECAP1, ECAP2, ECAP3, ECAP4, ECAP5, ECAP6
CPUSEL2	EQEP1, EQEP2, EQEP3
CPUSEL4	SD1, SD2
CPUSEL5	SCI_A, SCI_B, SCI_C, SCI_D
CPUSEL6	SPI_A, SPI_B, SPI_C
CPUSEL7	I2C_A, I2C_B
CPUSEL8	CAN_A, CAN_B
CPUSEL9	McBSP_A, McBSP_B
CPUSEL11	ADC_A, ADC_B, ADC_C, ADC_D
CPUSEL12	CMPSS1, CMPSS2, CMPSS3, CMPSS4, CMPSS5, CMPSS6, CMPSS7, CMPSS8
CPUSEL14	DAC_A, DAC_B, DAC_C

Note: DEVCFGLOCK1 register can be used to lock above registers (lock bit for each register)

The dual-core CPU select register selects either CPU1 or CPU2 as the clock source for each peripheral. The peripheral clock control register allows individual peripheral clock signals to be enabled or disabled. If a peripheral is not being used, its clock signal could be disabled, thus reducing power consumption.



Peripheral Clock Control Registers

*CpuSysRegs.PCLKCRx.bit.*PeripheralName = *1*

CPUx.SYSCLK

Peripheral Clock

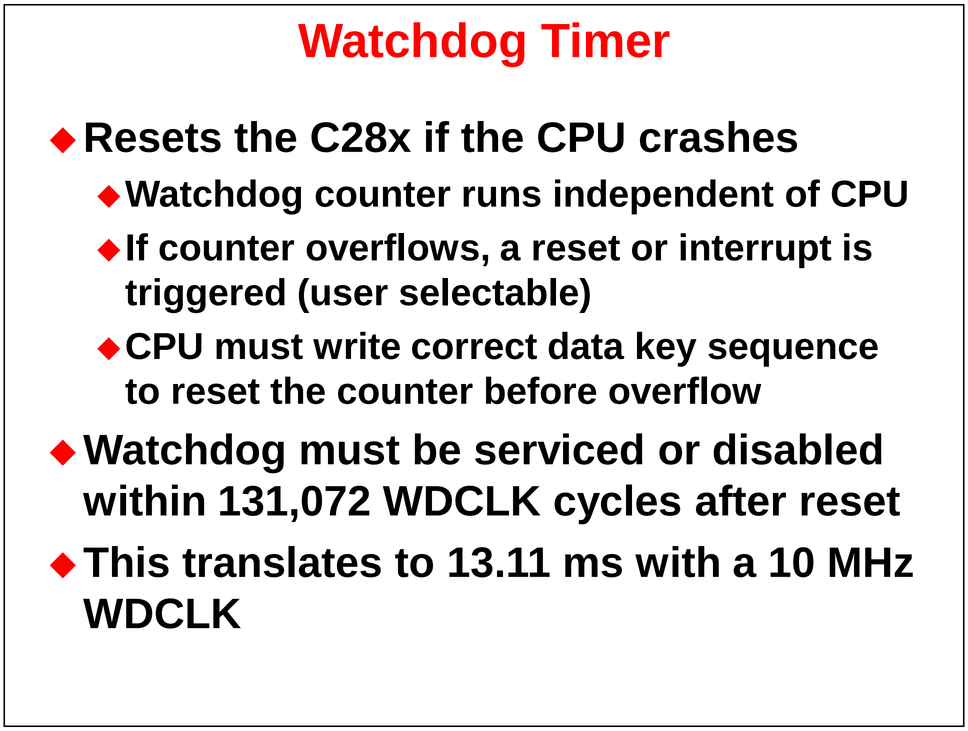
Module Enable Clock Bit **0 = disable** (default) **1 = enable**

Register	PeripheralName
PCLKCR0	CLA1, DMA, CPUTIMER0, CPUTIMER1, CPUTIMER2, HRPWM, TBCLKSYNC, GTBCLKSYNC
PCLKCR1	EMIF1, EMIF2
PCLKCR2	EPWM1, EPWM2, EPWM3, EPWM4, EPWM5, EPWM6, EPWM7, EPWM8, EPWM9, EPWM10, EPWM11, EPWM12
PCLKCR3	ECAP1, ECAP2, ECAP3, ECAP4, ECAP5, ECAP6
PCLKCR4	EQEP1, EQEP2, EQEP3
PCLKCR6	SD1, SD2
PCLKCR7	SCI_A, SCI_B, SCI_C, SCI_D
PCLKCR8	SPI_A, SPI_B, SPI_C
PCLKCR9	I2C_A, I2C_B
PCLKCR10	CAN_A, CAN_B
PCLKCR11	McBSP_A, McBSP_B, USB_A
PCLKCR12	uPP_A
PCLKCR13	ADC_A, ADC_B, ADC_C, ADC_D
PCLKCR14	CMPSS1, CMPSS2, CMPSS3, CMPSS4, CMPSS5, CMPSS6, CMPSS7, CMPSS8
PCLKCR16	DAC_A, DAC_B, DAC_C

Note: CPUSYSLOCK1 register can be used to lock above registers (lock bit for each register)

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.



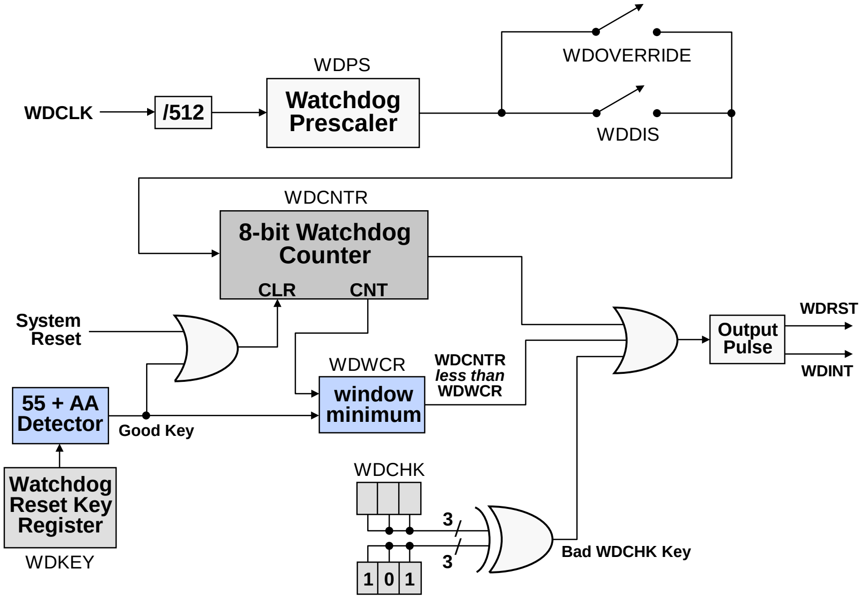
Watchdog Timer

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

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

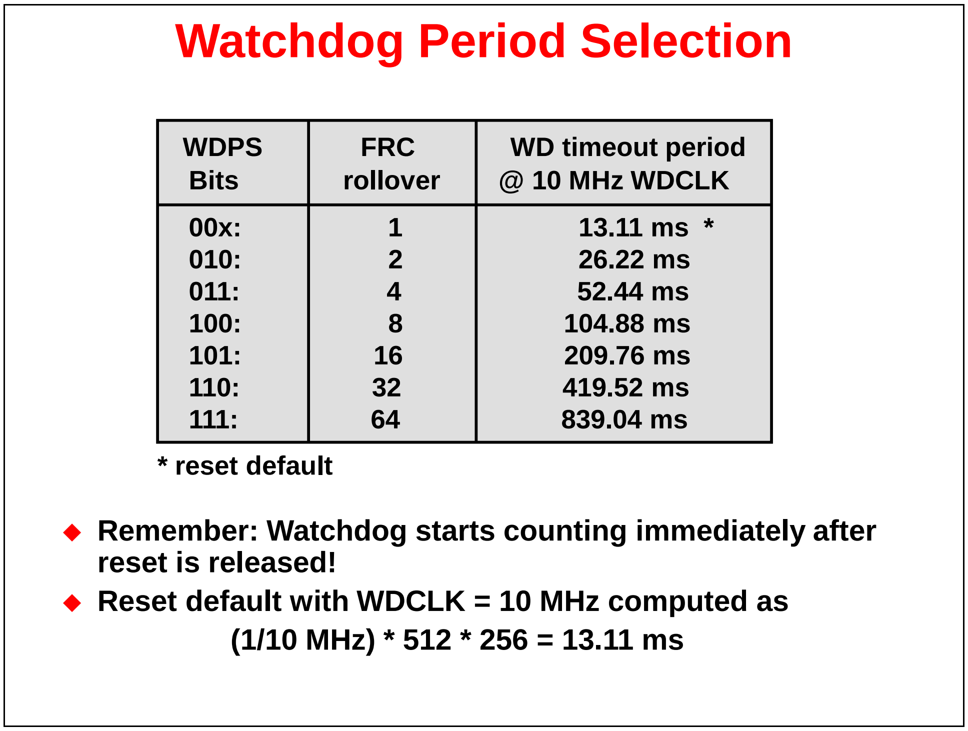
The watchdog timer starts running immediately after system power-up/reset, and must be dealt with by software soon after. Specifically, the watchdog must be serviced or disabled within 13.11 milliseconds (using a 10 MHz watchdog clock) after any reset before a watchdog initiated reset will occur. This 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.

Watchdog Timer Module



The watchdog clock is divided by 512 and prescaled, 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.



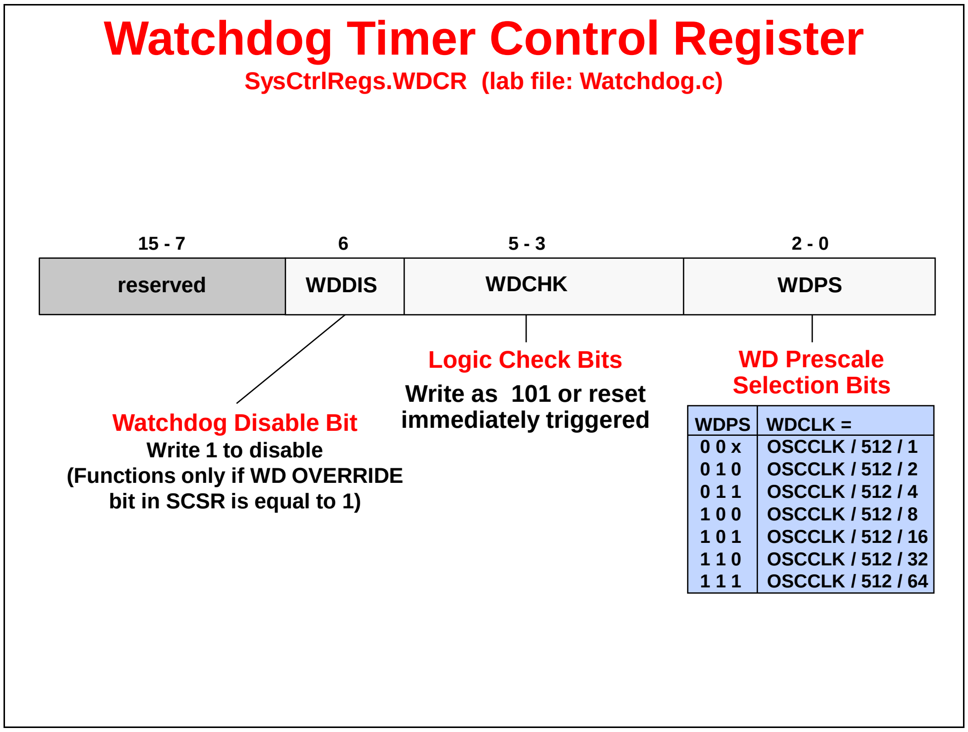
Watchdog Period Selection

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

* reset default

- Remember: Watchdog starts counting immediately after reset is released!
- Reset default with WDCLK = 10 MHz computed as

  (1/10 MHz) * 512 * 256 = 13.11 ms



Watchdog Timer Control Register

SysCtrlRegs.WDCR (lab file: Watchdog.c)

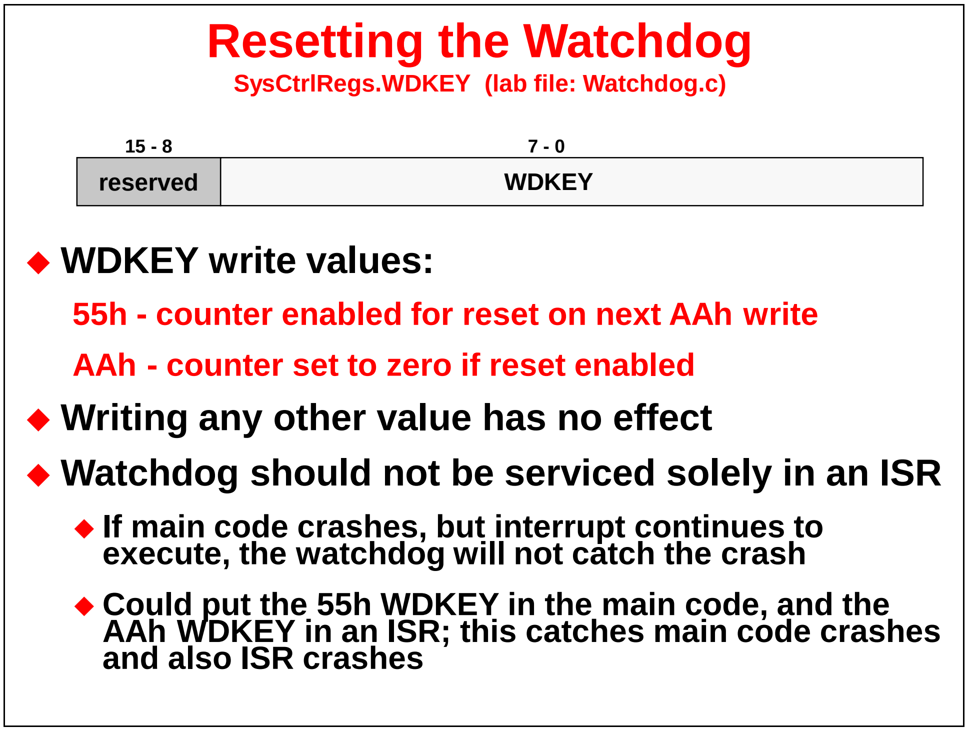
15 - 7	6	5 - 3	2 - 0
reserved	WDDIS	WDCHK	WDPS

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

Logic Check Bits
Write as 101 or reset immediately triggered

WD Prescale Selection Bits

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

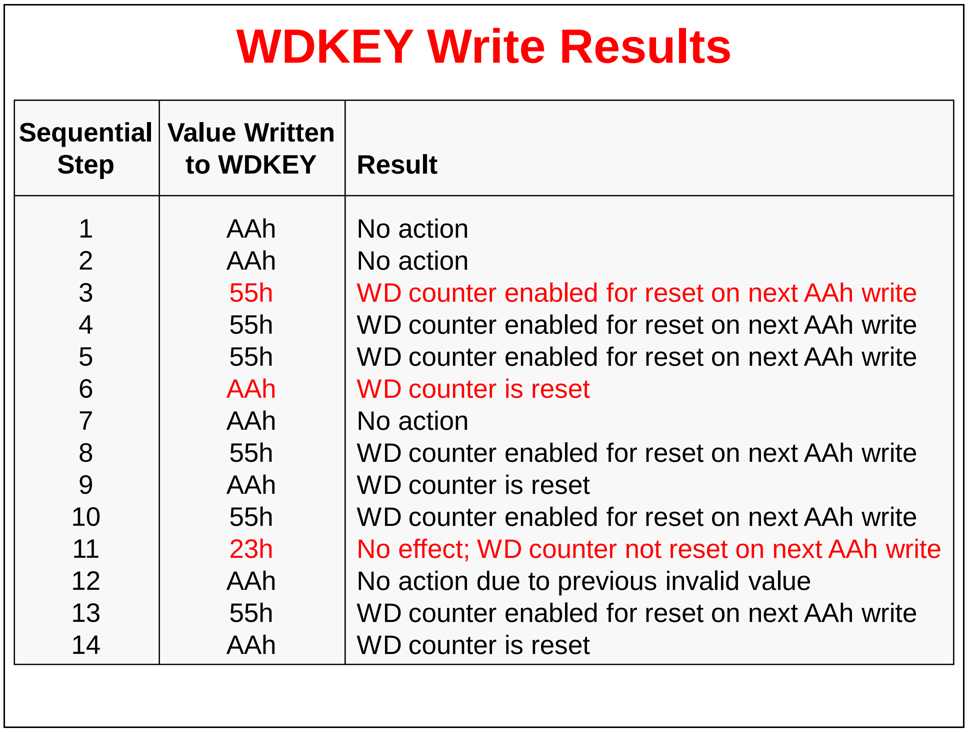


Resetting the Watchdog

SysCtrlRegs.WDKEY (lab file: Watchdog.c)

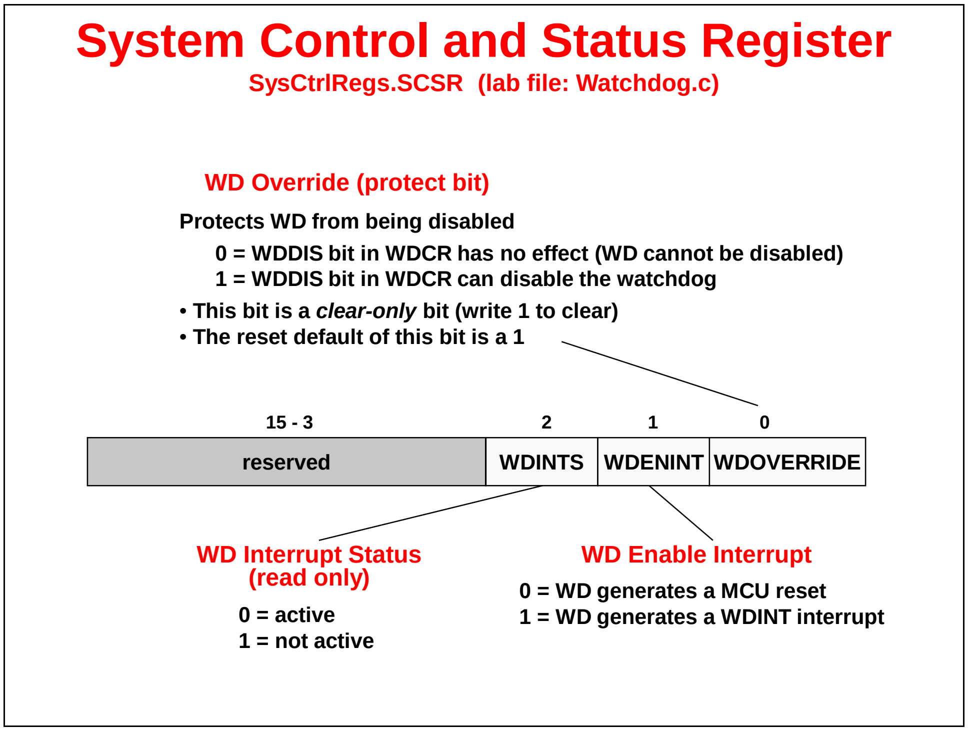
15 - 8	7 - 0
reserved	WDKEY

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



WDKEY Write Results

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



System Control and Status Register

SysCtrlRegs.SCSR (lab file: Watchdog.c)

WD Override (protect bit)

Protects WD from being disabled

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

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

15 - 3	2	1	0
reserved	WDINTS	WDENINT	WDOVERRIDE

WD Interrupt Status (read only)

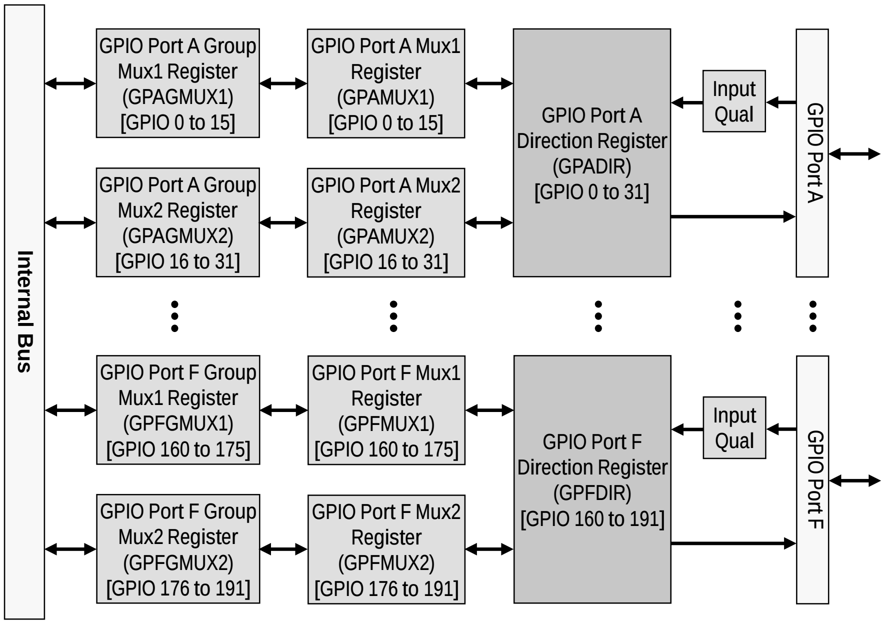
0 = active
1 = not active

WD Enable Interrupt

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

General Purpose Digital I/O

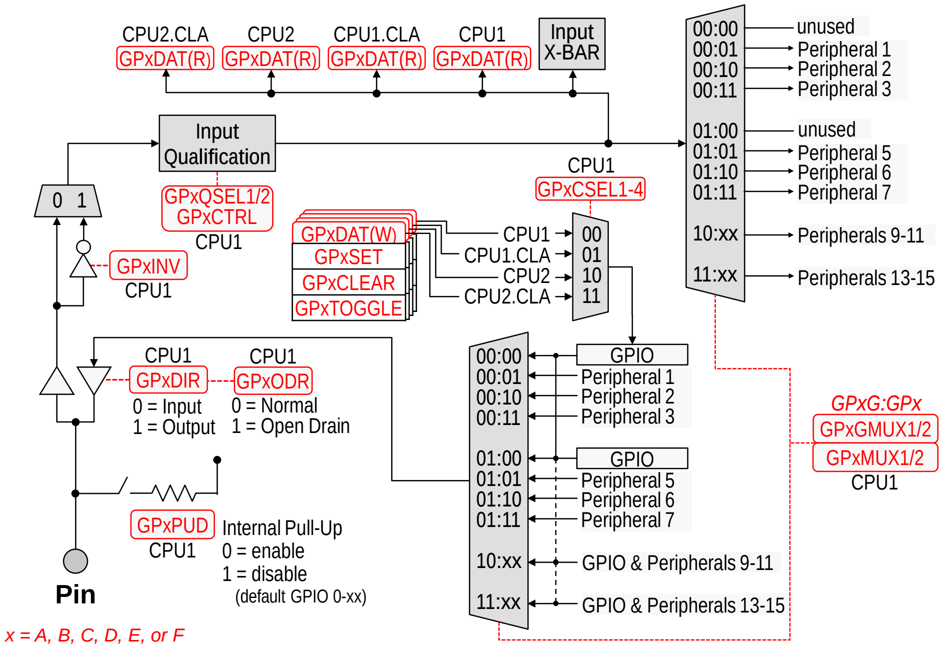
F28x7x GPIO Grouping Overview


Internal Bus
GPIO Port A Group Mux1 Register (GPAGMUX1) [GPIO 0 to 15]
GPIO Port A Mux1 Register (GPAMUX1) [GPIO 0 to 15]
GPIO Port A Group Mux2 Register (GPAGMUX2) [GPIO 16 to 31]
GPIO Port A Mux2 Register (GPAMUX2) [GPIO 16 to 31]
GPIO Port A Direction Register (GPADIR) [GPIO 0 to 31]
Input Qual
GPIO Port A
GPIO Port F Group Mux1 Register (GPFGMUX1) [GPIO 160 to 175]
GPIO Port F Mux1 Register (GPFMUX1) [GPIO 160 to 175]
GPIO Port F Group Mux2 Register (GPFGMUX2) [GPIO 176 to 191]
GPIO Port F Mux2 Register (GPFMUX2) [GPIO 176 to 191]
GPIO Port F Direction Register (GPFDIR) [GPIO 160 to 191]
Input Qual
GPIO Port F

The F2837xD 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 CPU1, CPU1 CLA, CPU2, or CPU2 CLA. However, the peripheral multiplexing and pin assignment can only be configured by CPU1. 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 I/O pins are grouped into six ports, and each port has 32 pins except for the sixth port which has nine pins (i.e. the remaining I/O 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.

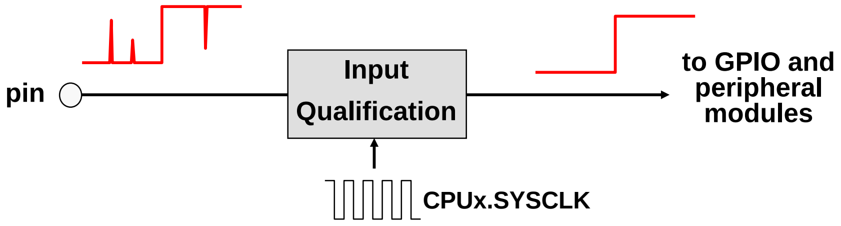
If the pin is configured as GPIO, a direction (DIR) register is used to specify the pin as either an input or output. By default, all GPIO pins are inputs. The current state of a GPIO pin corresponds to a bit value in a data (DAT) register, regardless if the pin is configured as GPIO or a peripheral 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.

F2837xD GPIO Pin Block Diagram

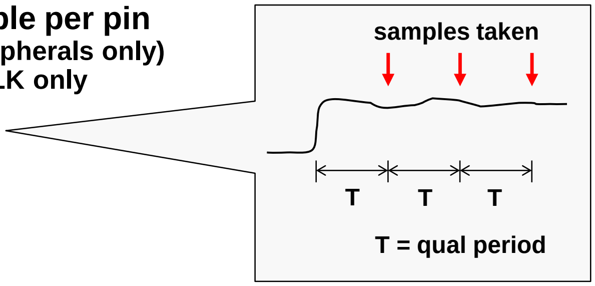


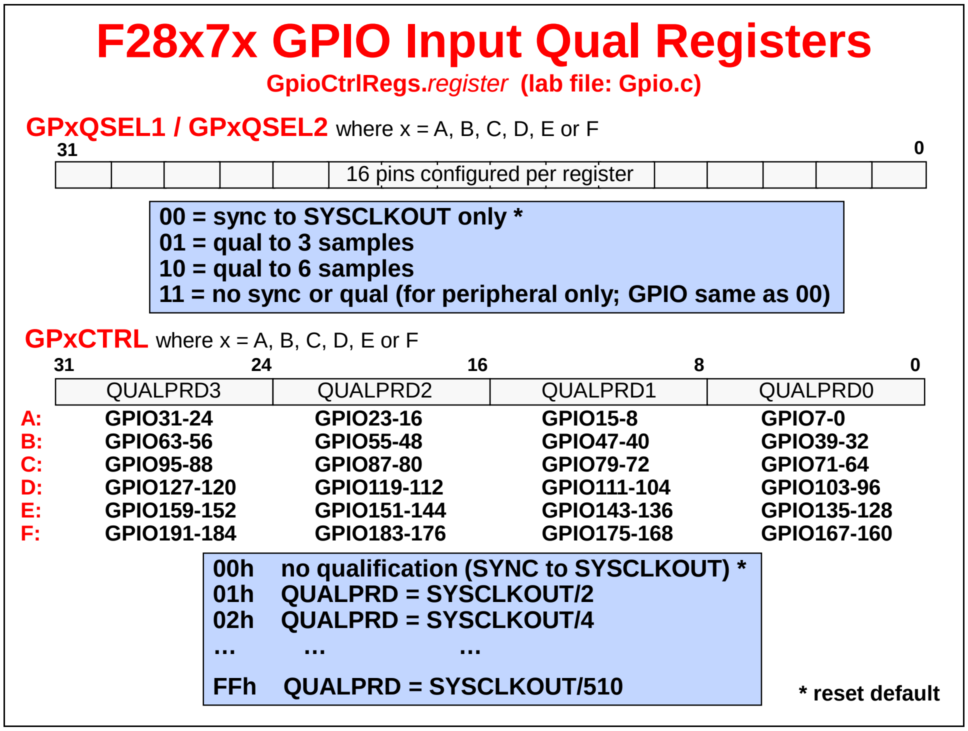
The input qualification scheme is very flexible, and the type of input qualification can be configured for each GPIO pin individually. In the case of a GPIO input pin, the qualification can be specified as only synchronize to SYSCLKOUT or qualification by a sampling window. For pins that are configured as peripheral inputs, the input can also be asynchronous in addition to synchronized to SYSCLKOUT or qualified by a sampling window.

F28x7x GPIO Input Qualification



- Qualification available on ports A - F
- Individually selectable per pin
  - no qualification (peripherals only)
  - sync to CPUx.SYSCLK only
  - qualify 3 samples
  - qualify 6 samples





F28x7x GPIO Input Qual Registers

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

GPxQSEL1 / GPxQSEL2 where x = A, B, C, D, E or F

31 — 0

16 pins configured per register

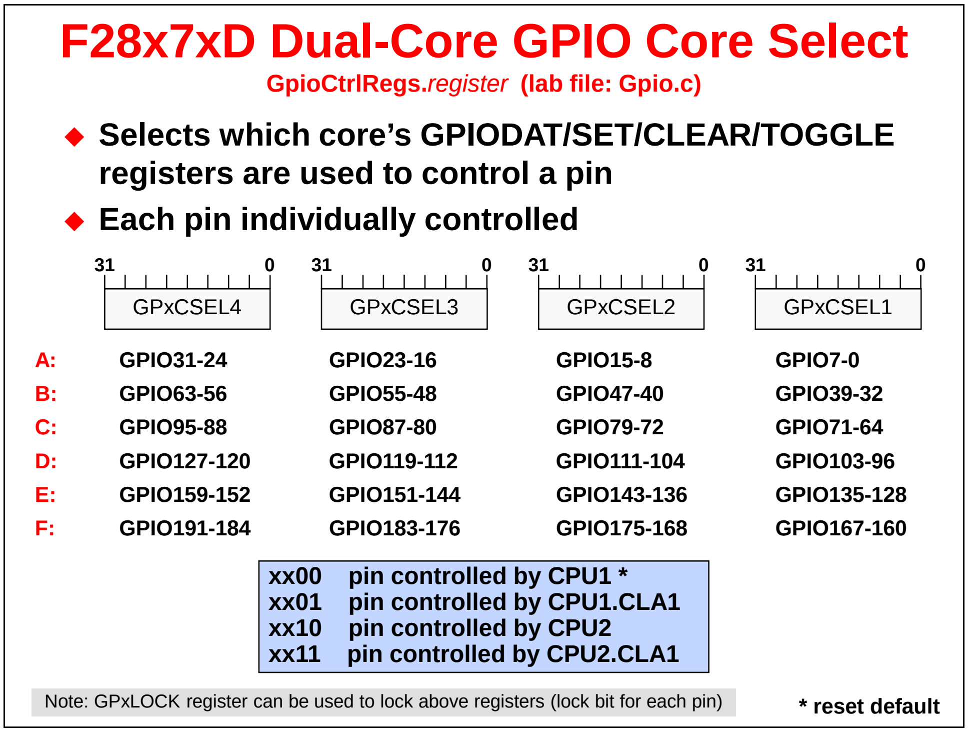
- 00 = sync to SYSCLKOUT only *
- 01 = qual to 3 samples
- 10 = qual to 6 samples
- 11 = no sync or qual (for peripheral only; GPIO same as 00)

GPxCTRL where x = A, B, C, D, E or F

	31–24	24–16	16–8	8–0
	QUALPRD3	QUALPRD2	QUALPRD1	QUALPRD0
A:	GPIO31-24	GPIO23-16	GPIO15-8	GPIO7-0
B:	GPIO63-56	GPIO55-48	GPIO47-40	GPIO39-32
C:	GPIO95-88	GPIO87-80	GPIO79-72	GPIO71-64
D:	GPIO127-120	GPIO119-112	GPIO111-104	GPIO103-96
E:	GPIO159-152	GPIO151-144	GPIO143-136	GPIO135-128
F:	GPIO191-184	GPIO183-176	GPIO175-168	GPIO167-160

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

* reset default



F28x7xD Dual-Core GPIO Core Select

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

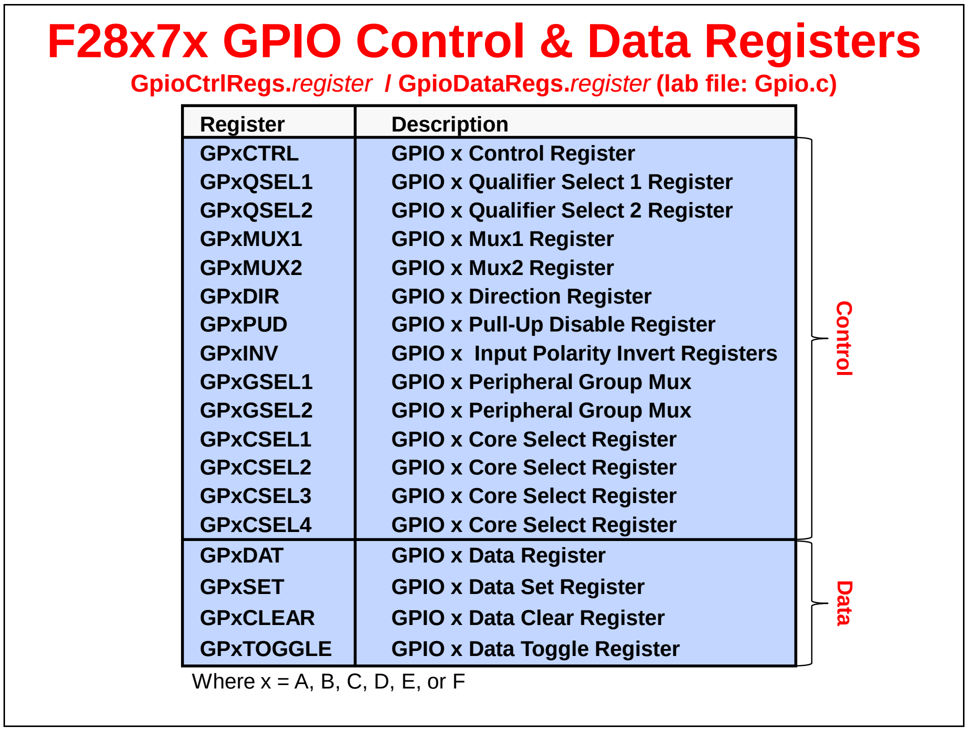
- Selects which core's GPIODAT/SET/CLEAR/TOGGLE registers are used to control a pin
- Each pin individually controlled

	GPxCSEL4 (31–0)	GPxCSEL3 (31–0)	GPxCSEL2 (31–0)	GPxCSEL1 (31–0)
A:	GPIO31-24	GPIO23-16	GPIO15-8	GPIO7-0
B:	GPIO63-56	GPIO55-48	GPIO47-40	GPIO39-32
C:	GPIO95-88	GPIO87-80	GPIO79-72	GPIO71-64
D:	GPIO127-120	GPIO119-112	GPIO111-104	GPIO103-96
E:	GPIO159-152	GPIO151-144	GPIO143-136	GPIO135-128
F:	GPIO191-184	GPIO183-176	GPIO175-168	GPIO167-160

xx00	pin controlled by CPU1 *
xx01	pin controlled by CPU1.CLA1
xx10	pin controlled by CPU2
xx11	pin controlled by CPU2.CLA1

Note: GPxLOCK register can be used to lock above registers (lock bit for each pin)

* reset default



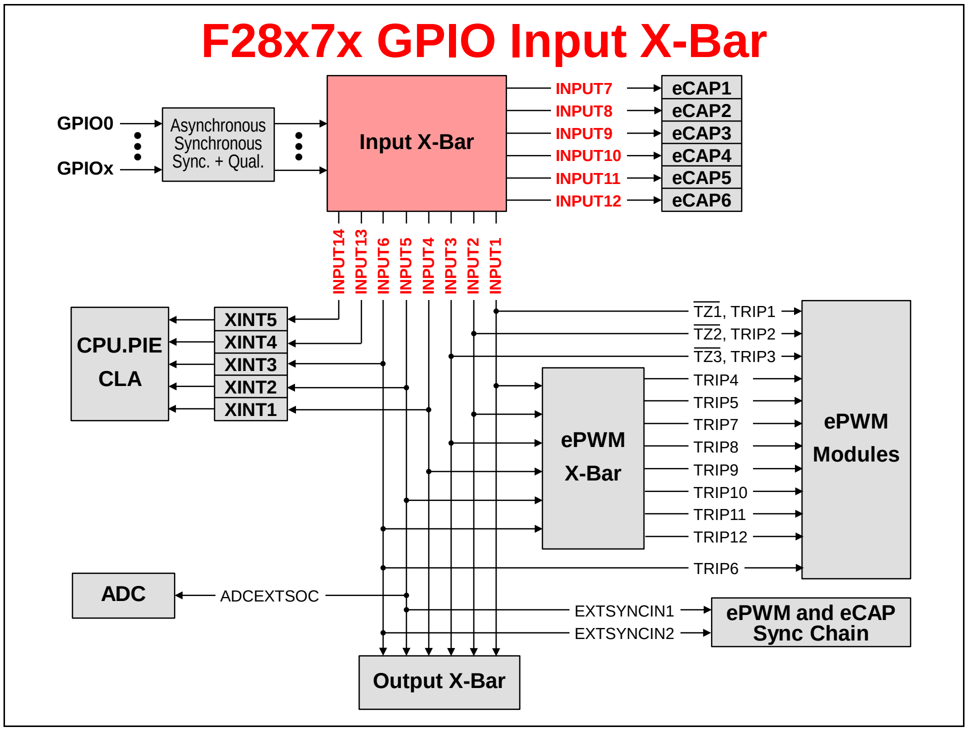
F28x7x GPIO Control & Data Registers

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

Register	Description	
GPxCTRL	GPIO x Control Register	Control
GPxQSEL1	GPIO x Qualifier Select 1 Register	
GPxQSEL2	GPIO x Qualifier Select 2 Register	
GPxMUX1	GPIO x Mux1 Register	
GPxMUX2	GPIO x Mux2 Register	
GPxDIR	GPIO x Direction Register	
GPxPUD	GPIO x Pull-Up Disable Register	
GPxINV	GPIO x Input Polarity Invert Registers	
GPxGSEL1	GPIO x Peripheral Group Mux	
GPxGSEL2	GPIO x Peripheral Group Mux	
GPxCSEL1	GPIO x Core Select Register	
GPxCSEL2	GPIO x Core Select Register	
GPxCSEL3	GPIO x Core Select Register	
GPxCSEL4	GPIO x Core Select Register	
GPxDAT	GPIO x Data Register	Data
GPxSET	GPIO x Data Set Register	
GPxCLEAR	GPIO x Data Clear Register	
GPxTOGGLE	GPIO x Data Toggle Register	

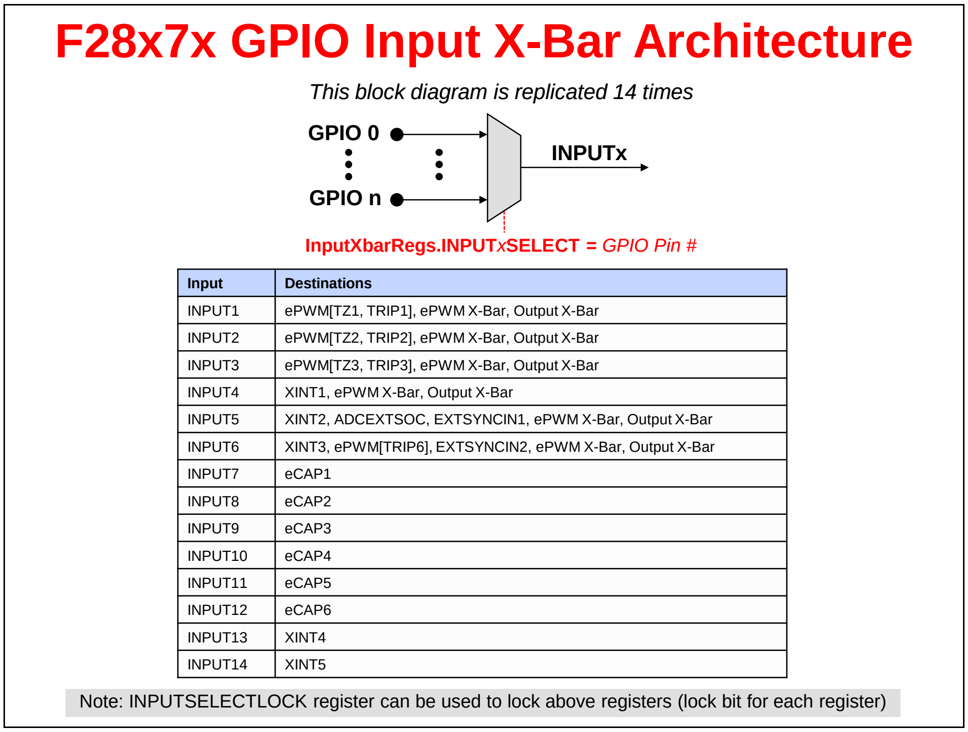
Where x = A, B, C, D, E, or F

GPIO Input X-Bar



F28x7x 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.



F28x7x GPIO Input X-Bar Architecture

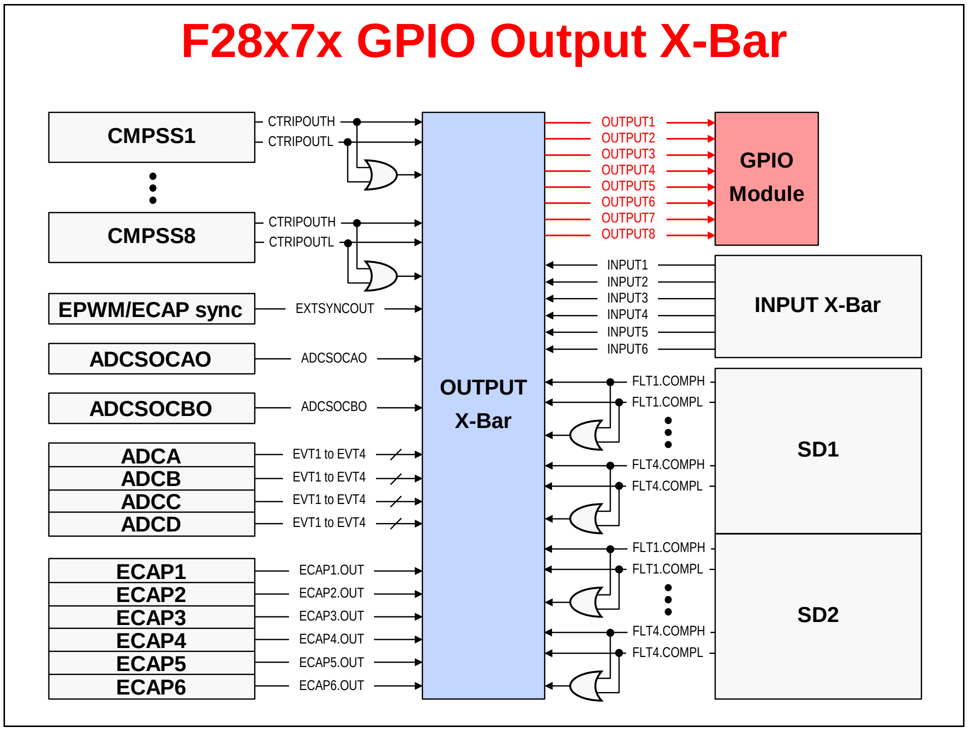
This block diagram is replicated 14 times

InputXbarRegs.INPUT*x*SELECT = *GPIO Pin #*

Input	Destinations
INPUT1	ePWM[TZ1, TRIP1], ePWM X-Bar, Output X-Bar
INPUT2	ePWM[TZ2, TRIP2], ePWM X-Bar, Output X-Bar
INPUT3	ePWM[TZ3, TRIP3], ePWM X-Bar, Output X-Bar
INPUT4	XINT1, ePWM X-Bar, Output X-Bar
INPUT5	XINT2, ADCEXTSOC, EXTSYNCIN1, ePWM X-Bar, Output X-Bar
INPUT6	XINT3, ePWM[TRIP6], EXTSYNCIN2, ePWM X-Bar, Output X-Bar
INPUT7	eCAP1
INPUT8	eCAP2
INPUT9	eCAP3
INPUT10	eCAP4
INPUT11	eCAP5
INPUT12	eCAP6
INPUT13	XINT4
INPUT14	XINT5

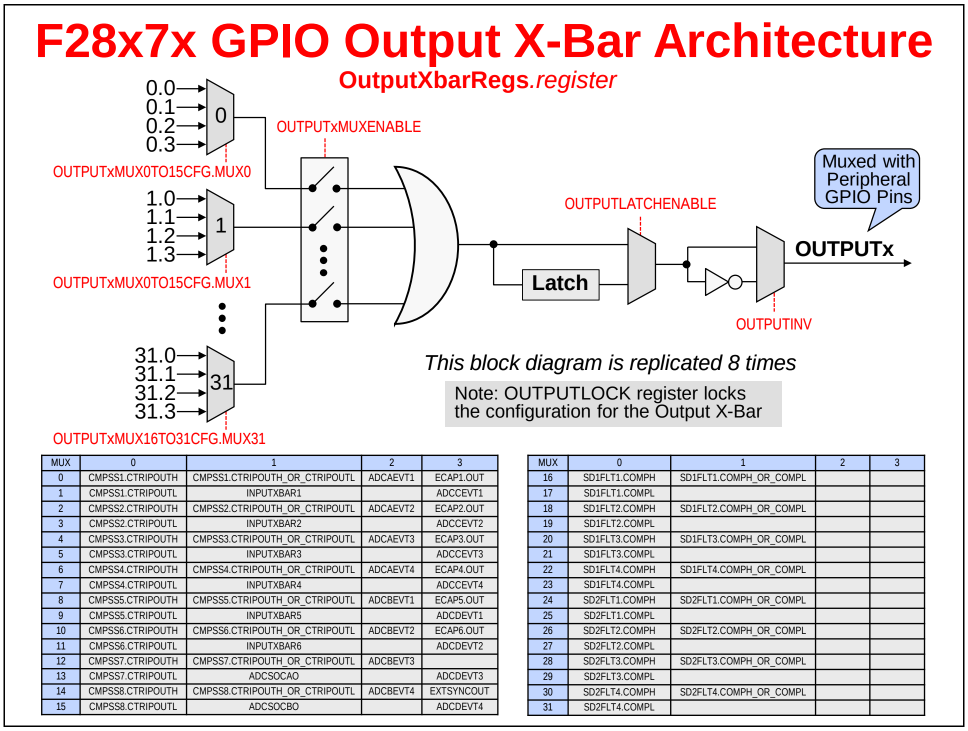
Note: INPUTSELECTLOCK register can be used to lock above registers (lock bit for each register)

GPIO Output X-Bar



F28x7x 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.



F28x7x GPIO Output X-Bar Architecture

OutputXbarRegs.*register*

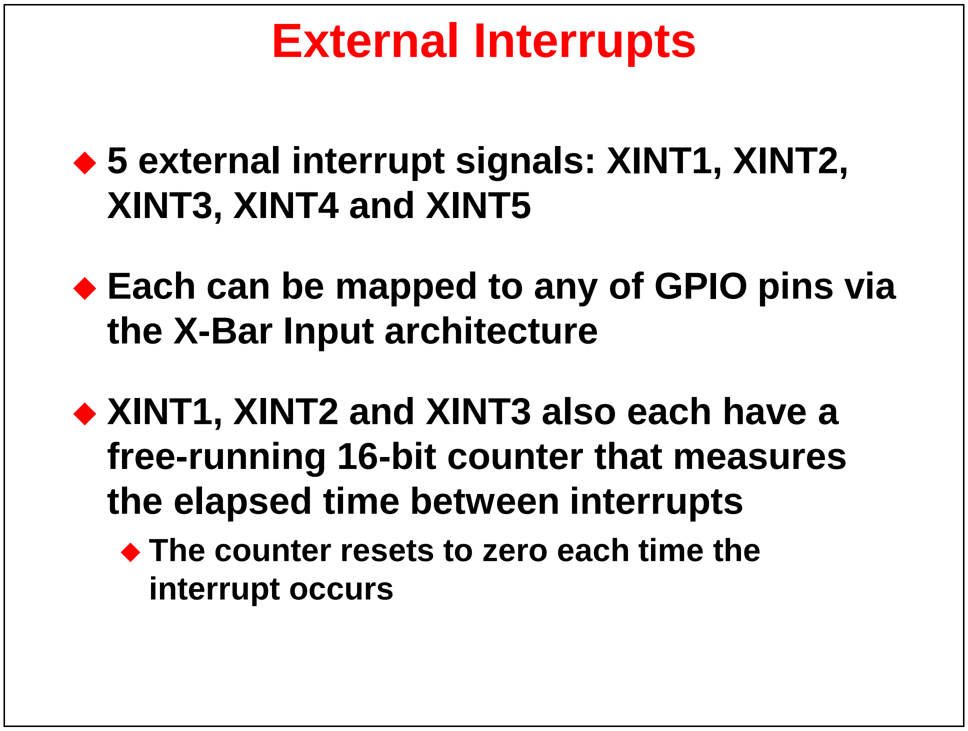
This block diagram is replicated 8 times

Note: OUTPUTLOCK register locks the configuration for the Output X-Bar

MUX	0	1	2	3
0	CMPSS1.CTRIPOUTH	CMPSS1.CTRIPOUTH_OR_CTRIPOUTL	ADCAEVT1	ECAP1.OUT
1	CMPSS1.CTRIPOUTL	INPUTXBAR1		ADCCEVT1
2	CMPSS2.CTRIPOUTH	CMPSS2.CTRIPOUTH_OR_CTRIPOUTL	ADCAEVT2	ECAP2.OUT
3	CMPSS2.CTRIPOUTL	INPUTXBAR2		ADCCEVT2
4	CMPSS3.CTRIPOUTH	CMPSS3.CTRIPOUTH_OR_CTRIPOUTL	ADCAEVT3	ECAP3.OUT
5	CMPSS3.CTRIPOUTL	INPUTXBAR3		ADCCEVT3
6	CMPSS4.CTRIPOUTH	CMPSS4.CTRIPOUTH_OR_CTRIPOUTL	ADCAEVT4	ECAP4.OUT
7	CMPSS4.CTRIPOUTL	INPUTXBAR4		ADCCEVT4
8	CMPSS5.CTRIPOUTH	CMPSS5.CTRIPOUTH_OR_CTRIPOUTL	ADCBEVT1	ECAP5.OUT
9	CMPSS5.CTRIPOUTL	INPUTXBAR5		ADCDEVT1
10	CMPSS6.CTRIPOUTH	CMPSS6.CTRIPOUTH_OR_CTRIPOUTL	ADCBEVT2	ECAP6.OUT
11	CMPSS6.CTRIPOUTL	INPUTXBAR6		ADCDEVT2
12	CMPSS7.CTRIPOUTH	CMPSS7.CTRIPOUTH_OR_CTRIPOUTL	ADCBEVT3	
13	CMPSS7.CTRIPOUTL	ADCSOCAO		ADCDEVT3
14	CMPSS8.CTRIPOUTH	CMPSS8.CTRIPOUTH_OR_CTRIPOUTL	ADCBEVT4	EXTSYNCOUT
15	CMPSS8.CTRIPOUTL	ADCSOCBO		ADCDEVT4

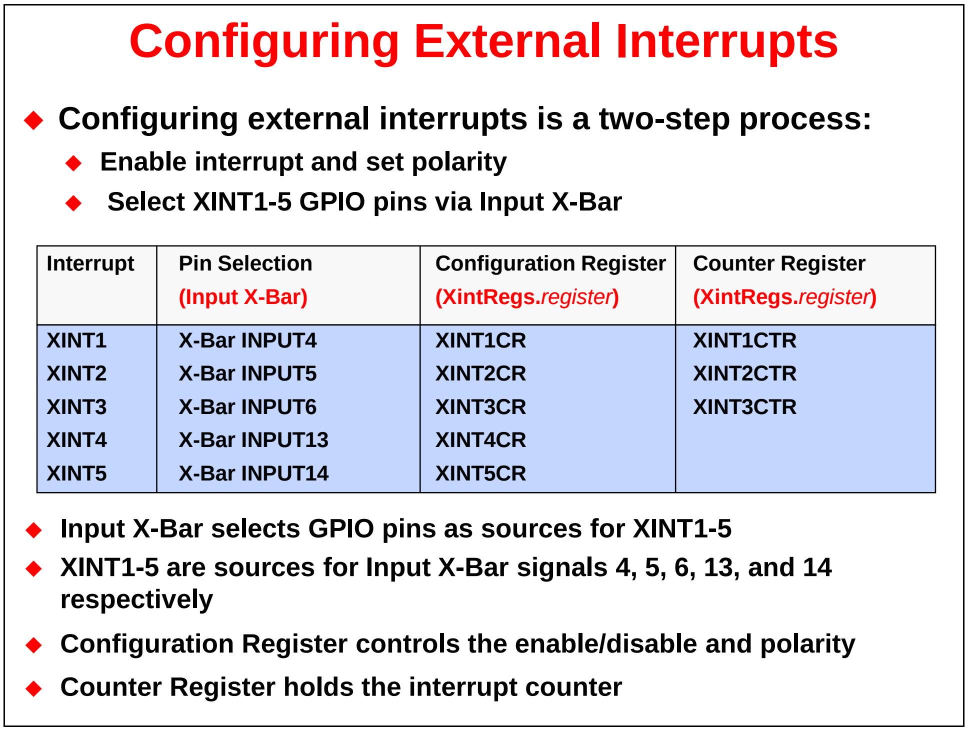
MUX	0	1	2	3
16	SD1FLT1.COMPH	SD1FLT1.COMPH_OR_COMPL		
17	SD1FLT1.COMPL			
18	SD1FLT2.COMPH	SD1FLT2.COMPH_OR_COMPL		
19	SD1FLT2.COMPL			
20	SD1FLT3.COMPH	SD1FLT3.COMPH_OR_COMPL		
21	SD1FLT3.COMPL			
22	SD1FLT4.COMPH	SD1FLT4.COMPH_OR_COMPL		
23	SD1FLT4.COMPL			
24	SD2FLT1.COMPH	SD2FLT1.COMPH_OR_COMPL		
25	SD2FLT1.COMPL			
26	SD2FLT2.COMPH	SD2FLT2.COMPH_OR_COMPL		
27	SD2FLT2.COMPL			
28	SD2FLT3.COMPH	SD2FLT3.COMPH_OR_COMPL		
29	SD2FLT3.COMPL			
30	SD2FLT4.COMPH	SD2FLT4.COMPH_OR_COMPL		
31	SD2FLT4.COMPL			

External Interrupts



External Interrupts

- 5 external interrupt signals: XINT1, XINT2, XINT3, XINT4 and XINT5
- Each can be mapped to any of GPIO pins via the X-Bar Input architecture
- XINT1, XINT2 and XINT3 also each have a free-running 16-bit counter that measures the elapsed time between interrupts
  - The counter resets to zero each time the interrupt occurs



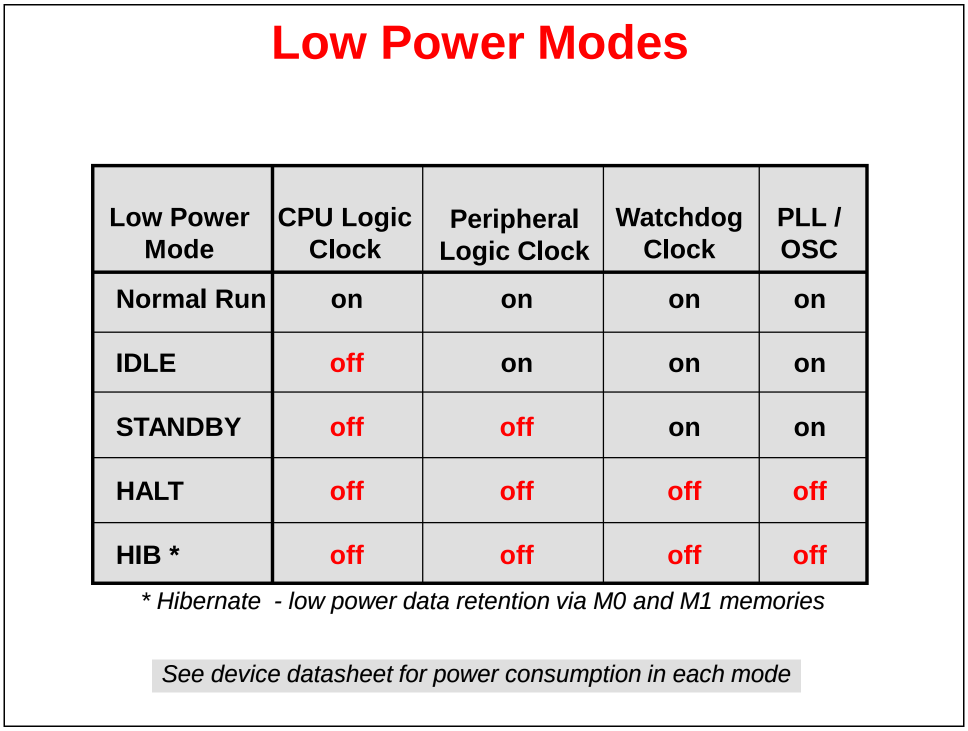
Configuring External Interrupts

- Configuring external interrupts is a two-step process:
  - Enable interrupt and set polarity
  - Select XINT1-5 GPIO pins via Input X-Bar

Interrupt	Pin Selection (Input X-Bar)	Configuration Register (XintRegs.*register*)	Counter Register (XintRegs.*register*)
XINT1	X-Bar INPUT4	XINT1CR	XINT1CTR
XINT2	X-Bar INPUT5	XINT2CR	XINT2CTR
XINT3	X-Bar INPUT6	XINT3CR	XINT3CTR
XINT4	X-Bar INPUT13	XINT4CR	
XINT5	X-Bar INPUT14	XINT5CR	

- Input X-Bar selects GPIO pins as sources for XINT1-5
- XINT1-5 are sources for Input X-Bar signals 4, 5, 6, 13, and 14 respectively
- Configuration Register controls the enable/disable and polarity
- Counter Register holds the interrupt counter

Low Power Modes

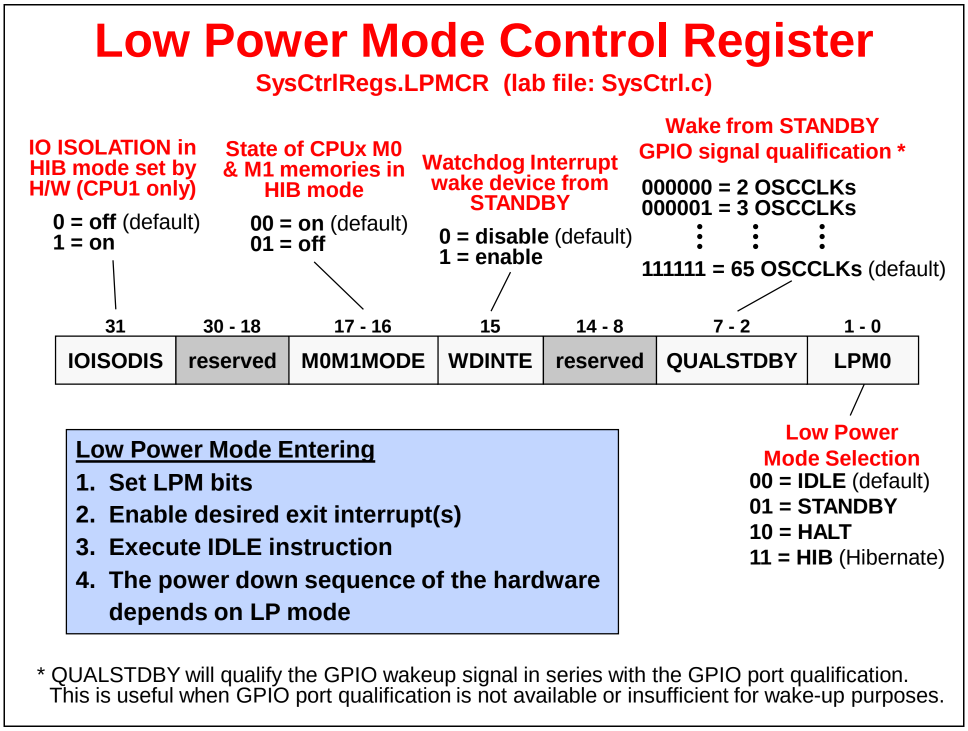


Low Power Modes

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

** Hibernate - low power data retention via M0 and M1 memories*

See device datasheet for power consumption in each mode



Low Power Mode Control Register

SysCtrlRegs.LPMCR (lab file: SysCtrl.c)

IO ISOLATION in HIB mode set by H/W (CPU1 only)
0 = off (default)
1 = on

State of CPUx M0 & M1 memories in HIB mode
00 = on (default)
01 = off

Watchdog Interrupt wake device from STANDBY
0 = disable (default)
1 = enable

Wake from STANDBY GPIO signal qualification *
000000 = 2 OSCCLKs
000001 = 3 OSCCLKs
⋮ ⋮ ⋮
111111 = 65 OSCCLKs (default)

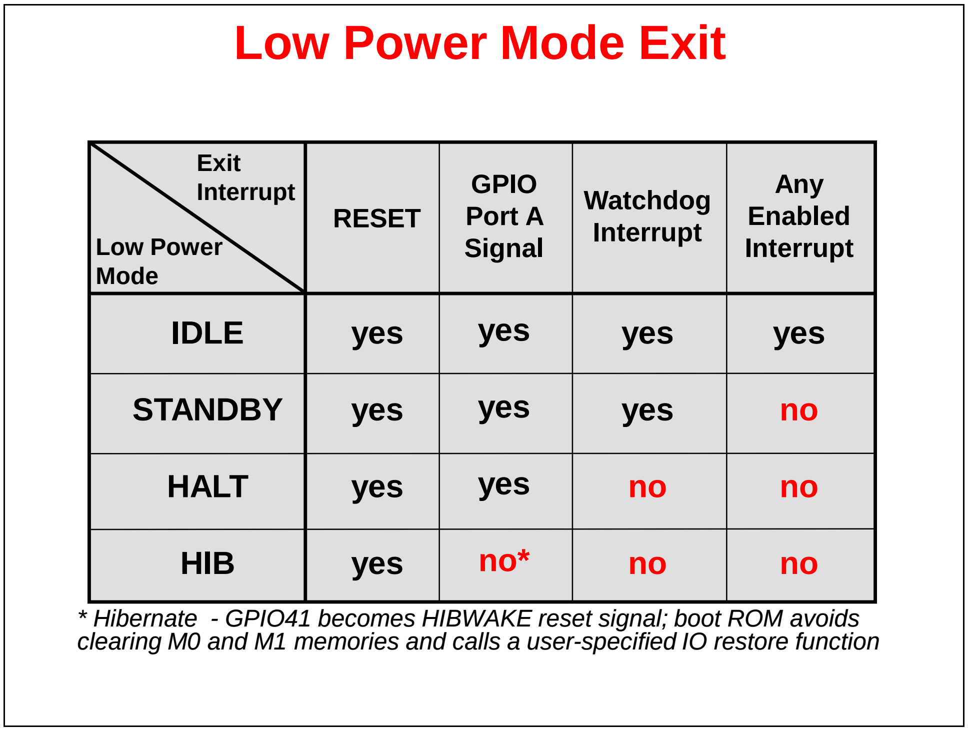
31	30 - 18	17 - 16	15	14 - 8	7 - 2	1 - 0
IOISODIS	reserved	M0M1MODE	WDINTE	reserved	QUALSTDBY	LPM0

Low Power Mode Selection
00 = IDLE (default)
01 = STANDBY
10 = HALT
11 = HIB (Hibernate)

Low Power Mode Entering

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

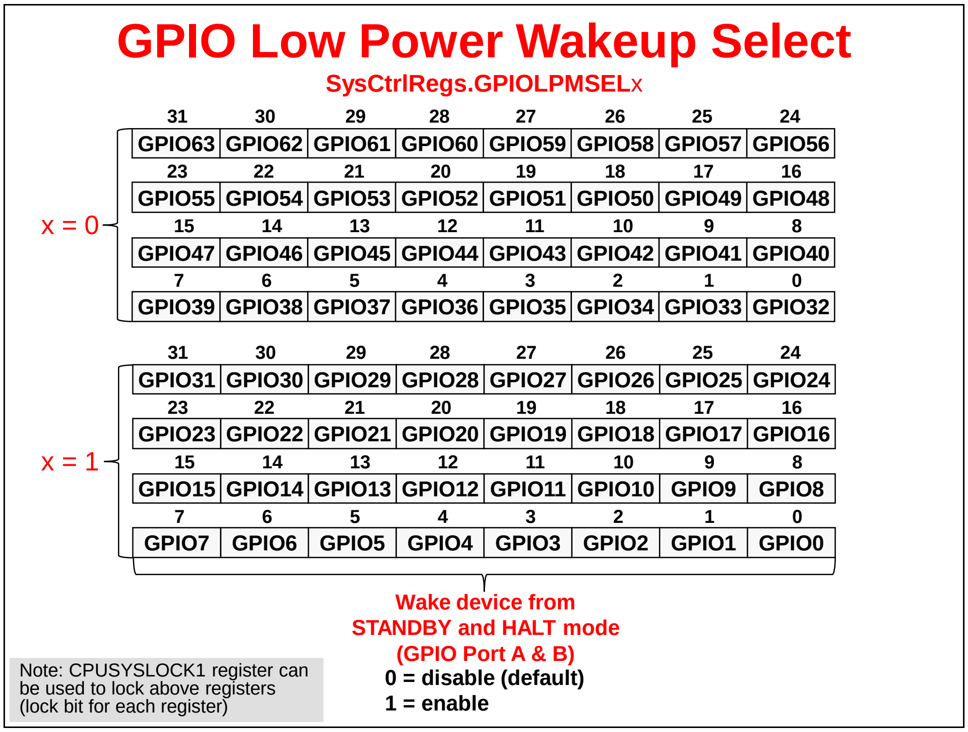
* QUALSTDBY will qualify the GPIO wakeup signal in series with the GPIO port qualification. This is useful when GPIO port qualification is not available or insufficient for wake-up purposes.



Low Power Mode Exit

Exit Interrupt / Low Power Mode	RESET	GPIO Port A Signal	Watchdog Interrupt	Any Enabled Interrupt
IDLE	yes	yes	yes	yes
STANDBY	yes	yes	yes	no
HALT	yes	yes	no	no
HIB	yes	no*	no	no

* Hibernate - GPIO41 becomes HIBWAKE reset signal; boot ROM avoids clearing M0 and M1 memories and calls a user-specified IO restore function



GPIO Low Power Wakeup Select

SysCtrlRegs.GPIOLPMSELx

x = 0

31	30	29	28	27	26	25	24
GPIO63	GPIO62	GPIO61	GPIO60	GPIO59	GPIO58	GPIO57	GPIO56
23	22	21	20	19	18	17	16
GPIO55	GPIO54	GPIO53	GPIO52	GPIO51	GPIO50	GPIO49	GPIO48
15	14	13	12	11	10	9	8
GPIO47	GPIO46	GPIO45	GPIO44	GPIO43	GPIO42	GPIO41	GPIO40
7	6	5	4	3	2	1	0
GPIO39	GPIO38	GPIO37	GPIO36	GPIO35	GPIO34	GPIO33	GPIO32

x = 1

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

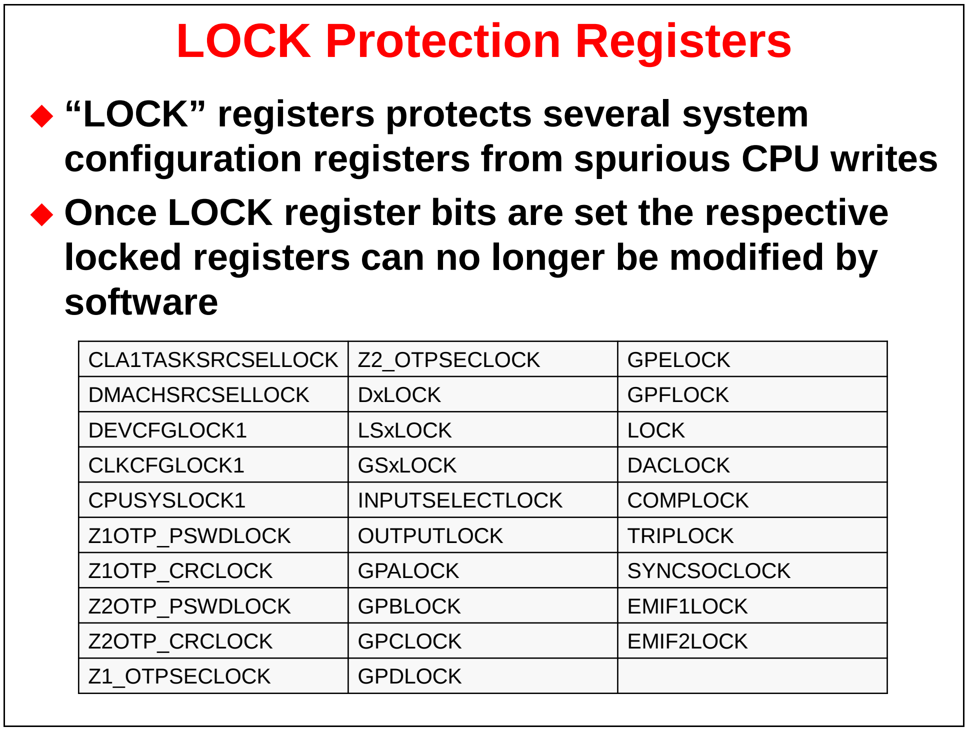
Wake device from STANDBY and HALT mode (GPIO Port A & B)

0 = disable (default)

1 = enable

Note: CPUSYSLOCK1 register can be used to lock above registers (lock bit for each register)

Register Protection

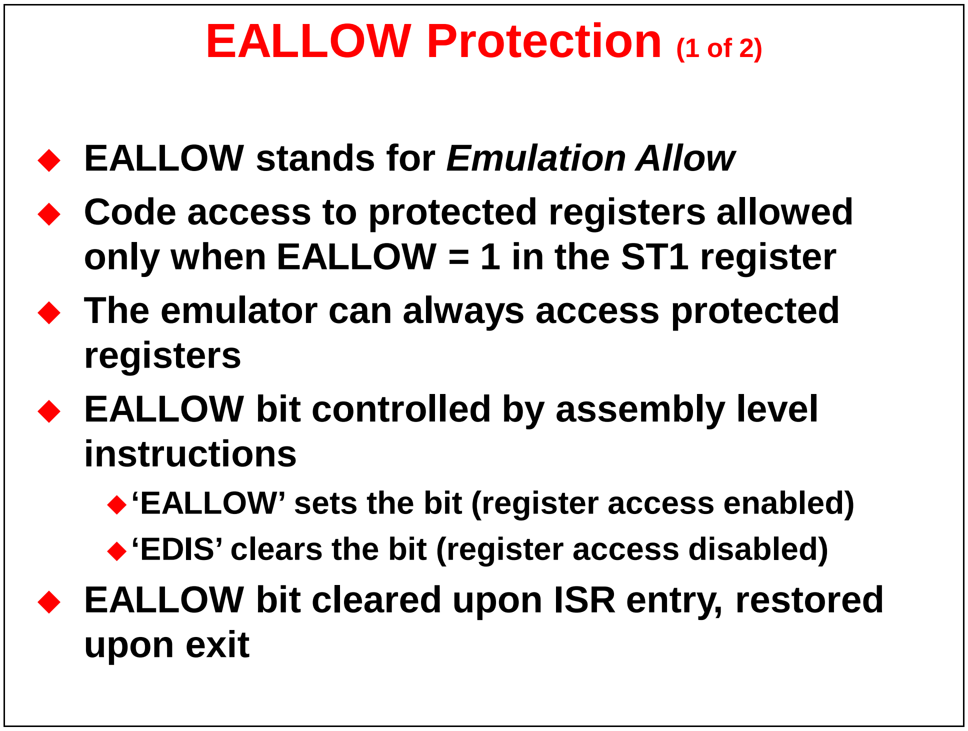


LOCK Protection Registers

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

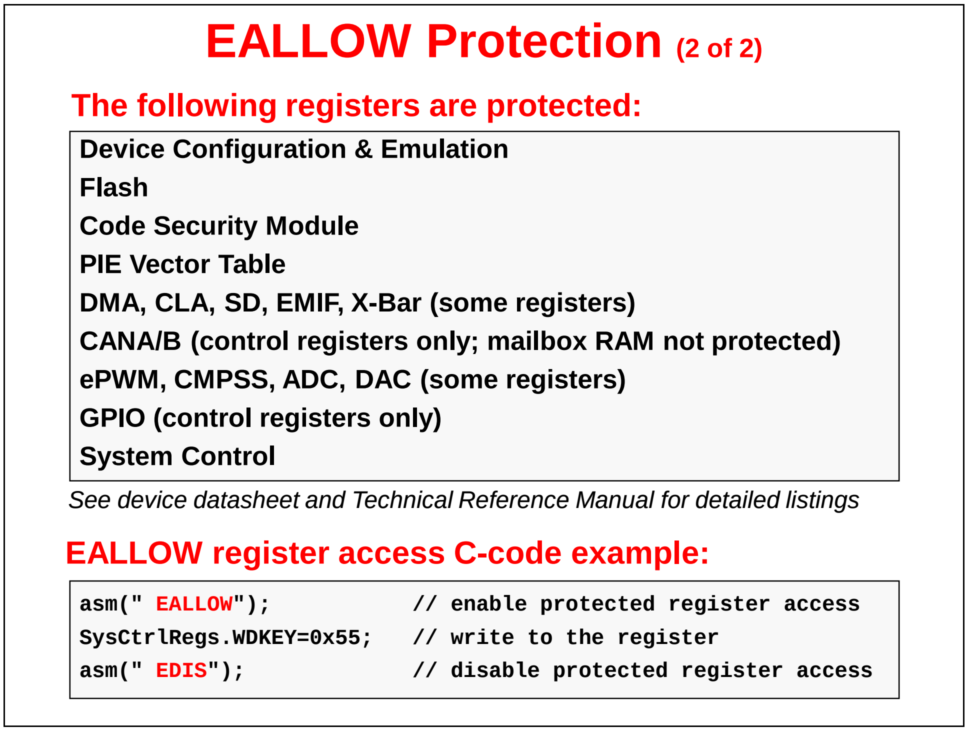
CLA1TASKSRCSELLOCK	Z2_OTPSECLOCK	GPELOCK
DMACHSRCSELLOCK	DxLOCK	GPFLOCK
DEVCFGLOCK1	LSxLOCK	LOCK
CLKCFGLOCK1	GSxLOCK	DACLOCK
CPUSYSLOCK1	INPUTSELECTLOCK	COMPLOCK
Z1OTP_PSWDLOCK	OUTPUTLOCK	TRIPLOCK
Z1OTP_CRCLOCK	GPALOCK	SYNCSOCLOCK
Z2OTP_PSWDLOCK	GPBLOCK	EMIF1LOCK
Z2OTP_CRCLOCK	GPCLOCK	EMIF2LOCK
Z1_OTPSECLOCK	GPDLOCK	

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 by software.



EALLOW Protection (1 of 2)

- EALLOW stands for *Emulation Allow*
- Code access to protected registers allowed only when EALLOW = 1 in the ST1 register
- The emulator can always access protected registers
- EALLOW bit controlled by assembly level instructions
  - ‘EALLOW’ sets the bit (register access enabled)
  - ‘EDIS’ clears the bit (register access disabled)
- EALLOW bit cleared upon ISR entry, restored upon exit



EALLOW Protection (2 of 2)

The following registers are protected:

Device 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 datasheet and Technical Reference Manual for detailed listings

EALLOW register access C-code example:

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

Lab 5: System Initialization

➢ Objective

The objective of this lab 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 system initialization for this lab will consist of the following:

- Setup the clock module – PLL, LOSPCP = /4, low-power modes to default values, enable all module clocks
- Disable the watchdog – clear WD flag, disable watchdog, WD prescale = 1
- Setup the watchdog and system control registers – DO NOT clear WD OVERRIDE bit, configure WD to generate a CPU reset
- Setup the shared I/O pins – set all GPIO pins to GPIO function (e.g. a "0" setting for GPIO group multiplexer "GPxGMUX1/2" and a "0" setting for GPIO multiplexer "GPxMUX1/2")

The first part of the lab exercise will setup the system initialization and test the watchdog operation by having the watchdog cause a reset. In the second part of the lab exercise the PIE vectors will be added and tested by using the watchdog to generate an interrupt. This lab will make use of the F2837xD C-code header files to simplify the programming of the device, as well as take care of the register definitions and addresses. Please review these files, and make use of them in the future, as needed.

➢ Procedure

Create a New Project

1. Create a new project (`File` → `New` → `CCS Project`) for this lab exercise. The top section should default to the options previously selected (setting the "Target" to "TMS320F28379D", and leaving the "Connection" box blank). Name the project **Lab5**. *Uncheck* the "Use default location" box. Using the "Browse…" button navigate to: `C:\C28x\Labs\Lab5\cpu01` then click `OK`. Set the "Linker Command File" to <none>, and be sure to set the "Project templetes and examples" to "Empty Project". Then click `Finish`.

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

```
CodeStartBranch.asm                    Lab_5_6_7.cmd
DelayUs.asm                            Main_5.c
F2837xD_GlobalVariableDefs.c           SysCtrl.c
F2837xD_Headers_nonBIOS_cpu1.cmd       Watchdog.c
Gpio.c                                 Xbar.c
```

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

Project Build Options

3. Setup the build options by right-clicking on `Lab5` in the Project Explorer window and select "Properties". We need to setup the include search path to include the peripheral register header files 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}/../../F2837xD_headers/include`

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

   Click `OK` to include each search path.

4. Next, 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 (first icon with green plus sign). Then in the "Enter Value" window type **CPU1**. This name is used in the project to conditionally include the peripheral register header files code specific to CPU1. Click `OK` to include the name. Finally, click `OK` to save and close the Properties window.

Modify Memory Configuration

5. 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 "SARAM" 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.

6. 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.

Setup System Initialization

7. Open and inspect `SysCtrl.c`. Notice that the clock sources, PLL, peripheral clocks, and low-power modes have been initialized.

8. Modify `Watchdog.c` to implement the system initialization as described in the objective for this lab exercise.

9. Open and inspect `Gpio.c`. Notice that the shared I/O pins have been set to the GPIO function. Also, in `Xbar.c` the crossbar switches have been set to their default values. Save your work.

Build and Load

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

11. Click the "`Debug`" button (green bug). A Launching Debug Session window will open. Select only CPU1 to load the program on (i.e. *uncheck* CPU2), and then click `OK`. Then the CCS Debug perspective view should open, the program will load automatically, and you should now be at the start of main().

12. 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_SARAM`

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

Run the Code – Watchdog Reset Disabled

13. 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.

14. 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).

15. 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 – CCS Issued CPU Reset

16. Perform a CCS CPU reset (soft reset) by clicking on the CPU Reset icon (or by selecting `Run → Reset → CPU Reset`). The program counter should now be at the entry point of the boot ROM code at 0x3FF16A. To view the boot ROM code click on `View Disassembly…`

17. Run your code. Where did your code stop? Are the results as expected? If things went as expected, your code should have stopped at the breakpoint. What happened is as follows. The ROM bootloader began execution and 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( ).

Run the Code – Watchdog Reset Enabled (Hardware Reset)

18. Open the Project Explorer window in the CCS Debug perspective view by selecting `View` → `Project Explorer`. Modify the `InitWatchdog()` function to enable the watchdog (WDCR). This will enable the watchdog to function and cause a reset. Save the file.

19. Build the project by clicking `Project` → `Build Project`. Select `Yes` to "Reload the program automatically".

> Alternatively, you add the "`Build`" button to the tool bar in the CCS Debug perspective (if it is not already there) so that it will 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`.

20. 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`.

21. This time we will have the watchdog issue a reset that will toggle the XRSn pin (i.e. perform a hard reset). Now run your code. Where did your code stop? Why did your code stop at an assembly ESTOP0 instruction in the boot ROM at 0x3FE493 and not as we expected at the breakpoint in main( )? Here is what happened. 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, it read the EMU_KEY and EMU_BMODE values from the PIE RAM which was previously set to RAMM0 boot mode. Again, note that these values do not change and are not affected by reset. When the F28x7x devices undergo a hardware reset (e.g. watchdog reset), the boot ROM code clears the RAM memory blocks. As a result, after the bootloader transferred execution to the beginning of our code at address 0x000000 in RAMM0, the memory block was cleared. The processor was then issued an illegal instruction which trapped us back in the boot ROM.

    This only happened because we are executing out of RAM. In a typical application, the Flash memory contains the program and the reset process would run as we would expect. This should explain why we did not see this behavior with the CCS CPU reset (soft reset where the RAM was not cleared). So what is the advantage of clearing memory during a hardware reset? This ensures that after the reset the original program code and data values will be in a known good state to provide a safer operation. It is important to understand that the watchdog did not behave differently depending on which type of reset was issued. It is the reset process that behaved differently from the different type of resets.

22. Since the watchdog reset in the previous step cleared the RAM blocks, we will now need to reload the program for the second part of this lab exercise. Reload the program by selecting:

    `Run` → `Load` → `Reload Program`

Setup PIE Vector for Watchdog Interrupt

The first part of this lab exercise used the watchdog to generate a CPU reset. This was tested using a breakpoint set at the beginning of main(). Next, we are going to use the watchdog to generate an interrupt. This part will demonstrate the interrupt concepts learned in the previous module.

23. Add (copy) the following files to the project from `C:\C28x\Labs\Lab5\source`:

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

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

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

    ```
            InitPieCtrl();
    ```

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

    PIE group #:___________   # within group:__________

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

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

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

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

28. Open and 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.

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

Build and Load

30. 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

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

32. 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 WAKE interrupt ISR.

Terminate Debug Session and Close Project

33. 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.

34. Next, close the project by right-clicking on `Lab5` in the Project Explorer window and select `Close Project`.

End of Exercise

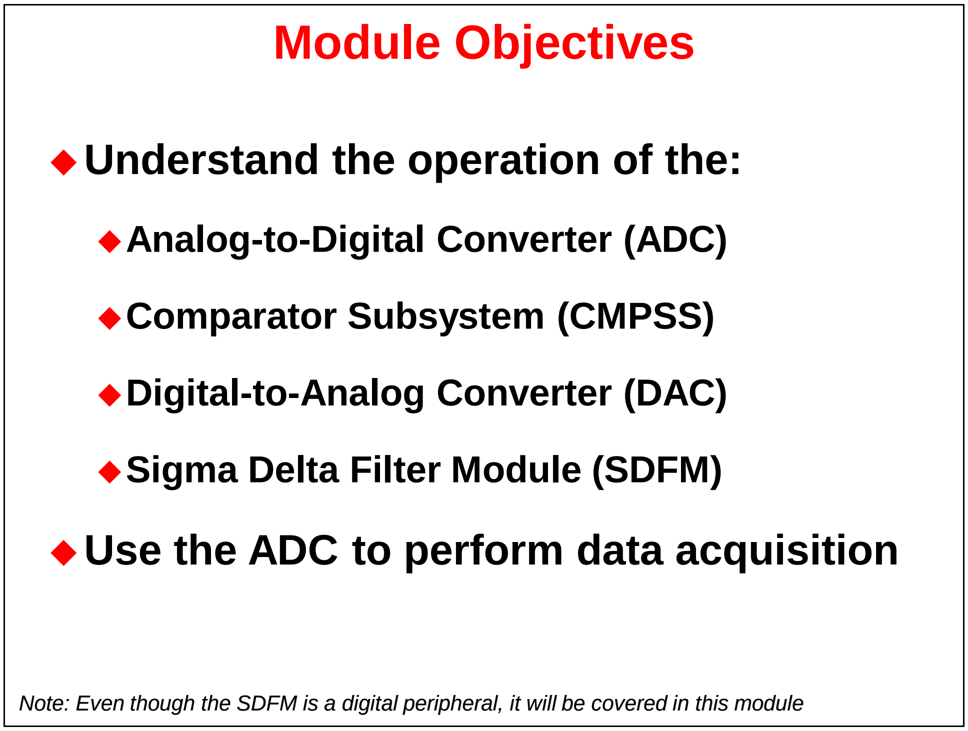
Note: By default, the watchdog timer is enabled out of reset. Code in the file `CodeStartBranch.asm` has been configured to disable the watchdog. This can be important for large C code projects. During this lab exercise, the watchdog was actually re-enabled (or disabled again) in the file `Watchdog.c`.

Analog Subsystem

Introduction

The Analog Subsystem consists of the Analog-to-Digital Converter (ADC), Comparator Subsystem (CMPSS), Digital-to-Analog Converter (DAC), and the Sigma Delta Filter Module (SDFM). This module will explain the operation of each subsystem. Even though the SDFM is a digital peripheral, it will be covered in this module.

Module Objectives



Module Objectives

- ◆ Understand the operation of the:
  - ◆ Analog-to-Digital Converter (ADC)
  - ◆ Comparator Subsystem (CMPSS)
  - ◆ Digital-to-Analog Converter (DAC)
  - ◆ Sigma Delta Filter Module (SDFM)
- ◆ Use the ADC to perform data acquisition

Note: Even though the SDFM is a digital peripheral, it will be covered in this module

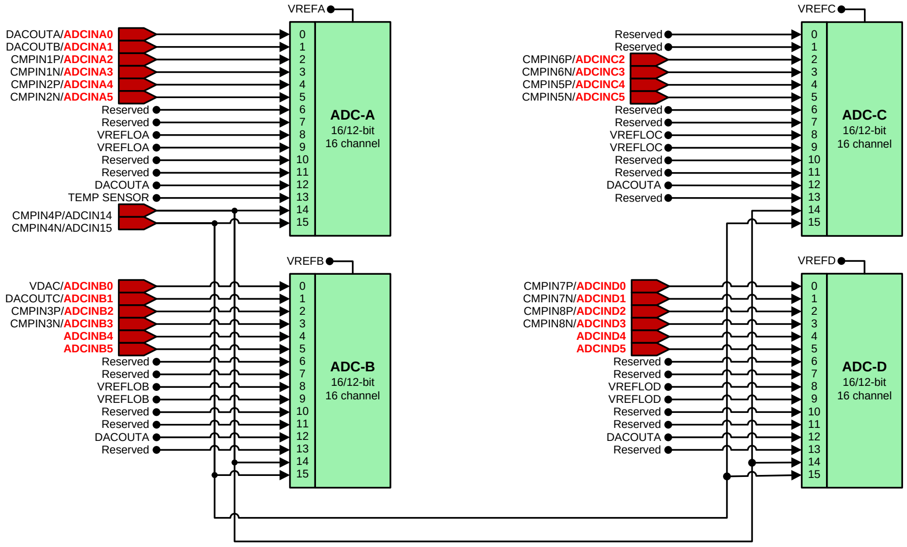
Analog Subsystem:

- Up to Four dual-mode ADCs
  - 16-bit mode
    - 1 MSPS each (up to 4 MSPS system)
    - Differential inputs
    - External reference
  - 12-bit mode
    - 3.5 MSPS each (up to 14 MSPS system)
    - Single-ended
    - External reference
- Up to Eight comparator subsystems
  - Each contains:
    - Two 12-bit reference DACs
    - Two comparators
    - Digital glitch filter
- Three 12-bit buffered DAC outputs
- Sigma-Delta Filter Module (SDFM)

Chapter Topics

Analog-to-Digital Converter (ADC)

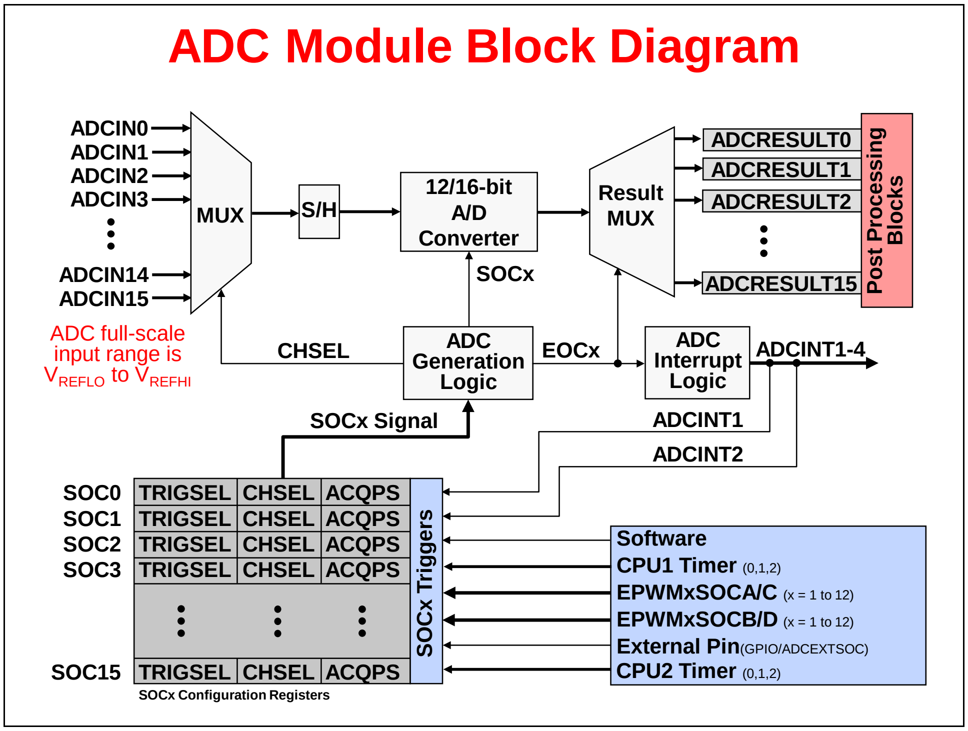
ADC Subsystem



****** Multiple ADC modules allow simultaneous sampling or independent operation ******

The F2837xD includes four independent high-performance ADC modules which can be accessed by both CPU subsystems, allowing the device to efficiently manage multiple analog signals for enhanced overall system throughput. 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 configurable resolution of either 16-bits or 12-bits. For 16-bit resolution, the ADC performs differential signal conversions with a performance of 1.1 MSPS, yielding 4.4 MSPS for the device. In differential signal mode, a pair of pins (positive input ADCINxP and negative input ADCINxN) is sampled and the input applied to the converter is the difference between the two pins (ADCINxP – ADCINxN). A benefit of differential signaling mode is the ability to cancel noise that may be introduced common to both inputs. For 12-bit resolution, the ADC performs single-ended signal conversions with a performance of 3.5 MSPS, yielding 14 MSPS for the device. In single-ended mode, a single pin (ADCINx) is sampled and applied to the input of the converter.

ADC Block and Functional Diagrams



ADC Module Block Diagram

ADC full-scale input range is V_{REFLO} to V_{REFHI}

SOC0	TRIGSEL	CHSEL	ACQPS
SOC1	TRIGSEL	CHSEL	ACQPS
SOC2	TRIGSEL	CHSEL	ACQPS
SOC3	TRIGSEL	CHSEL	ACQPS
	⋮	⋮	⋮
SOC15	TRIGSEL	CHSEL	ACQPS

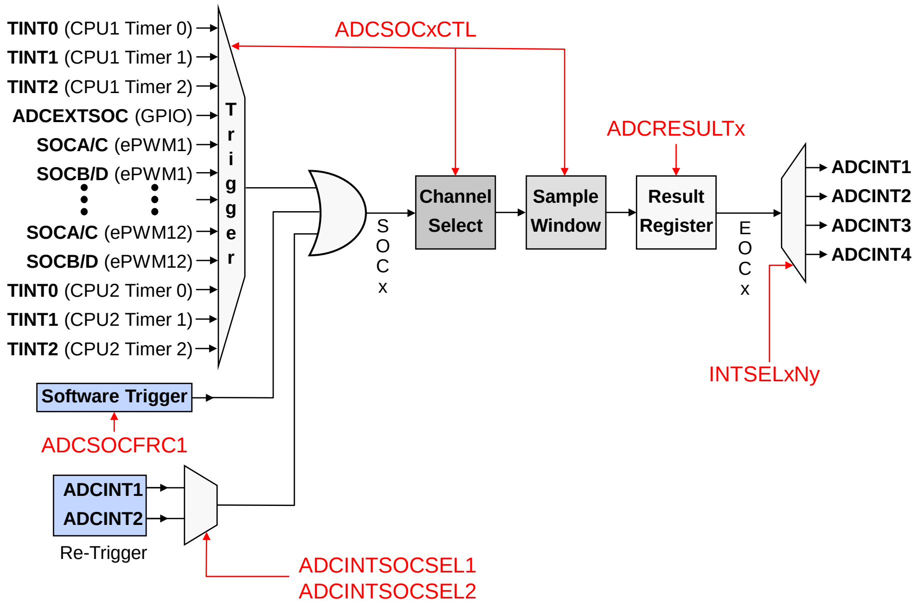
SOCx Configuration Registers

Software
CPU1 Timer (0,1,2)
EPWMxSOCA/C (x = 1 to 12)
EPWMxSOCB/D (x = 1 to 12)
External Pin(GPIO/ADCEXTSOC)
CPU2 Timer (0,1,2)

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 from each CPU subsystem, 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.

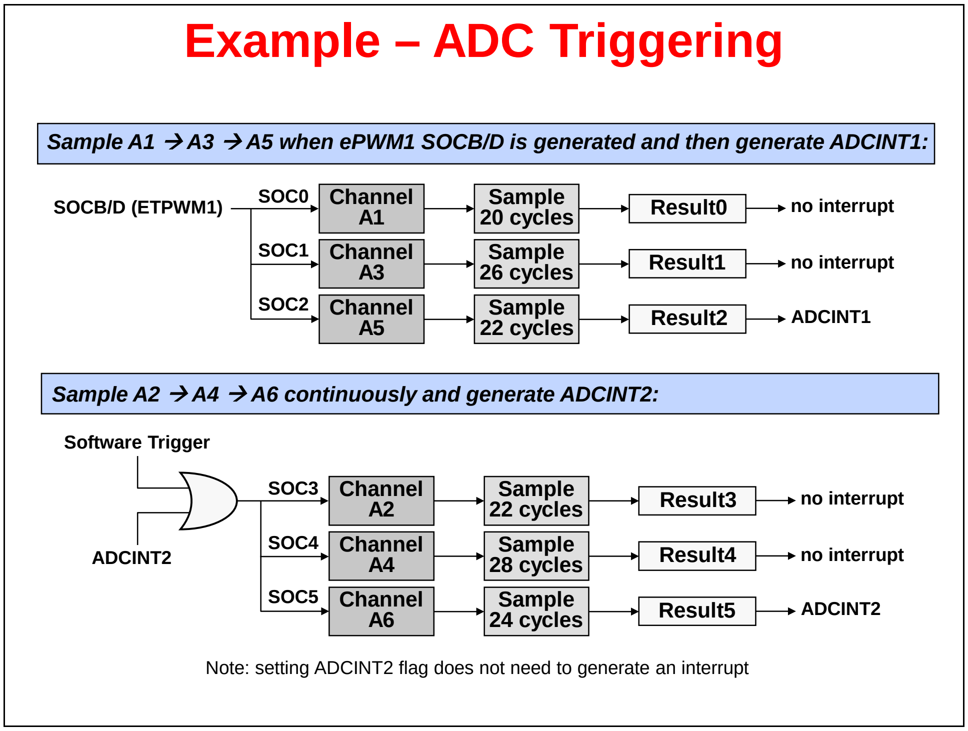
ADC SOCx Functional Diagram



This block diagram is replicated 16 times

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 red text indicates the register names.

ADC Triggering



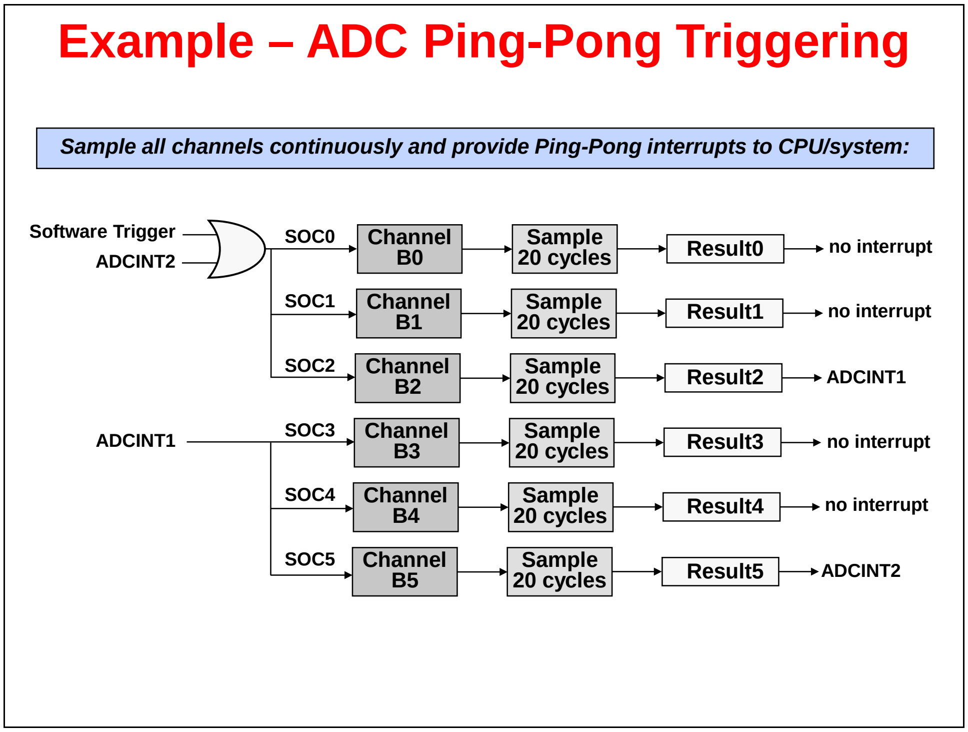
Example – ADC Triggering

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.

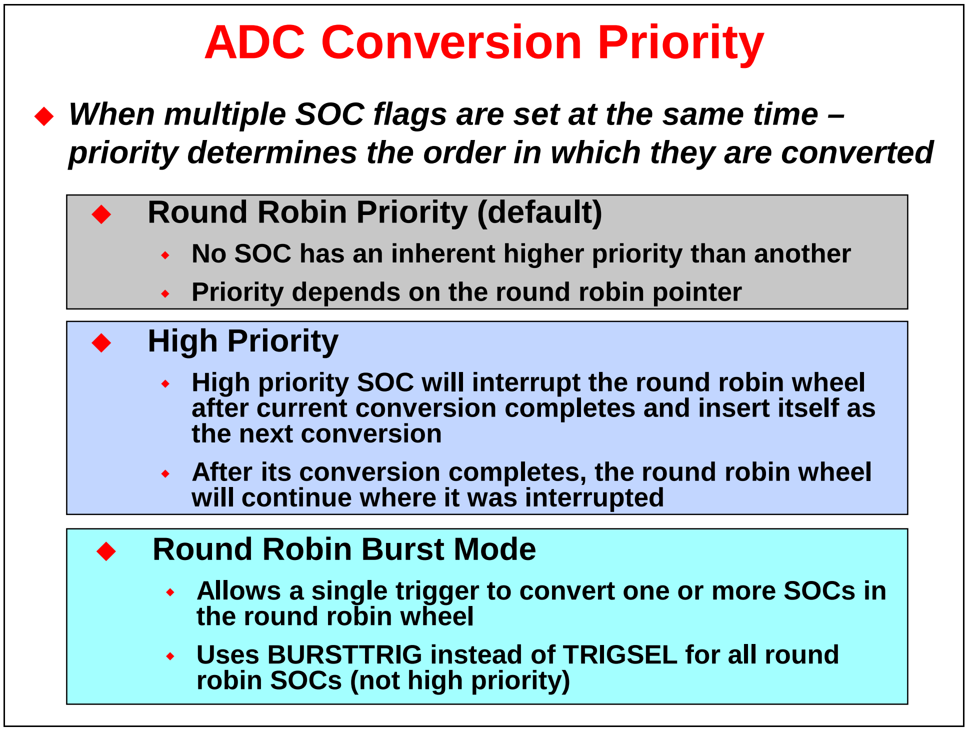


Example – ADC Ping-Pong Triggering

Sample all channels continuously and provide Ping-Pong interrupts to CPU/system:

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

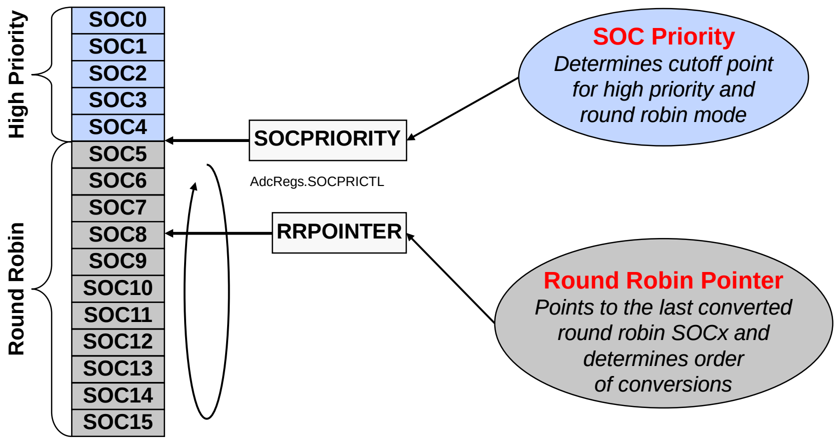


ADC Conversion Priority

- ***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 SOCs in the round robin wheel**
    - **Uses BURSTTRIG instead of TRIGSEL for all round robin SOCs (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



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

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

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

SOCPRIORITY configured as 0;
RRPOINTER configured as 15;
SOC0 is highest RR priority

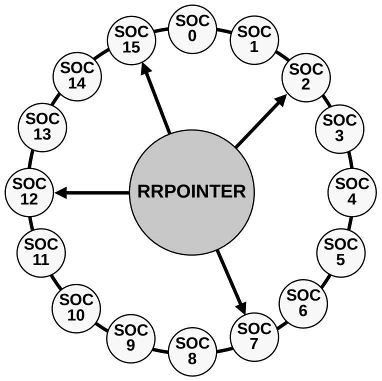
SOC7 trigger received

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

SOCPRIORITY configured as 4;
RRPOINTER configured as 15;
SOC4 is highest RR priority

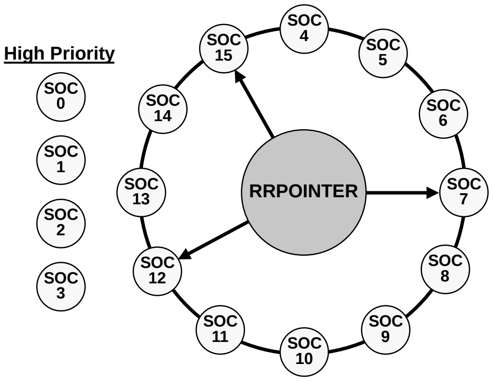
SOC7 trigger received

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

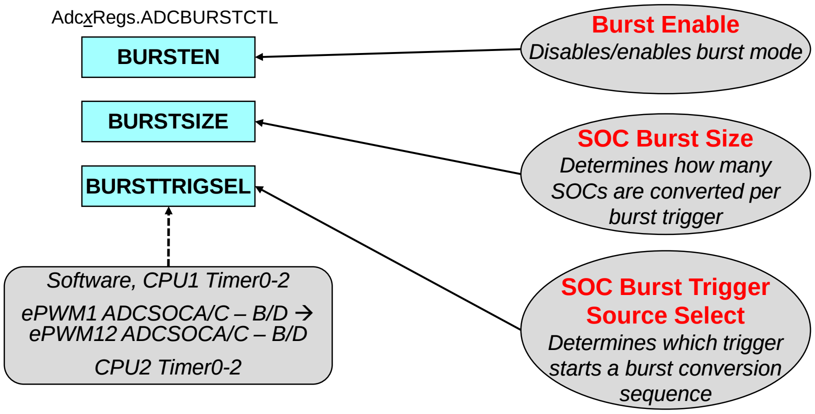
SOC2 & SOC12 triggers received simultaneously

SOC2 is converted;
RRPOINTER stays pointing to SOC7

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



Round Robin Burst Mode Diagram



AdcxRegs.ADCBURSTCTL

BURSTEN

BURSTSIZE

BURSTTRIGSEL

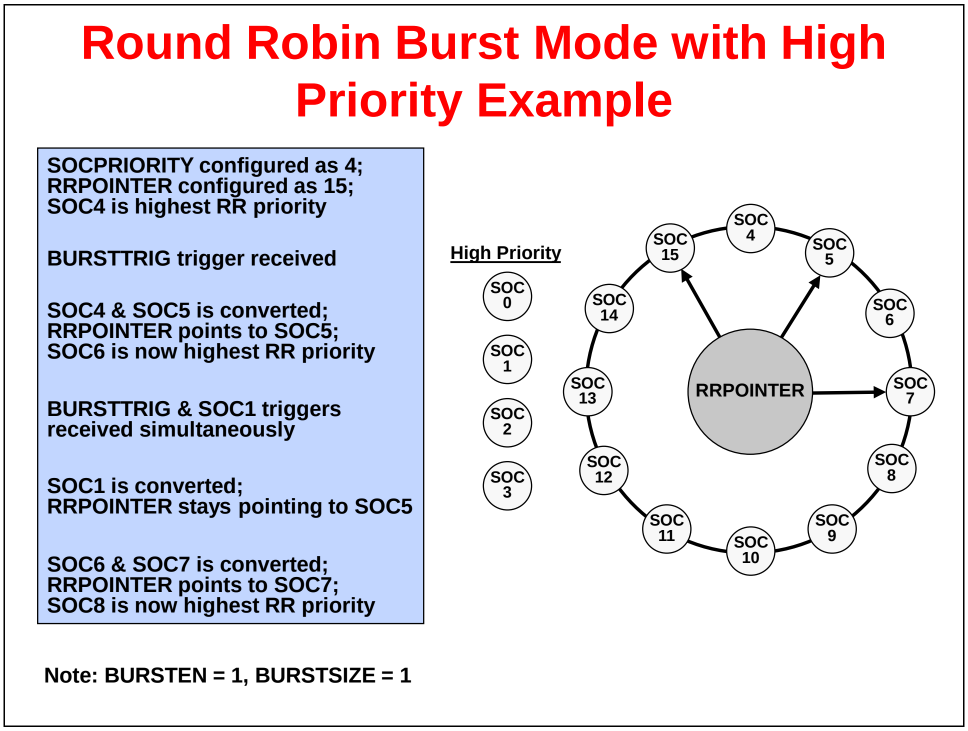
Software, CPU1 Timer0-2
ePWM1 ADCSOCA/C – B/D → ePWM12 ADCSOCA/C – B/D
CPU2 Timer0-2

Burst Enable
Disables/enables burst mode

SOC Burst Size
Determines how many SOCs are converted per burst trigger

SOC Burst Trigger Source Select
Determines which trigger starts a burst conversion sequence

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.



Round Robin Burst Mode with High Priority Example

SOCPRIORITY configured as 4;
RRPOINTER configured as 15;
SOC4 is highest RR priority

BURSTTRIG trigger received

SOC4 & SOC5 is converted;
RRPOINTER points to SOC5;
SOC6 is now highest RR priority

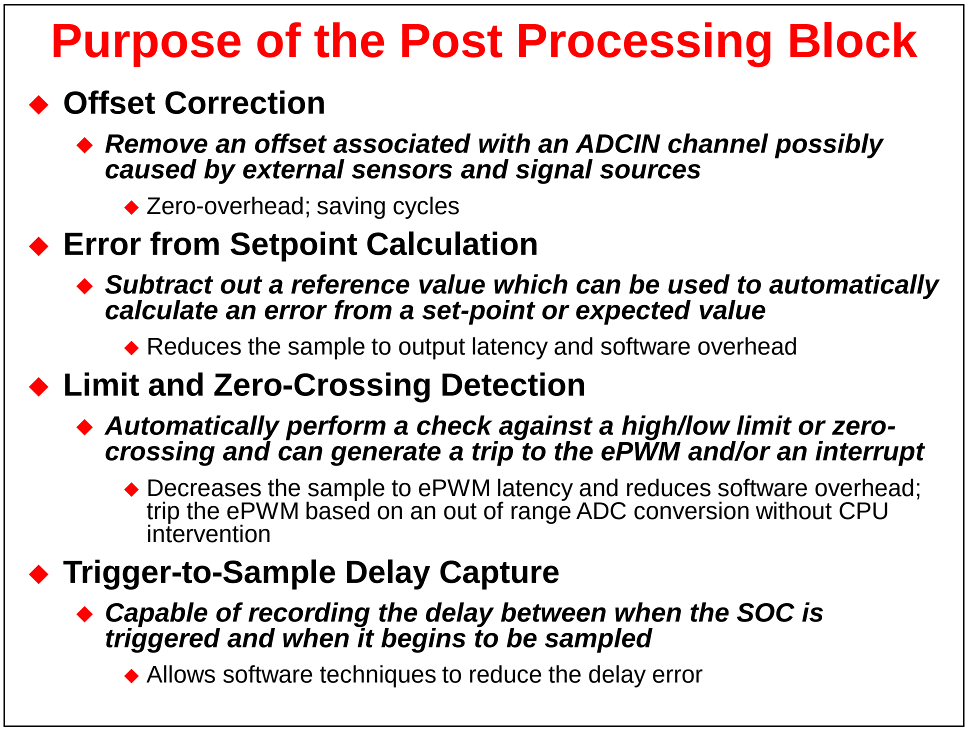
BURSTTRIG & SOC1 triggers received simultaneously

SOC1 is converted;
RRPOINTER stays pointing to SOC5

SOC6 & SOC7 is converted;
RRPOINTER points to SOC7;
SOC8 is now highest RR priority

Note: BURSTEN = 1, BURSTSIZE = 1

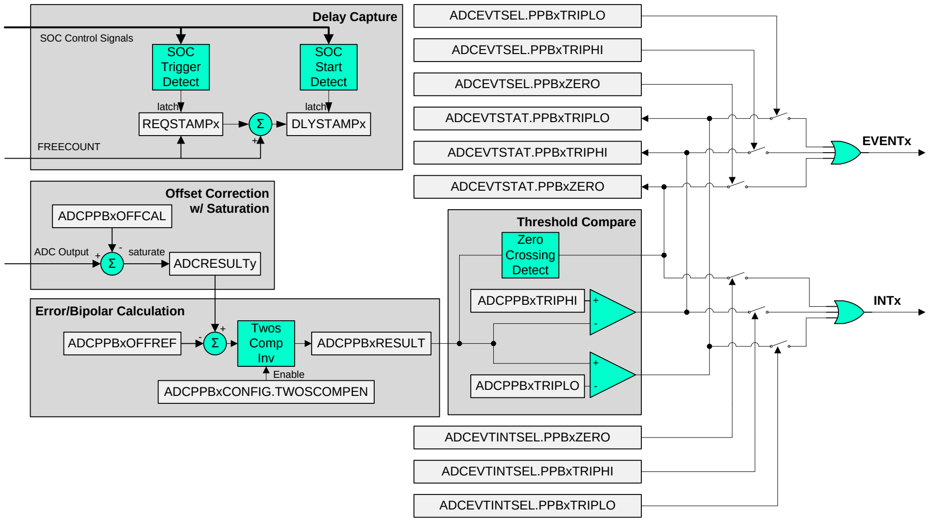
Post Processing Block



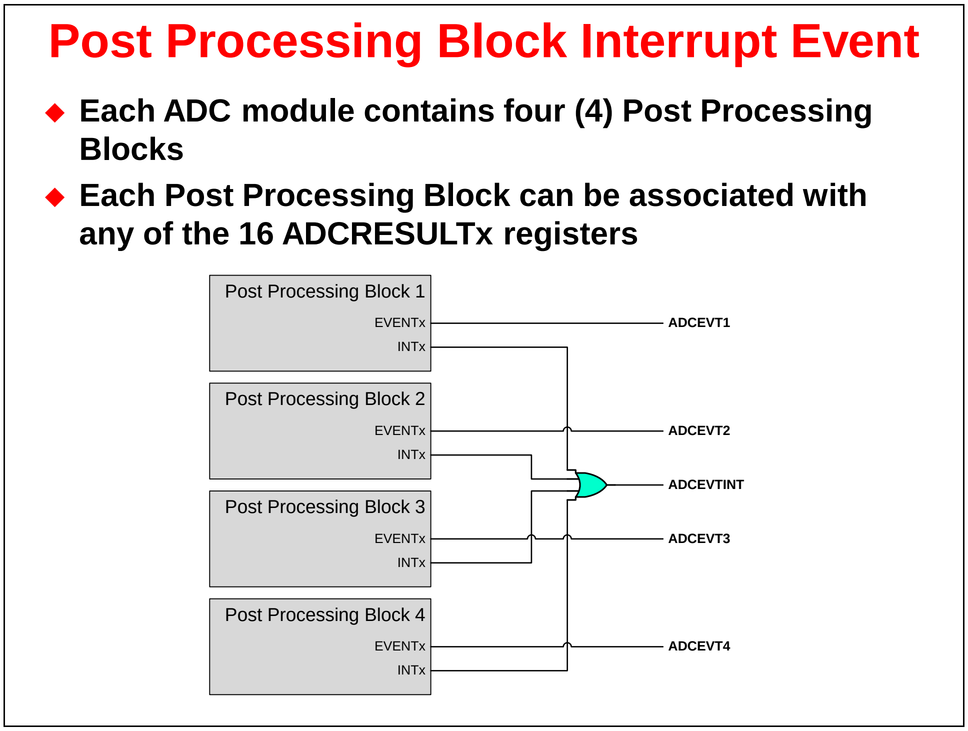
Purpose of the Post Processing Block

- Offset Correction
  - ***Remove an offset associated with an ADCIN channel possibly caused by external sensors and signal sources***
    - Zero-overhead; saving cycles
- Error from Setpoint 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

Post Processing Block - Diagram



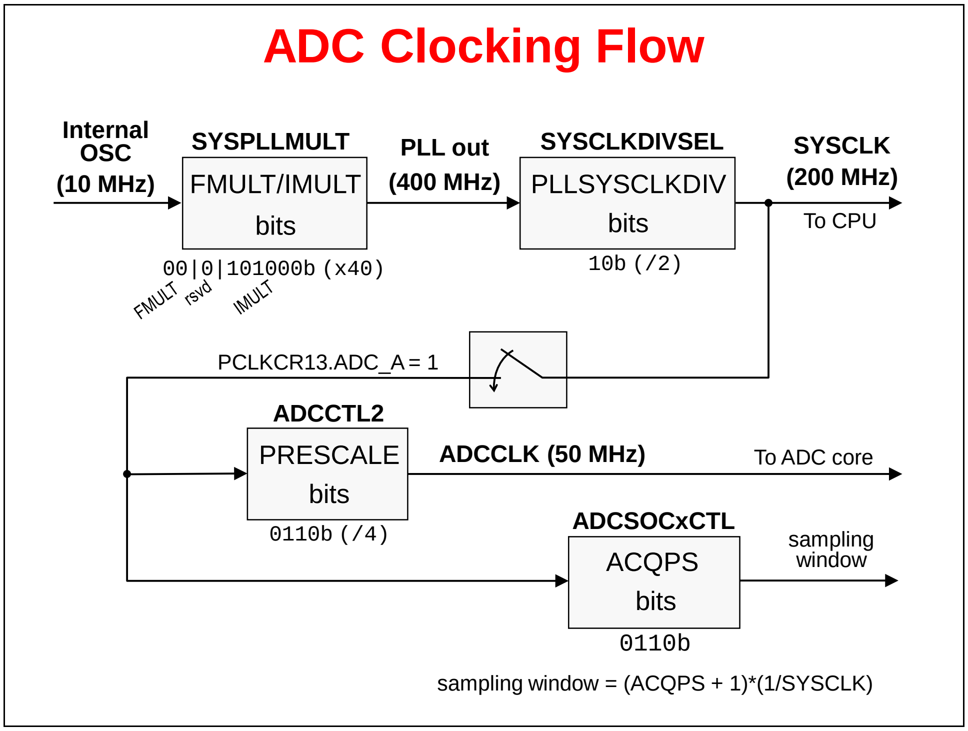
To further enhance the capabilities of the ADC, each ADC module incorporates four post-processing blocks (PPB), and each PPB can be linked to any of the ADC result registers. The PPBs can be used for offset correction, calculating an error from a set-point, detecting a limit and zero-crossing, and capturing 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 (4) Post Processing Blocks
- Each Post Processing Block can be associated with any of the 16 ADCRESULTx registers

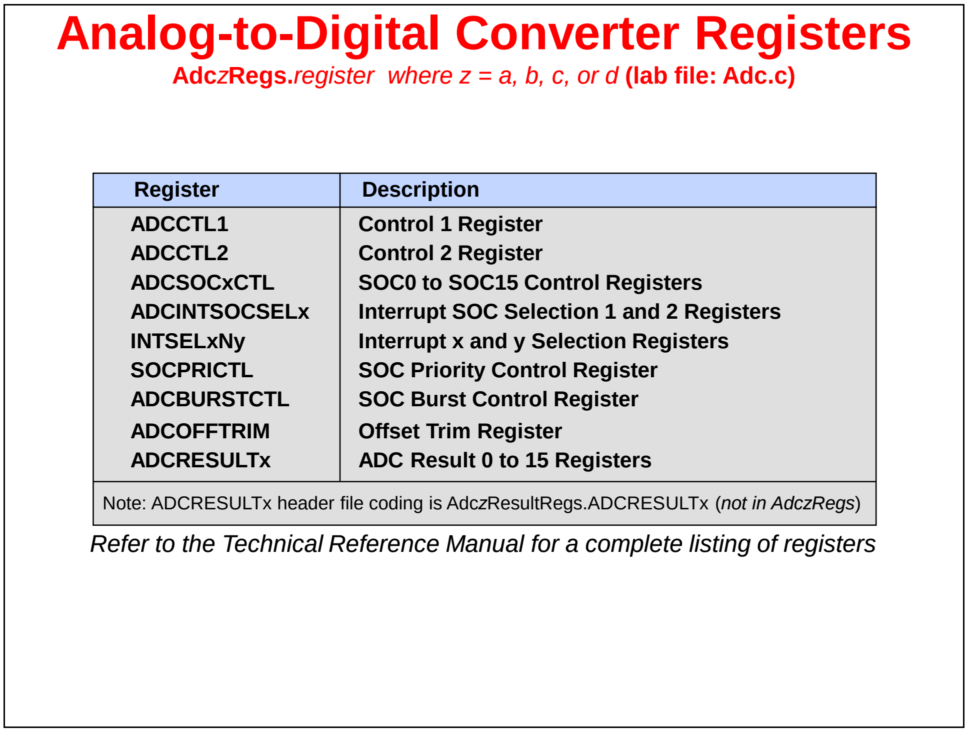
ADC Clocking Flow



ADC Clocking Flow

sampling window = (ACQPS + 1)*(1/SYSCLK)

ADC Registers



Analog-to-Digital Converter Registers

AdczRegs.*register where z = a, b, c, or d* **(lab file: Adc.c)**

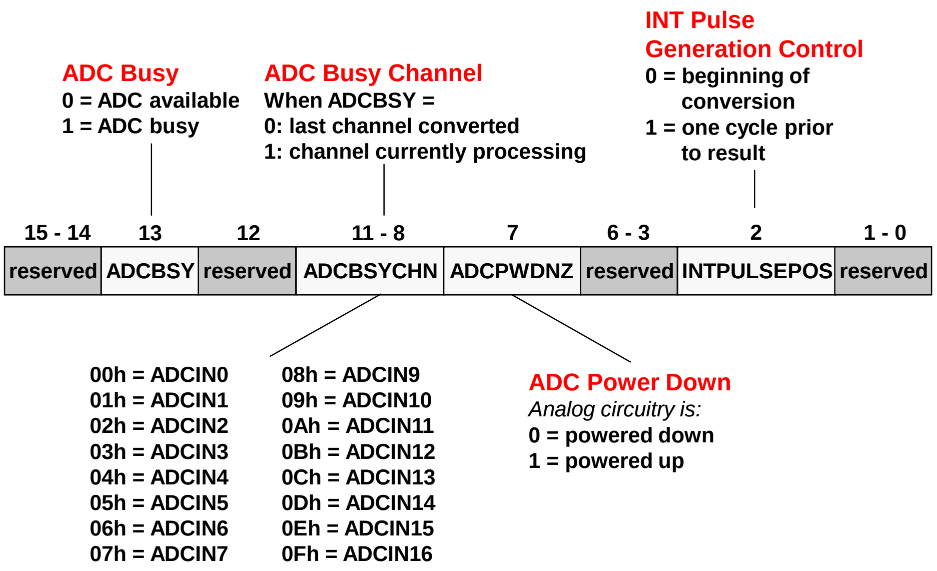
Register	Description
ADCCTL1	Control 1 Register
ADCCTL2	Control 2 Register
ADCSOCxCTL	SOC0 to SOC15 Control Registers
ADCINTSOCSELx	Interrupt SOC Selection 1 and 2 Registers
INTSELxNy	Interrupt x and y Selection Registers
SOCPRICTL	SOC Priority Control Register
ADCBURSTCTL	SOC Burst Control Register
ADCOFFTRIM	Offset Trim Register
ADCRESULTx	ADC Result 0 to 15 Registers

Note: ADCRESULTx header file coding is AdczResultRegs.ADCRESULTx (*not in AdczRegs*)

Refer to the Technical Reference Manual for a complete listing of registers

ADC Control Register 1

AdczRegs.ADCCTL1 *(z = a, b, c, or d)*



15 - 14	13	12	11 - 8	7	6 - 3	2	1 - 0
reserved	ADCBSY	reserved	ADCBSYCHN	ADCPWDNZ	reserved	INTPULSEPOS	reserved

ADC Busy
0 = ADC available
1 = ADC busy

ADC Busy Channel
When ADCBSY =
0: last channel converted
1: channel currently processing

INT Pulse Generation Control
0 = beginning of conversion
1 = one cycle prior to result

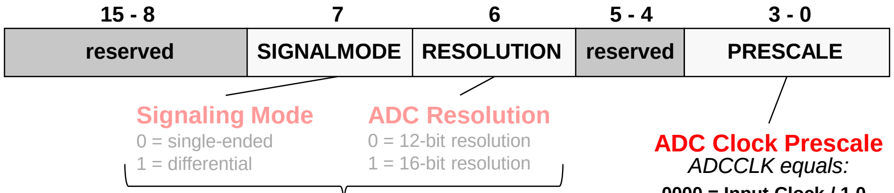
00h = ADCIN0
01h = ADCIN1
02h = ADCIN2
03h = ADCIN3
04h = ADCIN4
05h = ADCIN5
06h = ADCIN6
07h = ADCIN7

08h = ADCIN9
09h = ADCIN10
0Ah = ADCIN11
0Bh = ADCIN12
0Ch = ADCIN13
0Dh = ADCIN14
0Eh = ADCIN15
0Fh = ADCIN16

ADC Power Down
Analog circuitry is:
0 = powered down
1 = powered up

ADC Control Register 2

AdczRegs.ADCCTL2 *(z = a, b, c, or d)*



15 - 8	7	6	5 - 4	3 - 0
reserved	SIGNALMODE	RESOLUTION	reserved	PRESCALE

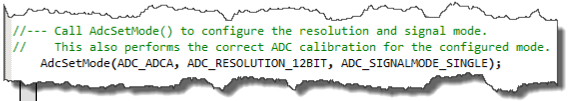
Signaling Mode
0 = single-ended
1 = differential

ADC Resolution
0 = 12-bit resolution
1 = 16-bit resolution

Configured by AdcSetMode() function in source code

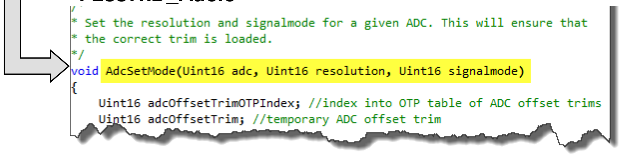
ADC Clock Prescale
ADCCLK equals:
0000 = Input Clock / 1.0
0001 = Invalid
0010 = Input Clock / 2.0
0011 = Input Clock / 2.5
0100 = Input Clock / 3.0
0101 = Input Clock / 3.5
0110 = Input Clock / 4.0
1000 = Input Clock / 4.5
1001 = Input Clock / 5.0
1010 = Input Clock / 5.5
1011 = Input Clock / 6.0
1100 = Input Clock / 6.5
1101 = Input Clock / 7.0
1110 = Input Clock / 7.5
1111 = Input Clock / 8.0

Adc.c



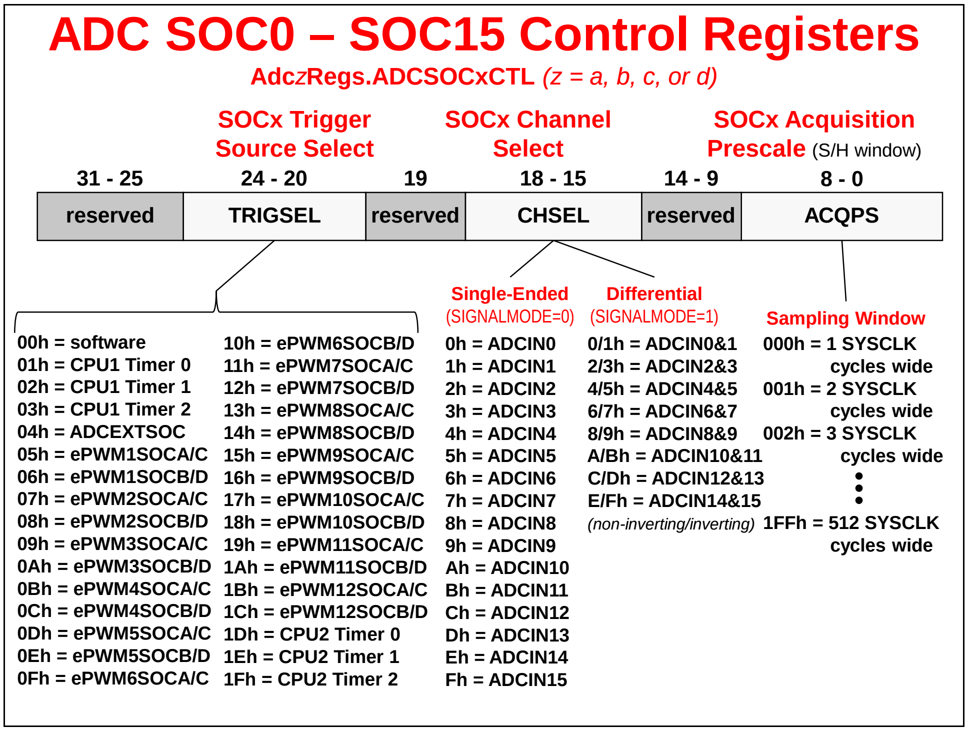
```
//--- Call AdcSetMode() to configure the resolution and signal mode.
//    This also performs the correct ADC calibration for the configured mode.
    AdcSetMode(ADC_ADCA, ADC_RESOLUTION_12BIT, ADC_SIGNALMODE_SINGLE);
```

F2837xD_Adc.c



```
/*
* Set the resolution and signalmode for a given ADC. This will ensure that
* the correct trim is loaded.
*/
void AdcSetMode(Uint16 adc, Uint16 resolution, Uint16 signalmode)
{
    Uint16 adcOffsetTrimOTPIndex; //index into OTP table of ADC offset trims
    Uint16 adcOffsetTrim; //temporary ADC offset trim
```

Definitions for selecting ADC signaling mode and resolution defined in F2837xD_Adc_defines.h



ADC SOC0 – SOC15 Control Registers

AdczRegs.ADCSOCxCTL *(z = a, b, c, or d)*

31 - 25	24 - 20	19	18 - 15	14 - 9	8 - 0
	SOCx Trigger Source Select		SOCx Channel Select		SOCx Acquisition Prescale (S/H window)
reserved	TRIGSEL	reserved	CHSEL	reserved	ACQPS

00h = software
01h = CPU1 Timer 0
02h = CPU1 Timer 1
03h = CPU1 Timer 2
04h = ADCEXTSOC
05h = ePWM1SOCA/C
06h = ePWM1SOCB/D
07h = ePWM2SOCA/C
08h = ePWM2SOCB/D
09h = ePWM3SOCA/C
0Ah = ePWM3SOCB/D
0Bh = ePWM4SOCA/C
0Ch = ePWM4SOCB/D
0Dh = ePWM5SOCA/C
0Eh = ePWM5SOCB/D
0Fh = ePWM6SOCA/C

10h = ePWM6SOCB/D
11h = ePWM7SOCA/C
12h = ePWM7SOCB/D
13h = ePWM8SOCA/C
14h = ePWM8SOCB/D
15h = ePWM9SOCA/C
16h = ePWM9SOCB/D
17h = ePWM10SOCA/C
18h = ePWM10SOCB/D
19h = ePWM11SOCA/C
1Ah = ePWM11SOCB/D
1Bh = ePWM12SOCA/C
1Ch = ePWM12SOCB/D
1Dh = CPU2 Timer 0
1Eh = CPU2 Timer 1
1Fh = CPU2 Timer 2

Single-Ended (SIGNALMODE=0)

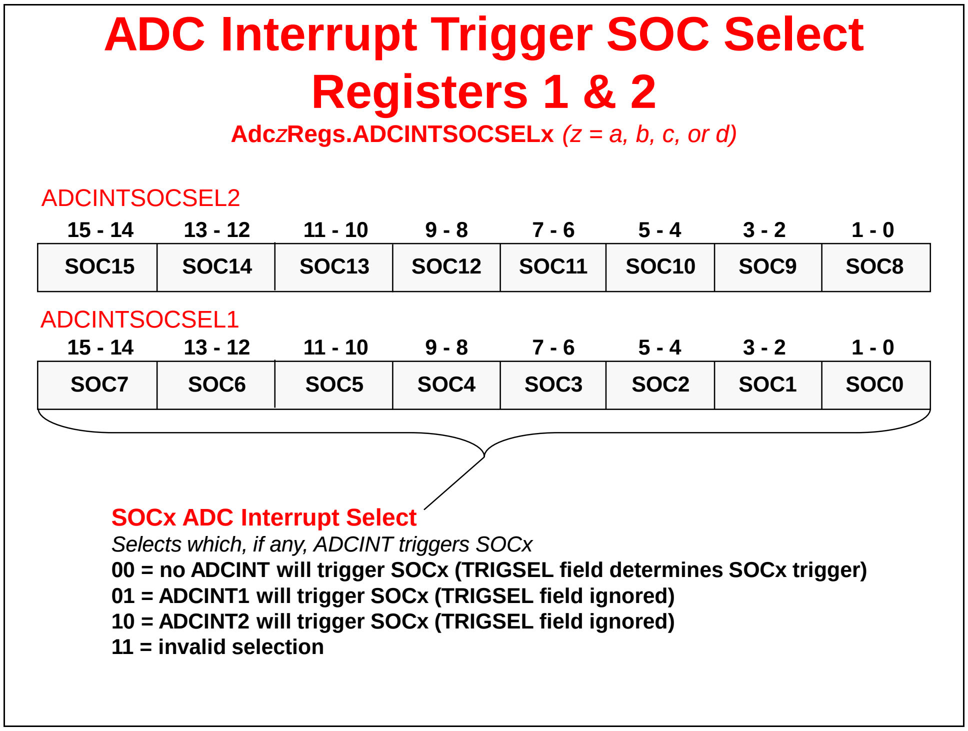
0h = ADCIN0
1h = ADCIN1
2h = ADCIN2
3h = ADCIN3
4h = ADCIN4
5h = ADCIN5
6h = ADCIN6
7h = ADCIN7
8h = ADCIN8
9h = ADCIN9
Ah = ADCIN10
Bh = ADCIN11
Ch = ADCIN12
Dh = ADCIN13
Eh = ADCIN14
Fh = ADCIN15

Differential (SIGNALMODE=1)

0/1h = ADCIN0&1
2/3h = ADCIN2&3
4/5h = ADCIN4&5
6/7h = ADCIN6&7
8/9h = ADCIN8&9
A/Bh = ADCIN10&11
C/Dh = ADCIN12&13
E/Fh = ADCIN14&15
(non-inverting/inverting)

Sampling Window

000h = 1 SYSCLK cycles wide
001h = 2 SYSCLK cycles wide
002h = 3 SYSCLK cycles wide
⋮
1FFh = 512 SYSCLK cycles wide



ADC Interrupt Trigger SOC Select Registers 1 & 2

AdczRegs.ADCINTSOCSELx *(z = a, b, c, or d)*

ADCINTSOCSEL2

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

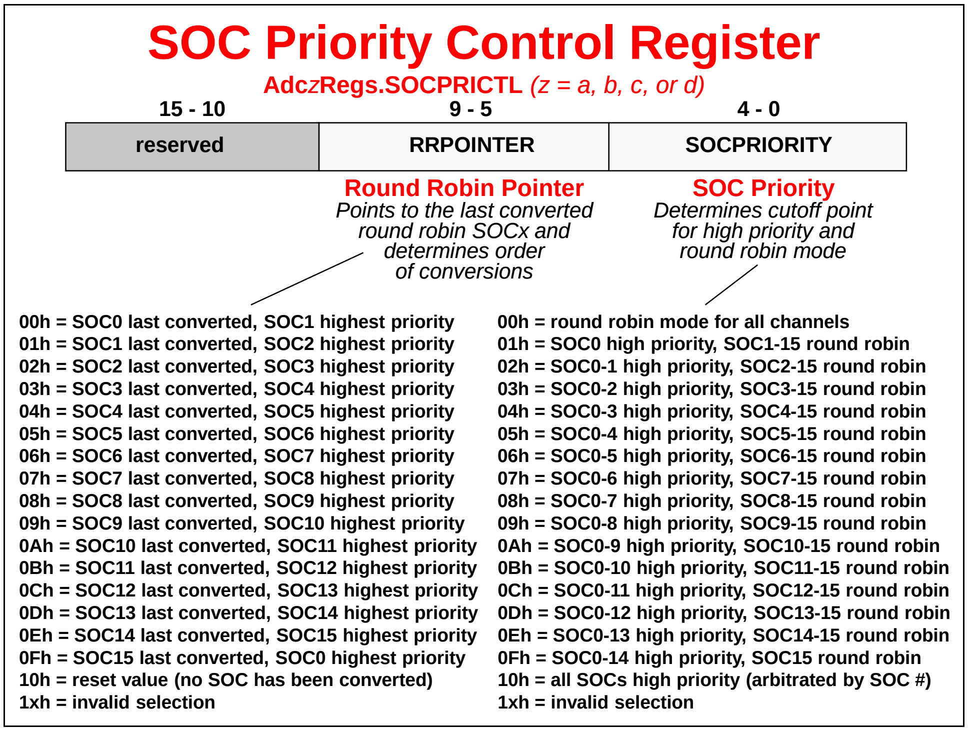
ADCINTSOCSEL1

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

SOCx ADC Interrupt Select

Selects which, if any, ADCINT triggers SOCx

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



SOC Priority Control Register

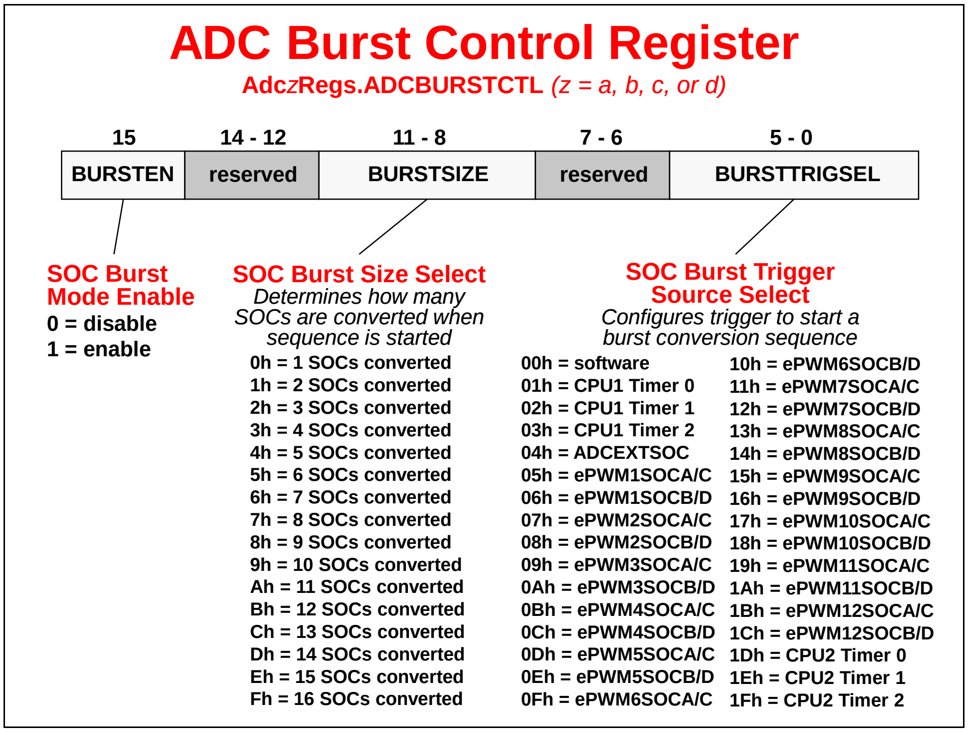
AdczRegs.SOCPRICTL *(z = a, b, c, or d)*

15 - 10	9 - 5	4 - 0
reserved	RRPOINTER	SOCPRIORITY

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

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

RRPOINTER	SOCPRIORITY
00h = SOC0 last converted, SOC1 highest priority	00h = round robin mode for all channels
01h = SOC1 last converted, SOC2 highest priority	01h = SOC0 high priority, SOC1-15 round robin
02h = SOC2 last converted, SOC3 highest priority	02h = SOC0-1 high priority, SOC2-15 round robin
03h = SOC3 last converted, SOC4 highest priority	03h = SOC0-2 high priority, SOC3-15 round robin
04h = SOC4 last converted, SOC5 highest priority	04h = SOC0-3 high priority, SOC4-15 round robin
05h = SOC5 last converted, SOC6 highest priority	05h = SOC0-4 high priority, SOC5-15 round robin
06h = SOC6 last converted, SOC7 highest priority	06h = SOC0-5 high priority, SOC6-15 round robin
07h = SOC7 last converted, SOC8 highest priority	07h = SOC0-6 high priority, SOC7-15 round robin
08h = SOC8 last converted, SOC9 highest priority	08h = SOC0-7 high priority, SOC8-15 round robin
09h = SOC9 last converted, SOC10 highest priority	09h = SOC0-8 high priority, SOC9-15 round robin
0Ah = SOC10 last converted, SOC11 highest priority	0Ah = SOC0-9 high priority, SOC10-15 round robin
0Bh = SOC11 last converted, SOC12 highest priority	0Bh = SOC0-10 high priority, SOC11-15 round robin
0Ch = SOC12 last converted, SOC13 highest priority	0Ch = SOC0-11 high priority, SOC12-15 round robin
0Dh = SOC13 last converted, SOC14 highest priority	0Dh = SOC0-12 high priority, SOC13-15 round robin
0Eh = SOC14 last converted, SOC15 highest priority	0Eh = SOC0-13 high priority, SOC14-15 round robin
0Fh = SOC15 last converted, SOC0 highest priority	0Fh = SOC0-14 high priority, SOC15 round robin
10h = reset value (no SOC has been converted)	10h = all SOCs high priority (arbitrated by SOC #)
1xh = invalid selection	1xh = invalid selection



ADC Burst Control Register

AdczRegs.ADCBURSTCTL *(z = a, b, c, or d)*

15	14 - 12	11 - 8	7 - 6	5 - 0
BURSTEN	reserved	BURSTSIZE	reserved	BURSTTRIGSEL

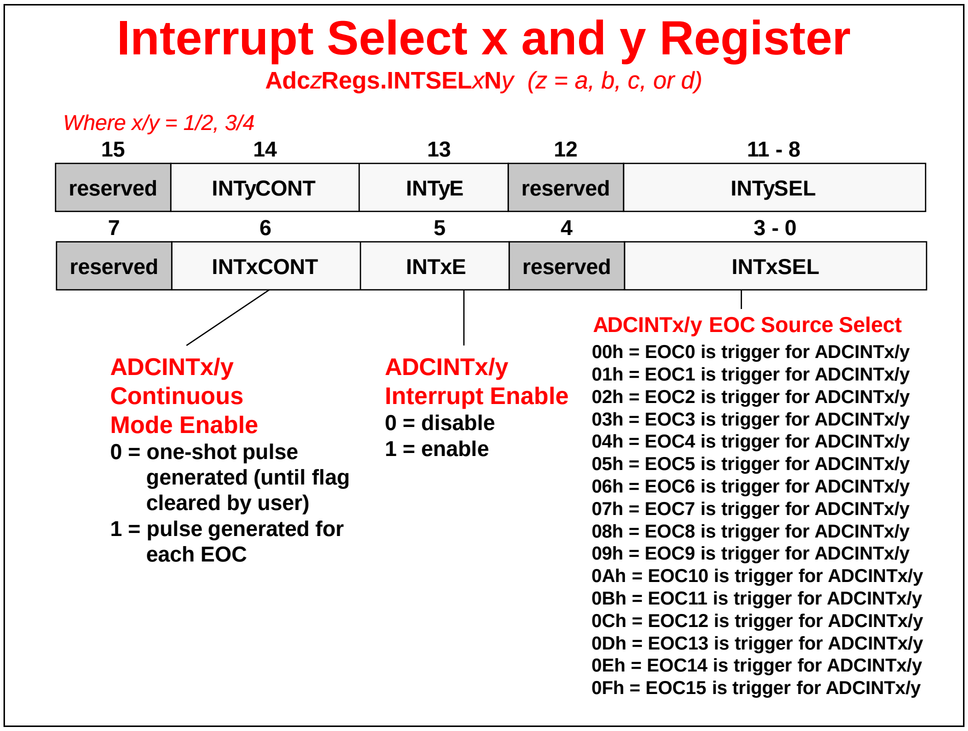
SOC Burst Mode Enable
0 = disable
1 = enable

SOC Burst Size Select
Determines how many SOCs are converted when sequence is started

0h = 1 SOCs converted
1h = 2 SOCs converted
2h = 3 SOCs converted
3h = 4 SOCs converted
4h = 5 SOCs converted
5h = 6 SOCs converted
6h = 7 SOCs converted
7h = 8 SOCs converted
8h = 9 SOCs converted
9h = 10 SOCs converted
Ah = 11 SOCs converted
Bh = 12 SOCs converted
Ch = 13 SOCs converted
Dh = 14 SOCs converted
Eh = 15 SOCs converted
Fh = 16 SOCs converted

SOC Burst Trigger Source Select
Configures trigger to start a burst conversion sequence

00h = software	10h = ePWM6SOCB/D
01h = CPU1 Timer 0	11h = ePWM7SOCA/C
02h = CPU1 Timer 1	12h = ePWM7SOCB/D
03h = CPU1 Timer 2	13h = ePWM8SOCA/C
04h = ADCEXTSOC	14h = ePWM8SOCB/D
05h = ePWM1SOCA/C	15h = ePWM9SOCA/C
06h = ePWM1SOCB/D	16h = ePWM9SOCB/D
07h = ePWM2SOCA/C	17h = ePWM10SOCA/C
08h = ePWM2SOCB/D	18h = ePWM10SOCB/D
09h = ePWM3SOCA/C	19h = ePWM11SOCA/C
0Ah = ePWM3SOCB/D	1Ah = ePWM11SOCB/D
0Bh = ePWM4SOCA/C	1Bh = ePWM12SOCA/C
0Ch = ePWM4SOCB/D	1Ch = ePWM12SOCB/D
0Dh = ePWM5SOCA/C	1Dh = CPU2 Timer 0
0Eh = ePWM5SOCB/D	1Eh = CPU2 Timer 1
0Fh = ePWM6SOCA/C	1Fh = CPU2 Timer 2



Interrupt Select x and y Register

Adc*z*Regs.INTSEL*x*N*y* *(z = a, b, c, or d)*

Where x/y = 1/2, 3/4

15	14	13	12	11 - 8
reserved	INTyCONT	INTyE	reserved	INTySEL

7	6	5	4	3 - 0
reserved	INTxCONT	INTxE	reserved	INTxSEL

ADCINTx/y Continuous Mode Enable

0 = one-shot pulse generated (until flag cleared by user)

1 = pulse generated for each EOC

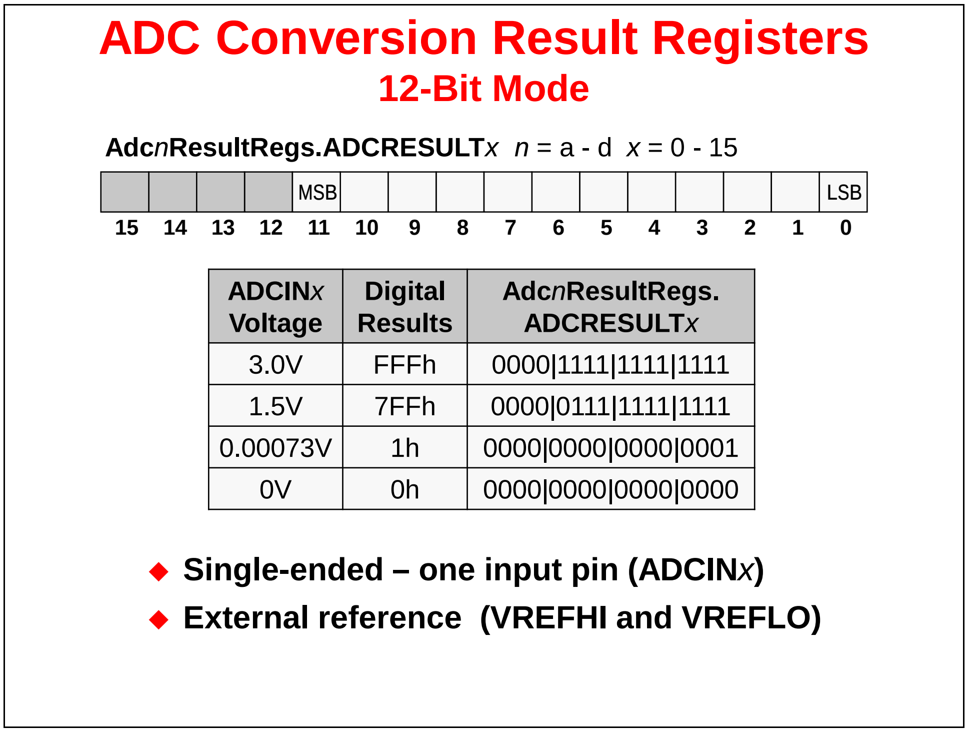
ADCINTx/y Interrupt Enable

0 = disable

1 = enable

ADCINTx/y EOC Source Select

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



ADC Conversion Result Registers

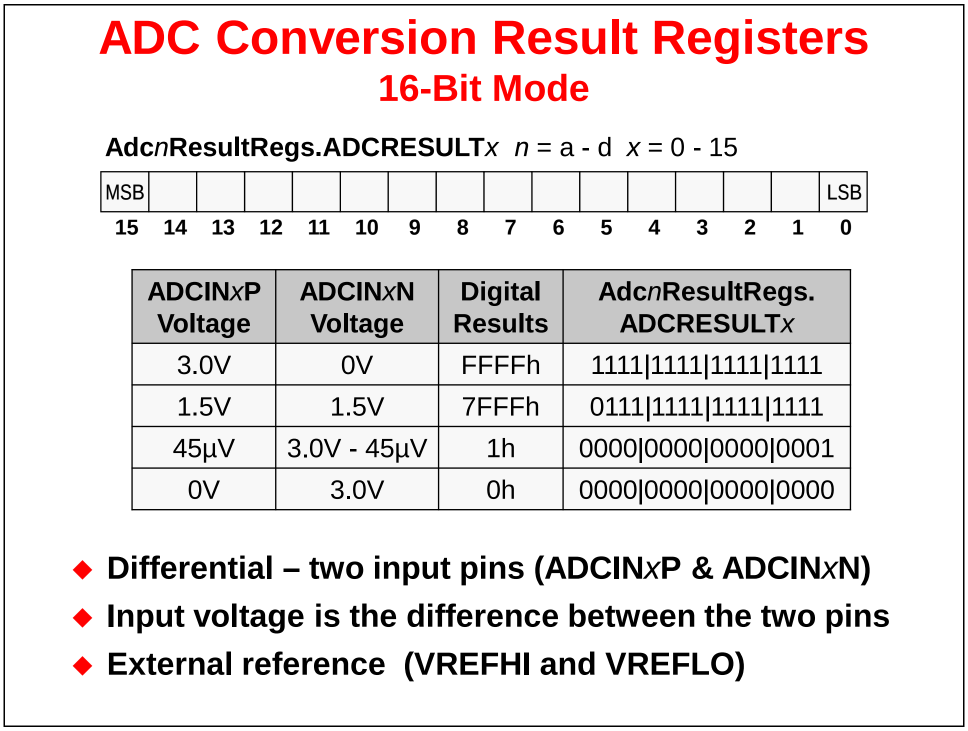
12-Bit Mode

Adc*n*ResultRegs.ADCRESULT*x* *n* = a - d *x* = 0 - 15

				MSB											LSB
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ADCIN*x* Voltage	Digital Results	Adc*n*ResultRegs. ADCRESULT*x*
3.0V	FFFh	0000|1111|1111|1111
1.5V	7FFh	0000|0111|1111|1111
0.00073V	1h	0000|0000|0000|0001
0V	0h	0000|0000|0000|0000

- **Single-ended – one input pin (ADCIN*x*)**
- **External reference (VREFHI and VREFLO)**



ADC Conversion Result Registers

16-Bit Mode

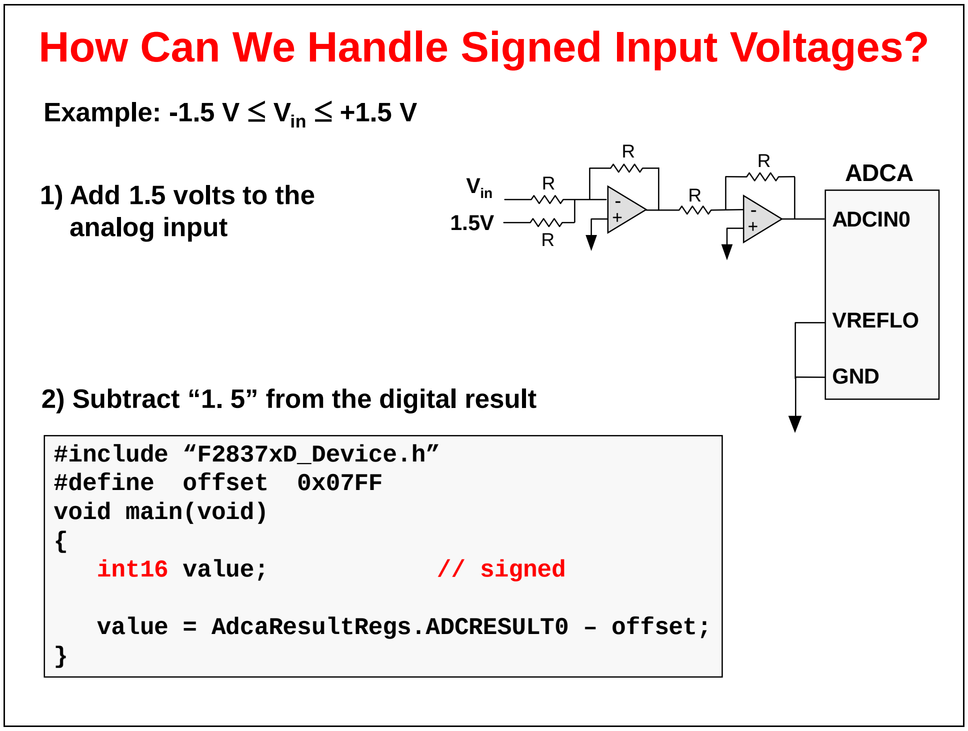
Adc*n*ResultRegs.ADCRESULT*x* *n* = a - d *x* = 0 - 15

MSB															LSB
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ADCIN*x*P Voltage	ADCIN*x*N Voltage	Digital Results	Adc*n*ResultRegs. ADCRESULT*x*
3.0V	0V	FFFFh	1111|1111|1111|1111
1.5V	1.5V	7FFFh	0111|1111|1111|1111
45µV	3.0V - 45µV	1h	0000|0000|0000|0001
0V	3.0V	0h	0000|0000|0000|0000

- **Differential – two input pins (ADCIN*x*P & ADCIN*x*N)**
- **Input voltage is the difference between the two pins**
- **External reference (VREFHI and VREFLO)**

Signed Input Voltages



How Can We Handle Signed Input Voltages?

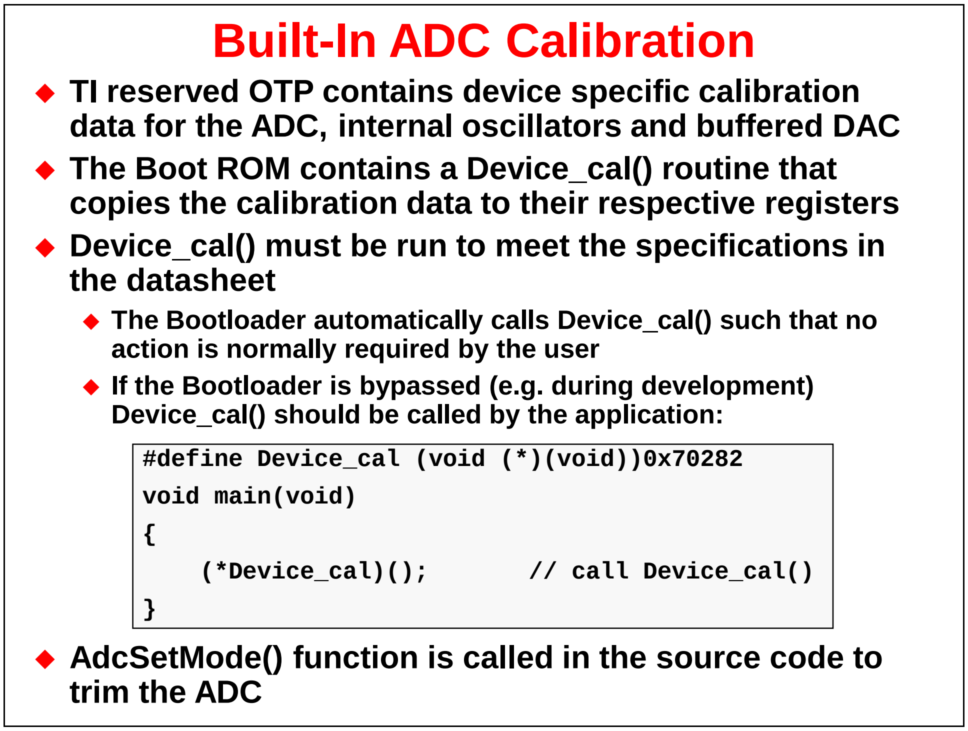
Example: -1.5 V ≤ V_{in} ≤ +1.5 V

1) Add 1.5 volts to the analog input

2) Subtract "1. 5" from the digital result

```
#include "F2837xD_Device.h"
#define  offset  0x07FF
void main(void)
{
   int16 value;            // signed

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

ADC Calibration and Reference

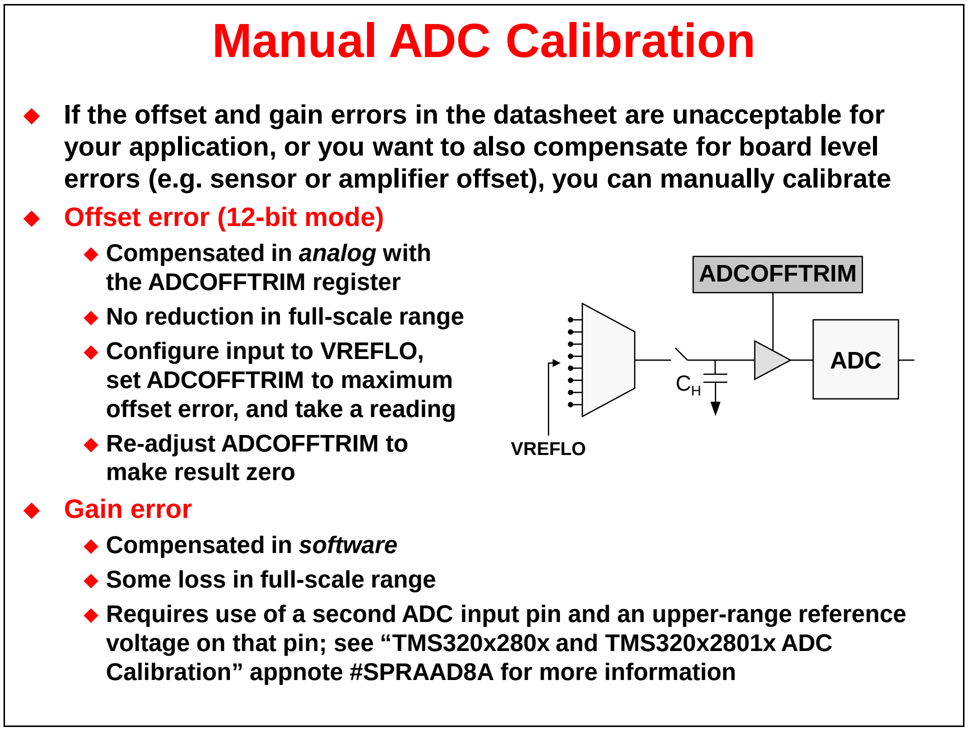


Built-In ADC Calibration

- 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))0x70282
void main(void)
{
    (*Device_cal)();       // call Device_cal()
}
```

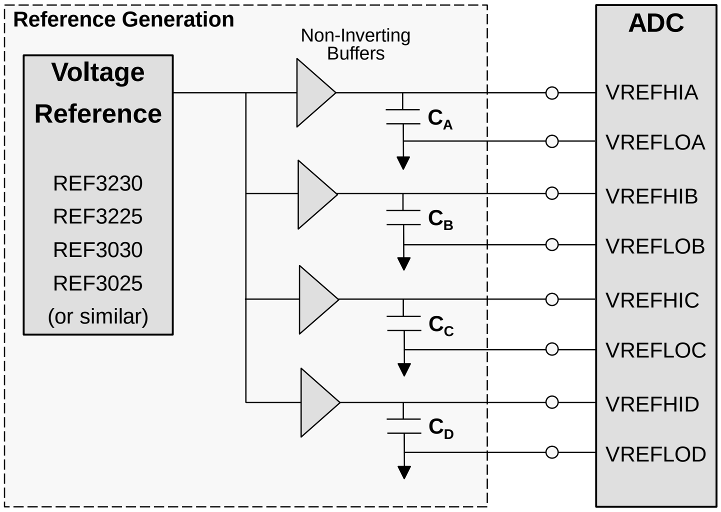
- AdcSetMode() function is called in the source code to trim the ADC



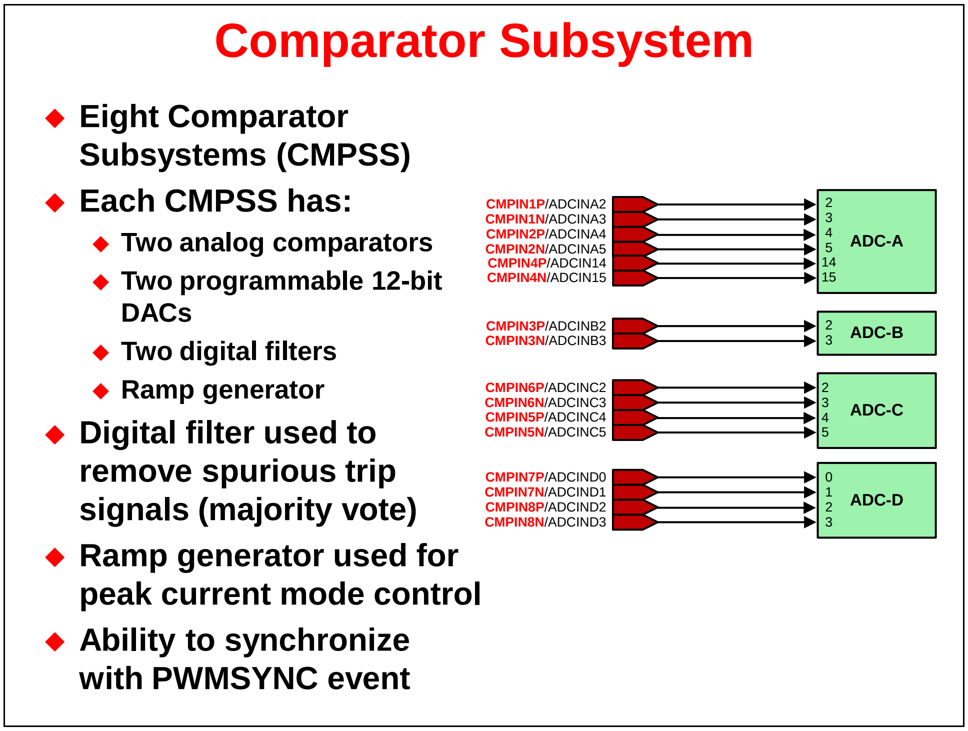
Manual ADC Calibration

- If the offset and gain errors in the datasheet are unacceptable for your application, or you want to also compensate for board level errors (e.g. sensor or amplifier offset), you can manually calibrate
- Offset error (12-bit mode)
  - Compensated in *analog* with the ADCOFFTRIM register
  - No reduction in full-scale range
  - Configure input to VREFLO, set ADCOFFTRIM to maximum offset error, and take a reading
  - Re-adjust ADCOFFTRIM to make result zero
- Gain error
  - Compensated in *software*
  - Some loss in full-scale range
  - Requires use of a second ADC input pin and an upper-range reference voltage on that pin; see "TMS320x280x and TMS320x2801x ADC Calibration" appnote #SPRAAD8A for more information

Analog Subsystem External Reference



Comparator Subsystem (CMPSS)



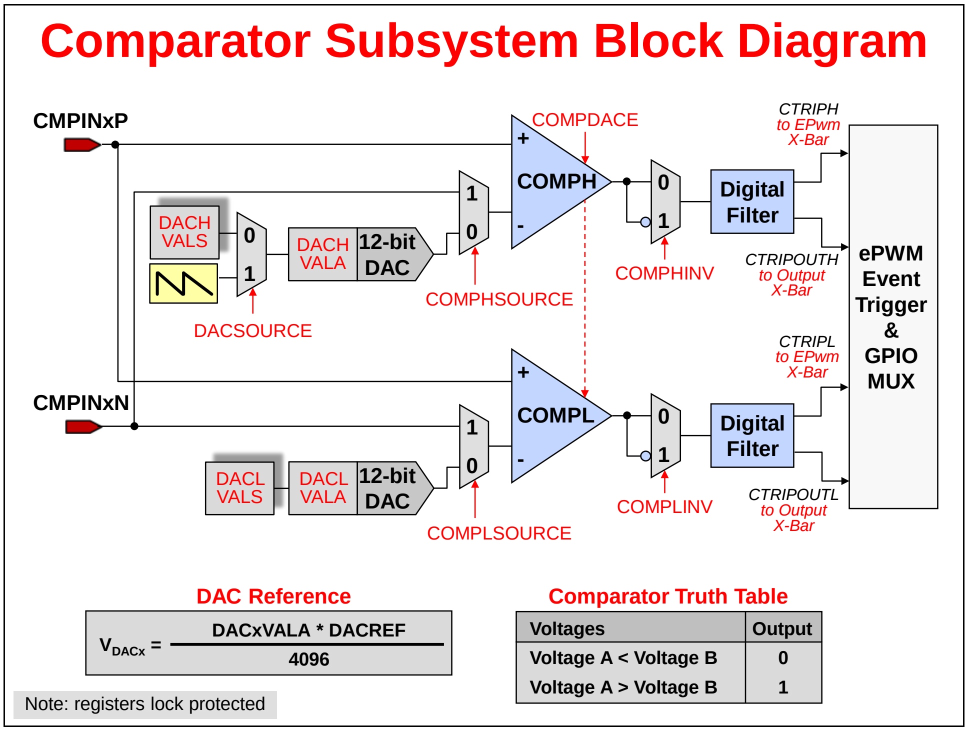
Comparator Subsystem

- Eight Comparator Subsystems (CMPSS)
- Each CMPSS has:
  - Two analog comparators
  - Two programmable 12-bit DACs
  - Two digital filters
  - Ramp generator
- Digital filter used to remove spurious trip signals (majority vote)
- Ramp generator used for peak current mode control
- Ability to synchronize with PWMSYNC event

CMPIN1P/ADCINA2
CMPIN1N/ADCINA3
CMPIN2P/ADCINA4
CMPIN2N/ADCINA5
CMPIN4P/ADCIN14
CMPIN4N/ADCIN15
2
3
4
5
14
15
ADC-A
CMPIN3P/ADCINB2
CMPIN3N/ADCINB3
2
3
ADC-B
CMPIN6P/ADCINC2
CMPIN6N/ADCINC3
CMPIN5P/ADCINC4
CMPIN5N/ADCINC5
2
3
4
5
ADC-C
CMPIN7P/ADCIND0
CMPIN7N/ADCIND1
CMPIN8P/ADCIND2
CMPIN8N/ADCIND3
0
1
2
3
ADC-D

The F2837xD includes eight 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



Comparator Subsystem Block Diagram

DAC Reference

$$V_{DACx} = \frac{\text{DACxVALA * DACREF}}{4096}$$

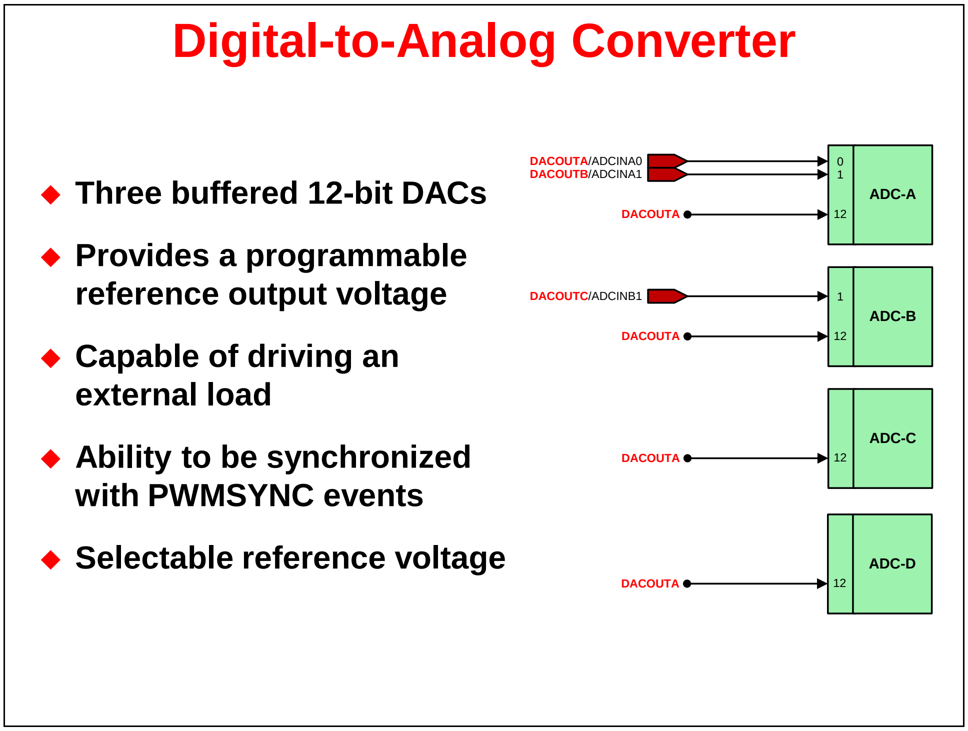
Comparator Truth Table

Voltages	Output
Voltage A < Voltage B	0
Voltage A > Voltage B	1

Note: registers lock protected

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 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 feed through a programmable digital filter that can remove spurious trip signals. The output of the CMPSS generates trip signals to the ePWM event trigger submodule and GPIO structure.

Digital-to-Analog Converter (DAC)

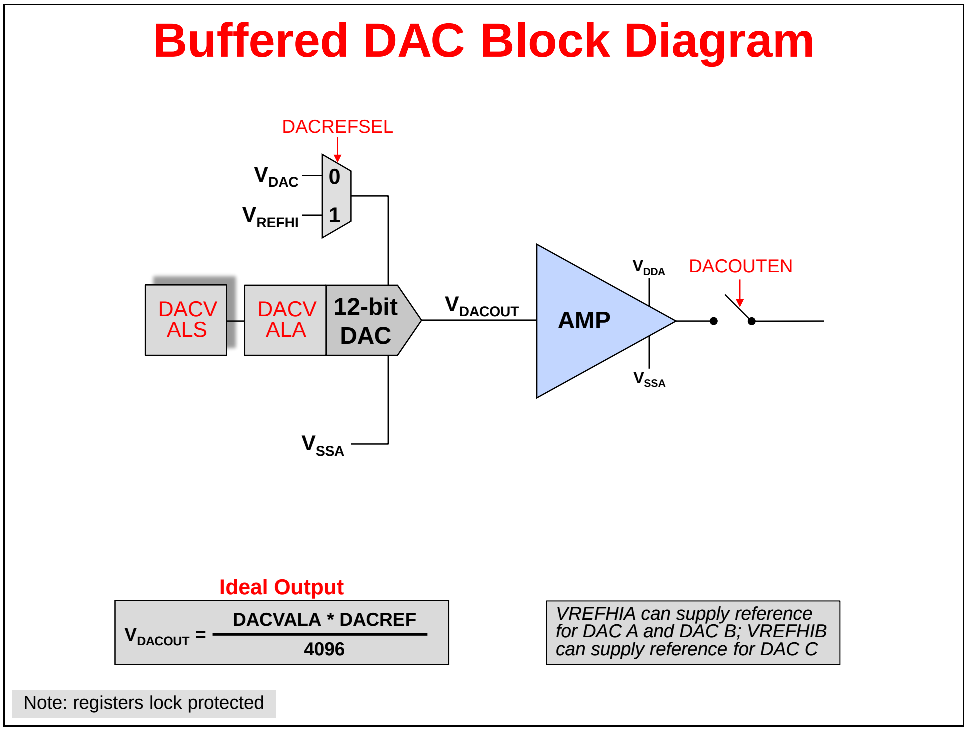


Digital-to-Analog Converter

- Three buffered 12-bit DACs
- Provides a programmable reference output voltage
- Capable of driving an external load
- Ability to be synchronized with PWMSYNC events
- Selectable reference voltage

The F2837xD includes three buffered 12-bit DAC modules that can provide a programmable reference output voltage 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



Buffered DAC Block Diagram

Ideal Output

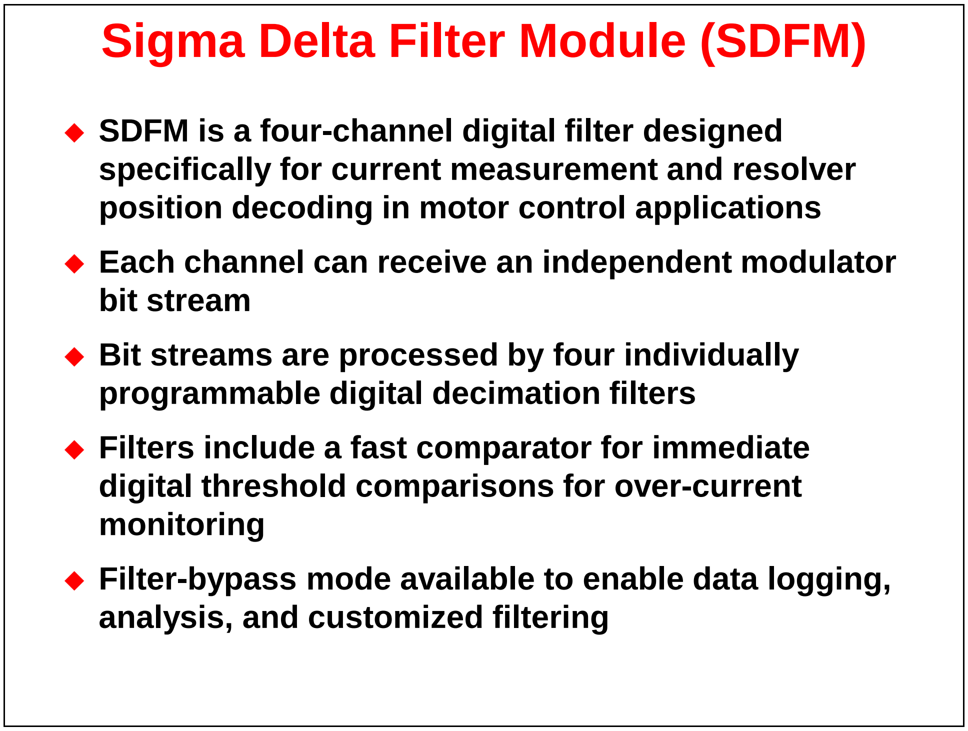
$$V_{DACOUT} = \frac{\textbf{DACVALA * DACREF}}{\textbf{4096}}$$

VREFHIA can supply reference for DAC A and DAC B; VREFHIB can supply reference for DAC C

Note: registers lock protected

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 below.

Sigma Delta Filter Module (SDFM)

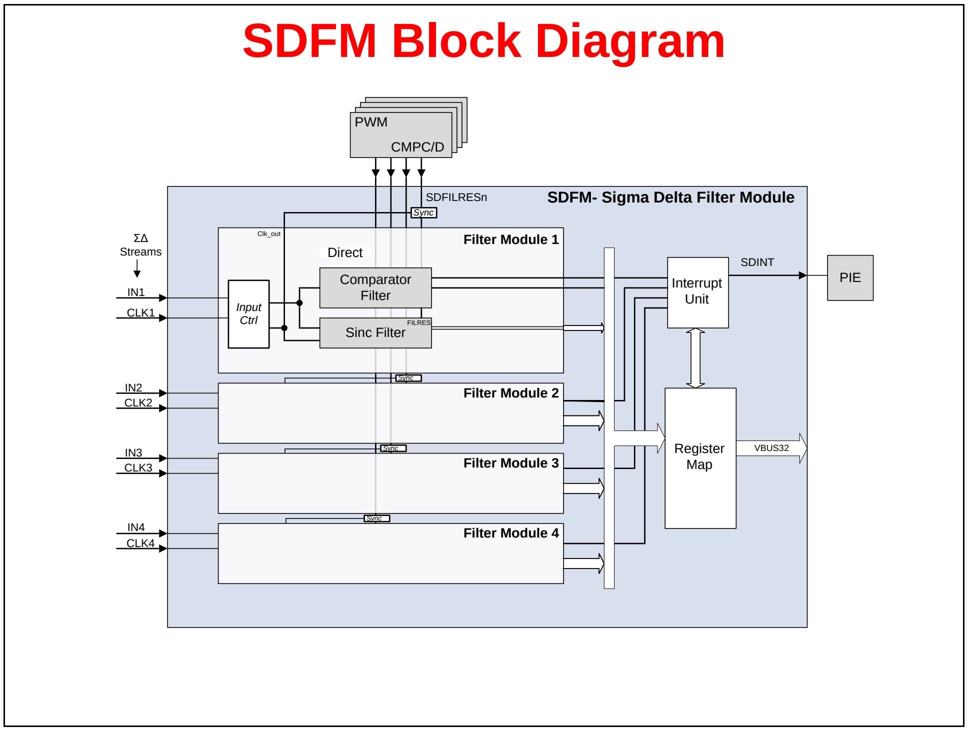


Sigma Delta Filter Module (SDFM)

- **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 modulator bit stream**
- **Bit streams are processed by four individually programmable digital decimation filters**
- **Filters include a fast comparator for immediate digital threshold comparisons for over-current monitoring**
- **Filter-bypass mode available to enable data logging, analysis, and customized filtering**

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.

SDFM Block Diagram

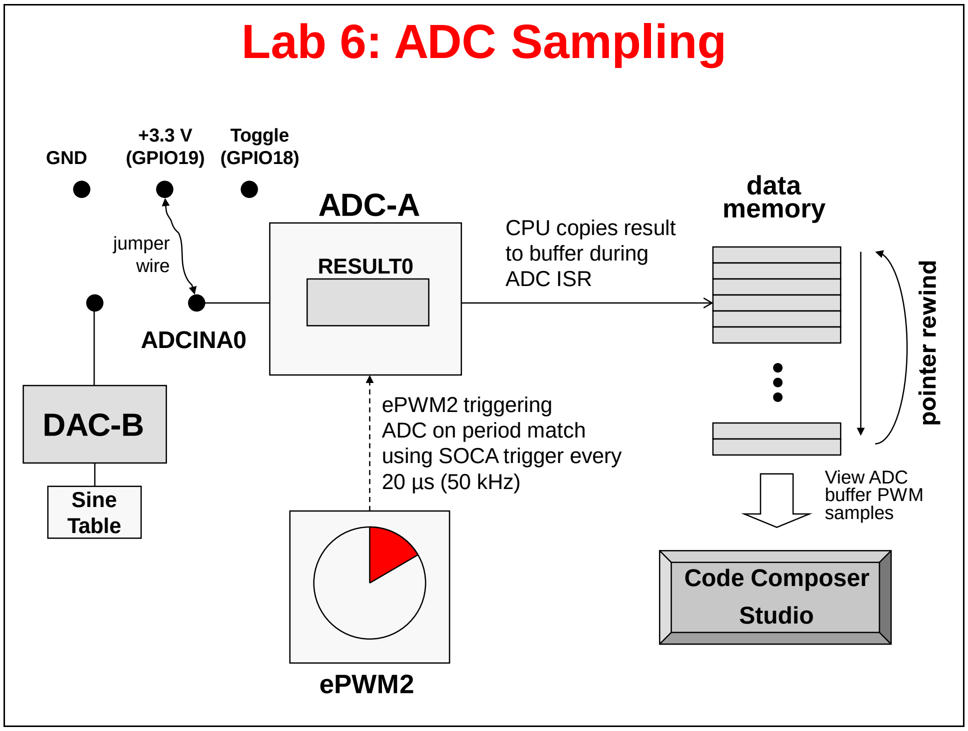


SDFM Block Diagram

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.

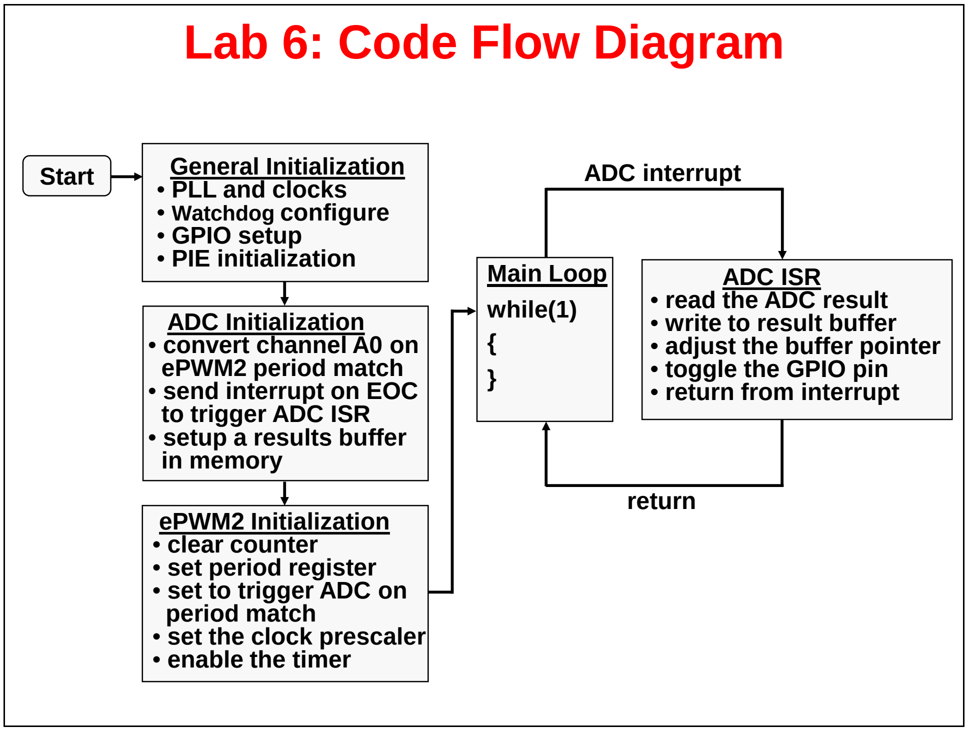

Lab 6: ADC Sampling

GND
+3.3 V (GPIO19)
Toggle (GPIO18)
jumper wire
ADCINA0
ADC-A
RESULT0
CPU copies result to buffer during ADC ISR
data memory
pointer rewind
DAC-B
Sine Table
ePWM2 triggering ADC on period match using SOCA trigger every 20 µs (50 kHz)
ePWM2
View ADC buffer PWM samples
Code Composer Studio

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

1. Using software
    a. SOCx bit (where x = 0 to 15) in the ADC SOC Force 1 Register (ADCSOCFRC1) causes a software initiated conversion
2. Automatically triggered on user selectable conditions
    a. CPU Timer 0/1/2 interrupt
    b. ePWMxSOCA / ePWMxSOCB (where x = 1 to 12)
       - 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 SOCx (where x = 0 to 15) selected by the ADC Interrupt Trigger SOC Select1/2 Register (ADCINTSOCSEL1/2)
3. Externally triggered using a pin
    a. ADCSOC pin (GPIO/ADCEXTSOC)

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 (GPIO18) high and low in the ADC interrupt service routine. The ADC ISR will also toggle LED D9 on the LaunchPad as a visual indication that the ISR is running. This pin will be connected to the ADC input pin, and sampled. After taking some data, Code Composer Studio will be used to plot the results. A flow chart of the code is shown in the following slide.



Lab 6: Code Flow Diagram

Start

General Initialization
- PLL and clocks
- Watchdog configure
- GPIO setup
- PIE initialization

ADC Initialization
- convert channel A0 on ePWM2 period match
- send interrupt on EOC to trigger ADC ISR
- setup a results buffer in memory

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

Main Loop

```
while(1)
{
}
```

ADC interrupt

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

return

Notes

- Program performs conversion on ADC channel A0 (ADCINA0 pin)
- ADC conversion is set at a 50 kHz sampling rate
- ePWM2 is triggering the ADC on period match using SOCA trigger
- Data is continuously stored in a circular buffer
- GPIO18 pin is also toggled in the ADC ISR
- ADC ISR will also toggle the LaunchPad LED D9 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 then click `Browse…` next to the "Select search-directory" box. Navigate to: `C:\C28x\Labs\Lab6\cpu01` and click `OK`. 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                                 Gpio.c
CodeStartBranch.asm                   Lab_5_6_7.cmd
Dac.c                                 Main_6.c
DefaultIsr_6.c                        PieCtrl.c
DelayUs.asm                           PieVect.c
EPwm_6.c                              Sinetable.c
F28x7xD_Adc.c                         SysCtrl.c
F2837xD_GlobalVariableDefs.c          Watchdog.c
F2837xD_Headers_nonBIOS_cpu1.cmd      Xbar.c
```

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()`, `InitEPwm()` and `InitDacb()` 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.c` to configure SOC0 in the ADC as follows:
   - SOC0 converts input ADCINA0 in single-sample mode
   - SOC0 has a 20 SYSCLK cycle acquisition window
   - SOC0 is triggered by the ePWM2 SOCA
   - SOC0 triggers ADCINT1 on end-of-conversion
   - All SOCs run round-robin

   Be sure to modify `Adc.c` and *not* `F2837xD_Adc.c` which is used for the ADC calibration.

4. Using the "PIE Interrupt Assignment Table" find the location for the ADC interrupt "`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.c` to do the following:

   - Enable the "ADCA1" interrupt in the PIE (Hint: use the PieCtrlRegs structure)
   - Enable the appropriate core interrupt in the IER register

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

Build and Load

7. Click the "`Build`" button and watch the tools run in the Console window. Check for errors in the Problems window.
8. Click the "`Debug`" button (green bug). A Launching Debug Session window will open. Select only CPU1 to load the program on (i.e. *uncheck* CPU2), and then click `OK`. Then the CCS Debug perspective view should open, the program will load automatically, and you should now be at the start of `main()`. If the device has been power cycled since the last lab exercise, be sure to configure the boot mode to EMU_BOOT_SARAM using the Scripts menu.

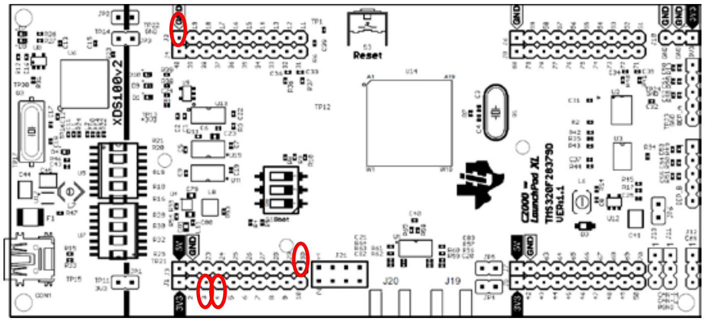
Run the Code

9. 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:



10. Using a jumper wire, connect the ADCINA0 (header J3, pin #30) to "GND" (header J2, 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. Note that you may not get exactly 0x0000 if the device you are using has positive offset error.
11. Adjust the jumper wire to connect the ADCINA0 (header J3, pin #30) to "+3.3V" (header J1, pin #3; GPIO-19) on the LaunchPad. (Note: pin # GPIO-19 has been set to "1" in Gpio.c). Then run the code again, and halt it after a few seconds. Verify that the ADC results buffer contains the expected value of ~0x0FFF. Note that you may not get exactly 0x0FFF if the device you are using has negative offset error.
12. Adjust the jumper wire to connect the ADCINA0 (header J3, pin #30) to GPIO18 (header J1, pin #4) 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?
13. 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.

14. Recall that the code toggled the GPIO18 pin alternately high and low. (Also, the ADC ISR is toggling the LED D9 on the LaunchPad as a visual indication that the ISR is running). If you had an oscilloscope available to display GPIO18, you would expect to see a square-wave. Why does Code Composer Studio plot resemble a triangle wave? What is the signal processing term for what is happening here?

15. Recall that the program toggled the GPIO18 pin at a 50 kHz rate. Therefore, a complete cycle (toggle high, then toggle low) occurs at half this rate, or 25 kHz. We therefore expect the period of the waveform to be 40 μs. 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.

16. The memory and graph windows displaying *AdcBuf* should still be open. The jumper wire between ADCINA0 (header J3, pin #30) and GPIO18 (header J1, pin #4) should still be connected. In real-time mode, we will have our window continuously refresh at the default rate. To view the refresh rate click:

`Window → Preferences…`

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

- "Continuous refresh interval (milliseconds)" = 500

Click OK.

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

17. 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.

18. Enable the Memory Browser for continuous refresh using the same procedure as the previous step.

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

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

These Script functions are executed by clicking:

`Scripts → Realtime Emulation Control → Function`

In the remaining lab exercises we will be using the first and third above Script functions to run and halt the code in real-time mode. Alternatively, the CPU Reset, Real-time mode, Resume, and Suspend buttons on the Code Composer Studio tool bar can be used.

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

`Scripts → Realtime Emulation Control → Run_Realtime_with_Reset`

21. ***Carefully*** remove and replace the jumper wire from GPIO18 (header J1, pin #4). Are the values updating in the Memory Browser and Single Time graph as expected?

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

`Scripts → Realtime Emulation Control → Full_Halt`

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

- Open and inspect `Main_6.c`. Notice that the global variable DEBUG_TOGGLE is used to control the toggling of the GPIO18 pin. This is the pin being read with the ADC.

- Highlight DEBUG_TOGGLE with the mouse, right click and select "`Add Watch Expression…`" and then select OK. The global variable DEBUG_TOGGLE should now be in the Expressions window with a value of "1".
- Enable the Expressions window for continuous refresh
- Run the code in real-time mode and change the value to "0". Are the results shown in the memory and graph window as expected? Change the value back to "1". As you can see, we are modifying data memory contents while the processor is running in real-time (i.e., we are not halting the MCU nor interfering with its operation in any way)! When done, fully halt the CPU.

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 (ADC-A1). Then using the jumper wire we will connect the DACB output to the ADCA input (ADC-A0) and display the sine wave in a graph window.

24. 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;
    }
    DacbRegs.DACVALS.all = DacOutput;
```

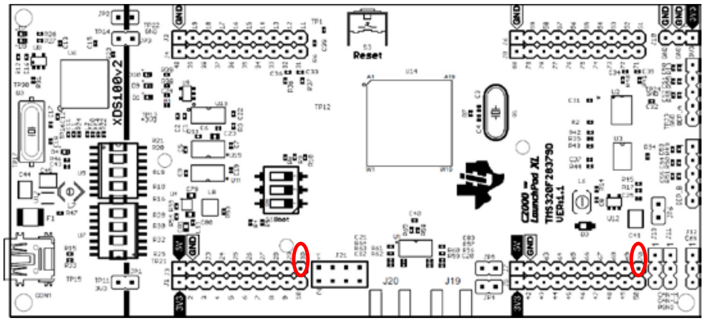
The variable `DacOffset` allows the user to adjust the DC output of DACB from the Expressions window in CCS. The variable Sine_Enable is a switch which adds a fixed frequency sine wave to the DAC offset. The sine wave is generated using a 25-point look-up table contained in the `SineTable.c` file. We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

25. 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.

26. Add the following variables to the Expressions window:

    - Sine_Enable
    - DacOffset

27. Adjust the jumper wire to connect the ADCINA0 (header J3, pin #30) to DACB (header J7, pin #70) on the LaunchPad. Refer to the following diagram for the pins that need to be connected.

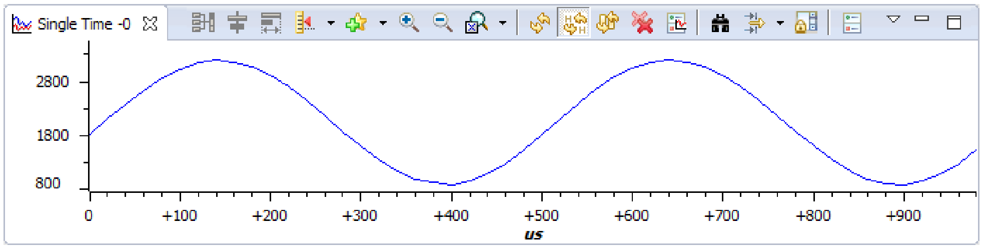


28. Run the code (real-time mode) using the Script function: `Scripts` → `Realtime Emulation Control` → `Run_Realtime_with_Reset`

29. 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).

30. Enable the sine generator by changing the variable `Sine_Enable` in the Expressions window to 1.

31. You should now see sinusoidal data in the graph window.



32. 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).

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

Terminate Debug Session and Close Project

34. 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.

35. Next, close the project by right-clicking on `Lab6` in the Project Explorer window and select `Close Project`.

Optional Exercise

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

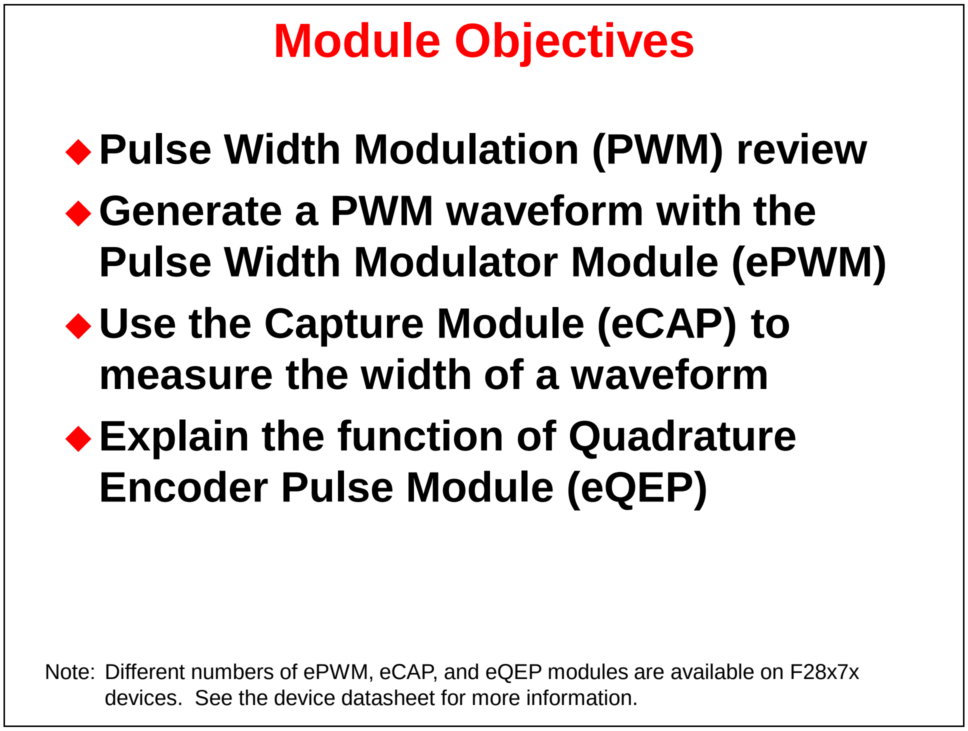
End of Exercise

Control Peripherals

Introduction

The C2000 high-performance control peripherals are an integral component for all digital control systems, and within the F2837xD these peripherals are common between the two CPU subsystems. After reset they are connected to the CPU1 subsystem, and a series of CPU Select registers are used to configure each peripheral individually to be either controlled CPU1 subsystem or CPU2 subsystem. This module starts with a review of pulse width modulation (PWM) and then explains how the ePWM is used for generating PWM waveforms. Also, the use of the eCAP and the eQEP will be discussed.

Module Objectives



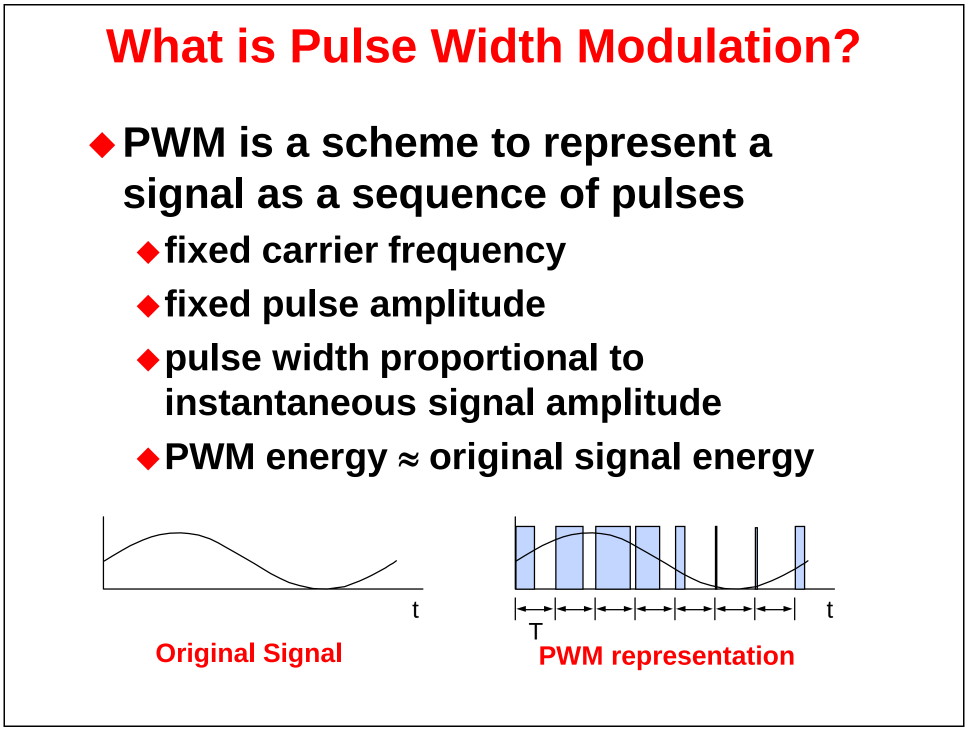
Module Objectives

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

Note: Different numbers of ePWM, eCAP, and eQEP modules are available on F28x7x devices. See the device datasheet for more information.

Chapter Topics

PWM Review



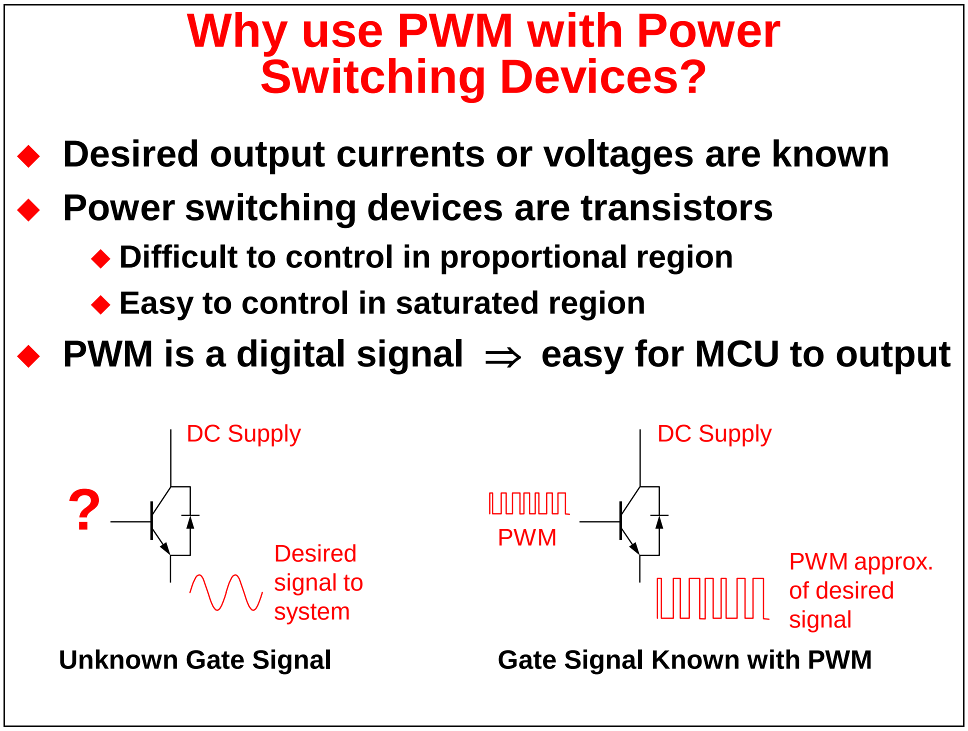
What is Pulse Width Modulation?

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

Original Signal

PWM representation

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



Why use PWM with Power Switching Devices?

- Desired output currents or voltages are known
- Power switching devices are transistors
  - Difficult to control in proportional region
  - Easy to control in saturated region
- PWM is a digital signal ⇒ easy for MCU to output

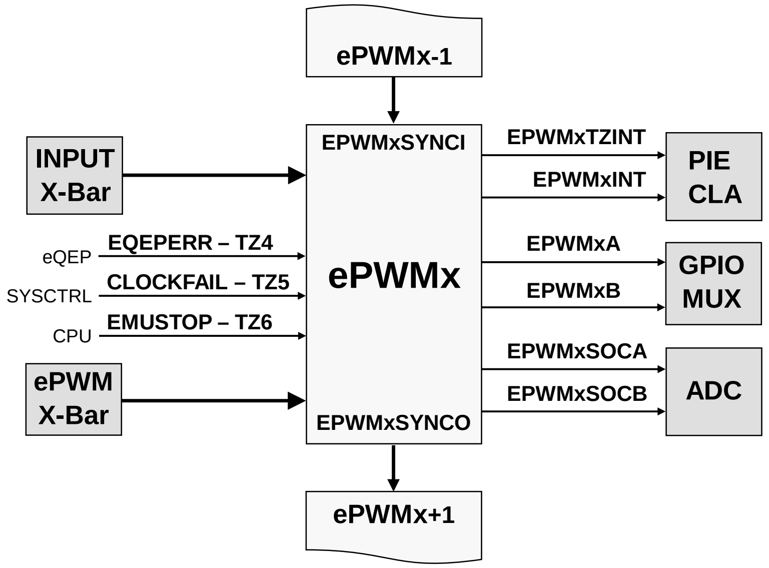
Unknown Gate Signal

Gate Signal Known with PWM

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

ePWM Module Signals and Connections



ePWMx-1

INPUT X-Bar

eQEP EQEPERR – TZ4

SYSCTRL CLOCKFAIL – TZ5

CPU EMUSTOP – TZ6

ePWM X-Bar

EPWMxSYNCI

ePWMx

EPWMxSYNCO

ePWMx+1

EPWMxTZINT

EPWMxINT

PIE CLA

EPWMxA

EPWMxB

GPIO MUX

EPWMxSOCA

EPWMxSOCB

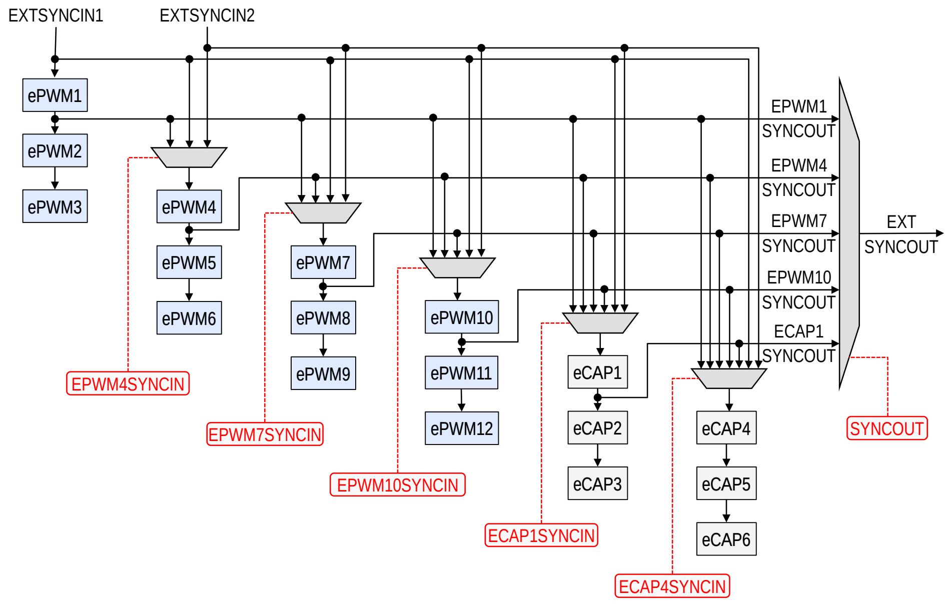
ADC

Note: the order in which the ePWM modules are connected is determined by the device synchronization scheme

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 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 starter 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.

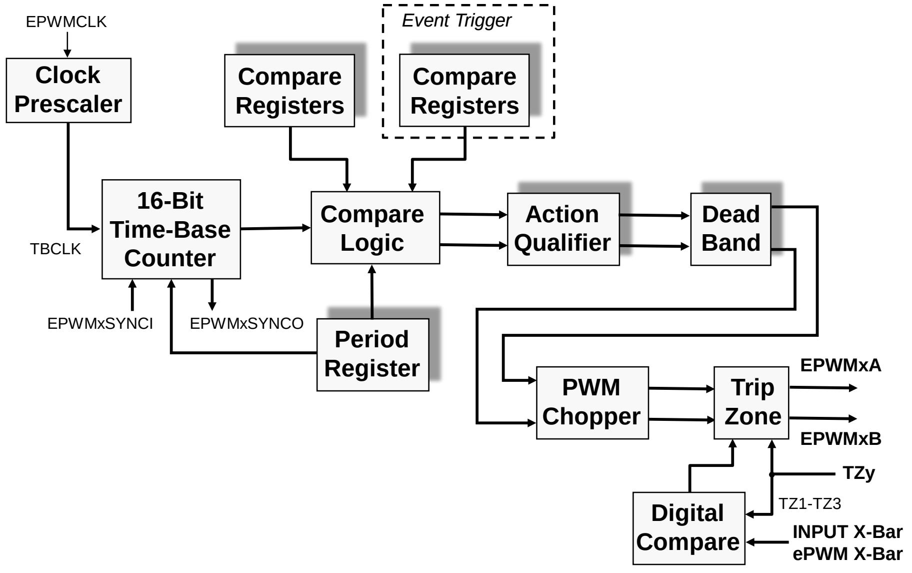
ePWM Synchronization Scheme

SyncSocRegs.SYNCSELECT



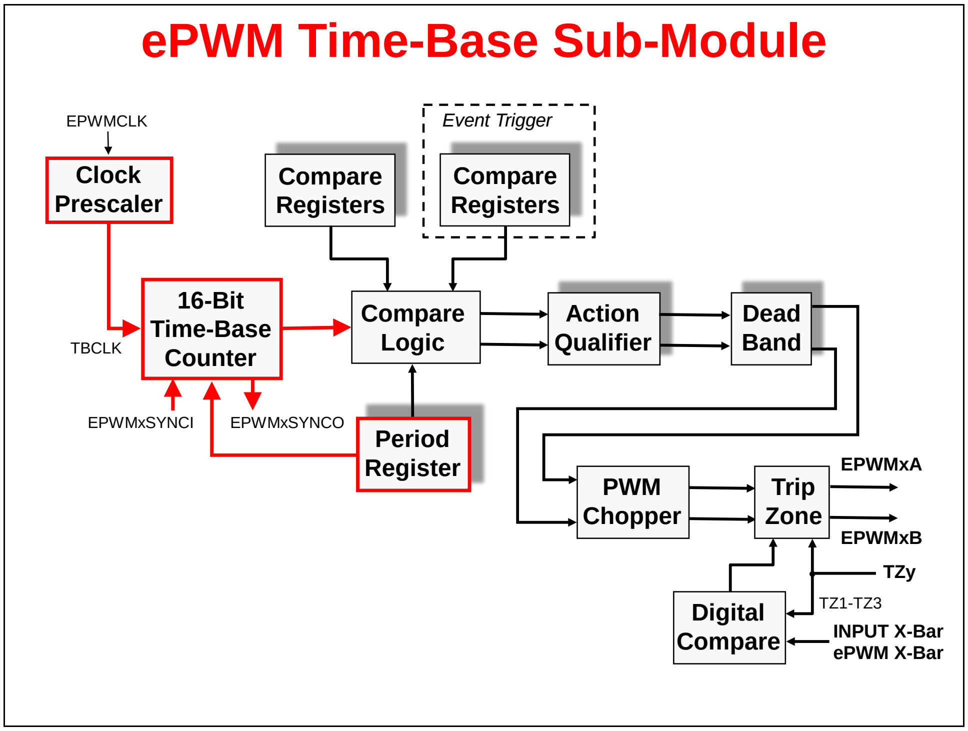
Various ePWM modules (and eCAP units) can be grouped together for synchronization.

ePWM Block Diagram



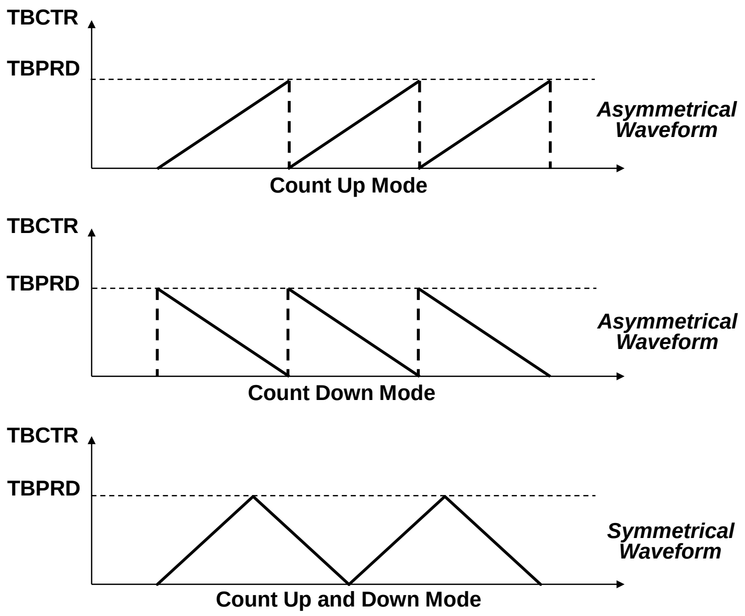
The ePWM module consists of eight submodules: time-base, counter-compare, action-qualifier, dead-band generator, PWM chopper, trip-zone, digital-compare, and event-trigger.

ePWM Time-Base Sub-Module

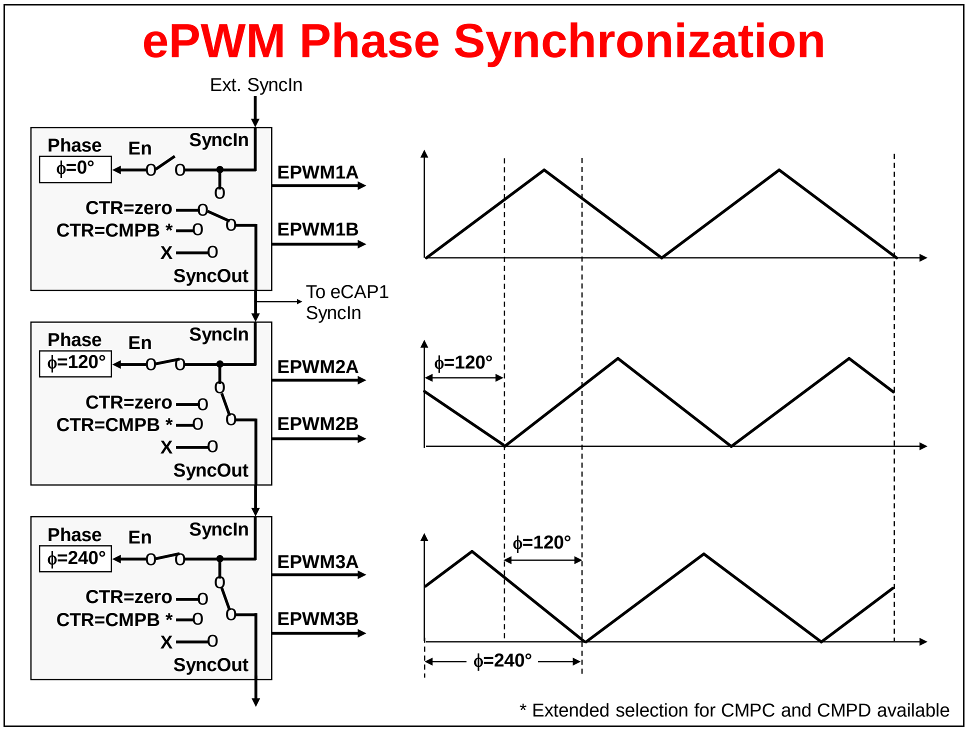


The time-base submodule consists of a dedicated 16-bit counter, along with built-in synchronization logic to allow multiple ePWM modules to work together as a single system. A clock pre-scaler divides the EPWM clock to the counter and a period register is used to control the frequency and period of the 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.

ePWM Time-Base Count Modes



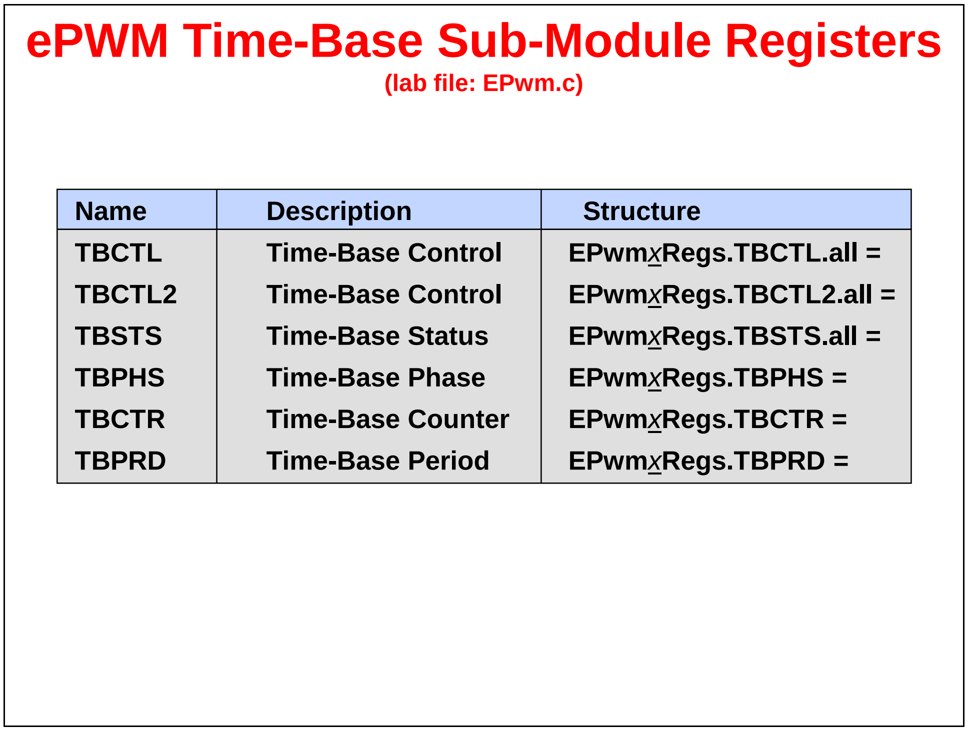
The time-base counter operates in three modes: up-count, down-count, and up-down-count. In up-count mode the time-base counter starts counting from zero and increments until it reaches the period register value, then the time-base counter resets to zero and the count sequence starts again. Likewise, 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.



ePWM Phase Synchronization

* Extended selection for CMPC and CMPD available

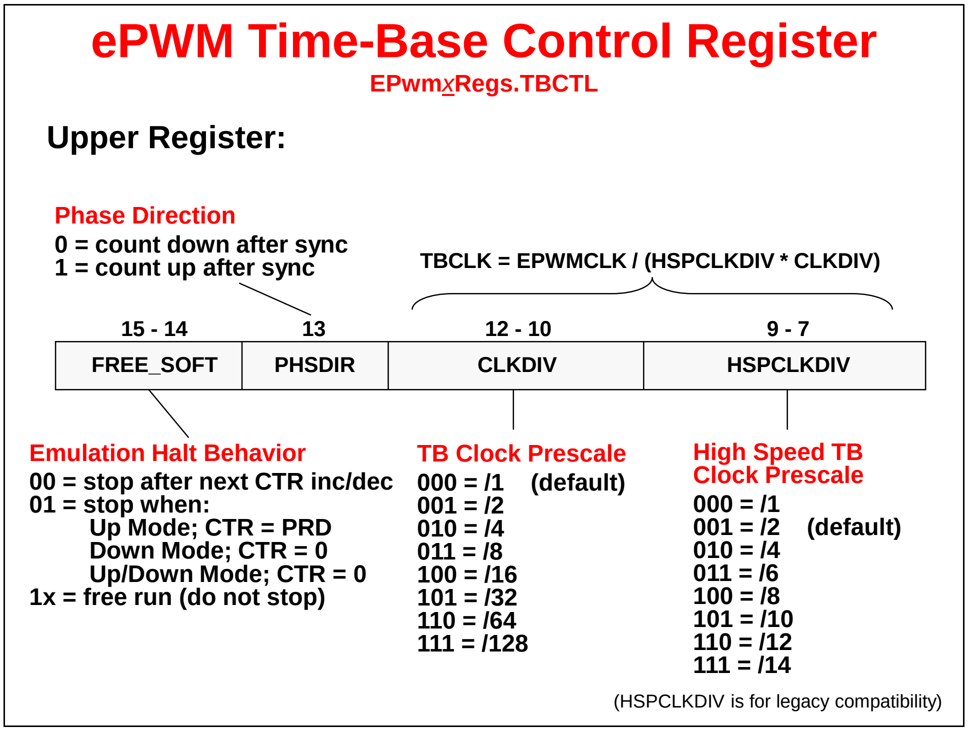
Synchronization allows multiple ePWM modules to work together as a single system. The synchronization is based on a synch-in signal, time-base counter equals zero, or time-base counter equals compare B register. Additionally, the waveform can be phase-shifted.



ePWM Time-Base Sub-Module Registers

(lab file: EPwm.c)

Name	Description	Structure
TBCTL	Time-Base Control	EPwm*x*Regs.TBCTL.all =
TBCTL2	Time-Base Control	EPwm*x*Regs.TBCTL2.all =
TBSTS	Time-Base Status	EPwm*x*Regs.TBSTS.all =
TBPHS	Time-Base Phase	EPwm*x*Regs.TBPHS =
TBCTR	Time-Base Counter	EPwm*x*Regs.TBCTR =
TBPRD	Time-Base Period	EPwm*x*Regs.TBPRD =



ePWM Time-Base Control Register

EPwmxRegs.TBCTL

Upper Register:

Phase Direction

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

TBCLK = EPWMCLK / (HSPCLKDIV * CLKDIV)

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

Emulation Halt Behavior

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

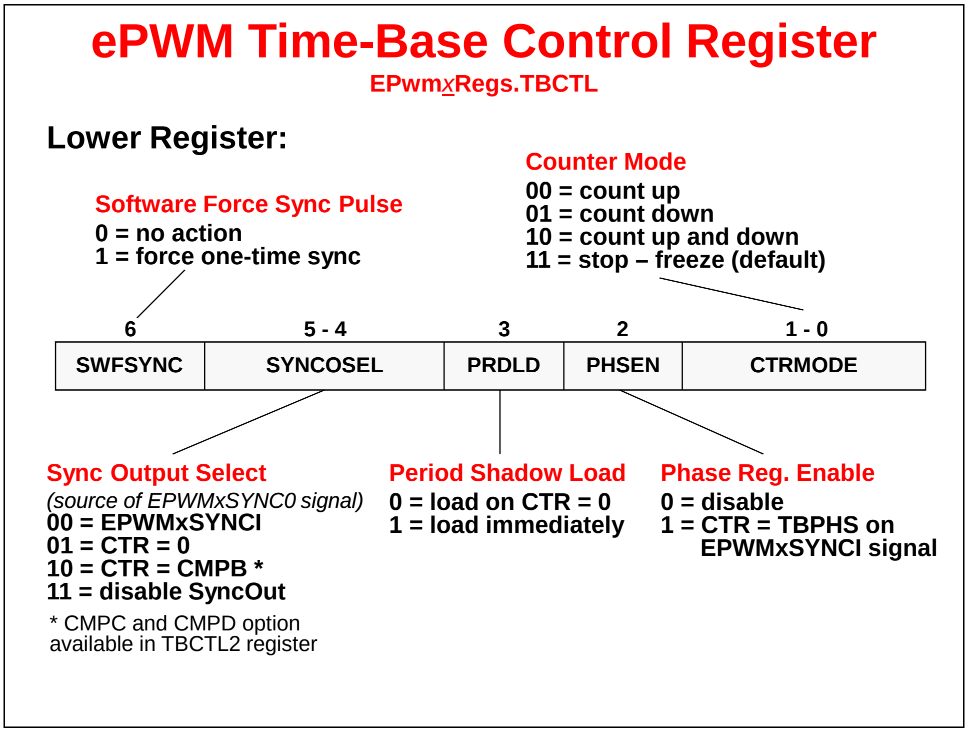
TB Clock Prescale

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

High Speed TB Clock Prescale

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

(HSPCLKDIV is for legacy compatibility)



ePWM Time-Base Control Register

EPwmxRegs.TBCTL

Lower Register:

Software Force Sync Pulse

0 = no action
1 = force one-time sync

Counter Mode

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

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

Sync Output Select

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

* CMPC and CMPD option
available in TBCTL2 register

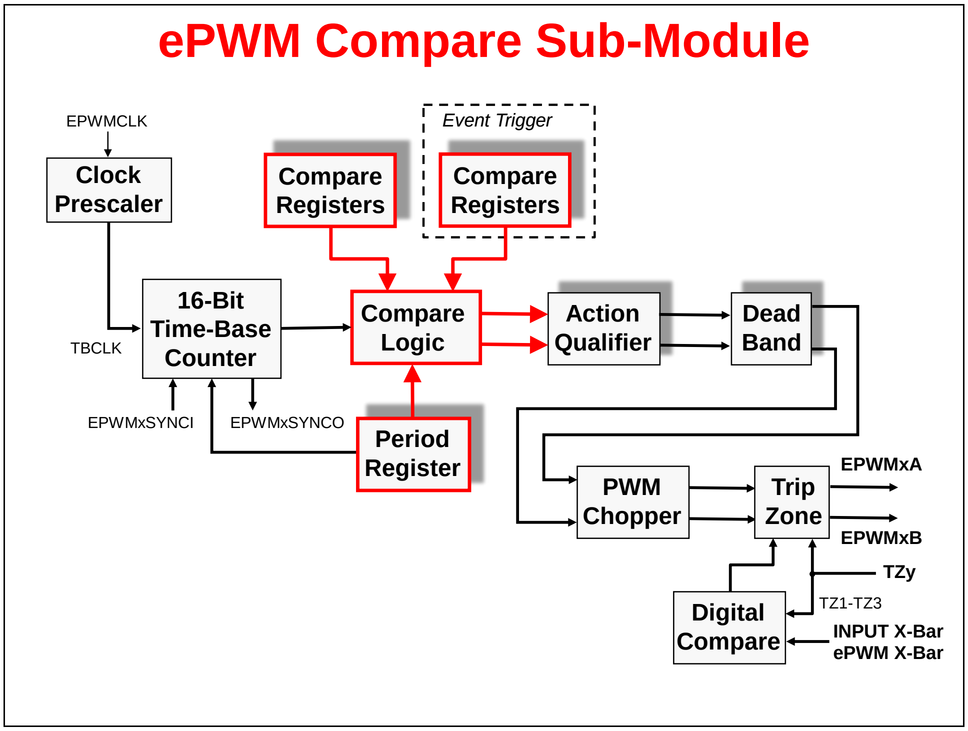
Period Shadow Load

0 = load on CTR = 0
1 = load immediately

Phase Reg. Enable

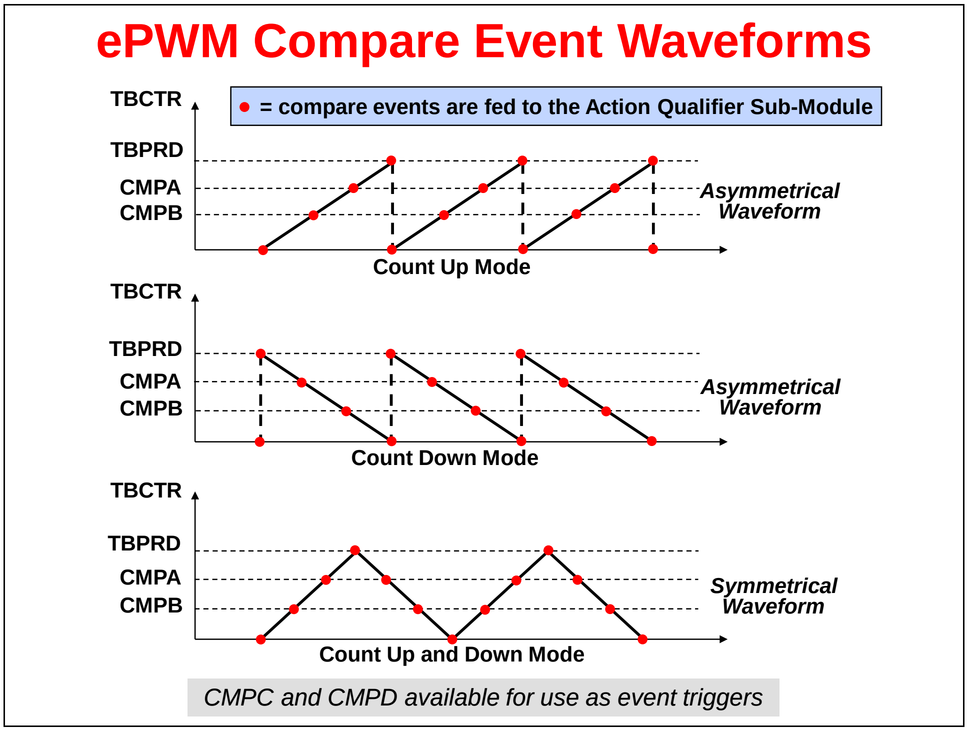
0 = disable
1 = CTR = TBPHS on
    EPWMxSYNCI signal

ePWM Compare Sub-Module



ePWM Compare Sub-Module

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.

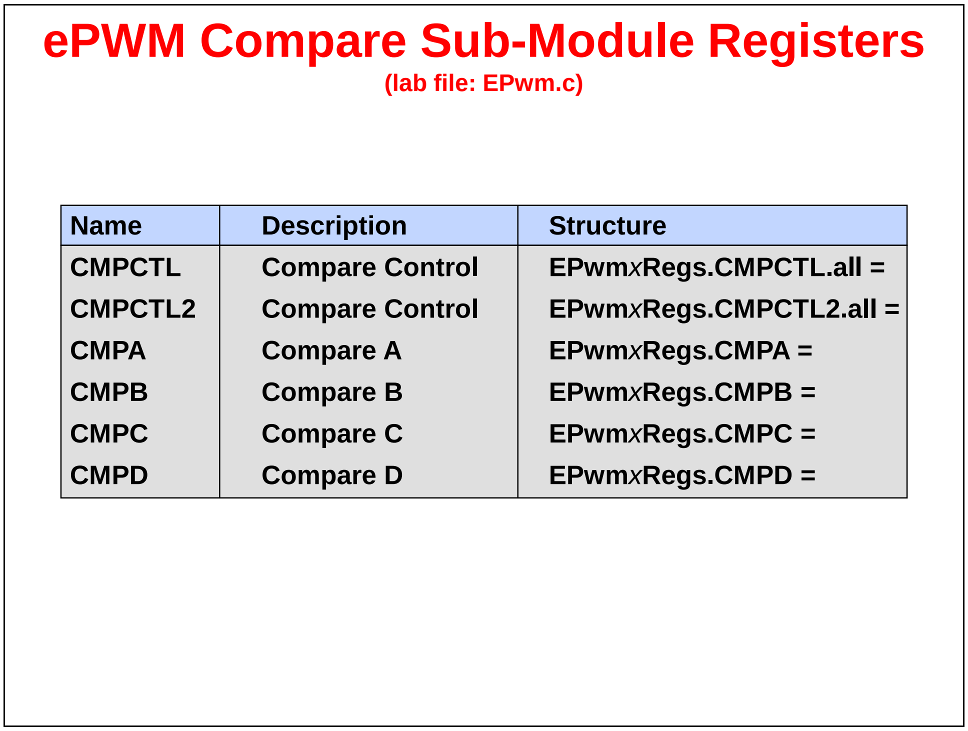


ePWM Compare Event Waveforms

● = compare events are fed to the Action Qualifier Sub-Module

CMPC and CMPD available for use as event triggers

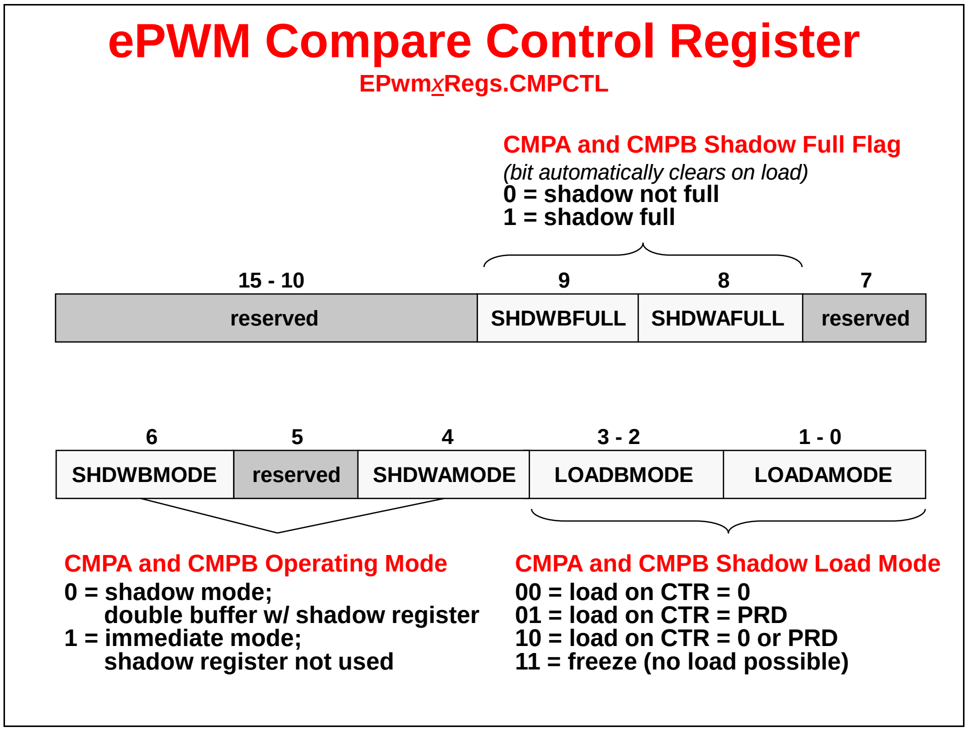
The above ePWM Compare Event Waveform diagram shows the compare matches which are fed into the action qualifier. Notice that with the count up and countdown mode, there are matches on the up-count and down-count.



ePWM Compare Sub-Module Registers

(lab file: EPwm.c)

Name	Description	Structure
CMPCTL	Compare Control	EPwm*x*Regs.CMPCTL.all =
CMPCTL2	Compare Control	EPwm*x*Regs.CMPCTL2.all =
CMPA	Compare A	EPwm*x*Regs.CMPA =
CMPB	Compare B	EPwm*x*Regs.CMPB =
CMPC	Compare C	EPwm*x*Regs.CMPC =
CMPD	Compare D	EPwm*x*Regs.CMPD =



ePWM Compare Control Register

EPwmxRegs.CMPCTL

CMPA and CMPB Shadow Full Flag

(bit automatically clears on load)

0 = shadow not full
1 = shadow full

15 - 10	9	8	7
reserved	SHDWBFULL	SHDWAFULL	reserved

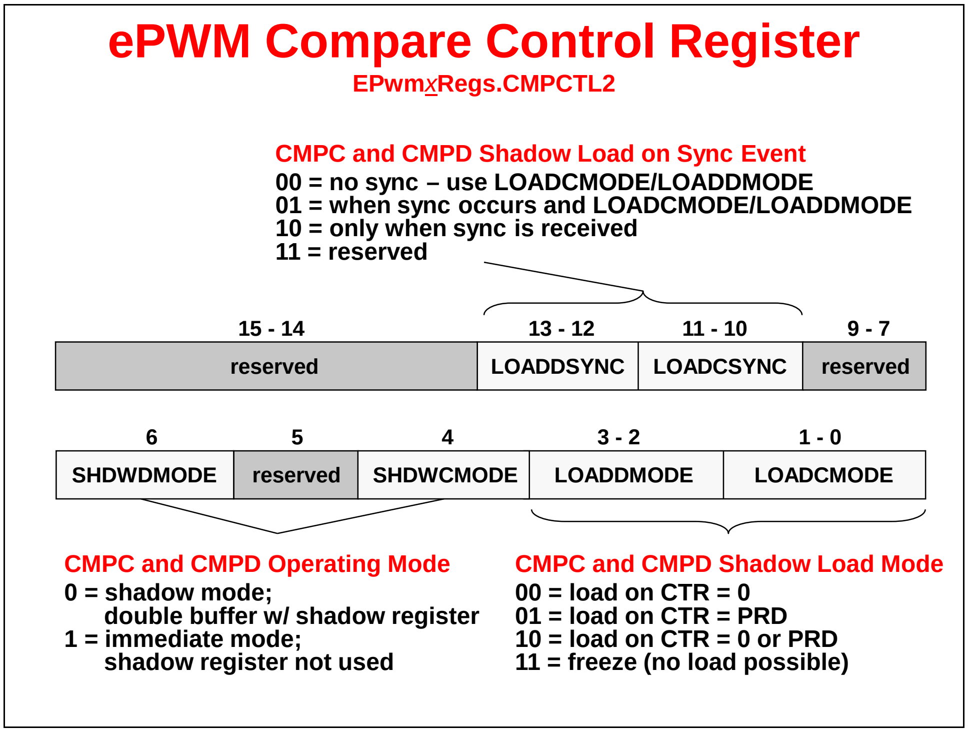
6	5	4	3 - 2	1 - 0
SHDWBMODE	reserved	SHDWAMODE	LOADBMODE	LOADAMODE

CMPA and CMPB Operating Mode

0 = shadow mode;
double buffer w/ shadow register
1 = immediate mode;
shadow register not used

CMPA and CMPB Shadow Load Mode

00 = load on CTR = 0
01 = load on CTR = PRD
10 = load on CTR = 0 or PRD
11 = freeze (no load possible)



ePWM Compare Control Register

EPwmxRegs.CMPCTL2

CMPC and CMPD Shadow Load on Sync Event

00 = no sync – use LOADCMODE/LOADDMODE
01 = when sync occurs and LOADCMODE/LOADDMODE
10 = only when sync is received
11 = reserved

15 - 14	13 - 12	11 - 10	9 - 7
reserved	LOADDSYNC	LOADCSYNC	reserved

6	5	4	3 - 2	1 - 0
SHDWDMODE	reserved	SHDWCMODE	LOADDMODE	LOADCMODE

CMPC and CMPD Operating Mode

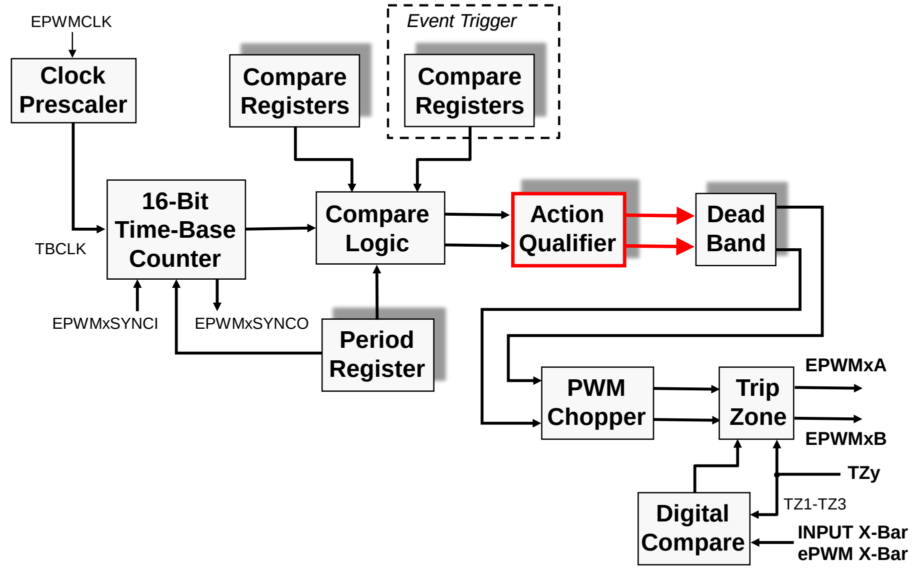
0 = shadow mode;
double buffer w/ shadow register
1 = immediate mode;
shadow register not used

CMPC and CMPD Shadow Load Mode

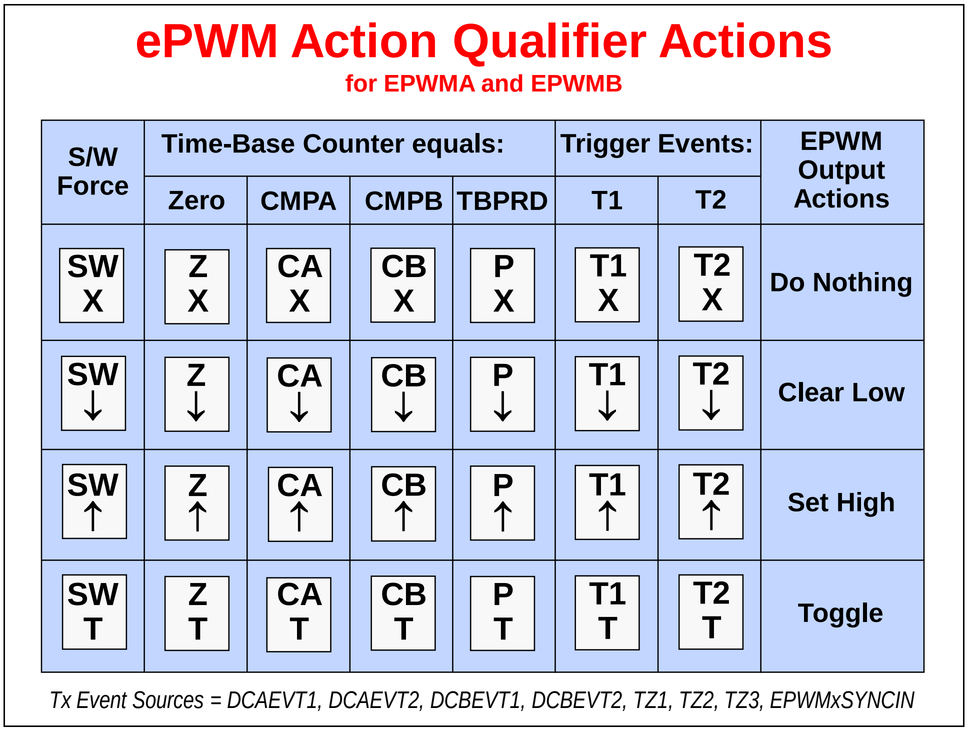
00 = load on CTR = 0
01 = load on CTR = PRD
10 = load on CTR = 0 or PRD
11 = freeze (no load possible)

ePWM Action Qualifier Sub-Module

ePWM Action Qualifier Sub-Module



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

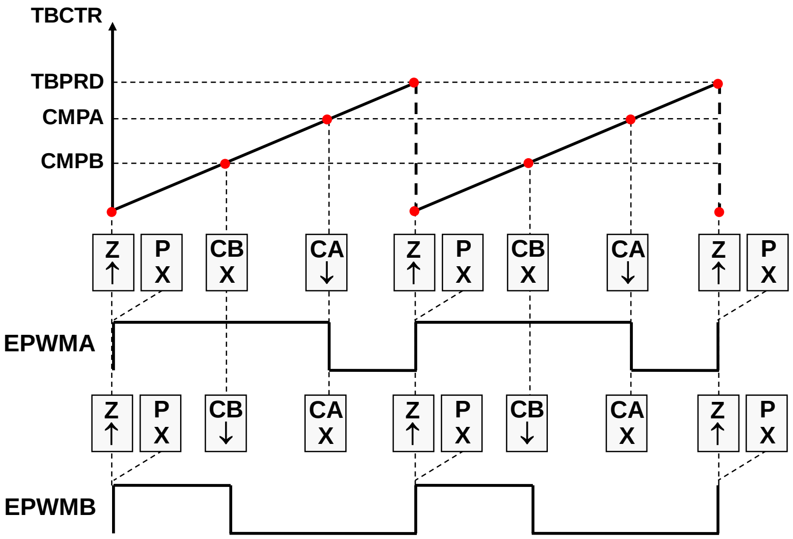
S/W Force	Time-Base Counter equals:				Trigger Events:		EPWM Output Actions
	Zero	CMPA	CMPB	TBPRD	T1	T2	
SW X	Z X	CA X	CB X	P X	T1 X	T2 X	Do Nothing
SW ↓	Z ↓	CA ↓	CB ↓	P ↓	T1 ↓	T2 ↓	Clear Low
SW ↑	Z ↑	CA ↑	CB ↑	P ↑	T1 ↑	T2 ↑	Set High
SW T	Z T	CA T	CB T	P T	T1 T	T2 T	Toggle

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 do 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 moveable 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.

ePWM Count Up Asymmetric Waveform

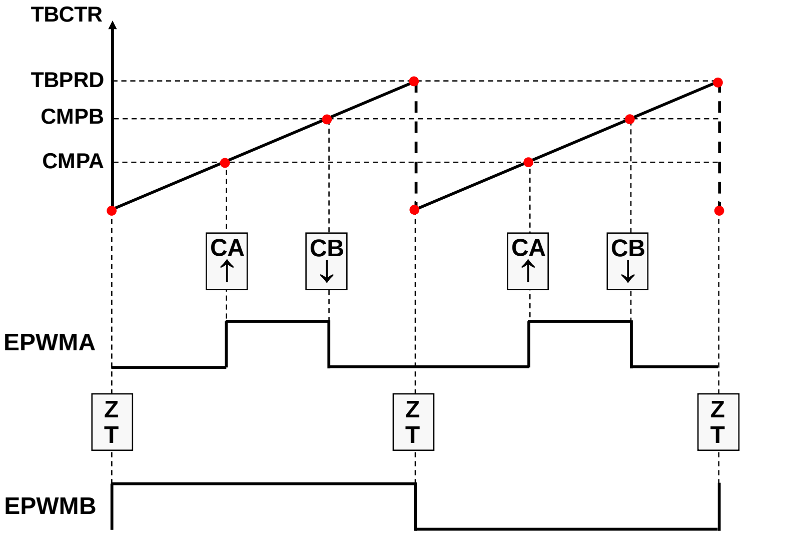
with Independent Modulation on EPWMA / B



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.

ePWM Count Up Asymmetric Waveform

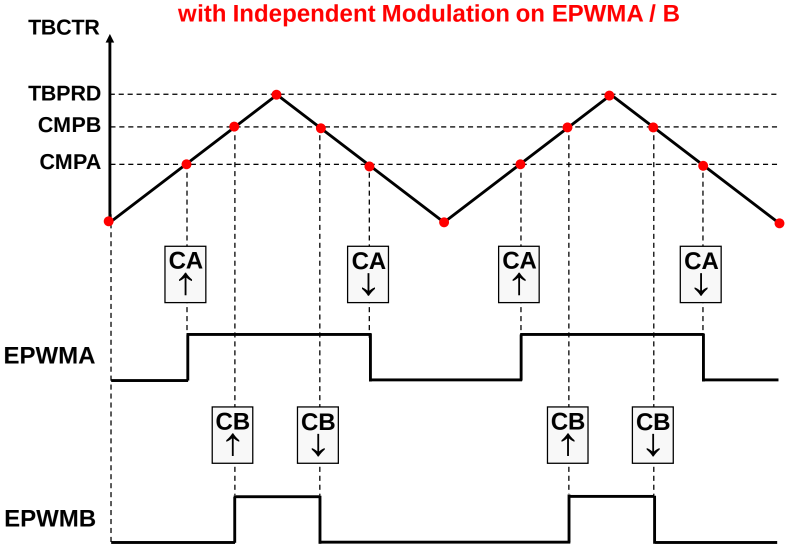
with Independent Modulation on EPWMA



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.

ePWM Count Up-Down Symmetric Waveform

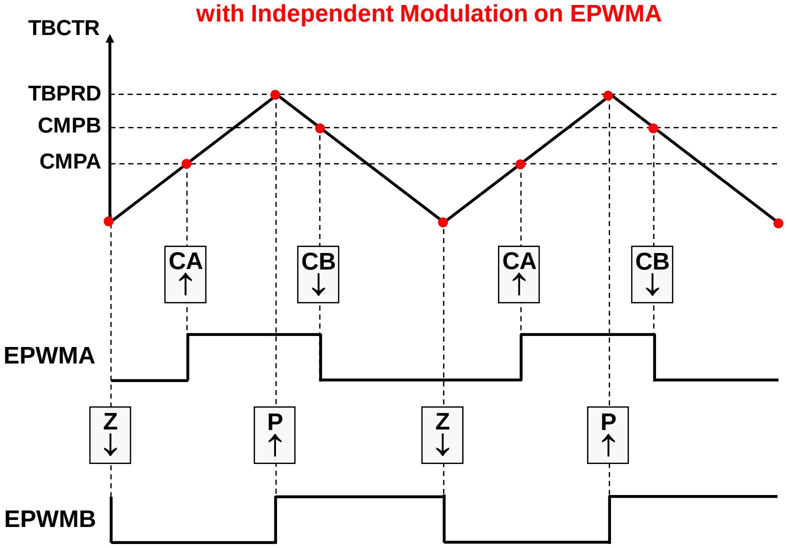
with Independent Modulation on EPWMA / B



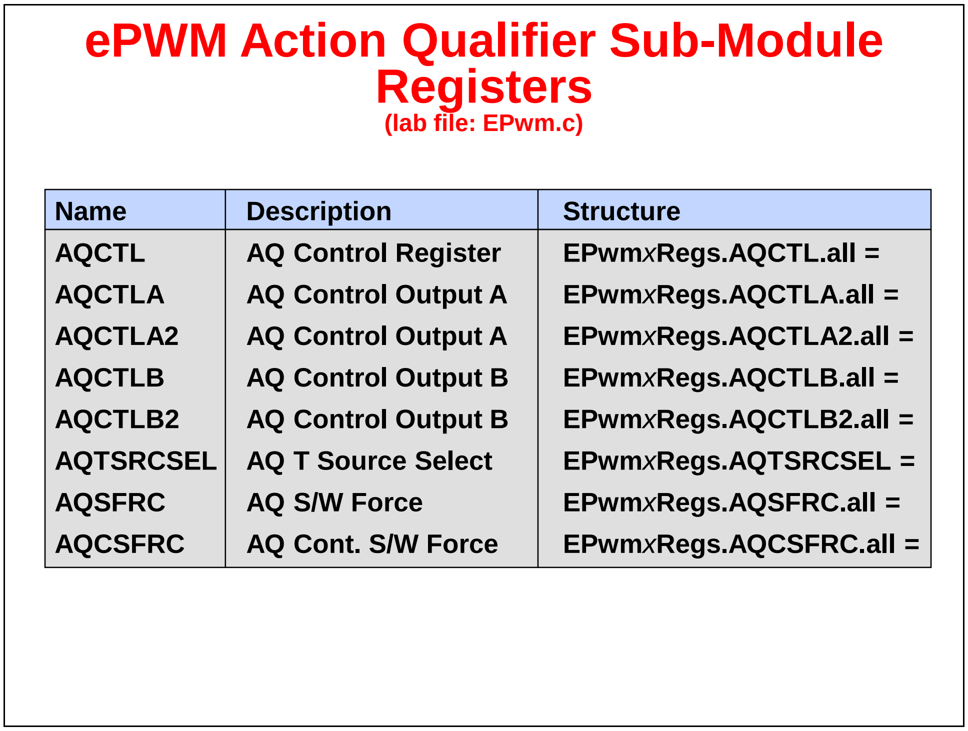
In the example above, there are different output actions on the up-count and down-count using a single compare register. The EPWMA and EPWMB outputs are being set high on the compare A and B up-count matches and cleared low on the compare A and B down-count matches.

ePWM Count Up-Down Symmetric Waveform

with Independent Modulation on EPWMA



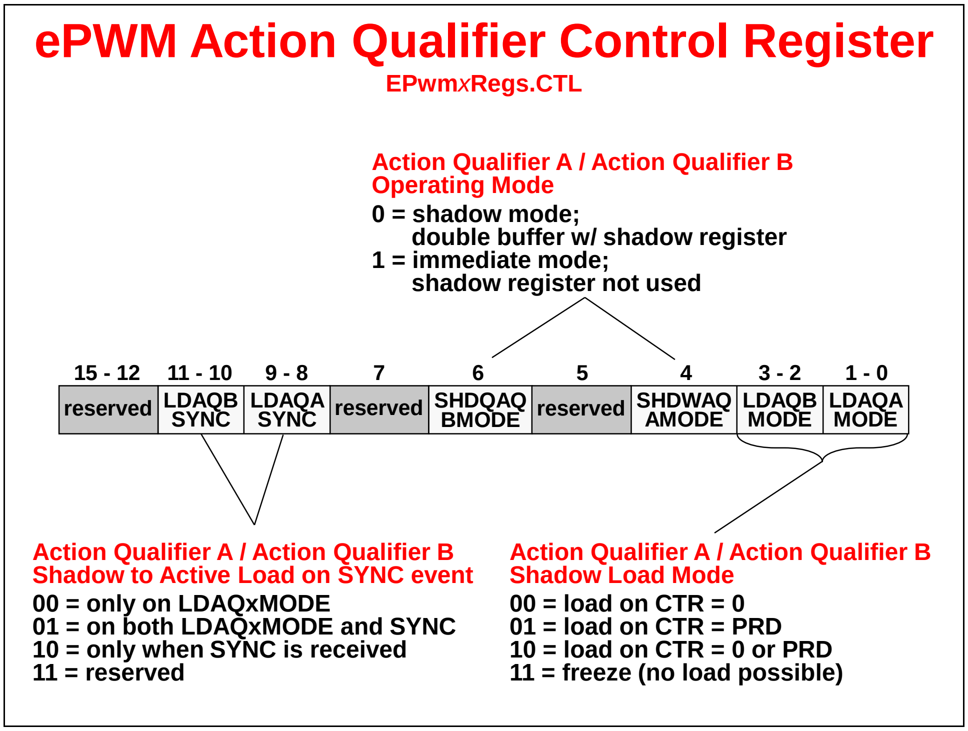
And finally in the example above, again using different output actions on the up-count and down-count, the EPWMA output is being set high on the compare A up-count match and being cleared low on the compare B down-count match. The EPWMB output is being cleared low on the zero match and being set high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



ePWM Action Qualifier Sub-Module Registers

(lab file: EPwm.c)

Name	Description	Structure
AQCTL	AQ Control Register	EPwm*x*Regs.AQCTL.all =
AQCTLA	AQ Control Output A	EPwm*x*Regs.AQCTLA.all =
AQCTLA2	AQ Control Output A	EPwm*x*Regs.AQCTLA2.all =
AQCTLB	AQ Control Output B	EPwm*x*Regs.AQCTLB.all =
AQCTLB2	AQ Control Output B	EPwm*x*Regs.AQCTLB2.all =
AQTSRCSEL	AQ T Source Select	EPwm*x*Regs.AQTSRCSEL =
AQSFRC	AQ S/W Force	EPwm*x*Regs.AQSFRC.all =
AQCSFRC	AQ Cont. S/W Force	EPwm*x*Regs.AQCSFRC.all =



ePWM Action Qualifier Control Register

EPwm*x*Regs.CTL

Action Qualifier A / Action Qualifier B
Operating Mode

0 = shadow mode;
double buffer w/ shadow register
1 = immediate mode;
shadow register not used

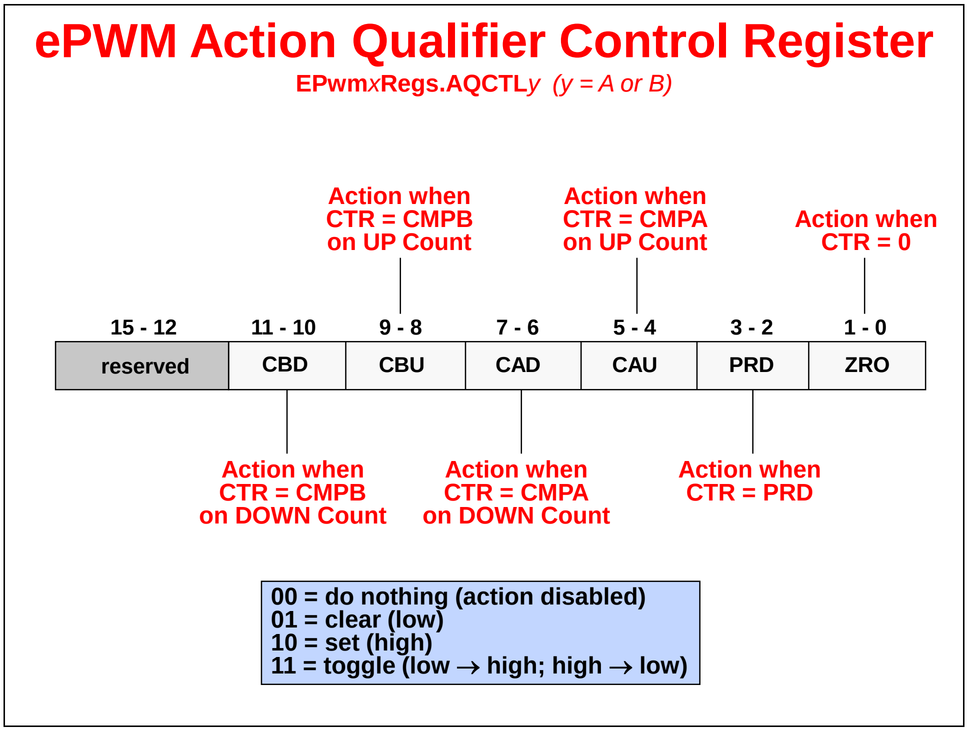
15 - 12	11 - 10	9 - 8	7	6	5	4	3 - 2	1 - 0
reserved	LDAQB SYNC	LDAQA SYNC	reserved	SHDQAQ BMODE	reserved	SHDWAQ AMODE	LDAQB MODE	LDAQA MODE

Action Qualifier A / Action Qualifier B
Shadow to Active Load on SYNC event

00 = only on LDAQxMODE
01 = on both LDAQxMODE and SYNC
10 = only when SYNC is received
11 = reserved

Action Qualifier A / Action Qualifier B
Shadow Load Mode

00 = load on CTR = 0
01 = load on CTR = PRD
10 = load on CTR = 0 or PRD
11 = freeze (no load possible)



ePWM Action Qualifier Control Register

EPwm*x*Regs.AQCTL*y* *(y = A or B)*

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

CBU: Action when CTR = CMPB on UP Count

CAU: Action when CTR = CMPA on UP Count

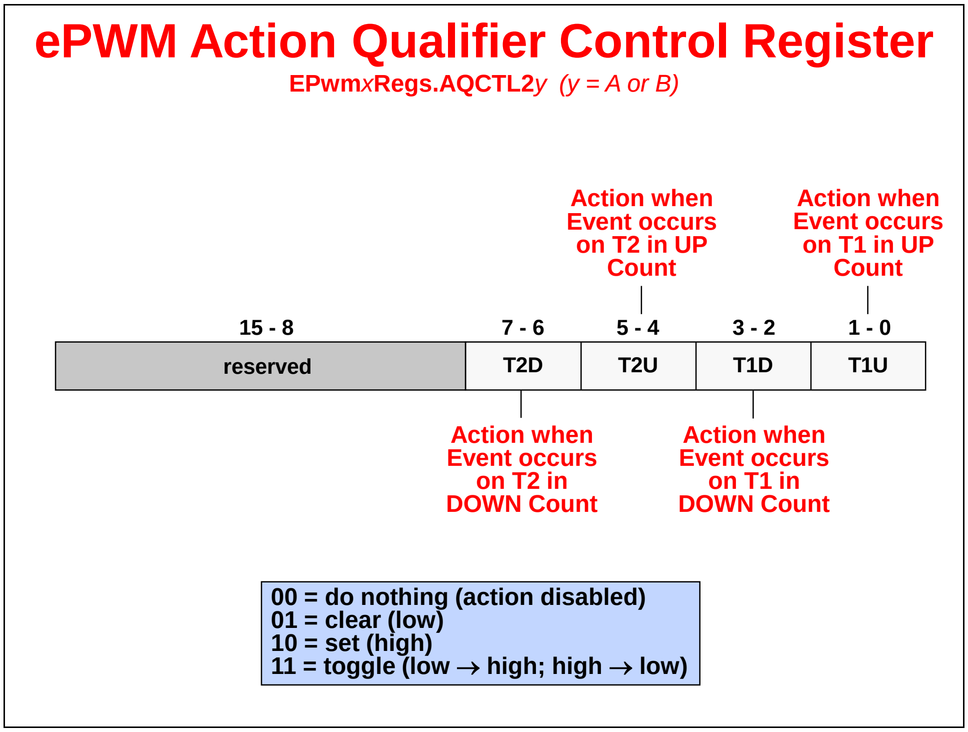
ZRO: Action when CTR = 0

CBD: Action when CTR = CMPB on DOWN Count

CAD: Action when CTR = CMPA on DOWN Count

PRD: Action when CTR = PRD

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



ePWM Action Qualifier Control Register

EPwm*x*Regs.AQCTL2*y* *(y = A or B)*

15 - 8	7 - 6	5 - 4	3 - 2	1 - 0
reserved	T2D	T2U	T1D	T1U

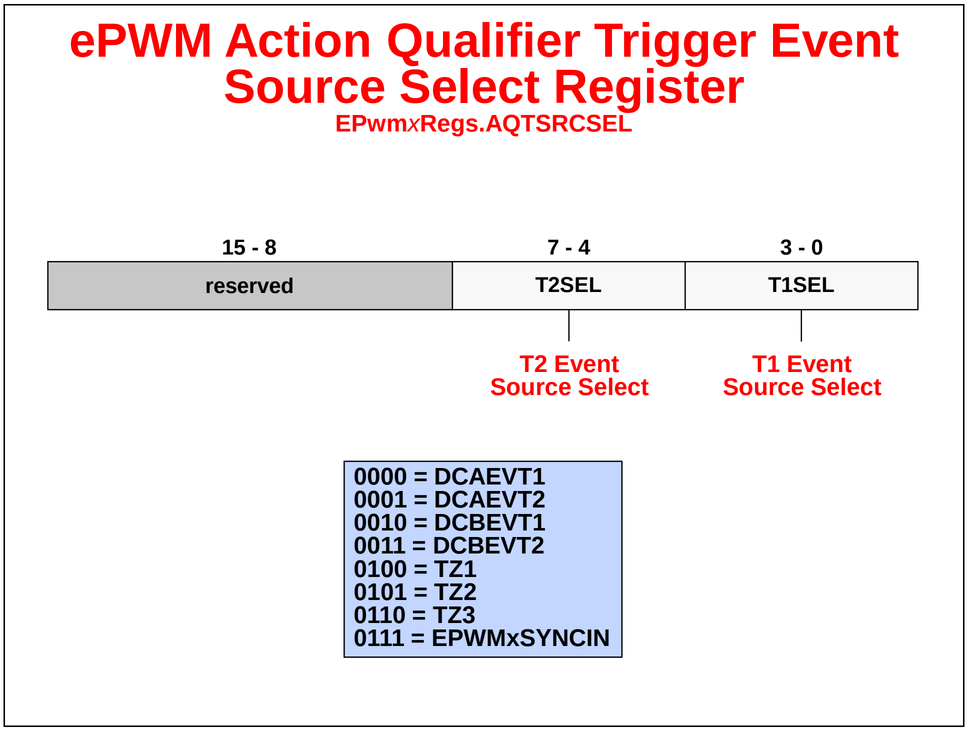
T2U: Action when Event occurs on T2 in UP Count

T1U: Action when Event occurs on T1 in UP Count

T2D: Action when Event occurs on T2 in DOWN Count

T1D: Action when Event occurs on T1 in DOWN Count

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



ePWM Action Qualifier Trigger Event Source Select Register

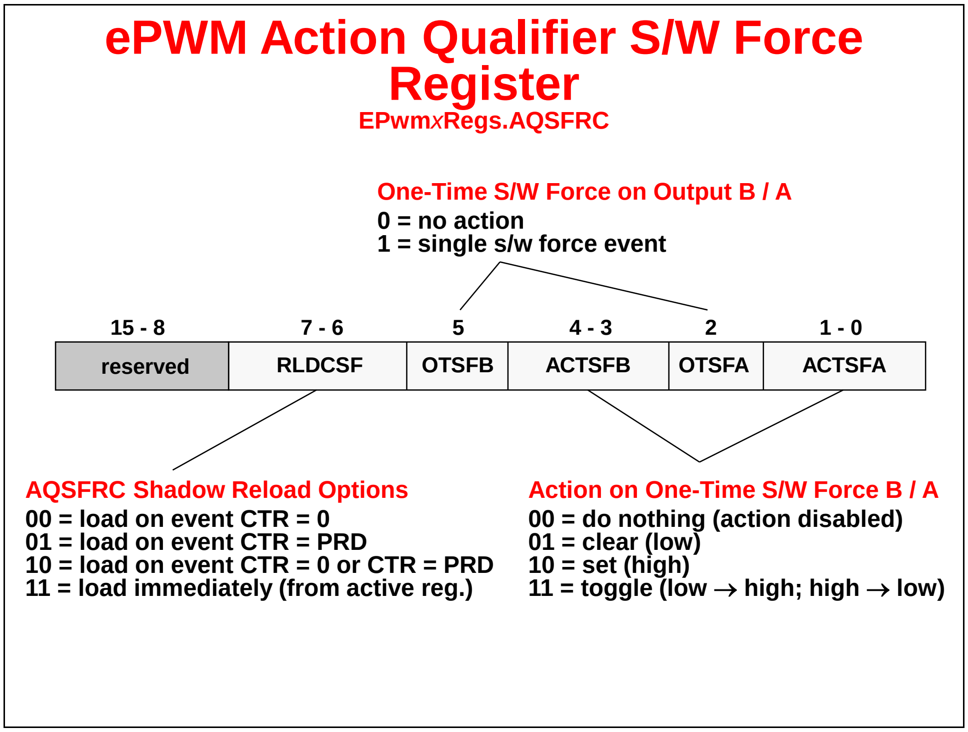
EPwm*x*Regs.AQTSRCSEL

15 - 8	7 - 4	3 - 0
reserved	T2SEL	T1SEL

T2 Event Source Select

T1 Event Source Select

0000 = DCAEVT1
0001 = DCAEVT2
0010 = DCBEVT1
0011 = DCBEVT2
0100 = TZ1
0101 = TZ2
0110 = TZ3
0111 = EPWMxSYNCIN



ePWM Action Qualifier S/W Force Register

EPwm*x*Regs.AQSFRC

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

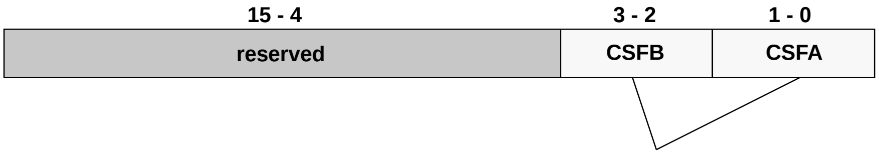
15 - 8	7 - 6	5	4 - 3	2	1 - 0
reserved	RLDCSF	OTSFB	ACTSFB	OTSFA	ACTSFA

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

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

ePWM Action Qualifier Continuous S/W Force Register

EPwm*x*Regs.AQCSFRC



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

Continuous S/W Force on Output B / A

00 = forcing disabled
01 = force continuous low on output
10 = force continuous high on output
11 = forcing disabled

Asymmetric and Symmetric Waveform Generation using the ePWM

PWM switching frequency:

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

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

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

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

PWM resolution:

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

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

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

PWM duty cycle:

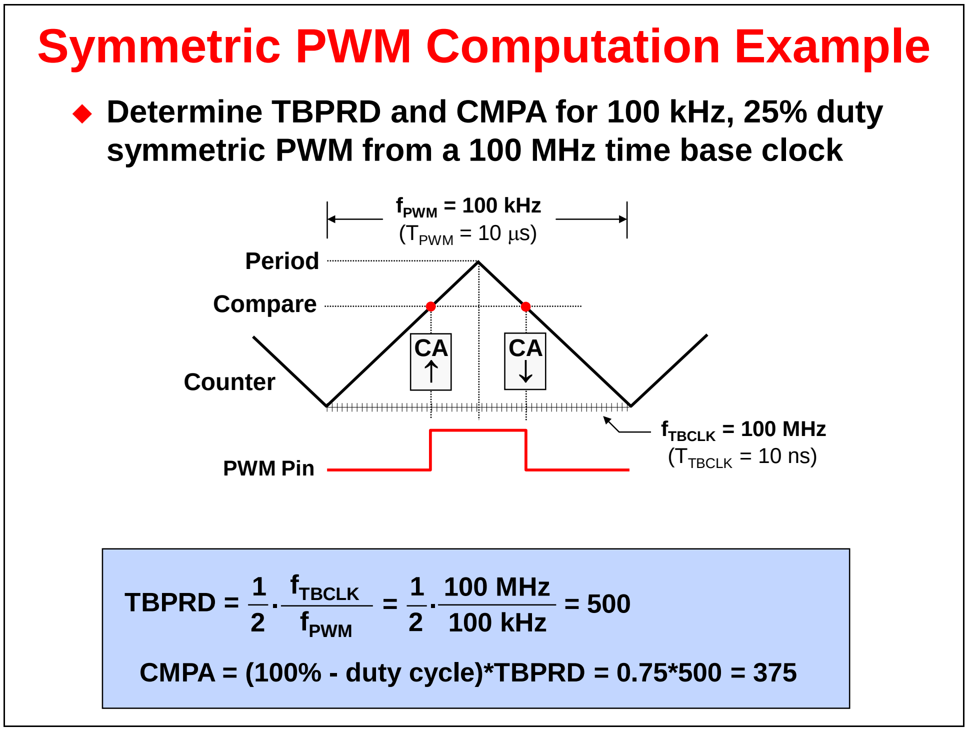
Duty cycle calculations are simple provided one remembers that the PWM signal is initially inactive during any particular timer period, and becomes active after the (first) compare match occurs. The timer compare register should be loaded with the value as follows:

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

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

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

PWM Computation Example

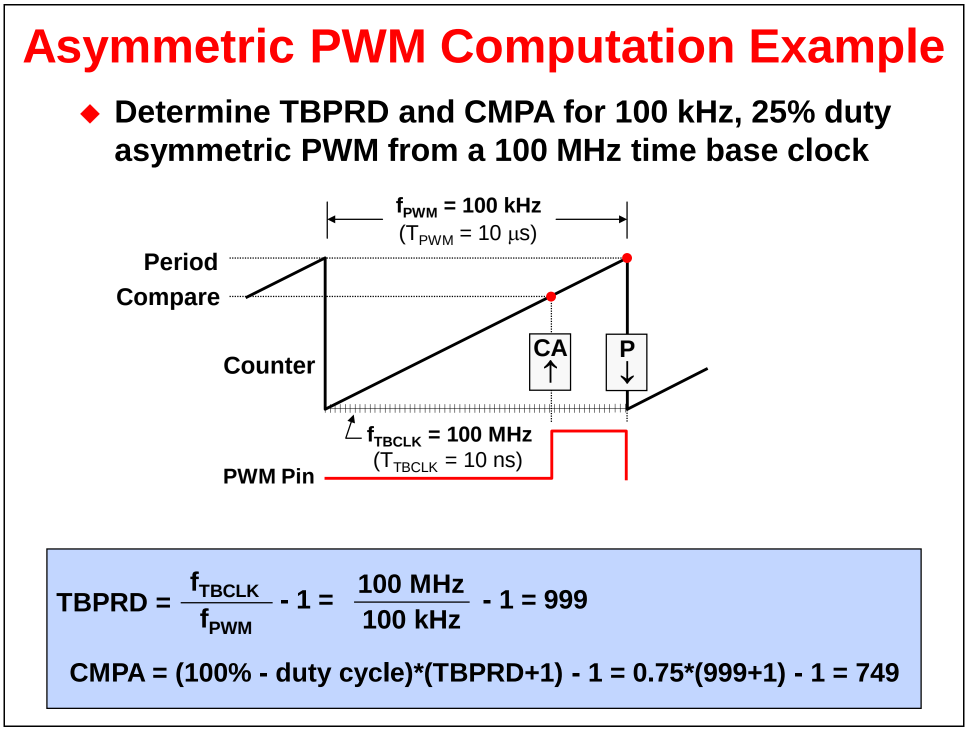


Symmetric PWM Computation Example

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

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

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



Asymmetric PWM Computation Example

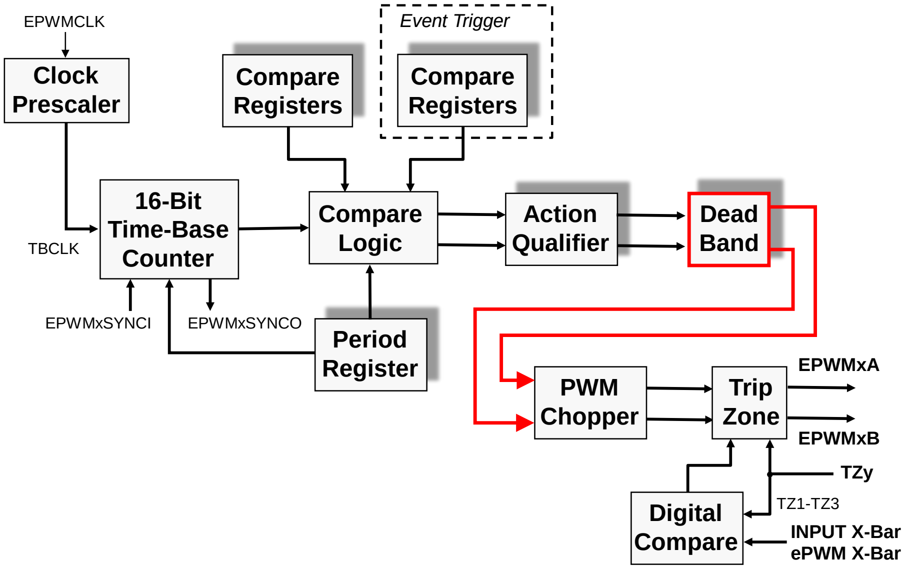
- Determine TBPRD and CMPA for 100 kHz, 25% duty asymmetric PWM from a 100 MHz time base clock

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

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

ePWM Dead-Band Sub-Module

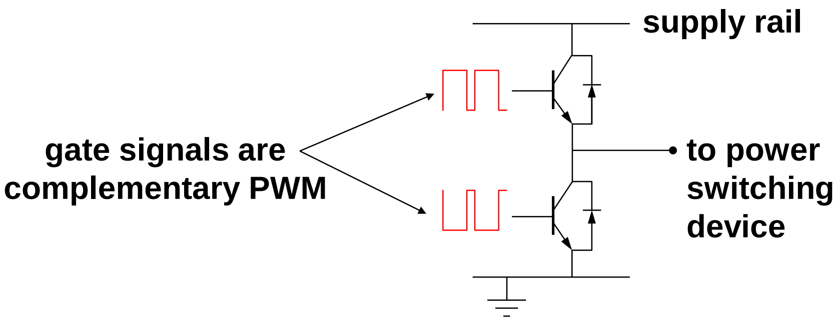
ePWM Dead-Band Sub-Module



EPWMCLK
Clock Prescaler
TBCLK
16-Bit Time-Base Counter
EPWMxSYNCI
EPWMxSYNCO
Compare Registers
Event Trigger
Compare Registers
Compare Logic
Period Register
Action Qualifier
Dead Band
PWM Chopper
Trip Zone
EPWMxA
EPWMxB
TZy
TZ1-TZ3
Digital Compare
INPUT X-Bar
ePWM X-Bar

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.

Motivation for Dead-Band

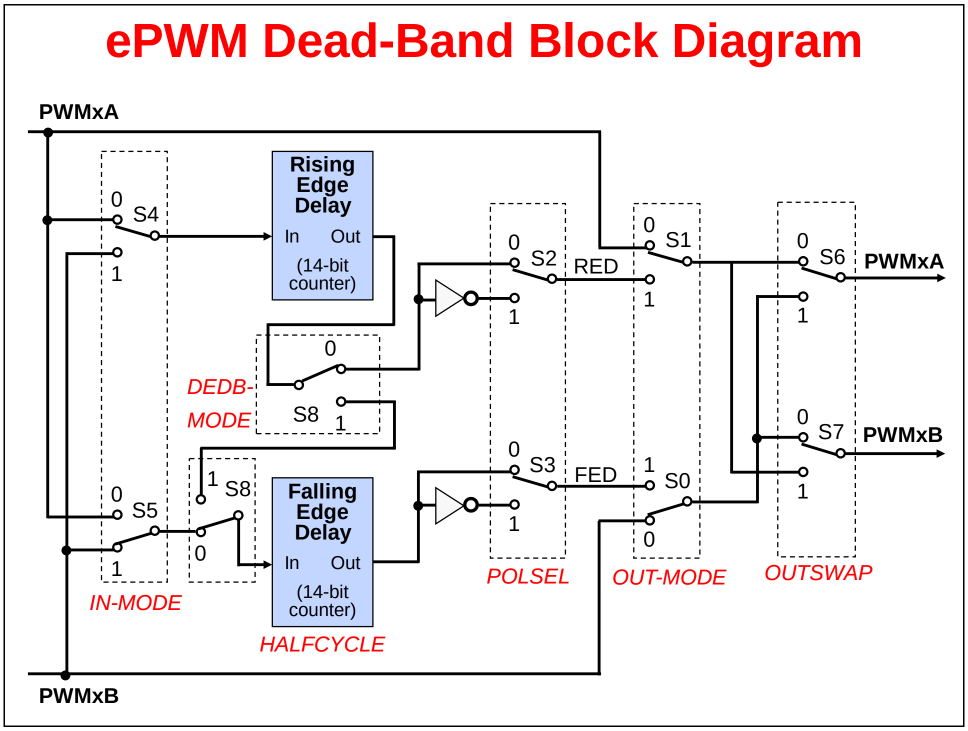


supply rail
gate signals are complementary PWM
to power switching device

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

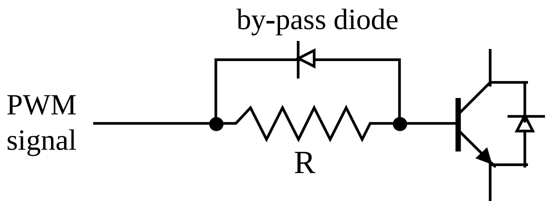
To explain further, power-switching devices turn on faster than they shut off. This issue would momentarily provide a path from supply rail to ground, giving us a short circuit. The dead-band sub-module alleviates this issue.

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



ePWM Dead-Band Block Diagram

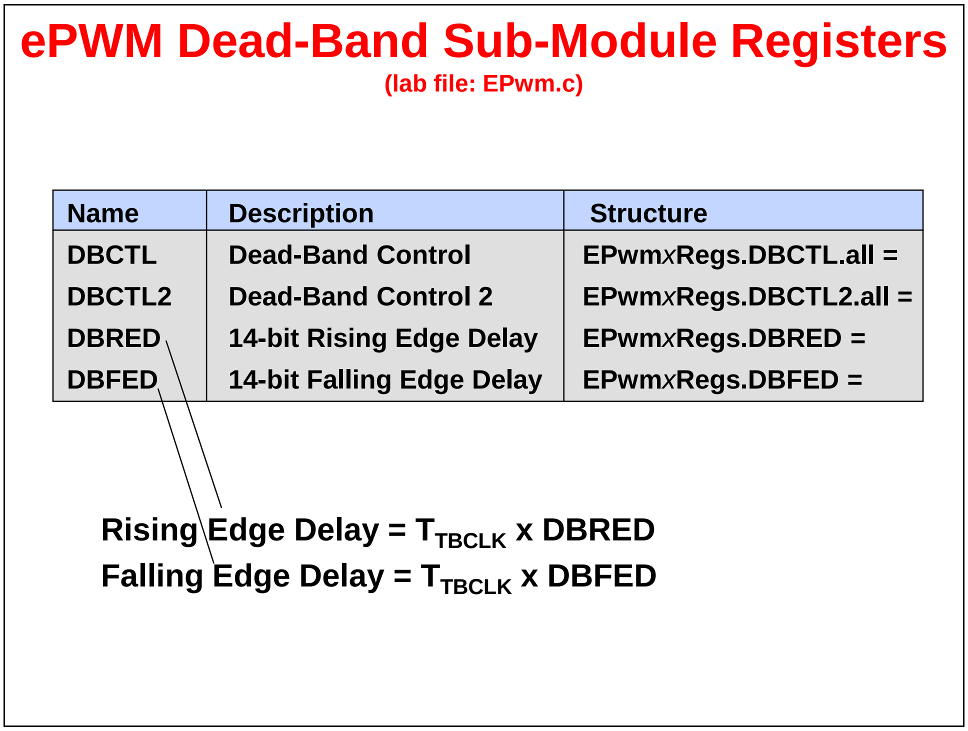
Two basic approaches exist for controlling shoot-through: modify the transistors, or modify the PWM gate signals controlling the transistors. In the first case, the opening time of the transistor gate must be increased so that it (slightly) exceeds the closing time. One way to accomplish this is by adding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



Shoot-through control via power circuit modification

The resistor acts to limit the current rise rate towards the gate during transistor opening, thus increasing the opening time. When closing the transistor however, current flows unimpeded from the gate via the by-pass diode and closing time is therefore not affected. While this passive approach offers an inexpensive solution that is independent of the control microprocessor, it is imprecise, the component parameters must be individually tailored to the power converter, and it cannot adapt to changing system conditions.

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



ePWM Dead-Band Sub-Module Registers

(lab file: EPwm.c)

Name	Description	Structure
DBCTL	Dead-Band Control	EPwm*x*Regs.DBCTL.all =
DBCTL2	Dead-Band Control 2	EPwm*x*Regs.DBCTL2.all =
DBRED	14-bit Rising Edge Delay	EPwm*x*Regs.DBRED =
DBFED	14-bit Falling Edge Delay	EPwm*x*Regs.DBFED =

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

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

ePWM Dead Band Control Registers

EPwmxRegs.DBCTL

15	14	13 - 12	11	10	9 - 8	7 - 6	5 - 4	3 - 2	1 - 0
HALF CYCLE	DEDB_ MODE	OUT SWAP	SHDW DBFED MODE	SHDW DBRED MODE	LOAD FED MODE	LOAD RED MODE	IN_ MODE	POL SEL	OUT_ MODE
	S8	S7 S6					S5 S4	S3 S2	S1 S0

0 = full cycle clocking
(TBCLK rate)
1 = half cycle clocking
(TBCLK*2 rate)

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

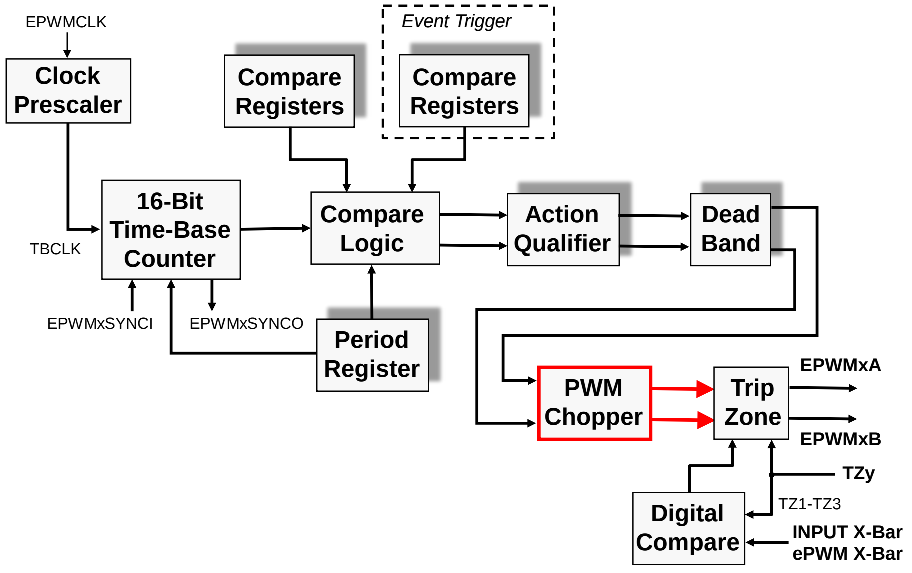
Shadow Load Mode
00 = load on CTR = 0
01 = load on CTR = PRD
10 = load on CTR = 0 or PRD
11 = freeze (no load possible)

EPwmxRegs.DBCTL2

15 - 3	2	1 - 0
reserved	SHDW DBCTL MODE	LOAD DBCTL MODE

ePWM Chopper Sub-Module

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.

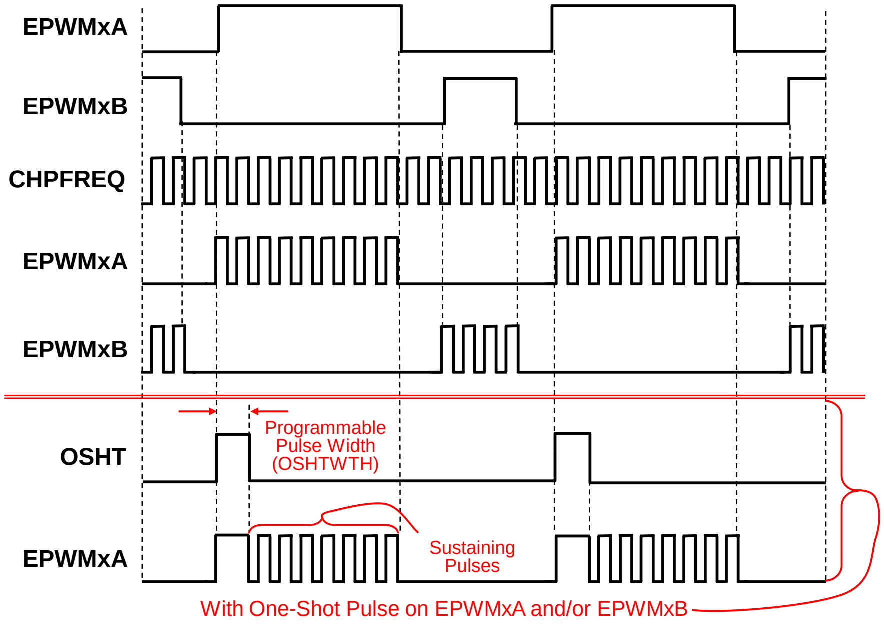
Programmable options are available to support the magnetic properties and characteristics of the transformer and associated circuitry.

Purpose of the PWM Chopper

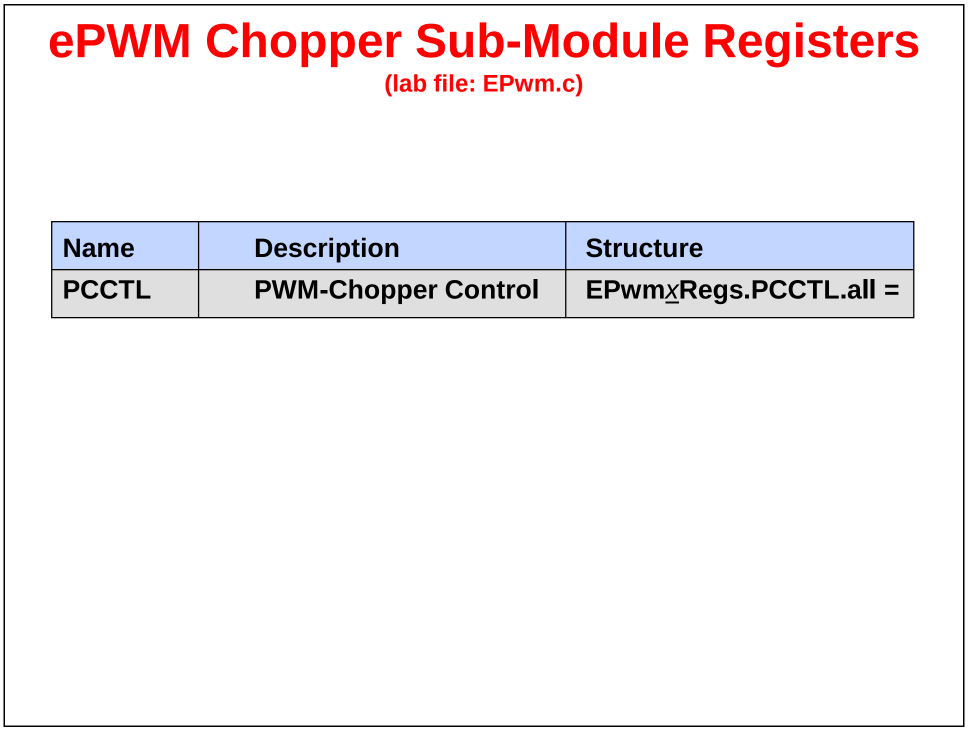
- **Allows a high frequency carrier signal to modulate the PWM waveform generated by the Action Qualifier and Dead-Band modules**
- **Used with pulse transformer-based gate drivers to control power switching elements**

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 Chopper Waveform



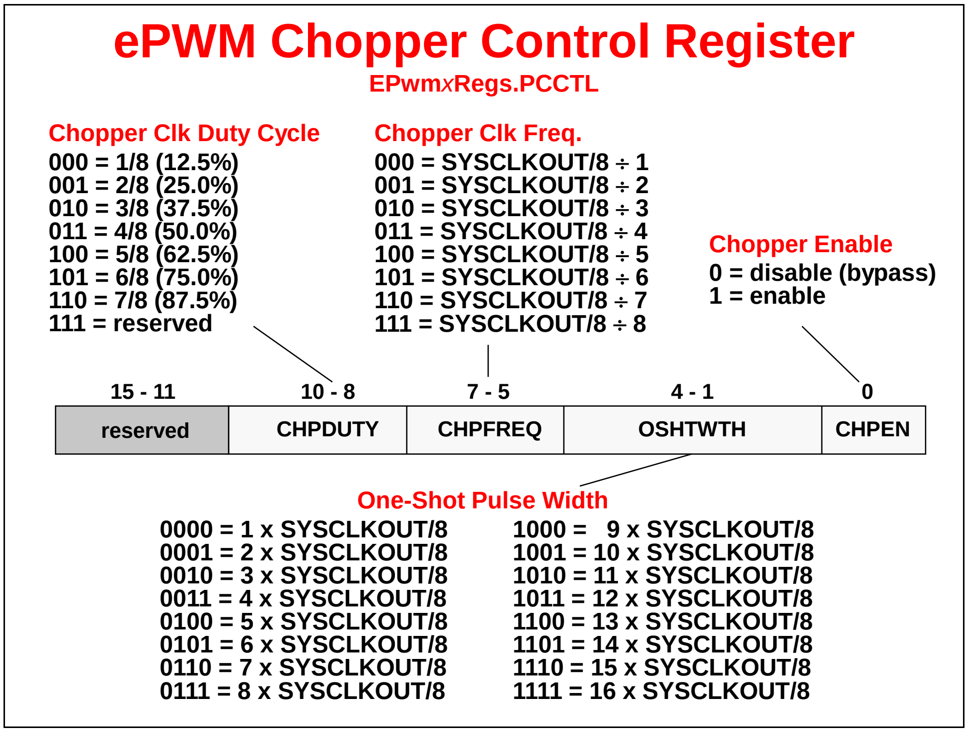
With One-Shot Pulse on EPWMxA and/or EPWMxB



ePWM Chopper Sub-Module Registers

(lab file: EPwm.c)

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



ePWM Chopper Control Register

EPwm*x*Regs.PCCTL

Chopper Clk Duty Cycle

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

Chopper Clk Freq.

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

Chopper Enable

0 = disable (bypass)
1 = enable

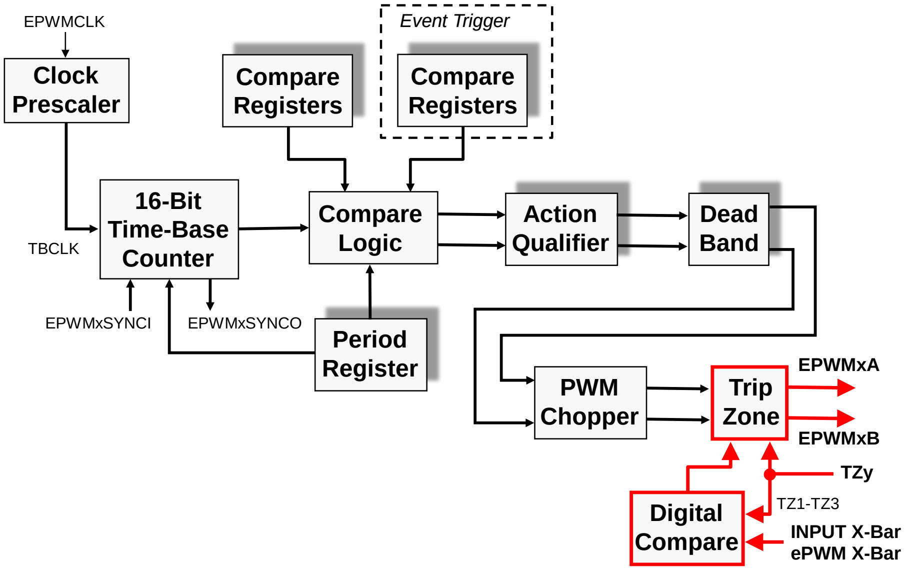
15 - 11	10 - 8	7 - 5	4 - 1	0
reserved	CHPDUTY	CHPFREQ	OSHTWTH	CHPEN

One-Shot Pulse Width

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

ePWM Trip-Zone and Digital Compare Sub-Modules

ePWM Trip-Zone and Digital Compare Sub-Modules

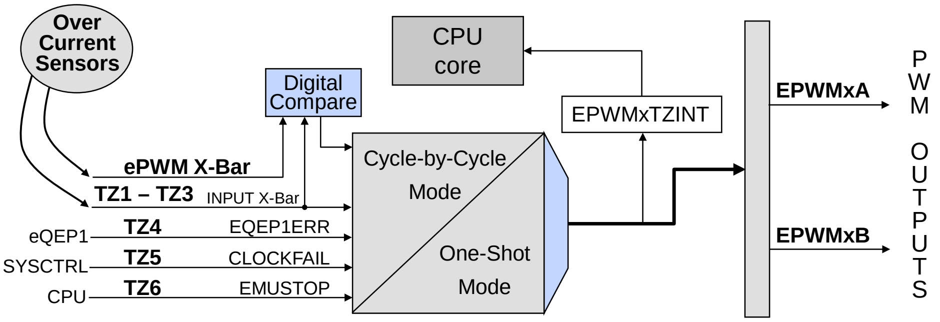


The trip zone and digital compare sub-modules provide a protection mechanism to protect the output pins from abnormalities, such as over-voltage, over-current, and excessive temperature rise.

Trip-Zone Features

- **Trip-Zone has a fast, clock independent logic path to high-impedance the EPWMxA/B output pins**
- **Interrupt latency may not protect hardware when responding to over current conditions or short-circuits through ISR software**
- **Supports: #1) one-shot trip for major short circuits or over current conditions**

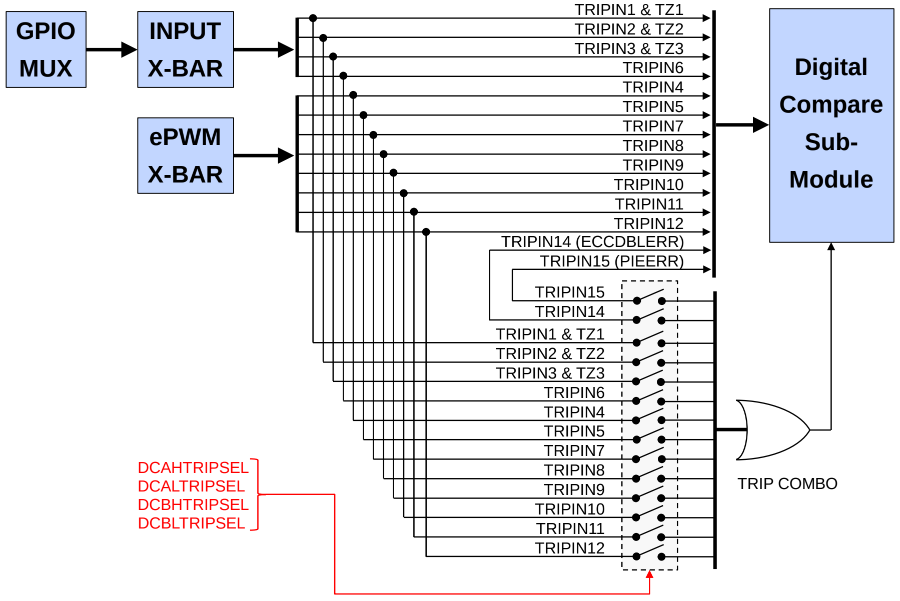
  #2) cycle-by-cycle trip for current limiting operation



The 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 inverted 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.

The power drive protection is a safety feature that is provided for the safe operation of systems such as power converters and motor drives. It can be used to inform the monitoring program of motor drive abnormalities such as over-voltage, over-current, and excessive temperature rise. If the power drive protection interrupt is unmasked, the PWM output pins will be put in a safe immediately after the pin is driven low. An interrupt will also be generated.

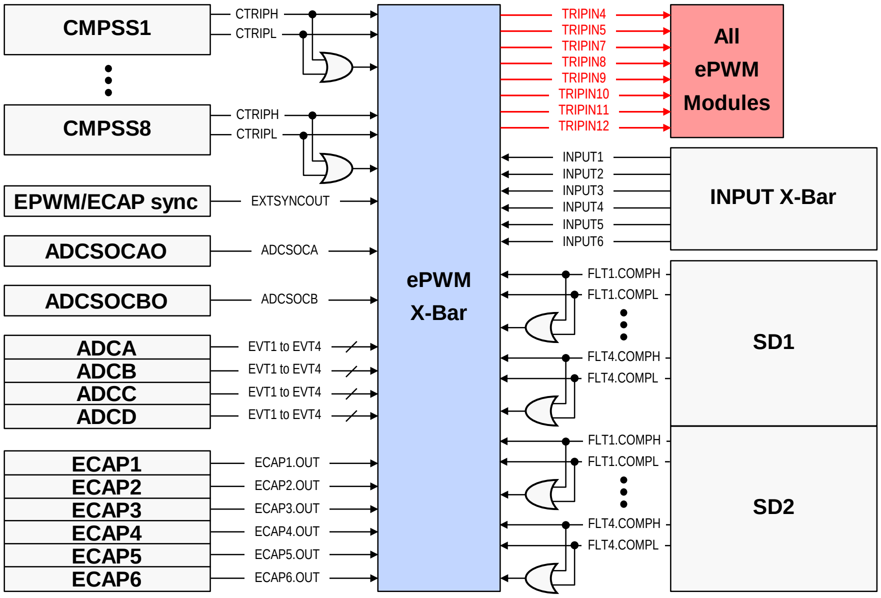
Digital Compare Trip Inputs



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.

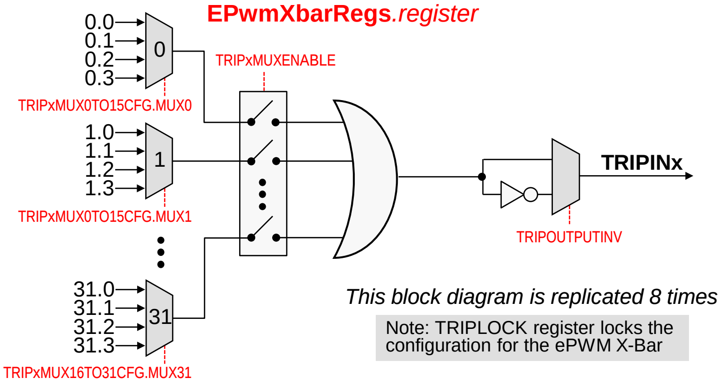
ePWM X-Bar



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.

ePWM X-Bar Architecture

EPwmXbarRegs.*register*



This block diagram is replicated 8 times

Note: TRIPLOCK register locks the configuration for the ePWM X-Bar

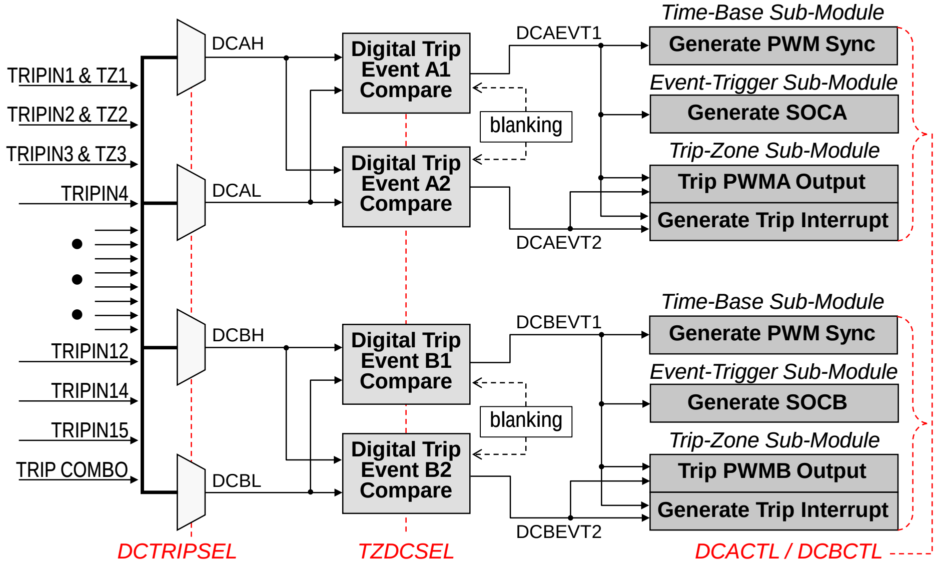
MUX	0	1	2	3
0	CMPSS1.CTRIPH	CMPSS1.CTRIPH_OR_CTRIPL	ADCAEVT1	ECAP1.OUT
1	CMPSS1.CTRIPL	INPUTXBAR1		ADCCEVT1
2	CMPSS2.CTRIPH	CMPSS2.CTRIPH_OR_CTRIPL	ADCAEVT2	ECAP2.OUT
3	CMPSS2.CTRIPL	INPUTXBAR2		ADCCEVT2
4	CMPSS3.CTRIPH	CMPSS3.CTRIPH_OR_CTRIPL	ADCAEVT3	ECAP3.OUT
5	CMPSS3.CTRIPL	INPUTXBAR3		ADCCEVT3
6	CMPSS4.CTRIPH	CMPSS4.CTRIPH_OR_CTRIPL	ADCAEVT4	ECAP4.OUT
7	CMPSS4.CTRIPL	INPUTXBAR4		ADCCEVT4
8	CMPSS5.CTRIPH	CMPSS5.CTRIPH_OR_CTRIPL	ADCBEVT1	ECAP5.OUT
9	CMPSS5.CTRIPL	INPUTXBAR5		ADCDEVT1
10	CMPSS6.CTRIPH	CMPSS6.CTRIPH_OR_CTRIPL	ADCBEVT2	ECAP6.OUT
11	CMPSS6.CTRIPL	INPUTXBAR6		ADCDEVT2
12	CMPSS7.CTRIPH	CMPSS7.CTRIPH_OR_CTRIPL	ADCBEVT3	
13	CMPSS7.CTRIPL	ADCSOCA		ADCDEVT3
14	CMPSS8.CTRIPH	CMPSS8.CTRIPH_OR_CTRIPL	ADCBEVT4	EXTSYNCOUT
15	CMPSS8.CTRIPL	ADCSOCB		ADCDEVT4

MUX	0	1	2	3
16	SD1FLT1.COMPH	SD1FLT1.COMPH_OR_COMPL		
17	SD1FLT1.COMPL			
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	SD2FLT1.COMPH	SD2FLT1.COMPH_OR_COMPL		
25	SD2FLT1.COMPL			
26	SD2FLT2.COMPH	SD2FLT2.COMPH_OR_COMPL		
27	SD2FLT2.COMPL			
28	SD2FLT3.COMPH	SD2FLT3.COMPH_OR_COMPL		
29	SD2FLT3.COMPL			
30	SD2FLT4.COMPH	SD2FLT4.COMPH_OR_COMPL		
31	SD2FLT4.COMPL			

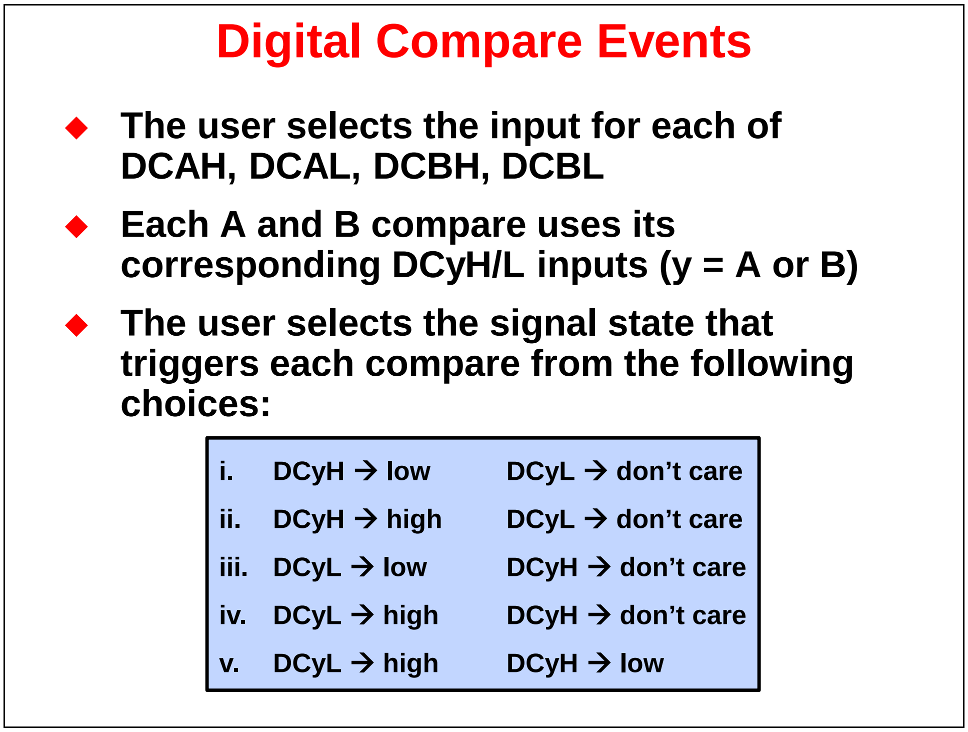
Purpose of the Digital Compare Sub-Module

- 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



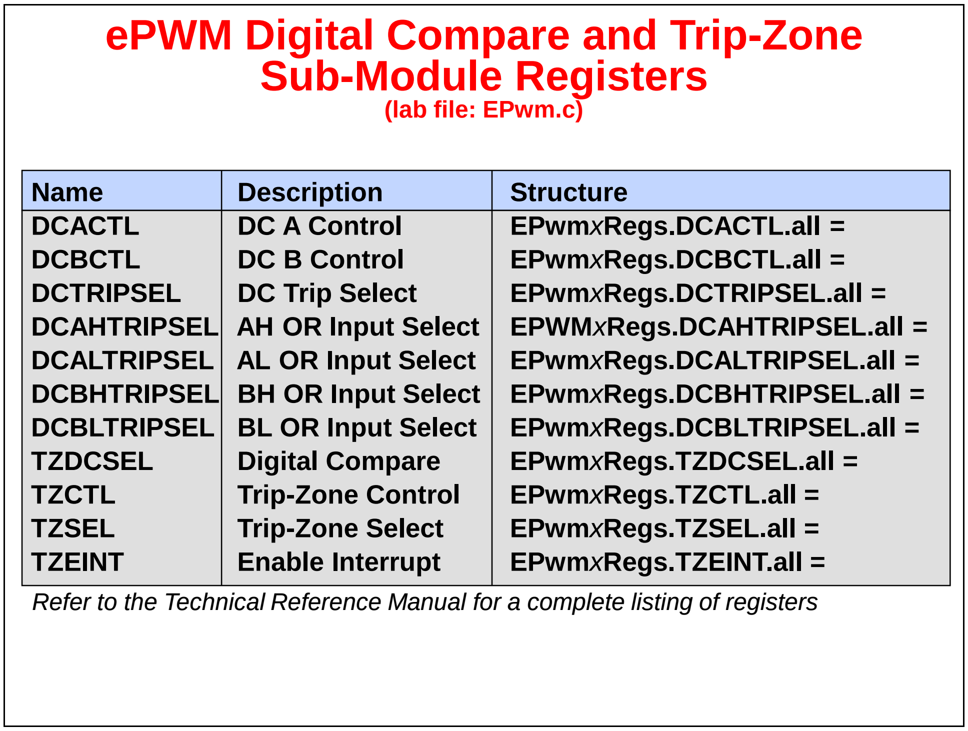
The digital-compare subsystem compares signals external to the ePWM module, such as a signal from the CMPSS analog comparators, to directly generate PWM events or actions which are then used by the trip-zone, time-base, and event-trigger submodules. These ‘compare’ events can trip the ePWM module, generate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

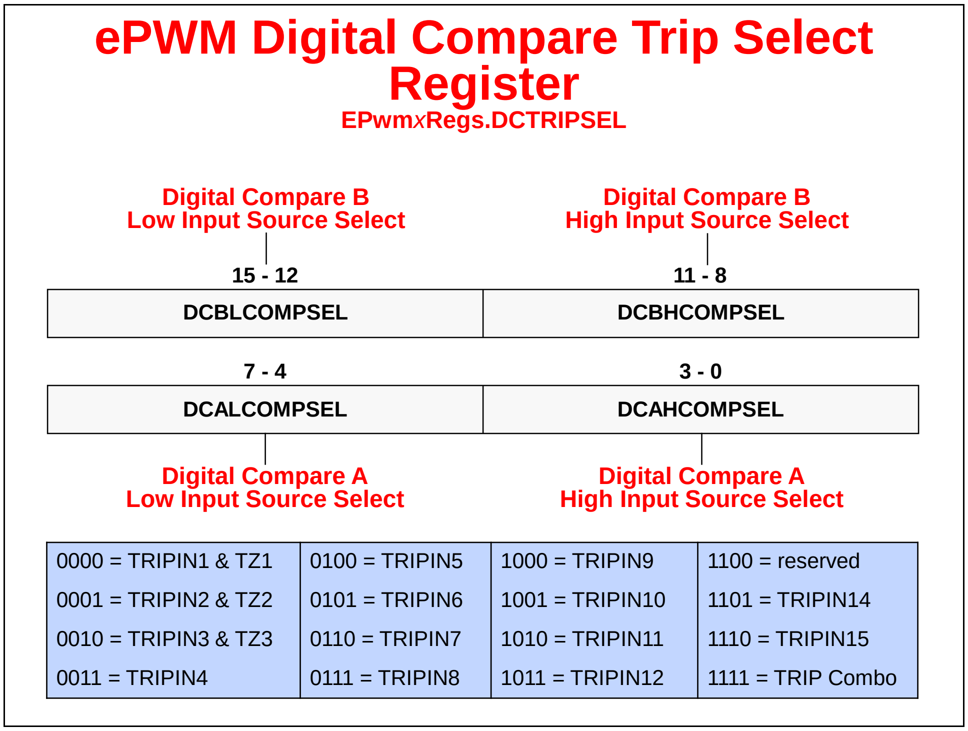


ePWM Digital Compare and Trip-Zone Sub-Module Registers

(lab file: EPwm.c)

Name	Description	Structure
DCACTL	DC A Control	EPwm*x*Regs.DCACTL.all =
DCBCTL	DC B Control	EPwm*x*Regs.DCBCTL.all =
DCTRIPSEL	DC Trip Select	EPwm*x*Regs.DCTRIPSEL.all =
DCAHTRIPSEL	AH OR Input Select	EPWM*x*Regs.DCAHTRIPSEL.all =
DCALTRIPSEL	AL OR Input Select	EPwm*x*Regs.DCALTRIPSEL.all =
DCBHTRIPSEL	BH OR Input Select	EPwm*x*Regs.DCBHTRIPSEL.all =
DCBLTRIPSEL	BL OR Input Select	EPwm*x*Regs.DCBLTRIPSEL.all =
TZDCSEL	Digital Compare	EPwm*x*Regs.TZDCSEL.all =
TZCTL	Trip-Zone Control	EPwm*x*Regs.TZCTL.all =
TZSEL	Trip-Zone Select	EPwm*x*Regs.TZSEL.all =
TZEINT	Enable Interrupt	EPwm*x*Regs.TZEINT.all =

Refer to the Technical Reference Manual for a complete listing of registers



ePWM Digital Compare Trip Select Register

EPwm*x*Regs.DCTRIPSEL

Digital Compare B
Low Input Source Select

Digital Compare B
High Input Source Select

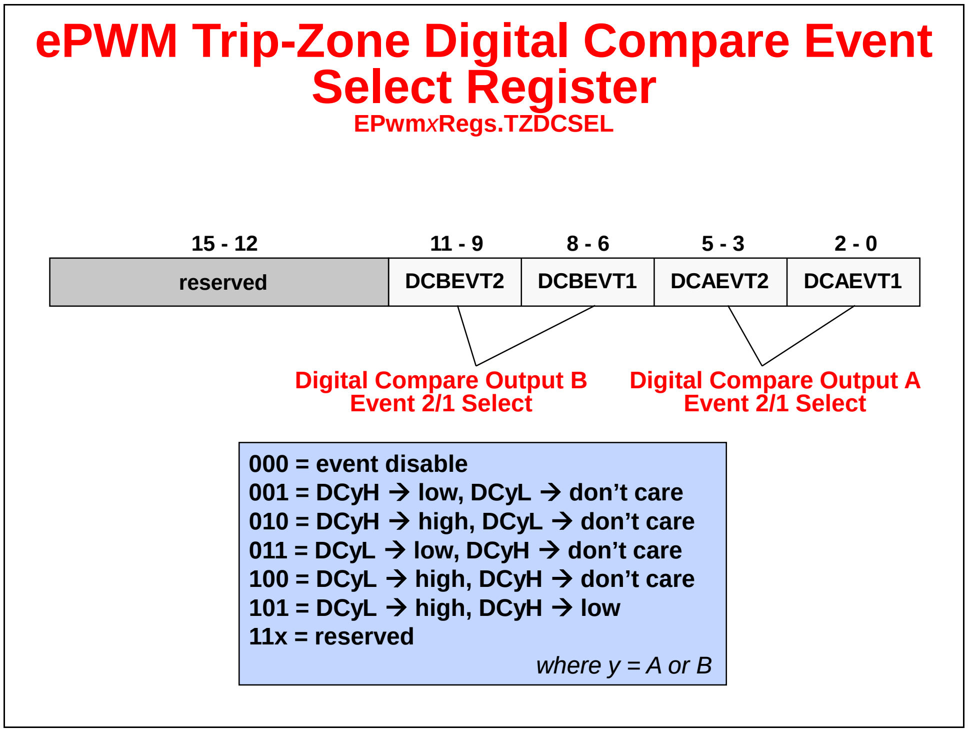
15 - 12	11 - 8
DCBLCOMPSEL	DCBHCOMPSEL

7 - 4	3 - 0
DCALCOMPSEL	DCAHCOMPSEL

Digital Compare A
Low Input Source Select

Digital Compare A
High Input Source Select

0000 = TRIPIN1 & TZ1	0100 = TRIPIN5	1000 = TRIPIN9	1100 = reserved
0001 = TRIPIN2 & TZ2	0101 = TRIPIN6	1001 = TRIPIN10	1101 = TRIPIN14
0010 = TRIPIN3 & TZ3	0110 = TRIPIN7	1010 = TRIPIN11	1110 = TRIPIN15
0011 = TRIPIN4	0111 = TRIPIN8	1011 = TRIPIN12	1111 = TRIP Combo



ePWM Trip-Zone Digital Compare Event Select Register

EPwm*x*Regs.TZDCSEL

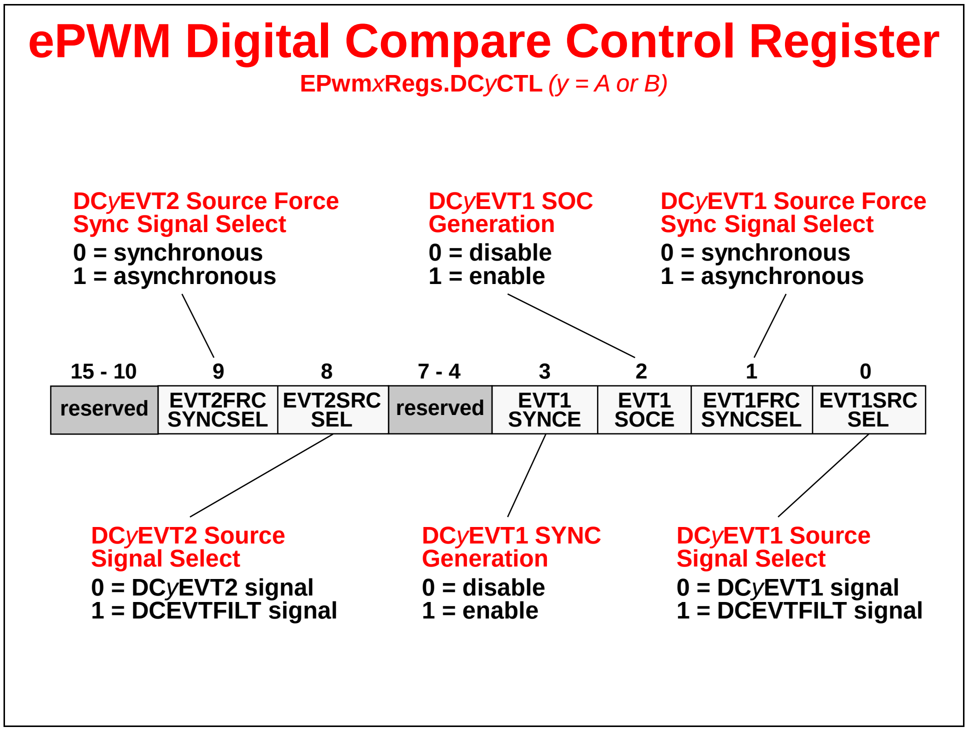
15 - 12	11 - 9	8 - 6	5 - 3	2 - 0
reserved	DCBEVT2	DCBEVT1	DCAEVT2	DCAEVT1

Digital Compare Output B
Event 2/1 Select

Digital Compare Output A
Event 2/1 Select

000 = event disable
001 = DCyH → low, DCyL → don’t care
010 = DCyH → high, DCyL → don’t care
011 = DCyL → low, DCyH → don’t care
100 = DCyL → high, DCyH → don’t care
101 = DCyL → high, DCyH → low
11x = reserved

where y = A or B



ePWM Digital Compare Control Register

EPwm*x*Regs.DC*y*CTL *(y = A or B)*

15 - 10	9	8	7 - 4	3	2	1	0
reserved	EVT2FRC SYNCSEL	EVT2SRC SEL	reserved	EVT1 SYNCE	EVT1 SOCE	EVT1FRC SYNCSEL	EVT1SRC SEL

DC*y*EVT2 Source Force Sync Signal Select
0 = synchronous
1 = asynchronous

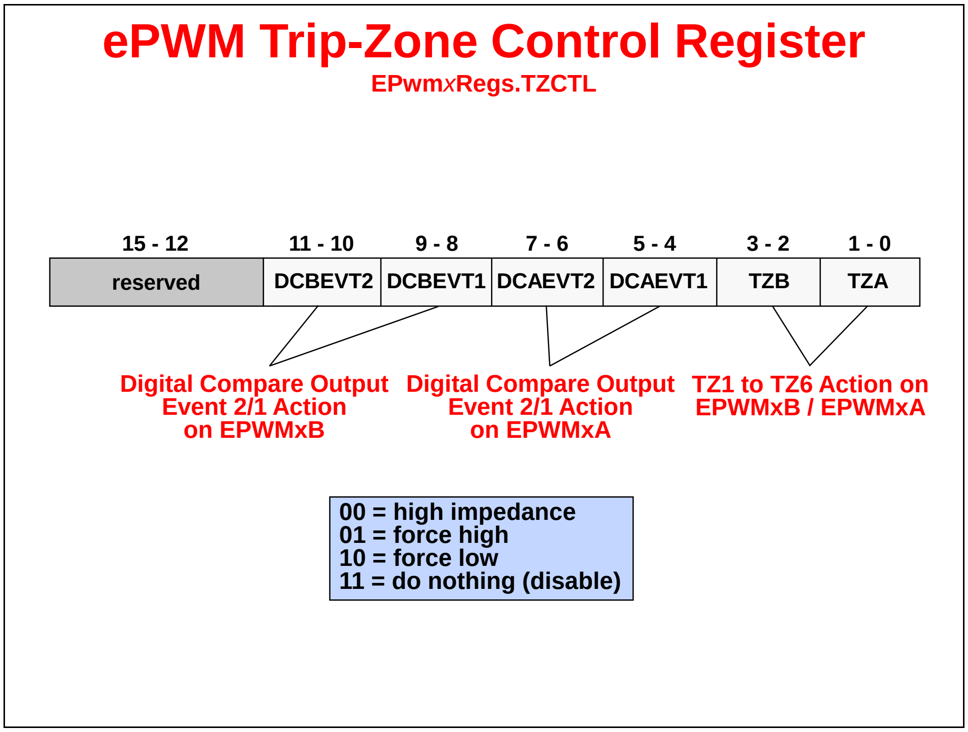
DC*y*EVT2 Source Signal Select
0 = DC*y*EVT2 signal
1 = DCEVTFILT signal

DC*y*EVT1 SYNC Generation
0 = disable
1 = enable

DC*y*EVT1 SOC Generation
0 = disable
1 = enable

DC*y*EVT1 Source Force Sync Signal Select
0 = synchronous
1 = asynchronous

DC*y*EVT1 Source Signal Select
0 = DC*y*EVT1 signal
1 = DCEVTFILT signal



ePWM Trip-Zone Control Register

EPwm*x*Regs.TZCTL

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

Digital Compare Output Event 2/1 Action on EPWMxB

Digital Compare Output Event 2/1 Action on EPWMxA

TZ1 to TZ6 Action on EPWMxB / EPWMxA

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

ePWM Trip-Zone Select Register

EPwm*x*Regs.TZSEL

One-Shot Trip Zone

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

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

15	14	13	12	11	10	9	8
DCBEVT1	DCAEVT1	OSHT6	OSHT5	OSHT4	OSHT3	OSHT2	OSHT1

7	6	5	4	3	2	1	0
DCBEVT2	DCAEVT2	CBC6	CBC5	CBC4	CBC3	CBC2	CBC1

Cycle-by-Cycle Trip Zone

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

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

ePWM Trip-Zone Enable Interrupt Register

EPwm*x*Regs.TZEINT

15 - 7	6	5	4	3	2	1	0
reserved	DCBEVT2	DCBEVT1	DCAEVT2	DCAEVT1	OST	CBC	reserved

Digital Compare Output B Event 2/1 Interrupt Enable

0 = disable
1 = enable

Digital Compare Output A Event 2/1 Interrupt Enable

0 = disable
1 = enable

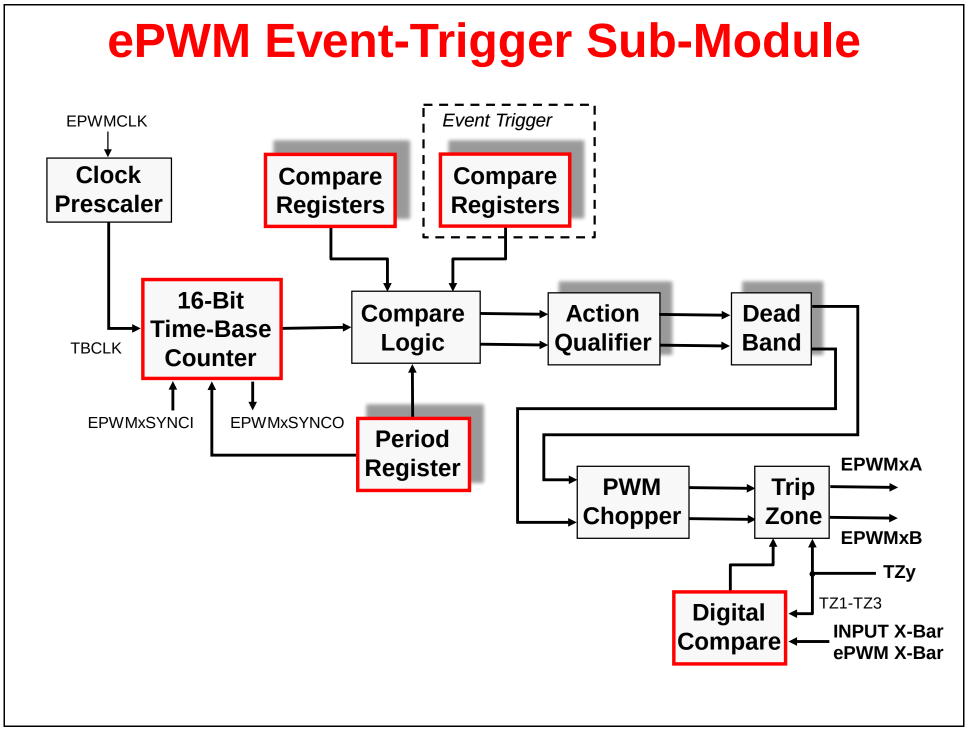
One-Shot Interrupt Enable

0 = disable
1 = enable

Cycle-by-Cycle Interrupt Enable

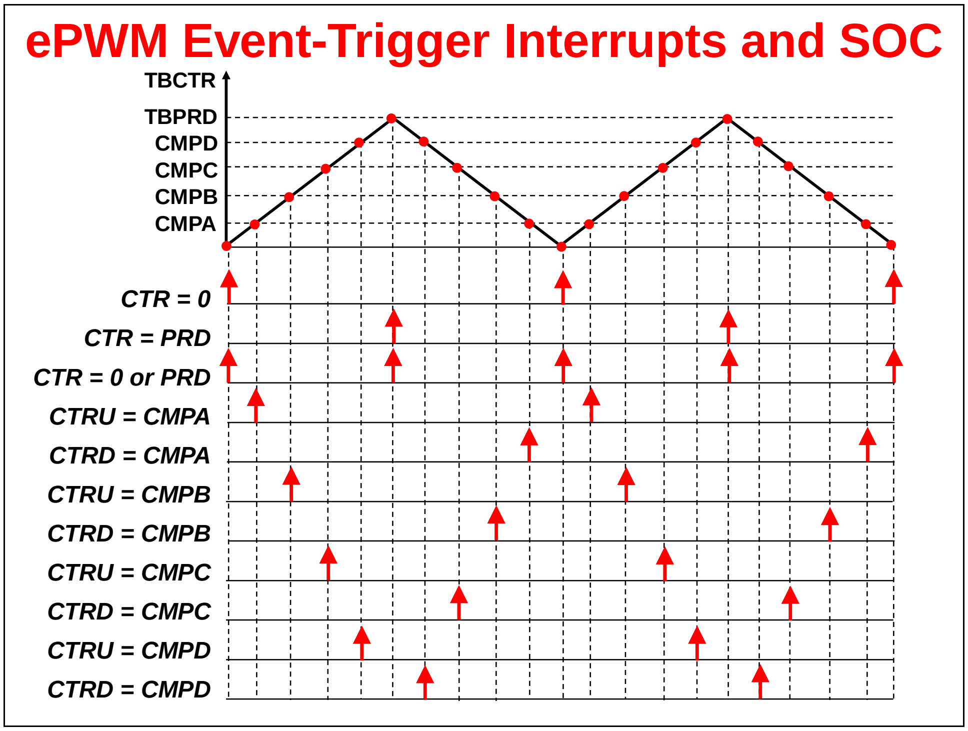
0 = disable
1 = enable

ePWM Event-Trigger Sub-Module



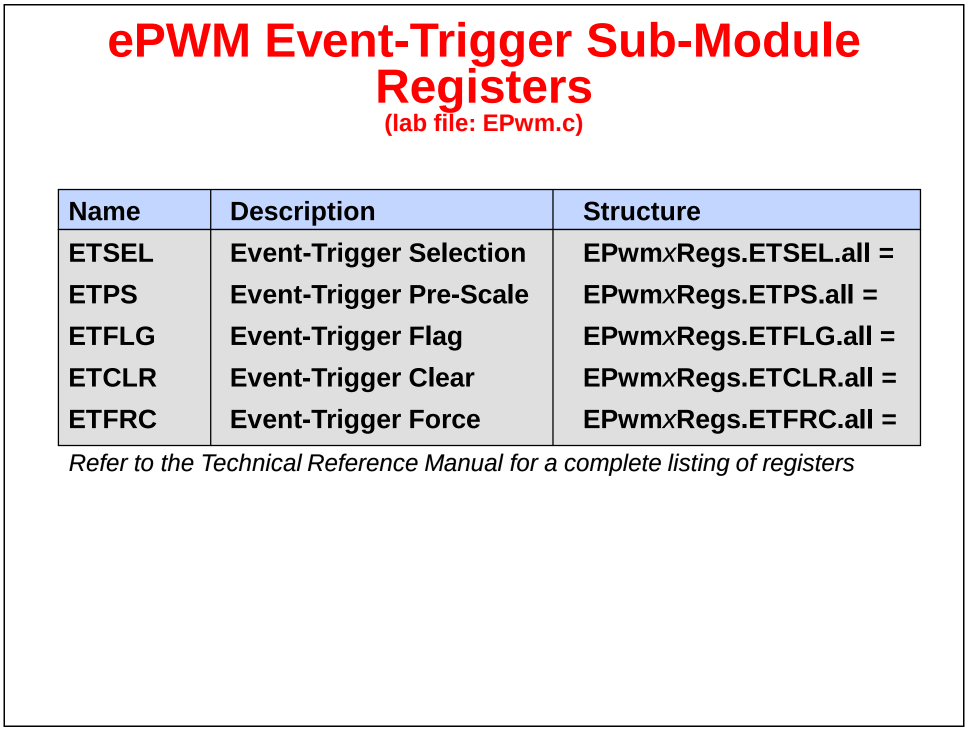
ePWM Event-Trigger Sub-Module

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.



ePWM Event-Trigger Interrupts and SOC

These event triggers can occur when the time-base counter equals zero, period, zero or period, 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.



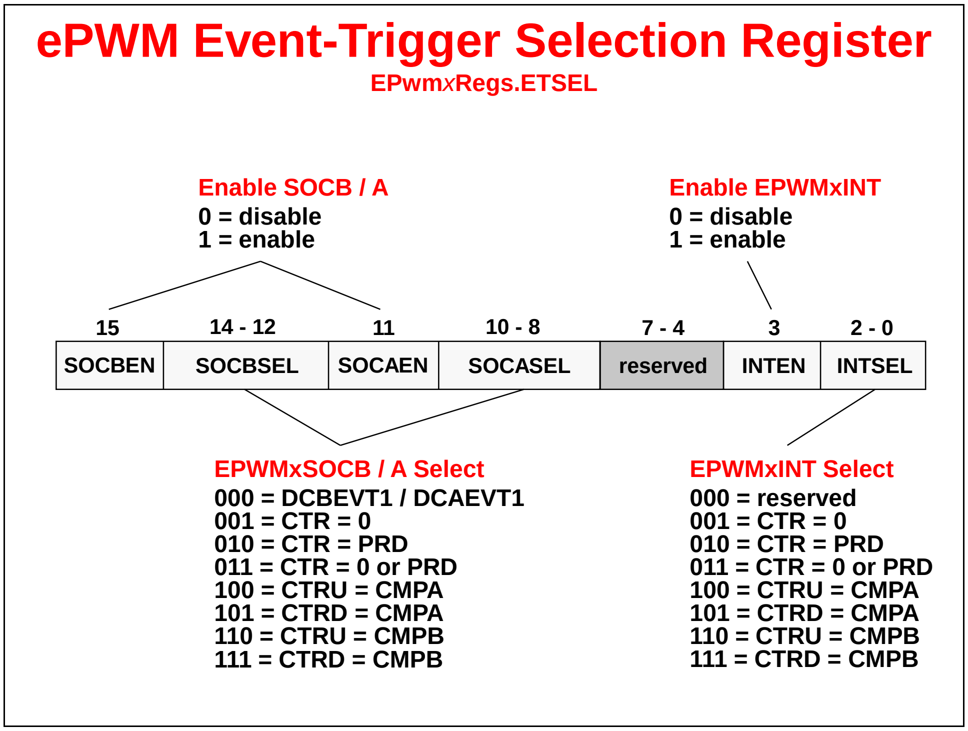
ePWM Event-Trigger Sub-Module Registers

(lab file: EPwm.c)

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

Refer to the Technical Reference Manual for a complete listing of registers

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.



ePWM Event-Trigger Selection Register

EPwm*x*Regs.ETSEL

Enable SOCB / A
0 = disable
1 = enable

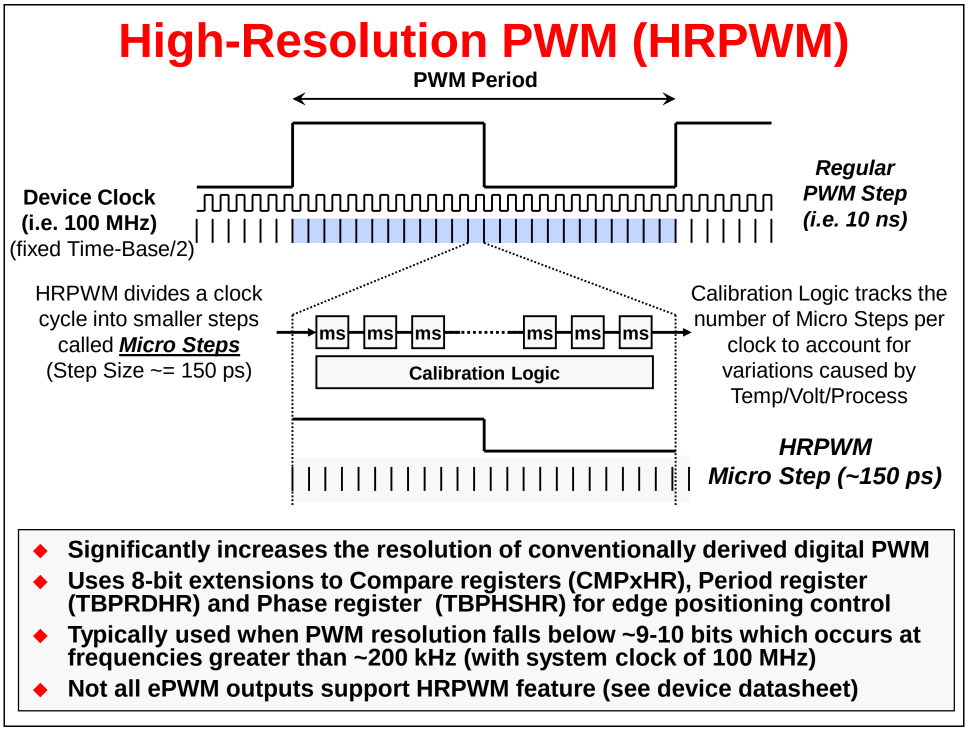
Enable EPWMxINT
0 = disable
1 = enable

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

EPWMxSOCB / A Select
000 = DCBEVT1 / DCAEVT1
001 = CTR = 0
010 = CTR = PRD
011 = CTR = 0 or PRD
100 = CTRU = CMPA
101 = CTRD = CMPA
110 = CTRU = CMPB
111 = CTRD = CMPB

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

High Resolution PWM (HRPWM)



High-Resolution PWM (HRPWM)

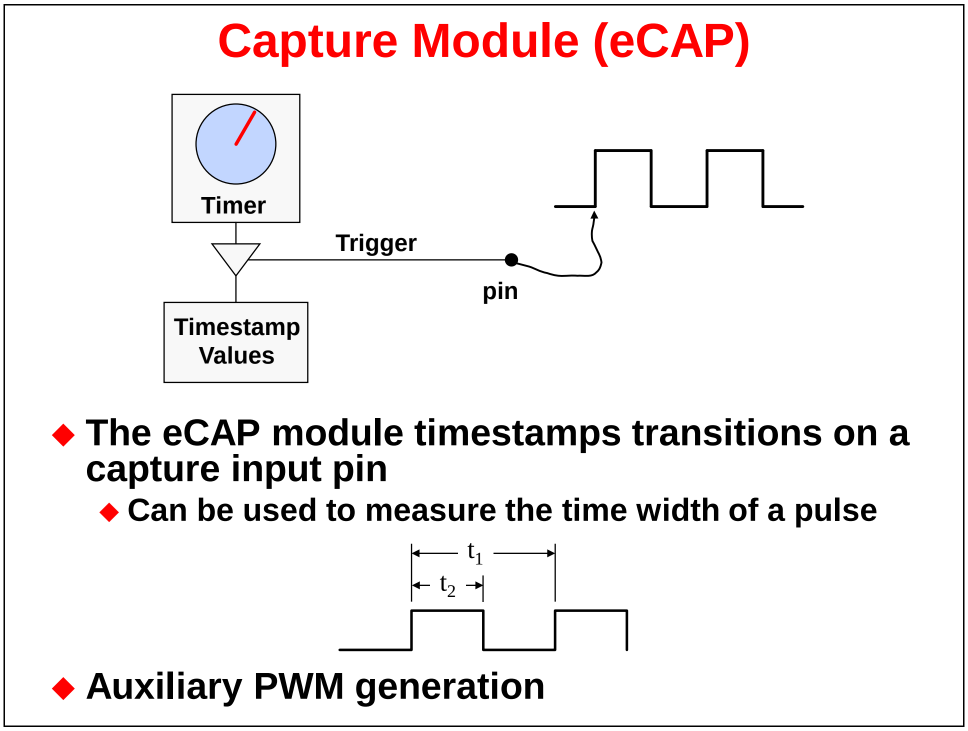
HRPWM divides a clock cycle into smaller steps called ***Micro Steps*** (Step Size ~= 150 ps)

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

- **Significantly increases the resolution of conventionally derived digital PWM**
- **Uses 8-bit extensions to Compare registers (CMPxHR), Period register (TBPRDHR) and Phase register (TBPHSHR) for edge positioning control**
- **Typically used when PWM resolution falls below ~9-10 bits which occurs at frequencies greater than ~200 kHz (with system clock of 100 MHz)**
- **Not all ePWM outputs support HRPWM feature (see device datasheet)**

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



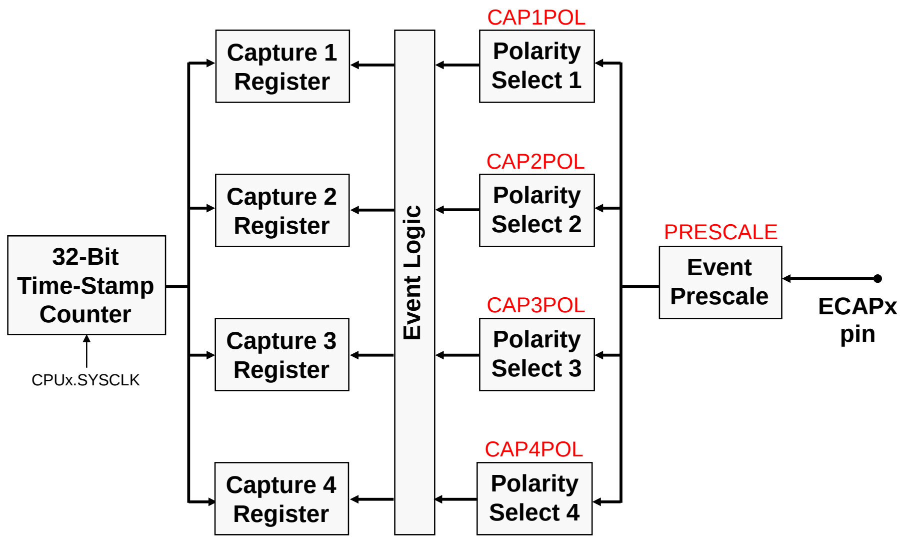
Capture Module (eCAP)

- **The eCAP module timestamps transitions on a capture input pin**
  - **Can be used to measure the time width of a pulse**
- **Auxiliary PWM generation**

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.

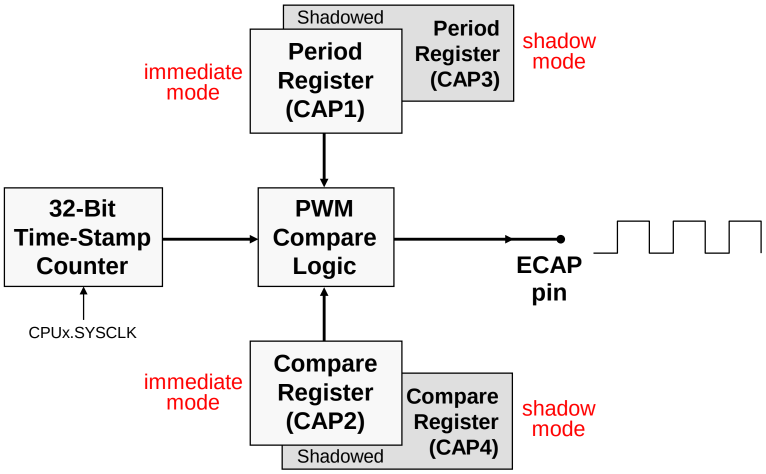
Capture units can be configured to trigger an A/D conversion that is synchronized with an external event. There are several potential advantages to using the capture for this function over the ADCSOC pin associated with the ADC module. First, the ADCSOC pin is level triggered, and therefore only low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

eCAP Module Block Diagram – Capture Mode



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

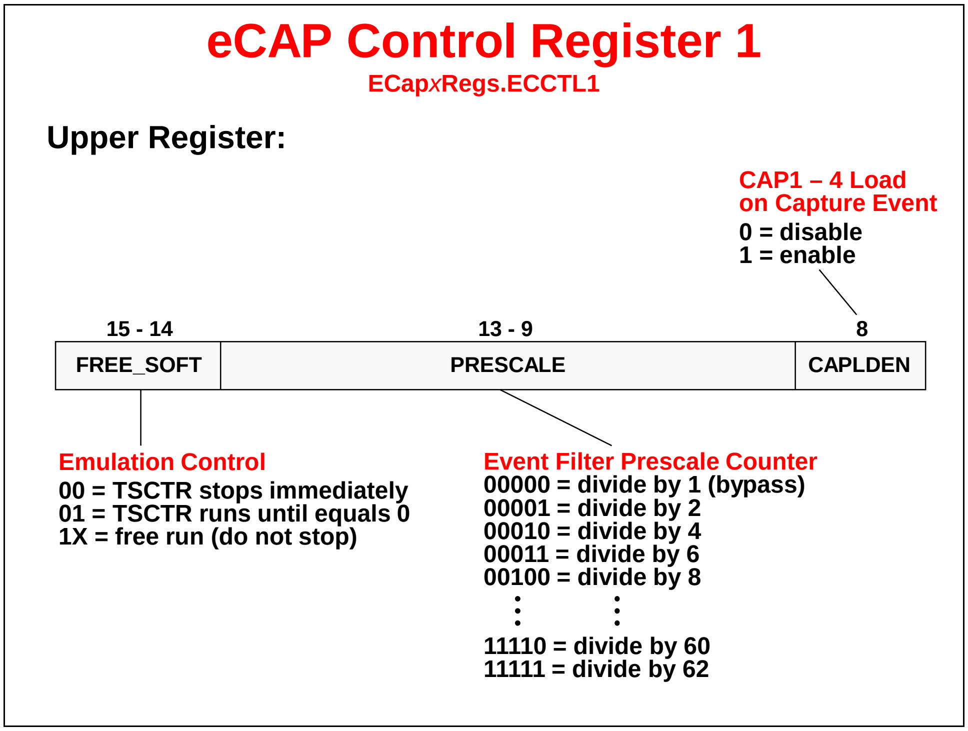


If the module is not used in capture mode, the eCAP module can be configured to operate as a single channel asymmetrical PWM module (i.e. time-base counter operates in count-up mode).

eCAP Module Registers

(lab file: ECap.c)

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



eCAP Control Register 1

ECCapxRegs.ECCTL1

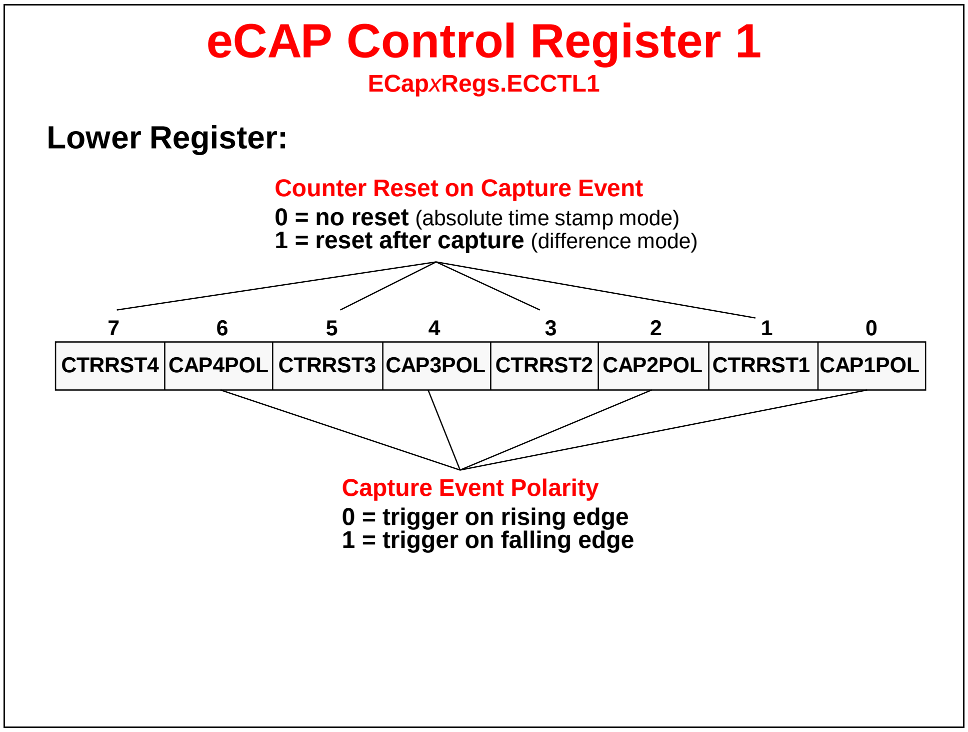
Upper Register:

15 - 14	13 - 9	8
FREE_SOFT	PRESCALE	CAPLDEN

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

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

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



eCAP Control Register 1

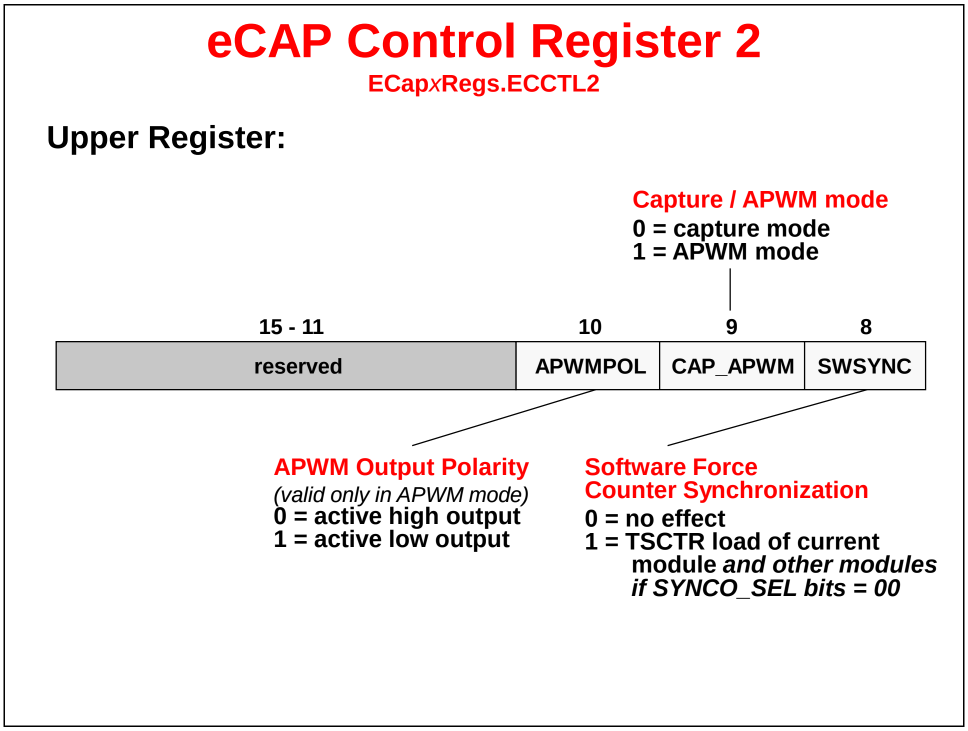
ECCapxRegs.ECCTL1

Lower Register:

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

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

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



eCAP Control Register 2

ECap*x*Regs.ECCTL2

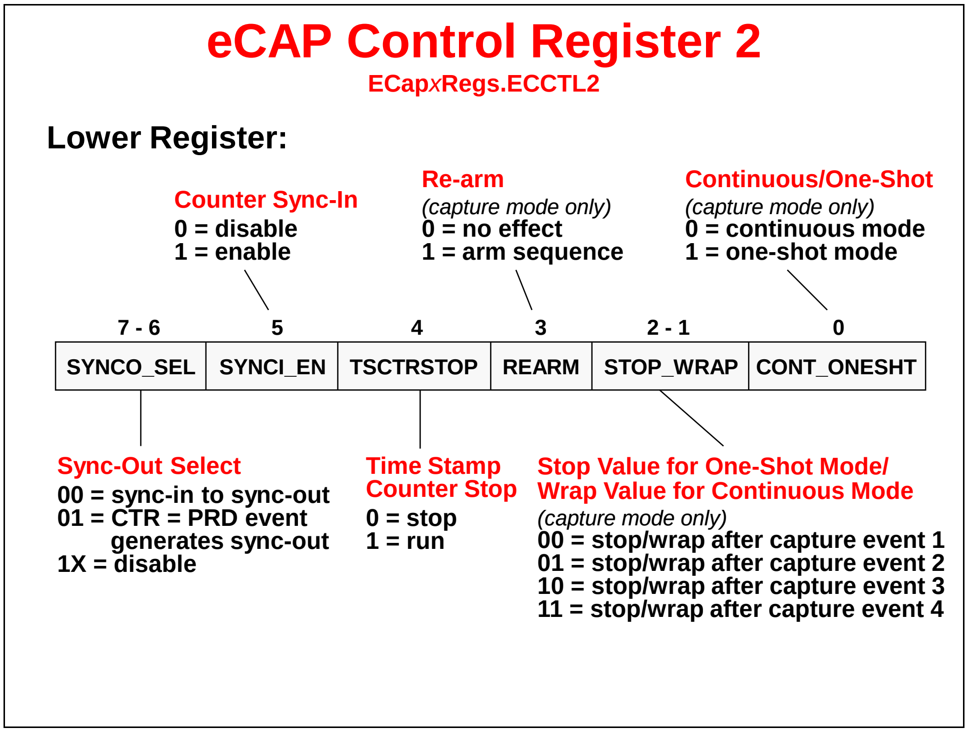
Upper Register:

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

15 - 11	10	9	8
reserved	APWMPOL	CAP_APWM	SWSYNC

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

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



eCAP Control Register 2

ECap*x*Regs.ECCTL2

Lower Register:

Counter Sync-In
0 = disable
1 = enable

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

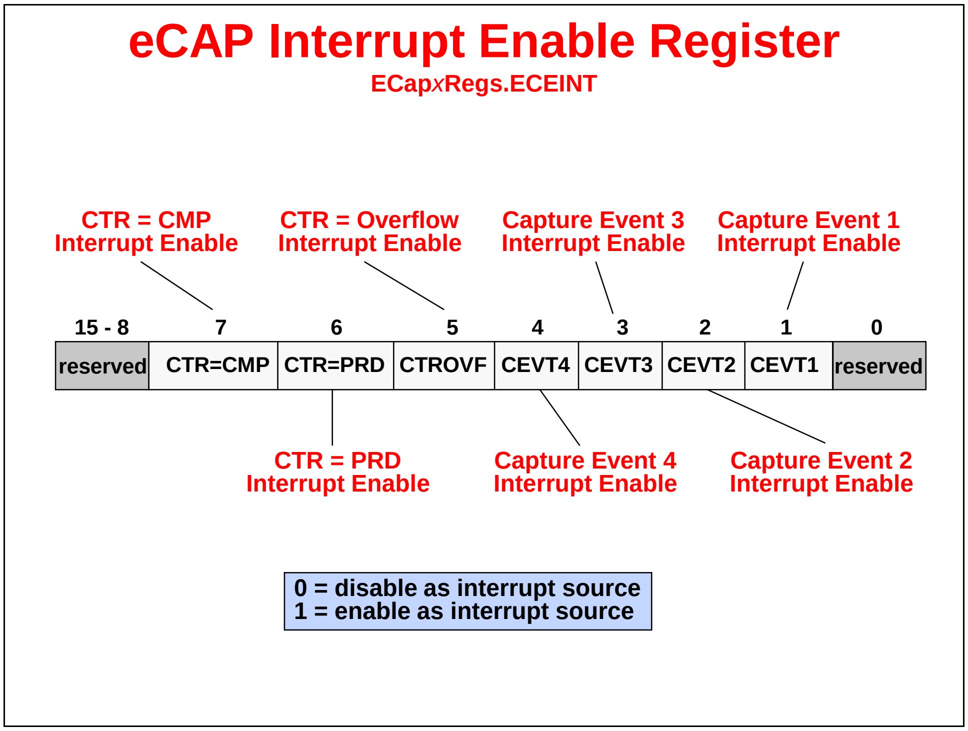
Continuous/One-Shot
(capture mode only)
0 = continuous mode
1 = one-shot mode

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

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

Time Stamp Counter Stop
0 = stop
1 = run

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



eCAP Interrupt Enable Register

ECapxRegs.ECEINT

15 - 8	7	6	5	4	3	2	1	0
reserved	CTR=CMP	CTR=PRD	CTROVF	CEVT4	CEVT3	CEVT2	CEVT1	reserved

CTR = CMP Interrupt Enable
CTR = Overflow Interrupt Enable
Capture Event 3 Interrupt Enable
Capture Event 1 Interrupt Enable
CTR = PRD Interrupt Enable
Capture Event 4 Interrupt Enable
Capture Event 2 Interrupt Enable

0 = disable as interrupt source
1 = enable as interrupt source

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

eQEP

What is an Incremental Quadrature Encoder?

A digital (angular) position sensor

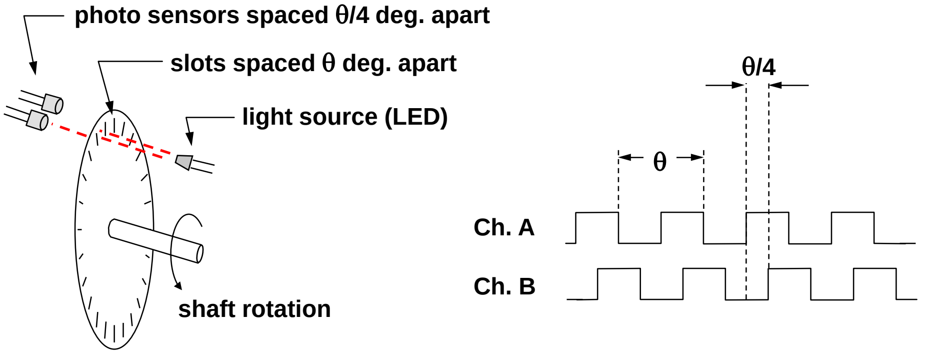


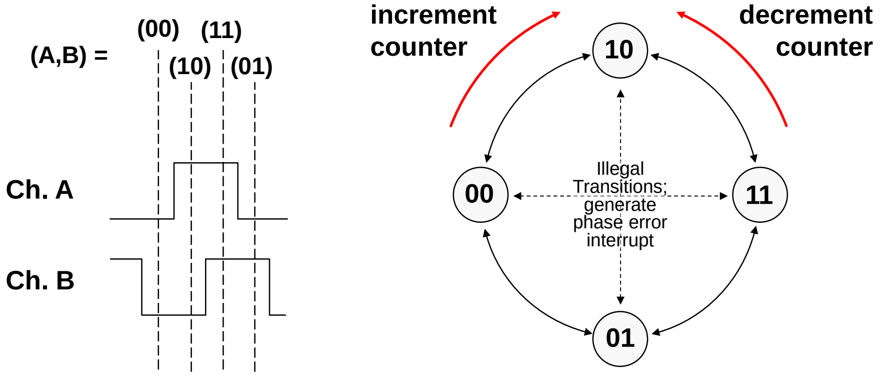
photo sensors spaced θ/4 deg. apart
slots spaced θ deg. apart
light source (LED)
shaft rotation
θ/4
θ
Ch. A
Ch. B

Incremental Optical Encoder **Quadrature Output from Photo Sensors**

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.

How is Position Determined from Quadrature Signals?

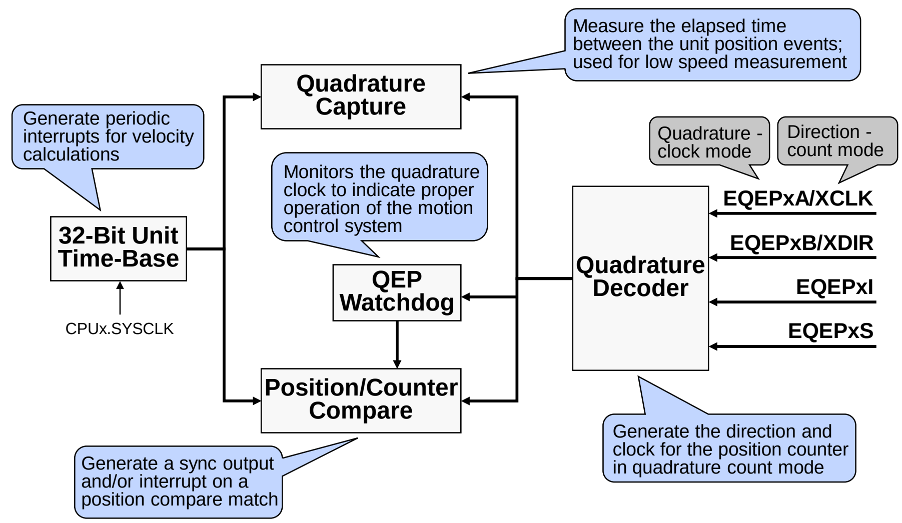
Position resolution is θ/4 degrees



Quadrature Decoder State Machine

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

eQEP Module Block Diagram



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

Generate periodic interrupts for velocity calculations

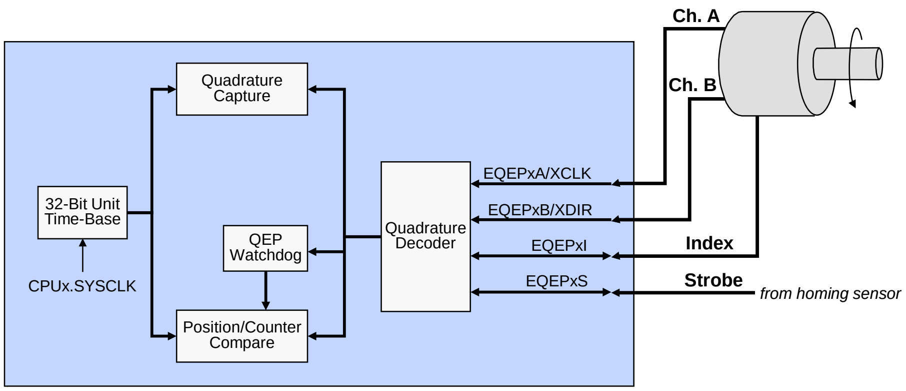
Monitors the quadrature clock to indicate proper operation of the motion control system

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

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

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.

eQEP Module Connections

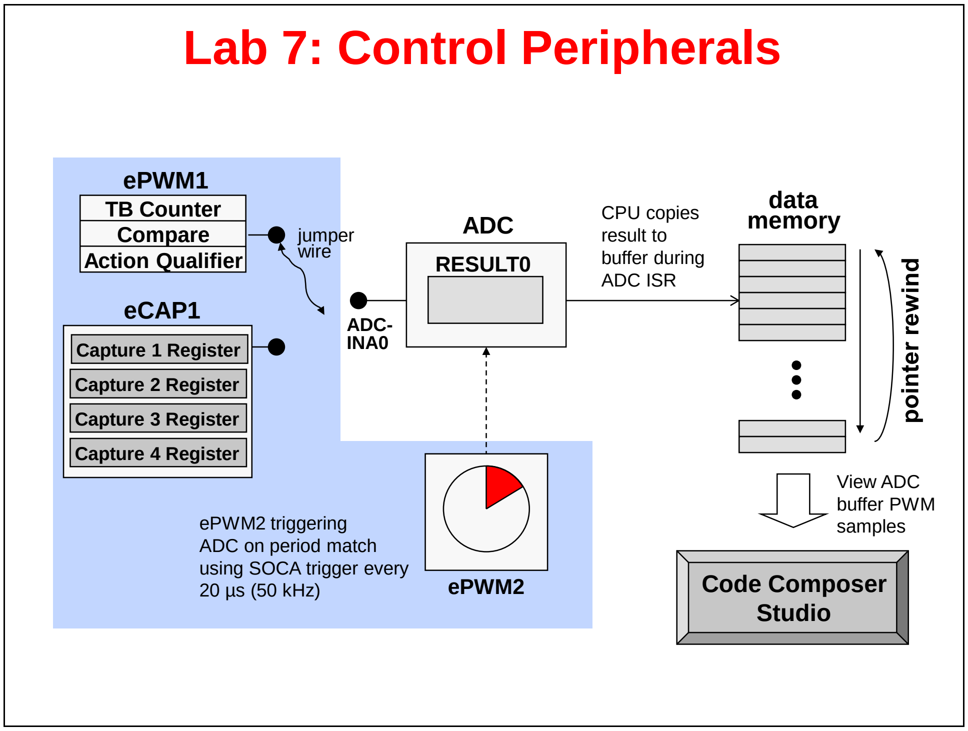


The above figure shows a summary of the connections to the eQEP module.

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 then click `Browse…` next to the "Select search-directory" box. Navigate to: `C:\C28x\Labs\Lab7\cpu01` and click `OK`. 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`
`CodeStartBranch.asm`
`Dac.c`
`DefaultIsr_7.c`
`DelayUs.asm`
`ECap.c`
`EPwm.c`
`F2837xD_Adc.c`
`F2837xD_GlobalVariableDefs.c`
`F2837xD_Headers_nonBIOS_cpu1.cmd`
`Gpio.c`
`Lab_5_6_7.cmd`
`Main_7.c`
`PieCtrl.c`
`PieVect.c`
`SineTable.c`
`SysCtrl.c`
`Watchdog.c`
`Xbar.c`

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

Setup Shared I/O and ePWM1

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

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

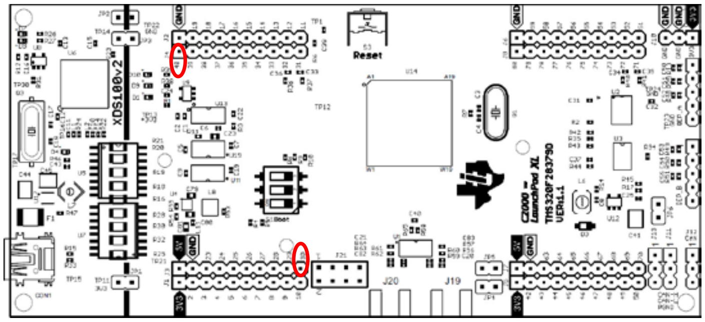
Build and Load

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

5. Click the "`Debug`" button (green bug). A Launching Debug Session window will open. Select only CPU1 to load the program on (i.e. ***uncheck*** CPU2), and then click `OK`. Then the CCS Debug perspective view should open, the program will load automatically, and you should now be at the start of `main()`. If the device has been power cycled since the last lab exercise, be sure to configure the boot mode to EMU_BOOT_SARAM using the Scripts menu.

Run the Code – PWM Waveform

6. Using a jumper wire, connect the PWM1A (header J4, pin #40) to ADCINA0 (header J3, pin #30) on the LaunchPad. Refer to the following diagram for the pins that need to be connected.



7. 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) using the Script function: `Scripts` → `Realtime Emulation Control` → `Run_Realtime_with_Reset`. Watch the window update. Verify that the ADC result buffer contains the updated values.

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

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

Select OK to save the graph options.

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

Frequency Domain Graphing Feature of Code Composer Studio

11. Code Composer Studio also has the ability to make frequency domain plots. It does this by using the PC to perform a Fast Fourier Transform (FFT) of the DSP data. Let's make

a frequency domain plot of the contents in the ADC results buffer (i.e. the PWM waveform).

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

Acquisition Buffer Size	50
DSP Data Type	16-bit unsigned integer
Sampling Rate (Hz)	50000
Start Address	AdcBuf
Data Plot Style	Bar
FFT Order	10

Select `OK` to save the graph options.

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

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

Setup eCAP1 to Measure Width of Pulse

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

14. Add (copy) `ECap.c` to the project from `C:\C28x\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. Edit `Xbar.c` and adjust the input selection register INPUT7SELECT for GPIO24 (header J4, pin #34) to feed the eCAP1 function. Simply set the register to 24.

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

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

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

PIE group #:___________ # within group:__________

This information will be used in the next step.

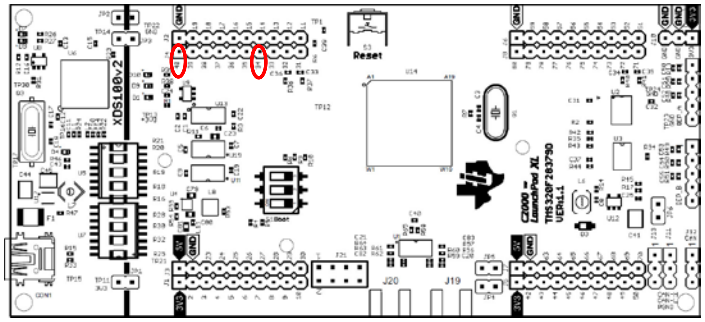
20. Modify the end of ECap.c to do the following:
    - Enable the "ECAP1" interrupt in the PIE (Hint: use the PieCtrlRegs structure)
    - Enable the appropriate core interrupt in the IER register

Build and Load

21. Save all changes to the files and 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 PWM1A (header J4, pin #40) to ECAP1 (header J4, pin #34, 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) by using the Script function: `Scripts` → `Realtime Emulation Control` → `Run_Realtime_with_Reset`. Notice the values for *PwmDuty* and *PwmPeriod*.

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

<u>Questions:</u>

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

Optional Exercise – Modulate the PWM Waveform

If you finish early, you might want to experiment with the code by observing the effects of changing the ePWM1 CMPA register using real-time emulation. Be sure that the jumper wire is connecting PWM1A (header J4, pin #40) to ADCINA0 (header J3, pin #30), and the Single Time graph is displayed. The graph must be enabled for continuous refresh.

a) Run the code in real-time mode.
b) Open an Expressions window to the EPwm1Regs.CMPA register – in EPwm.c highlight the "EPwm1Regs" structure and right click, then select Add Watch Expression… and then OK.
c) In the Expressions window open "EPwm1Regs", then open "CMPA" and open "bit".
d) The Expressions window must be enabled for continuous refresh.
e) Under "bit" change the "CMPA" 18750 value (within a range of 2500 and 22500).
f) Notice the effect on the PWM waveform in the graph.

You have just modulated the PWM waveform by manually changing the CMPA value. Next, we will modulate the PWM automatically by having the ADC ISR change the CMPA value.

a) 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.
b) In Main.c add "PWM_MODULATE" to the Expressions window using the same procedure above.
c) Then with the code running in real-time mode, change the "PWM_MODULATE" from 0 to 1 and observe the PWM waveform in the graph. Also, in the Expressions window notice the CMPA value being updated.

(If you do not have time to work on this optional exercise, you may want to try this later).

Terminate Debug Session and Close Project

27. 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.

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

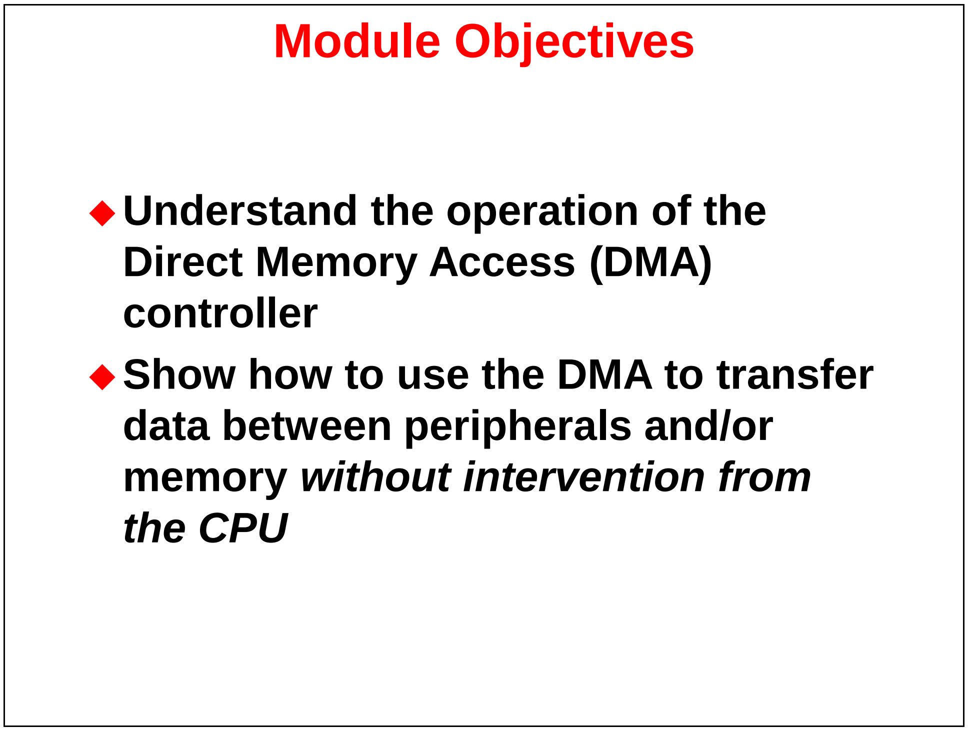
End of Exercise

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*

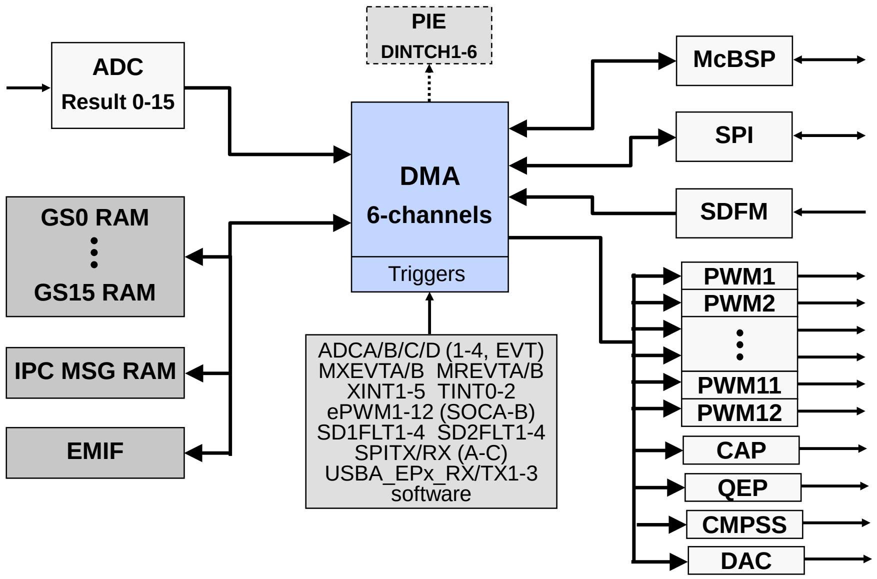
The DMA module provides a hardware method of transferring data between peripherals and/or memory without intervention from the CPU, effectively freeing up the CPU for other functions. Each CPU subsystem has its own DMA and using the DMA is ideal when an application requires a significant amount of time spent moving large amounts of data from off-chip memory 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 G15, IPC RAM, EMIF, transfer to or from the McBSP and SPI, 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

Chapter Topics

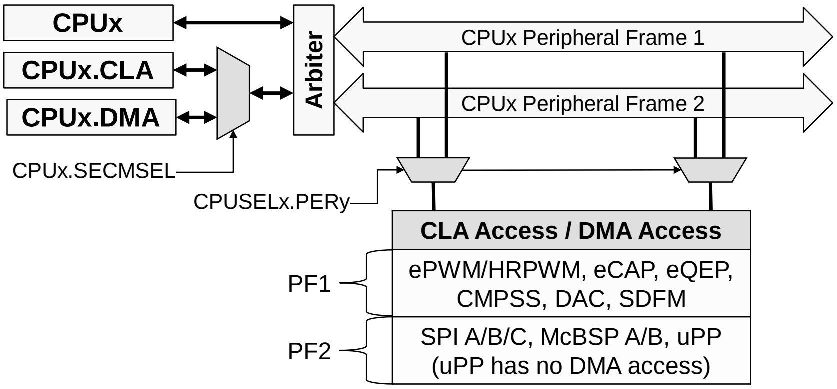
Direct Memory Access (DMA)

DMA Triggers, Sources, and Destinations



DMA / CLA Common Peripheral Access

Common peripherals can be accessed by the CPU and either DMA or CLA



CpuSysRegs.SECMSEL

15 - 4	3 - 2	1 - 0
reserved	PF2SEL	PF1SEL

x0 = connected to CLA *
x1 = connected to DMA

Note: CPUSELx bit associated with each peripheral defines if the peripheral is connected to CPU1 or CPU2

* Default (lock bit protected)

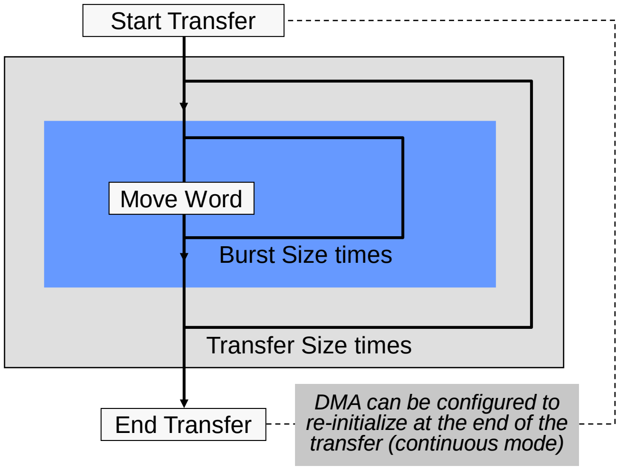
Basic Operation

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



DMA can be configured to re-initialize at the end of the transfer (continuous mode)

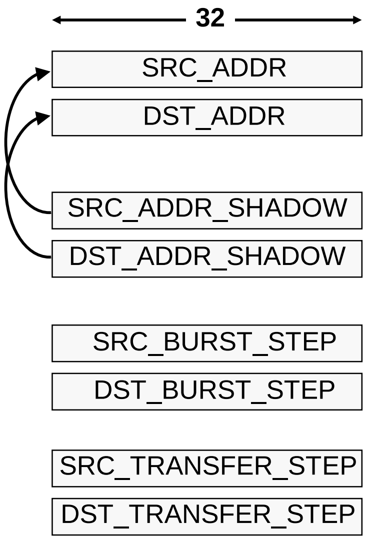
Basic Address Control Registers

Active pointers

Pointer shadow registers copied to active pointers at start of transfer

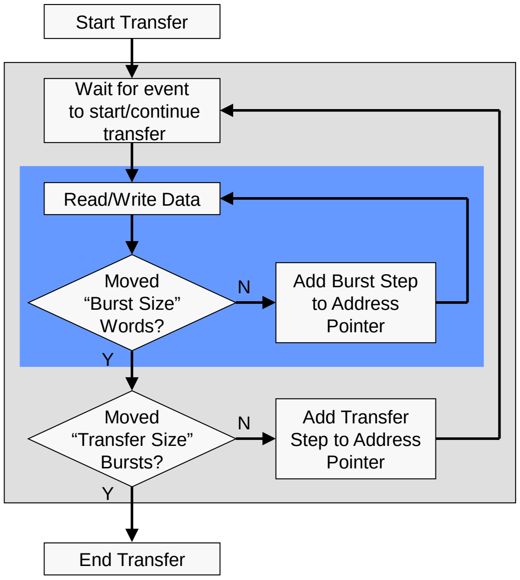
Signed value added to active pointer after each word

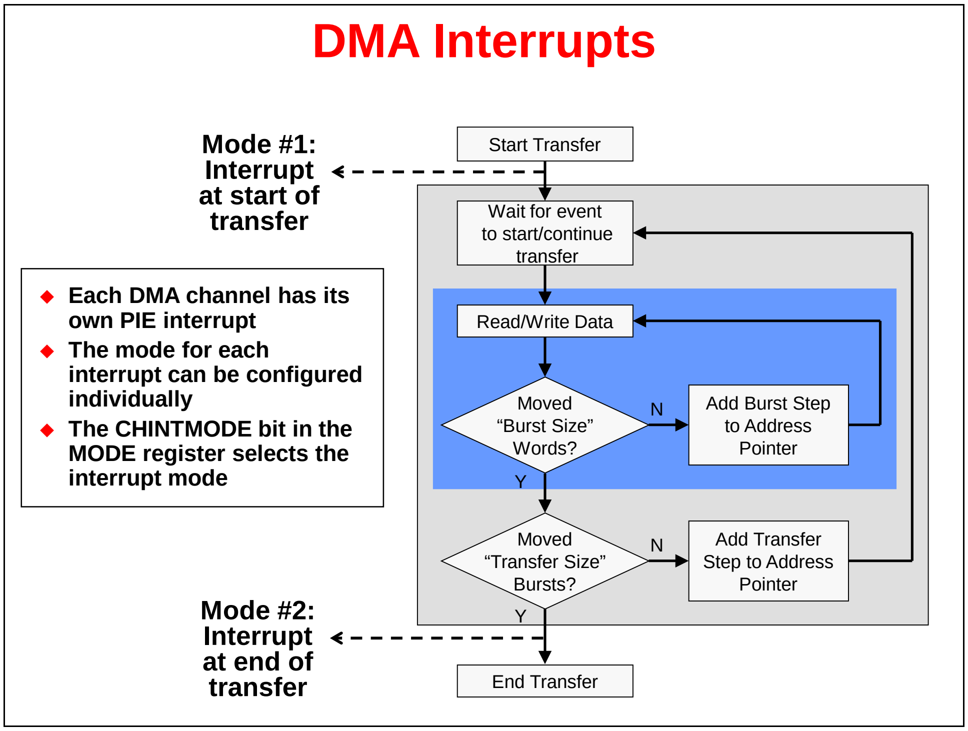
Signed value added to active pointer after each burst



Simplified State Machine Example

3 words/burst
2 bursts/transfer


Start Transfer
Wait for event to start/continue transfer
Read/Write Data
Moved "Burst Size" Words?
N
Add Burst Step to Address Pointer
Y
Moved "Transfer Size" Bursts?
N
Add Transfer Step to Address Pointer
Y
End Transfer

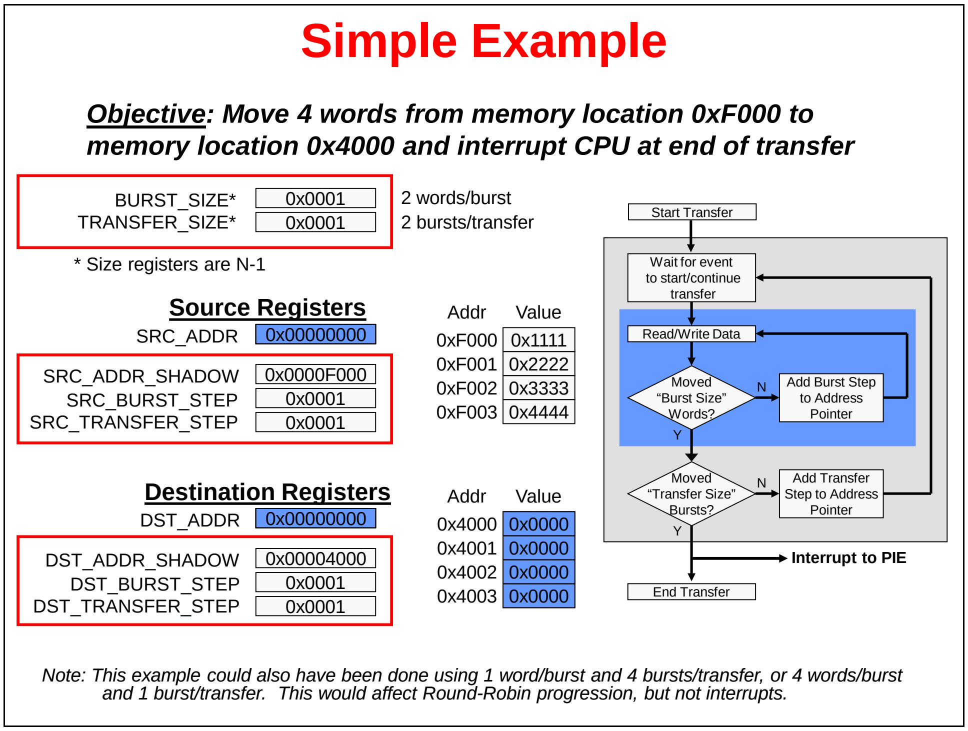


DMA Interrupts

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

Mode #1: Interrupt at start of transfer
Start Transfer
Wait for event to start/continue transfer
Read/Write Data
Moved “Burst Size” Words?
N
Add Burst Step to Address Pointer
Y
Moved “Transfer Size” Bursts?
N
Add Transfer Step to Address Pointer
Y
Mode #2: Interrupt at end of transfer
End Transfer

DMA Examples



Simple Example

***Objective*: Move 4 words from memory location 0xF000 to memory location 0x4000 and interrupt CPU at end of transfer**

BURST_SIZE*	0x0001	2 words/burst
TRANSFER_SIZE*	0x0001	2 bursts/transfer

* Size registers are N-1

Source Registers

SRC_ADDR	0x00000000
SRC_ADDR_SHADOW	0x0000F000
SRC_BURST_STEP	0x0001
SRC_TRANSFER_STEP	0x0001

Addr	Value
0xF000	0x1111
0xF001	0x2222
0xF002	0x3333
0xF003	0x4444

Destination Registers

DST_ADDR	0x00000000
DST_ADDR_SHADOW	0x00004000
DST_BURST_STEP	0x0001
DST_TRANSFER_STEP	0x0001

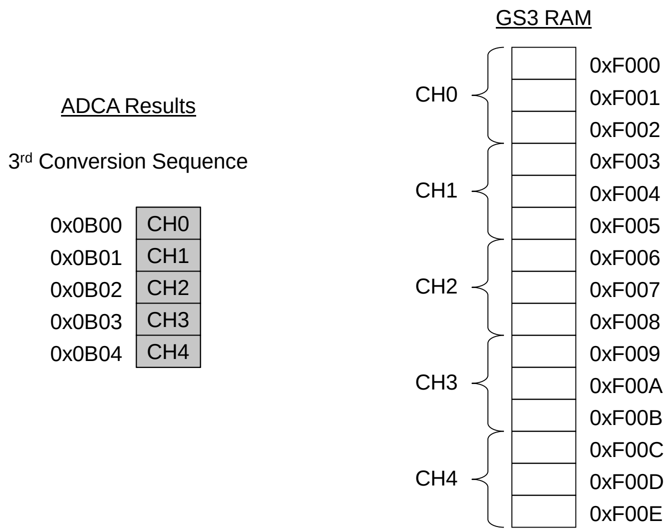
Addr	Value
0x4000	0x0000
0x4001	0x0000
0x4002	0x0000
0x4003	0x0000

Start Transfer
Wait for event to start/continue transfer
Read/Write Data
Moved “Burst Size” Words?
N
Add Burst Step to Address Pointer
Y
Moved “Transfer Size” Bursts?
N
Add Transfer Step to Address Pointer
Y
Interrupt to PIE
End Transfer

Note: This example could also have been done using 1 word/burst and 4 bursts/transfer, or 4 words/burst and 1 burst/transfer. This would affect Round-Robin progression, but not interrupts.

Data Binning Example

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



ADCA Results

3rd Conversion Sequence

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

GS3 RAM

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

Data Binning Example Register Setup

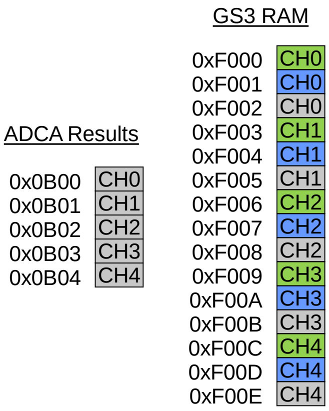
***Objective**: Bin 3 samples of 5 ADC channels, then interrupt the CPU*

ADC Registers:

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	0xFFFC	(-4)
DST_ADDR_SHADOW	0x0000F000	starting address**
DST_BURST_STEP	0x0003	
DST_TRANSFER_STEP	0xFFF5	(-11)



ADCA Results

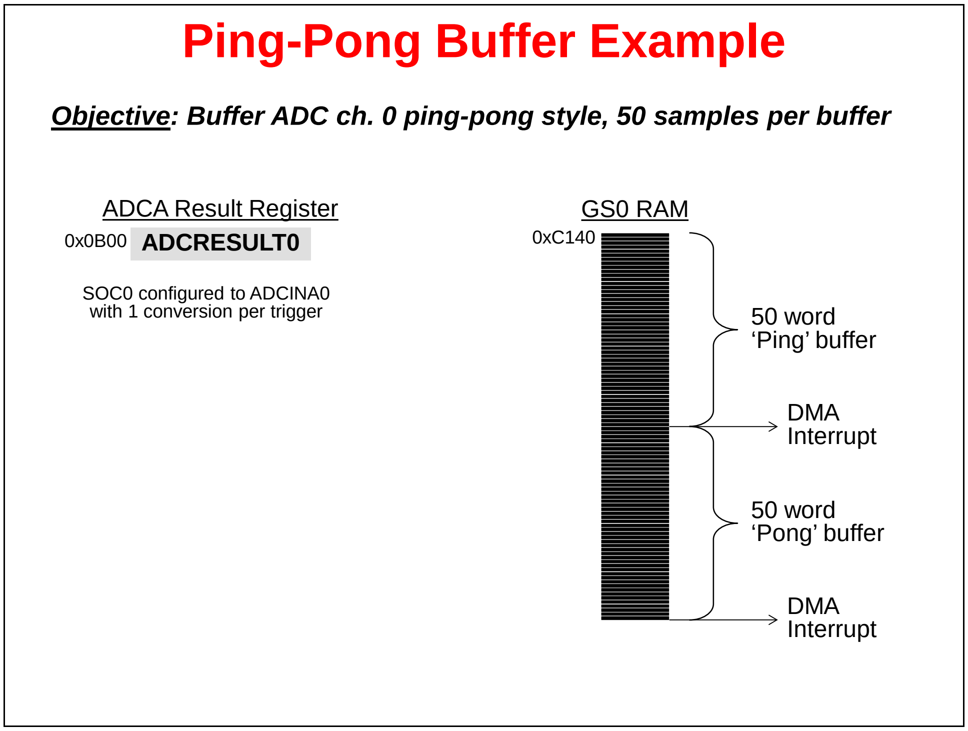
0x0B00	CH0
0x0B01	CH1
0x0B02	CH2
0x0B03	CH3
0x0B04	CH4

GS3 RAM

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

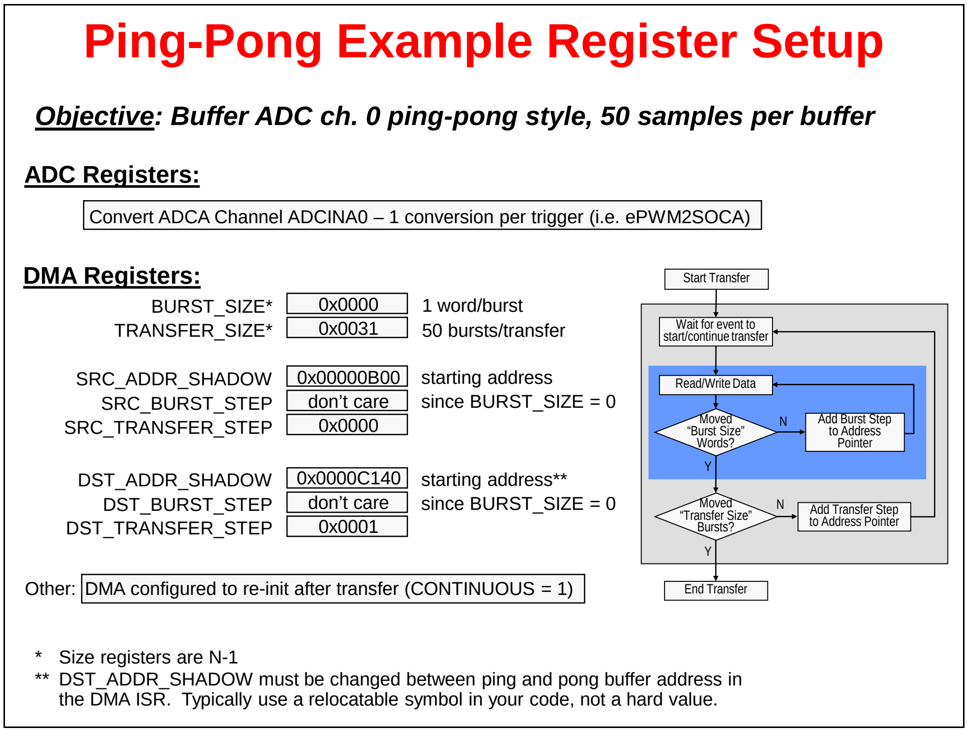
* Size registers are N-1

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



Ping-Pong Buffer Example

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



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	0x00000B00	starting address
SRC_BURST_STEP	don't care	since BURST_SIZE = 0
SRC_TRANSFER_STEP	0x0000	
DST_ADDR_SHADOW	0x0000C140	starting address**
DST_BURST_STEP	don't care	since BURST_SIZE = 0
DST_TRANSFER_STEP	0x0001	

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

Start Transfer
Wait for event to start/continue transfer
Read/Write Data
Moved "Burst Size" Words?
N
Add Burst Step to Address Pointer
Y
Moved "Transfer Size" Bursts?
N
Add Transfer Step to Address Pointer
Y
End Transfer

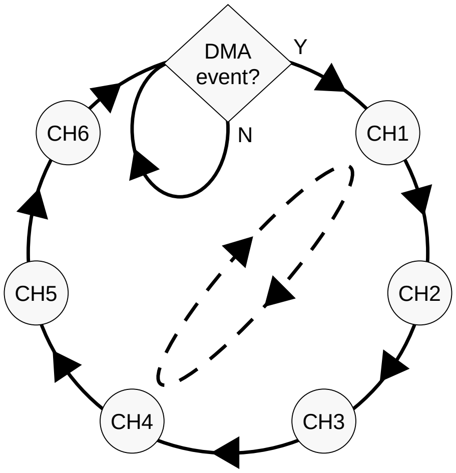
* Size registers are N-1

** 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.

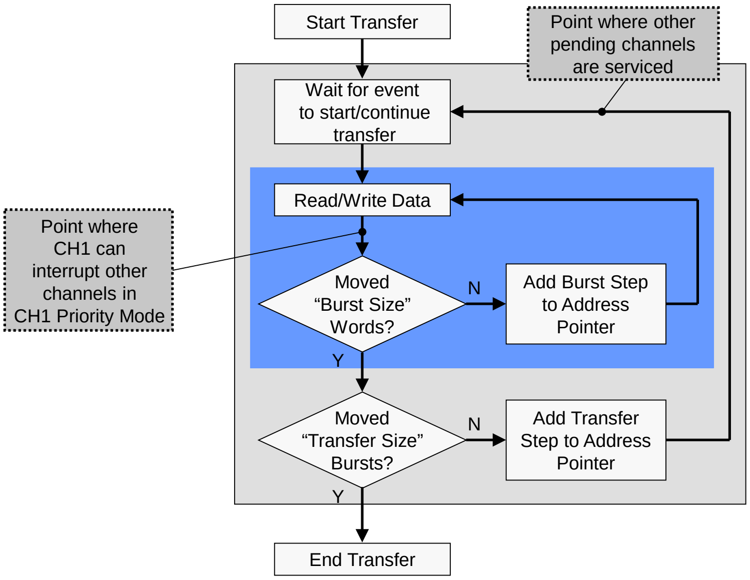
Channel Priority Modes

Channel Priority Modes

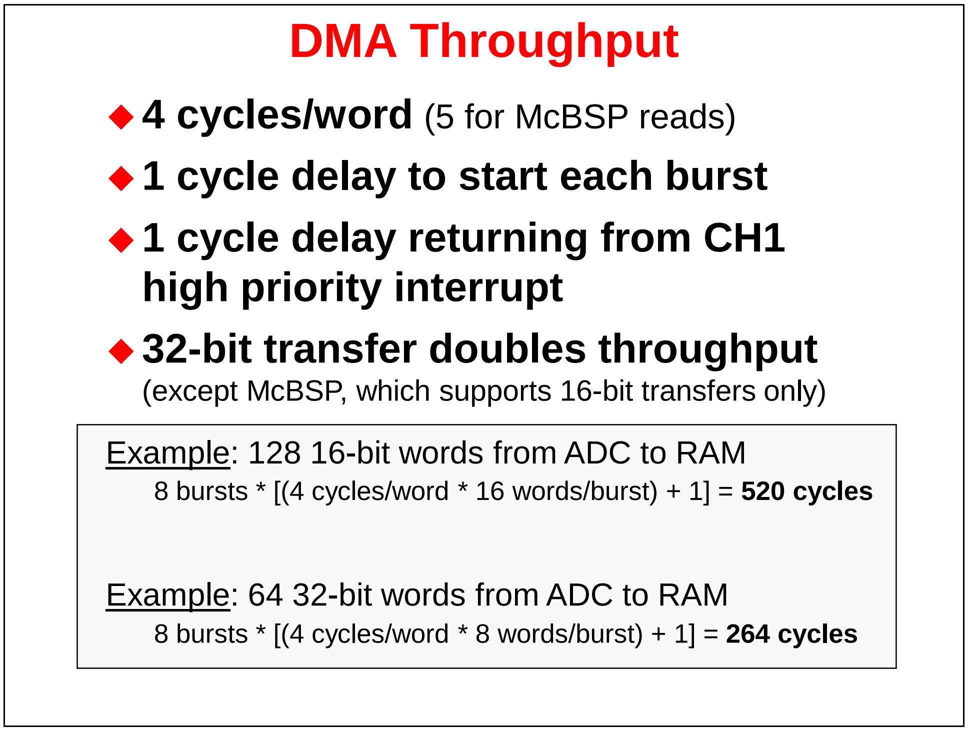
- **Round Robin Mode:**
  - **All channels have equal priority**
  - **After each enabled channel has transferred a *burst of words*, the next enabled channel is serviced in round robin fashion**
- **Channel 1 High Priority Mode:**
  - **Same as Round Robin *except* CH1 can interrupt DMA state machine**
  - **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


Start Transfer
Point where other pending channels are serviced
Wait for event to start/continue transfer
Read/Write Data
Point where CH1 can interrupt other channels in CH1 Priority Mode
Moved "Burst Size" Words?
N
Add Burst Step to Address Pointer
Y
Moved "Transfer Size" Bursts?
N
Add Transfer Step to Address Pointer
Y
End Transfer

DMA Throughput



DMA Throughput

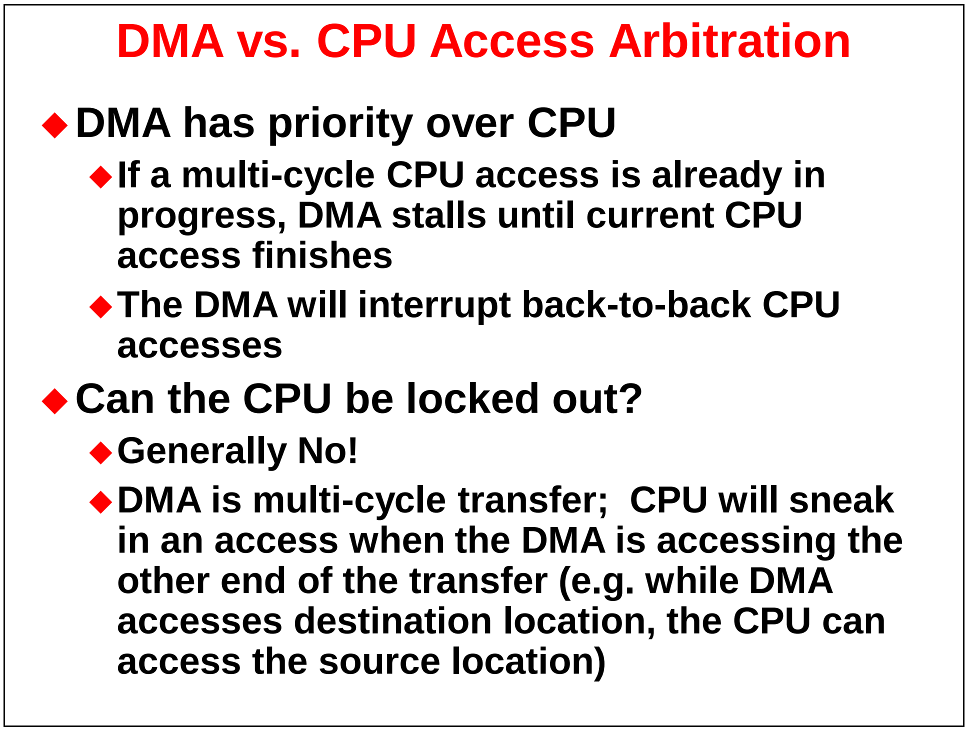
- **4 cycles/word** (5 for McBSP reads)
- **1 cycle delay to start each burst**
- **1 cycle delay returning from CH1 high priority interrupt**
- **32-bit transfer doubles throughput** (except McBSP, which supports 16-bit transfers only)

Example: 128 16-bit words from ADC to RAM

8 bursts * [(4 cycles/word * 16 words/burst) + 1] = **520 cycles**

Example: 64 32-bit words from ADC to RAM

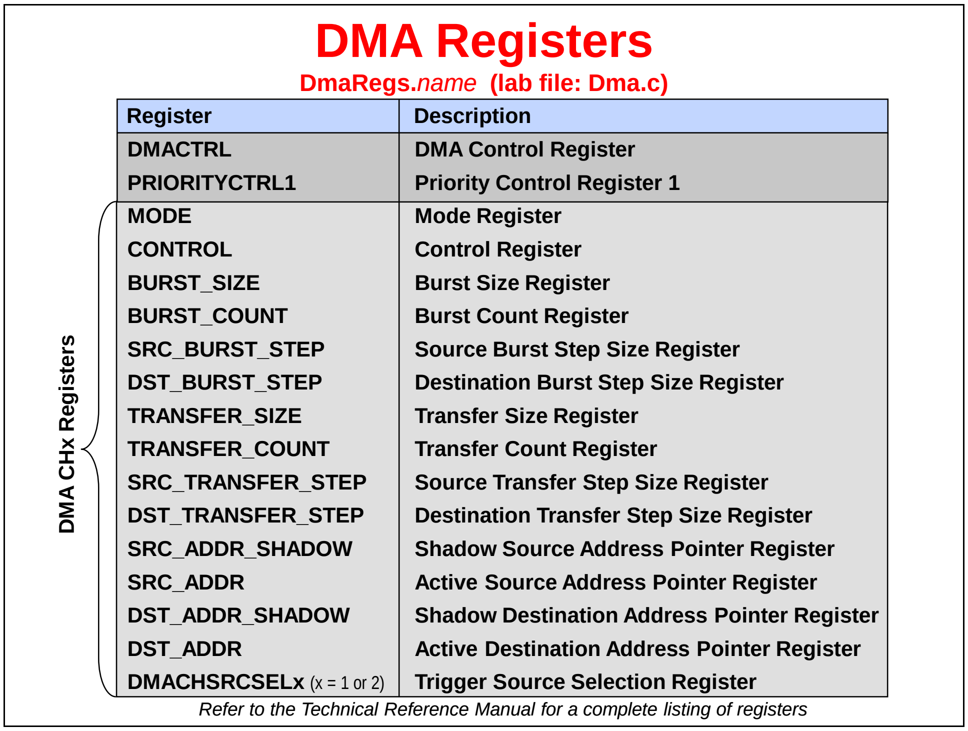
8 bursts * [(4 cycles/word * 8 words/burst) + 1] = **264 cycles**



DMA vs. CPU Access Arbitration

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

DMA Registers



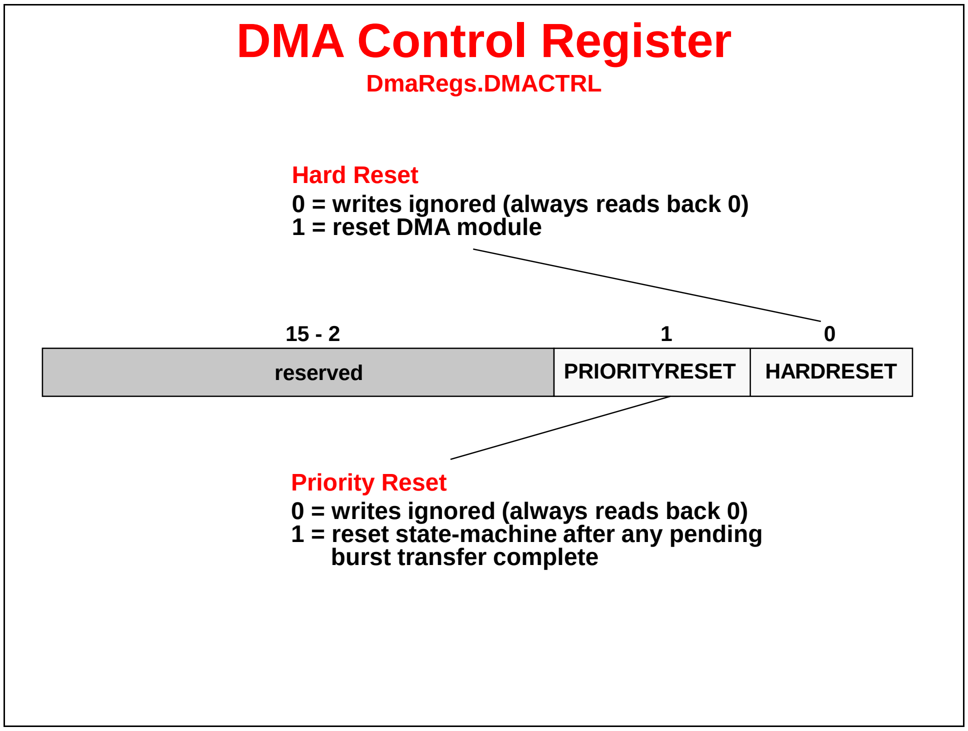
DMA Registers

DmaRegs.*name* (lab file: Dma.c)

Register	Description
DMACTRL	DMA Control Register
PRIORITYCTRL1	Priority Control Register 1
MODE	Mode Register
CONTROL	Control Register
BURST_SIZE	Burst Size Register
BURST_COUNT	Burst Count Register
SRC_BURST_STEP	Source Burst Step Size Register
DST_BURST_STEP	Destination Burst Step Size Register
TRANSFER_SIZE	Transfer Size Register
TRANSFER_COUNT	Transfer Count Register
SRC_TRANSFER_STEP	Source Transfer Step Size Register
DST_TRANSFER_STEP	Destination Transfer Step Size Register
SRC_ADDR_SHADOW	Shadow Source Address Pointer Register
SRC_ADDR	Active Source Address Pointer Register
DST_ADDR_SHADOW	Shadow Destination Address Pointer Register
DST_ADDR	Active Destination Address Pointer Register
DMACHSRCSELx (x = 1 or 2)	Trigger Source Selection Register

DMA CHx Registers

Refer to the Technical Reference Manual for a complete listing of registers



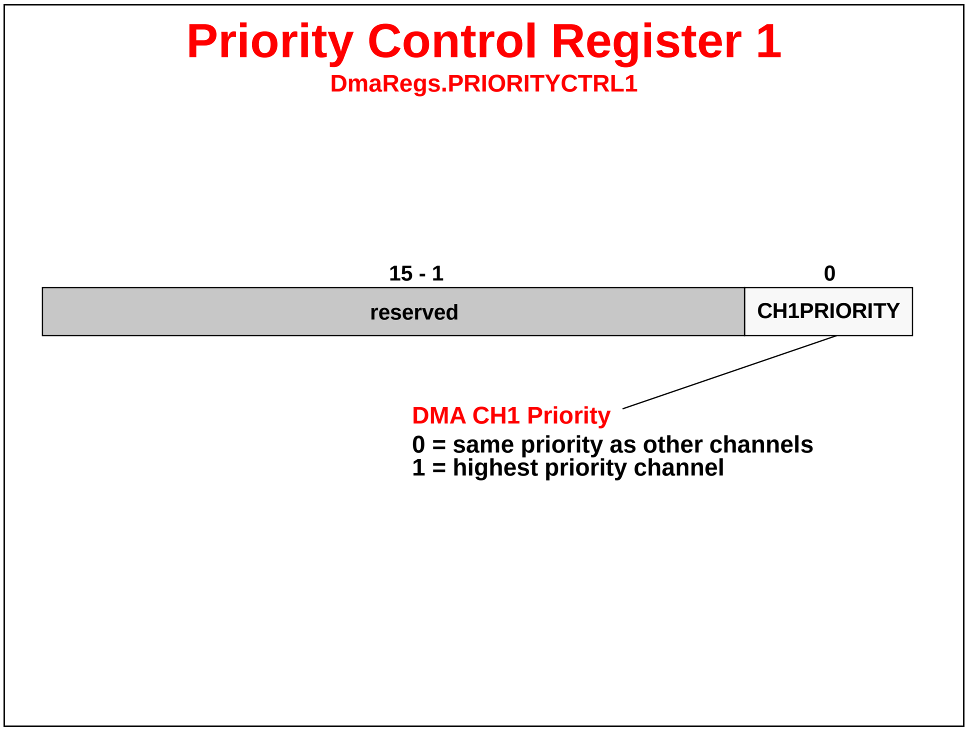
DMA Control Register

DmaRegs.DMACTRL

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

15 - 2	1	0
reserved	PRIORITYRESET	HARDRESET

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

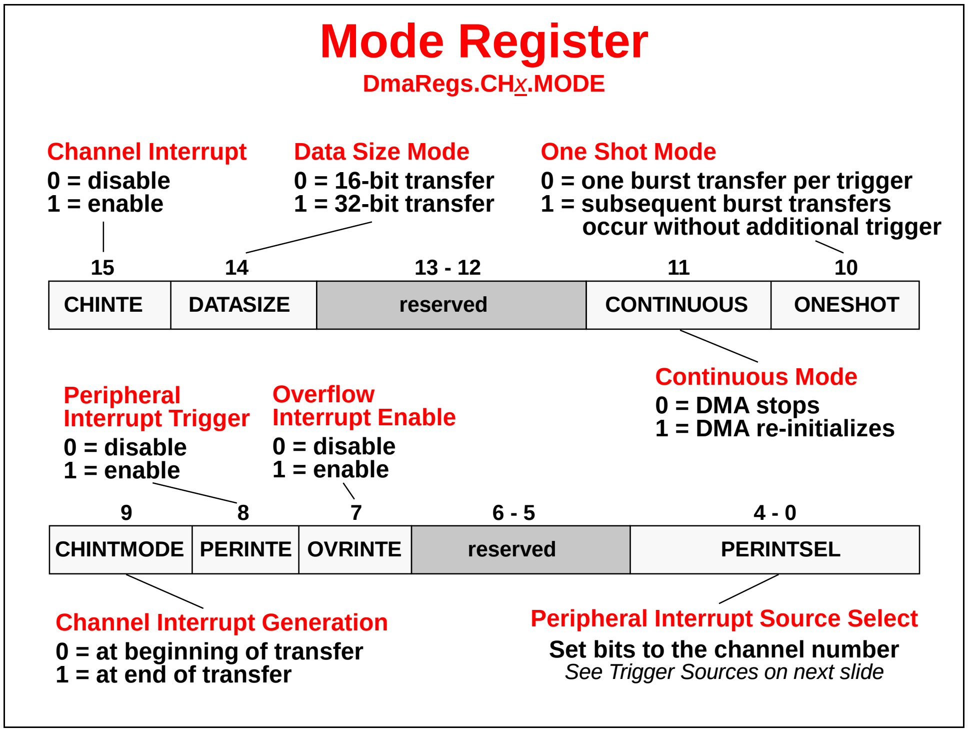


Priority Control Register 1

DmaRegs.PRIORITYCTRL1

15 - 1	0
reserved	CH1PRIORITY

DMA CH1 Priority
0 = same priority as other channels
1 = highest priority channel



Mode Register

DmaRegs.CH*x*.MODE

Channel Interrupt
0 = disable
1 = enable

Data Size Mode
0 = 16-bit transfer
1 = 32-bit transfer

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

15	14	13 - 12	11	10
CHINTE	DATASIZE	reserved	CONTINUOUS	ONESHOT

Peripheral Interrupt Trigger
0 = disable
1 = enable

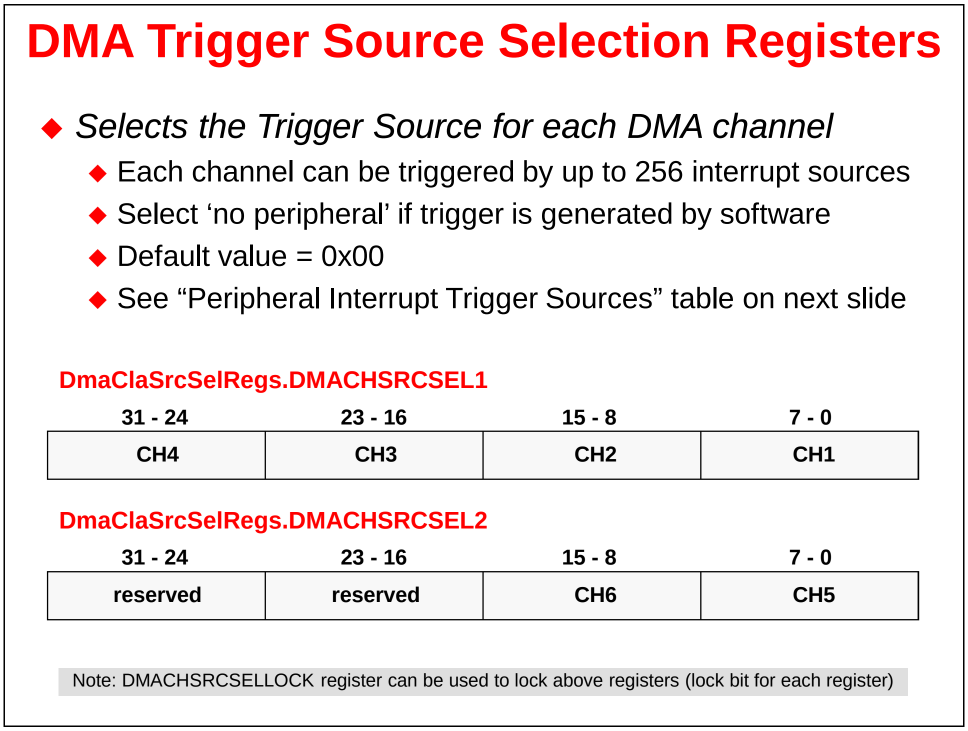
Overflow Interrupt Enable
0 = disable
1 = enable

Continuous Mode
0 = DMA stops
1 = DMA re-initializes

9	8	7	6 - 5	4 - 0
CHINTMODE	PERINTE	OVRINTE	reserved	PERINTSEL

Channel Interrupt Generation
0 = at beginning of transfer
1 = at end of transfer

Peripheral Interrupt Source Select
Set bits to the channel number
See Trigger Sources on next slide



DMA Trigger Source Selection Registers

- *Selects the Trigger Source for each DMA channel*
  - Each channel can be triggered by up to 256 interrupt sources
  - Select ‘no peripheral’ if trigger is generated by software
  - Default value = 0x00
  - See “Peripheral Interrupt Trigger Sources” table on next slide

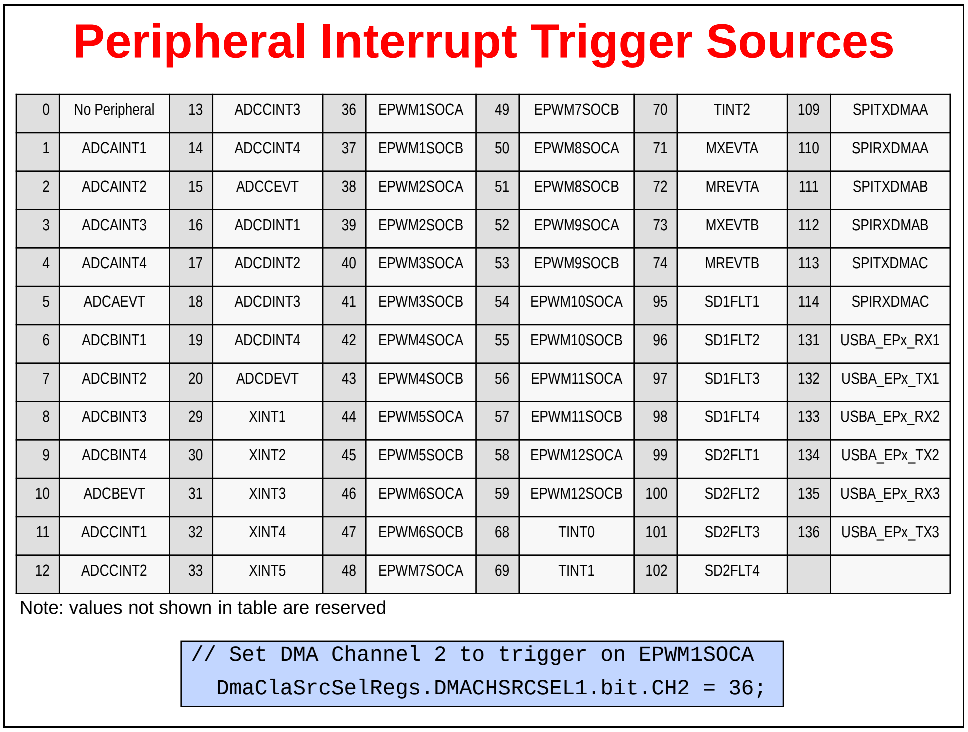
DmaClaSrcSelRegs.DMACHSRCSEL1

31 - 24	23 - 16	15 - 8	7 - 0
CH4	CH3	CH2	CH1

DmaClaSrcSelRegs.DMACHSRCSEL2

31 - 24	23 - 16	15 - 8	7 - 0
reserved	reserved	CH6	CH5

Note: DMACHSRCSELLOCK register can be used to lock above registers (lock bit for each register)

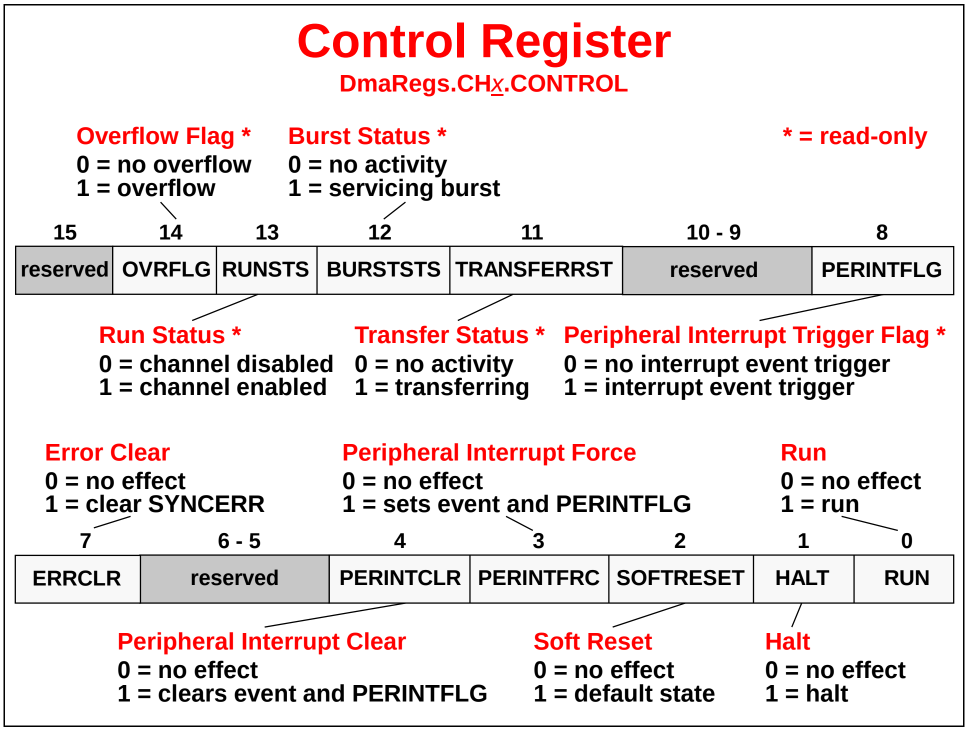


Peripheral Interrupt Trigger Sources

0	No Peripheral	13	ADCCINT3	36	EPWM1SOCA	49	EPWM7SOCB	70	TINT2	109	SPITXDMAA
1	ADCAINT1	14	ADCCINT4	37	EPWM1SOCB	50	EPWM8SOCA	71	MXEVTA	110	SPIRXDMAA
2	ADCAINT2	15	ADCCEVT	38	EPWM2SOCA	51	EPWM8SOCB	72	MREVTA	111	SPITXDMAB
3	ADCAINT3	16	ADCDINT1	39	EPWM2SOCB	52	EPWM9SOCA	73	MXEVTB	112	SPIRXDMAB
4	ADCAINT4	17	ADCDINT2	40	EPWM3SOCA	53	EPWM9SOCB	74	MREVTB	113	SPITXDMAC
5	ADCAEVT	18	ADCDINT3	41	EPWM3SOCB	54	EPWM10SOCA	95	SD1FLT1	114	SPIRXDMAC
6	ADCBINT1	19	ADCDINT4	42	EPWM4SOCA	55	EPWM10SOCB	96	SD1FLT2	131	USBA_EPx_RX1
7	ADCBINT2	20	ADCDEVT	43	EPWM4SOCB	56	EPWM11SOCA	97	SD1FLT3	132	USBA_EPx_TX1
8	ADCBINT3	29	XINT1	44	EPWM5SOCA	57	EPWM11SOCB	98	SD1FLT4	133	USBA_EPx_RX2
9	ADCBINT4	30	XINT2	45	EPWM5SOCB	58	EPWM12SOCA	99	SD2FLT1	134	USBA_EPx_TX2
10	ADCBEVT	31	XINT3	46	EPWM6SOCA	59	EPWM12SOCB	100	SD2FLT2	135	USBA_EPx_RX3
11	ADCCINT1	32	XINT4	47	EPWM6SOCB	68	TINT0	101	SD2FLT3	136	USBA_EPx_TX3
12	ADCCINT2	33	XINT5	48	EPWM7SOCA	69	TINT1	102	SD2FLT4		

Note: values not shown in table are reserved

```
// Set DMA Channel 2 to trigger on EPWM1SOCA
  DmaClaSrcSelRegs.DMACHSRCSEL1.bit.CH2 = 36;
```



Control Register

DmaRegs.CHx.CONTROL

* = read-only

Overflow Flag *
0 = no overflow
1 = overflow

Burst Status *
0 = no activity
1 = servicing burst

15	14	13	12	11	10 - 9	8
reserved	OVRFLG	RUNSTS	BURSTSTS	TRANSFERRST	reserved	PERINTFLG

Run Status *
0 = channel disabled
1 = channel enabled

Transfer Status *
0 = no activity
1 = transferring

Peripheral Interrupt Trigger Flag *
0 = no interrupt event trigger
1 = interrupt event trigger

Error Clear
0 = no effect
1 = clear SYNCERR

Peripheral Interrupt Force
0 = no effect
1 = sets event and PERINTFLG

Run
0 = no effect
1 = run

7	6 - 5	4	3	2	1	0
ERRCLR	reserved	PERINTCLR	PERINTFRC	SOFTRESET	HALT	RUN

Peripheral Interrupt Clear
0 = no effect
1 = clears event and PERINTFLG

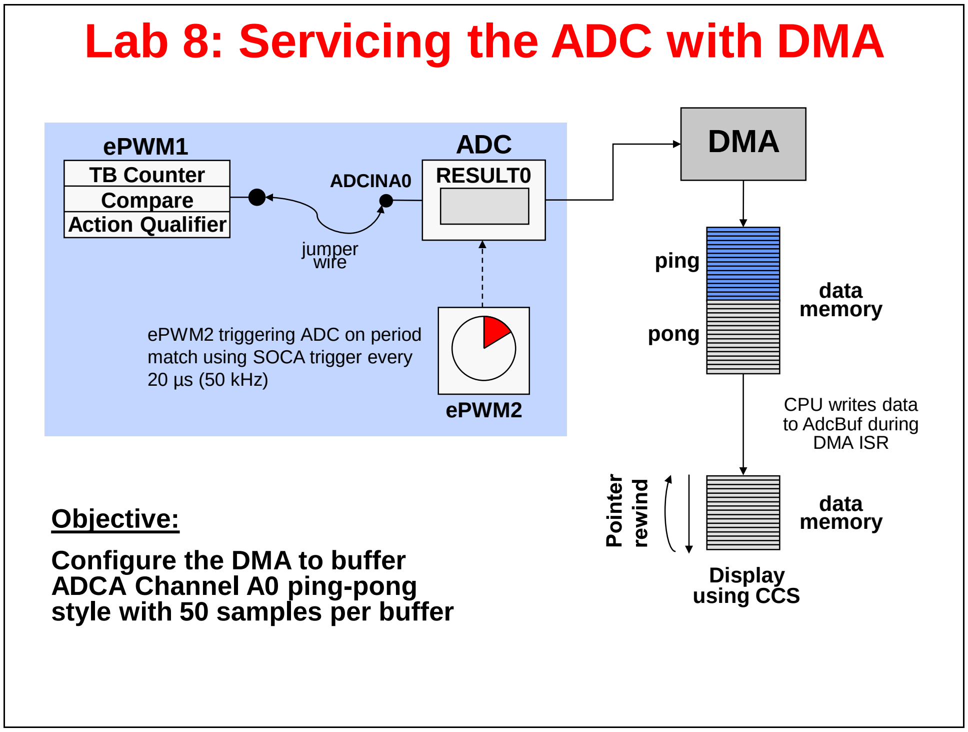
Soft Reset
0 = no effect
1 = default state

Halt
0 = no effect
1 = halt

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.



Lab 8: Servicing the ADC with DMA

Objective:

Configure the DMA to buffer ADCA Channel A0 ping-pong style with 50 samples per buffer

➢ 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 then click `Browse…` next to the "Select search-directory" box. Navigate to: `C:\C28x\Labs\Lab8\cpu01` and click `OK`. 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
CodeStartBranch.asm
Dac.c
DefaultIsr_8.c
DelayUs.asm
Dma.c
ECap.c
EPwm.c
F2837xD_Adc.c
F2837xD_GlobalVariableDefs.c
F2837xD_Headers_nonBIOS_cpu1.cmd
Gpio.c
Lab_8.cmd
Main_8.c
PieCtrl.c
PieVect.c
SineTable.c
SysCtrl.c
Watchdog.c
Xbar.c
```

Inspect Lab_8.cmd

2. Open and inspect `Lab_8.cmd`. Notice that a section called “`dmaMemBufs`” is being linked to `RAMGS4`. 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.c` to implement the DMA operation as described in the objective for this lab exercise. Configure the DMA Channel 1 Mode Register (MODE) so that the peripheral interrupt source select is set to channel 1. Enable the peripheral interrupt trigger and set the channel for interrupt generation at the start of transfer. Configure for 16-bit data transfers with one burst per trigger and auto re-initialization at the end of the transfer. Enable the channel interrupt. Configure the DMA Trigger Selection Register (DMACHSRCSELx) so that the ADCAINT1 is the peripheral interrupt trigger source. In the DMA Channel 1 Control Register (CONTROL) clear the error and peripheral interrupt bits. Enable the channel 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 “`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.c` to do the following:

   - Enable the "`DMA_CH1`" interrupt in the PIE (Hint: use the `PieCtrlRegs` structure)
   - Enable the appropriate core interrupt in the IER register

8. Open and inspect `DefaultIsr_8.c`. Notice that this file contains the DMA interrupt service routine. 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). A Launching Debug Session window will open. Select only CPU1 to load the program on (i.e. ***uncheck*** CPU2), and then click `OK`. Then the CCS Debug perspective view should open, the program will load automatically, and you should now be at the start of `main()`. If the device has been power cycled since the last lab exercise, be sure to configure the boot mode to EMU_BOOT_SARAM using the Scripts menu.

Run the Code – Test the DMA Operation

Note: For the next step, check to be sure that the jumper wire connecting PWM1A (header J4, pin #40) to ADCINA0 (header J3, pin #30) is in place on the LaunchPad.

11. Run the code in real-time mode using the Script function: `Scripts` → `Realtime Emulation Control` → `Run_Realtime_with_Reset`. 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. Fully halt the CPU (real-time mode) by using the Script function: `Scripts` → `Realtime Emulation Control` → `Full_Halt`.

Terminate Debug Session and Close Project

15. Terminate the active debug session using the `Terminate` button. This will close the debugger and return 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`.

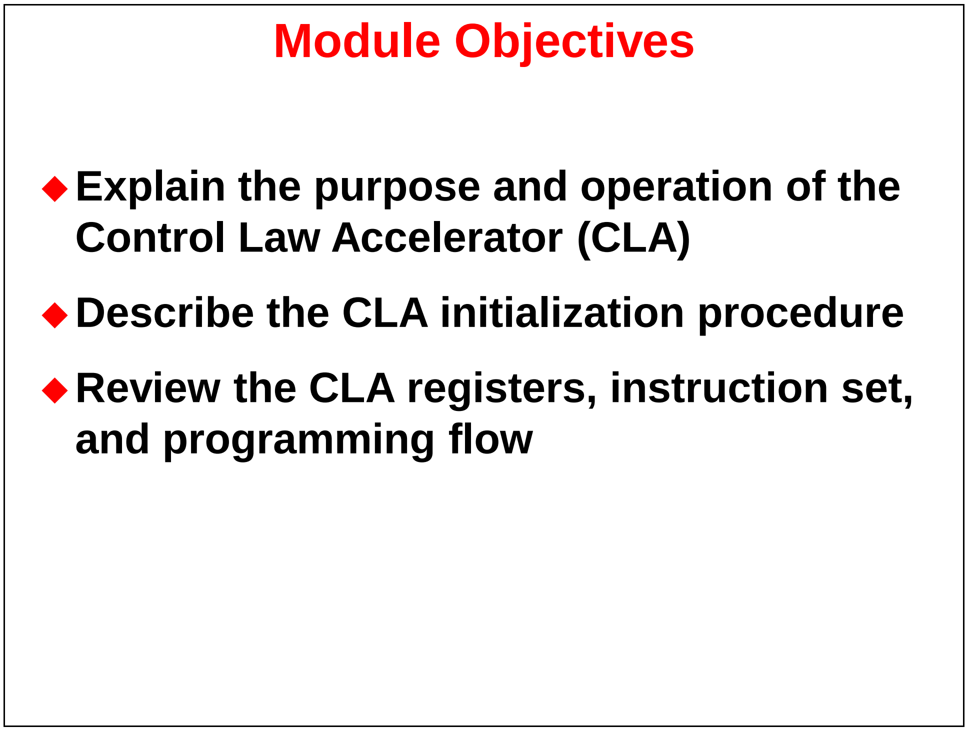
End of Exercise

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 independently and in parallel with 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, SPI, McBSP, uPP 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.

Module Objectives

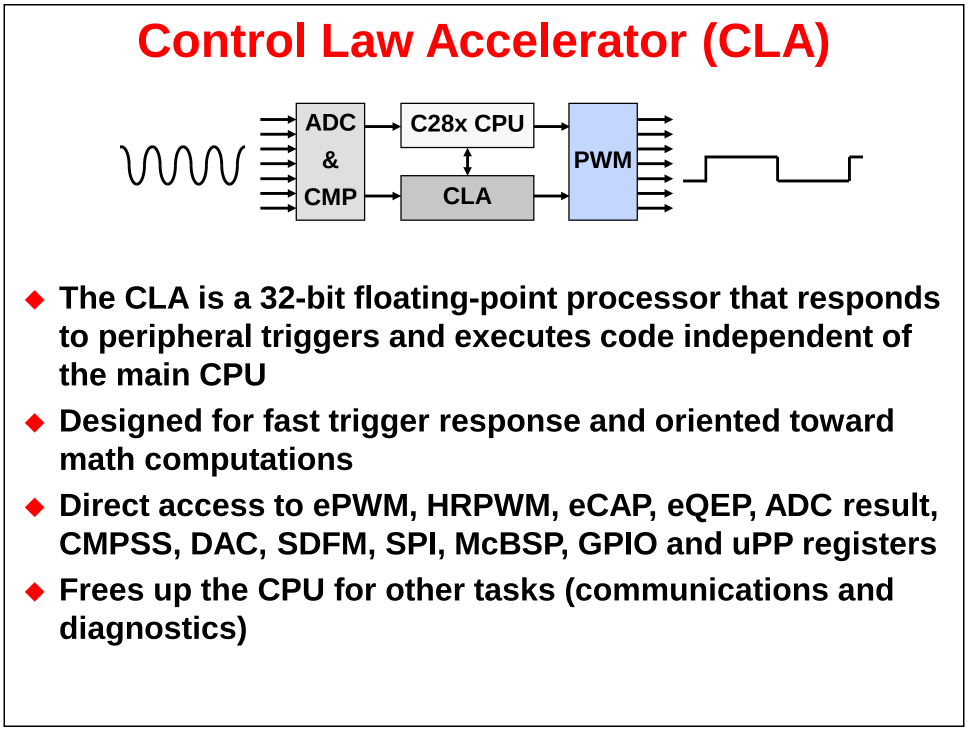


Module Objectives

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

Chapter Topics

Control Law Accelerator (CLA)


ADC
&
CMP
C28x CPU
CLA
PWM

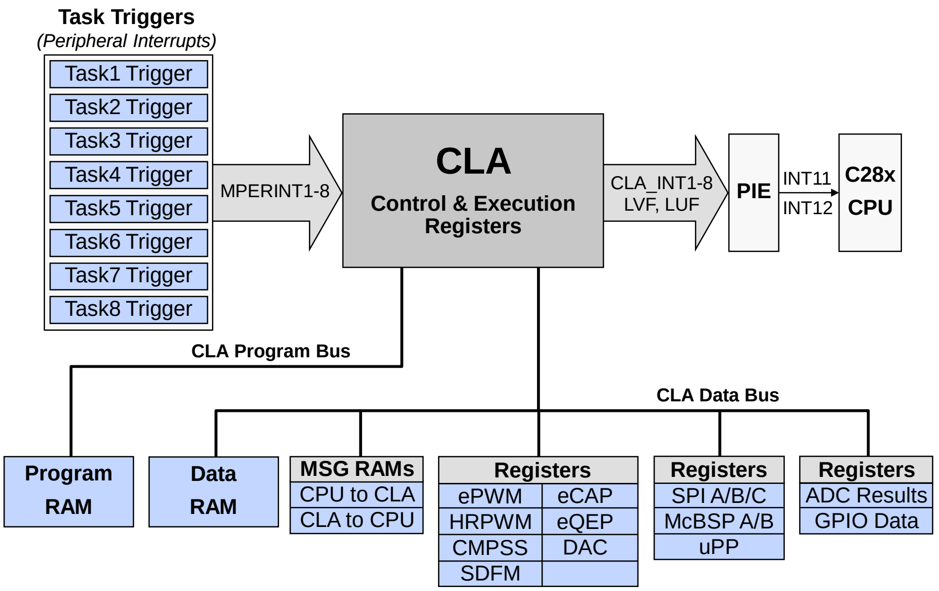
Control Law Accelerator (CLA)

- **The CLA is a 32-bit floating-point processor that responds to peripheral triggers and executes code independent of the main CPU**
- **Designed for fast trigger response and oriented toward math computations**
- **Direct access to ePWM, HRPWM, eCAP, eQEP, ADC result, CMPSS, DAC, SDFM, SPI, McBSP, GPIO and uPP registers**
- **Frees up the CPU for other tasks (communications and diagnostics)**

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. Each CPU subsystem has its own CLA that 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 Block Diagram



Task Triggers
(Peripheral Interrupts)
Task1 Trigger
Task2 Trigger
Task3 Trigger
Task4 Trigger
Task5 Trigger
Task6 Trigger
Task7 Trigger
Task8 Trigger
MPERINT1-8
CLA
Control & Execution Registers
CLA_INT1-8
LVF, LUF
PIE
INT11
INT12
C28x CPU
CLA Program Bus
CLA Data Bus
Program RAM
Data RAM
MSG RAMs
CPU to CLA
CLA to CPU
Registers
ePWM
eCAP
HRPWM
eQEP
CMPSS
DAC
SDFM
Registers
SPI A/B/C
McBSP A/B
uPP
Registers
ADC Results
GPIO Data

CLA Memory and Register Access

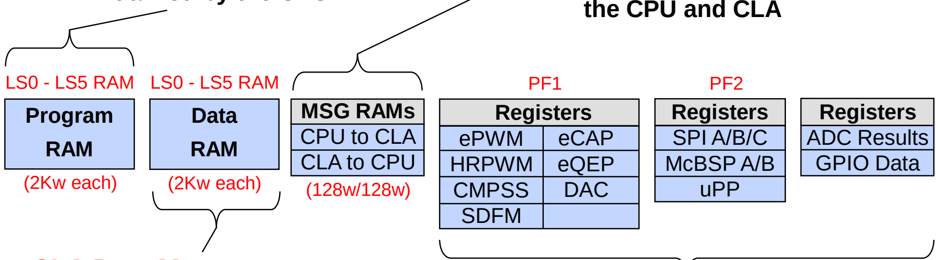
CLA Memory and Register Access

CLA Program Memory

- Contains CLA program code
- Mapped to the CPU at reset
- Initialized by the CPU

Message RAMs

- Used to pass data between the CPU and CLA
- Always mapped to both the CPU and CLA



LS0 - LS5 RAM
Program RAM
(2Kw each)
LS0 - LS5 RAM
Data RAM
(2Kw each)
MSG RAMs
CPU to CLA
CLA to CPU
(128w/128w)
PF1
Registers
ePWM
eCAP
HRPWM
eQEP
CMPSS
DAC
SDFM
PF2
Registers
SPI A/B/C
McBSP A/B
uPP
Registers
ADC Results
GPIO Data

CLA Data Memory

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

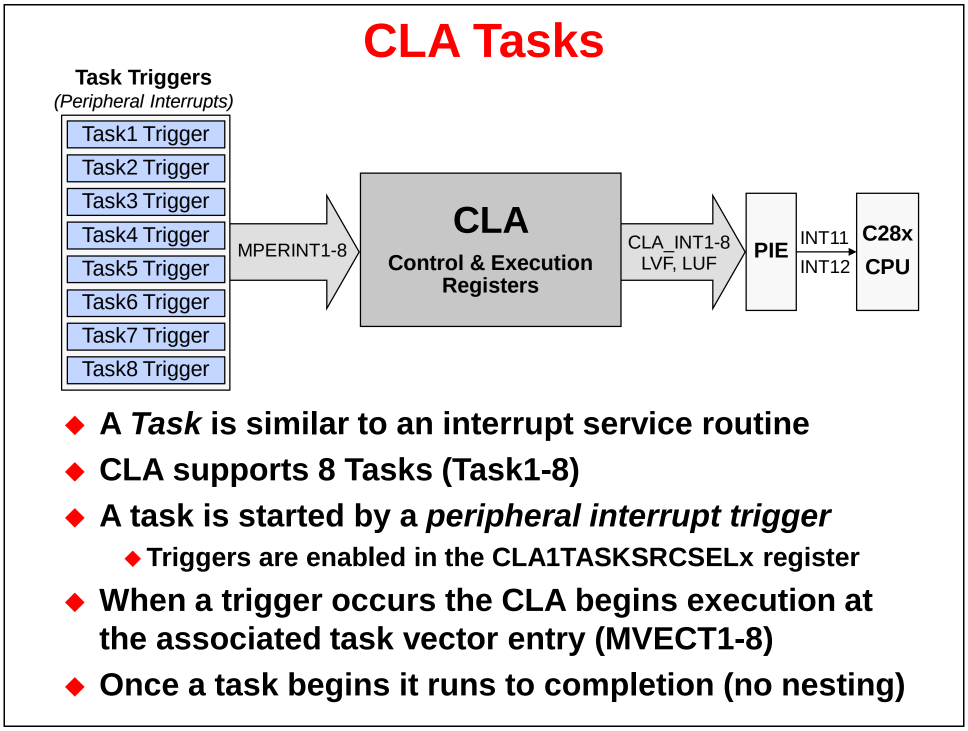
Note: CPU1.CLA1 has access to EMIF2 for data only

Peripheral Register Access

- Provides direct access to peripherals
- Either the CLA or DMA can have access to a PF, but not both
- CPUx.SECMSEL register selects CLA or DMA per PF (default is CLA)

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.

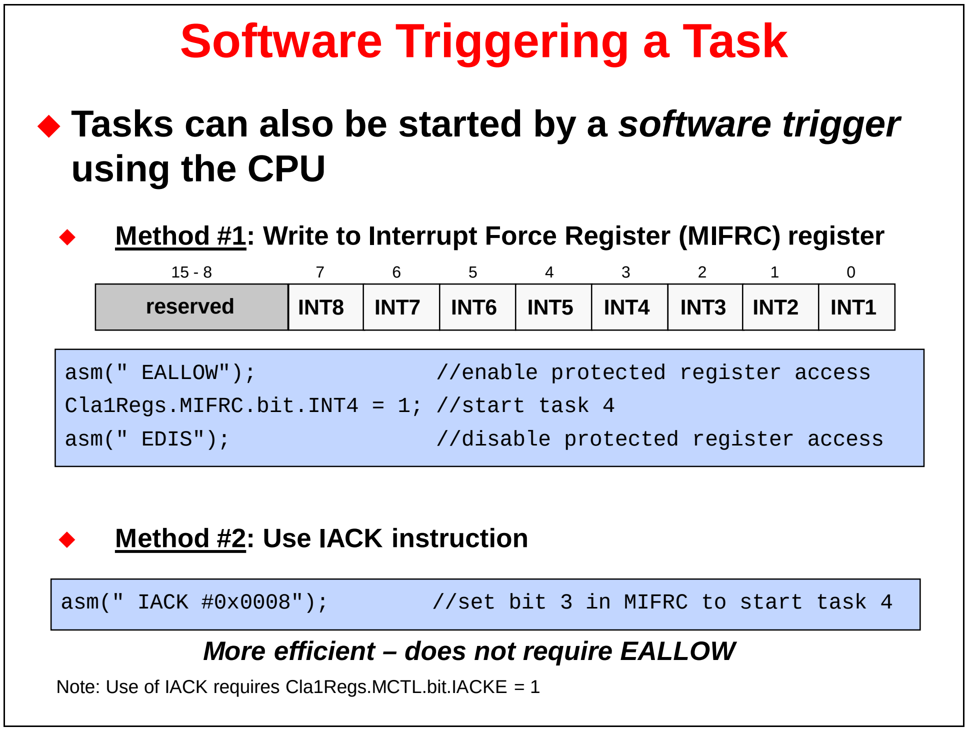
CLA Tasks



CLA Tasks

- **A *Task* is similar to an interrupt service routine**
- **CLA supports 8 Tasks (Task1-8)**
- **A task is started by a *peripheral interrupt trigger***
  - **Triggers are enabled in the CLA1TASKSRCSELx register**
- **When a trigger occurs the CLA begins execution at the associated task vector entry (MVECT1-8)**
- **Once a task begins it runs to completion (no nesting)**

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. Tasks can be written in C or assembly code, where typically the user will use assembly code for high performance time-critical tasks, and C for non-critical tasks. 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. Since the CLA is a software programmable accelerator, it is very flexible and can be modified for different applications.



Software Triggering a Task

- Tasks can also be started by a *software trigger* using the CPU
  - **Method #1**: Write to Interrupt Force Register (MIFRC) register

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

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

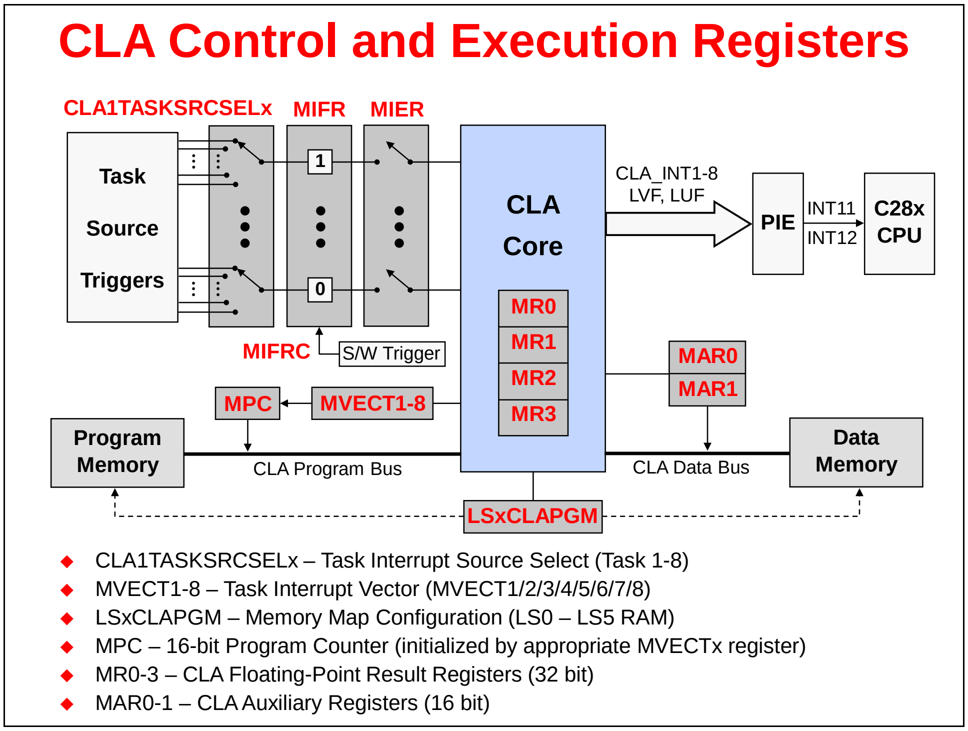
- **Method #2**: Use IACK instruction

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

More efficient – does not require EALLOW

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

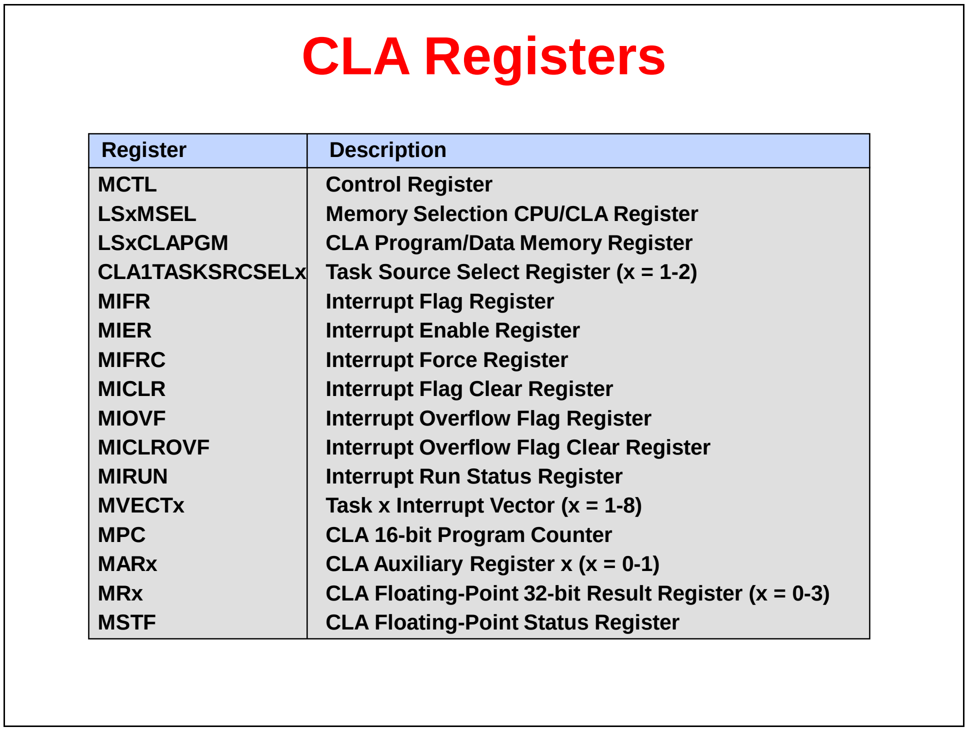
CLA Control and Execution Registers



CLA Control and Execution Registers

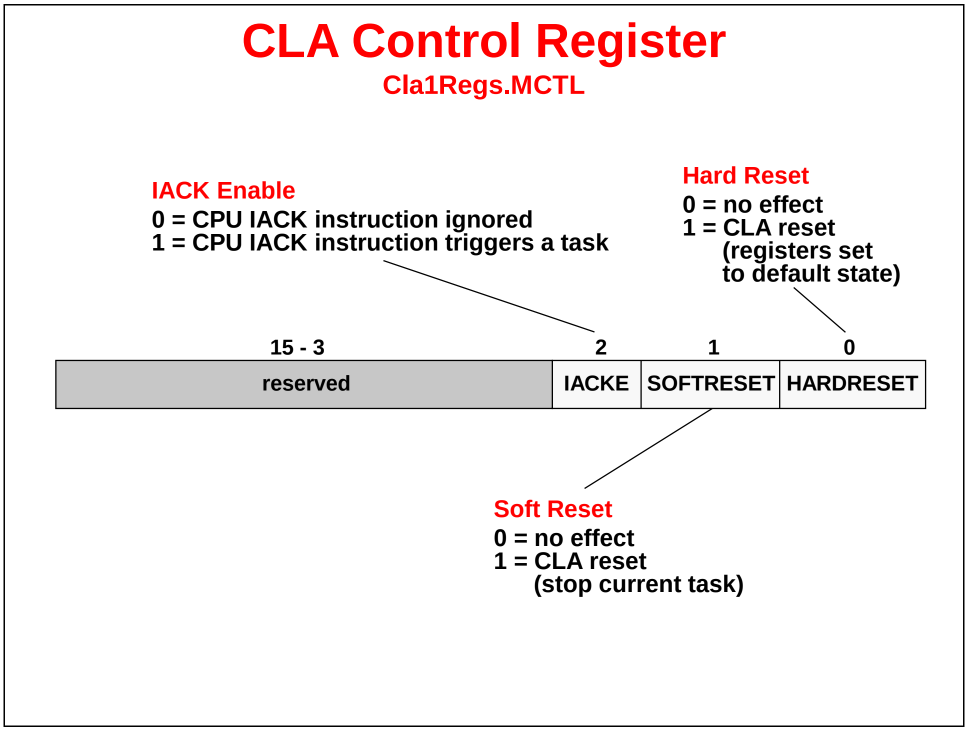
- CLA1TASKSRCSELx – Task Interrupt Source Select (Task 1-8)
- MVECT1-8 – Task Interrupt Vector (MVECT1/2/3/4/5/6/7/8)
- LSxCLAPGM – Memory Map Configuration (LS0 – LS5 RAM)
- MPC – 16-bit Program Counter (initialized by appropriate MVECTx register)
- MR0-3 – CLA Floating-Point Result Registers (32 bit)
- MAR0-1 – CLA Auxiliary Registers (16 bit)

CLA Registers



CLA Registers

Register	Description
MCTL	Control Register
LSxMSEL	Memory Selection CPU/CLA Register
LSxCLAPGM	CLA Program/Data Memory Register
CLA1TASKSRCSELx	Task Source Select Register (x = 1-2)
MIFR	Interrupt Flag Register
MIER	Interrupt Enable Register
MIFRC	Interrupt Force Register
MICLR	Interrupt Flag Clear Register
MIOVF	Interrupt Overflow Flag Register
MICLROVF	Interrupt Overflow Flag Clear Register
MIRUN	Interrupt Run Status Register
MVECTx	Task x Interrupt Vector (x = 1-8)
MPC	CLA 16-bit Program Counter
MARx	CLA Auxiliary Register x (x = 0-1)
MRx	CLA Floating-Point 32-bit Result Register (x = 0-3)
MSTF	CLA Floating-Point Status Register



CLA Control Register

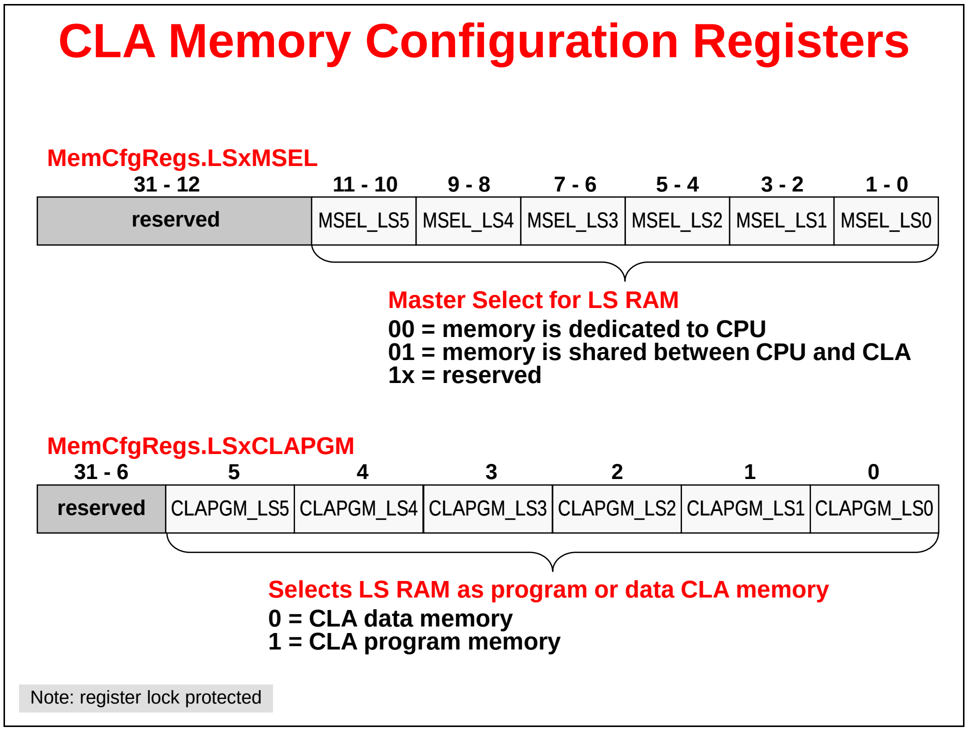
Cla1Regs.MCTL

IACK Enable
0 = CPU IACK instruction ignored
1 = CPU IACK instruction triggers a task

Hard Reset
0 = no effect
1 = CLA reset
(registers set to default state)

15 - 3	2	1	0
reserved	IACKE	SOFTRESET	HARDRESET

Soft Reset
0 = no effect
1 = CLA reset
(stop current task)



CLA Memory Configuration Registers

MemCfgRegs.LSxMSEL

31 - 12	11 - 10	9 - 8	7 - 6	5 - 4	3 - 2	1 - 0
reserved	MSEL_LS5	MSEL_LS4	MSEL_LS3	MSEL_LS2	MSEL_LS1	MSEL_LS0

Master Select for LS RAM

00 = memory is dedicated to CPU
01 = memory is shared between CPU and CLA
1x = reserved

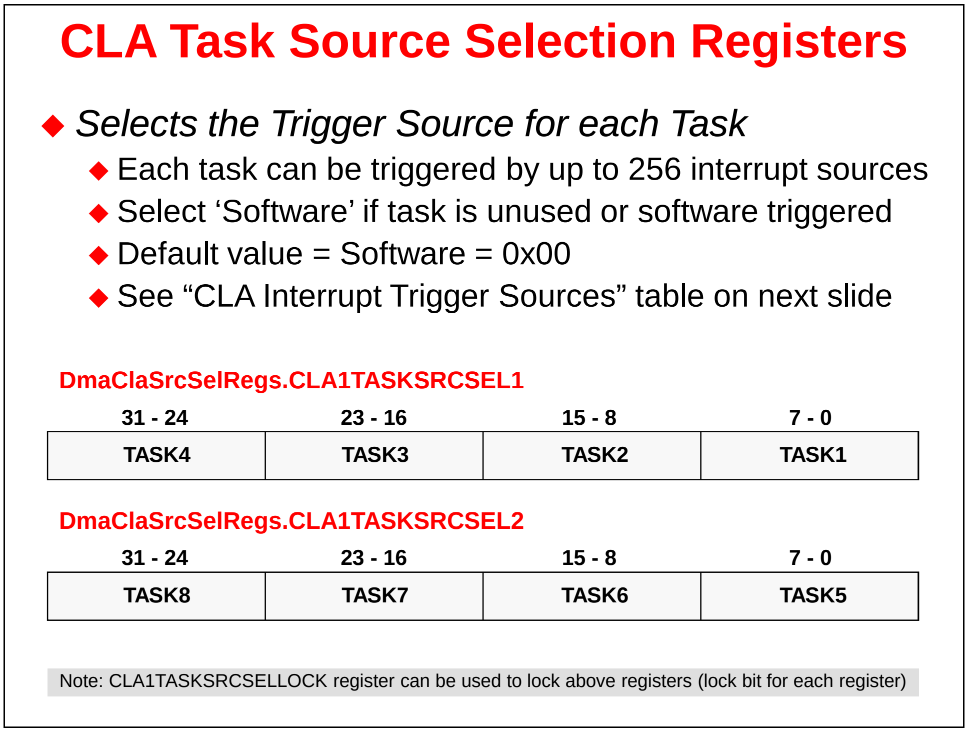
MemCfgRegs.LSxCLAPGM

31 - 6	5	4	3	2	1	0
reserved	CLAPGM_LS5	CLAPGM_LS4	CLAPGM_LS3	CLAPGM_LS2	CLAPGM_LS1	CLAPGM_LS0

Selects LS RAM as program or data CLA memory

0 = CLA data memory
1 = CLA program memory

Note: register lock protected



CLA Task Source Selection Registers

- *Selects the Trigger Source for each Task*
  - Each task can be triggered by up to 256 interrupt sources
  - Select ‘Software’ if task is unused or software triggered
  - Default value = Software = 0x00
  - See “CLA Interrupt Trigger Sources” table on next slide

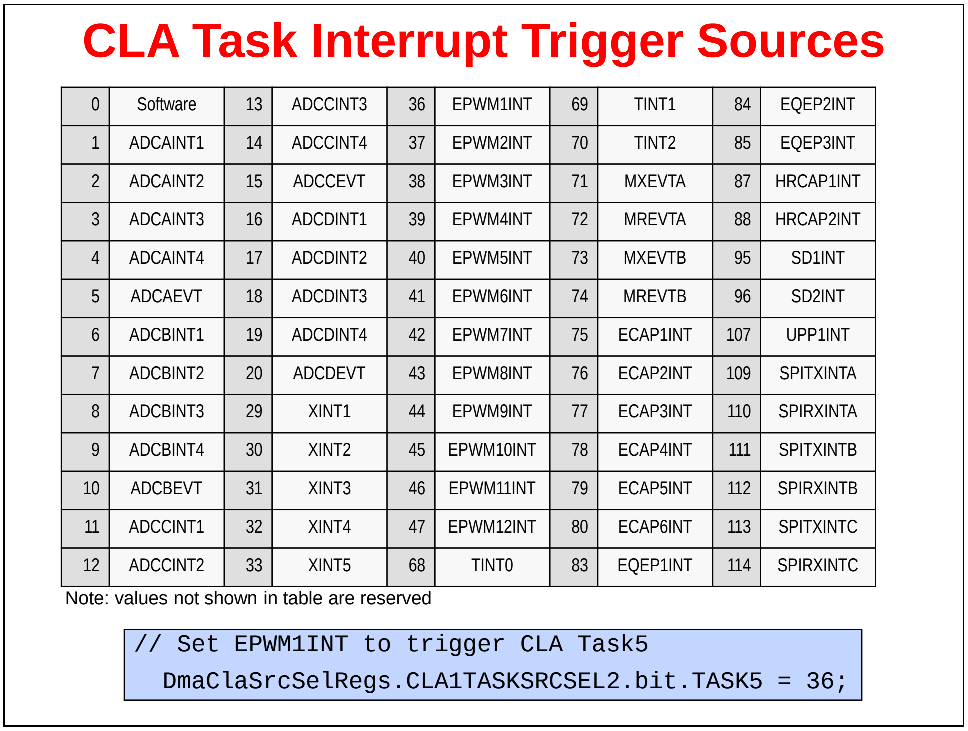
DmaClaSrcSelRegs.CLA1TASKSRCSEL1

31 - 24	23 - 16	15 - 8	7 - 0
TASK4	TASK3	TASK2	TASK1

DmaClaSrcSelRegs.CLA1TASKSRCSEL2

31 - 24	23 - 16	15 - 8	7 - 0
TASK8	TASK7	TASK6	TASK5

Note: CLA1TASKSRCSELLOCK register can be used to lock above registers (lock bit for each register)

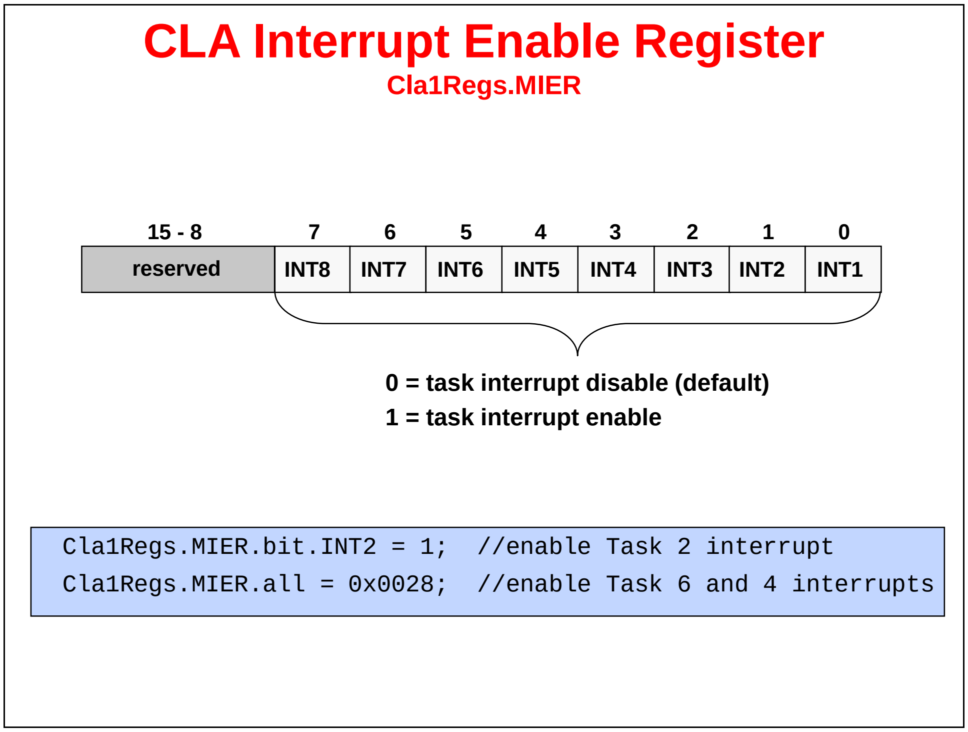


CLA Task Interrupt Trigger Sources

0	Software	13	ADCCINT3	36	EPWM1INT	69	TINT1	84	EQEP2INT
1	ADCAINT1	14	ADCCINT4	37	EPWM2INT	70	TINT2	85	EQEP3INT
2	ADCAINT2	15	ADCCEVT	38	EPWM3INT	71	MXEVTA	87	HRCAP1INT
3	ADCAINT3	16	ADCDINT1	39	EPWM4INT	72	MREVTA	88	HRCAP2INT
4	ADCAINT4	17	ADCDINT2	40	EPWM5INT	73	MXEVTB	95	SD1INT
5	ADCAEVT	18	ADCDINT3	41	EPWM6INT	74	MREVTB	96	SD2INT
6	ADCBINT1	19	ADCDINT4	42	EPWM7INT	75	ECAP1INT	107	UPP1INT
7	ADCBINT2	20	ADCDEVT	43	EPWM8INT	76	ECAP2INT	109	SPITXINTA
8	ADCBINT3	29	XINT1	44	EPWM9INT	77	ECAP3INT	110	SPIRXINTA
9	ADCBINT4	30	XINT2	45	EPWM10INT	78	ECAP4INT	111	SPITXINTB
10	ADCBEVT	31	XINT3	46	EPWM11INT	79	ECAP5INT	112	SPIRXINTB
11	ADCCINT1	32	XINT4	47	EPWM12INT	80	ECAP6INT	113	SPITXINTC
12	ADCCINT2	33	XINT5	68	TINT0	83	EQEP1INT	114	SPIRXINTC

Note: values not shown in table are reserved

```
// Set EPWM1INT to trigger CLA Task5
   DmaClaSrcSelRegs.CLA1TASKSRCSEL2.bit.TASK5 = 36;
```



CLA Interrupt Enable Register

Cla1Regs.MIER

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

0 = task interrupt disable (default)

1 = task interrupt enable

```
Cla1Regs.MIER.bit.INT2 = 1;   //enable Task 2 interrupt
Cla1Regs.MIER.all = 0x0028;   //enable Task 6 and 4 interrupts
```

CLA Initialization

CLA Initialization

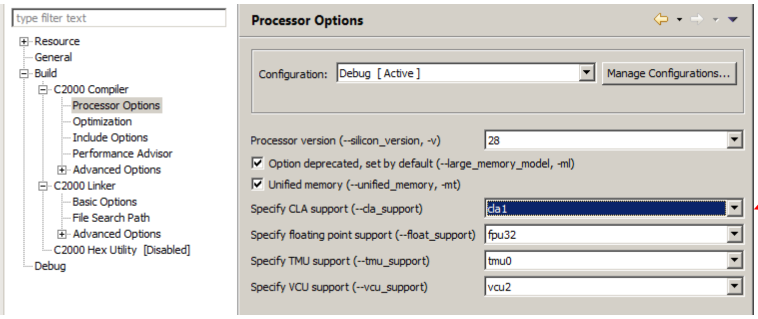
Performed by the CPU during software initialization

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 (PCLKCR3 register)
   - Populate the CLA task interrupt vectors (MVECT1-8 registers)
   - Select the desired task interrupt sources (CLA1TASKSRCSELx register)
   - If desired, set Cla1Regs.MCTL.bit.IACKE = 1 to enable IACK instruction to start tasks using software (avoids EALLOW)
   - Map CLA program RAM and data RAMs to CLA space
4. **Configure desired CLA task completion interrupts in the PIE**
5. **Enable CLA task triggers in the MIER register**
6. **Initialize the desired peripherals to trigger the CLA tasks**

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

Enabling CLA Support in CCS

- **Set the "Specify CLA support" project option to 'cla1'**
- **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**



Processor Options

Configuration: Debug [Active]

Processor version (--silicon_version, -v) 28

☑ Option deprecated, set by default (--large_memory_model, -ml)

☑ Unified memory (--unified_memory, -mt)

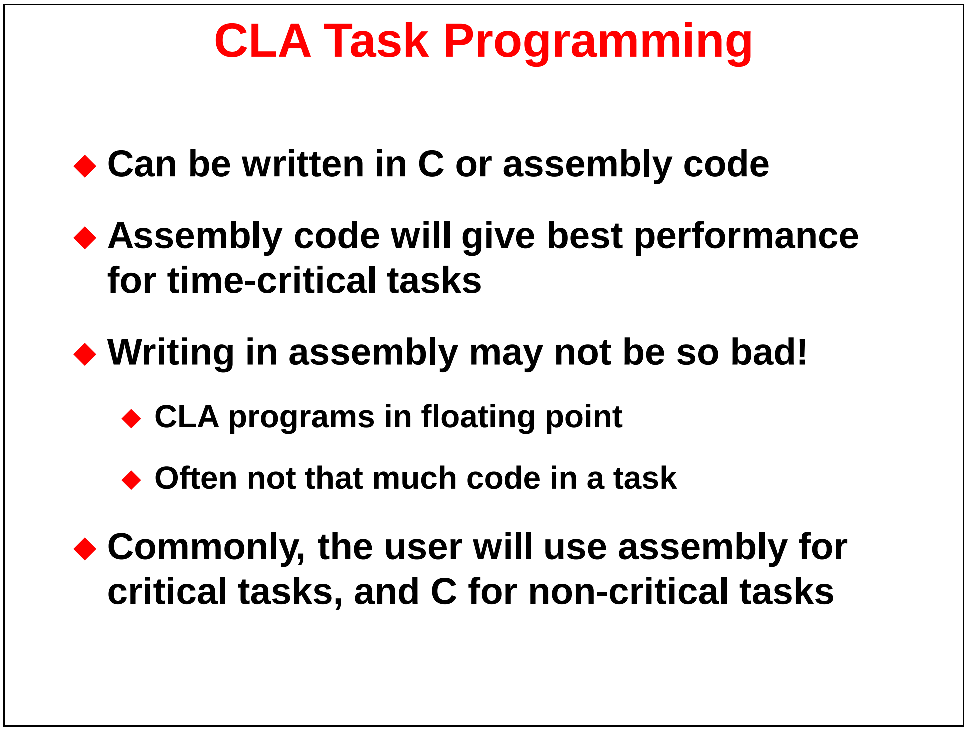
Specify CLA support (--cla_support) cla1

Specify floating point support (--float_support) fpu32

Specify TMU support (--tmu_support) tmu0

Specify VCU support (--vcu_support) vcu2

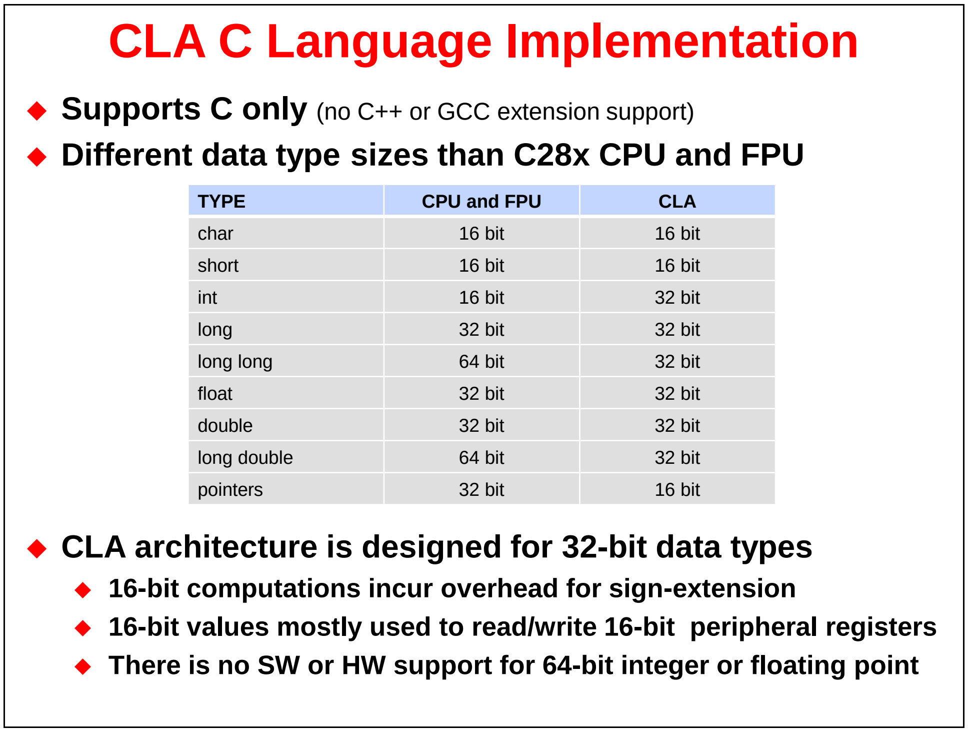
CLA Task Programming



CLA Task Programming

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

CLA C Language Implementation and Restrictions

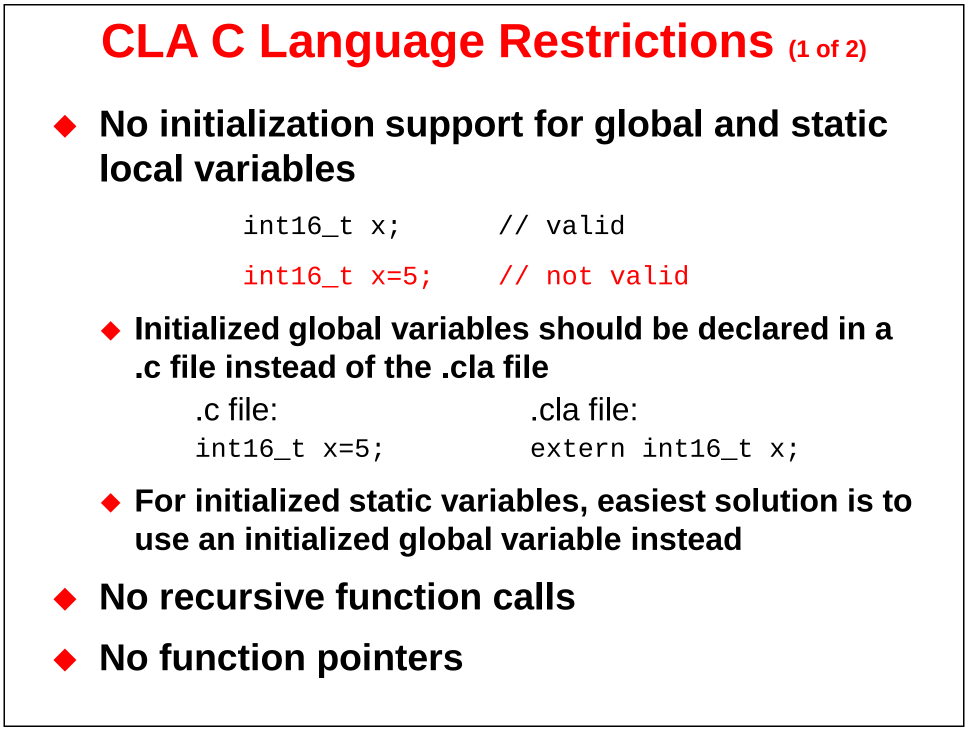


CLA C Language Implementation

- **Supports C only** (no C++ or GCC extension support)
- **Different data type sizes than C28x CPU 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 C Language Restrictions (1 of 2)

- No initialization support for global and static local variables

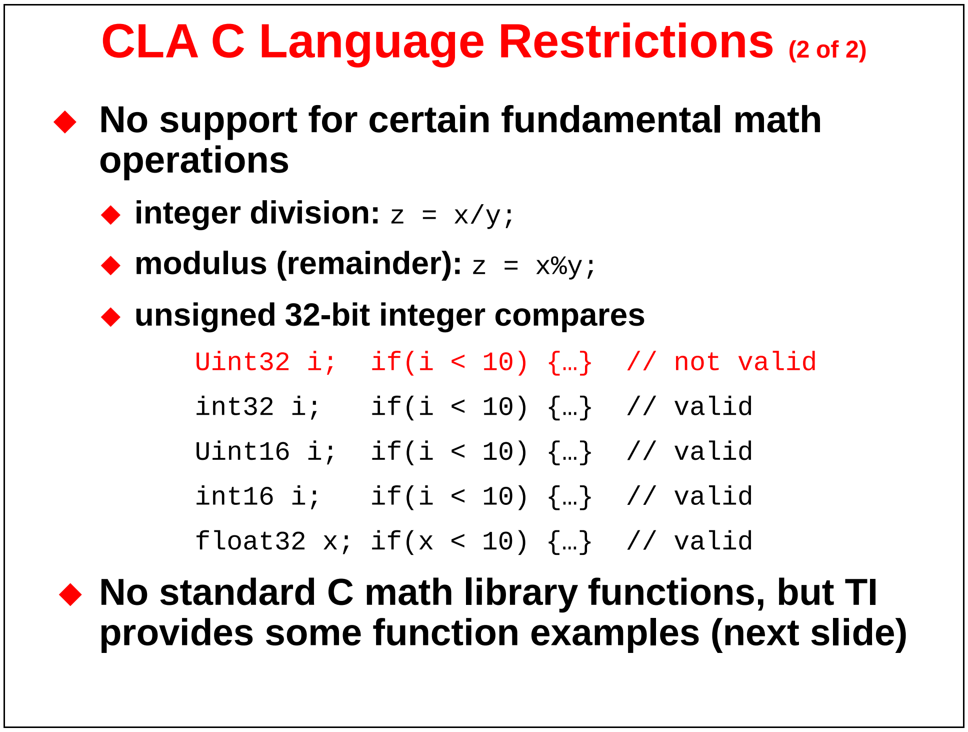
```
int16_t x;      // valid
int16_t x=5;    // not valid
```

  - 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 C Language Restrictions (2 of 2)

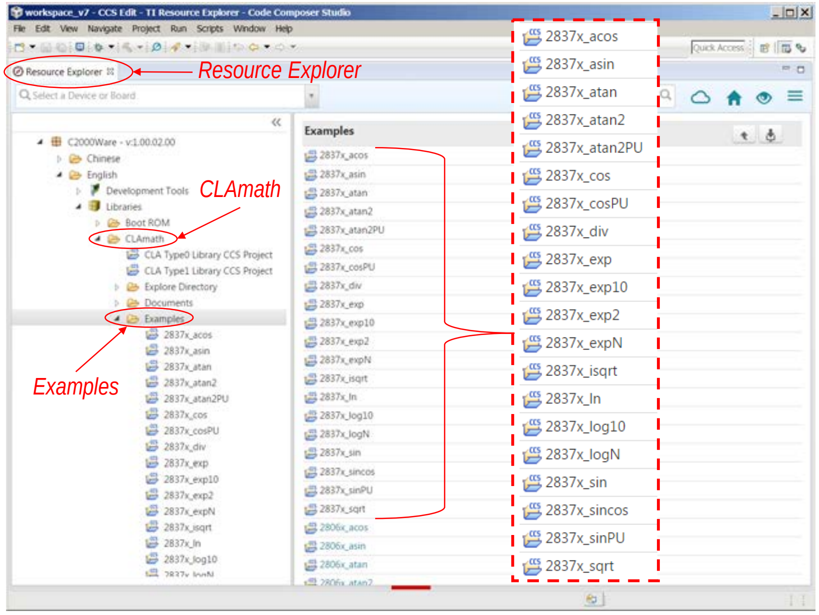
- No support for certain fundamental math operations
  - integer division: `z = x/y;`
  - modulus (remainder): `z = x%y;`
  - unsigned 32-bit integer compares

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

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

C2000Ware™ - CLA Software Support

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

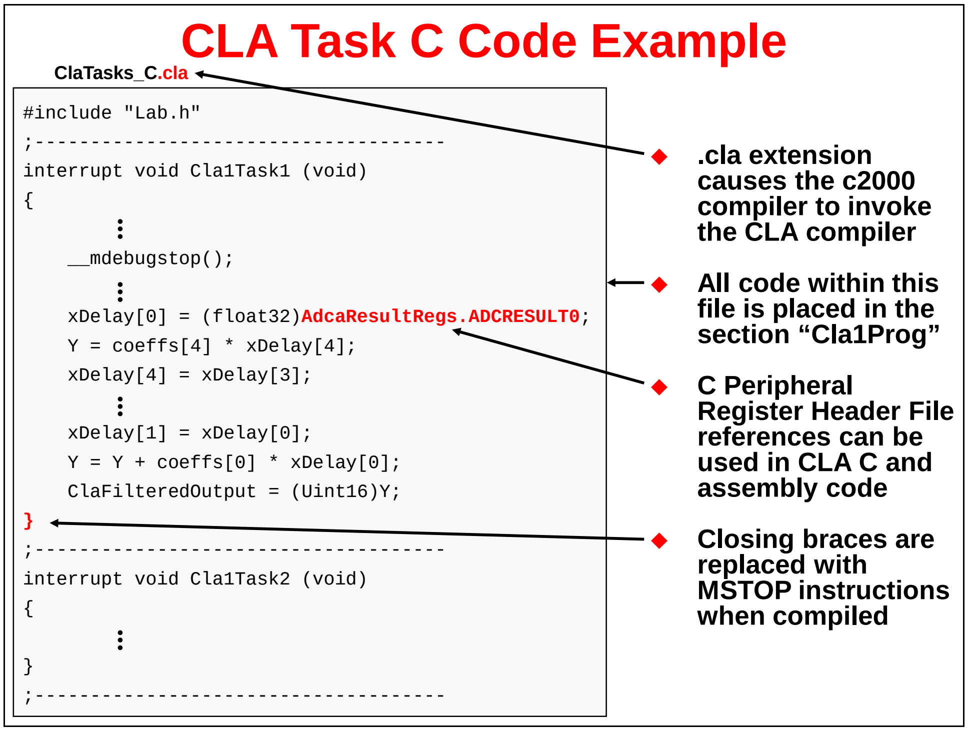


CLA Compiler Scratchpad Memory Area

- **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
     ⋮
}
```



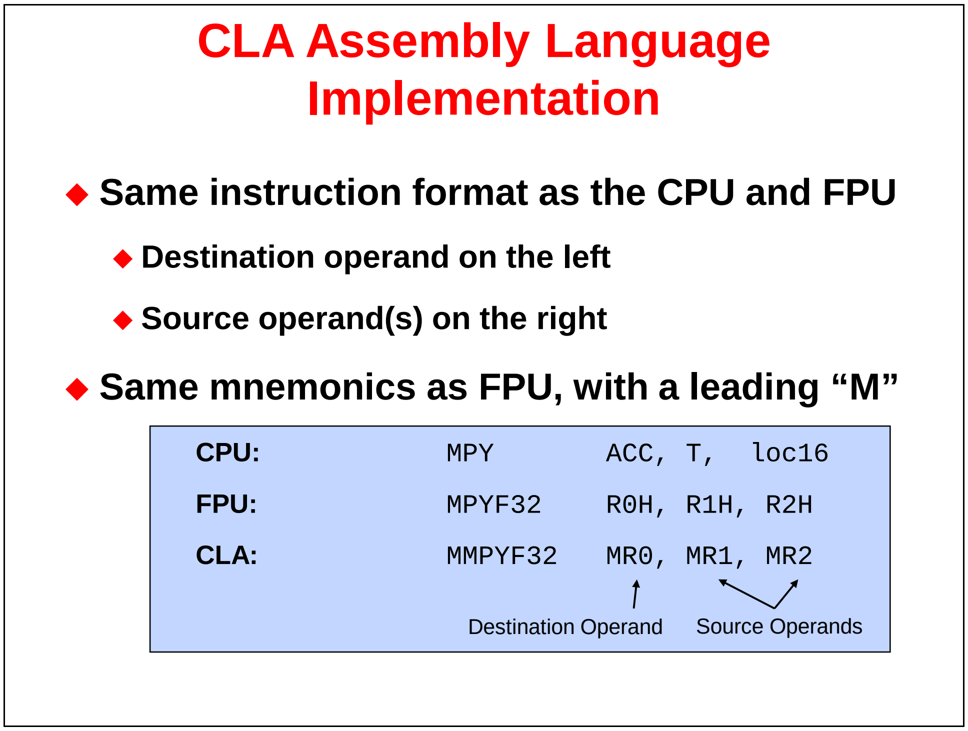
CLA Task C Code Example

ClaTasks_C.cla

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

- .cla extension causes the c2000 compiler to invoke the CLA compiler
- All code within this file is placed in the section “Cla1Prog”
- C Peripheral Register Header File references can be used in CLA C and assembly code
- Closing braces are replaced with MSTOP instructions when compiled

CLA Assembly Language Implementation

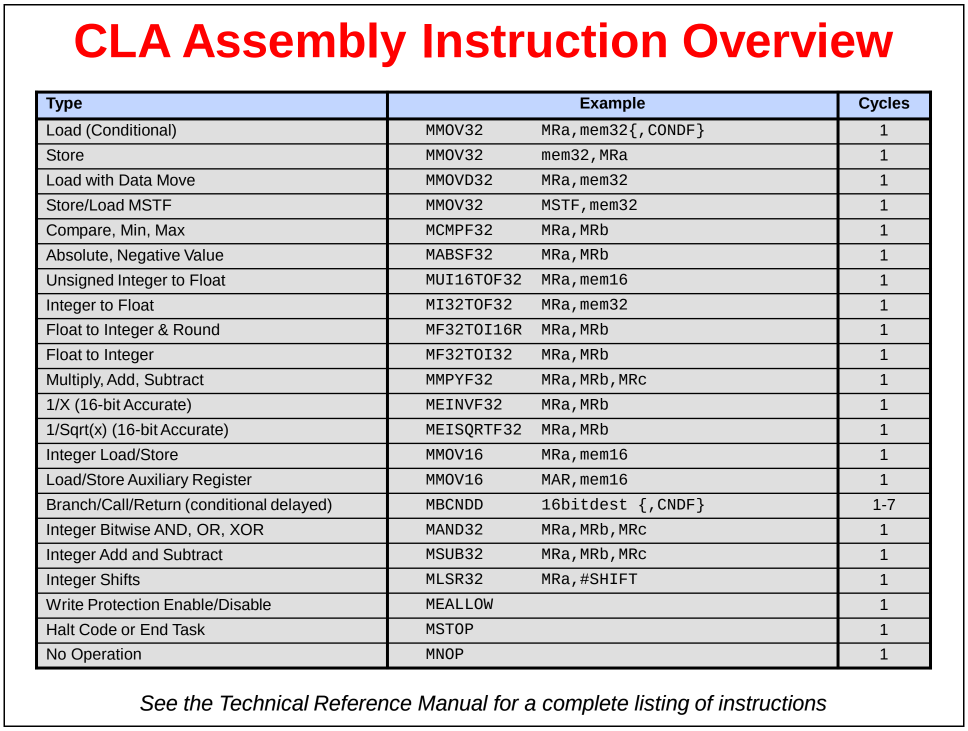


CLA Assembly Language Implementation

- Same instruction format as the CPU and FPU
  - Destination operand on the left
  - Source operand(s) on the right
- Same mnemonics as FPU, with a leading “M”

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

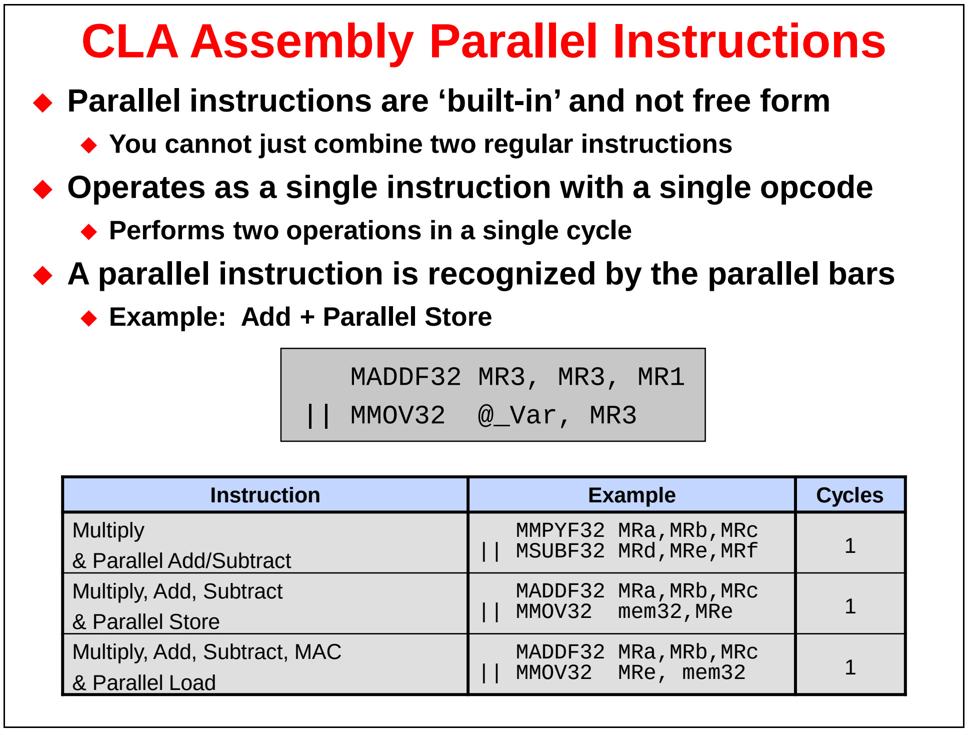
Destination Operand
Source Operands



CLA Assembly Instruction Overview

Type	Example	Cycles
Load (Conditional)	`MMOV32 MRa,mem32{,CONDF}`	1
Store	`MMOV32 mem32,MRa`	1
Load with Data Move	`MMOVD32 MRa,mem32`	1
Store/Load MSTF	`MMOV32 MSTF,mem32`	1
Compare, Min, Max	`MCMPF32 MRa,MRb`	1
Absolute, Negative Value	`MABSF32 MRa,MRb`	1
Unsigned Integer to Float	`MUI16TOF32 MRa,mem16`	1
Integer to Float	`MI32TOF32 MRa,mem32`	1
Float to Integer & Round	`MF32TOI16R MRa,MRb`	1
Float to Integer	`MF32TOI32 MRa,MRb`	1
Multiply, Add, Subtract	`MMPYF32 MRa,MRb,MRc`	1
1/X (16-bit Accurate)	`MEINVF32 MRa,MRb`	1
1/Sqrt(x) (16-bit Accurate)	`MEISQRTF32 MRa,MRb`	1
Integer Load/Store	`MMOV16 MRa,mem16`	1
Load/Store Auxiliary Register	`MMOV16 MAR,mem16`	1
Branch/Call/Return (conditional delayed)	`MBCNDD 16bitdest {,CNDF}`	1-7
Integer Bitwise AND, OR, XOR	`MAND32 MRa,MRb,MRc`	1
Integer Add and Subtract	`MSUB32 MRa,MRb,MRc`	1
Integer Shifts	`MLSR32 MRa,#SHIFT`	1
Write Protection Enable/Disable	`MEALLOW`	1
Halt Code or End Task	`MSTOP`	1
No Operation	`MNOP`	1

See the Technical Reference Manual for a complete listing of instructions



CLA Assembly Parallel Instructions

- Parallel instructions are 'built-in' and not free form
  - You cannot just combine two regular instructions
- Operates as a single instruction with a single opcode
  - Performs two operations in a single cycle
- A parallel instruction is recognized by the parallel bars
  - Example: Add + Parallel Store

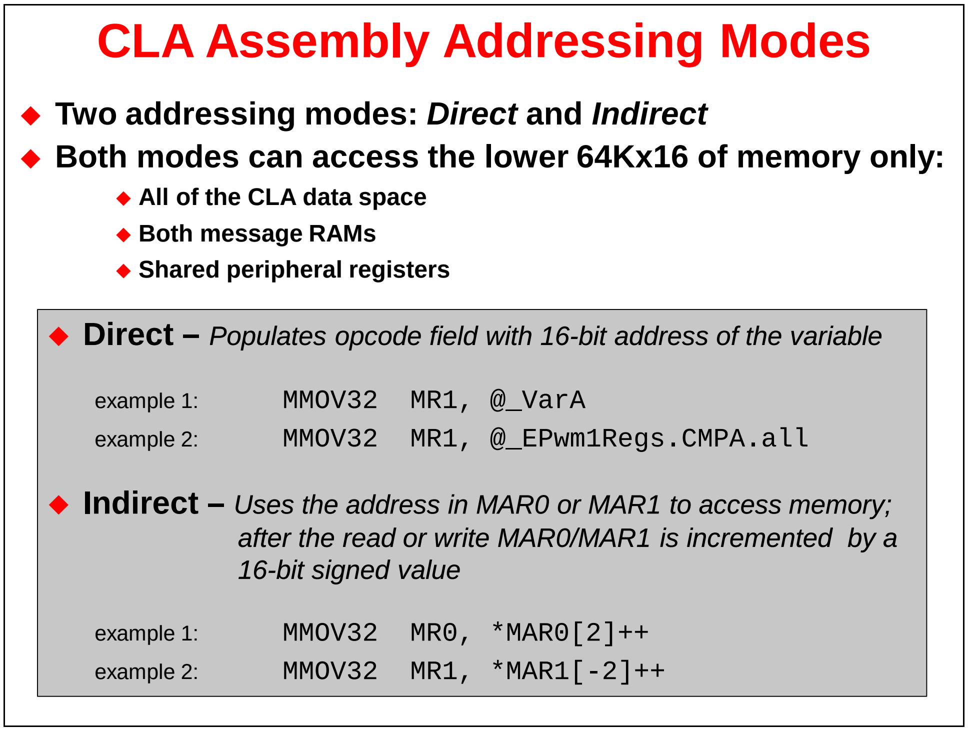
```
   MADDF32 MR3, MR3, MR1
|| MMOV32  @_Var, MR3
```

Instruction	Example	Cycles

Multiply
& Parallel Add/Subtract
	`MMPYF32 MRa,MRb,MRc`
	`|| MSUBF32 MRd,MRe,MRf`
		1

Multiply, Add, Subtract
& Parallel Store
	`MADDF32 MRa,MRb,MRc`
	`|| MMOV32 mem32,MRe`
		1

Multiply, Add, Subtract, MAC
& Parallel Load
	`MADDF32 MRa,MRb,MRc`
	`|| MMOV32 MRe, mem32`
		1



CLA Assembly Addressing Modes

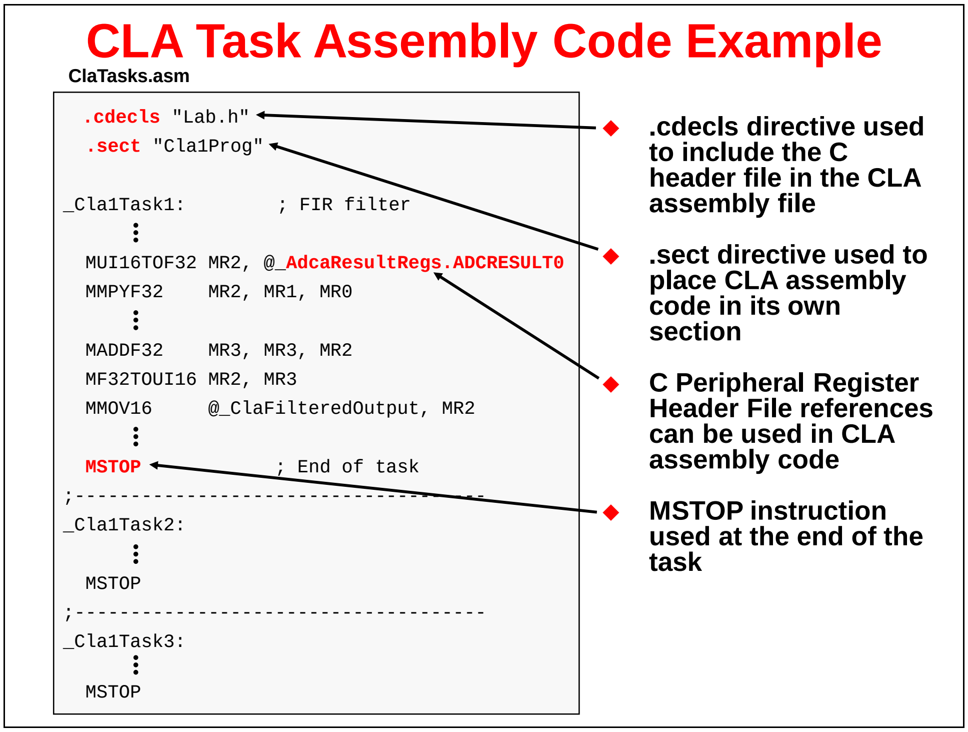
- Two addressing modes: *Direct* and *Indirect*
- Both modes can access the lower 64Kx16 of memory only:
  - All of the CLA data space
  - Both message RAMs
  - Shared peripheral registers

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

```
example 1:      MMOV32   MR1, @_VarA
example 2:      MMOV32   MR1, @_EPwm1Regs.CMPA.all
```

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

```
example 1:      MMOV32   MR0, *MAR0[2]++
example 2:      MMOV32   MR1, *MAR1[-2]++
```

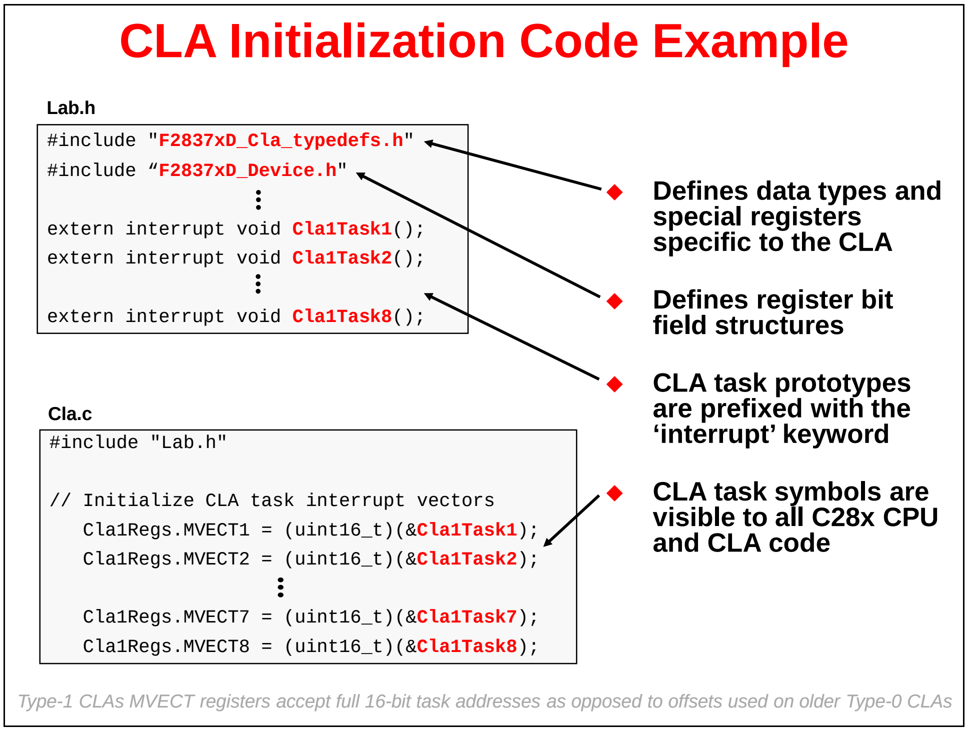


CLA Task Assembly Code Example

ClaTasks.asm

```
  .cdecls "Lab.h"
  .sect "Cla1Prog"

_Cla1Task1:        ; FIR filter
      ⋮
  MUI16TOF32 MR2, @_AdcaResultRegs.ADCRESULT0
  MMPYF32    MR2, MR1, MR0
      ⋮
  MADDF32    MR3, MR3, MR2
  MF32TOUI16 MR2, MR3
  MMOV16     @_ClaFilteredOutput, MR2
      ⋮
  MSTOP              ; End of task
;-------------------------------------
_Cla1Task2:
      ⋮
  MSTOP
;-------------------------------------
_Cla1Task3:
      ⋮
  MSTOP
```

- .cdecls directive used to include the C header file in the CLA assembly file
- .sect directive used to place CLA assembly code in its own section
- C Peripheral Register Header File references can be used in CLA assembly code
- MSTOP instruction used at the end of the task



CLA Initialization Code Example

Lab.h

```
#include "F2837xD_Cla_typedefs.h"
#include “F2837xD_Device.h"
                 ⋮
extern interrupt void Cla1Task1();
extern interrupt void Cla1Task2();
                 ⋮
extern interrupt void Cla1Task8();
```

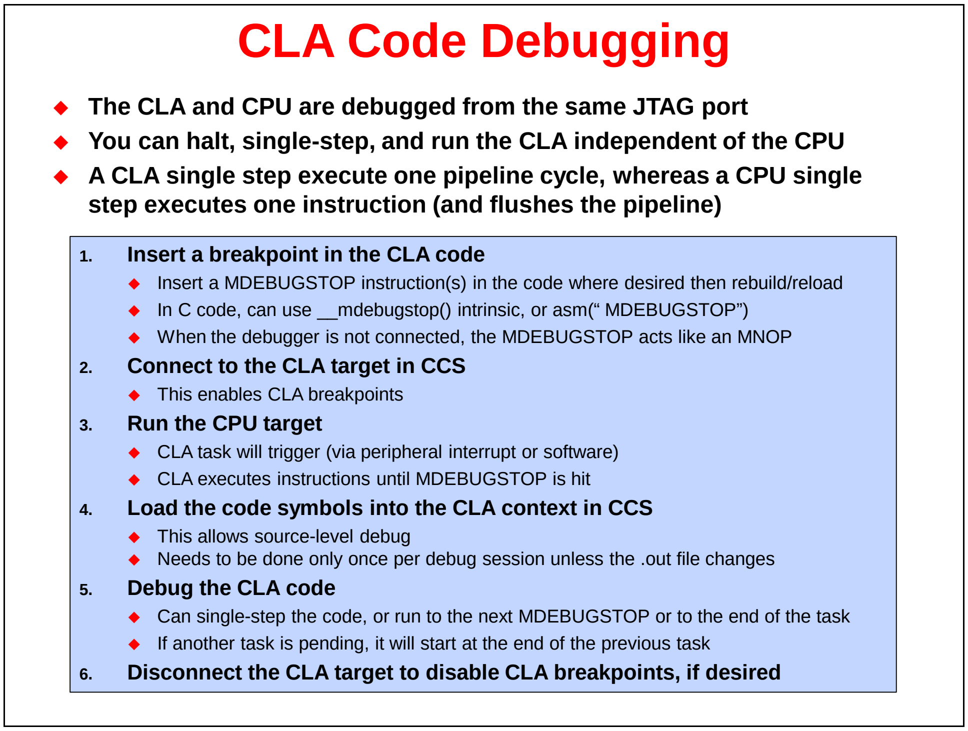
Cla.c

```
#include "Lab.h"

// Initialize CLA task interrupt vectors
   Cla1Regs.MVECT1 = (uint16_t)(&Cla1Task1);
   Cla1Regs.MVECT2 = (uint16_t)(&Cla1Task2);
                    ⋮
   Cla1Regs.MVECT7 = (uint16_t)(&Cla1Task7);
   Cla1Regs.MVECT8 = (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**

Type-1 CLAs MVECT registers accept full 16-bit task addresses as opposed to offsets used on older Type-0 CLAs

CLA Code Debugging



CLA Code Debugging

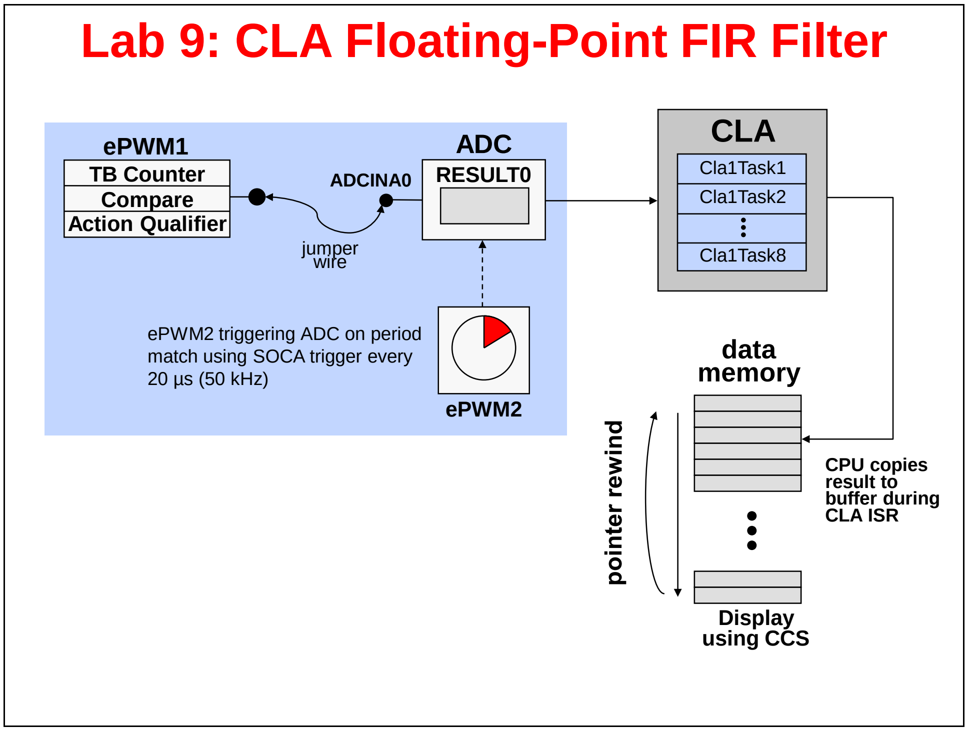
- **The CLA and CPU are debugged from the same JTAG port**
- **You can halt, single-step, and run the CLA independent of the CPU**
- **A CLA single step execute one pipeline cycle, whereas a CPU single step executes one instruction (and flushes the pipeline)**

1. **Insert a breakpoint in the CLA code**
   - Insert a MDEBUGSTOP instruction(s) in the code where desired then rebuild/reload
   - In C code, can use __mdebugstop() intrinsic, or asm(“ MDEBUGSTOP”)
   - When the debugger is not connected, the MDEBUGSTOP 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 MDEBUGSTOP 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 MDEBUGSTOP 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**

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.



Lab 9: CLA Floating-Point FIR Filter

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.

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

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

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

➢ **Procedure**

Open the Project

1. A project named `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 then click `Browse…` next to the "Select search-directory" box. Navigate to: `C:\C28x\Labs\Lab9\cpu01` and click `OK`. 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
Cla_9.c
ClaTasks.asm
ClaTasks_C.cla
CodeStartBranch.asm
Dac.c
DefaultIsr_9_10.c
DelayUs.asm
Dma.c
ECap.c
EPwm.c
F2837xD_Adc.c
F2837xD_GlobalVariableDefs.c
F2837xD_Headers_nonBIOS_cpu1.cmd
Gpio.c
Lab_9.cmd
Main_9.c
PieCtrl.c
PieVect.c
SineTable.c
SysCtrl.c
Watchdog.c
Xbar.c
```

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

Enabling CLA Support in CCS

2. Open the build options by right-clicking on `Lab9` in the Project Explorer window and select `Properties`. Then under "C2000 Compiler" select "`Processor Options`". Notice the "Specify CLA support" is set to `cla1`. This is needed to compile and assemble CLA code. Click `OK` to close the Properties window.

Inspect Lab_9.cmd

3. 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.

4. 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.

5. Open `ClaTasks_C.cla` and notice Task 1 has been configured to run an FIR filter. Within this code the ADC result integer (i.e. the filter input) is being first converted to floating-point, and then at the end the floating-point filter output is being converted back to integer. Also, notice Task 8 is being used to initialize the filter delay line. The .cla extension is recognized by the compiler as a CLA C file, and the compiler will generate CLA specific code.

6. Edit `Cla_9.c` to implement the CLA operation as described in the objective for this lab exercise. Set RAMLS0, RAMLS1, RAMLS2, and RAMLS4 memory blocks as shared between the CPU and CLA. Configure the RAMLS4 memory block to be mapped to CLA program memory space. Configure the RAMLS0, RAMLS1 and RAMLS2 memory blocks to be mapped to CLA data memory space. Note that the RAMLS0 memory block will be used for the CLA C compiler scratchpad. Set Task 1 peripheral interrupt source to ADCAINT1 and set the other Task peripheral interrupt source inputs to “software” (i.e. none). Enable CLA Task 1 interrupt. Enable the use of the IACK instruction to trigger a task, and then enable Task 8 interrupt.

7. Open `Main_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()`.

8. 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.

9. 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.

10. Using the “PIE Interrupt Assignment Table” find the location for the CLA Task 1 interrupt “`CLA1_1`” and fill in the following information:

    PIE group #:___________   # within group:__________

    This information will be used in the next step.

11. Modify the end of `Cla_9.c` to do the following:

    - Enable the “`CLA1_1`” interrupt in the PIE (Hint: use the `PieCtrlRegs` structure)
    - Enable the appropriate core interrupt in the IER register

12. Open and inspect `DefaultIsr_9_10.c`. Notice that this file contains the CLA interrupt service routine. Save all modified files.

Build and Load

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

14. Click the “`Debug`” button (green bug). A Launching Debug Session window will open. Select only CPU1 to load the program on (i.e. ***uncheck*** CPU2), and then click `OK`. Then the CCS Debug perspective view should open, the program will load automatically, and you should now be at the start of `main()`. If the device has been power cycled since the last lab exercise, be sure to configure the boot mode to EMU_BOOT_SARAM using the Scripts menu.

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

Note: For the next step, check to be sure that the jumper wire connecting PWM1A (header J4, pin #40) to ADCINA0 (header J3, pin #30) is in place on the LaunchPad.

15. Run the code in real-time mode using the Script function: `Scripts` → `Realtime Emulation Control` → `Run_Realtime_with_Reset`. Open and watch the memory browser window update. Verify that the ADC result buffer contains updated values.

16. Setup a dual-time graph of the filtered and unfiltered ADC results buffer. Click: `Tools` → `Graph` → `Dual Time` and set the following values:

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

17. The graphical display should show the filtered PWM waveform in the Dual Time A display and the unfiltered waveform in the Dual Time B display. You should see that the results match the previous lab exercise.

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

Change Task 1 to FIR Filter in Assembly

Previously, the initialization and filter tasks were implemented in C. In this part, we will not be using the C implementation of the FIR filter located at Task 1 in `ClaTasks_C.cla`. Instead, we will add `ClaTasks.asm` to the project and use the assembly implementation of the FIR filter located at Task 1 in this file. The CLA setup code in `Cla_9.c` and the filter initialization C-code located at Task 8 in `ClaTasks_C.cla` will not need to change.

19. Open `ClaTasks_C.cla` and at the beginning of Task 1 change the #if preprocessor directive from 1 to 0. The sections of code between the #if and #endif will not be compiled. This has the same effect as commenting out this code. We need to do this to avoid a conflict with the Task 1 in `ClaTask.asm` file.
20. Add (copy) `ClaTasks.asm` to project from `C:\C28x\Labs\Lab9\source`.
21. Open `ClaTasks.asm` and notice that the .cdecls directive is being used to include the C header file in the CLA assembly file. Therefore, we can use the Peripheral Register Header File references in the CLA assembly code. Next, notice Task 1 has been configured to run an FIR filter. Within this code special instructions have been used to convert the ADC result integer (i.e. the filter input) to floating-point and the floating-point filter output back to integer. Notice at Task 2 the assembly preprocessor .if directive is set to 0. The assembly preprocessor .endif directive is located at the end of Task 8. With this setting, Tasks 2 through 8 will not be assembled, again avoiding a conflict with Task 2 through 8 in the `ClaTasks_C.cla` file. Save all modified files.

Build and Load

22. 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 – Test the CLA Operation (Tasks in C and ASM)

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

Change All Tasks to Assembly

In this part, we will be using the assembly implementation of the FIR filter and filter delay line initialization routine located at Task 1 and Task 8, respectively, in the `ClaTasks.asm` file. The setup in `Cla_9.c` will remain the same. The `ClaTasks_C.cla` is no longer needed and will be excluded from the build. As a result, the CLA C compiler is not used and the CLA C compiler scratchpad area allocated by the linker command file will not be needed.

25. Open `ClaTasks.asm` and at the beginning of Task 2 change the assembly preprocessor .if directive to 1. Recall that the assembly preprocessor .endif directive is located at the end of Task 8. Now Task 2 through Task 8 will be assembled, along with Task 1.
26. Exclude `ClaTasks_C.cla` from the project to avoid conflicts with `ClaTasks.asm`. In the Project Explorer window right-click on `ClaTasks_C.cla` and select "Exclude from Build". This file is no longer needed since all of the tasks are now in `ClaTasks.asm`.

Build and Load

27. 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 – Test the CLA Operation (Tasks in ASM)

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

Terminate Debug Session and Close Project

30. 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.

31. Next, close the project by right-clicking on `Lab9` in the Project Explorer window and select `Close Project`.

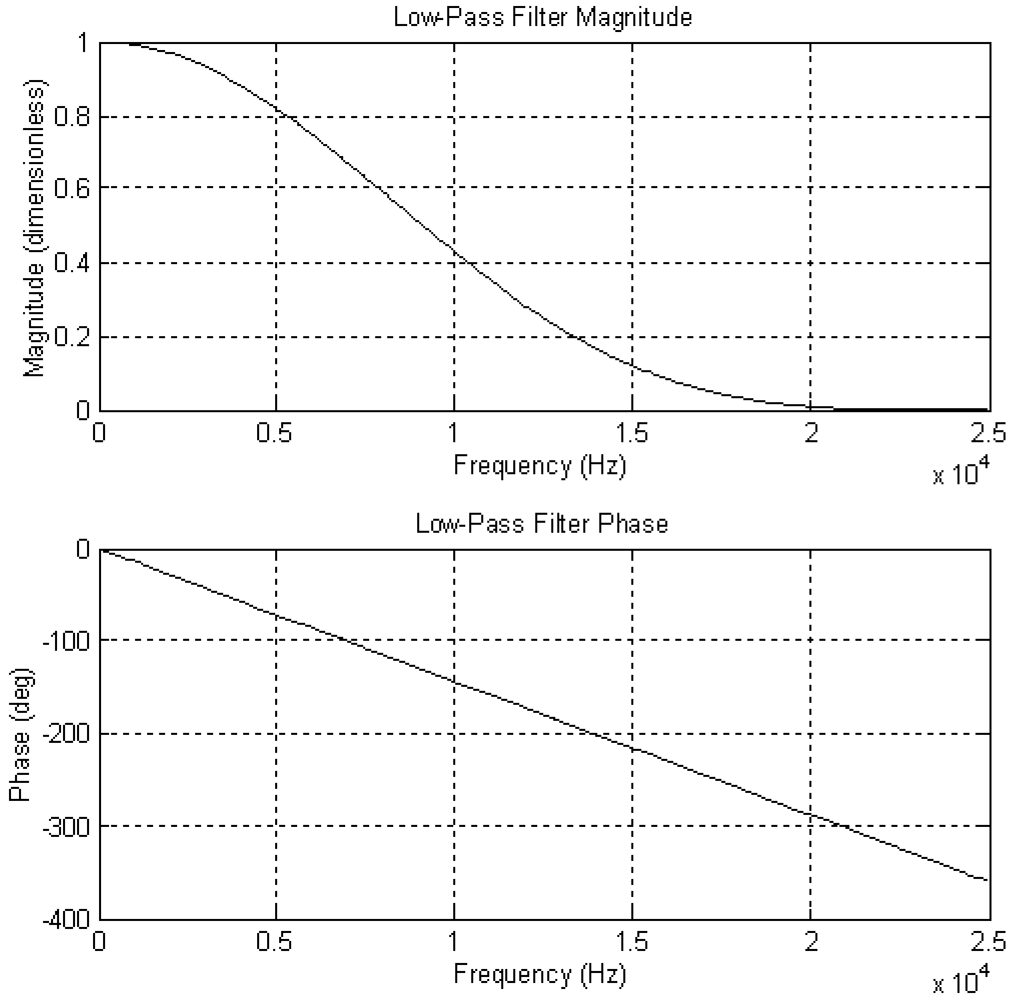
End of Exercise

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

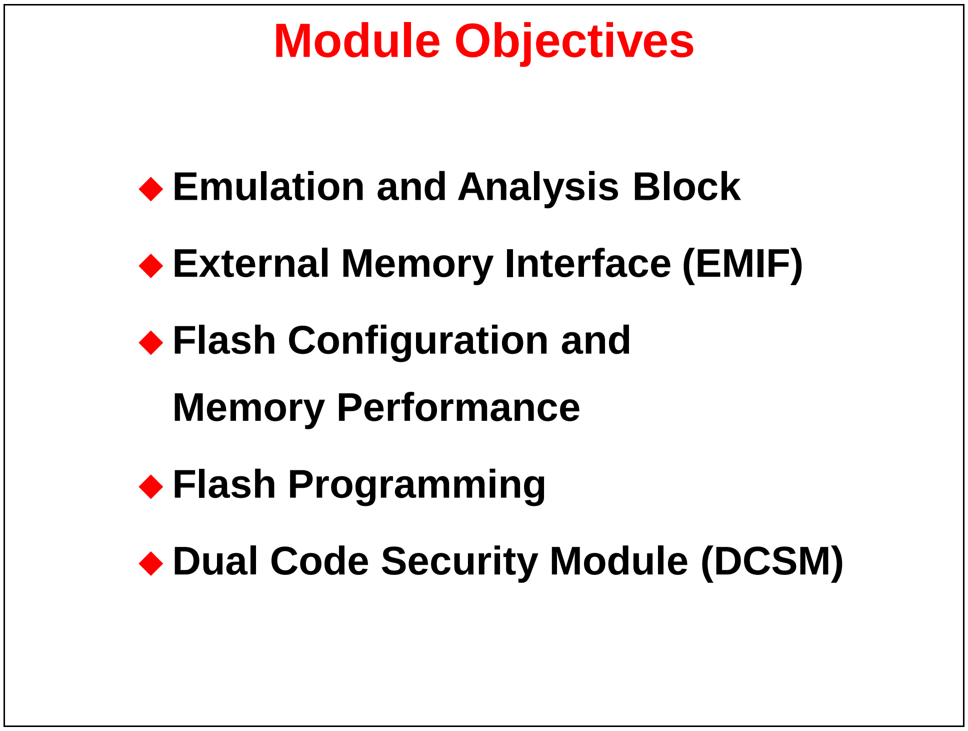


System Design

Introduction

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

Module Objectives



Module Objectives

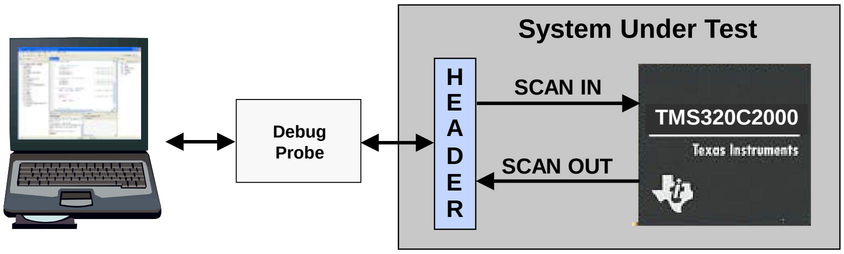
- Emulation and Analysis Block
- External Memory Interface (EMIF)
- Flash Configuration and Memory Performance
- Flash Programming
- Dual Code Security Module (DCSM)

Chapter Topics

Emulation and Analysis Block

JTAG Emulation System

(based on IEEE 1149.1 Boundary Scan Standard)



Some Available Debug Probes

XDS100 CLASS -
BlackHawk: USB100
Spectrum Digital: XDS100

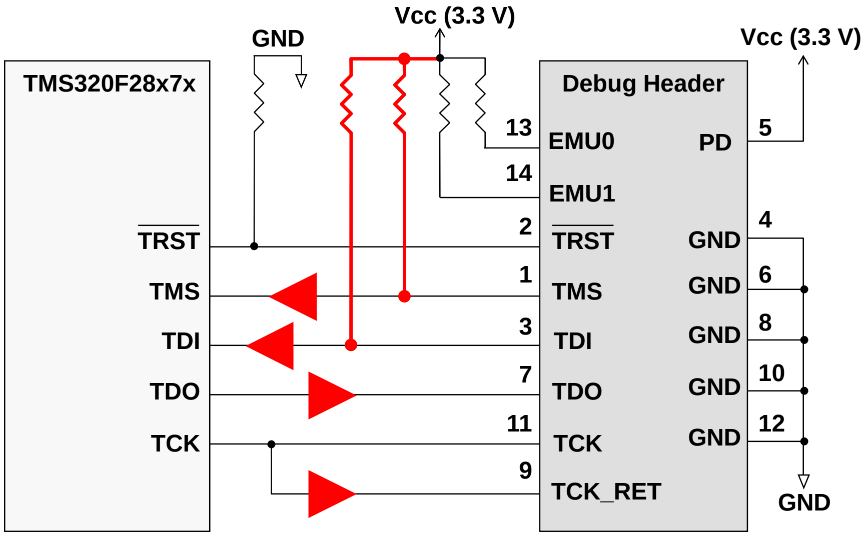
These debug probes are ultra-low cost, and have an unlimited free license for use with Code Composer Studio

XDS200 CLASS -
BlackHawk: USB200
Spectrum Digital: XDS200

These debug probes offer a balance of low cost with good performance compared to the XDS100 class debug probes

Note: XDS510 CLASS debug probes are not recommended (obsolete); and for C2000, XDS560 CLASS debug probes are not recommended since they are expensive and do not offer any advantage over the XDS200 CLASS debug probes

Emulation Connections to the Device



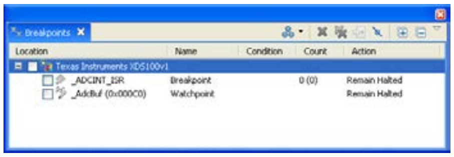
🟥 = If distance between device and header is greater than 6 inches

On-Chip Emulation Analysis Block:
Capabilities

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

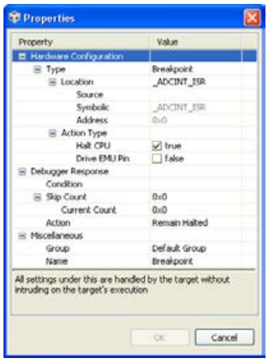
Analysis Configuration		Debug Activity
2 Hardware Breakpoints	⇒	Halt on a specified instruction (for debugging in Flash)
2 Address Watchpoints	⇒	A memory location is getting corrupted; halt the processor when any value is written to this location
1 Address Watchpoint with Data	⇒	Halt program execution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

On-Chip Emulation Analysis Block:
Hardware Breakpoints and Watchpoints



Location	Name	Condition	Count	Action
Texas Instruments XDS100v1				
□ _ADCINT_ISR	Breakpoint		0 (0)	Remain Halted
□ _AdcBuf (0x000C0)	Watchpoint			Remain Halted

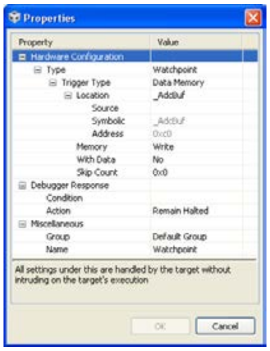
View → Breakpoints



Property	Value
Hardware Configuration	
Type	Breakpoint
Location	_ADCINT_ISR
Source	
Symbolic	_ADCINT_ISR
Address	0x0
Action Type	
Halt CPU	☑ true
Drive EMU Pin	□ false
Debugger Response	
Condition	
Skip Count	0x0
Current Count	0x0
Action	Remain Halted
Miscellaneous	
Group	Default Group
Name	Breakpoint

All settings under this are handled by the target without intruding on the target's execution

Hardware Breakpoint Properties



Property	Value
Hardware Configuration	
Type	Watchpoint
Trigger Type	Data Memory
Location	_AdcBuf
Source	
Symbolic	_AdcBuf
Address	0xc0
Memory	Write
With Data	No
Skip Count	0x0
Debugger Response	
Condition	
Action	Remain Halted
Miscellaneous	
Group	Default Group
Name	Watchpoint

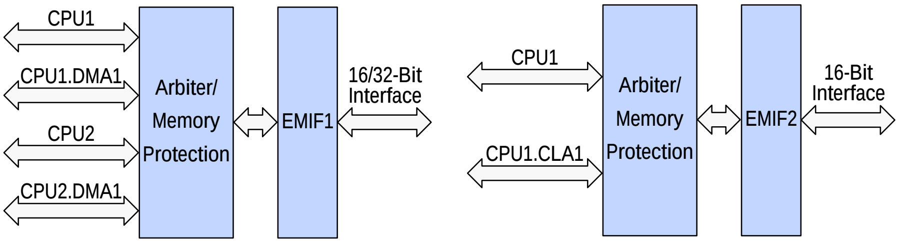
All settings under this are handled by the target without intruding on the target's execution

Hardware Watchpoint Properties

External Memory Interface (EMIF)

External Memory Interface (EMIF)

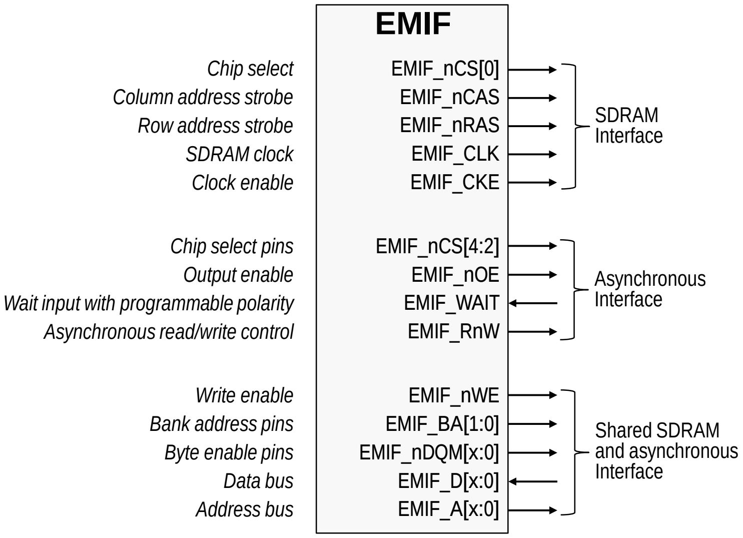
- Provides a means for the CPU, DMA, and CLA to connect to various memory devices
- Support for synchronous (SDRAM) and asynchronous (SRAM, NOR Flash) memories
- F2837xD includes two EMIFs
  - EMIF1 – 16/32-bit interface shared between CPU1 and CPU2
  - EMIF2 – 16-bit interface dedicated to CPU1

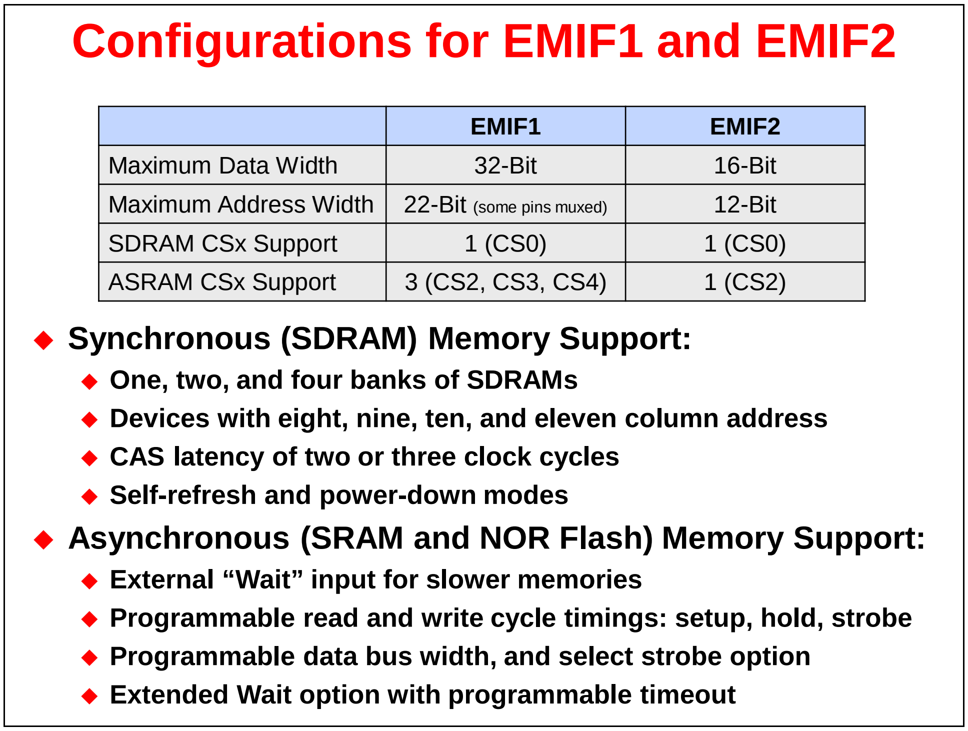


EMIF1 shared between CPU1 & CPU2

EMIF2 dedicated to CPU1

External Memory Interface Signals

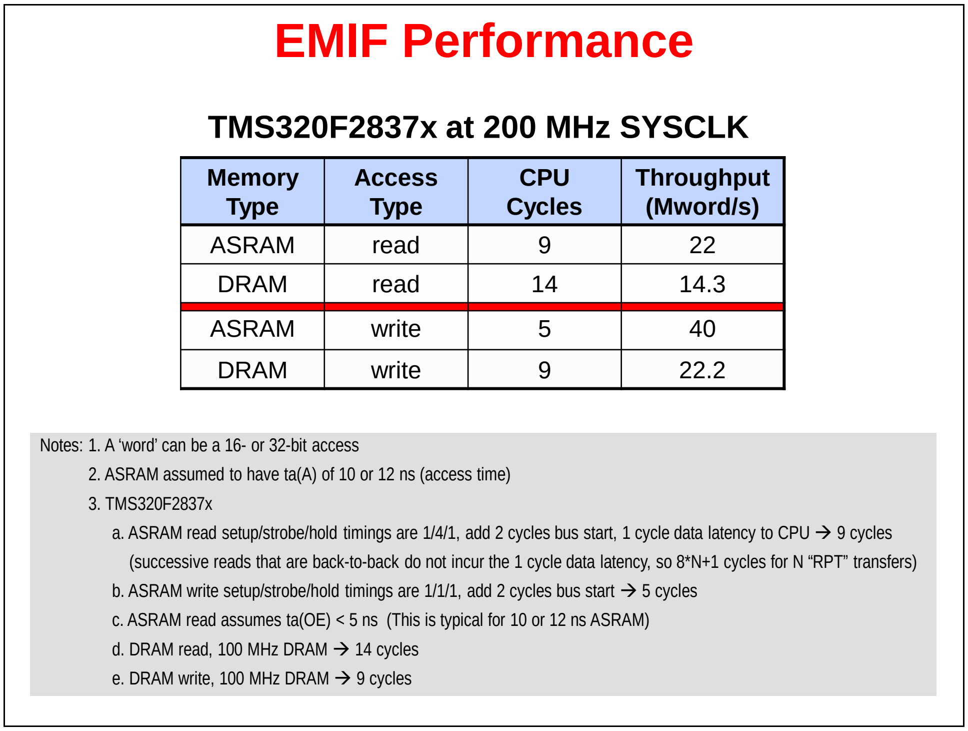




Configurations for EMIF1 and EMIF2

	EMIF1	EMIF2
Maximum Data Width	32-Bit	16-Bit
Maximum Address Width	22-Bit (some pins muxed)	12-Bit
SDRAM CSx Support	1 (CS0)	1 (CS0)
ASRAM CSx Support	3 (CS2, CS3, CS4)	1 (CS2)

- **Synchronous (SDRAM) Memory Support:**
  - **One, two, and four banks of SDRAMs**
  - **Devices with eight, nine, ten, and eleven column address**
  - **CAS latency of two or three clock cycles**
  - **Self-refresh and power-down modes**
- **Asynchronous (SRAM and NOR Flash) Memory Support:**
  - **External "Wait" input for slower memories**
  - **Programmable read and write cycle timings: setup, hold, strobe**
  - **Programmable data bus width, and select strobe option**
  - **Extended Wait option with programmable timeout**



EMIF Performance

TMS320F2837x at 200 MHz SYSCLK

Memory Type	Access Type	CPU Cycles	Throughput (Mword/s)
ASRAM	read	9	22
DRAM	read	14	14.3
ASRAM	write	5	40
DRAM	write	9	22.2

Notes: 1. A 'word' can be a 16- or 32-bit access

2. ASRAM assumed to have ta(A) of 10 or 12 ns (access time)

3. TMS320F2837x

a. ASRAM read setup/strobe/hold timings are 1/4/1, add 2 cycles bus start, 1 cycle data latency to CPU → 9 cycles (successive reads that are back-to-back do not incur the 1 cycle data latency, so 8*N+1 cycles for N "RPT" transfers)

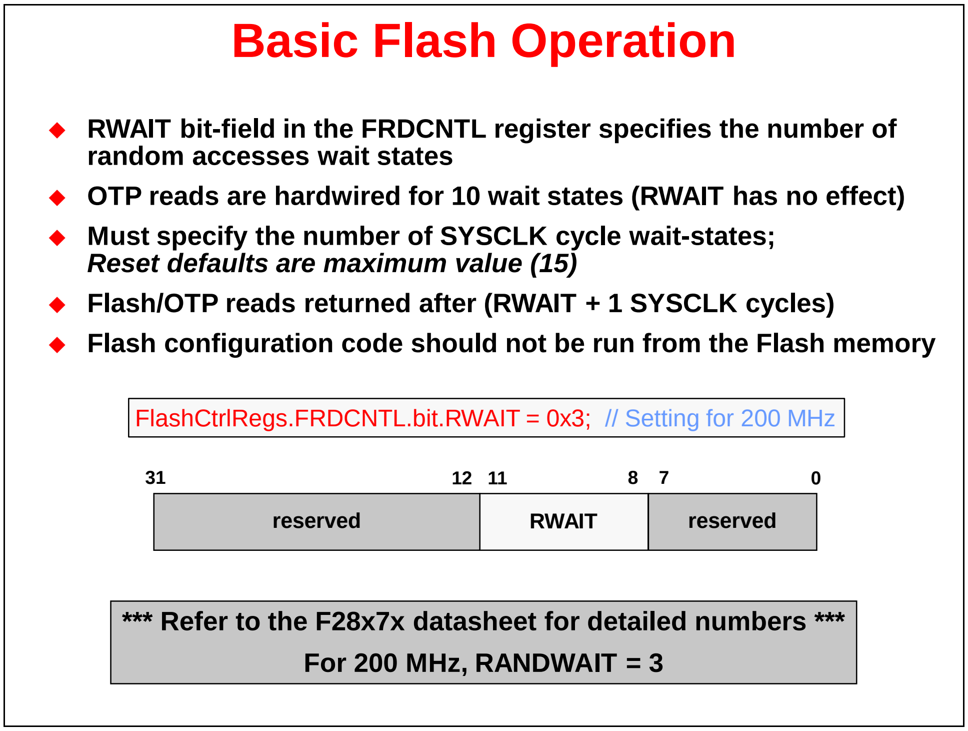
b. ASRAM write setup/strobe/hold timings are 1/1/1, add 2 cycles bus start → 5 cycles

c. ASRAM read assumes ta(OE) < 5 ns (This is typical for 10 or 12 ns ASRAM)

d. DRAM read, 100 MHz DRAM → 14 cycles

e. DRAM write, 100 MHz DRAM → 9 cycles

Flash Configuration and Memory Performance



Basic Flash Operation

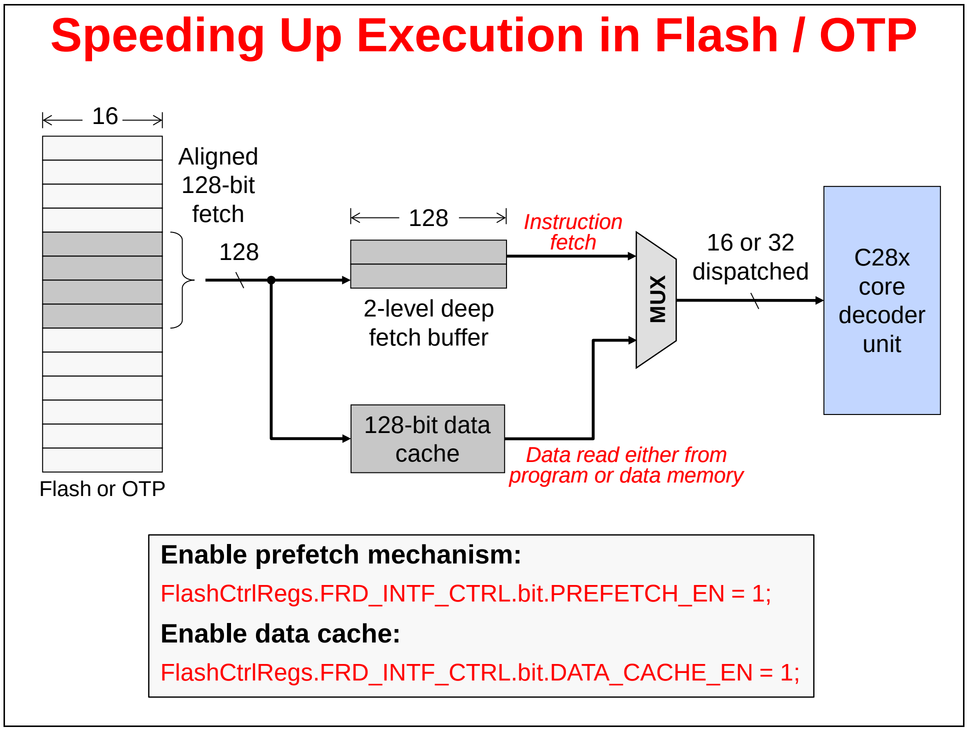
- RWAIT bit-field in the FRDCNTL register specifies the number of random accesses wait states
- OTP reads are hardwired for 10 wait states (RWAIT has no effect)
- Must specify the number of SYSCLK cycle wait-states; *Reset defaults are maximum value (15)*
- Flash/OTP reads returned after (RWAIT + 1 SYSCLK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

```
FlashCtrlRegs.FRDCNTL.bit.RWAIT = 0x3;  // Setting for 200 MHz
```

31 – 12	11 – 8	7 – 0
reserved	RWAIT	reserved

***** Refer to the F28x7x datasheet for detailed numbers *****

For 200 MHz, RANDWAIT = 3



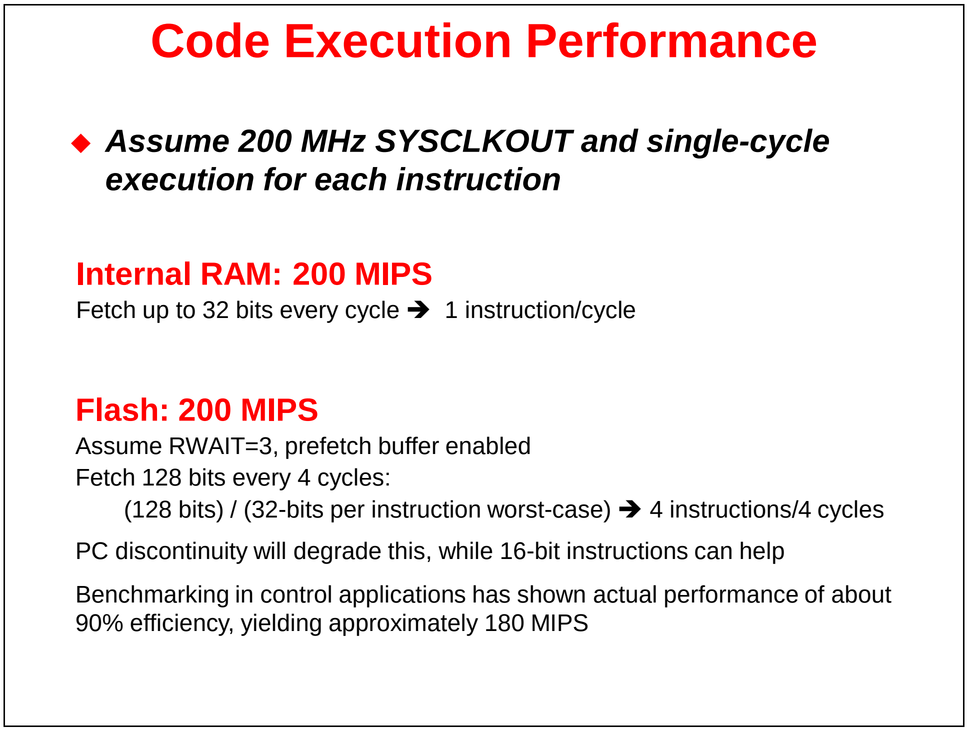
Speeding Up Execution in Flash / OTP

Enable prefetch mechanism:

```
FlashCtrlRegs.FRD_INTF_CTRL.bit.PREFETCH_EN = 1;
```

Enable data cache:

```
FlashCtrlRegs.FRD_INTF_CTRL.bit.DATA_CACHE_EN = 1;
```



Code Execution Performance

- ***Assume 200 MHz SYSCLKOUT and single-cycle execution for each instruction***

Internal RAM: 200 MIPS

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

Flash: 200 MIPS

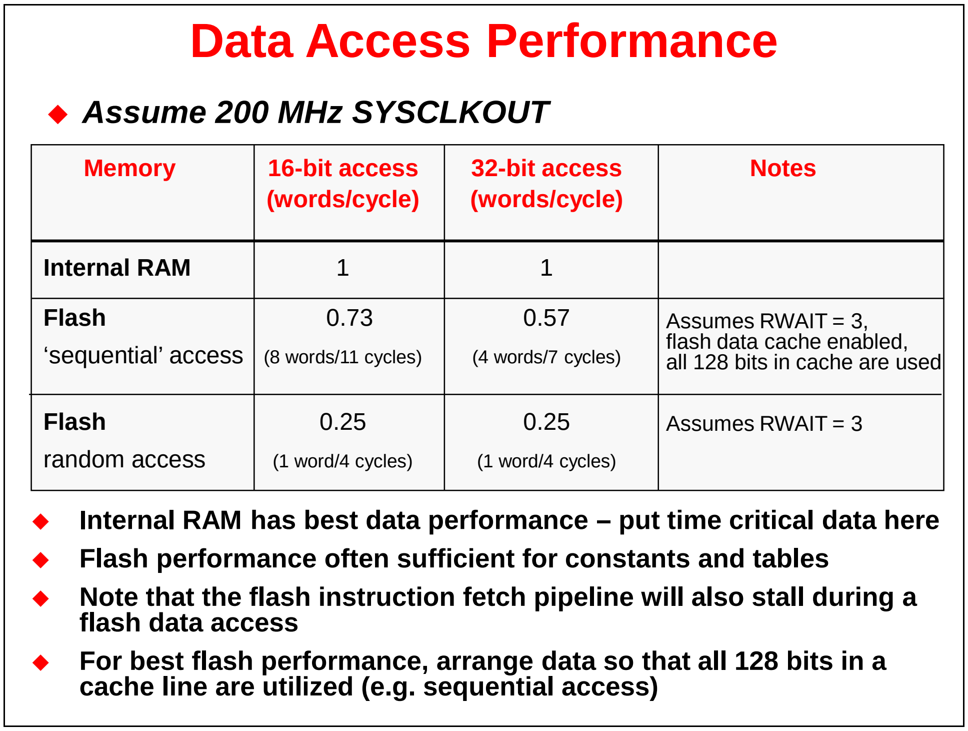
Assume RWAIT=3, prefetch buffer enabled

Fetch 128 bits every 4 cycles:

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

PC discontinuity will degrade this, while 16-bit instructions can help

Benchmarking in control applications has shown actual performance of about 90% efficiency, yielding approximately 180 MIPS



Data Access Performance

- ***Assume 200 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 = 3, 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 = 3

- **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)

Setting Flash charge pump fallback power mode to active:

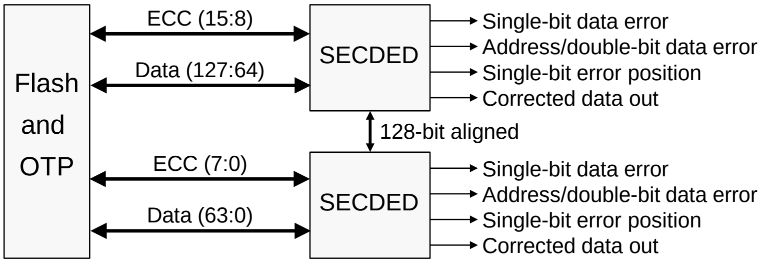
```
FlashCtrlRegs.FPAC1.bit.PMPPWR = 0x1;  // 0: sleep, 1: active
```

Setting fallback power mode to active:

```
FlashCtrlRegs.FBFALLBACK.bit.BNKPWR0 = 0x3;  // 0: sleep, 1: standby,
                                             // 2: reserved, 3: active
```

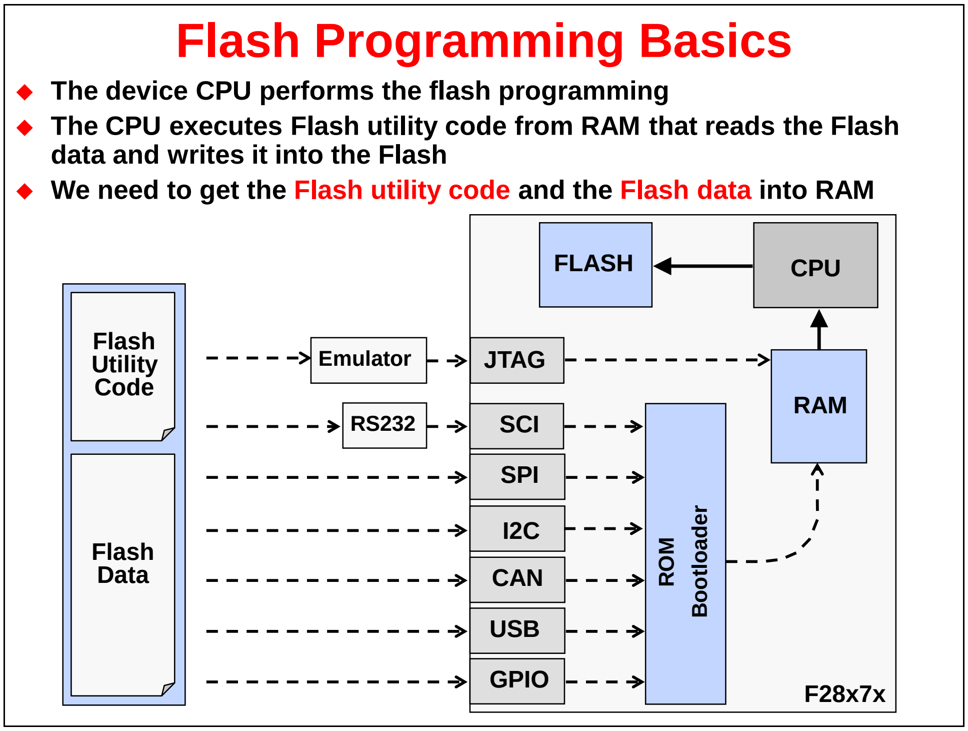
Error Correction Code (ECC) Protection

- Provides capability to screen out Flash/OTP memory faults (enabled at reset)
- Single error correction and double error detection (SECDED)
- 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 SECDED 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



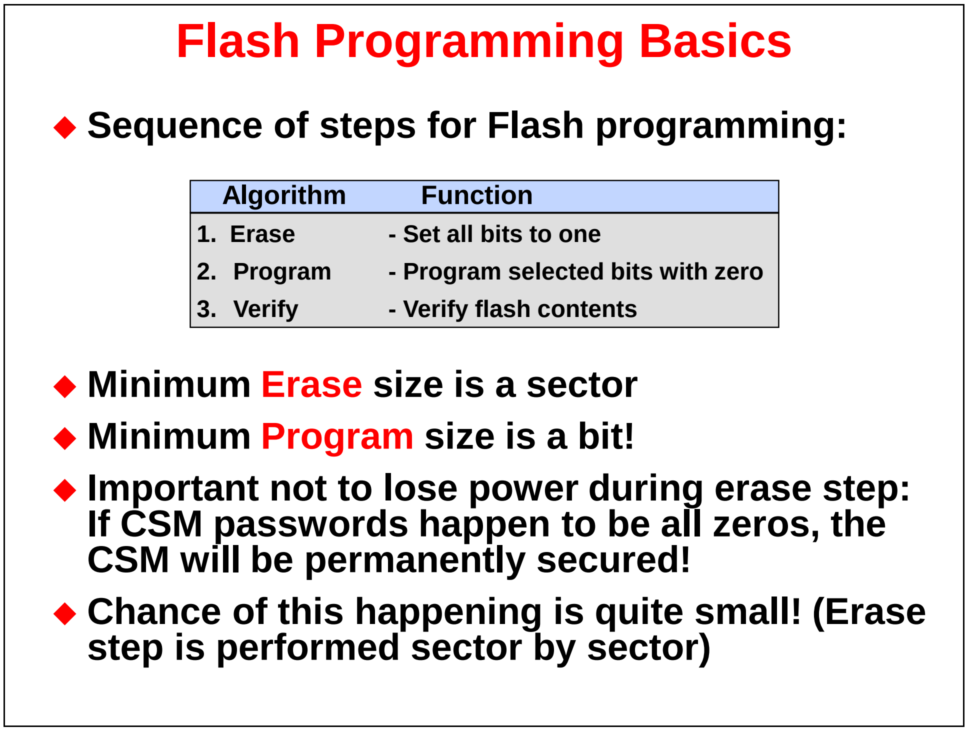
```
FlashEccRegs.ECC_ENABLE.bit.ENABLE = 0xA;  // 0xA enable; other values disable
```

Flash Programming



Flash Programming Basics

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

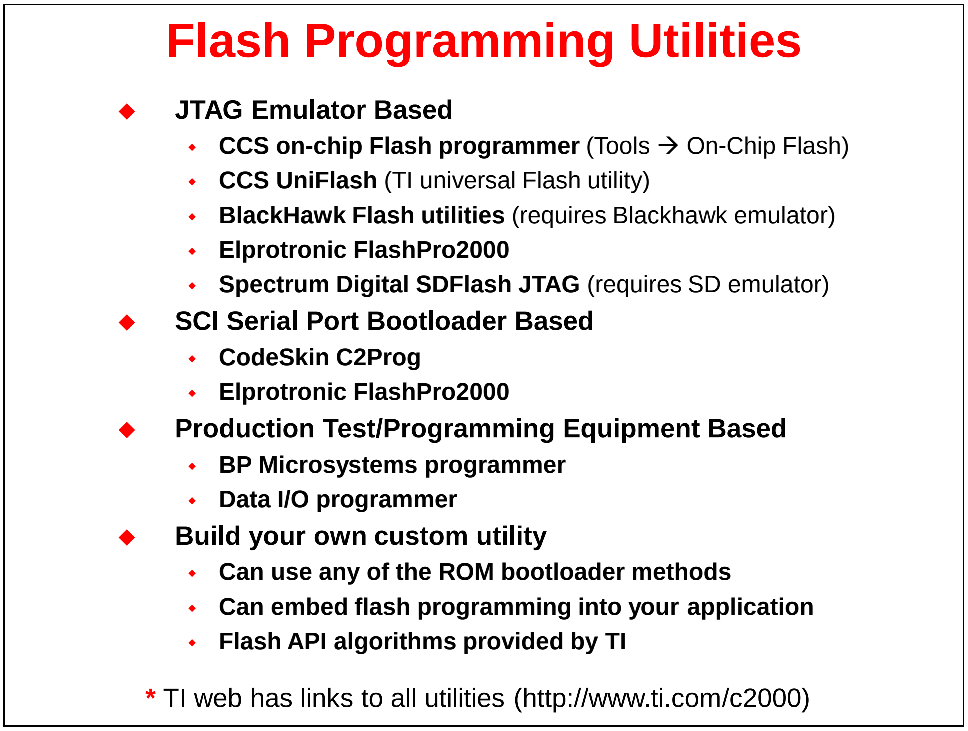


Flash Programming Basics

- Sequence of steps for Flash programming:

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

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

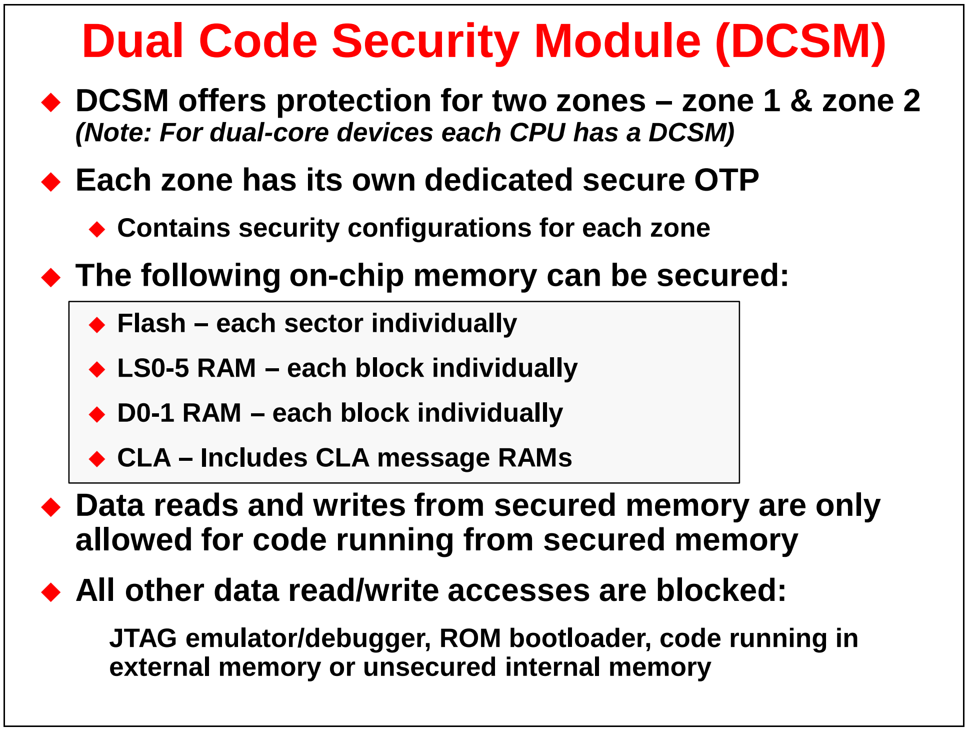


Flash Programming Utilities

- **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**
  - **Spectrum Digital SDFlash JTAG** (requires SD emulator)
- **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)

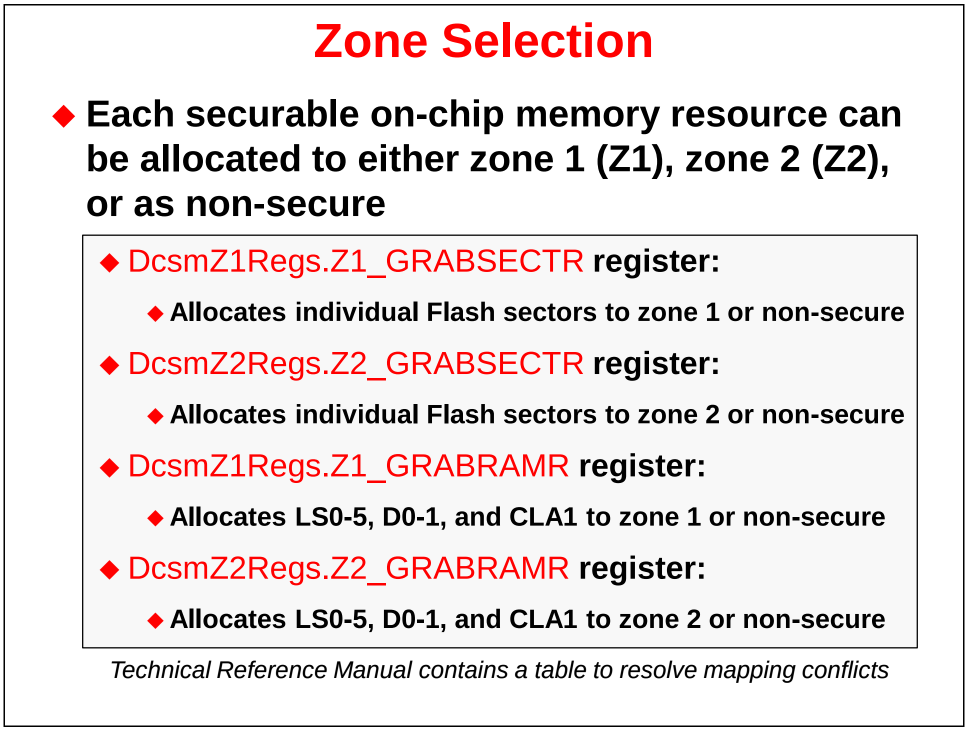
Dual Code Security Module (DCSM)



Dual Code Security Module (DCSM)

- **DCSM offers protection for two zones – zone 1 & zone 2**
  (Note: For dual-core devices each CPU has a DCSM)
- **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**
  - **LS0-5 RAM – each block individually**
  - **D0-1 RAM – each block individually**
  - **CLA – Includes CLA message RAMs**
- **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 LS0-5, D0-1, and CLA1 to zone 1 or non-secure**
  - DcsmZ2Regs.Z2_GRABRAMR **register:**
    - **Allocates LS0-5, D0-1, and CLA1 to zone 2 or non-secure**

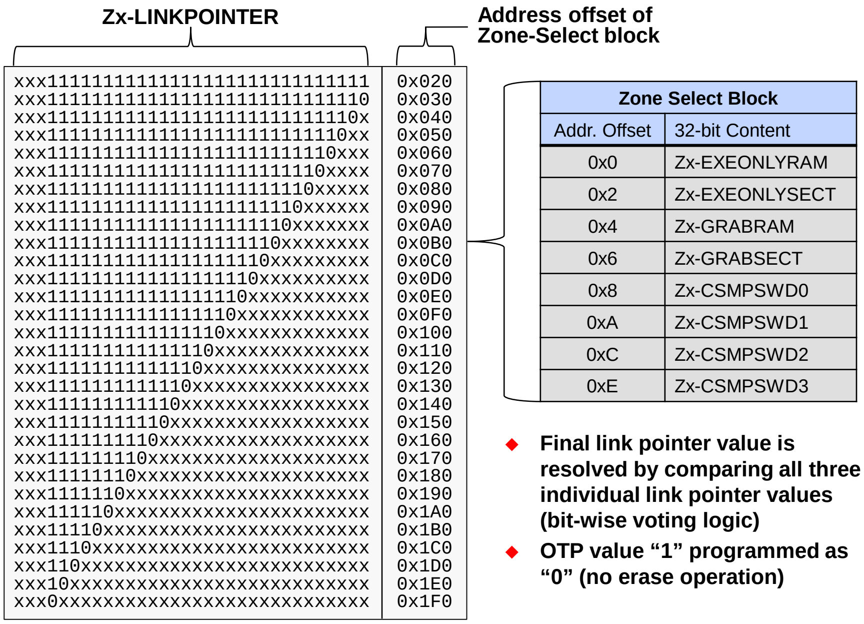
Technical Reference Manual contains a table to resolve mapping conflicts

CSM Passwords

Zx_CSMPSWD0
Zx_CSMPSWD1
Zx_CSMPSWD2
Zx_CSMPSWD3

- **Each zone is secured by its own 128-bit (four 32-bit words) user defined CSM password**
- **Passwords for each zone is stored in its dedicated OTP location**
  - ***Location based on a zone-specific link pointer***
- **128-bit CSMKEY registers are used to secure and unsecure the device**
- **Password locations for each zone can be locked and secured by programming PSWDLOCK fields in the OTP with any value other than “1111” (0xF)**

Zone Select Bits in OTP



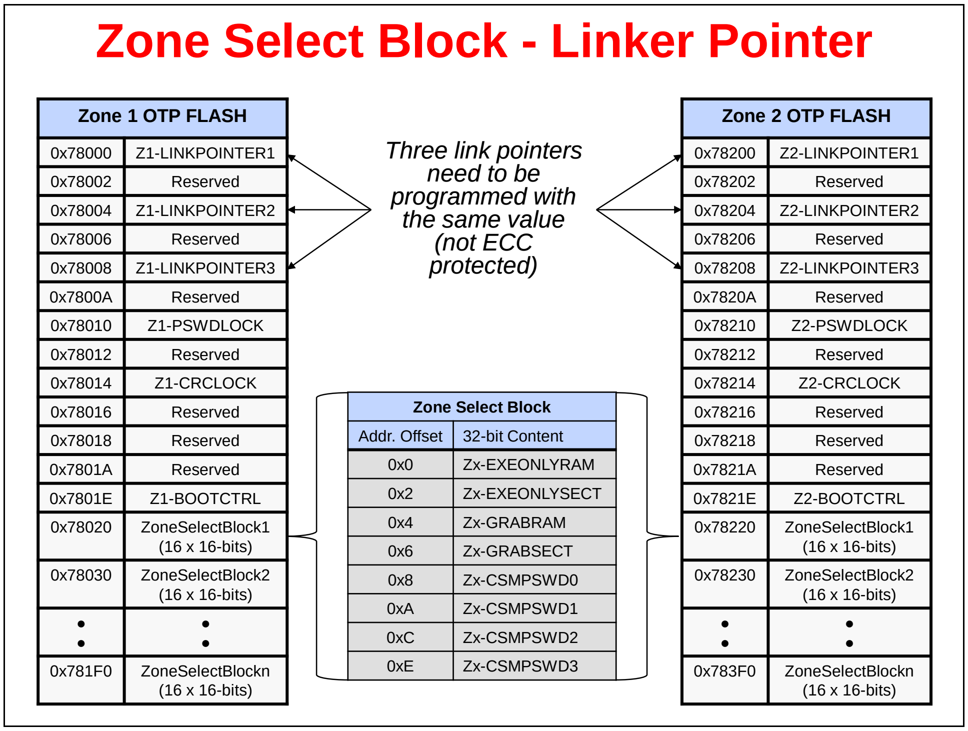
Zx-LINKPOINTER

Address offset of Zone-Select block

```
xxx11111111111111111111111111111  0x020
xxx11111111111111111111111111110  0x030
xxx1111111111111111111111111110x  0x040
xxx111111111111111111111111110xx  0x050
xxx11111111111111111111111110xxx  0x060
xxx1111111111111111111111110xxxx  0x070
xxx111111111111111111111110xxxxx  0x080
xxx11111111111111111111110xxxxxx  0x090
xxx1111111111111111111110xxxxxxx  0x0A0
xxx111111111111111111110xxxxxxxx  0x0B0
xxx11111111111111111110xxxxxxxxx  0x0C0
xxx1111111111111111110xxxxxxxxxx  0x0D0
xxx111111111111111110xxxxxxxxxxx  0x0E0
xxx11111111111111110xxxxxxxxxxxx  0x0F0
xxx1111111111111110xxxxxxxxxxxxx  0x100
xxx111111111111110xxxxxxxxxxxxxx  0x110
xxx11111111111110xxxxxxxxxxxxxxx  0x120
xxx1111111111110xxxxxxxxxxxxxxxx  0x130
xxx111111111110xxxxxxxxxxxxxxxxx  0x140
xxx11111111110xxxxxxxxxxxxxxxxxx  0x150
xxx1111111110xxxxxxxxxxxxxxxxxxx  0x160
xxx111111110xxxxxxxxxxxxxxxxxxxx  0x170
xxx11111110xxxxxxxxxxxxxxxxxxxxx  0x180
xxx1111110xxxxxxxxxxxxxxxxxxxxxx  0x190
xxx111110xxxxxxxxxxxxxxxxxxxxxxx  0x1A0
xxx11110xxxxxxxxxxxxxxxxxxxxxxxx  0x1B0
xxx1110xxxxxxxxxxxxxxxxxxxxxxxxx  0x1C0
xxx110xxxxxxxxxxxxxxxxxxxxxxxxxx  0x1D0
xxx10xxxxxxxxxxxxxxxxxxxxxxxxxxx  0x1E0
xxx0xxxxxxxxxxxxxxxxxxxxxxxxxxxx  0x1F0
```

Zone Select Block	
Addr. Offset	32-bit Content
0x0	Zx-EXEONLYRAM
0x2	Zx-EXEONLYSECT
0x4	Zx-GRABRAM
0x6	Zx-GRABSECT
0x8	Zx-CSMPSWD0
0xA	Zx-CSMPSWD1
0xC	Zx-CSMPSWD2
0xE	Zx-CSMPSWD3

- **Final link pointer value is resolved by comparing all three individual link pointer values (bit-wise voting logic)**
- **OTP value “1” programmed as “0” (no erase operation)**



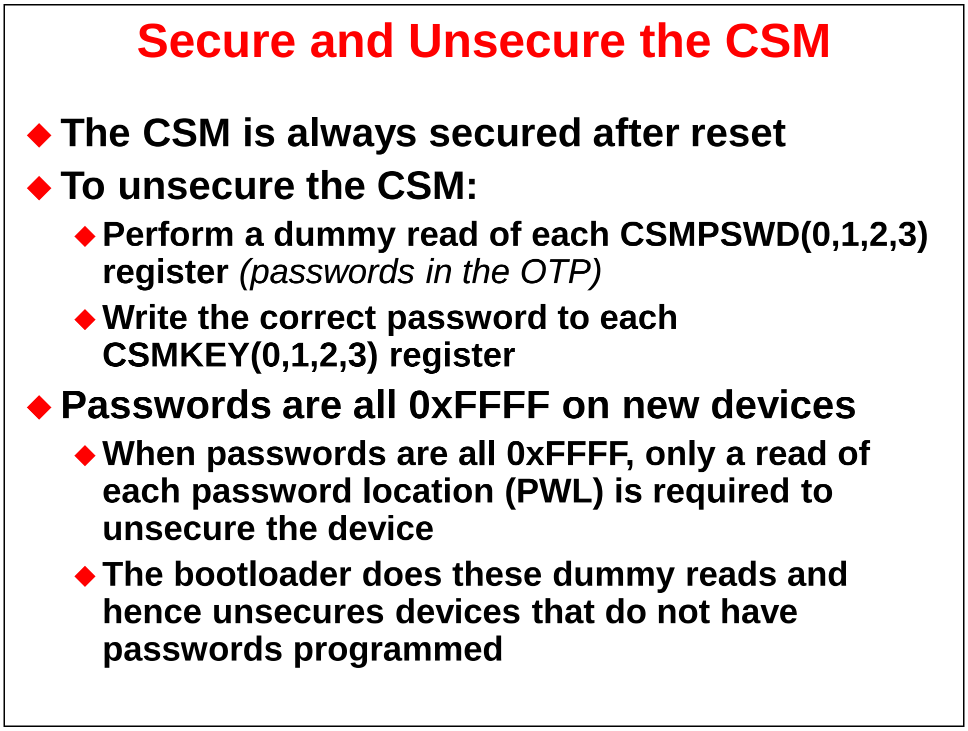
Zone Select Block - Linker Pointer

Zone 1 OTP FLASH	
0x78000	Z1-LINKPOINTER1
0x78002	Reserved
0x78004	Z1-LINKPOINTER2
0x78006	Reserved
0x78008	Z1-LINKPOINTER3
0x7800A	Reserved
0x78010	Z1-PSWDLOCK
0x78012	Reserved
0x78014	Z1-CRCLOCK
0x78016	Reserved
0x78018	Reserved
0x7801A	Reserved
0x7801E	Z1-BOOTCTRL
0x78020	ZoneSelectBlock1 (16 x 16-bits)
0x78030	ZoneSelectBlock2 (16 x 16-bits)
:	:
0x781F0	ZoneSelectBlockn (16 x 16-bits)

Three link pointers need to be programmed with the same value (not ECC protected)

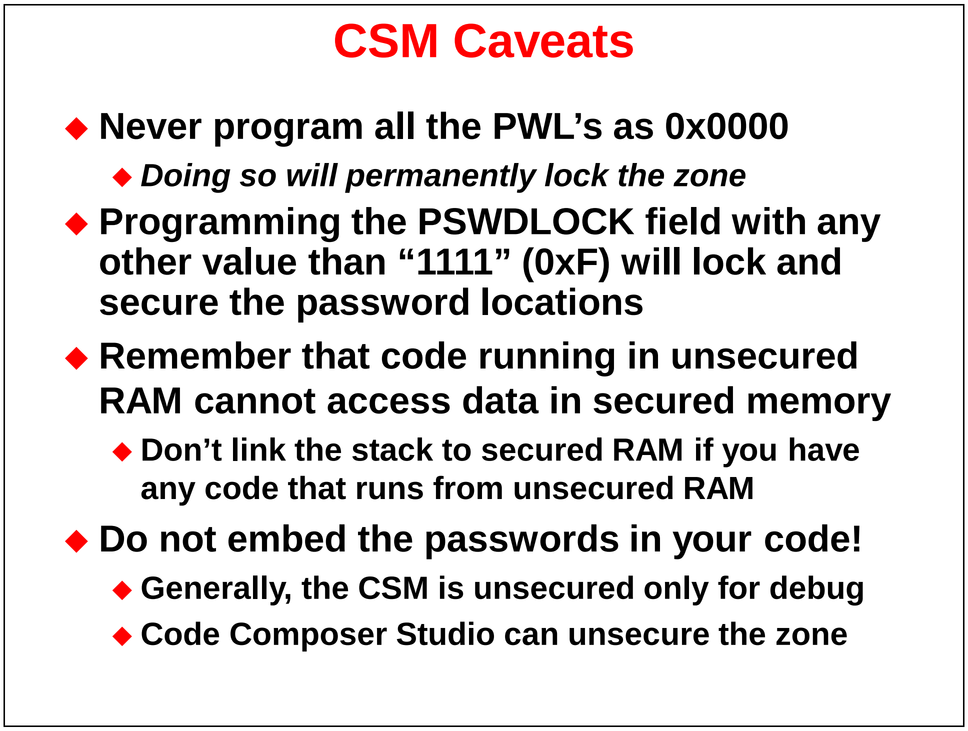
Zone Select Block	
Addr. Offset	32-bit Content
0x0	Zx-EXEONLYRAM
0x2	Zx-EXEONLYSECT
0x4	Zx-GRABRAM
0x6	Zx-GRABSECT
0x8	Zx-CSMPSWD0
0xA	Zx-CSMPSWD1
0xC	Zx-CSMPSWD2
0xE	Zx-CSMPSWD3

Zone 2 OTP FLASH	
0x78200	Z2-LINKPOINTER1
0x78202	Reserved
0x78204	Z2-LINKPOINTER2
0x78206	Reserved
0x78208	Z2-LINKPOINTER3
0x7820A	Reserved
0x78210	Z2-PSWDLOCK
0x78212	Reserved
0x78214	Z2-CRCLOCK
0x78216	Reserved
0x78218	Reserved
0x7821A	Reserved
0x7821E	Z2-BOOTCTRL
0x78220	ZoneSelectBlock1 (16 x 16-bits)
0x78230	ZoneSelectBlock2 (16 x 16-bits)
:	:
0x783F0	ZoneSelectBlockn (16 x 16-bits)



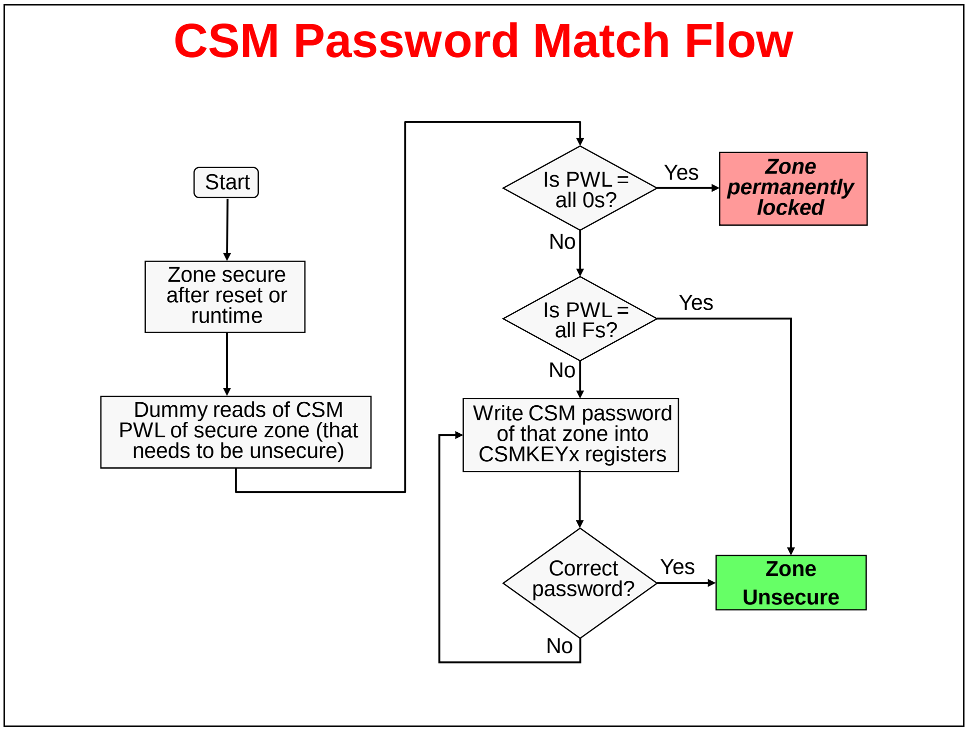
Secure and Unsecure the CSM

- **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**
- **Passwords are all 0xFFFF on new devices**
  - **When passwords are all 0xFFFF, only a read of each password location (PWL) is required to unsecure the device**
  - **The bootloader does these dummy reads and hence unsecures devices that do not have passwords programmed**



CSM Caveats

- Never program all the PWL’s as 0x0000
  - *Doing so will permanently lock the zone*
- Programming the PSWDLOCK field with any other value than “1111” (0xF) will lock and secure the password locations
- Remember that code running in unsecured RAM cannot access data in secured memory
  - Don’t link the stack to secured RAM if you have any code that runs from unsecured RAM
- Do not embed the passwords in your code!
  - Generally, the CSM is unsecured only for debug
  - Code Composer Studio can unsecure the zone



CSM Password Match Flow

Start

Zone secure after reset or runtime

Dummy reads of CSM PWL of secure zone (that needs to be unsecure)

Is PWL = all 0s?

Yes

Zone permanently locked

No

Is PWL = all Fs?

Yes

No

Write CSM password of that zone into CSMKEYx registers

Correct password?

Yes

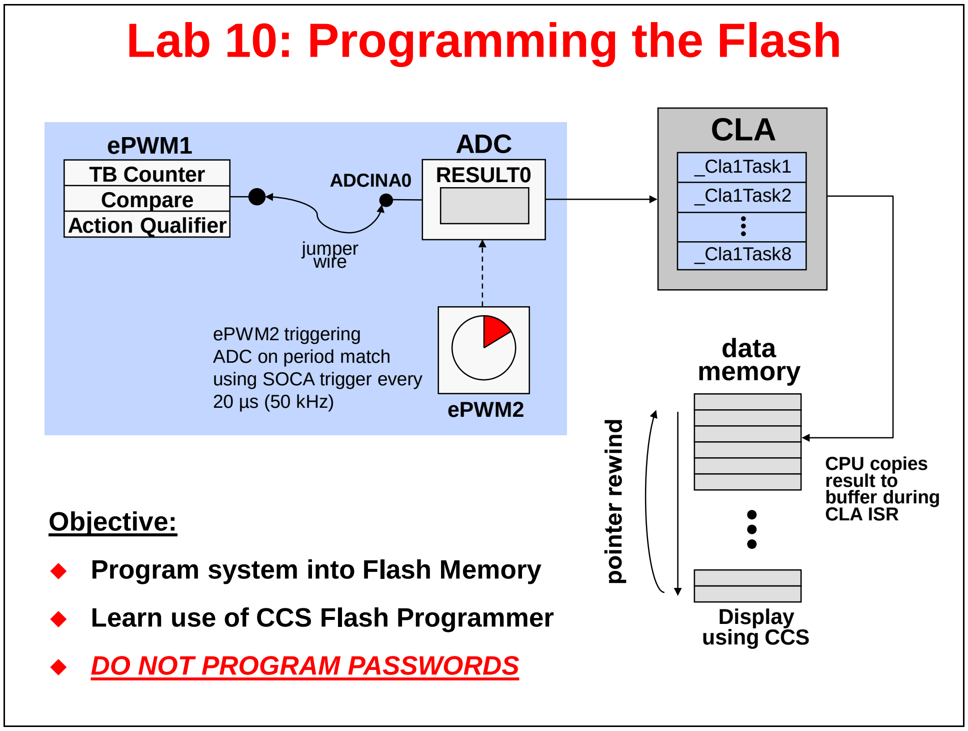
Zone Unsecure

No

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 TMS320F28379D 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.



Lab 10: Programming the Flash

Objective:

- Program system into Flash Memory
- Learn use of CCS Flash Programmer
- *DO NOT PROGRAM PASSWORDS*

➢ 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 then click `Browse…` next to the "Select search-directory" box. Navigate to: `C:\C28x\Labs\Lab10\cpu01` and click `OK`. 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
Cla_10.c
ClaTasks.asm
ClaTasks_C.cla
CodeStartBranch.asm
Dac.c
DefaultIsr_9_10.c
DelayUs.asm
Dma.c
ECap.c
EPwm.c
F2837xD_Adc.c
F2837xD_GlobalVariableDefs.c
F2837xD_Headers_nonBIOS_cpu1.cmd
Flash.c
Gpio.c
Lab_10.cmd
Main_10.c
PieCtrl.c
PieVect.c
SineTable.c
SysCtrl.c
Watchdog.c
Xbar.c
```

Note: The `Flash.c` file will be added during the lab exercise.

Link Initialized Sections to Flash

Initialized sections, such as code and constants, must contain valid values at device power-up. Stand-alone operation of an 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.

2. 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_A. The FLASH_A 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_BCDEFGHIJKLMN. 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.

3. Edit `Lab_10.cmd` to link the following compiler sections to on-chip flash memory block FLASH_BCDEFGHIJKLMN:

   Compiler Sections:

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

Copying Interrupt Vectors from Flash to RAM

The interrupt vectors must be located in on-chip flash memory and at power-up needs to be copied to the PIE RAM as part of the device initialization procedure. The code that performs this copy is located in InitPieCtrl(). The C-compiler runtime support library contains a memory copy function called *memcpy()* which will be used to perform the copy.

4. Open and inspect InitPieCtrl() in `PieCtrl.c`. Notice the memcpy() function used to initialize (copy) the PIE vectors. At the end of the file a structure is used to enable the PIE.

Initializing the Flash Control Registers

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

5. Add (copy) `Flash.c` to the project from `C:\C28x\Labs\Lab10\source`.

6. Open and inspect `Flash.c`. The C compiler CODE_SECTION pragma is used to place the InitFlash() function into a linkable section named "secureRamFuncs".

7. The "secureRamFuncs" section will be linked using the user linker command file `Lab_10.cmd`. In `Lab_10.cmd` the "secureRamFuncs" 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 "secureRamFuncs" to, we are specifiying "PAGE = 0" (which is program memory).

8. Open and inspect `Main_10.c`. Notice that the memory copy function memcpy() is being used to copy the section "secureRamFuncs", which contains the initialization function for the flash control registers.

9. Add a line of code in main() to call the InitFlash() function. There are no passed parameters or return values. You just type

   ```
   InitFlash();
   ```

   at the desired spot in main().

Dual Code Security Module and Passwords

The DCSM module provides protection against unwanted copying (i.e. pirating!) of your code from flash, OTP, LS0-5 RAM blocks, D0-1 RAM blocks, and CLA memory 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, dummy passwords of 0xFFFFFFFF 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 VALUES FROM 0xFFFFFFFF***.

Executing from Flash after Reset

The F28379D 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.

10. 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.

11. 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 emulator is connected, the device will be in emulation boot mode and will use the EMU_KEY and EMU_BMODE values in the PIE RAM to determine the boot mode. This mode was utilized in the previous lab exercises. In this lab exercise, we will be disconnecting the emulator and running in stand-alone boot mode (but do not disconnect the emulator yet!). The bootloader will read the OTP_KEY and OTP_BMODE values from their locations in the OTP. The behavior when these values have not been programmed (i.e., both 0xFF) 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 memory copy function *memcpy()* will once again 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.

12. 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.

13. Open `Cla_10.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 MemCfgRegs structure is used to configure the RAMLS4 block as CLA program memory space. Close the opened files.

Build – Lab.out

14. 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.

15. Program the flash memory by clicking the “Debug” button (green bug). A Launching Debug Session window will open. Select only CPU1 to load the program on (i.e. *uncheck* CPU2), and then click `OK`. 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

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

    `Run → Reset → CPU Reset`

    The program counter should now be at address 0x3FF16A 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`.

17. Under Scripts on the menu bar click:

    `EMU Boot Mode Select → EMU_BOOT_FLASH`

    This has the debugger load values into EMU_KEY and EMU_BMODE so that the bootloader will jump to "Flash" at address 0x080000.

18. Next click:

    `Run → Go Main`

    The code should stop at the beginning of your main()routine. If you got to that point succesfully, it confirms that the flash has been programmed properly, that the bootloader is properly configured for jump to flash mode, and that the codestart section has been linked to the proper address.

19. You can now run the CPU, and you should observe the LED D9 on the LaunchPad blinking. 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.

20. Halt the CPU.

Terminate Debug Session and Close Project

21. 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.

22. 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 GPIO72 and GPIO84 pins are used to determine the boot mode. On the LaunchPad switch SW1 controls the boot options for the F28379D device. Check that switch SW1 positions 1 and 2 are set to the default “1 – on” position (both switches up). This will configure the device (in stand-alone boot mode) to GetMode. Since the OTP_KEY has not been programmed, the default GetMode will be boot from flash. Details of the switch positions can be found in the LaunchPad User’s Guide.

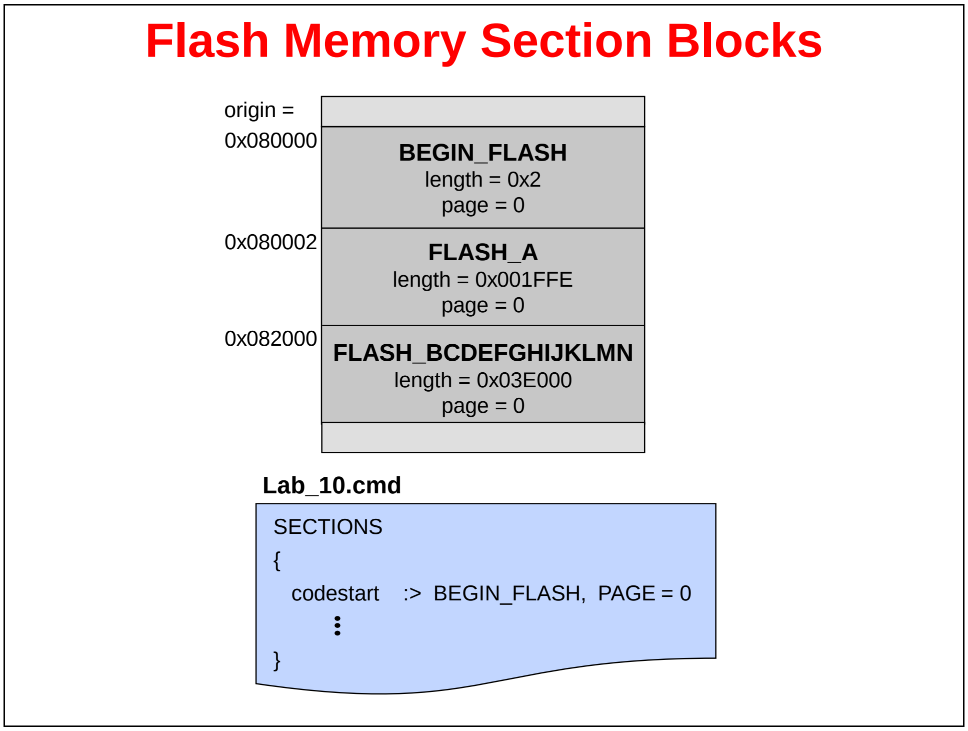
23. Close Code Composer Studio.

24. Disconnect the USB cable from the LaunchPad (i.e. remove power from the LaunchPad).

25. 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 12 Reference: Programming the Flash



Flash Memory Section Blocks

origin = 0x080000

BEGIN_FLASH
length = 0x2
page = 0

0x080002

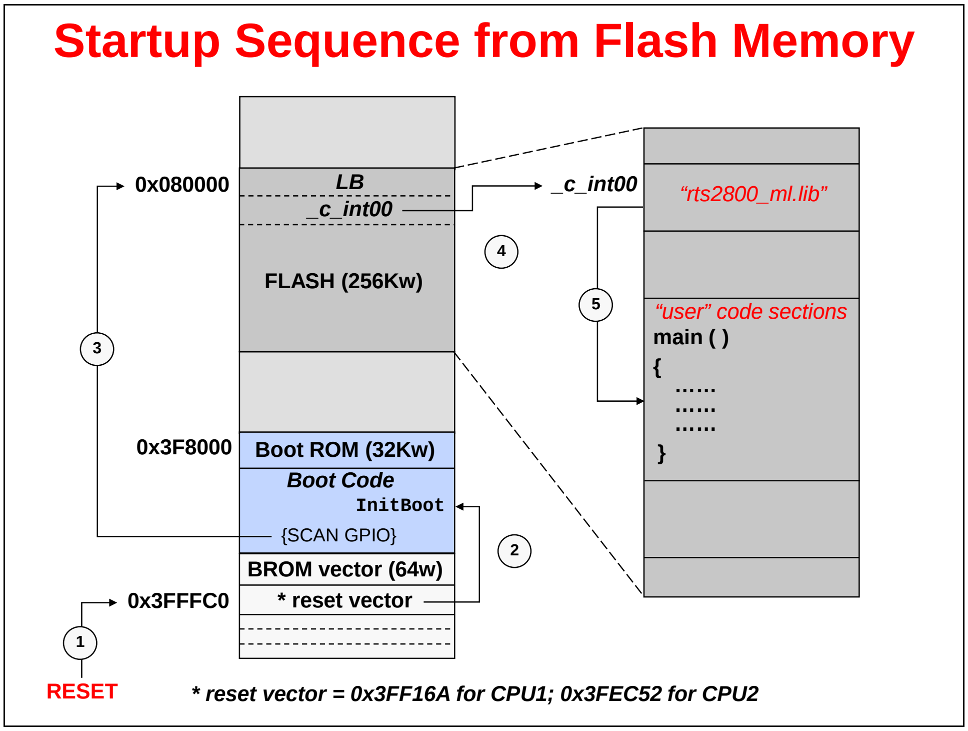
FLASH_A
length = 0x001FFE
page = 0

0x082000

FLASH_BCDEFGHIJKLMN
length = 0x03E000
page = 0

Lab_10.cmd

```
SECTIONS
{
   codestart    :>  BEGIN_FLASH,  PAGE = 0
        ⋮
}
```



Startup Sequence from Flash Memory

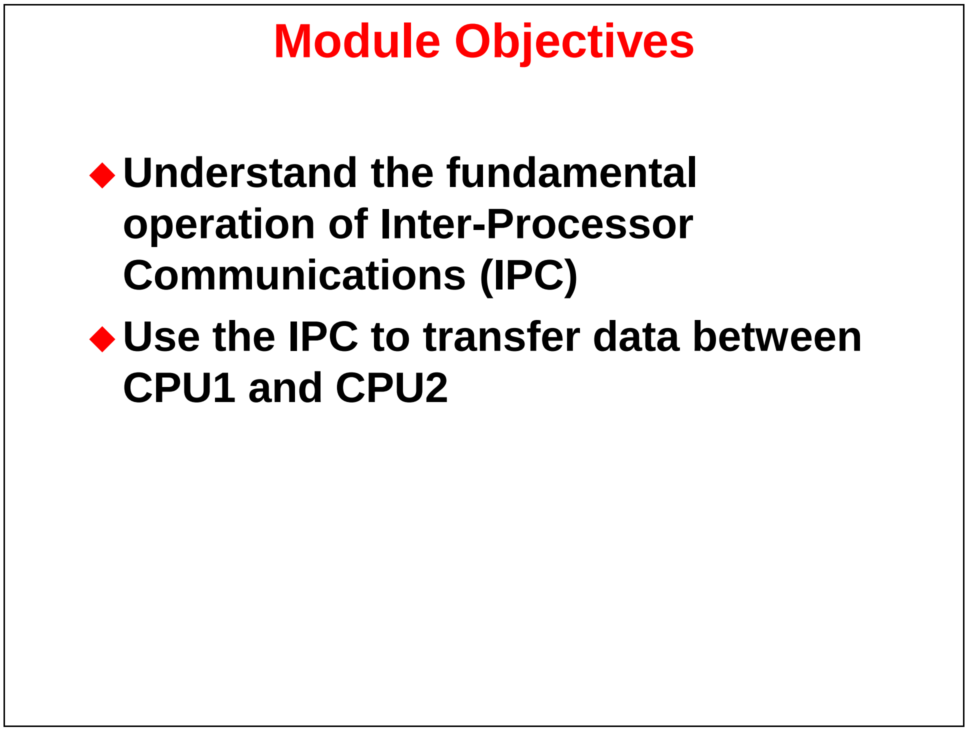
*** reset vector = 0x3FF16A for CPU1; 0x3FEC52 for CPU2**

Dual-Core Inter-Processor Communications

Introduction

This module explains the use and operation of the Inter-Processor Communications (IPC). The IPC allows communication between the two CPU subsystems (i.e. CPU1 and CPU2).

Module Objectives

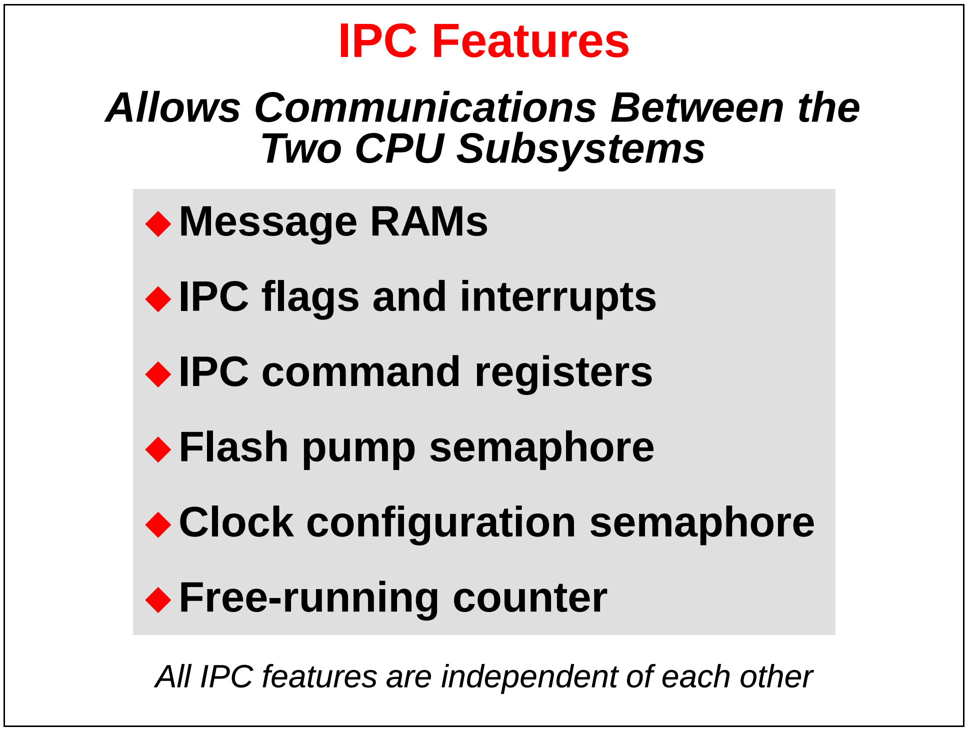


Module Objectives

- ◆ Understand the fundamental operation of Inter-Processor Communications (IPC)
- ◆ Use the IPC to transfer data between CPU1 and CPU2

Chapter Topics

Inter-Processor Communications



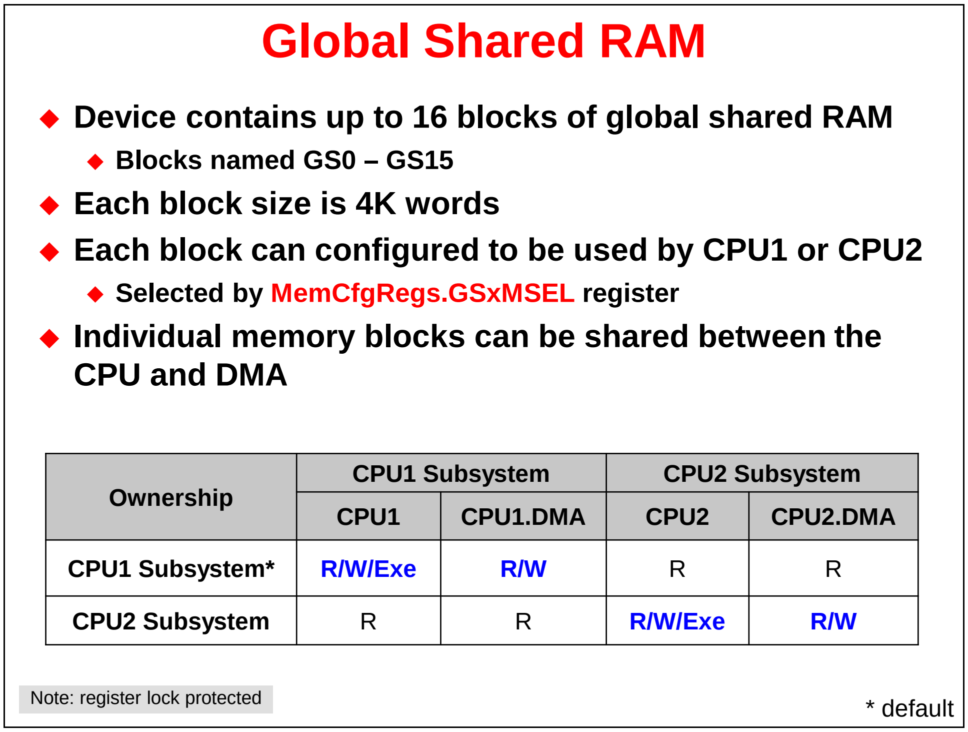
IPC Features

Allows Communications Between the Two CPU Subsystems

- **Message RAMs**
- **IPC flags and interrupts**
- **IPC command registers**
- **Flash pump semaphore**
- **Clock configuration semaphore**
- **Free-running counter**

All IPC features are independent of each other

IPC Global Shared RAM and Message RAM



Global Shared RAM

- **Device contains up to 16 blocks of global shared RAM**
  - **Blocks named GS0 – GS15**
- **Each block size is 4K words**
- **Each block can configured to be used by CPU1 or CPU2**
  - **Selected by MemCfgRegs.GSxMSEL register**
- **Individual memory blocks can be shared between the CPU and DMA**

Ownership	**CPU1 Subsystem**		**CPU2 Subsystem**	
	CPU1	**CPU1.DMA**	**CPU2**	**CPU2.DMA**
CPU1 Subsystem*	**R/W/Exe**	**R/W**	R	R
CPU2 Subsystem	R	R	**R/W/Exe**	**R/W**

Note: register lock protected

* default

There are up to 16 blocks of shared RAM on F2837xD devices. These shared RAM blocks are typically used by the application, but can also be used for transferring messages and data.

Each block can individually be owned by either CPU1 or CPU2.

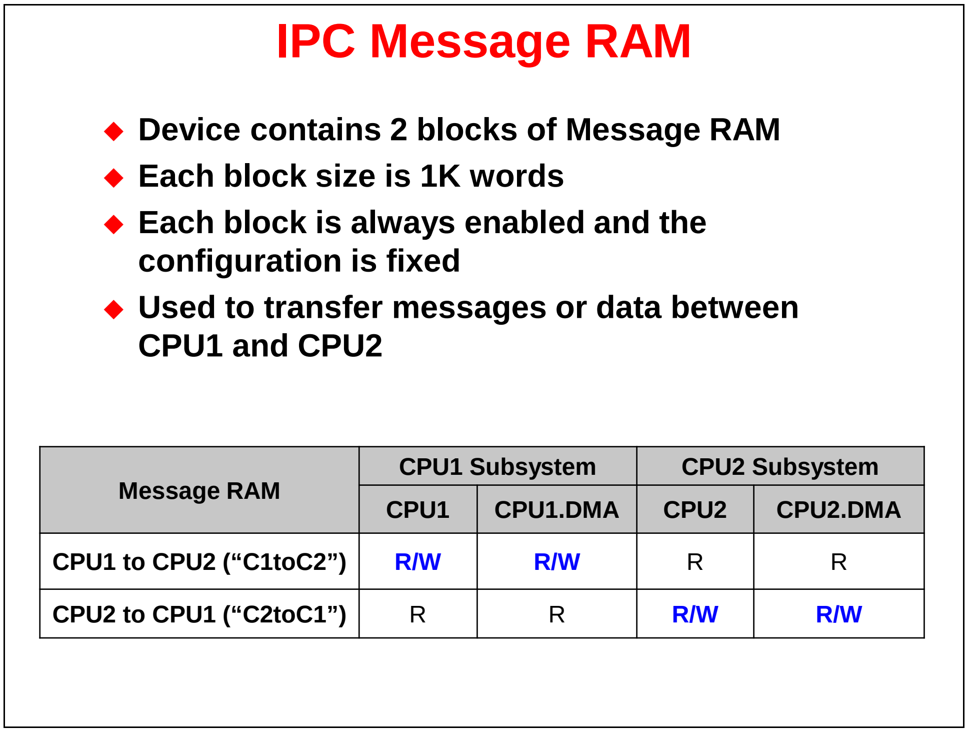
CPU1 core ownership:

At reset, CPU1 owns all of the shared RAM blocks. In this configuration CPU1 core can freely use the memory blocks. CPU1 can read, write or execute from the block and CPU1.DMA can read or write.

On the CPU2 core, CPU2 and CPU2.DMA can only read from these blocks. Blocks owned by the CPU1 core can be used by the CPU1 to send CPU2 messages. This is referred to as “C1toC2”.

CPU2 core ownership:

After reset, the CPU1 application can assign ownership of blocks to the CPU2 subsystem. In this configuration, CPU2 core can freely use the blocks. CPU2 can read, write or execute from the block and the CPU2.DMA can read or write. CPU1 core, however can only read from the block. Blocks owned by CPU2 core can be used can be used to send messages from the CPU2 to CPU1. This is referred to as “C2toC1”.



IPC Message RAM

- **Device contains 2 blocks of Message RAM**
- **Each block size is 1K words**
- **Each block is always enabled and the configuration is fixed**
- **Used to transfer messages or data between CPU1 and CPU2**

Message RAM	**CPU1 Subsystem**		**CPU2 Subsystem**	
	CPU1	**CPU1.DMA**	**CPU2**	**CPU2.DMA**
CPU1 to CPU2 (“C1toC2”)	**R/W**	**R/W**	R	R
CPU2 to CPU1 (“C2toC1”)	R	R	**R/W**	**R/W**

The F2837xD has two dedicated message RAM blocks. Each block is 1K words in length. Unlike the shared RAM blocks, these blocks provide communication in one direction only and cannot be reconfigured.

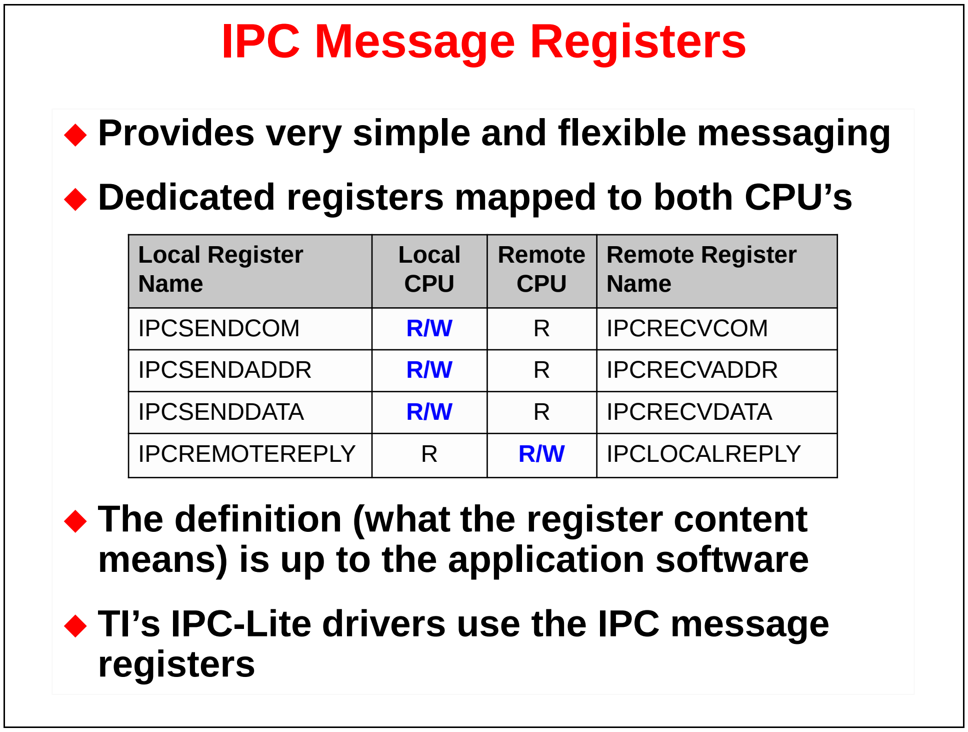
CPU1 to CPU2 “C1toC2” message RAM:

The first message RAM is the CPU1 to CPU2 or C1toC2. This block can be read or written to by the CPU1 and read by the CPU2. CPU1 can write a message to this block and then the CPU2 can read it.

CPU2 to CPU1 “C2toC1” message RAM:

The second message RAM is the CPU2 to CPU1 or C2toC1. This block can be read or written to by CPU2 and read by CPU1. This means CPU2 can write a message to this block and then

CPU1 can read it. After the sending CPU writes a message it can inform the receiver CPU that it is available through an interrupt or flag.



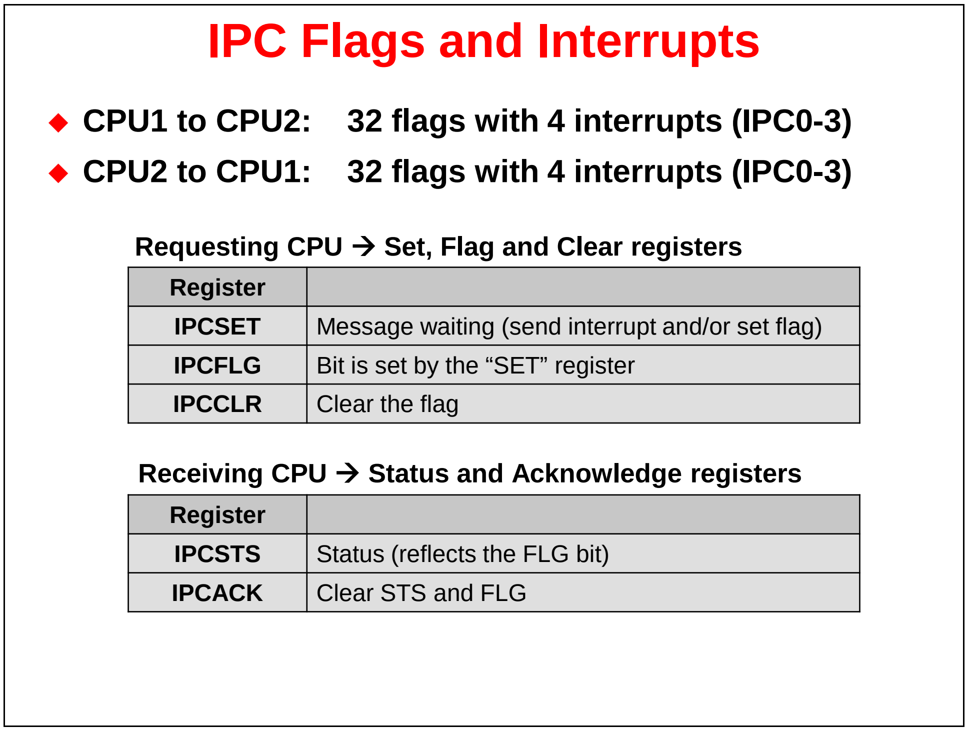
IPC Message Registers

- Provides very simple and flexible messaging
- Dedicated registers mapped to both CPU's

Local Register Name	Local CPU	Remote CPU	Remote Register Name
IPCSENDCOM	R/W	R	IPCRECVCOM
IPCSENDADDR	R/W	R	IPCRECVADDR
IPCSENDDATA	R/W	R	IPCRECVDATA
IPCREMOTEREPLY	R	R/W	IPCLOCALREPLY

- The definition (what the register content means) is up to the application software
- TI's IPC-Lite drivers use the IPC message registers

Interrupts and Flags



IPC Flags and Interrupts

- CPU1 to CPU2: 32 flags with 4 interrupts (IPC0-3)
- CPU2 to CPU1: 32 flags with 4 interrupts (IPC0-3)

Requesting CPU → Set, Flag and Clear registers

Register	
IPCSET	Message waiting (send interrupt and/or set flag)
IPCFLG	Bit is set by the "SET" register
IPCCLR	Clear the flag

Receiving CPU → Status and Acknowledge registers

Register	
IPCSTS	Status (reflects the FLG bit)
IPCACK	Clear STS and FLG

When the sending CPU wants to inform the receiver that a message is ready, it can make use of an interrupt or flag. There are identical IPC interrupt and flag resources on both CPU1 core and CPU2 core.

4 Interrupts:

There are 4 interrupts that CPU1 can send to CPU2 (and vice-versa) through the Peripheral Interrupt Expansion (PIE) module. Each of the interrupts has a dedicated vector within the PIE, IPC0 – IPC3.

28 Flags:

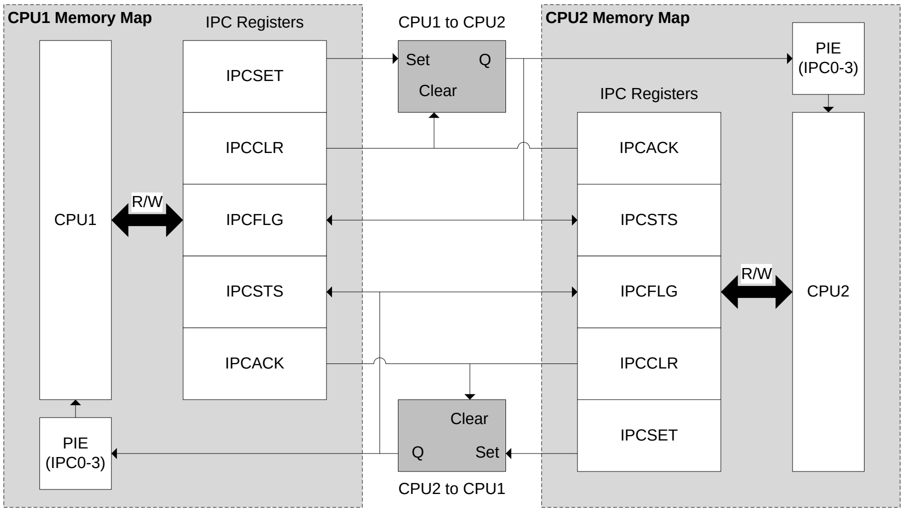
In addition, there are 28 flags available to each of the CPU cores. These flags can be used for messages that are not time critical or they can be used to send status back to originating processor. The flags and interrupts can be used however the application sees fit and are not tied to particular operation in hardware.

Registers: Set, Flag, Clear, Status and Acknowledge

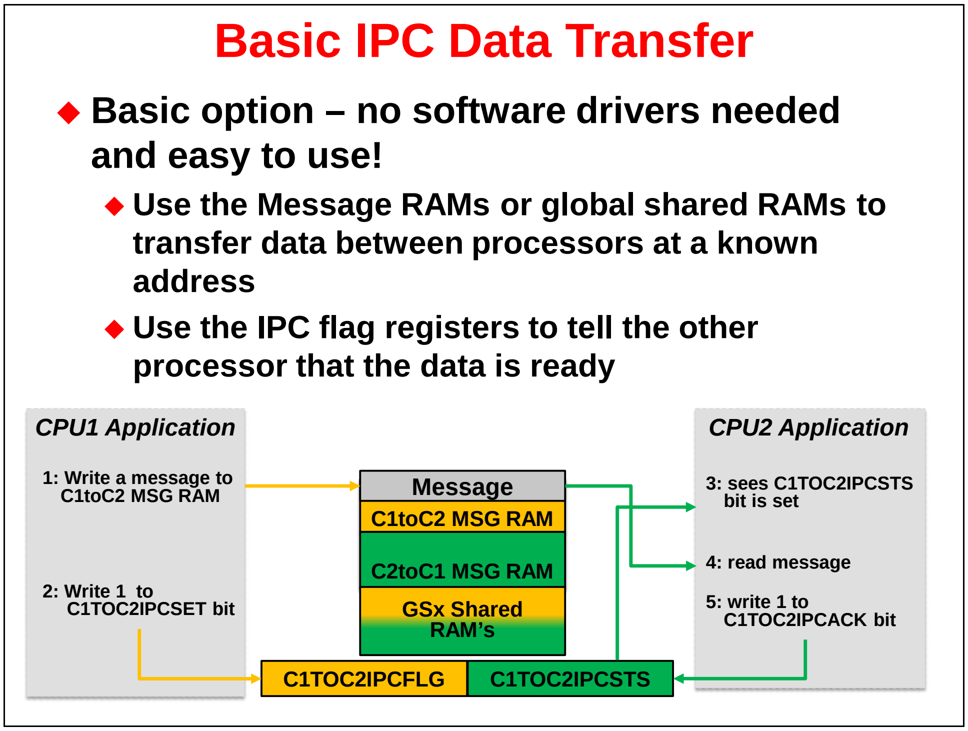
The registers to control the IPC interrupts and flags are 32-bits:

Bits [3:0] = interrupt & flag
Bits [31:4] = flag only

Messaging with IPC Flags and Interrupts



IPC Data Transfer



Basic IPC Data Transfer

- Basic option – no software drivers needed and easy to use!
  - Use the Message RAMs or global shared RAMs to transfer data between processors at a known address
  - Use the IPC flag registers to tell the other processor that the data is ready

CPU1 Application

1: Write a message to C1toC2 MSG RAM

2: Write 1 to C1TOC2IPCSET bit

Message

C1toC2 MSG RAM

C2toC1 MSG RAM

GSx Shared RAM's

C1TOC2IPCFLG

C1TOC2IPCSTS

CPU2 Application

3: sees C1TOC2IPCSTS bit is set

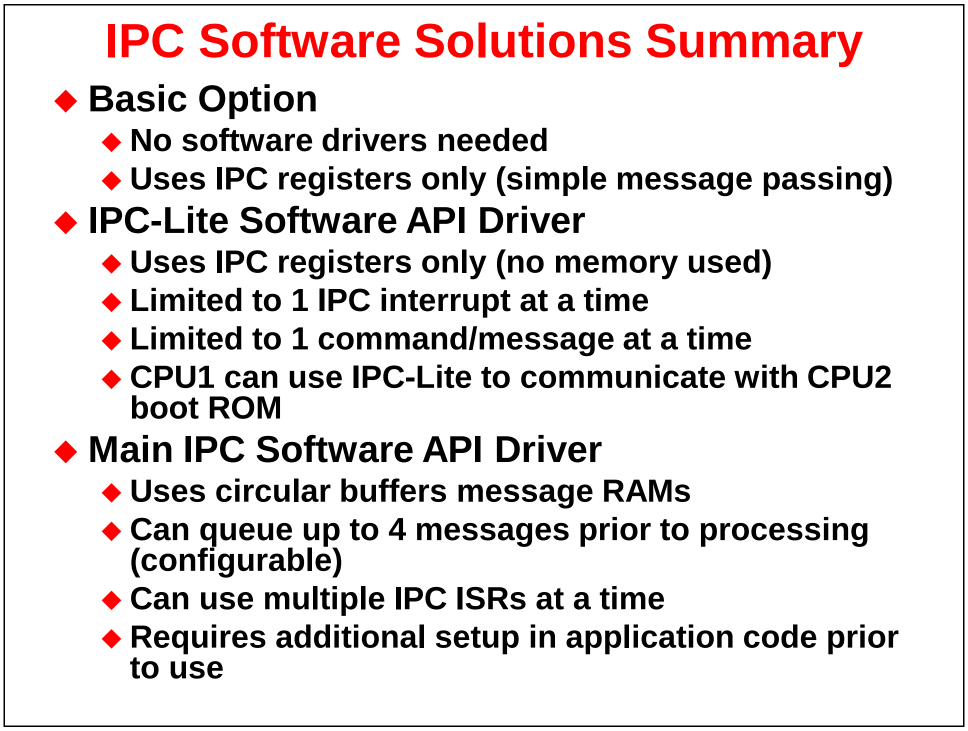
4: read message

5: write 1 to C1TOC2IPCACK bit

The F2837xD IPC is very easy to use. At the most basic level, the application does not need any separate software drivers to communicate between processors. It can utilize the message RAM's and shared RAM blocks to pass data between processors at a fixed address known to both processors. Then the sending processor can use the IPC flag registers merely to flag to the receiving processor that the data is ready. Once the receiving processor has grabbed the data, it will then acknowledge the corresponding IPC flag to indicate that it is ready for more messages.

As an example:

1. First, CPU1 would write a message to the CPU2 in C1toC2 MSG RAM.
2. Then the CPU1 would write a 1 to the appropriate flag bit in the C1TOC2IPCSET register. This sets the C1TOC2IPCFLG, which also sets the C1TOC2IPCSTS register on CPU2, letting CPU2 know that a message is available.
3. Then CPU2 sees that a bit in the C1TOC2IPCSTS register is set.
4. Next CPU2 reads the message from the C1toC2 MSG RAM and then
5. It writes a 1 to the same bit in the C1TOC2IPCACK register to acknowledge that it has received the message. This subsequently clears the flag bit in C1TOC2IPCFLG and C1TOC2IPCSTS.
6. CPU1 can then send more messages using that particular flag bit.



IPC Software Solutions Summary

- **Basic Option**
  - **No software drivers needed**
  - **Uses IPC registers only (simple message passing)**
- **IPC-Lite Software API Driver**
  - **Uses IPC registers only (no memory used)**
  - **Limited to 1 IPC interrupt at a time**
  - **Limited to 1 command/message at a time**
  - **CPU1 can use IPC-Lite to communicate with CPU2 boot ROM**
- **Main IPC Software API Driver**
  - **Uses circular buffers message RAMs**
  - **Can queue up to 4 messages prior to processing (configurable)**
  - **Can use multiple IPC ISRs at a time**
  - **Requires additional setup in application code prior to use**

There are three options to use the IPC on the device.

Basic option: A very simple option that does not require any drivers. This option only requires IPC registers to implement very simple flagging of messages passed between processors.

Driver options: If the application code needs a set of basic IPC driver functions for reading or writing data, setting/clearing bits, and function calls, then there are 2 IPC software driver solutions provided by TI.

IPC-Lite:

- Only uses the IPC registers. No additional memory such as message RAM or shared RAM is needed.
- Only one IPC ISR can be used at a time.
- Can only process one message at a time.
- CPU1 can use IPC lite to communicate with the CPU2 boot ROM. The CPU2 boot ROM processes basic IPC read, write, bit manipulation, function call, and branch commands.

Main IPC Software API Driver: (This is a more feature filled IPC solution)

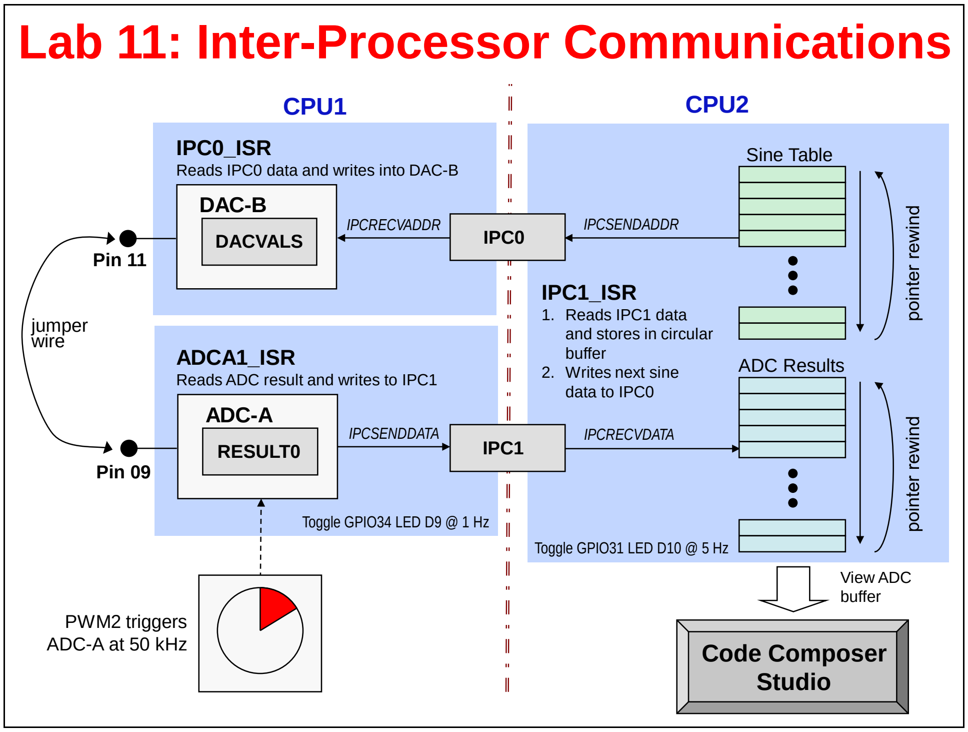
- Utilizes circular buffers in C2toC1 and C1toC2 message RAM’s.
- Allows application to queue up to 4 messages prior to processing (configurable).
- Allows application to use multiple IPC ISR’s at a time.
- Requires additional setup in application code prior to use.

In addition to the above, SYS/BIOS 6 will provide a new transport module to work with the shared memory and IPC resources on the F2837x.

Lab 11: Inter-Processor Communications

➢ Objective

The objective of this lab exercise is to demonstrate and become familiar with the operation of the IPC module. We will be using the basic IPC features to send data in both directions between CPU1 and CPU2. A typical dual-core F2837xD application consists of two separate and completely independent CCS projects. One project is for CPU1, and the other project is for CPU2. As in the previous lab exercises, PWM2 will be configured to provide a 50 kHz SOC signal to ADC-A. An End-of-Conversion ISR on CPU1 will read each result and write it into a data register in the IPC. An IPC interrupt will then be triggered on CPU2 which fetches this data and stores it in a circular buffer. The same ISR grabs a data point from a sine table and loads it into a different IPC register for transmission to CPU1. This triggers an interrupt on CPU1 to fetch the sine data and write it into DAC-B. The DAC-B output is connected by a jumper wire to the ADCINA0 pin. If the program runs as expected, the sine table and ADC results buffer on CPU2 should contain very similar data.


Lab 11: Inter-Processor Communications
CPU1
CPU2
IPC0_ISR
Reads IPC0 data and writes into DAC-B
DAC-B
DACVALS
IPCRECVADDR
IPC0
IPCSENDADDR
Sine Table
pointer rewind
Pin 11
jumper wire
IPC1_ISR
1. Reads IPC1 data and stores in circular buffer
2. Writes next sine data to IPC0
ADCA1_ISR
Reads ADC result and writes to IPC1
ADC-A
RESULT0
IPCSENDDATA
IPC1
IPCRECVDATA
ADC Results
pointer rewind
Pin 09
Toggle GPIO34 LED D9 @ 1 Hz
Toggle GPIO31 LED D10 @ 5 Hz
View ADC buffer
PWM2 triggers ADC-A at 50 kHz
Code Composer Studio

➢ Procedure

Open the Projects – CPU1 & CPU2

1. Two projects named `Lab11_cpu01` and `Lab11_cpu02` have been created for this lab exercise. Open *both* projects by clicking on `Project` → `Import CCS Projects`. The “Import CCS Eclipse Projects” window will open then click `Browse…` next to the “Select search-directory” box. Navigate to: `C:\C28x\Labs\Lab11` and click `OK`.

   Both projects will appear in the “Discovered projects” window. Click `Select All` and click `Finish` to import the project. All build options for each project have been configured the same as the previous lab exercise.

   The files used in the CPU1 project are:

```
Adc.c
CodeStartBranch.asm
Dac.c
DefaultIsr_11_cpu1.c
DelayUs.asm
EPwm_11.c
F2837xD_Adc.c
F2837xD_GlobalVariableDefs.c
F2837xD_Headers_nonBIOS_cpu1.cmd
Gpio.c
Lab_11_cpu1.cmd
Main_11_cpu1.c
PieCtrl.c
PieVect.c
SysCtrl.c
Watchdog.c
```

The files used in the CPU2 project are:

```
CodeStartBranch.asm
DefaultIsr_11_cpu2.c
F2837xD_GlobalVariableDefs.c
F2837xD_Headers_nonBIOS_cpu1.cmd
Lab_11_cpu2.cmd
Main_11_cpu2.c
PieCtrl.c
PieVect.c
SineTable.c
Watchdog.c
```

Inspect the Project – CPU1

2. Click on the project name `Lab11_cpu01` in the Project Explorer window to set the project active. Then click on the plus sign (+) to the left of Lab11_cpu01 to expand the file list.

3. Open and inspect `Main_11_cpu1.c`. Notice the synchronization handshake code using IPC17 during initialization:

```
//--- Wait here until CPU02 is ready
    while (IpcRegs.IPCSTS.bit.IPC17 == 0) ;  // Wait for CPU02 to set IPC17
    IpcRegs.IPCACK.bit.IPC17 = 1;            // Acknowledge and clear IPC17
```

CPU1 will start first and then wait until CPU2 releases it from the while() loop. This only needs to be done once. In effect, CPU1 is waiting until CPU2 is ready to accept IPC interrupts, thereby making sure that the CPUs are ready for messaging through the IPC.

4. Open and inspect `DefaultIsr_11_cpu1.c`. This file contains two interrupt service routines – one (`ADCA1_ISR`) at PIE1.1 reads the ADC results which is sent over IPC1 to CPU2, and the other (`IPC0_ISR`) at PIE1.13 reads the incoming sine table point for the DAC which is sent over IPC0 from CPU2. Additionally, ADCA1_ISR toggles the LaunchPad LED D9 at 1 Hz as a visual indication that it is running.

In `ADCA1_ISR()` the ADC result value being sent to CPU2 is written via the `IPCSENDDATA` register. In `IPC0_ISR()` the incoming data from CPU2 for the DAC is read via the `IPCRECVADDR` register. These registers are part of the IPC module and provide an easy way to transmit single data words between CPUs without using memory.

Inspect the Project – CPU2

5. Click on the project name `Lab11_cpu02` in the Project Explorer window to set the project active. Then click on the plus sign (+) to the left of Lab11_cpu02 to expand the file list.

6. Open and inspect `Main_11_cpu2.c`. Notice the synchronization handshake code used to release CPU1 from its while() loop:

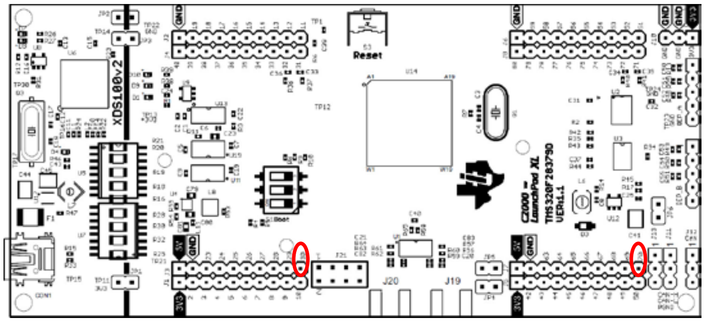
```
//--- Let CPU1 know that CPU2 is ready
    IpcRegs.IPCSET.bit.IPC17 = 1;            // Set IPC17 to release CPU1
```

7. Open and inspect `DefaultIsr_11_cpu2.c`. This file contains a single interrupt service routine – (`IPC1_ISR`) at PIE1.14 reads the incoming ADC results which is sent over IPC1 from CPU1, and writes the next sine table point for the DAC which is sent over IPC0 to CPU1. Additionally, IPC1_ISR toggles the LaunchPad LED D10 at 5 Hz as a visual indication that it is running.

In `IPC1_ISR()` the incoming ADC result value from CPU1 is read via the `IPCRECVDATA` register, and the sine data to CPU1 is written via the `IPCSENDADDR` register. The `IPCSENDDATA` and `IPCRECVDATA` registers are mapped to the same address on each CPU, as are the `IPCSENDADDR` and `IPCRECVADDR` registers.

Jumper Wire Connection

8. Using a jumper wire, connect the ADCINA0 (header J3, pin #30) to DACB (header J7, pin #70) on the LaunchPad. Refer to the following diagram for the pins that need to be connected.

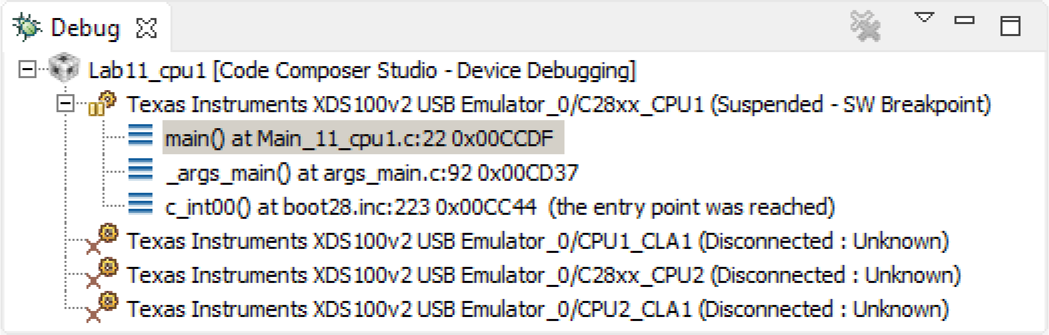


Build and Load the Project

9. In the Project Explorer window click on the `Lab11_cpu01` project to set it active. Then click the Build button and watch the tools run in the Console window. Check for any errors in the Problems window. Repeat this step for the `Lab11_cpu02` project.

10. Again, in the Project Explorer window click on the `Lab11_cpu01` project to set it active. Click on the `Debug` button (green bug). A Launching Debug Session window will open. Select only CPU1 to load the program on (i.e. *__uncheck__* CPU2), and then click `OK`. The CCS Debug perspective view should open, then CPU1 will connect to the target and the program will load automatically.

11. The Debug window reflects the current status of CPU1 and CPU2.



Debug

Lab11_cpu1 [Code Composer Studio - Device Debugging]
Texas Instruments XDS100v2 USB Emulator_0/C28xx_CPU1 (Suspended - SW Breakpoint)
main() at Main_11_cpu1.c:22 0x00CCDF
_args_main() at args_main.c:92 0x00CD37
c_int00() at boot28.inc:223 0x00CC44 (the entry point was reached)
Texas Instruments XDS100v2 USB Emulator_0/CPU1_CLA1 (Disconnected : Unknown)
Texas Instruments XDS100v2 USB Emulator_0/C28xx_CPU2 (Disconnected : Unknown)
Texas Instruments XDS100v2 USB Emulator_0/CPU2_CLA1 (Disconnected : Unknown)

Notice that CPU1 is currently connected and CPU2 is “Disconnected”. This means that CCS has no control over CPU2 thus far; it is freely running from the view of CCS. Of course CPU2 is under control of CPU1 and since we have not executed an IPC command yet, CPU2 is stopped by an “Idle” mode instruction in the Boot ROM.

12. Next, we need to connect to and load the program on CPU2. Right-click at the line “Texas Instruments XDS100v2 USB Emulator_0/C28xx_CPU2” and select `Connect Target`.

13. With the line “Texas Instruments XDS100v2 USB Emulator_0/C28xx_CPU2” still highlighted, load the program:

`Run → Load → Load Program…`

Browse to the file: `C:\C28x\Labs\Lab11\cpu02\Debug\Lab11_cpu02.out` and select `OK` to load the program.

14. Again, with the line "Texas Instruments XDS100v2 USB Emulator_0/C28xx_CPU1" highlighted, set the bootloader mode using the menu bar by clicking:

    `Scripts → EMU Boot Mode Select → EMU_BOOT_SARAM`

    Use the same procedure above to set the bootloader mode for CPU2. If the device has been power cycled between lab exercises, or within this lab exercise, be sure to configure the boot mode to `EMU_BOOT_SARAM` using the Scripts menu for both CPU1 and CPU2.

Run the Code

15. In the Debug window, click on the line "Texas Instruments XDS100v2 USB Emulator_0/C28xx_CPU1". Run the code on CPU1 by clicking the green `Resume` button. At this point CPU1 is waiting for CPU2 to be ready.
16. In the Debug window, click on the line "Texas Instruments XDS100v2 USB Emulator_0/C28xx_CPU2". As before, run the code on CPU2 by clicking the `Resume` button. Using the IPC17, CPU2 communicates to CPU1 that it is now ready. On the LaunchPad, LED D9 connected to CPU1 should be blinking at approximately 1 Hz and LED D10 connected to CPU2 should be blinking at approximately 5 Hz.
17. In the Debug window select CPU1. Halt the CPU1 code after a few seconds by clicking on the `Suspend` button.
18. Then in the Debug window select CPU2. Halt the CPU2 code by using the same procedure.

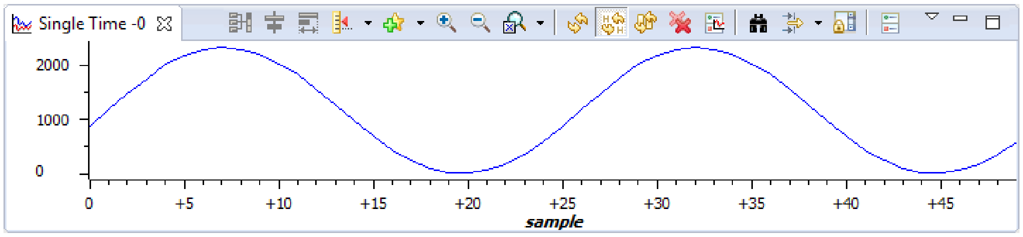
View the ADC Results

19. 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	sample

    Select `OK` to save the graph options.

20. If the IPC communications is working, the ADC results buffer on CPU2 should contain the sine data transmitted from the look-up table. The graph view should look like:



Run the Code - Real-Time Emulation Mode

21. We will now run the code in real-time emulation mode. Enable the graph window for continuous refresh. On the graph window toolbar, left-click on "`Enable Continuous Refresh`" (the yellow icon with the arrows rotating in a circle over a pause sign). This will allow the graph to continuously refresh in real-time while the program is running.

22. In the Debug window highlight the line "Texas Instruments XDS100v2 USB Emulator_0/C28xx_CPU1". Run the code on CPU1 in real-time mode by clicking:

    `Scripts → Realtime Emulation Control → Run_Realtime_with_Reset`

23. Next, in the Debug window highlight the line "Texas Instruments XDS100v2 USB Emulator_0/C28xx_CPU2". Run the code on CPU2 in real-time mode by using the same procedure above.

    The graph should now be updating in real-time.

24. *Carefully* remove and replace the jumper wire from the DACB output (header J7, pin #70) to the ADCINA0 input (header J3, pin #30). The ADC results graph should disappear and be replaced by a flat line when the jumper wire is removed. This shows that the sine data is being transmitted over IPC0 to CPU1, and (after being sent from DAC to ADC) received from CPU1 over IPC1.

25. Now we will view the IPC registers while the code is running in real-time emulation mode on CPU1 and CPU2. Open `Main_11_cpu1.c` (or `Main_11_cpu2.c`), highlight the "IpcRegs" structure and right click, then select `Add Watch Expression…` and click `OK`. Enable the Expressions window for continuous refresh.

26. In the Debug window highlight the line "Texas Instruments XDS100v2 USB Emulator_0/C28xx_CPU1". Then in the Expressions window open "IpcRegs", scroll down and notice the `IPCSENDDATA` and `IPCRECVADDR` registers is updating, as expected for CPU1. Also, notice that `IPCSENDADDR` and `IPCRECVDATA` registers, as well as the graph (ADC buffer) are not updated on CPU1.

27. In the Debug window highlight the line "Texas Instruments XDS100v2 USB Emulator_0/C28xx_CPU2". Then in the Expressions window open "IpcRegs", scroll down and notice the `IPCRECVDATA` and `IPCSENDADDR` registers, and the graph is updating, as expected for CPU2. Likewise, notice that `IPCRECVADDR` and `IPCSENDDATA` registers are not updated on CPU2.

28. Again, in the Debug window highlight the line "Texas Instruments XDS100v2 USB Emulator_0/C28xx_CPU1". Fully halt the code on CPU1 in real-time mode by clicking:

    `Scripts → Realtime Emulation Control → Full_Halt`

29. Next, fully halt the code on CPU2 in real-time mode by using the same procedure.

Terminate Debug Session and Close Project

30. 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.

31. Next, close the `Lab11_cpu01` and `Lab11_cpu02` projects by right-clicking on each project in the Project Explorer window and select `Close Project`.

End of Exercise

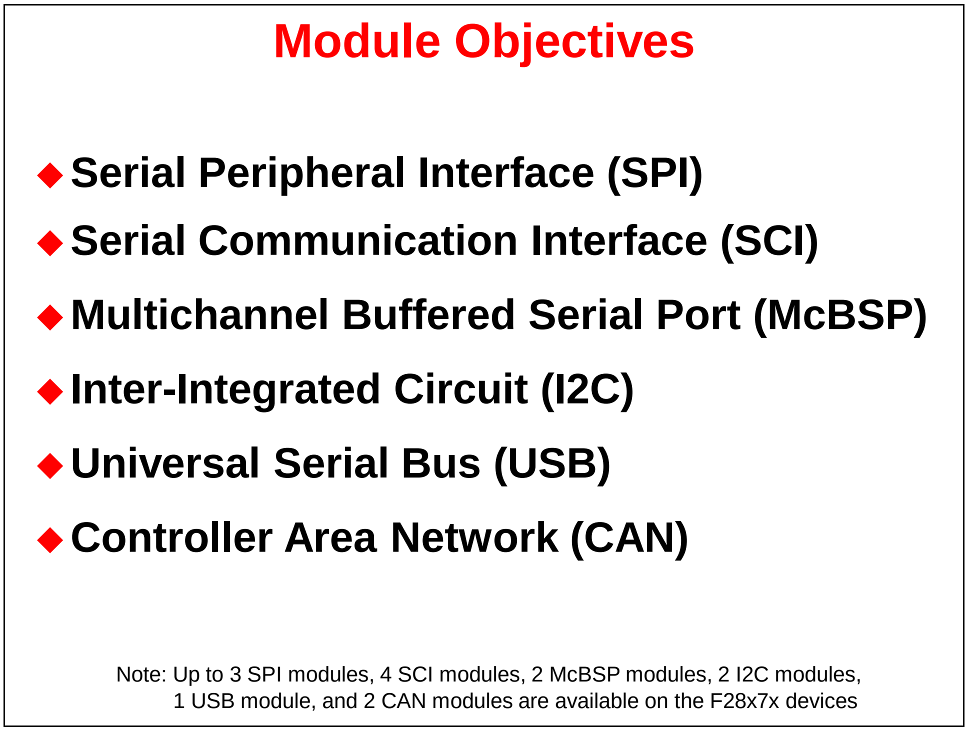
Communications

Introduction

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

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

Module Objectives



Module Objectives

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

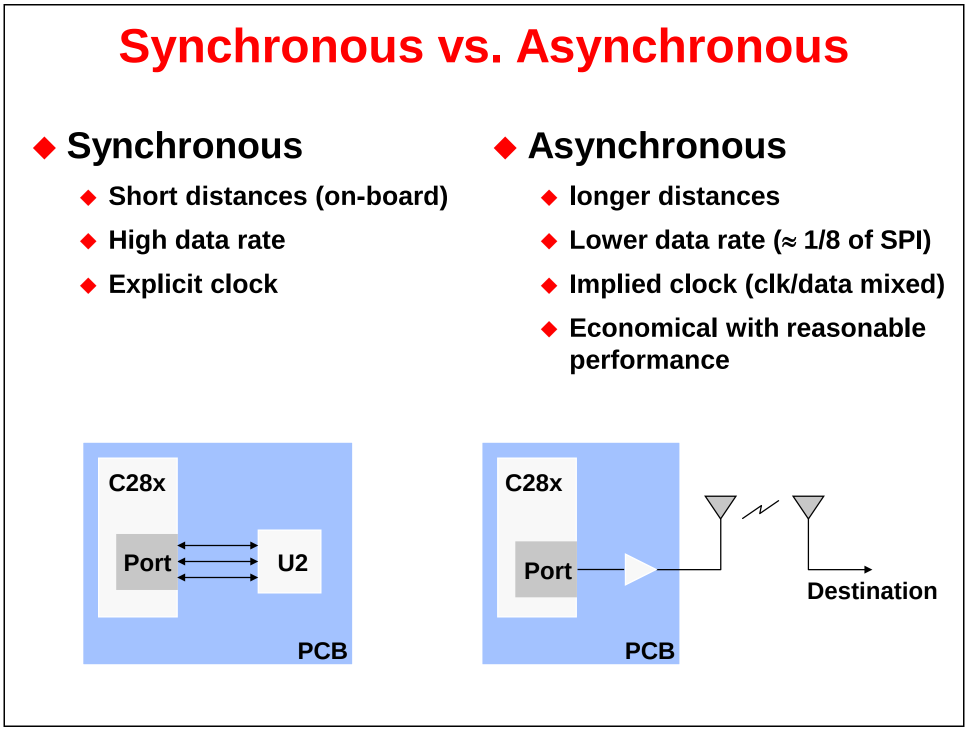
Note: Up to 3 SPI modules, 4 SCI modules, 2 McBSP modules, 2 I2C modules,
1 USB module, and 2 CAN modules are available on the F28x7x devices

The F2837xD dual-core MCU includes numerous communications peripherals that extend the connectivity of the device. There are up to three Serial Peripheral Interface (SPI) modules, four Serial Communication Interface (SCI) modules, two Multi-channel Buffered Serial Port (McBSP) modules, two Inter-Integrated Circuit (I2C) modules, two Controller Area Network (CAN) modules, one Universal Serial Bus (USB) module, and one Universal Parallel Port (uPP) module. These peripherals can be assigned to either the CPU1 subsystem or the CPU2 subsystem, except for the USB and uPP which is dedicated to only the CPU1 subsystem.

Chapter Topics

Communications Techniques

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



Synchronous vs. Asynchronous

- **Synchronous**
  - **Short distances (on-board)**
  - **High data rate**
  - **Explicit clock**
- **Asynchronous**
  - **longer distances**
  - **Lower data rate (≈ 1/8 of SPI)**
  - **Implied clock (clk/data mixed)**
  - **Economical with reasonable performance**

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

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

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

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

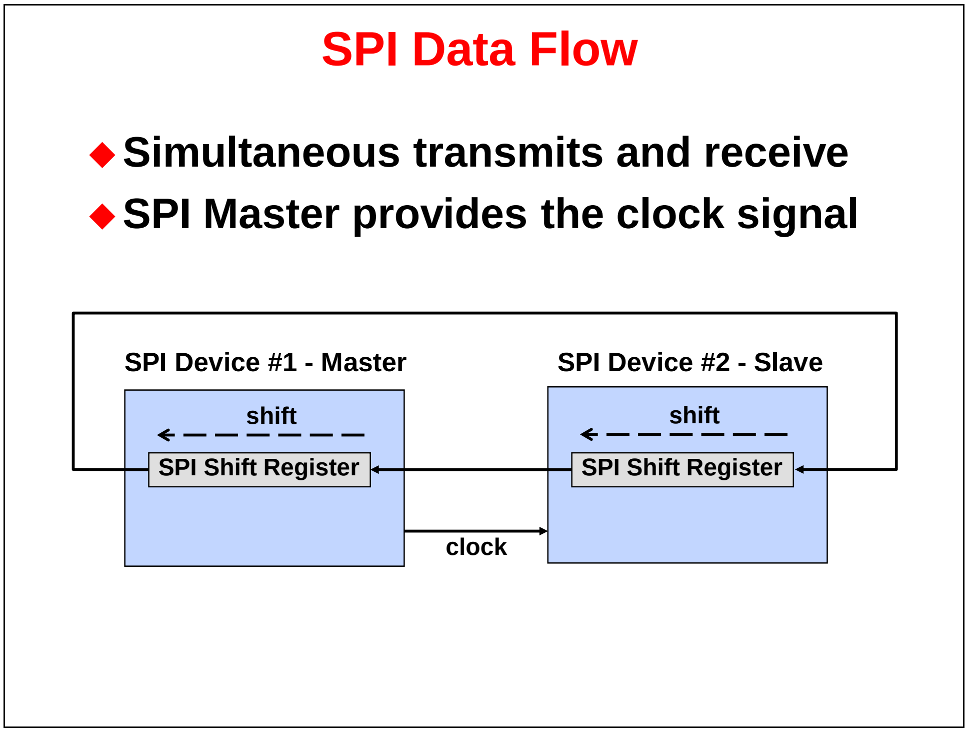
Serial Peripheral Interface (SPI)

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

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

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

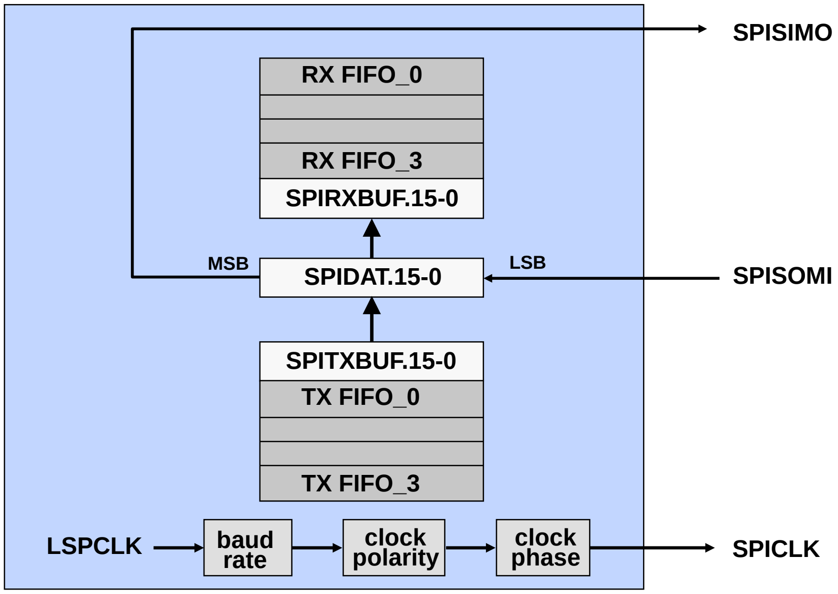


SPI Data Flow

- Simultaneous transmits and receive
- SPI Master provides the clock signal

SPI Block Diagram

C28x - SPI Master Mode Shown

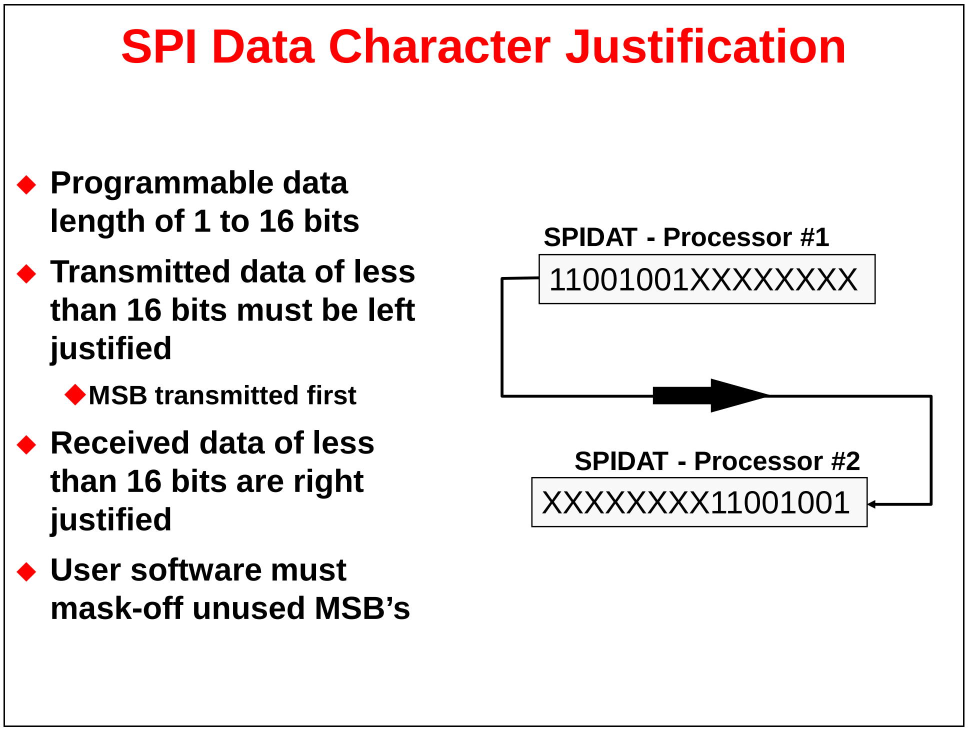


SPI Transmit / Receive Sequence

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

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

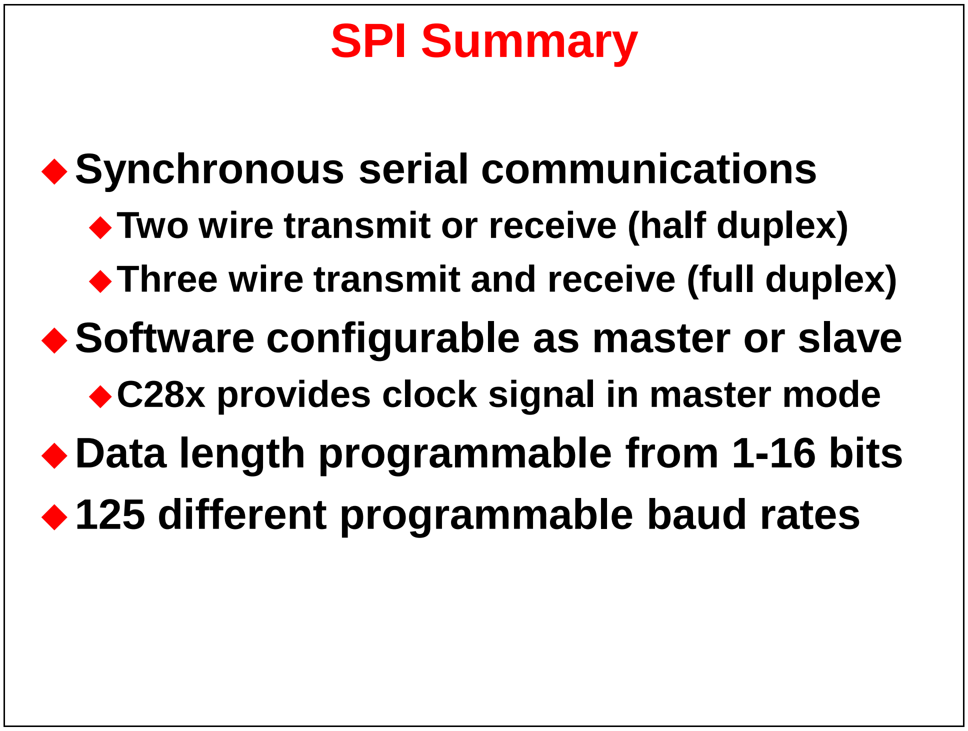
Received data is shifted into SPIDAT from the left, MSB first. However, the entire sixteen bits of SPIDAT is copied into SPIBUF after the character transmission is complete such that received characters of less than 16 bits will be right-justified in SPIBUF. The non-utilized higher significance bits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



SPI Data Character Justification

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

SPI Summary



SPI Summary

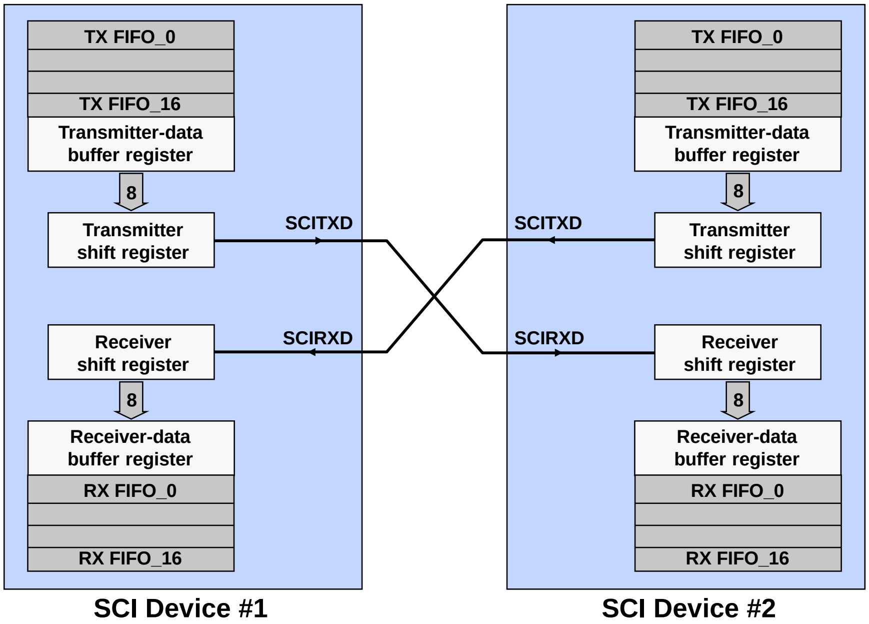
- **Synchronous serial communications**
  - **Two wire transmit or receive (half duplex)**
  - **Three wire transmit and receive (full duplex)**
- **Software configurable as master or slave**
  - **C28x provides clock signal in master mode**
- **Data length programmable from 1-16 bits**
- **125 different programmable baud rates**

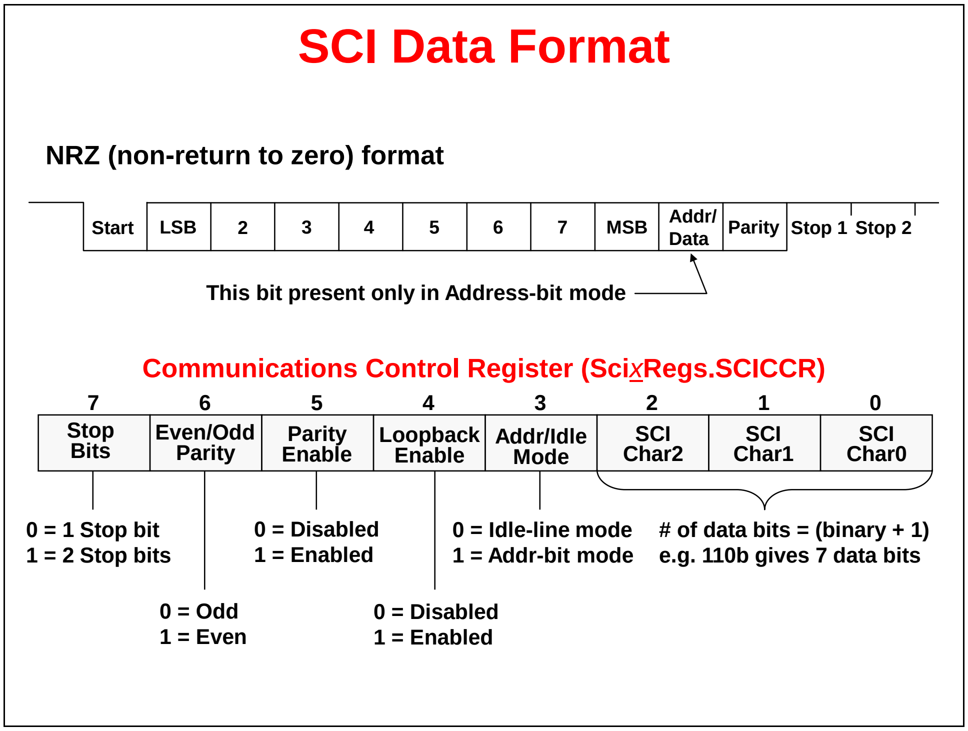
Serial Communications Interface (SCI)

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.

SCI Pin Connections

(Full Duplex Shown)





SCI Data Format

NRZ (non-return to zero) format

Start	LSB	2	3	4	5	6	7	MSB	Addr/ Data	Parity	Stop 1	Stop 2

This bit present only in Address-bit mode

Communications Control Register (SciꭓRegs.SCICCR)

7	6	5	4	3	2	1	0
Stop Bits	Even/Odd Parity	Parity Enable	Loopback Enable	Addr/Idle Mode	SCI Char2	SCI Char1	SCI Char0
0 = 1 Stop bit 1 = 2 Stop bits	0 = Odd 1 = Even	0 = Disabled 1 = Enabled	0 = Disabled 1 = Enabled	0 = Idle-line mode 1 = Addr-bit mode	# of data bits = (binary + 1) e.g. 110b gives 7 data bits		

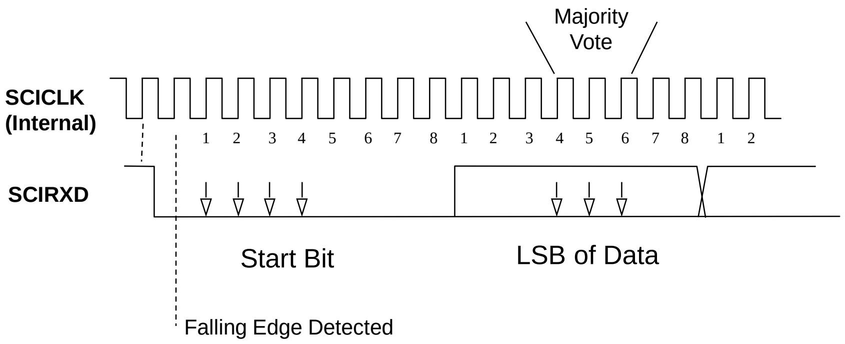
The basic unit of data is called a **character** and is 1 to 8 bits in length. Each character of data is formatted with a start bit, 1 or 2 stop bits, an optional parity bit, and an optional address/data bit. A character of data along with its formatting bits is called a **frame**. Frames are organized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

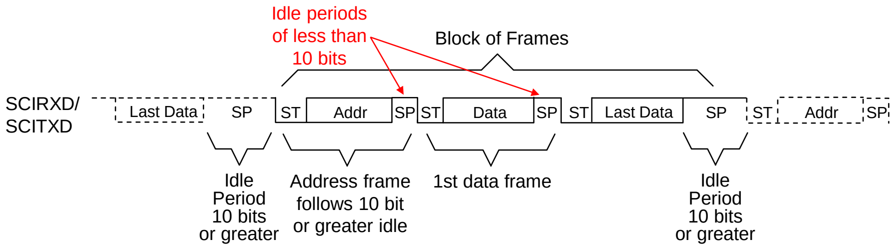
Multiprocessor Wake-Up Modes

- 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

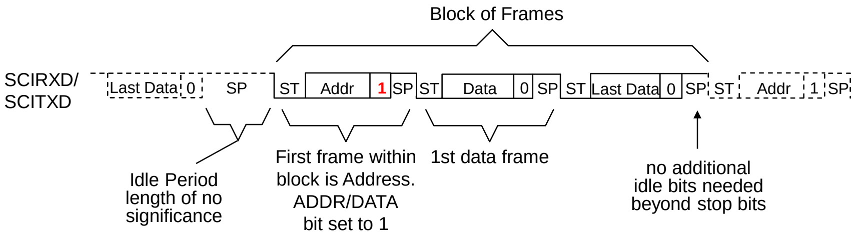
Idle-Line Wake-Up Mode

- 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

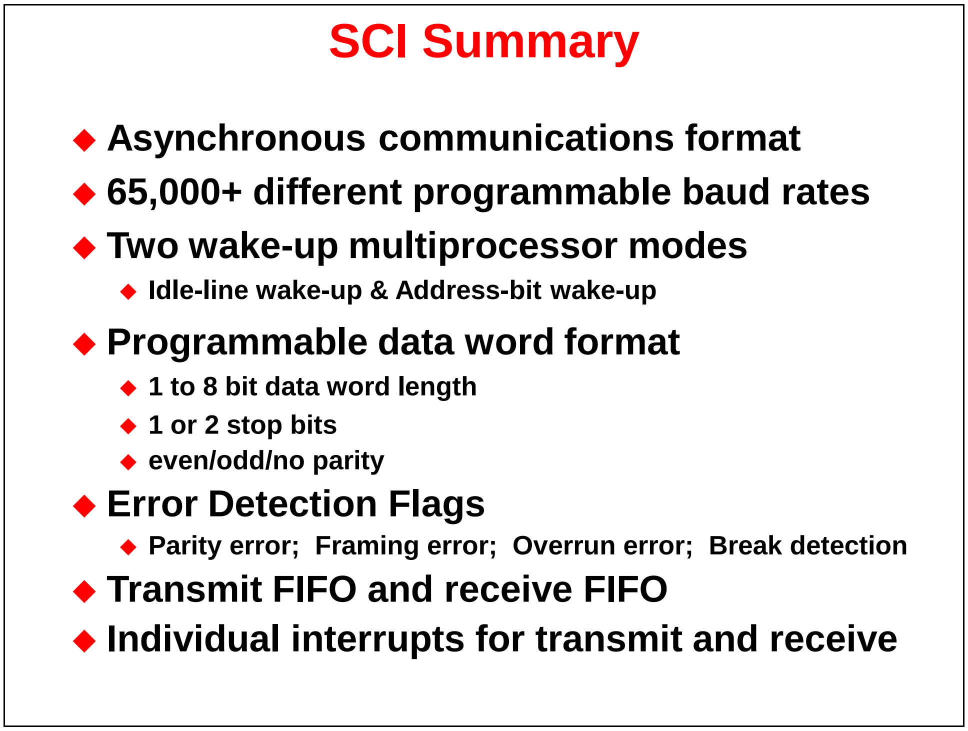
- All frames contain an extra address bit
- Receiver wakes up when address bit detected
- Automatic setting of Addr/Data bit in frame by setting TXWAKE = 1 prior to writing address to SCITXBUF



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

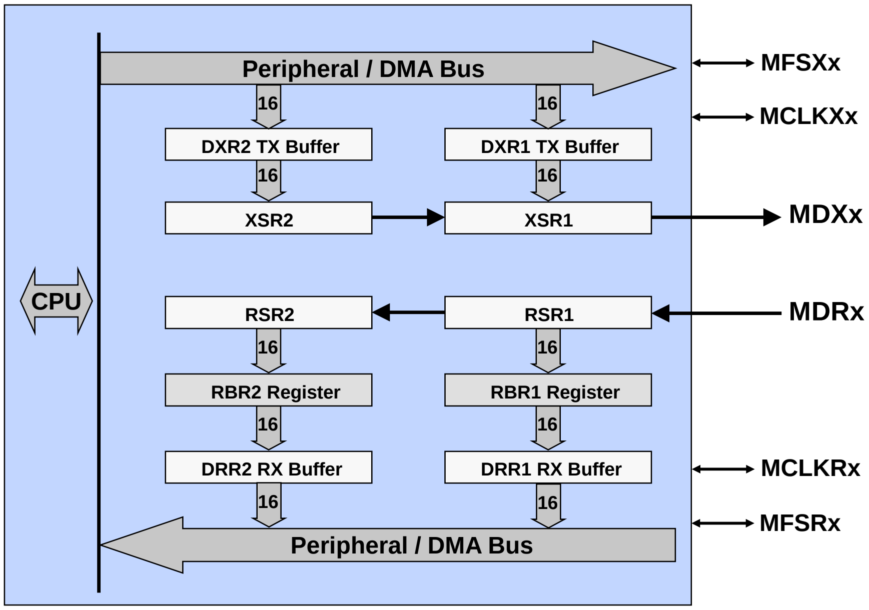


SCI Summary

- **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**

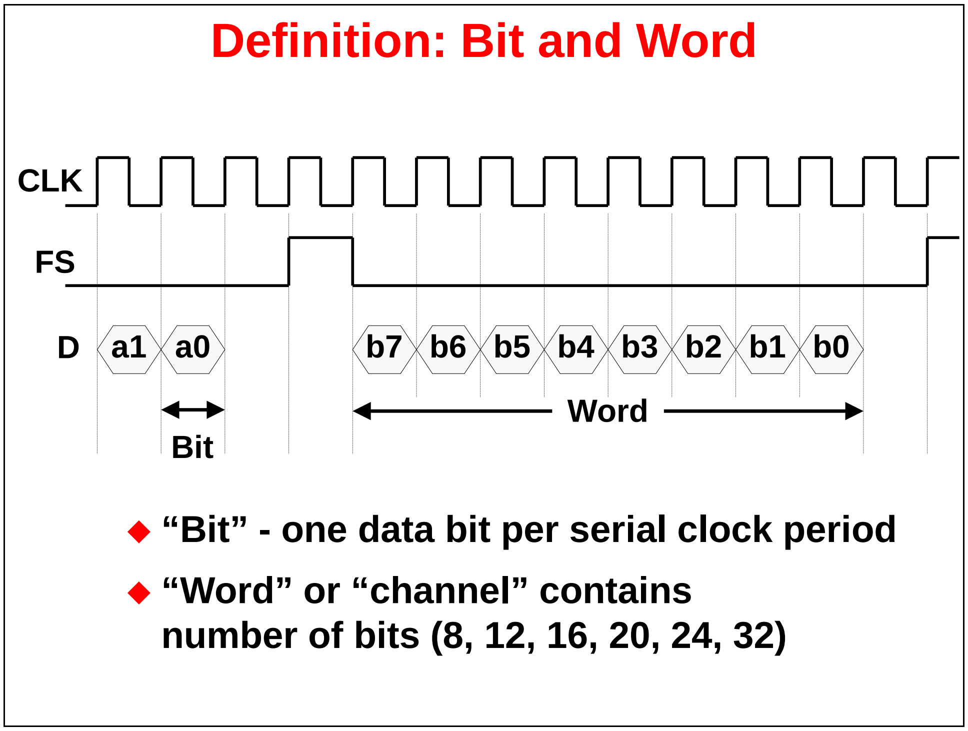
Multichannel Buffered Serial Port (McBSP)

McBSP Block Diagram



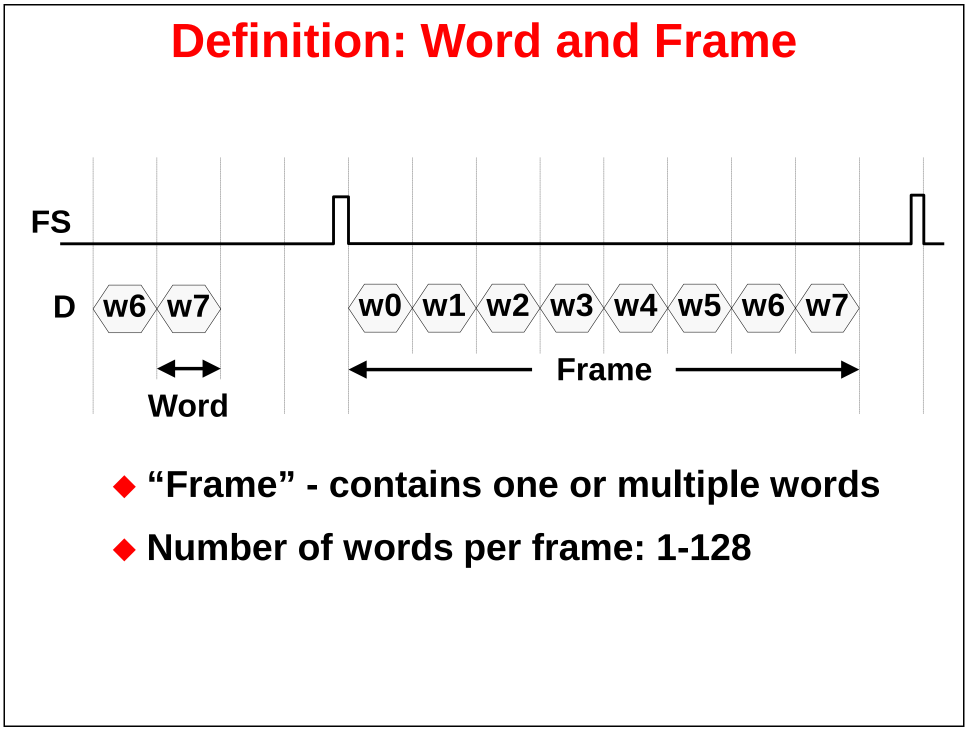
The McBSP provides a high-speed direct interface to codecs, analog interface chips (AICs), and other serially connected A/D and D/A devices. It has double-buffered transmission and triple-buffered reception for supporting continuous data streams. There are 128 channels for transmission and reception, and data size selections of 8, 12, 16, 20, 24, and 32 bits, along with µ-law and A-law companding.

Definition: Bit, Word, and Frame



Definition: Bit and Word

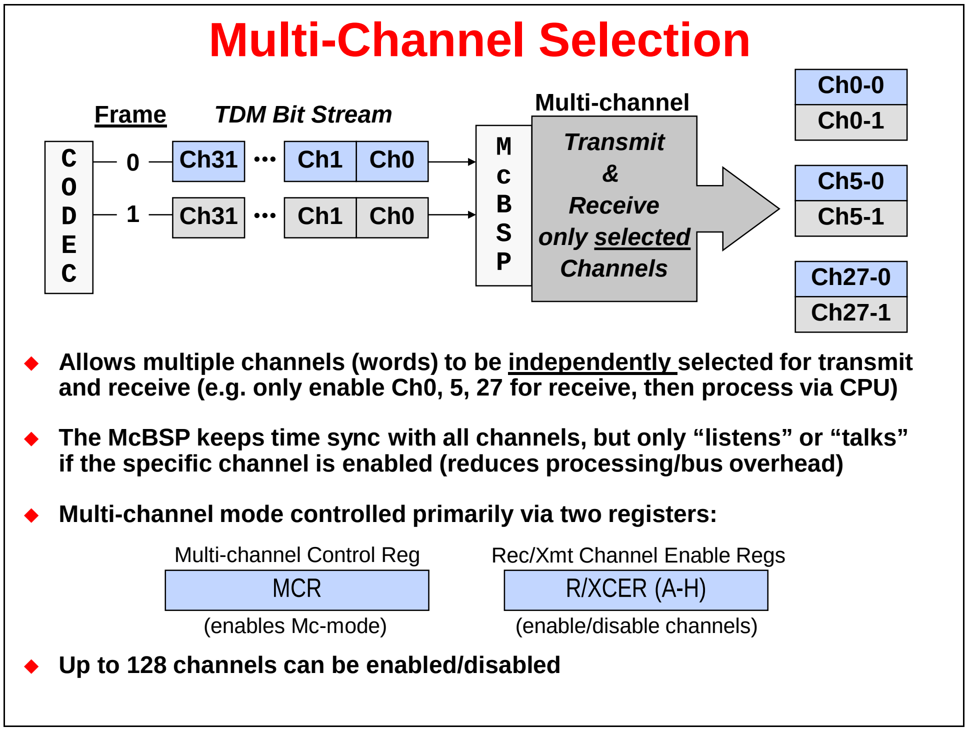
- “Bit” - one data bit per serial clock period
- “Word” or “channel” contains number of bits (8, 12, 16, 20, 24, 32)



Definition: Word and Frame

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

Multi-Channel Selection



Multi-Channel Selection

Frame

TDM Bit Stream

CODEC

0 Ch31 ··· Ch1 Ch0

1 Ch31 ··· Ch1 Ch0

McBSP

Multi-channel

Transmit & Receive only selected Channels

Ch0-0

Ch0-1

Ch5-0

Ch5-1

Ch27-0

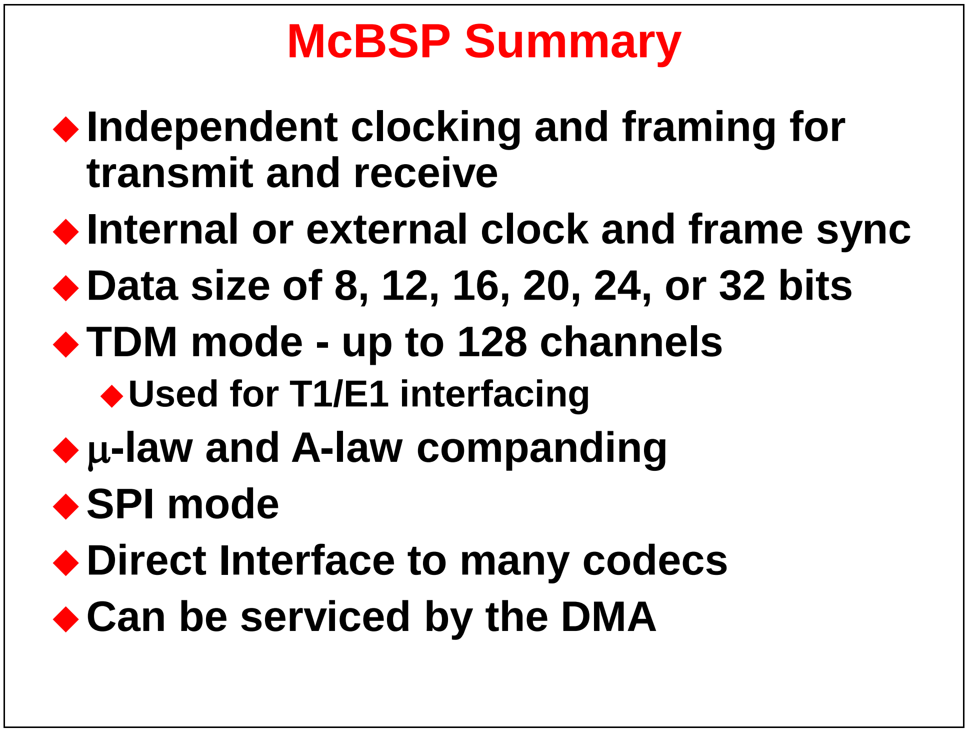
Ch27-1

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

Multi-channel Control Reg	Rec/Xmt Channel Enable Regs
MCR	R/XCER (A-H)
(enables Mc-mode)	(enable/disable channels)

- Up to 128 channels can be enabled/disabled

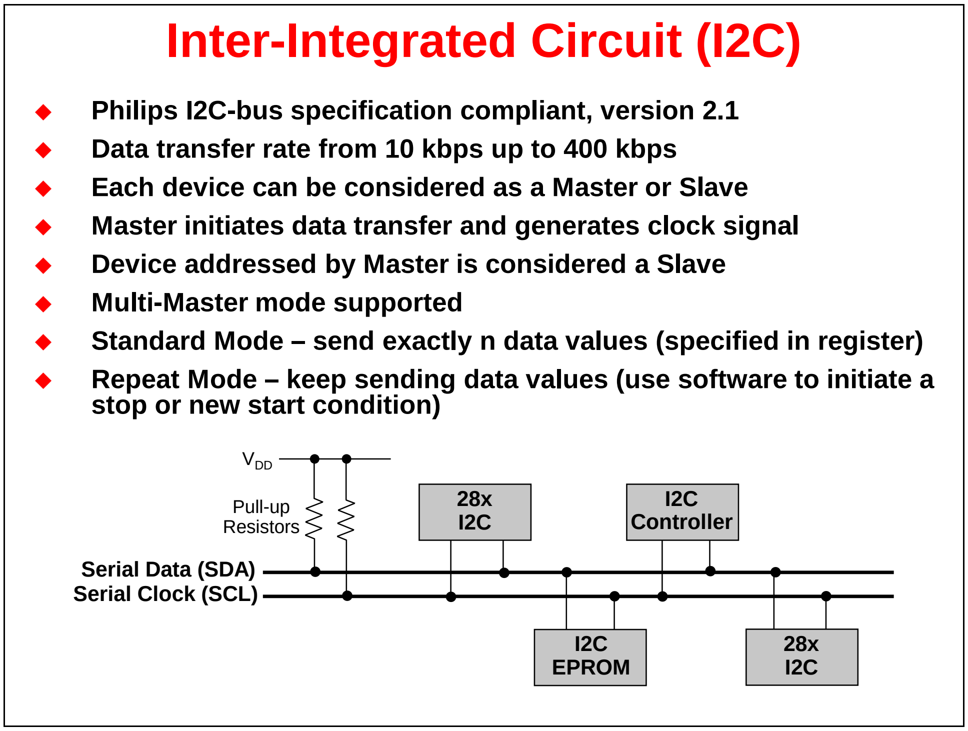
McBSP Summary



McBSP Summary

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

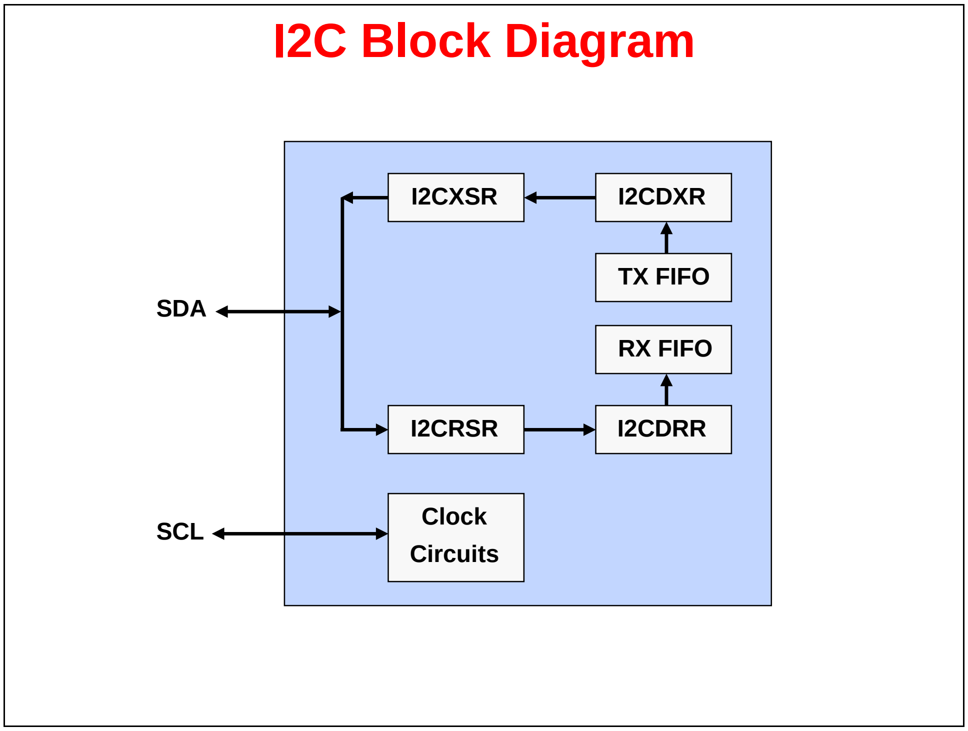
Inter-Integrated Circuit (I2C)



Inter-Integrated Circuit (I2C)

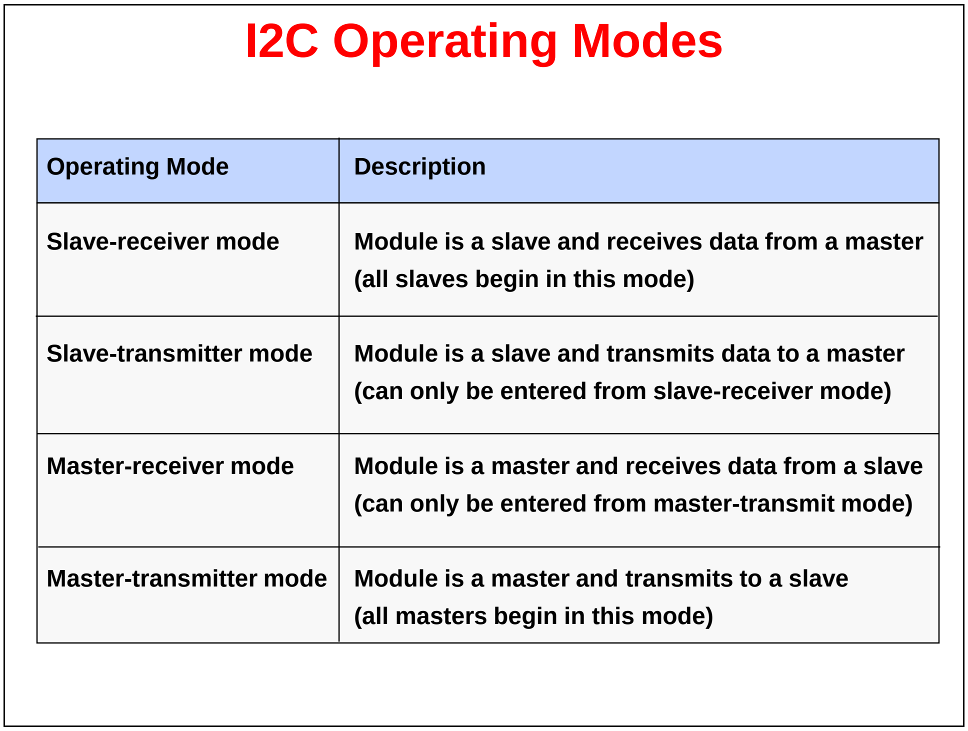
- Philips I2C-bus specification compliant, version 2.1
- Data transfer rate from 10 kbps up to 400 kbps
- Each device can be considered as a Master or Slave
- Master initiates data transfer and generates clock signal
- Device addressed by Master is considered a Slave
- Multi-Master mode supported
- Standard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

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.



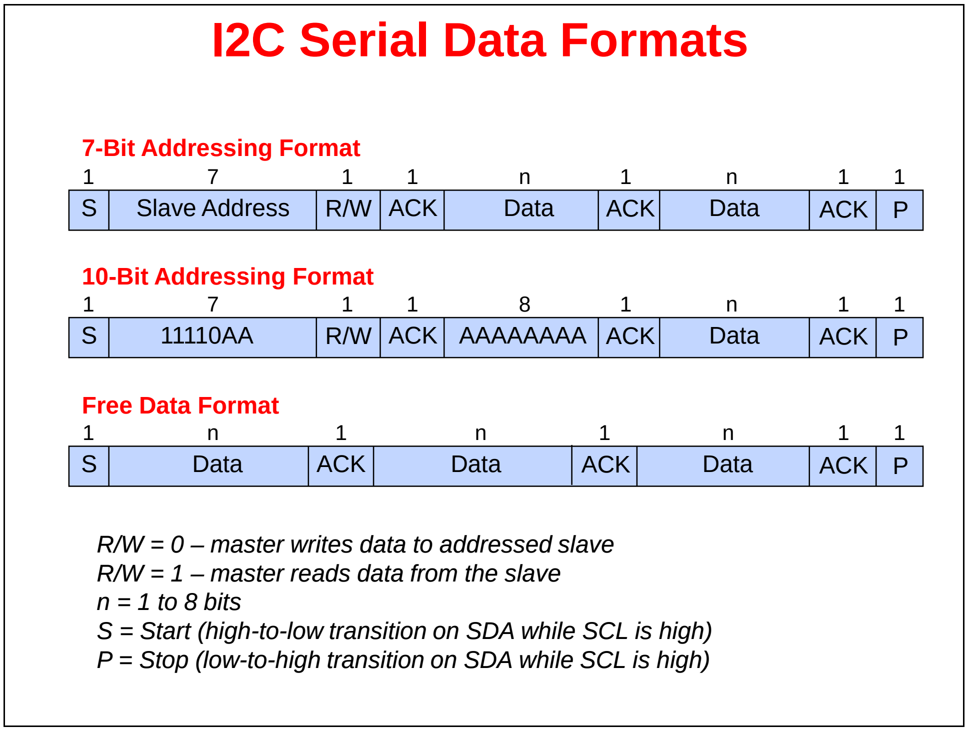
I2C Block Diagram

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

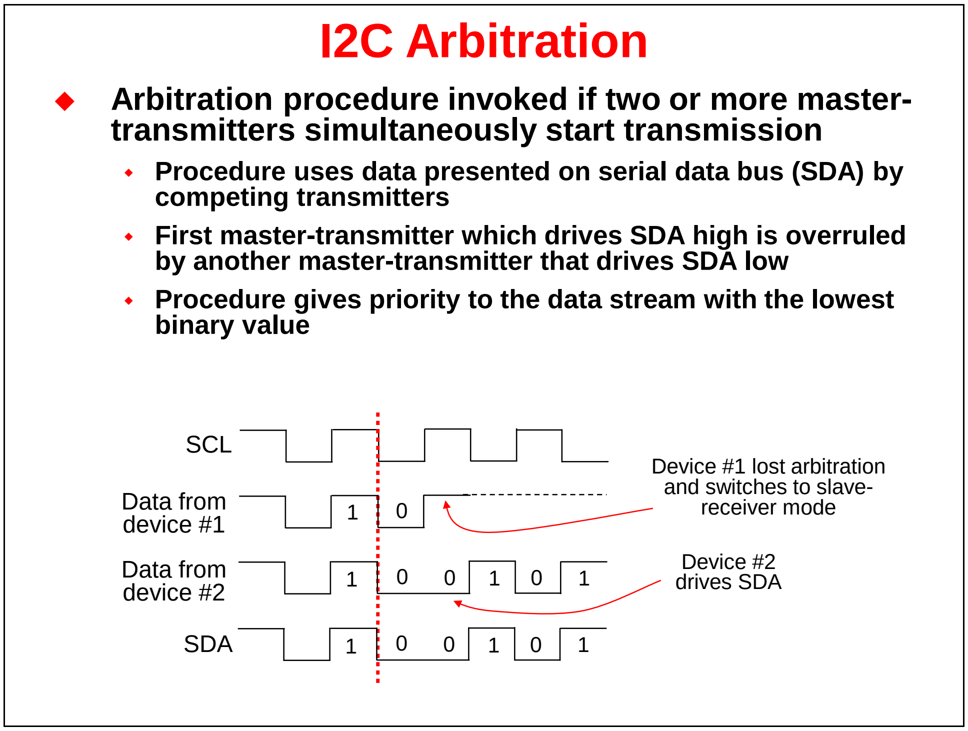
10-Bit Addressing Format

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

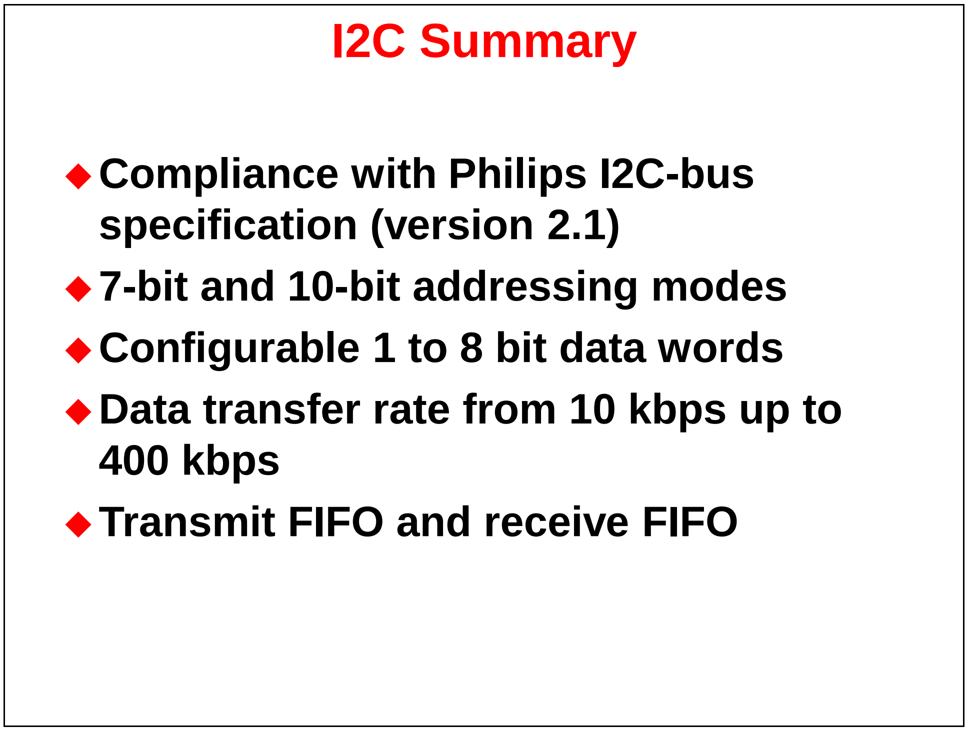
R/W = 0 – master writes data to addressed slave
R/W = 1 – master reads data from the slave
n = 1 to 8 bits
S = Start (high-to-low transition on SDA while SCL is high)
P = Stop (low-to-high transition on SDA while SCL is high)



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**

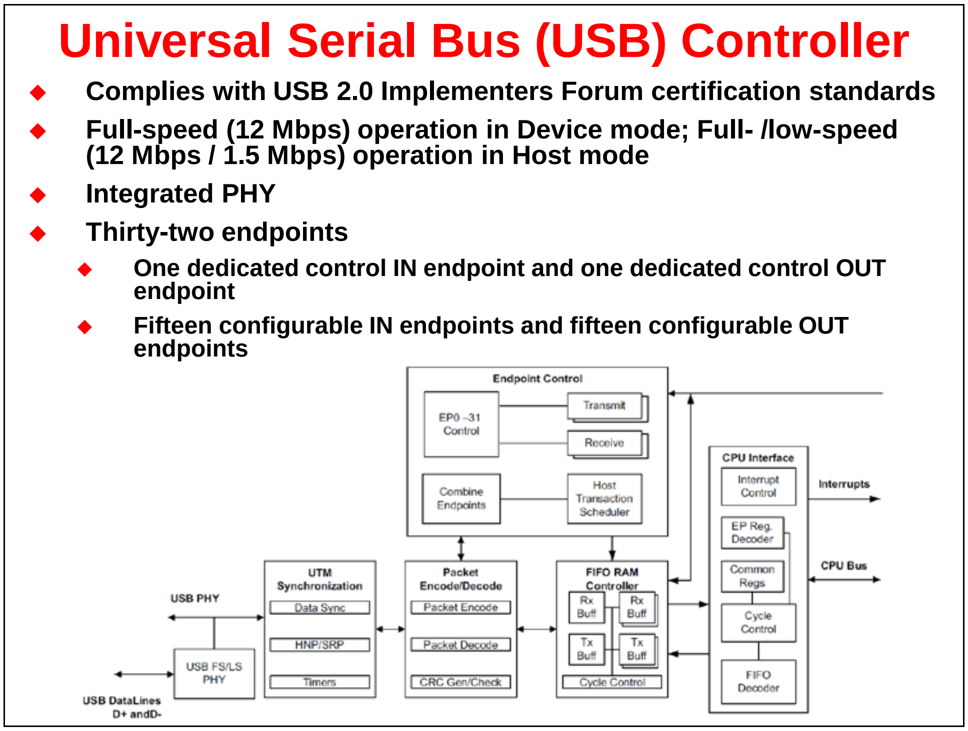
I2C Summary



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**

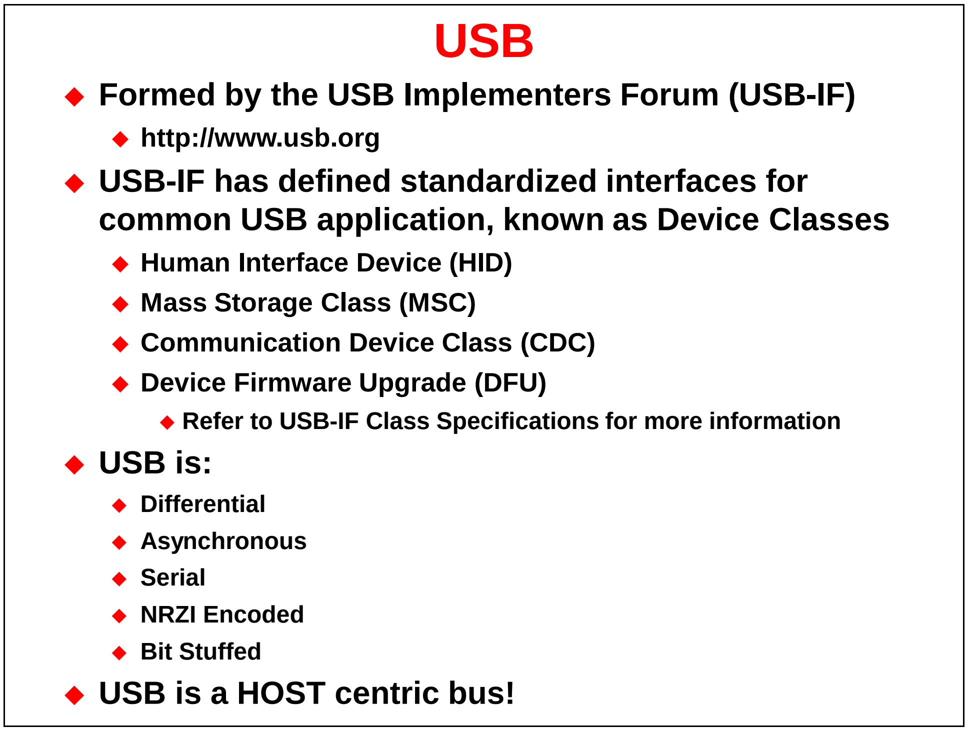
Universal Serial Bus (USB)



Universal Serial Bus (USB) Controller

- Complies with USB 2.0 Implementers Forum certification standards
- Full-speed (12 Mbps) operation in Device mode; Full- /low-speed (12 Mbps / 1.5 Mbps) operation in Host mode
- Integrated PHY
- Thirty-two endpoints
  - One dedicated control IN endpoint and one dedicated control OUT endpoint
  - Fifteen configurable IN endpoints and fifteen configurable OUT endpoints

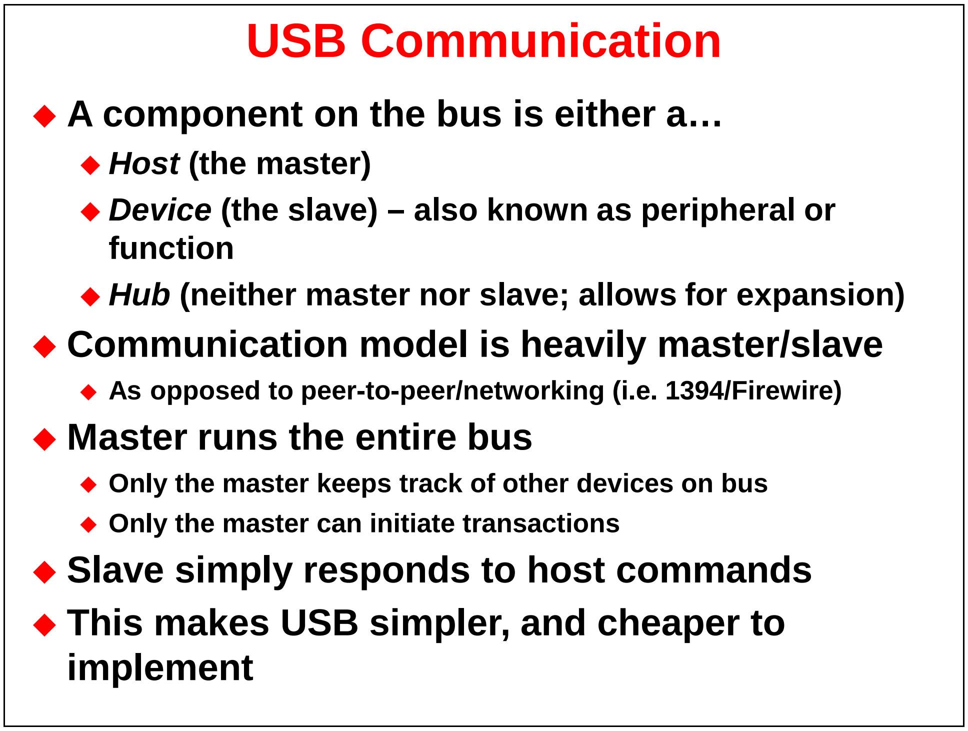
The USB operates as a full-speed function controller during point-to-point communications with a USB host. It complies with the USB 2.0 standard, and a dynamically sizeable FIFO supports queuing of multiple packets.



USB

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

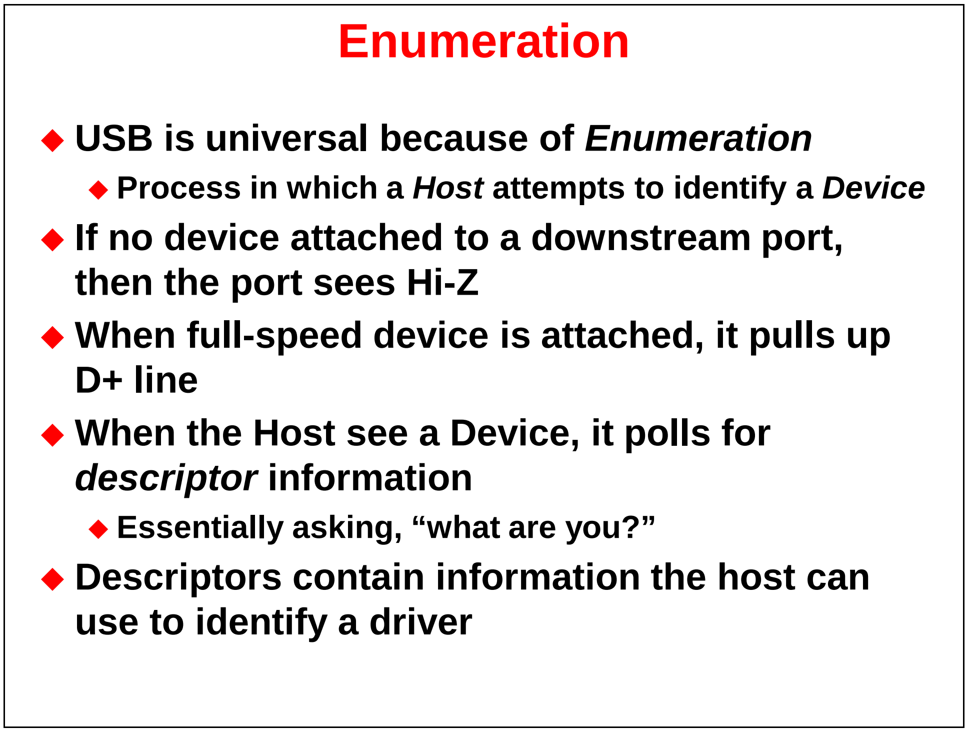
USB Communication



USB Communication

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

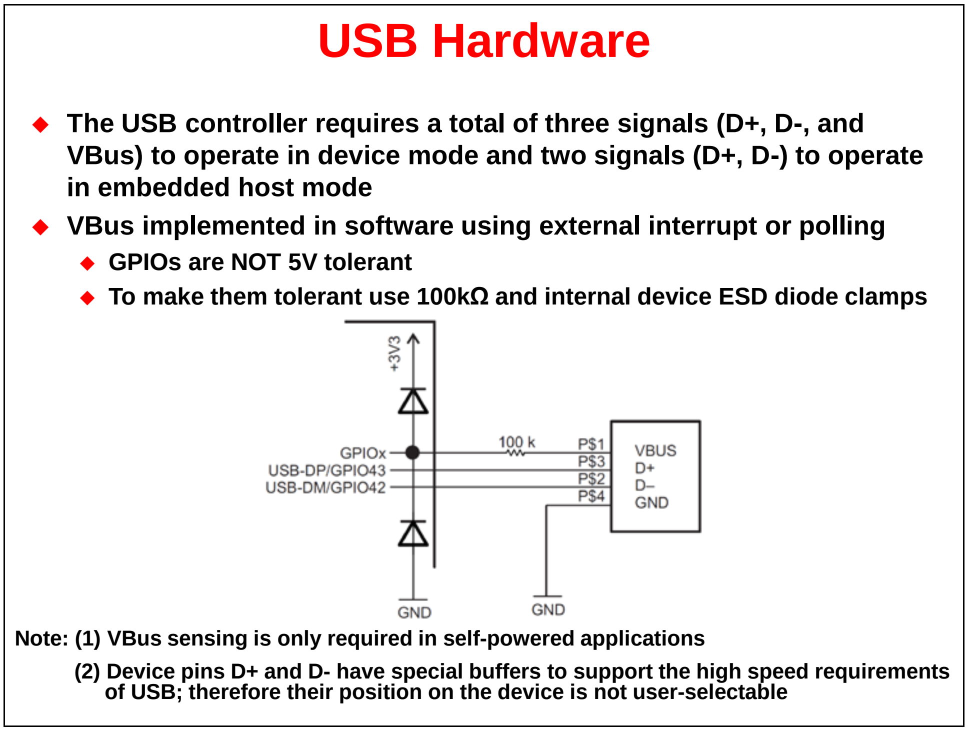
Enumeration



Enumeration

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

F28x USB Hardware



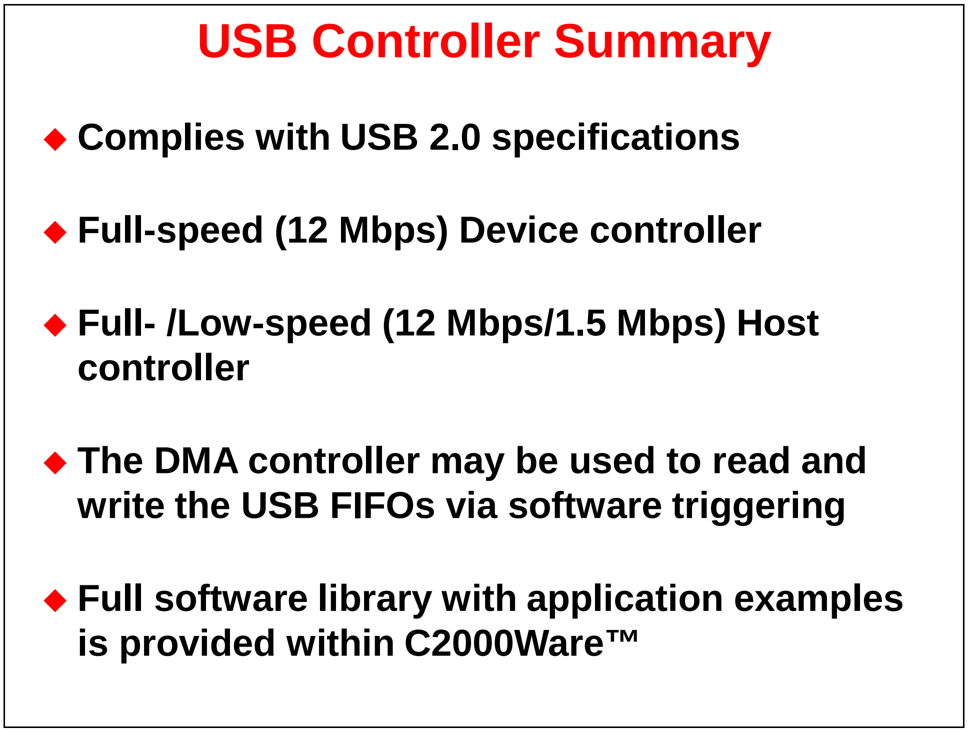
USB Hardware

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

Note: (1) VBus sensing is only required in self-powered applications

(2) Device pins D+ and D- have special buffers to support the high speed requirements of USB; therefore their position on the device is not user-selectable

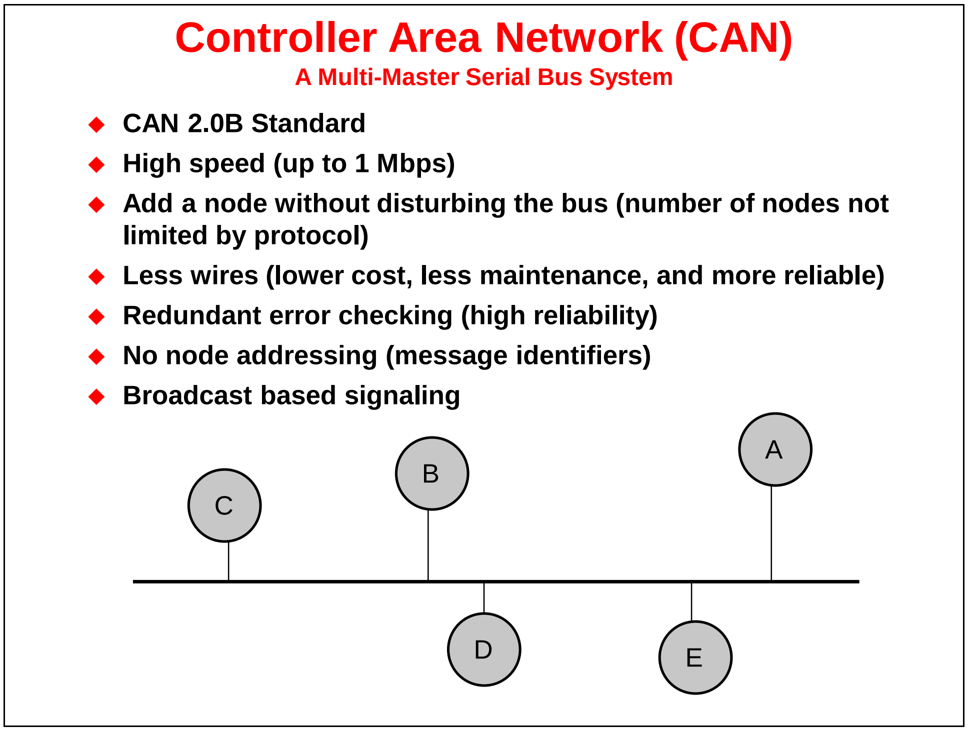
USB Controller Summary



USB Controller Summary

- Complies with USB 2.0 specifications
- Full-speed (12 Mbps) Device controller
- Full- /Low-speed (12 Mbps/1.5 Mbps) Host controller
- The DMA controller may be used to read and write the USB FIFOs via software triggering
- Full software library with application examples is provided within C2000Ware™

Controller Area Network (CAN)



Controller Area Network (CAN)

A Multi-Master Serial Bus System

- **CAN 2.0B Standard**
- **High speed (up to 1 Mbps)**
- **Add a node without disturbing the bus (number of nodes not limited by protocol)**
- **Less wires (lower cost, less maintenance, and more reliable)**
- **Redundant error checking (high reliability)**
- **No node addressing (message identifiers)**
- **Broadcast based signaling**

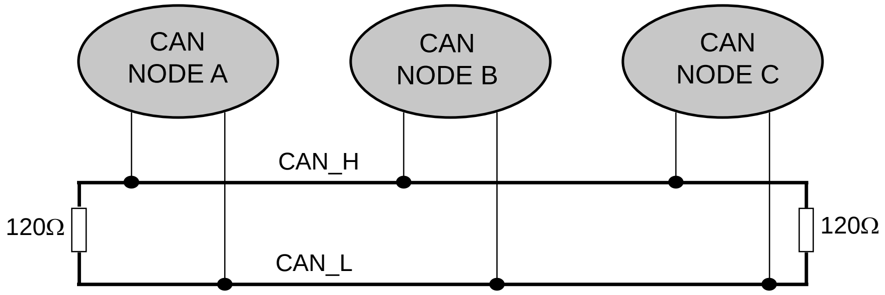
The CAN module is a serial communications protocol that efficiently supports distributed real-time control with a high level of security. It supports bit-rates up to 1 Mbit/s and is compliant with the CAN 2.0B protocol specification.

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

CAN Bus

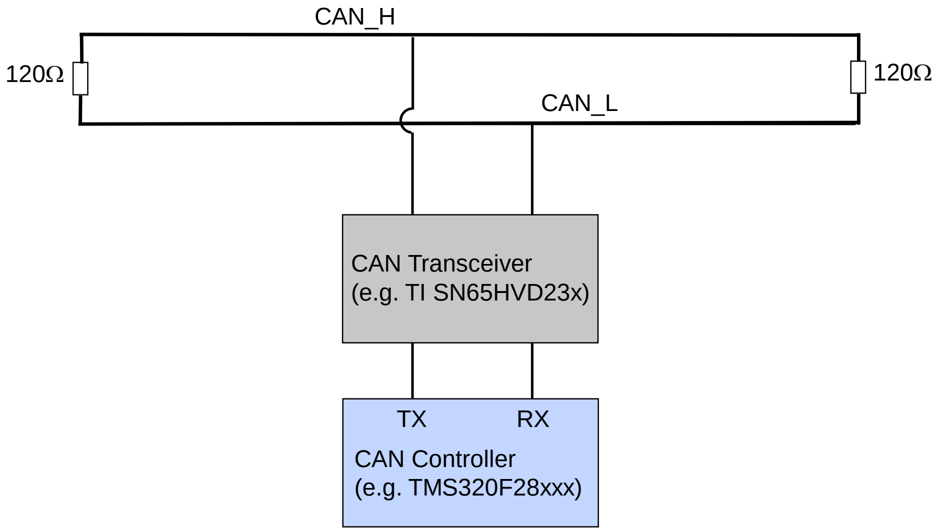
- Two wire differential bus (usually twisted pair)
- Max. bus length depend on transmission rate
  - 40 meters @ 1 Mbps



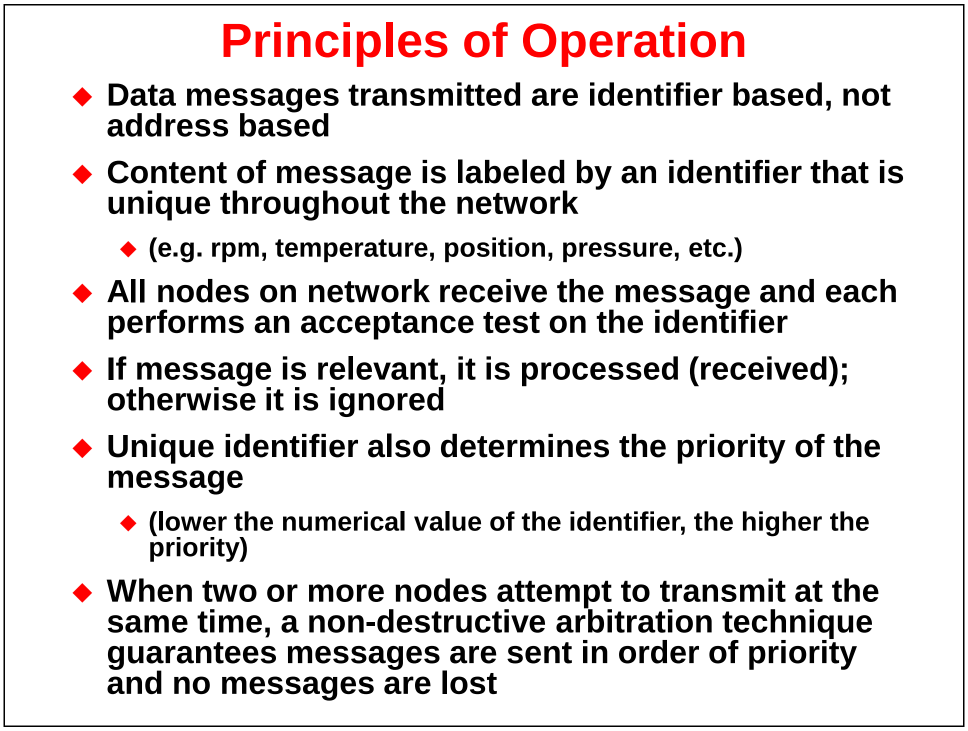
The MCU communicates to the CAN Bus using a transceiver. The CAN bus is a twisted pair wire and the transmission rate depends on the bus length. If the bus is less than 40 meters the transmission rate is capable up to 1 Mbit/second.

CAN Node

Wired-AND Bus Connection

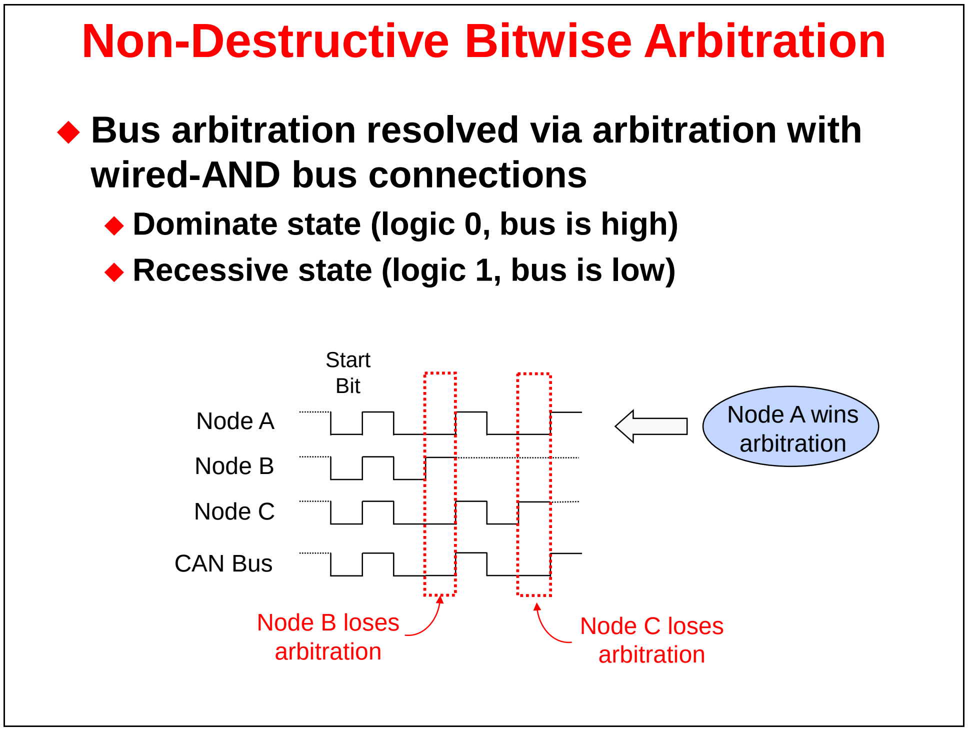


Principles of Operation



Principles of Operation

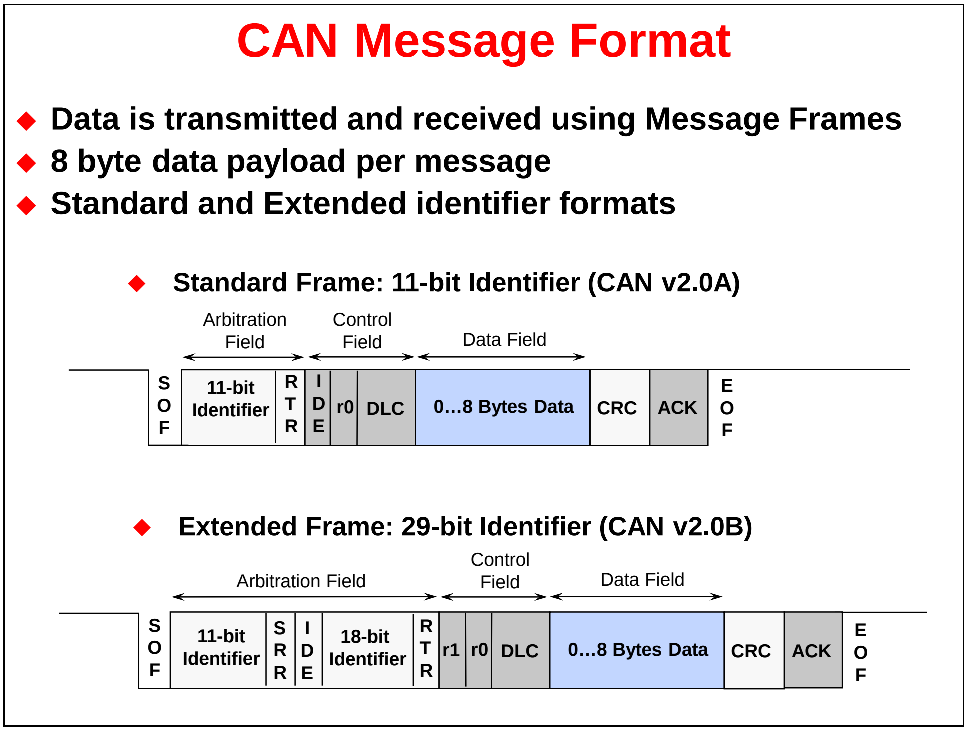
- Data messages transmitted are identifier based, not address based
- Content of message is labeled by an identifier that is unique throughout the network
  - (e.g. rpm, temperature, position, pressure, etc.)
- All nodes on network receive the message and each performs an acceptance test on the identifier
- If message is relevant, it is processed (received); otherwise it is ignored
- Unique identifier also determines the priority of the message
  - (lower the numerical value of the identifier, the higher the priority)
- When two or more nodes attempt to transmit at the same time, a non-destructive arbitration technique guarantees messages are sent in order of priority and no messages are lost



Non-Destructive Bitwise Arbitration

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

Message Format and Block Diagram



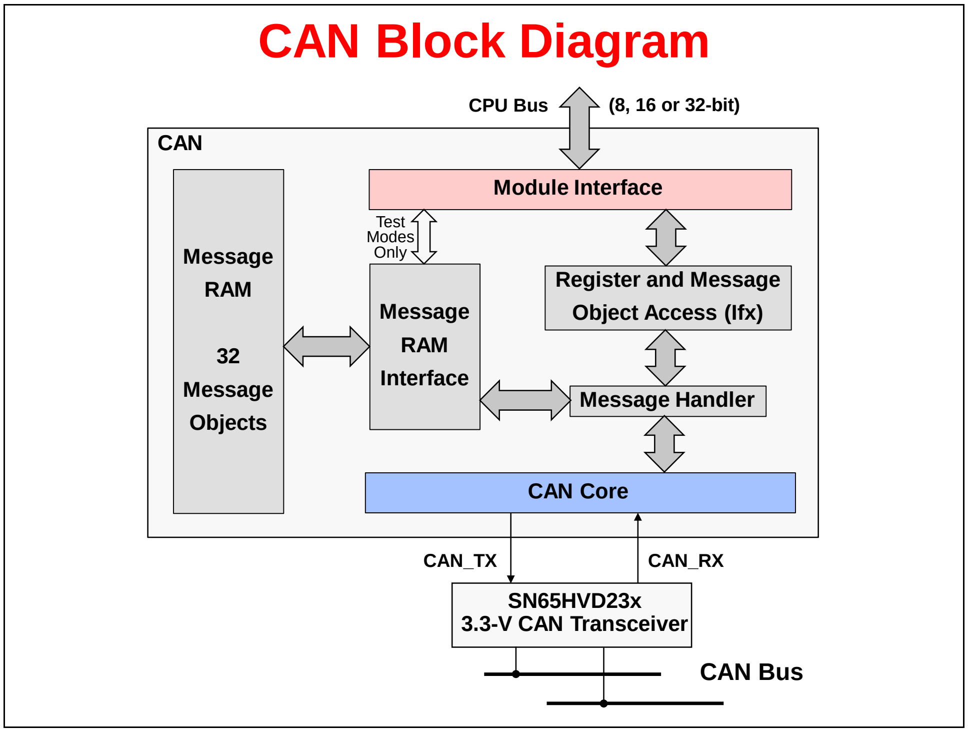
CAN Message Format

- Data is transmitted and received using Message Frames
- 8 byte data payload per message
- Standard and Extended identifier formats

- Standard Frame: 11-bit Identifier (CAN v2.0A)

- Extended Frame: 29-bit Identifier (CAN v2.0B)

The MCU CAN module is a full CAN Controller. It contains a message handler for transmission and reception management, and frame storage. The specification is CAN 2.0B Active – that is, the module can send and accept standard (11-bit identifier) and extended frames (29-bit identifier).



CAN Block Diagram

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

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

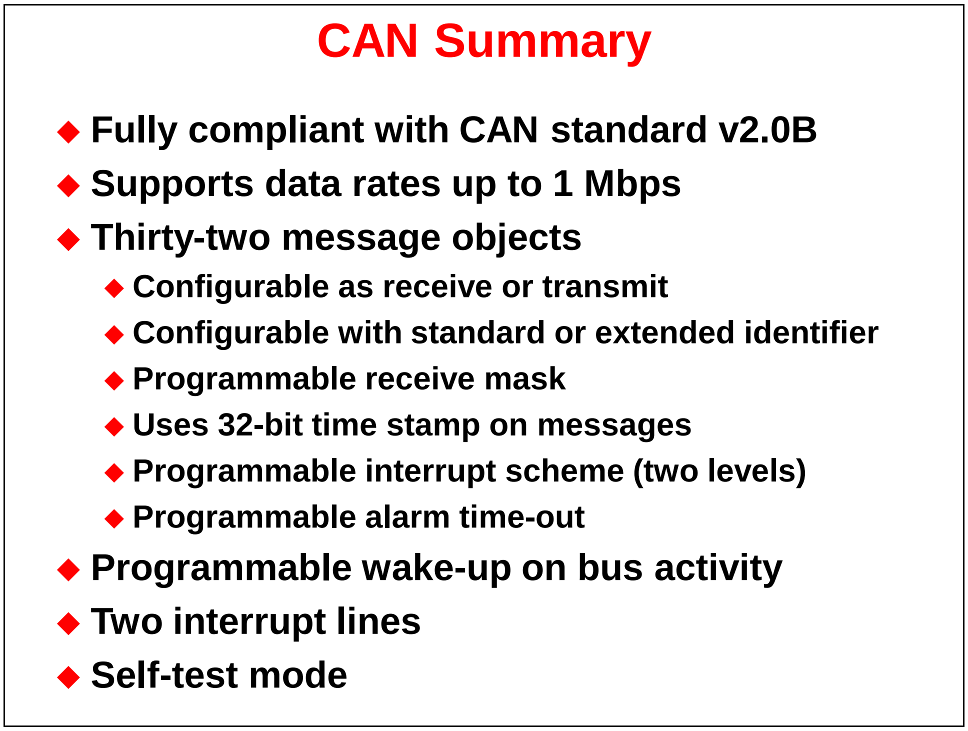
The CAN module mailboxes are divided into several parts:

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

The CAN module contains registers which are divided into five groups:

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

CAN Summary



CAN Summary

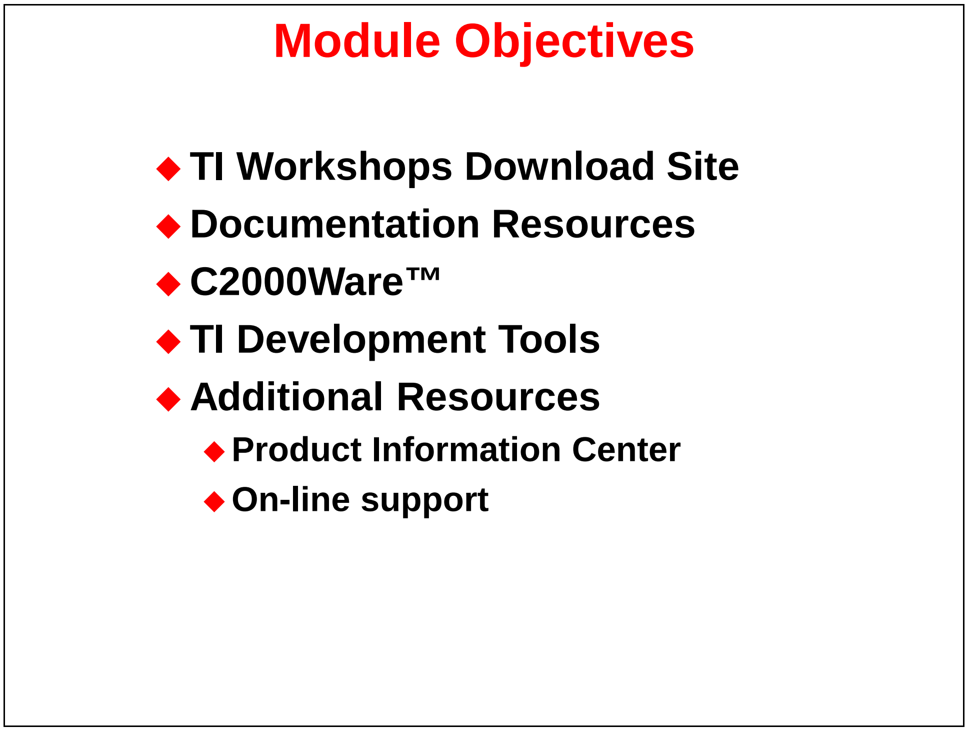
- **Fully compliant with CAN standard v2.0B**
- **Supports data rates up to 1 Mbps**
- **Thirty-two message objects**
  - **Configurable as receive or transmit**
  - **Configurable with standard or extended identifier**
  - **Programmable receive mask**
  - **Uses 32-bit time stamp on messages**
  -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

Support Resources

Introduction

This module contains various references to support the development process.

Module Objectives



Module Objectives

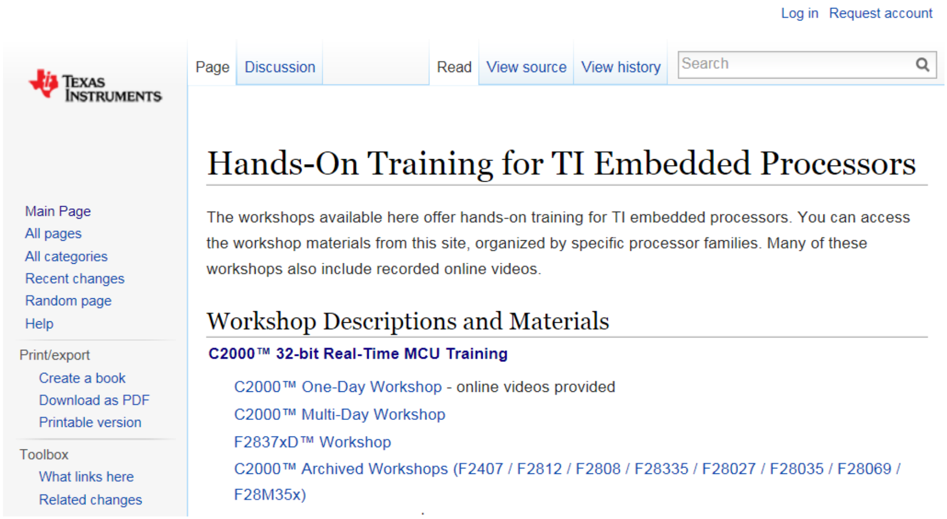
- TI Workshops Download Site
- Documentation Resources
- C2000Ware™
- TI Development Tools
- Additional Resources
  - Product Information Center
  - On-line support

Chapter Topics

TI Support Resources

C2000 Workshop Download Wiki

C2000 Workshop Download Wiki



Hands-On Training for TI Embedded Processors

The workshops available here offer hands-on training for TI embedded processors. You can access the workshop materials from this site, organized by specific processor families. Many of these workshops also include recorded online videos.

Workshop Descriptions and Materials

C2000™ 32-bit Real-Time MCU Training

- C2000™ One-Day Workshop - online videos provided
- C2000™ Multi-Day Workshop
- F2837xD™ Workshop
- C2000™ Archived Workshops (F2407 / F2812 / F2808 / F28335 / F28027 / F28035 / F28069 / F28M35x)

http://www.ti.com/hands-on-training

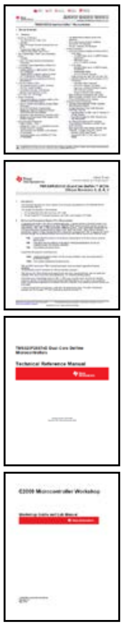
At the C2000 Workshop Download Wiki you will find all of the materials for the C2000 One-day and Multi-day Workshops, as well as the C2000 archived workshops, which include support for the following device families:

- F2407
- F2812
- F2808
- F28335
- F28027
- F28035
- F28069
- F28M35x

Documentation Resources

Documentation Resources

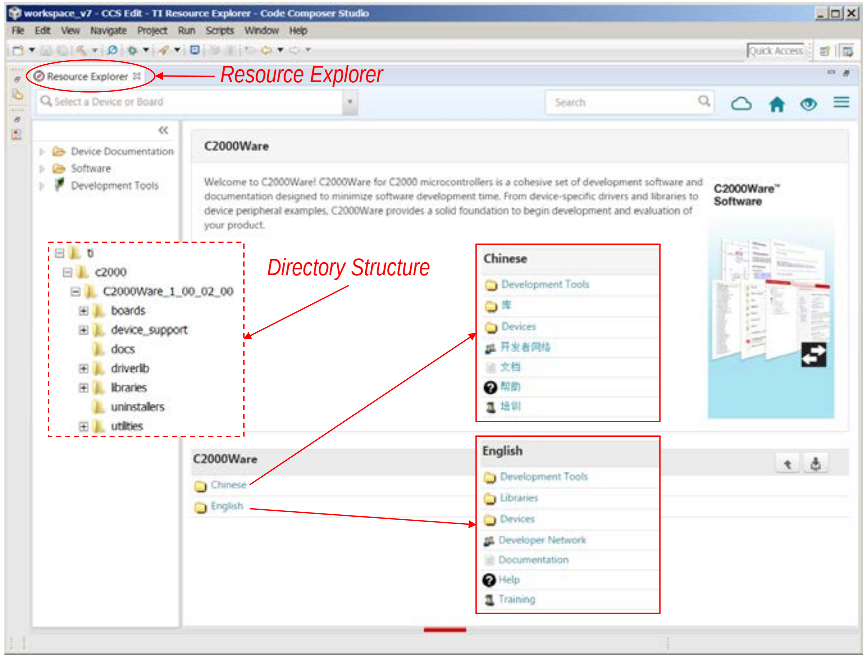
- Data Sheet
  - Contains device electrical characteristics and timing specifications
  - Key document for hardware engineers
- Silicon Errata
  - Contains deviations from original specifications
  - Includes silicon revision history
- Technical Reference Manual (TRM)
  - Contains architectural descriptions 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™

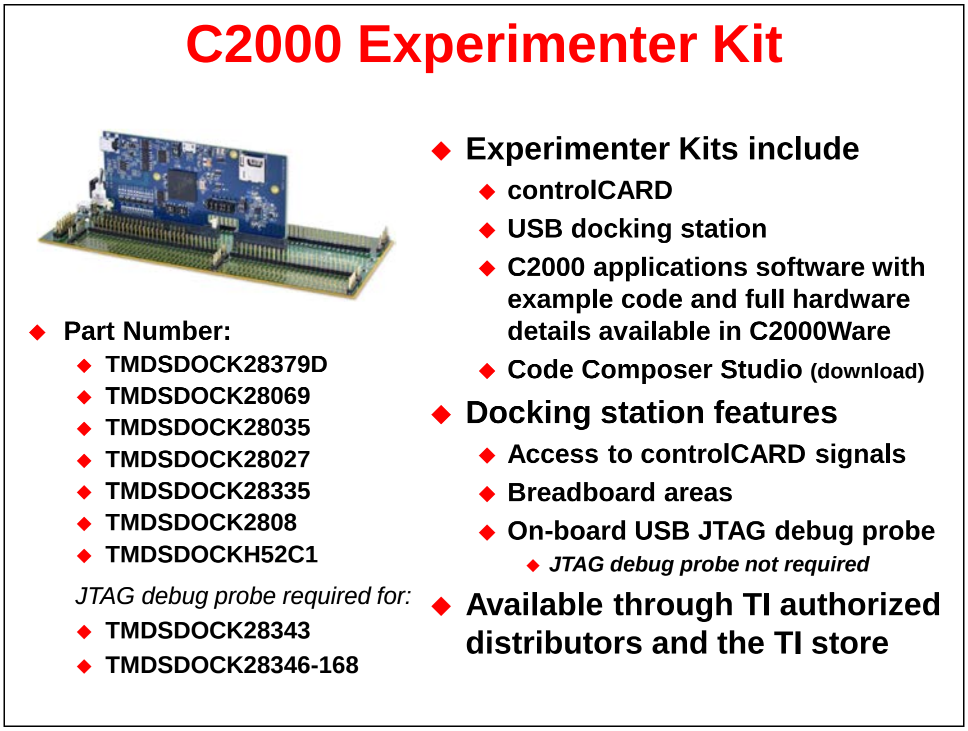
C2000Ware™



Welcome to C2000Ware! C2000Ware for C2000 microcontrollers is a cohesive set of development software and documentation designed to minimize software development time. From device-specific drivers and libraries to device peripheral examples, C2000Ware provides a solid foundation to begin development and evaluation of your product.

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:
  - TMDSDOCK28379D
  - TMDSDOCK28069
  - TMDSDOCK28035
  - TMDSDOCK28027
  - TMDSDOCK28335
  - TMDSDOCK2808
  - TMDSDOCKH52C1

JTAG debug probe required for:

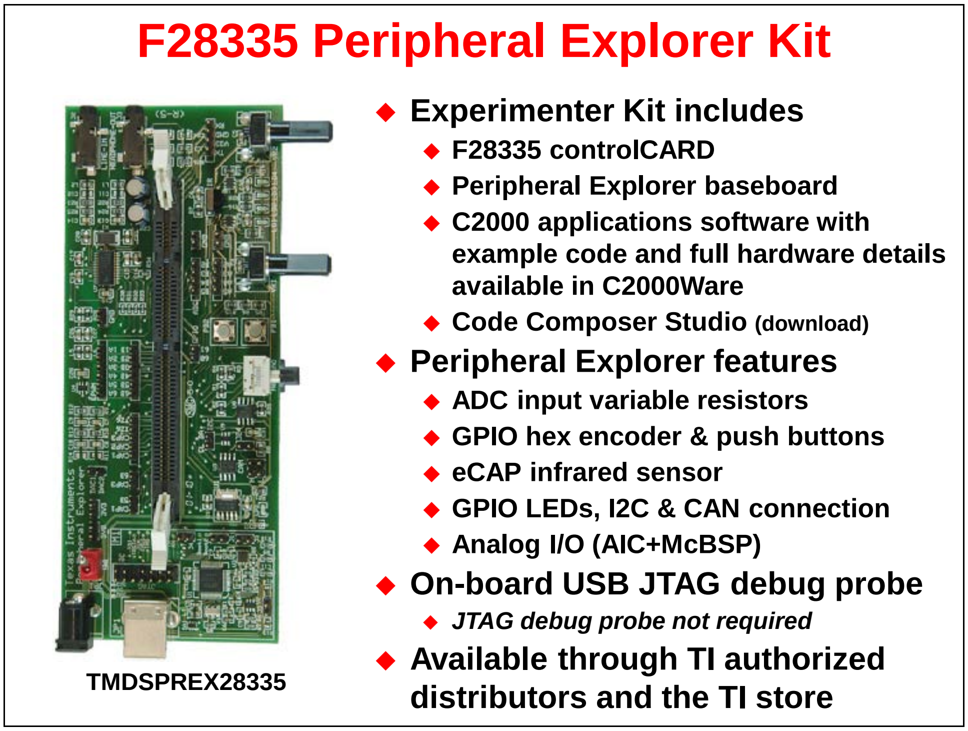
  - TMDSDOCK28343
  - TMDSDOCK28346-168

- 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*
- 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



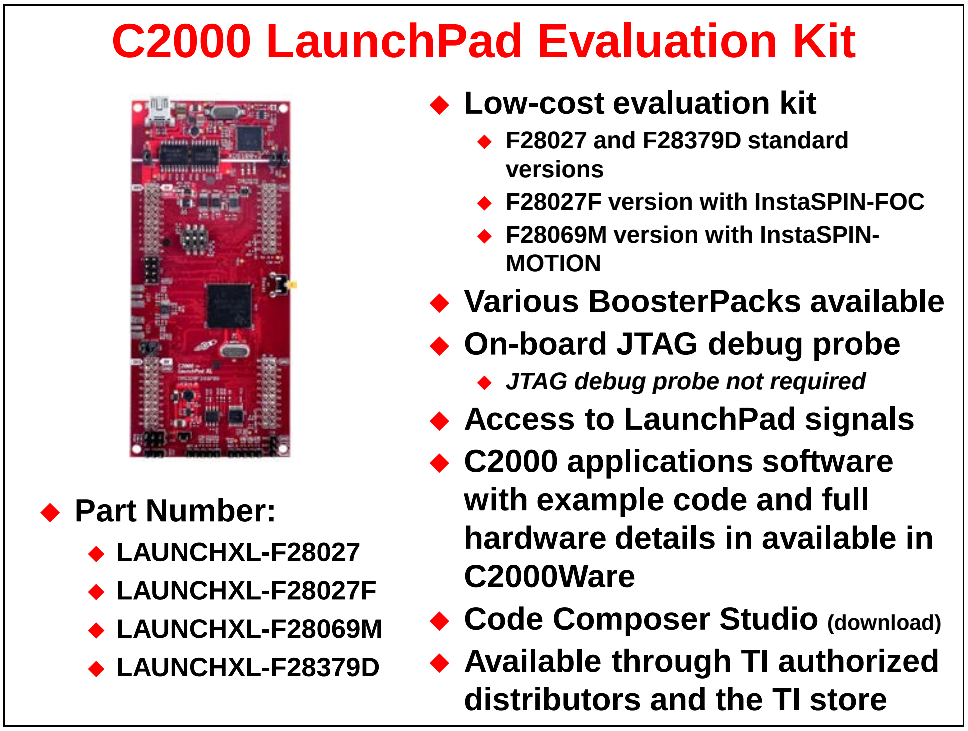
F28335 Peripheral Explorer Kit

TMDSPREX28335

- **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**

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



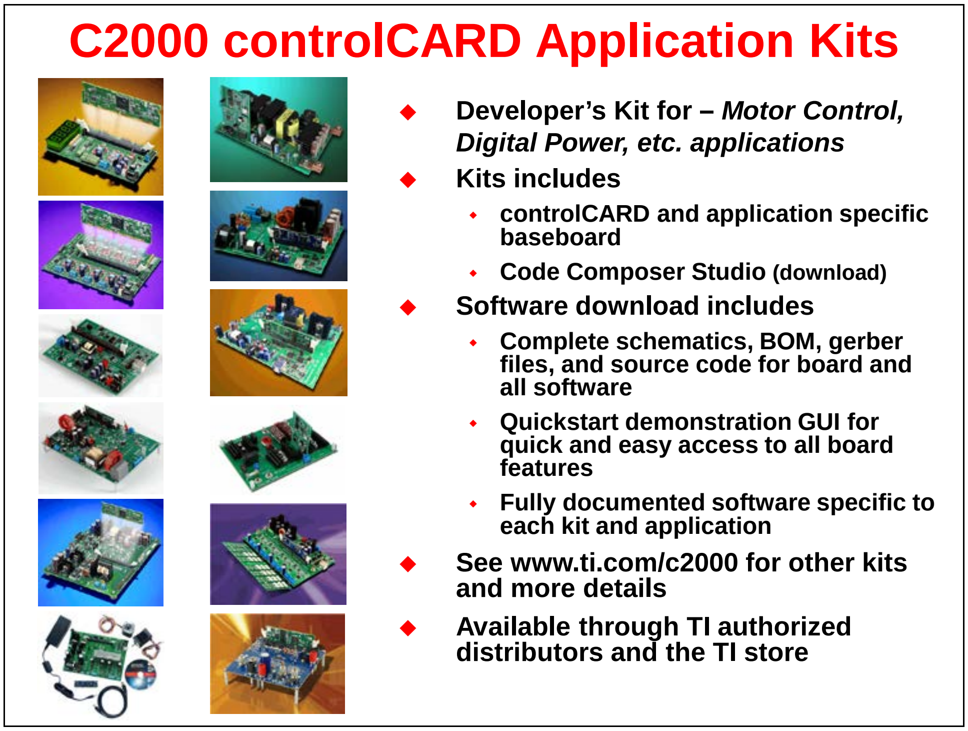
C2000 LaunchPad Evaluation Kit

- 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

- Part Number:
  - LAUNCHXL-F28027
  - LAUNCHXL-F28027F
  - LAUNCHXL-F28069M
  - LAUNCHXL-F28379D

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

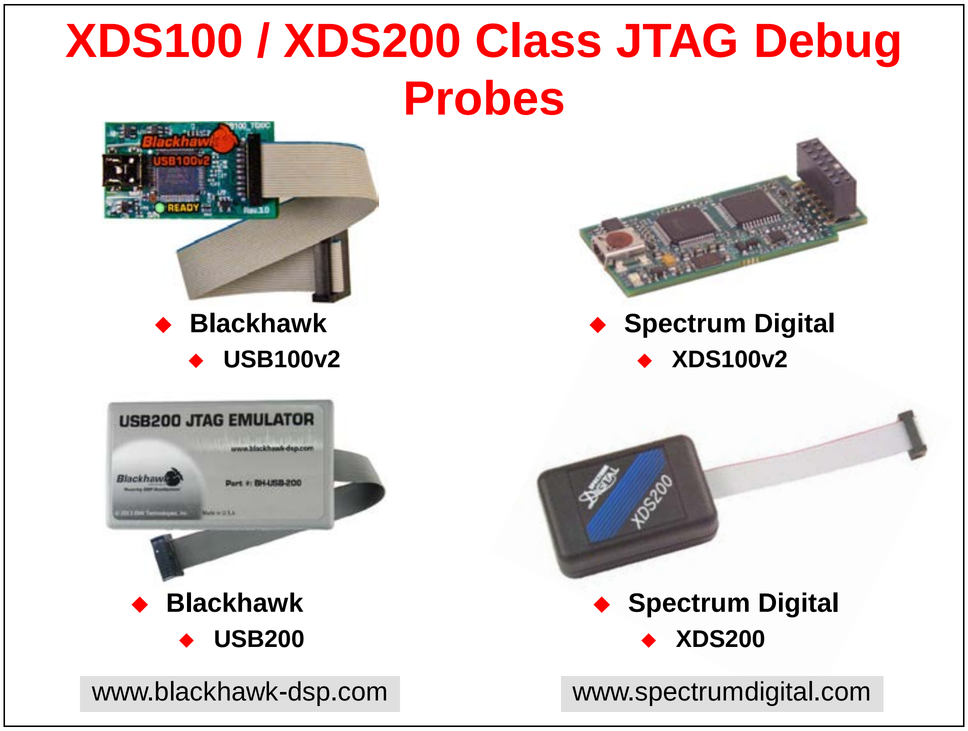


C2000 controlCARD Application Kits

- 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 / XDS200 Class JTAG Debug Probes



XDS100 / XDS200 Class JTAG Debug Probes

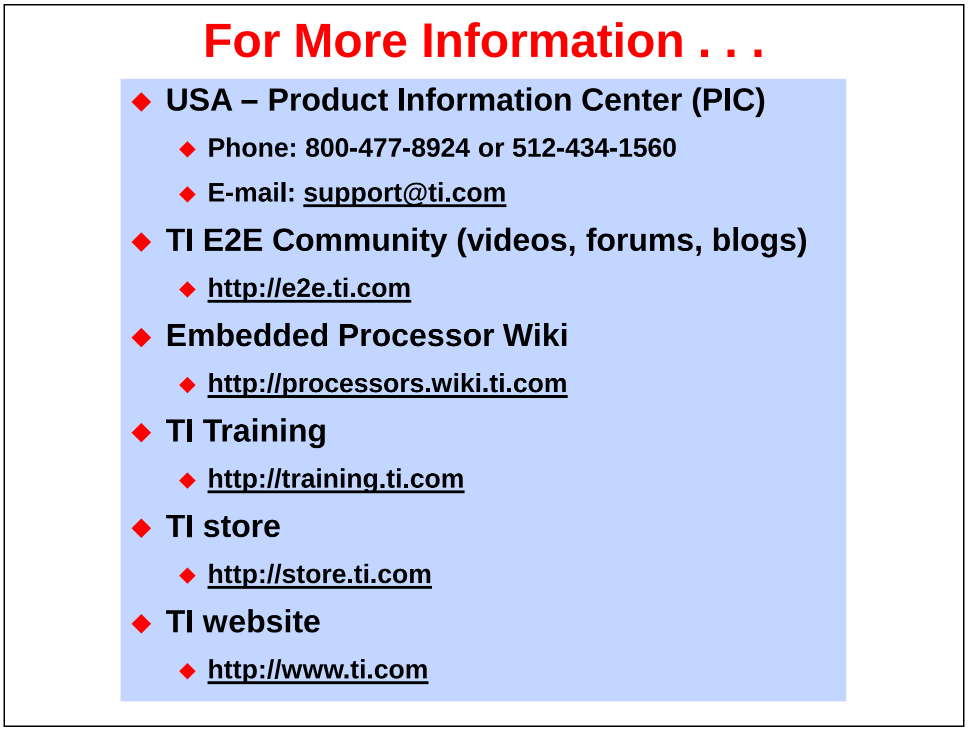
- Blackhawk
  - USB100v2
- Spectrum Digital
  - XDS100v2
- Blackhawk
  - USB200
- Spectrum Digital
  - XDS200

www.blackhawk-dsp.com

www.spectrumdigital.com

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

Product Information Resources



For More Information . . .

- USA – Product Information Center (PIC)
  - Phone: 800-477-8924 or 512-434-1560
  - E-mail: support@ti.com
- TI E2E Community (videos, forums, blogs)
  - 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 – F28379D Experimenter Kit

Overview

This appendix provides a quick reference and mapping of the header pins used on the F28379D LaunchPad and F28379D Experimenter Kit. This allows either development board to be used with the workshop.

Chapter Topics

F28379D Experimenter Kit

Initial Hardware Setup

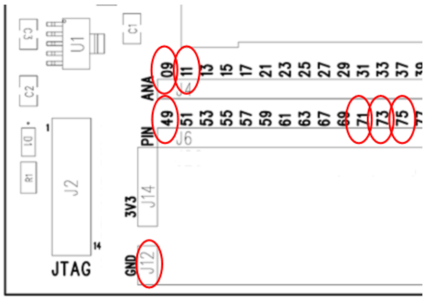
- **F28379D Experimenter Kit:**

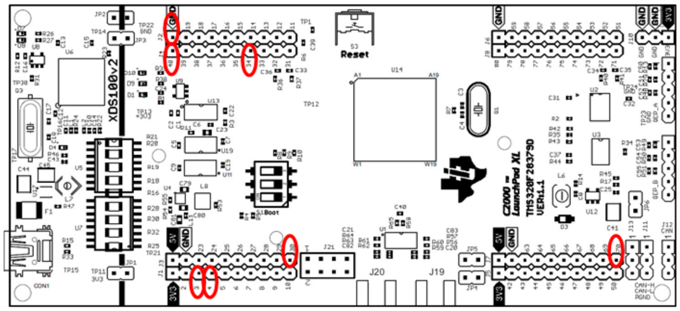
Insert the F28379D controlCARD into the Docking Station connector slot. Using the two (2) supplied USB cables – plug the USB Standard Type A connectors into the computer USB ports and plug the USB Mini-B connectors as follows:

- A:J1 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	J3-30
GND	GND	J2-20 (GND)
GPIO19	Pin # 73	J1-3
GPIO18	Pin # 71	J1-4
DACOUTB	ANA header, Pin # 11	J7-70
PWM1A	Pin # 49	J4-40
ECAP1 (via Input X-bar)	Pin # 75 (GPIO24)	J4-34 (GPIO24)

controlCARD and LaunchPad LED Mapping

Function	controlCARD	LaunchPad
LED – Power	LED LD1 (green)	LED D1 (green)
LED – GPIO31	LED LD2 (red)	LED D10 (blue)
LED – GPIO34	LED LD3 (red)	LED D9 (red)

Stand-Alone Operation (No Emulator)

When the device is in stand-alone boot mode, the state of GPIO72 and GPIO84 pins are used to determine the boot mode. On the controlCARD switch SW1 controls the boot options for the F28379D device. Check that switch SW1 positions 1 and 2 are set to the default "1 – on" position (both switches up). This will configure the device (in stand-alone boot mode) to GetMode. Since the OTP_KEY has not been programmed, the default GetMode will be boot from flash. Details of the switch positions can be found in the controlCARD information guide.