C2000™ MCU 1-Day Workshop

Workshop Guide and Lab Manual

TEXAS INSTRUMENTS

C2000 MCU 1-Day Workshop
Revision 2.0
November 2016

Important Notice

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

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

Customers are responsible for their applications using TI components.

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

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

Revision History

April 2014 – Revision 1.0

October 2014 – Revision 1.1

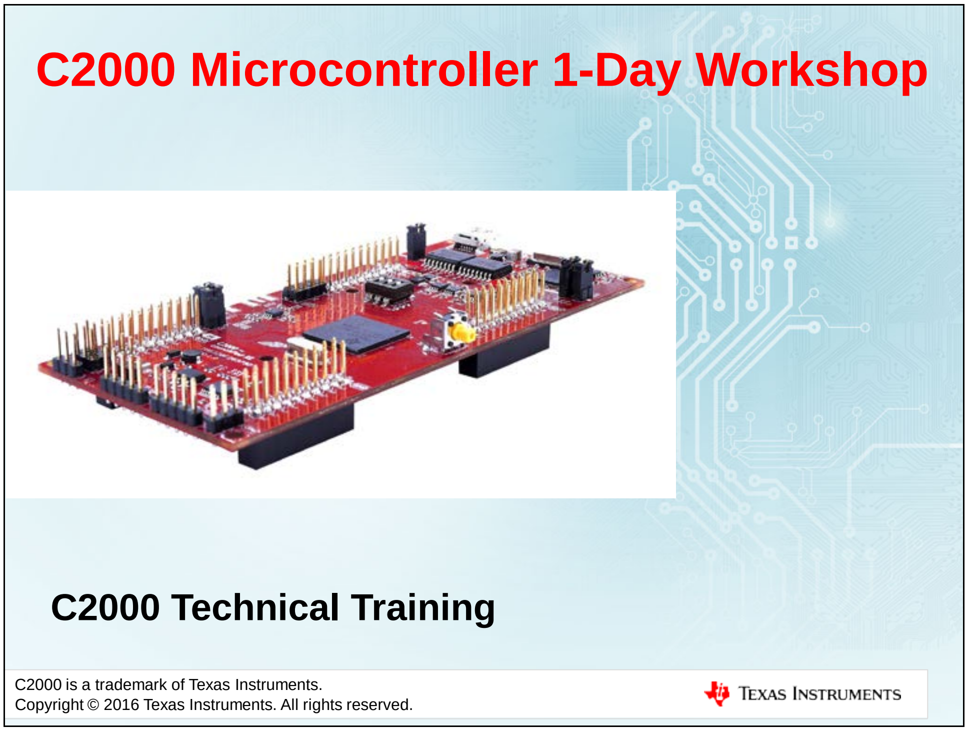
November 2016 – Revision 2.0

Mailing Address

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

Workshop Topics

Workshop Introduction

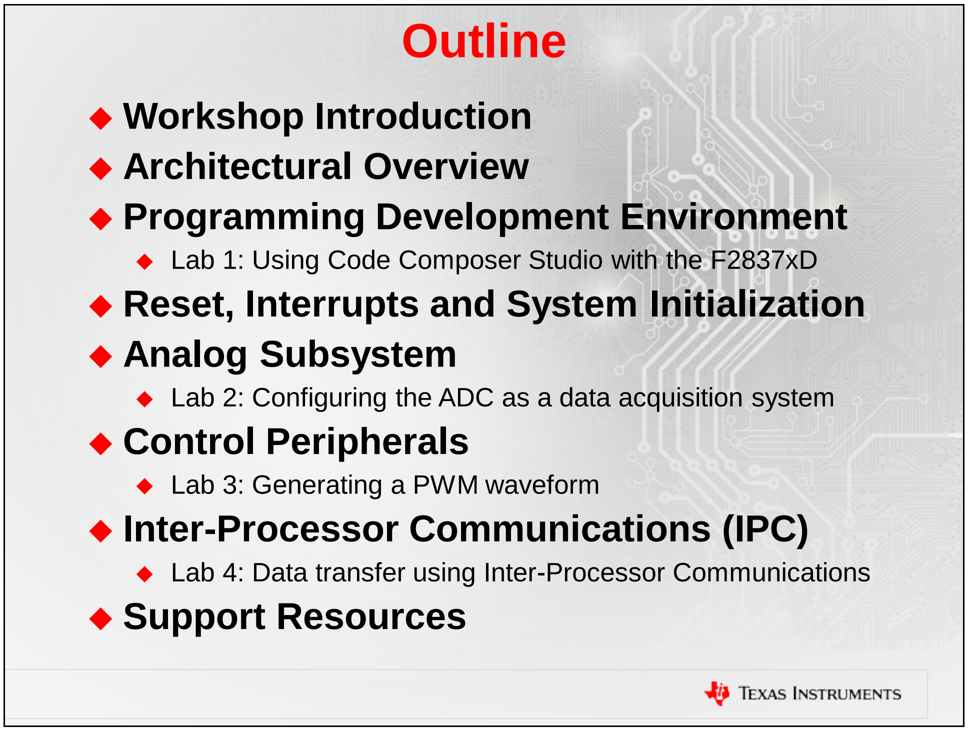


C2000 Microcontroller 1-Day Workshop

C2000 Technical Training

Texas Instruments

Outline



Outline

- Workshop Introduction
- Architectural Overview
- Programming Development Environment
  - Lab 1: Using Code Composer Studio with the F2837xD
- Reset, Interrupts and System Initialization
- Analog Subsystem
  - Lab 2: Configuring the ADC as a data acquisition system
- Control Peripherals
  - Lab 3: Generating a PWM waveform
- Inter-Processor Communications (IPC)
  - Lab 4: Data transfer using Inter-Processor Communications
- Support Resources

Required Workshop Materials

Required Workshop Materials

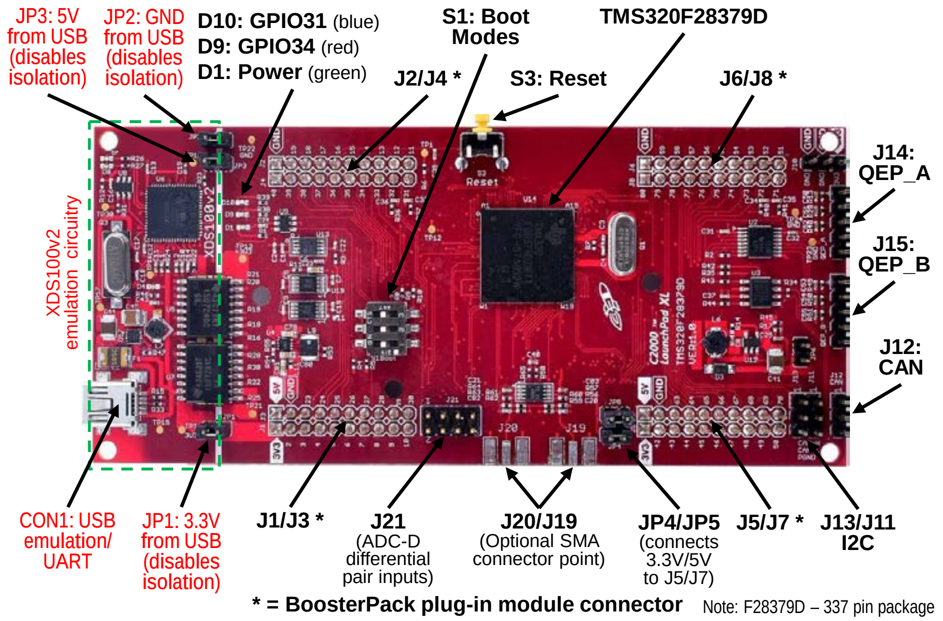
- http://processors.wiki.ti.com/index.php/C2000_One-Day_Workshop
- F28379D LaunchPad (LAUNCHXL-F28379D)
- Install Code Composer Studio v6.2.0
- Run the workshop installer

  C2000 MCU 1-Day Workshop-2.0-Setup.exe

  - Lab Files / Solution Files
  - Student Guide

F28379D LaunchPad

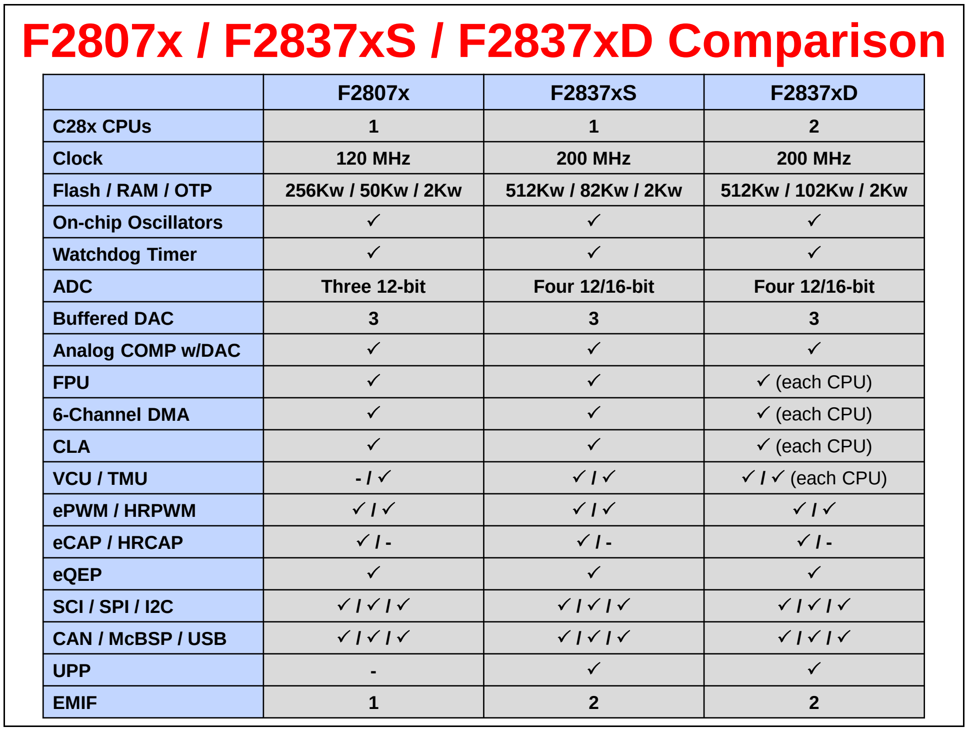
F28379D LaunchPad


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

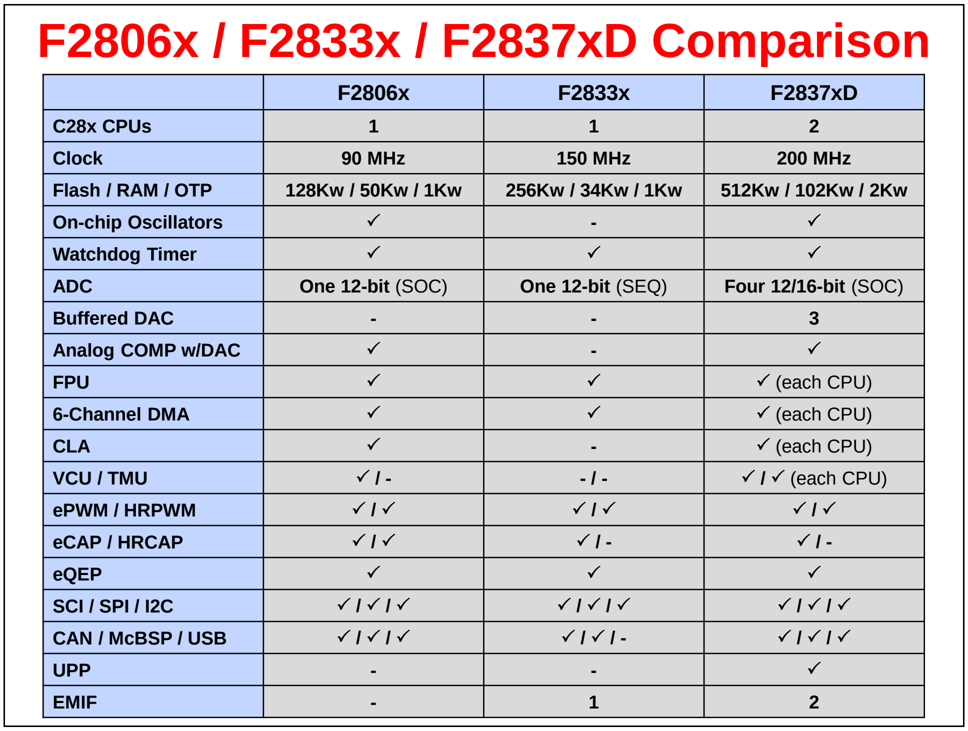
Note: F28379D – 337 pin package

F28x7x Piccolo / Delfino Comparison



F2807x / F2837xS / F2837xD 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	✓	✓	✓
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



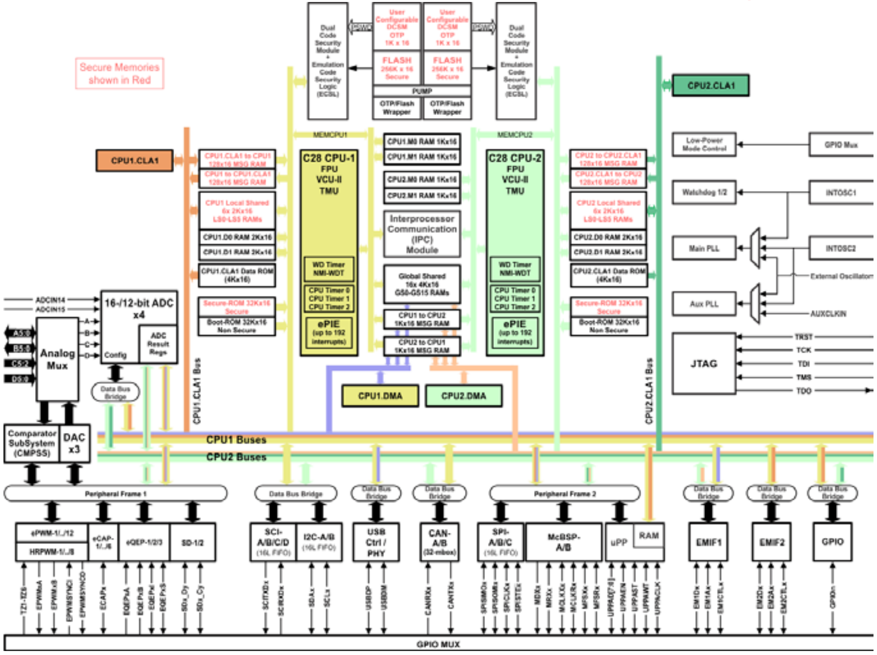
F2806x / F2833x / F2837xD Comparison

	F2806x	F2833x	F2837xD
C28x CPUs	1	1	2
Clock	90 MHz	150 MHz	200 MHz
Flash / RAM / OTP	128Kw / 50Kw / 1Kw	256Kw / 34Kw / 1Kw	512Kw / 102Kw / 2Kw
On-chip Oscillators	✓	-	✓
Watchdog Timer	✓	✓	✓
ADC	One 12-bit (SOC)	One 12-bit (SEQ)	Four 12/16-bit (SOC)
Buffered DAC	-	-	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

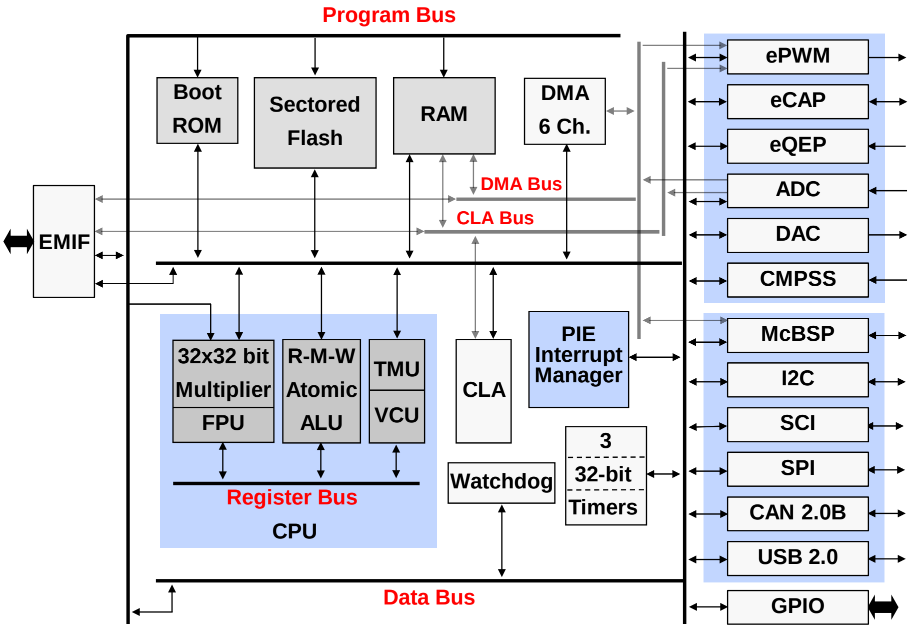
Architectural Overview

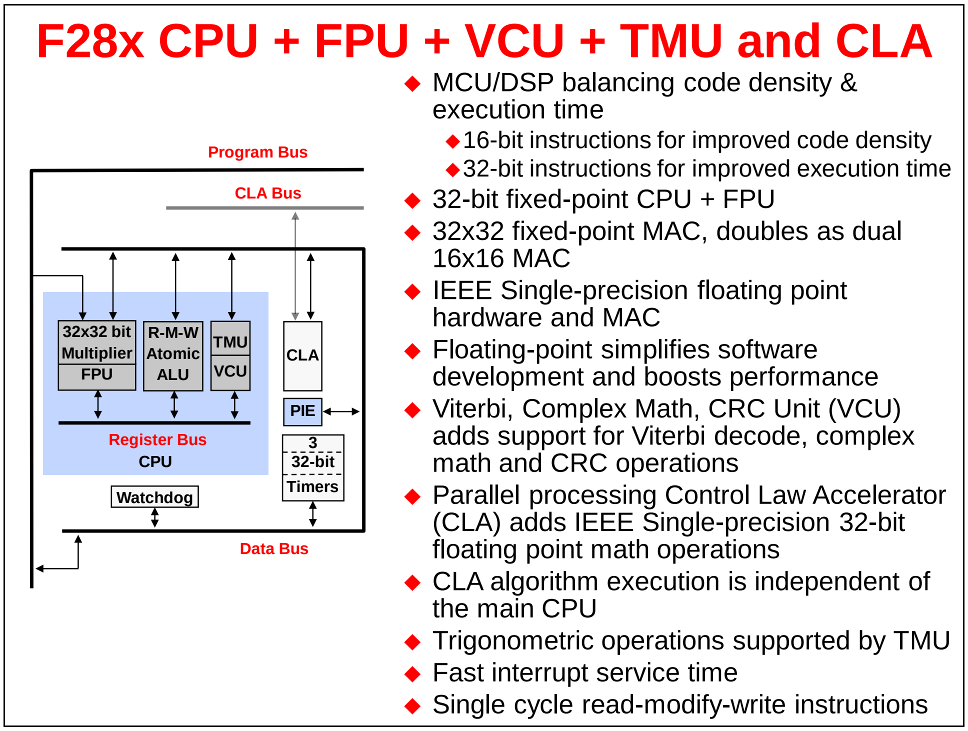
F2837xD Block Diagram

F2837xD – Dual Core Block Diagram



TMS320F28x7x Core Block Diagram

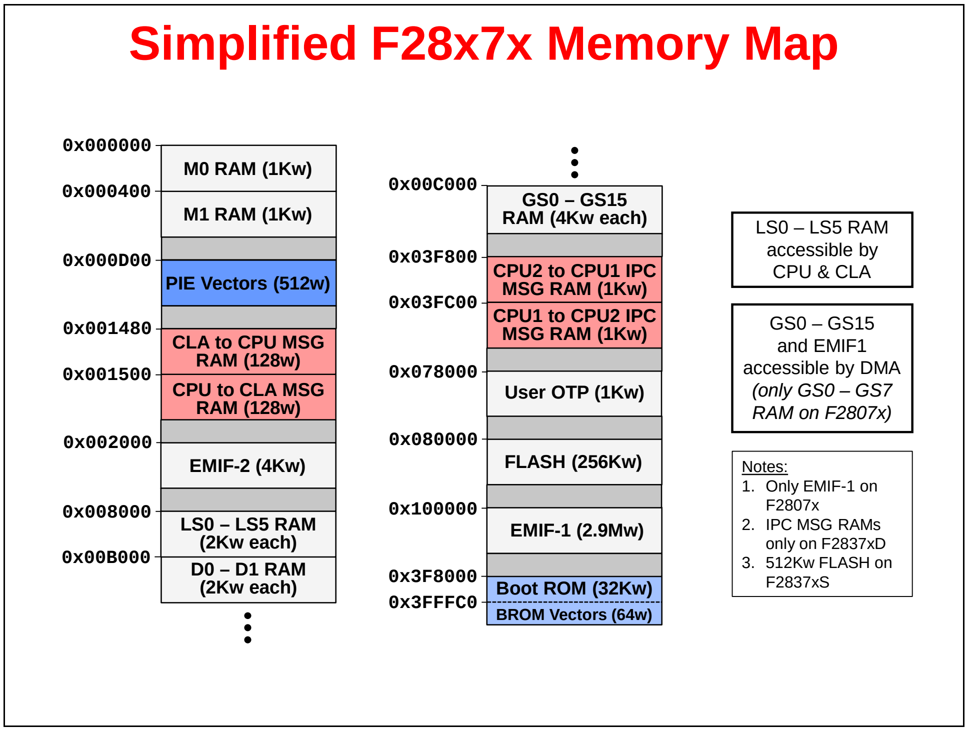




F28x CPU + FPU + VCU + TMU and CLA

- 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

Simplified F28x7x Memory Map



Simplified F28x7x Memory Map

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

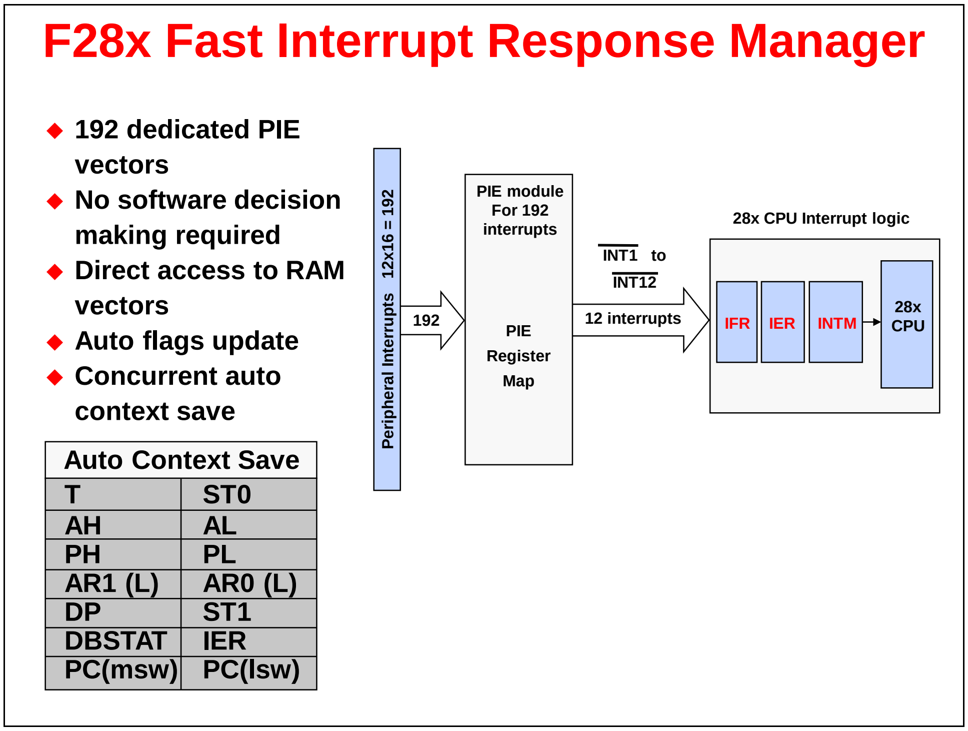
0x00C000	GS0 – GS15 RAM (4Kw each)
0x03F800	CPU2 to CPU1 IPC MSG RAM (1Kw)
0x03FC00	CPU1 to CPU2 IPC MSG RAM (1Kw)
0x078000	User OTP (1Kw)
0x080000	FLASH (256Kw)
0x100000	EMIF-1 (2.9Mw)
0x3F8000	Boot ROM (32Kw)
0x3FFFC0	BROM Vectors (64w)

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. 512Kw FLASH on F2837xS

Interrupt Response Manager

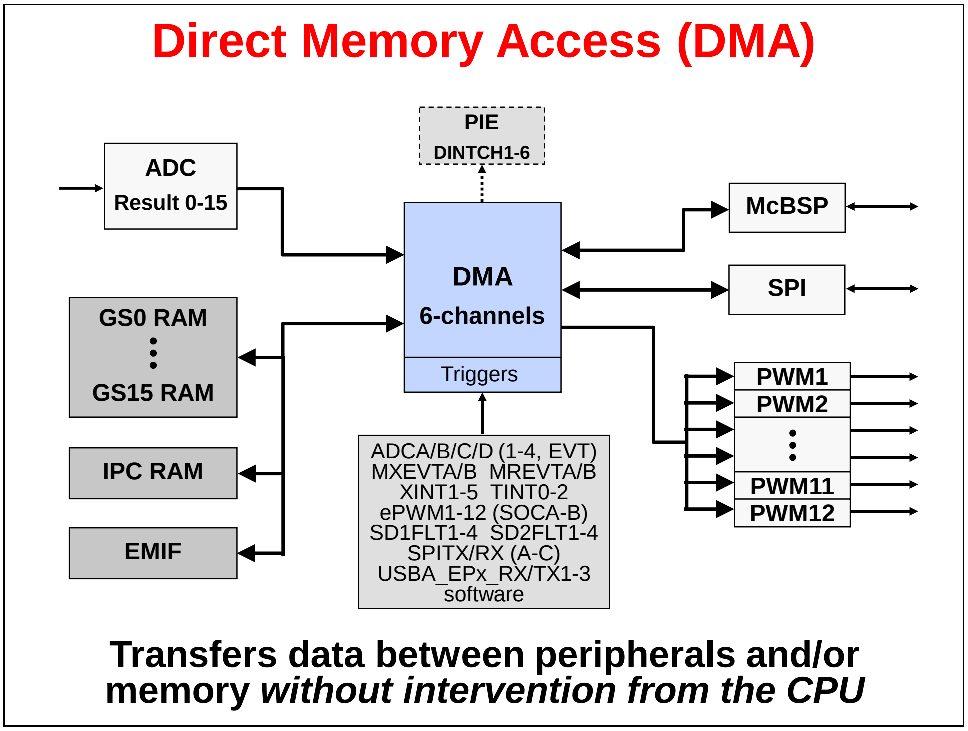


F28x Fast Interrupt Response Manager

- 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)

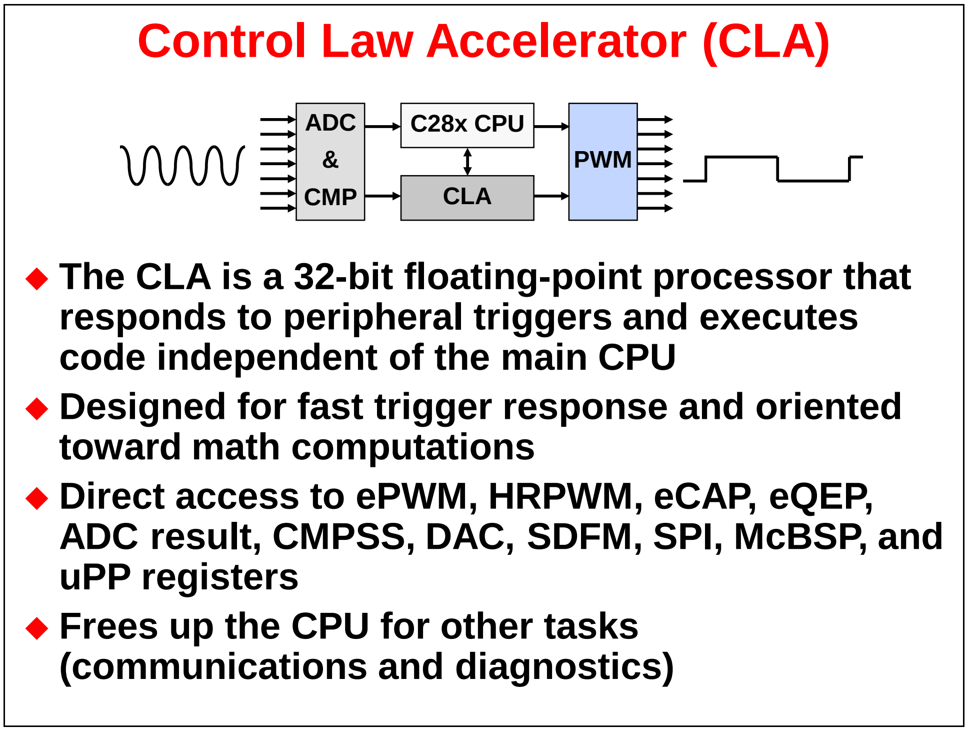
Direct Memory Access (DMA)



Direct Memory Access (DMA)

Transfers data between peripherals and/or memory *without intervention from the CPU*

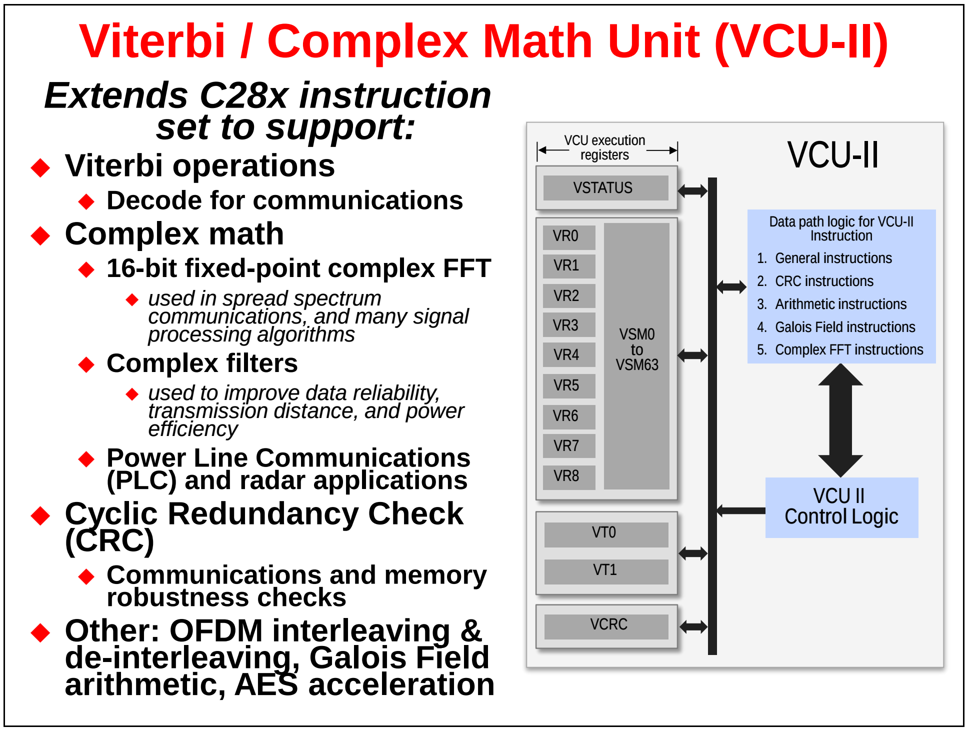
Control Law Accelerator (CLA)



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, and uPP registers
- Frees up the CPU for other tasks (communications and diagnostics)

Viterbi / Complex Math Unit (VCU)

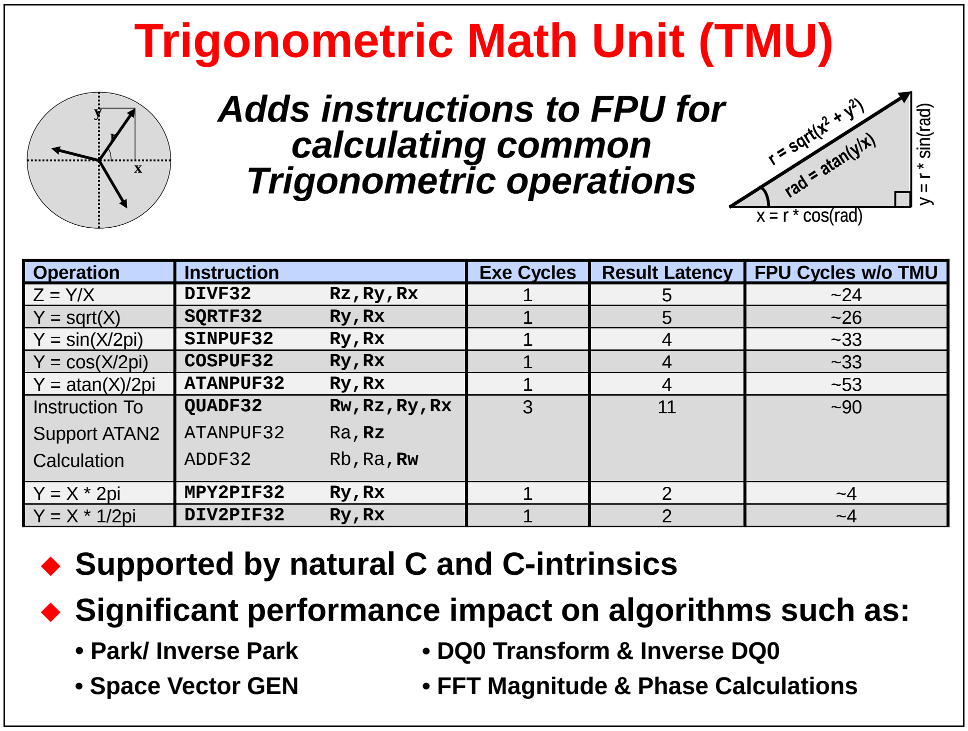


Viterbi / Complex Math Unit (VCU-II)

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

Trigonometric Math Unit (TMU)



Trigonometric Math Unit (TMU)

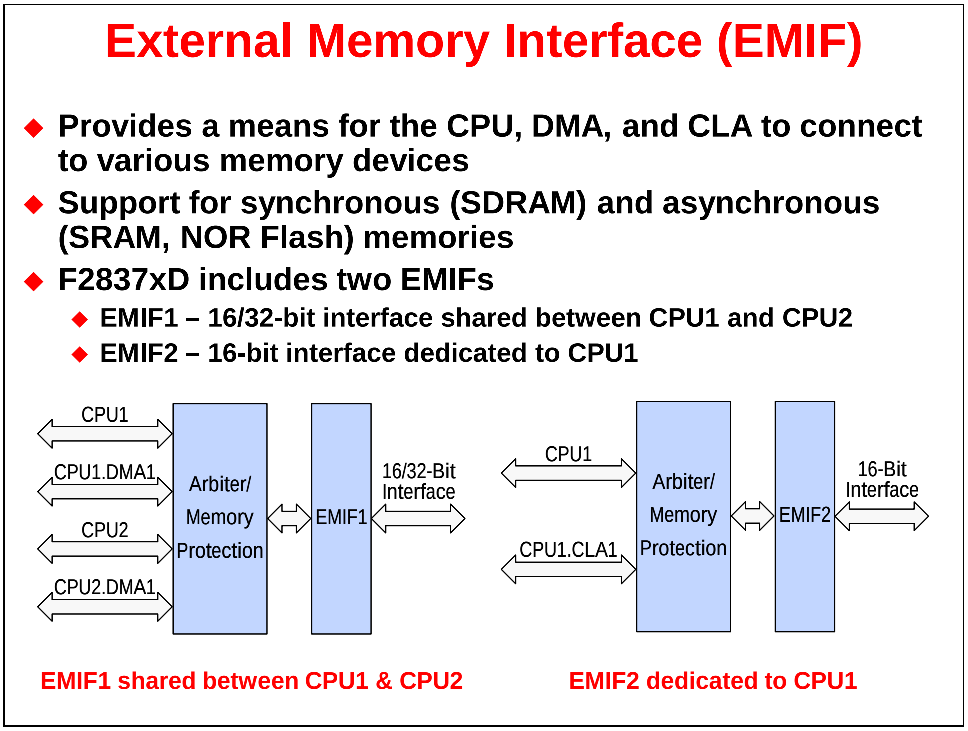
Adds instructions to FPU for calculating common Trigonometric operations

y
r
x
r = sqrt(x² + y²)
rad = atan(y/x)
y = r * sin(rad)
x = r * cos(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 Rw,Rz,Ry,Rx** ATANPUF32 Ra,**Rz** ADDF32 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**

External Memory Interface (EMIF)



External Memory Interface (EMIF)

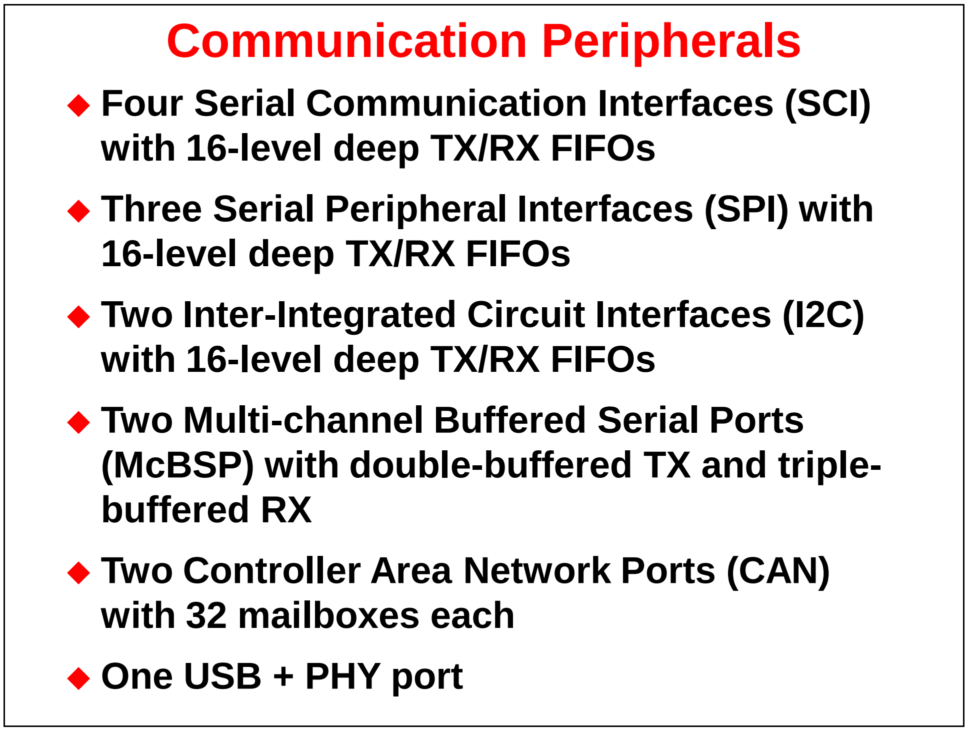
- **Provides a means for the CPU, DMA, and CLA to connect to various memory devices**
- **Support for synchronous (SDRAM) and asynchronous (SRAM, NOR Flash) memories**
- **F2837xD includes two EMIFs**
  - **EMIF1 – 16/32-bit interface shared between CPU1 and CPU2**
  - **EMIF2 – 16-bit interface dedicated to CPU1**

CPU1
CPU1.DMA1
CPU2
CPU2.DMA1
Arbiter/ Memory Protection
EMIF1
16/32-Bit Interface
CPU1
CPU1.CLA1
Arbiter/ Memory Protection
EMIF2
16-Bit Interface

EMIF1 shared between CPU1 & CPU2

EMIF2 dedicated to CPU1

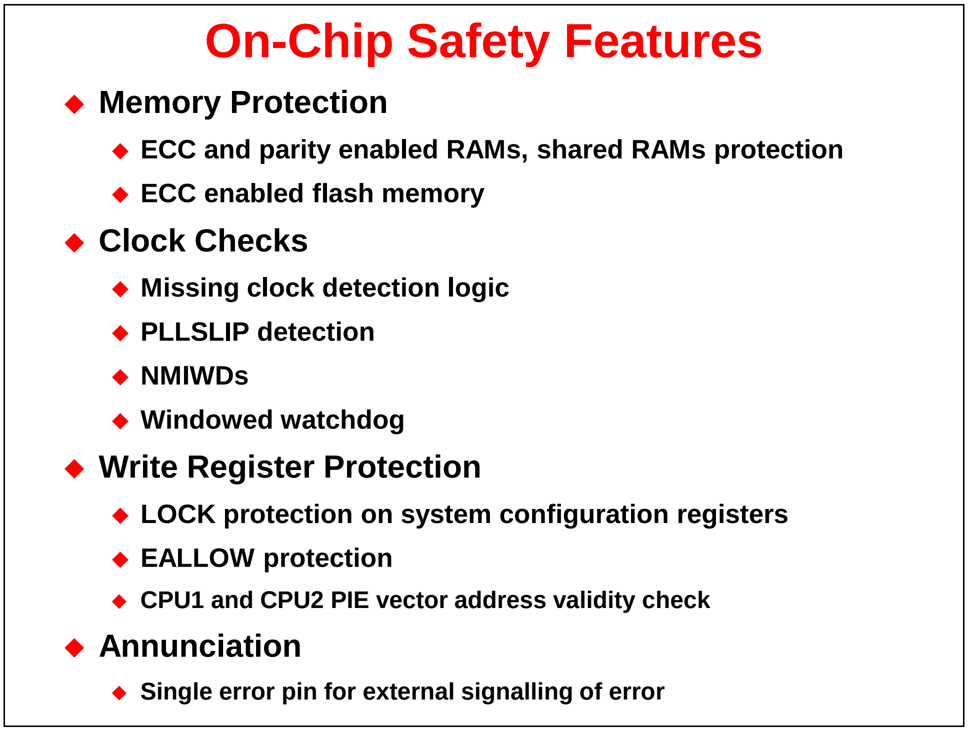
Communication Peripherals



Communication Peripherals

- Four Serial Communication Interfaces (SCI) with 16-level deep TX/RX FIFOs
- Three Serial Peripheral Interfaces (SPI) with 16-level deep TX/RX FIFOs
- Two Inter-Integrated Circuit Interfaces (I2C) with 16-level deep TX/RX FIFOs
- Two Multi-channel Buffered Serial Ports (McBSP) with double-buffered TX and triple-buffered RX
- Two Controller Area Network Ports (CAN) with 32 mailboxes each
- One USB + PHY port

On-Chip Safety Features



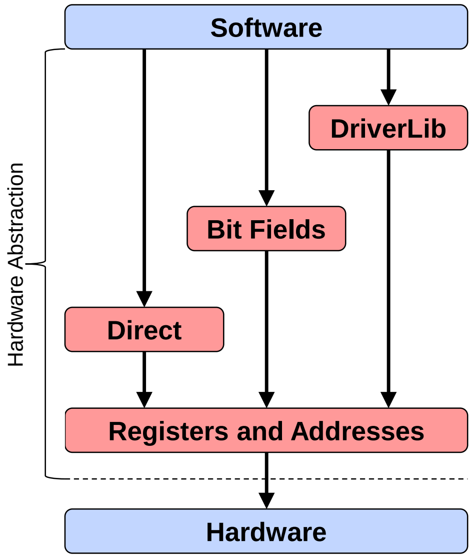
On-Chip Safety Features

- 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

Programming Development Environment

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

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

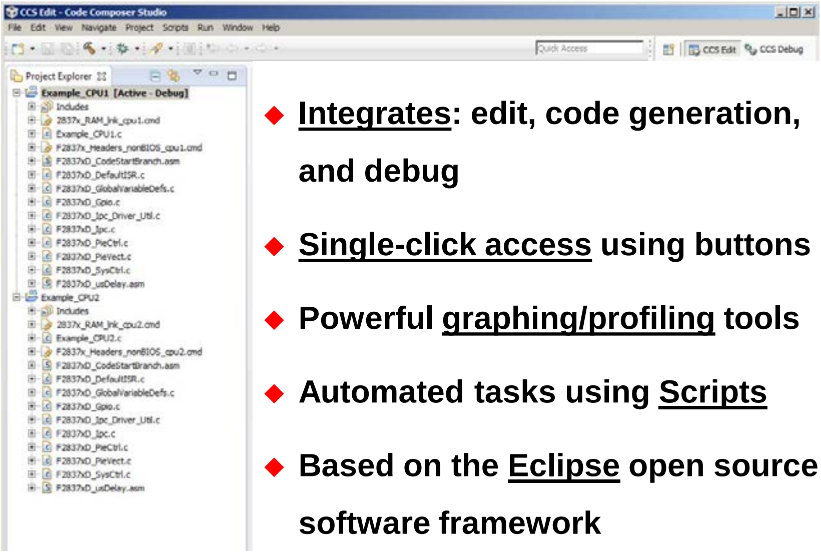
```
EPWM_setCounterCompareValue(EPWM2_BASE, EPWM_COUNTER_COMPARE_A, duty);
```

- 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\controlSUITE\device_support\`
- controlSUITE can be downloaded at `www.ti.com\controlSUITE`

Code Composer Studio

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

Code Composer Studio: IDE



```
Example_CPU1 [Active - Debug]
    Includes
    2837x_RAM_lnk_cpu1.cmd
    Example_CPU1.c
    F2837x_Headers_nonBIOS_cpu1.cmd
    F2837xD_CodeStartBranch.asm
    F2837xD_DefaultISR.c
    F2837xD_GlobalVariableDefs.c
    F2837xD_Gpio.c
    F2837xD_Ipc_Driver_Util.c
    F2837xD_Ipc.c
    F2837xD_PieCtrl.c
    F2837xD_PieVect.c
    F2837xD_SysCtrl.c
    F2837xD_usDelay.asm
Example_CPU2
    Includes
    2837x_RAM_lnk_cpu2.cmd
    Example_CPU2.c
    F2837x_Headers_nonBIOS_cpu2.cmd
    F2837xD_CodeStartBranch.asm
    F2837xD_DefaultISR.c
    F2837xD_GlobalVariableDefs.c
    F2837xD_Gpio.c
    F2837xD_Ipc_Driver_Util.c
    F2837xD_Ipc.c
    F2837xD_PieCtrl.c
    F2837xD_PieVect.c
    F2837xD_SysCtrl.c
    F2837xD_usDelay.asm
```

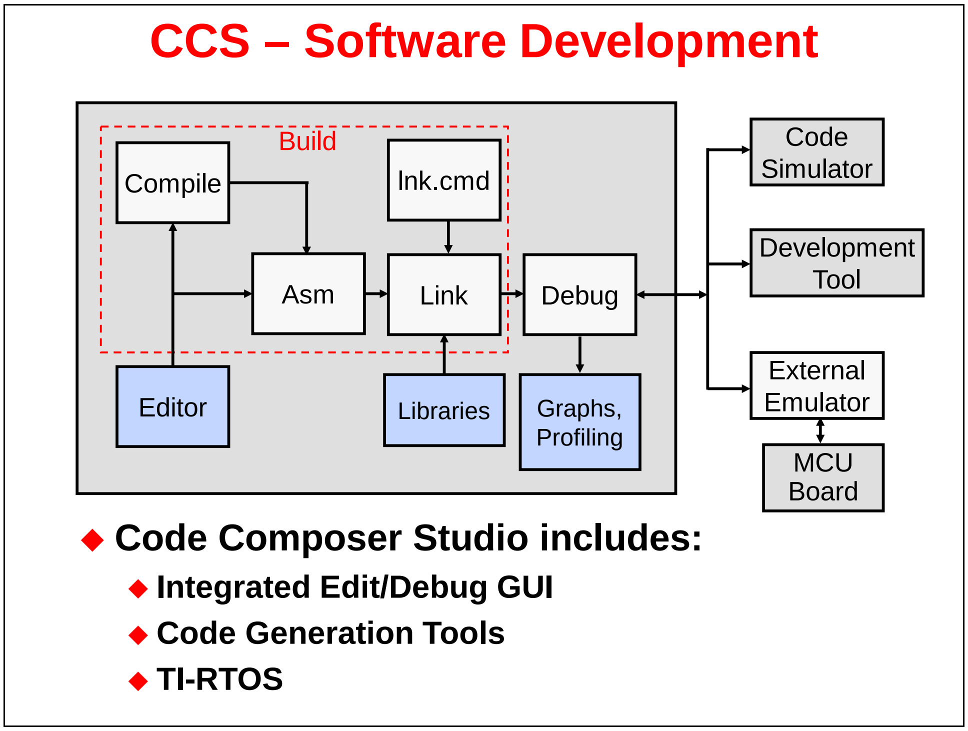
- **Integrates: edit, code generation, and debug**
- **Single-click access using buttons**
- **Powerful graphing/profiling tools**
- **Automated tasks using Scripts**
- **Based on the Eclipse open source software framework**

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.

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 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*.



CCS – Software Development

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

CCS 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.

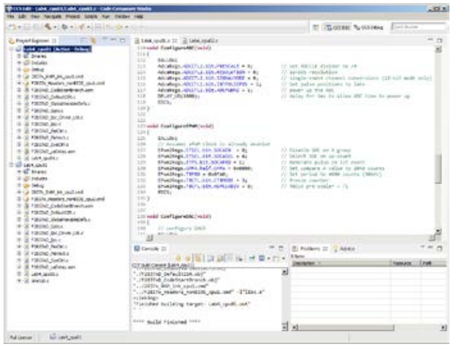
Code developed independently of hardware concerns increases the benefits of modularity by allowing the programmer to focus on the code and not waste time managing memory and moving code as other code components grow or shrink. A linker is invoked to allocate systems hardware to the modules desired to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

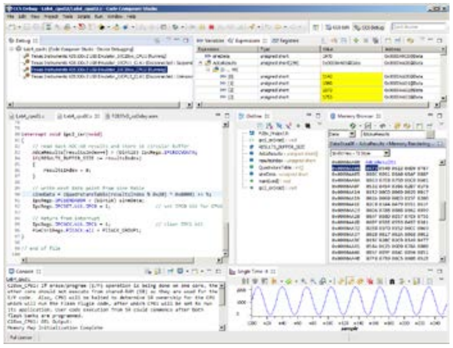
Edit and Debug Perspective

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

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



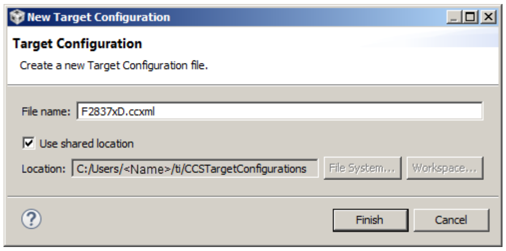


- Edit Perspective
  - Displays views used during code development
    - C/C++ project, editor, etc.
- Debug Perspective
  - Displays views used for debugging
    - Menus and toolbars associated with debugging, watch and memory windows, graphs, etc.

Target Configuration

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

Creating a Target Configuration


New Target Configuration

Target Configuration

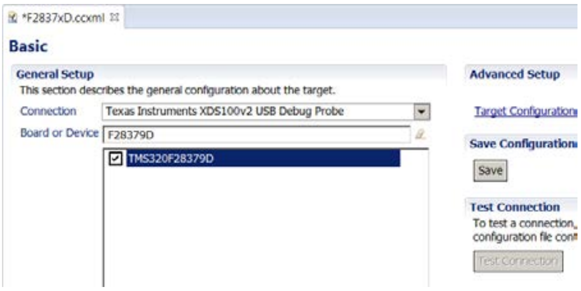
Create a new Target Configuration file.

File name: F2837xD.ccxml

☑ Use shared location

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

- File → New → Target Configuration File


*F2837xD.ccxml

Basic

General Setup

This section describes the general configuration about the target.

Connection: Texas Instruments XDS100v2 USB Debug Probe

Board or Device: F28379D

☑ TMS320F28379D

Advanced Setup

Target Configuration

Save Configuration

Test Connection

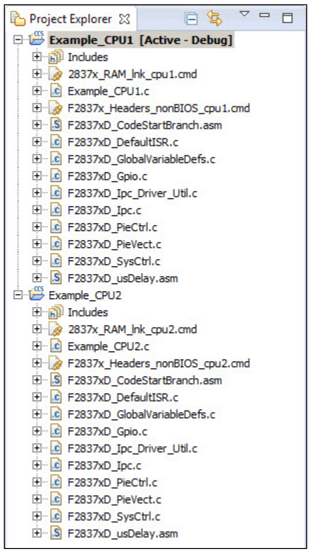
To test a connection, configuration file con

- Select connection type
- Select device
- Save configuration

CCS Project and Build Options

CCS works with a *project* paradigm. Essentially, within CCS you create a project for each executable program you wish to create. Projects store all the information required to build the executable. For example, it lists things like: the source files, the header files, the target system's memory-map, and program build options.

CCSv6 Project



Project Explorer

```
Example_CPU1  [Active - Debug]
    Includes
    2837x_RAM_lnk_cpu1.cmd
    Example_CPU1.c
    F2837x_Headers_nonBIOS_cpu1.cmd
    F2837xD_CodeStartBranch.asm
    F2837xD_DefaultISR.c
    F2837xD_GlobalVariableDefs.c
    F2837xD_Gpio.c
    F2837xD_Ipc_Driver_Util.c
    F2837xD_Ipc.c
    F2837xD_PieCtrl.c
    F2837xD_PieVect.c
    F2837xD_SysCtrl.c
    F2837xD_usDelay.asm
Example_CPU2
    Includes
    2837x_RAM_lnk_cpu2.cmd
    Example_CPU2.c
    F2837x_Headers_nonBIOS_cpu2.cmd
    F2837xD_CodeStartBranch.asm
    F2837xD_DefaultISR.c
    F2837xD_GlobalVariableDefs.c
    F2837xD_Gpio.c
    F2837xD_Ipc_Driver_Util.c
    F2837xD_Ipc.c
    F2837xD_PieCtrl.c
    F2837xD_PieVect.c
    F2837xD_SysCtrl.c
    F2837xD_usDelay.asm
```

Project files contain:

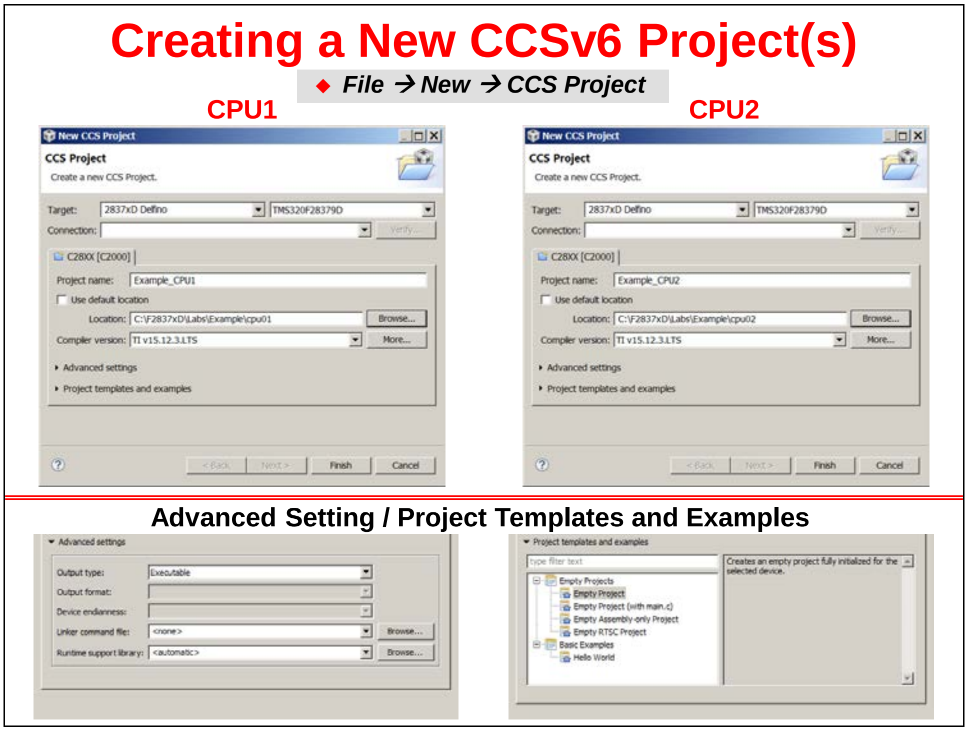
- **List of files:**
  - **Source (C, assembly)**
  - **Libraries**
  - **SYS/BIOS configuration file**
  - **Linker command files**
- **Project settings:**
  - **Build options (compiler, assembler, linker, and TI-RTOS)**
  - **Build configurations**

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.

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 CCSv6 Project(s)

- File → New → CCS Project

CPU1

New CCS Project

CCS Project

Create a new CCS Project.

Target: 2837xD Delfino | TMS320F28379D

Connection:

C28XX [C2000]

Project name: Example_CPU1

□ Use default location

Location: C:\F2837xD\Labs\Example\cpu01

Compiler version: TI v15.12.3.LTS

CPU2

New CCS Project

CCS Project

Create a new CCS Project.

Target: 2837xD Delfino | TMS320F28379D

Connection:

C28XX [C2000]

Project name: Example_CPU2

□ Use default location

Location: C:\F2837xD\Labs\Example\cpu02

Compiler version: TI v15.12.3.LTS

Advanced Setting / Project Templates and Examples

Advanced settings

Output type: Executable

Output format:

Device endianness:

Linker command file: <none>

Runtime support library: <automatic>

Project templates and examples

type filter text

- Empty Projects
  - Empty Project
  - Empty Project (with main.c)
  - Empty Assembly-only Project
  - Empty RTSC Project
- Basic Examples
  - Hello World

Creates an empty project fully initialized for the selected device.

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. The following slide is a sample of the configuration options.

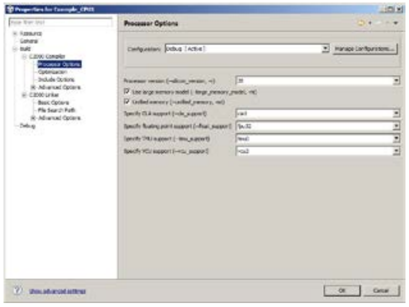
There is a one-to-one relationship between the items in the text box on the main page and the GUI check and drop-down box selections. Once you have mastered the various options, you can probably find yourself just typing in the options.

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

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

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

CCSv6 Build Options – Compiler / Linker



Configuration: Debug [Active]

Processor version (--silicon_version, -v) 28

☑ Use large memory model (--large_memory_model, -ml)

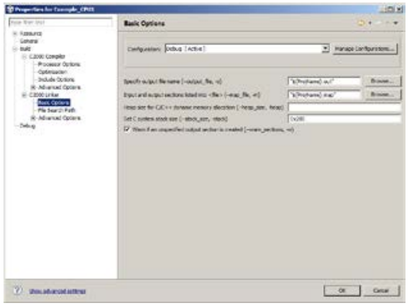
☑ Unified memory (--unified_memory, -mt)

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"

Input and output sections listed into <file> (--map_file, -m) "${ProjName}.map"

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

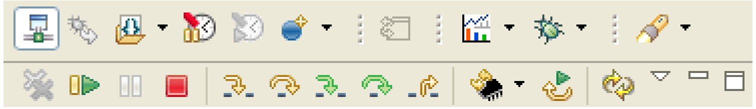
Set C system stack size (--stack_size, -stack) 0x200

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

- **Separate build options for each project – CPU1 & CPU2**
- **Compiler**
  - **Categories for code generation tools – controls many aspects of the build process, such as:**
    - **Optimization level**
    - **Target device**
    - **Compiler / assembly / link options**
- **Linker**
  - **Categories for linking – specify various link options**
  - ***${PROJECT_ROOT}* specifies the current project directory**

CCSv6 Debug Environment

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



The common debugging and program execution descriptions are shown below:

Start debugging

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

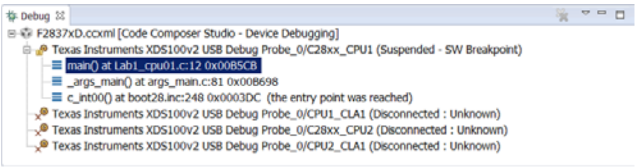
Program execution

Image	Name	Description	Availability
⏸	Halt	Halts the selected target. The rest of the debug views will update automatically with most recent target data.	Target Menu Debug View Toolbar
▶	Run	Resumes the execution of the currently loaded program from the current PC location. Execution continues until a breakpoint is encountered.	Target Menu Debug View Toolbar
⇨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

Dual Subsystem Debug

Launching Dual Subsystem Debug (1)

- 1st subsystem (CCS Edit Perspective) -
  - Clicking “Debug” button ⚙ will automatically:
    - Launch the debugger
    - Connects to target
    - Programs flash memory

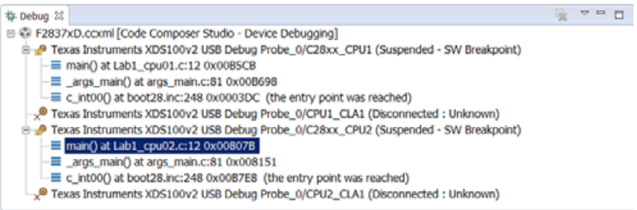


```
F2837xD.ccxml [Code Composer Studio - Device Debugging]
  Texas Instruments XDS100v2 USB Debug Probe_0/C28xx_CPU1 (Suspended - SW Breakpoint)
    main() at Lab1_cpu01.c:12 0x00B5CB
    _args_main() at args_main.c:81 0x00B698
    c_int00() at boot28.inc:248 0x0003DC  (the entry point was reached)
  Texas Instruments XDS100v2 USB Debug Probe_0/CPU1_CLA1 (Disconnected : Unknown)
  Texas Instruments XDS100v2 USB Debug Probe_0/C28xx_CPU2 (Disconnected : Unknown)
  Texas Instruments XDS100v2 USB Debug Probe_0/CPU2_CLA1 (Disconnected : Unknown)
```

- Note 2nd subsystem is disconnected
- Next step will connect 2nd subsystem

Launching Dual Subsystem Debug (2)

- 2nd subsystem (CCS Debug Perspective) -
  - In Debug window right-click on emulator and select “Connect target”
  - Highlight emulator and load program (flash)
    - Run → Load → Load Program…

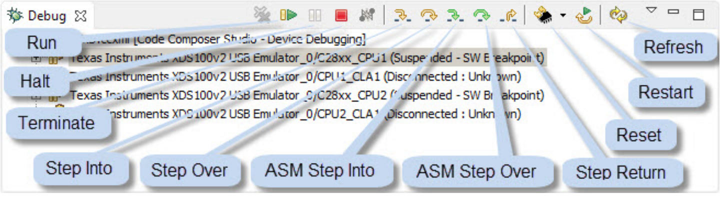


```
F2837xD.ccxml [Code Composer Studio - Device Debugging]
  Texas Instruments XDS100v2 USB Debug Probe_0/C28xx_CPU1 (Suspended - SW Breakpoint)
    main() at Lab1_cpu01.c:12 0x00B5CB
    _args_main() at args_main.c:81 0x00B698
    c_int00() at boot28.inc:248 0x0003DC  (the entry point was reached)
  Texas Instruments XDS100v2 USB Debug Probe_0/CPU1_CLA1 (Disconnected : Unknown)
  Texas Instruments XDS100v2 USB Debug Probe_0/C28xx_CPU2 (Suspended - SW Breakpoint)
    main() at Lab1_cpu02.c:12 0x00807B
    _args_main() at args_main.c:81 0x008151
    c_int00() at boot28.inc:248 0x00B7E8  (the entry point was reached)
  Texas Instruments XDS100v2 USB Debug Probe_0/CPU2_CLA1 (Disconnected : Unknown)
```

- Both subsystems are connected
- Next step is dual subsystem start-up sequence

Dual Subsystem Debug Start-up

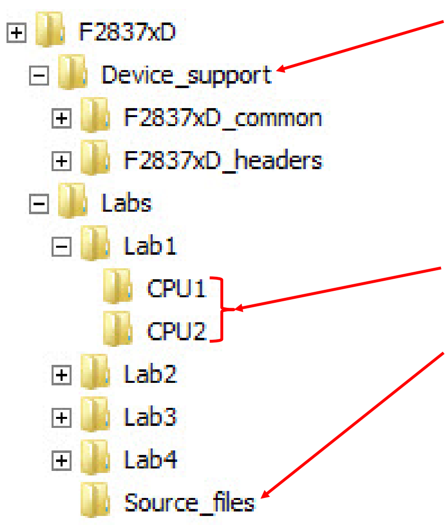
- Start-up sequence
  1. Reset CPU1 subsystem
  2. Reset CPU2 subsystem
  3. Run CPU1 subsystem
  4. Run CPU2 subsystem
  5. Stop and debug either subsystem
- Debug window controls “selected” subsystem for the debug interaction
  - Highlight appropriate subsystem for debug



[Code Composer Studio - Device Debugging]
Texas Instruments XDS100v2 USB Emulator_0/C28xx_CPU1 (Suspended - SW Breakpoint)
Texas Instruments XDS100v2 USB Emulator_0/CPU1_CLA1 (Disconnected : Unknown)
Texas Instruments XDS100v2 USB Emulator_0/C28xx_CPU2 (Suspended - SW Breakpoint)
Texas Instruments XDS100v2 USB Emulator_0/CPU2_CLA1 (Disconnected : Unknown)

Lab File Directory Structure

Lab File Directory Structure



Supporting Files and Libraries

- 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 and libraries

Source Files are “Added” to the Project Folder

Original Source Files

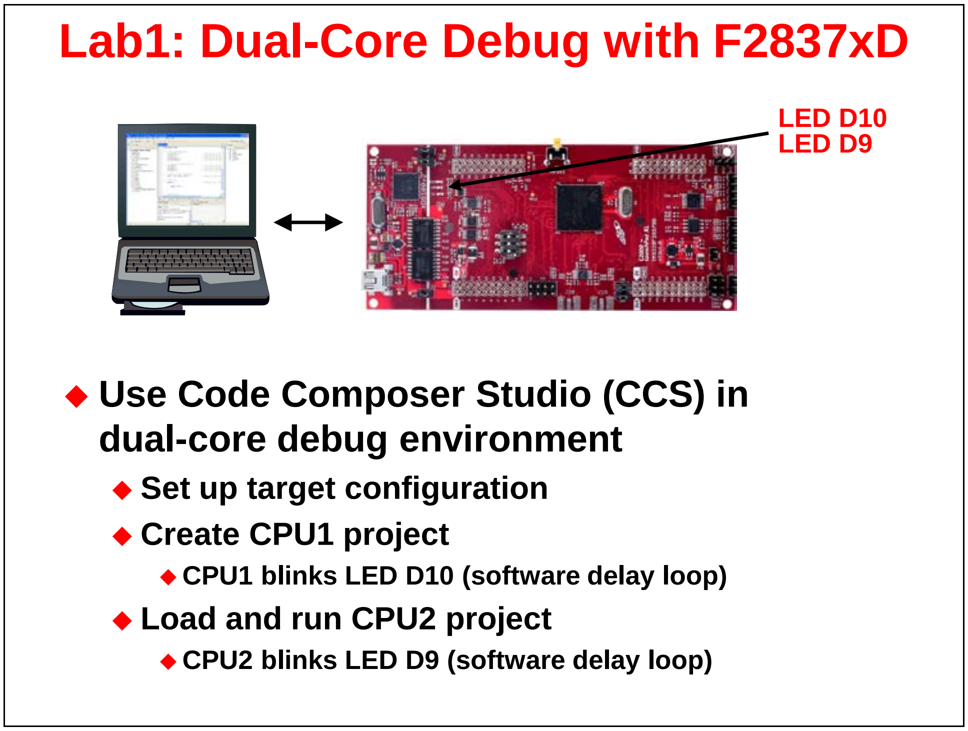
- All modified files are in the Project Folder
- Original source files are always available for reuse, if a file becomes corrupted

Note: CCSv6 will automatically add ALL files contained in the folder where the project is created

Lab 1: Dual-Core Debug with F2837xD

➢ Objective

The objective of this lab exercise is to become familiar with the Code Composer Studio (CCS) development environment while using a dual core F2837xD device. Details on setting up the target configuration, creating a new project, setting build options, and connecting to the dual-core device will be explained. A typical F2837xD application consists of two separate and completely independent CCS projects. One project is for CPU1, and the other project is for CPU2. A project contains all the files needed to develop an executable output file (.out) which can be run on the F2837xD device. In this lab exercise we will have CPU1 blink LED D10 and the CPU2 blink LED D9.



Lab1: Dual-Core Debug with F2837xD

LED D10
LED D9

- **Use Code Composer Studio (CCS) in dual-core debug environment**
  - **Set up target configuration**
  - **Create CPU1 project**
    - **CPU1 blinks LED D10 (software delay loop)**
  - **Load and run CPU2 project**
    - **CPU2 blinks LED D9 (software delay loop)**

➢ 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 *controlSUITE* files is included with the lab files. This provides portability, making the workshop files self-contained and independent of other support files or resources. The lab directions for this workshop are based on all software installed in their default locations.

➢ Procedure

Start Code Composer Studio and Open a Workspace

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

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

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

   The workbench will open in the "CCS Edit" perspective view. Notice the CCS Edit icon in the upper right-hand corner. A perspective defines the initial layout views of the workbench windows, toolbars, and menus which are appropriate for a specific type of task (i.e. code development or debugging). This minimizes clutter to the user interface. The "CCS Edit" perspective is used to create or build 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.

Set Up Target Configuration

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

   `File → New → Target Configuration File`

   In the file name field type **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 **F28379D** to filter the options. In the box below, check the box to select "F28379D". Click `Save` to save the configuration, then close the "F2837xD.ccxml" set up 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 – CPU1

6. A *project* contains all the files needed to develop an executable output file (.out) which will run on the MCU hardware. To create a new project for CPU1 click:

   `File → New → CCS Project`

   A CCS Project window will open. At the top of this window, filter the "Target" options by using the pull-down list on the left and choose "2837xD Delfino". In the pull-down list immediately to the right, choose the "TMS320F28379D" device.

   Leave the "Connection" box blank since we already set up the target configuration.

7. The next section selects the project settings. In the Project name field type **Lab1_cpu01**. *Uncheck* the "Use default location" box. Click the Browse… button and navigate to:

   `C:\F2837xD\Labs\Lab1\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.

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

   A new project has now been created. Notice the "Project Explorer" window contains Lab1_cpu01. 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.

Add Files to Project – CPU1

Note: The local copy of the supporting files and libraries in this workshop are identical to the required controlSUITE files. The workshop lab exercises will make use of these files as often as possible. When adding files to the project, a window will appear asking to "copy" or "link" the files. Selecting "Copy files" will make a copy of the original file to work with in the local project directory. Selecting "Link files" will set a reference to the original file and will use the original file. Typically, "link files" is used when the files will not be modified. To avoid accidently modifying the original files, we will use "copy files" throughout this workshop and work with the local copy in the project directory.

For convenience, all of the needed source files for this lab exercise are located in the same folder.

10. To add the source files to the project, right-click on Lab1_cpu01 in the "Project Explorer" window and select:

    `Add Files…`

    or click: `Project → Add Files…`

    Navigate to `C:\F2837xD\Labs\Source_files`. Select all of the files in this folder and click `Open`. Next, add ("copy files") the files to the project by clicking `OK`. The files used in this project are:

```
2837xD_RAM_lnk_cpu1.cmd
F2837xD_CodeStartBranch.asm
F2837xD_DefaultISR.c
F2837xD_GlobalVariableDefs.c
F2837xD_Gpio.c
F2837xD_Headers_nonBIOS_cpu1.cmd
F2837xD_PieCtrl.c
F2837xD_PieVect.c
F2837xD_SysCtrl.c
F2837xD_usDelay.asm
Lab1_cpu01.c
```

In the Project Explorer window, click the plus sign (+) to the left of Lab1_cpu01 and notice that the files are listed.

Project Build Options – CPU1

11. Configure the build options by right-clicking on Lab1_cpu01 in the "Project Explorer" window and select "Properties". We need to set up the include search path to include the peripheral register header files. Under "C2000 Compiler" select "Include Options". In the search path box ("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}/../../../Device_support/F2837xD_headers/include

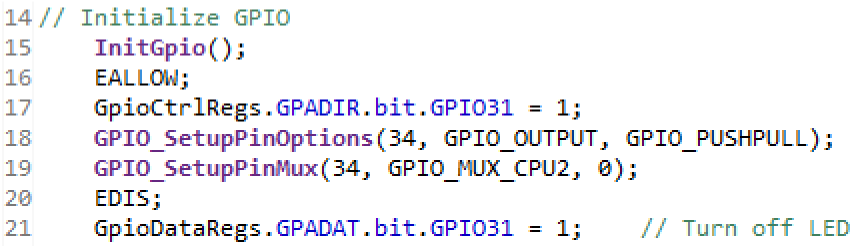
    ${PROJECT_ROOT}/../../../Device_support/F2837xD_common/include

    Click OK to include each search path.

12. Next, we need to configure the predefined symbols. Under "C2000 Compiler" select "Advanced Options" and then "Predefined Symbols". In the predefined name box ("Pre-define NAME") click the Add icon (first icon with green plus sign). Then in the "Enter Value" window type (one at a time): **CPU1** and **_LAUNCHXL_F28379D** (note leading underscore). Click OK to include each name. These names are used in the project to conditionally include the peripheral register header files code specific to CPU1 and the LaunchPad. Finally, click OK to save and close the Properties window.

Inspect the Project – CPU1

13. Open and inspect Lab1_cpu01.c by double clicking on the filename in the Project Explorer window. The code in this lab exercise will be running from internal RAM. In function main(), the code lines shown below are used to configure the GPIO pins. On the LaunchPad, GPIO31 and GPIO34 are used to blink LEDs D10 and D9, respectively.

    ```
    // Initialize GPIO
       InitGpio();
       EALLOW;
       GpioCtrlRegs.GPADIR.bit.GPIO31 = 1;
       GPIO_SetupPinOptions(34, GPIO_OUTPUT, GPIO_PUSHPULL);
       GPIO_SetupPinMux(34, GPIO_MUX_CPU2, 0);
       EDIS;
       GpioDataRegs.GPADAT.bit.GPIO31 = 1;    // Turn off LED
    ```

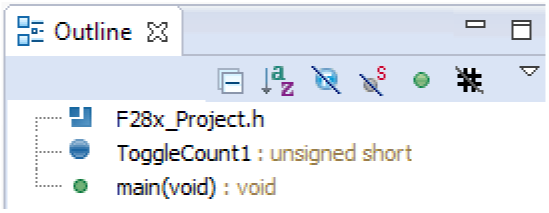
    Since CPU1 has control over all the IO pins, GPIO31 can be manipulated directly by CPU1. However, for this lab exercise, we would like to have CPU2 control GPIO34 so it can blink D9. This will be accomplished using the IPC (Inter-Processor Communications) module on the device. The function calls are used here to set up the GPIO pin so it is ready for CPU2 to use.

14. At the bottom of function main() is an infinite "for" loop. The instructions inside the loop blink LED D10 on the LaunchPad at a rate determined by the DELAY_US() macro. The LED status is changed by the code lines which write to the GPIO31 pin.

15. CCS contains an outline viewer which displays the components of each source file. Open the outline viewer by clicking:

    `View → Outline`

    Notice that the outline window contents change as each source file is viewed in the editor. For the source file "Lab1_cpu01" the outline window contains:



    The list is short since this is a very simple project, but for more complex source files the "Outline" view provides a useful way of finding symbols and function calls within the file.

Open a New Project – CPU2

16. A project named `Lab1_cpu02` 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:\F2837xD\Labs\Lab1\cpu02` and click `OK`. Then click `Finish` to import the project. All build options have been configured the same as the previous project (CPU1). The files used in this project are:

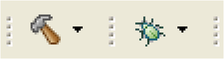
```
2837xD_RAM_lnk_cpu2.cmd                F2837xD_PieCtrl.c
F2837xD_CodeStartBranch.asm            F2837xD_PieVect.c
F2837xD_DefaultISR.c                   F2837xD_SysCtrl.c
F2837xD_GlobalVariableDefs.c           F2837xD_usDelay.asm
F2837xD_Gpio.c                         Lab1_cpu02.c
F2837xD_Headers_nonBIOS_cpu2.cmd
```

Inspect the Project – CPU2

17. Open and inspect `Lab1_cpu02.c` by double clicking on the filename in the Project Explorer window. The code for CPU2 is almost identical to that for CPU1. One difference is the timings of the LED status changes at the bottom of main(). Locate these lines. Notice that the code which toggles the I/O pin uses the function `GPIO_WritePin()`. As mentioned, this uses the Inter-Processor Communications (IPC) module to send the data from CPU2 to CPU1, which has control over the GPIO pins.

Build and Load the Projects – CPU1 & CPU2

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



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

Note: 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. In this lab exercise, code will be running from RAM only.

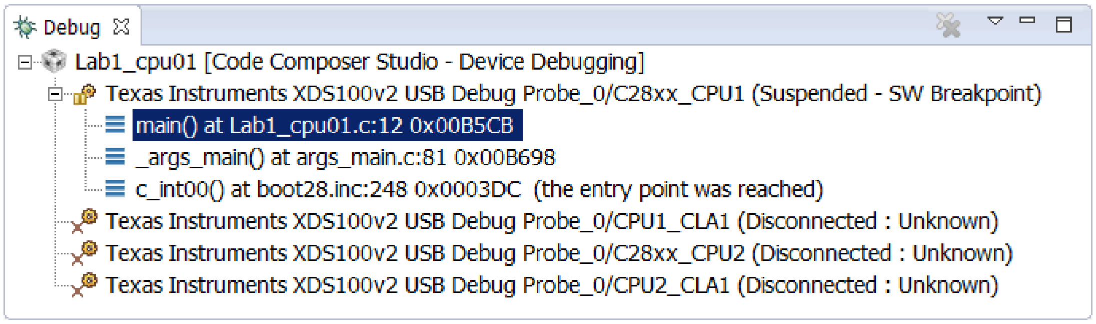
19. In the Project Explorer window click on the “Lab1_cpu01” project to set it active. Then click the “Build” button (hammer) and watch the tools run in the “Console” window. Check for any errors in the “Problems” window. Repeat this step for the “Lab1_cpu02” project.

20. Again, in the Project Explorer window click on the “Lab1_cpu01” project to set it active. CCS in the “CCS Edit” perspective view can automatically save modified source files, build the program, open the “CCS Debug” perspective view, connect and download it to the target (load RAM memory or program flash memory), and then run the program to the beginning main(), in a single step.

    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, and then click `OK`.

    The CCS Debug icon in the upper right-hand corner indicates that we are now in the “CCS Debug” perspective view. The program ran through the C-environment initialization routine in the run-time support library and stopped at “main()” in Lab1_cpu01.c. The blue arrow in the left hand column of the source code window indicates the current position of the CPU1 program counter (PC). The “Debug” window reflects the current status of CPU1 and CPU2.



```
Lab1_cpu01 [Code Composer Studio - Device Debugging]
  Texas Instruments XDS100v2 USB Debug Probe_0/C28xx_CPU1 (Suspended - SW Breakpoint)
    main() at Lab1_cpu01.c:12 0x00B5CB
    _args_main() at args_main.c:81 0x00B698
    c_int00() at boot28.inc:248 0x0003DC  (the entry point was reached)
  Texas Instruments XDS100v2 USB Debug Probe_0/CPU1_CLA1 (Disconnected : Unknown)
  Texas Instruments XDS100v2 USB Debug Probe_0/C28xx_CPU2 (Disconnected : Unknown)
  Texas Instruments XDS100v2 USB Debug Probe_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 Inter Processor Communication (IPC) command yet, CPU2 is stopped by an “Idle” mode instruction in the Boot ROM.

21. Next, we need to connect to and load the program on CPU2. Right-click at the line “Texas Instruments XDS100v2 USB Debug Probe_0/C28xx_CPU2” and select “`Connect Target`”.

22. With the line “Texas Instruments XDS100v2 USB Debug Probe_0/C28xx_CPU2” still highlighted, load the program:

    `Run` → `Load` → `Load Program…`

    Browse to the file: `C:\F2837xD\Labs\Lab1\cpu02\Debug\Lab1_cpu02.out` and select `OK` to load the program.

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. Next, we will examine the use of an “Expressions” window.

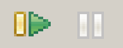
23. To add global variables to the “Expressions” window, click the “Expressions” tab near the top of the CCS window. (Note that the expressions window can be manually opened by clicking:

View → Expressions on the menu bar). In the Expression window an ampersand, which means the "address of", is not used. The Expressions window knows we are specifying a symbol.

24. In main() for each CPU there is a counter which keeps track of the number of times each LED has changed state. We will monitor these variables. In the empty box in the "Expression" column (click on the text *"Add new expression"*), type **ToggleCount1** and then enter.
25. Repeat the above step to add the variable **ToggleCount2** to the Expressions window.

Running the Code – CPU1 & CPU2

Two buttons on the horizontal toolbar are commonly used to control program execution. Hover your mouse over each button as you read the following descriptions:



Button	Name	Description
1	Resume	Run the selected target (F8)
2	Suspend	Halt the selected target (Alt+F8)

26. In the Debug window, click on the line "Texas Instruments XDS100v2 USB Debug Probe_0/C28xx_CPU1". Then run the code on CPU1 by clicking the green "Resume" button. LED D10 on the LaunchPad should now be blinking at approximately 1Hz.
27. In the Debug window, click on the line "Texas Instruments XDS100v2 USB Debug Probe_0/C28xx_CPU2". As before, then run the code on CPU2 by clicking the "Resume" button. LED D9 should now also be blinking, though at a different frequency than D10.
28. Halt the CPU2 program by clicking on the "Suspend" button. In the Expressions window the ToggleCount2 variable should have recorded a small number of LED state changes. Notice that the ToggleCount1 variable is not recognized on CPU2
29. Click on CPU1 in the Debug window and halt the program using the "Suspend" button. Again, the ToggleCount1 variable should have a small number while ToggleCount2 is unrecognized.

In the forthcoming labs we will explore several other features of the CCS environment, including real-time debugging and the graph plotting capabilities of the software.

Terminate Debug Session and Close Project

30. The "Terminate" button will terminate the active debug session, close the debugger and return CCS to the "CCS Edit" perspective view.

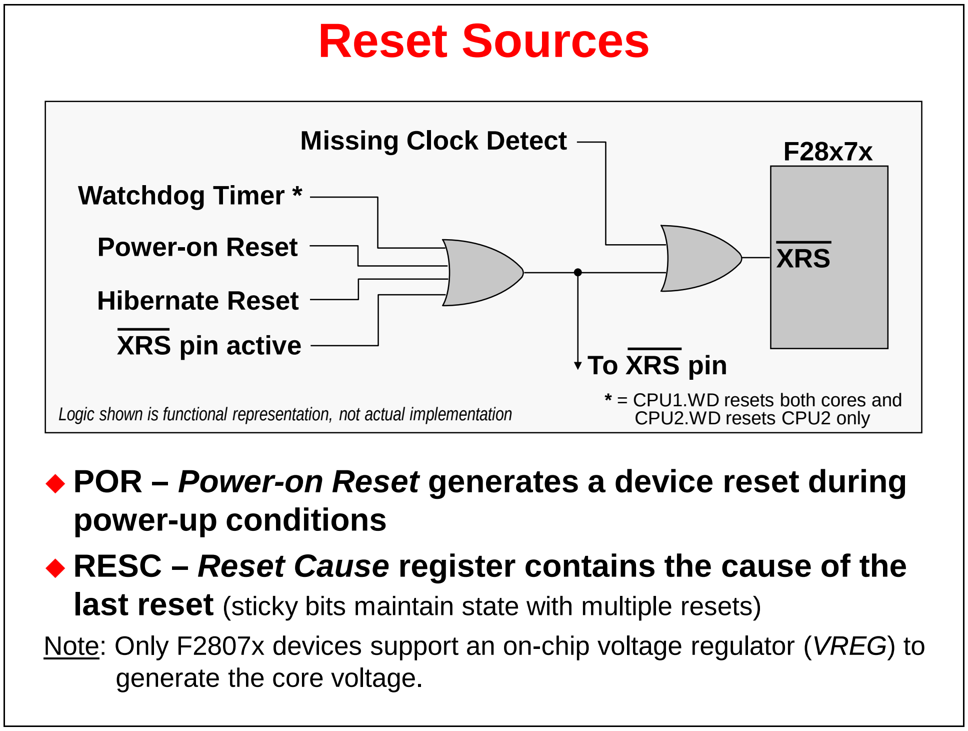
    Click: Run → Terminate or use the Terminate icon: ⏹

31. Next, close the Lab1_cpu01 and Lab1_cpu02 projects by right-clicking on each project in the Project Explorer window and select Close Project.

End of Exercise

Reset, Interrupts and System Initialization

Reset Sources



Reset Sources

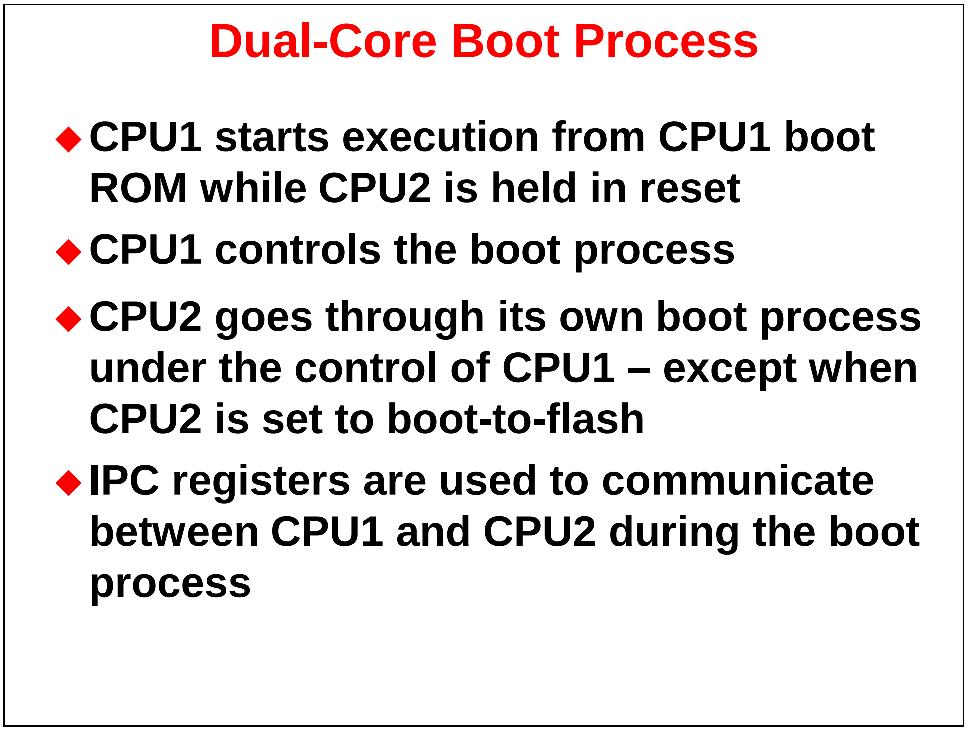
Logic shown is functional representation, not actual implementation

* = 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)

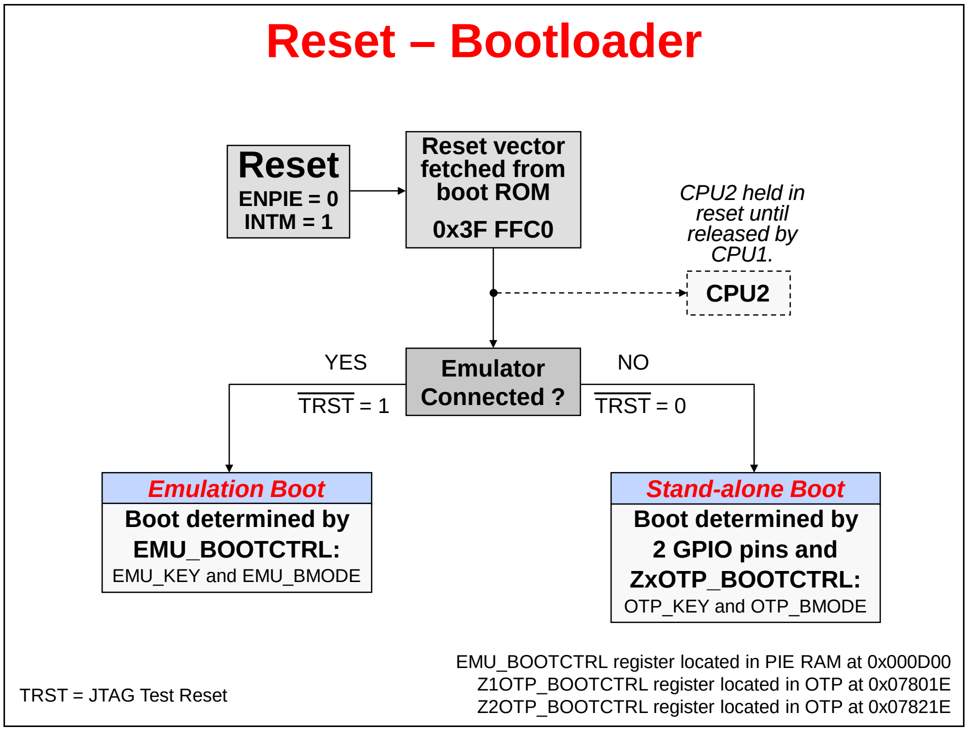
Note: Only F2807x devices support an on-chip voltage regulator (*VREG*) to generate the core voltage.

Boot Process



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**

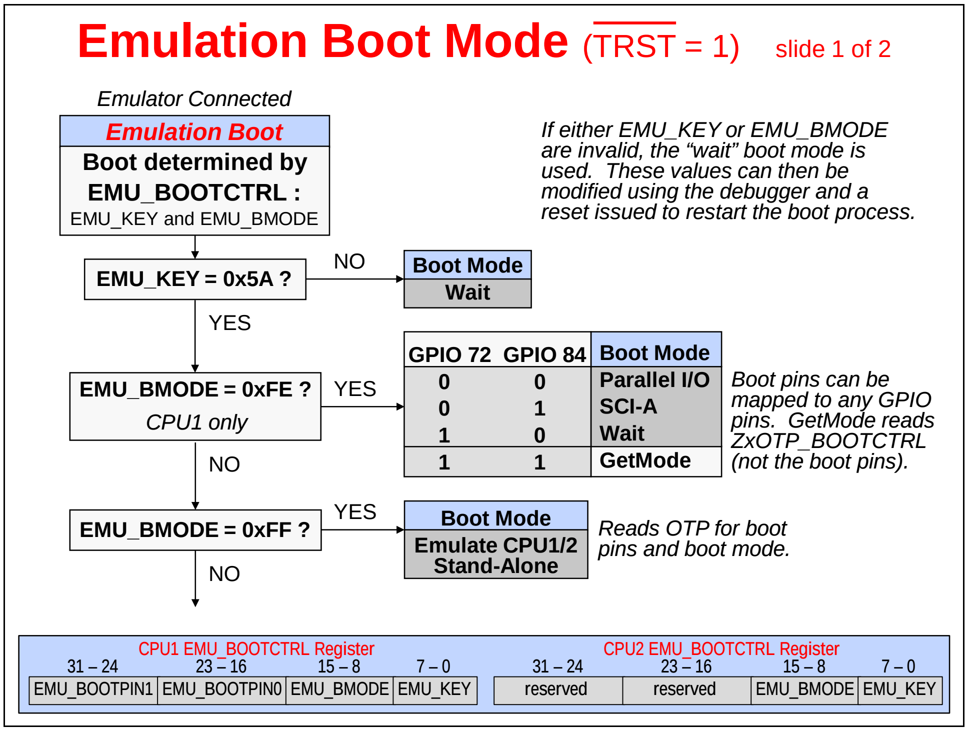


Reset – Bootloader

TRST = JTAG Test Reset

EMU_BOOTCTRL register located in PIE RAM at 0x000D00
Z1OTP_BOOTCTRL register located in OTP at 0x07801E
Z2OTP_BOOTCTRL register located in OTP at 0x07821E

Emulation Boot Mode



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

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.

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 ZxOTP_BOOTCTRL (not the boot pins).

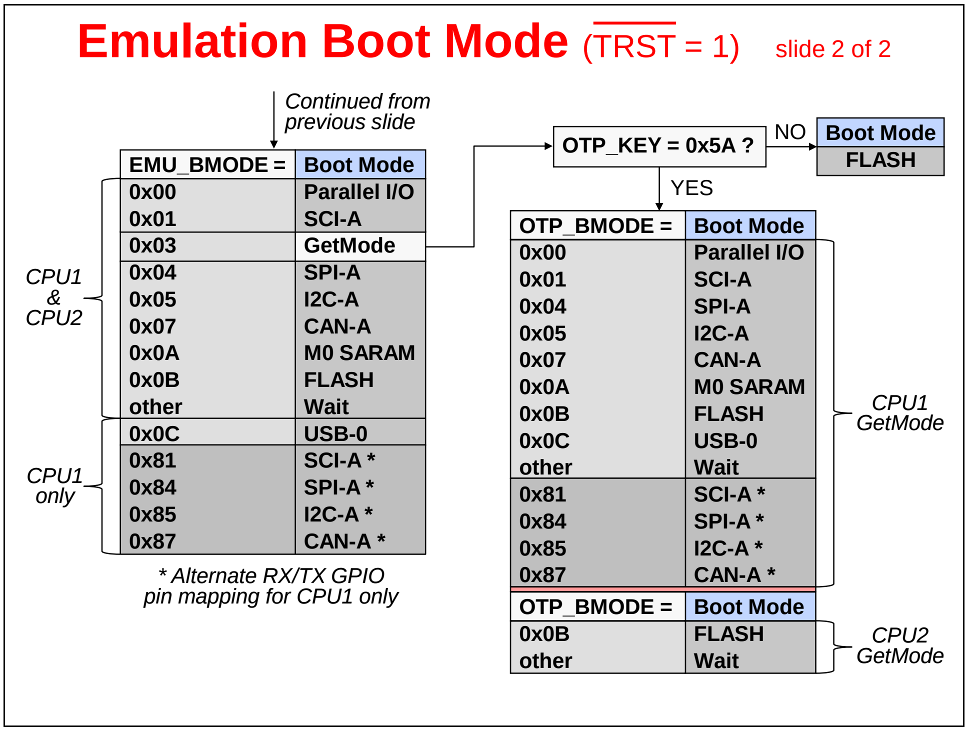
Reads OTP for boot pins and boot mode.

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 (TRST = 1) slide 2 of 2

Continued from previous slide

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

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

OTP_KEY = 0x5A ?

NO

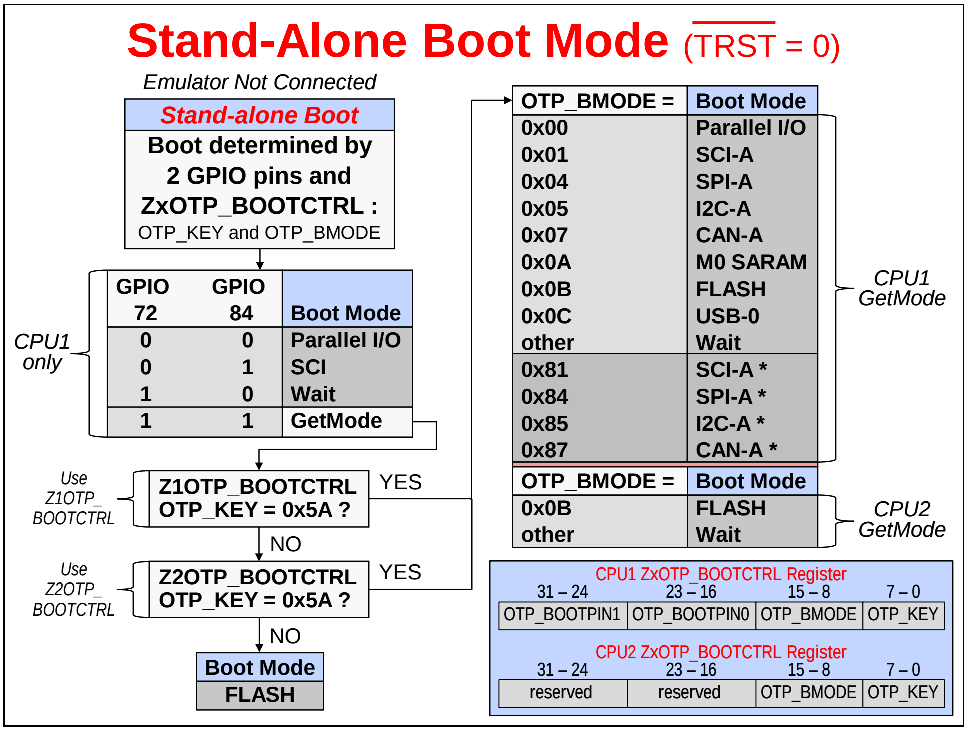
Boot Mode
FLASH

YES

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

OTP_BMODE =	Boot Mode
0x0B	FLASH
other	Wait

Stand-Alone Boot Mode



Stand-Alone Boot Mode (TRST = 0)

Emulator Not Connected

Stand-alone Boot

Boot determined by 2 GPIO pins and ZxOTP_BOOTCTRL :
OTP_KEY and OTP_BMODE

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

Z1OTP_BOOTCTRL
OTP_KEY = 0x5A ?

YES

NO

Z2OTP_BOOTCTRL
OTP_KEY = 0x5A ?

YES

NO

Boot Mode
FLASH

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

OTP_BMODE =	Boot Mode
0x0B	FLASH
other	Wait

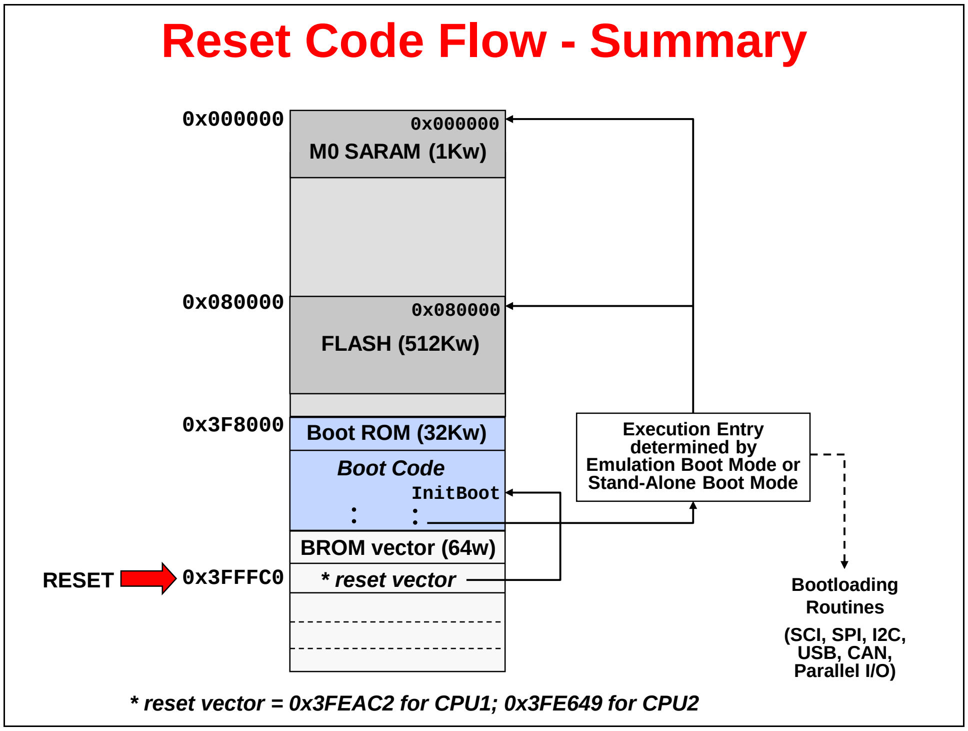
CPU1 ZxOTP_BOOTCTRL Register

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

CPU2 ZxOTP_BOOTCTRL Register

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

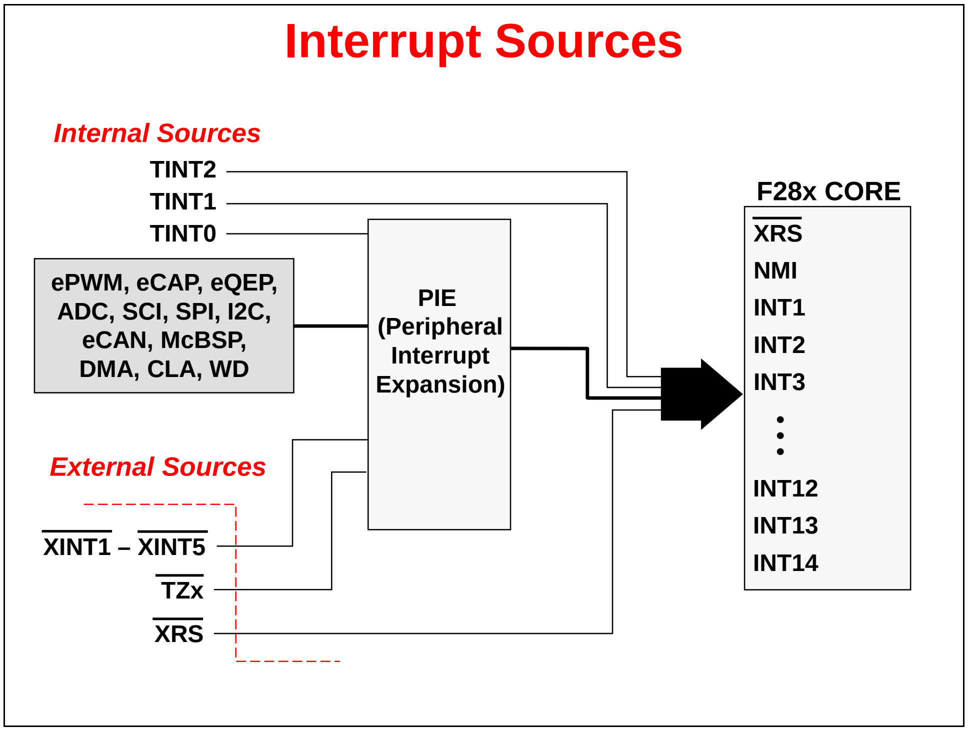
Reset Code Flow – Summary



Reset Code Flow - Summary

** reset vector = 0x3FEAC2 for CPU1; 0x3FE649 for CPU2*

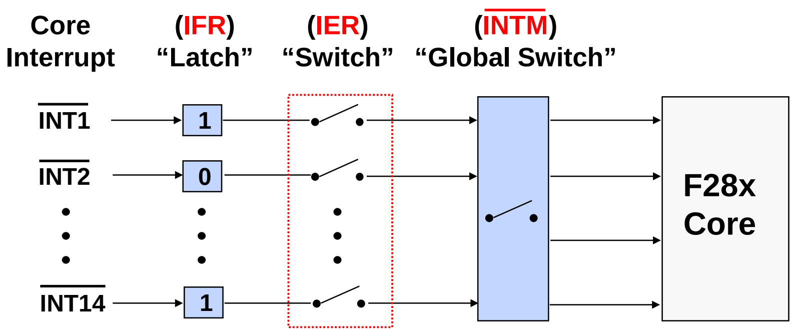
Interrupt Sources



Interrupt Sources

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

Core Interrupt Registers

Interrupt Flag Register (IFR) (pending = 1 / absent = 0)

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

Interrupt Enable Register (IER) (enable = 1 / disable = 0)

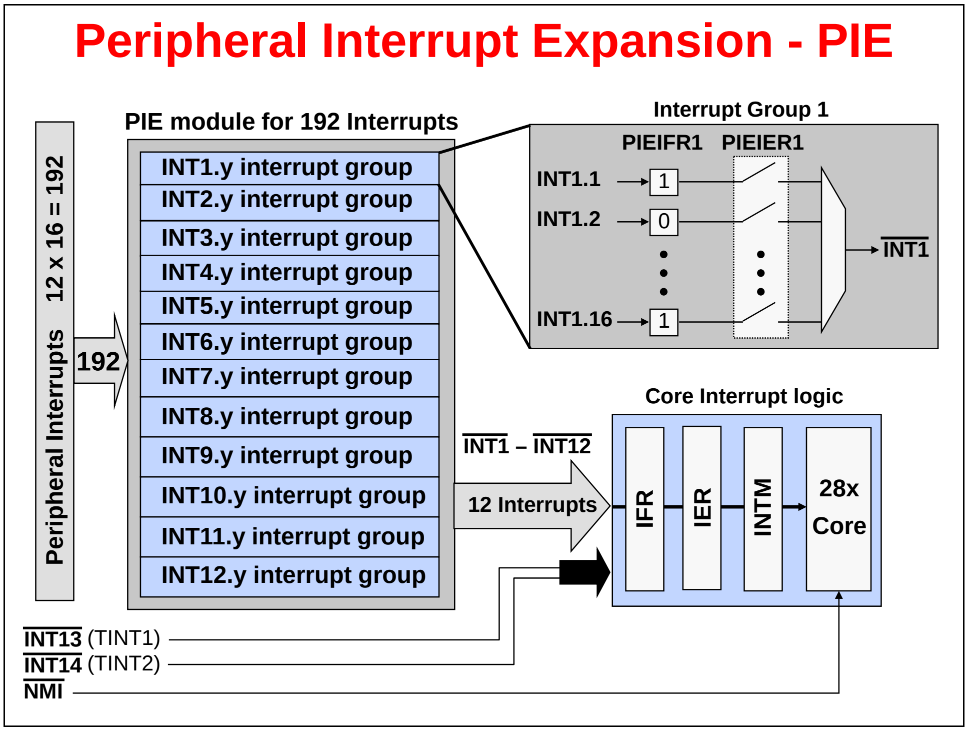
15	14	13	12	11	10	9	8
RTOSINT	DLOGINT	INT14	INT13	INT12	INT11	INT10	INT9
INT8	INT7	INT6	INT5	INT4	INT3	INT2	INT1
7	6	5	4	3	2	1	0

Interrupt Global Mask Bit (INTM)

		Bit 0	
ST1		$\overline{\text{INTM}}$	(enable = 0 / disable = 1)

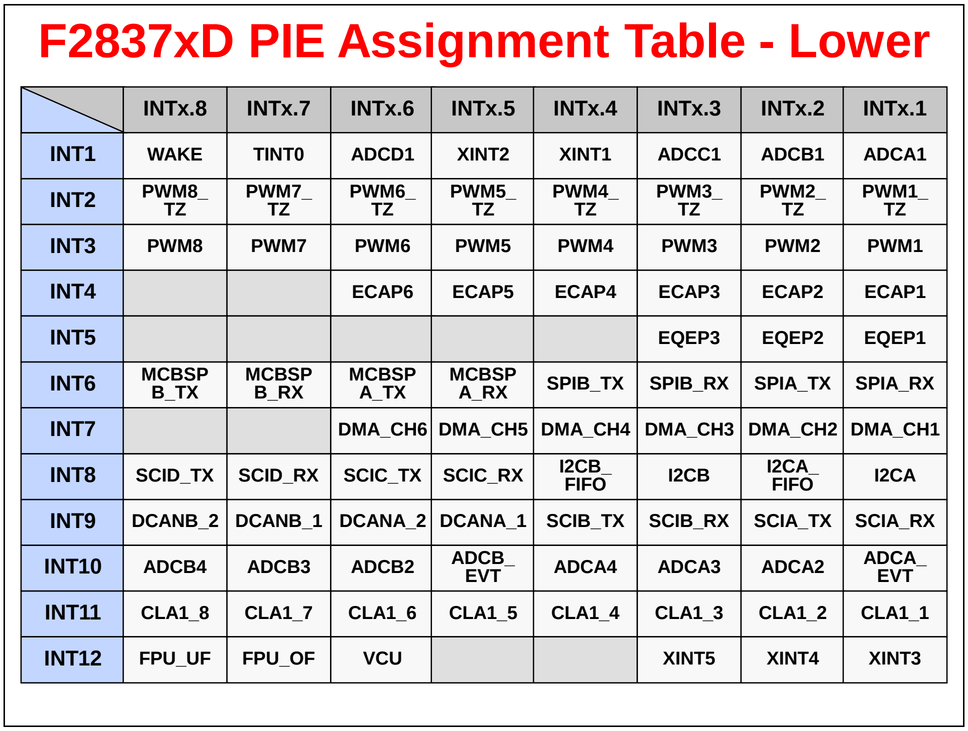
```
/*** Interrupt Enable Register ***/
extern cregister volatile unsigned int IER;
     IER  |= 0x0008;              //enable INT4 in IER
     IER &= 0xFFF7;               //disable INT4 in IER
/*** Global Interrupts ***/
     asm(" CLRC  INTM");     //enable global interrupts
     asm(" SETC  INTM");     //disable global interrupts
```

Peripheral Interrupt Expansion – PIE



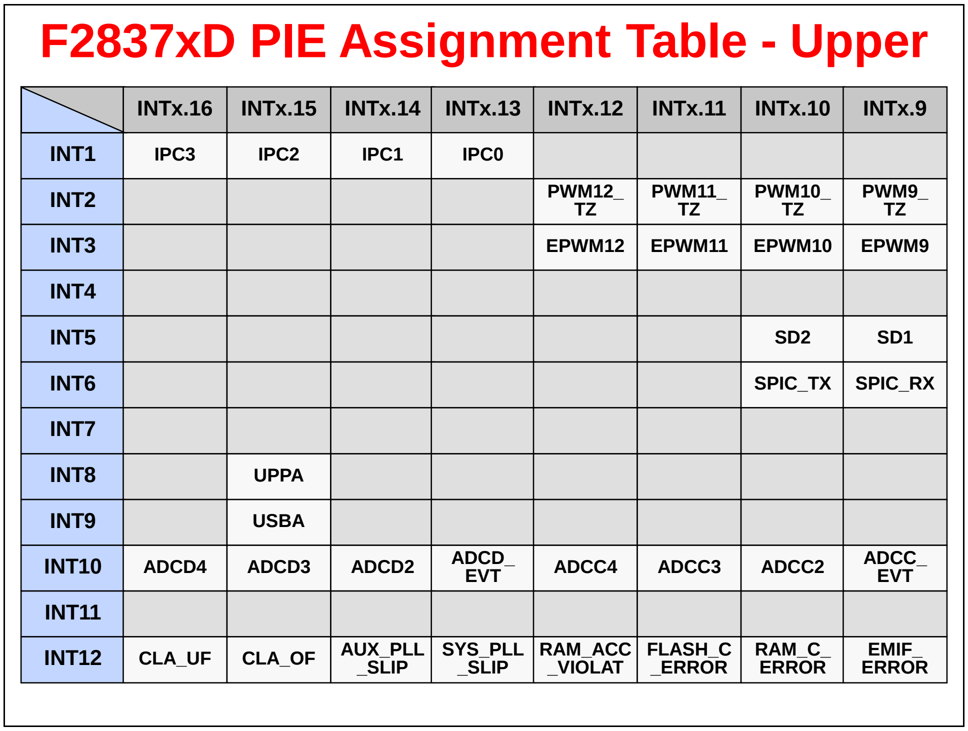
Peripheral Interrupt Expansion - PIE

F2837xD PIE Assignment Table



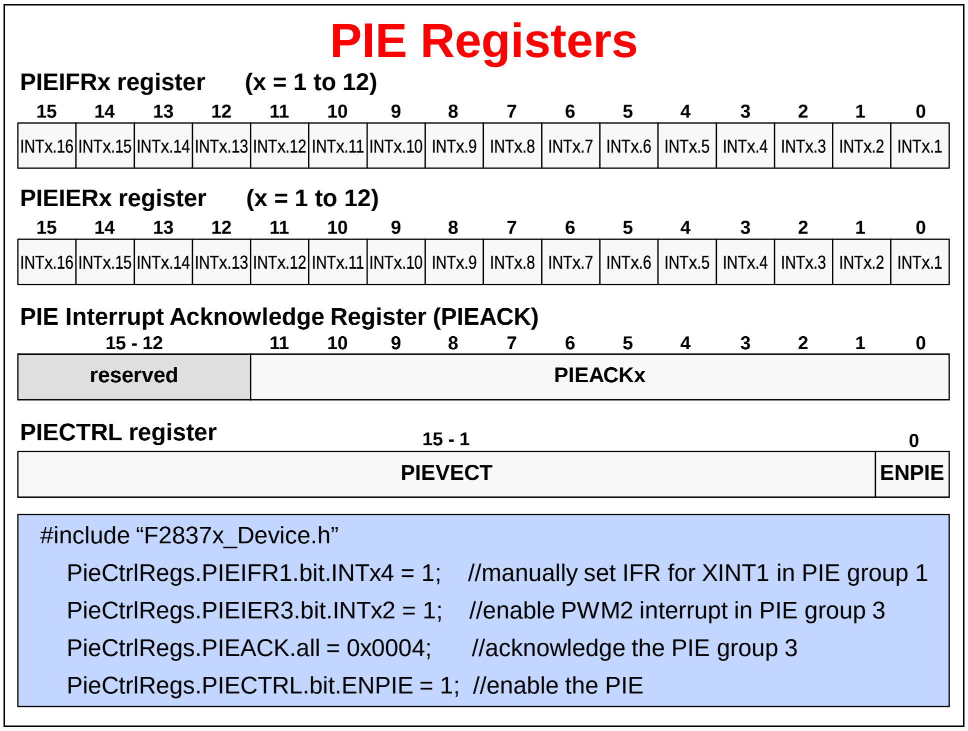
F2837xD PIE Assignment Table - Lower

	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	MCBSPB_TX	MCBSPB_RX	MCBSPA_TX	MCBSPA_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	DCANB_2	DCANB_1	DCANA_2	DCANA_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



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



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)

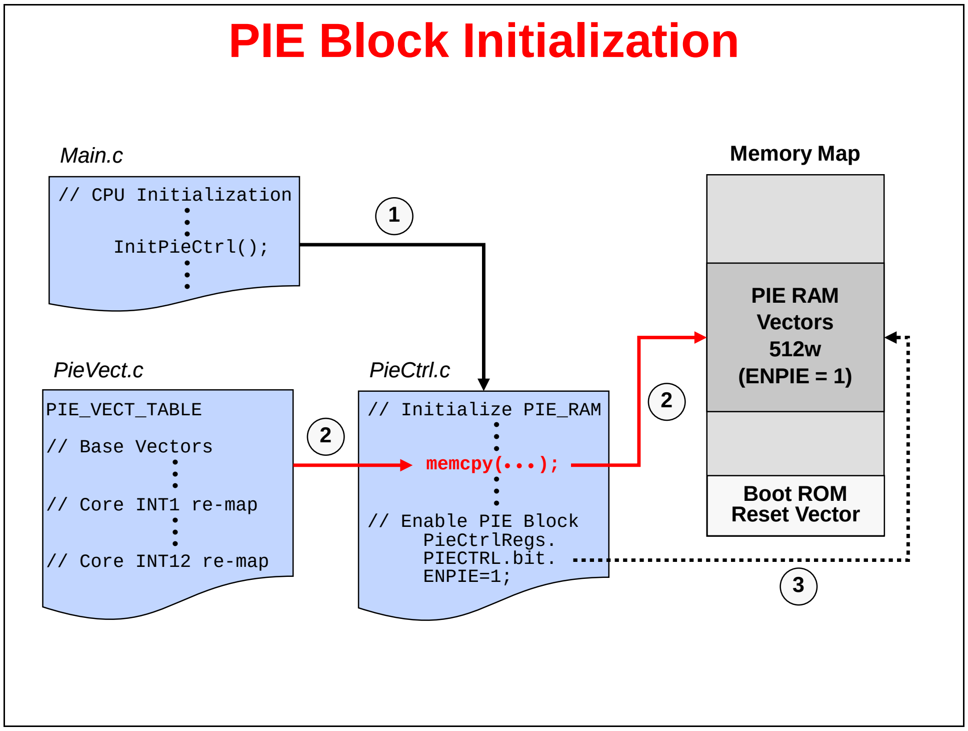
15 - 12	11	10	9	8	7	6	5	4	3	2	1	0
reserved	PIEACKx											

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

Main.c

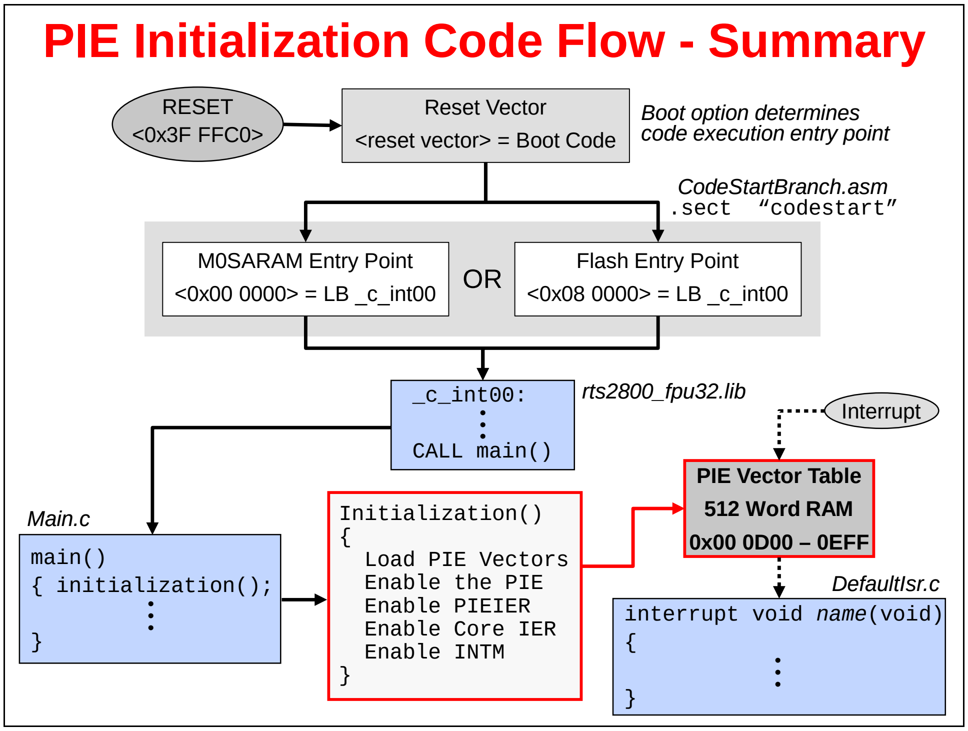
```
// CPU Initialization
        :
     InitPieCtrl();
        :
```

PieVect.c

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

PieCtrl.c

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



PIE Initialization Code Flow - Summary

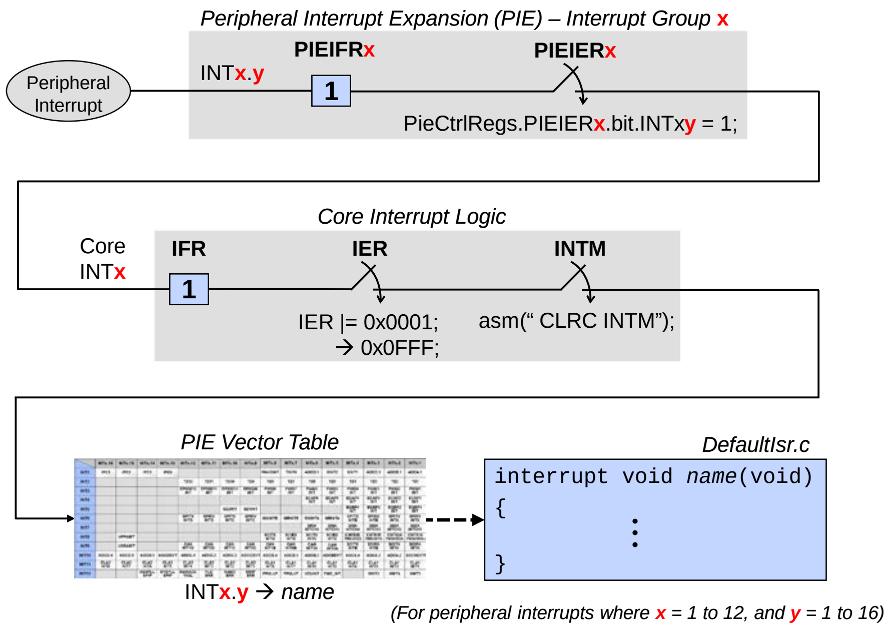
```
_c_int00:
     :
CALL main()
```

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

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

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

Interrupt Signal Flow – Summary



Peripheral Interrupt Expansion (PIE) – Interrupt Group x

Peripheral Interrupt

INTx.y

PIEIFRx

1

PIEIERx

PieCtrlRegs.PIEIERx.bit.INTxy = 1;

Core Interrupt Logic

Core INTx

IFR

1

IER

IER |= 0x0001;
→ 0x0FFF;

INTM

asm(" CLRC INTM");

PIE Vector Table

INTx.y → *name*

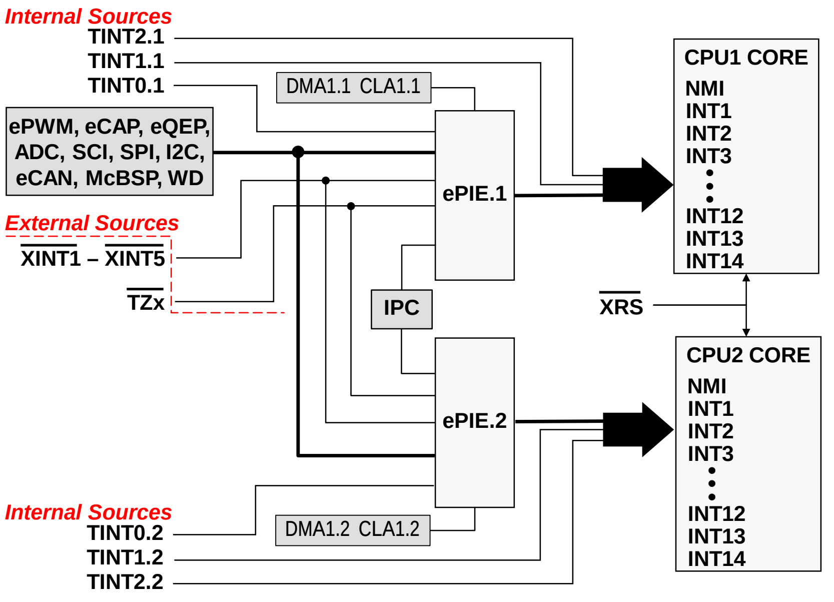
DefaultIsr.c

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

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

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, eCAN, McBSP, WD

External Sources

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

$\overline{\text{TZx}}$

ePIE.1

IPC

ePIE.2

CPU1 CORE

NMI
INT1
INT2
INT3
⋮
INT12
INT13
INT14

$\overline{\text{XRS}}$

CPU2 CORE

NMI
INT1
INT2
INT3
⋮
INT12
INT13
INT14

DMA1.2 CLA1.2

Internal Sources

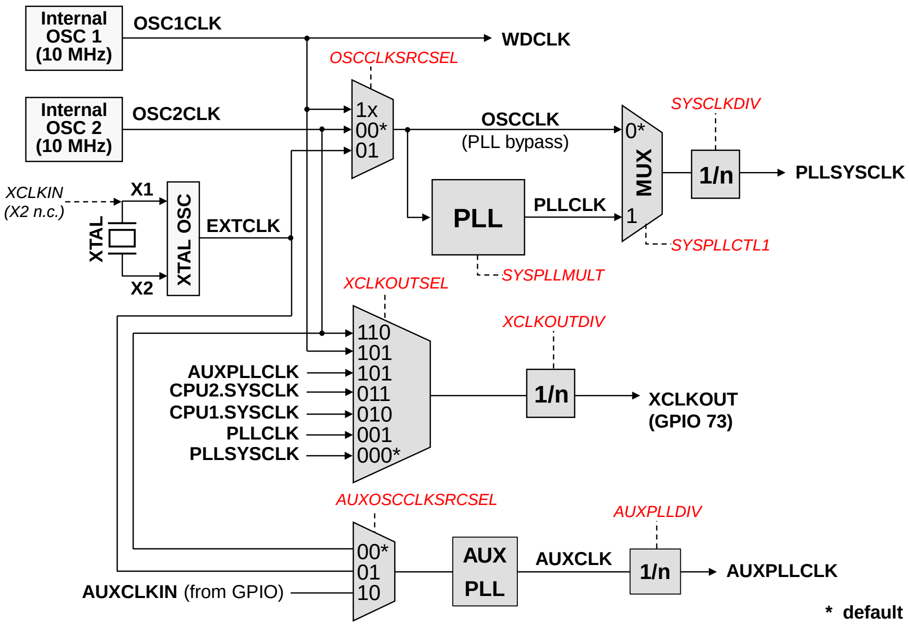
TINT0.2

TINT1.2

TINT2.2

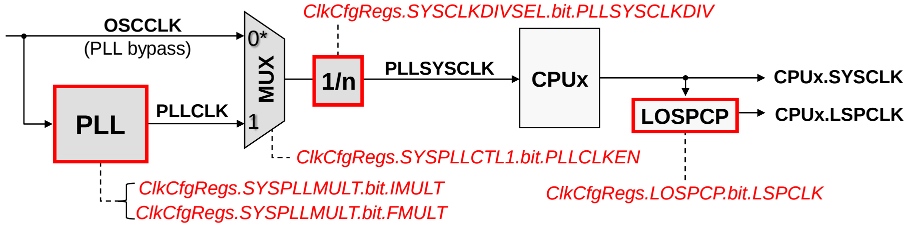
F28x7x Oscillator / PLL Clock Module

F28x7x Oscillator / PLL Clock Module



* default

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

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

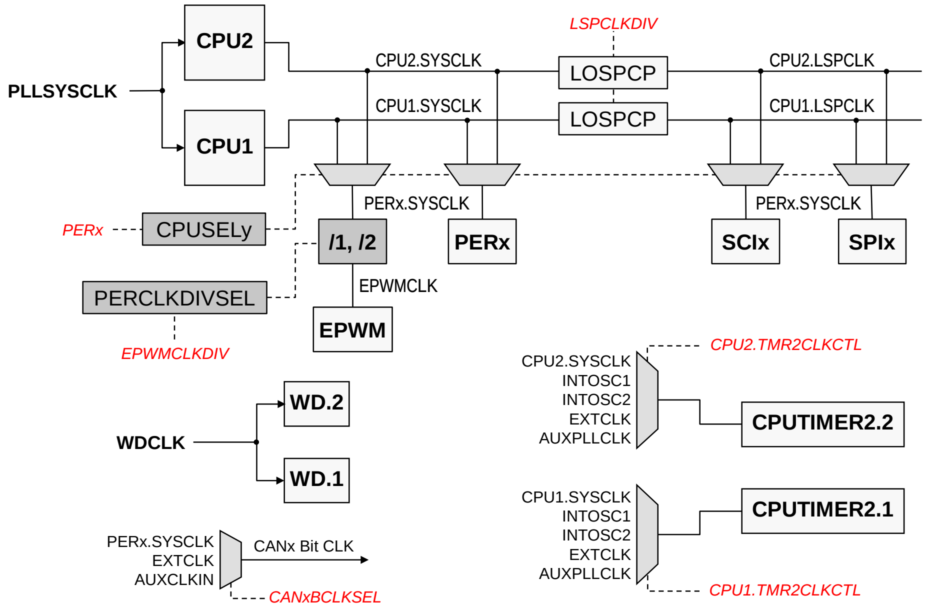
SYSPLL DIVSEL	n
111111	/126
• • •	• • •
000010	/4 *
000001	/2
000000	/1

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

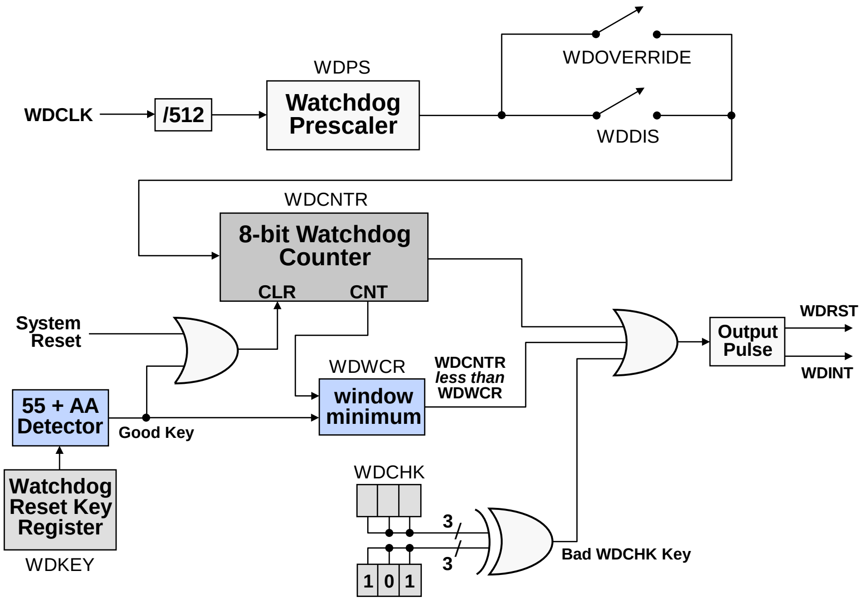
* default

F2837xD Dual-Core System Clock



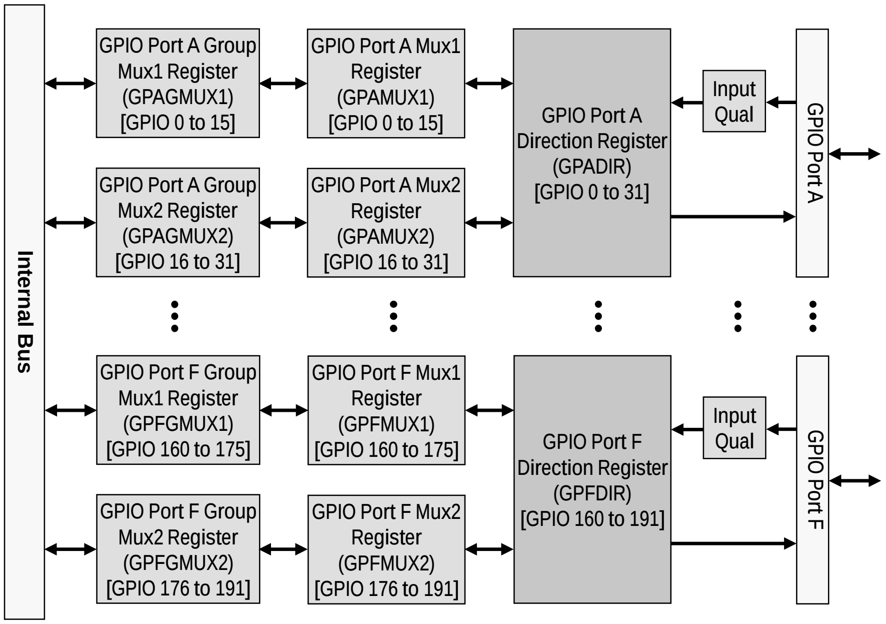
Watchdog Timer Module

Watchdog Timer Module



F28x7x General-Purpose Input-Output

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]

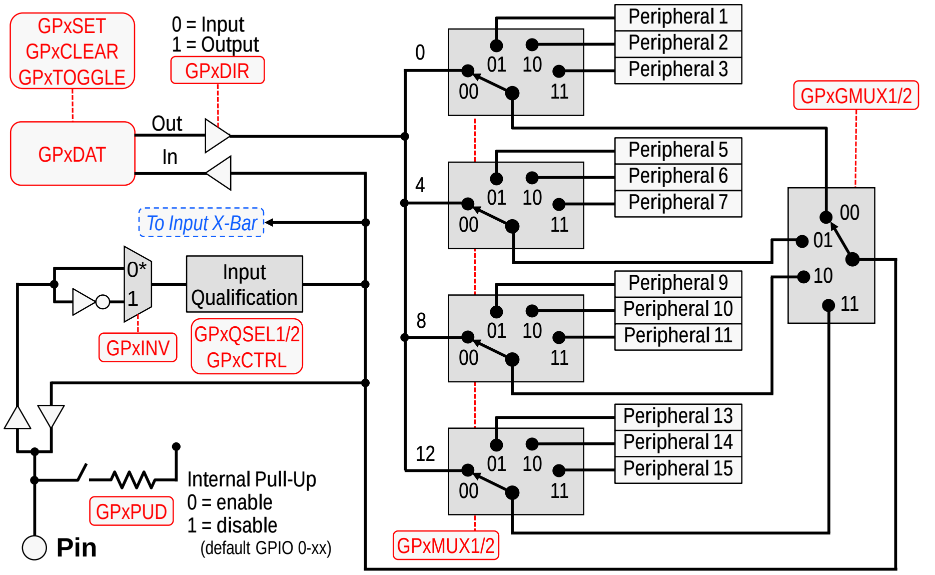
GPIO Port F Mux2 Register (GPFMUX2) [GPIO 176 to 191]

GPIO Port F Direction Register (GPFDIR) [GPIO 160 to 191]

Input Qual

GPIO Port F

F28x7x GPIO Pin Block Diagram



GPxSET
GPxCLEAR
GPxTOGGLE

GPxDAT

0 = Input
1 = Output
GPxDIR

Out

In

To Input X-Bar

0*
1

GPxINV

Input Qualification

GPxQSEL1/2
GPxCTRL

GPxPUD

Internal Pull-Up
0 = enable
1 = disable
(default GPIO 0-xx)

Pin

0
4
8
12

00 01 10 11

Peripheral 1
Peripheral 2
Peripheral 3
Peripheral 5
Peripheral 6
Peripheral 7
Peripheral 9
Peripheral 10
Peripheral 11
Peripheral 13
Peripheral 14
Peripheral 15

GPxMUX1/2

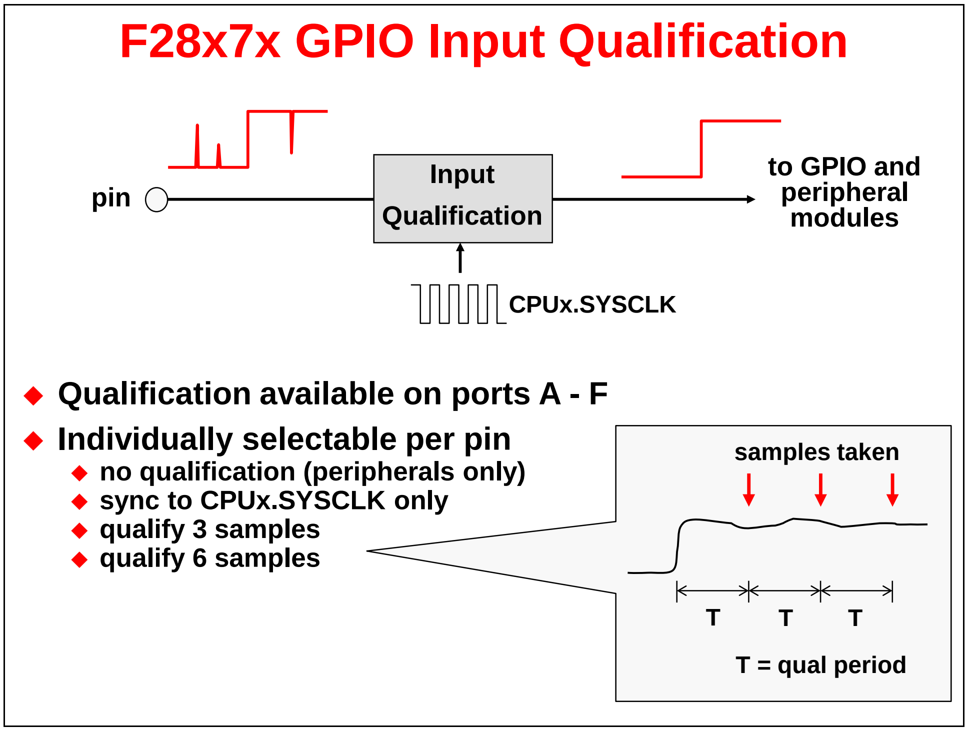
GPxGMUX1/2

Output X-Bar muxed with Peripheral GPIO pins

See device datasheet for pin function selection matrices

Logic shown is functional representation, not actual implementation

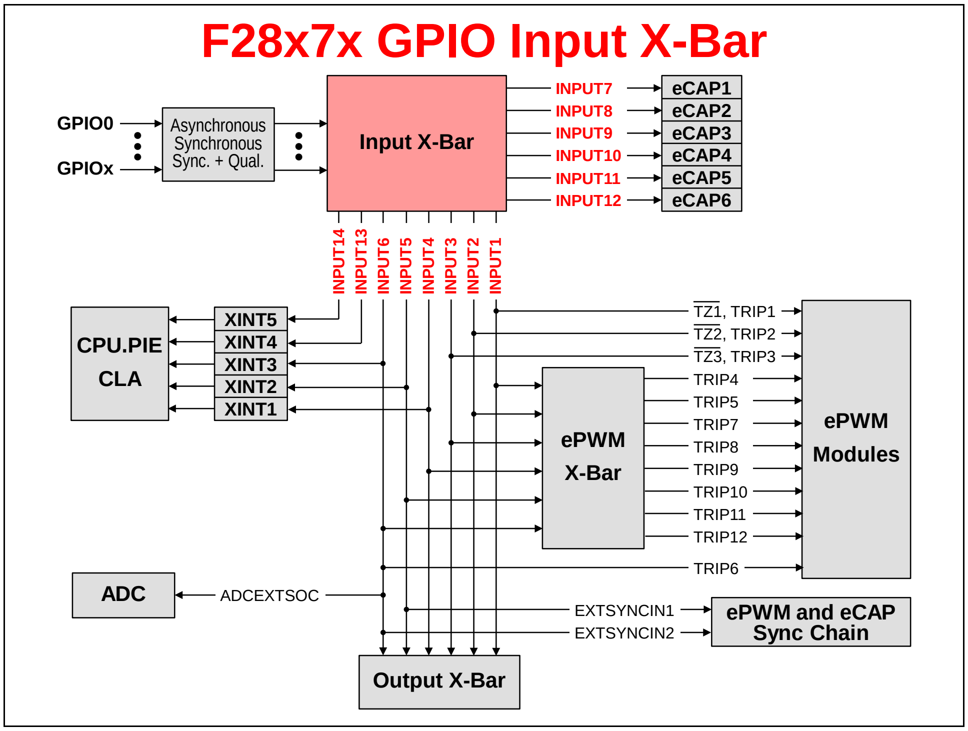
* = Default x = A, B, C, D, E, or F



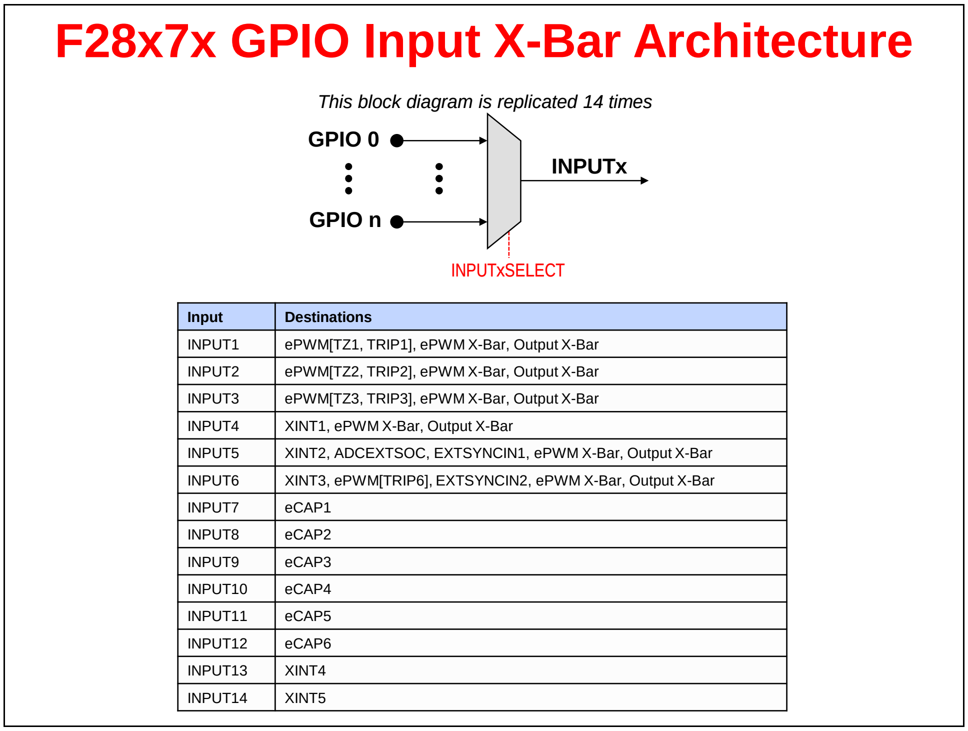
F28x7x GPIO Input Qualification

- 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

GPIO Input X-Bar



F28x7x GPIO Input X-Bar

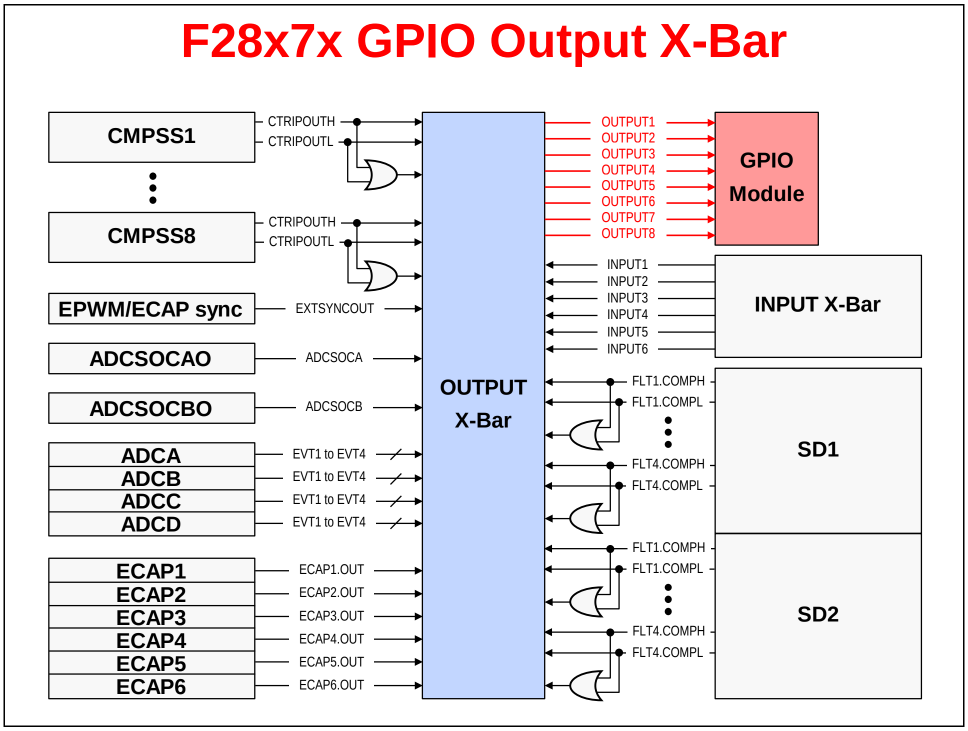


F28x7x GPIO Input X-Bar Architecture

This block diagram is replicated 14 times

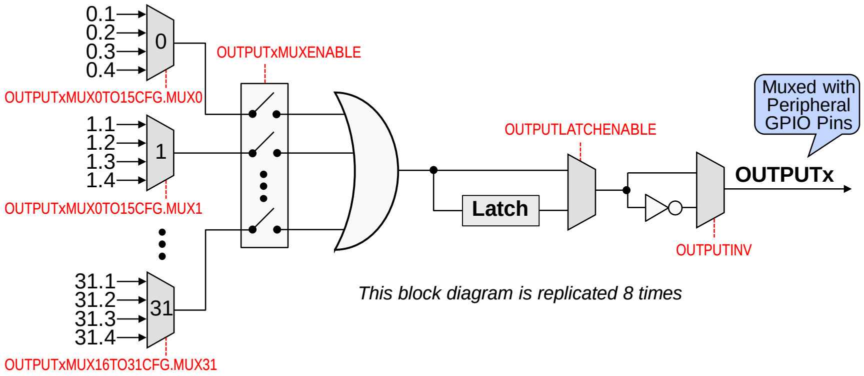
Input	Destinations
INPUT1	ePWM[TZ1, TRIP1], ePWM X-Bar, Output X-Bar
INPUT2	ePWM[TZ2, TRIP2], ePWM X-Bar, Output X-Bar
INPUT3	ePWM[TZ3, TRIP3], ePWM X-Bar, Output X-Bar
INPUT4	XINT1, ePWM X-Bar, Output X-Bar
INPUT5	XINT2, ADCEXTSOC, EXTSYNCIN1, ePWM X-Bar, Output X-Bar
INPUT6	XINT3, ePWM[TRIP6], EXTSYNCIN2,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

GPIO Output X-Bar



F28x7x GPIO Output X-Bar

F28x7x GPIO Output X-Bar Architecture

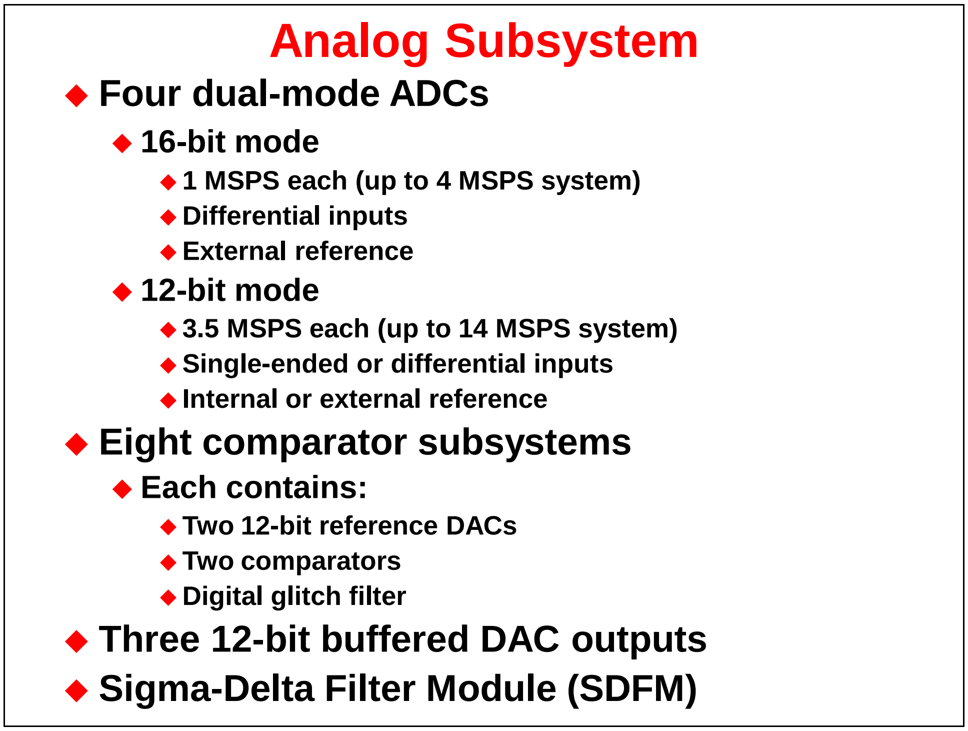


This block diagram is replicated 8 times

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

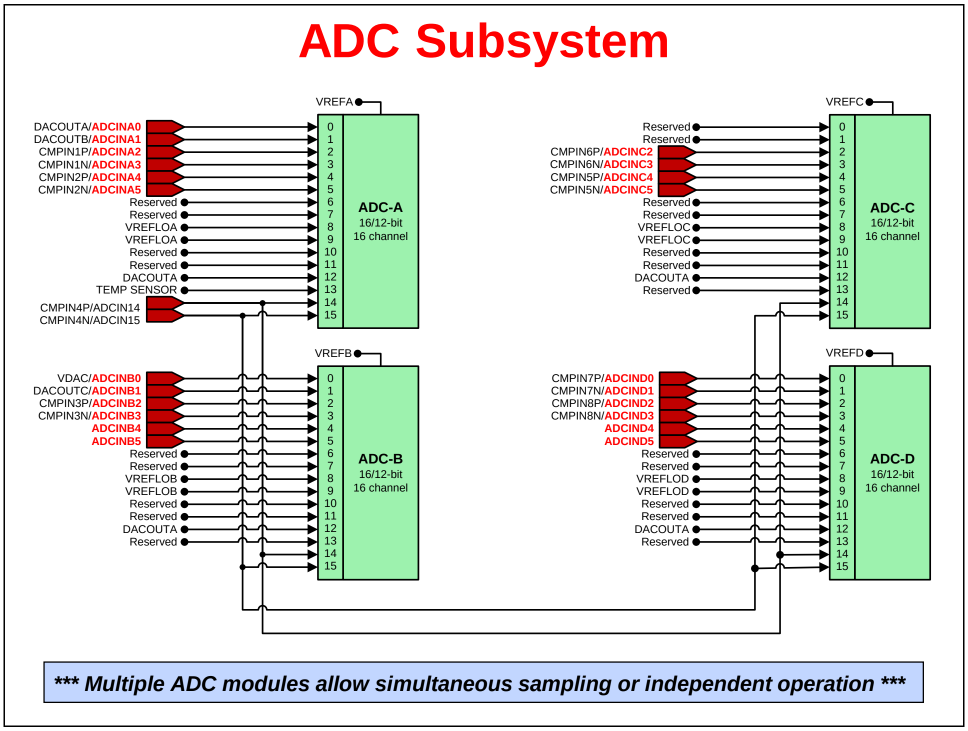
Analog Subsystem



Analog Subsystem

- 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 or differential inputs
    - Internal or external reference
- 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)

ADC Subsystem

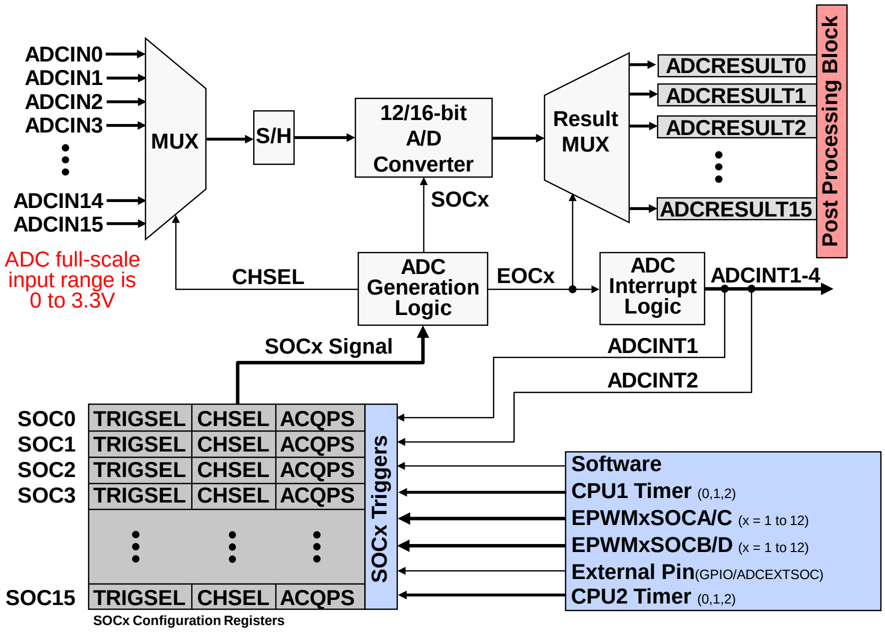


ADC Subsystem

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

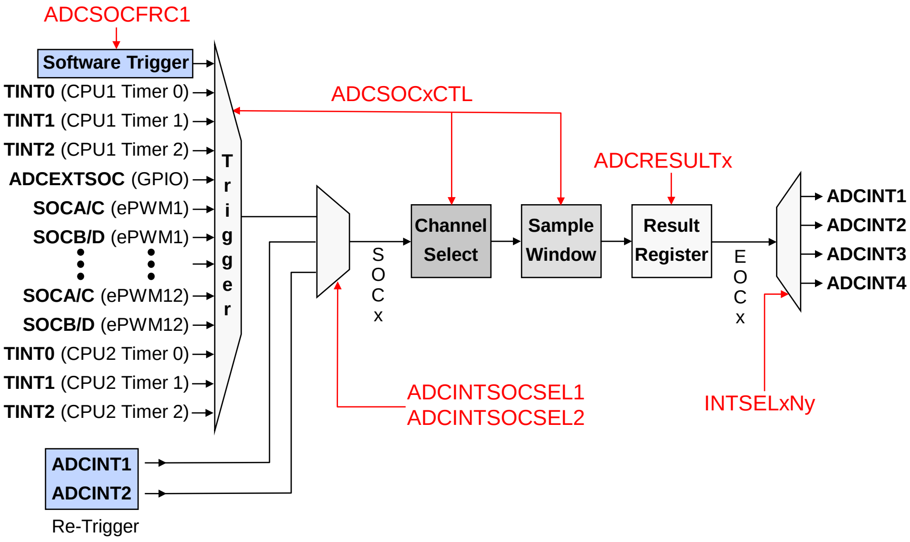
ADC Module Block Diagram

ADC Module Block Diagram



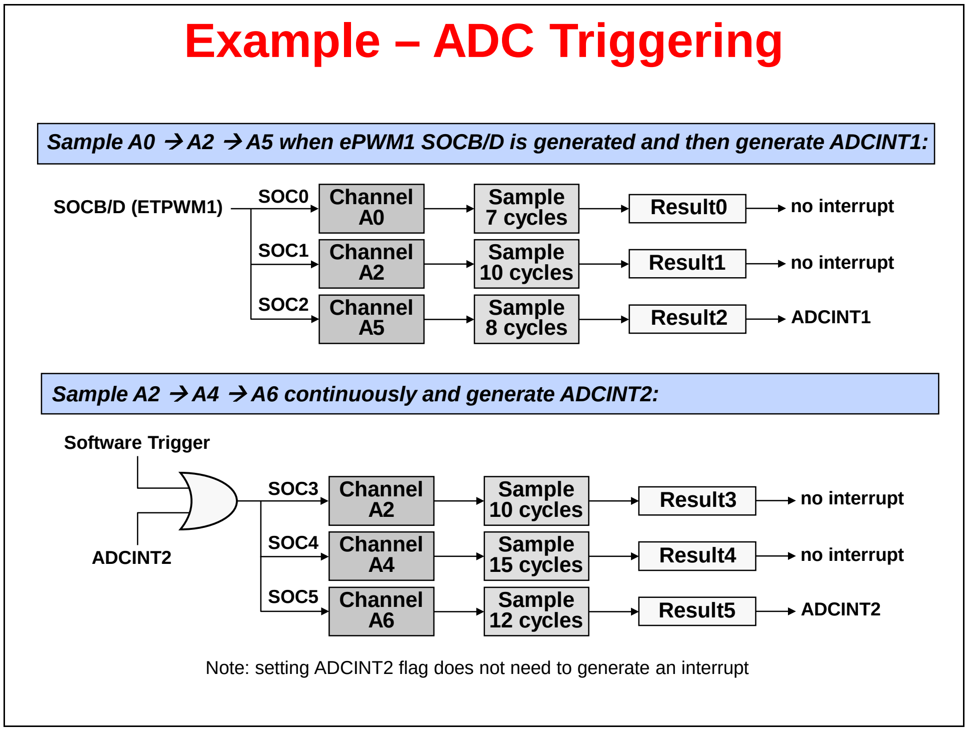
SOC0	TRIGSEL	CHSEL	ACQPS
SOC1	TRIGSEL	CHSEL	ACQPS
SOC2	TRIGSEL	CHSEL	ACQPS
SOC3	TRIGSEL	CHSEL	ACQPS
	⋮	⋮	⋮
SOC15	TRIGSEL	CHSEL	ACQPS

ADC SOCx Functional Diagram



This block diagram is replicated 16 times

ADC Triggering

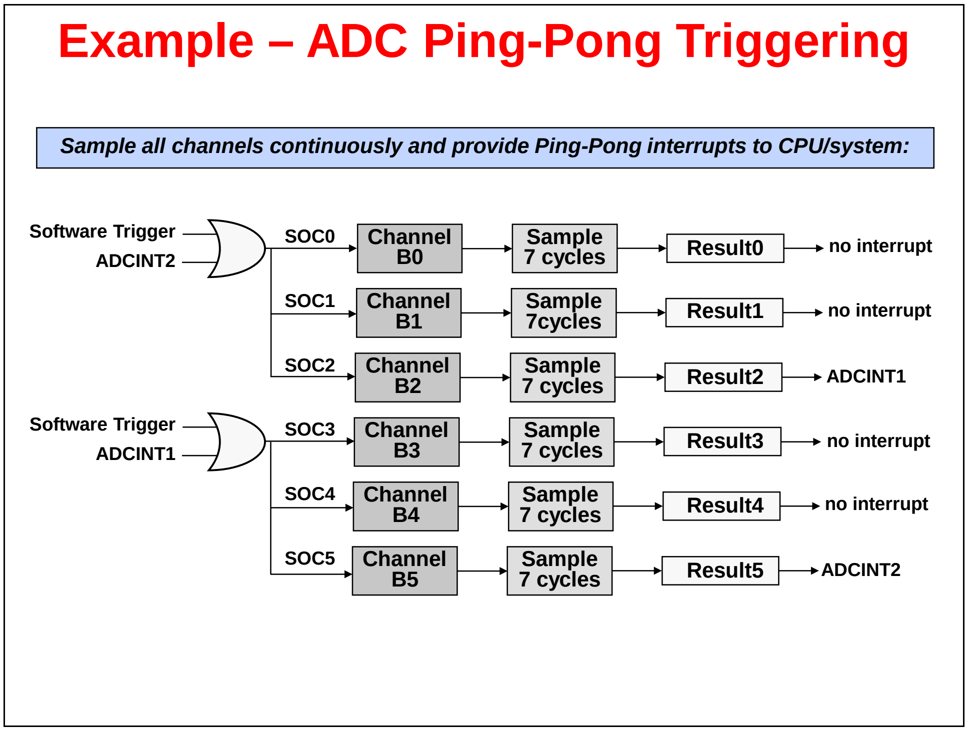


Example – ADC Triggering

Sample A0 → A2 → A5 when ePWM1 SOCB/D is generated and then generate ADCINT1:

Sample A2 → A4 → A6 continuously and generate ADCINT2:

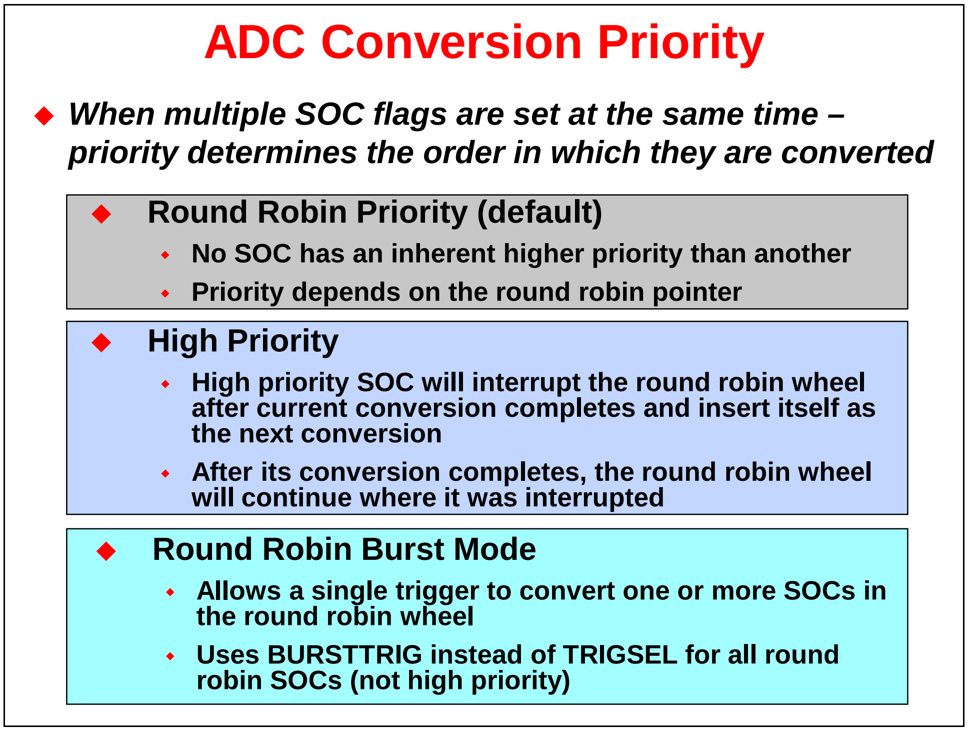
Note: setting ADCINT2 flag does not need to generate an interrupt



Example – ADC Ping-Pong Triggering

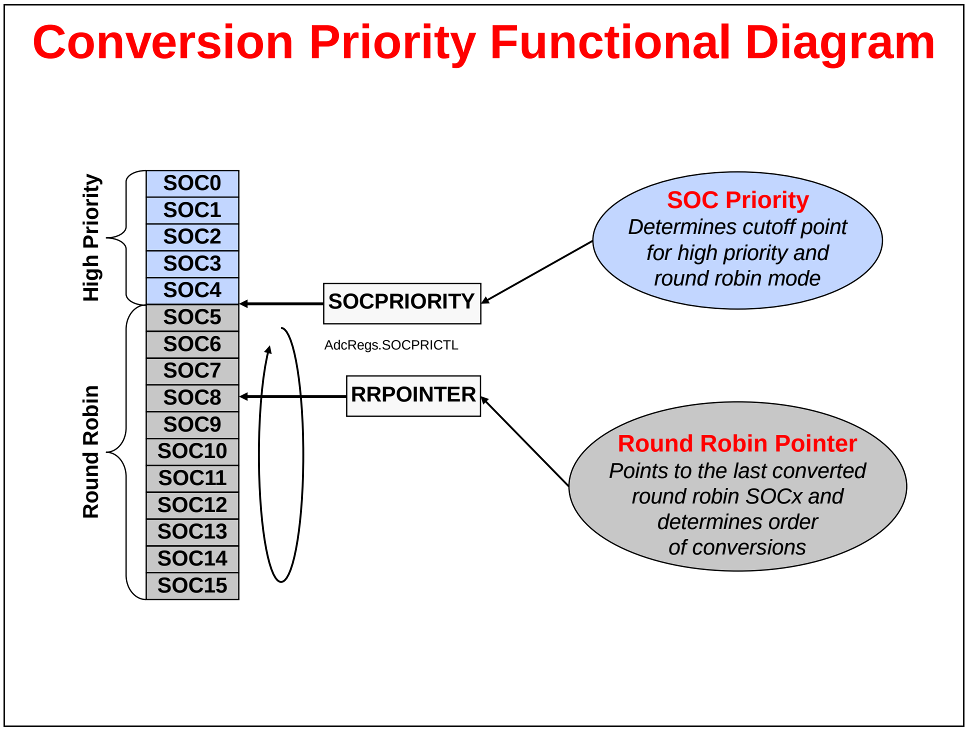
Sample all channels continuously and provide Ping-Pong interrupts to CPU/system:

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)**

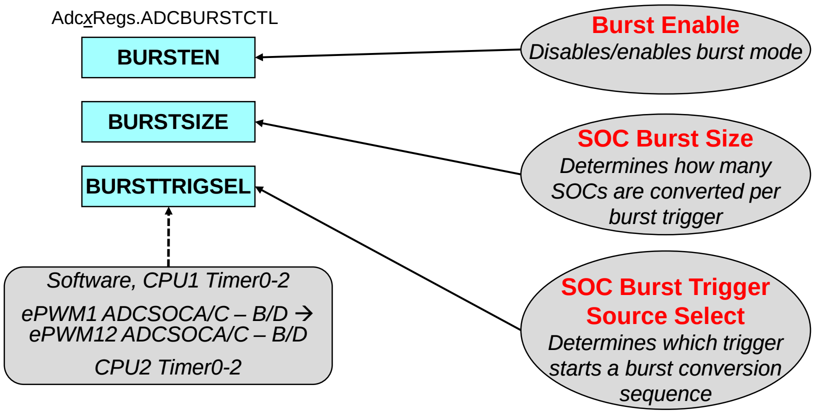


Conversion Priority Functional Diagram

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

Round Robin Burst Mode Diagram

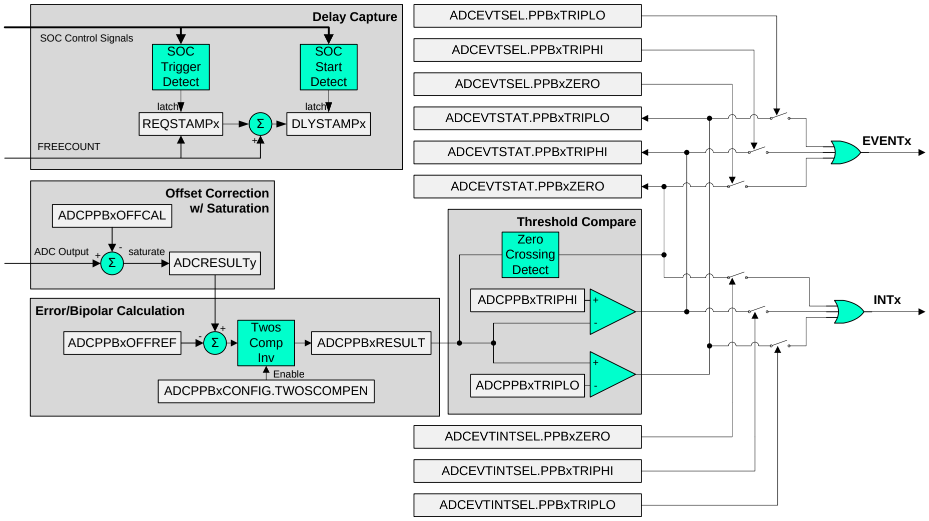

AdcxRegs.ADCBURSTCTL
BURSTEN
BURSTSIZE
BURSTTRIGSEL
Software, CPU1 Timer0-2
ePWM1 ADCSOCA/C – B/D → ePWM12 ADCSOCA/C – B/D
CPU2 Timer0-2
Burst Enable
Disables/enables burst mode
SOC Burst Size
Determines how many SOCs are converted per burst trigger
SOC Burst Trigger Source Select
Determines which trigger starts a burst conversion sequence

Post Processing Block

Purpose of the Post Processing Block

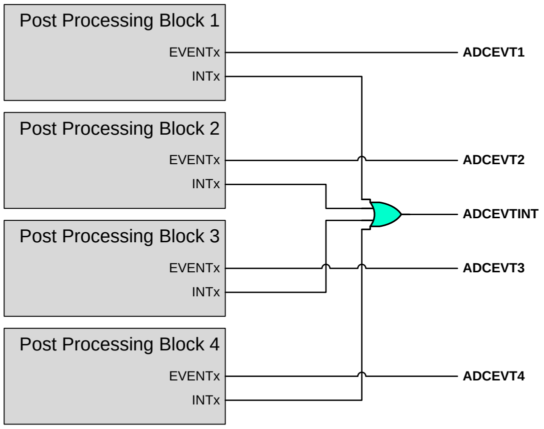
- Offset Correction
  - ***Remove an offset associated with an ADCIN channel possibly caused by external sensors and signal sources***
    - Zero-overhead; saving cycles
- Error from Setpoint Calculation
  - ***Subtract out a reference value which can be used to automatically calculate an error from 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

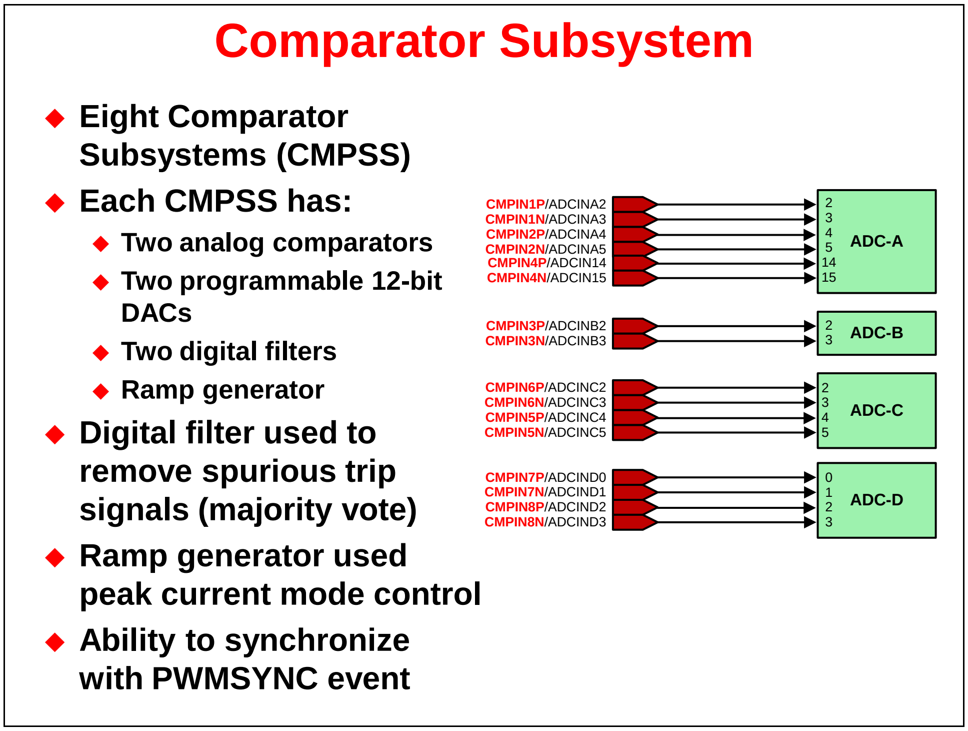


Post Processing Block Interrupt Event

- Each ADC module contains four (4) Post Processing Blocks
- Each Post Processing Block and be associated with any of the 16 ADCRESULTx registers



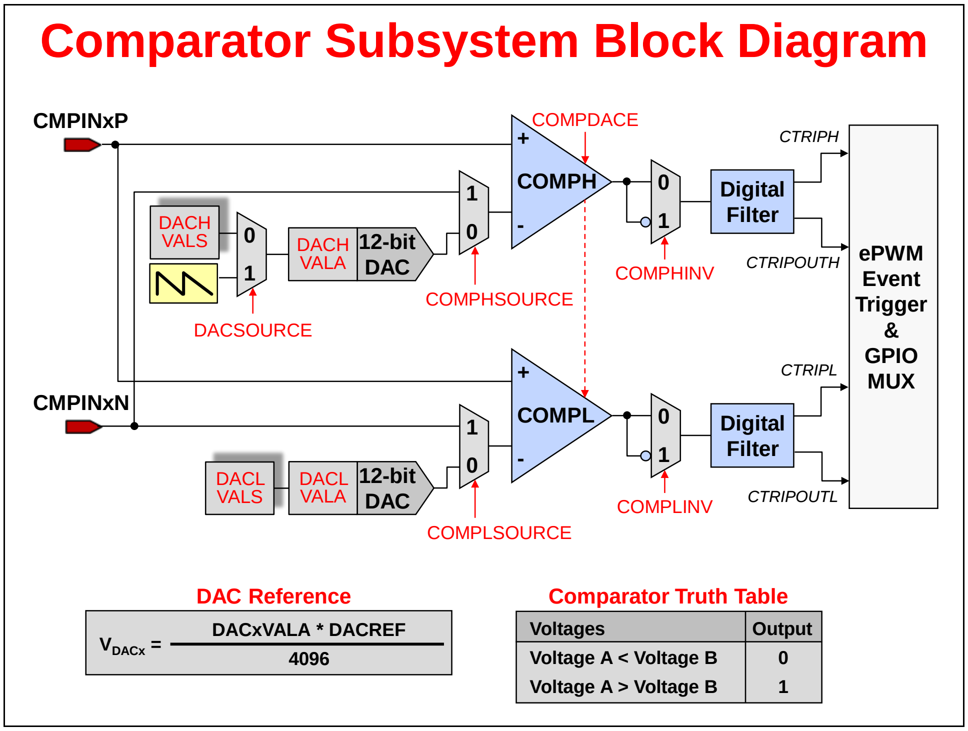
Comparator Subsystem



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 peak current mode control
- Ability to synchronize with PWMSYNC event

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



Comparator Subsystem Block Diagram

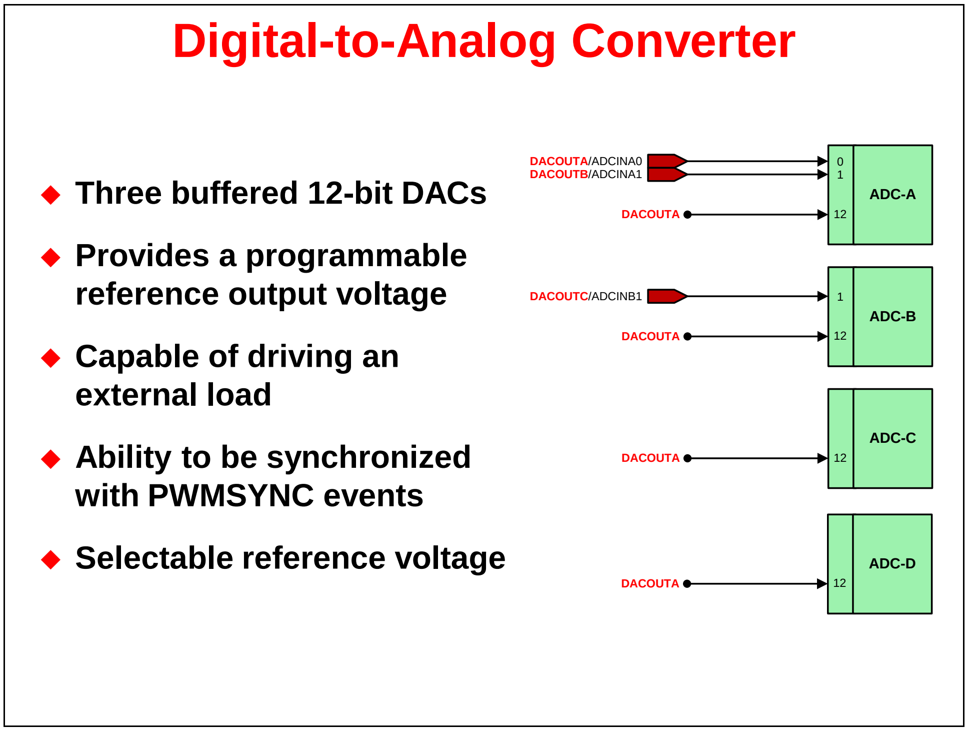
DAC Reference

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

Comparator Truth Table

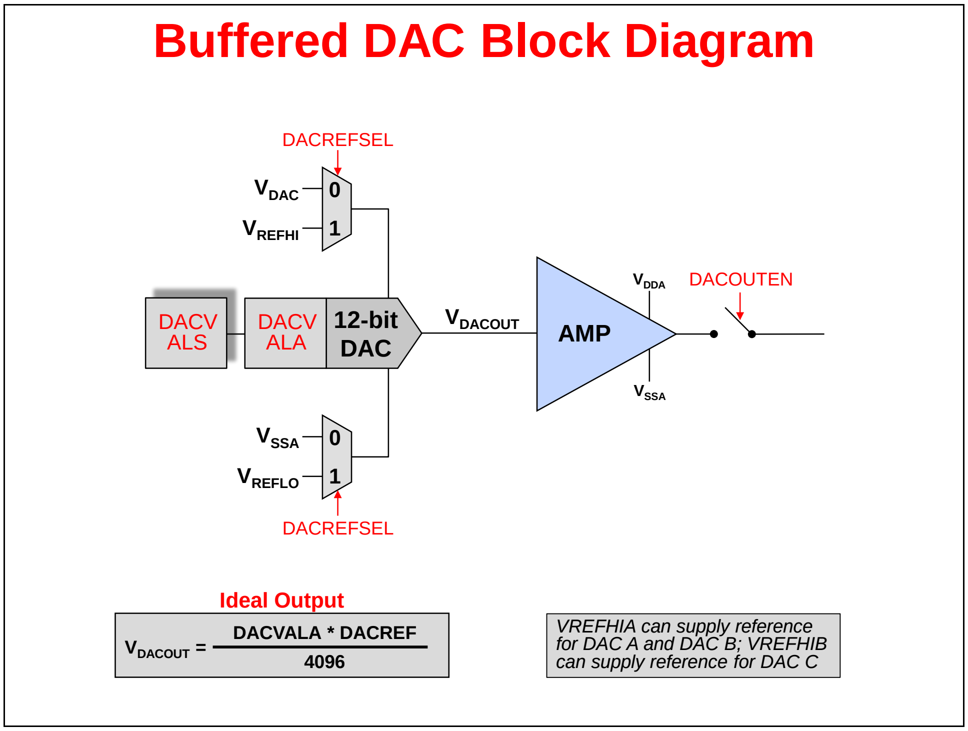
Voltages	Output
Voltage A < Voltage B	0
Voltage A > Voltage B	1

Digital-to-Analog Converter



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



Buffered DAC Block Diagram

Ideal Output

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

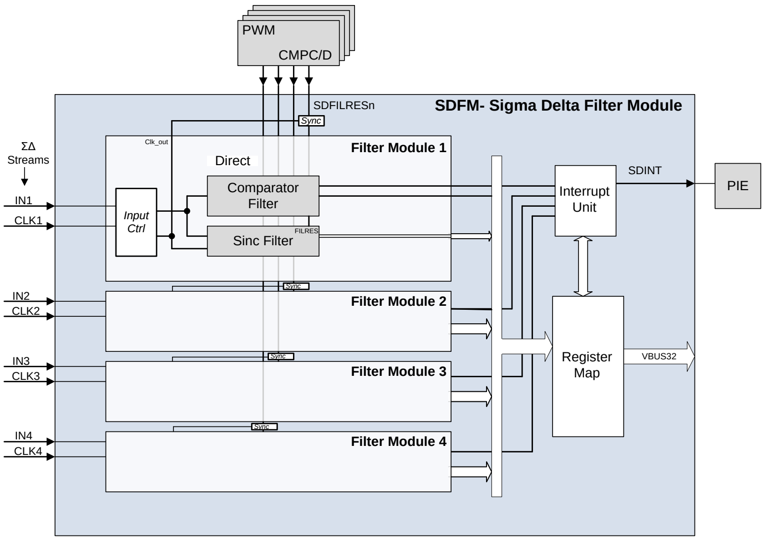
VREFHIA can supply reference for DAC A and DAC B; VREFHIB can supply reference for DAC C

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

SDFM Block Diagram

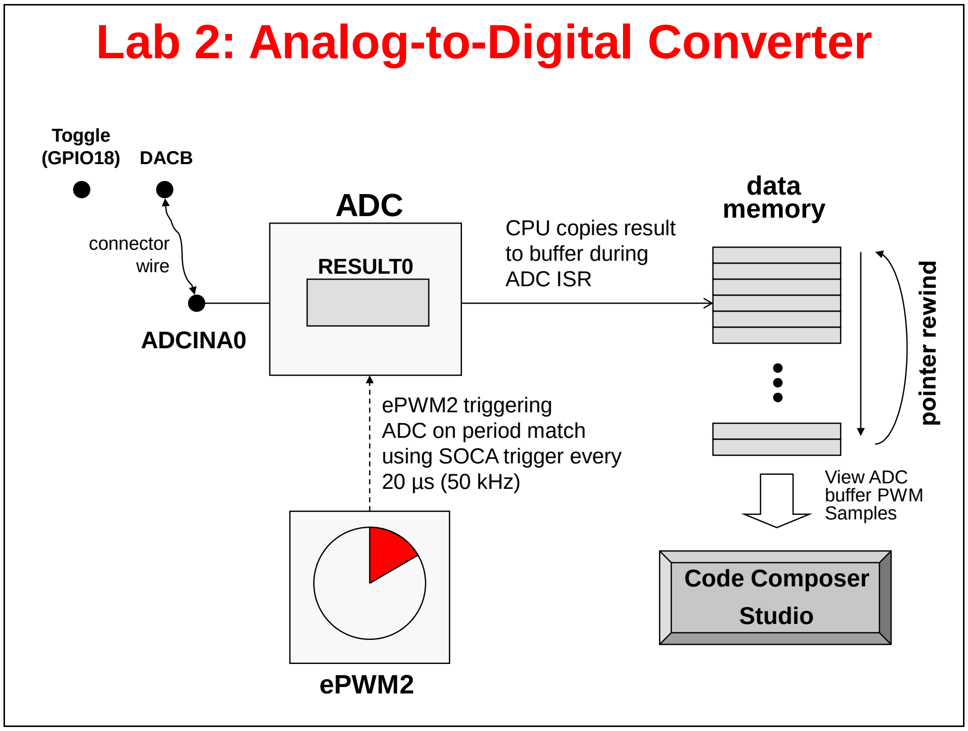


Lab 2: Analog-to-Digital Converter

➢ Objective

The objective of this lab exercise is to demonstrate and become familiar with the operation of the on-chip analog-to-digital converter. In this lab exercise all the code will run on CPU1 (CPU2 will not be used). The ADC will be configured to sample a single input channel at a 50 kHz sampling rate. We will use ePWM2A to automatically trigger the SOCA signal at the desired sampling rate (ePWM period match CTR=PRD SOC). The ADC end-of-conversion interrupt will be used to prompt CPU1 to copy the results of the ADC conversion into a circular memory buffer (AdcaResults).

In order to generate an interesting input signal, the code also alternately toggles a GPIO pin high and low in the ADC interrupt service routine. This pin will be connected to the ADC input pin by means of a jumper wire. Using Code Composer Studio the sampled data will be viewed in memory and displayed with the graphing feature. We will then configure one of the internal DACs to generate a fixed frequency sine wave with programmable offset and measure this signal in the same way.


Lab 2: Analog-to-Digital Converter

➢ Procedure

Open the Project

1. A project named `Lab2_cpu01` has been created for this lab. Open the project by clicking on `Project` → `Import CCS Projects`. The "Import CCS Eclipse Projects" window will open then click `Browse…` next to the "Select search-directory" box. Navigate to: `C:\F2837xD\Labs\Lab2\cpu01` and click `OK`. Then click `Finish` to import the project. All build options have been configured the same as the previous lab.

   Click on the project name in the Project Explorer window to set the project active. Then click on the plus sign (+) to the left of Lab2_cpu01 to expand the file list.

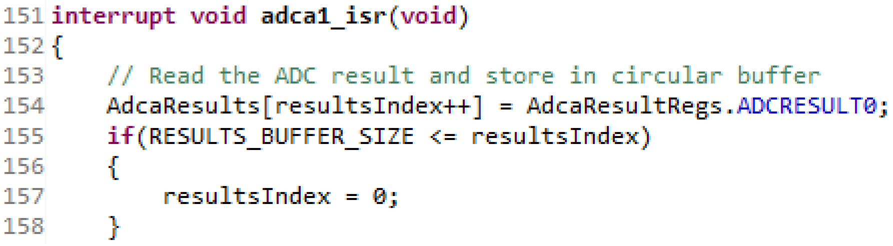
Inspect the Project

2. Open and inspect `Lab2_cpu01.c`. The initialization code immediately following main() is similar to that used in lab 1. Notice the inclusion of the following four functions which set up the ADC, PWM and DAC. The last function configures the ADC to be triggered by an EPWM event and to generate a CPU interrupt.

```
ConfigureADC()
ConfigureEPWM()
ConfigureDAC()
SetupADCEpwm()
```

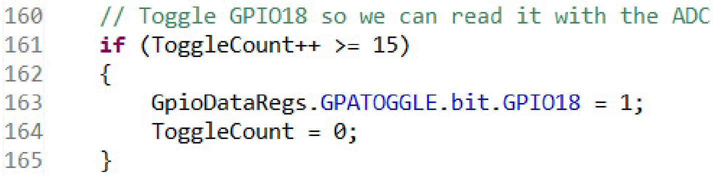
The code for these functions is located further down in the same file.

3. At the bottom of the file is the Interrupt Service Routine (ISR) `adca1_isr`. This is triggered by an end-of-conversion event from ADC-A. The ISR code reads and stores the newest ADC result in the buffer `AdcaResults`. The variable `resultsIndex` keeps track of the last entry in the buffer and wraps around to the first entry when the end of the buffer is reached. This implements a circular buffer to store a continuous stream of incoming ADC data.



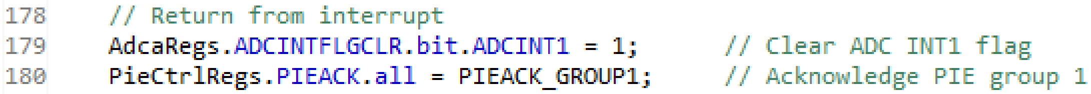
```
interrupt void adca1_isr(void)
{
    // Read the ADC result and store in circular buffer
    AdcaResults[resultsIndex++] = AdcaResultRegs.ADCRESULT0;
    if(RESULTS_BUFFER_SIZE <= resultsIndex)
    {
        resultsIndex = 0;
    }
```

Also, the ISR contains code to toggle the GPIO18 pin which be measured with the ADC. This pin toggles between 0V and +3.3V every sixteen interrupts. If everything works as expected, the `AdcaResults` buffer should contain a repeating sequence of 16 readings of close to 0x0000 followed by another 16 readings close to 0x0FFF (i.e. full scale).



```
    // Toggle GPIO18 so we can read it with the ADC
    if (ToggleCount++ >= 15)
    {
        GpioDataRegs.GPATOGGLE.bit.GPIO18 = 1;
        ToggleCount = 0;
    }
```

The last two lines in the ISR clear the interrupt flag at the ADC and acknowledge the PIE level group interrupt so that the next ADC EOC event will trigger an interrupt.

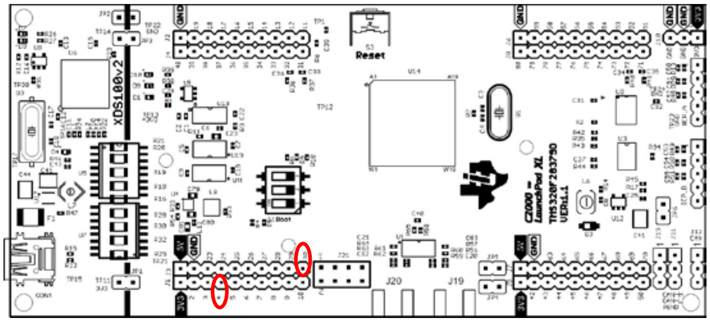


```
    // Return from interrupt
    AdcaRegs.ADCINTFLGCLR.bit.ADCINT1 = 1;      // Clear ADC INT1 flag
    PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;     // Acknowledge PIE group 1
```

Jumper Wire Connection

In order to have a meaningful input signal to the ADC, a jumper wire will connect the ADC input pin to the GPIO18 pin. This pin has been set up in the ADC ISR to alternately toggle between 0V and +3.3V.

4. On the LaunchPad locate connector J3, pin #30 (ADCINA0). Connect one end of the jumper wire to this pin, and the other end of the jumper wire to the adjacent connector J1, pin #4 (GPIO18). Refer to the following diagram for the pins that need to be connected using the jumper wire.



Build and Load the Project

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

6. Click the “Debug” button (green bug). A Launching Debug Session window will open. Select only CPU1 to load the program on, and then click OK. The “CCS Debug” perspective view should open, the program will load automatically, and you should now be at the start of main().

7. 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 (runtime support library) and stopped at the start of main(). CCS did not do a device reset, and as a result the bootloader was bypassed.

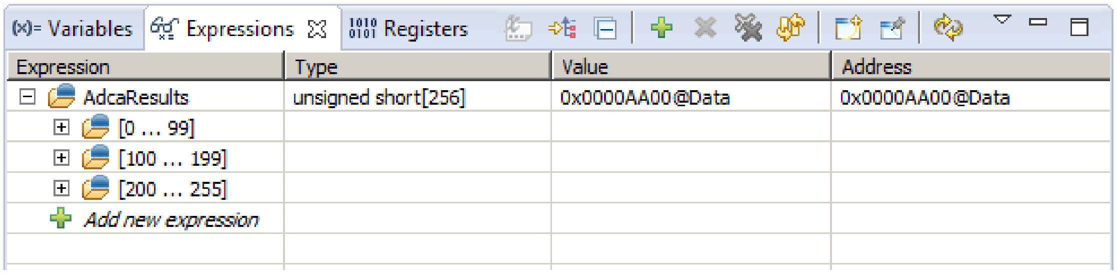
   In the event the device undergoes a reset, the proper boot mode needs to be set. Therefore, we must configure the device by loading values into EMU_KEY and EMU BMODE so the bootloader will jump to “M0 SARAM” at address 0x000000. Set the bootloader mode using the menu bar by clicking:

   `Scripts → EMU Boot Mode Select → EMU_BOOT_SARAM`

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

View the ADC Results

8. Click the “Expressions” tab near the top of the CCS window. In the empty box in the “Expression” column (click on the text *“Add new expression”*), type **AdcaResults** and then enter. This will add the ADC results buffer to the watch window. Click on the “+” symbol to the left of the buffer name. Notice the buffer is divided into three separate groups of 100 elements or less. Expand the first of these so we can inspect the ADC results later.



Expression	Type	Value	Address
⊟ AdcaResults	unsigned short[256]	0x0000AA00@Data	0x0000AA00@Data
⊞ [0 … 99]			
⊞ [100 … 199]			
⊞ [200 … 255]			
Add new expression			

Run the Code

9. Run the code by using the “Resume” button on the toolbar, or by using `Run` → `Resume` on the menu bar (or F8 key). LED D10 should be blinking at a period of approximately 1 second.

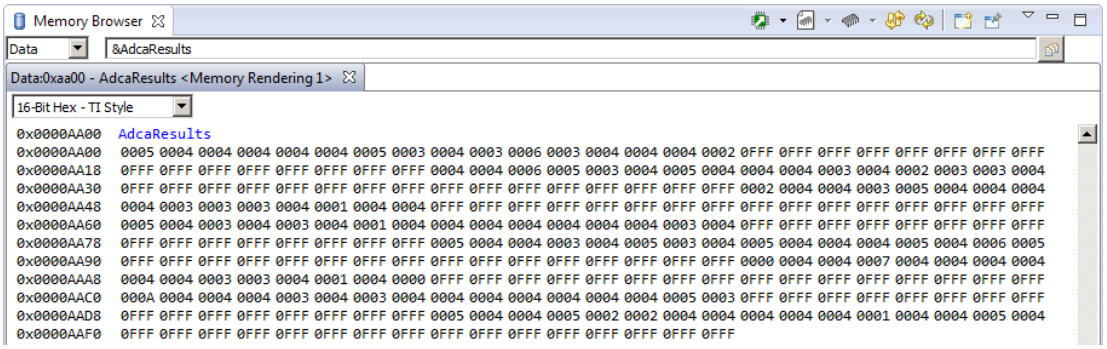
10. Halt the code after a few seconds by using the “Suspend” button on the toolbar, or by using `Run` → `Suspend` on the menu bar (or Alt-F8 key).

11. Observe the contents of the AdcaResults buffer in the Expressions window. If the code is running as expected, you should see a series of sixteen readings close to 0, followed by another series close to full scale (4095).

View the ADC Results Buffer in Memory

12. Open a memory browser by clicking `View` → `Memory Browser`.

13. In the box marked “Enter location here”, type **&AdcaResults** and then enter. The memory browser will display the contents of the ADC results buffer. The browser should contain a series of entries of 0x0FFF and 0x0000, indicating the data is from the toggling GPIO pin.



Data &AdcaResults

Data:0xaa00 - AdcaResults <Memory Rendering 1>

16-Bit Hex - TI Style

```
0x0000AA00  AdcaResults
0x0000AA00   0005 0004 0004 0004 0004 0004 0005 0003 0004 0003 0006 0003 0004 0004 0004 0002 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF
0x0000AA18   0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0004 0004 0006 0005 0003 0004 0005 0004 0004 0004 0003 0004 0002 0003 0003 0004
0x0000AA30   0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0002 0004 0004 0003 0005 0004 0004 0004
0x0000AA48   0004 0003 0003 0003 0004 0001 0004 0004 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF
0x0000AA60   0005 0004 0003 0004 0003 0004 0001 0004 0004 0004 0004 0004 0004 0004 0003 0004 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF
0x0000AA78   0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0005 0004 0004 0003 0004 0005 0003 0004 0005 0004 0004 0004 0005 0004 0006 0005
0x0000AA90   0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0000 0004 0004 0007 0004 0004 0004 0004
0x0000AAA8   0004 0004 0003 0003 0004 0001 0004 0000 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF
0x0000AAC0   000A 0004 0004 0004 0003 0004 0003 0004 0004 0004 0004 0004 0004 0004 0005 0003 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF
0x0000AAD8   0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0005 0004 0004 0005 0002 0002 0004 0004 0004 0004 0004 0001 0004 0004 0005 0004
0x0000AAF0   0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF 0FFF
```

Graph the ADC Data

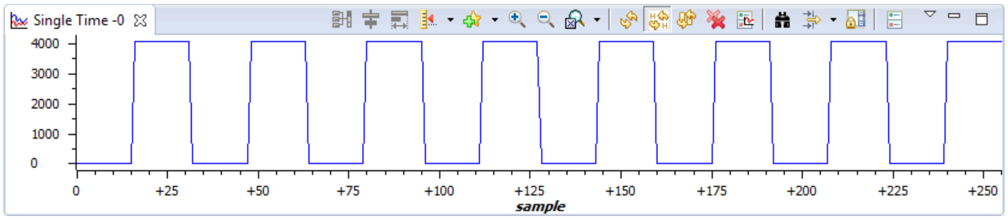
CCS can display the ADC results in the form of a time graph. This provides a clear visualization of the signal at the ADC input.

14. Open and set up a graph to plot a 256-point window of the ADC results buffer. Click: `Tools` → `Graph` → `Single Time` and set the following values:

Acquisition Buffer Size	256
DSP Data Type	16-bit unsigned integer
Sampling Rate (Hz)	50000
Start Address	AdcaResults
Display Data Size	256
Time Display Unit	sample

Select `OK` to save the graph options.

The graph view should look like:



Using Real-Time Emulation Mode

Real-time emulation is a special emulation feature that allows the windows within Code Composer Studio to 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.

15. 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.

16. Enable the Memory Browser and Expressions window for continuous refresh using the same procedure as the previous step.

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

    `Scripts → Realtime Emulation Control → Run_Realtime_with_Reset`

18. *Carefully* remove and replace the connector wire from the ADC input. Are the values updating as expected? The ADC results should be zero when the jumper wire is removed.

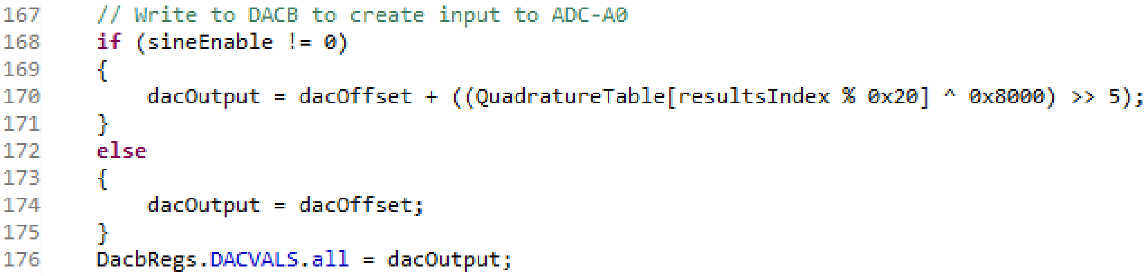
19. Fully halt the CPU in real-time mode. Click:

    `Scripts → Realtime Emulation Control → Full_Halt`

Sampling a Sine Wave

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

20. Notice the following code lines in the `adca1_isr()`in Lab2_cpu01.c source file:



```
// Write to DACB to create input to ADC-A0
if (sineEnable != 0)
{
    dacOutput = dacOffset + ((QuadratureTable[resultsIndex % 0x20] ^ 0x8000) >> 5);
}
else
{
    dacOutput = dacOffset;
}
DacbRegs.DACVALS.all = dacOutput;
```

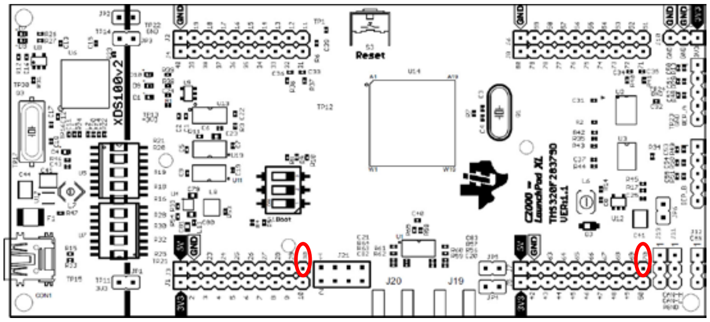
The variable `dacOffset` allows the user to adjust the DC output from DAC-B from an Expressions window in CCS. The variable sineEnable is a switch which adds a fixed frequency sine wave to the DAC offset. The sine wave is generated using a 32-point look-up table contained in the source file `sinetab.c`. We will plot the sine wave in a graph window while manually adjusting the offset.

21. Open and inspect `sinetab.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 40 signed integer points which represent five quadrants of sinusoidal data. The first 32 points are a complete cycle. In the source code we need to sequentially access each of the first 32 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 DAC-B.

22. In the Expressions window collapse the AdcaResults buffer variable by clicking on the "-" symbol to the left of the variable name. Then add the following variables to the Expressions window:

    - `sineEnable`
    - `dacOffset`

23. Remove the jumper wire from connector J1, pin #4 (GPIO18) and connect it to connector J7, pin #70 (DACOUTB). Refer to the following diagram for the pins that need to be connected using the jumper wire.

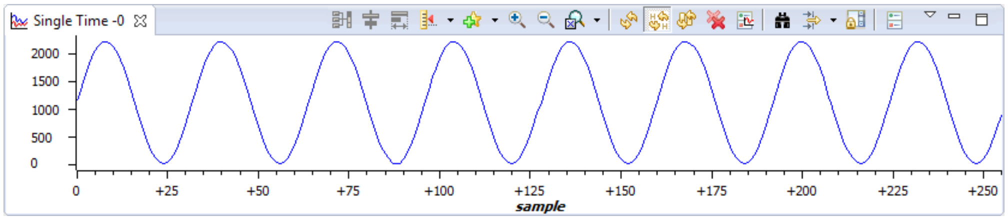


24. Run the code (real-time mode) using the Script function: `Scripts` → `Realtime Emulation Control` → `Run_Realtime_with_Reset`

25. 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).

26. Enable the sine generator by changing the variable `sineEnable` in the Expressions window to 1.

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



28. 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).

29. Fully halt the code (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 CCS to the “CCS Edit” perspective” view.

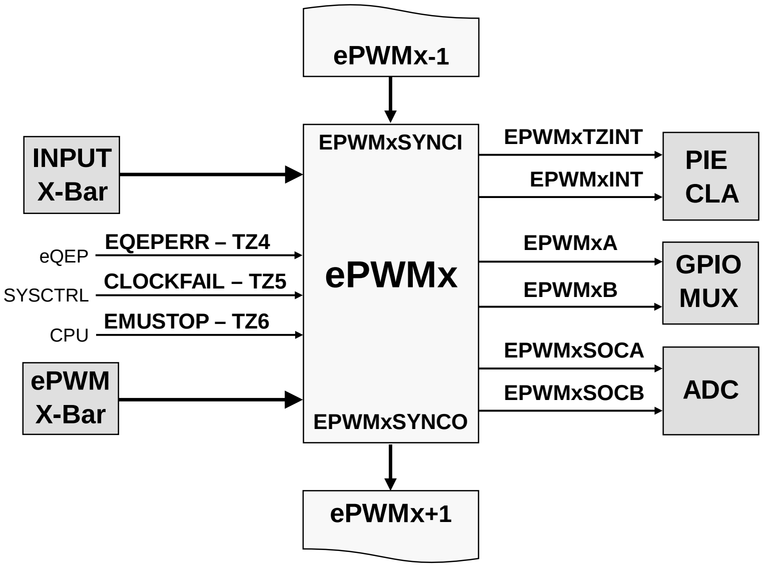
31. Next, close the project by right-clicking on Lab2_cpu01 in the Project Explorer window and select `Close Project`.

End of Exercise

Control Peripherals

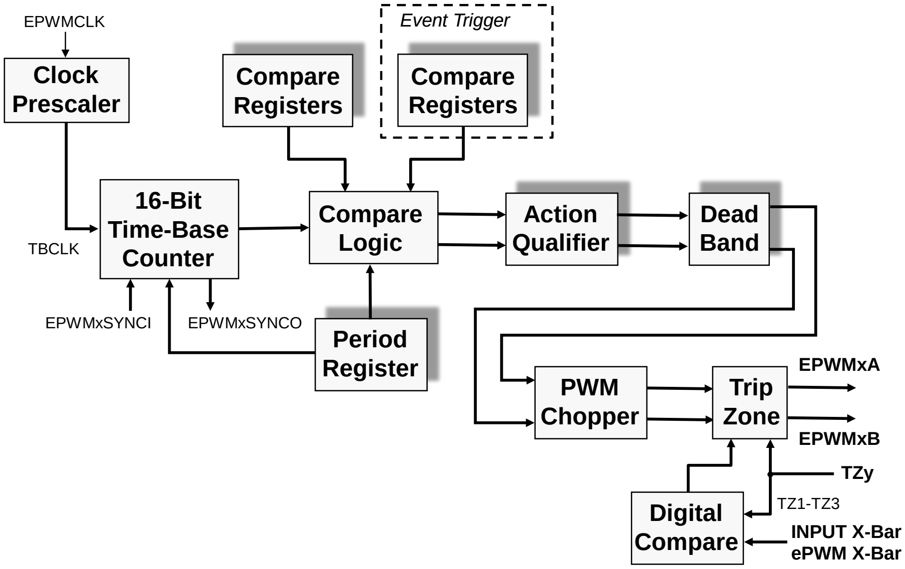
ePWM Module Signals and Connections

ePWM Module Signals and Connections



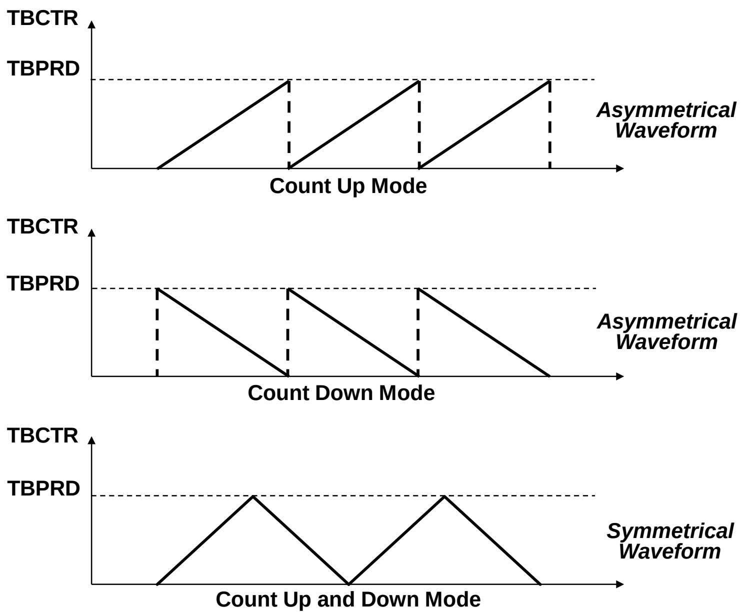
ePWM Block Diagram

ePWM Block Diagram

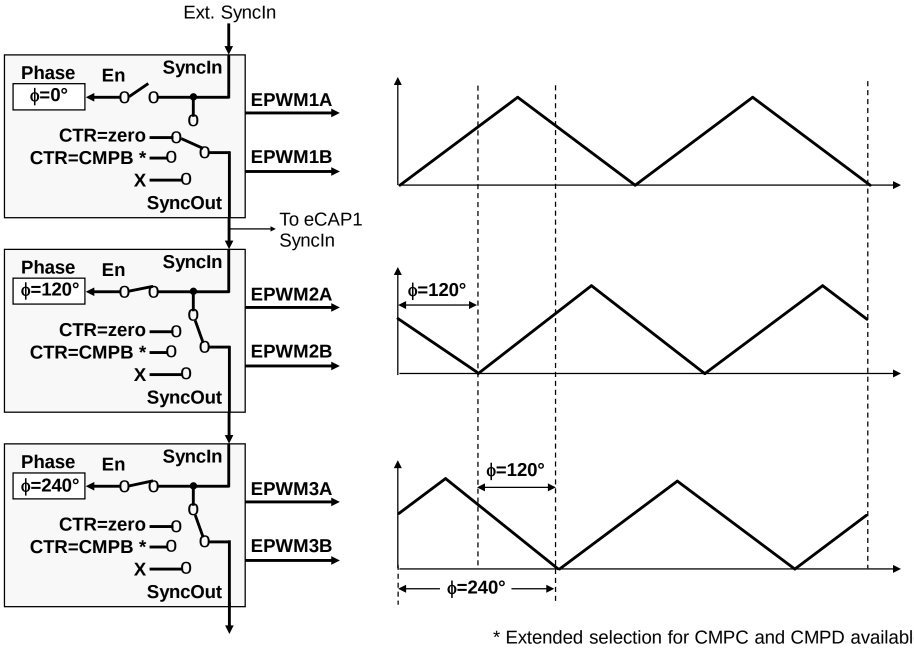


ePWM Time-Base Sub-Module

ePWM Time-Base Count Modes



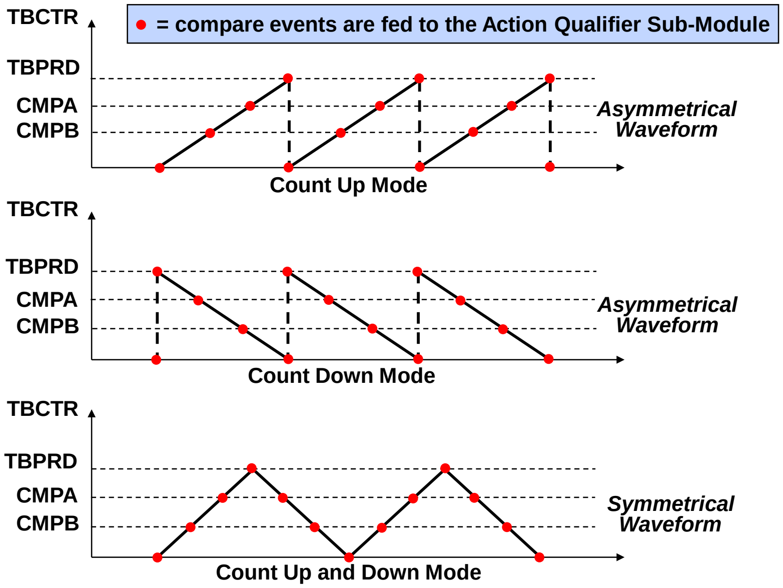
ePWM Phase Synchronization



* Extended selection for CMPC and CMPD available

ePWM Compare Sub-Module

ePWM Compare Event Waveforms



CMPC and CMPD available for use as event triggers

ePWM Action Qualifier Sub-Module

ePWM Action Qualifier Actions

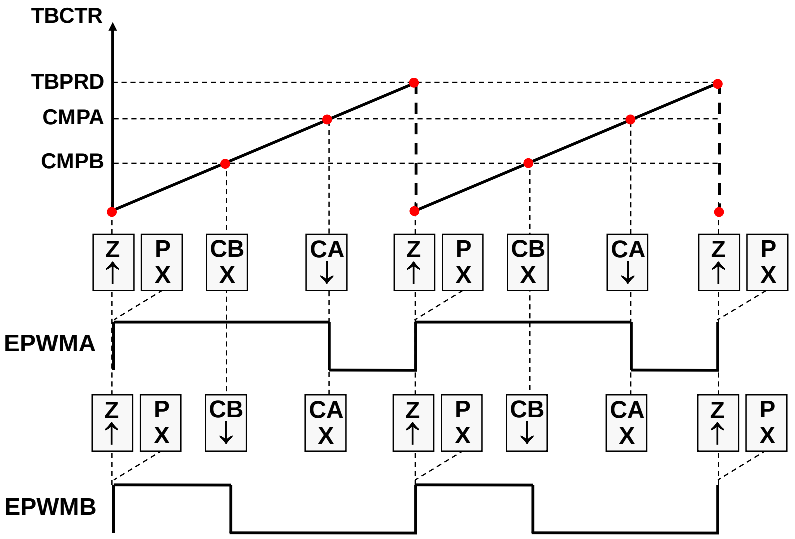
for EPWMA and EPWMB

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

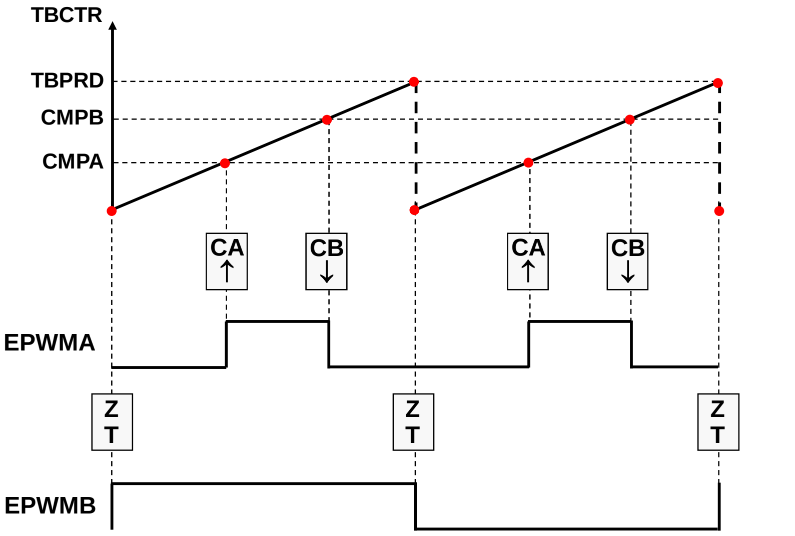
ePWM Count Up Asymmetric Waveform

with Independent Modulation on EPWMA / B



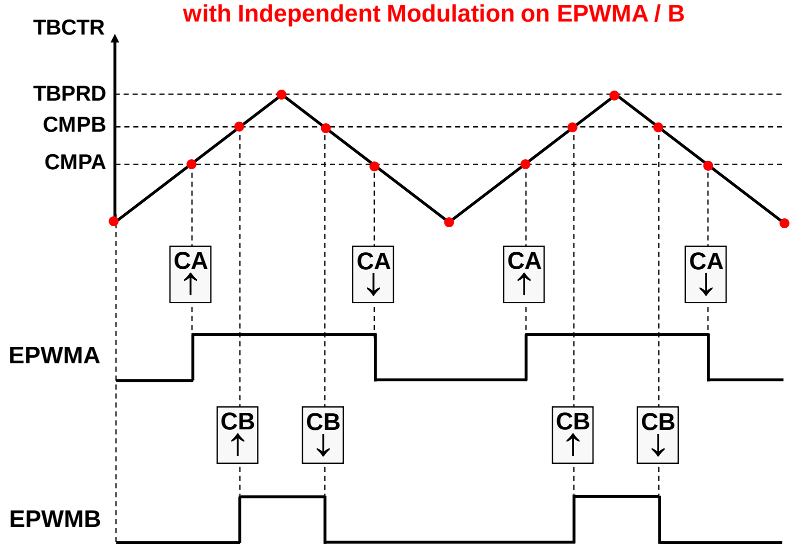
ePWM Count Up Asymmetric Waveform

with Independent Modulation on EPWMA



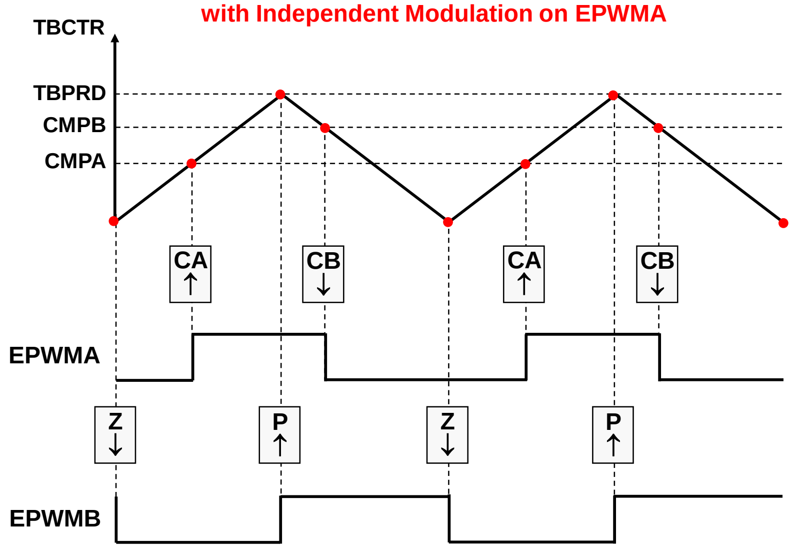
ePWM Count Up-Down Symmetric Waveform

with Independent Modulation on EPWMA / B



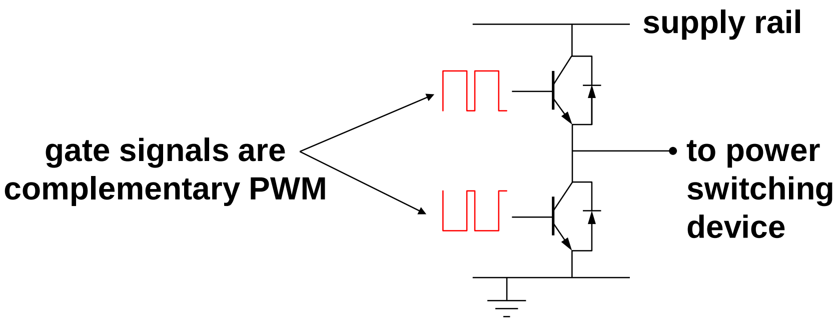
ePWM Count Up-Down Symmetric Waveform

with Independent Modulation on EPWMA



ePWM Dead-Band Sub-Module

Motivation for Dead-Band

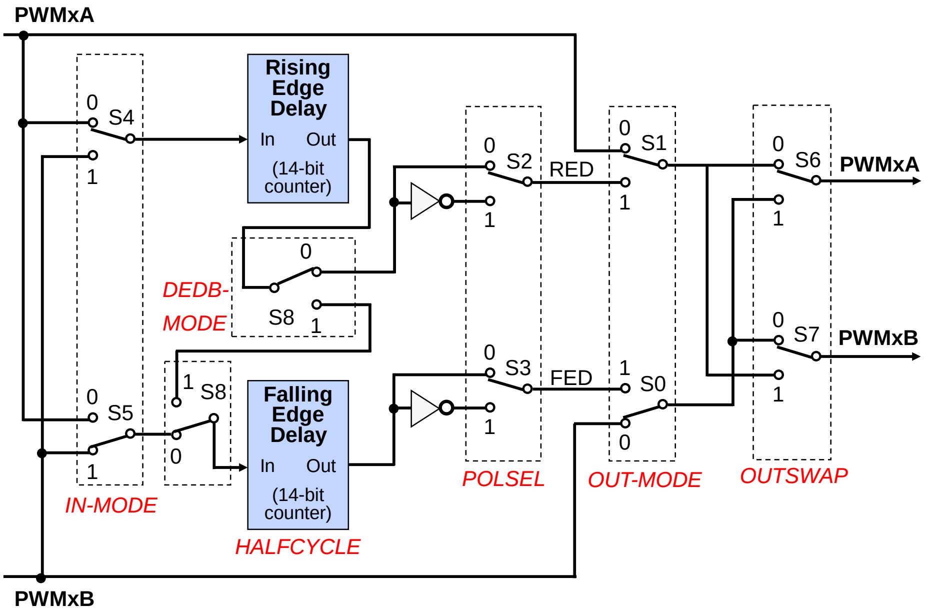

supply rail

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!

ePWM Dead-Band Block Diagram


PWMxA

Rising Edge Delay

In Out

(14-bit counter)

Falling Edge Delay

In Out

(14-bit counter)

S4 S5 S8 S2 S3 S1 S0 S6 S7

RED

FED

DEDB-MODE

IN-MODE

HALFCYCLE

POLSEL

OUT-MODE

OUTSWAP

PWMxA

PWMxB

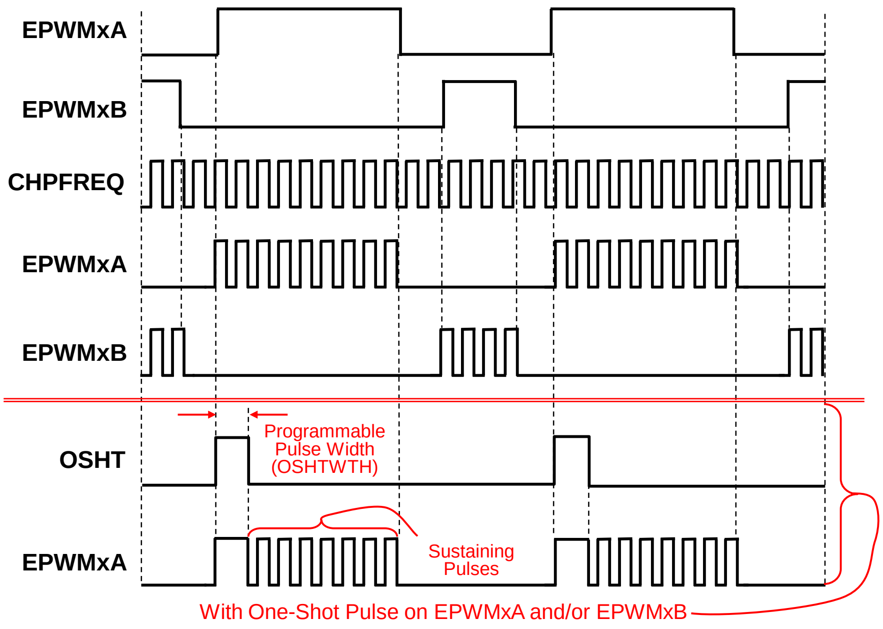
PWMxB

ePWM Chopper Sub-Module

Purpose of the PWM Chopper

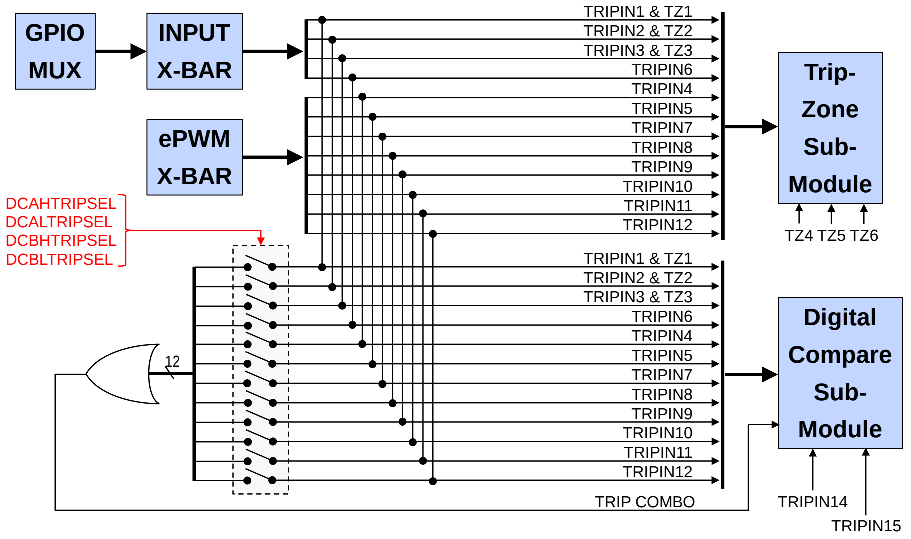
- Allows a high frequency carrier signal to modulate the PWM waveform generated by the Action Qualifier and Dead-Band modules
- Used with pulse transformer-based gate drivers to control power switching elements

ePWM Chopper Waveform


EPWMxA
EPWMxB
CHPFREQ
EPWMxA
EPWMxB
OSHT
Programmable Pulse Width (OSHTWTH)
EPWMxA
Sustaining Pulses
With One-Shot Pulse on EPWMxA and/or EPWMxB

ePWM Trip-Zone and Digital Compare Sub-Module

Trip-Zone and Digital Compare Inputs



GPIO MUX

INPUT X-BAR

ePWM X-BAR

DCAHTRIPSEL
DCALTRIPSEL
DCBHTRIPSEL
DCBLTRIPSEL

12

TRIPIN1 & TZ1
TRIPIN2 & TZ2
TRIPIN3 & TZ3
TRIPIN6
TRIPIN4
TRIPIN5
TRIPIN7
TRIPIN8
TRIPIN9
TRIPIN10
TRIPIN11
TRIPIN12

Trip-Zone Sub-Module

TZ4 TZ5 TZ6

TRIPIN1 & TZ1
TRIPIN2 & TZ2
TRIPIN3 & TZ3
TRIPIN6
TRIPIN4
TRIPIN5
TRIPIN7
TRIPIN8
TRIPIN9
TRIPIN10
TRIPIN11
TRIPIN12

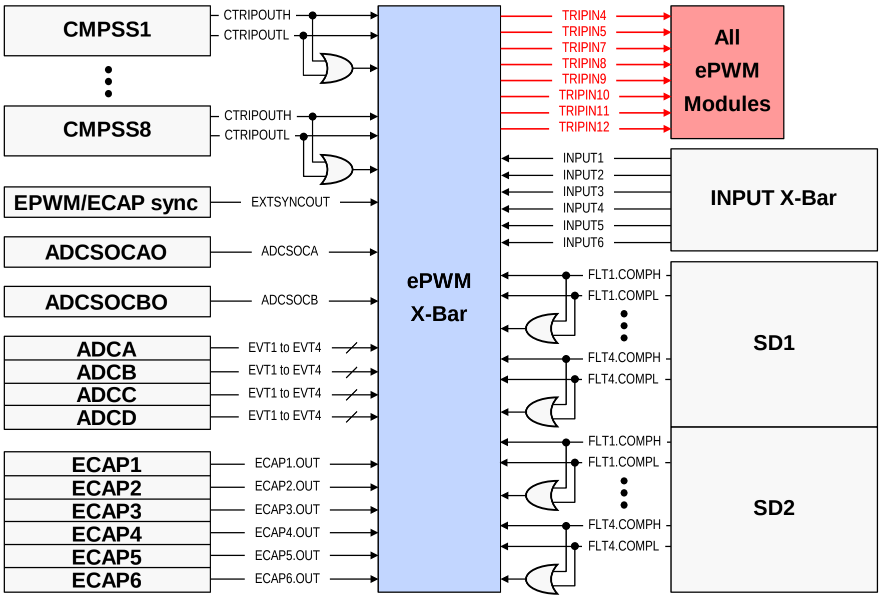
Digital Compare Sub-Module

TRIP COMBO

TRIPIN14

TRIPIN15

ePWM X-Bar



CMPSS1
CTRIPOUTH
CTRIPOUTL

CMPSS8
CTRIPOUTH
CTRIPOUTL

EPWM/ECAP sync
EXTSYNCOUT

ADCSOCAO
ADCSOCA

ADCSOCBO
ADCSOCB

ADCA
ADCB
ADCC
ADCD
EVT1 to EVT4
EVT1 to EVT4
EVT1 to EVT4
EVT1 to EVT4

ECAP1
ECAP2
ECAP3
ECAP4
ECAP5
ECAP6
ECAP1.OUT
ECAP2.OUT
ECAP3.OUT
ECAP4.OUT
ECAP5.OUT
ECAP6.OUT

ePWM X-Bar

TRIPIN4
TRIPIN5
TRIPIN7
TRIPIN8
TRIPIN9
TRIPIN10
TRIPIN11
TRIPIN12

All ePWM Modules

INPUT1
INPUT2
INPUT3
INPUT4
INPUT5
INPUT6

INPUT X-Bar

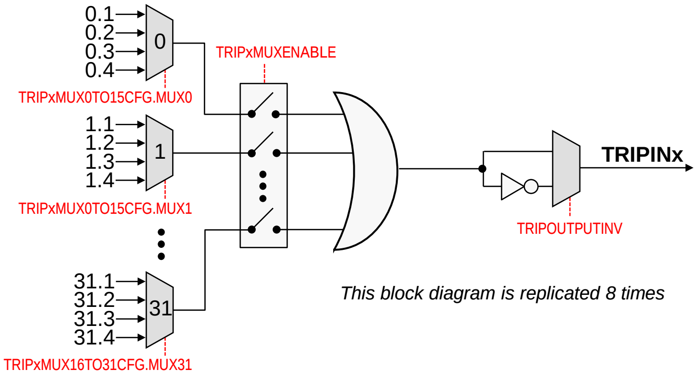
FLT1.COMPH
FLT1.COMPL
FLT4.COMPH
FLT4.COMPL

SD1

FLT1.COMPH
FLT1.COMPL
FLT4.COMPH
FLT4.COMPL

SD2

ePWM X-Bar Architecture



This block diagram is replicated 8 times

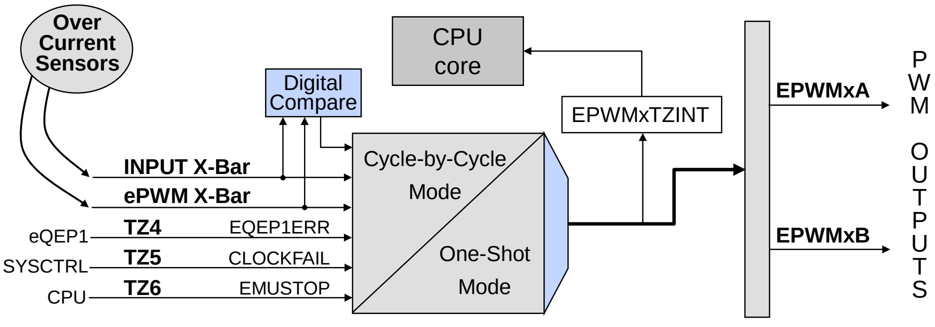
MUX	0	1	2	3
0	CMPSS1.CTRIPOUTH	CMPSS1.CTRIPH_OR_CTRIPL	ADCAEVT1	ECAP1.OUT
1	CMPSS1.CTRIPOUTL	INPUTXBAR1		ADCCEVT1
2	CMPSS2.CTRIPOUTH	CMPSS2.CTRIPH_OR_CTRIPL	ADCAEVT2	ECAP2.OUT
3	CMPSS2.CTRIPOUTL	INPUTXBAR2		ADCCEVT2
4	CMPSS3.CTRIPOUTH	CMPSS3.CTRIPH_OR_CTRIPL	ADCAEVT3	ECAP3.OUT
5	CMPSS3.CTRIPOUTL	INPUTXBAR3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

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**

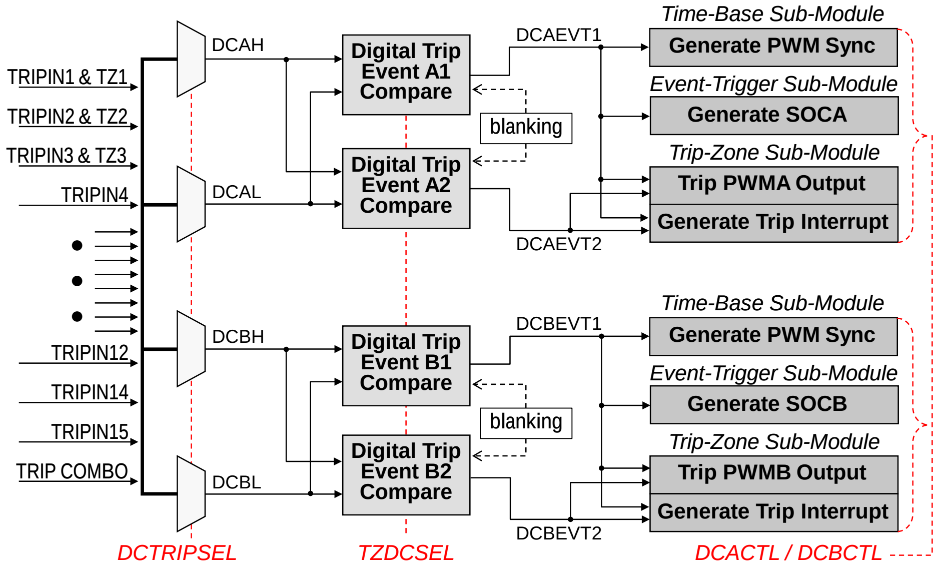
  #2) cycle-by-cycle trip for current limiting operation

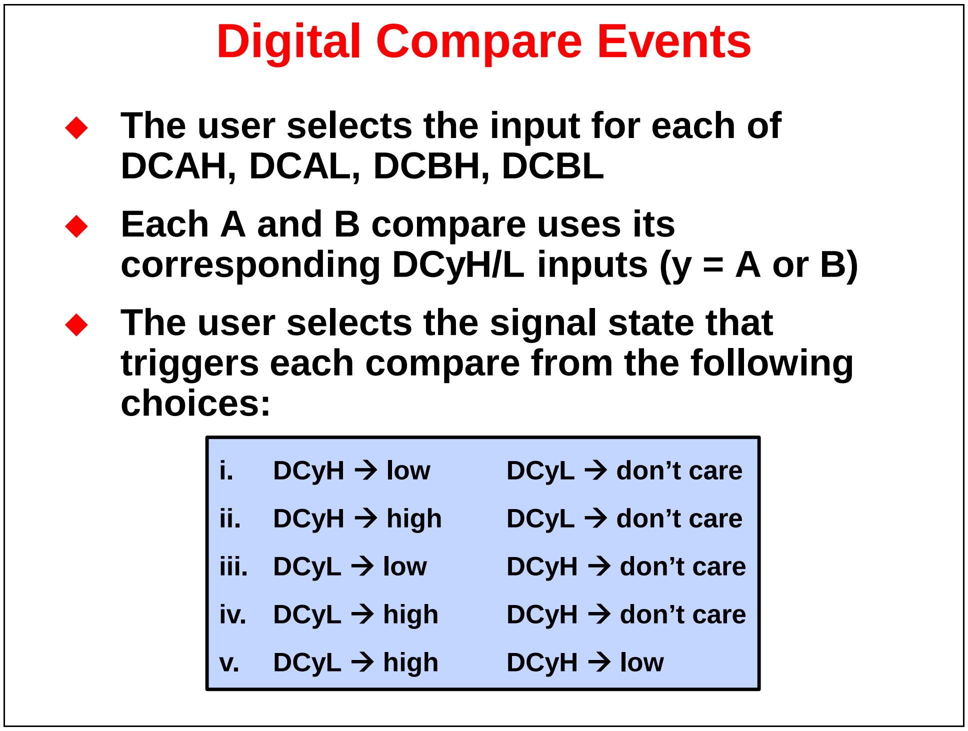


Purpose of the Digital Compare Sub-Module

- **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



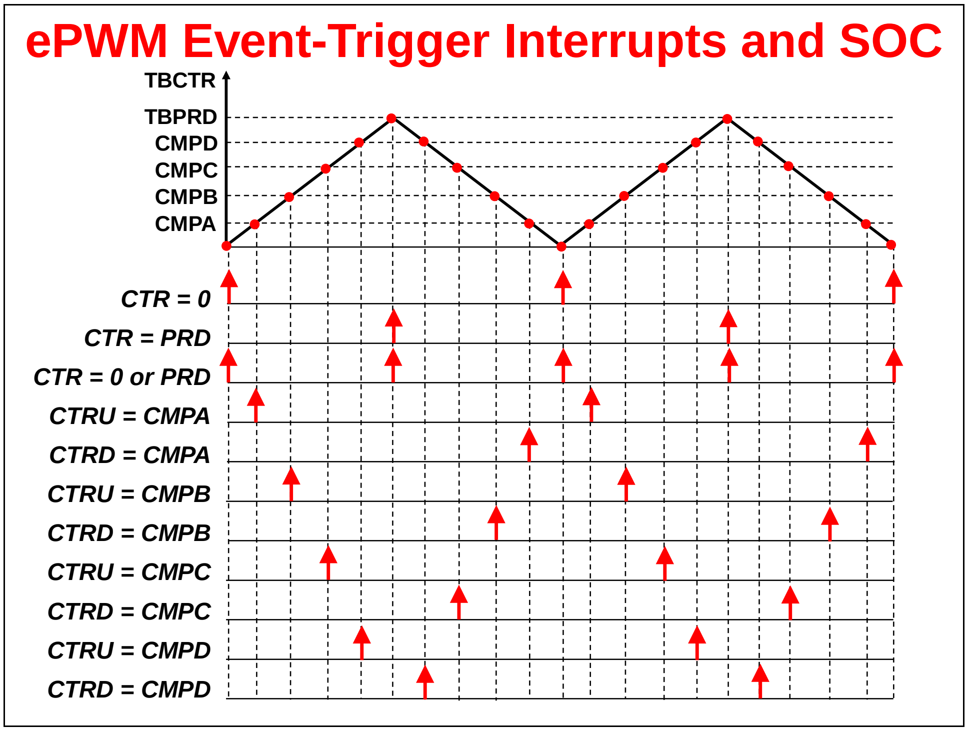


Digital Compare Events

- 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

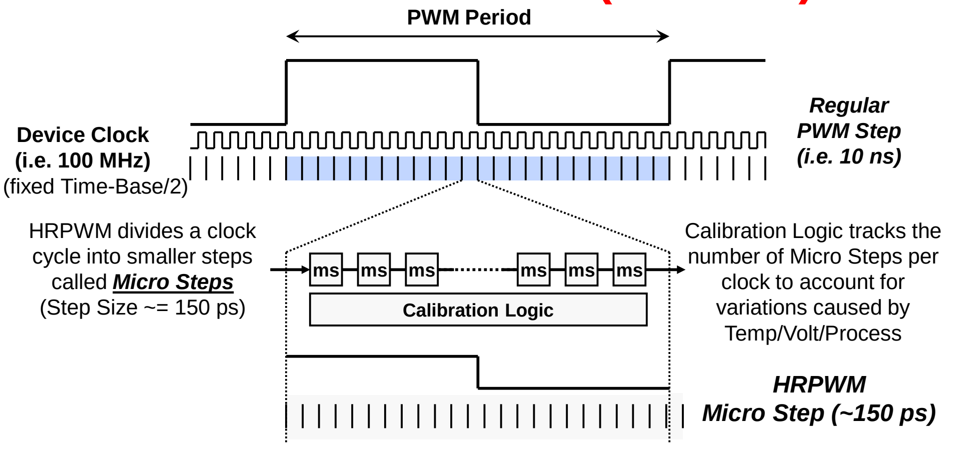
ePWM Event-Trigger Sub-Module



ePWM Event-Trigger Interrupts and SOC

Hi-Resolution PWM (HRPWM)

Hi-Resolution PWM (HRPWM)



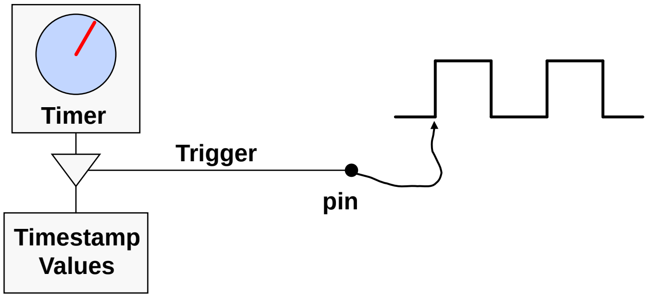
HRPWM divides a clock cycle into smaller steps called ***Micro Steps*** (Step Size ~= 150 ps)

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

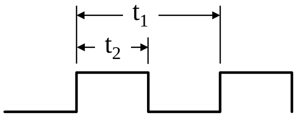
- Significantly increases the resolution of conventionally derived digital PWM
- Uses 8-bit extensions to Compare registers (CMPxHR), Period register (TBPRDHR) and Phase register (TBPHSHR) for edge positioning control
- Typically used when PWM resolution falls below ~9-10 bits which occurs at frequencies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

Capture Module (eCAP)

Capture Module (eCAP)

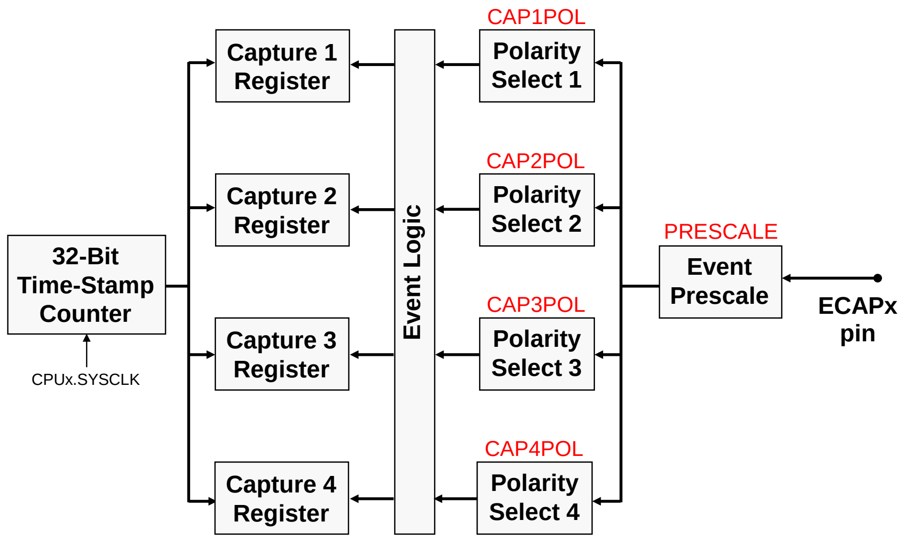


- **The eCAP module timestamps transitions on a capture input pin**
  - **Can be used to measure the time width of a pulse**

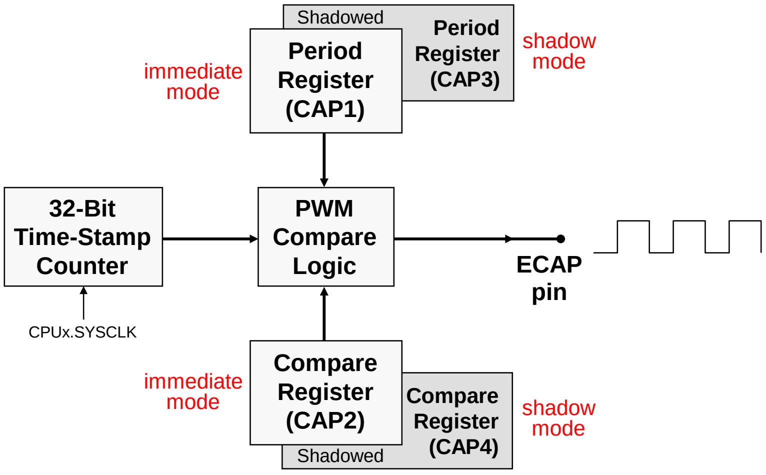


- **Auxiliary PWM generation**

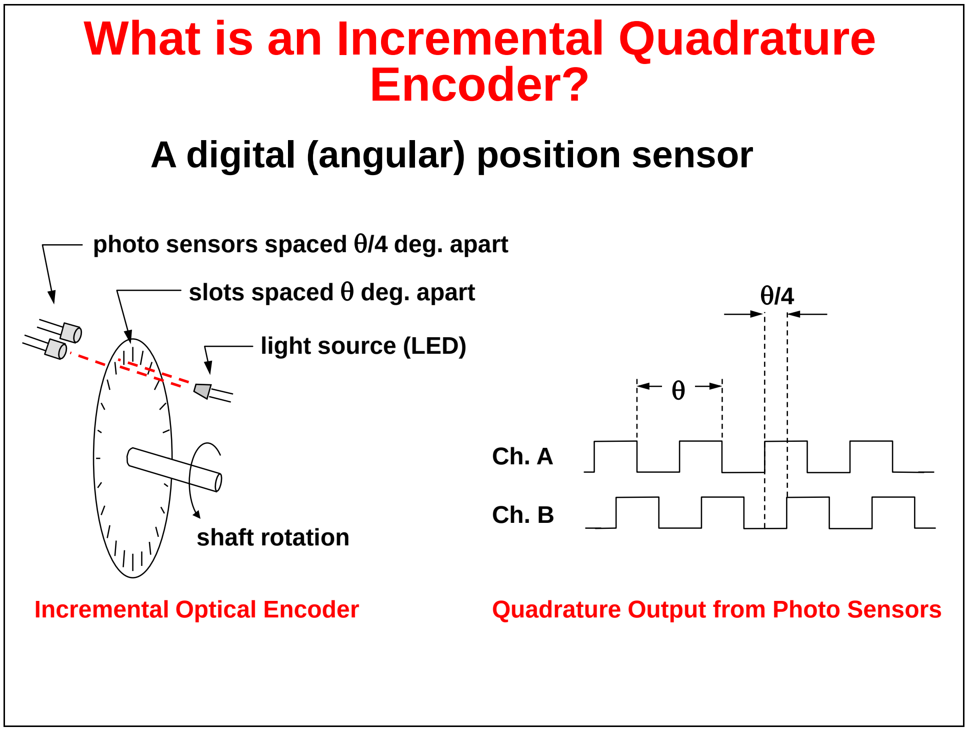
eCAP Module Block Diagram – Capture Mode


CAP1POL
CAP2POL
CAP3POL
CAP4POL
PRESCALE
Capture 1 Register
Capture 2 Register
Capture 3 Register
Capture 4 Register
Event Logic
Polarity Select 1
Polarity Select 2
Polarity Select 3
Polarity Select 4
Event Prescale
ECAPx pin
32-Bit Time-Stamp Counter
CPUx.SYSCLK

eCAP Module Block Diagram – APWM Mode


Shadowed
Period Register (CAP3)
shadow mode
immediate mode
Period Register (CAP1)
32-Bit Time-Stamp Counter
CPUx.SYSCLK
PWM Compare Logic
ECAP pin
immediate mode
Compare Register (CAP2)
Compare Register (CAP4)
Shadowed
shadow mode

Quadrature Encoder Pulse Module (eQEP)



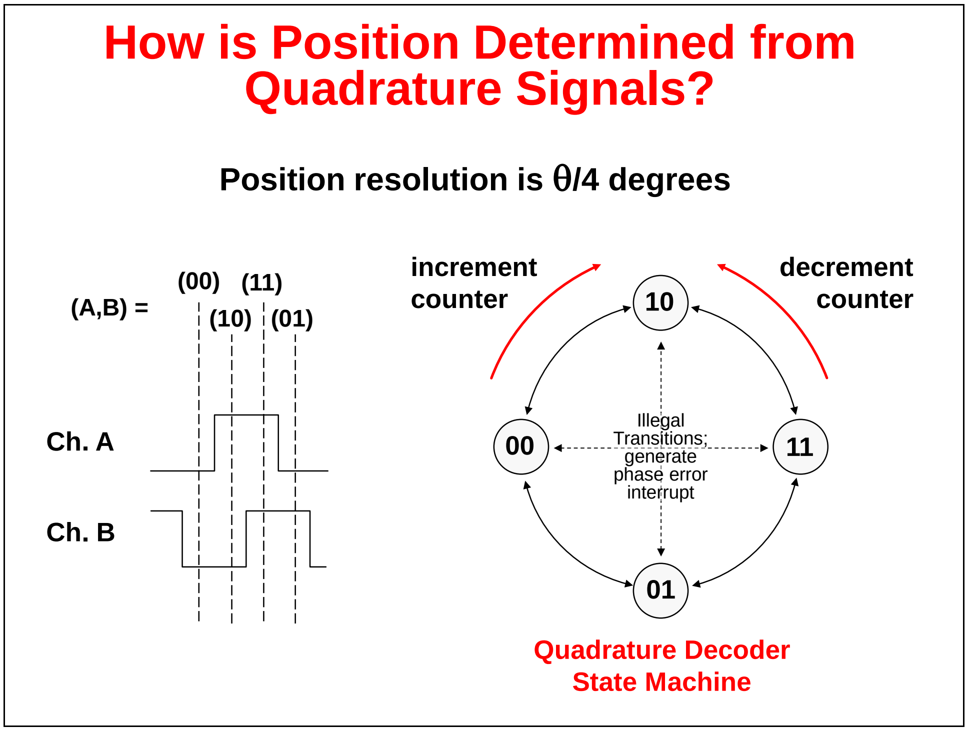
What is an Incremental Quadrature Encoder?

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

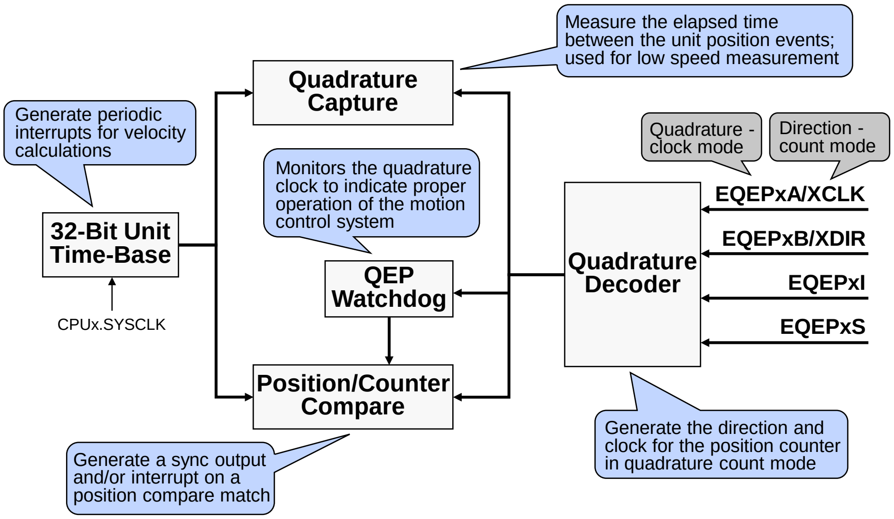


How is Position Determined from Quadrature Signals?

Position resolution is θ/4 degrees

Quadrature Decoder State Machine

eQEP Module Block Diagram



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

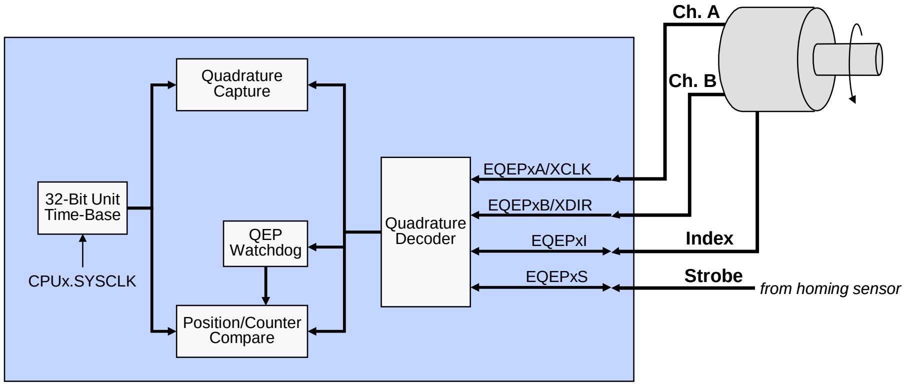
Generate periodic interrupts for velocity calculations

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

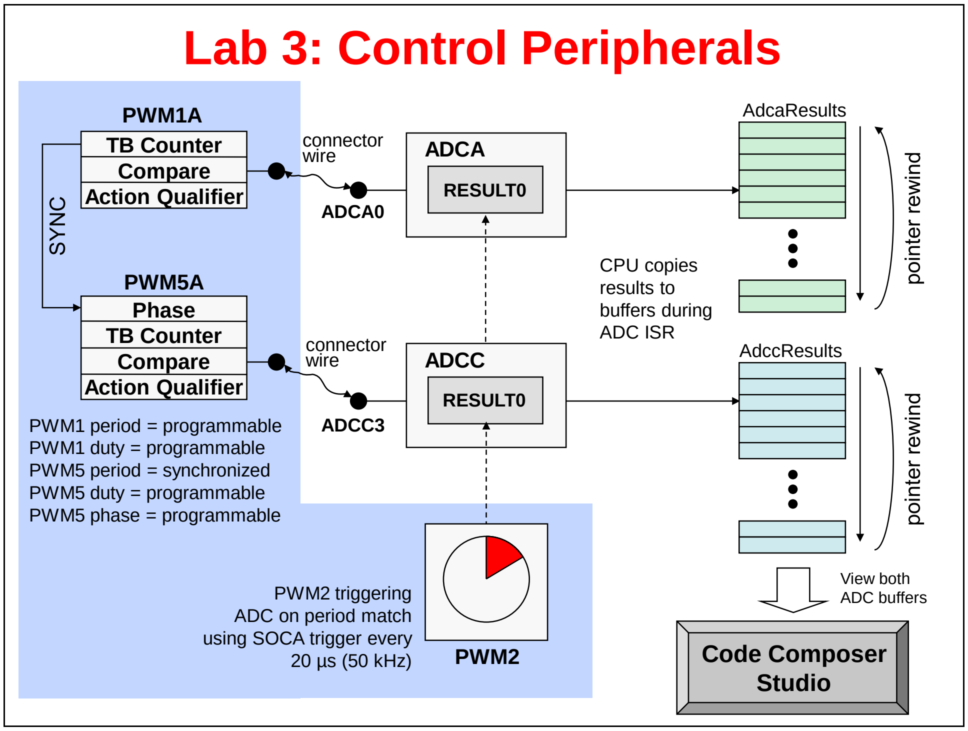
eQEP Module Connections



Lab 3: Control Peripherals

➢ Objective

The objective of this lab exercise is to demonstrate and become familiar with the operation of the PWM modules. In this lab exercise all the code will run on CPU1 (CPU2 will not be used). PWM1A will be configured to generate a PWM waveform with programmable frequency and duty cycle. PWM5A will be phase locked to PWM1A and will share the same period, however its duty cycle and phase offset are also programmable. PWM2 will be configured to generate a fixed 50 kHz sample trigger for ADC-A and ADC-C. These ADCs will sample the two PWM waveforms and the results will be stored in two circular buffers in data memory. We will open two time graph windows in CCS to observe the contents of these buffers while the PWM variables are adjusted.



Lab 3: Control Peripherals

PWM1 period = programmable
PWM1 duty = programmable
PWM5 period = synchronized
PWM5 duty = programmable
PWM5 phase = programmable

CPU copies results to buffers during ADC ISR

PWM2 triggering ADC on period match using SOCA trigger every 20 µs (50 kHz)

➢ Procedure

Open the Project

1. A project named `Lab3_cpu01` has been created for this lab. Open the project by clicking on `Project` → `Import CCS Projects`. The "Import CCS Eclipse Projects" window will open then click `Browse…` next to the "Select search-directory" box. Navigate to: `C:\F2837xD\Labs\Lab3\cpu01` and click `OK`. Then click `Finish` to import the project. All build options have been configured the same as the previous lab.

   Click on the project name in the Project Explorer window to set the project active. Then click on the plus sign (+) to the left of Lab3_cpu01 to expand the file list.

Inspect the Project

2. Open and inspect `Lab3_cpu01.c`. The initialization code immediately following main() is similar to that used in lab 2. Notice the inclusion of the following three functions which configure the PWM modules.

```
InitEPwm1()
InitEPwm2()
InitEPwm5()
```

The code for these functions is located further down in the same file.

3. Scroll down the file and locate the function `InitEPwm1()`. Inspect the code and notice the following line:

```
EPwm1Regs.TBCTL.bit.SYNCOSEL = 1;
```

This configures the TB module to generate a SYNC output on a CTR = 0 match. Notice also the setting of the PHSEN bit in the same register. This bit disables the SYNC input to this module.

4. Scroll further down the file and locate the function `InitEPwm5()`. Inspect the code and notice the setting of the PHSEN bit in this module. This bit enables synchronization from the SYNC input from EPWM1.

At the bottom of this function are the following lines used to configure the AQ module:

```
EPwm5Regs.AQCTLA.bit.ZRO = 2;
EPwm5Regs.AQCTLA.bit.CAU = 1;
```

These define a HIGH output on a CTR = zero event and a LOW output on a compare match when counting UP. The result is an asynchronous PWM with trailing edge duty cycle modulation. ePWM1 is configured in the same way.

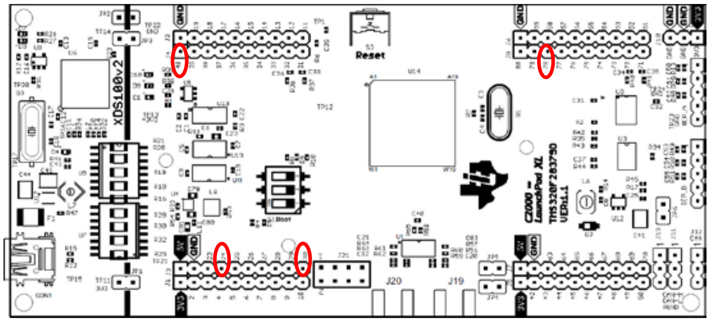
5. At the bottom of the file is the ADC Interrupt Service Routine `adca1_isr()`. As in the previous lab exercise, this interrupt is triggered by an end-of-conversion (EOC) event from ADC-A. The ISR code reads and stores the newest ADCINA0 result in the buffer `AdcaResults` and the newest ADCINC3 result in buffer `AdccResults`. Since ADC-A and ADC-C are configured similarly, their conversion time will be the same and we only need one ISR to collect both readings.

6. Notice the code near the bottom of the ISR which manipulates the variables `pretrig` and `trigger`. The ISR code has been written so that the first sample in both buffers is taken on a rising edge of PWM1A. When we view the results in a graph window, this makes it easier to see the effects of changes to PWM duty cycle and phase offset.

Jumper Wire Connection

7. We now need to connect the PWM1A output pin to the ADCINA0 input pin, and the PWM5A output pin to the ADCINC3 input pin. From Lab 2, one end of the jumper wire should still be connected to connector J3, pin #30 (ADCINA0). Connect the other end of the jumper wire to connector J4, pin #40 (PWM1A).

8. Using another jumper wire, carefully make a connection between connector J3, pin #24 (ADCINC3) and connector J8, pin #78 (PWM5A). Refer to the following diagram for the pins that need to be connected using the jumper wires.



Build and Load the Project

9. Click the “Build” button and watch the tools run in the Console window. Check for any 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, and then click `OK`. 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

11. Run the code by using the “Resume” button on the toolbar, or by using `Run` → `Resume` on the menu bar (or F8 key). LED D10 should be blinking at a period of approximately 1 second.

12. Halt the code after a few seconds by using the “Suspend” button on the toolbar,or by using `Run` → `Suspend` on the menu bar (or Alt-F8 key).

View the ADC Results

13. The Memory Browser should still be open from the previous lab exercise. If not, then open a memory browser by clicking `View` → `Memory Browser`. In the box marked “Enter location here”, type **&AdcaResults** and then enter.

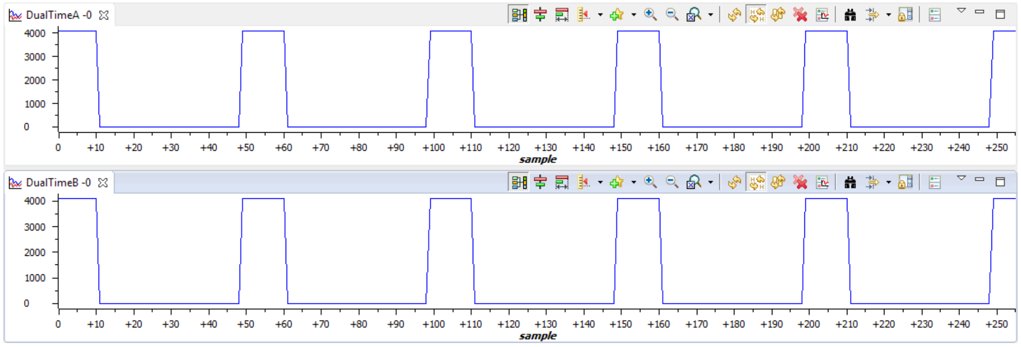
    Observe the contents of the AdcaResults buffer in the Memory Browser. If the code is running as expected, you should see a series of readings close to 0, followed by another series close to full scale (4095), similar to the first part of lab 2. This is the output from PWM1A.

14. If the graph from the previous lab exercise is still open, close it now. Open and set up a Dual Time graph to plot a 256-point window of both ADC results buffers. Click: `Tools` → `Graph` → `Dual Time` and set the following values:

Acquisition Buffer Size	256
DSP Data Type	16-bit unsigned integer
Sampling Rate (Hz)	50000
Start Address A	AdcaResults
Start Address B	AdccResults
Display Data Size	256
Time Display Unit	sample

    Select `OK` to save the graph options.

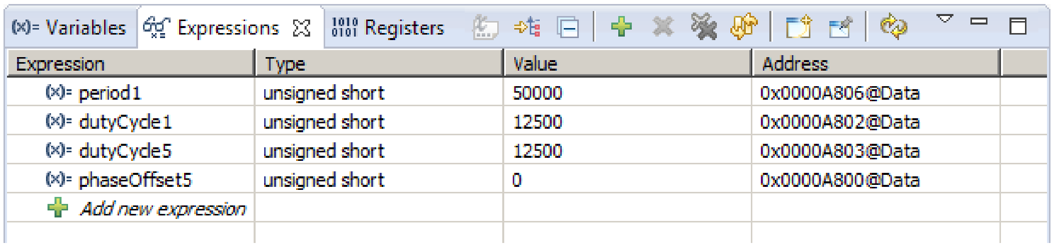
15. We would like to be able to view both graphs at the same time. To do this, position the mouse cursor on the tab of the graph DualTimeA-0, then click and hold down the left mouse button while dragging the graph to a different part of the workspace. Choose an area where both graphs can be viewed simultaneously before releasing the mouse button. The graphs view should look like:



16. The Expressions window should still be open from the previous lab exercise. If not, then click the “Expressions” tab near the top of the CCS window. Add the following variables to the Expressions window:

    - `period1`
    - `dutyCycle1`
    - `dutyCycle5`
    - `phaseOffset5`

    The other expressions are not needed for this lab exercise and can safely be deleted from the Expression list, if desired.



Expression	Type	Value	Address
(x)= period1	unsigned short	50000	0x0000A806@Data
(x)= dutyCycle1	unsigned short	12500	0x0000A802@Data
(x)= dutyCycle5	unsigned short	12500	0x0000A803@Data
(x)= phaseOffset5	unsigned short	0	0x0000A800@Data
Add new expression			

Run the Code - Real-Time Emulation Mode

17. We need to enable the graph windows 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.

18. Enable the Expressions window for continuous refresh using the same procedure as the previous step.

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

    `Scripts → Realtime Emulation Control → Run_Realtime_with_Reset`

20. *Carefully* remove and replace the connector wire to the ADCINA0 input (connector J3, pin #30). The ADC results graph A should be zero when the jumper wire is removed.

    Next, *carefully* remove and replace the connector wire to the ADCINC3 input (connector J3, pin #24). The ADC results graph B should be zero when the jumper wire is removed. This confirms both buffers are updating in real-time.

Adjust the PWM Settings

21. We will adjust the PWM settings and check the effects in the graph. First, click on the `period1` variable value in the Expressions window and change its value to 30000. What effect did this have on the PWM signals?

22. Restore the `period1` variable to its original value of 50000.

23. Next, change the duty cycle variables `dutyCycle1` and `dutyCycle5` while observing the PWM signals. In both cases be careful to choose a number between about 1000 and 49000. Were the changes to the PWM signals as expected?

24. Now change the `phaseOffset5` variable to a positive number between 0 and 49000. What effect did this have?

25. Set the PWM variables as follows:

    period1 = 50000

    dutyCycle1 = 25000

    dutyCycle5 = 25000

    phaseOffset5 = 25000

    What is the relationship between these PWM waveforms called?

26. Finally, set the variable `period1` to 75000. What happened and why?

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

    `Scripts → Realtime Emulation Control → Full_Halt`

28. Run the code in real-time mode. Click:

    `Scripts → Realtime Emulation Control → Run_Realtime_with_Reset`

    Notice the original waveforms should now be displayed.

29. Again, fully halt the CPU in real-time mode. Click:

    `Scripts → Realtime Emulation Control → Full_Halt`

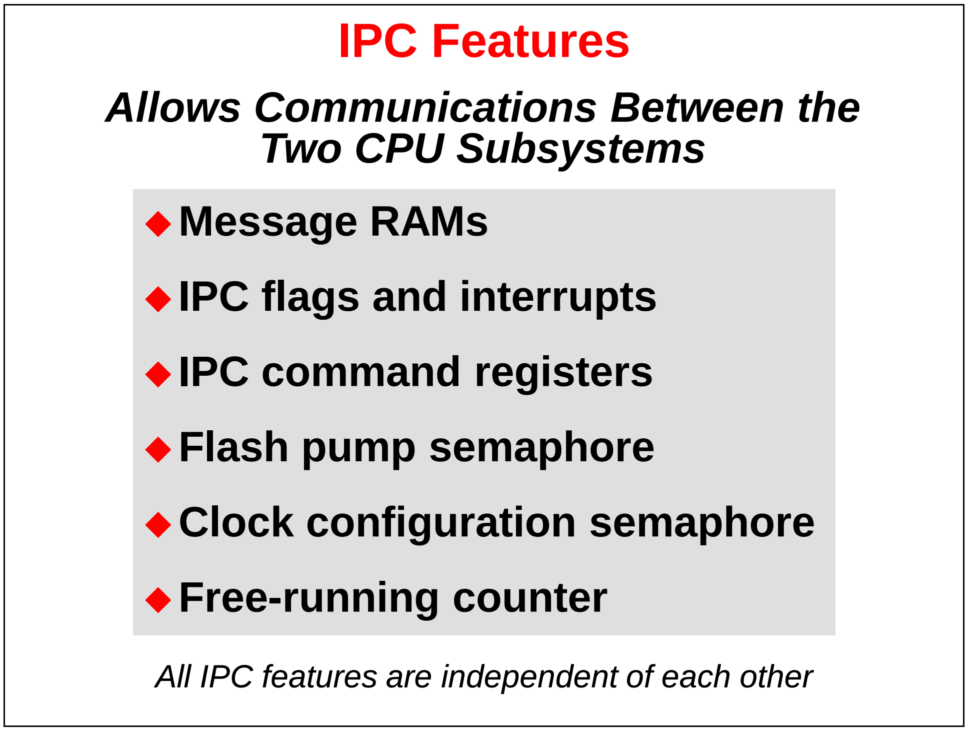
Terminate Debug Session and Close Project

30. Terminate the active debug session using the “Terminate” button. This will close the debugger and return CCS to the “CCS Edit” perspective” view.

31. Next, close the project by right-clicking on Lab3_cpu01 in the Project Explorer window and select `Close Project`.

End of Exercise

Inter-Processor Communications (IPC)



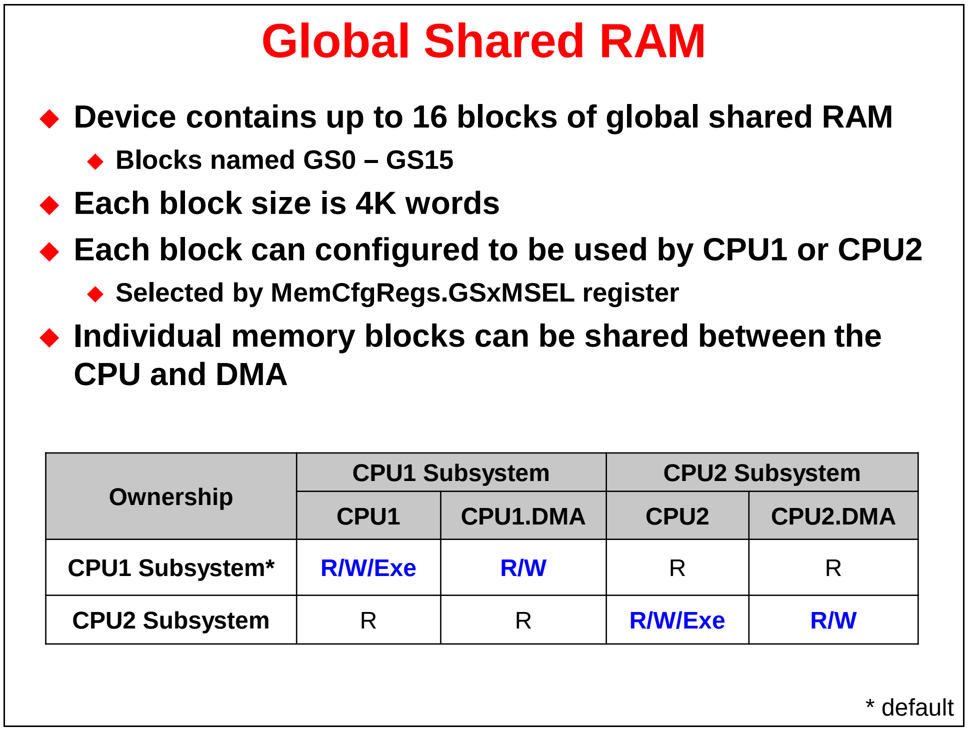
IPC Features

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 SARAM and Message SARAM



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**

* default

There are up to 16 blocks of shared SARAM on F2837xD devices. These shared SARAM 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.

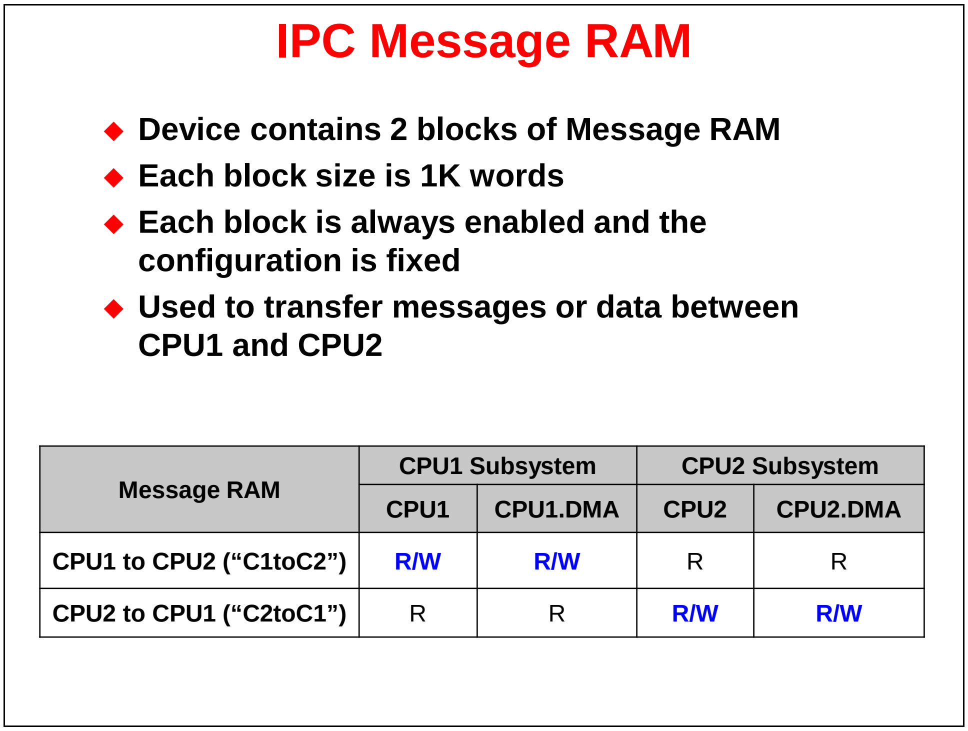
CPU1 core ownership:

At reset, CPU1 owns all of the shared SARAM 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**

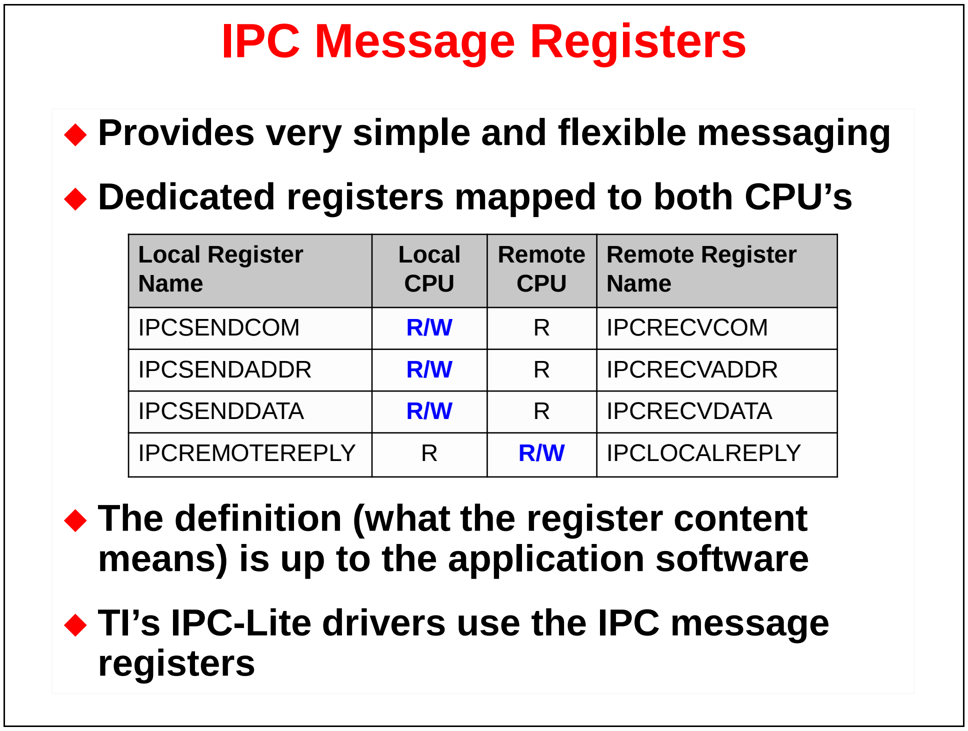
The F2837xD has two dedicated message RAM blocks. Each block is 1K words in length. Unlike the shared SARAM blocks, these blocks provide communication in one direction only and cannot be reconfigured.

CPU1 to CPU2 “C1toC2” message RAM:

The first message SARAM 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 SARAM 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.



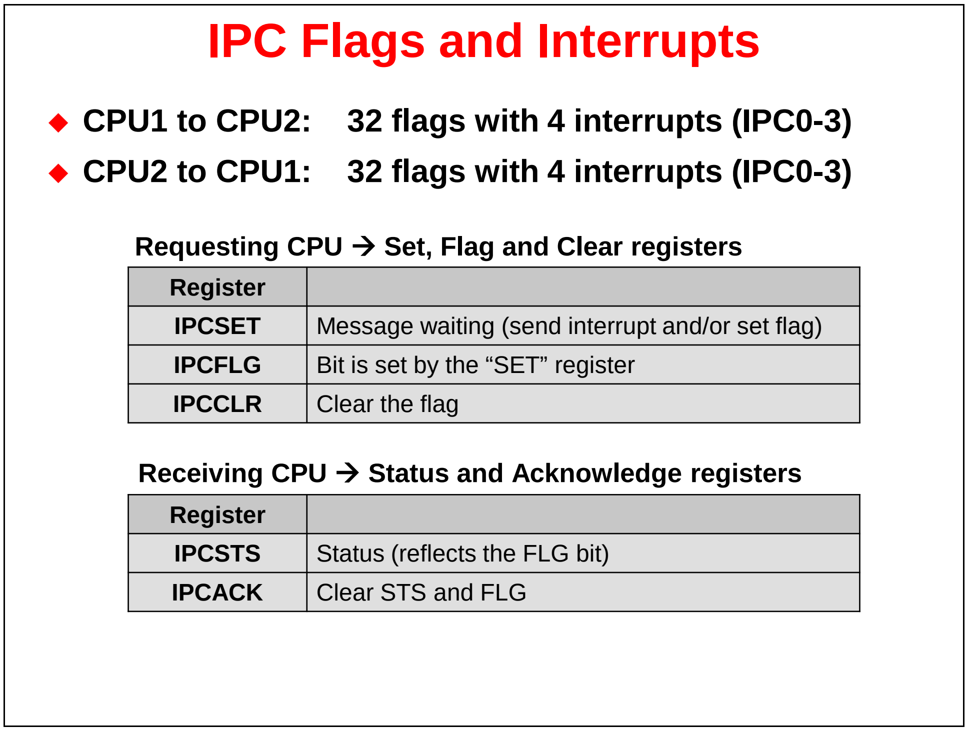
IPC Message Registers

- 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 wishes 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 through the Peripheral Interrupt Expansion (PIE) module. Each of the interrupts has a dedicated vector within the PIE.

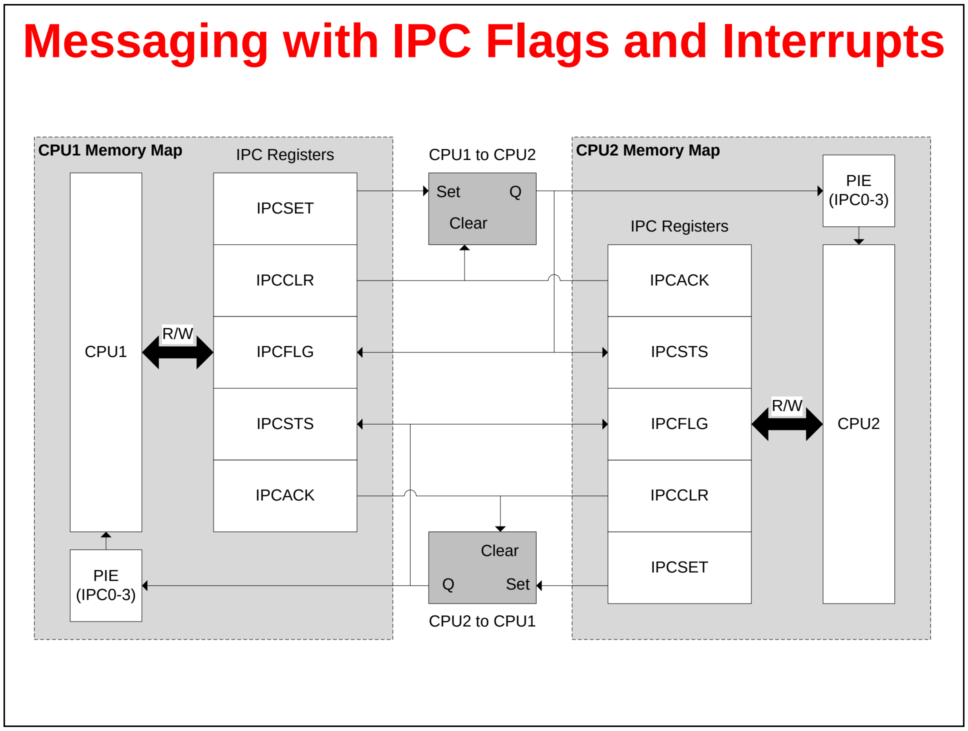
28 Flags:

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

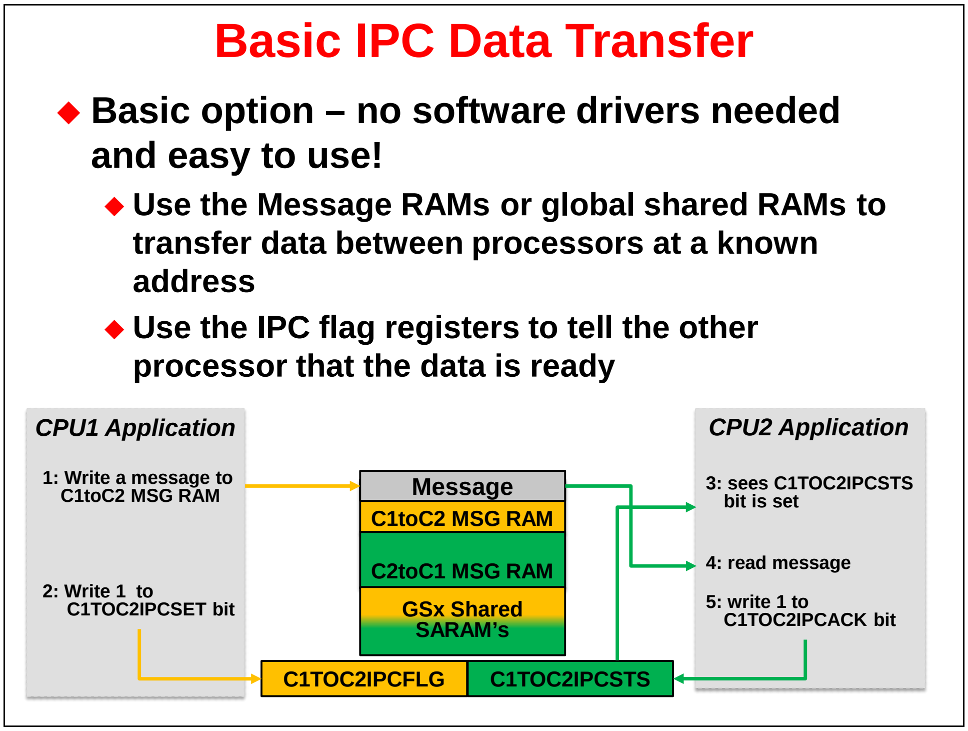
The registers to control the IPC interrupts and flags are 32-bits:

Bits [3:0] = interrupt & flag
Bits [31:4] = flag only



Messaging with IPC Flags and Interrupts

IPC Data Transfer



Basic IPC Data Transfer

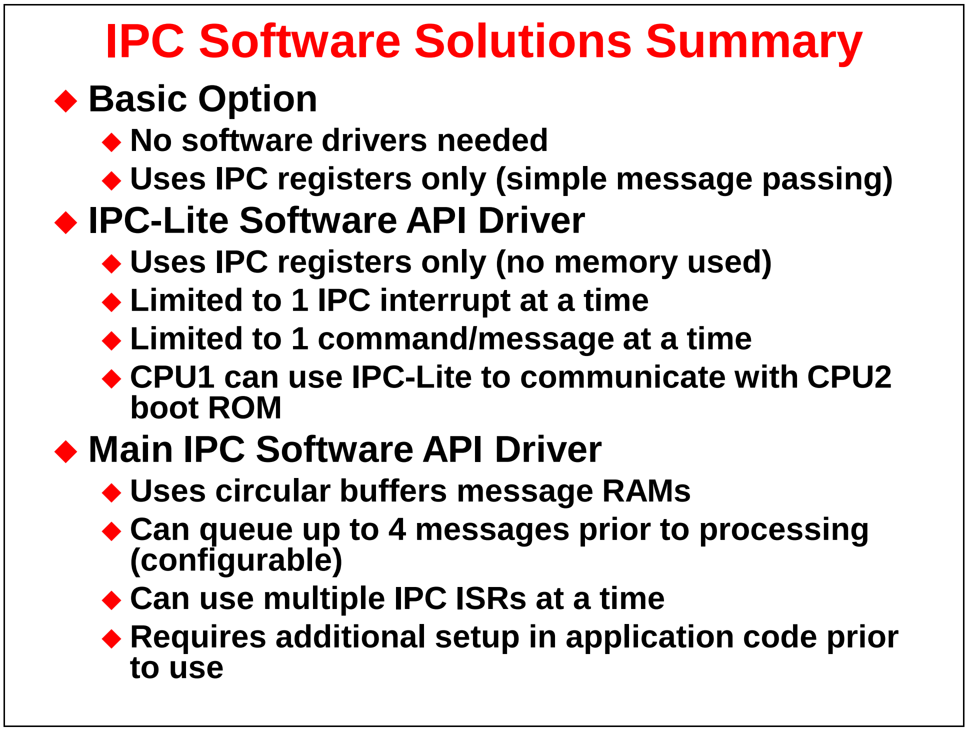
- **Basic option – no software drivers needed and easy to use!**
  - **Use the Message RAMs or global shared RAMs to transfer data between processors at a known address**
  - **Use the IPC flag registers to tell the other processor that the data is ready**

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 SARAM'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 SARAM 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)

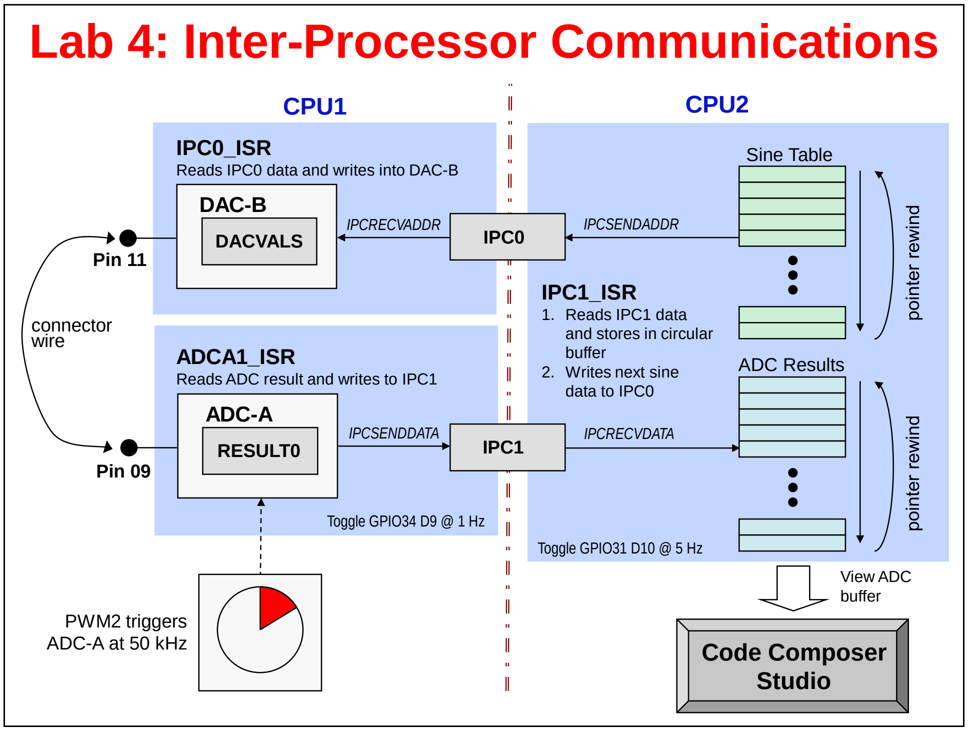
- Utilizes circular buffers in C2toC1 and C1toC2 message RAM’s.
- Allows application to queue up to 4 messages prior to processing (configurable).
- Allows application to use multiple IPC ISR’s at a time.
- Requires additional set up in application code prior to use.

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 4: 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. As in the previous lab exercise, 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.



➢ Procedure

Open the Projects – CPU1 & CPU2

1. Two projects named `Lab4_cpu01` and `Lab4_cpu02` has been created for this lab. 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:\F2837xD\Labs\Lab4` 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.

Inspect the Project – CPU1

2. Click on the project name `Lab4_cpu01` in the Project Explorer window to set the project active. Then click on the plus sign (+) to the left of Lab4_cpu01 to expand the file list.

3. Open and inspect `Lab4_cpu01.c`. This file contains two interrupt service routines – one (`ipc1_isr`) to read the incoming sine data over IPC, and the other (`adca1_isr`) to read the ADC results. The code for these routines is located near the bottom of the file.

4. In `ipc1_isr()` incoming data from CPU2 is read via the `IPCRECVADDR` register. In `adca1_isr()` the ADC result to CPU2 is written via the `IPCSENDDATA` 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

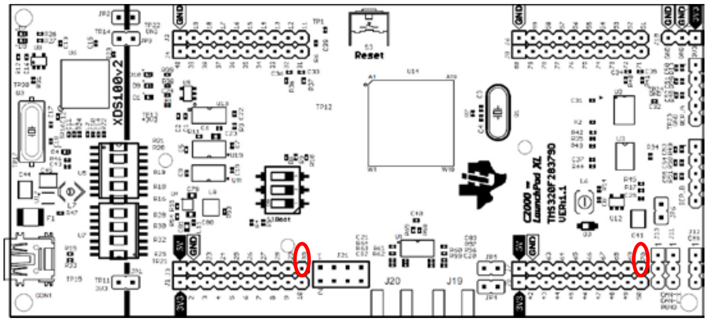
5. Click on the project name `Lab4_cpu02` in the Project Explorer window to set the project active. Then click on the plus sign (+) to the left of Lab4_cpu02 to expand the file list.

6. Open and inspect `Lab4_cpu02.c`. This file contains a single interrupt service routine – (`ipc2_isr`) to read the incoming ADC data from CPU1 and write the next sine table point to CPU1. The code for this routine is located at the bottom of the file.

7. In `ipc2_isr()` incoming ADC data 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. We need to connect the DACOUTB output pin to the ADCINA0 input pin, as was done in the Lab2 exercise. Using the jumper wire, carefully make a connection between connector J3, pin #30 (ADCINA0) and connector J7, pin #70 (DACOUTB). Remove all other jumper wires. Refer to the following diagram for the pins that need to be connected using the jumper wire.



Build and Load the Project

9. In the Project Explorer window click on the “Lab4_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 “Lab4_cpu02” project.

10. Again, in the Project Explorer window click on the “Lab4_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, 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. Next, we need to connect to and load the program on CPU2. Right-click at the line “Texas Instruments XDS100v2 USB Debug Probe_0/C28xx_CPU2” and select “`Connect Target`”.
12. With the line “Texas Instruments XDS100v2 USB Debug Probe_0/C28xx_CPU2” still highlighted, load the program:

    `Run → Load → Load Program…`

    Browse to the file: `C:\F2837xD\Labs\Lab4\cpu02\Debug\Lab4_cpu02.out` and select `OK` to load the program.
13. Again, with the line “Texas Instruments XDS100v2 USB Debug Probe_0/C28xx_CPU2” still highlighted, set the bootloader mode using the menu bar by clicking:

    `Scripts → EMU Boot Mode Select → EMU_BOOT_SARAM`

    CPU1 bootloader mode was already set in the previous lab exercise. 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 for both CPU1 and CPU2.

Run the Code

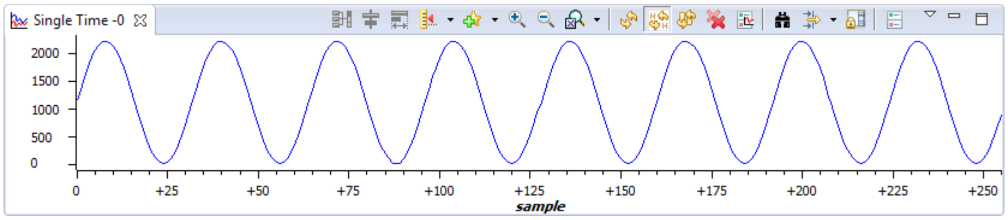
14. In the Debug window, click on the line “Texas Instruments XDS100v2 USB Debug Probe_0/C28xx_CPU1”. Run the code on CPU1 by clicking the green “Resume” button. At this point CPU1 is waiting for CPU2 to be ready.
15. In the Debug window, click on the line “Texas Instruments XDS100v2 USB Debug Probe_0/C28xx_CPU2”. As before, run the code on CPU2 by clicking the “Resume” button. Using the IPC, CPU2 communicates to CPU1 that it is now ready. LED D10 connected to CPU1 on the LaunchPad should be blinking at a period of approximately 1 second. Note that LED D9 connected to CPU2 will not be used in this lab exercise.
16. In the Debug window select CPU1. Halt the CPU1 code after a few seconds by clicking on the “Suspend” button.
17. Then in the Debug window select CPU2. Halt the CPU2 code by using the same procedure.

View the ADC Results

18. If the graph from the previous lab exercise is still open, close it now. Open and set up a graph to plot a 256-point window of the ADC results buffer. Click: `Tools → Graph → Single Time` and set the following values:

Acquisition Buffer Size	256
DSP Data Type	16-bit unsigned integer
Sampling Rate (Hz)	50000
Start Address	AdcaResults
Display Data Size	256
Time Display Unit	sample

    Select `OK` to save the graph options.
19. 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

20. 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.

21. In the Debug window highlight the line "Texas Instruments XDS100v2 USB Debug Probe_0/C28xx_CPU1". Run the code on CPU1 in real-time mode by clicking:

    `Scripts → Realtime Emulation Control → Run_Realtime_with_Reset`

22. Next, in the Debug window highlight the line "Texas Instruments XDS100v2 USB Debug Probe_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.

23. *Carefully* remove and replace the connector wire from the DAC-B output (connector J7, pin #70) or to the ADCINA0 input (connector 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 data is being transmitted over the IPC from CPU2, and (after being sent from DAC to ADC) received from CPU1, also over the IPC.

24. Again, in the Debug window highlight the line "Texas Instruments XDS100v2 USB Debug Probe_0/C28xx_CPU1". Fully halt the code on CPU1 in real-time mode by clicking:

    `Scripts → Realtime Emulation Control → Full_Halt`

25. Next, fully halt the code on CPU2 in real-time mode by using the same procedure.

Terminate Debug Session and Close Project

26. The "Terminate" button will terminate the active debug session, close the debugger and return CCS to the "CCS Edit" perspective view.

    Click: `Run → Terminate` or use the Terminate icon: 🟥

    Next, close the Lab4_cpu01 and Lab4_cpu02 projects by right-clicking on each project in the Project Explorer window and select `Close Project`.

End of Exercise

Support Resources

C2000 MCU Multi-day Training Course

C2000 MCU Multi-day Training Course

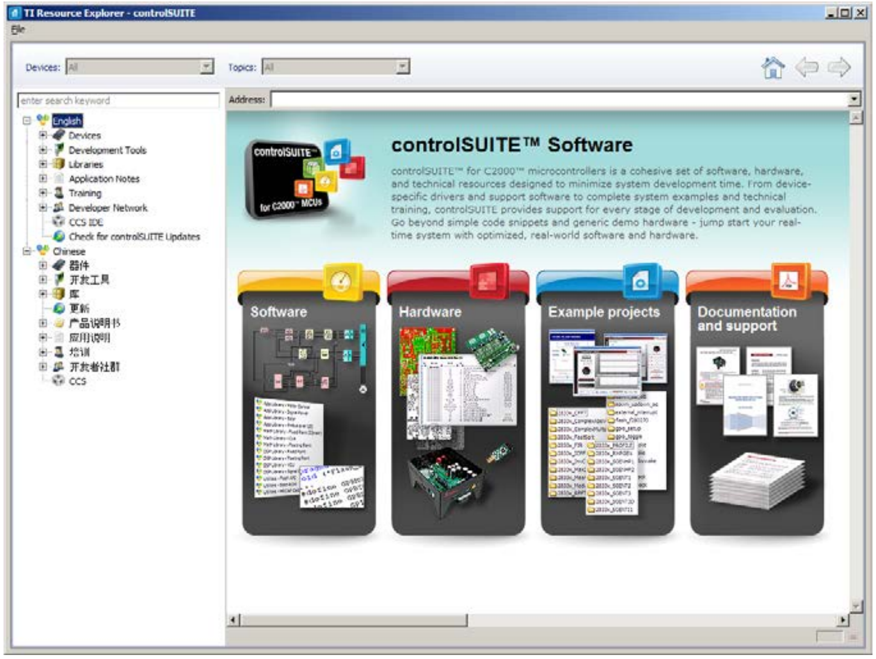
In-depth hands-on TMS320F28379D Design and Peripheral Training

TMS320F28379D Workshop Outline

- Architectural Overview
- Programming Development Environment
- Peripheral Register Header Files
- Reset and Interrupts
- System Initialization
- Analog Subsystem
- Control Peripherals
- Direct Memory Access (DMA)
- Control Law Accelerator (CLA)
- System Design
- Dual-Core Inter-Processor Communications (IPC)
- Communications
- Support Resources

controlSUITE™

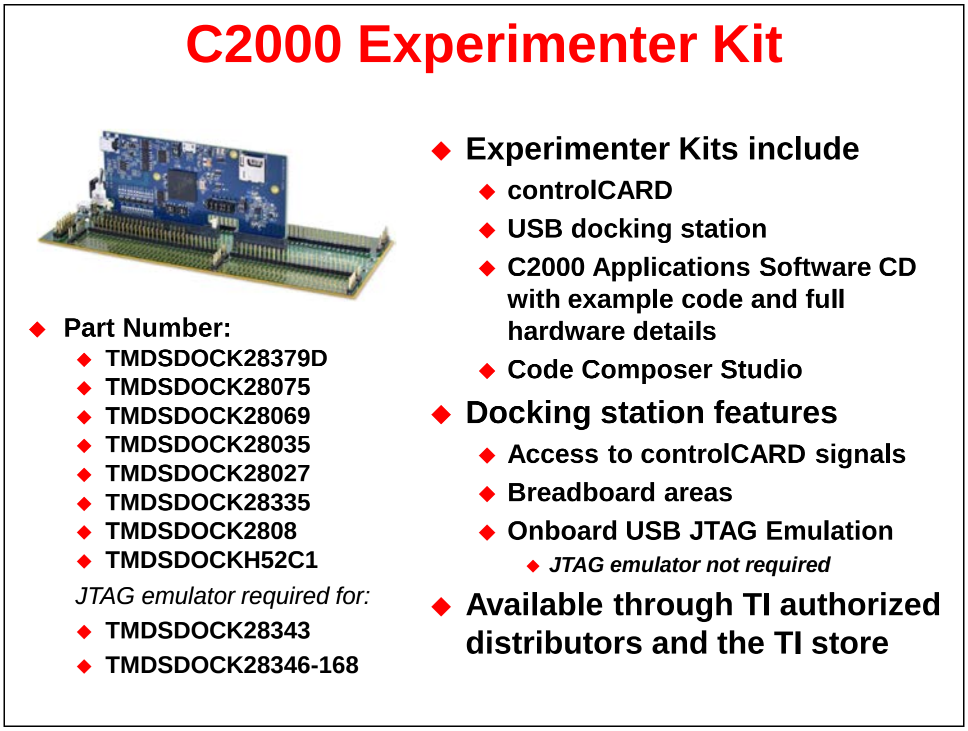
controlSUITE™



controlSUITE™ Software

controlSUITE™ for C2000™ microcontrollers is a cohesive set of software, hardware, and technical resources designed to minimize system development time. From device-specific drivers and support software to complete system examples and technical training, controlSUITE provides support for every stage of development and evaluation. Go beyond simple code snippets and generic demo hardware - jump start your real-time system with optimized, real-world software and hardware.

Experimenter's Kit



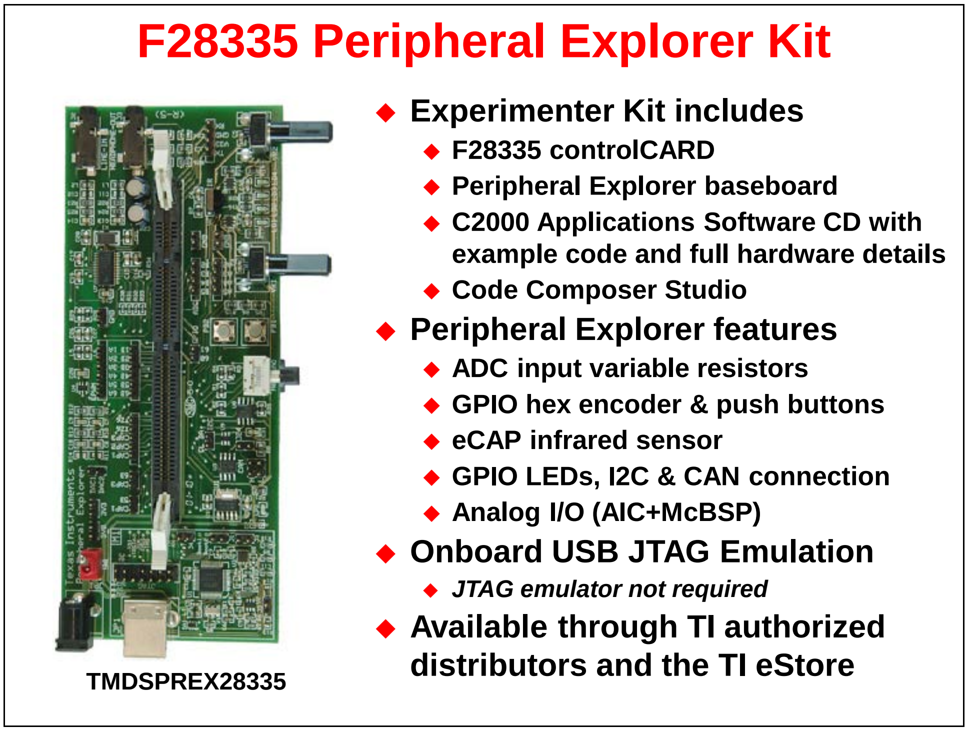
C2000 Experimenter Kit

- Part Number:
  - TMDSDOCK28379D
  - TMDSDOCK28075
  - TMDSDOCK28069
  - TMDSDOCK28035
  - TMDSDOCK28027
  - TMDSDOCK28335
  - TMDSDOCK2808
  - TMDSDOCKH52C1

  JTAG emulator required for:
  - TMDSDOCK28343
  - TMDSDOCK28346-168

- Experimenter Kits include
  - controlCARD
  - USB docking station
  - C2000 Applications Software CD with example code and full hardware details
  - Code Composer Studio
- Docking station features
  - Access to controlCARD signals
  - Breadboard areas
  - Onboard USB JTAG Emulation
    - ***JTAG emulator not required***
- Available through TI authorized distributors and the TI store

Perpheral Explorer Kit

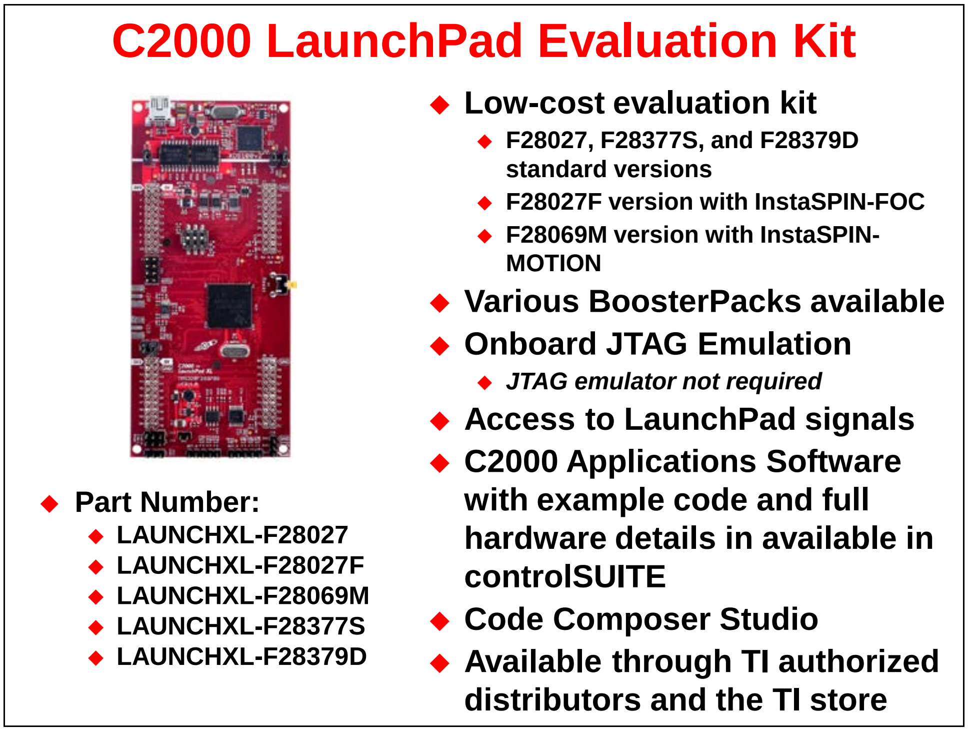


F28335 Peripheral Explorer Kit

TMDSPREX28335

- Experimenter Kit includes
  - F28335 controlCARD
  - Peripheral Explorer baseboard
  - C2000 Applications Software CD with example code and full hardware details
  - Code Composer Studio
- Peripheral Explorer features
  - ADC input variable resistors
  - GPIO hex encoder & push buttons
  - eCAP infrared sensor
  - GPIO LEDs, I2C & CAN connection
  - Analog I/O (AIC+McBSP)
- Onboard USB JTAG Emulation
  - ***JTAG emulator not required***
- Available through TI authorized distributors and the TI eStore

LaunchPad Evaluation Kit

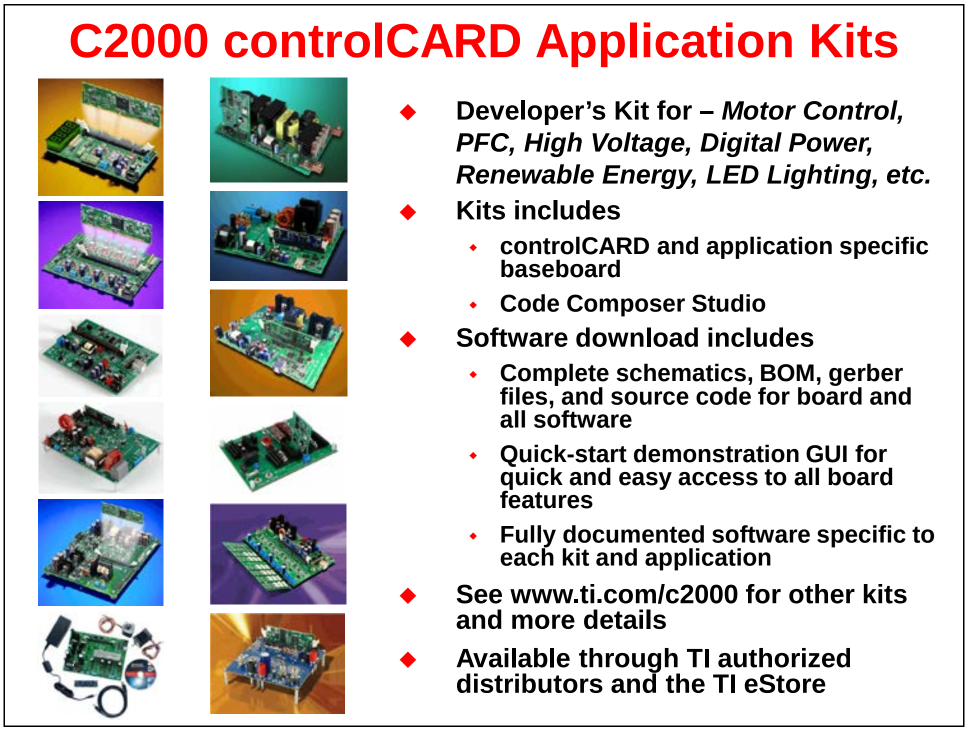


C2000 LaunchPad Evaluation Kit

- Low-cost evaluation kit
  - F28027, F28377S, and F28379D standard versions
  - F28027F version with InstaSPIN-FOC
  - F28069M version with InstaSPIN-MOTION
- Various BoosterPacks available
- Onboard JTAG Emulation
  - *JTAG emulator not required*
- Access to LaunchPad signals
- C2000 Applications Software with example code and full hardware details in available in controlSUITE
- Code Composer Studio
- Available through TI authorized distributors and the TI store

- Part Number:
  - LAUNCHXL-F28027
  - LAUNCHXL-F28027F
  - LAUNCHXL-F28069M
  - LAUNCHXL-F28377S
  - LAUNCHXL-F28379D

Application Kits



C2000 controlCARD Application Kits

- Developer's Kit for – *Motor Control, PFC, High Voltage, Digital Power, Renewable Energy, LED Lighting, etc.*
- Kits includes
  - controlCARD and application specific baseboard
  - Code Composer Studio
- Software download includes
  - Complete schematics, BOM, gerber files, and source code for board and all software
  - Quick-start 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 eStore

XDS100 / XDS200 Class JTAG Emulators

XDS100 / XDS200 Class JTAG Emulators

- Blackhawk
  - USB100v2

- Spectrum Digital
  - XDS100v2

- Blackhawk
  - USB200

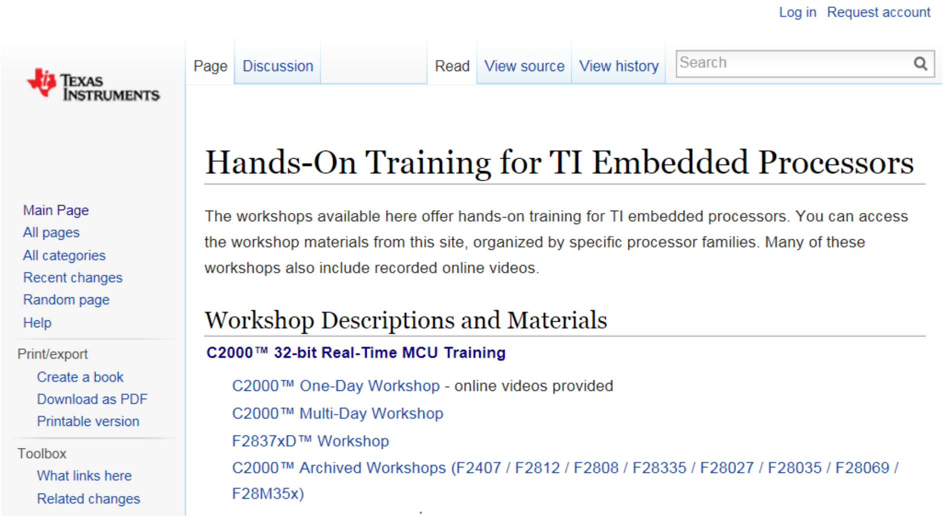
- Spectrum Digital
  - XDS200

www.blackhawk-dsp.com

www.spectrumdigital.com

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.

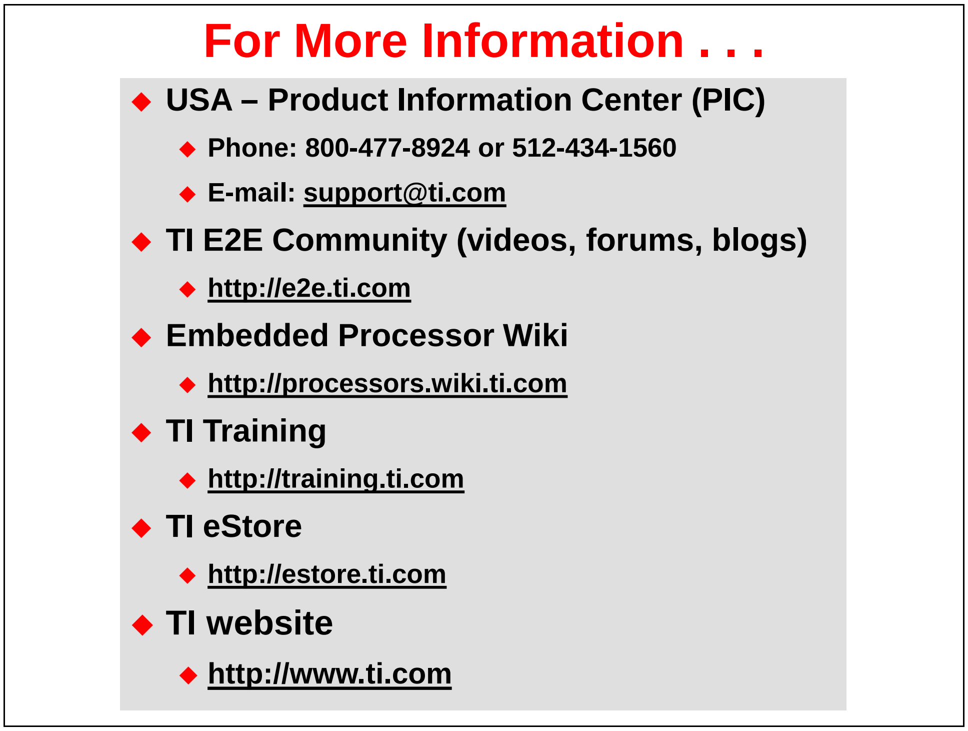
Workshop Descriptions and Materials

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

For More Information…



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 eStore
  - http://estore.ti.com
- TI website
  - http://www.ti.com

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.

➢ Initial Hardware Set Up

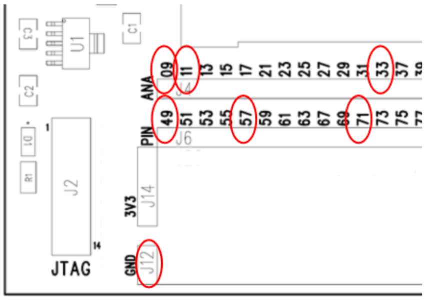
- **F28379D Experimenter Kit:**

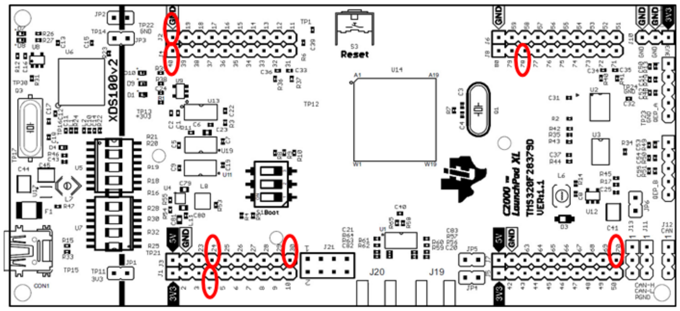
Insert the F28379D controlCARD into the Docking Station connector slot. Using the two (2) supplied USB cables – plug the USB Standard Type A connectors into the computer USB ports and plug the USB Mini-B connectors as follows:

- 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.

Experimenter Kit and LaunchPad Mapping





Function	Experimenter Kit	LaunchPad
ADCINA0	ANA header, Pin # 09	J3-30
ADCINC3	ANA header, Pin # 33	J3-24
GND	GND	J2-20 (GND)
GPIO18	Pin # 71	J1-4
DACOUTB	ANA header, Pin # 11	J7-70
PWM1A	Pin # 49	J4-40
PWM5A	Pin # 57	J8-78

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.