



C2000™ MCU 1-Day Workshop

Workshop Guide and Lab Manual

*F28xMCUodw
Revision 4.0
February 2012*



Technical Training
Organization

Important Notice

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

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

Customers are responsible for their applications using TI components.

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

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

Copyright © 2009 – 2012 Texas Instruments Incorporated

Revision History

April 2009 – Revision 1.0

October 2009 – Revision 1.1

June 2010 – Revision 2.0

December 2010 – Revision 2.1

October 2011 – Revision 3.0

February 2012 – Revision 4.0

Mailing Address

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

Workshop Topics

<i>Workshop Topics</i>	3
<i>Workshop Introduction</i>	4
<i>Architecture Overview</i>	8
<i>Programming Development Environment</i>	12
Code Composer Studio	12
Linking Sections in Memory	14
<i>Lab 1: Linker Command File</i>	17
<i>Peripheral Register Header Files</i>	23
<i>Reset, Interrupts and System Initialization</i>	30
Reset	30
Interrupts	32
Peripheral Interrupt Expansion (PIE)	34
Oscillator / PLL Clock Module	36
Watchdog Timer Module.....	37
GPIO.....	38
<i>Lab 2: System Initialization</i>	41
<i>Control Peripherals</i>	46
ADC Module	46
Pulse Width Modulation.....	48
ePWM.....	49
eCAP	61
eQEP.....	63
<i>Lab 3: Control Peripherals</i>	65
<i>Flash Programming</i>	71
Flash Programming Basics	71
Programming Utilities and CCS Flash Programmer.....	72
Code Security Module and Password	73
<i>Lab 4: Programming the Flash</i>	75
<i>The Next Step</i>	82
Training	82
controlSUITE	82
Development Tools.....	83
C2000 Workshop Download Wiki	86
Development Support.....	86

Workshop Introduction

C2000 Microcontroller 1-Day Workshop



Texas Instruments
Technical Training



TEXAS
INSTRUMENTS

C2000 is trademarks of Texas Instruments. Copyright © 2012 Texas Instruments. All rights reserved.



C2000 MCU 1-Day Workshop Outline

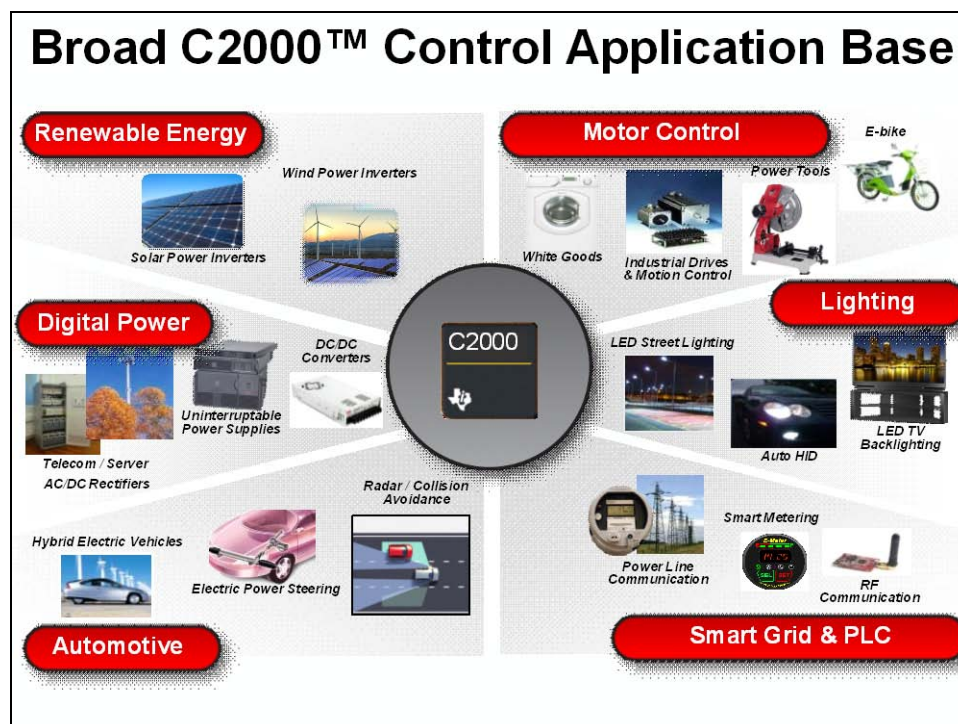
- ◆ Workshop Introduction
- ◆ Architecture Overview
- ◆ Programming Development Environment
 - ◆ Lab: Linker command file
- ◆ Peripheral Register Header Files
- ◆ Reset, Interrupts and System Initialization
 - ◆ Lab: Watchdog and interrupts
- ◆ Control Peripherals
 - ◆ Lab: Generate and graph a PWM waveform
- ◆ Flash Programming
 - ◆ Lab: Run the code from flash memory
- ◆ The Next Step...

Required Workshop Materials

- ◆ http://processors.wiki.ti.com/index.php/C2000_Piccolo_One-Day_Workshop_Home_Page
- ◆ F28069 controlSTICK kit
- ◆ Install Code Composer Studio v5.1.1
- ◆ Run the workshop installer
C2000 Microcontroller 1-Day Workshop-4.0-Setup.exe
 - ◆ Lab Files / Solution Files
 - ◆ Student Guide and Documentation

TI Embedded Processing Portfolio

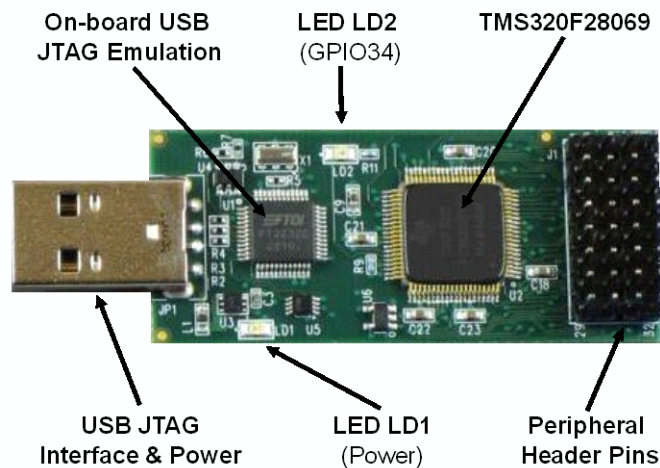
TI Embedded Processors						
Microcontrollers (MCUs)		ARM®-Based Processors		Digital Signal Processors (DSPs)		
16-bit ultra-low power MCUs	32-bit real-time MCUs	32-bit ARM Cortex™-M3 MCUs	ARM Cortex-A8 MPUs	DSP DSP+ARM	Multi-core DSP	Ultra Low power DSP
MSP430™ Up to 25 MHz Flash 1 KB to 256 KB Analog I/O, ADC, LCD, USB, RF Measurement, Sensing, General Purpose	C2000™ Delfino Piccolo™ 40MHz to 300 MHz Flash, RAM 16 KB to 512 KB PWM, ADC, CAN, SPI, I²C Motor Control, Digital Power, Lighting, Ren. Energy	Stellaris® ARM® Cortex™-M3 Up to 100 MHz Flash 8 KB to 256 KB USB, ENET, MAC+PHY, CAN, ADC, PWM, SPI Connectivity, Security, Motion Control, HMI, Industrial Automation	Sitara™ ARM® Cortex™-A8 & ARM9 300MHz to >1GHz Cache, RAM, ROM USB, CAN, PCIe, EMAC Industrial computing, POS & portable data terminals	C6000™ DaVinci™ video processors OMAP™ 300MHz to >1Ghz +Accelerator Cache, RAM, ROM USB, ENET, PCIe, SATA, SPI Floating/Fixed Point Video, Audio, Voice, Security, Confer.	C6000™ 24,000 MMACS Cache, RAM, ROM SRIO, EMAC, DMA, PCIe Telecom T&M, media gateways, base stations	C5000™ Up to 300 MHz +Accelerator Up to 320KB RAM, Up to 128KB ROM USB, ADC, McBSP, SPI, I²C Audio, Voice, Medical, Biometrics
 Software & Development Tools 						



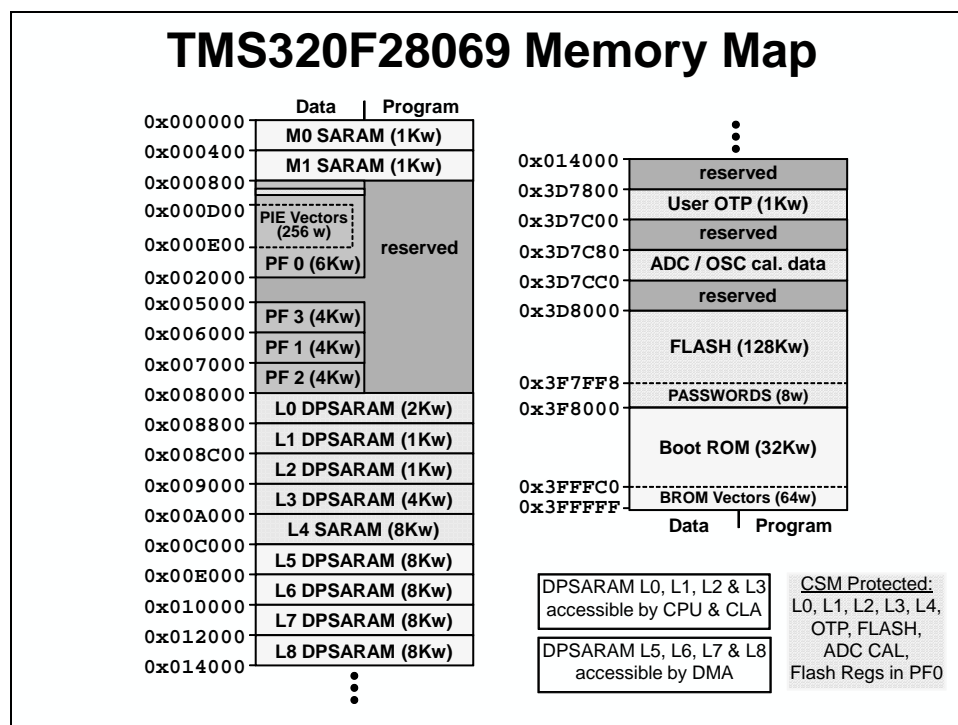
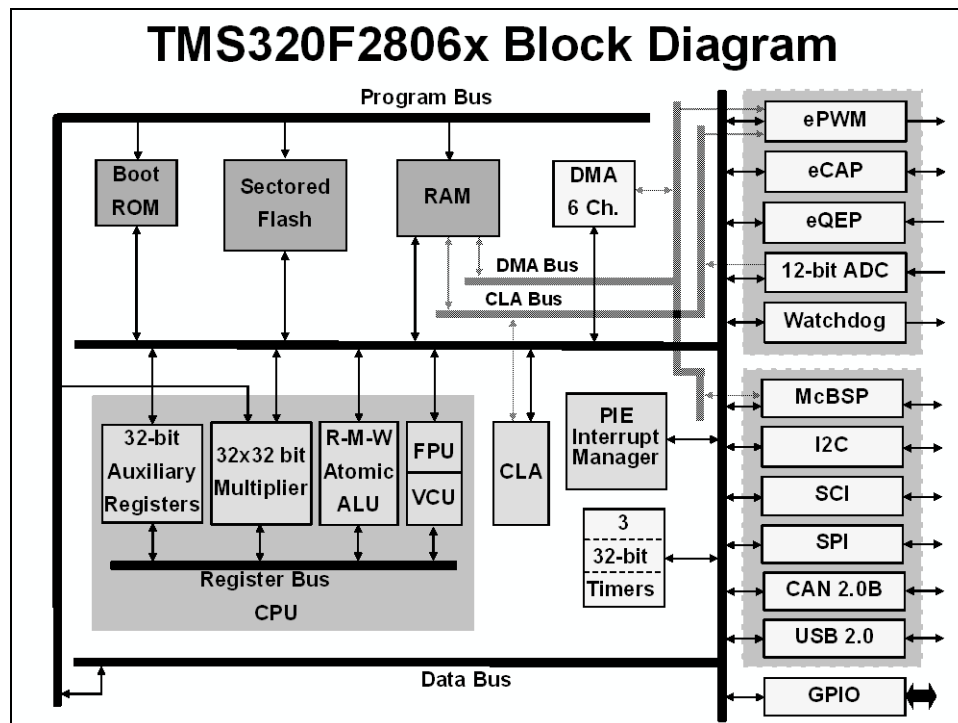
C2000 Delfino / Piccolo Comparison

	F2833x	F2803x	F2806x
Clock	150 MHz	60 MHz	80 MHz
Flash / RAM	128Kw / 34Kw	64Kw / 10Kw	128Kw / 50Kw
On-chip Oscillators	-	2	2
VREG / POR / BOR	-	✓	✓
Watchdog Timer	✓	✓	✓
12-bit ADC	SEQ - based	SOC - based	SOC - based
Analog COMP w/ DAC	-	✓	✓
FPU	✓	-	✓
6-Channel DMA	✓	-	✓
CLA	-	✓	✓
VCU	-	-	✓
ePWM / HR ePWM	✓ / ✓	✓ / ✓	✓ / ✓
eCAP / HR eCAP	✓ / -	✓ / -	✓ / ✓
eQEP	✓	✓	✓
SCI / SPI / I2C	✓	✓	✓
LIN	-	✓	-
McBSP	✓	-	✓
USB	-	-	✓
External Interface	✓	-	-

TMS320F28069 controlSTICK



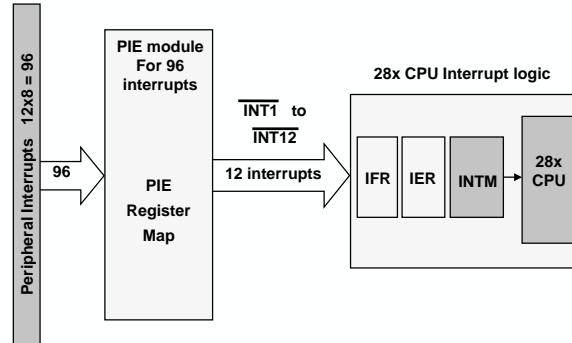
Architecture Overview



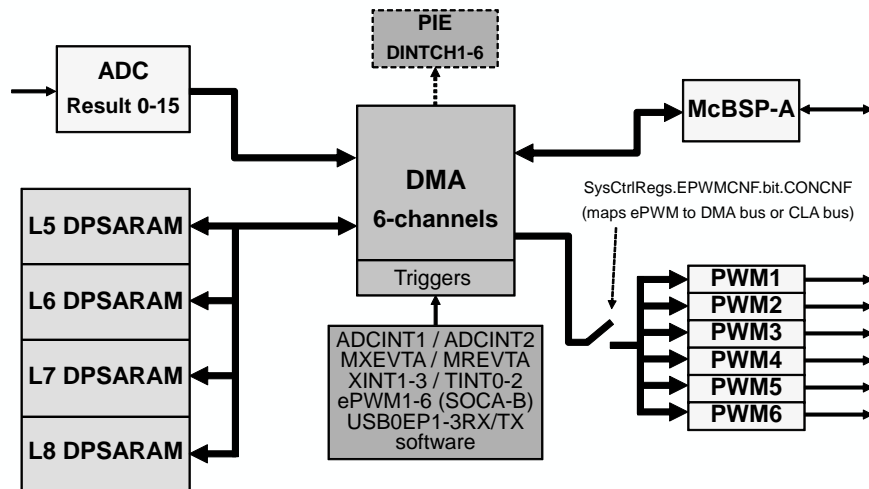
F28x Fast Interrupt Response Manager

- ◆ 96 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)

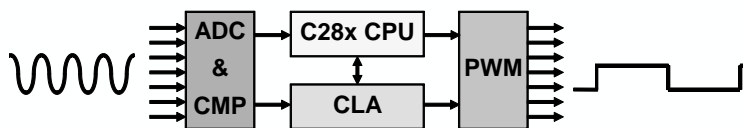


Direct Memory Access (DMA)



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

Control Law Accelerator (CLA)



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

Viterbi, Complex Math, CRC Unit (VCU)

Extends C28x instruction set to support:

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

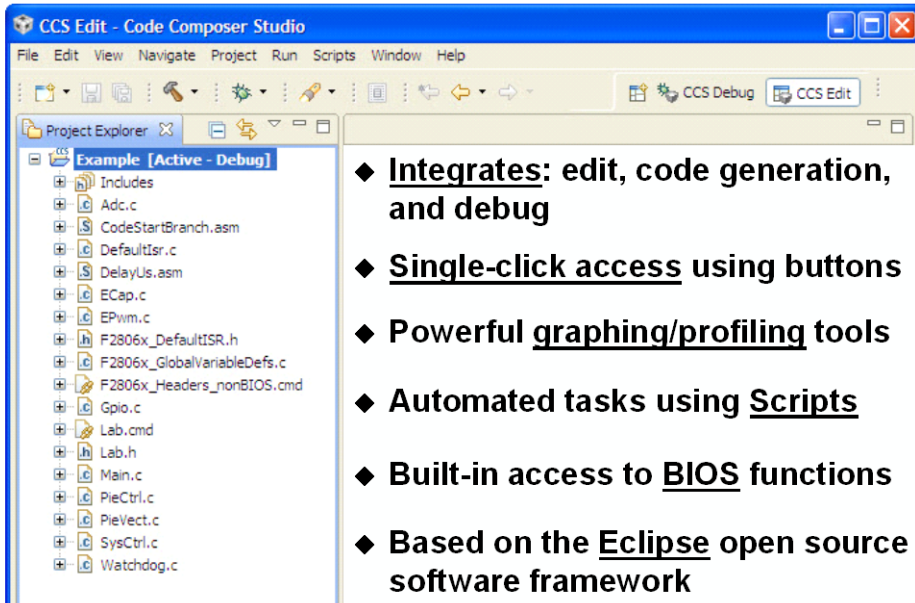
Architecture Summary

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

Programming Development Environment

Code Composer Studio

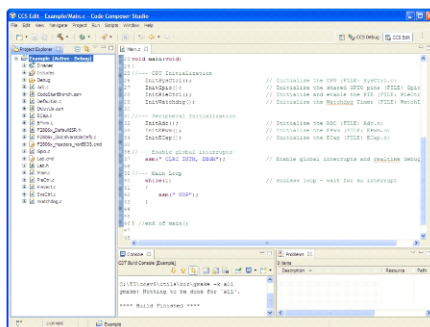
Code Composer Studio: IDE



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

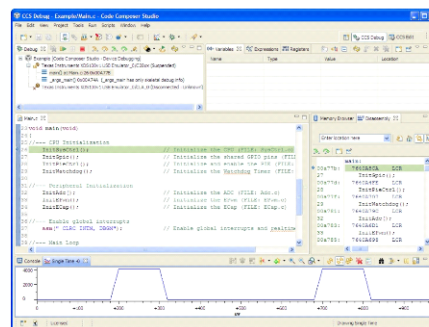
Edit and Debug Perspective (CCSv5)

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



◆ Edit Perspective

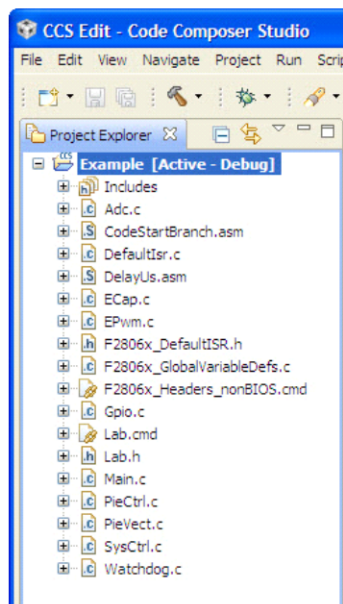
- ◆ Displays views used during code development
 - ◆ C/C++ project, editor, etc.



◆ Debug Perspective

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

CCSv5 Project



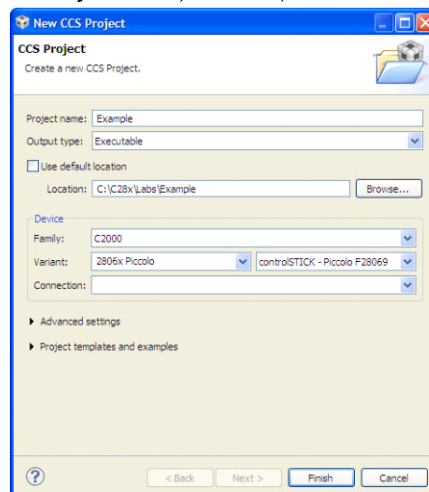
Project files contain:

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

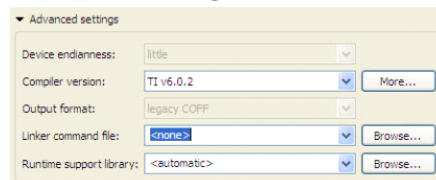
Creating a New CCSv5 Project

◆ **File → New → CCS Project**

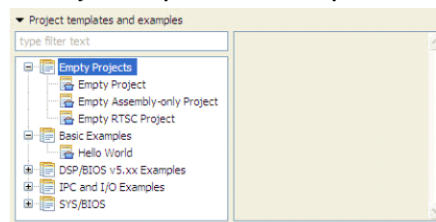
1. Project Name, Location, and Device



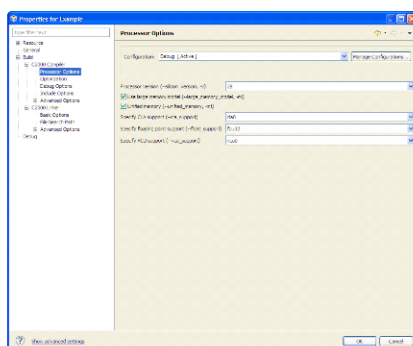
2. Advanced Settings



3. Project Templates and Examples

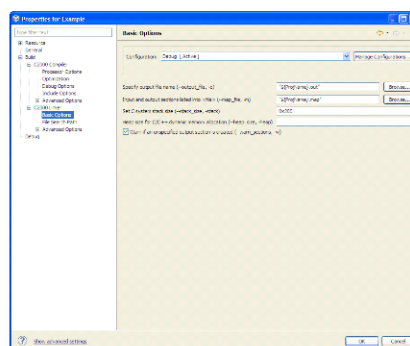


CCSv5 Build Options – Compiler / Linker



◆ Compiler

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



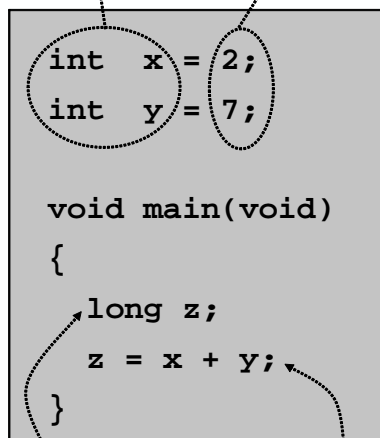
◆ Linker

- ◆ 9 categories for linking
 - ◆ Specify various link options
- ◆ `$(PROJECT_ROOT)` specifies the current project directory

Linking Sections in Memory

Sections

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



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

Compiler Section Names

Initialized Sections

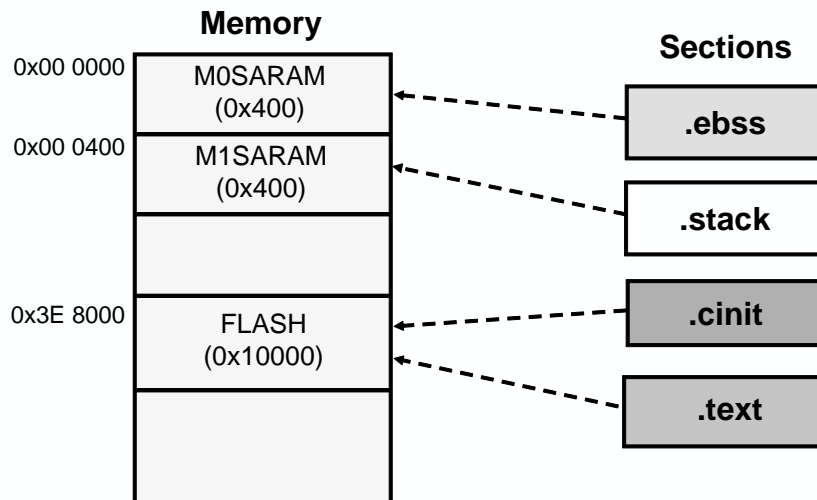
Name	Description	Link Location
.text	code	FLASH
.cinit	initialization values for global and static variables	FLASH
.econst	constants (e.g. const int k = 3;)	FLASH
.switch	tables for switch statements	FLASH
.pinit	tables for global constructors (C++)	FLASH

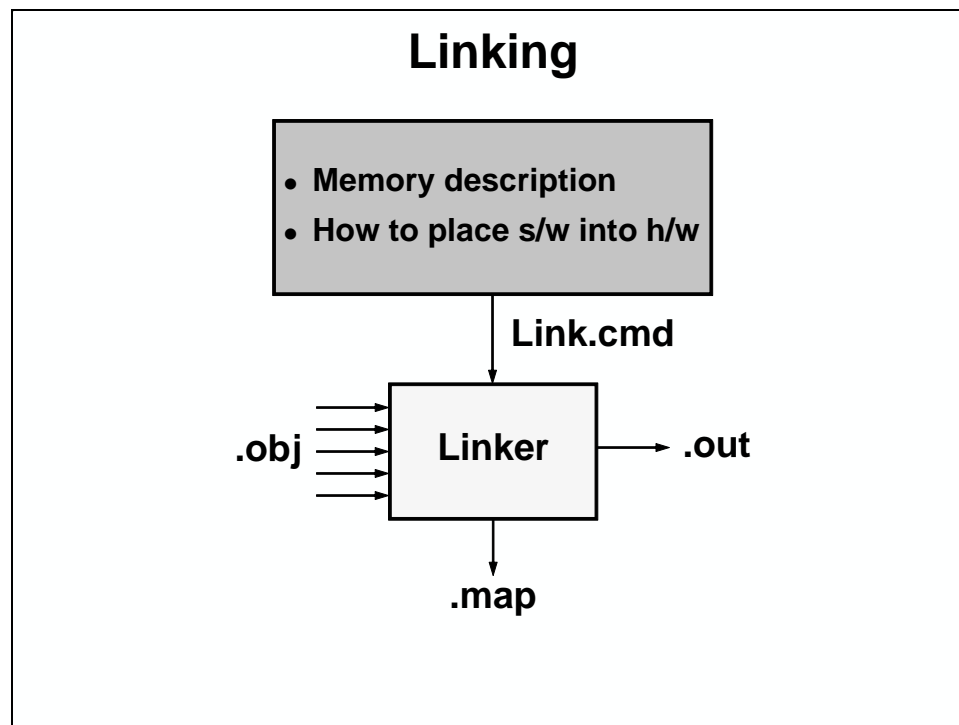
Uninitialized Sections

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

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

Placing Sections in Memory





Linker Command File

```

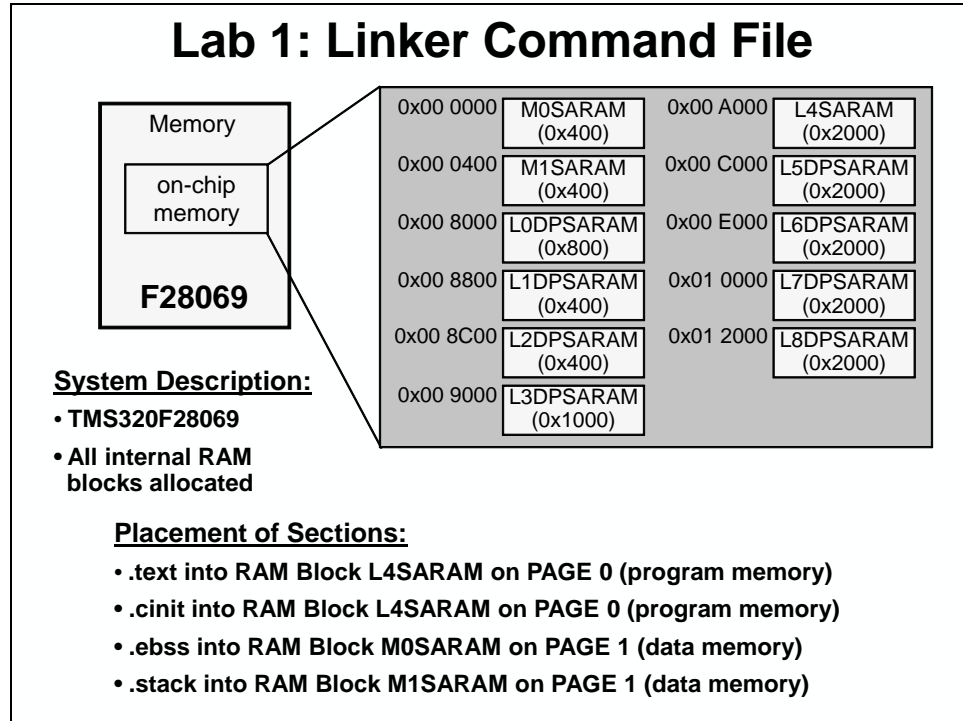
MEMORY
{
    PAGE 0:          /* Program Memory */
    FLASH:           origin = 0x3E8000, length = 0x10000

    PAGE 1:          /* Data Memory */
    M0SARAM:         origin = 0x000000, length = 0x400
    M1SARAM:         origin = 0x000400, length = 0x400
}
SECTIONS
{
    .text:>          FLASH      PAGE = 0
    .ebss:>          M0SARAM    PAGE = 1
    .cinit:>         FLASH      PAGE = 0
    .stack:>         M1SARAM    PAGE = 1
}
  
```


Lab 1: Linker Command File

➤ Objective

Use a linker command file to link the C program file (Lab1.c) into the system described below.



System Description

- TMS320F28069
- All internal RAM blocks allocated

Placement of Sections:

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

➤ Procedure

Start Code Composer Studio and Open a Workspace

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

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

2. The first time CCS opens a “Welcome to Code Composer Studio v5” page appears. Close the page by clicking the X on the “TI Resource Explorer” tab. You should now have an empty workbench. The term workbench refers to the desktop development environment. Maximize CCS to fill your screen.

The workbench will open in the “CCS Edit Perspective” view. Notice the `CCS Edit` icon in the upper right-hand corner. A perspective defines the initial layout views of the workbench windows, toolbars, and menus which are appropriate for a specific type of task (i.e. code development or debugging). This minimizes clutter to the user interface. The “CCS Edit Perspective” is used to create or build C/C++ projects. A “CCS Debug Perspective” view will automatically be enabled when the debug session is started. This perspective is used for debugging C/C++ projects.

Setup Target Configuration

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

File → New → Target Configuration File

In the file name field type **F28069_ctrlSTK.ccxml**. This is just a descriptive name since multiple target configuration files can be created. Leave the “Use shared location” box checked and select **Finish**.

4. In the next window that appears, select the emulator using the “Connection” pull-down list and choose “Texas Instruments XDS100v1 USB Emulator”. In the “Board or Device” box type **F28069** to filter the options. In the box below, check the box to select “controlSTICK – Piccolo F28069”. Click **Save** to save the configuration, then close the “F28069_ctrlSTK.ccxml” setup window by clicking the X on the tab.

5. To view the target configurations, click:

View → Target Configurations

and click the plus sign (+) to the left of **User Defined**. Notice that the **F28069_ctrlSTK.ccxml** file is listed and set as the default. If it is not set as the default, right-click on the .ccxml file and select “Set as Default”. Close the Target Configurations window by clicking the X on the tab.

Create a New Project

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

File → New → CCS Project

In the Project name field type **Lab1**. Uncheck the “Use default location” box. Click the **Browse...** button and navigate to:

C:\C28x\Labs\Lab1\Project

Click OK.

7. The next section selects the device. Select the “Family” using the pull-down list and choose “C2000”. Set the “Variant” filter using the pull-down list to “2806x Piccolo” and choose the “controlSTICK – Piccolo F28069”. Leave the “Connection” box blank. We have already set up the target configuration.
8. Next, open the “Advanced setting” section and set the “Linker command file” to “<none>”. We will be using our own linker command file, rather than the one supplied by CCS. Leave the “Runtime Support Library” set to “<automatic>”. This will automatically select the “rts2800_fpu32.lib” runtime support library for floating-point devices.
9. Now open the “Project templates and examples” section and select the very top “Empty Project” template. (Note: Do not select the second one from the top - this option will create an empty main.c file in the project, which is not needed for this lab exercise). Click Finish.
10. A new project has now been created. Notice the Project Explorer window contains Lab1. The project is set Active and the output files will be located in the Debug folder. At this point, the project does not include any source files. The next step is to add the source files to the project.
11. To add the source files to the project, right-click on Lab1 in the Project Explorer window and select:
Add Files...
or click: Project → Add Files...

and make sure you’re looking in C:\C28x\Labs\Lab1\Files. With the “files of type” set to view all files (*.*) select Lab1.c and Lab1.cmd then click OPEN. A “File Operation” window will open, choose “Copy files” and click OK. This will add the files to the project.
12. In the Project Explorer window, click the plus sign (+) to the left of Lab1 and notice that the files are listed.

Project Build Options

13. There are numerous build options in the project. Most default option settings are sufficient for getting started. We will inspect a couple of the default options at this time. Right-click on Lab1 in the Project Explorer window and select Properties or click:
Project → Properties
14. A “Properties” window will open and in the section on the left under “Build” be sure that the “C2000 Compiler” and “C2000 Linker” options are visible. Next, under “C2000 Linker” select the “Basic Options”. Notice that .out and .map files are being specified. The .out file is the executable code that will be loaded into the MCU. The .map file will contain a linker report showing memory usage and section addresses in memory.
15. Next in the “Basic Options” set the Stack Size to **0x200**.

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

Linker Command File – Lab1.cmd

17. Open and inspect `Lab1.cmd` by double clicking on the filename in the project window. Notice that the `Memory{ }` declaration describes the system memory shown on the “Lab1: Linker Command File” slide in the objective section of this lab exercise. Memory blocks `L3DPSARAM` and `L4SARAM` have been placed in program memory on page 0, and the other memory blocks have been placed in data memory on page 1.
18. In the `Sections{ }` area notice that the sections defined on the slide have been “linked” into the appropriate memories. Also, notice that a section called `.reset` has been allocated. The `.reset` section is part of the `rts2800_fpu32.lib` and is not needed. By putting the `TYPE = DSECT` modifier after its allocation the linker will ignore this section and not allocate it. Close the inspected file.

Build and Load the Project

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



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

20. Click the “Build” button and watch the tools run in the `Console` window. Check for errors in the `Problems` window (we have deliberately put an error in `Lab1.c`). When you get an error, you will see the error message in the `Problems` window. Expand the problem by clicking on the plus sign (+) to the left of the “Errors”. Then simply double-click the error message. The editor will automatically open to the source file containing the error, with the code line highlighted with a question mark (?).
21. Fix the error by adding a semicolon at the end of the “`z = x + y`” statement. For future knowledge, realize that a single code error can sometimes generate multiple error messages at build time. This was not the case here.
22. Build the project again. There should be no errors this time.
23. CCS can automatically save modified source files, build the project, open the debug perspective view, connect and download it to the target, and then run the program to the beginning of the main function.

Click on the “Debug” button (green bug) or click `Run` → `Debug`.

Notice the CCS `Debug` icon in the upper right-hand corner indicating that we are now in the “CCS Debug Perspective” view. The program ran through the C-environment initialization routine in the `rts2800_fpu32.lib` and stopped at `main()` in `Lab1.c`.

Debug Environment Windows

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

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

Click: View → Memory Browser on the menu bar.

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

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

25. Notice the “Variables” window automatically opened and the local variables `x` and `y` are present. The variables window will always contain the local variables for the code function currently being executed.

(Note that local variables actually live on the stack. You can also view local variables in a memory browser by setting the address to “SP” after the code function has been entered).

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

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


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

Single-stepping the Code

27. Click the “Variables” tab at the top of the window to watch the local variables. Single-step through `main()` by using the <F5> key (or you can use the Step Into button on the horizontal toolbar). Check to see if the program is working as expected. What is the value for “z” when you get to the end of the program?

Terminate Debug Session and Close Project

28. 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: 

29. Next, close the project by right-clicking on Lab1 in the Project Explorer window and select Close Project.

End of Exercise

Peripheral Register Header Files

Traditional Approach to C Coding

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

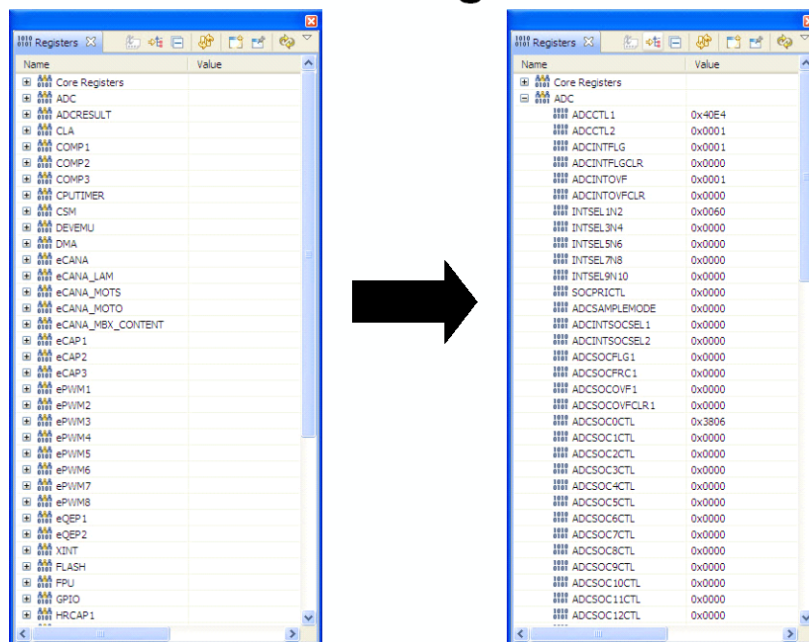
- Advantages**
- Simple, fast and easy to type
 - Variable names exactly match register names (easy to remember)
- Disadvantages**
- Requires individual masks to be generated to manipulate individual bits
 - Cannot easily display bit fields in debugger window
 - Will generate less efficient code in many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

Structure Approach to C Coding

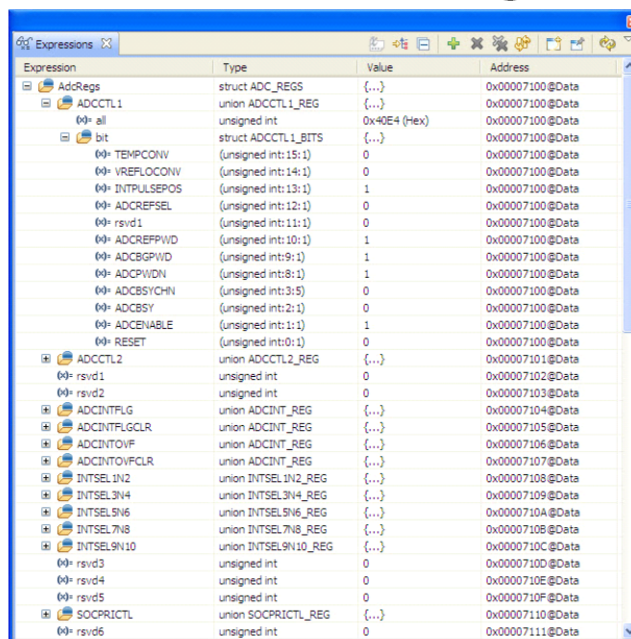
```
void main(void)
{
    AdcRegs.ADCCTL1.all = 0x1234;           //write entire register
    AdcRegs.ADCCTL1.bit.ADCENABLE = 1;     //enable ADC module
}
```

- Advantages**
- Easy to manipulate individual bits
 - Watch window is amazing! (next slide)
 - Generates most efficient code (on C28x)
- Disadvantages**
- Can be difficult to remember the structure names (Editor Auto Complete feature to the rescue!)
 - More to type (again, Editor Auto Complete feature to the rescue)

Built-in CCSv5 Register Window



CCSv5 Watch Window using Structures



Structure Naming Conventions

◆ The F2806x header files define:

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

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

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

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

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

Editor Auto Complete to the Rescue!

```

20
21//--- Reset the ADC module
22// Note: The ADC is already reset after a DSP reset, but this example is just showing
23// good coding practice to reset the peripheral before configuring it as you never
24// know why the DSP has started the code over again from the beginning).
25  AdcRegs.ADCCTL1.bit.RESET = 1;    // Reset the ADC
26
27// Must wait 2 ADCCLK periods for the reset to take effect.
28// Note that ADCCLK = SYSCLKOUT for F2806x devices.
29  asm("NOP");
30  asm("NOP");
31
32  AdcRegs.ADCCTL1.bit.RESET = 1;
33
34
35//--- Power-up and configure the ADC
36  AdcRegs.ADCCTL1.all = 0x00E4;    // Power-up reference and main ADC
37// bit 15      0:   RESET, ADC software reset, 0=no effect, 1=resets the ADC
38// bit 14      0:   ADCENABLE, ADC enable, 0=disabled, 1=enabled
39// bit 13      0:   ADCBSY, ADC busy, read-only
40// bit 12-8    0's:  ADCBSYCHM, ADC busy channel, read-only
41// bit 7       1:   ADCBFWN, ADC power down, 0=powered down, 1=powered up
42// bit 6       1:   ADCBFPWD, ADC bandgap power down, 0=powered down, 1=powered up
43// bit 5       1:   ADCREFPWD, ADC reference power down, 0=powered down, 1=powered up
44// bit 4       0:   reserved
45// bit 3       0:   ADCREFSEL, ADC reference select, 0=internal, 1=external
46// bit 2       1:   INTFULSEPOS, INT pulse generation, 0=start of conversion, 1=end of conversion
47// bit 1       0:   VREFLOCONV, VREFLO convert, 0=VREFLO not connected, 1=VREFLO connected to B5
48// bit 0       0:   TEMPCONV, Temperature sensor convert. 0=ADCINAS is pin, 1=ADCINAS is temp sensor
49
50  AdcRegs.ADCCLK2.all = 0x0001;    // ADC clock configuration
51// bit 15-3    0's:  reserved
52// bit 2       0:   CLMDIV4EN, ADC clock divider. 0=no effect, 1=CPUCCLK/4 if CLMDIV2EN=1 (else no effect)
53// bit 1       0:   ADCNONOVERLAP, 0=overlap sample and conversion, 1=no overlap
54// bit 0       1:   CLMDIV2EN, ADC clock divider. 0=CPUCCLK, 1=CPUCCLK/2
55

```

F2806x Header File Package

(<http://www.ti.com, controlSUITE>)

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

◆ \F2806x_headers\include	→ .h files
◆ \F2806x_headers\cmd	→ linker .cmd files
◆ \F2806x_common\gel	→ .gel files for CCS
◆ \F2806x_examples	→ code examples
◆ \doc	→ documentation

controlSUITE Header File Package located at C:\TI\controlSUITE\device_support\

Peripheral Structure .h files (1 of 2)

- ◆ Contain bits field structure definitions for each peripheral register

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

```
#include "F2806x_Device.h"

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

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

F2806x_Adc.h

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

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

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

Peripheral Structure .h files (2 of 2)

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

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

- ◆ **F2806x_Device.h**
 - ◆ Main include file
 - ◆ Will include all other .h files
 - ◆ Include this file (*directly or indirectly*) in each source file:

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

Global Variable Definitions File

F2806x_GlobalVariableDefs.c

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

F2806x_GlobalVariableDefs.c

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

- ◆ Add this file to your CCS project:

F2806x_GlobalVariableDefs.c

Linker Command Files for the Structures

F2806x_nonBIOS.cmd and F2806x_BIOS.cmd

F2806x_GlobalVariableDefs.c

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

- ◆ Links each structure to the address of the peripheral using the structures named section

F2806x_Headers_nonBIOS.cmd

```
MEMORY
{
  PAGE1:
  ...
  ADC:  origin=0x007100, length=0x000080
  ...
}
SECTIONS
{
  ...
  AdcRegsFile:  > ADC      PAGE = 1
  ...
}
```

- ◆ non-BIOS and BIOS versions of the .cmd file

- ◆ Add one of these files to your CCS project:

F2806x_nonBIOS.cmd

or

F2806x_BIOS.cmd

Peripheral Specific Examples

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

adc_soc	eqep_pos_speed	mcbasp_loopback_interrupts
adc_temp_sensor	external_interrupt	mcbasp_spi_loopback
adc_temp_sensor_conv	flash_f28069	osc_comp
cla_adc	fpu_hardware	sci_echoback
cla_adc_fir	fpu_software	scia_loopback
cla_adc_fir_flash	gpio_setup	scia_loopback_interrupts
cpu_timer	gpio_toggle	spi_loopback
dma_ram_to_ram	hrcap_capture_hrpwm	spi_loopback_interrupts
ecan_back2back	hrcap_capture_pwm	sw_prioritized_interrupts
ecap_apwm	hrpwm	timed_led_blink
ecap_capture_pwm	hrpwm_duty_sfo_v6	usb_dev_bulk
epwm_blanking_window	hrpwm_mult_ch_prdudown_sfo_v6	usb_dev_chidcdc
epwm_dcevent_trip	hrpwm_prdup_sfo_v6	usb_dev_keyboard
epwm_dcevent_trip_comp	hrpwm_prdudown_sfo_v6	usb_dev_mouse
epwm_deadband	hrpwm_slider	usb_dev_serial
epwm_real-time_interrupts	i2c_eeprom	usb_host_keyboard
epwm_timer_interrupts	lpm_haltwake	usb_host_mouse
epwm_trip_zone	lpm_idlewake	usb_host_msc
epwm_up_aq	lpm_standbywake	watchdog
epwm_updown_aq	mcbasp_loopback	
eqep_freqcal	mcbasp_loopback_dma	

Peripheral Register Header Files Summary

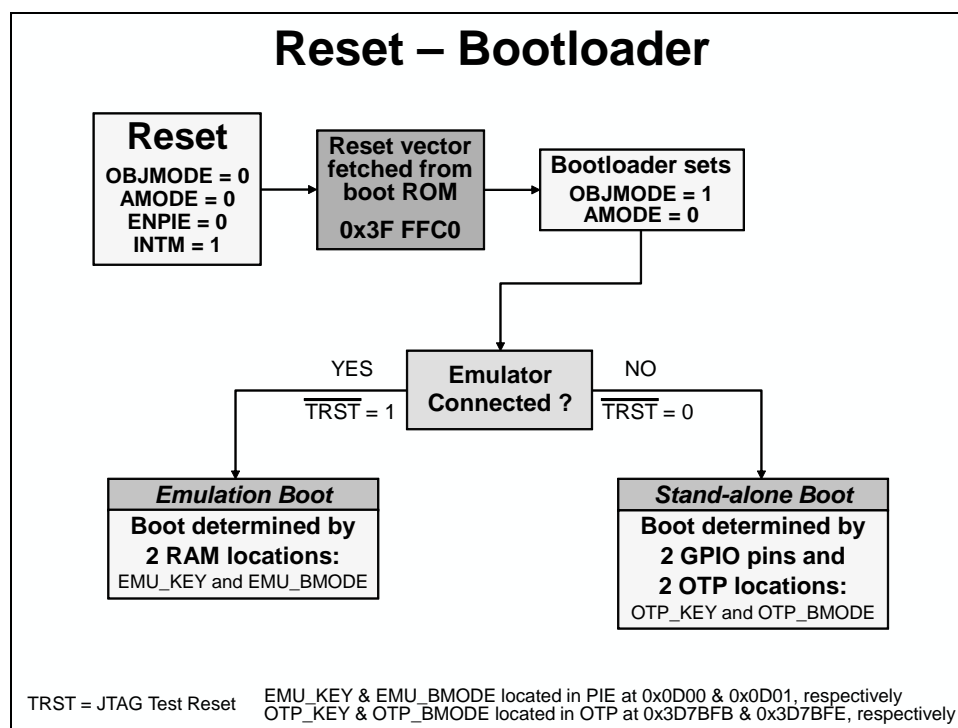
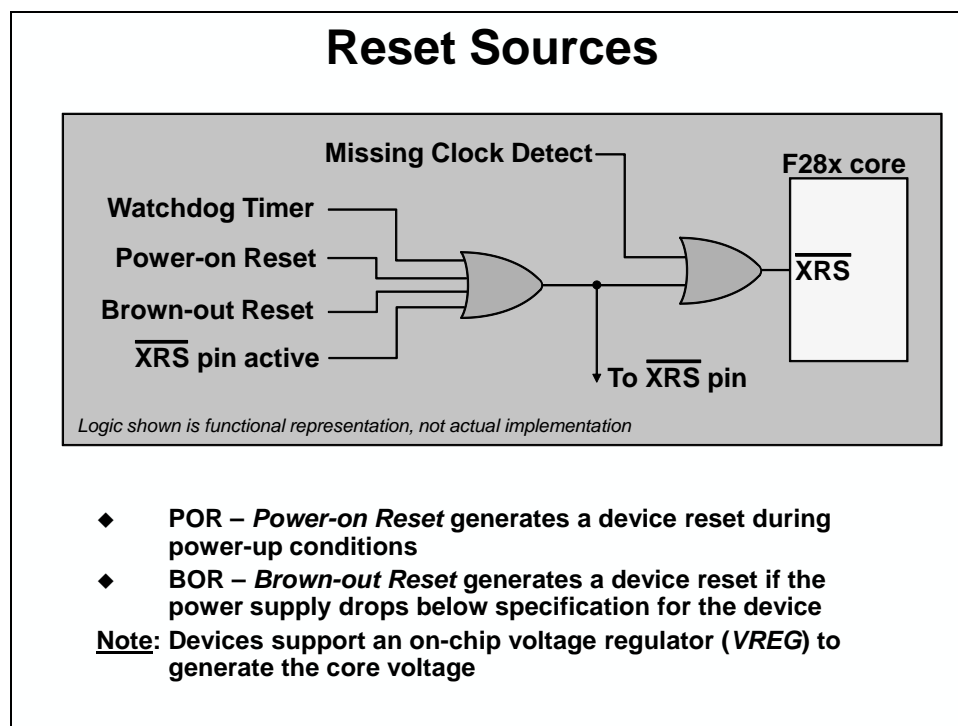
- ◆ Easier code development
- ◆ Easy to use
- ◆ Generates most efficient code
- ◆ Increases effectiveness of CCS watch window
- ◆ TI has already done all the work!
 - ◆ Use the correct header file package for your device:

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

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

Reset, Interrupts and System Initialization

Reset



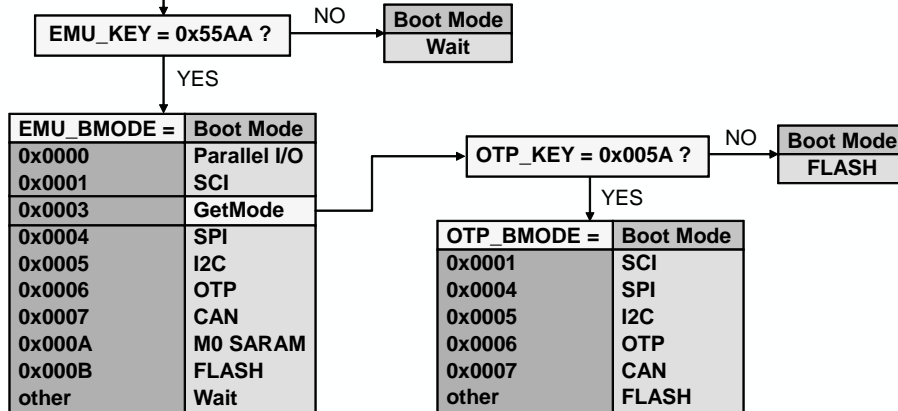
Emulation Boot Mode ($\overline{\text{TRST}} = 1$)

Emulator Connected

Emulation Boot

Boot determined by
2 RAM locations:
EMU_KEY and EMU_BMODE

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



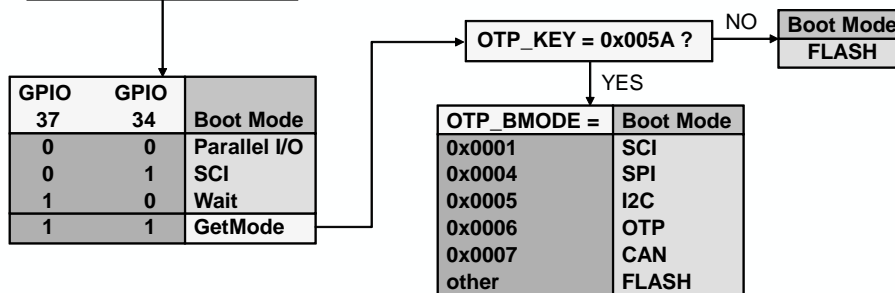
Stand-Alone Boot Mode ($\overline{\text{TRST}} = 0$)

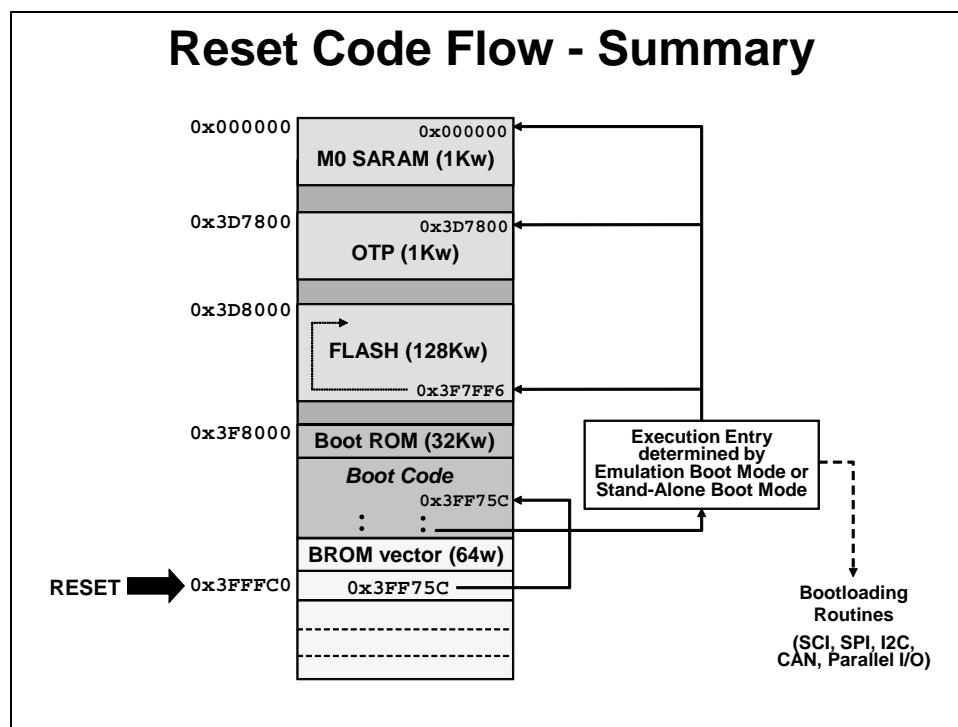
Emulator Not Connected

Stand-alone Boot

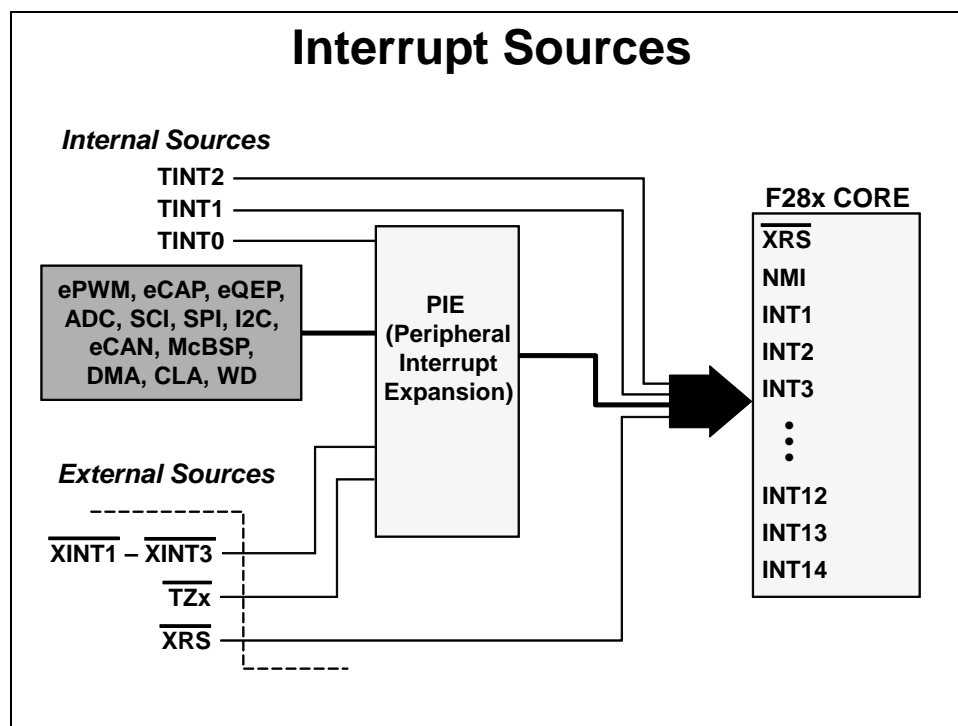
Boot determined by
2 GPIO pins and
2 OTP locations:
OTP_KEY and OTP_BMODE

Note that the boot behavior for unprogrammed OTP is the "FLASH" boot mode



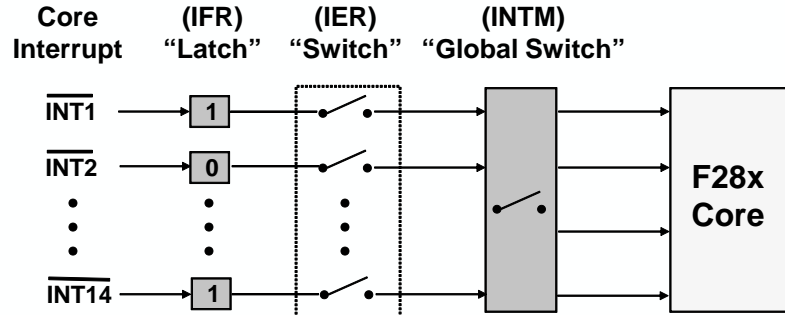


Interrupts



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

INTM

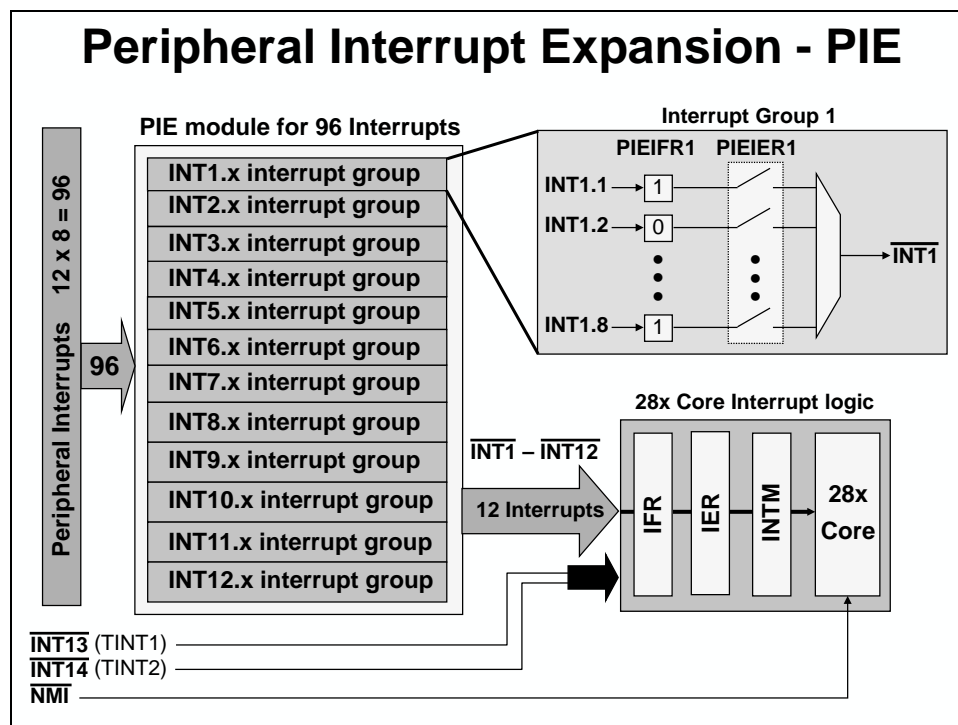
(enable = 0 / disable = 1)

```

/** Interrupt Enable Register */
extern register volatile unsigned int IER;
IER |= 0x0008;           //enable INT4 in IER
IER &= 0xFF7;           //disable INT4 in IER
/** Global Interrupts */
asm(" CLRC INTM");       //enable global interrupts
asm(" SETC INTM");       //disable global interrupts

```

Peripheral Interrupt Expansion (PIE)



F2806x PIE Interrupt Assignment Table

	INTx.8	INTx.7	INTx.6	INTx.5	INTx.4	INTx.3	INTx.2	INTx.1
INT1	WAKEINT	TINT0	ADCINT9	XINT2	XINT1		ADCINT2	ADCINT1
INT2	EPWM8_TZINT	EPWM7_TZINT	EPWM6_TZINT	EPWM5_TZINT	EPWM4_TZINT	EPWM3_TZINT	EPWM2_TZINT	EPWM1_TZINT
INT3	EPWM8_INT	EPWM7_INT	EPWM6_INT	EPWM5_INT	EPWM4_INT	EPWM3_INT	EPWM2_INT	EPWM1_INT
INT4	HRCAP2_INT	HRCAP1_INT				ECAP3_INT	ECAP2_INT	ECAP1_INT
INT5				HRCAP4_INT	HRCAP3_INT		EQEP2_INT	EQEP1_INT
INT6			MXINTA	MRINTA	SPITX_INTB	SPIRX_INTB	SPITX_INTA	SPIRX_INTA
INT7			DINTCH6	DINTCH5	DINTCH4	DINTCH3	DINTCH2	DINTCH1
INT8							I2CINT2A	I2CINT1A
INT9			ECAN1_INTA	ECAN0_INTA	SCITX_INTB	SCIRX_INTB	SCITX_INTA	SCIRX_INTA
INT10	ADCINT8	ADCINT7	ADCINT6	ADCINT5	ADCINT4	ADCINT3	ADCINT2	ADCINT1
INT11	CLA1_INT8	CLA1_INT7	CLA1_INT6	CLA1_INT5	CLA1_INT4	CLA1_INT3	CLA1_INT2	CLA1_INT1
INT12	LUF	LVF						XINT3

PIE Registers

PIEIFRx register (x = 1 to 12)

15-8	7	6	5	4	3	2	1	0
reserved	INTx.8	INTx.7	INTx.6	INTx.5	INTx.4	INTx.3	INTx.2	INTx.1

PIEIERx register (x = 1 to 12)

15-8	7	6	5	4	3	2	1	0
reserved	INTx.8	INTx.7	INTx.6	INTx.5	INTx.4	INTx.3	INTx.2	INTx.1

PIE Interrupt Acknowledge Register (PIEACK)

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

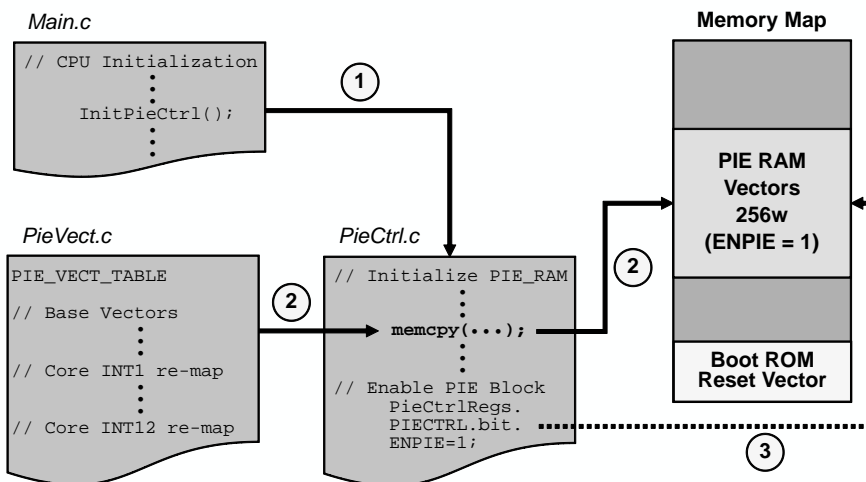
PIECTRL register

15-1	0
PIEVECT	ENPIE

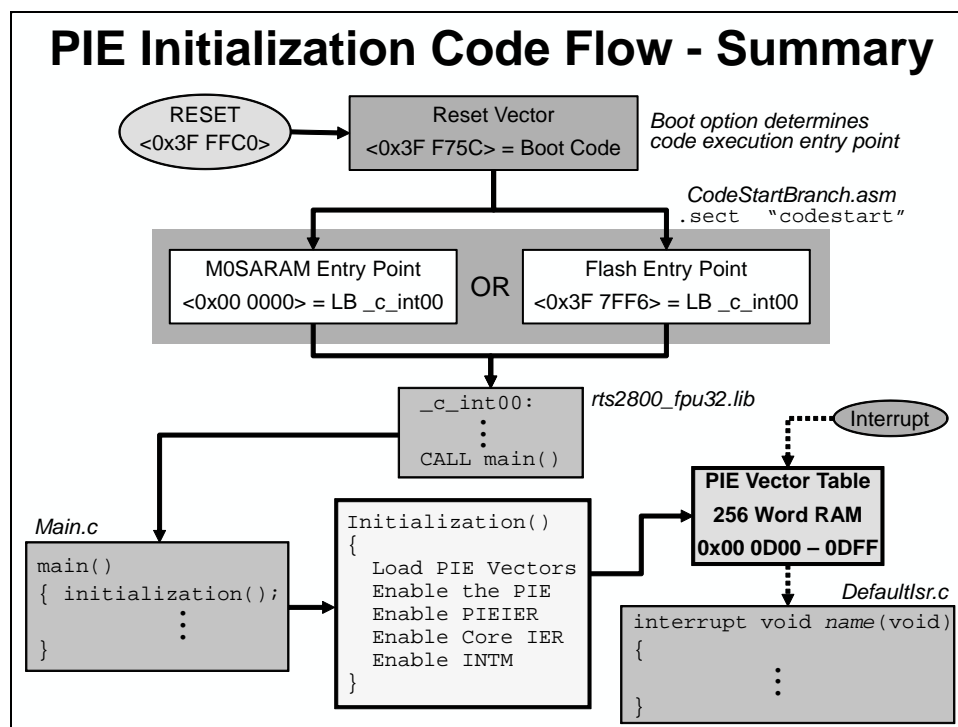
```
#include "F2806x_Device.h"
```

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

PIE Block Initialization



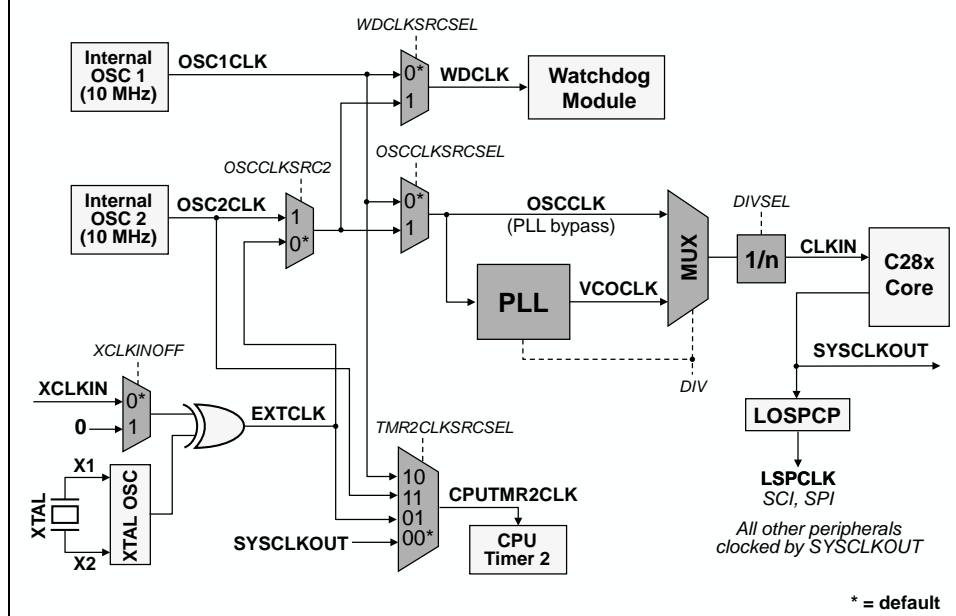
PIE Initialization Code Flow - Summary

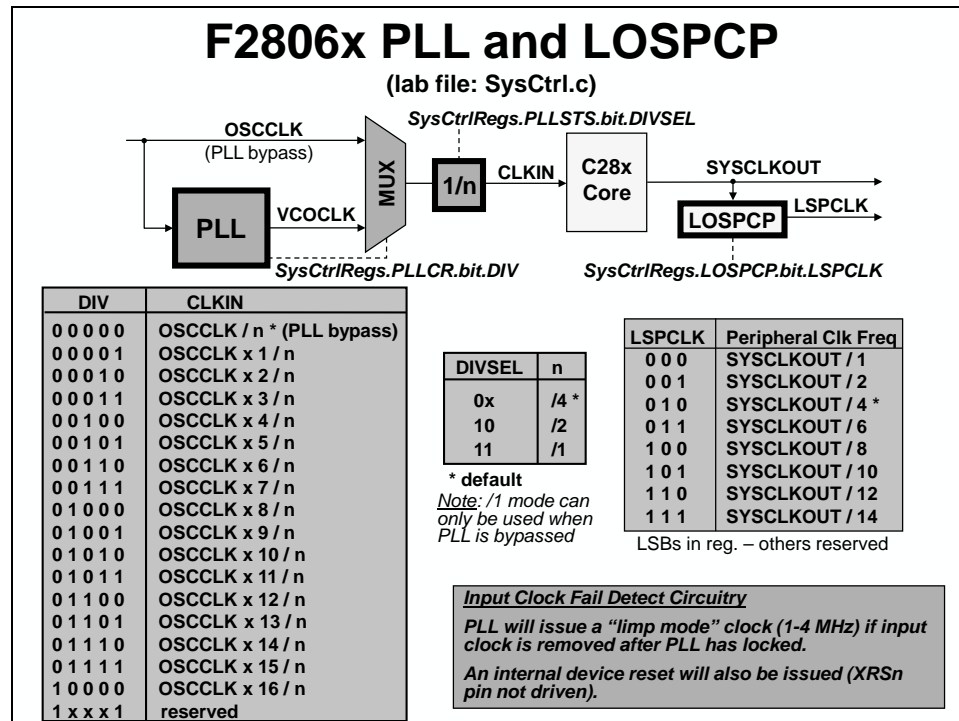


Oscillator / PLL Clock Module

F2806x Oscillator / PLL Clock Module

(lab file: SysCtrl.c)

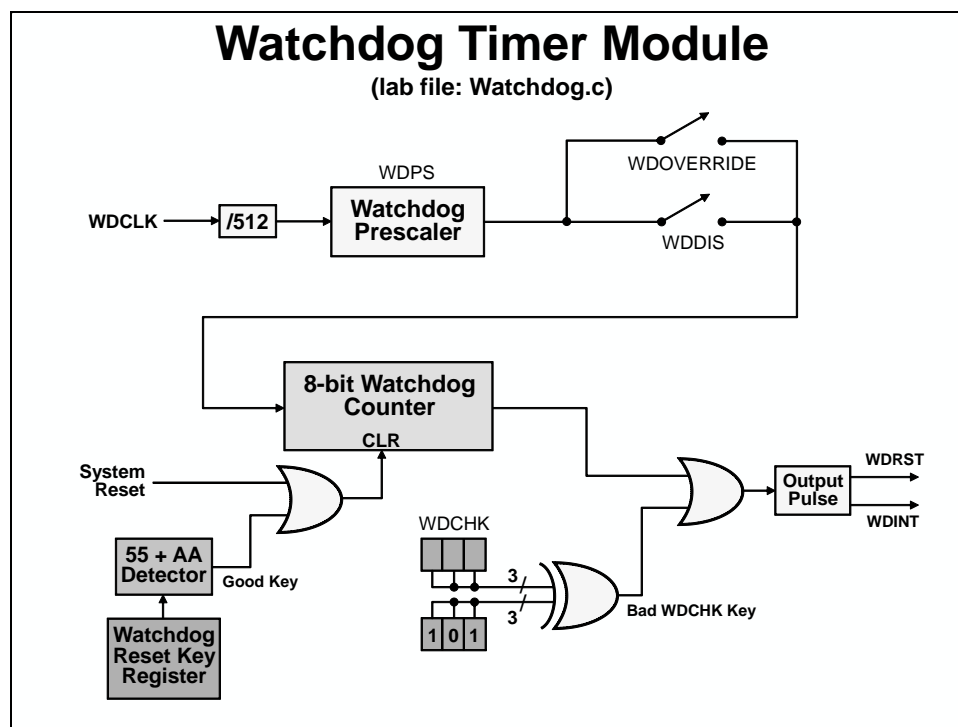




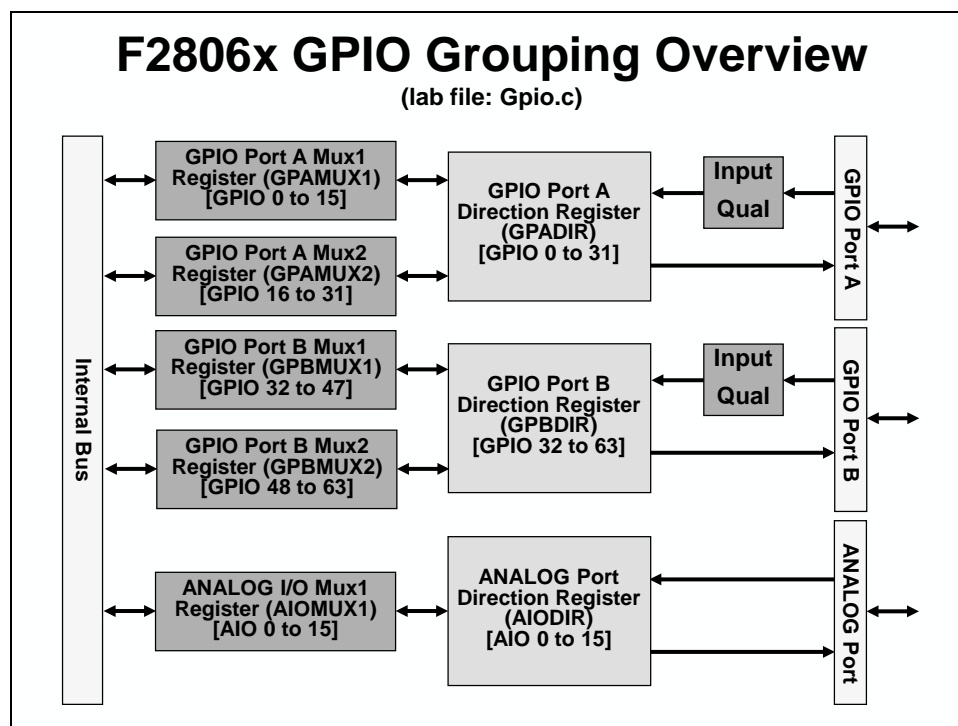
Watchdog Timer Module

Watchdog Timer

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



GPIO





Lab 2: System Initialization

- ◆ LAB2 files have been provided
- ◆ LAB2 consists of two parts:
 - Part 1
 - ◆ Test behavior of watchdog when disabled and enabled
 - Part 2
 - ◆ Initialize peripheral interrupt expansion (PIE) vectors and use watchdog to generate an interrupt
- ◆ Modify, build, and test code using Code Composer Studio

Lab 2: System Initialization

➤ Objective

The objective of this lab is to perform the processor system initialization. Additionally, the peripheral interrupt expansion (PIE) vectors will be initialized and tested. The system initialization for this lab will consist of the following:

- Setup the clock module – PLL, LOSPCP = /4, low-power modes to default values, enable all module clocks
- Disable the watchdog – clear WD flag, disable watchdog, WD prescale = 1
- Setup the watchdog and system control registers – DO NOT clear WD OVERRIDE bit, configure WD to generate a CPU reset
- Setup the shared I/O pins – set all GPIO pins to GPIO function (e.g. a "00" setting for GPIO function, and a "01", "10", or "11" setting for peripheral function)

The first part of the lab exercise will setup the system initialization and test the watchdog operation by having the watchdog cause a reset. In the second part of the lab exercise the PIE vectors will be tested by using the watchdog to generate an interrupt. This lab will make use of the F2806x C-code header files to simplify the programming of the device, as well as take care of the register definitions and addresses. Please review these files, and make use of them in the future, as needed.

➤ Procedure

Open the Project

1. A project named Lab2 has been created for this lab. Open the project by clicking on **Project** → **Import Existing CCS/CCE Eclipse Project**. The "Import" window will open then click **Browse...** next to the "Select search-directory" box. Navigate to: `C:\C28x\Labs\Lab2\Project` and click **OK**. Then click **Finish** to import the project.
2. In the **Project Explorer** window, click the plus sign (+) to the left of Lab2 to view the project files. All Build Options have been configured for this lab. The files used in this lab are:

CodeStartBranch.asm	Lab.h
DefaultIsr_2.c	Lab_2_3.cmd
DelayUs.asm	Main_2.c
F2806x_DefaultIsr.h	PieCtrl.c
F2806x_GlobalVariableDefs.c	PieVect.c
F2806x_Headers_nonBIOS.cmd	SysCtrl.c
Gpio.c	Watchdog.c

Modified Memory Configuration

3. Open and inspect the linker command file `Lab_2_3.cmd`. Notice that the user defined section "codestart" is being linked to a memory block named `BEGIN_M0`. The codestart section contains code that branches to the code entry point of the project. The bootloader must branch to the codestart section at the end of the boot process. Recall that the emulation boot mode "M0 SARAM" branches to address `0x000000` upon bootloader completion.

The linker command file (`Lab_2_3.cmd`) has a new memory block named `BEGIN_M0`: origin = `0x000000`, length = `0x0002`, in program memory. Additionally, the existing memory block `M0SARAM` in data memory has been modified to avoid overlaps with this new memory block.

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

System Initialization

5. Open and inspect `SysCtrl.c`. Notice that the PLL and module clocks have been enabled.
6. Open and inspect `Watchdog.c`. Notice that the watchdog control register (`WDCR`) is configured to disable the watchdog, and the system control and status register (`SCSR`) is configured to generate a reset.
7. Open and inspect `Gpio.c`. Notice that the shared I/O pins have been set to the GPIO function, except for `GPIO0` which will be used in the next lab exercise. Close the inspected files.

Build and Load

8. Click the "Build" button and watch the tools run in the `Console` window. Check for errors in the `Problems` window.
9. Click the "Debug" button (green bug). The "CCS Debug Perspective" view should open, the program will load automatically, and you should now be at the start of `main()`.
10. After CCS loaded the program in the previous step, it set the program counter (PC) to point to `_c_int00`. It then ran through the C-environment initialization routine in the `rt2800_fpu32.lib` and stopped at the start of `main()`. CCS did not do a device reset, and as a result the bootloader was bypassed.

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

Scripts → EMU Boot Mode Select → `EMU_BOOT_SARAM`

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

Run the Code – Watchdog Reset

11. Place the cursor in the “main loop” section (on the `asm(" NOP");` instruction line) and right click the mouse key and select `Run To Line`. This is the same as setting a breakpoint on the selected line, running to that breakpoint, and then removing the breakpoint.
12. Place the cursor on the first line of code in `main()` and set a breakpoint by double clicking in the line number field to the left of the code line. Notice that line is highlighted with a blue dot indicating that the breakpoint has been set. The breakpoint is set to prove that the watchdog is disabled. If the watchdog causes a reset, code execution will stop at this breakpoint.
13. Run your code for a few seconds by using the “Resume” button on the toolbar, or by using `Run → Resume` on the menu bar (or F8 key). After a few seconds halt your code by using the “Suspend” button on the toolbar, or by using `Run → Suspend` on the menu bar (or alt-F8 key). Where did your code stop? Are the results as expected? If things went as expected, your code should be in the “main loop”.
14. Switch to the “CCS Edit Perspective” view by clicking the `CCS Edit` icon in the upper right-hand corner. Modify the `InitWatchdog()` function to enable the watchdog (WDCR). In `Watchdog.c` change the WDCR register value to `0x00A8`. This will enable the watchdog to function and cause a reset. Save the file.
15. Click the “Build” button. Select `Yes` to “Reload the program automatically”. Switch back to the “CCS Debug Perspective” view by clicking the `CCS Debug` icon in the upper right-hand corner.
16. Like before, place the cursor in the “main loop” section (on the `asm(" NOP");` instruction line) and right click the mouse key and select `Run To Line`.
17. Run your code. Where did your code stop? Are the results as expected? If things went as expected, your code should have stopped at the breakpoint. What happened is as follows. While the code was running, the watchdog timed out and reset the processor. The reset vector was then fetched and the ROM bootloader began execution. Since the device is in emulation boot mode (i.e. the emulator is connected) the bootloader read the `EMU_KEY` and `EMU_BMODE` values from the PIE RAM. These values were previously set for boot to M0 SARAM bootmode by CCS. Since these values did not change and are not affected by reset, the bootloader transferred execution to the beginning of our code at address `0x000000` in the M0SARAM, and execution continued until the breakpoint was hit in `main()`.

Setup PIE Vector for Watchdog Interrupt

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

18. Switch to the “CCS Edit Perspective” view by clicking the CCS `Edit` icon in the upper right-hand corner. Notice that the following files are included in the project:

```
DefaultIsr_2.c
PieCtrl.c
PieVect.c
```

19. In `Main_2.c`, uncomment the code used to call the `InitPieCtrl()` function. There are no passed parameters or return values, so the call code is simply:

```
InitPieCtrl();
```

20. Using the “PIE Interrupt Assignment Table” shown in the slides find the location for the watchdog interrupt, “WAKEINT”. This is used in the next step.

PIE group #:_____ # within group:_____

21. In `main()` notice the code used to enable global interrupts (INTM bit), and in `InitWatchdog()` the code used to enable the “WAKEINT” interrupt in the PIE (using the `PieCtrlRegs` structure) and to enable core INT1 (IER register).
22. Modify the system control and status register (SCSR) to cause the watchdog to generate a WAKEINT rather than a reset. In `Watchdog.c` change the SCSR register value to **0x0002**. Save the modified files.
23. Open and inspect `DefaultIsr_2.c`. This file contains interrupt service routines. The ISR for WAKEINT has been trapped by an emulation breakpoint contained in an inline assembly statement using “ESTOP0”. This gives the same results as placing a breakpoint in the ISR. We will run the lab exercise as before, except this time the watchdog will generate an interrupt. If the registers have been configured properly, the code will be trapped in the ISR.
24. Open and inspect `PieCtrl.c`. This file is used to initialize the PIE RAM and enable the PIE. The interrupt vector table located in `PieVect.c` is copied to the PIE RAM to setup the vectors for the interrupts. Close the modified and inspected files.

Build and Load

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

Run the Code – Watchdog Interrupt

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

Terminate Debug Session and Close Project

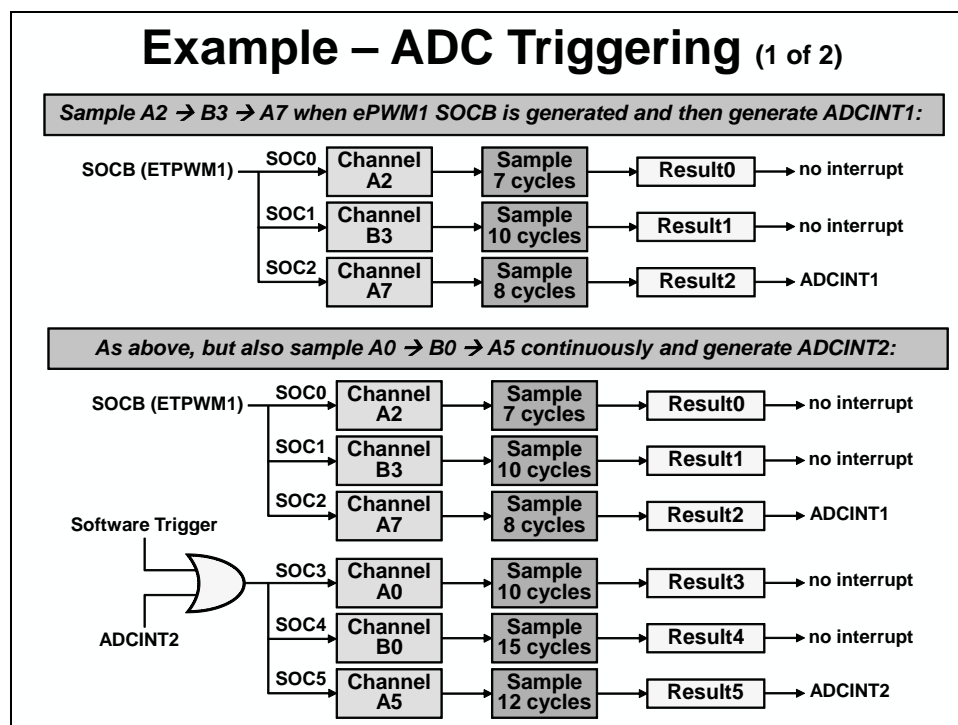
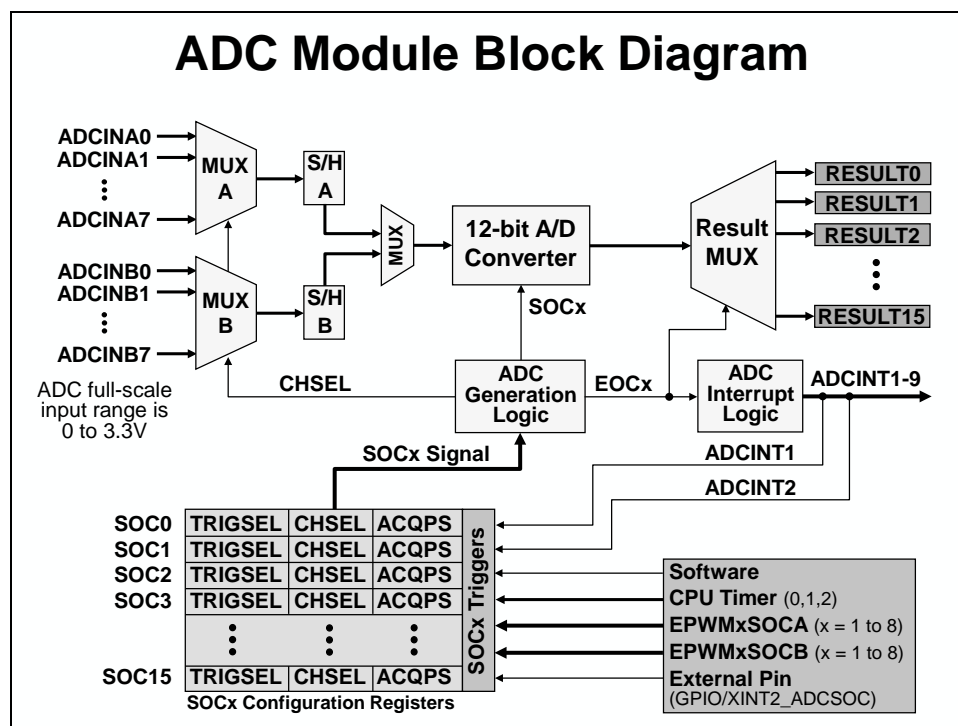
28. Terminate the active debug session using the `Terminate` button. This will close the debugger and return CCS to the “CCS Edit Perspective” view.
29. Next, close the project by right-clicking on `Lab2` in the `Project Explorer` window and select `Close Project`.

End of Exercise

Note: By default, the watchdog timer is enabled out of reset. Code in the file `CodeStartBranch.asm` has been configured to disable the watchdog. This can be important for large C code projects (ask your instructor if this has not already been explained). During this lab exercise, the watchdog was actually re-enabled (or disabled again) in the file `Watchdog.c`.

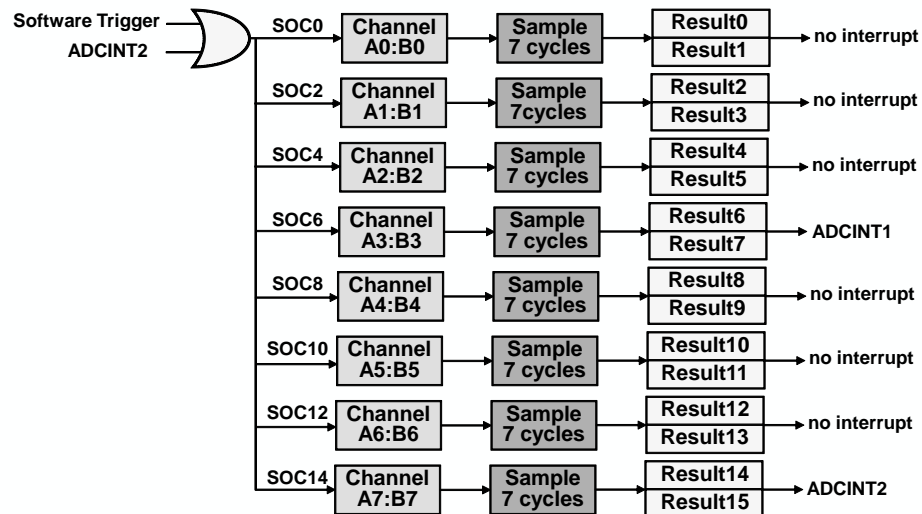
Control Peripherals

ADC Module

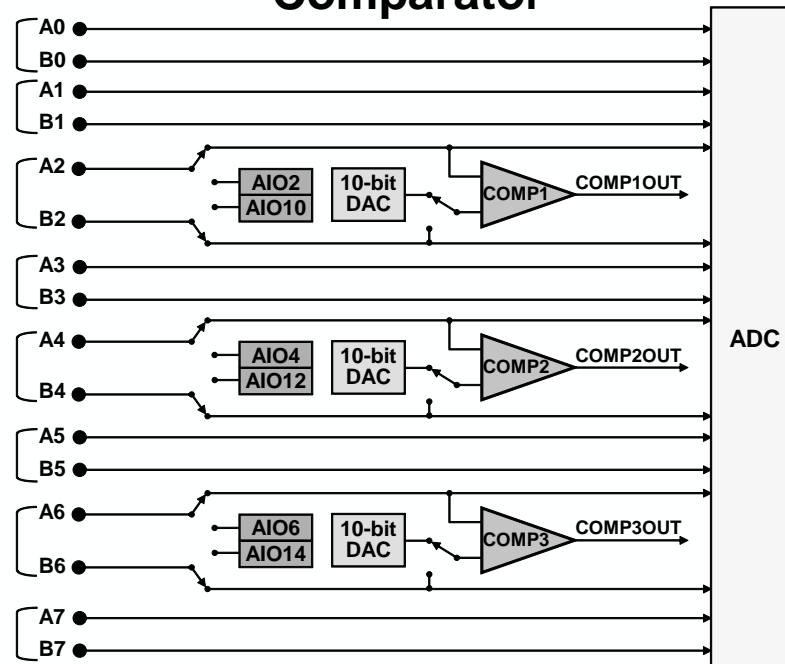


Example – ADC Triggering (2 of 2)

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



Comparator



ADC Control Registers (file: Adc.c)

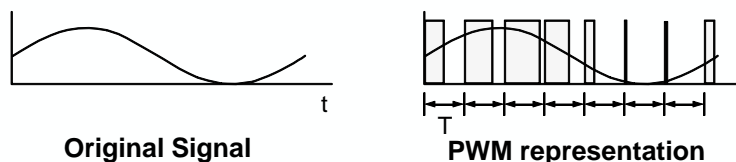
- ◆ **ADCTRL1** (ADC Control Register 1)
 - ◆ module reset, ADC enable
 - ◆ busy/busy channel
 - ◆ reference select
 - ◆ Interrupt generation control
- ◆ **ADCxSCTL** (SOC0 to SOC15 Control Registers)
 - ◆ trigger source
 - ◆ channel
 - ◆ acquisition sampling window
- ◆ **ADCINTSSELx** (Interrupt SOC Selection 1 and 2 Registers)
 - ◆ selects ADCINT1 / ADCINT2 trigger for SOCx
- ◆ **ADCxSMR** (Sampling Mode Register)
 - ◆ sequential sampling / simultaneous sampling
- ◆ **INTSELxNy** (Interrupt x and y Selection Registers)
 - ◆ EOC0 – EOC15 source select for ADCINT1-9
- ◆ **ADCRESULTx** (ADC Result 0 to 15 Registers)

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

Pulse Width Modulation

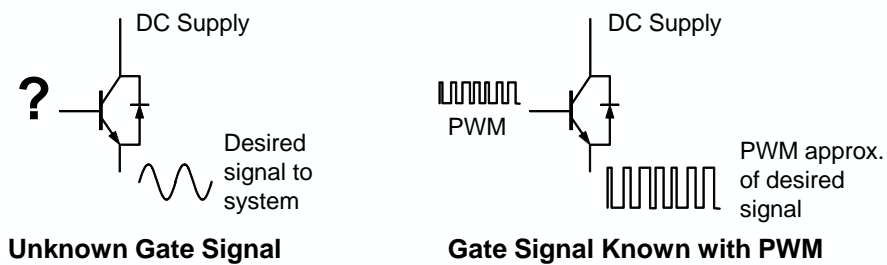
What is Pulse Width Modulation?

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



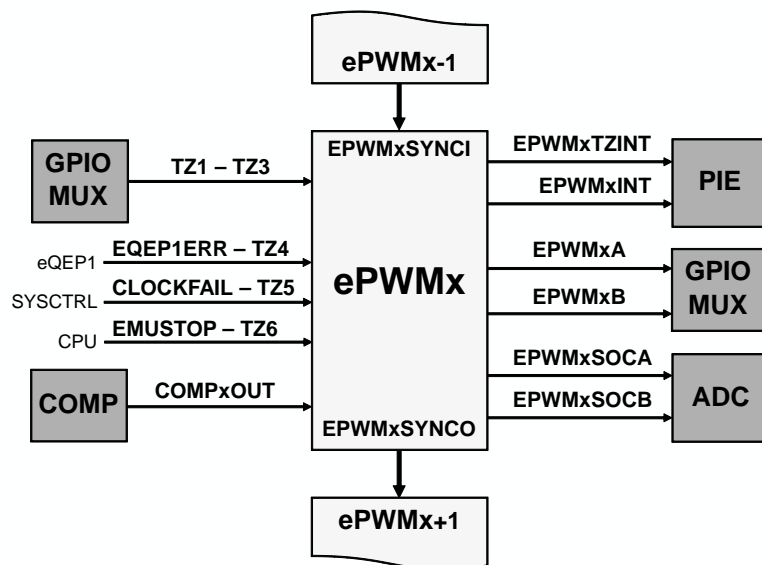
Why use PWM with Power Switching Devices?

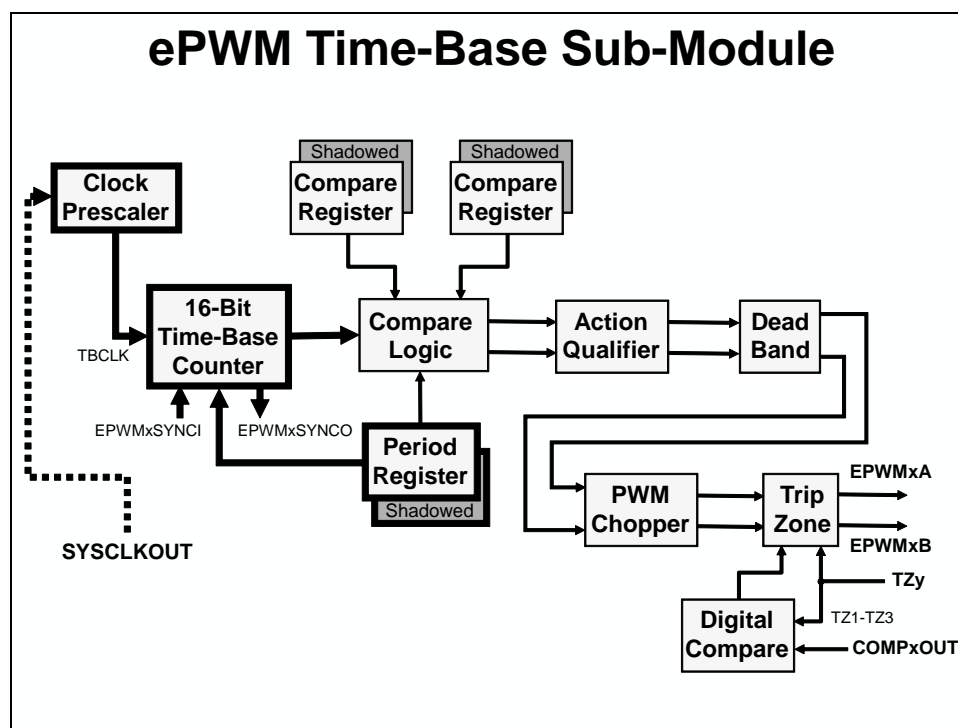
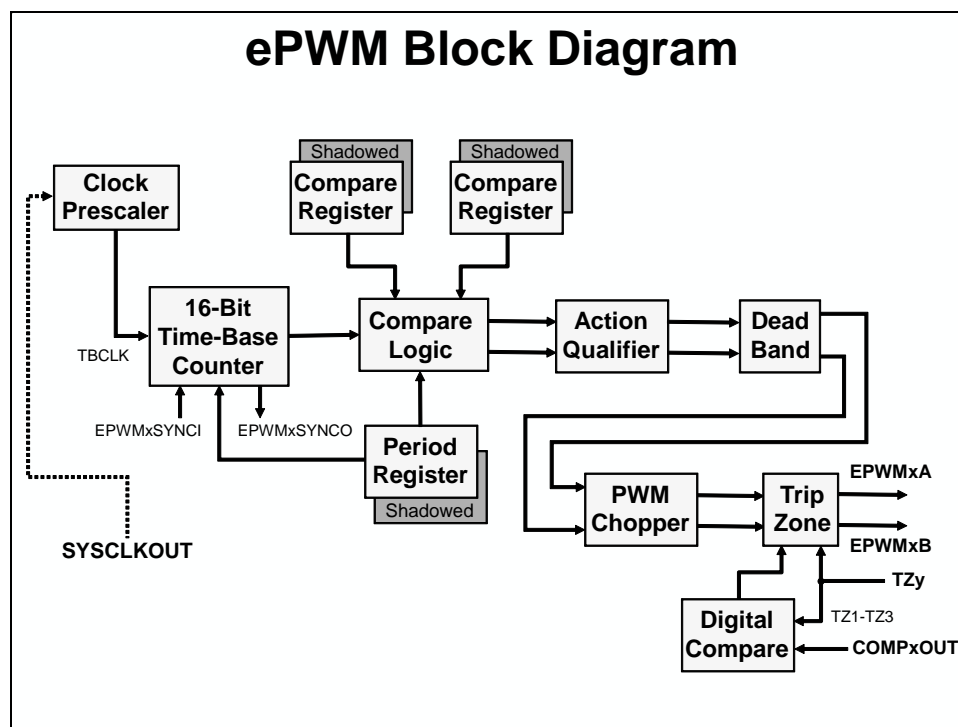
- ◆ Desired output currents or voltages are known
- ◆ Power switching devices are transistors
 - ◆ Difficult to control in proportional region
 - ◆ Easy to control in saturated region
- ◆ PWM is a digital signal \Rightarrow easy for MCU to output



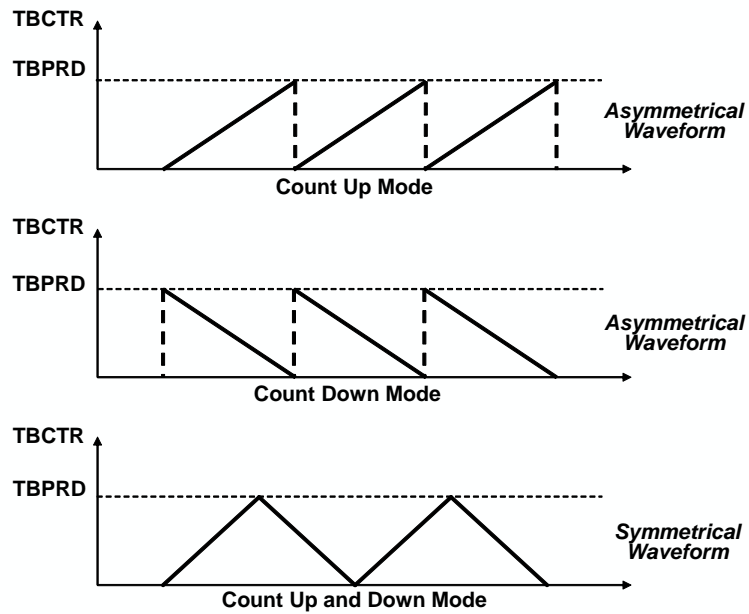
ePWM

ePWM Module Signals and Connections

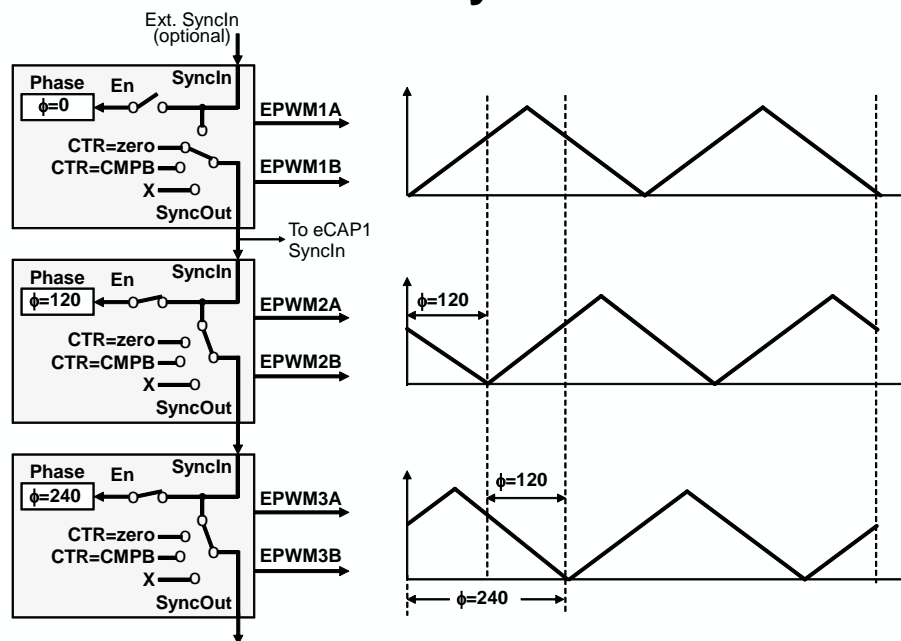


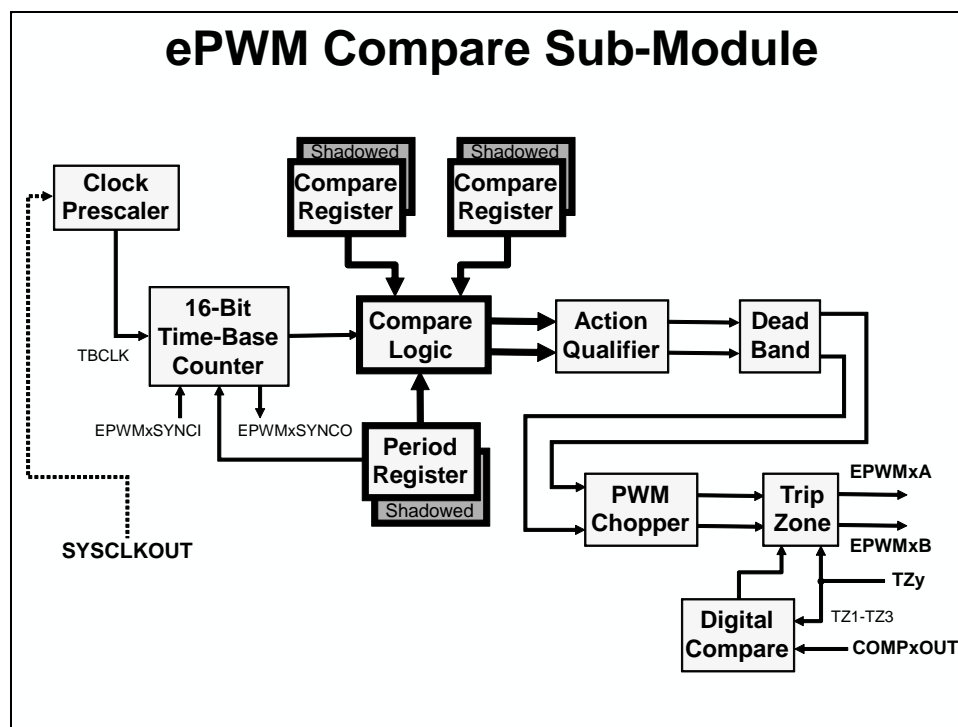


ePWM Time-Base Count Modes

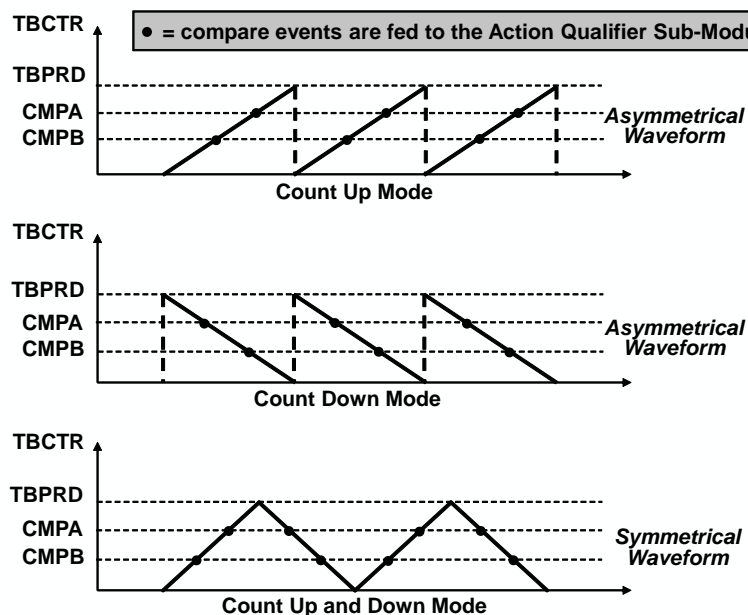


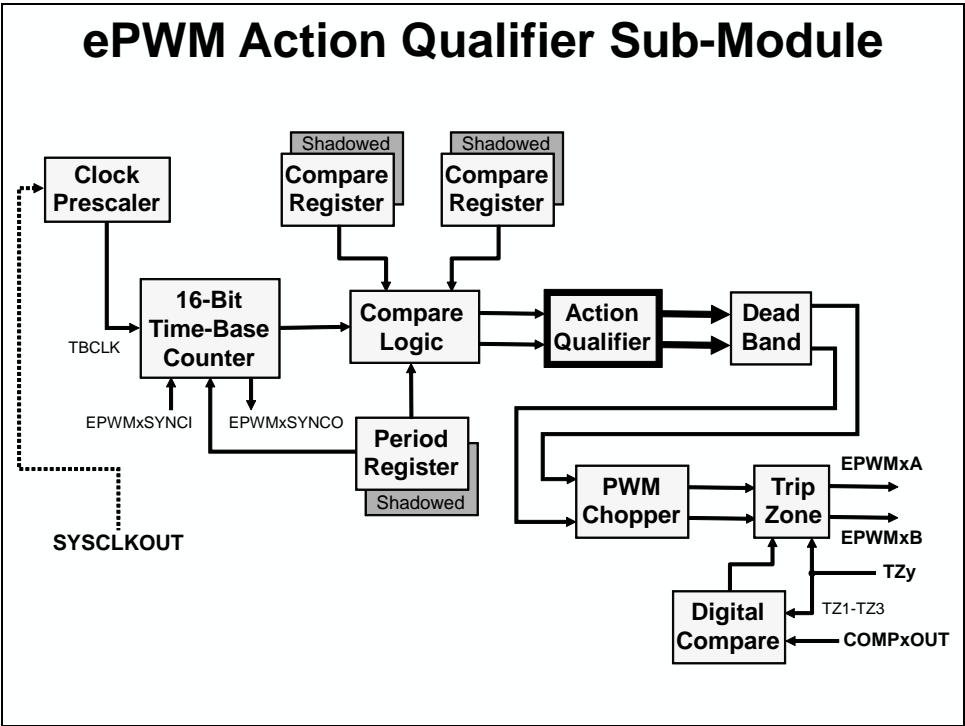
ePWM Phase Synchronization





ePWM Compare Event Waveforms



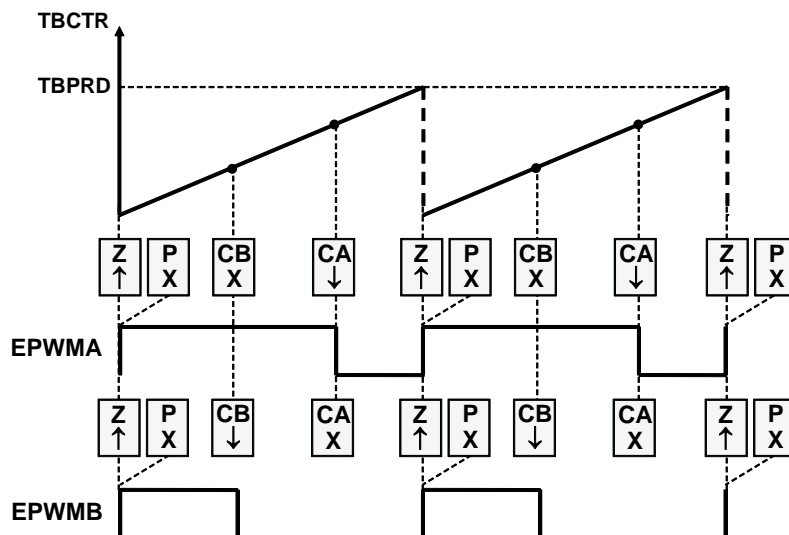


ePWM Action Qualifier Actions
for EPWMA and EPWMB

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

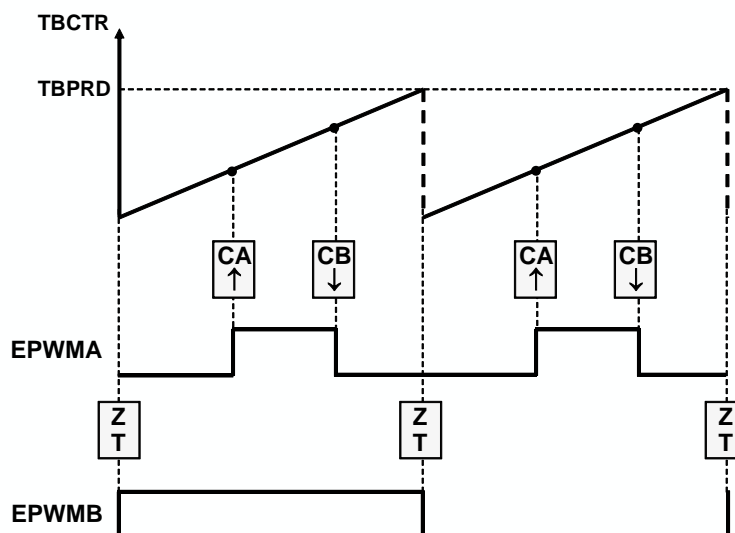
ePWM Count Up Asymmetric Waveform

with Independent Modulation on EPWMA / B



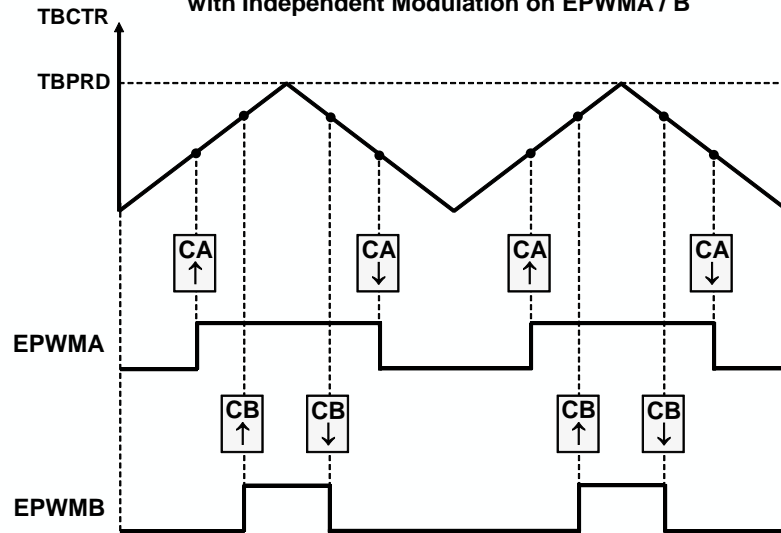
ePWM Count Up Asymmetric Waveform

with Independent Modulation on EPWMA



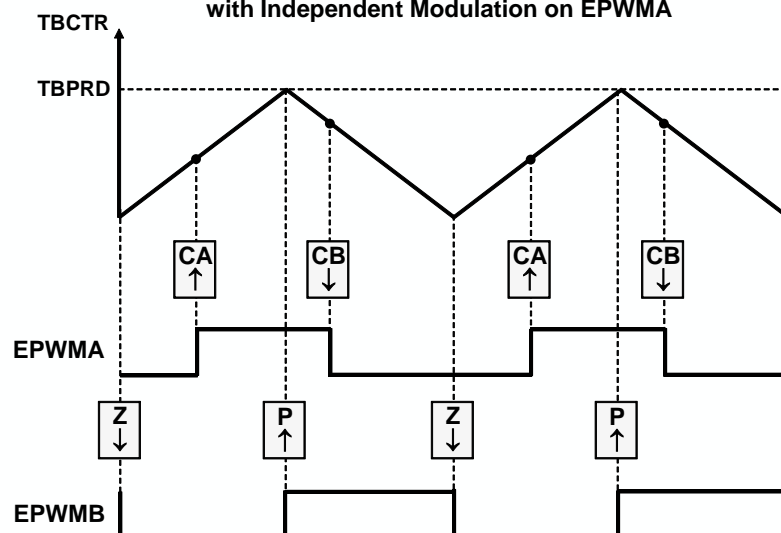
ePWM Count Up-Down Symmetric Waveform

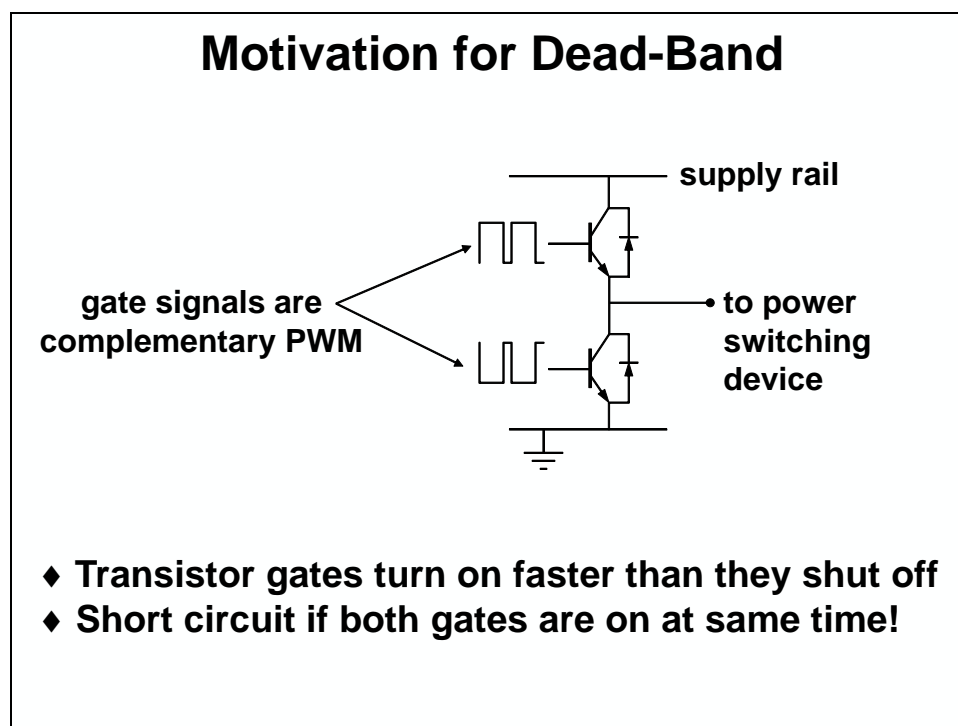
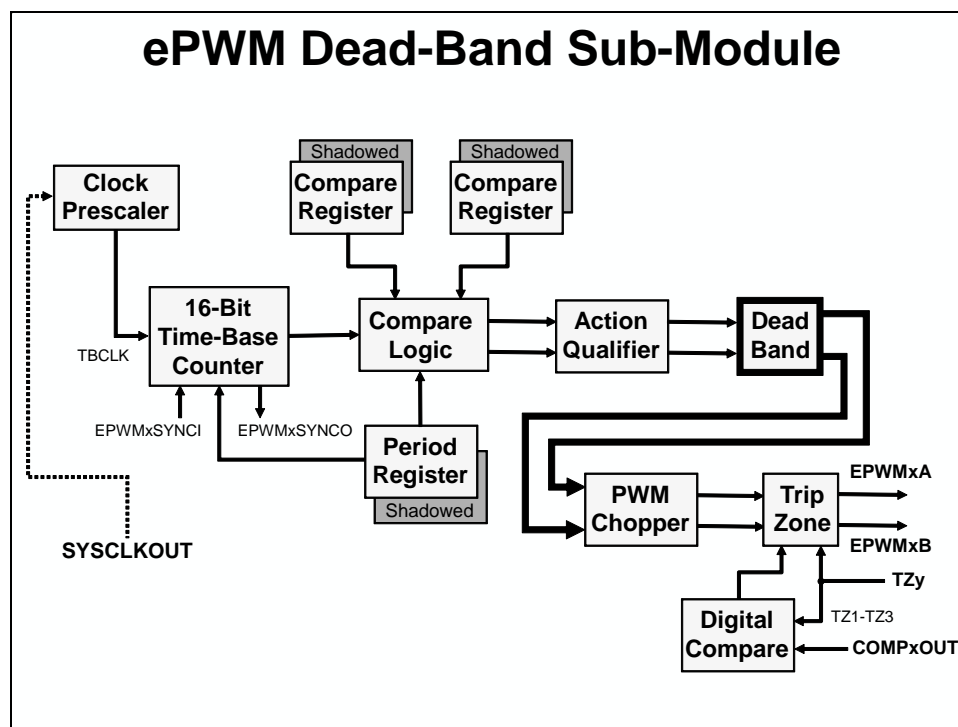
with Independent Modulation on EPWMA / B



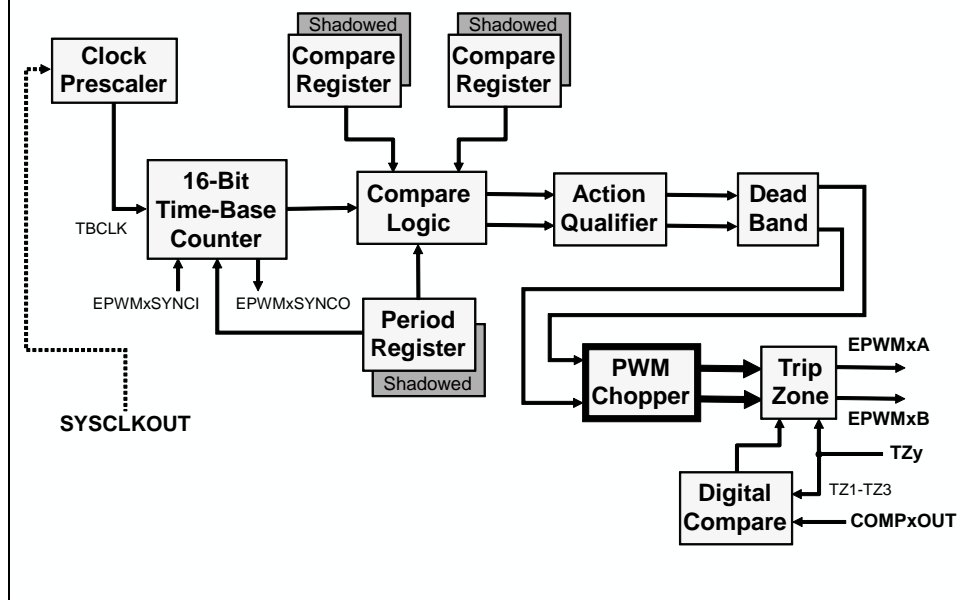
ePWM Count Up-Down Symmetric Waveform

with Independent Modulation on EPWMA

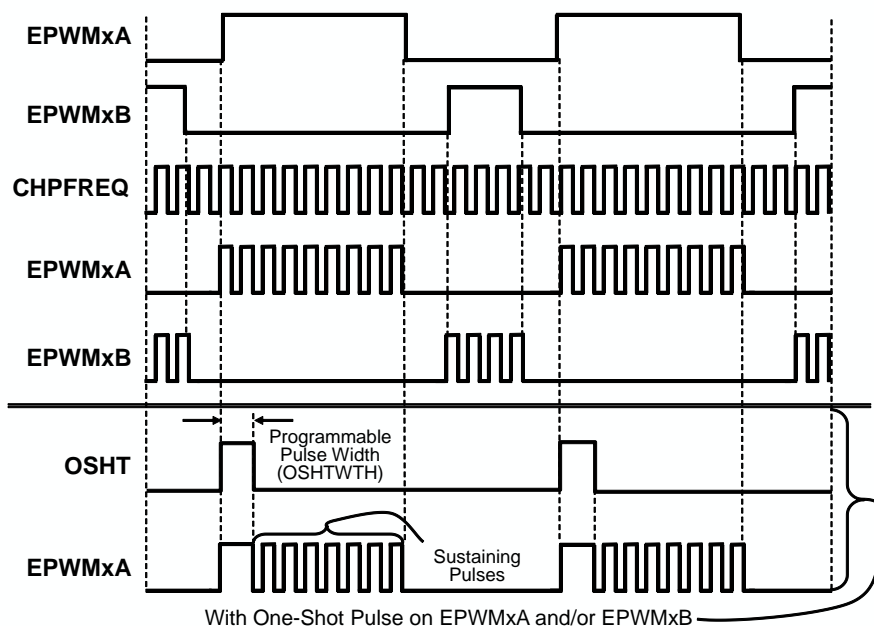


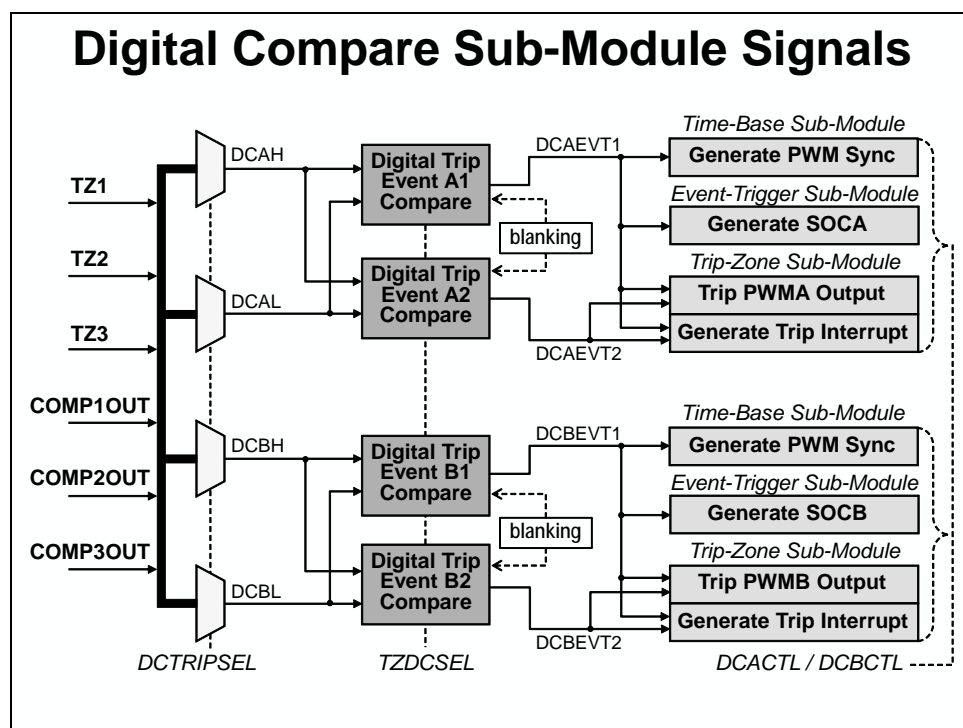
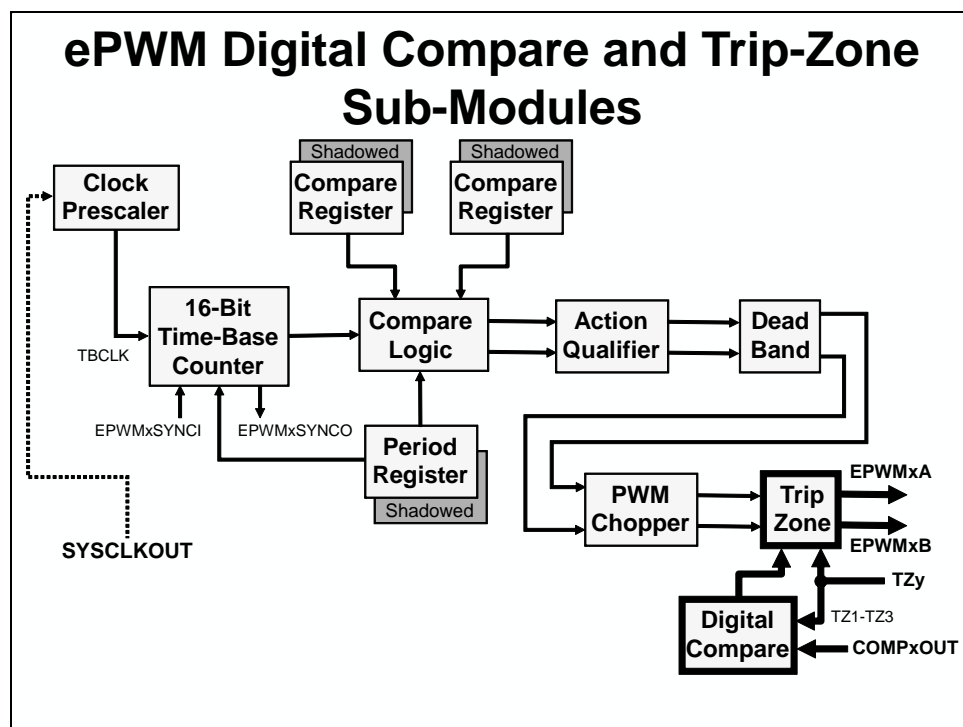


ePWM PWM Chopper Sub-Module



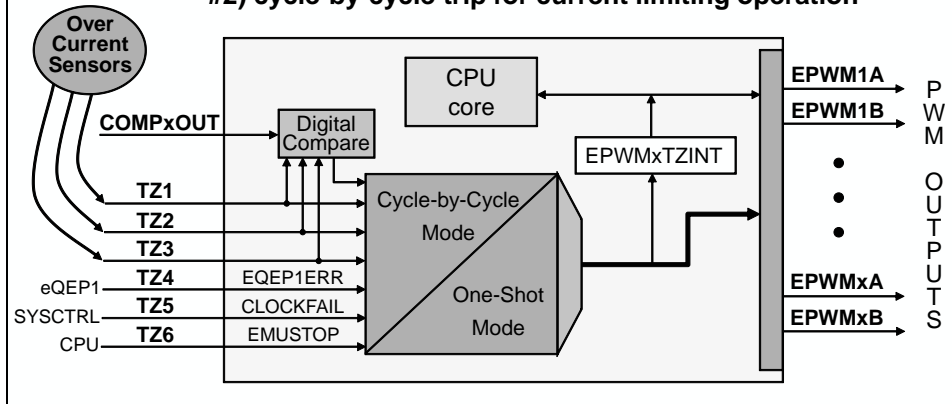
ePWM Chopper Waveform



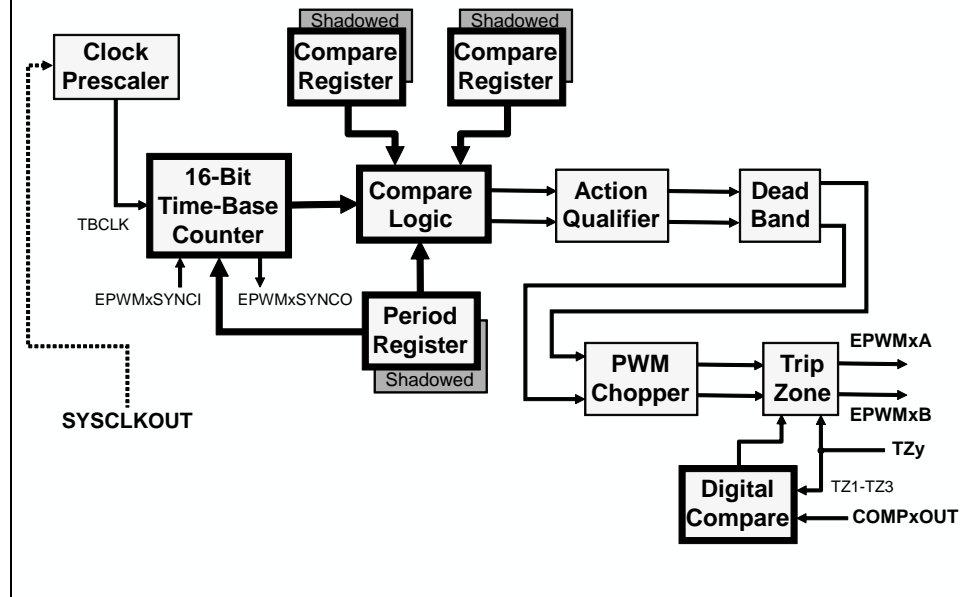


Trip-Zone Features

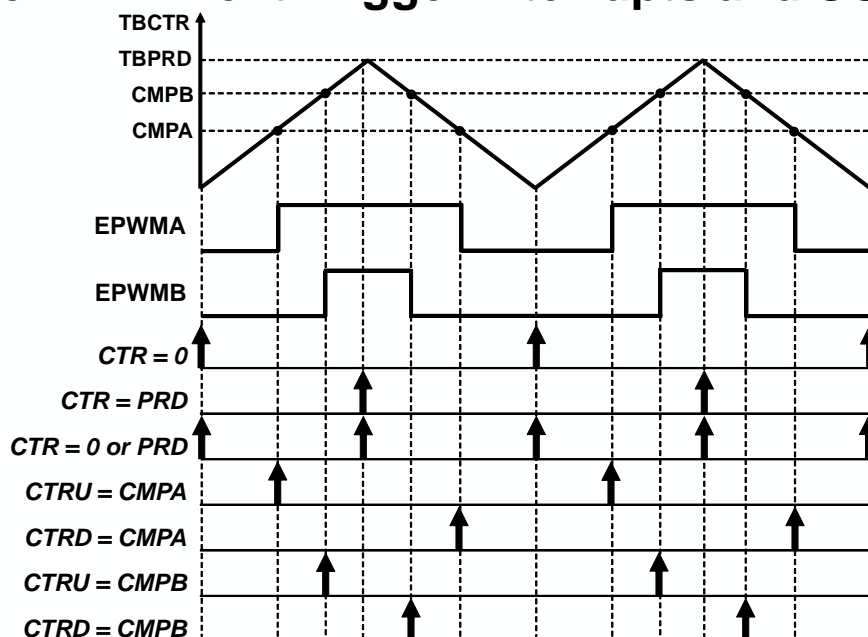
- ◆ Trip-Zone has a fast, clock independent logic path to high-impedance the EPWMxA/B output pins
- ◆ Interrupt latency may not protect hardware when responding to over current conditions or short-circuits through ISR software
- ◆ Supports: #1) one-shot trip for major short circuits or over current conditions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



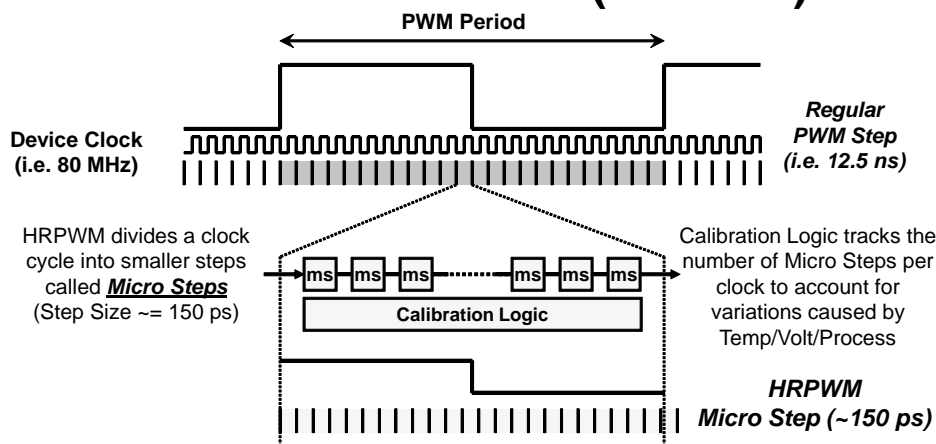
ePWM Event-Trigger Sub-Module



ePWM Event-Trigger Interrupts and SOC



Hi-Resolution PWM (HRPWM)



- ◆ 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 ~160 kHz (with system clock of 80 MHz)
- ◆ Not all ePWM outputs support HRPWM feature (see device datasheet)

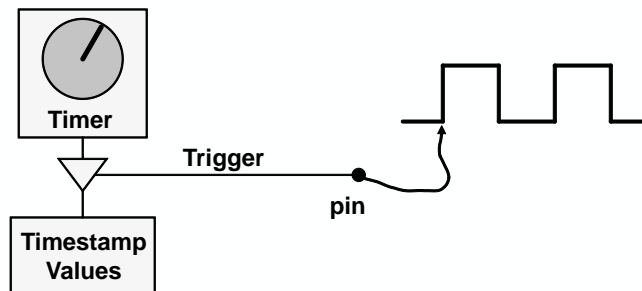
ePWM Control Registers (file: EPwm.c)

- ◆ **TBCTL** (Time-Base Control)
 - ◆ counter mode (up, down, up & down, stop); clock prescale; period shadow load; phase enable/direction; sync select
- ◆ **CMPCTL** (Compare Control)
 - ◆ compare load mode; operating mode (shadow / immediate)
- ◆ **AQCTLA/B** (Action Qualifier Control Output A/B)
 - ◆ action on up/down CTR = CMPA/B, PRD, 0 (nothing/set/clear/toggle)
- ◆ **DBCTL** (Dead-Band Control)
 - ◆ in/out-mode (disable / delay PWMx A/B); polarity select
- ◆ **PCCTL** (PWM-Chopper Control)
 - ◆ enable / disable; chopper CLK freq. & duty cycle; 1-shot pulse width
- ◆ **DCTRIPSEL** (Digital Compare Trip Select)
 - ◆ Digital compare A/B high/low input source select
- ◆ **TZCTL** (Trip-Zone Control)
 - ◆ enable / disable; action (force high / low / high-Z / nothing)
- ◆ **ETSEL** (Event-Trigger Selection)
 - ◆ interrupt & SOCA/B enable / disable; interrupt & SOCA/B select

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

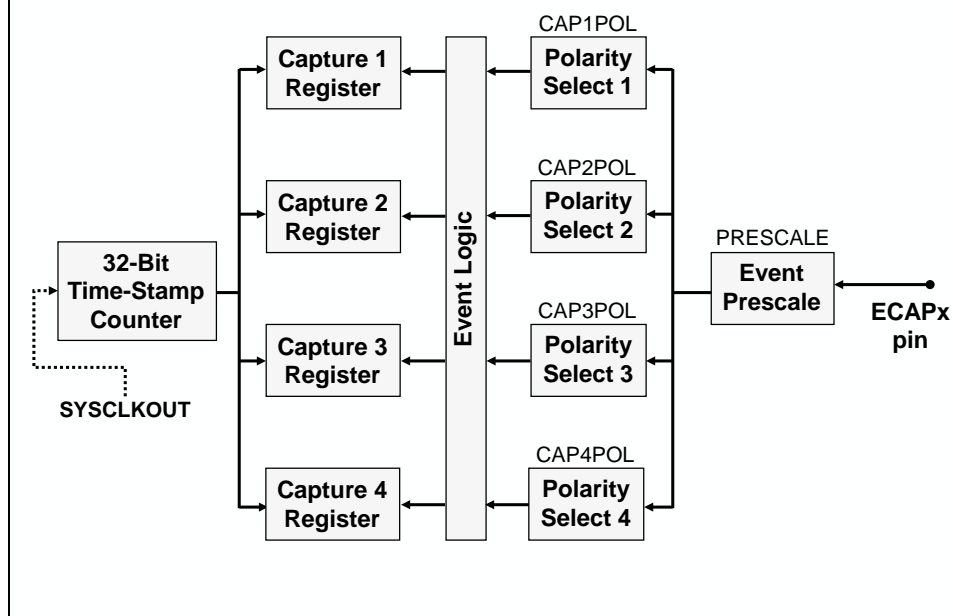
eCAP

Capture Module (eCAP)

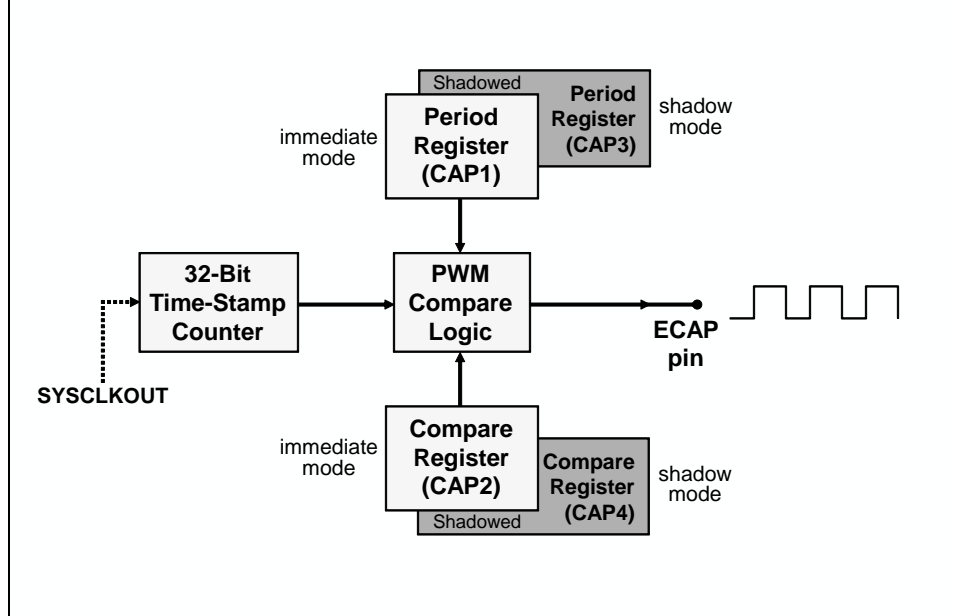


- ◆ The eCAP module timestamps transitions on a capture input pin

eCAP Module Block Diagram – Capture Mode



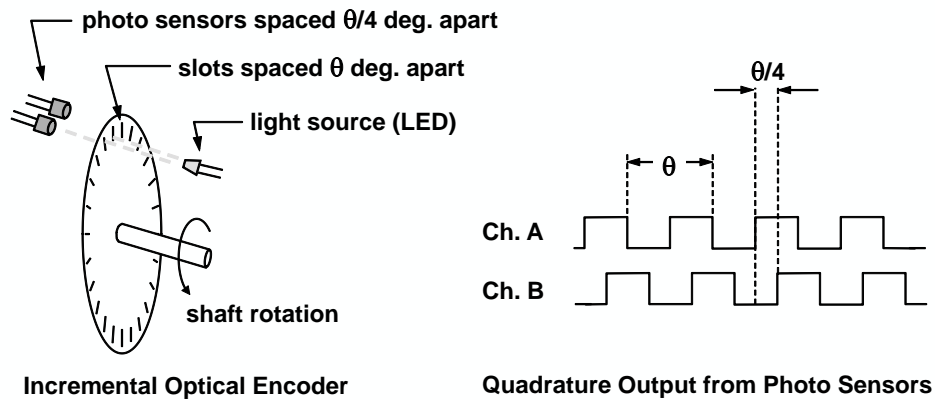
eCAP Module Block Diagram – APWM Mode



eQEP

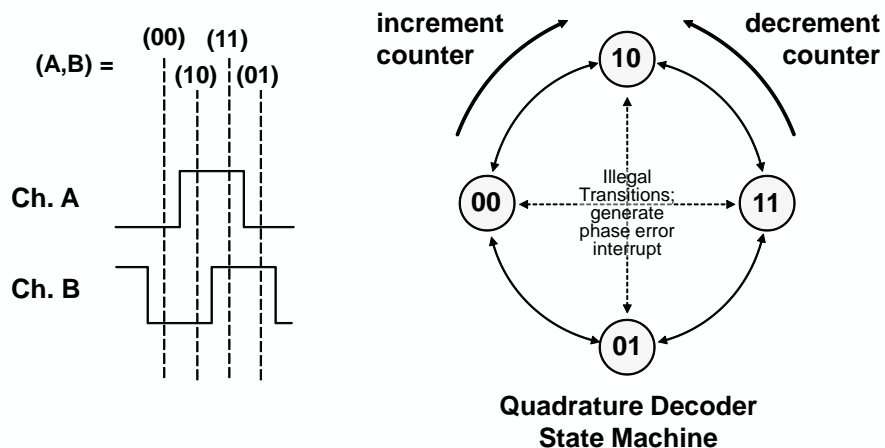
What is an Incremental Quadrature Encoder?

A digital (angular) position sensor

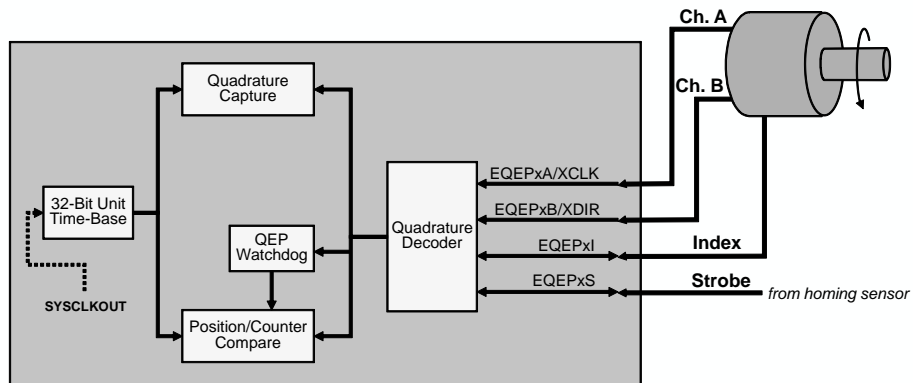


How is Position Determined from Quadrature Signals?

Position resolution is $\theta/4$ degrees



eQEP Module Connections



Lab 3: Control Peripherals

➤ Objective

The objective of this lab is to demonstrate and become familiar with the operation of the on-chip analog-to-digital converter and ePWM. ePWM1A will be setup to generate a 2 kHz, 25% duty cycle symmetric PWM waveform. The waveform will then be sampled with the on-chip analog-to-digital converter and displayed using the graphing feature of Code Composer Studio. The ADC has been setup to sample a single input channel at a 50 kHz sampling rate and store the conversion result in a buffer in the MCU memory. This buffer operates in a circular fashion, such that new conversion data continuously overwrites older results in the buffer.

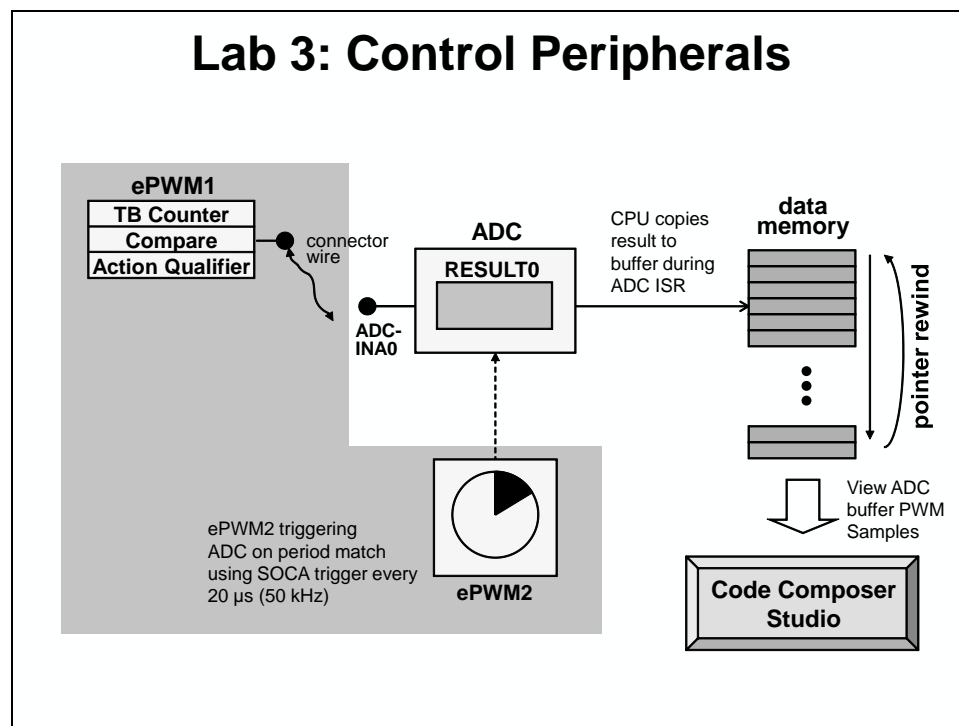
Two ePWM modules have been configured for this lab exercise:

ePWM1A – PWM Generation

- Used to generate a 2 kHz, 25% duty cycle symmetric PWM waveform

ePWM2 – ADC Conversion Trigger

- Used as a timebase for triggering ADC samples (period match trigger SOCA)



The software in this exercise configures the ePWM modules and the ADC. It is entirely interrupt driven. The ADC end-of-conversion interrupt will be used to prompt the CPU to copy the results of the ADC conversion into a results buffer in memory. This buffer pointer will be managed in a circular fashion, such that new conversion results will continuously overwrite older conversion results in the buffer. The ADC interrupt service routine (ISR) will also toggle LED LD2 on the TMS320F28069 controlSTICK as a visual indication that the ISR is running.

Notes

- ePWM1A is used to generate a 2 kHz PWM waveform
- Program performs conversion on ADC channel A0 (ADCINA0 pin)
- ADC conversion is set at a 50 kHz sampling rate
- ePWM2 is triggering the ADC on period match using SOCA trigger
- Data is continuously stored in a circular buffer
- Data is displayed using the graphing feature of Code Composer Studio
- ADC ISR will also toggle the LED LD2 as a visual indication that it is running

➤ Procedure

Open the Project

1. A project named Lab3 has been created for this lab. Open the project by clicking on **Project** → **Import Existing CCS/CCE Eclipse Project**. The “Import” window will open then click **Browse...** next to the “Select search-directory” box. Navigate to: `C:\C28x\Labs\Lab3\Project` and click **OK**. Then click **Finish** to import the project.
2. In the **Project Explorer** window, click the plus sign (+) to the left of Lab3 to view the project files. All Build Options have been configured for this lab. The files used in this lab are:

<code>Adc.c</code>	<code>Gpio.c</code>
<code>CodeStartBranch.asm</code>	<code>Lab.h</code>
<code>DefaultIsr_3_4.c</code>	<code>Lab_2_3.cmd</code>
<code>DelayUs.asm</code>	<code>Main_3.c</code>
<code>EPwm.c</code>	<code>PieCtrl.c</code>
<code>F2806x_DefaultIsr.h</code>	<code>PieVect.c</code>
<code>F2806x_GlobalVariableDefs.c</code>	<code>SysCtrl.c</code>
<code>F2806x-Headers_nonBIOS.cmd</code>	<code>Watchdog.c</code>

Setup GPIO and ePWM1

Note: *DO NOT* make any changes to `Gpio.c` and `EPwm.c` – **ONLY INSPECT**

3. Open and inspect `Gpio.c` by double clicking on the filename in the project window. Notice that the shared I/O pin in GPIO0 has been set for the ePWM1A function. Next, open and inspect `EPwm.c` and see that the ePWM1 has been setup to implement the PWM waveform as described in the objective for this lab. Notice the values used in the following registers: TBCTL (set clock prescales to divide-by-1, no software force, sync and phase disabled), TBPRD, CMPA, CMPCTL (load on 0 or PRD), and AQCTLA (set on up count and clear on down count for output A). Software force, deadband, PWM chopper and trip action has been disabled. (Note that the last steps enable the timer count mode and enable the clock to the ePWM module). See the global variable names and values that have been set using `#define` in the beginning of the `Lab.h` file. Notice that ePWM2 has been initialized earlier in the code for the ADC. Close the inspected files.

Build and Load

- Click the “Build” button and watch the tools run in the Console window. Check for errors in the Problems window.
- Click the “Debug” button (green bug). The “CCS Debug Perspective” view should open, the program load automatically, and you should now be at the start of `Main()`. If the device has been power cycled since the last lab exercise, be sure to configure the boot mode to `EMU_BOOT_SARAM` using the Scripts menu.

Run the Code – PWM Waveform

- Open a memory browser window to view some of the contents of the ADC results buffer. To open a memory browser window click: `View` → `Memory Browser` on the menu bar. The address label for the ADC results buffer is `AdcBuf` (type `&AdcBuf`) in the “Data” memory page. Select `Go` to view the contents of the ADC result buffer.

Note: *Exercise care when connecting any wires, as the power to the controlSTICK is on, and we do not want to damage the controlSTICK!* Details of pin assignments can be found on the last page of this lab exercise.

- Using a connector wire provided, connect the PWM1A (pin # 17) to ADCINA0 (pin # 3) on the controlSTICK.
- Run your code for a few seconds by using the `Resume` button on the toolbar, or using `Run` → `Resume` on the menu bar. After a few seconds halt your code by using the `Suspend` button on the toolbar, or by using `Run` → `Suspend` on the menu bar. Verify that the ADC result buffer contains the updated values.
- Open and setup a graph to plot a 50-point window of the ADC results buffer. Click: `Tools` → `Graph` → `Single Time` and set the following values:

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

Select `OK` to save the graph options.

- The graphical display should show the generated 2 kHz, 25% duty cycle symmetric PWM waveform. The period of a 2 kHz signal is 500 μs. You can confirm this by measuring the period of the waveform using the “measurement marker mode” graph feature. In the graph window toolbar, left-click on the ruler icon with the red arrow. Note when you hover your mouse over the icon, it will show “Toggle Measurement

Marker Mode”. Move the mouse to the first measurement position and left-click. Again, left-click on the Toggle Measurement Marker Mode icon. Move the mouse to the second measurement position and left-click. The graph will automatically calculate the difference between the two values taken over a complete waveform period. When done, clear the measurement points by right-clicking on the graph and select Remove All Measurement Marks (or Ctrl+Alt+M).

Frequency Domain Graphing Feature of Code Composer Studio

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

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

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

Select OK to save the graph options.

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

Using Real-time Emulation

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.

13. The memory and single time graph windows displaying *AdcBuf* should still be open. The connector wire between PWM1A (pin # 17) and ADCINA0 (pin # 3) should still be connected. In real-time mode, we will have our window continuously refresh at the default rate. To view the refresh rate click:

Window → Preferences...

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

- “Continuous refresh interval (milliseconds)” = 500

Click OK.

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

14. Next we need to enable the graph window for continuous refresh. Select the “Single Time” graph. In the graph window toolbar, left-click on the yellow icon with the arrows rotating in a circle over a pause sign. Note when you hover your mouse over the icon, it will show “Enable Continuous Refresh”. This will allow the graph to continuously refresh in real-time while the program is running.

15. Enable the memory window for continuous refresh using the same procedure as the previous step.

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

Scripts → Realtime Emulation Control → Run_Realtime_with_Reset

17. **Carefully** remove and replace the connector wire from ADCINA0 (pin # 3). Are the values updating as expected?

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

Scripts → Realtime Emulation Control → Full_Halt

Terminate Debug Session and Close Project

19. Terminate the active debug session using the Terminate button. This will close the debugger and return CCS to the “CCS Edit Perspective” view.
20. Next, close the project by right-clicking on Lab3 in the Project Explorer window and select Close Project.

Optional Exercise

You might want to experiment with this code by changing some of the values or just modify the code. Try generating another waveform of a different frequency and duty cycle. Also, try to generate complementary pair PWM outputs. Next, try to generate additional simultaneous waveforms by using other ePWM modules. Hint: don’t forget to setup the proper shared I/O pins, etc. (This optional exercise requires some further working knowledge of the ePWM. Additionally, it may require more time than is allocated for this lab. Therefore, you may want to try this after the class).

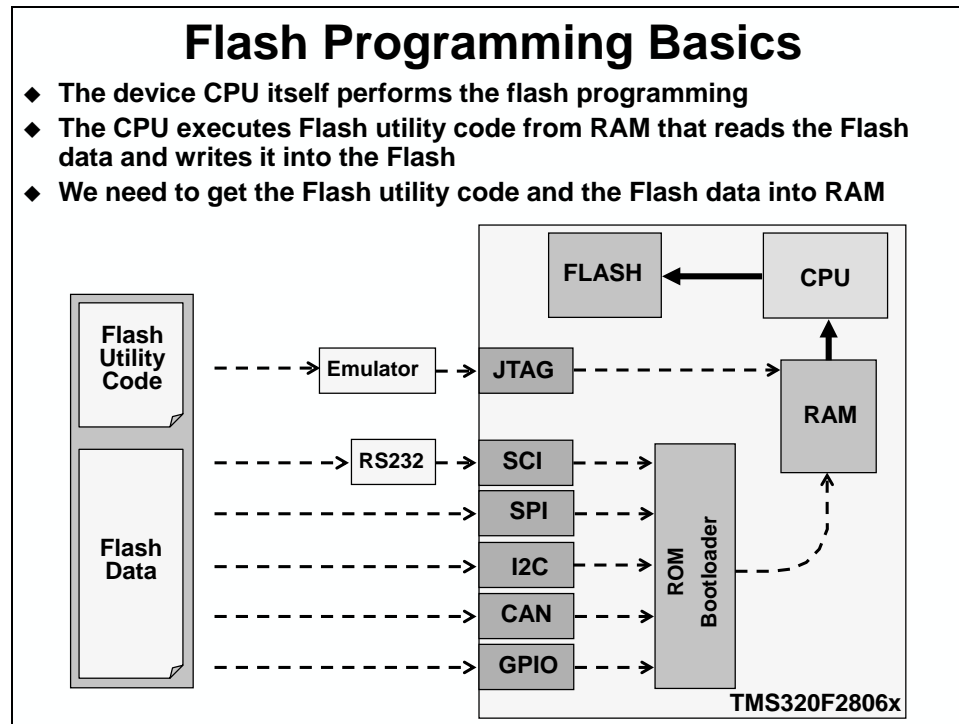
End of Exercise

Lab Reference: F28069 controlSTICK Header Pin Diagram

1 ADC-A6 COMP3 (+VE)	2 ADC-A2 COMP1 (+VE)	3 ADC-A0	4 3V3
5 ADC-A4 COMP2 (+VE)	6 ADC-B1	7 EPWM-4B GPIO-07	8 TZ1 GPIO-12
9 SCL-A GPIO-33	10 ADC-B6 COMP3 (-VE)	11 EPWM-4A GPIO-06	12 ADC-A1
13 SDA-A GPIO-32	14 ADC-B0	15 EPWM-3B GPIO-05	16 5V0 (Disabled by Default)
17 EPWM-1A GPIO-00	18 ADC-B4 COMP2 (-VE)	19 EPWM-3A GPIO-04	20 SPISOMI-A GPIO-17
21 EPWM-1B GPIO-01	22 ADC-A5	23 EPWM-2B GPIO-03	24 SPISIMO-A GPIO-16
25 SPISTE-A GPIO-19	26 ADC-B2 COMP1 (-VE)	27 EPWM-2A GPIO-02	28 GND
29 SPICLK-A GPIO-18	30 GPIO-34 (LED)	31 PWM1A-DAC (Filtered)	32 GND

Flash Programming

Flash Programming Basics



Flash Programming Basics

- ◆ Sequence of steps for Flash programming:

Algorithm	Function
1. Erase	- Set all bits to zero, then to one
2. Program	- Program selected bits with zero
3. Verify	- Verify flash contents

- ◆ Minimum Erase size is a sector (8Kw or 16Kw)
- ◆ Minimum Program size is a bit!
- ◆ Important not to lose power during erase step:
If CSM passwords happen to be all zeros, the CSM will be permanently locked!
- ◆ Chance of this happening is quite small! (Erase step is performed sector by sector)

Programming Utilities and CCS Flash Programmer

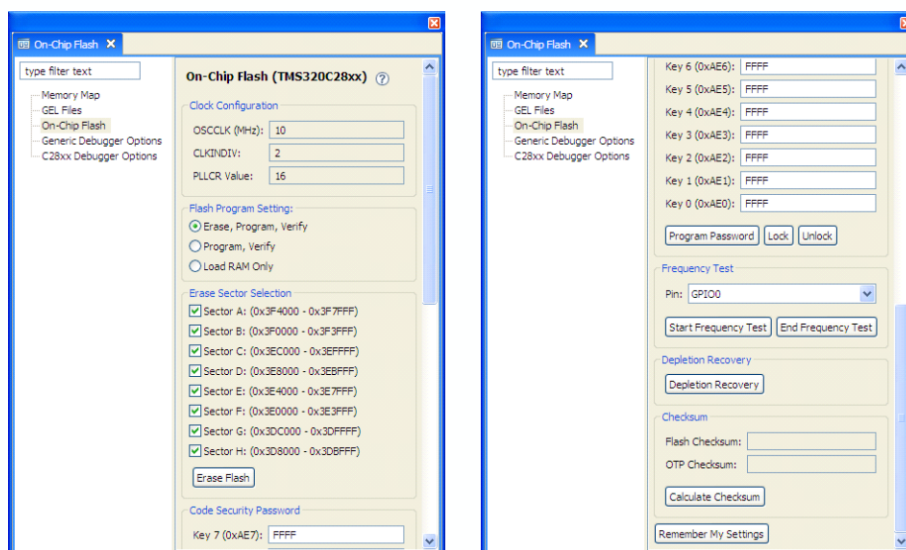
Flash Programming Utilities

- ◆ JTAG Emulator Based
 - Code Composer Studio on-chip Flash programmer
 - BlackHawk Flash utilities (requires Blackhawk emulator)
 - Elprotronic FlashPro2000
 - Spectrum Digital SDFlash JTAG (requires SD emulator)
 - Signum System Flash utilities (requires Signum emulator)
- ◆ SCI Serial Port Bootloader Based
 - Code-Skin (<http://www.code-skin.com>)
 - Elprotronic FlashPro2000
- ◆ Production Test/Programming Equipment Based
 - BP Micro programmer
 - Data I/O programmer
- ◆ Build your own custom utility
 - Can use any of the ROM bootloader methods
 - Can embed flash programming into your application
 - Flash API algorithms provided by TI

* TI web has links to all utilities (<http://www.ti.com/c2000>)

CCS On-Chip Flash Programmer

- ◆ On-Chip Flash programmer is integrated into the CCS debugger



- ◆ Tools → On-Chip Flash

Code Security Module and Password

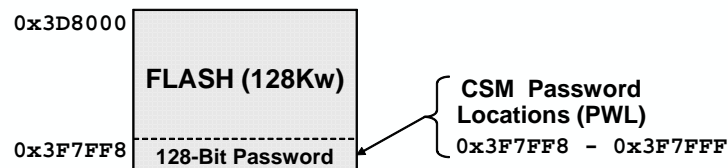
Code Security Module (CSM)

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

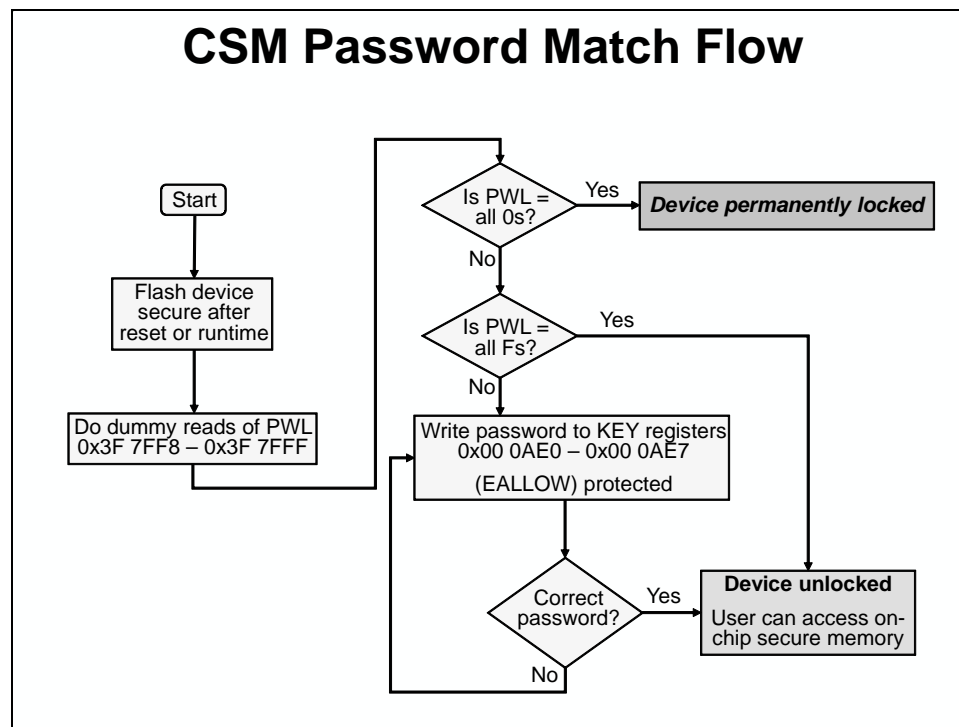
0x000A80	Flash Registers
0x008000	L0 DPSARAM (2Kw)
0x008800	L1 DPSARAM (1Kw)
0x008C00	L2 DPSARAM (1Kw)
0x009000	L3 DPSARAM (4Kw)
0x00A000	L4 DPSARAM (8Kw)
0x00C000	reserved
0x3D7800	User OTP (1Kw)
0x3D7C00	reserved
0x3D7C80	ADC / OSC cal. data
0x3D7CC0	reserved
0x3D8000	FLASH (128Kw)
0x3F7FF8	PASSWORDS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

- ◆ Data reads and writes from restricted memory are only allowed for code running from restricted memory
- ◆ All other data read/write accesses are blocked:
JTAG emulator/debugger, ROM bootloader, code running in external memory or unrestricted internal memory

CSM Password



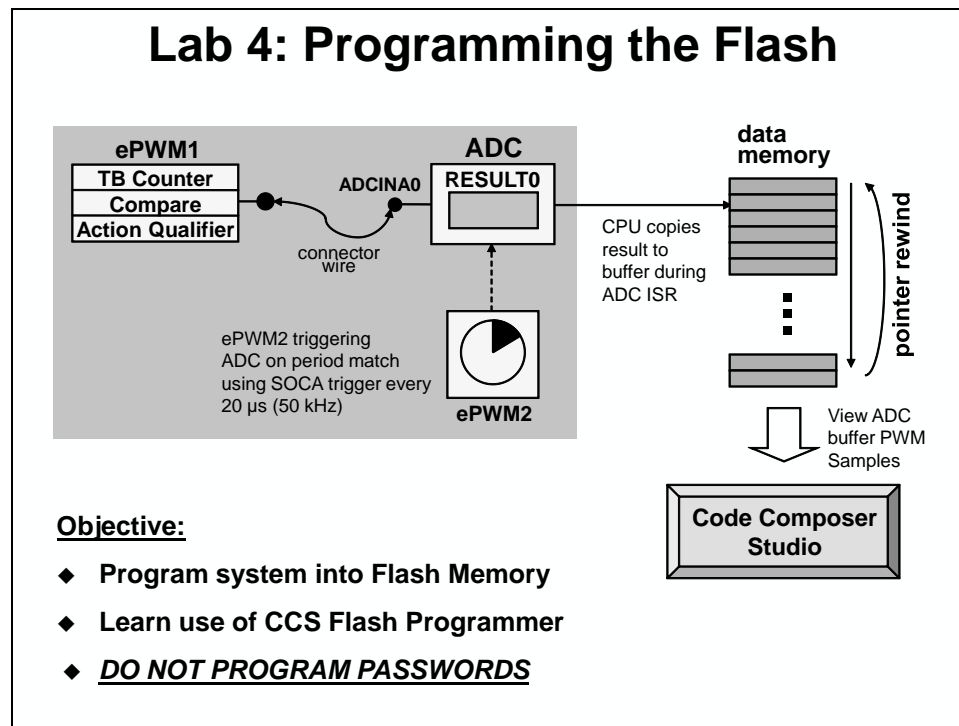
- ◆ 128-bit user defined password is stored in Flash
- ◆ 128-bit KEY registers are used to lock and unlock the device
 - ◆ Mapped in memory space 0x00 0AE0 – 0x00 0AE7
 - ◆ Registers “EALLOW” protected



Lab 4: Programming the Flash

➤ Objective

The objective of this lab is to program and execute code from the on-chip flash memory. The TMS320F28069 device has been designed for standalone operation in an embedded system. Using the on-chip flash eliminates the need for external non-volatile memory or a host processor from which to bootload. In this lab, the steps required to properly configure the software for execution from internal flash memory will be covered.



➤ Procedure

Open the Project

1. A project named Lab4 has been created for this lab. Open the project by clicking on Project → Import Existing CCS/CCE Eclipse Project. The “Import” window will open then click Browse... next to the “Select search-directory” box. Navigate to: C:\C28x\Labs\Lab4\Project and click OK. Then click Finish to import the project.
2. In the Project Explorer window, click the plus sign (+) to the left of Lab4 to view the project files. All Build Options have been configured for this lab. The files used in this lab are:

<code>Adc.c</code>	<code>Gpio.c</code>
<code>CodeStartBranch.asm</code>	<code>Lab.h</code>
<code>DefaultIsr_3_4.c</code>	<code>Lab_4.cmd</code>
<code>DelayUs.asm</code>	<code>Main_4.c</code>
<code>EPwm.c</code>	<code>Passwords.asm</code>
<code>F2806x_DefaultIsr.h</code>	<code>PieCtrl.c</code>
<code>F2806x_GlobalVariableDefs.c</code>	<code>PieVect.c</code>
<code>F2806x_Headers_nonBIOS.cmd</code>	<code>SysCtrl.c</code>
<code>Flash.c</code>	<code>Watchdog.c</code>

Link Initialized Sections to Flash

Initialized sections, such as code and constants, must contain valid values at device power-up. Stand-alone operation of an F28069 embedded system means that no emulator is available to initialize the device RAM. Therefore, all initialized sections must be linked to the on-chip flash memory.

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

- Open and inspect the linker command file `Lab_4.cmd`. Notice that a memory block named `FLASH_ABCDEFGH` has been created at `origin = 0x3D8000`, `length = 0x01FF80` on Page 0. This flash memory block length has been selected to avoid conflicts with other required flash memory spaces. See the reference slide at the end of this lab exercise for further details showing the address origins and lengths of the various memory blocks used.
- In `Lab_4.cmd` the following compiler sections have been linked to on-chip flash memory block `FLASH_ABCDEFGH`:

Compiler Sections:

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

Copying Interrupt Vectors from Flash to RAM

The interrupt vectors must be located in on-chip flash memory and at power-up needs to be copied to the PIE RAM as part of the device initialization procedure. The code that performs this copy is located in `InitPieCtrl()`. The C-compiler runtime support library contains a memory copy function called `memcpy()` which will be used to perform the copy.

- Open and inspect `InitPieCtrl()` in `PieCtrl.c`. Notice the `memcpy()` function used to initialize (copy) the PIE vectors. At the end of the file a structure is used to enable the PIE.

Initializing the Flash Control Registers

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

6. Open and inspect `Flash.c`. The C compiler `CODE_SECTION` pragma is used to place the `InitFlash()` function into a linkable section named “secureRamFuncs”.
7. The “secureRamFuncs” section will be linked using the user linker command file `Lab_4.cmd`. Open and inspect `Lab_4.cmd`. The “secureRamFuncs” will load to flash (load address) but will run from L4SARAM (run address). Also notice that the linker has been asked to generate symbols for the load start, load end, and run start addresses.

While not a requirement from a MCU hardware or development tools perspective (since the C28x MCU has a unified memory architecture), historical convention is to link code to program memory space and data to data memory space. Therefore, notice that for the L4SARAM memory we are linking “secureRamFuncs” to, we are specifying “PAGE = 0” (which is program memory).

8. Open and inspect `Main_4.c`. Notice that the memory copy function `memcpy()` is being used to copy the section “secureRamFuncs”, which contains the initialization function for the flash control registers.
9. The following line of code in `main()` is used call the `InitFlash()` function. Since there are no passed parameters or return values the code is just:

```
InitFlash();
```

at the desired spot in `main()`.

Code Security Module and Passwords

The CSM module provides protection against unwanted copying (i.e. pirating!) of your code from flash, OTP memory, and the L0, L1, L2, L3 and L4 RAM blocks. The CSM uses a 128-bit password made up of 8 individual 16-bit words. They are located in flash at addresses 0x3F7FF8 to 0x3F7FFF. During this lab, dummy passwords of 0xFFFF will be used – therefore only dummy reads of the password locations are needed to unsecure the CSM. **DO NOT PROGRAM ANY REAL PASSWORDS INTO THE DEVICE.** After development, real passwords are typically placed in the password locations to protect your code. We will not be using real passwords in the workshop.

The CSM module also requires programming values of 0x0000 into flash addresses 0x3F7F80 through 0x3F7FF5 in order to properly secure the CSM. Both tasks will be accomplished using a simple assembly language file `Passwords.asm`.

10. Open and inspect `Passwords.asm`. This file specifies the desired password values (**DO NOT CHANGE THE VALUES FROM 0xFFFF**) and places them in an initialized

section named “passwords”. It also creates an initialized section named “csm_rsvd” which contains all 0x0000 values for locations 0x3F7F80 to 0x3F7FF5 (length of 0x76).

11. Open `Lab_4.cmd` and notice that the initialized sections for “passwords” and “csm_rsvd” are linked to memories named `PASSWORDS` and `CSM_RSVD`, respectively.

Executing from Flash after Reset

The F28069 device contains a ROM bootloader that will transfer code execution to the flash after reset. When the boot mode selection is set for “Jump to Flash” mode, the bootloader will branch to the instruction located at address 0x3F7FF6 in the flash. An instruction that branches to the beginning of your program needs to be placed at this address. Note that the CSM passwords begin at address 0x3F7FF8. There are exactly two words available to hold this branch instruction, and not coincidentally, a long branch instruction “LB” in assembly code occupies exactly two words. Generally, the branch instruction will branch to the start of the C-environment initialization routine located in the C-compiler runtime support library. The entry symbol for this routine is `_c_int00`. Recall that C code cannot be executed until this setup routine is run. Therefore, assembly code must be used for the branch. We are using the assembly code file named `CodeStartBranch.asm`.

12. Open and inspect `CodeStartBranch.asm`. This file creates an initialized section named “codestart” that contains a long branch to the C-environment setup routine. This section has been linked to a block of memory named `BEGIN_FLASH`.
13. In the earlier lab exercises, the section “codestart” was directed to the memory named `BEGIN_M0`. Open and inspect `Lab_4.cmd` and notice that the section “codestart” will now be directed to `BEGIN_FLASH`. Close the inspected files.

On power up the reset vector will be fetched and the ROM bootloader will begin execution. If the emulator is connected, the device will be in emulator boot mode and will use the `EMU_KEY` and `EMU_BMODE` values in the PIE RAM to determine the bootmode. This mode was utilized in an earlier lab. In this lab, we will be disconnecting the emulator and running in stand-alone boot mode (but do not disconnect the emulator yet!). The bootloader will read the `OTP_KEY` and `OTP_BMODE` values from their locations in the OTP. The behavior when these values have not been programmed (i.e., both 0xFFFF) or have been set to invalid values is boot to flash bootmode.

Build – Lab.out

14. Click the “Build” button to generate the `Lab.out` file to be used with the CCS Flash Programmer. Check for errors in the Problems window.

Programming the On-Chip Flash Memory

In CCS the on-chip flash programmer is integrated into the debugger. When the program is loaded CCS will automatically determine which sections reside in flash memory based on the linker command file. CCS will then program these sections into the on-chip flash memory. Additionally, in order to effectively debug with CCS, the symbolic debug information (e.g., symbol and label addresses, source file links, etc.) will automatically load so that CCS knows

where everything is in your code. Clicking the “Debug” button in the “CCS Edit Perspective” will automatically launch the debugger, connect to the target, and program the flash memory in a single step.

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

Running the Code – Using CCS

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

Run → Reset → Reset CPU

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

17. Under Scripts on the menu bar click:

EMU Boot Mode Select → EMU_BOOT_FLASH.

This has the debugger load values into EMU_KEY and EMU_BMODE so that the bootloader will jump to "FLASH" at address 0x3F7FF6.

18. Single-Step by using the <F5> key (or you can use the Step Into button on the horizontal toolbar) through the bootloader code until you arrive at the beginning of the codestart section in the CodeStartBranch.asm file. (Be patient, it will take about 125 single-steps). Notice that we have placed some code in CodeStartBranch.asm to give an option to first disable the watchdog, if selected.
19. Step a few more times until you reach the start of the C-compiler initialization routine at the symbol _c_int00.
20. Now do Run → Go Main. The code should stop at the beginning of your main() routine. If you got to that point successfully, it confirms that the flash has been programmed properly, that the bootloader is properly configured for jump to flash mode, and that the codestart section has been linked to the proper address.
21. You can now run the CPU, and you should observe the LED on the controlSTICK blinking. Try resetting the CPU, select the EMU_BOOT_FLASH boot mode, and then hitting run (without doing all the stepping and the Go Main procedure). The LED should be blinking again.
22. Halt the CPU.

Terminate Debug Session and Close Project

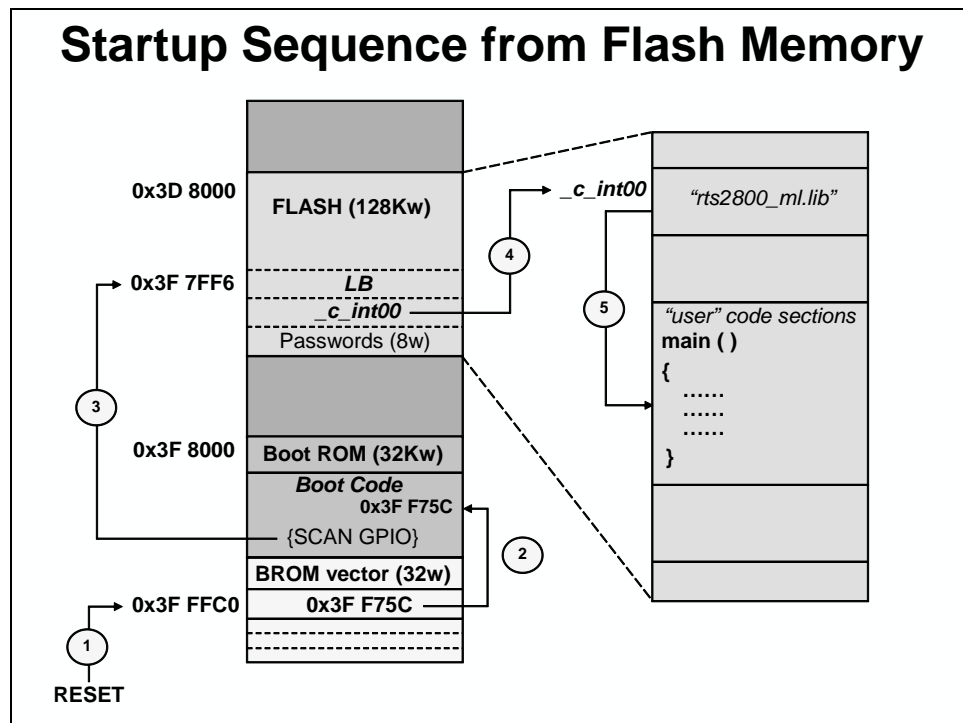
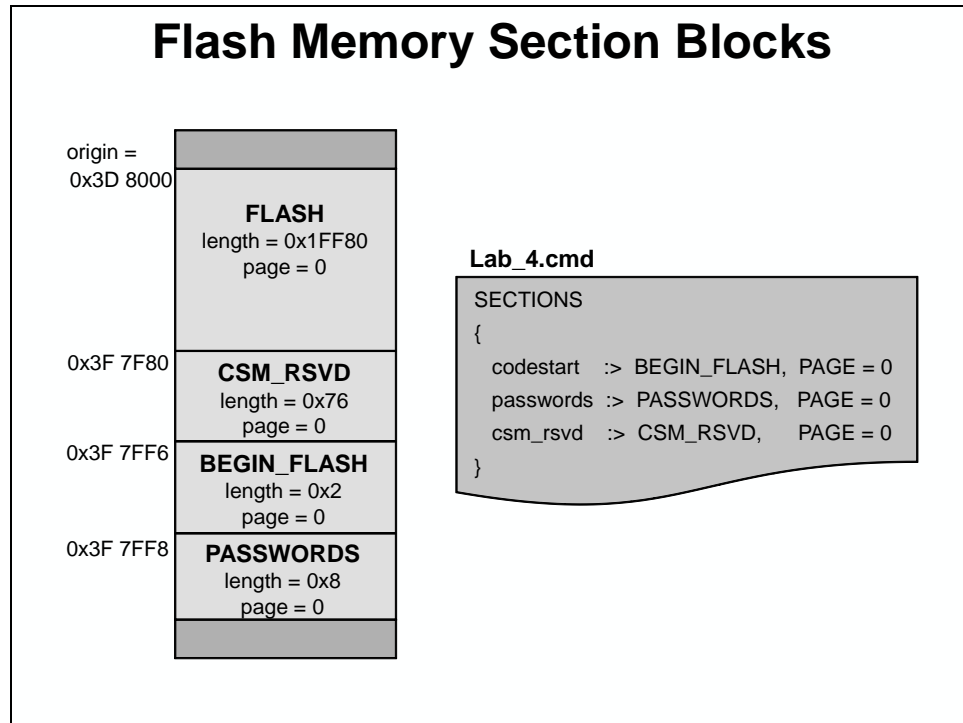
23. Terminate the active debug session using the Terminate button. This will close the debugger and return CCS to the “CCS Edit Perspective” view.
24. Next, close the project by right-clicking on Lab4 in the Project Explorer window and select Close Project.

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

25. Close Code Composer Studio.
26. Disconnect the controlSTICK from the computer USB port.
27. Re-connect the controlSTICK to the computer USB port.
28. The LED should be blinking, showing that the code is now running from flash memory.

End of Exercise

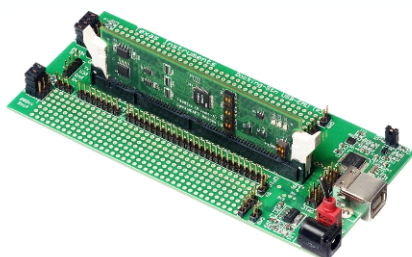
Lab 4 Reference: Programming the Flash



The Next Step...

Training

C2000 MCU Multi-day Training Course

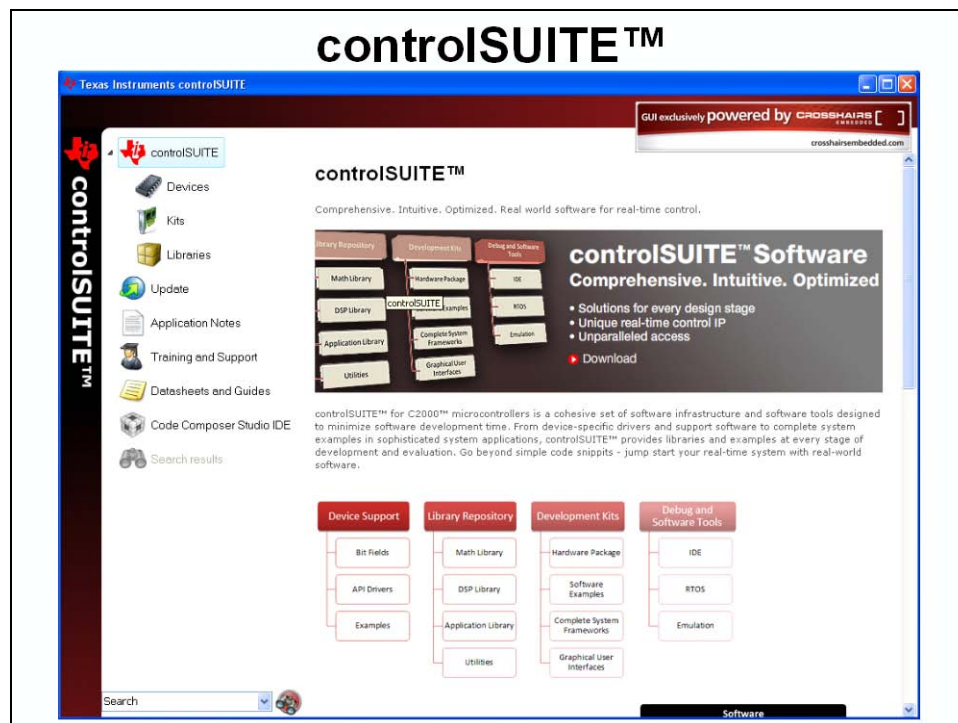


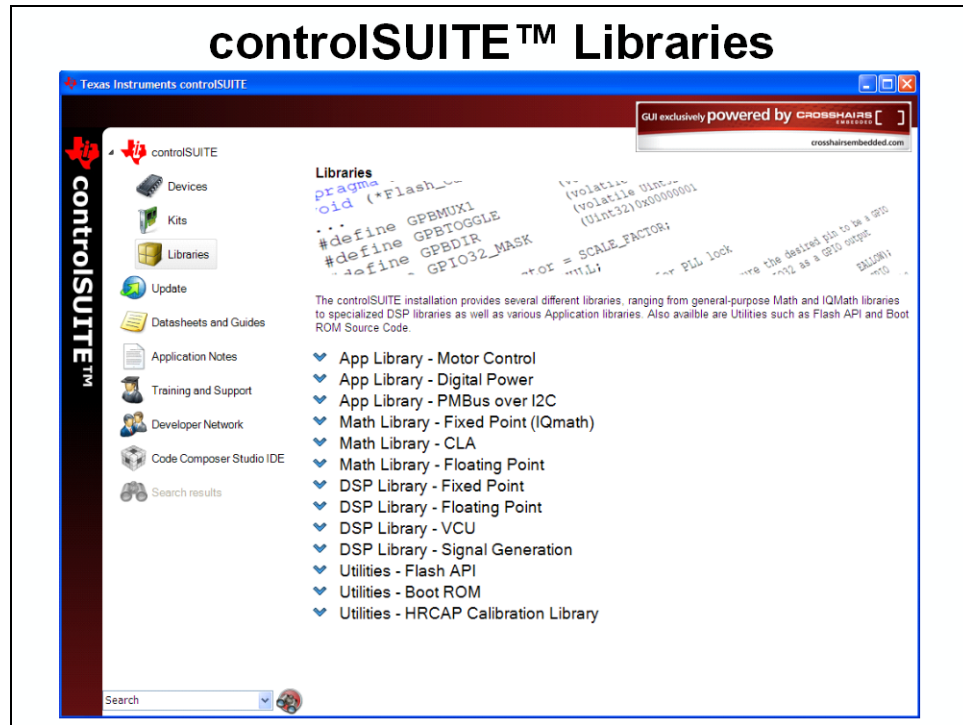
**In-depth hands-on
TMS320F28069 Design
and Peripheral
Training**

TMS320F2806x Workshop Outline

- Architectural Overview
- Programming Development Environment
- Peripheral Register Header Files
- Reset and Interrupts
- System Initialization
- Analog-to-Digital Converter
- Control Peripherals
- Numerical Concepts and Iqmath
- Direct Memory Access (DMA)
- Control Law Accelerator (CLA)
- Viterbi, Complex Math, CRC Unit (VCU)
- System Design
- Communications
- DSP/BIOS
- Support Resources

controlSUITE

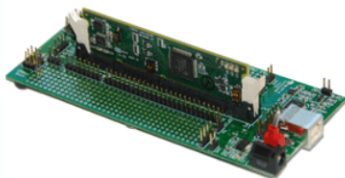




Development Tools

C2000 Experimenter's Kits

F28069, F28035, F28027, F28335, F2808



◆ Part Number:

- ◆ TMDXDOCK28069
- ◆ TMDSDOCK28035
- ◆ TMDSDOCK28027
- ◆ TMDSDOCK28335
- ◆ TMDSDOCK2808

◆ Experimenter Kits include

- ◆ F28069, F28035, F28027, F28335 or F2808 controlCARD
- ◆ USB docking station
- ◆ C2000 Applications Software CD with example code and full hardware details
- ◆ Code Composer Studio v5

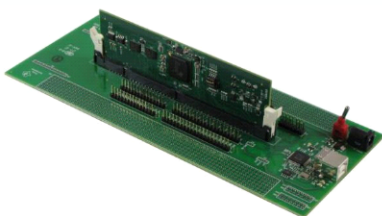
◆ Docking station features

- ◆ Access to controlCARD signals
- ◆ Breadboard areas
- ◆ Onboard USB JTAG Emulation
 - ◆ JTAG emulator not required

◆ Available through TI authorized distributors and the TI eStore

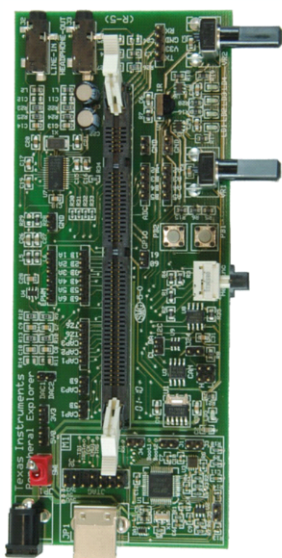
C2834x Experimenter's Kits

C28343, C28346



- ◆ **Part Number:**
 - ◆ TMDXDOCK28343
 - ◆ TMDSDOCK28346-168
- ◆ **Experimenter Kits include**
 - ◆ C2834x controlCARD
 - ◆ Docking station
 - ◆ C2000 Applications Software CD with example code and full hardware details
 - ◆ Code Composer Studio v5
 - ◆ 5V power supply
- ◆ **Docking station features**
 - ◆ Access to controlCARD signals
 - ◆ Breadboard areas
 - ◆ JTAG emulator required – sold separately
- ◆ **Available through TI authorized distributors and the TI eStore**

F28335 Peripheral Explorer Kit



TMDSPREX28335

- ◆ **Experimenter Kit includes**
 - ◆ F28335 controlCARD
 - ◆ Peripheral Explorer baseboard
 - ◆ C2000 Applications Software CD with example code and full hardware details
 - ◆ Code Composer Studio v5
- ◆ **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**

C2000 controlSTICK Evaluation Tool

F28069, F28027



◆ **Part Number:**

- ◆ TMDX28069USB
- ◆ TMDS28027USB

- ◆ **Low-cost USB evaluation tool**
- ◆ **Onboard JTAG Emulation**
 - ◆ *JTAG emulator not required*
- ◆ **Access to controlSTICK signals**
- ◆ **C2000 Applications Software CD with example code and full hardware details**
- ◆ **Code Composer Studio v5**
- ◆ **Available through TI authorized distributors and the TI eStore**

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
 - ◆ Full version of Code Composer Studio v5
- ◆ **Software download includes**
 - ◆ Complete schematics, BOM, gerber files, and source code for board and all software
 - ◆ Quickstart demonstration GUI for quick and easy access to all board features
 - ◆ Fully documented software specific to each kit and application
- ◆ **See www.ti.com/c2000 for other kits and more details**
- ◆ **Available through TI authorized distributors and the TI eStore**

C2000 Workshop Download Wiki

C2000 Workshop Download Wiki

[Log in / create account](#)

Page [Discussion](#) [Read](#) [View source](#) [View history](#) [Go](#) [Search](#)

Hands-On Training for TI Embedded Processors

Hands-On Training for TI Embedded Processors

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

On demand and live training can also be found at [TI eTechDays](#). You can sign up for the live events which are typically scheduled 2-3 times per year.

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

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

C2000™ 32-bit Real-Time MCU Training

C2000™ One-Day Workshop

C2000™ One-Day Workshop agenda, locations, and schedule [↗](#)
[Online materials and labs](#)

C2000™ Multi-Day Workshop

C2000™ Multi-Day Workshop agenda, locations, and schedule [↗](#)
[Online materials and labs](#)

C2000™ Archived Workshops

The archived workshops are for F24xx, F28xx, and F28xx one-day and multi-day workshops. The materials, labs and solutions can be found [here](#)
[C2000 archived workshops](#)

<http://processors.wiki.ti.com/index.php/Training>

Development Support

For More Information . . .

- ◆ **USA – Product Information Center (PIC)**
 - ◆ Phone: 800-477-8924 or 972-644-5580
 - ◆ 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://www.ti.com/training>
- ◆ **TI eStore**
 - ◆ <http://estore.ti.com>
- ◆ **TI website**
 - ◆ <http://www.ti.com>