TMS320F28335™ One-Day Workshop

Workshop Guide and Lab Manual

C28xodw
Revision 5.2
January 2009

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 © 2003 – 2009 Texas Instruments Incorporated

Revision History

February 2003 – Revision 1.0

March 2003 – Revision 1.1

July 2004 – Revision 2.0

August 2005 – Revision 3.0

October 2005 – Revision 3.1

April 2007 – Revision 4.0

June 2008 – Revision 5.0

November 2008 – Revision 5.1

January 2009 – Revision 5.2

Mailing Address

Texas Instruments
Training Technical Organization
7839 Churchill Way
M/S 3984
Dallas, Texas 75251-1903

Workshop Topics

Workshop Topics......3

Workshop Introduction......4

Architecture Overview......7

Programming Development Environment......10
  Code Composer Studio......10
  Linking Sections in Memory......12

Lab 1: Linker Command File......15

Peripheral Register Header Files......19

Reset, Interrupts and System Initialization......26
  Reset......26
  Interrupts......28
  Peripheral Interrupt Expansion (PIE)......29
  Oscillator / PLL Clock Module......31
  Watchdog Timer Module......32
  GPIO......33

Lab 2: System Initialization......35

Control Peripherals......39
  ADC Module......39
  Pulse Width Modulation......40
  ePWM......41
  eCAP......52
  eQEP......54

Lab 3: Control Peripherals......56

Flash Programming......62
  Flash Programming Basics......62
  Programming Utilities and CCS Plug-in......63
  Code Security Module and Password......64

Lab 4: Programming the Flash......66

The Next Step…......73
  Training......73
  Development Tools......74
  Development Support......77

Workshop Introduction

TMS320C28x™ 1-Day Workshop

Texas Instruments
Technical Training

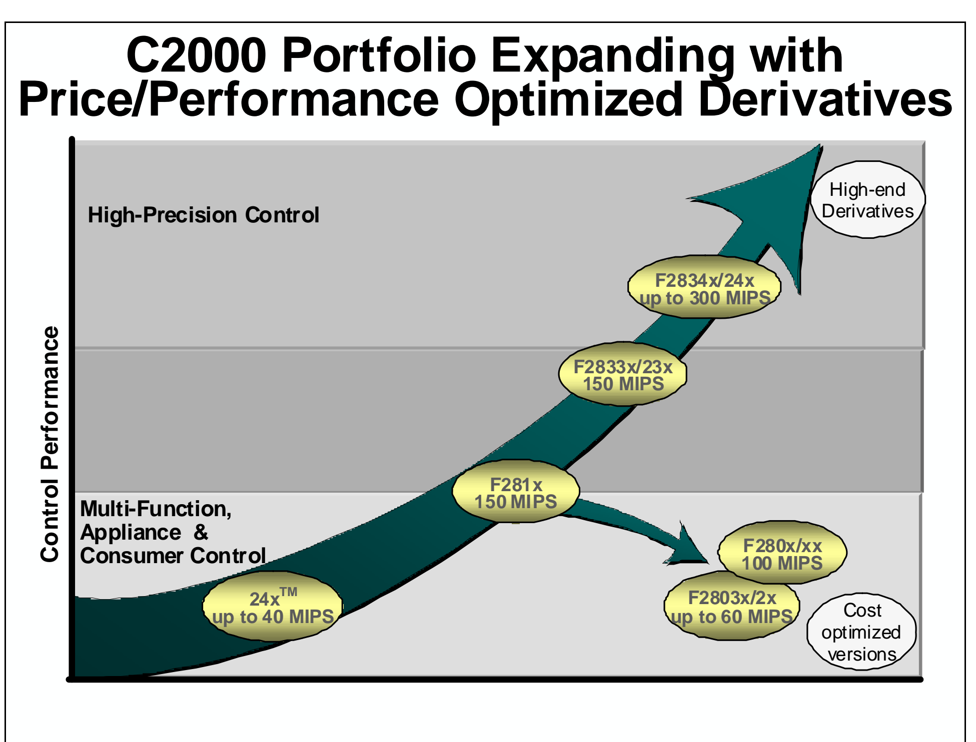
C28x is a trademark of Texas Instruments.
eZdsp is a trademark of Spectrum Digital, Inc.
Copyright © 2009 Texas Instruments. All rights reserved.

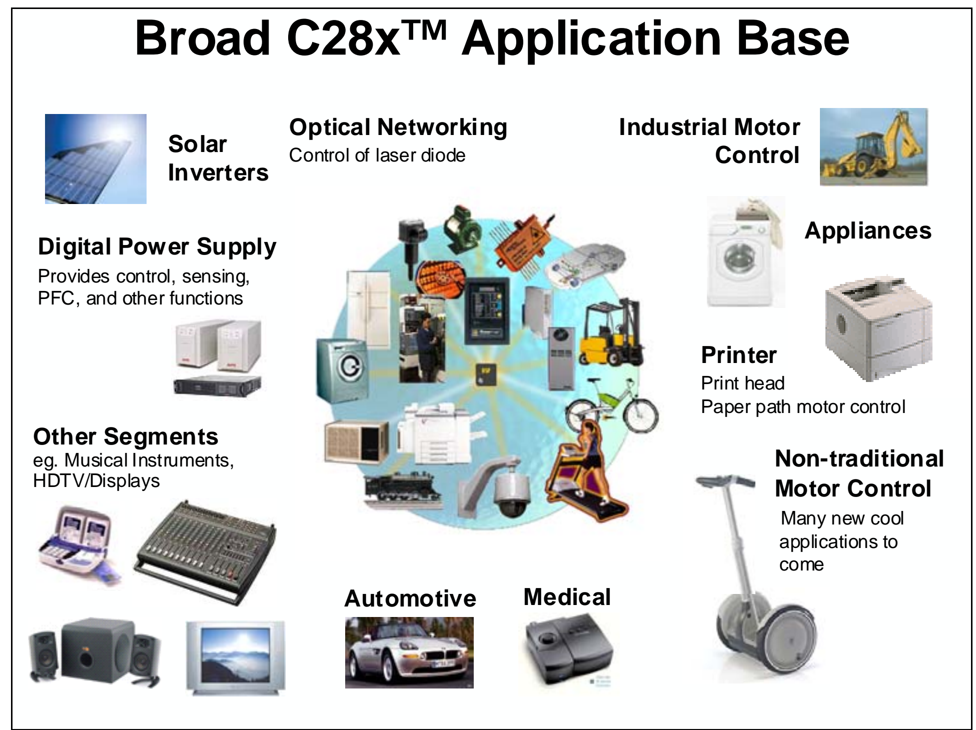
C28x 1-Day Workshop Outline

- **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…**

Introductions

- Name
- Company
- Project Responsibilities
- DSP / Microcontroller Experience
- TMS320 DSP Experience
- Hardware / Software - Assembly / C
- Interests





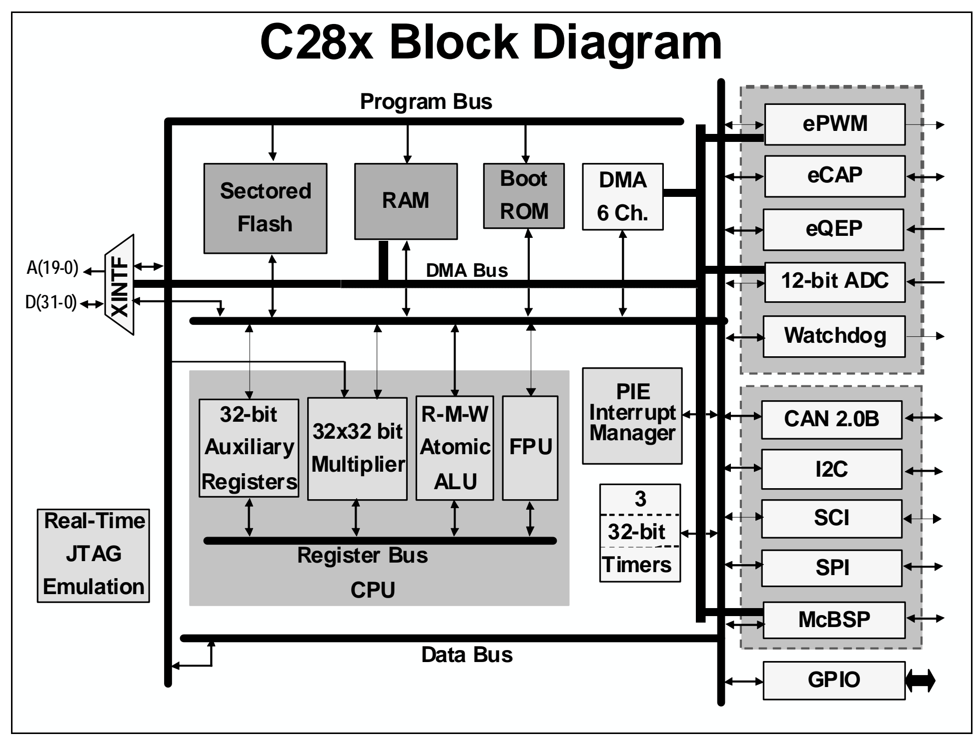
High Performance Controllers
F2833x / F2823x

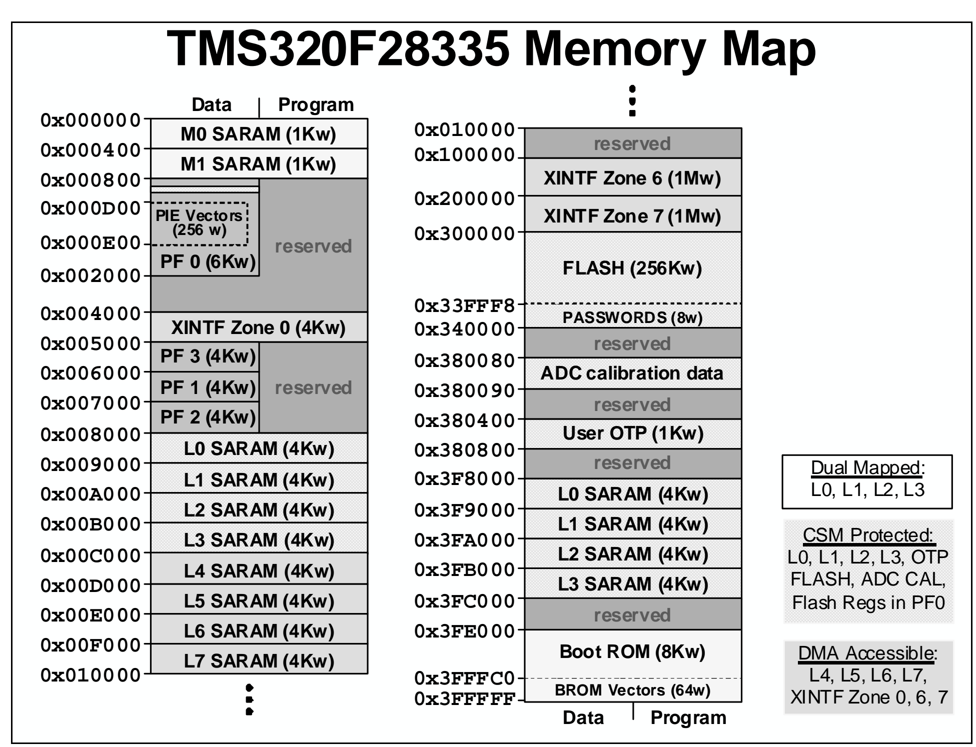
	MHz	FPU	Flash	RAM	DMA	PWM/ HRPWM	CAP/ QEP	Communication Ports
F28335	150	Yes	256	34	Yes	18/6	6/2	SPI, 3x SCI, I²C, 2x McBSP, 2x CAN
F28334	150	Yes	128	34	Yes	18/6	4/2	SPI, 3x SCI, I²C, 2x McBSP, 2x CAN
F28332	100	Yes	64	26	Yes	16/4	4/2	SPI, 2x SCI, I²C, McBSP, 2x CAN
F28235	150	No	256	34	Yes	18/6	6/2	SPI, 3x SCI, I²C, 2x McBSP, 2x CAN
F28234	150	No	128	34	Yes	18/6	4/2	SPI, 3x SCI, I²C, 2x McBSP, 2x CAN
F28232	100	No	64	26	Yes	16/4	4/2	SPI, 2x SCI, I²C, McBSP, 2x CAN

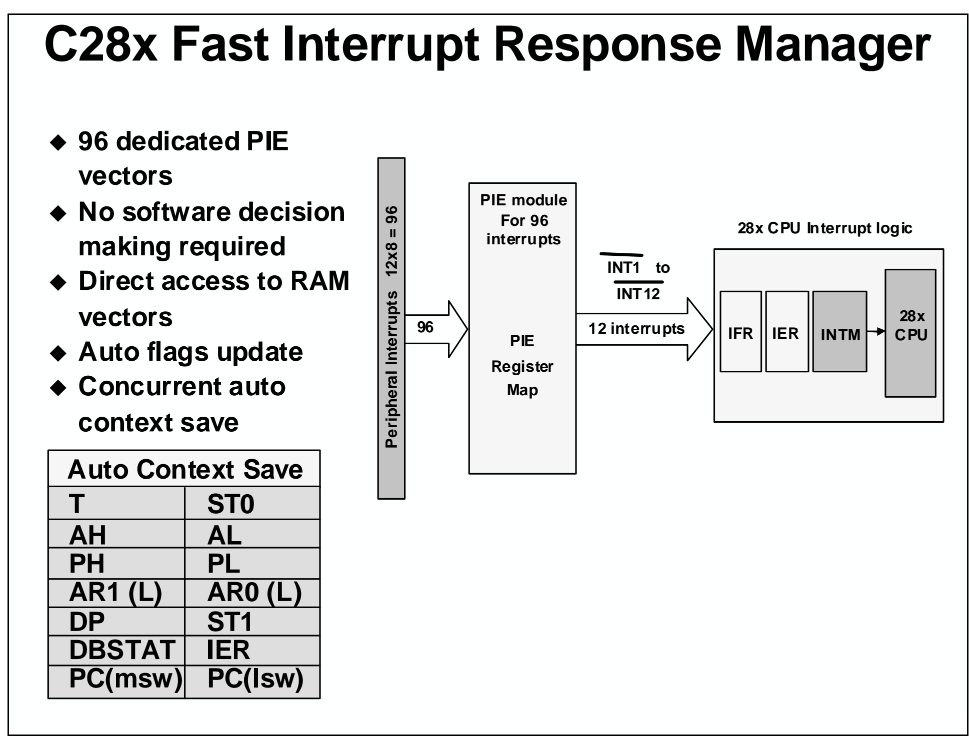
- All devices above are 100% pin-compatible and 100% Software compatible
- All devices have 16/32-bit EMIF, 16 channel ADC at 12.5 MSPS, and 88 GPIO

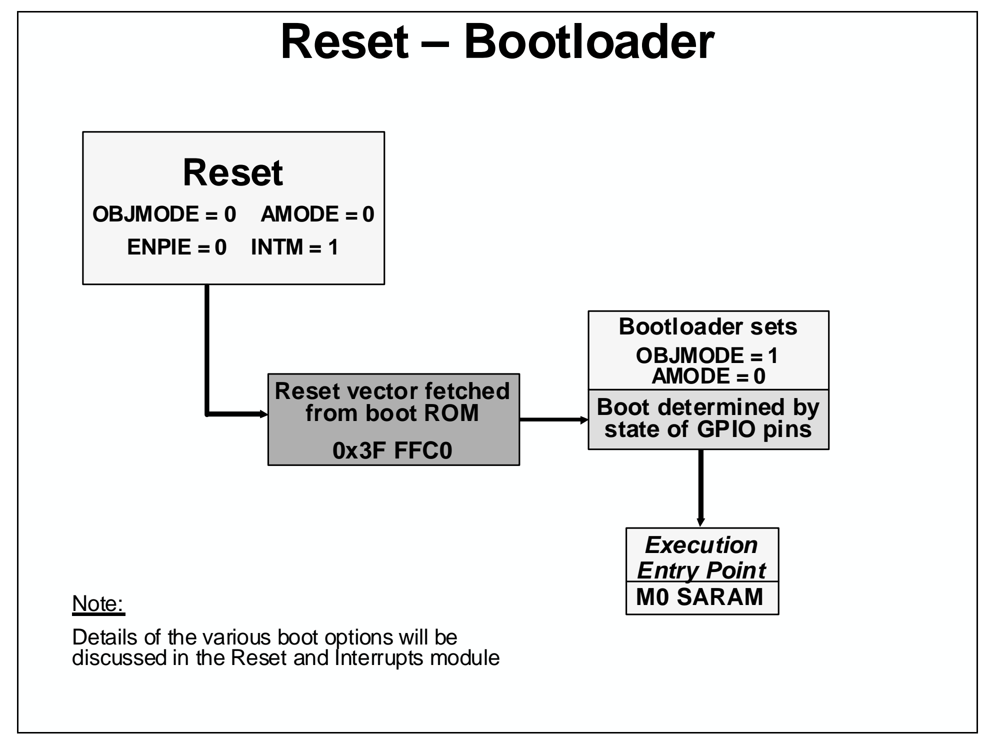
For details and information on other C28x family members refer to the "DSP Selection Guide" and specific "Data Manuals"

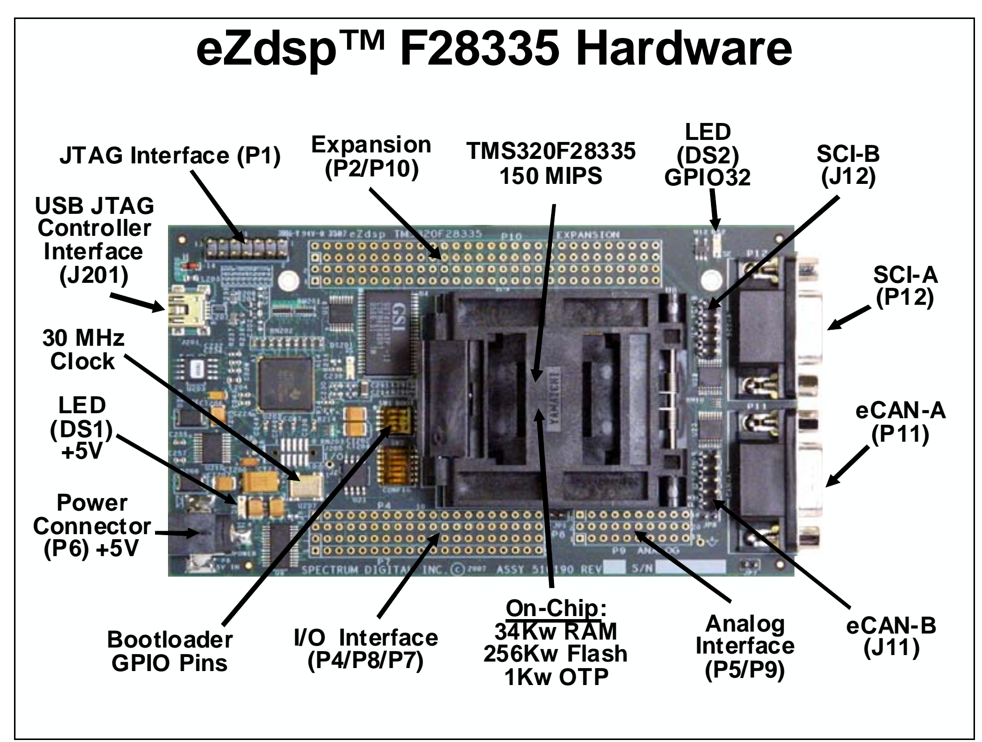
Architecture Overview





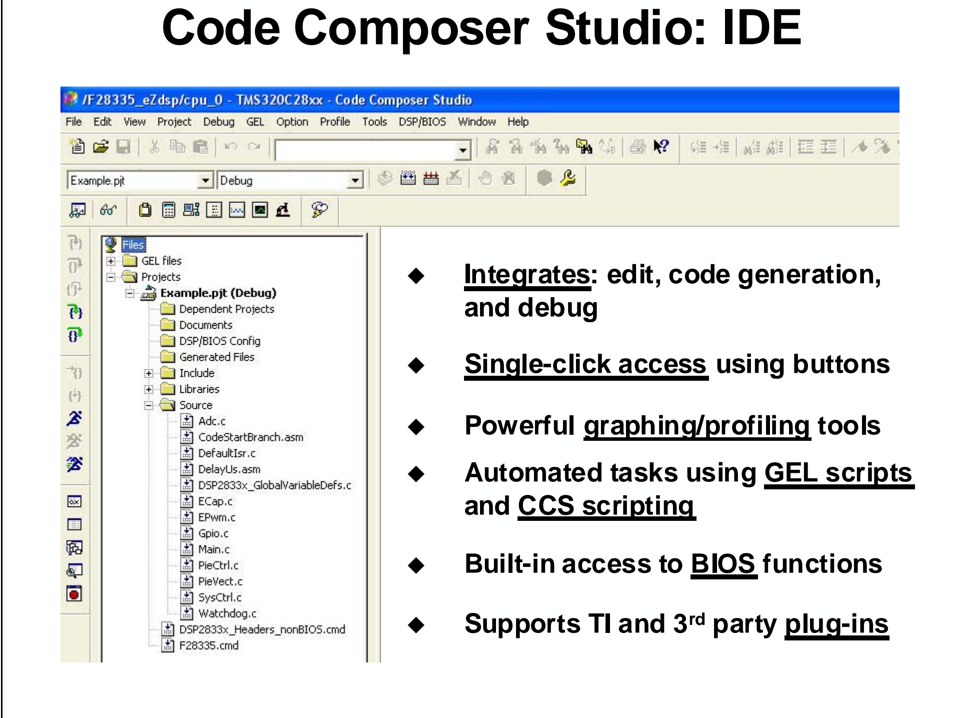

C28x Fast Interrupt Response Manager
96 dedicated PIE vectors
No software decision making required
Direct access to RAM vectors
Auto flags update
Concurrent auto context save
Auto Context Save
T ST0
AH AL
PH PL
AR1 (L) AR0 (L)
DP ST1
DBSTAT IER
PC(msw) PC(lsw)
Peripheral Interrupts 12x8 = 96
96
PIE module For 96 interrupts
PIE Register Map
INT1 to INT12
12 interrupts
28x CPU Interrupt logic
IFR
IER
INTM
28x CPU

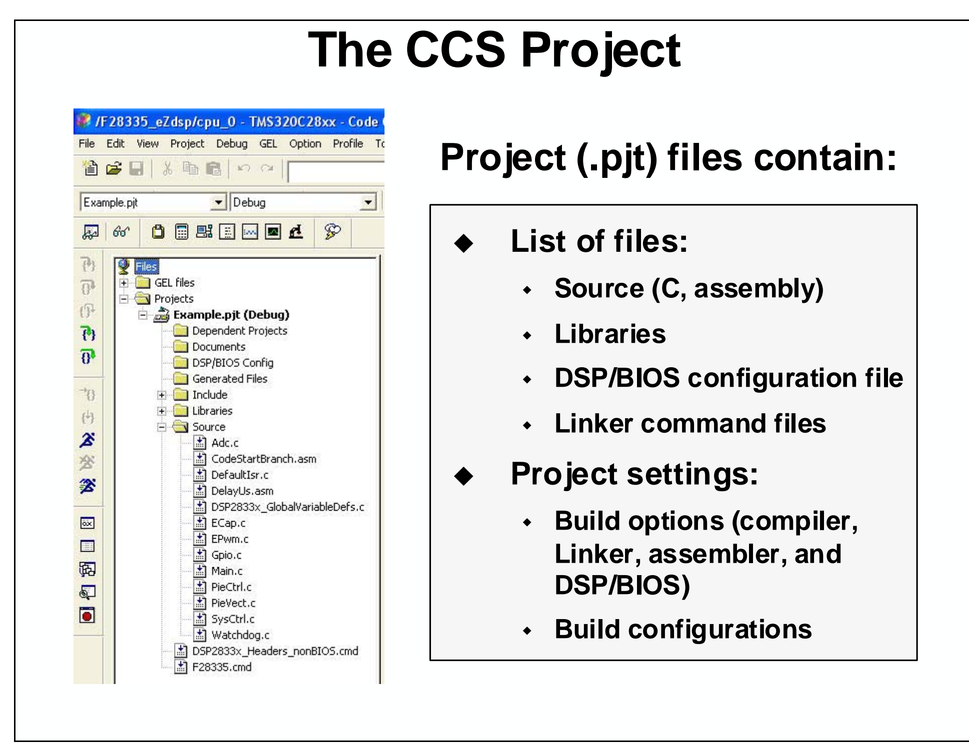

Reset – Bootloader
Reset
OBJMODE = 0 AMODE = 0
ENPIE = 0 INTM = 1
Reset vector fetched from boot ROM
0x3F FFC0
Bootloader sets
OBJMODE = 1
AMODE = 0
Boot determined by state of GPIO pins
Execution Entry Point
M0 SARAM
Note:
Details of the various boot options will be discussed in the Reset and Interrupts module

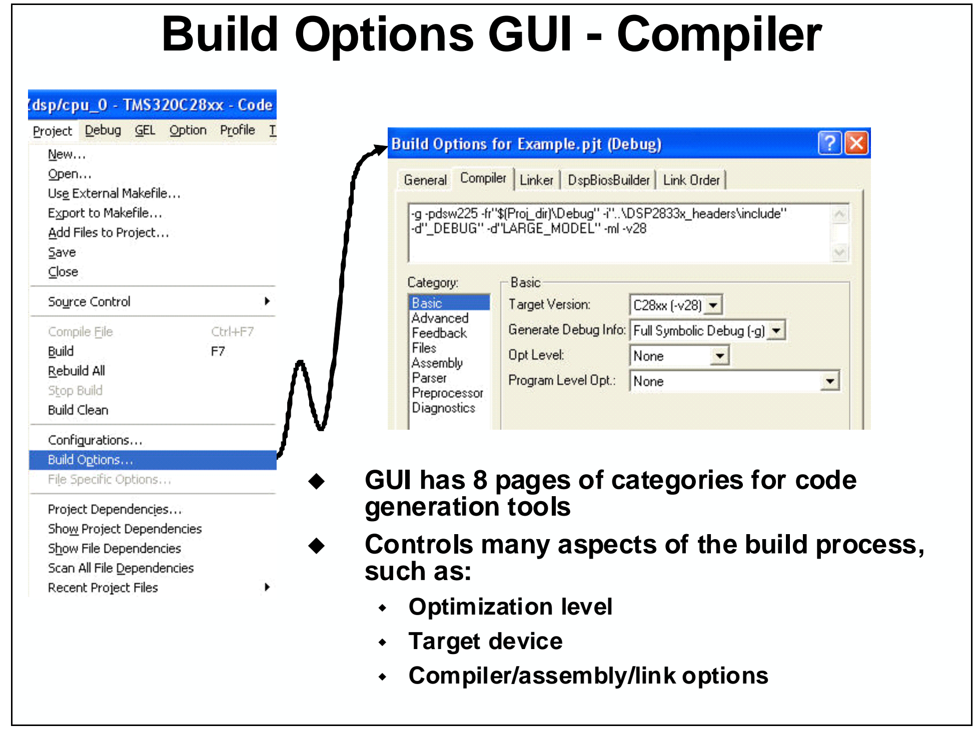

eZdsp™ F28335 Hardware
JTAG Interface (P1)
Expansion (P2/P10)
TMS320F28335 150 MIPS
LED (DS2) GPIO32
SCI-B (J12)
USB JTAG Controller Interface (J201)
SCI-A (P12)
30 MHz Clock
LED (DS1) +5V
eCAN-A (P11)
Power Connector (P6) +5V
Bootloader GPIO Pins
I/O Interface (P4/P8/P7)
On-Chip: 34Kw RAM 256Kw Flash 1Kw OTP
Analog Interface (P5/P9)
eCAN-B (J11)

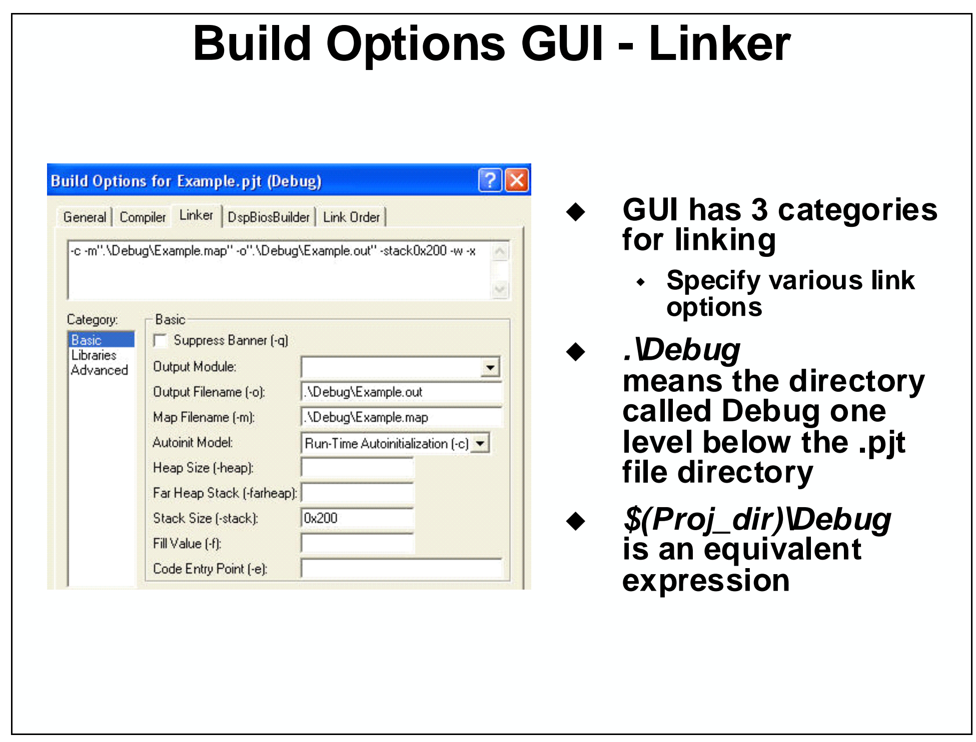
Programming Development Environment

Code Composer Studio



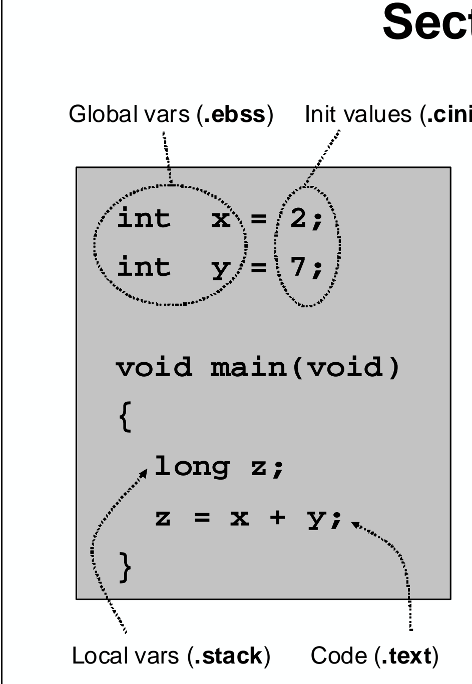



Build Options GUI - Compiler
GUI has 8 pages of categories for code generation tools
Controls many aspects of the build process, such as:
Optimization level
Target device
Compiler/assembly/link options
Build Options for Example.pjt (Debug)
General
Compiler
Linker
DspBiosBuilder
Link Order
Category:
Basic
Advanced
Feedback
Files
Assembly
Parser
Preprocessor
Diagnostics
Target Version:
Generate Debug Info:
Opt Level:
Program Level Opt.:
Project
Debug
New...
Open...
Use External Makefile...
Export to Makefile...
Add Files to Project...
Save
Close
Source Control
Compile File
Build
Rebuild All
Stop Build
Build Clean
Configurations...
Build Options...
File Specific Options...
Project Dependencies...
Show Project Dependencies
Show File Dependencies
Scan All File Dependencies
Recent Project Files


Build Options GUI - Linker
GUI has 3 categories for linking
Specify various link options
.\Debug means the directory called Debug one level below the .pjt file directory
$(Proj_dir)\Debug is an equivalent expression
Build Options for Example.pjt (Debug)
General
Compiler
Linker
DspBiosBuilder
Link Order
-c -m".\Debug\Example.map" -o".\Debug\Example.out" -stack0x200 -w -x
Category:
Basic
Libraries
Advanced
Suppress Banner (-q)
Output Module:
Output Filename (-o):
Map Filename (-m):
Autoinit Model:
Heap Size (-heap):
Far Heap Stack (-farheap):
Stack Size (-stack):
Fill Value (-f):
Code Entry Point (-e):

Linking Sections in Memory

Sections



- 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

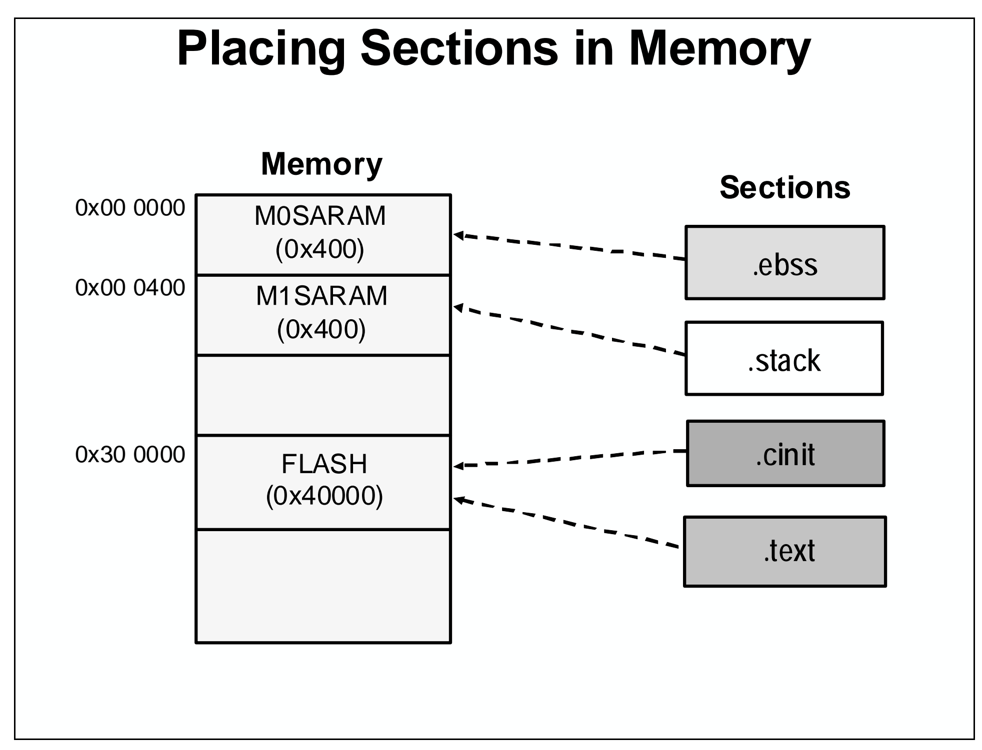
Initialized Sections

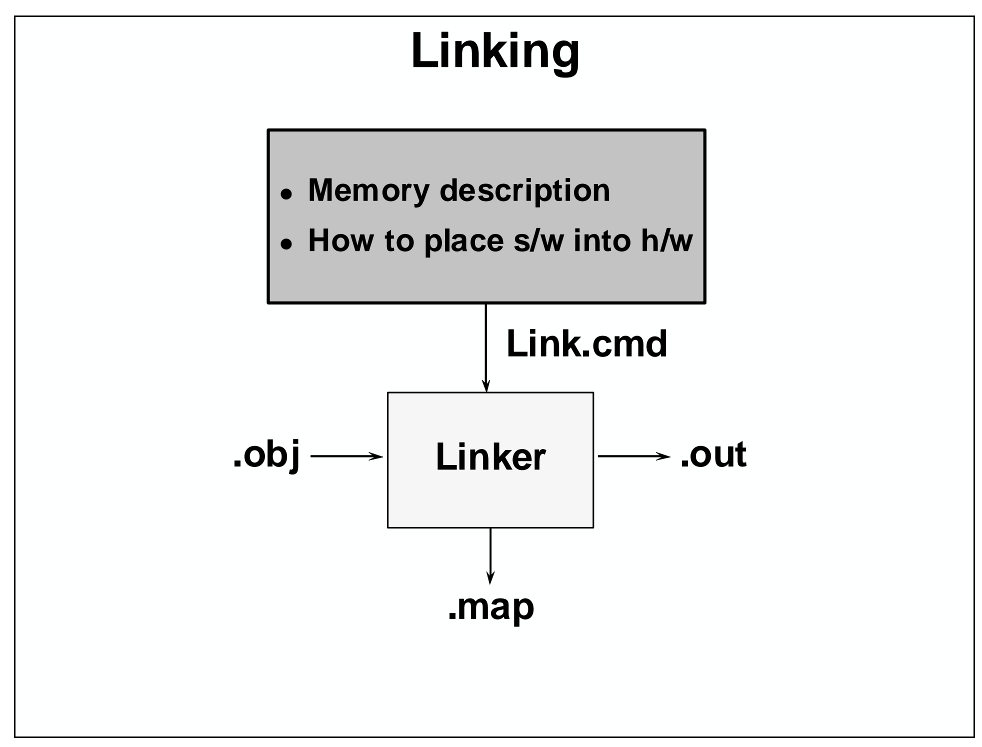
Name	Description	Link Location
.text	code	FLASH
.cinit	initialization values for global and static variables	FLASH
.econst	constants (e.g. const int k = 3;)	FLASH
.switch	tables for switch statements	FLASH
.pinit	tables for global constructors (C++)	FLASH

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
Memory
Sections
0x00 0000
M0SARAM
(0x400)
0x00 0400
M1SARAM
(0x400)
0x30 0000
FLASH
(0x40000)
.ebss
.stack
.cinit
.text


Linking
• Memory description
• How to place s/w into h/w
Link.cmd
.obj
Linker
.out
.map

Linker Command File

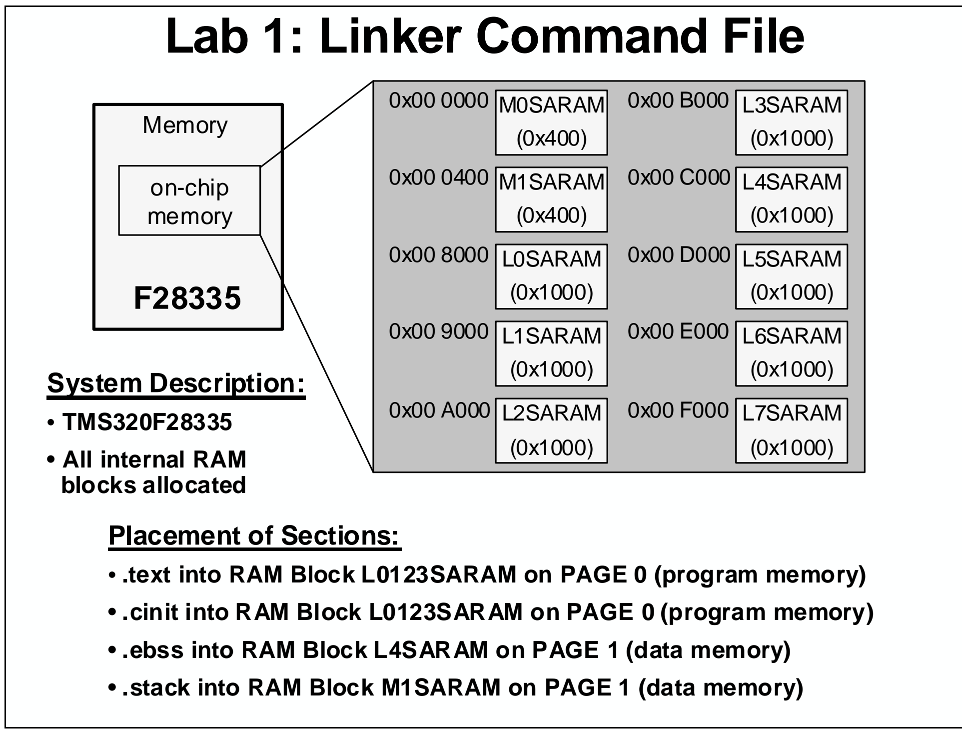
```
MEMORY
{
  PAGE 0:         /* Program Memory */
   FLASH:   origin = 0x300000, length = 0x40000

  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

- TMS320F28335
- All internal RAM blocks allocated

Placement of Sections:

- .text into RAM Block L0123SARAM on PAGE 0 (program memory)
- .cinit into RAM Block L0123SARAM on PAGE 0 (program memory)
- .ebss into RAM Block L4SARAM on PAGE 1 (data memory)
- .stack into RAM Block M1SARAM on PAGE 1 (data memory)

➢ **Procedure**

Open a Project

1. Double click on the Code Composer Studio icon on the desktop. Maximize Code Composer Studio to fill your screen. Code Composer Studio has a *Connect/Disconnect* feature which allows the target to be dynamically connected and disconnected. This will reset the JTAG link and also enable "hot swapping" a target board. Connect to the target.

Click: `Debug` → `Connect`

The menu bar (at the top) lists File ... Help. Note the horizontal tool bar below the menu bar and the vertical tool bar on the left-hand side. The window on the left is the project window and the large right-hand window is your workspace.

2. A *project* is all the files you will need to develop an executable output file (`.out`) which can be run on the DSP hardware. A project named `Lab1.pjt` has been created for this lab. Open the project by clicking:

   `Project` → `Open…`

   and look in `C:\C28x\LABS\LAB1`. This *.pjt* file will invoke all the necessary tools (compiler, assembler, linker) to build the project. It will also create a `debug` folder that will hold immediate output files.

3. In the project window on the left, click the plus sign (+) to the left of `Project`. Now, click on the plus sign next to `Lab1.pjt`. Notice that the `Lab1.cmd` file is listed. Click on `Source` to see the current source file list (i.e. `Lab1.c`).

4. A test file named `Lab1.c` has been added to the project. This file will be used in this exercise to demonstrate some features of Code Composer Studio.

Project Build Options

5. There are numerous build options in the project. The default option settings are sufficient for getting started. We will inspect a couple of the default linker options at this time.

   Click: `Project` → `Build Options…`

6. Select the Linker tab. Notice that .out and .map files are being created. The .out file is the executable code that will be loaded into the DSP. The .map file will contain a linker report showing memory usage and section addresses in memory. The Stack Size has been set to 0x200.

7. Select `OK` and the Build Options window will close.

Linker Command File – Lab1.cmd

8. 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 L0SARAM, L1SARAM, L2SARM, and L3SARAM have been combined into a single memory block called L0123SARAM. This combined memory block has been placed in program memory on page 0, and the other memory blocks have been placed in data memory on page 1.

9. 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_ml.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

10. The top four buttons on the horizontal toolbar control code generation. Hover your mouse over each button as you read the following descriptions:

Button	Name	Description
1	Compile File	Compile, assemble the current open file
2	Incremental Build	Compile, assemble only changed files, then link
3	Rebuild All	Compile, assemble all files, then link
4	Stop Build	Stop code generation

11. Code Composer Studio can automatically load the output file after a successful build. On the menu bar click: `Option` → `Customize…` and select the `"Program/Project/CIO"` tab, check `"Load Program After Build"`.

    Also, Code Composer Studio can automatically connect to the target when started. Select the `"Debug Properties"` tab, check `"Connect to the target at startup"`, then click `OK`.

12. Click the `"Build"` button and watch the tools run in the build window. Check for errors (we have deliberately put an error in `Lab1.c`). When you get an error, scroll the build window at the bottom of the Code Composer Studio screen until you see the error message (in red), and simply double-click the error message. The editor will automatically open the source file containing the error, and position the mouse cursor at the correct code line.

13. Fix the error by adding a semicolon at the end of the `"z = x + y"` statement. For future knowlege, realize that a single code error can sometimes generate multiple error messages at build time. This was not the case here.

14. Rebuild the project (there should be no errors this time). The output file should automatically load. The Program Counter should be pointing to `_c_int00` in the Disassembly Window.

15. Under `Debug` on the menu bar click `"Go Main"`. This will run through the DSP/BIOS C-environment initialization routine and stop at `main()` in `Lab1.c`.

Debug Enviroment 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 windows, and watch windows.

16. Open a *memory window* to view the global variable `"z"`.

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

Type "`&z`" into the address field and then enter. Note that you must use the ampersand (meaning "address of") when using a symbol in a memory window address box. Also note that Code Composer Studio is case sensitive.

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

17. Open the *watch window* to view the local variables `x` and `y`.

    Click: `View` → `Watch Window` on the menu bar.

    Click the "Watch Locals" tab and notice that the local variables `x` and `y` are already present. The watch 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 window by setting the address to "SP" after the code function has been entered).

18. We can also add global variables to the watch window if desired. Let's add the global variable "`z`".

    Click the "Watch 1" tab at the bottom of the watch window. In the empty box in the "Name" column, type "`z`" and then enter. Note that you do not use an ampersand here. The watch window knows you are specifying a symbol. Check that the watch window and memory window both report the same value for "`z`". Trying changing the value in one window, and notice that the value also changes in the other window.

Single-stepping the Code

19. Click the "Watch Locals" tab at the bottom of the watch window. Single-step through `main()` by using the <F11> key (or you can use the `Single Step` button on the vertical 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?

End of Exercise

Peripheral Register Header Files

Traditional Approach to C Coding

```
#define ADCTRL1      (volatile unsigned  int *)0x00007100
#define ADCTRL2      (volatile unsigned  int *)0x00007101
                      ...
void main(void)
{
    *ADCTRL1 = 0x1234;            //write entire register
    *ADCTRL2 |= 0x4000;           //reset sequencer #1
}
```

Advantages
- Simple, fast and easy to type
- Variable names exactly match register names (easy to remember)

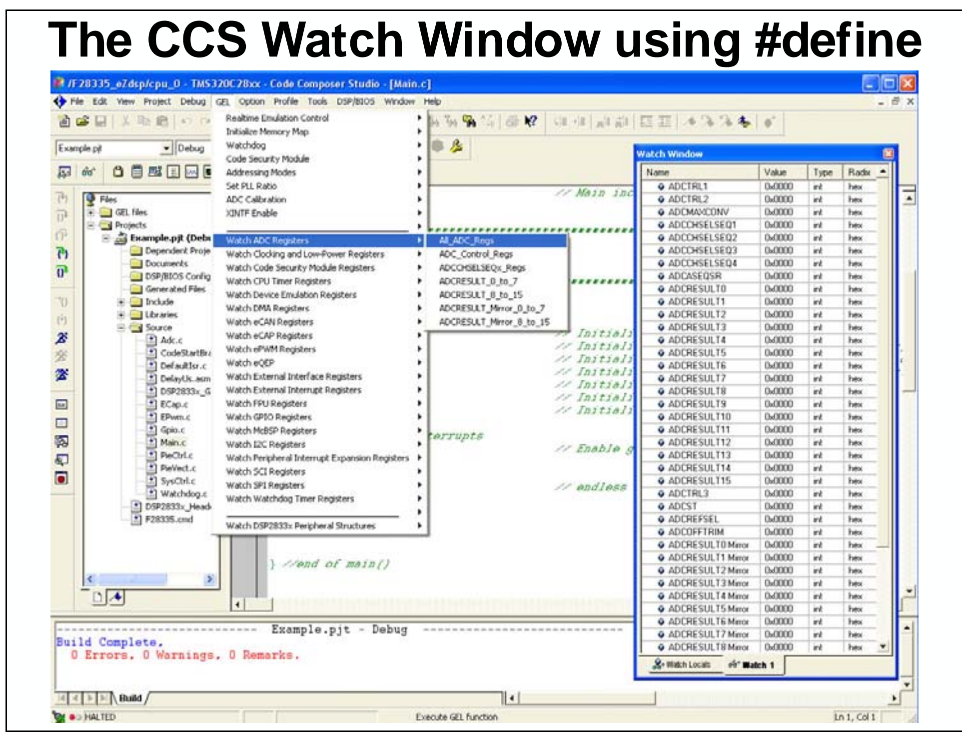
Disadvantages
- Requires individual masks to be generated to manipulate individual bits
- Cannot easily display bit fields in Watch window
- Will generate less efficient code in many cases

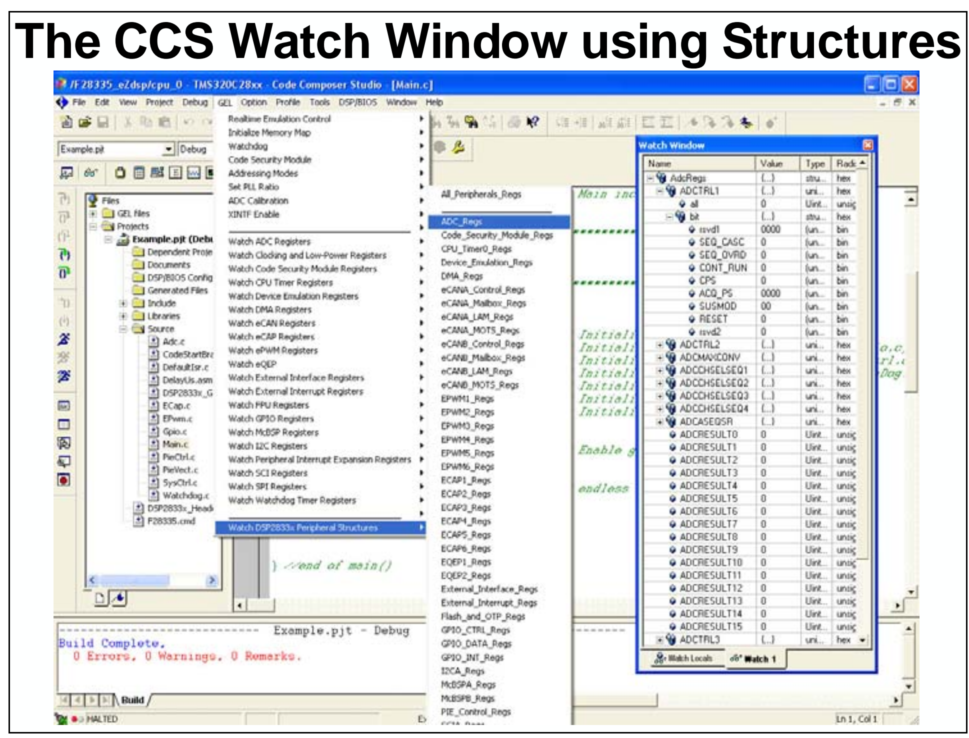
Structure Approach to C Coding

```
void main(void)
{
    AdcRegs.ADCTRL1.all = 0x1234;        //write entire register
    AdcRegs.ADCTRL2.bit.RST_SEQ1 = 1;    //reset sequencer #1
}
```

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

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


The CCS Watch Window using #define


The CCS Watch Window using Structures

Structure Naming Conventions

- **The DSP2833x header files define:**
  - All of the peripheral structures
  - All of the register names
  - All of the bit field names
  - All of the register addresses

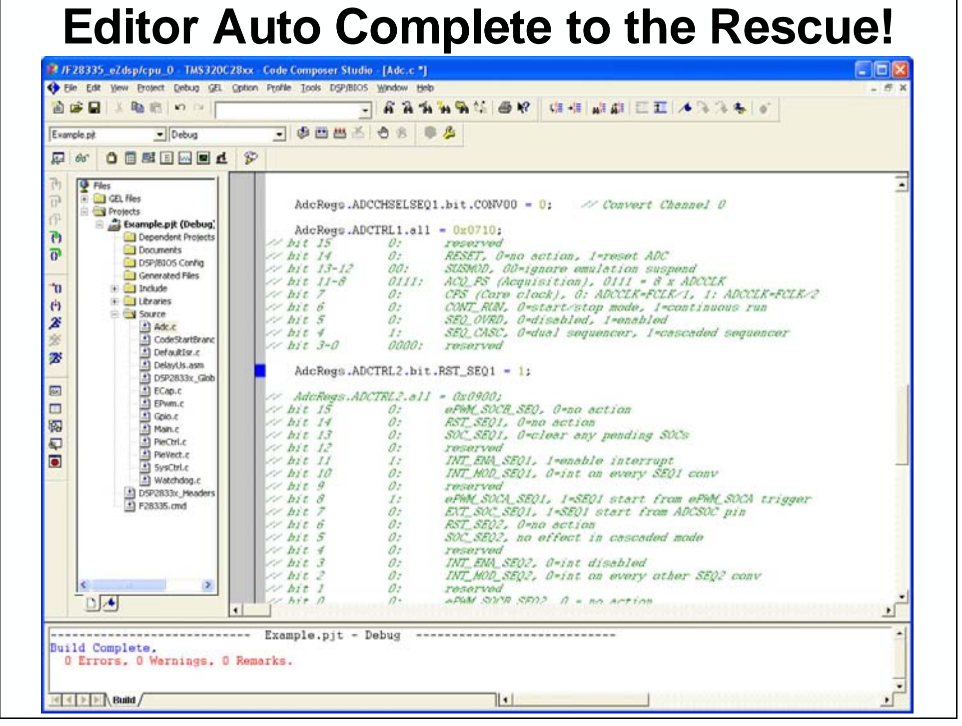
```
PeripheralName.RegisterName.all            // Access full 16 or 32-bit register
PeripheralName.RegisterName.half.LSW       // Access low 16-bits of 32-bit register
PeripheralName.RegisterName.half.MSW       // Access high 16-bits of 32-bit register
PeripheralName.RegisterName.bit.FieldName  // Access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

Notes: [1] "PeripheralName" are assigned by TI and found in the DSP2833x 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!



DSP2833x Header File Package

(http://www.ti.com, literature # SPRC530)

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

\DSP2833x_headers\include	→ .h files
\DSP2833x_headers\cmd	→ linker .cmd files
\DSP2833x_headers\gel	→ .gel files for CCS
\DSP2833x_examples	→ ‘2833x examples
\DSP2823x_examples	→ ‘2823x examples
\doc	→ documentation

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 "DSP2833x_Device.h"

Void InitAdc(void)
{
   /* Reset the ADC module */
   AdcRegs.ADCTRL1.bit.RESET = 1;

   /* configure the ADC register */
   AdcRegs.ADCTRL1.all = 0x0710;
};
```

DSP2833x_Adc.h

```
/* ADC Individual Register Bit Definitions */
struct ADCTRL1_BITS {     // bits  description
   Uint16  rsvd1:4;          // 3:0 reserved
   Uint16  SEQ_CASC:1;    // 4 Cascaded sequencer mode
   Uint16  SEQ_OVRD:1     // 5 Sequencer override
   Uint16  CONT_RUN:1;    // 6 Continuous run
   Uint16  CPS:1;            // 7 ADC core clock prescaler
   Uint16  ACQ_PS:4;       // 11:8 Acquisition window size
   Uint16  SUSMOD:2;      // 13:12 Emulation suspend mode
   Uint16  RESET:1;         // 14 ADC reset
   Uint16  rsvd2:1;          // 15 reserved
};

/* Allow access to the bit fields or entire register */
union ADCTRL1_REG {
  Uint16           all;
  struct ADCTRL1_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**

DSP2833x_Device.h	DSP2833x_DevEmu.h	DSP2833x_SysCtrl.h
DSP2833x_PieCtrl.h	DSP2833x_Adc.h	DSP2833x_CpuTimers.h
DSP2833x_ECan.h	DSP2833x_ECap.h	DSP2833x_EPwm.h
DSP2833x_EQep.h	DSP2833x_Gpio.h	DSP2833x_I2c.h
DSP2833x_Sci.h	DSP2833x_Spi.h	DSP2833x_XIntrupt.h
DSP2833x_PieVect.h	DSP2833x_DefaultIsr.h	DSP2833x_DMA.h
DSP2833x_Mcbsp.h	DSP2833x_Xintf.h	

- ***DSP2833x_Device.h***
  - Main include file (for ‘2833x and ‘2823x devices)
  - Will include all other .h files
  - Include this file in each source file:

```
#include "DSP2833x_Device.h"
```

Global Variable Definitions File

DSP2833x_GlobalVariableDefs.c

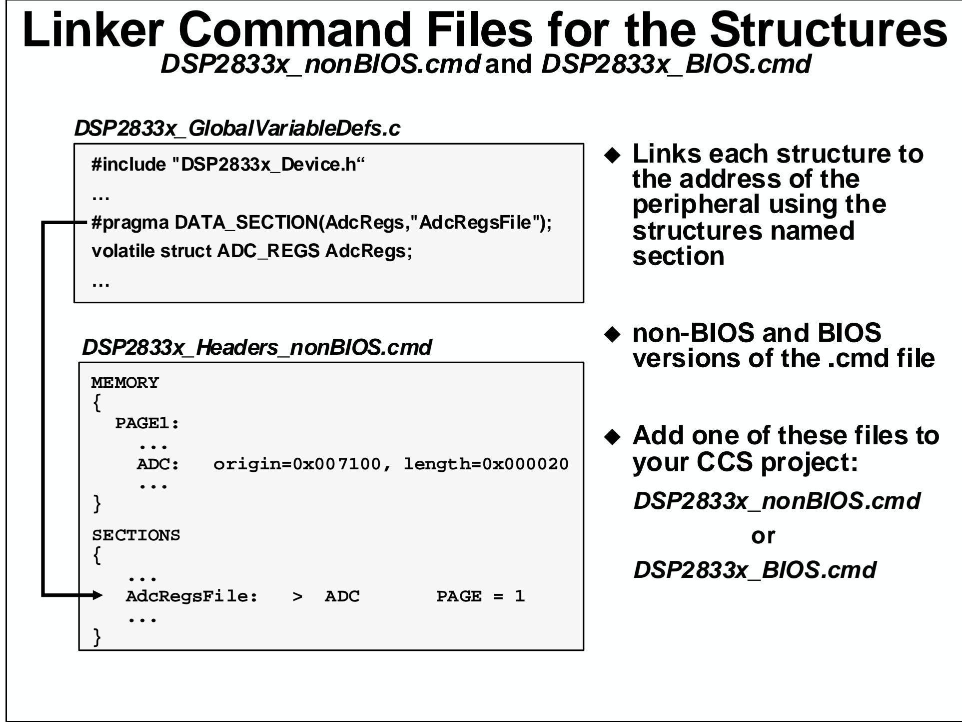
- **Declares a global instantiation of the structure for each peripheral**
- **Each structure put in its own section using a DATA_SECTION pragma to allow linking to correct memory (see next slide)**

DSP2833x_GlobalVariableDefs.c

```
#include "DSP2833x_Device.h"
…
#pragma DATA_SECTION(AdcRegs,"AdcRegsFile");
volatile struct ADC_REGS AdcRegs;
…
```

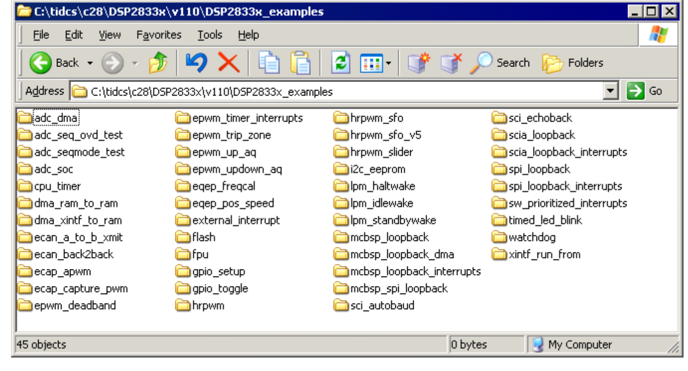
- **Add this file to your CCS project:**

  DSP2833x_GlobalVariableDefs.c



Peripheral Specific Examples

- Example(s) projects for each peripheral
- Helpful to get you started
- Seperate projects for '2833x and '2823x
  - '2823x projects configured for no FPU



Peripheral Register Header Files Summary

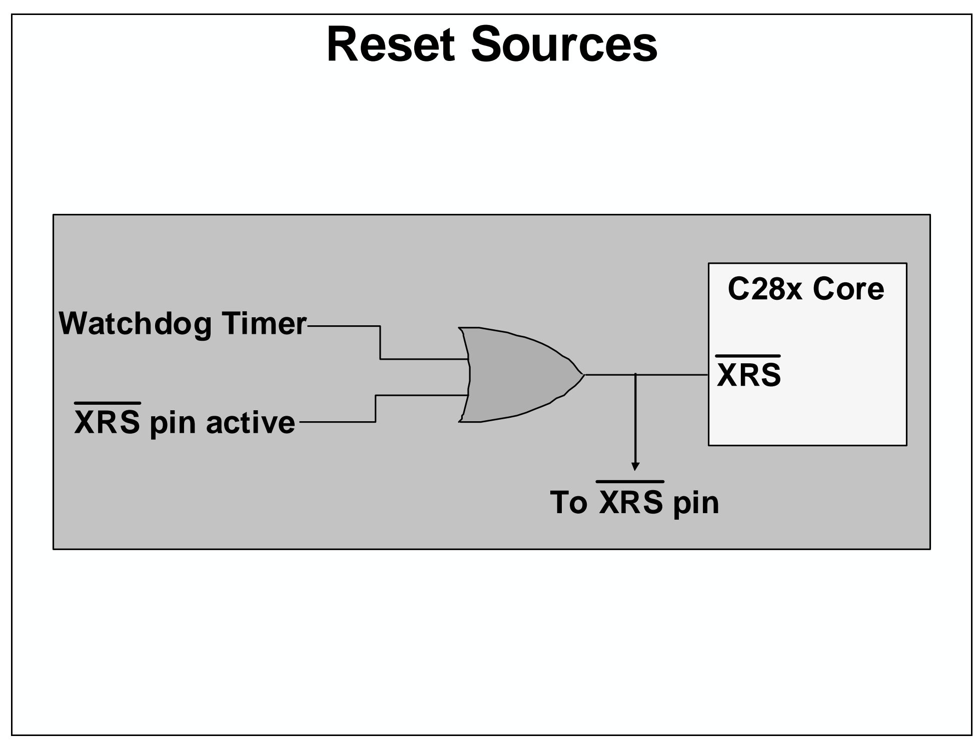
- Easier code development
- Easy to use
- Generates most efficient code
- Increases effectiveness of CCS watch window
- TI has already done all the work!
  - Use the correct header file package for your device:

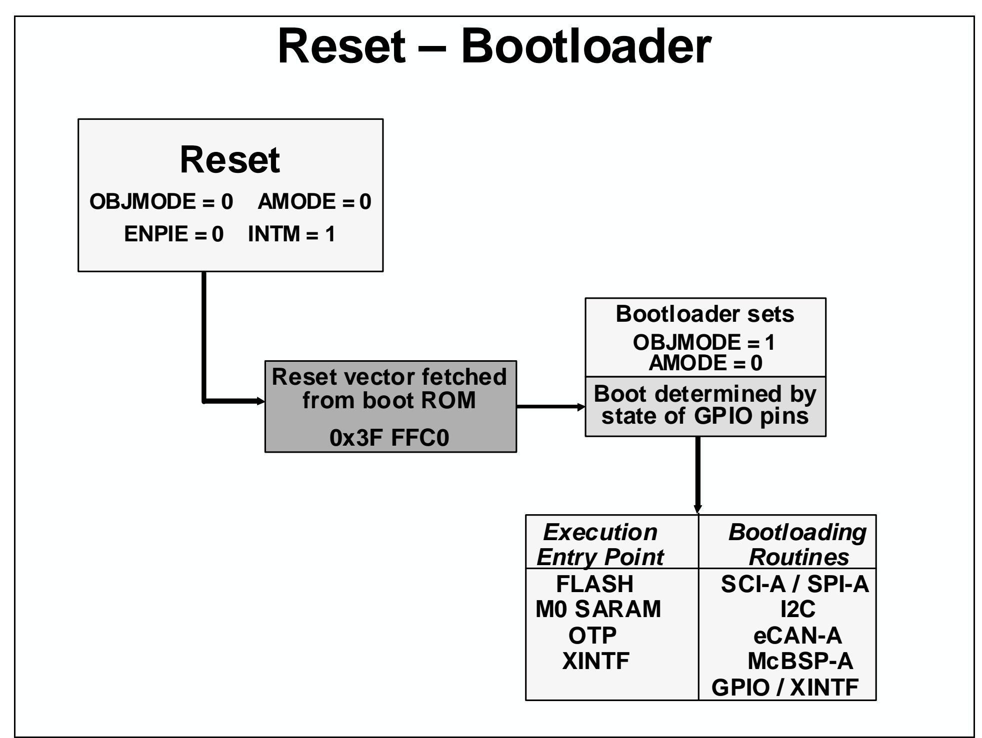
Device	Literature #
F2833x and F2823x	# SPRC530
F280x and F2801x	# SPRC191
F2804x	# SPRC324
F281x	# SPRC097

Go to http://www.ti.com and enter the literature number in the keyword search box

Reset, Interrupts and System Initialization

Reset

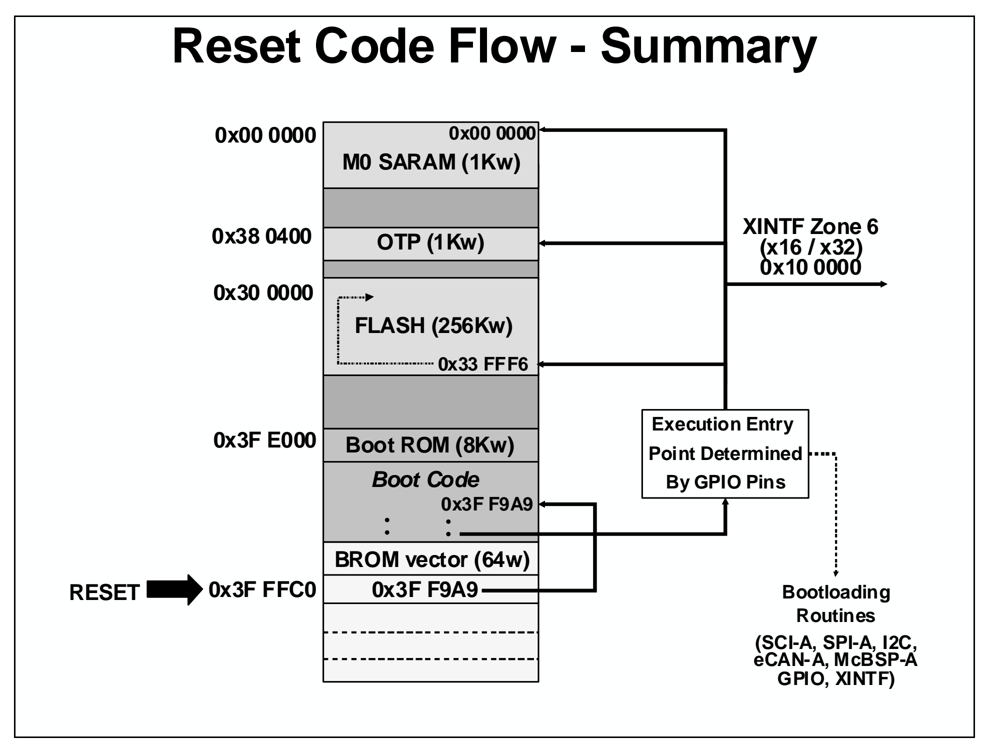




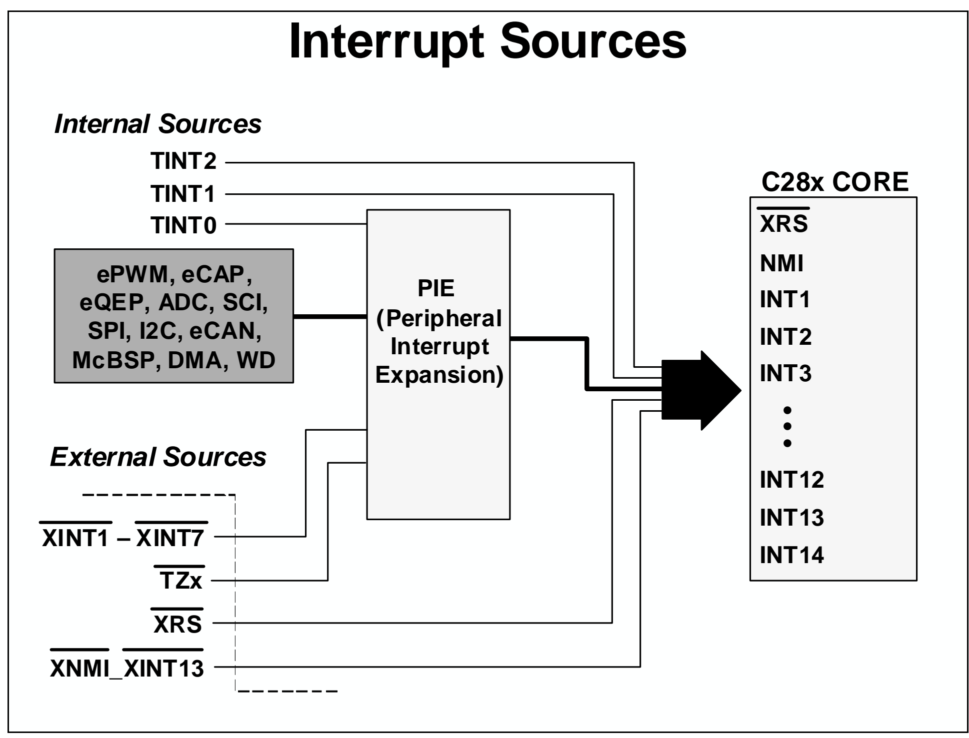
Bootloader Options

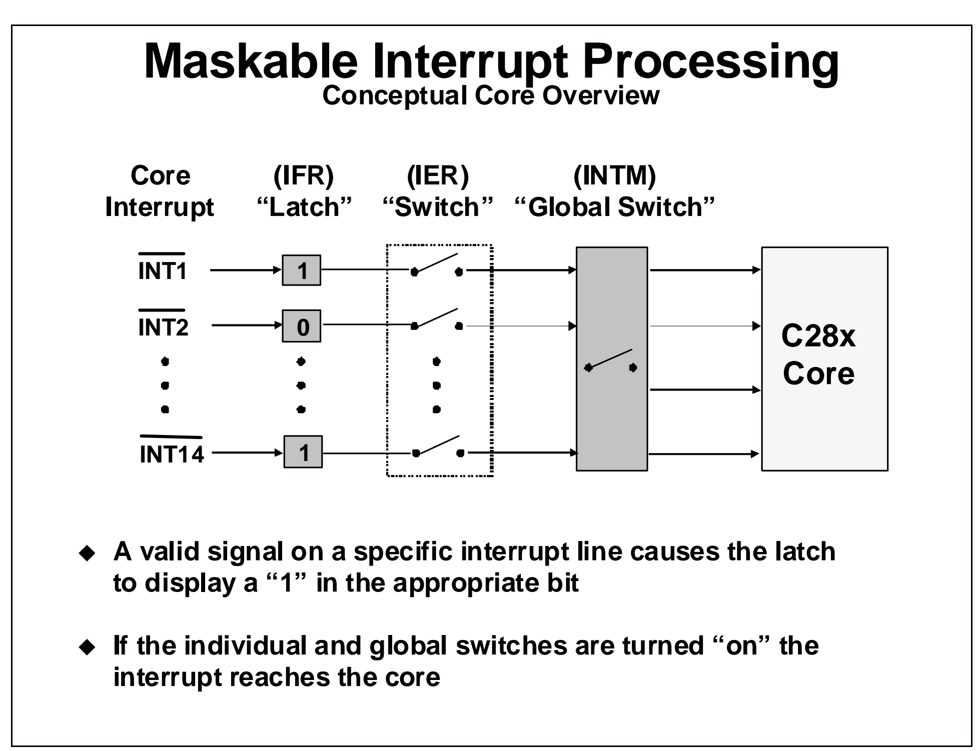
GPIO pins 87 / XA15	86 / XA14	85 / XA13	84 / XA12	
1	1	1	1	jump to *FLASH* address 0x33 FFF6
1	1	1	0	bootload code to on-chip memory via *SCI-A*
1	1	0	1	bootload external EEPROM to on-chip memory via *SPI-A*
1	1	0	0	bootload external EEPROM to on-chip memory via *I2C*
1	0	1	1	Call CAN_Boot to load from *eCAN-A* mailbox 1
1	0	1	0	bootload code to on-chip memory via *McBSP-A*
1	0	0	1	jump to *XINTF* Zone 6 address 0x10 0000 for 16-bit data
1	0	0	0	jump to *XINTF* Zone 6 address 0x10 0000 for 32-bit data
0	1	1	1	jump to *OTP* address 0x38 0400
0	1	1	0	bootload code to on-chip memory via *GPIO port A* (parallel)
0	1	0	1	bootload code to on-chip memory via *XINTF* (parallel)
0	1	0	0	jump to *M0 SARAM* address 0x00 0000
0	0	1	1	branch to check boot mode
0	0	1	0	branch to Flash without ADC calibration (TI debug only)
0	0	0	1	branch to M0 SARAM without ADC calibration (TI debug only)
0	0	0	0	branch to SCI-A without ADC calibration (TI debug only)

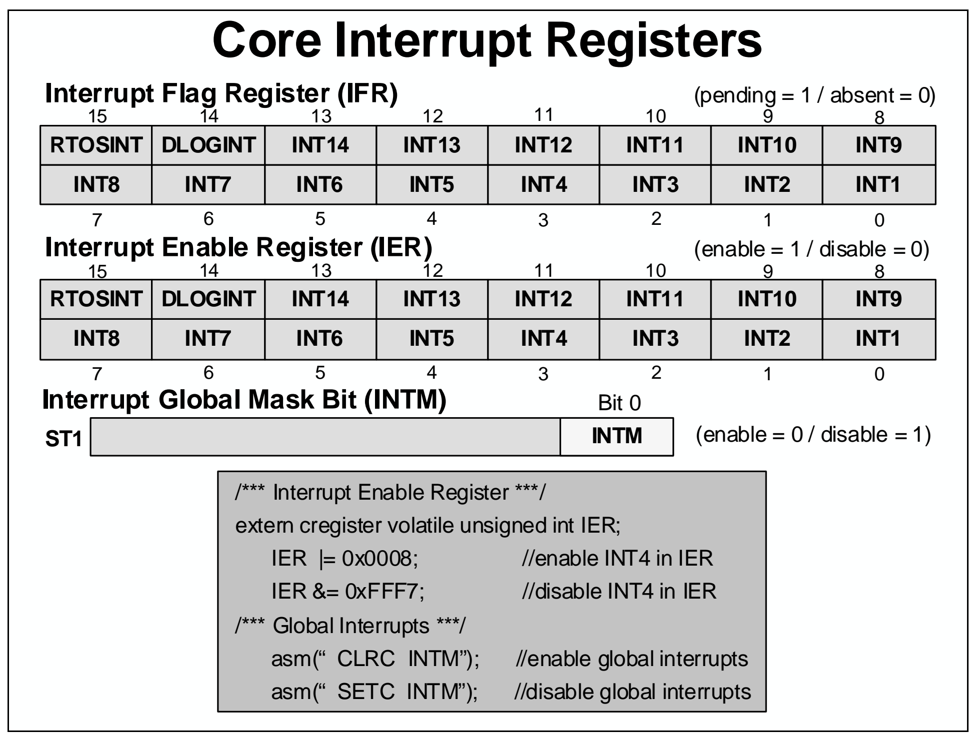
Reset Code Flow - Summary



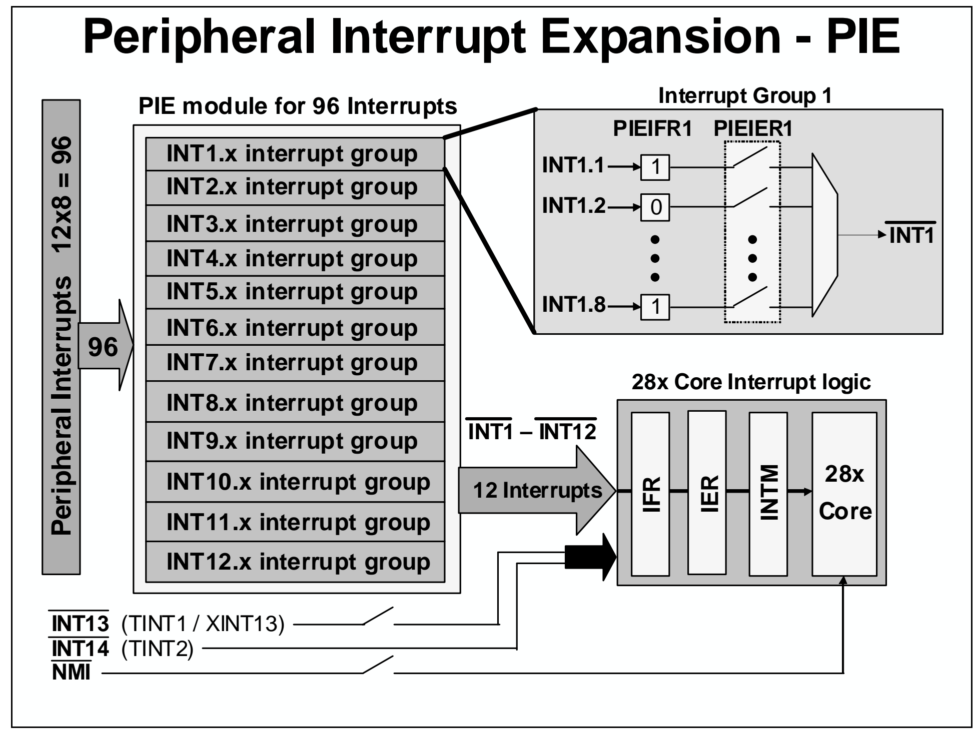
Interrupts

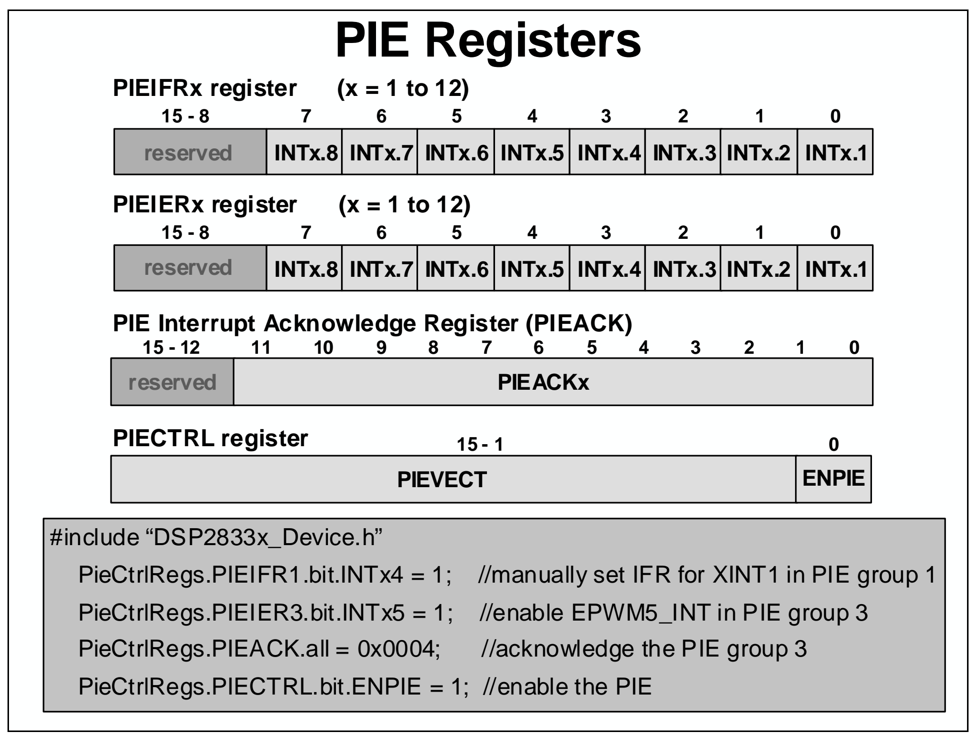






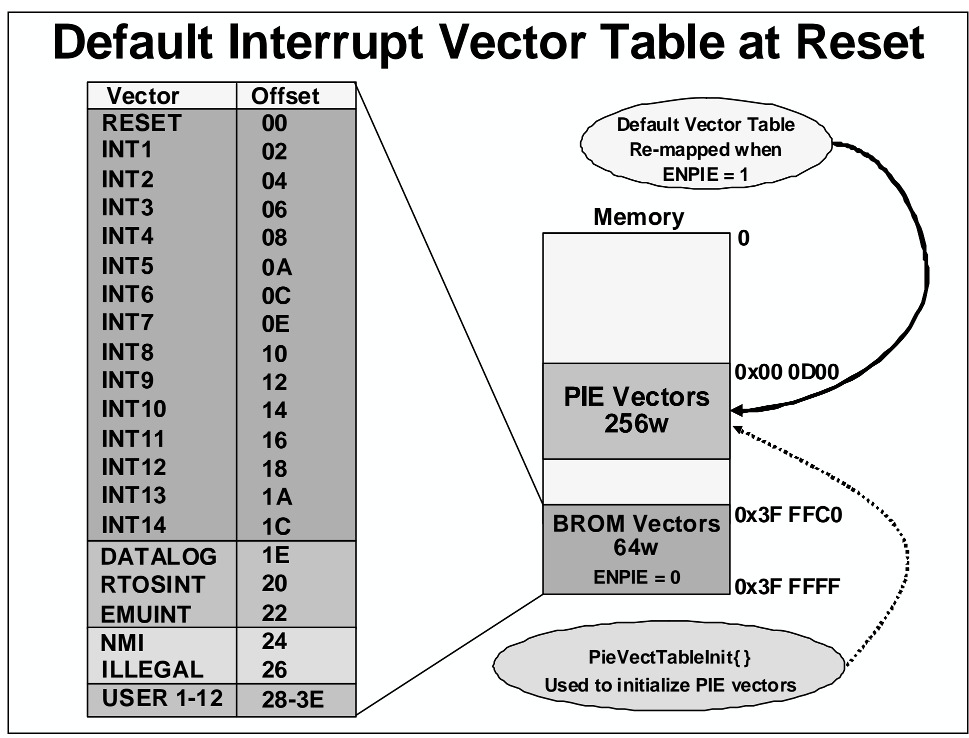
Peripheral Interrupt Expansion (PIE)



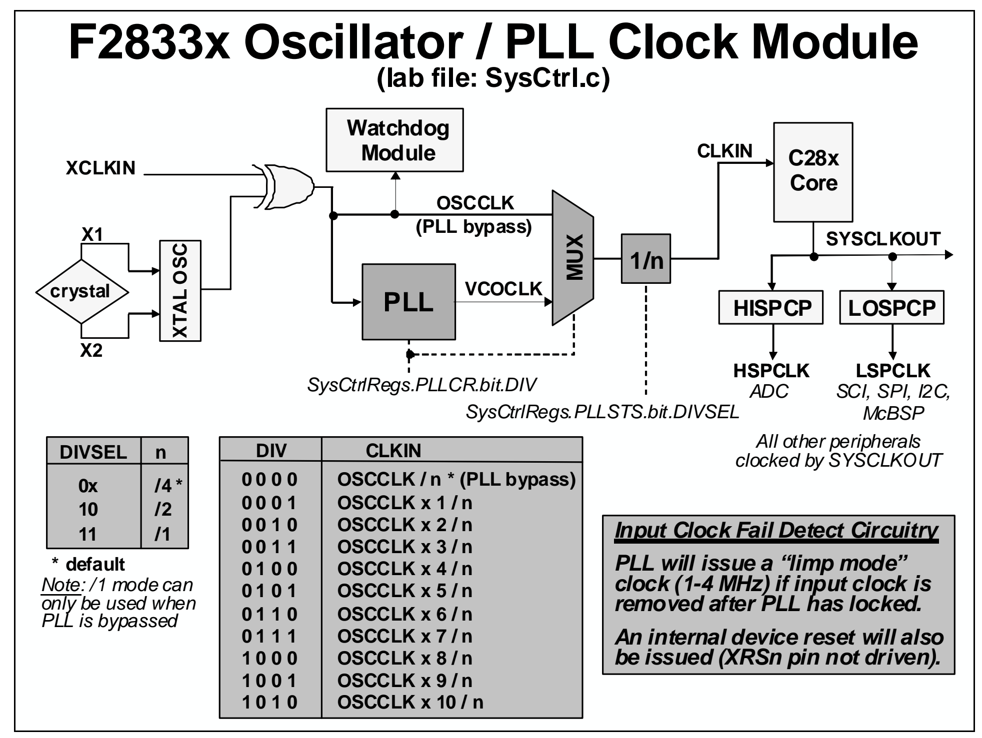


F2833x PIE Interrupt Assignment Table

	INTx.8	INTx.7	INTx.6	INTx.5	INTx.4	INTx.3	INTx.2	INTx.1
INT1	WAKEINT	TINT0	ADCINT	XINT2	XINT1		SEQ2INT	SEQ1INT
INT2			EPWM6_TZINT	EPWM5_TZINT	EPWM4_TZINT	EPWM3_TZINT	EPWM2_TZINT	EPWM1_TZINT
INT3			EPWM6_INT	EPWM5_INT	EPWM4_INT	EPWM3_INT	EPWM2_INT	EPWM1_INT
INT4			ECAP6_INT	ECAP5_INT	ECAP4_INT	ECAP3_INT	ECAP2_INT	ECAP1_INT
INT5							EQEP2_INT	EQEP1_INT
INT6			MXINTA	MRINTA	MXINTB	MRINTB	SPITXINTA	SPIRXINTA
INT7			DINTCH6	DINTCH5	DINTCH4	DINTCH3	DINTCH2	DINTCH1
INT8			SCITXINTC	SCIRXINTC			I2CINT2A	I2CINT1A
INT9	ECAN1_INTB	ECAN0_INTB	ECAN1_INTA	ECAN0_INTA	SCITXINTB	SCIRXINTB	SCITXINTA	SCIRXINTA
INT10								
INT11								
INT12	LUF	LVF		XINT7	XINT6	XINT5	XINT4	XINT3



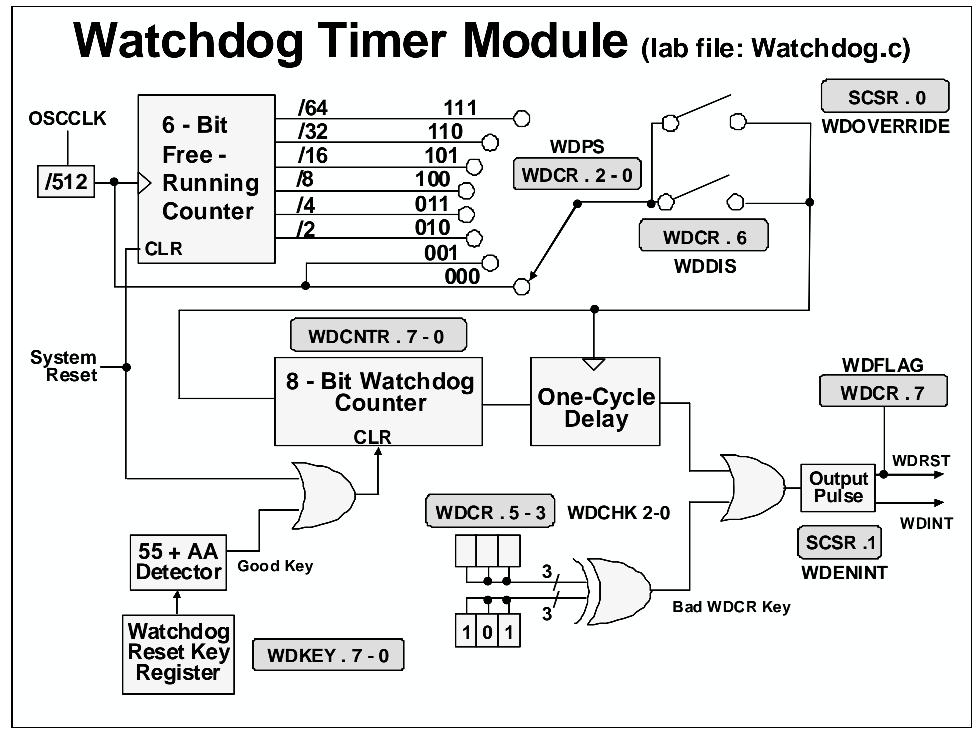
Oscillator / PLL Clock Module



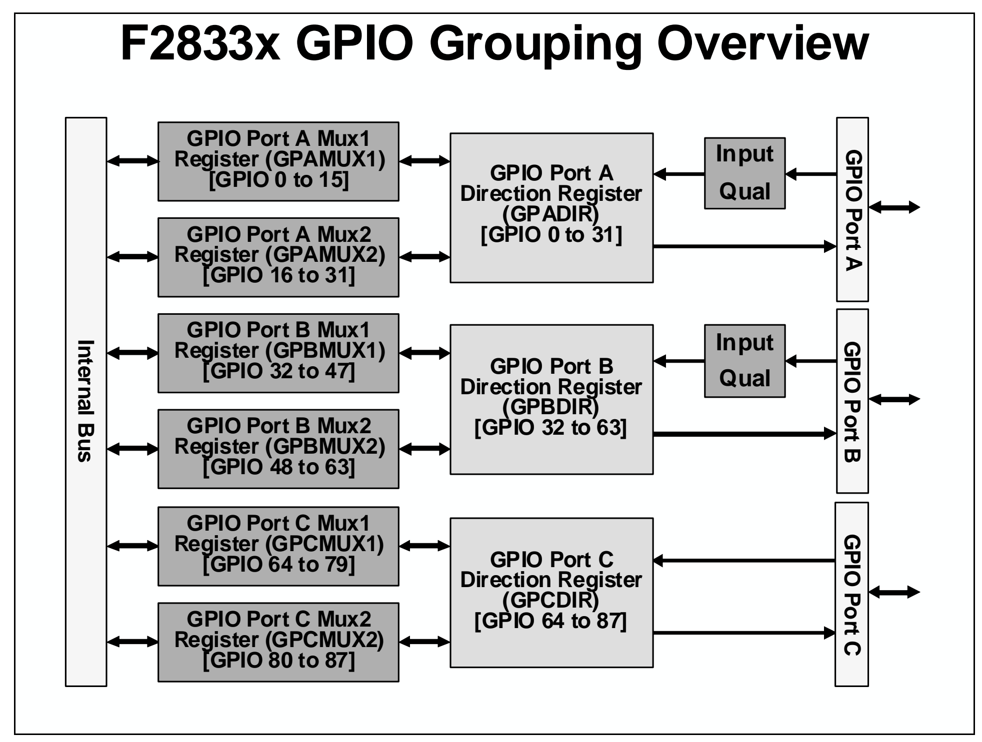
Watchdog Timer Module

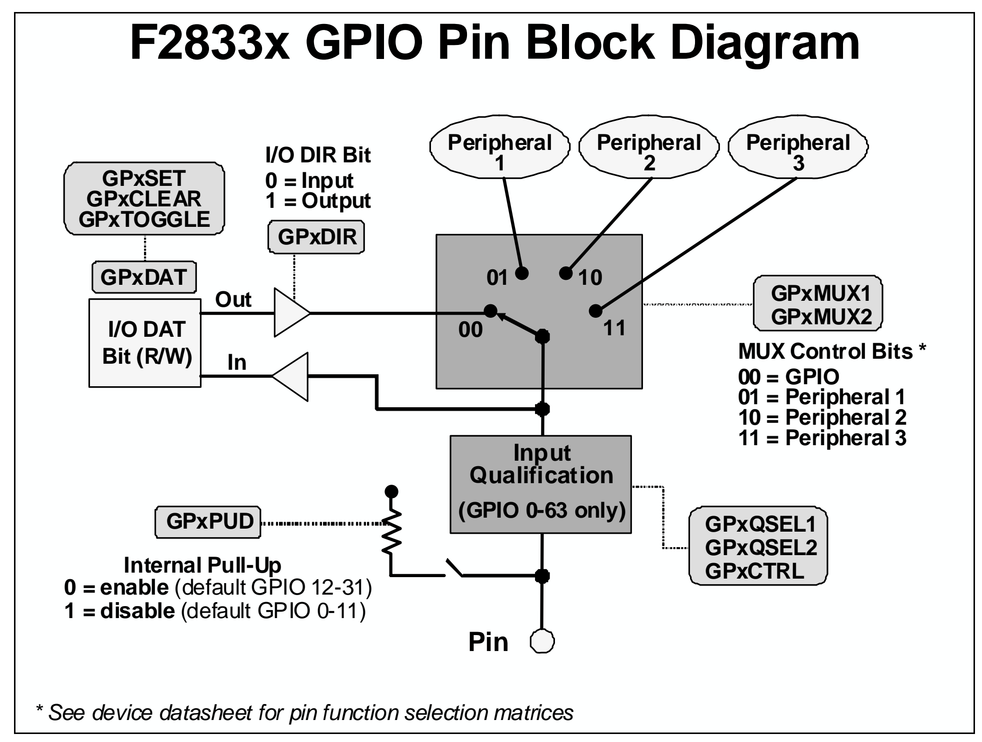
Watchdog Timer

- **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 instructions after reset**
- **This translates to 4.37 ms with a 30 MHz OSCCLK**



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, HISPCP = /1, LOSPCP = /4, low-power modes to default values, enable all module clocks
- Disable the watchdog – clear WD flag, disable watchdog, WD prescale = 1
- Setup watchdog system control register – DO NOT clear WD OVERRIDE bit, WD generate a DSP reset
- Setup 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 tested by using the watchdog to generate an interrupt. This lab will make use of the DSP2833x 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**

Project File

1. A project named `Lab2.pjt` has been created for this lab. Open the project by clicking on `Project` → `Open…` and look in `C:\C28x\LABS\LAB2`. All Build Options have been configured. The files used in this lab are:

```
CodeStartBranch.asm                 Lab_2_3.cmd
DefaultIsr_2.c                      Main_2.c
DelayUs.asm                         PieCtrl.c
DSP2833x_GlobalVariableDefs.c       PieVect.c
DSP2833x_Headers_nonBIOS.cmd        SysCtrl.c
Gpio.c                              Watchdog.c
```

Note that include files, such as `DSP2833x_Device.h` and `Lab.h`, are automatically added at project build time. (Also, `DSP2833x_DefaultIsr.h` is automatically added and will be used with the interrupts in the second part of this lab exercise).

Modified Memory Configuration

2. 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 "Jump to M0 SARAM" bootloader mode 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.

System Initialization

3. Open and inspect SysCtrl.c. Notice that the PLL and module clocks have been enabled.

4. Open and inspect Watchdog.c. Notice that watchdog control register (WDCR) is configured to disable the watchdog, and the system control and status register (SCSR) is configured to generate a reset.

5. 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

6. Click the "Build" button and watch the tools run in the build window. The output file should automatically load.

7. Under Debug on the menu bar click "Reset CPU".

8. Under Debug on the menu bar click "Go Main". You should now be at the start of Main().

Run the Code – Watchdog Reset

9. Place the cursor on the first line of code in main() and set a breakpoint by right clicking the mouse key and select Toggle Software Breakpoint. Notice that line is highlighted with a red dot indicating that the breakpoint has been set. Alternately, you can double-click in the gray field to the left of the code line to set the breakpoint. The breakpoint is set to prove that the watchdog is disabled. If the watchdog causes a reset, code execution will stop at this breakpoint.

10. Place the cursor in the "main loop" section (on the asm(" NOP"); instruction line) and right click the mouse key and select Run To Cursor. This is the same as setting a breakpoint on the selected line, running to that breakpoint, and then removing the breakpoint.

11. Run your code for a few seconds by using the <F5> key, or using the Run button on the vertical toolbar, or using Debug → Run on the menu bar. After a few seconds halt your code by using Shift <F5>, or the Halt button on the vertical toolbar. Where did your

code stop? Are the results as expected? If things went as expected, your code should be in the `"main loop"`.

12. Modify the `InitWatchdog()` function to enable the watchdog – in `Watchdog.c` change the `WDCR` register value to `0x00A8`. This will enable the watchdog to function and cause a reset. Save the file and click the `"Build"` button. Then reset the CPU by clicking on `Debug` → `Reset CPU`. Under Debug on the menu bar click `"Go Main"`.

13. Single-step your code off of the breakpoint.

14. Run your code. Where did your code stop? Are the results as expected? If things went as expected, your code should stop at the breakpoint.

Setup PIE Vector for Watchdog Interrupt

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

15. Notice that the following files are included in the project:
    `DefaultIsr_2.c`
    `PieCtrl.c`
    `PieVect.c`

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

    ```
    InitPieCtrl();
    ```

17. 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: _______

18. 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).

19. 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 this modified file.

20. 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.

21. 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

22. Click the "Build" button. Then reset the CPU, and then "Go Main".

Run the Code – Watchdog Interrupt

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

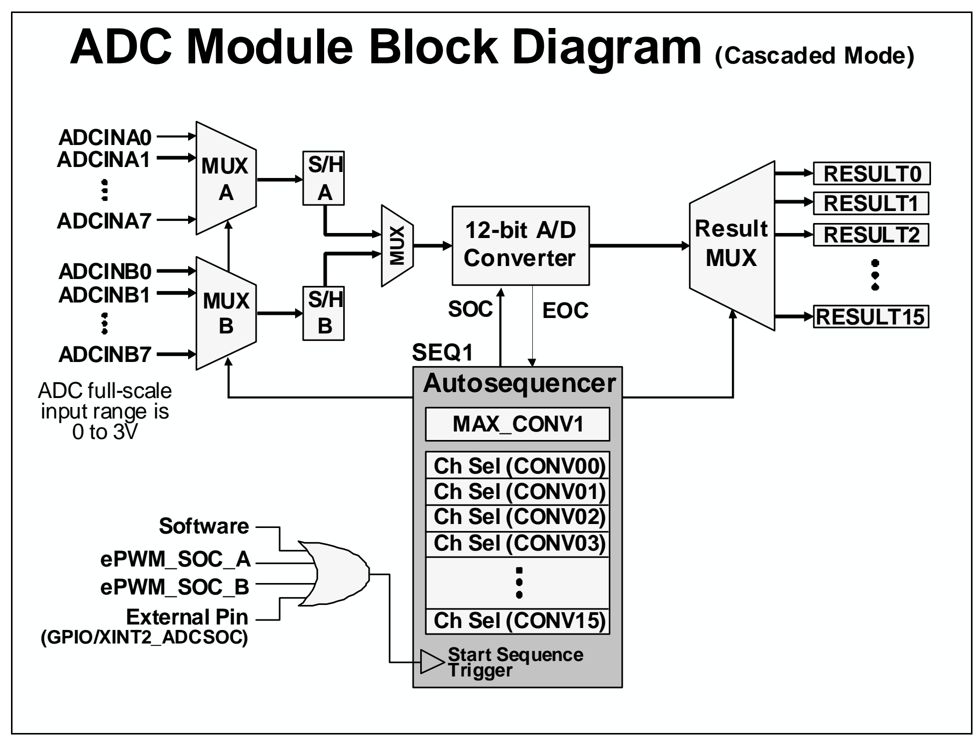
24. Run your code. Where did your code stop? Are the results as expected? If things went as expected, your code should stop at the "ESTOP0" instruction in the WAKEINT ISR.

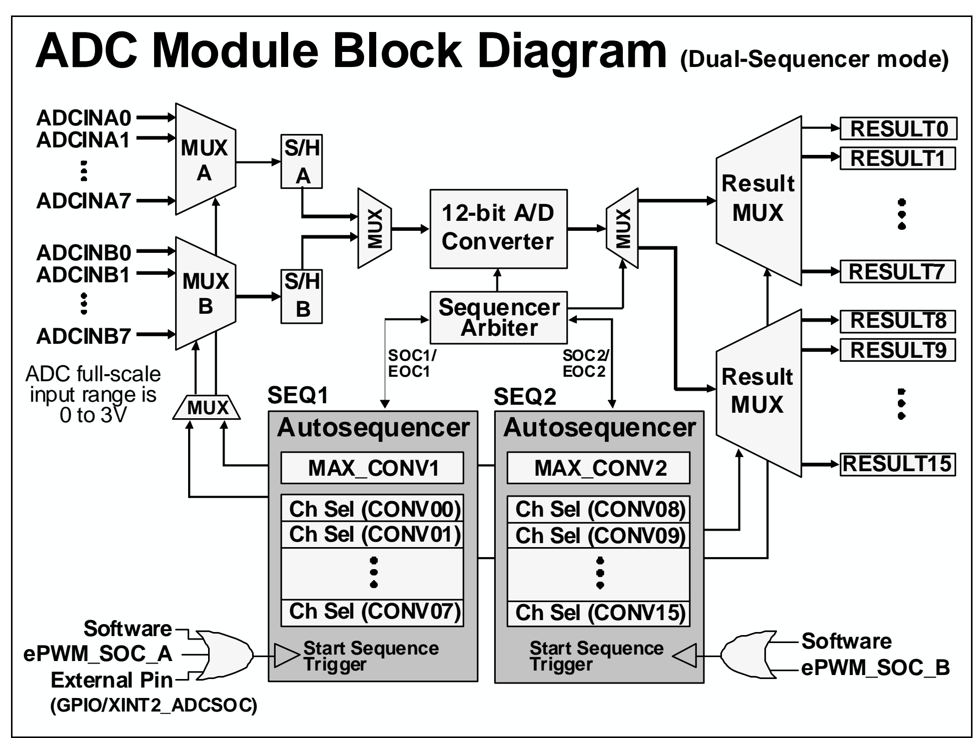
End of Exercise

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





ADC Control Registers (file: Adc.c)

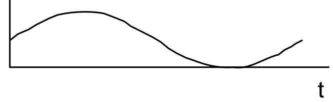
- ADCTRL1 (ADC Control Register 1)
  - module reset
  - continuous run / stop EOS
  - sequencer mode (cascaded / dual)
  - acquisition time prescale (S/H)
- ADCTRL2 (ADC Control Register 2)
  - ePWM SOC; start conversion (s/w trigger); ePWM SOC mask bit
  - reset SEQ
  - interrupt enable; interrupt mode: every EOS / every other EOS
- ADCTRL3 (ADC Control Register 3)
  - ADC clock prescale
  - sampling mode (sequential / simultaneous)
- ADCMAXCONV (ADC Maximum Conversion Register)
  - maximum number of autoconversions
- ADCCHSELSEQx {x=1-4} (ADC Channel Select Register)
  - Channel select sequencing
- ADCRESULTx {x=0-15} (ADC Results Register)

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

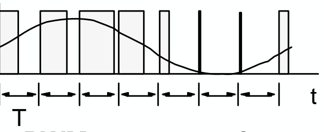
Pulse Width Modulation

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 ≈ original signal energy



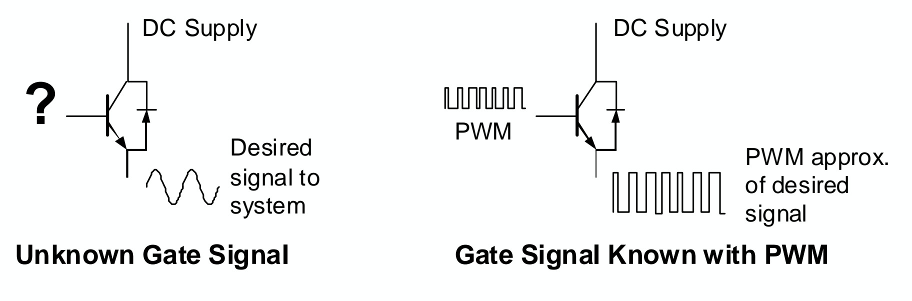
Original Signal



PWM representation

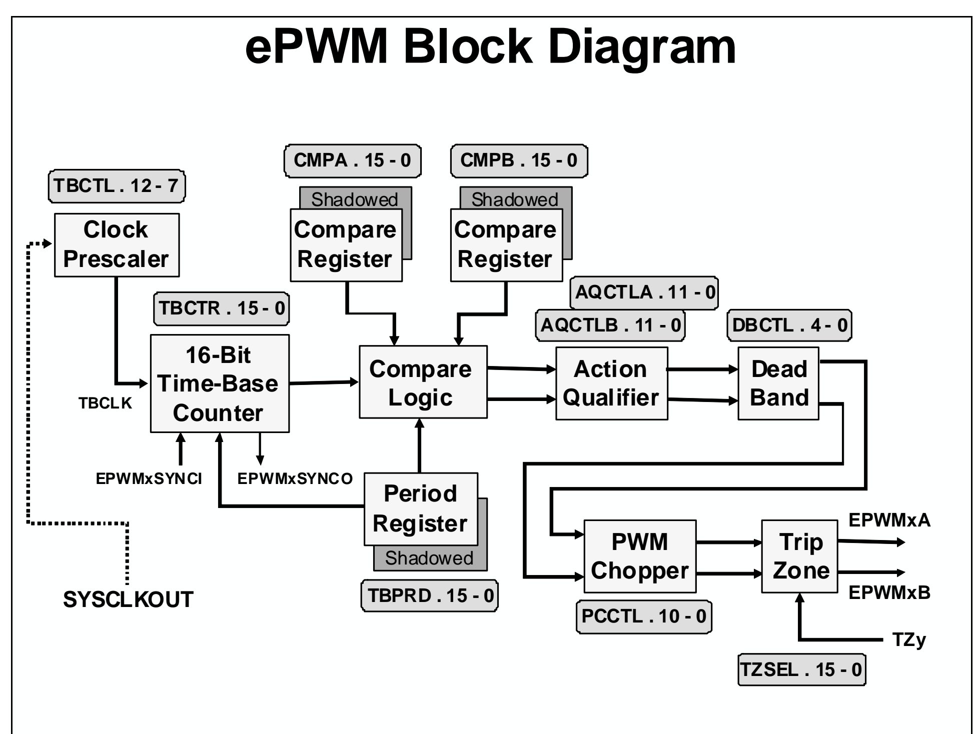
Why use PWM with Power Switching Devices?

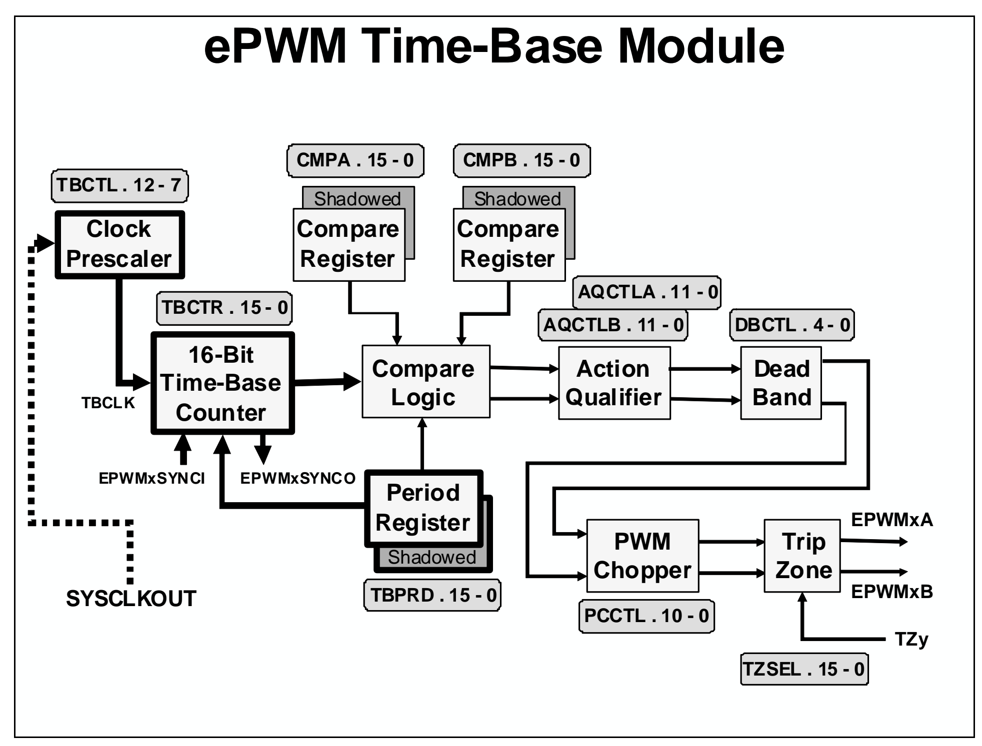
- Desired output currents or voltages are known
- Power switching devices are transistors
  - Difficult to control in proportional region
  - Easy to control in saturated region
- PWM is a digital signal ⇒ easy for DSP to output

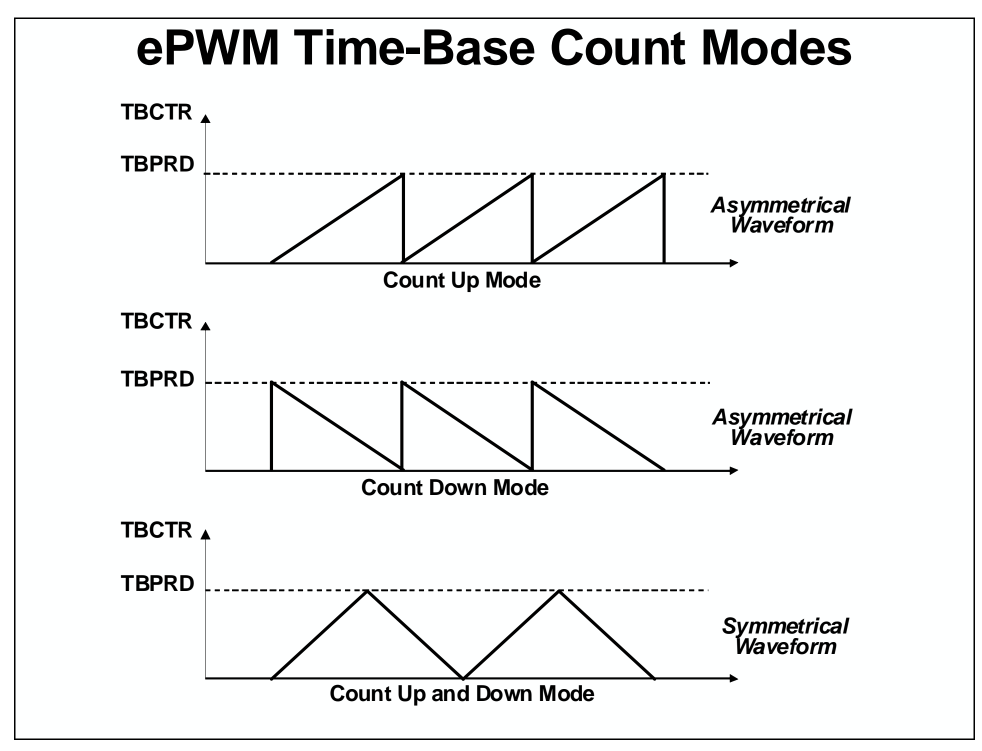


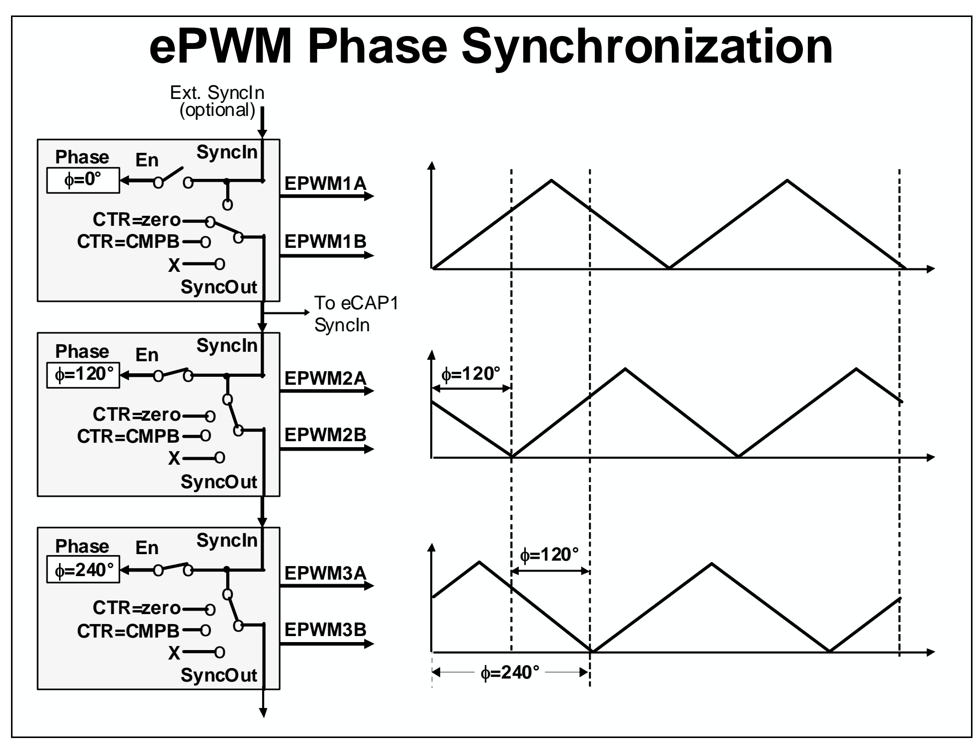
ePWM

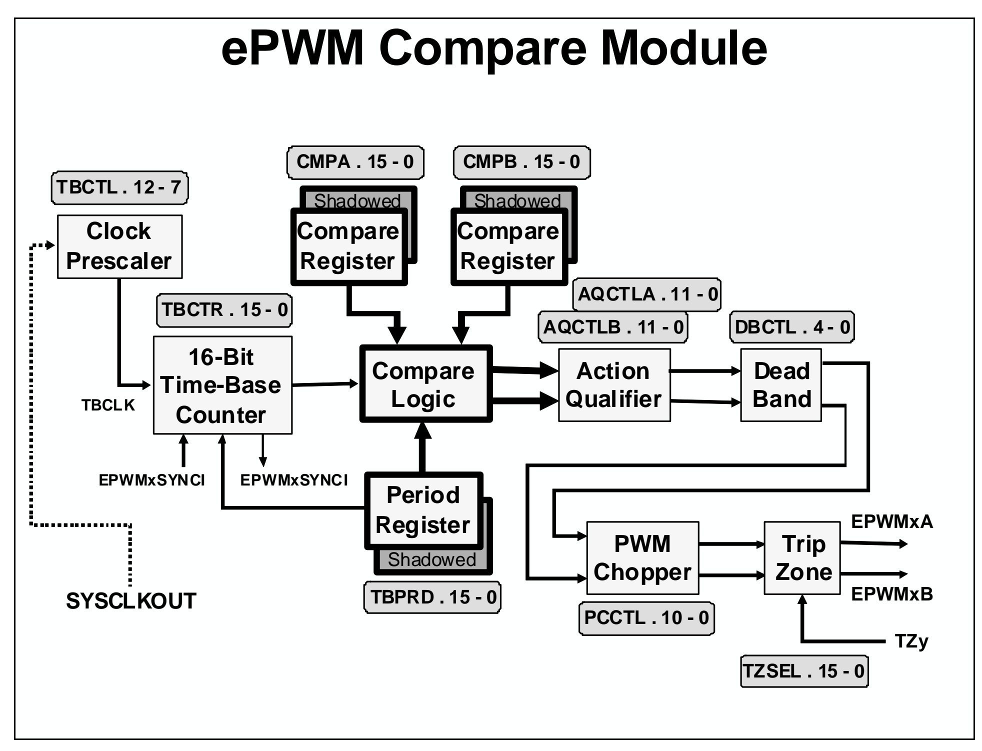
ePWM Block Diagram

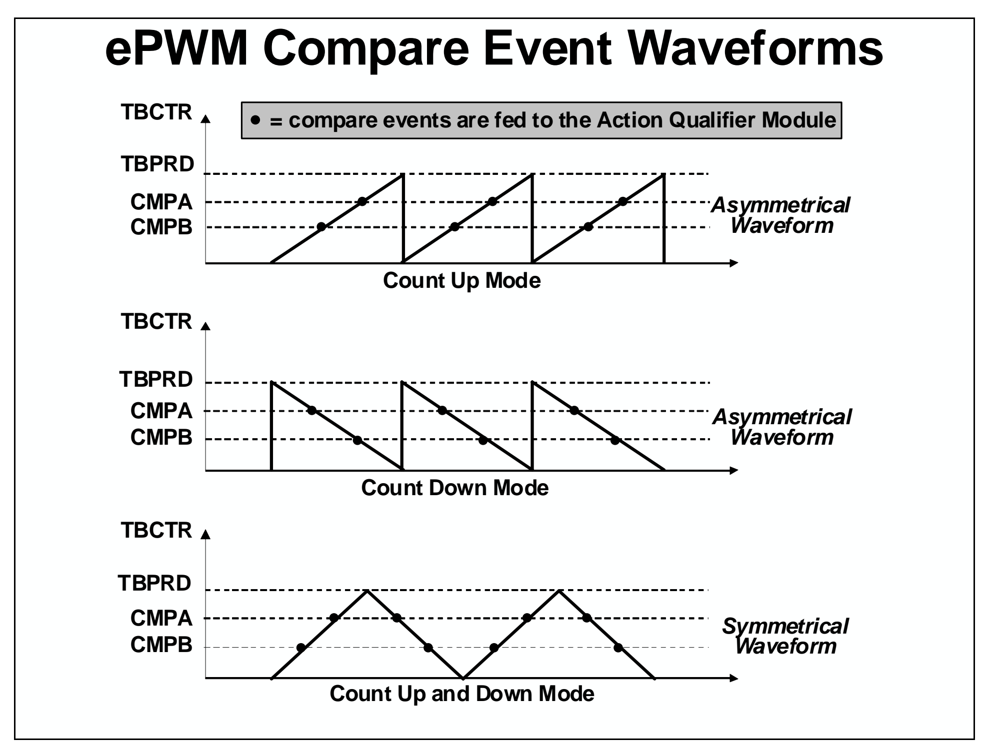


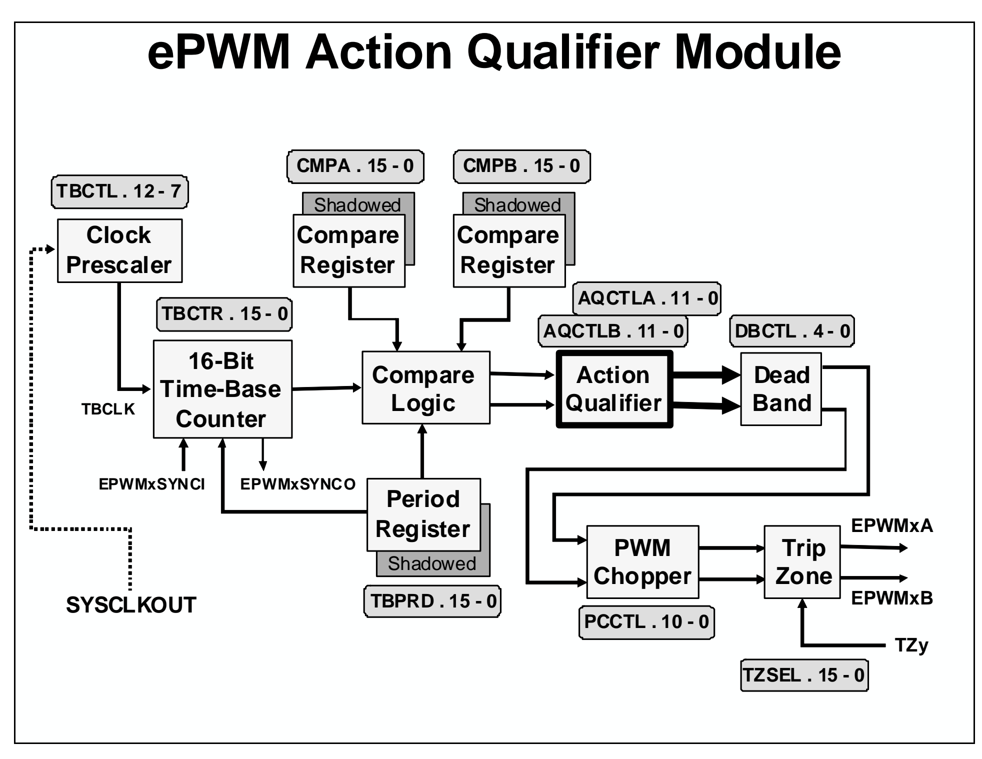

ePWM Time-Base Module
TBCTL . 12 - 7
Clock Prescaler
CMPA . 15 - 0
Shadowed
Compare Register
CMPB . 15 - 0
Shadowed
Compare Register
AQCTLA . 11 - 0
AQCTLB . 11 - 0
DBCTL . 4 - 0
TBCTR . 15 - 0
16-Bit Time-Base Counter
TBCLK
Compare Logic
Action Qualifier
Dead Band
EPWMxSYNCI
EPWMxSYNCO
Period Register
Shadowed
TBPRD . 15 - 0
SYSCLKOUT
PWM Chopper
PCCTL . 10 - 0
Trip Zone
EPWMxA
EPWMxB
TZy
TZSEL . 15 - 0


ePWM Time-Base Count Modes
TBCTR
TBPRD
Asymmetrical Waveform
Count Up Mode
TBCTR
TBPRD
Asymmetrical Waveform
Count Down Mode
TBCTR
TBPRD
Symmetrical Waveform
Count Up and Down Mode


ePWM Phase Synchronization
Ext. SyncIn (optional)
Phase
φ=0°
En
SyncIn
CTR=zero
CTR=CMPB
X
SyncOut
EPWM1A
EPWM1B
To eCAP1 SyncIn
Phase
φ=120°
En
SyncIn
CTR=zero
CTR=CMPB
X
SyncOut
EPWM2A
EPWM2B
φ=120°
Phase
φ=240°
En
SyncIn
CTR=zero
CTR=CMPB
X
SyncOut
EPWM3A
EPWM3B
φ=120°
φ=240°


ePWM Compare Module
TBCTL . 12 - 7
Clock Prescaler
CMPA . 15 - 0
Shadowed
Compare Register
CMPB . 15 - 0
Shadowed
Compare Register
TBCTR . 15 - 0
16-Bit Time-Base Counter
TBCLK
Compare Logic
AQCTLA . 11 - 0
AQCTLB . 11 - 0
Action Qualifier
DBCTL . 4 - 0
Dead Band
EPWMxSYNCI
EPWMxSYNCI
Period Register
Shadowed
TBPRD . 15 - 0
SYSCLKOUT
PWM Chopper
PCCTL . 10 - 0
Trip Zone
EPWMxA
EPWMxB
TZy
TZSEL . 15 - 0

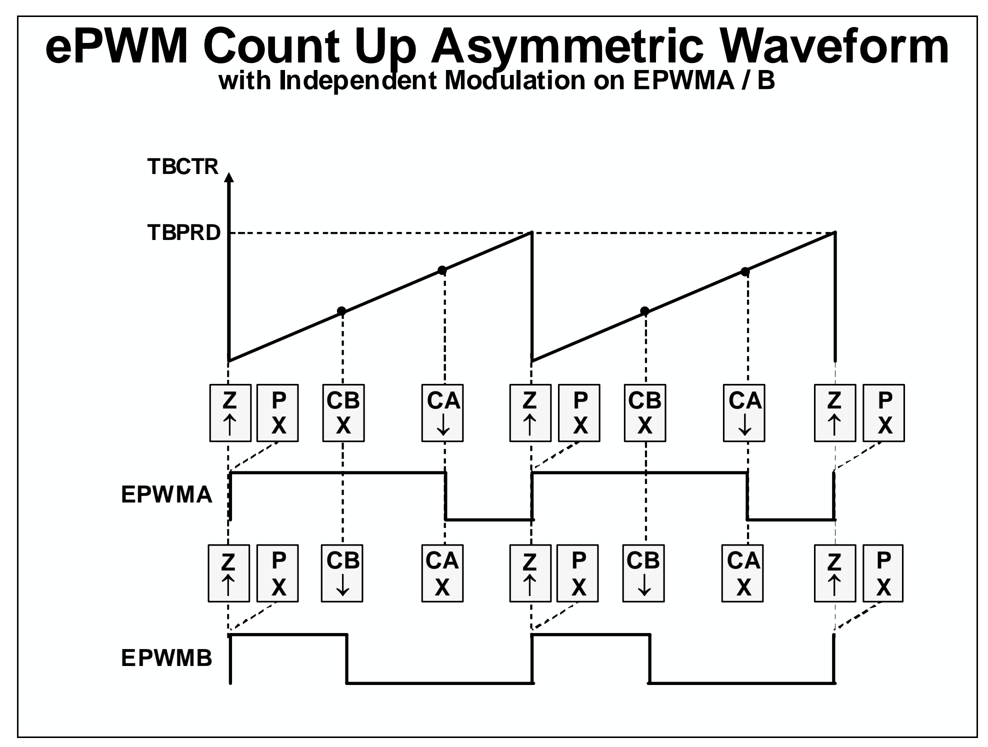

ePWM Compare Event Waveforms
• = compare events are fed to the Action Qualifier Module
TBCTR
TBPRD
CMPA
CMPB
Asymmetrical Waveform
Count Up Mode
TBCTR
TBPRD
CMPA
CMPB
Asymmetrical Waveform
Count Down Mode
TBCTR
TBPRD
CMPA
CMPB
Symmetrical Waveform
Count Up and Down Mode

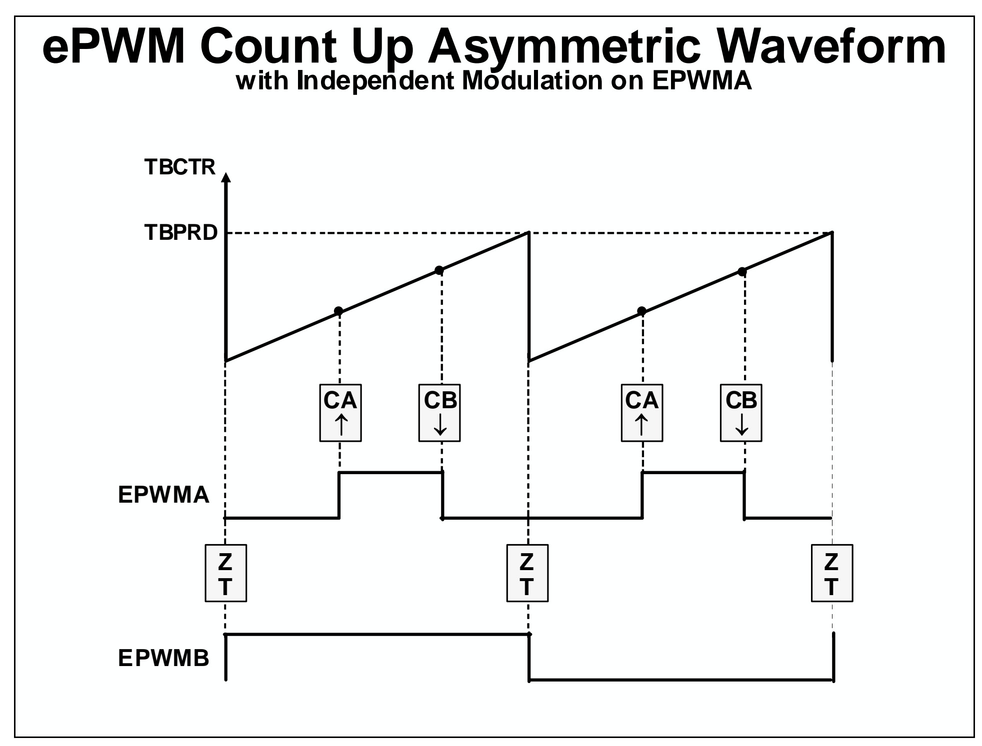

ePWM Action Qualifier Module
TBCTL . 12 - 7
Clock Prescaler
CMPA . 15 - 0
Shadowed
Compare Register
CMPB . 15 - 0
Shadowed
Compare Register
TBCTR . 15 - 0
16-Bit Time-Base Counter
TBCLK
EPWMxSYNCI
EPWMxSYNCO
Compare Logic
AQCTLA . 11 - 0
AQCTLB . 11 - 0
Action Qualifier
DBCTL . 4 - 0
Dead Band
Period Register
Shadowed
TBPRD . 15 - 0
SYSCLKOUT
PWM Chopper
PCCTL . 10 - 0
Trip Zone
EPWMxA
EPWMxB
TZy
TZSEL . 15 - 0

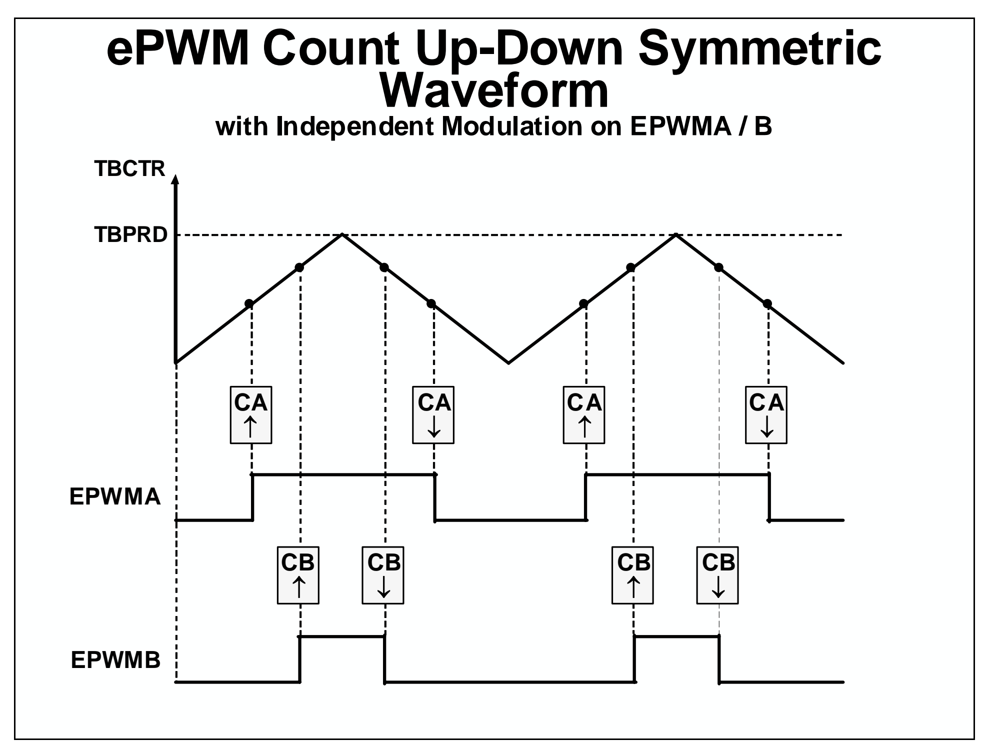
ePWM Action Qualifier Actions

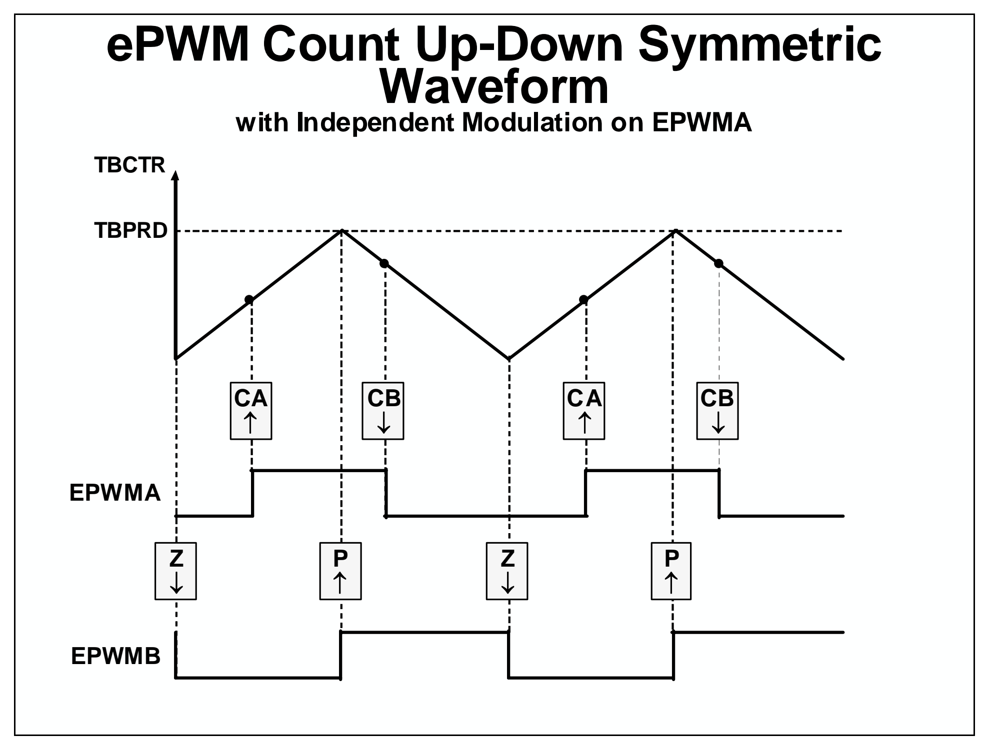
for EPWMA and EPWMB

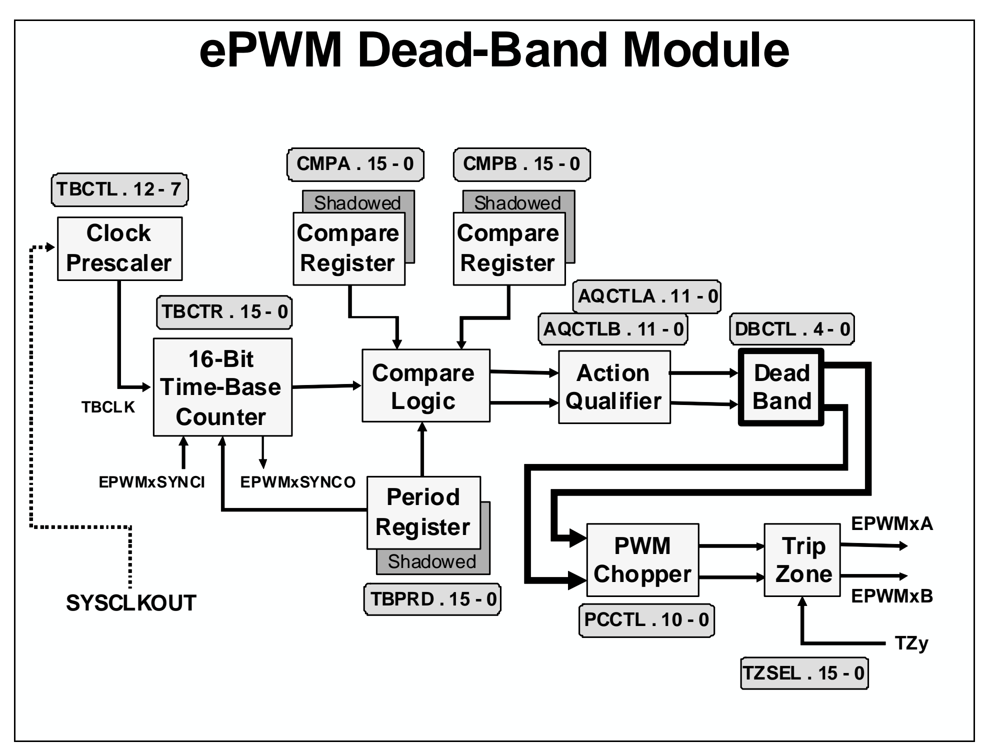
S/W Force	Time-Base Counter equals:				EPWM Output Actions
	Zero	CMPA	CMPB	TBPRD	
SW X	Z X	CA X	CB X	P X	Do Nothing
SW ↓	Z ↓	CA ↓	CB ↓	P ↓	Clear Low
SW ↑	Z ↑	CA ↑	CB ↑	P ↑	Set High
SW T	Z T	CA T	CB T	P T	Toggle



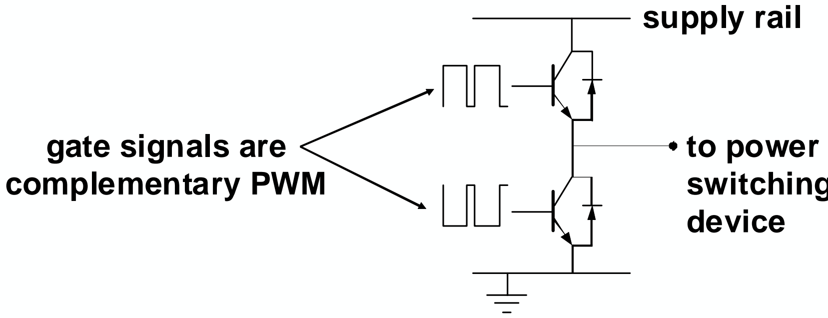

ePWM Count Up Asymmetric Waveform
with Independent Modulation on EPWMA
TBCTR
TBPRD
CA ↑
CB ↓
CA ↑
CB ↓
EPWMA
Z T
Z T
Z T
EPWMB


ePWM Count Up-Down Symmetric Waveform
with Independent Modulation on EPWMA / B
TBCTR
TBPRD
CA ↑
CA ↓
CA ↑
CA ↓
EPWMA
CB ↑
CB ↓
CB ↑
CB ↓
EPWMB


ePWM Count Up-Down Symmetric Waveform
with Independent Modulation on EPWMA
TBCTR
TBPRD
CA ↑
CB ↓
CA ↑
CB ↓
EPWMA
Z ↓
P ↑
Z ↓
P ↑
EPWMB

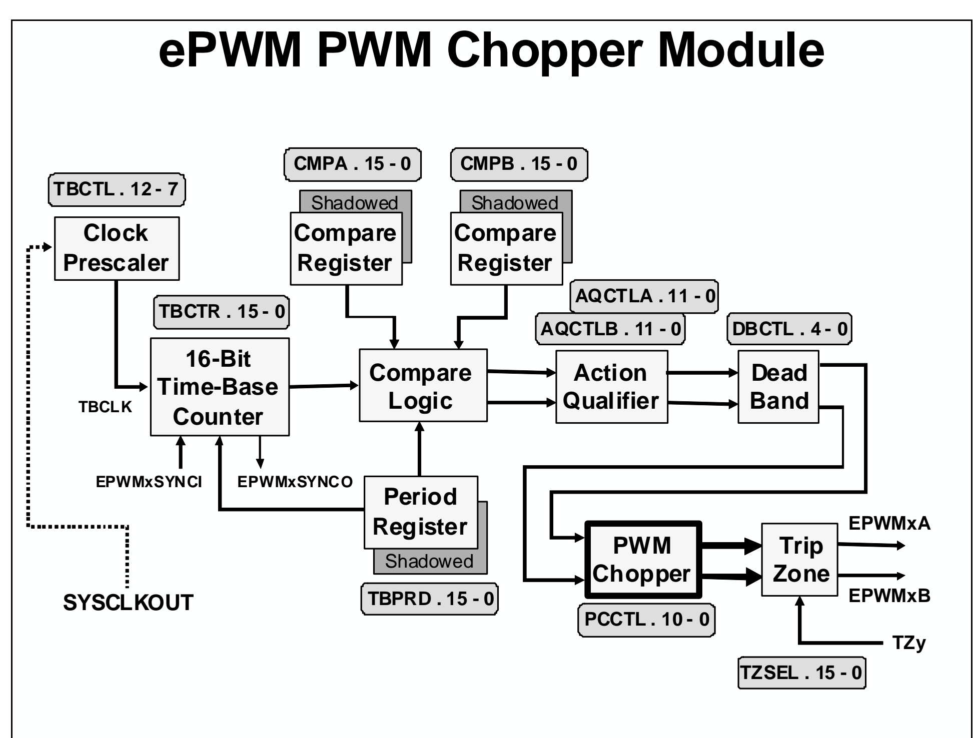

ePWM Dead-Band Module
CMPA . 15 - 0
CMPB . 15 - 0
TBCTL . 12 - 7
Shadowed
Compare Register
Shadowed
Compare Register
Clock Prescaler
TBCTR . 15 - 0
AQCTLA . 11 - 0
AQCTLB . 11 - 0
DBCTL . 4 - 0
16-Bit Time-Base Counter
Compare Logic
Action Qualifier
Dead Band
TBCLK
EPWMxSYNCI
EPWMxSYNCO
Period Register
Shadowed
TBPRD . 15 - 0
PWM Chopper
Trip Zone
EPWMxA
EPWMxB
PCCTL . 10 - 0
TZy
TZSEL . 15 - 0
SYSCLKOUT

Motivation for Dead-Band



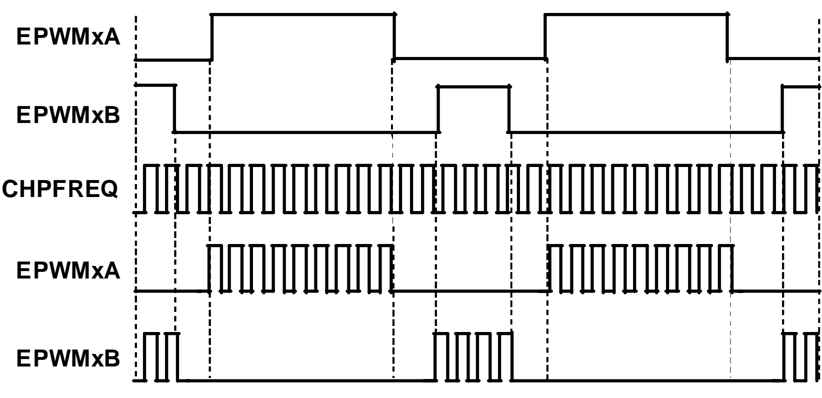
- Transistor gates turn on faster than they shut off
- Short circuit if both gates are on at same time!

ePWM PWM Chopper Module

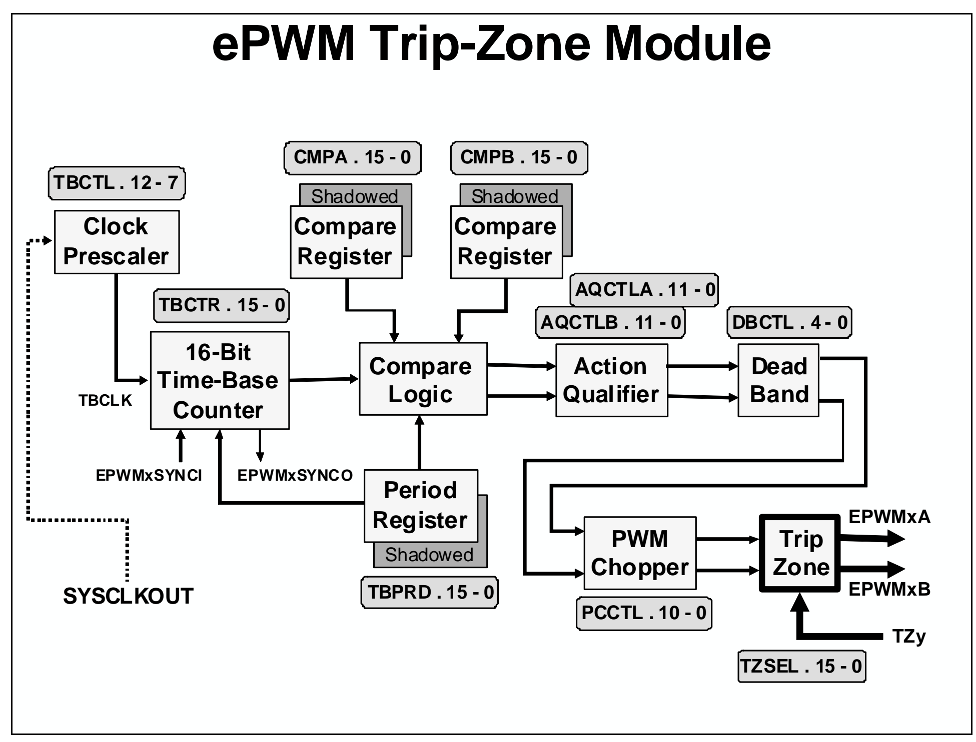


ePWM Chopper Waveform

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

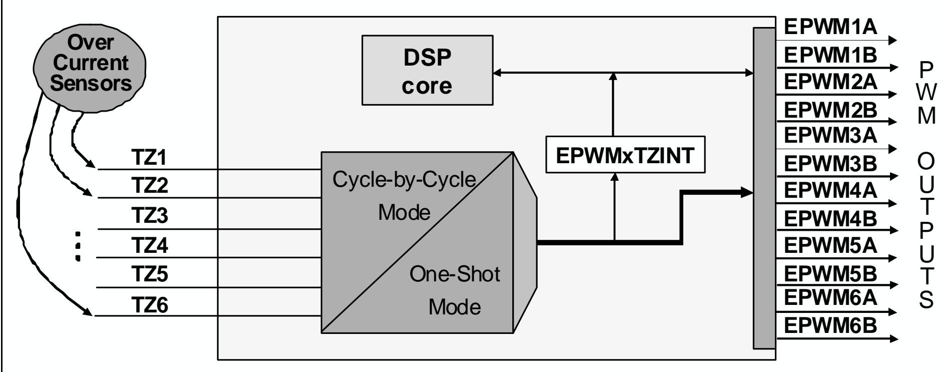


ePWM Trip-Zone Module

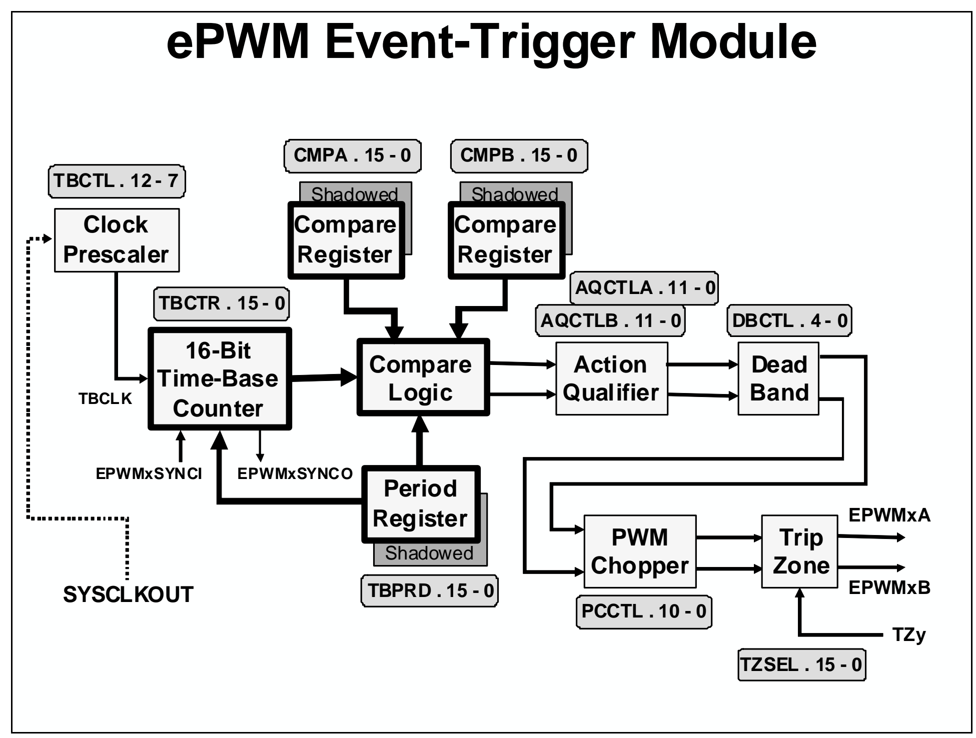


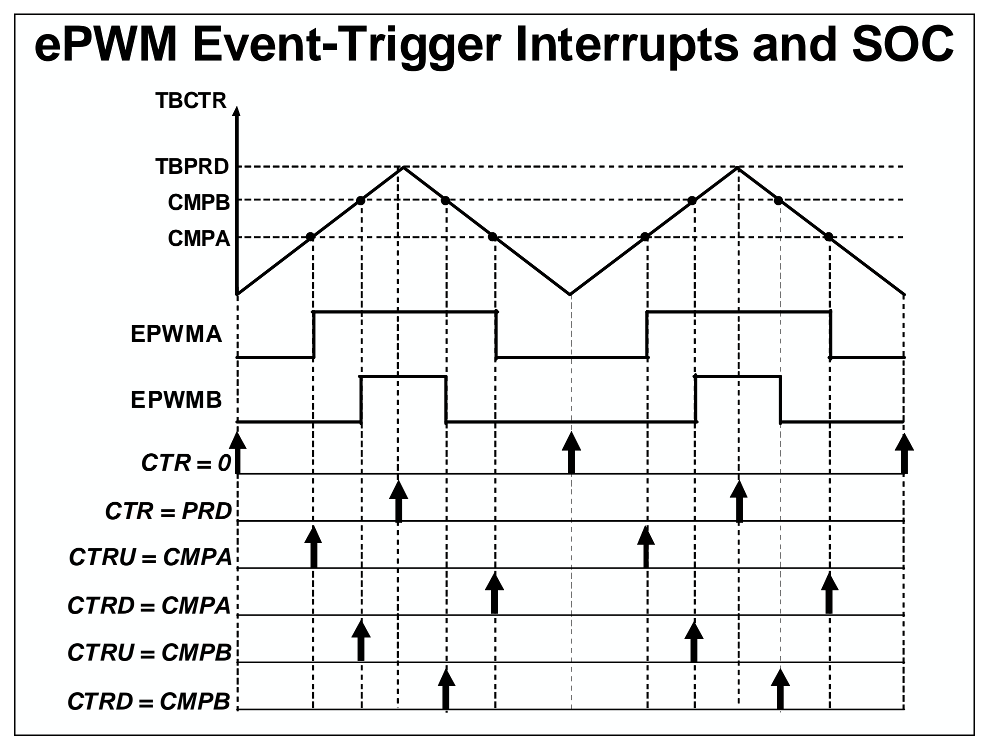
Trip-Zone Module Features

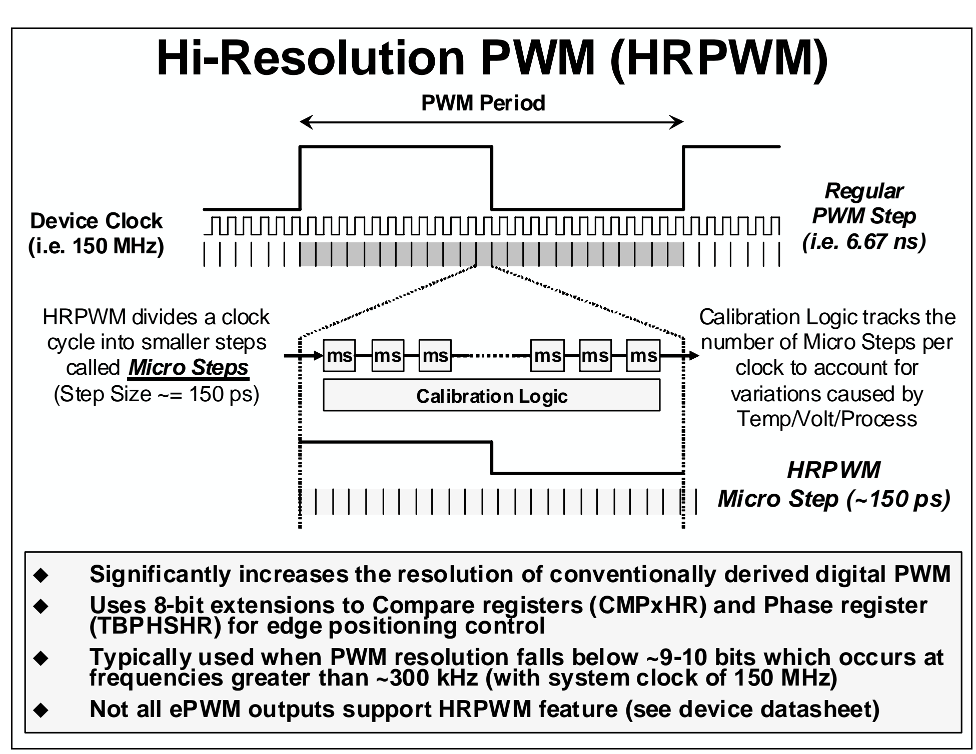
- Trip-Zone has a fast, clock independent logic path to high-impedance the EPWMxA/B output pins
- Interrupt latency may not protect hardware when responding to over current conditions or short-circuits through ISR software
- Supports: #1) one-shot trip for major short circuits or over current conditions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



ePWM Event-Trigger Module




ePWM Event-Trigger Interrupts and SOC
TBCTR
TBPRD
CMPB
CMPA
EPWMA
EPWMB
CTR = 0
CTR = PRD
CTRU = CMPA
CTRD = CMPA
CTRU = CMPB
CTRD = CMPB


Hi-Resolution PWM (HRPWM)
PWM Period
Device Clock
(i.e. 150 MHz)
Regular
PWM Step
(i.e. 6.67 ns)
HRPWM divides a clock cycle into smaller steps called Micro Steps
(Step Size ~= 150 ps)
ms
Calibration Logic
Calibration Logic tracks the number of Micro Steps per clock to account for variations caused by Temp/Volt/Process
HRPWM
Micro Step (~150 ps)
Significantly increases the resolution of conventionally derived digital PWM
Uses 8-bit extensions to Compare registers (CMPxHR) and Phase register (TBPHSHR) for edge positioning control
Typically used when PWM resolution falls below ~9-10 bits which occurs at frequencies greater than ~300 kHz (with system clock of 150 MHz)
Not all ePWM outputs support HRPWM feature (see device datasheet)

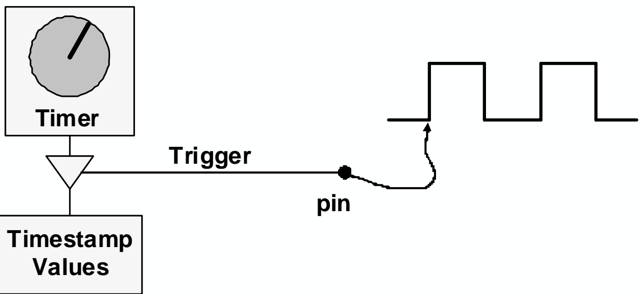
ePWM Control Registers (file: EPwm.c)

- **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 PWMxA/B); polarity select
- **PCCTL (PWM-Chopper Control)**
  - enable / disable; chopper CLK freq. & duty cycle; 1-shot pulse width
- **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

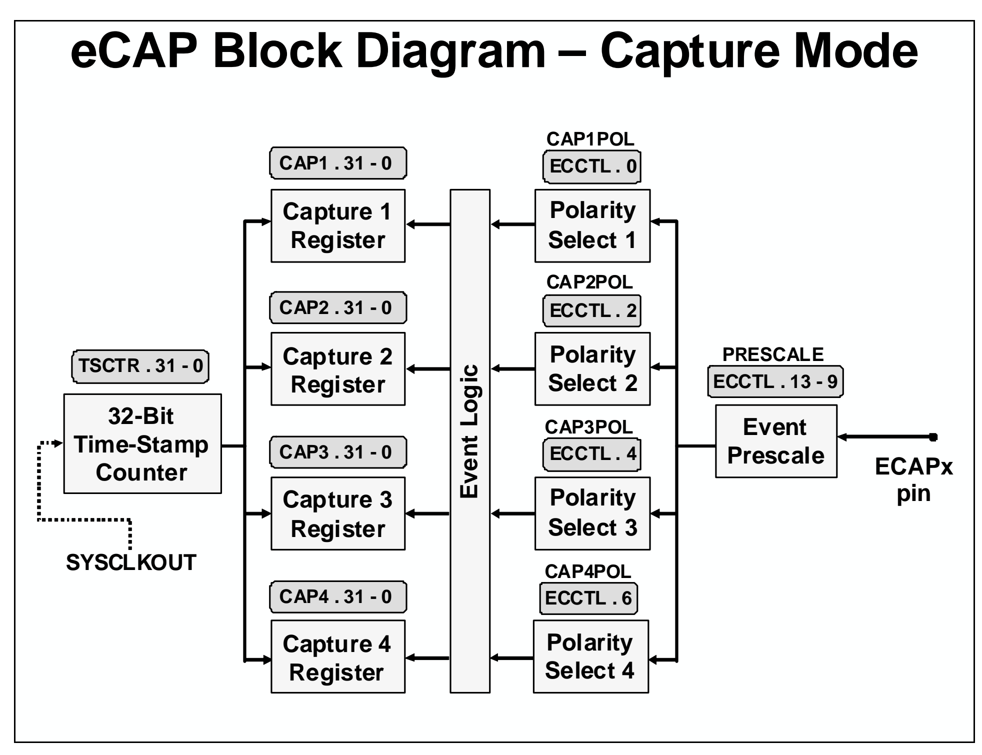
Note: refer to the reference guide for a complete listing of registers

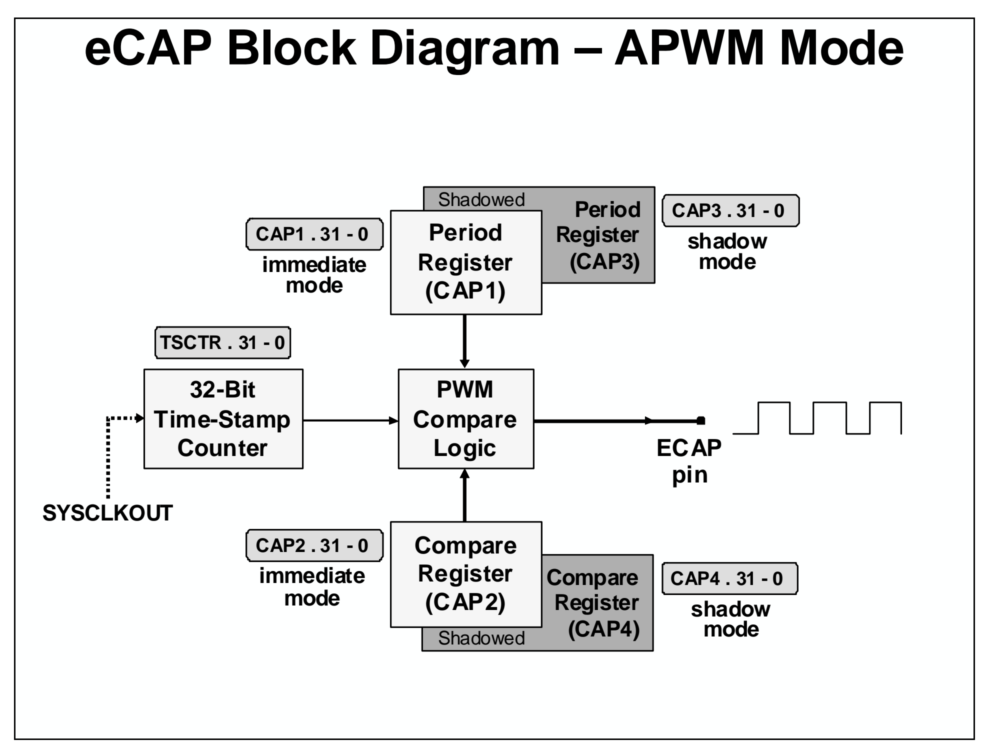
eCAP

Capture Units (eCAP)

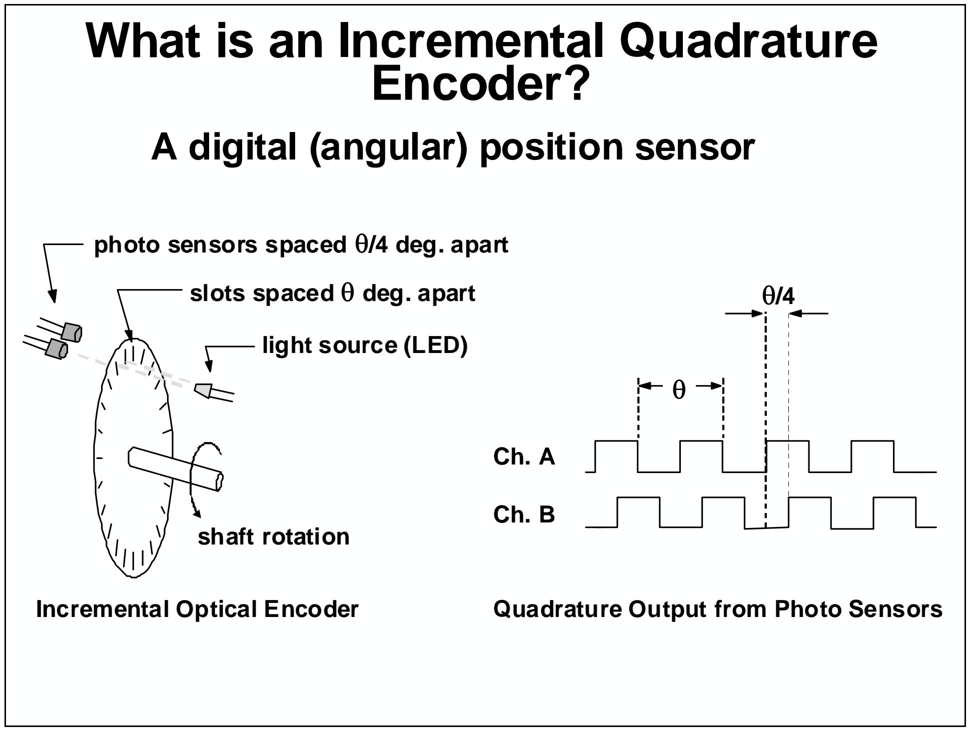


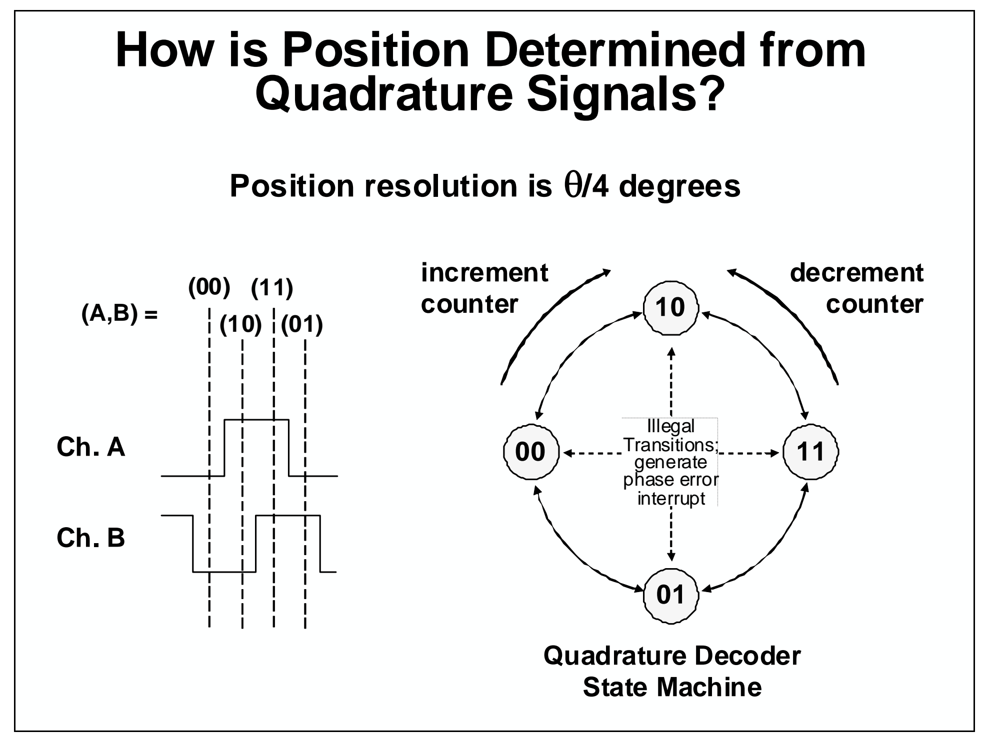
- **The eCAP module timestamps transitions on a capture input pin**

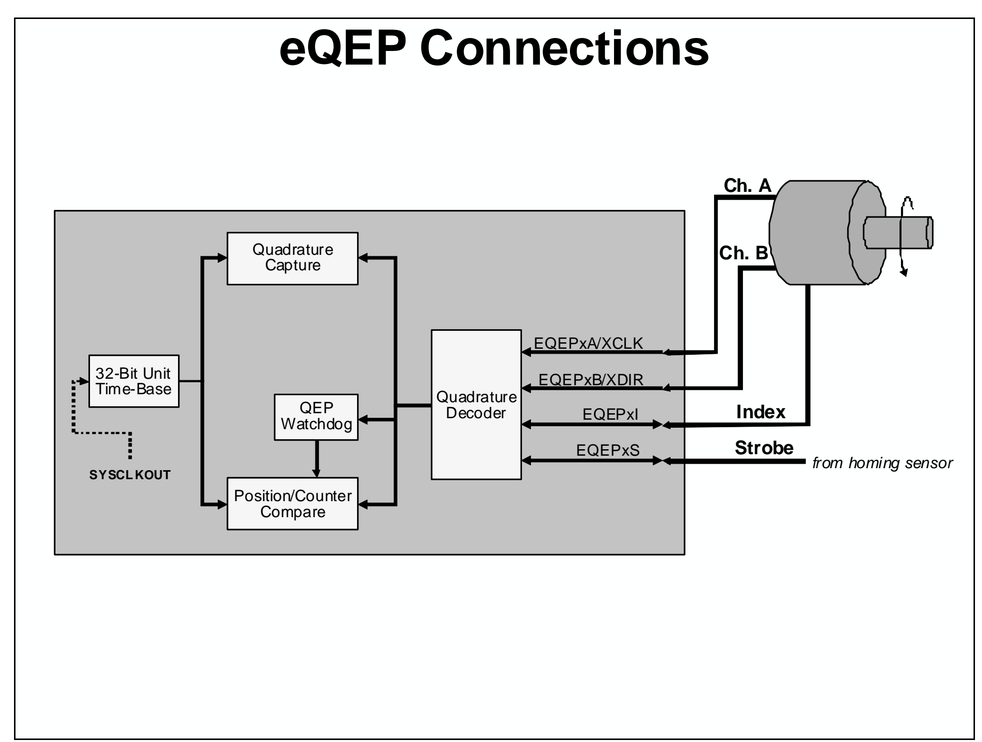

eCAP Block Diagram – Capture Mode
CAP1 . 31 - 0
Capture 1 Register
CAP2 . 31 - 0
Capture 2 Register
CAP3 . 31 - 0
Capture 3 Register
CAP4 . 31 - 0
Capture 4 Register
TSCTR . 31 - 0
32-Bit Time-Stamp Counter
SYSCLKOUT
Event Logic
CAP1POL
ECCTL . 0
Polarity Select 1
CAP2POL
ECCTL . 2
Polarity Select 2
CAP3POL
ECCTL . 4
Polarity Select 3
CAP4POL
ECCTL . 6
Polarity Select 4
PRESCALE
ECCTL . 13 - 9
Event Prescale
ECAPx pin


eCAP Block Diagram – APWM Mode
Shadowed
Period Register (CAP3)
CAP3 . 31 - 0
shadow mode
CAP1 . 31 - 0
immediate mode
Period Register (CAP1)
TSCTR . 31 - 0
32-Bit Time-Stamp Counter
SYSCLKOUT
PWM Compare Logic
ECAP pin
CAP2 . 31 - 0
immediate mode
Compare Register (CAP2)
Compare Register (CAP4)
Shadowed
CAP4 . 31 - 0
shadow mode

eQEP






eQEP Connections
Ch. A
Ch. B
Quadrature Capture
32-Bit Unit Time-Base
QEP Watchdog
Quadrature Decoder
Position/Counter Compare
SYSCLKOUT
EQEPxA/XCLK
EQEPxB/XDIR
EQEPxI
EQEPxS
Index
Strobe
from homing sensor

Lab 3: Control Peripherals

➢ Objective

The objective of this lab is to demonstrate the techniques discussed in this module 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 48 kHz sampling rate and store the conversion result in a buffer in the DSP memory. This buffer operates in a circular fashion, such that new conversion data continuously overwrites older results in the buffer.

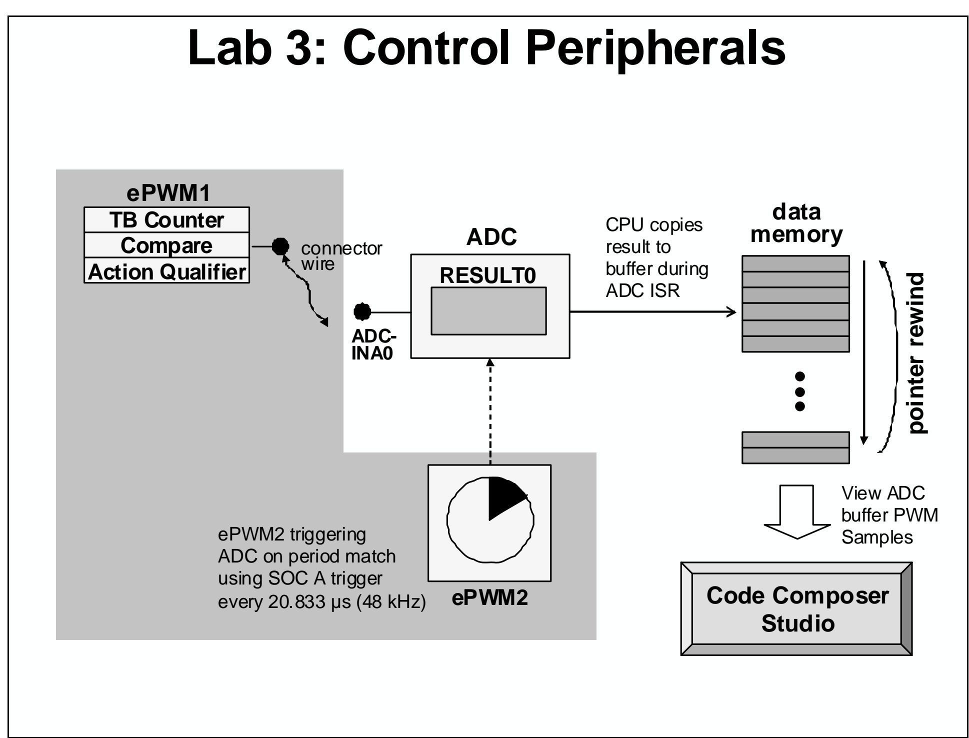
Two ePWM modules have been configured for this lab exercise:

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 SOC A)



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 DS2 on the eZdsp™ 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 48 kHz sampling rate
- ePWM2 is triggering the ADC on period match using SOC A 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 eZdsp™ LED DS2 as a visual indication that it is running

➢ **Procedure**

Project File

1. A project named `Lab3.pjt` has been created for this lab. Open the project by clicking on `Project` → `Open…` and look in `C:\C28x\LABS\LAB3`. All Build Options have been configured. The files used in this lab are:

```
Adc.c                                  Gpio.c
CodeStartBranch.asm                    Lab_2_3.cmd
DefaultIsr_3_4.c                       Main_3.c
DelayUs.asm                            PieCtrl.c
DSP2833x_GlobalVariableDefs.c          PieVect.c
DSP2833x_Headers_nonBIOS.cmd           SysCtrl.c
EPwm.c                                 Watchdog.c
```

Setup of Shared I/O, General-Purpose Timer1 and Compare1

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

---

2. 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

3. Click the "Build" button and watch the tools run in the build window. The output file should automatically load.

4. Under Debug on the menu bar click "Reset CPU".

5. Under Debug on the menu bar click "Go Main". You should now be at the start of Main().

Run the Code – PWM Waveform

6. Open a memory window to view some of the contents of the ADC results buffer. To open a memory window click: View → Memory… on the menu bar. The address label for the ADC results buffer is *AdcBuf*.

Note: ***Exercise care when connecting any wires, as the power to the eZdsp™ is on, and we do not want to damage the eZdsp™!*** Details of pin assignments can be found in Appendix A.

7. Using a connector wire provided, connect the PWM1A (pin # P8-9) to ADCINA0 (pin # P9-2) on the eZdsp™.

8. Run your code for a few seconds by using the <F5> key, or using the Run button on the vertical toolbar, or using Debug → Run on the menu bar. After a few seconds halt your code by using Shift <F5>, or the Halt button on the vertical toolbar. Verify that the ADC result buffer contains the updated values.

9. Open and setup a graph to plot a 48-point window of the ADC results buffer.
Click: View → Graph → Time/Frequency… and set the following values:

Start Address	AdcBuf
Acquisition Buffer Size	48
Display Data Size	48
DSP Data Type	16-bit unsigned integer
Sampling Rate (Hz)	48000
Time Display Unit	μs

Select OK to save the graph options.

10. The graphical display should show the generated 2 kHz, 25% duty cycle symmetric PWM waveform. The period of a 2 kHz signal is 500 μs. You can confirm this by measuring the period of the waveform using the graph (you may want to enlarge the graph window using the mouse). The measurement is best done with the mouse. The lower left-hand corner of the graph window will display the X and Y-axis values.

Subtract the X-axis values taken over a complete waveform period (you can use the PC calculator program found in Microsoft Windows to do this).

Frequency Domain Graphing Feature of Code Composer Studio

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

    Click: `View` → `Graph` → `Time/Frequency…` and set the following values:

Display Type	FFT Magnitude
Start Address	AdcBuf
Acquisition Buffer Size	48
FFT Framesize	48
DSP Data Type	16-bit unsigned integer
Sampling Rate (Hz)	48000

    Select `OK` to save the graph options.

12. On the plot window, left-click the mouse to move the vertical marker line and 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 DSP 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 DSP behavior. This is very useful when tuning control law parameters on-the-fly, for example.

13. Reset the CPU, and then enable real-time mode by selecting:

    `Debug` → `Real-time Mode`

14. A message box *may* appear. Select `YES` to enable debug events. This will set bit 1 (DBGM bit) of status register 1 (ST1) to a "0". The DBGM is the debug enable mask bit. When the DBGM bit is set to "0", memory and register values can be passed to the host processor for updating the debugger windows.

15. The memory and graph windows displaying *AdcBuf* should still be open. The connector wire between PWM1A (pin # P8-9) and ADCINA0 (pin # P9-2) should still be connected. In real-time mode, we would like to have our window continuously refresh. Click:

    `View → Real-time Refresh Options…`

    and check "`Global Continuous Refresh`". Use the default refresh rate of 100 ms and select `OK`. Alternately, we could have right clicked on each window individually and selected "`Continuous Refresh`".

    Note: "`Global Continuous Refresh`" causes all open windows to refresh at the refresh rate. This can be problematic when a large number of windows are open, as bandwidth over the emulation link is limited. Updating too many windows can cause the refresh frequency to bog down. In that case, either close some windows, or disable global refresh and selectively enable "`Continuous Refresh`" for individual windows of interest instead.

16. Run the code and watch the windows update in real-time mode. ***Carefully*** remove and replace the connector wire from PWM1A (pin # P8-9). Are the values updating as expected?

17. Fully halting the DSP when in real-time mode is a two-step process. First, halt the processor with `Debug → Halt`. Then uncheck the "`Real-time mode`" to take the DSP out of real-time mode (`Debug → Real-time Mode`).

Real-time Mode using GEL Functions

18. Code Composer Studio includes GEL (General Extension Language) functions which automate entering and exiting real-time mode. Four functions are available:

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

    These GEL functions can be executed by clicking:

    `GEL → Realtime Emulation Control →` *GEL Function*

    If you would like, try repeating the previous step using the following GEL functions:

    `GEL → Realtime Emulation Control → Run_Realtime_with_Reset`

    `GEL → Realtime Emulation Control → Full_Halt`

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, the student may want to try this after the class).

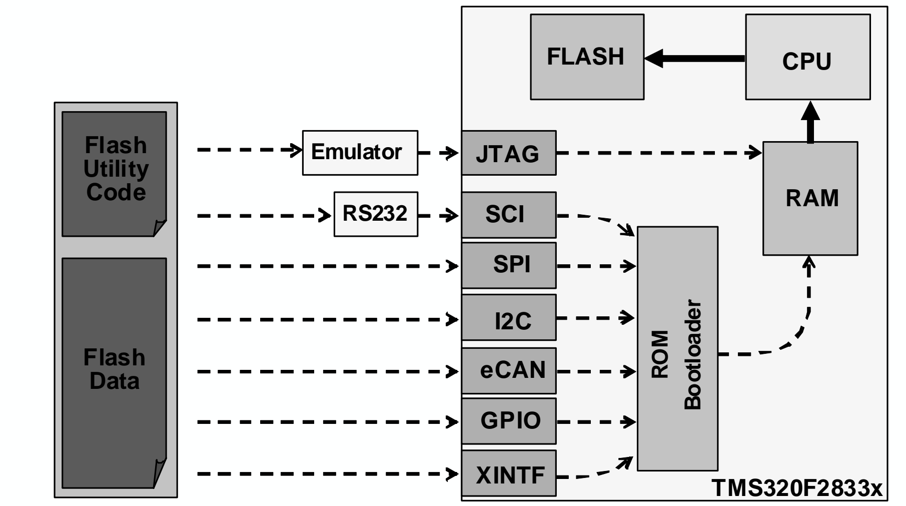
End of Exercise

Flash Programming

Flash Programming Basics

Flash Programming Basics

- The DSP CPU itself performs the flash programming
- The CPU executes Flash utility code from RAM that reads the Flash data and writes it into the Flash
- We need to get the Flash utility code and the Flash data into RAM



Flash Programming Basics

- Sequence of steps for Flash programming:

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

- Minimum Erase size is a sector (32Kw 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 Plug-in

Flash Programming Utilities

- **Code Composer Studio Plug-in (uses JTAG)**
- **Third-party JTAG utilities**
  - **SDFlash JTAG from Spectrum Digital (requires SD emulator)**
  - **Signum System Flash utilities (requires Signum emulator)**
  - **BlackHawk Flash utilities (requires Blackhawk emulator)**
- **SDFlash Serial utility (uses SCI boot)**
- **Gang Programmers (use GPIO boot)**
  - **BP Micro programmer**
  - **Data I/O programmer**
- **Build your own custom utility**
  - **Use a different ROM bootloader method than SCI**
  - **Embed flash programming into your application**
  - **Flash API algorithms provided by TI**

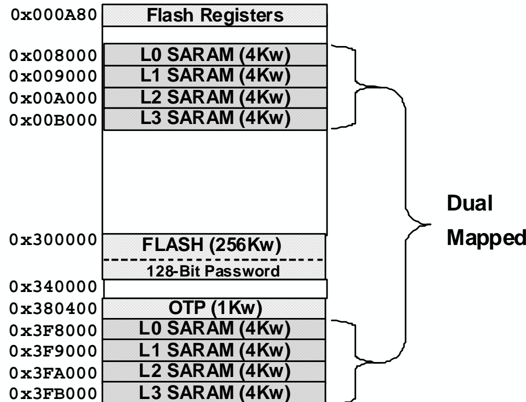
*** TI web has links to all utilities (http://www.ti.com/c2000)**

Code Composer Studio Flash Plug-In

Code Security Module and Password

Code Security Module (CSM)

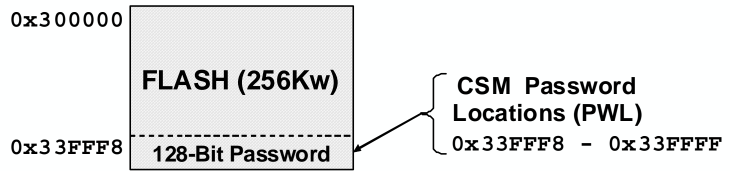
- Access to the following on-chip memory is restricted:



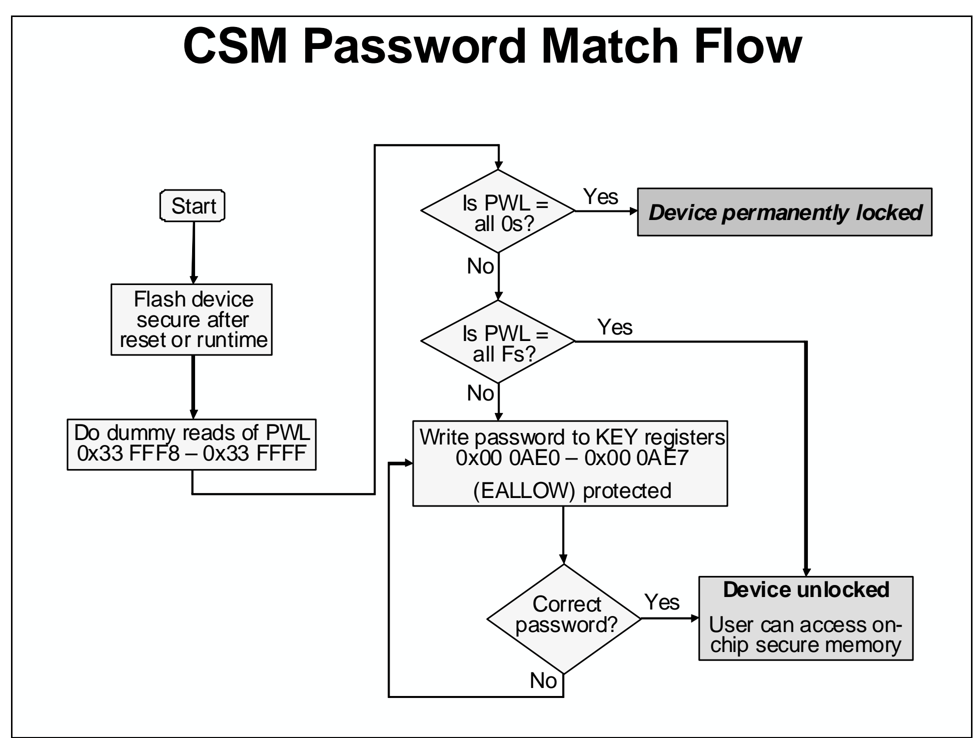
- Data reads and writes from restricted memory are only allowed for code running from restricted memory
- All other data read/write accesses are blocked:
  JTAG emulator/debugger, ROM bootloader, code running in external memory or unrestricted internal memory

CSM Password

- Prevents reverse engineering and protects valuable intellectual property



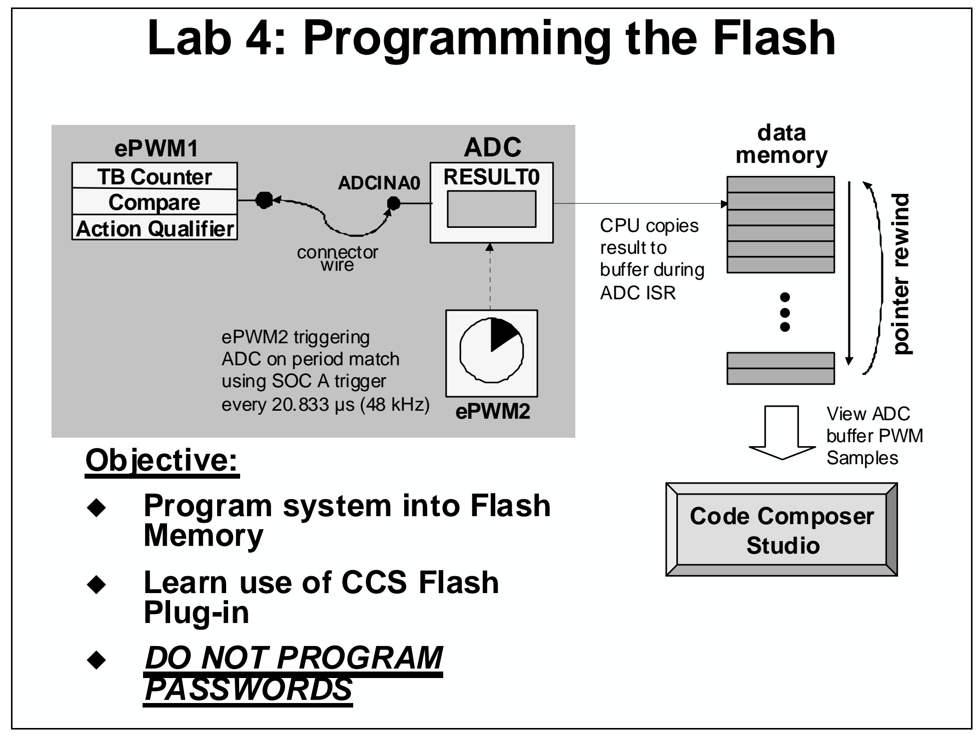
- 128-bit user defined password is stored in Flash
- 128-bit Key Register used to lock and unlock the device
  - Mapped in memory space 0x00 0AE0 – 0x00 0AE7
- 128-bits = 2^{128} = 3.4×10^{38} possible passwords
- To try 1 password every 8 cycles at 150 MHz, it would take at least 5.8×10^{23} years to try all possible combinations!


CSM Password Match Flow
Start
Flash device secure after reset or runtime
Do dummy reads of PWL 0x33 FFF8 – 0x33 FFFF
Is PWL = all 0s?
Yes
Device permanently locked
No
Is PWL = all Fs?
Yes
No
Write password to KEY registers 0x00 0AE0 – 0x00 0AE7
(EALLOW) protected
Correct password?
Yes
No
Device unlocked
User can access on-chip secure memory

Lab 4: Programming the Flash

➢ **Objective**

The objective of this lab is to demonstrate the techniques discussed in this module and to program and execute code from the on-chip flash memory. The TMS320F28335 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**

Project File

1. A project named `Lab4.pjt` has been created for this lab. Open the project by clicking on `Project` → `Open…` and look in `C:\C28x\Labs\Lab4`. All Build Options have been configured like the previous lab. The files used in this lab are:

```
Adc.c                                Gpio.c
CodeStartBranch.asm                  Lab_4.cmd
DefaultIsr_3_4.c                     Main_4.c
DelayUs.asm                          PieCtrl.c
DSP2833x_GlobalVariableDefs.c        PieVect.c
DSP2833x_Headers_nonBIOS.cmd         SysCtrl.c
EPwm.c                               Watchdog.c
```

Link Initialized Sections to Flash

Initialized sections, such as code and constants, must contain valid values at device power-up. For a stand-alone embedded system with the F28335 device, these initialized sections must be linked to the on-chip flash memory. Note that a stand-alone embedded system must operate without an emulator or debugger in use, and no host processor is used to perform bootloading.

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

2. Open and inspect the linker command file `Lab_4.cmd`. Notice that a memory block named `FLASH_ABCDEFGH` has been been created at origin = 0x300000, length = 0x03FF80 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.

3. In `Lab_4.cmd` the following compiler sections have been linked to on-chip flash memory block `FLASH_ABCDEFGH`:

Compiler Sections
.text
.cinit
.const
.econst
.pinit
.switch

Copying Interrupt Vectors from Flash to RAM

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

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

Initializing the Flash Control Registers

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

5. Open and inspect `Flash.c`. The C compiler CODE_SECTION pragma is used to place the `InitFlash()` function into a linkable section named `"secureRamFuncs"`.

6. 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 `L0123SARAM` (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 DSP hardware or development tools perspective (since the C28x DSP 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 `L0123SARAM` memory we are linking `"secureRamFuncs"` to, we are specifiying "PAGE = 0" (which is program memory).

7. 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.

8. 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 and L3 RAM blocks. The CSM uses a 128-bit password made up of 8 individual 16-bit words. They are located in flash at addresses 0x33FFF8 to 0x33FFFF. 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 0x33FF80 through 0x33FFF5 in order to properly secure the CSM. Both tasks will be accomplished using a simple assembly language file `Passwords.asm`.

9. 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 0x33FF80 to 0x33FFF5 (length of 0x76).

10. 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 F28335 device contains a ROM bootloader that will transfer code execution to the flash after reset. When the boot mode selection pins are set for "Jump to Flash" mode, the bootloader will branch to the instruction located at address 0x33FFF6 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 0x33FFF8. 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`.

11. 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`.

12. 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.

13. The eZdsp™ board needs to be configured for "Jump to Flash" bootmode. Move switch SW1 positions 1, 2, 3 and 4 to the "1" position (all switches to the Left) to accomplish this. Details of switch positions can be found in Appendix A. This switch controls the pullup/down resistor on the GPIO84, GPIO85, GPIO86 and GPIO87 pins, which are the pins sampled by the bootloader to determine the bootmode. (For additional information on configuring the "Jump to Flash" bootmode see the TMS320x2833x DSP Boot ROM Reference Guide, and also the eZdsp F28335 Technical Reference).

Build – Lab.out

14. At this point we need to build the project, but not have CCS automatically load it since CCS cannot load code into the flash! (the flash must be programmed). On the menu bar click: `Option` → `Customize…` and select the `"Program/Project CIO"` tab. *Uncheck* `"Load Program After Build"`.

    CCS has a feature that automatically steps over functions without debug information. This can be useful for accelerating the debug process provided that you are not interested

in debugging the function that is being stepped-over. While single-stepping in this lab exercise we do not want to step-over any functions. Therefore, select the `"Debug Properties"` tab. *Uncheck* `"Step over functions without debug information when source stepping"`, then click `OK`.

15. Click the `"Build"` button to generate the `Lab.out` file to be used with the CCS Flash Plug-in.

CCS Flash Plug-in

16. Open the Flash Plug-in tool by clicking :

    `Tools → F28xx On-Chip Flash Programmer`

17. A Clock Configuration window *may* open. If needed, in the Clock Configuration window set “OSCCLK (MHz):” to 30, “DIVSEL:” to /2, and “PLLCR Value:” to 10. Then click `OK`. In the next Flash Programmer Settings window confirm that the selected DSP device to program is F28335 and all options have been checked. Click `OK`.

18. Notice that the eZdsp™ board uses a 30 MHz oscillator (located on the board near LED DS1). Confirm the “Clock Configuration” in the upper left corner has the OSCCLK set to 30 MHz, the DIVSEL set to /2, and the PLLCR value set to 10. Recall that the PLL is divided by two, which gives a SYSCLKOUT of 150 MHz.

19. Confirm that all boxes are checked in the “Erase Sector Selection” area of the plug-in window. We want to erase all the flash sectors.

20. We will not be using the plug-in to program the “Code Security Password”. ***Do not modify the Code Security Password fields.***

21. In the “Operation” block, notice that the “COFF file to Program/Verify” field automatically defaults to the current `.out` file. Check to be sure that “Erase, Program, Verify” is selected. We will be using the default wait states, as shown on the slide in this module.

22. Click “Execute Operation” to program the flash memory. Watch the programming status update in the plug-in window.

23. After successfully programming the flash memory, close the programmer window.

Running the Code – Using CCS

24. In order to effectively debug with CCS, we need to load the symbolic debug information (e.g., symbol and label addresses, source file links, etc.) so that CCS knows where everything is in your code. Click:

    `File → Load Symbols → Load Symbols Only…`

    and select `Lab4.out` in the `Debug` folder.

25. Reset the DSP. The program counter should now be at 0x3FF9A9, which is the start of the bootloader in the Boot ROM.

26. Single-Step <F11> 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.

27. Step a few more times until you reach the start of the C-compiler initialization routine at the symbol _c_int00.

28. Now do `Debug` → `Go Main`. The code should stop at the beginning of your `main()` routine. If you got to that point succesfully, it confirms that the flash has been programmed properly, and that the bootloader is properly configured for jump to flash mode, and that the codestart section has been linked to the proper address.

29. You can now RUN the DSP, and you should observe the LED on the board blinking. Try resetting the DSP and hitting RUN (without doing all the stepping and the Go Main procedure). The LED should be blinking again.

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

30. Close Code Composer Studio.

31. Disconnect the USB cable (emulator) from the eZdsp™ board.

32. Remove the power from the board.

33. Re-connect the power to the board.

34. The LED should be blinking, showing that the code is now running from flash memory.

Return Switch SW1 Back to Default Positions

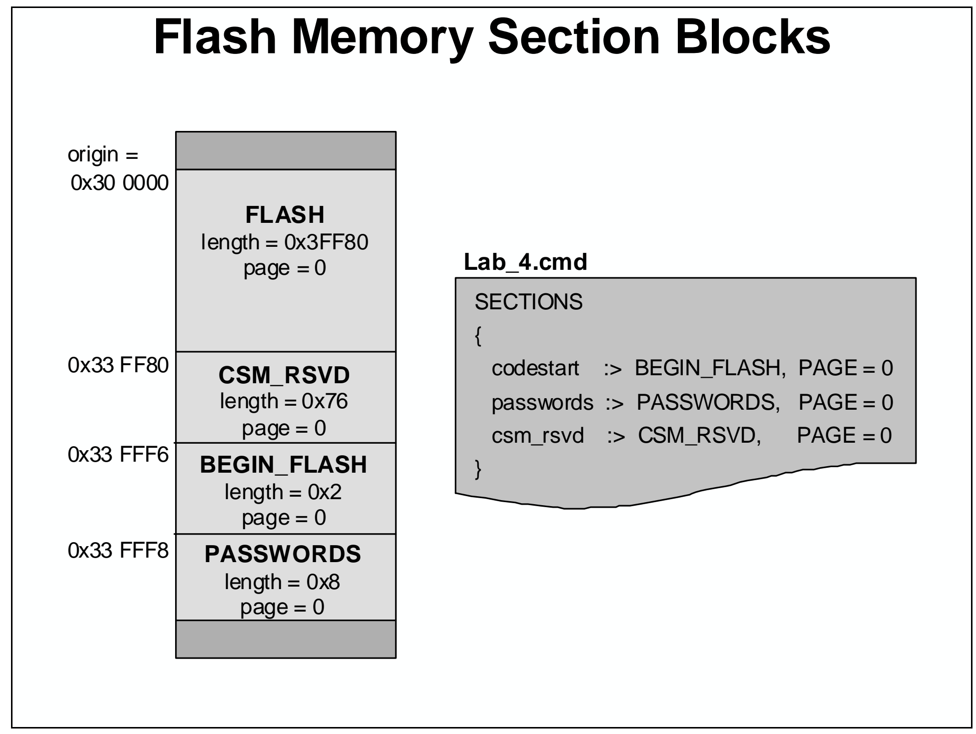
35. Remove the power from the board.

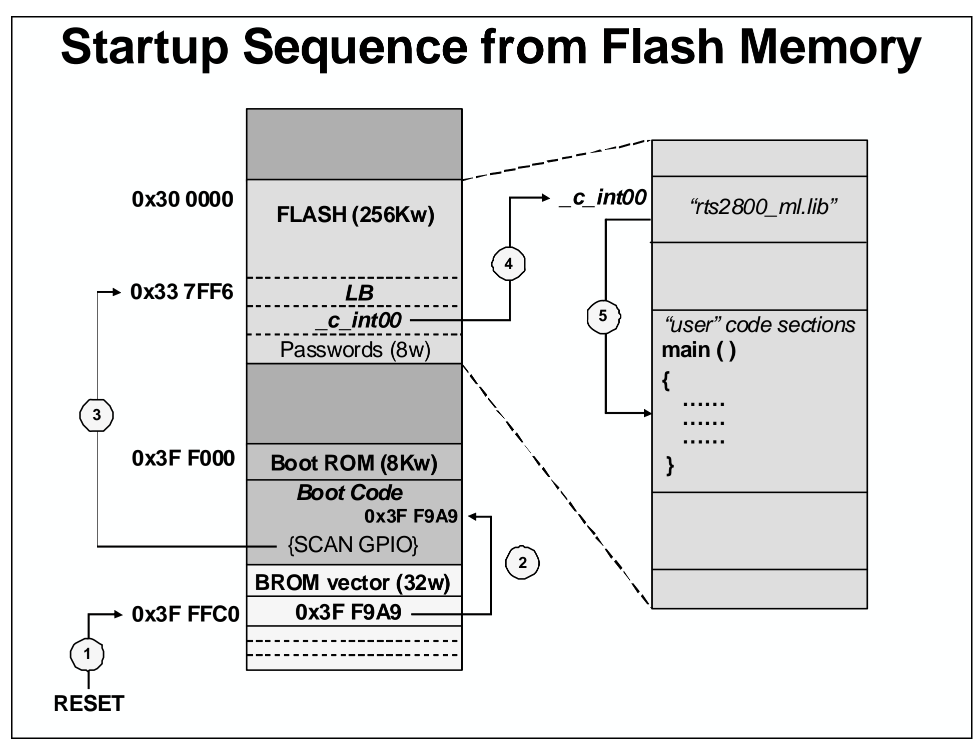
36. Please return the settings of switch SW1 back to the default positions "Jump to M0SARAM" bootmode as shown in the table below (see Appendix A for switch position details):

Position 4 GPIO87	Position 3 GPIO86	Position 2 GPIO85	Position 1 GPIO84	Boot Mode
Right – 0	Left – 1	Right – 0	Right – 0	M0 SARAM

End of Exercise

Lab 4 Reference: Programming the Flash





The Next Step…

Training

C28x Multi-day Training Course

In-depth hands-on TMS320F28335 Design and Peripheral Training

TMS320C28x 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)
- System Design
- Communications
- DSP/BIOS
- Support Resources

C2000 Digital Power Supply Workshop

Digital Power Experimenter Kit

C2000 DPS Workshop Outline
- Introduction to Digital Power Supply Design
- Driving the Power Stage with PWM Waveforms
- Controlling the Power Stage with Feedback
- Tuning the Loop for Good Transient Response
- Summary and Conclusion

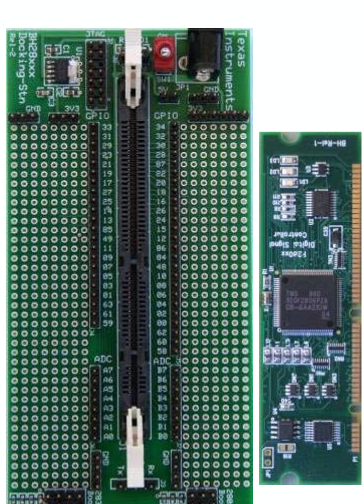
Provides hands-on Introduction to Digital Power Concepts

Development Tools

C2000 controlCARDs

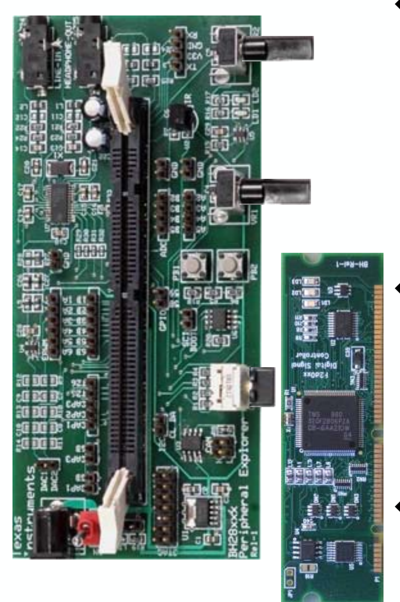
- New low cost single-board controllers perfect for initial software development and small volume system builds
- Small form factor (9cm x 2.5cm) with standard 100-pin DIMM interface
  - analog I/O, digital I/O, and JTAG signals available at DIMM interface
- Galvanically isolated RS-232 interface
- Single 5V power supply required (not included)
- Available through TI authorized distributors and on the TI web
  - Part Numbers:
    - TMDSCNCD2808 (100 MHz F2808)
    - TMDSCNCD28044 (100 MHz F28044)
    - TMDSCNCD28335 (150 MHz F28335)

C2000 Experimenter Kits



- Experimenter Kits include
  - F2808 or F28335 controlCARD
  - Docking station (motherboard)
  - C2000 Applications Software CD with example code and full hardware details
  - Code Composer Studio v3.3 with code size limit of 32KB
  - 5V DC power supply
- Docking station features
  - Access to all controlCARD signals
  - Breadboard areas
  - RS-232 an JTAG connectors
- Available through TI authorized distributors and on the TI web
  - Part Numbers:
    - TMDSDOCK2808
    - TMDSDOCK28335

C2000 Peripheral Explorer Kit

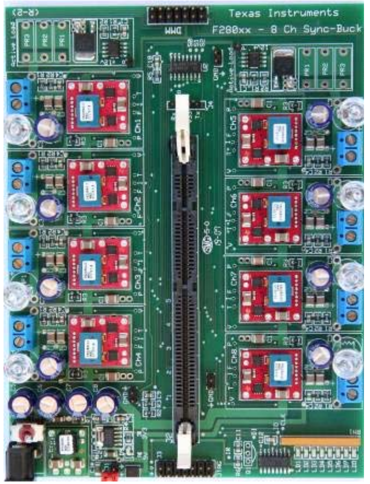


- Experimenter Kit includes
  - F28335 controlCARD
  - Peripheral Explorer (motherboard)
  - C2000 Applications Software CD with example code and full hardware details
  - Code Composer Studio v3.3 with code size limit of 32KB
  - 5V DC power supply
- Peripheral Explorer features
  - ADC input variable resistors
  - GPIO hex encoder & push buttons
  - eCAP infrared sensor
  - GPIO LEDs, I2C & CAN connection
  - Analog I/O (AIC+McBSP)
- Available through TI authorized distributors and on the TI web
  - Part Number:
    - TMDSPREX28335

C2000 Digital Power Experimenter Kit

- DPEK includes
  - 2-rail DC/DC EVM using TI PowerTrain™ modules (10A)
  - F2808 controlCARD
  - On-board digital multi-meter and active load for transient response tuning
  - C2000 Applications Software CD with example code and full hardware details
  - Digital Power Supply Workshop teaching material and lab software
  - Code Composer Studio v3.3 with code size limit of 32KB
  - 9V DC power supply
- Available through TI authorized distributors and on the TI web
  - Part Number: TMDSDCDC2KIT

C2000 DC/DC Developer’s Kit

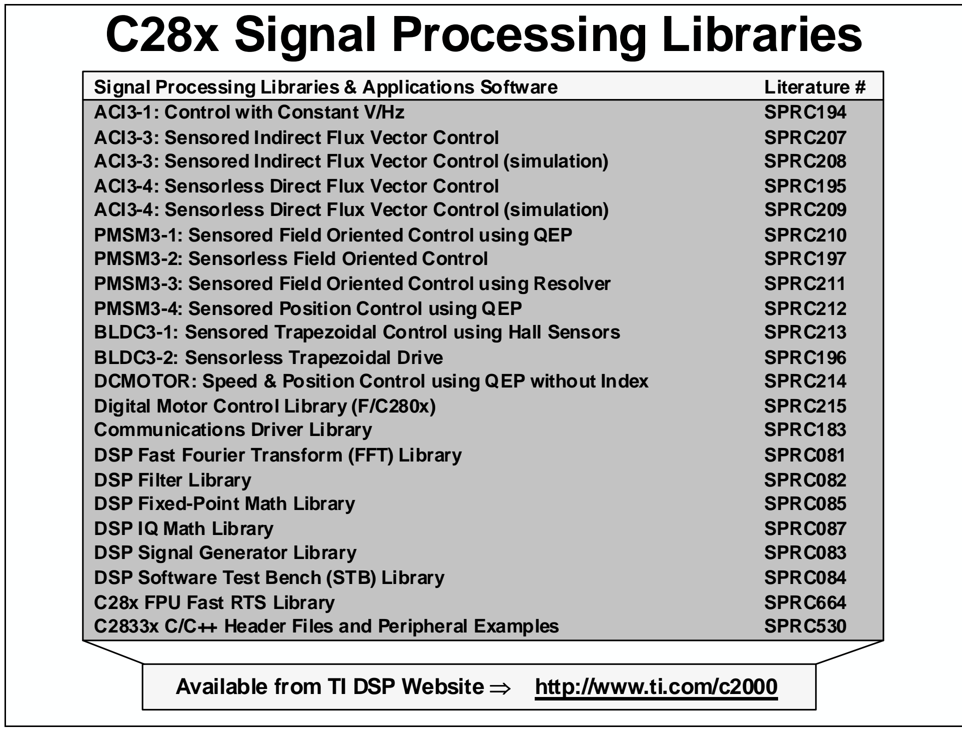


- DC/DC Kit includes
  - 8-rail DC/DC EVM using TI PowerTrain™ modules (10A)
  - F28044 controlCARD
  - C2000 Applications Software CD with example code and full hardware details
  - Code Composer Studio v3.3 with code size limit of 32KB
  - 9V DC power supply
- Available through TI authorized distributors and on the TI web
  - Part Number: TMDSDCDC8KIT

C2000 AC/DC Developer’s Kit

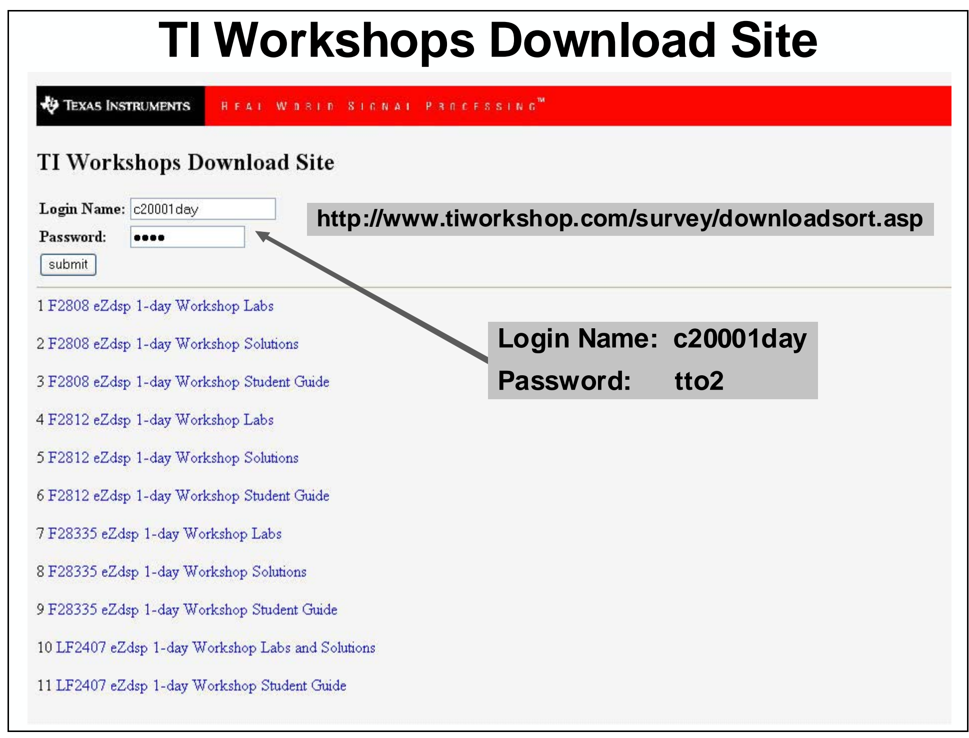
- AC/DC Kit includes
  - AC/DC EVM with interleaved PFC and phase-shifted full-bridge
  - F2808 controlCARD
  - C2000 Applications Software CD with example code and full hardware details
  - Code Composer Studio v3.3 with code size limit of 32KB
- AC/DC EVM features
  - 12VAC in, 80W/10A output
  - Primary side control
  - Synchronous rectification
  - Peak current mode control
  - Two-phase PFC with current balancing
- Available through TI authorized distributors and on theTI web
  - Part Number: TMDSACDCKIT

Development Support


C28x Signal Processing Libraries

Signal Processing Libraries & Applications Software	Literature #
ACI3-1: Control with Constant V/Hz	SPRC194
ACI3-3: Sensored Indirect Flux Vector Control	SPRC207
ACI3-3: Sensored Indirect Flux Vector Control (simulation)	SPRC208
ACI3-4: Sensorless Direct Flux Vector Control	SPRC195
ACI3-4: Sensorless Direct Flux Vector Control (simulation)	SPRC209
PMSM3-1: Sensored Field Oriented Control using QEP	SPRC210
PMSM3-2: Sensorless Field Oriented Control	SPRC197
PMSM3-3: Sensored Field Oriented Control using Resolver	SPRC211
PMSM3-4: Sensored Position Control using QEP	SPRC212
BLDC3-1: Sensored Trapezoidal Control using Hall Sensors	SPRC213
BLDC3-2: Sensorless Trapezoidal Drive	SPRC196
DCMOTOR: Speed & Position Control using QEP without Index	SPRC214
Digital Motor Control Library (F/C280x)	SPRC215
Communications Driver Library	SPRC183
DSP Fast Fourier Transform (FFT) Library	SPRC081
DSP Filter Library	SPRC082
DSP Fixed-Point Math Library	SPRC085
DSP IQ Math Library	SPRC087
DSP Signal Generator Library	SPRC083
DSP Software Test Bench (STB) Library	SPRC084
C28x FPU Fast RTS Library	SPRC664
C2833x C/C++ Header Files and Peripheral Examples	SPRC530

Available from TI DSP Website ⇒ http://www.ti.com/c2000



Customers Are Using C2000™ Products For …

Motor Control

Active suspension
Air conditioners
Aircraft A/C
Bonding machines
Building automation
Cameras
Car A/C
CD and DVD drives
Check readers
CNC control
Compressors
Copiers
Door openers
Elevator motor control
Encoders
Fan control
Food mixers
Fork lifts
Fuel pumps
Golf trainers
Hair removers
Heart/lung machines
Human transporters
Industrial drives
Inverters
Lawn mowers
Magnetic bearings
Mass flow control
Medical pumps
Missile control
Plotters
Postal sorters
Power assisted steering
Power drills
Power generators
Printers
Refrigerators
Robot control
Sewing machines
Ship propulsion control
Stepper motor control
Textile machines
Toy trains
Treadmills
Vacuum cleaners
Vibration control
Wafer testers
Washing machines
Windmill control

Digital Power

Battery charging
Frequency converters
Fuel cell control
Server power control
Solar energy control
Uninterr. power supplies

Optical Networking

TEC control
Optical switch control
Tunable laser control

Others

Adaptive cruise control
Airbag control
Antitheft systems
Blood analyzers
Data encryption systems
E-meters
Gas sensors
GPS systems
Ignition control
Induction ovens
Park assist systems
Power line modems
Radar control
Reactor monitoring
RF ID systems
Spectrum analyzers
Telecom switches
Tire pressure sensing
Ultrasound scanners
Welding equipment
Wireless modems
Color/light sensors

… and many more

For More Information . . .

Internet

Website: `http://www.ti.com`

FAQ: http://www-k.ext.ti.com/sc/technical_support/knowledgebase.htm

- Device information
- Application notes
- Technical documentation
- my.ti.com
- News and events
- Training

Enroll in Technical Training: **http://www.ti.com/sc/training**

USA - Product Information Center (PIC)

Phone: `800-477-8924` **or** `972-644-5580`

Email: `support@ti.com`

- Information and support for <u>all</u> TI Semiconductor products/tools
- Submit suggestions and errata for tools, silicon and documents

European Product Information Center (EPIC)

Web: http://www-k.ext.ti.com/sc/technical_support/pic/euro.htm

Phone:

Language	Number
Belgium (English)	+32 (0) 27 45 55 32
France	+33 (0) 1 30 70 11 64
Germany	+49 (0) 8161 80 33 11
Israel (English)	1800 949 0107 (free phone)
Italy	800 79 11 37 (free phone)
Netherlands (English)	+31 (0) 546 87 95 45
Spain	+34 902 35 40 28
Sweden (English)	+46 (0) 8587 555 22
United Kingdom	+44 (0) 1604 66 33 99
Finland (English)	+358(0) 9 25 17 39 48

Fax: **All Languages** +49 (0) 8161 80 2045

Email: epic@ti.com

- Literature, Sample Requests and Analog EVM Ordering
- Information, Technical and Design support for all Catalog TI Semiconductor products/tools
- Submit suggestions and errata for tools, silicon and documents

Appendix A – eZdsp™ F28335

Note: This appendix only provides a description of the eZdsp™ F28335 interfaces used in this workshop. For a complete description of all features and details, please see the *eZdsp™ F28335 Technical Reference* manual.

Module Topics

Appendix A – eZdsp™ F28335 .... **A-1**

Module Topics *A-2*

eZdsp™ F28335 .... *A-3*
eZdsp™ F28335 Connector / Header and Pin Diagram A-3
P2 – Expansion Interface A-5
P4/P8/P7 – I/O Interface A-6
P5/P9 – Analog Interface A-8
P10 – Expansion Interface A-9
SW1 – Boot Load Option Switch A-10
DS1/DS2 – LEDs A-11
TP1/TP2/TP3/TP4 – Test Points A-11

eZdsp™ F28335

eZdsp™ F28335 Connector / Header and Pin Diagram

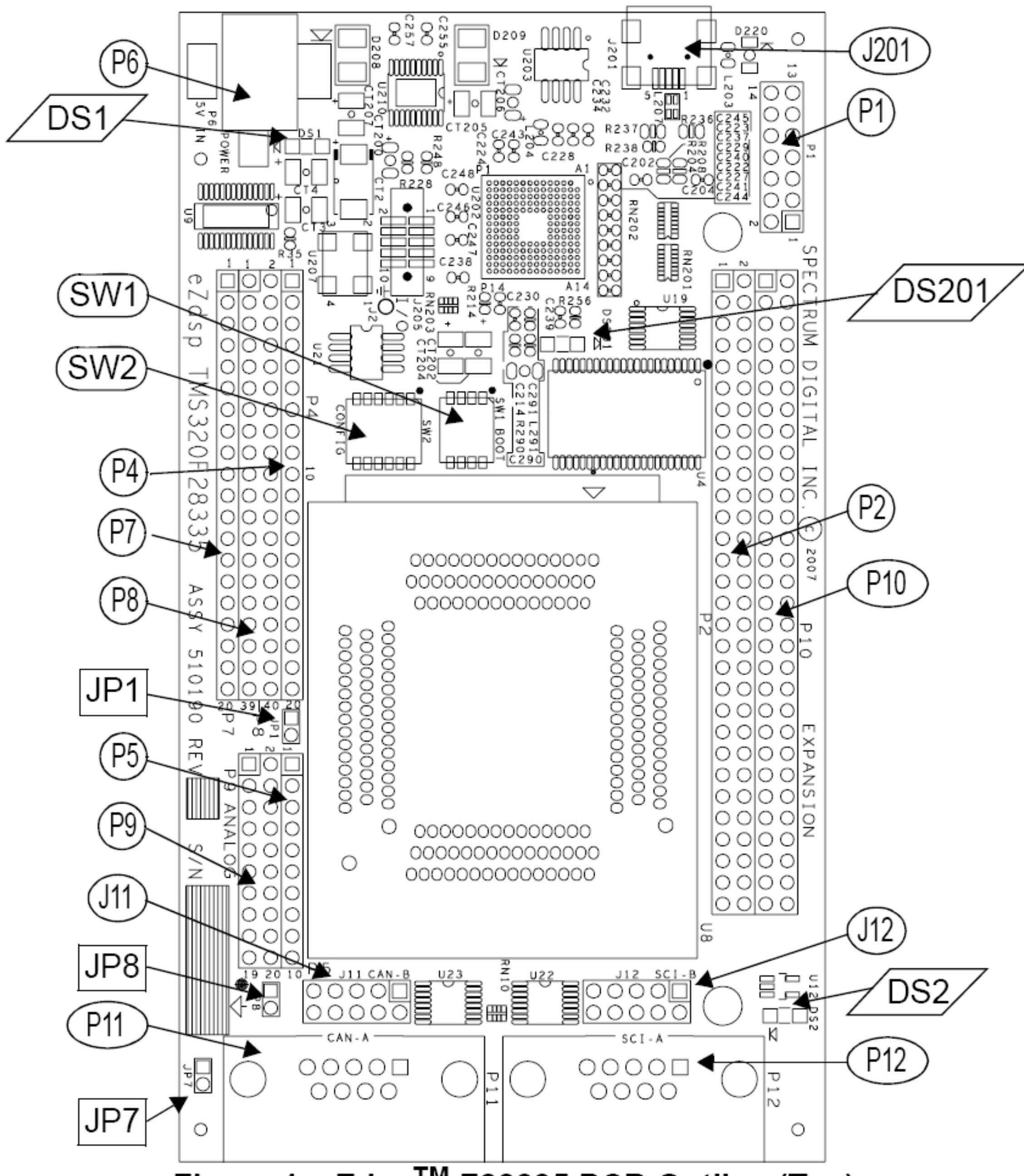


Figure 1, eZdsp™ F28335 PCB Outline (Top)

Table 1: eZdsp^{TM} F28335 Connectors

Connector	Function
P1	JTAG Interface
P2	Expansion
P4/P8/P7	I/O Interface
P5/P9	Analog Interface
P6	Power Connector
P10	Expansion
P11	CAN-A
P12	SCI-A
J11	CAN-B
J12	SCI-B
J201	Embedded JTAG

P2 – Expansion Interface

The positions of the 60 pins on the P2 connector are shown in the figure below.

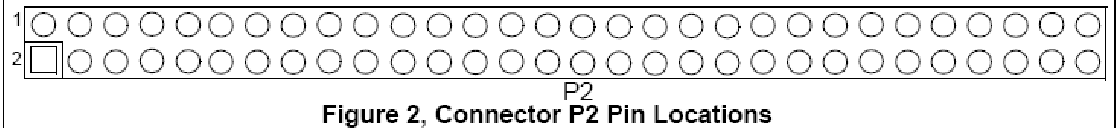


Figure 2, Connector P2 Pin Locations

The definition of P2, which has the I/O signal interface is shown below.

Table 2: P2, Expansion Interface Connector

Pin #	Signal	Pin #	Signal
1	+3.3V/+5V/NC *	2	+3.3/+5V/NC *
3	GPIO79_XD0	4	GPIO78_XD1
5	GPIO77_XD2	6	GPIO76_XD3
7	GPIO75_XD4	8	GPIO74_XD5
9	GPIO73_XD6	10	GPIO72_XD7
11	GPIO71_XD8	12	GPIO70_XD9
13	GPIO69_XD10	14	GPIO68_XD11
15	GPIO67_XD12	16	GPIO66_XD13
17	GPIO65_XD14	18	GPIO64_XD15
19	GPIO40_XA0_XWE1n	20	GPIO41_XA1
21	GPIO42_XA2	22	GPIO43_XA3
23	GPIO44_XA4	24	GPIO45_XA5
25	GPIO46_XA6	26	GPIO47_XA7
27	GPIO80_XA8	28	GPIO81_XA9
29	GPIO82_XA10	30	GPIO83_XA11
31	GPIO84_XA12	32	GPIO85_XA13
33	GPIO86_XA14	34	GPIO87_XA15
35	GND	36	GND
37	GPIO36_SCIRXDA-XZCS0n	38	GPIO37_ECAP2_XZCS7n
39	GPIO34_ECAP1_XREADY	40	B_GPIO28_SCIRXDA_XZCS6n
41	GPIO35_SCIRXDA_XRNW	42	10K Pull-up
43	GPIO38_WE0n	44	XRDn
45	+3.3V	46	No connect
47	DSP_RSn	48	XCLKOUT
49	GND	50	GND
51	GND	52	GND
53	GPIO39_XA16	54	GPIO31_CANTXA_XA17
55	GPIO30_CANRXA_XA18	56	GPIO14_TZ3n_XHOLDn_SCITXB_MCLKXB
57	GPIO15_XHOLDAn_SCIRXDB_MFSXB	58	GPIO29_SCITXDA_XA19
59	No connect	60	No connect

* Default is No Connect (NC). User can jumper to +3.3V or +5V on backside of eZdsp with JR5.

P4/P8/P7 – I/O Interface

The connectors P4, P8, and P7 present the I/O signals from the DSC. The layout of these connectors are shown below.

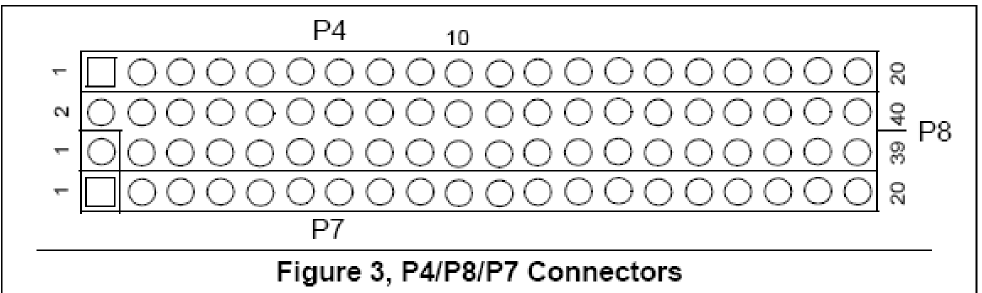


Figure 3, P4/P8/P7 Connectors

The pin definition of the P4 connector is shown in the table below.

Table 3: P4, I/O Connectors

Pin #	Signal
1	+3.3V/+5V/NC *
2	No connect
3	GPIO22_EQEP1S_MCLKRA_SCITXDB
4	GPIO7_EPWM4B_MCLKRA_ECAP2
5	GPIO23_EQEP1_MFSXA_SCIRXDB
6	GPIO5_EPWM3B_MFSRA_ECAP1
7	GPIO20_EAEP1A_MXDA_CANTXB
8	GPIO21_EQEP1B_MDRA_CANRXB
9	No connect
10	GND
11	GPIO3_EPWM2B_ECAP5_MCLKRB
12	GPIO1_EPWM1B/ECAP6/MFSRB
13	No connect
14	No connect
15	No connect
16	No connect
17	No connect
18	GPIO14_TZ3n_XHOLD_SCITXDB_MCLKXB
19	GPIO15_TZ4n_XHOLDA_SCIRXDB_MFSXB
20	GND

Table 4: P8, I/O Connectors

Pin #	Signal	Pin #	Signal
1	+3.3V/+5V/NC *	2	+3.3V/+5V/NC *
3	MUX_GPIO29_SCITXDA_XA19	4	MUX_GPIO28_SCIRXDA_XZCS6n
5	GPIO14_TZ3n_XHOLD_SCITXDB_MCLKXB	6	GPIO20_EAEP1A_MXDA_CANTXB
7	GPIO21_EQEP18_MDRA_CANRXB	8	GPIO23_EQEP1_MFSXA_SCIRXDB
9	GPIO0_EPWM1A	10	GPIO1_EPWM1B/ECAP6/MFSRB
11	GPIO2_EPWM2A	12	GPIO3_EPWM2B_ECAP5_MCLKRB
13	GPIO4_EPWM3A	14	GPIO5_EPWM3B_MFSRA_ECAP1
15	GPIO27_ECAP4_EQEP2S_MFSXB	16	GPIO6_EPWMN4A_EPWMSYNCI/EPWMSYNCO
17	GPIO13_TZ2N_CANRXB_MDRB	18	GPIO34_ECAP1_XREADY
19	GND	20	GND
21	GPIO7_EPWM4B_MCLKRA_ECAP2	22	GPIO15TZ4n_XHOLDA_SCIRXDB_MFSXB
23	GPIO16_SPISIMOA_CANTXB_TZ5n	24	GPIO17_SPISOMIA_CANRXB_TZ6n
25	GPIO18_SPICLKA_SCITXDB_CANRXA	26	GPIO19_SPISTAn_SCIRXDB_CANTXA
27	_MUX_GPIO31_CANRXA_XA17	28	MUX_GPIO30_CANRXA_XA18
29	MUX_GPIO11_EPWM6B_SCIRXDB_ECAP4	30	MUX_GPIO8EPWM5A_CANTXB_ADCSOCA0nP3
31	MUX_GPIO9_EPWM5B_SCITXDB_ECAP3	32	MUX_GPIO10_EPWM6A_CANRXB_ADCASOCB0n
33	MUX_GPIO22	34	GPIO25_ECAP2_EPEQ2B_MDRB
35	GPIO26_ECAP3_EQEP21_MCLKXB	36	GPIO32_SDAA_EPWMSYNCI_ADCSOCAOn
37	GPIO12_TZ1N_CANTXB_MDXB	38	GPIO33_SCLA_EPWNSYNCVO_ADCSOCBOn
39	GND	40	GND

* Default is No Connect (NC). User can jumper to +3.3V or +5V on backside of eZdsp with JR4.

The P7 connector is supplied for backwards compatibility. Signals from other connectors can be wired to this connector to support existing user interfaces. The pin definition of P7 connector is shown in the table below.

Table 5: P7, I/O Connector

Pin #	Signal	Pin #	Signal
1	No connect	11	No connect
2	No connect	12	No connect
3	No connect	13	No connect
4	No connect	14	No connect
5	No connect	15	No connect
6	No connect	16	No connect
7	No connect	17	No connect
8	No connect	18	No connect
9	No connect	19	No connect
10	No connect	20	GND

P5/P9 – Analog Interface

The position of the 30 pins on the P5/P9 connectors are shown in the diagram below as viewed from the top of the eZdsp.

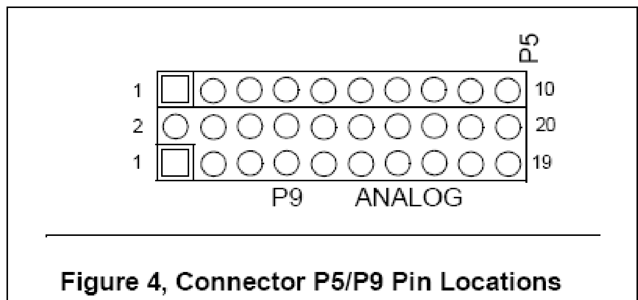


Figure 4, Connector P5/P9 Pin Locations

The definition of P5/P9 signals are shown in the table below.

Table 6: P5/P9, Analog Interface Connector

P5 Pin #	Signal	P9 Pin #	Signal	P9 Pin #	Signal
1	ADCINB0	1	GND	2	ADCINA0
2	ADCINB1	3	GND	4	ADCINA1
3	ADCINB2	5	GND	6	ADCINA2
4	ADCINB3	7	GND	8	ADCINA3
5	ADCINB4	9	GND	10	ADCINA4
6	ADCINB5	11	GND	12	ADCINA5
7	ADCINB6	13	GND	14	ADCINA6
8	ADCINB7	15	GND	16	ADCINA7
9	ADCREFM	17	GND	18	ADCLO *
10	ADCREFP	19	GND	20	No connect

* Connect ADCLO to AGND or ADCLO of target system for proper ADC operation.

P10 – Expansion Interface

The positions of the 60 pins on the P10 connector are shown in the figure below.

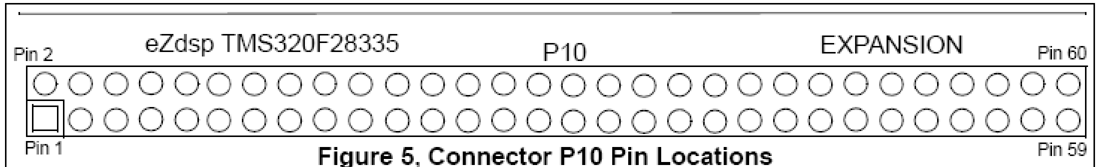


Figure 5, Connector P10 Pin Locations

The definition of P10, which has the I/O signal interface is shown below.

Table 7: P10, Expansion Interface Connector

Pin #	Signal	Pin #	Signal
1	+3.3V/+5V/NC	2	+3.3V/+5V/NC
3	GPIO63_SCITXDC_XD16	4	GPIO62_SCIRXDC_XD17
5	GPIO61_MFSRB_XD18	6	GPIO60_MCLKRB_XD19
7	GPIO59_MFSRA_XD20	8	GPIO58_MCLKRA_XD21
9	GPIO57_SPISTEAn_XD22	10	GPIO56_SPICLKA_XD23
11	GPIO55_SPISOMIA_XD24	12	GPIO54_SPISIMOA_XD25
13	GPIO53EQEP1_XD26	14	GPIO52_EQEP1S_XD27
15	GPIO51_EAEP1B_XD28	16	GPIO50_EQEP1A_XD29
17	GPIO49_ECAP6_XD30	18	GPIO48_ECAP5_XD31
19	No connect	20	No Connect
21	No connect	22	No Connect
23	No connect	24	No Connect
25	No connect	26	No Connect
27	No connect	28	No Connect
29	No connect	30	No Connect
31	No connect	32	No Connect
33	No connect	34	No Connect
35	No connect	36	No Connect
37	No connect	38	No Connect
39	No connect	40	No Connect
41	No connect	42	No Connect
43	No connect	44	No Connect
45	No connect	46	No Connect
47	No connect	48	No Connect
49	No connect	50	No Connect
51	No connect	52	No Connect
53	No connect	54	No Connect
55	No connect	56	No Connect
57	No connect	58	No Connect
59	GND	60	GND

SW1 – Boot Load Option Switch

Switch SW1 is used to select the boot load option used by the F28335 processor on power up. These selections are shown in the table below.

Table 8: **SW1**, Boot Load Option Switch

PIN	Position 4 Boot-3 XA15	Position 3 Boot-2 XA14	Position 2 Boot-1 XA13	Position 1 Boot-0 XA12	Boot Mode
1111	OFF	OFF	OFF	OFF	Jump to Flash
1110	OFF	OFF	OFF	ON	SCI-A boot
1101	OFF	OFF	ON	OFF	SPI-A boot *
1100	OFF	OFF	ON	ON	I^2C-A boot
1011	OFF	ON	OFF	OFF	eCAN-A boot
1010	OFF	ON	OFF	ON	McBSP-A boot
1001	OFF	ON	ON	OFF	Jump to XINTX x16
1000	OFF	ON	ON	ON	Jump to XINTX x32
0111	ON	OFF	OFF	OFF	Jump to OTP
0110	ON	OFF	OFF	ON	Parallel GPIO I/O boot
0101	ON	OFF	ON	OFF	Parallel XINTF boot
0100	ON	OFF	ON	ON	Jump to SARAM
0011	ON	ON	OFF	OFF	Branch to check boot mode
0010	ON	ON	OFF	ON	Branch to Flash, skip ADC CAL
0001	ON	ON	ON	OFF	Branch to SARAM, skip ADC CAL
0000	ON	ON	ON	ON	Branch to SCI, skip ADC CAL

* As shipped from the factory.

The figure below shows the layout of SW1.

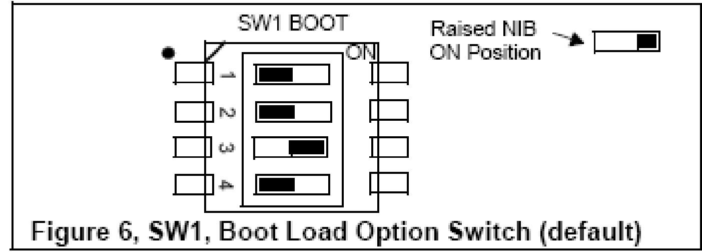


Figure 6, SW1, Boot Load Option Switch (default)

Position 4 GPIO87	Position 3 GPIO86	Position 2 GPIO85	Position 1 GPIO84	Boot Mode
Right – 0	Left – 1	Right – 0	Right – 0	M0 SARAM
Left – 1	Left – 1	Left – 1	Left – 1	FLASH

DS1/DS2 – LEDs

The eZdsp™ F28335 has three light-emitting diodes. DS1 indicates the presence of +5 volts and is normally 'on' when power is applied to the board. LED DS2 is under control of the GPIO32 line from the processor. DS201 is connected to the embedded USB emulator and shows the status of the emulation link. These are shown in the table below.

Table 9: LEDs

LED #	Color	Controlling Signal
DS1	Green	+5 Volts
DS2	Green	GPIO32
DS201	Green	Embedded emulation link status

TP1/TP2/TP3/TP4 – Test Points

Table 10: Test Points

Test Point	Signal
TP1	AGND
TP2	XCLKOUT
TP3	U8(DSP) Pin 81, TEST1
TP4	U8(DSP) Pin 82, TEST2