C2000™ Piccolo™ 1-Day Workshop

Workshop Guide and Lab Manual

F28xPodw
Revision 2.1
December 2010

TTO

Technical Training
Organization

Important Notice

Revision History

April 2009 – Revision 1.0

October 2009 – Revision 1.1

June 2010 – Revision 2.0

December 2010 – Revision 2.1

Mailing Address

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

Workshop Topics

Workshop Introduction

C2000™ Piccolo™ 1-Day Workshop

Texas Instruments
Technical Training

TTO
Technical Training Organization

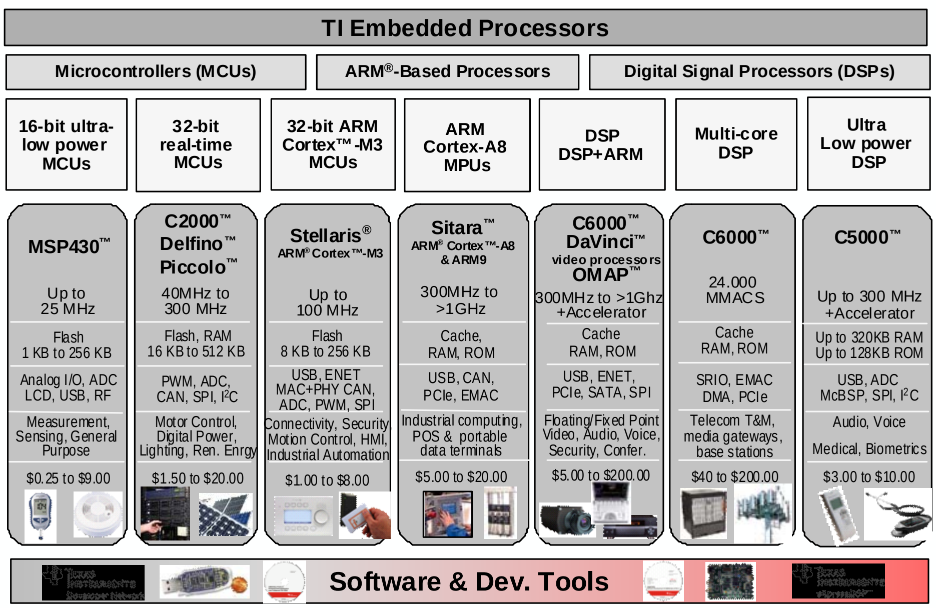
C2000 Piccolo 1-Day Workshop Outline

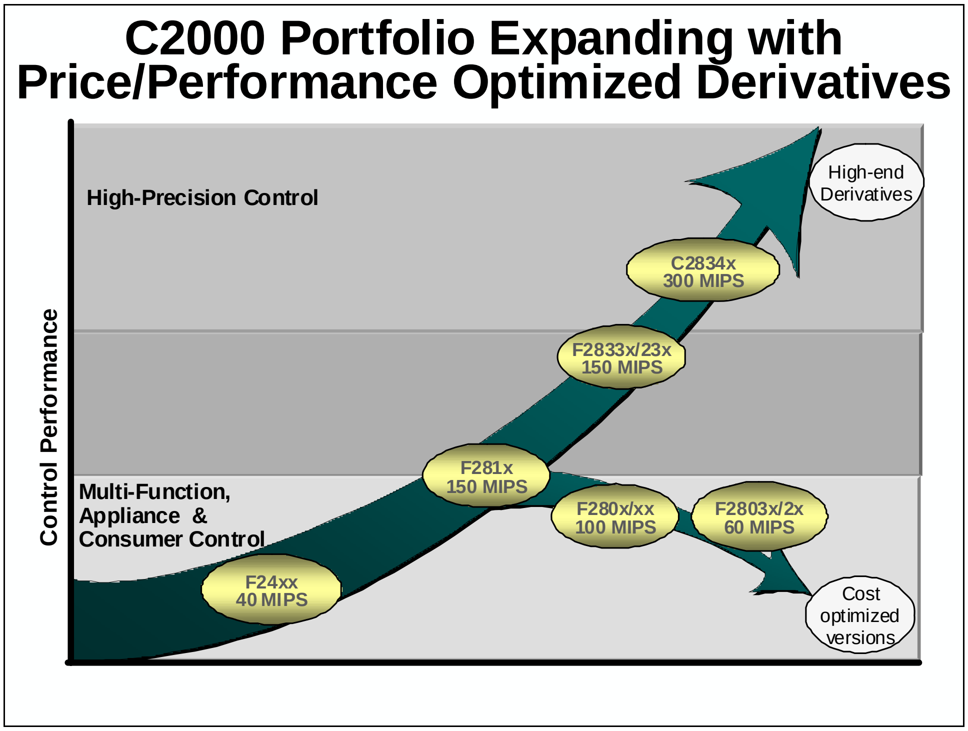
- **Workshop Introduction**
- **Architecture Overview**
- **Programming Development Environment**
  - *Lab: Linker command file*
- **Peripheral Register Header Files**
- **Reset, Interrupts and System Initialization**
  - *Lab: Watchdog and interrupts*
- **Control Peripherals**
  - *Lab: Generate and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

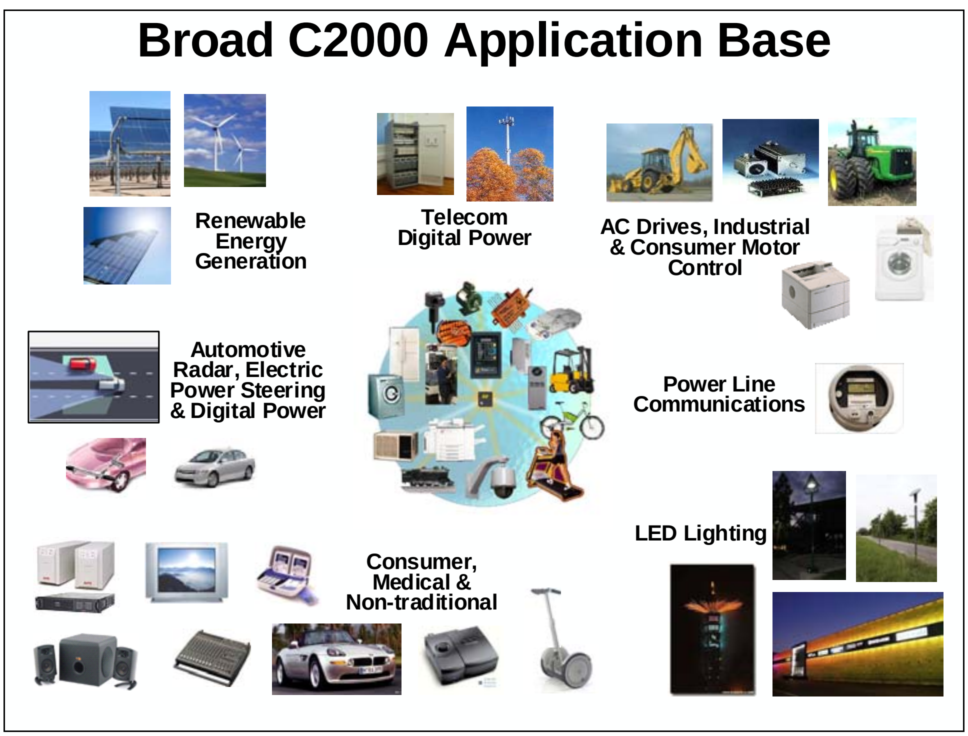
Introductions

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

TI Embedded Processing Portfolio




C2000 Portfolio Expanding with Price/Performance Optimized Derivatives
High-Precision Control
Multi-Function, Appliance & Consumer Control
Control Performance
F24xx 40 MIPS
F281x 150 MIPS
F2833x/23x 150 MIPS
C2834x 300 MIPS
F280x/xx 100 MIPS
F2803x/2x 60 MIPS
High-end Derivatives
Cost optimized versions


Broad C2000 Application Base
Renewable Energy Generation
Telecom Digital Power
AC Drives, Industrial & Consumer Motor Control
Automotive Radar, Electric Power Steering & Digital Power
Power Line Communications
LED Lighting
Consumer, Medical & Non-traditional

C2000 Piccolo™ Microcontroller Family
F2802x / F2803x

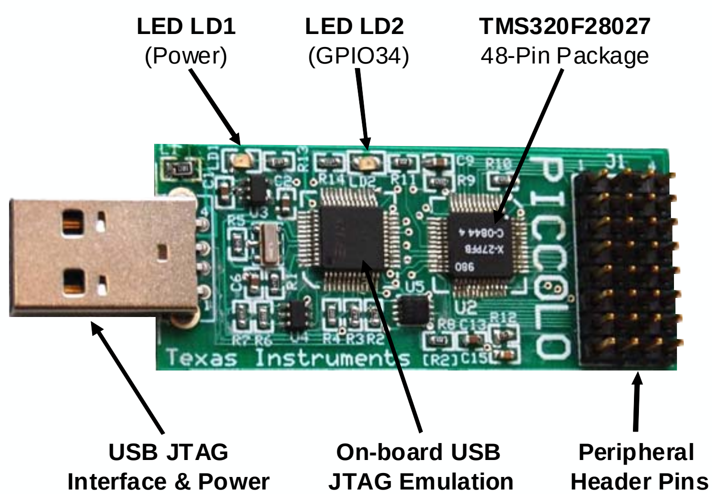
	MHz	Flash (x16)	RAM (x16)	CLA	Analog Comp*	ADC* (ch)	PWM / (HR)*	CAP	QEP	Communication Ports
F280200	40	8Kw	3Kw	No	1 / 2	7 / 13	8 (0)	0	0	SPI, SCI, I2C
F28020	40	16Kw	3Kw	No	1 / 2	7 / 13	8 (0)	1	0	SPI, SCI, I2C
F28021	40	32Kw	5Kw	No	1 / 2	7 / 13	8 (0)	1	0	SPI, SCI, I2C
F28022	50	16Kw	6Kw	No	1 / 2	7 / 13	8 (4)	1	0	SPI, SCI, I2C
F28023	50	32Kw	6Kw	No	1 / 2	7 / 13	8 (4)	1	0	SPI, SCI, I2C
F28026	60	16Kw	6Kw	No	1 / 2	7 / 13	8 (4)	1	0	SPI, SCI, I2C
F28027	60	32Kw	6Kw	No	1 / 2	7 / 13	8 (4)	1	0	SPI, SCI, I2C
F28030	60	16Kw	6Kw	No	3	14 / 16	12(0) / 14(0)	1	1	SPI,SCI,I2C,LIN,eCAN
F28031	60	32Kw	8Kw	No	3	14 / 16	12(0) / 14(0)	1	1	SPI,SCI,I2C,LIN,eCAN
F28032	60	32Kw	10Kw	No	3	14 / 16	12(6) / 14(7)	1	1	SPI,SCI,I2C,LIN,eCAN
F28033	60	32Kw	10Kw	Yes	3	14 / 16	12(6) / 14(7)	1	1	SPI,SCI,I2C,LIN,eCAN
F28034	60	64Kw	10Kw	No	3	14 / 16	12(6) / 14(7)	1	1	SPI,SCI,I2C,LIN,eCAN
F28035	60	64Kw	10Kw	Yes	3	14 / 16	12(6) / 14(7)	1	1	SPI,SCI,I2C,LIN,eCAN

** number dependent on package type: F2802x – 38/48 pins, F2803x – 64/80 pins*

- All devices have VREG, POR/BOR, Watchdog, OTP, CPU Timers

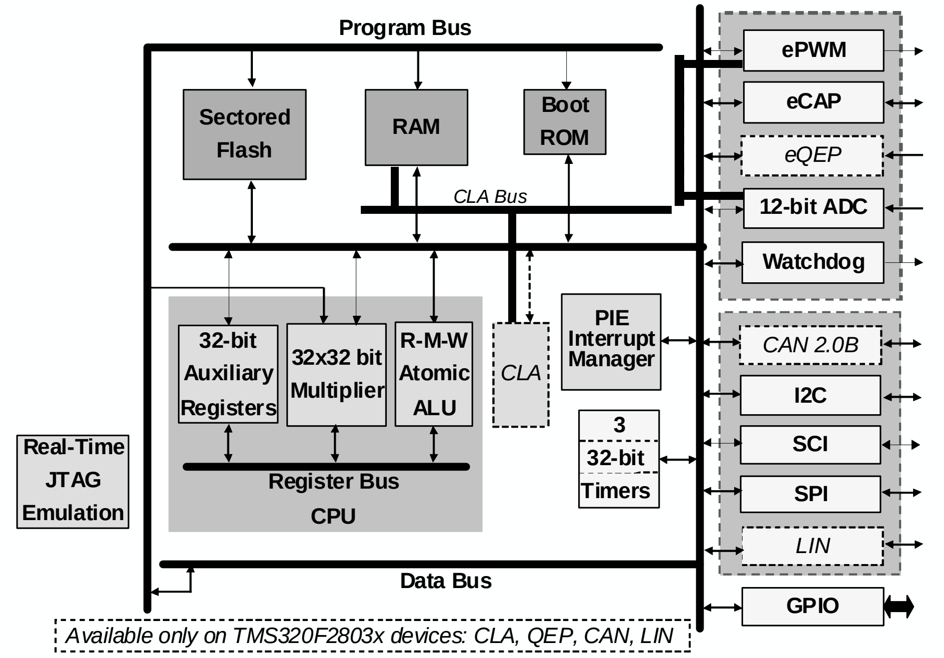
For details and information on other C2000 family members refer to the "Embedded Processing Guide" and specific "Data Manuals"

Piccolo™ controlSTICK

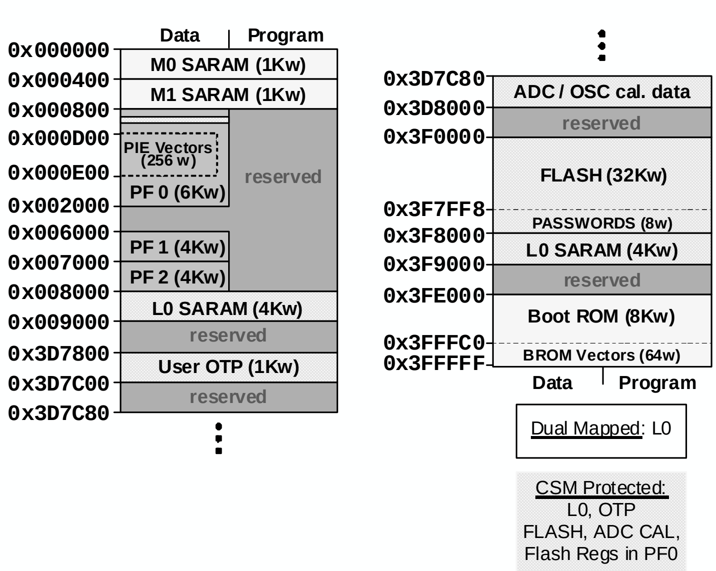


Architecture Overview

TMS320F2802x/3x Block Diagram



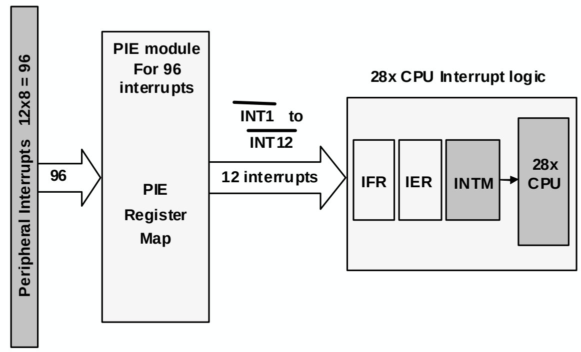
TMS320F28027 Memory Map



F28x Fast Interrupt Response Manager

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

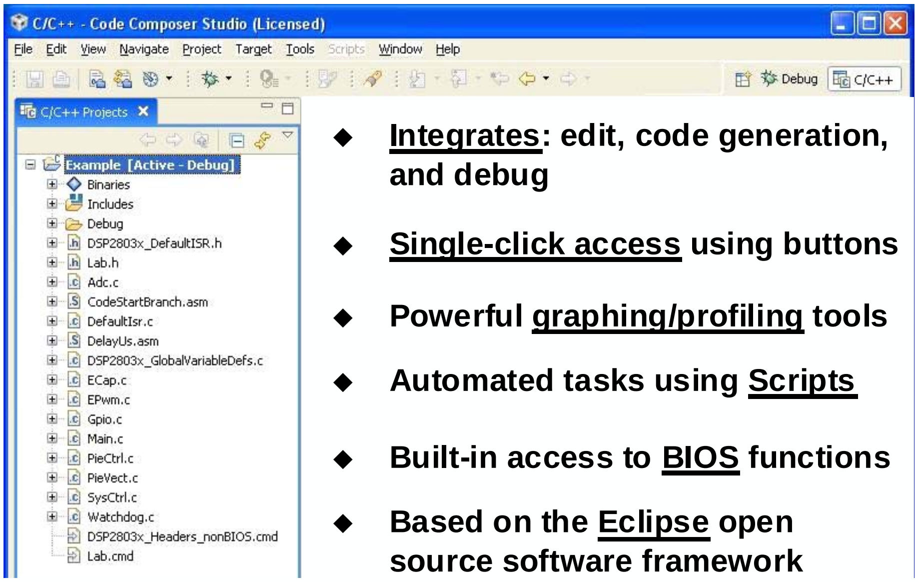
Auto Context Save	
T	ST0
AH	AL
PH	PL
AR1 (L)	AR0 (L)
DP	ST1
DBSTAT	IER
PC(msw)	PC(lsw)



Programming Development Environment

Code Composer Studio

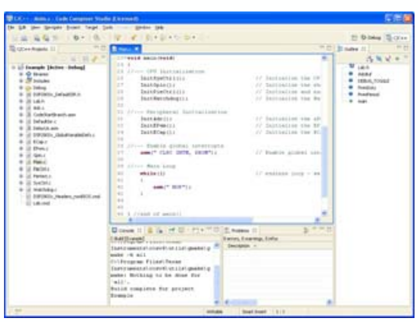
Code Composer Studio: IDE

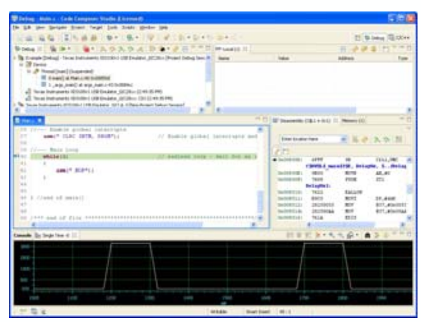


- **Integrates**: edit, code generation, and debug
- **Single-click access** using buttons
- Powerful **graphing/profiling** tools
- Automated tasks using **Scripts**
- Built-in access to **BIOS** functions
- Based on the **Eclipse** open source software framework

C/C++ and Debug Perspective (CCSv4)

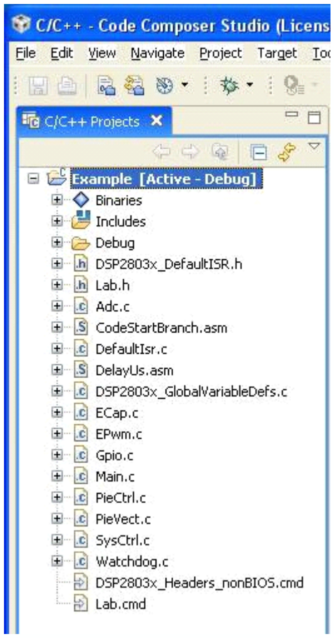
- Each perspective provides a set of functionality aimed at accomplishing a specific task





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

CCSv4 Project



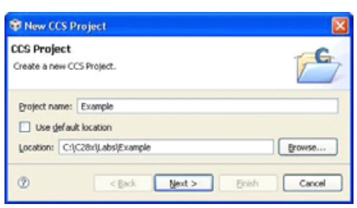
Project files contain:

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

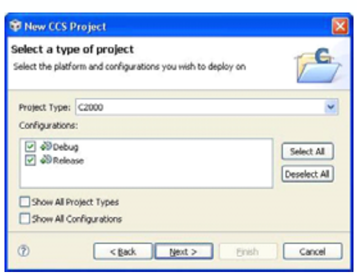
Creating a New CCSv4 Project

- ***File → New → CCS Project***

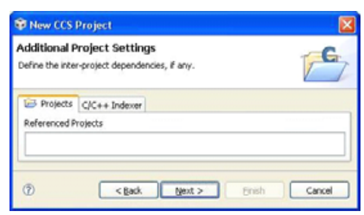
1



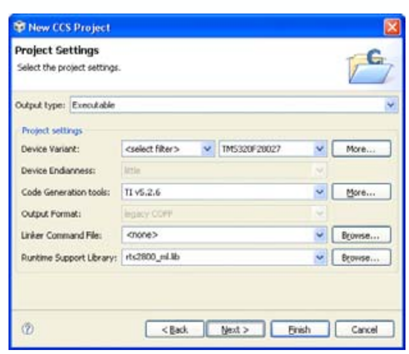
2



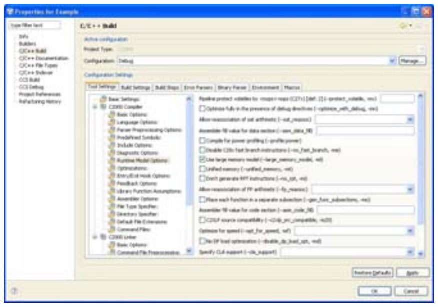
3

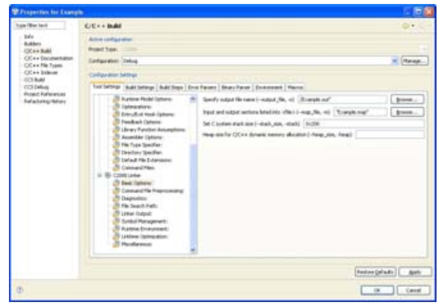


4



CCSv4 Build Options – Compiler / Linker





- **Compiler**
  - **16 categories for code generation tools**
  - **Controls many aspects of the build process, such as:**
    - **Optimization level**
    - **Target device**
    - **Compiler / assembly / link options**
- **Linker**
  - **9 categories for linking**
    - **Specify various link options**
  - ***${PROJECT_ROOT}* specifies the current project directory**

Linking Sections in Memory

Sections

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

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

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

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

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

Compiler Section Names

Initialized Sections

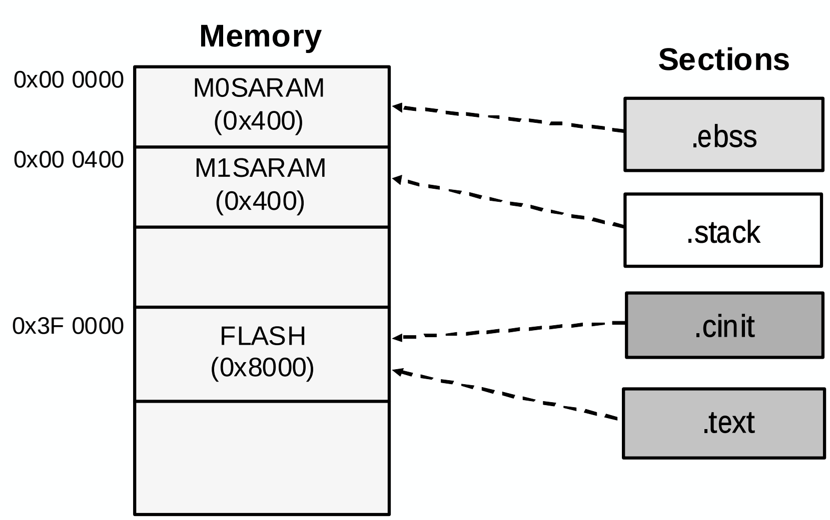
Name	Description	Link Location
.text	code	FLASH
.cinit	initialization values for global and static variables	FLASH
.econst	constants (e.g. const int k = 3;)	FLASH
.switch	tables for switch statements	FLASH
.pinit	tables for global constructors (C++)	FLASH

Uninitialized Sections

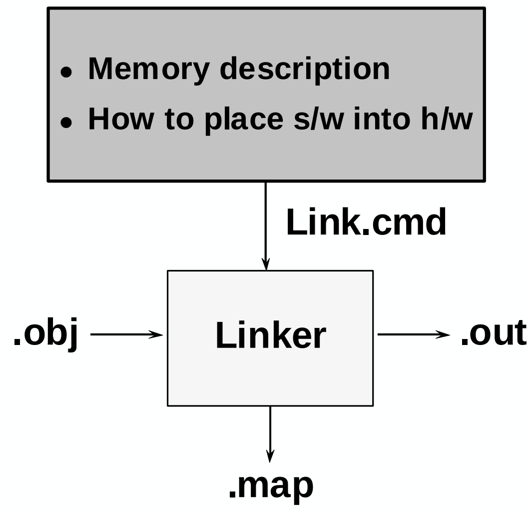
Name	Description	Link Location
.ebss	global and static variables	RAM
.stack	stack space	low 64Kw RAM
.esysmem	memory for far malloc functions	RAM

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

Placing Sections in Memory



Linking



Linker Command File

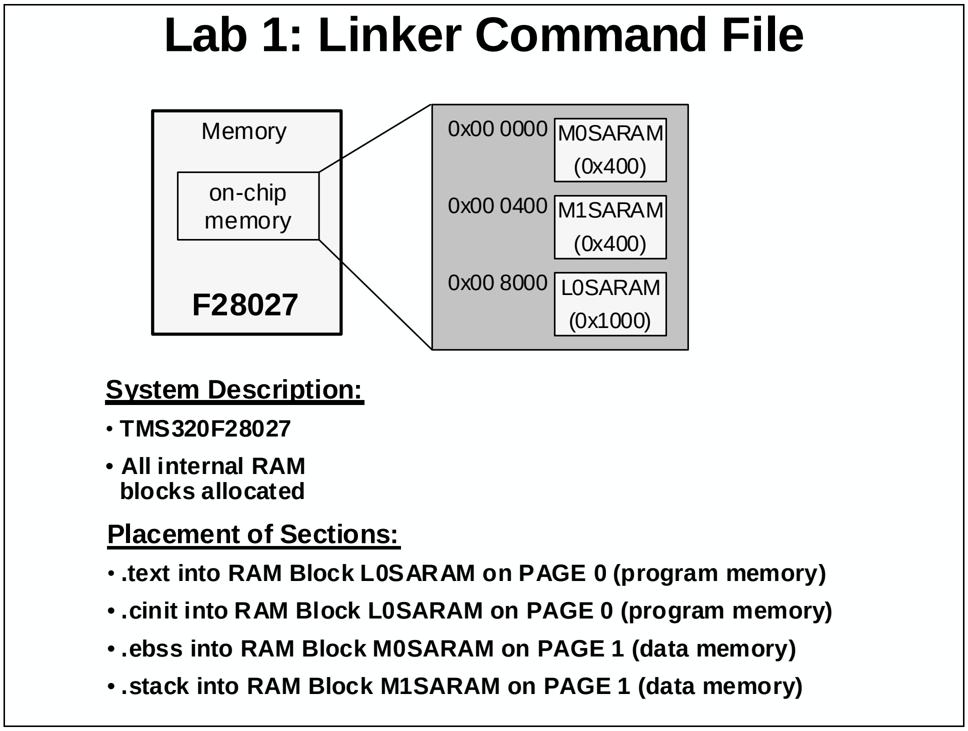
```
MEMORY
{
  PAGE 0:        /* Program Memory */
   FLASH:   origin = 0x3F0000, length = 0x8000

  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

- TMS320F28027
- All internal RAM blocks allocated

Placement of Sections:

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

➢ Procedure

Start Code Composer Studio and Open a Workspace

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

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

2. The first time CCS opens a "Welcome to Code Composer Studio v4" page appears. Close the page by clicking on the CCS icon in the upper right or by clicking the `X` on the "Welcome" tab. You should now have an empty workbench. The term workbench refers to the desktop development environment. Maximize CCS to fill your screen.

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

Setup Target Configuration

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

   `Target → New Target Configuration…`

   In the file name field type **F28027_ctrlSTK.ccxml**. This is just a descriptive name since multiple target configuration files can be created. Leave the "Use shared location" box checked and select `Finish`.

4. In the next window that appears, select the emulator using the "Connection" pull-down list and choose "`Texas Instruments XDS100v1 USB Emulator`". In the box below, check the box to select "`controlSTICK – Piccolo F28027`". Click `Save` to save the configuration, then close the "Cheat Sheets" and "F28027_ctrlSTK.ccxml" setup window by clicking the `X` on the tabs.

5. To view the target configurations, click:

   `View → Target Configurations`

   and click the plus sign (+) to the left of `User Defined`. Notice that the F28027_ctrlSTK.ccxml file is listed and set as the default. If it is not set as the default, right-click on the .ccxml file and select "Set as Default". Close the Target Configurations window by clicking the `X` on the tab.

Create a New Project

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

   `File → New → CCS Project`

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

`C:\C28x\Labs\Lab1\Project`

Click `OK` and then click `Next`.

7. The next window that appears selects the platform and configurations. Select the "Project Type" using the pull-down list and choose "`C2000`". In the "Configurations" box below, leave the "Debug" and "Release" boxes checked. This will create folders that will hold the output files. Click `Next`.

8. In the next window, inter-project dependencies (if any) are defined. Select `Next`.

9. In the last window, the CCS project settings are selected. Change the "Device Variant" using the pull-down list to "`TMS320F28027`". Next, using the pull-down list change the "Linker Command File" to "`<none>`". We will be using our own linker command file, rather than the one supplied by CCS. The "Runtime Support Library" will be automatically set to "`rts2800_ml.lib`". This will select the large memory model runtime support library. Click `Finish`.

10. A new project has now been created. Notice the `C/C++ Projects` window contains `Lab1`. The project is set `Active` and the output files will be located in the `Debug` folder. At this point, the project does not include any source files. The next step is to add the source files to the project.

11. To add the source files to the project, right-click on `Lab1` in the `C/C++ Projects` window and select:

    `Add Files to Project…`

    or click: `Project → Add Files to Active Project…`

    and make sure you're looking in `C:\C28x\Labs\Lab1\Files`. With the "files of type" set to view all files (`*.*`) select `Lab1.c` and `Lab1.cmd` then click `OPEN`. This will add the files to the project.

12. In the `C/C++ Projects` window, click the plus sign (+) to the left of `Lab1` and notice that the files are listed.

Project Build Options

13. There are numerous build options in the project. Most default option settings are sufficient for getting started. We will inspect a couple of the default options at this time. Right-click on Lab1 in the `C/C++ Projects` window and select `Properties` or click:

    `Project → Properties`

14. A "Properties" window will open and in the section on the left be sure that "C/C++ Build" category is selected. In the "Configuration Settings" section make sure that the `Tool Settings` tab is selected. Next, under "C2000 Linker" select the "`Basic Options`". Notice that .out and .map files are being specified. The .out file is the executable code that will be loaded into the MCU. The .map file will contain a linker report showing memory usage and section addresses in memory.

15. Next in the "`Basic Options`" set the Stack Size to **`0x200`**.

16. Under "C2000 Compiler" select the "`Runtime Model Options`". Notice the "Use large memory model" and "Unified memory" boxes are checked. Select `OK` to save and close the Properties window.

Linker Command File – Lab1.cmd

17. Open and inspect `Lab1.cmd` by double clicking on the filename in the project window. Notice that the `Memory{}` declaration describes the system memory shown on the "Lab1: Linker Command File" slide in the objective section of this lab exercise. Memory block L0SARAM has been placed in program memory on page 0, and the other memory blocks have been placed in data memory on page 1.

18. In the `Sections{}` area notice that the sections defined on the slide have been "linked" into the appropriate memories. Also, notice that a section called .reset has been allocated. The .reset section is part of the rts2800_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

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

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

20. Click the "`Build`" button and watch the tools run in the `Console` window. Check for errors in the `Problems` window (we have deliberately put an error in `Lab1.c`). When you get an error, you will see the error message (in red) in the `Problems` window, and simply double-click the error message. The editor will automatically open to the source file containing the error, and position the mouse cursor at the correct code line.

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

22. Build the project again. There should be no errors this time.

23. CCS can automatically save modified source files, build the program, open the debug perspective view, connect and download it to the target, and then run the program to the beginning of the main function.

    Click on the "`Debug`" button (green bug) or click `Target → Debug Active Project`.

Notice the `Debug` icon in the upper right-hand corner indicating that we are now in the "Debug Perspective" view. The program ran through the C-environment initialization routine in the rts2800_ml.lib and stopped at `main()` in `Lab1.c`.

Debug Environment Windows

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

24. Open a "Memory" window to view the global variable "`z`".

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

    Type **`&z`** into the address field and select "`Data`" memory page. 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 Hex" in 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.

25. Notice the "Local(1)" window automatically opened and the local variables `x` and `y` are present. The local 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).

26. 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 top of the watch window. In the empty box in the "Name" column, type **`z`** and then enter. An ampersand is not used here. The watch window knows you are specifying a symbol. (Note that the watch window can be manually opened by clicking: `View` → `Watch Window` on the menu bar).

    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

27. Click the "Local (1)" tab at the top of the watch window. Single-step through `main()` by using the <F5> key (or you can use the `Step Into` button on the horizontal toolbar). Check to see if the program is working as expected. What is the value for "`z`" when you get to the end of the program?

Terminate Debug Session and Close Project

28. The `Terminate All` button will terminate the active debug session, close the debugger and return CCS to the "C/C++ Perspective" view.

    Click: `Target` → `Terminate All` or use the `Terminate All` icon: 

    Close the Terminate Debug Session "Cheat Sheet" by clicking on the `X` on the tab.

29. Next, close the project by right-clicking on `Lab1` in the `C/C++ Projects` window and select `Close Project`.

End of Exercise

Peripheral Register Header Files

Traditional Approach to C Coding

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

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

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

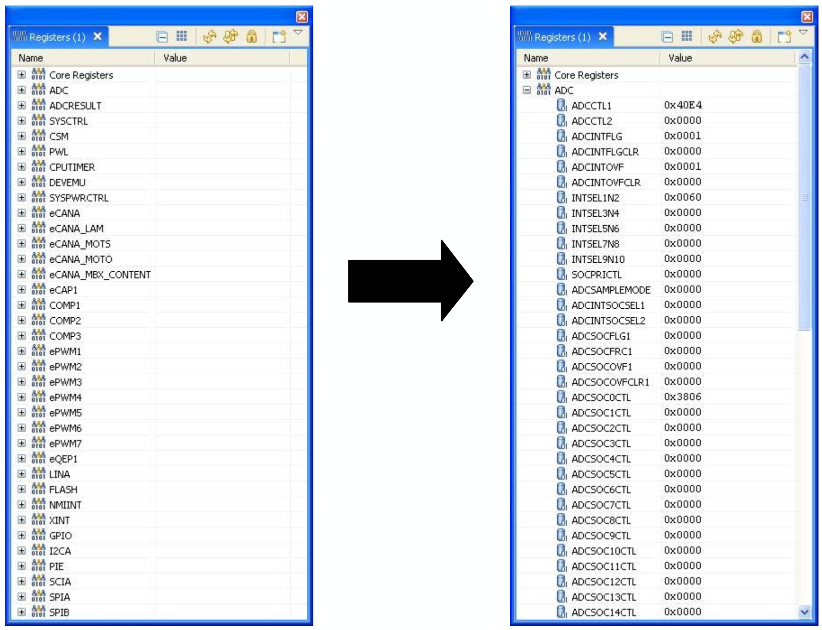
Structure Approach to C Coding

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

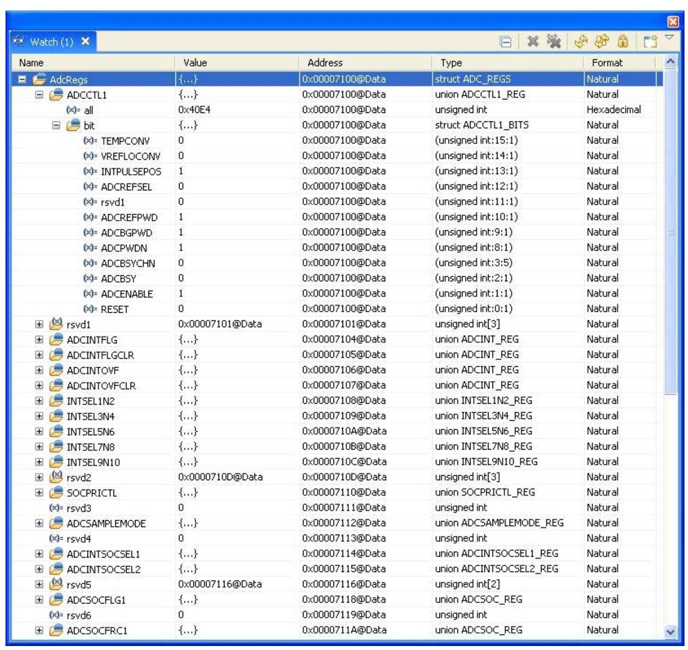
Advantages
- Easy to manipulate individual bits
- Watch window is amazing! (next slide)
- Generates most efficient code (on C28x)

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

Built-in CCSv4 Register Window



CCSv4 Watch Window using Structures



Structure Naming Conventions

- **The DSP2802x header files define:**
  - **All of the peripheral structures**
  - **All of the register names**
  - **All of the bit field names**
  - **All of the register addresses**

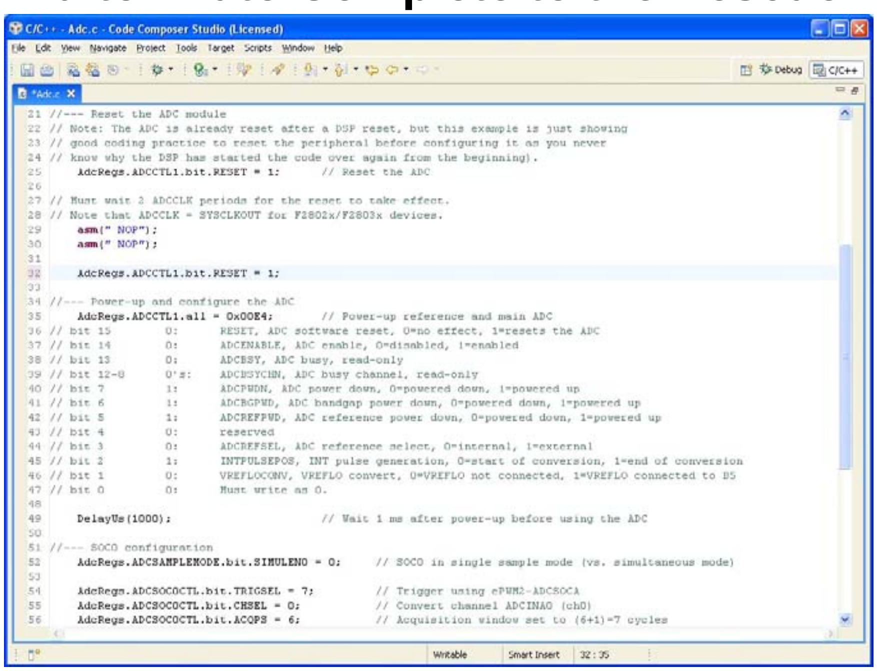
PeripheralName.RegisterName.all	**// Access full 16 or 32-bit register**
PeripheralName.RegisterName.half.LSW	**// Access low 16-bits of 32-bit register**
PeripheralName.RegisterName.half.MSW	**// Access high 16-bits of 32-bit register**
PeripheralName.RegisterName.bit.FieldName	**// Access specified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

Notes: [1] "PeripheralName" are assigned by TI and found in the DSP2802x 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!



DSP2802x Header File Package

(http://www.ti.com, literature # SPRC832)

- **Contains everything needed to use the structure approach**
- **Defines all peripheral register bits and register addresses**
- **Header file package includes:**
  - **\DSP2802x_headers\include → .h files**
  - **\DSP2802x_headers\cmd → linker .cmd files**
  - **\DSP2802x_headers\gel → .gel files for CCS**
  - **\DSP2802x_examples → CCS3 examples**
  - **\DSP2802x_examples_ccsv4 → CCS4 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 "DSP2802x_Device.h"

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

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

DSP2802x_Adc.h

```
// ADC Individual Register Bit Definitions:
struct ADCCTL1_BITS {         // bits  description
    Uint16  TEMPCONV:1;       // 0 Temperature sensor connection
    Uint16  VREFLOCONV:1;     // 1 VSSA connection
    Uint16  INTPULSEPOS:1;    // 2 INT pulse generation control
    Uint16  ADCREFSEL:1;      // 3 Internal/external reference select
    Uint16  rsvd1:1;          // 4 reserved
    Uint16  ADCREFPWD:1;      // 5 Reference buffers powerdown
    Uint16  ADCBGPWD:1;       // 6 ADC bandgap powerdown
    Uint16  ADCPWDN:1;        // 7 ADC powerdown
    Uint16  ADCBSYCHN:5;      // 12:8 ADC busy on a channel
    Uint16  ADCBSY:1;         // 13 ADC busy signal
    Uint16  ADCENABLE:1;      // 14 ADC enable
    Uint16  RESET:1;          // 15 ADC master reset
};
// Allow access to the bit fields or entire register:
union ADCCTL1_REG {
   Uint16               all;
   struct ADCCTL1_BITS   bit;
};
// ADC External References & Function Declarations:
extern volatile struct ADC_REGS AdcRegs;
```

Peripheral Structure .h files (2 of 2)

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

DSP2802x_Adc.h	DSP2802x_BootVars.h	DSP2802x_Comp.h
DSP2802x_CpuTimers.h	DSP2802x_DevEmu.h	DSP2802x_Device.h
DSP2802x_ECap.h	DSP2802x_EPwm.h	DSP2802x_Gpio.h
DSP2802x_I2c.h	DSP2802x_NmiIntrupt.h	DSP2802x_PieCtrl.h
DSP2802x_PieVect.h	DSP2802x_Sci.h	DSP2802x_Spi.h
DSP2802x_SysCtrl.h	DSP2802x_XIntrupt.h	

- ***DSP2802x_Device.h***
  - **Main include file**
  - **Will include all other .h files**
  - **Include this file (*directly or indirectly*) in each source file:**

    ```
    #include "DSP2802x_Device.h"
    ```

Global Variable Definitions File

DSP2802x_GlobalVariableDefs.c

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

DSP2802x_GlobalVariableDefs.c

```
#include "DSP2802x_Device.h"
…
#pragma DATA_SECTION(AdcRegs,"AdcRegsFile");
volatile struct ADC_REGS AdcRegs;
…
```

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

  DSP2802x_GlobalVariableDefs.c

Linker Command Files for the Structures

DSP2802x_nonBIOS.cmd

DSP2802x_GlobalVariableDefs.c

```
#include "DSP2802x_Device.h"
...
#pragma DATA_SECTION(AdcRegs,"AdcRegsFile");
volatile struct ADC_REGS AdcRegs;
...
```

DSP2802x_Headers_nonBIOS.cmd

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

- **Links each structure to the address of the peripheral using the structures named section**
- **Add this file to your CCS project:**

  DSP2802x_nonBIOS.cmd

Peripheral Specific Examples

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

adc_soc	epwm_updown_aq	lpm_idlewake
adc_temp_sensor	external_interrupt	lpm_standbywake
cpu_timer	flash	sci_echoback
ecap_apwm	gpio_setup	scia_loopback
ecap_capture_pwm	gpio_toggle	scia_loopback_interrupts
epwm_blanking_window	hrpwm	spi_loopback
epwm_dcevent_trip	hrpwm_duty_sfo_v6	spi_loopback_interrupts
epwm_dcevent_trip_comp	hrpwm_prdup_sfo_v6	sw_prioritized_interrupts
epwm_deadband	hrpwm_prdupdown_sfo_v6	timed_led_blink
epwm_timer_interrupts	hrpwm_slider	watchdog
epwm_trip_zone	i2c_eeprom	
epwm_up_aq	lpm_haltwake	

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:

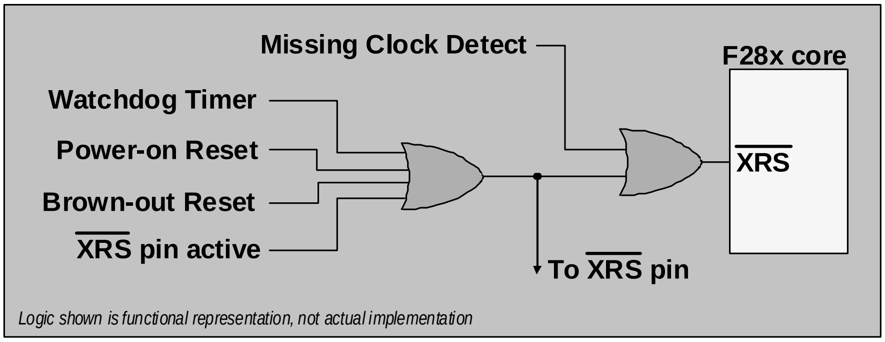
F2802x	# SPRC832
F2803x	# SPRC892
F2833x and F2823x	# SPRC530
F280x and F2801x	# SPRC191
F2804x	# SPRC324
F281x	# SPRC097

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

Reset, Interrupts and System Initialization

Reset

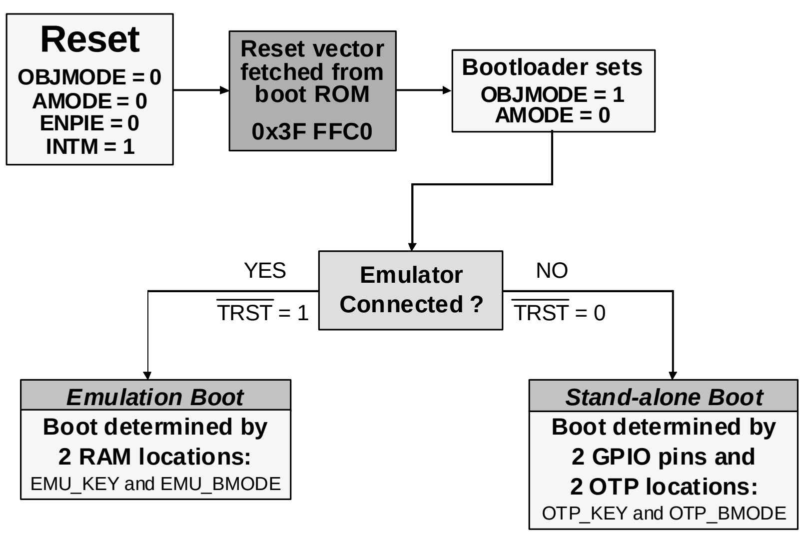
Reset Sources



- **POR – *Power-on Reset* generates a device reset during power-up conditions**
- **BOR – *Brown-out Reset* generates a device reset if the power supply drops below specification for the device**

Note: Devices support an on-chip voltage regulator (*VREG*) to generate the core voltage

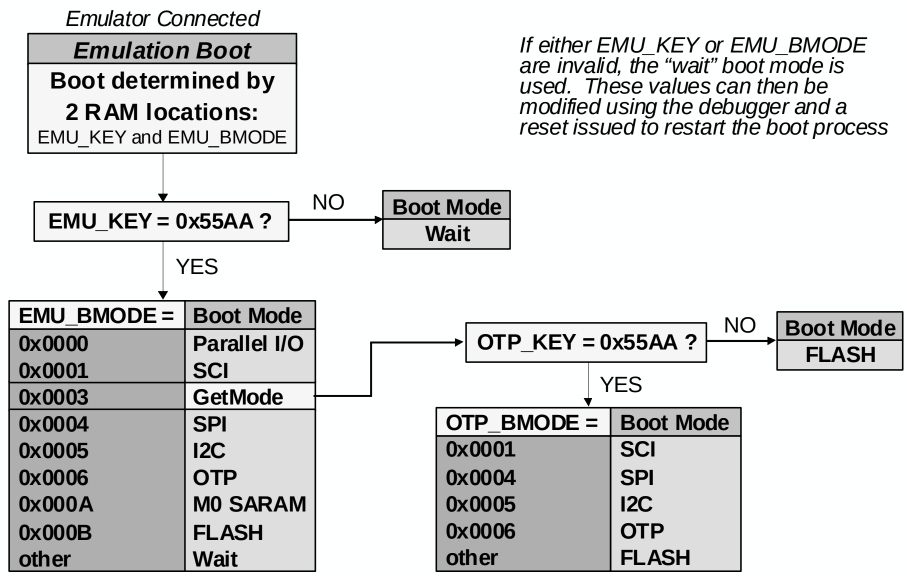
Reset – Bootloader



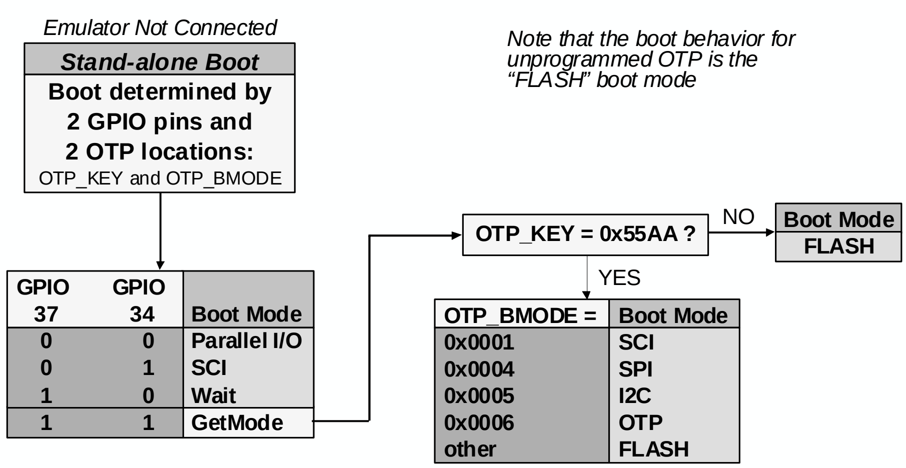
TRST = JTAG Test Reset

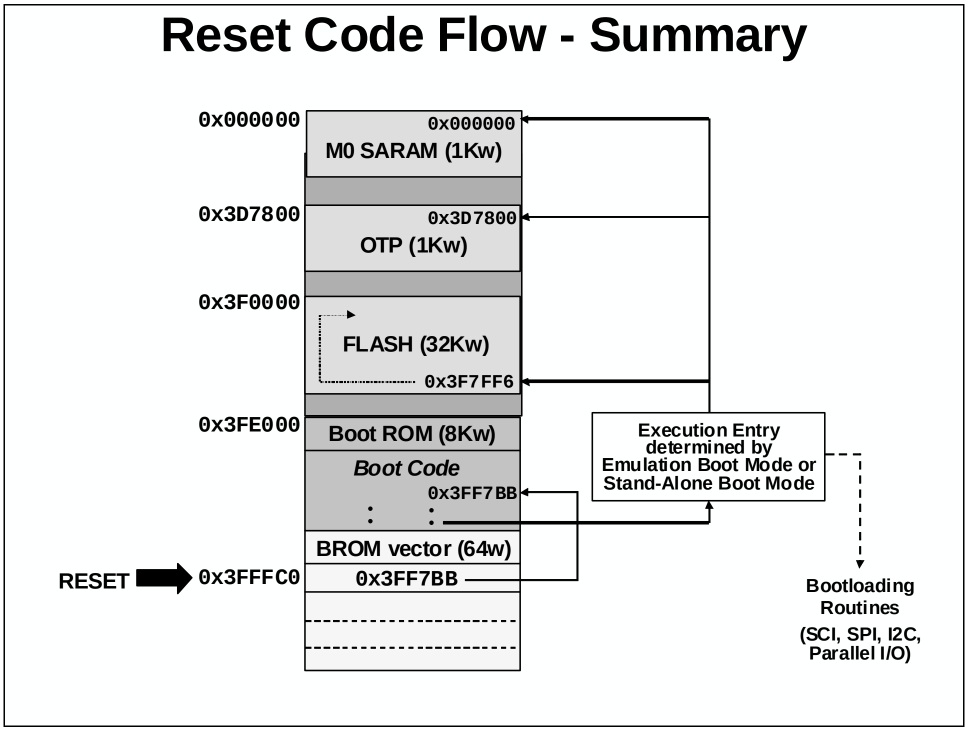
EMU_KEY & EMU_BMODE located in PIE at 0x0D00 & 0x0D01, respectively
OTP_KEY & OTP_BMODE located in OTP at 0x3D78FE & 0x3D78FF, respectively

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

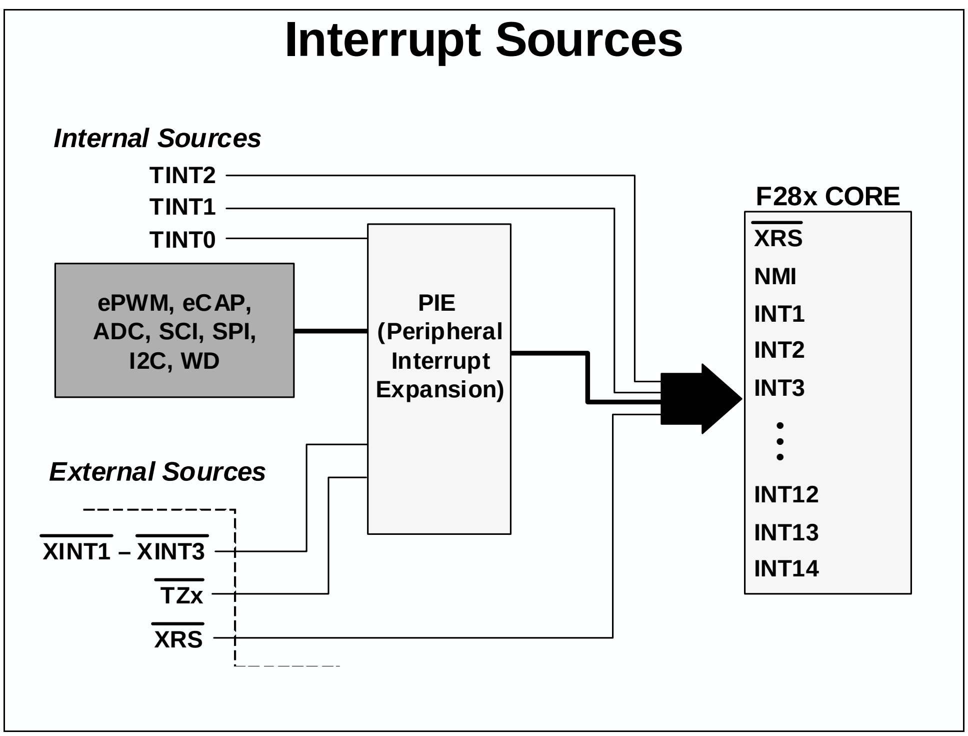


Stand-Alone Boot Mode ($\overline{\text{TRST}}$ = 0)



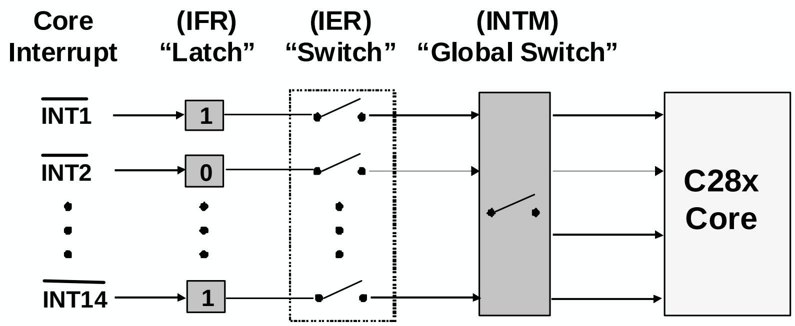

Reset Code Flow - Summary
0x000000
0x000000
M0 SARAM (1Kw)
0x3D7800
0x3D7800
OTP (1Kw)
0x3F0000
FLASH (32Kw)
0x3F7FF6
0x3FE000
Boot ROM (8Kw)
Boot Code
0x3FF7BB
Execution Entry determined by Emulation Boot Mode or Stand-Alone Boot Mode
BROM vector (64w)
RESET
0x3FFFC0
0x3FF7BB
Bootloading Routines
(SCI, SPI, I2C, Parallel I/O)

Interrupts


Interrupt Sources
Internal Sources
TINT2
TINT1
TINT0
ePWM, eCAP, ADC, SCI, SPI, I2C, WD
PIE (Peripheral Interrupt Expansion)
F28x CORE
XRS
NMI
INT1
INT2
INT3
INT12
INT13
INT14
External Sources
XINT1 – XINT3
TZx
XRS

Maskable Interrupt Processing

Conceptual Core Overview



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

Core Interrupt Registers

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

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

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

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

Interrupt Global Mask Bit (INTM)

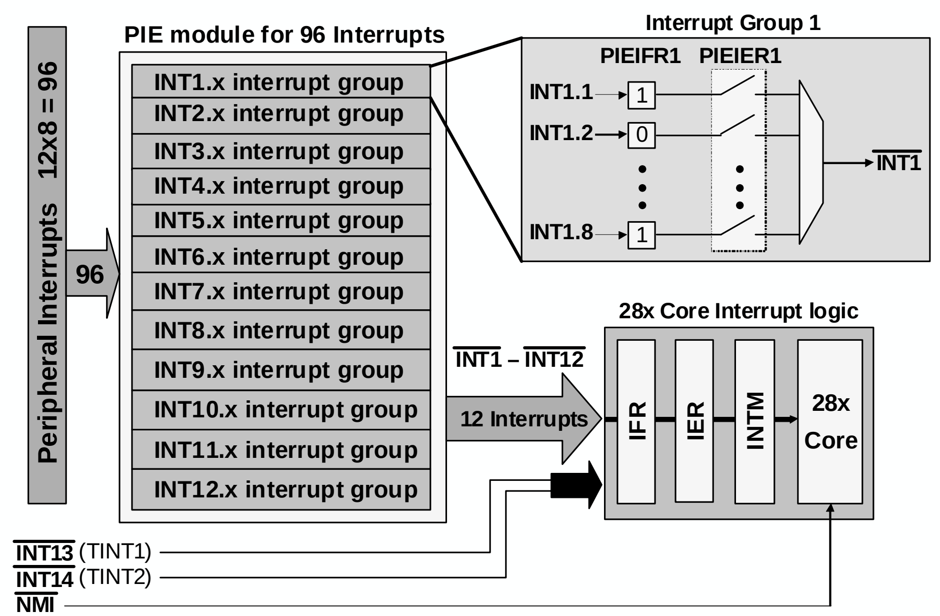
		Bit 0
ST1		INTM

(enable = 0 / disable = 1)

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

Peripheral Interrupt Expansion (PIE)

Peripheral Interrupt Expansion - PIE



F2802x PIE Interrupt Assignment Table

	INTx.8	INTx.7	INTx.6	INTx.5	INTx.4	INTx.3	INTx.2	INTx.1
INT1	WAKEINT	TINT0	ADCINT9	XINT2	XINT1		ADCINT2	ADCINT1
INT2					EPWM4 _TZINT	EPWM3 _TZINT	EPWM2 _TZINT	EPWM1 _TZINT
INT3					EPWM4 _INT	EPWM3 _INT	EPWM2 _INT	EPWM1 _INT
INT4								ECAP1 _INT
INT5								
INT6							SPITX INTA	SPIRX INTA
INT7								
INT8							I2CINT2A	I2CINT1A
INT9							SCITX INTA	SCIRX INTA
INT10	ADCINT8	ADCINT7	ADCINT6	ADCINT5	ADCINT4	ADCINT3	ADCINT2	ADCINT1
INT11								
INT12								XINT3

PIE Registers

PIEIFRx register (x = 1 to 12)

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

PIEIERx register (x = 1 to 12)

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

PIE Interrupt Acknowledge Register (PIEACK)

15 - 12	11 - 0
reserved	PIEACKx

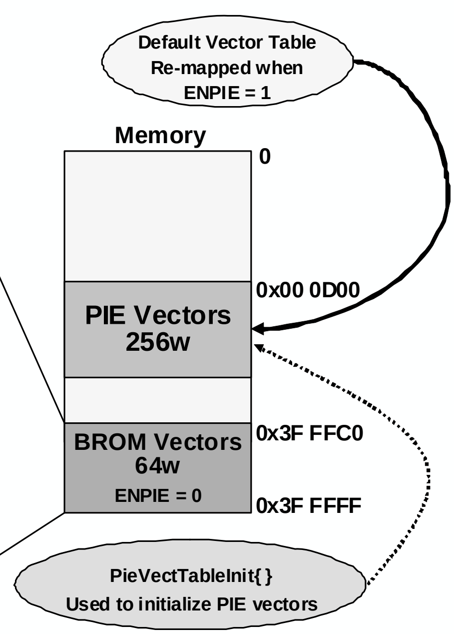
PIECTRL register

15 - 1	0
PIEVECT	ENPIE

```
#include "DSP2802x_Device.h"
   PieCtrlRegs.PIEIFR1.bit.INTx4 = 1;    //manually set IFR for XINT1 in PIE group 1
   PieCtrlRegs.PIEIER3.bit.INTx2 = 1;    //enable EPWM2_INT in PIE group 3
   PieCtrlRegs.PIEACK.all = 0x0004;      //acknowledge the PIE group 3
   PieCtrlRegs.PIECTRL.bit.ENPIE = 1;  //enable the PIE
```

Default Interrupt Vector Table at Reset

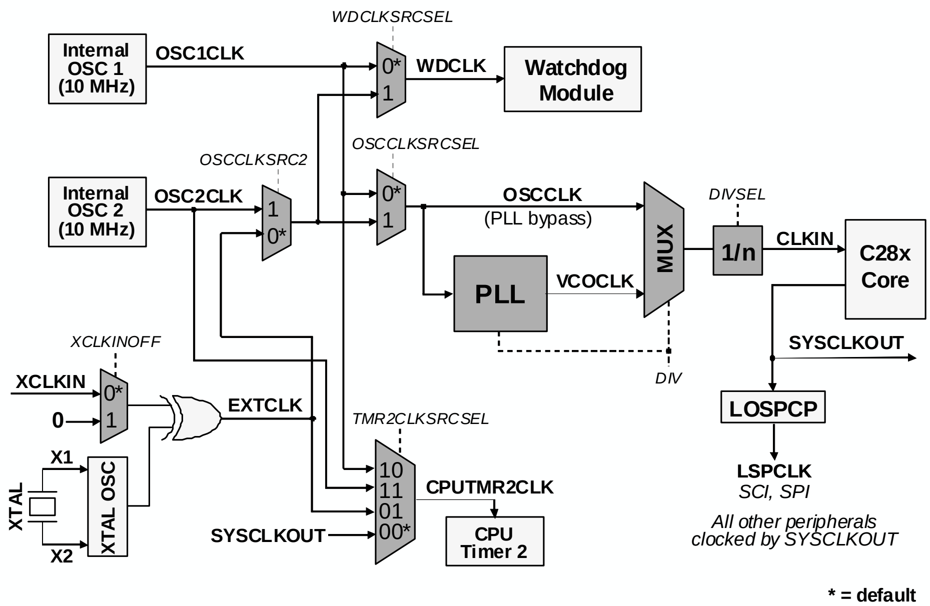
Vector	Offset
RESET	00
INT1	02
INT2	04
INT3	06
INT4	08
INT5	0A
INT6	0C
INT7	0E
INT8	10
INT9	12
INT10	14
INT11	16
INT12	18
INT13	1A
INT14	1C
DATALOG	1E
RTOSINT	20
EMUINT	22
NMI	24
ILLEGAL	26
USER 1-12	28-3E



Oscillator / PLL Clock Module

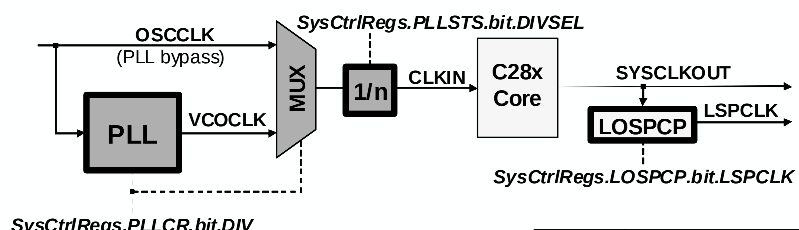
F2802x Oscillator / PLL Clock Module

(lab file: SysCtrl.c)



F2802x PLL and LOSPCP

(lab file: SysCtrl.c)



DIV	CLKIN
0 0 0 0	OSCCLK / n * (PLL bypass)
0 0 0 1	OSCCLK x 1 / n
0 0 1 0	OSCCLK x 2 / n
0 0 1 1	OSCCLK x 3 / n
0 1 0 0	OSCCLK x 4 / n
0 1 0 1	OSCCLK x 5 / n
0 1 1 0	OSCCLK x 6 / n
0 1 1 1	OSCCLK x 7 / n
1 0 0 0	OSCCLK x 8 / n
1 0 0 1	OSCCLK x 9 / n
1 0 1 0	OSCCLK x 10 / n
1 0 1 1	OSCCLK x 11 / n
1 1 0 0	OSCCLK x 12 / n

DIVSEL	n
0x	/4 *
10	/2
11	/1

*** default**

Note: /1 mode can only be used when PLL is bypassed

LSPCLK	Peripheral Clk Freq
0 0 0	SYSCLKOUT / 1
0 0 1	SYSCLKOUT / 2
0 1 0	SYSCLKOUT / 4 *
0 1 1	SYSCLKOUT / 6
1 0 0	SYSCLKOUT / 8
1 0 1	SYSCLKOUT / 10
1 1 0	SYSCLKOUT / 12
1 1 1	SYSCLKOUT / 14

Input Clock Fail Detect Circuitry

PLL will issue a "limp mode" clock (1-4 MHz) if input clock is removed after PLL has locked.

An internal device reset will also be issued (XRSn pin not driven).

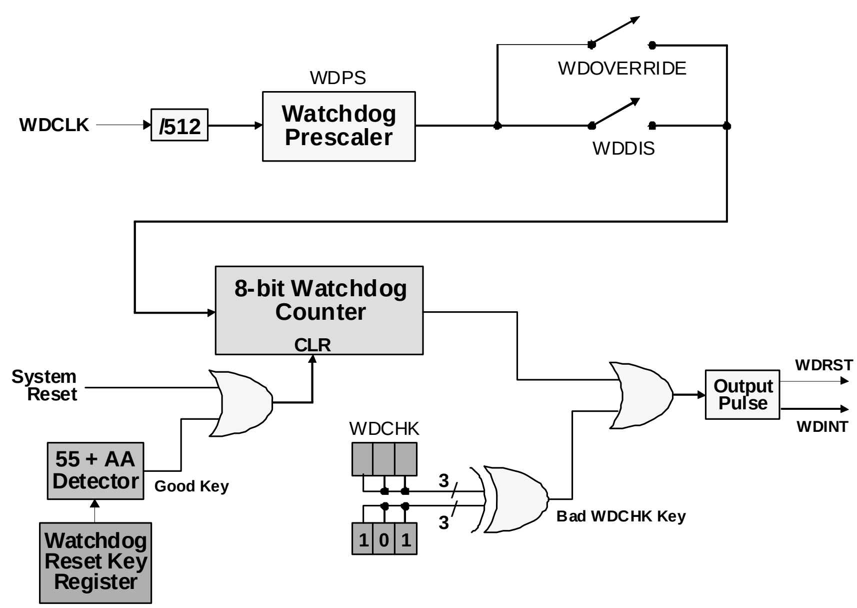
Watchdog Timer Module

Watchdog Timer

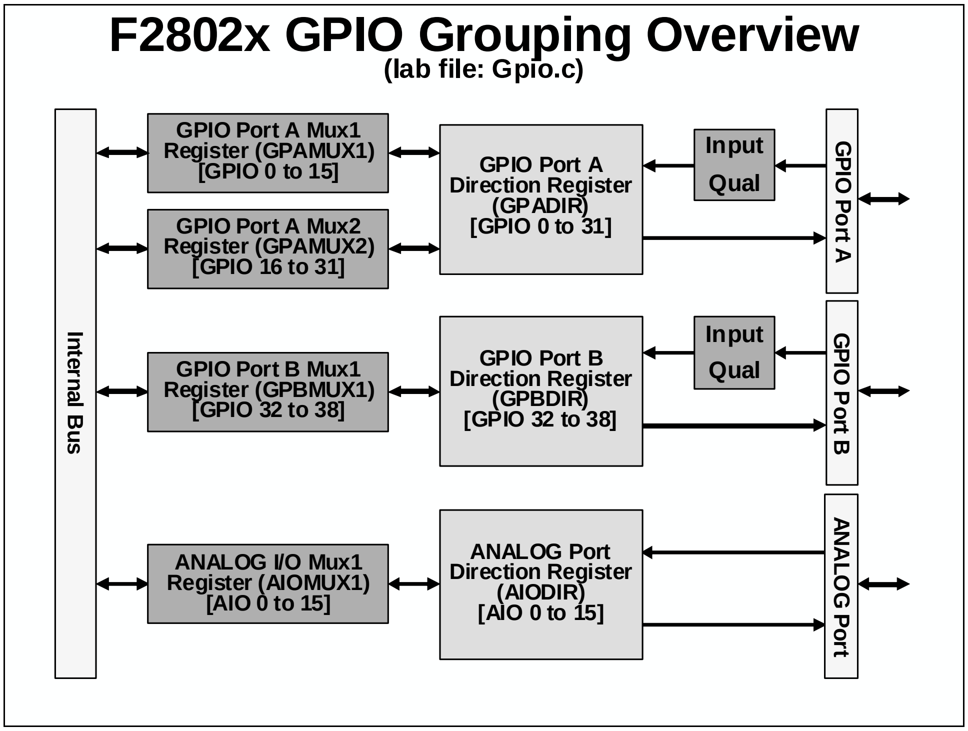
- Resets the C28x if the CPU crashes
  - Watchdog counter runs independent of CPU
  - If counter overflows, a reset or interrupt is triggered (user selectable)
  - CPU must write correct data key sequence to reset the counter before overflow
- Watchdog must be serviced or disabled within 131,072 WDCLK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

Watchdog Timer Module

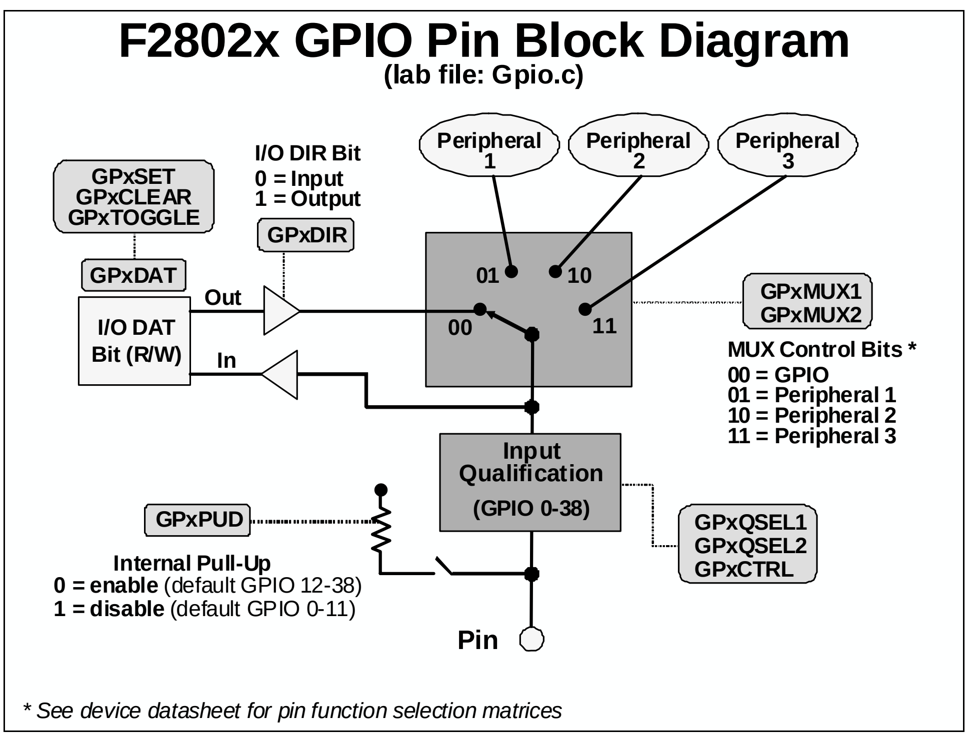
(lab file: Watchdog.c)



GPIO

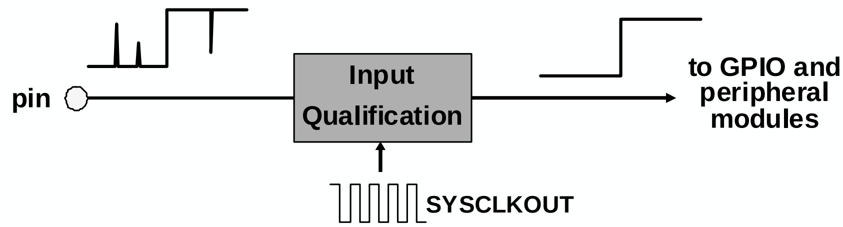


F2802x GPIO Grouping Overview
(lab file: Gpio.c)
Internal Bus
GPIO Port A Mux1 Register (GPAMUX1) [GPIO 0 to 15]
GPIO Port A Mux2 Register (GPAMUX2) [GPIO 16 to 31]
GPIO Port A Direction Register (GPADIR) [GPIO 0 to 31]
Input Qual
GPIO Port A
GPIO Port B Mux1 Register (GPBMUX1) [GPIO 32 to 38]
GPIO Port B Direction Register (GPBDIR) [GPIO 32 to 38]
Input Qual
GPIO Port B
ANALOG I/O Mux1 Register (AIOMUX1) [AIO 0 to 15]
ANALOG Port Direction Register (AIODIR) [AIO 0 to 15]
ANALOG Port

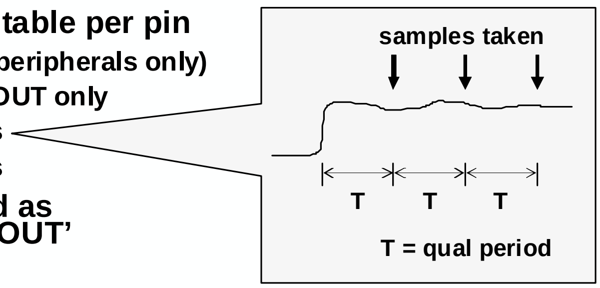


F2802x GPIO Pin Block Diagram
(lab file: Gpio.c)
GPxSET
GPxCLEAR
GPxTOGGLE
GPxDAT
I/O DAT Bit (R/W)
Out
In
I/O DIR Bit
0 = Input
1 = Output
GPxDIR
Peripheral 1
Peripheral 2
Peripheral 3
01
10
00
11
GPxMUX1
GPxMUX2
MUX Control Bits *
00 = GPIO
01 = Peripheral 1
10 = Peripheral 2
11 = Peripheral 3
Input Qualification
(GPIO 0-38)
GPxQSEL1
GPxQSEL2
GPxCTRL
GPxPUD
Internal Pull-Up
0 = enable (default GPIO 12-38)
1 = disable (default GPIO 0-11)
Pin
* See device datasheet for pin function selection matrices

F2802x GPIO Input Qualification



- **Qualification available on ports A & B (GPIO 0 - 38) only**
- **Individually selectable per pin**
  - **no qualification (peripherals only)**
  - **sync to SYSCLKOUT only**
  - **qualify 3 samples**
  - **qualify 6 samples**
- **AIO pins are fixed as ‘sync to SYSCLKOUT’**



Lab 2: System Initialization

- LAB2 files have been provided
- LAB2 consists of two parts:

  <u>Part 1</u>
  - Test behavior of watchdog when disabled and enabled

  <u>Part 2</u>
  - Initialize peripheral interrupt expansion (PIE) vectors and use watchdog to generate an interrupt
- Modify, build, and test code using Code Composer Studio

Lab 2: System Initialization

➢ **Objective**

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

- Setup the clock module – PLL, LOSPCP = /4, low-power modes to default values, enable all module clocks
- Disable the watchdog – clear WD flag, disable watchdog, WD prescale = 1
- Setup watchdog system control register – DO NOT clear WD OVERRIDE bit, WD generate a CPU 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 be tested by using the watchdog to generate an interrupt. This lab will make use of the DSP2802x C-code header files to simplify the programming of the device, as well as take care of the register definitions and addresses. Please review these files, and make use of them in the future, as needed.

➢ **Procedure**

Open the Project

1. A project named `Lab2` has been created for this lab. Open the project by clicking on `Project → Import Existing CCS/CCE Eclipse Project`. The “Import” window will open then click `Browse`… next to the “Select root directory” box. Navigate to: `C:\C28x\Labs\Lab2\Project` and click `OK`. Then click `Finish` to import the project.

2. In the `C/C++ Projects` window, click the plus sign (+) to the left of `Lab2` to view the project files. All Build Options have been configured for this lab. The files used in this lab are:

```
CodeStartBranch.asm                    Lab_2_3.cmd
DefaultIsr_2.c                         Main_2.c
DelayUs.asm                            PieCtrl.c
DSP2802x_GlobalVariableDefs.c          PieVect.c
DSP2802x_Headers_nonBIOS.cmd           SysCtrl.c
Gpio.c                                 Watchdog.c
```

Modified Memory Configuration

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

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

System Initialization

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

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

6. 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

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

8. Click the “`Debug`” button (green bug). The “Debug Perspective” view should open, the program will load automatically, and you should now be at the start of `main()`.

9. After CCS loaded the program in the previous step, it set the program counter (PC) to point to `_c_int00`. It then ran through the C-environment initialization routine in the `rts2800_ml.lib` and stopped at the start of `main()`. CCS did not do a device reset, and as a result the bootloader was bypassed.

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

   `Scripts → EMU Boot Mode Select → EMU_BOOT_SARAM`

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

Run the Code – Watchdog Reset

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 Line`. This is the same as setting a breakpoint on the selected line, running to that breakpoint, and then removing the breakpoint.

11. Place the cursor on the first line of code in `main()` and set a breakpoint by right clicking the mouse key and select `Toggle Breakpoint`. Notice that line is highlighted with a blue dot indicating that the breakpoint has been set. Alternately, you can double-click in the line number 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.

12. Run your code for a few seconds by using the `Run` button on the toolbar, or using `Target → Run` on the menu bar. After a few seconds halt your code by using the `Halt` button on the toolbar, or by using `Target → Halt`. Where did your code stop? Are the results as expected? If things went as expected, your code should be in the "`main loop`".

13. Switch to the "C/C++ Perspective" view by clicking the `C/C++` icon in the upper right-hand corner. Modify the `InitWatchdog()` function to enable the watchdog (WDCR). In `Watchdog.c` change the `WDCR` register value to **`0x00A8`**. This will enable the watchdog to function and cause a reset. Save the file.

14. Click the "`Build`" button. Select `Yes` to "Reload the program automatically". Switch back to the "Debug Perspective" view by clicking the `Debug` icon in the upper right-hand corner.

15. Like before, place the cursor in the "`main loop`" section (on the `asm(" NOP");` instruction line) and right click the mouse key and select `Run To Line`.

16. Run your code. Where did your code stop? Are the results as expected? If things went as expected, your code should have stopped at the breakpoint. What happened is as follows. While the code was running, the watchdog timed out and reset the processor. The reset vector was then fetched and the ROM bootloader began execution. Since the device is in emulation boot mode (i.e. the emulator is connected) the bootloader read the EMU_KEY and EMU_BMODE values from the PIE RAM. These values were previously set for boot to M0 SARAM bootmode by CCS. Since these values did not change and are not affected by reset, the bootloader transferred execution to the beginning of our code at address 0x000000 in the M0SARAM, and execution continued until the breakpoint was hit in main( ).

Setup PIE Vector for Watchdog Interrupt

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

17. Switch to the "C/C++ Perspective" view by clicking the `C/C++` icon in the upper right-hand corner. Notice that the following files are included in the project:
    ```
    DefaultIsr_2.c
    PieCtrl.c
    PieVect.c
    ```

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

19. 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:__________

20. 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).

21. Modify the system control and status register `(SCSR)` to cause the watchdog to generate a WAKEINT rather than a reset. In `Watchdog.c` change the `SCSR` register value to **`0x0002`**. Save the modified files.

22. 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.

23. 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

24. Click the "`Build`" button and select `Yes` to "Reload the program automatically". Switch to the "Debug Perspective" view by clicking the `Debug` icon in the upper right-hand corner.

Run the Code – Watchdog Interrupt

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

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

Terminate Debug Session and Close Project

27. Terminate the active debug session using the `Terminate All` button. This will close the debugger and return CCS to the "C/C++ Perspective" view.

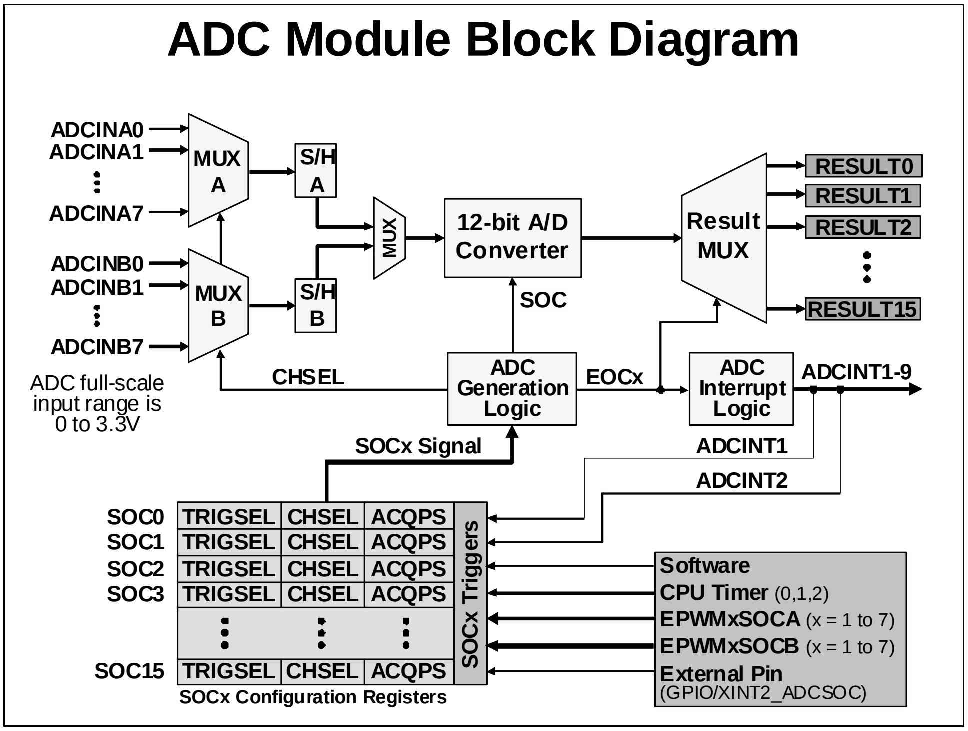
28. Next, close the project by right-clicking on `Lab2` in the `C/C++ Projects` window and select `Close Project`.

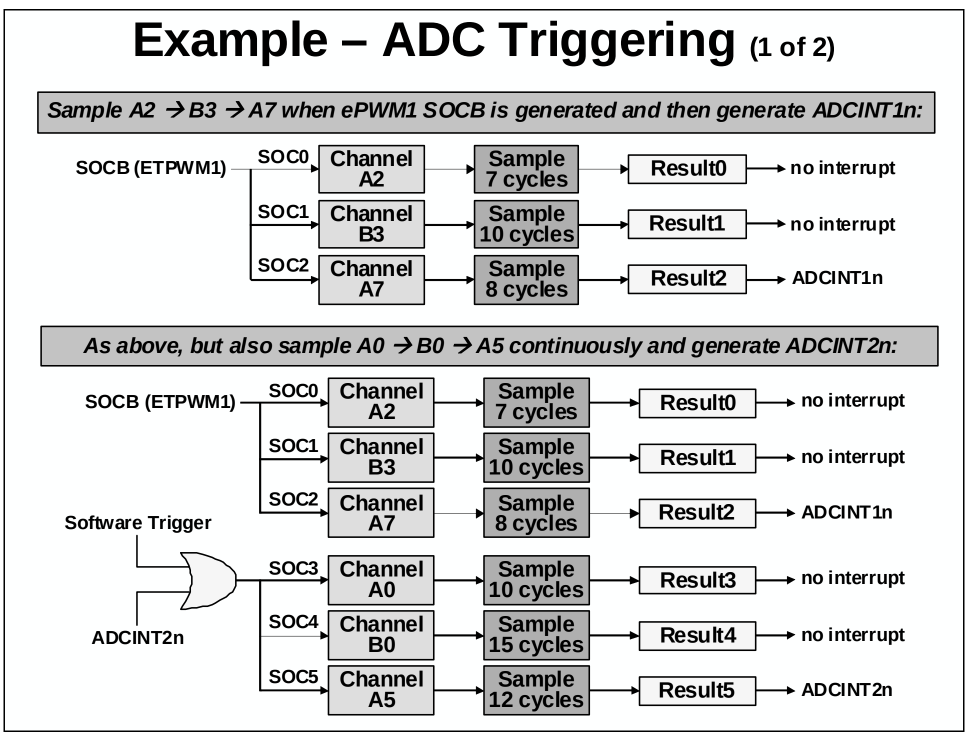
End of Exercise

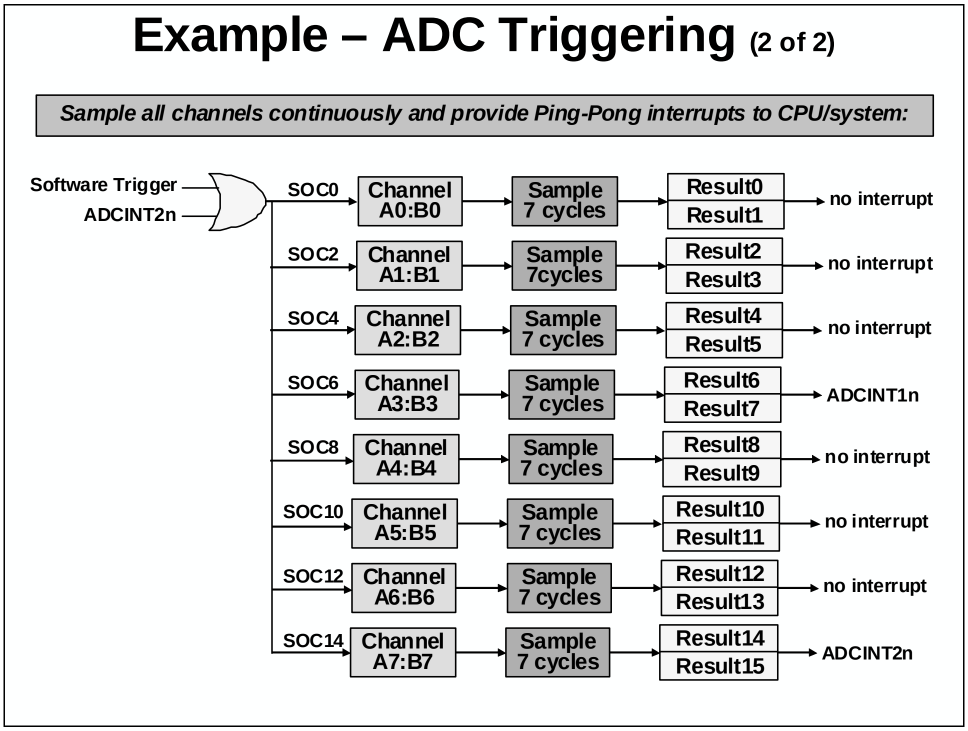
Note: By default, the watchdog timer is enabled out of reset. Code in the file `CodeStartBranch.asm` has been configured to disable the watchdog. This can be important for large C code projects (ask your instructor if this has not already been explained). During this lab exercise, the watchdog was actually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

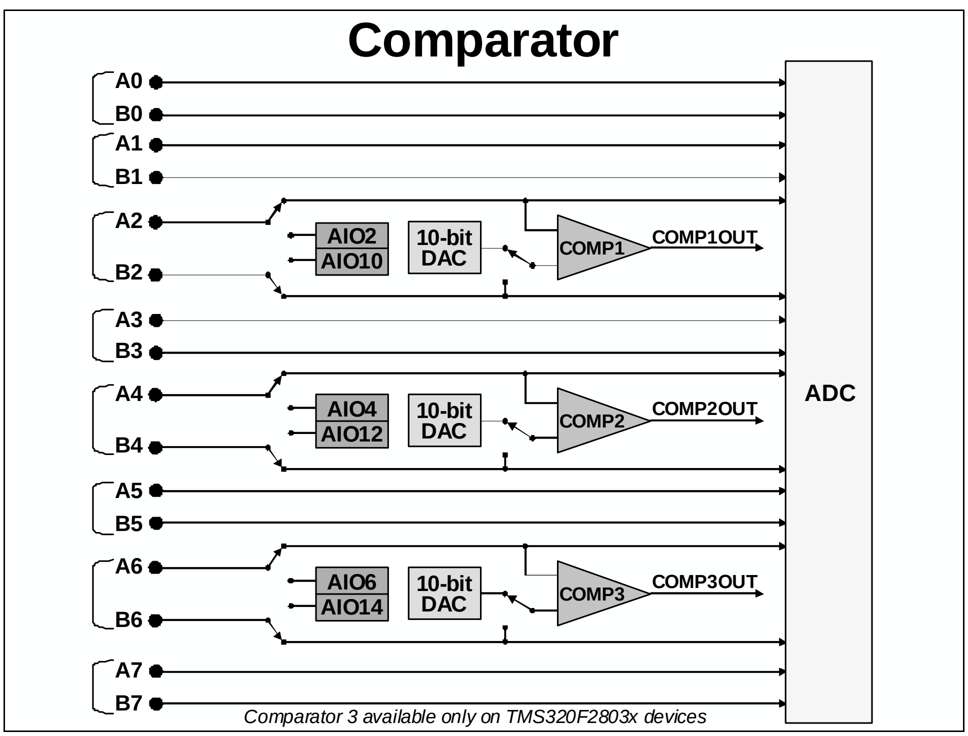
Control Peripherals

ADC Module






Example – ADC Triggering (2 of 2)
Sample all channels continuously and provide Ping-Pong interrupts to CPU/system:
Software Trigger
ADCINT2n
SOC0
SOC2
SOC4
SOC6
SOC8
SOC10
SOC12
SOC14
Channel A0:B0
Channel A1:B1
Channel A2:B2
Channel A3:B3
Channel A4:B4
Channel A5:B5
Channel A6:B6
Channel A7:B7
Sample 7 cycles
Sample 7cycles
Result0
Result1
Result2
Result3
Result4
Result5
Result6
Result7
Result8
Result9
Result10
Result11
Result12
Result13
Result14
Result15
no interrupt
ADCINT1n
ADCINT2n


Comparator
A0
B0
A1
B1
A2
B2
A3
B3
A4
B4
A5
B5
A6
B6
A7
B7
AIO2
AIO10
AIO4
AIO12
AIO6
AIO14
10-bit DAC
COMP1
COMP2
COMP3
COMP1OUT
COMP2OUT
COMP3OUT
ADC
Comparator 3 available only on TMS320F2803x devices

ADC Control Registers (file: Adc.c)

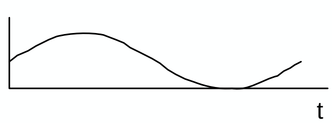
- ADCTRL1 (ADC Control Register 1)
  - module reset, ADC enable
  - busy/busy channel
  - reference select
  - Interrupt generation control
- ADCSOCxCTL (SOC0 to SOC15 Control Registers)
  - trigger source
  - channel
  - acquisition sampling window
- ADCINTSOCSELx (Interrupt SOC Selection 1 and 2 Registers)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

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

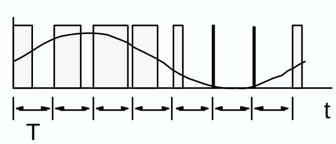
Pulse Width Modulation

What is Pulse Width Modulation?

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



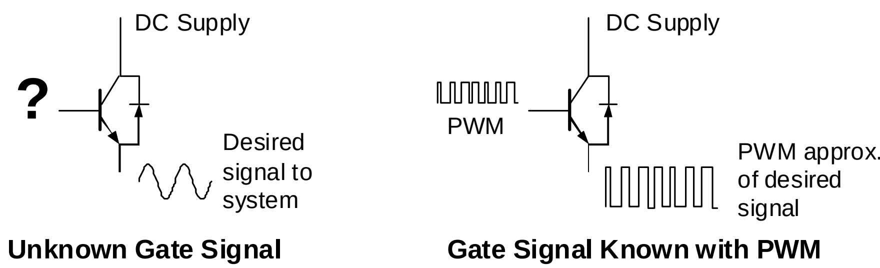
Original Signal



PWM representation

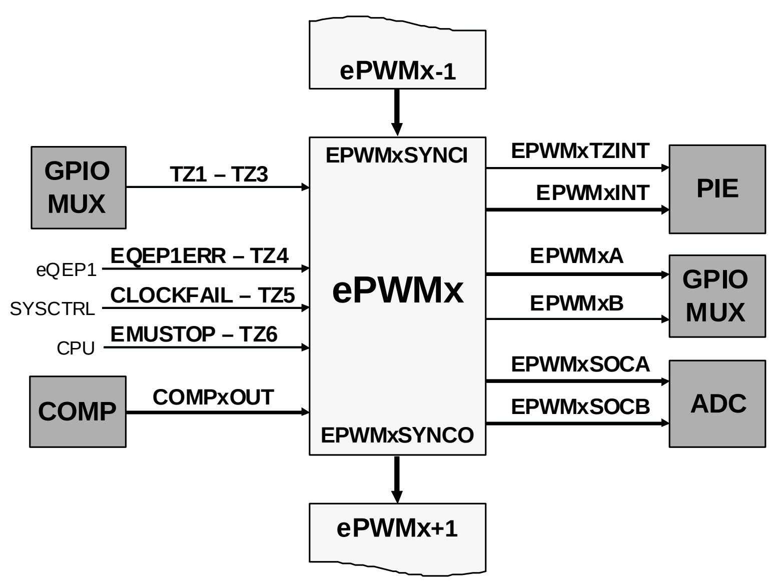
Why use PWM with Power Switching Devices?

- 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

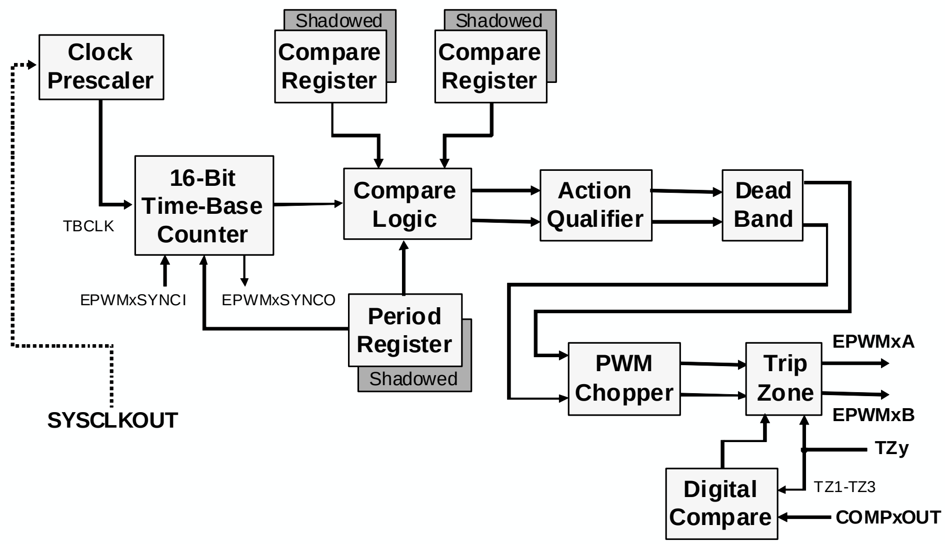


ePWM

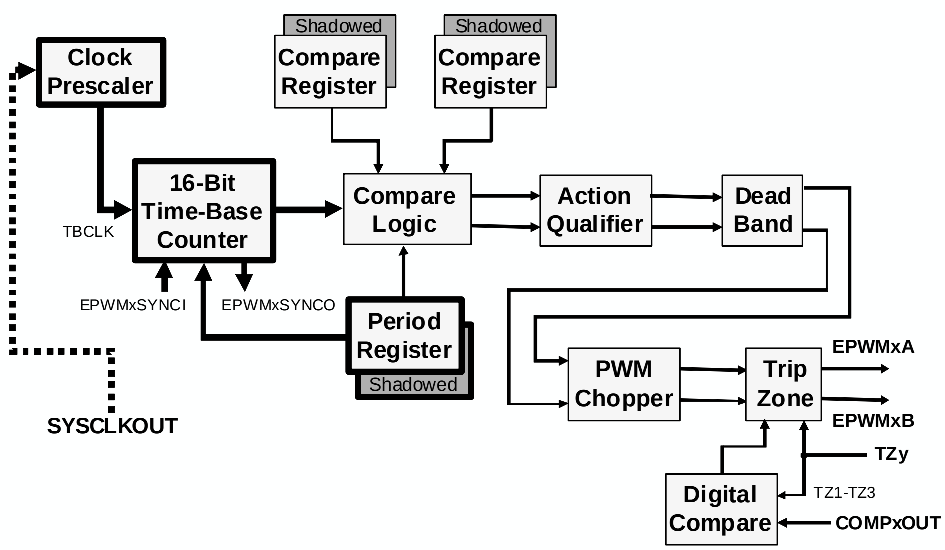
ePWM Module Signals and Connections



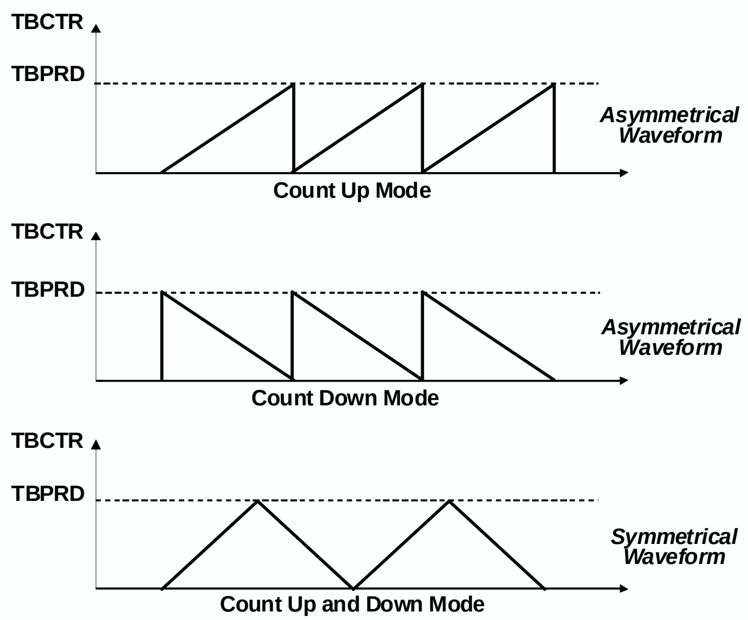
ePWM Block Diagram



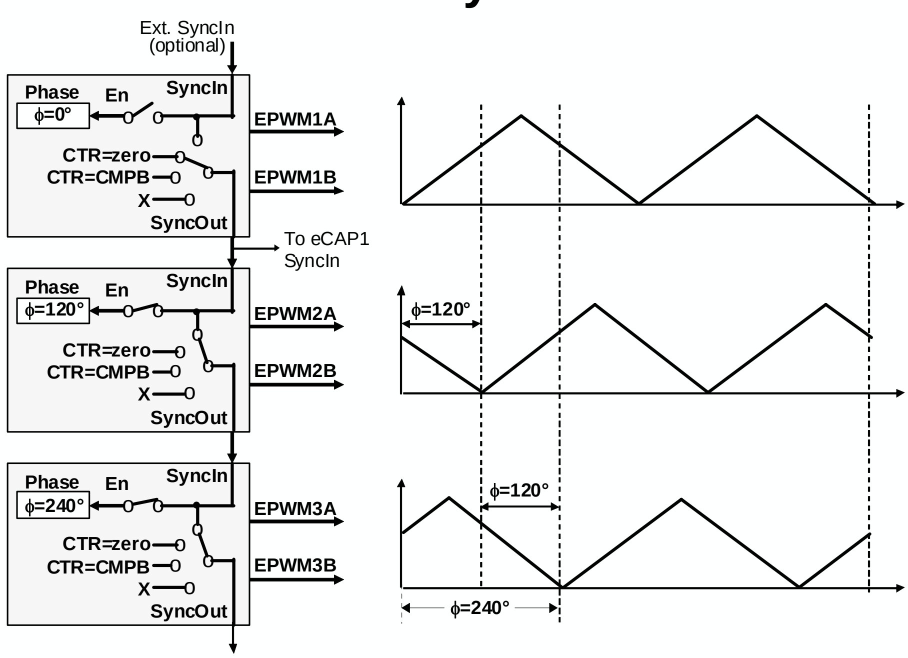
ePWM Time-Base Sub-Module



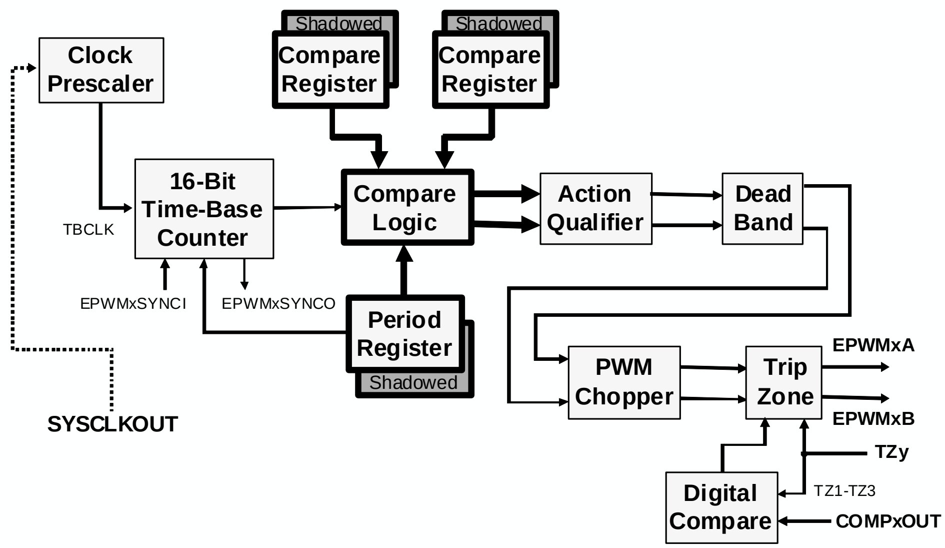
ePWM Time-Base Count Modes



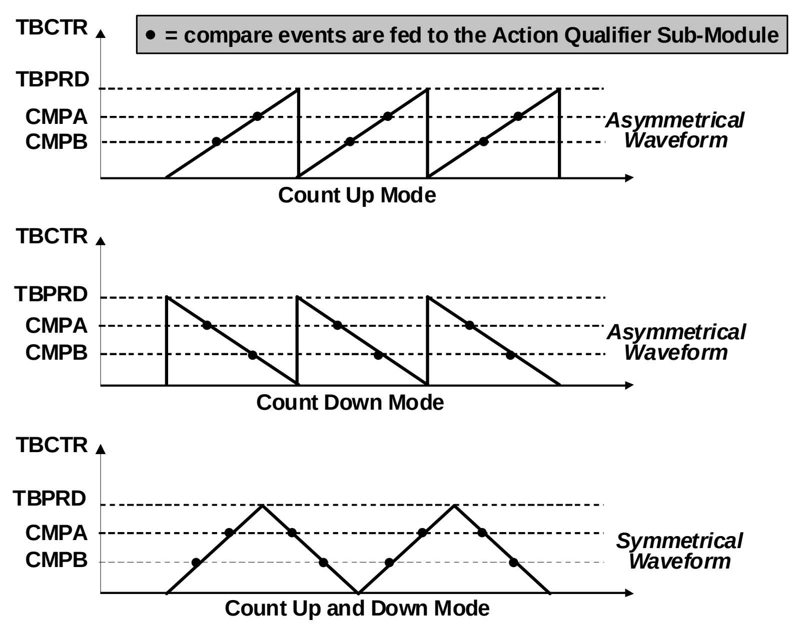
ePWM Phase Synchronization



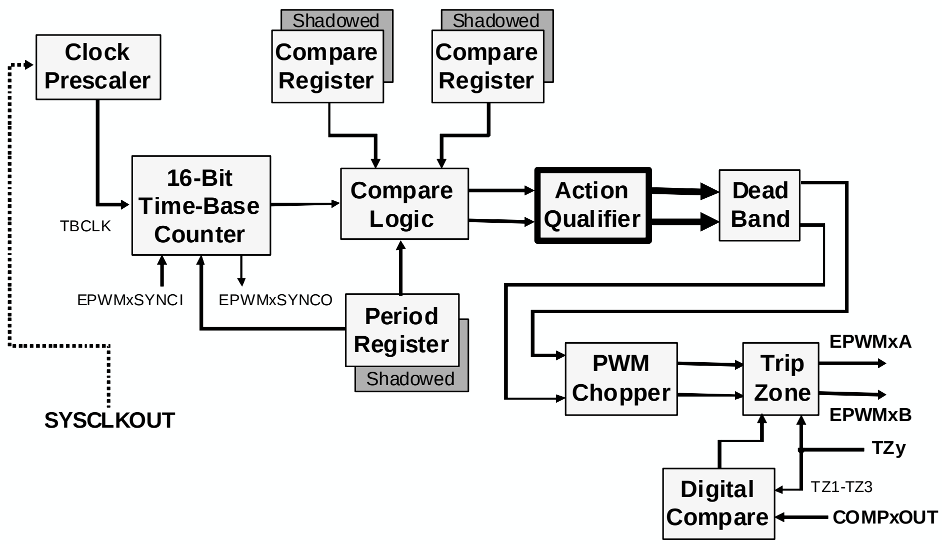
ePWM Compare Sub-Module



ePWM Compare Event Waveforms



ePWM Action Qualifier Sub-Module

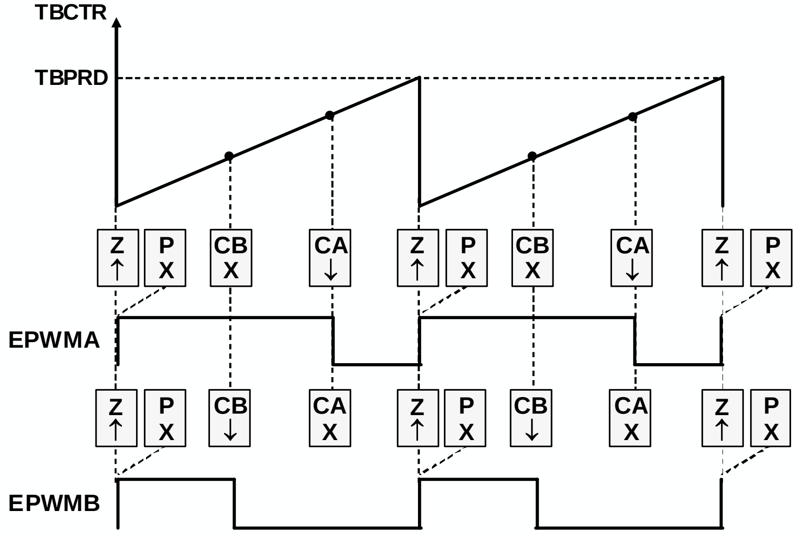


ePWM Action Qualifier Actions
for EPWMA and EPWMB

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

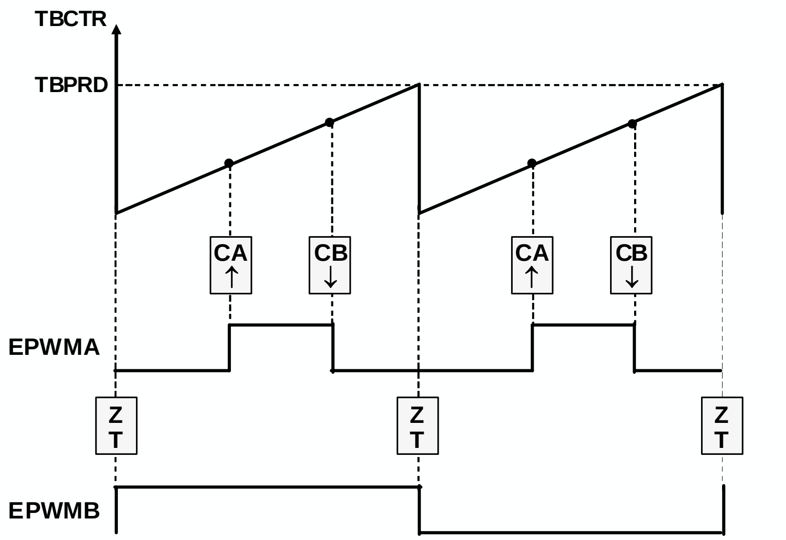
ePWM Count Up Asymmetric Waveform

with Independent Modulation on EPWMA / B



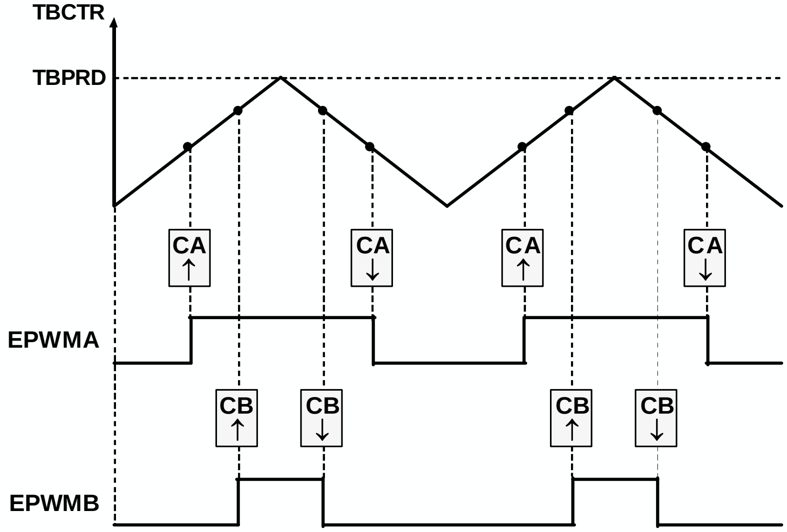
ePWM Count Up Asymmetric Waveform

with Independent Modulation on EPWMA



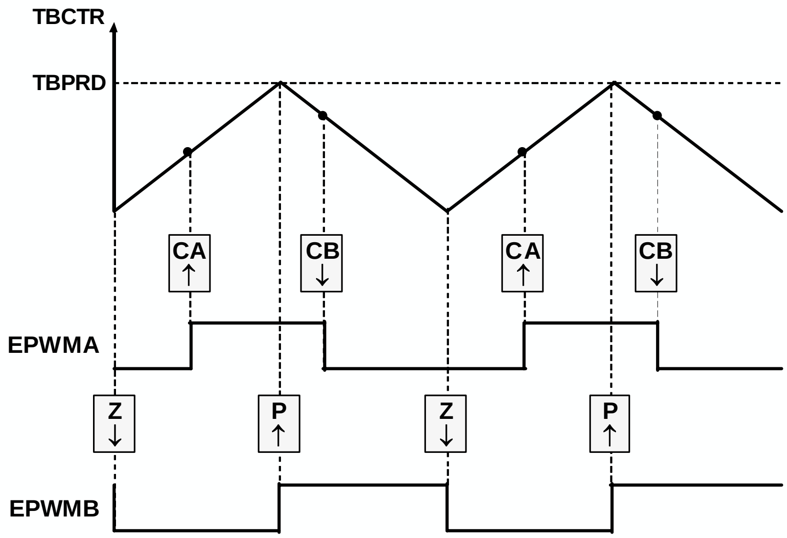
ePWM Count Up-Down Symmetric Waveform

with Independent Modulation on EPWMA / B

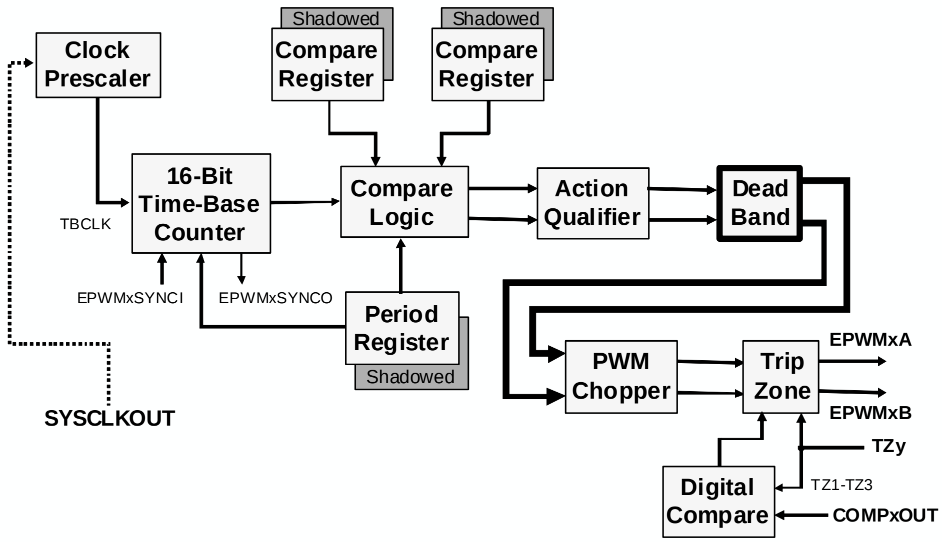


ePWM Count Up-Down Symmetric Waveform

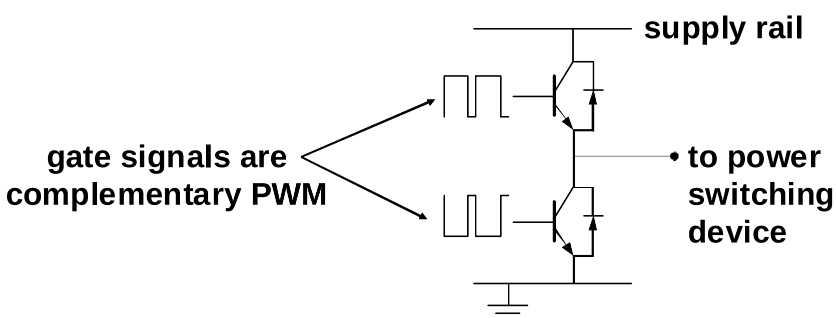
with Independent Modulation on EPWMA



ePWM Dead-Band Sub-Module

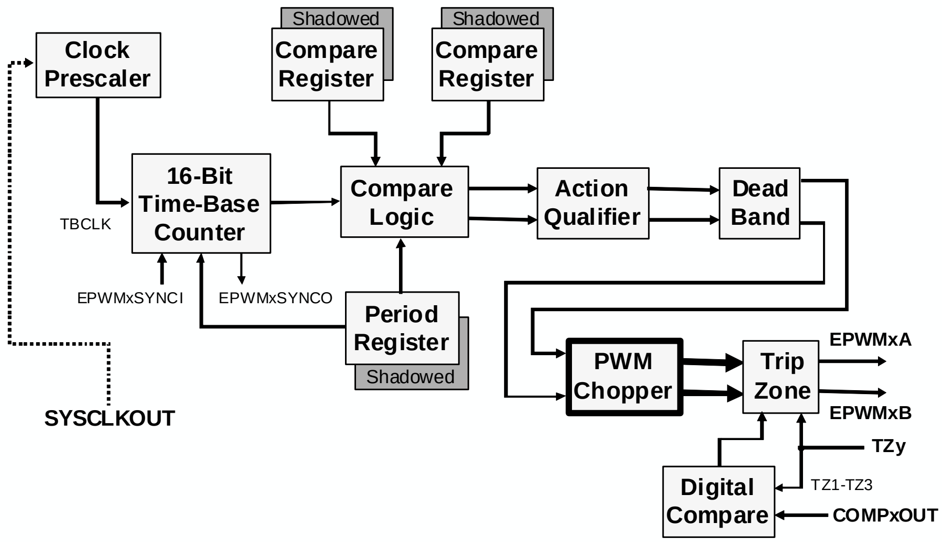


Motivation for Dead-Band



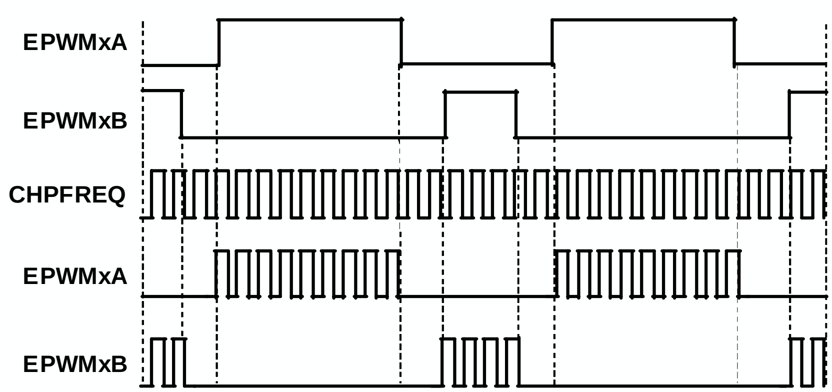
- ♦ **Transistor gates turn on faster than they shut off**
- ♦ **Short circuit if both gates are on at same time!**

ePWM PWM Chopper Sub-Module

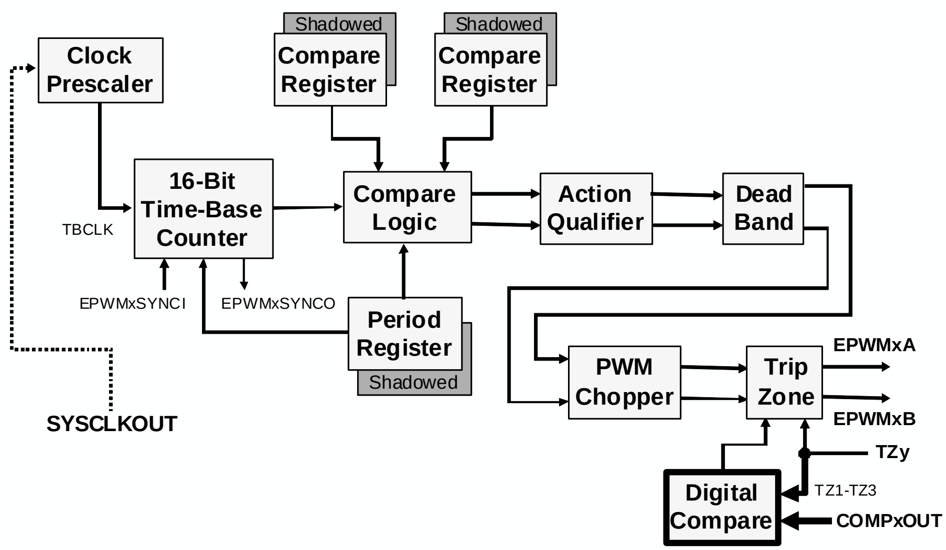


ePWM Chopper Waveform

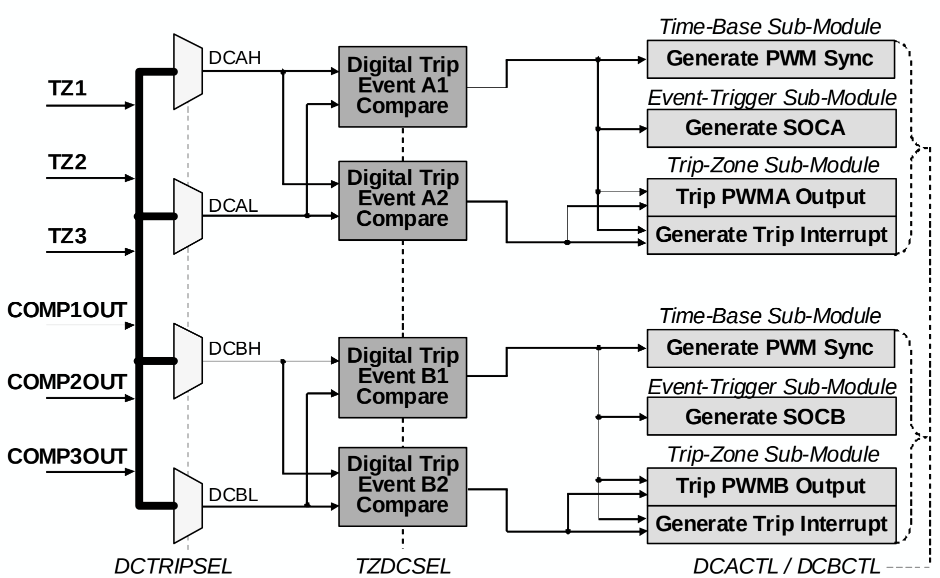
- Allows a high frequency carrier signal to modulate the PWM waveform generated by the Action Qualifier and Dead-Band modules
- Used with pulse transformer-based gate drivers to control power switching elements



ePWM Digital Compare Sub-Module

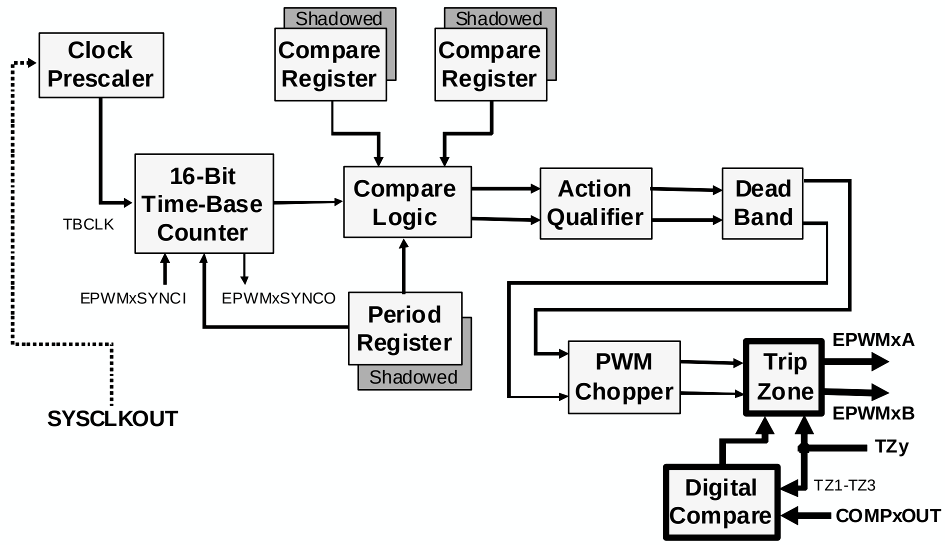


Digital Compare Sub-Module Signals



The Digital Compare sub-module compares signals external to the ePWM module to directly generate events which are then fed to the Event-Trigger, Trip-Zone, and Time-Base sub-modules

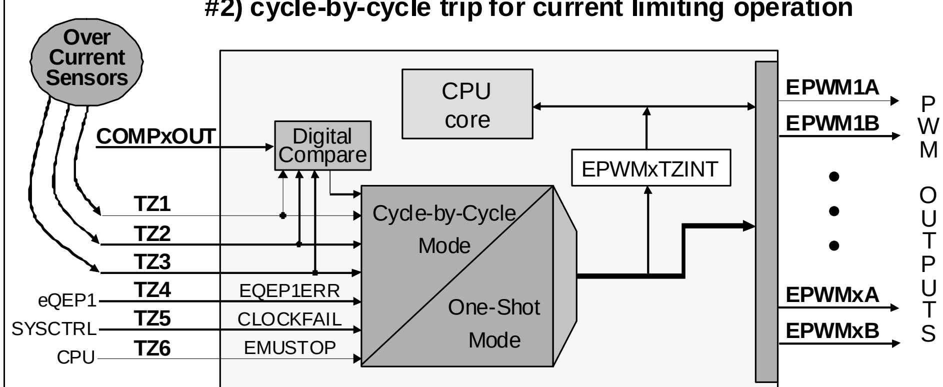
ePWM Trip-Zone Sub-Module

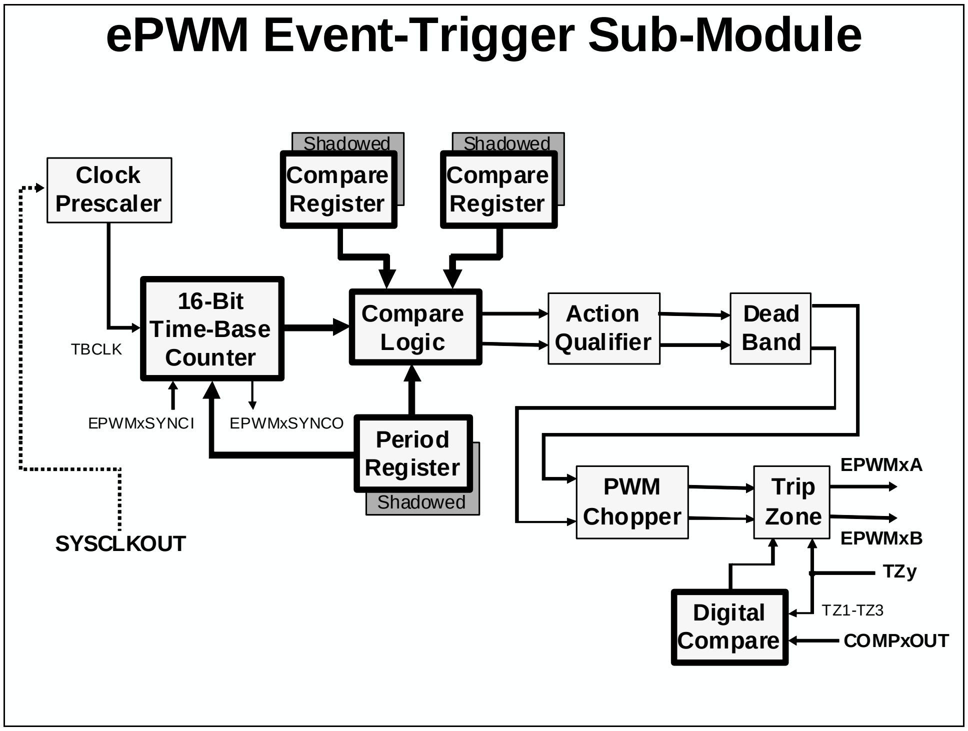


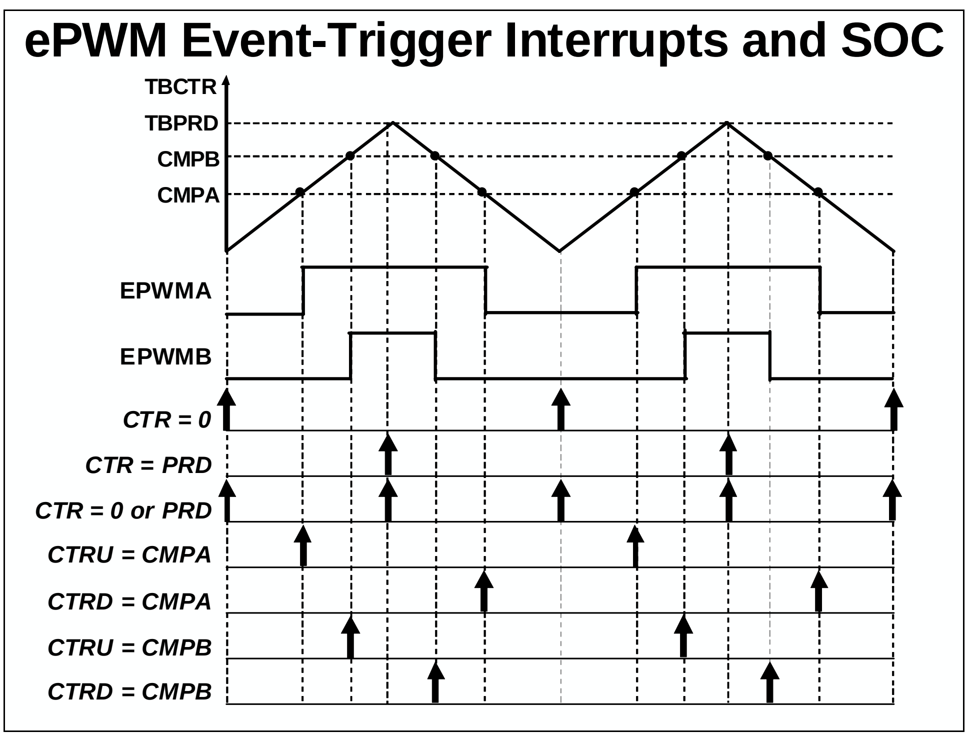
Trip-Zone Features

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

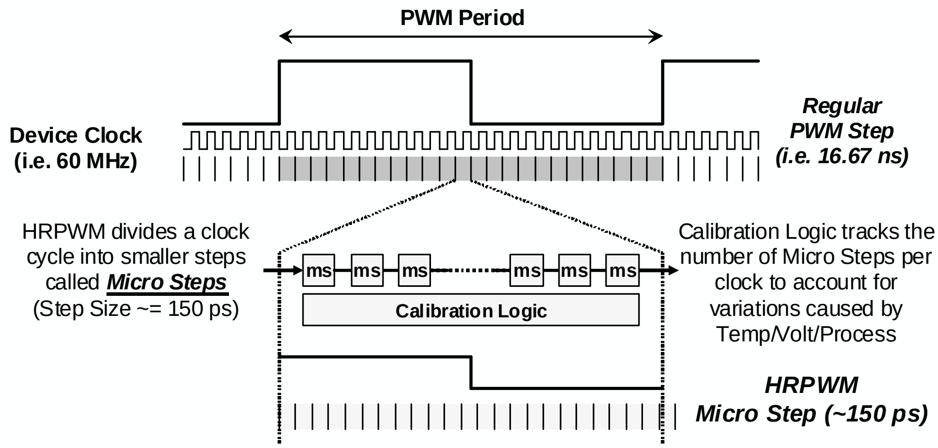
  #2) cycle-by-cycle trip for current limiting operation




ePWM Event-Trigger Sub-Module
Clock Prescaler
Shadowed
Compare Register
Shadowed
Compare Register
16-Bit Time-Base Counter
Compare Logic
Action Qualifier
Dead Band
TBCLK
EPWMxSYNCI
EPWMxSYNCO
Period Register
Shadowed
SYSCLKOUT
PWM Chopper
Trip Zone
EPWMxA
EPWMxB
TZy
Digital Compare
TZ1-TZ3
COMPxOUT


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

Hi-Resolution PWM (HRPWM)



- **Significantly increases the resolution of conventionally derived digital PWM**
- **Uses 8-bit extensions to Compare registers (CMPxHR), Period register (TBPRDHR) and Phase register (TBPHSHR) for edge positioning control**
- **Typically used when PWM resolution falls below ~9-10 bits which occurs at frequencies greater than ~120 kHz (with system clock of 60 MHz)**
- **Not all ePWM outputs support HRPWM feature (see device datasheet)**

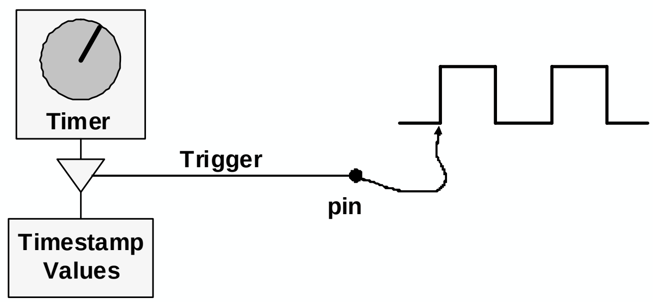
ePWM Control Registers (file: EPwm.c)

- **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**
- **DCTRIPSEL (Digital Compare Trip Select)**
  - **Digital compare A/B high/low input source select**
- **TZCTL (Trip-Zone Control)**
  - **enable /disable; action (force high / low / high-Z /nothing)**
- **ETSEL (Event-Trigger Selection)**
  - **interrupt & SOCA/B enable / disable; interrupt & SOCA/B select**

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

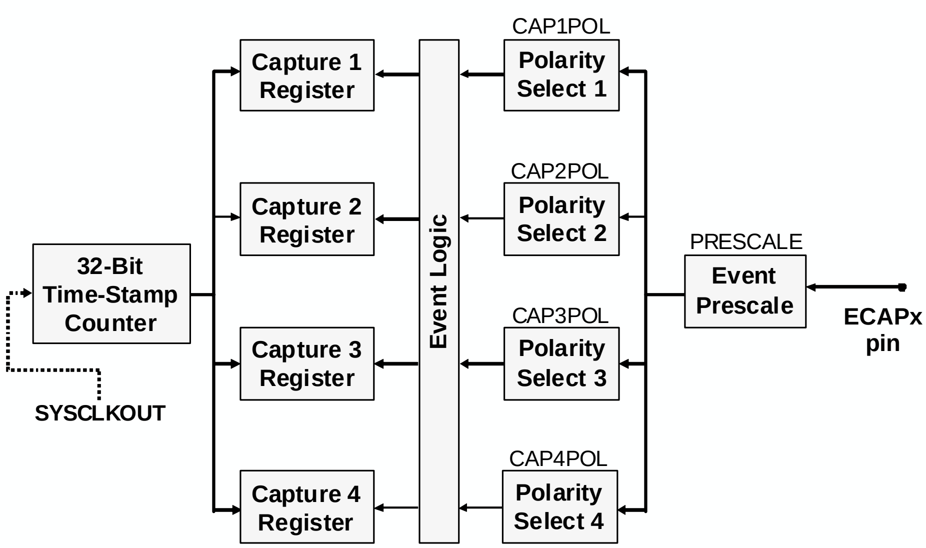
eCAP

Capture Module (eCAP)

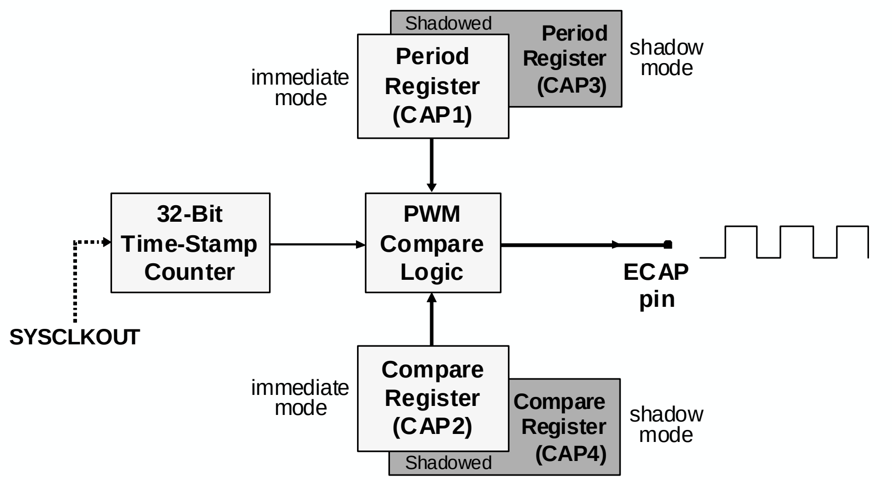


- The eCAP module timestamps transitions on a capture input pin

eCAP Module Block Diagram – Capture Mode



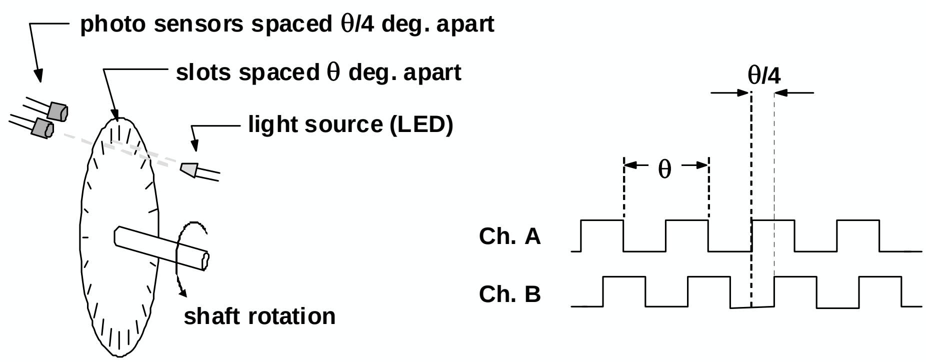
eCAP Module Block Diagram – APWM Mode



eQEP

What is an Incremental Quadrature Encoder?

A digital (angular) position sensor



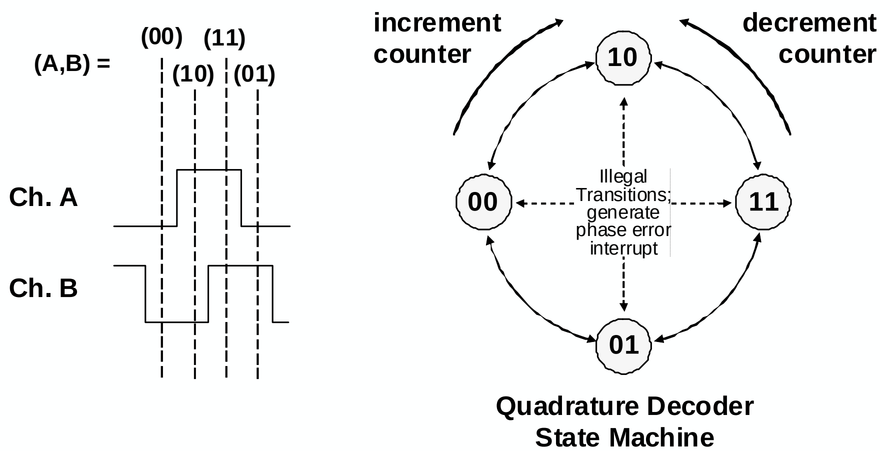
Incremental Optical Encoder

Quadrature Output from Photo Sensors

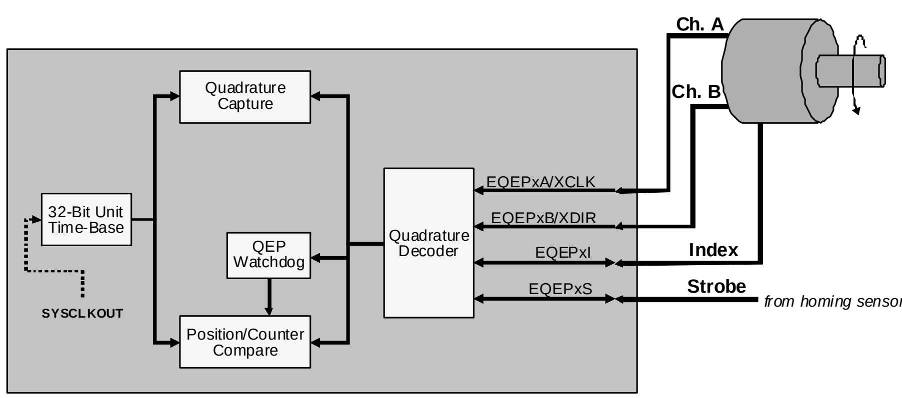
Note: eQEP available only on the TMS320F2803x devices

How is Position Determined from Quadrature Signals?

Position resolution is θ/4 degrees



eQEP Module Connections



Lab 3: Control Peripherals

➢ Objective

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

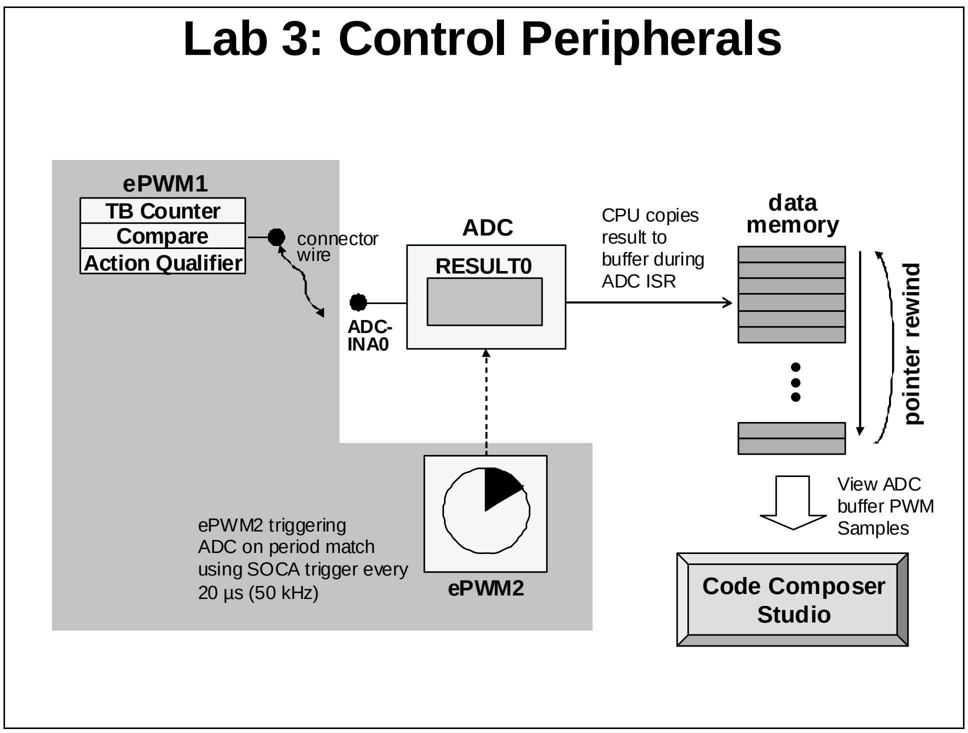
Two ePWM modules have been configured for this lab exercise:

<u>ePWM1A – PWM Generation</u>

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

<u>ePWM2 – ADC Conversion Trigger</u>

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



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

Notes

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

➢ **Procedure**

Open the Project

1. A project named `Lab3` has been created for this lab. Open the project by clicking on `Project` → `Import Existing CCS/CCE Eclipse Project`. The "Import" window will open then click `Browse`… next to the "Select root directory" box. Navigate to: `C:\C28x\Labs\Lab3\Project` and click `OK`. Then click `Finish` to import the project.

2. In the `C/C++ Projects` window, click the plus sign (+) to the left of `Lab3` to view the project files. All Build Options have been configured for this lab. The files used in this lab are:

```
Adc.c                                Gpio.c
CodeStartBranch.asm                  Lab_2_3.cmd
DefaultIsr_3_4.c                     Main_3.c
DelayUs.asm                          PieCtrl.c
DSP2802x_GlobalVariableDefs.c        PieVect.c
DSP2802x_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***

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

Build and Load

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

5. Click the "`Debug`" button (green bug). The "Debug Perspective" view should open, the program load automatically, and you should now be at the start of `Main()`. If the device has been power cycled since the last lab exercise, be sure to configure the boot mode to EMU_BOOT_SARAM using the Scripts menu.

Run the Code – PWM Waveform

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* in the "Data" memory page.

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

---

7. Using a connector wire provided, connect the PWM1A (pin # 17) to ADCINA0 (pin # 3) on the controlSTICK.

8. Run your code for a few seconds by using the `Run` button on the toolbar, or using `Target` → `Run` on the menu bar. After a few seconds halt your code by using the `Halt` button on the toolbar, or by using `Target` → `Halt`. Verify that the ADC result buffer contains the updated values.

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

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

Select `OK` to save the graph options.

10. The graphical display should show the generated 2 kHz, 25% duty cycle symmetric PWM waveform. The period of a 2 kHz signal is 500 μs. You can confirm this by measuring the period of the waveform using the "measurement marker mode" graph feature. Right-click on the graph and select `Measurement Marker Mode`. Move

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

Frequency Domain Graphing Feature of Code Composer Studio

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

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

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

    Select `OK` to save the graph options.

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

Using Real-time Emulation

Real-time emulation is a special emulation feature that allows the windows within Code Composer Studio to be updated at up to a 10 Hz rate *while the MCU is running*. This not only allows graphs and watch windows to update, but also allows the user to change values in watch or memory windows, and have those changes affect the MCU behavior. This is very useful when tuning control law parameters on-the-fly, for example.

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

    `Window` → `Preferences…`

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

- "Continuous refresh interval (milliseconds)" = 1000

Click OK.

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

14. Next we need to enable the graph window for continuous refresh. In the upper right-hand corner of the graph window, left-click on the yellow icon with the arrows rotating in a circle over a pause sign. Note when you hover your mouse over the icon, it will show "`Enable Continuous Refresh`". This will allow the graph to continuously refresh in real-time while the program is running.

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

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

    `Scripts → Realtime Emulation Control → Run_Realtime_with_Reset`

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

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

    `Scripts → Realtime Emulation Control → Full_Halt`

Terminate Debug Session and Close Project

19. Terminate the active debug session using the `Terminate All` button. This will close the debugger and return CCS to the "C/C++ Perspective" view.

20. Next, close the project by right-clicking on `Lab2` in the `C/C++ Projects` window and select `Close Project`.

Optional Exercise

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

End of Exercise

Lab Reference: Piccolo™ controlSTICK Header Pin Diagram

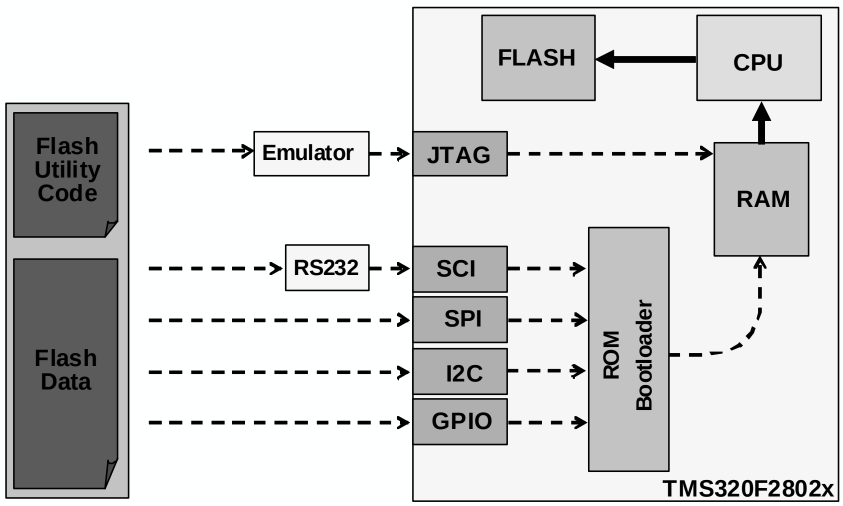
1 **ADC-A7**	**2** **ADC-A2** **COMP1 (+VE)**	**3** **ADC-A0** **Vref-HI**	**4** **3V3**
5 **ADC-A4** **COMP2 (+VE)**	**6** **ADC-B1**	**7** **EPWM-4B** **GPIO-07**	**8** **TZ1** **GPIO-12**
9 **SCL** **GPIO-33**	**10** **ADC-B6**	**11** **EPWM-4A** **GPIO-06**	**12** **ADC-A1**
13 **SDA** **GPIO-32**	**14** **ADC-B7**	**15** **EPWM-3B** **GPIO-05**	**16** **5V0**
17 **EPWM-1A** **GPIO-00**	**18** **ADC-B4** **COMP2 (-VE)**	**19** **EPWM-3A** **GPIO-04**	**20** **SPISOMI** **GPIO-17**
21 **EPWM-1B** **GPIO-01**	**22** **ADC-B3**	**23** **EPWM-2B** **GPIO-03**	**24** **SPISIMO** **GPIO-16**
25 **SPISTE** **GPIO-19**	**26** **ADC-B2** **COMP1 (-VE)**	**27** **EPWM-2A** **GPIO-02**	**28** **GND**
29 **SPICLK** **GPIO-18**	**30** **GPIO-34** **(LED)**	**31** **PWM1A-DAC** **(Filtered)**	**32** **GND**

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 (4Kw or 8Kw)
- Minimum Program size is a bit!
- Important not to lose power during erase step: If CSM passwords happen to be all zeros, the CSM will be permanently locked!
- Chance of this happening is quite small! (Erase step is performed sector by sector)

Programming Utilities and CCS Flash Programmer

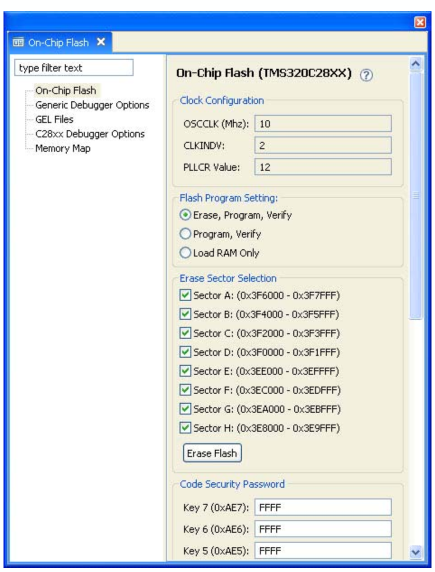
Flash Programming Utilities

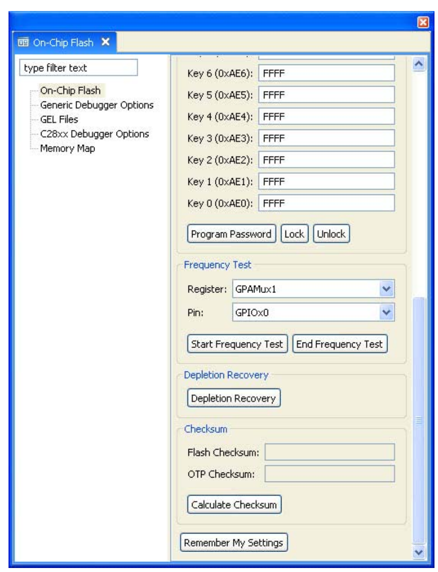
- **JTAG Emulator Based**
  - **Code Composer Studio on-chip Flash programmer**
  - **BlackHawk Flash utilities (requires Blackhawk emulator)**
  - **Elprotronic FlashPro2000**
  - **Spectrum Digital SDFlash JTAG (requires SD emulator)**
  - **Signum System Flash utilities (requires Signum emulator)**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

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

CCS On-Chip Flash Programmer

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

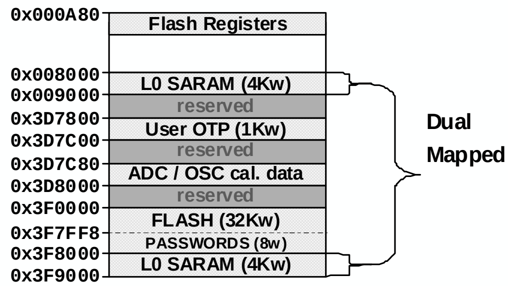




Code Security Module and Password

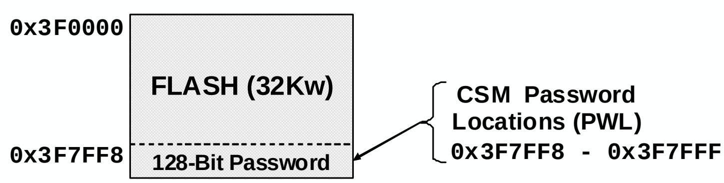
Code Security Module (CSM)

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



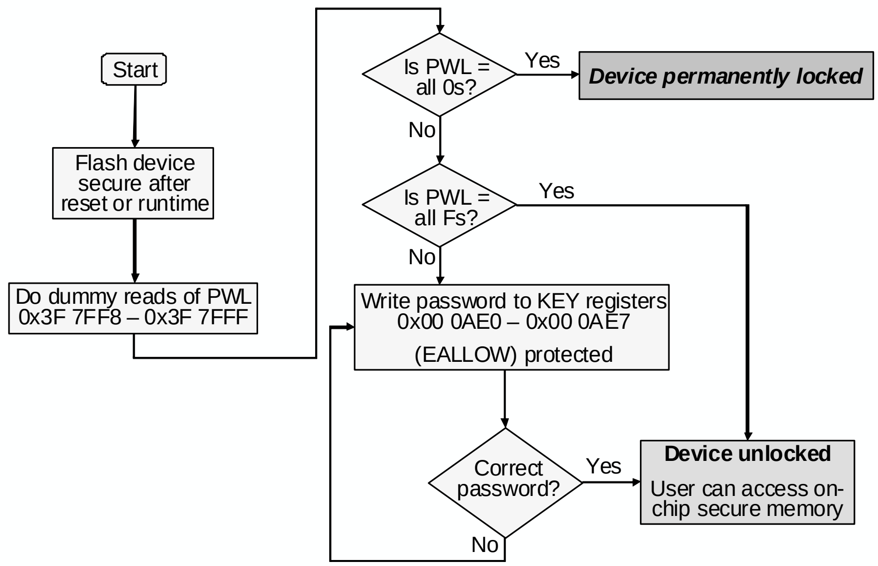
- Data reads and writes from restricted memory are only allowed for code running from restricted memory
- All other data read/write accesses are blocked:
  JTAG emulator/debugger, ROM bootloader, code running in external memory or unrestricted internal memory

CSM Password



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

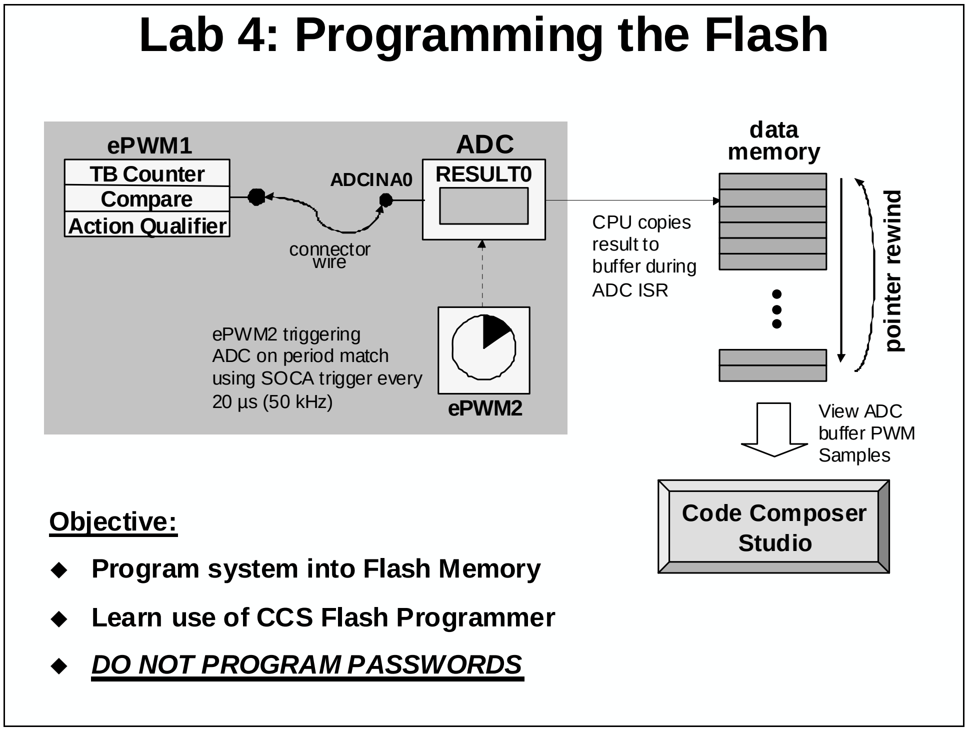
CSM Password Match Flow



Lab 4: Programming the Flash

➢ **Objective**

The objective of this lab is to program and execute code from the on-chip flash memory. The TMS320F28027 device has been designed for standalone operation in an embedded system. Using the on-chip flash eliminates the need for external non-volatile memory or a host processor from which to bootload. In this lab, the steps required to properly configure the software for execution from internal flash memory will be covered.



➢ **Procedure**

Open the Project

1. A project named `Lab4` has been created for this lab. Open the project by clicking on `Project → Import Existing CCS/CCE Eclipse Project`. The "Import" window will open then click `Browse`... next to the "Select root directory" box. Navigate to: `C:\C28x\Labs\Lab4\Project` and click `OK`. Then click `Finish` to import the project.

2. In the `C/C++ Projects` window, click the plus sign (+) to the left of `Lab4` to view the project files. All Build Options have been configured for this lab. The files used in this lab are:

```
Adc.c                               Gpio.c
CodeStartBranch.asm                 Lab_4.cmd
DefaultIsr_3_4.c                    Main_4.c
DelayUs.asm                         Passwords.asm
DSP2802x_GlobalVariableDefs.c       PieCtrl.c
DSP2802x_Headers_nonBIOS.cmd        PieVect.c
EPwm.c                              SysCtrl.c
Flash.c                             Watchdog.c
```

Link Initialized Sections to Flash

Initialized sections, such as code and constants, must contain valid values at device power-up. Stand-alone operation of an F28027 embedded system means that no emulator is available to initialize the device RAM. Therefore, all initialized sections must be linked to the on-chip flash memory.

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

3. Open and inspect the linker command file `Lab_4.cmd`. Notice that a memory block named `FLASH_ABCD` has been been created at origin = 0x3F0000, length = 0x007F80 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.

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

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.

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

Initializing the Flash Control Registers

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

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

7. The "`secureRamFuncs`" section will be linked using the user linker command file `Lab_4.cmd`. Open and inspect `Lab_4.cmd`. The "`secureRamFuncs`" will load to flash (load address) but will run from `L0SARAM` (run address). Also notice that the linker has been asked to generate symbols for the load start, load end, and run start addresses.

   While not a requirement from a MCU hardware or development tools perspective (since the C28x MCU has a unified memory architecture), historical convention is to link code to program memory space and data to data memory space. Therefore, notice that for the `L0SARAM` memory we are linking "`secureRamFuncs`" to, we are specifiying "PAGE = 0" (which is program memory).

8. Open and inspect `Main_4.c`. Notice that the memory copy function memcpy() is being used to copy the section "`secureRamFuncs`", which contains the initialization function for the flash control registers.

9. The following line of code in `main()` is used call the `InitFlash()` function. Since there are no passed parameters or return values the code is just:

   ```
   InitFlash();
   ```

   at the desired spot in `main()`.

Code Security Module and Passwords

The CSM module provides protection against unwanted copying (i.e. pirating!) of your code from flash, OTP memory, and the L0SARAM block. The CSM uses a 128-bit password made up of 8 individual 16-bit words. They are located in flash at addresses 0x3F7FF8 to 0x3F7FFF. During this lab, dummy passwords of 0xFFFF will be used – therefore only dummy reads of the password locations are needed to unsecure the CSM. ***DO NOT PROGRAM ANY REAL PASSWORDS INTO THE DEVICE***. After development, real passwords are typically placed in the password locations to protect your code. We will not be using real passwords in the workshop.

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

10. Open and inspect `Passwords.asm`. This file specifies the desired password values ***<u>(DO NOT CHANGE THE VALUES FROM 0xFFFF)</u>*** and places them in an initialized section named "`passwords`". It also creates an initialized section named "`csm_rsvd`" which contains all 0x0000 values for locations 0x3F7F80 to 0x3F7FF5 (length of 0x76).

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

Executing from Flash after Reset

The F28027 device contains a ROM bootloader that will transfer code execution to the flash after reset. When the boot mode selection is set for "Jump to Flash" mode, the bootloader will branch to the instruction located at address 0x3F7FF6 in the flash. An instruction that branches to the beginning of your program needs to be placed at this address. Note that the CSM passwords begin at address 0x3F7FF8. There are exactly two words available to hold this branch instruction, and not coincidentally, a long branch instruction "LB" in assembly code occupies exactly two words. Generally, the branch instruction will branch to the start of the C-environment initialization routine located in the C-compiler runtime support library. The entry symbol for this routine is *_c_int00*. Recall that C code cannot be executed until this setup routine is run. Therefore, assembly code must be used for the branch. We are using the assembly code file named `CodeStartBranch.asm`.

12. Open and inspect `CodeStartBranch.asm`. This file creates an initialized section named "`codestart`" that contains a long branch to the C-environment setup routine. This section has been linked to a block of memory named `BEGIN_FLASH`.

13. In the earlier lab exercises, the section "`codestart`" was directed to the memory named `BEGIN_M0`. Open and inspect `Lab_4.cmd` and notice that the section "`codestart`" will now be directed to `BEGIN_FLASH`. Close the inspected files.

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

Build – Lab.out

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

CCS Flash Plug-in

In CCS (version 4.x) the on-chip flash programmer is integrated into the debugger. When the program is loaded CCS will automatically determine which sections reside in flash memory based on the linker command file. CCS will then program these sections into the on-chip flash memory. Additionally, in order to effectively debug with CCS, the symbolic debug information (e.g., symbol and label addresses, source file links, etc.) will automatically load so that CCS knows where everything is in your code.

Clicking the "`Debug`" button in the `C/C++ Perspective` will automatically launch the debugger, connect to the target, and program the flash memory in a single step.

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

16. Flash programming options are configured with the "On-Chip Flash" control panel. Open the control panel by clicking:

    `Tools → On-Chip Flash`

    Scroll the control panel and notice the various options that can be selected. You will see that specific actions such as "Erase Flash" can be performed.

    The CCS on-chip flash programmer was automatically configured to use the Piccolo™ 10 MHz internal oscillator as the device clock during programming. Notice the "Clock Configuration" settings has the OSCCLK set to 10 MHz, the DIVSEL set to /2, and the PLLCR value set to 12. Recall that the PLL is divided by two, which gives a SYSCLKOUT of 60 MHz.

    The flash programmer should be set for "Erase, Program, Verify" and all boxes in the "Erase Sector Selection" should be checked. We want to erase all the flash sectors.

    We will not be using the on-chip flash programmer to program the "Code Security Password". ***Do not modify the Code Security Password fields.*** They should remain as all 0xFFFF.

17. Close the "On-Chip Flash" control panel by clicking the `X` on the tab.

Running the Code – Using CCS

18. Reset the CPU using the "Reset CPU" button or click:

    `Target → Reset → Reset CPU`

    The program counter should now be at address 0x3FF7BB in the "Disassembly" window, which is the start of the bootloader in the Boot ROM.

19. Under `Scripts` on the menu bar click:
    `EMU Boot Mode Select → EMU_BOOT_FLASH`.
    This has the debugger load values into EMU_KEY and EMU_BMODE so that the bootloader will jump to "FLASH" at address 0x3F7FF6.

20. Single-Step by using the <F5> key (or you can use the `Step Into` button on the horizontal toolbar) through the bootloader code until you arrive at the beginning of the codestart section in the `CodeStartBranch.asm` file. (Be patient, it will take about 125 single-steps). Notice that we have placed some code in `CodeStartBranch.asm` to give an option to first disable the watchdog, if selected.

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

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

23. You can now RUN the CPU, and you should observe the LED on the controlSTICK blinking. Try resetting the CPU, select the `EMU_BOOT_FLASH` boot mode, and then hitting RUN (without doing all the stepping and the Go Main procedure). The LED should be blinking again.

24. HALT the CPU.

Terminate Debug Session and Close Project

25. Terminate the active debug session using the `Terminate All` button. This will close the debugger and return CCS to the "C/C++ Perspective" view.

26. Next, close the project by right-clicking on `Lab4` in the `C/C++ Projects` window and select `Close Project`.

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

27. Close Code Composer Studio.

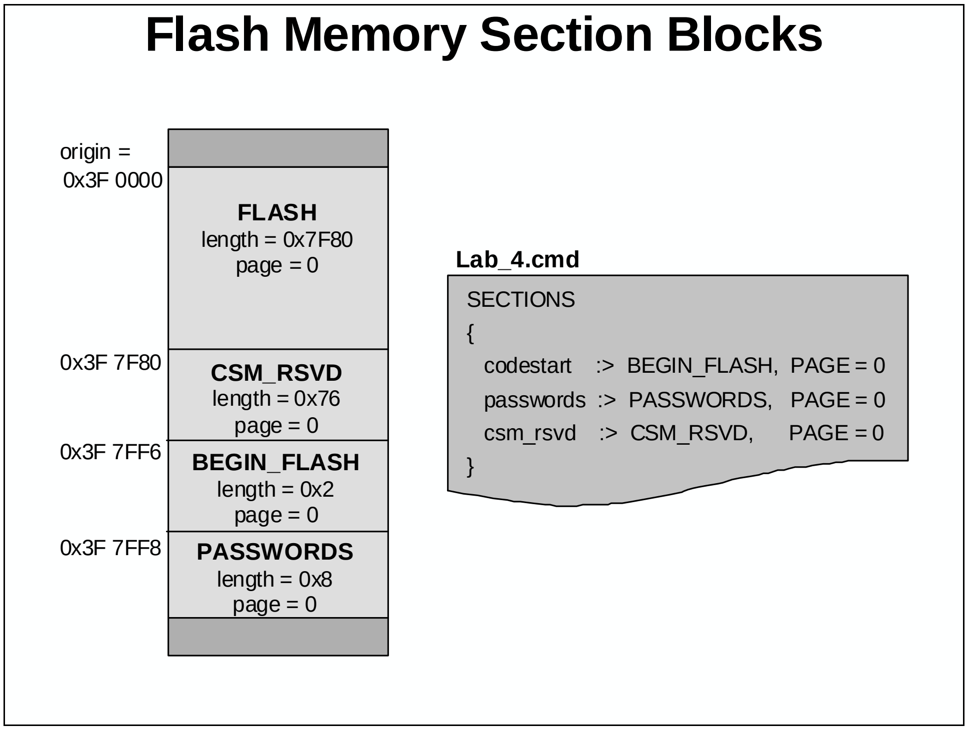
28. Disconnect the controlSTICK from the computer USB port.

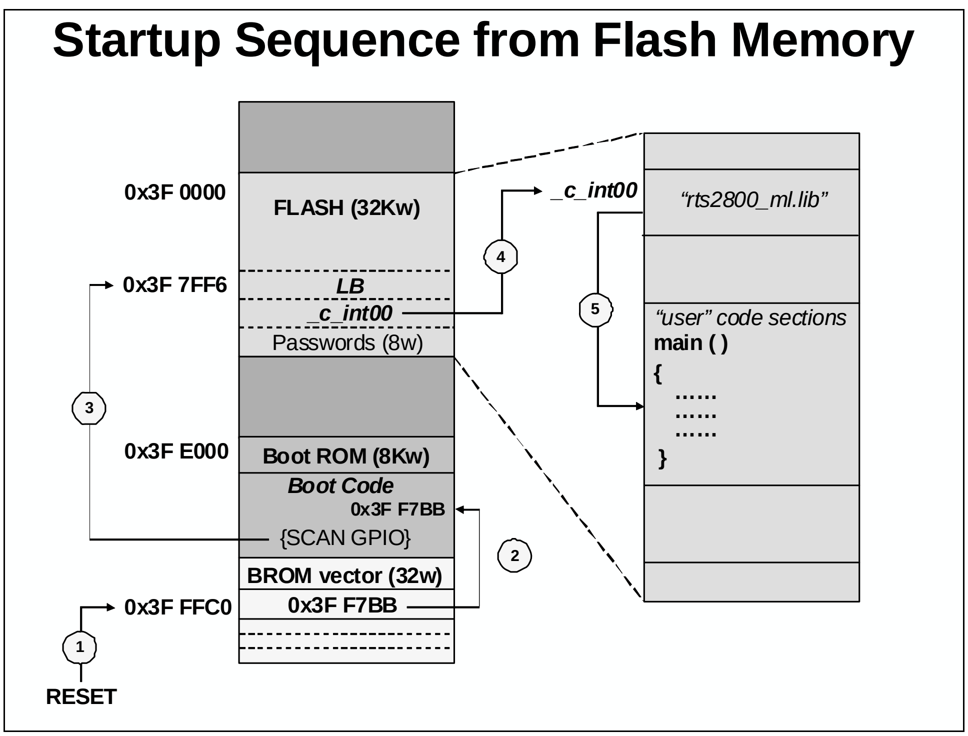
29. Re-connect the controlSTICK to the computer USB port.

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

End of Exercise

Lab 4 Reference: Programming the Flash





The Next Step…

Training

F28x Multi-day Training Course

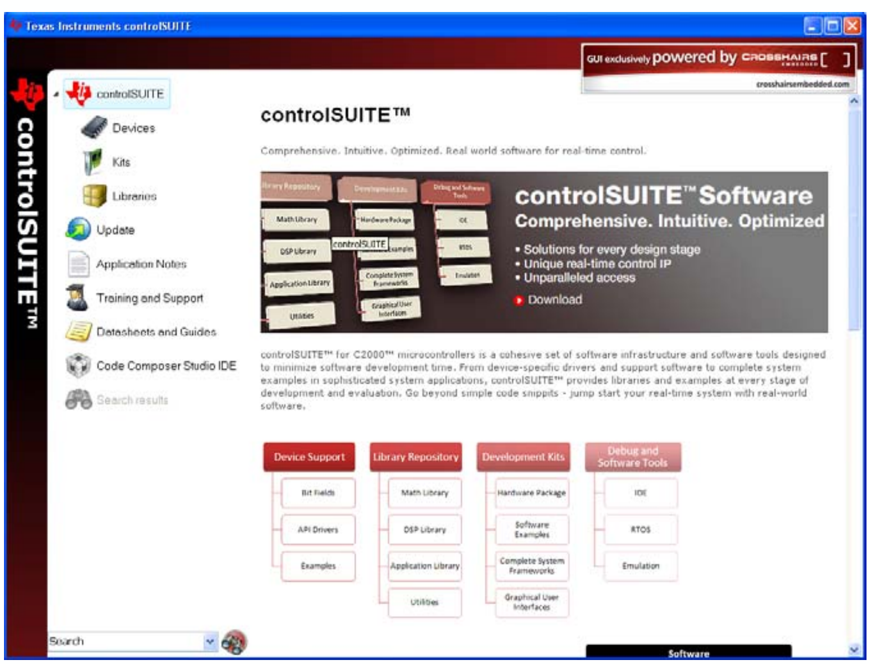
In-depth hands-on TMS320F28035 Design and Peripheral Training

TMS320F2803x 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
- Control Law Accelerator (CLA)
- System Design
- Communications
- DSP/BIOS
- Support Resources

controlSUITE

controlSUITE™

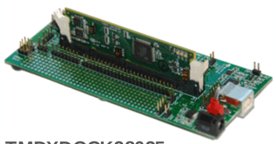


Development Tools

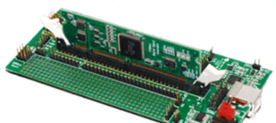
C2000 Experimenter's Kits

F28027, F28035, F2808, F28335

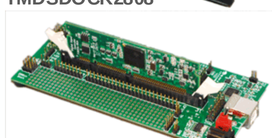
TMDXDOCK28027



TMDXDOCK28035



TMDSDOCK2808

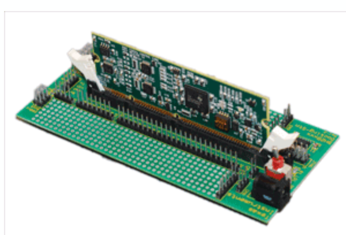


TMDSDOCK28335

- **Experimenter Kits include**
  - F28027, F28035, F2808 or F28335 controlCARD
  - USB docking station
  - C2000 Applications Software CD with example code and full hardware details
  - Code Composer Studio v3.3 with code size limit of 32KB
- **Docking station features**
  - Access to controlCARD signals
  - Breadboard areas
  - Onboard USB JTAG Emulation
    - *JTAG emulator not required*
- **Available through TI authorized distributors and the TI eStore**

C2834x Experimenter's Kits

C28343, C28346

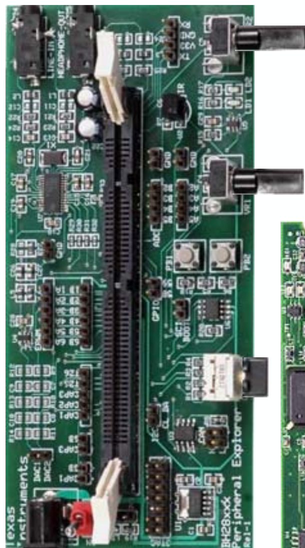


TMDXDOCK28343

TMDXDOCK28346-168

- **Experimenter Kits include**
  - C2834x controlCARD
  - Docking station
  - C2000 Applications Software CD with example code and full hardware details
  - Code Composer Studio v3.3 with code size limit of 32KB
  - 5V power supply
- **Docking station features**
  - Access to controlCARD signals
  - Breadboard areas
  - *JTAG emulator required – sold separately*
- **Available through TI authorized distributors and the TI eStore**

F28335 Peripheral Explorer Kit

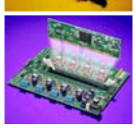


TMDSPREX28335

- Experimenter Kit includes
  - F28335 controlCARD
  - Peripheral Explorer baseboard
  - C2000 Applications Software CD with example code and full hardware details
  - Code Composer Studio 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)
- *JTAG emulator required – sold separately*
- Available through TI authorized distributors and the TI eStore

C2000 controlCARD Application Kits

Digital Power Experimenter's Kit



Digital Power Developer's Kit

Resonant DC/DC Developer's Kit

Renewable Energy Developer's Kit

AC/DC Developer's Kit

Dual Motor Control and PFC Developer's Kit

- Kits includes
  - controlCARD and application specific baseboard
  - Full version of Code Composer Studio v3.3 with 32KB code size limit
- Software download includes
  - Complete schematics, BOM, gerber files, and source code for board and all software
  - Quickstart demonstration GUI for quick and easy access to all board features
  - Fully documented software specific to each kit and application
- See www.ti.com/c2000 for more details
- Available through TI authorized distributors and the TI eStore

C2000 Signal Processing Libraries

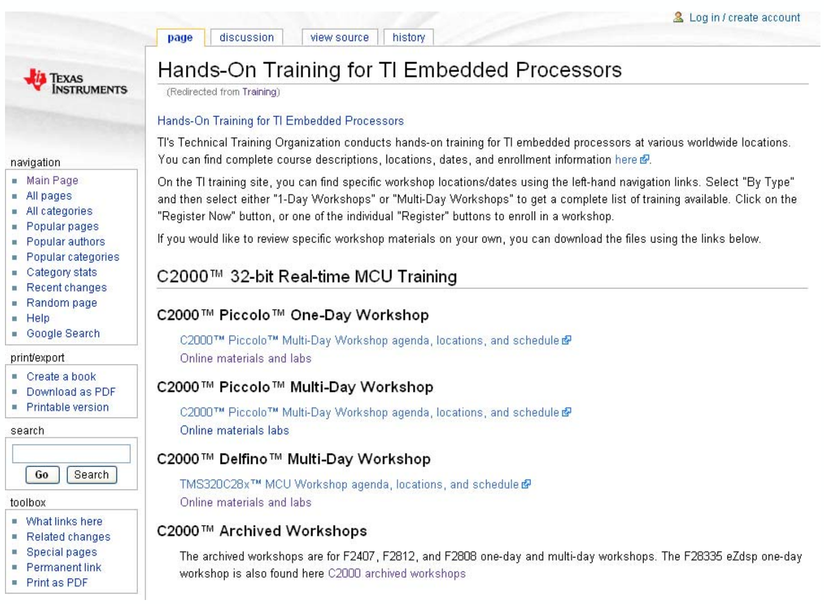
C2000 Signal Processing Libraries

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
DSP2802x C/C++ Header Files and Peripheral Examples	SPRC832

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

C2000 Workshop Download Wiki

C2000 Workshop Download Wiki



http://processors.wiki.ti.com/index.php/Hands-On_Training_for_TI_Embedded_Processors

Development Support

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 all 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

NOTES: