TEXAS INSTRUMENTS

C2000™ Piccolo™ Workshop

Workshop Guide and Lab Manual

F28xPmdw
Revision 1.0
September 2009

TTO
Technical Training
Organization

Important Notice

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

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

Customers are responsible for their applications using TI components.

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

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

Revision History

September 2009 – Revision 1.0

Mailing Address

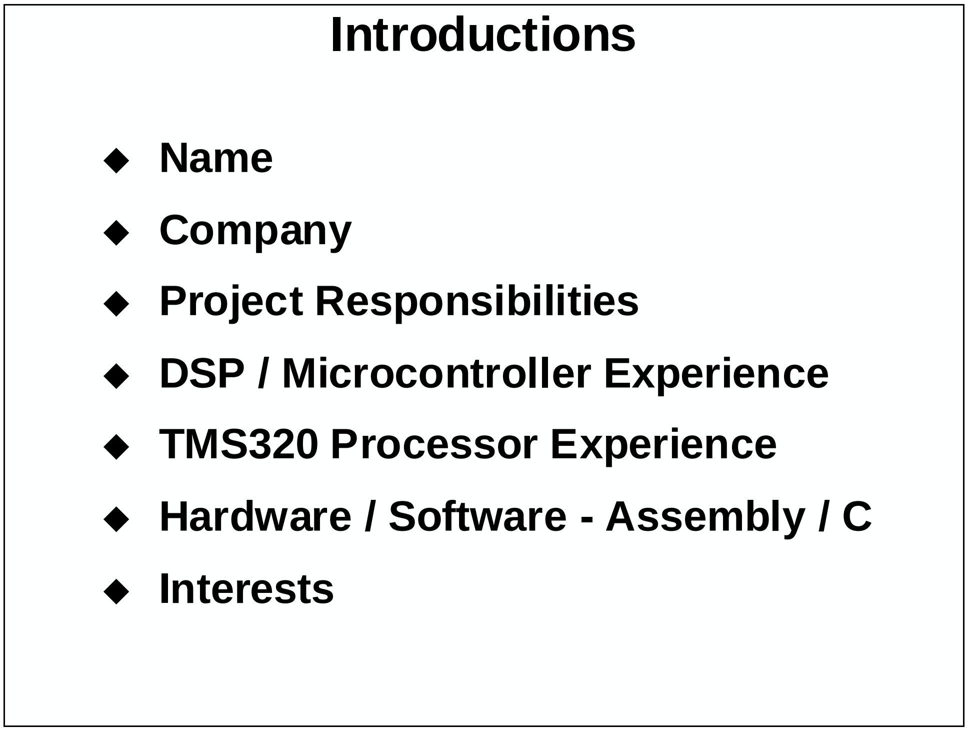
Texas Instruments
Training Technical Organization
7839 Churchill Way
M/S 3984
Dallas, Texas 75251-1903

C2000™ Piccolo™ Workshop

C2000™ Piccolo™ Workshop

Texas Instruments
Technical Training

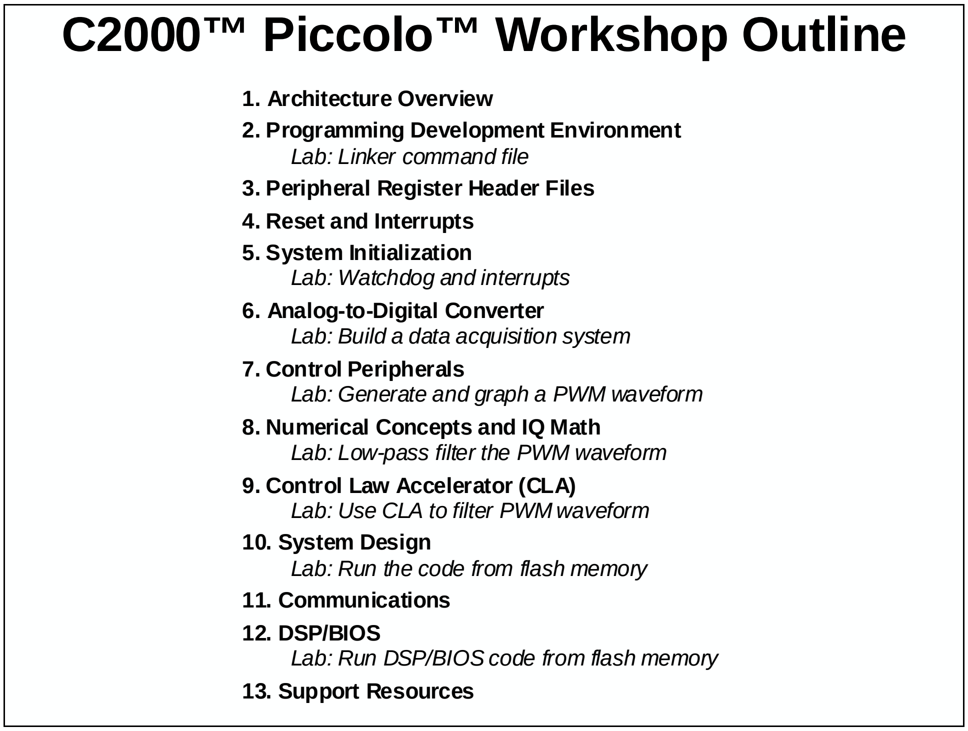
Introductions



Introductions

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

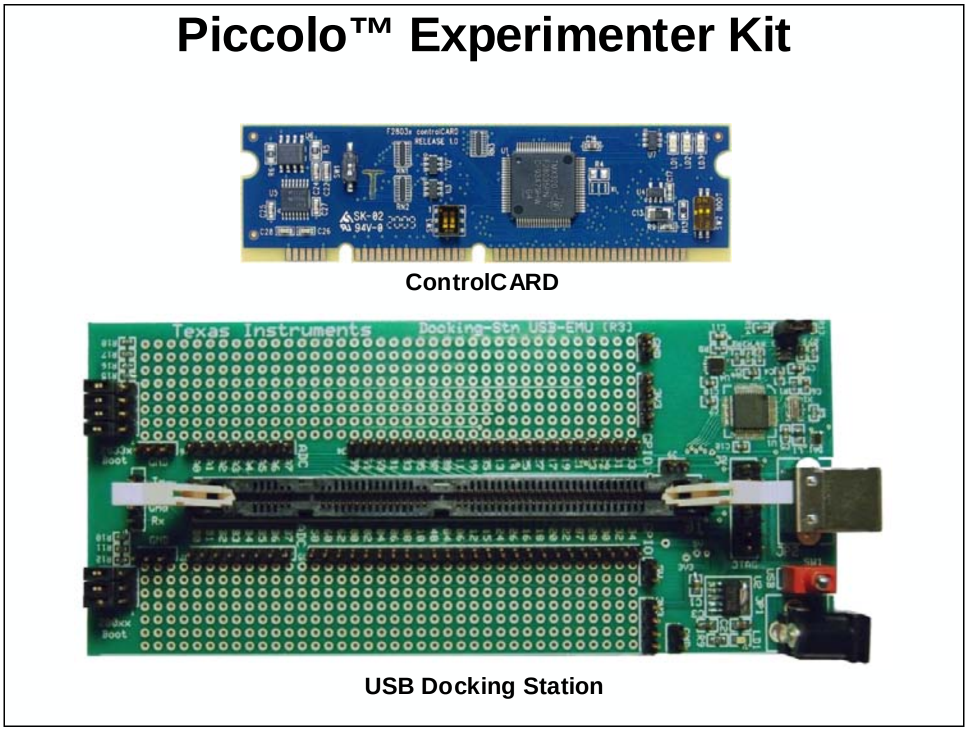
C2000™ Piccolo™ Workshop Outline



C2000™ Piccolo™ Workshop Outline

1. **Architecture Overview**
2. **Programming Development Environment**
   Lab: Linker command file
3. **Peripheral Register Header Files**
4. **Reset and Interrupts**
5. **System Initialization**
   Lab: Watchdog and interrupts
6. **Analog-to-Digital Converter**
   Lab: Build a data acquisition system
7. **Control Peripherals**
   Lab: Generate and graph a PWM waveform
8. **Numerical Concepts and IQ Math**
   Lab: Low-pass filter the PWM waveform
9. **Control Law Accelerator (CLA)**
   Lab: Use CLA to filter PWM waveform
10. **System Design**
    Lab: Run the code from flash memory
11. **Communications**
12. **DSP/BIOS**
    Lab: Run DSP/BIOS code from flash memory
13. **Support Resources**

C2000™ Experimenter Kit



Piccolo™ Experimenter Kit

ControlCARD

USB Docking Station

Architecture Overview

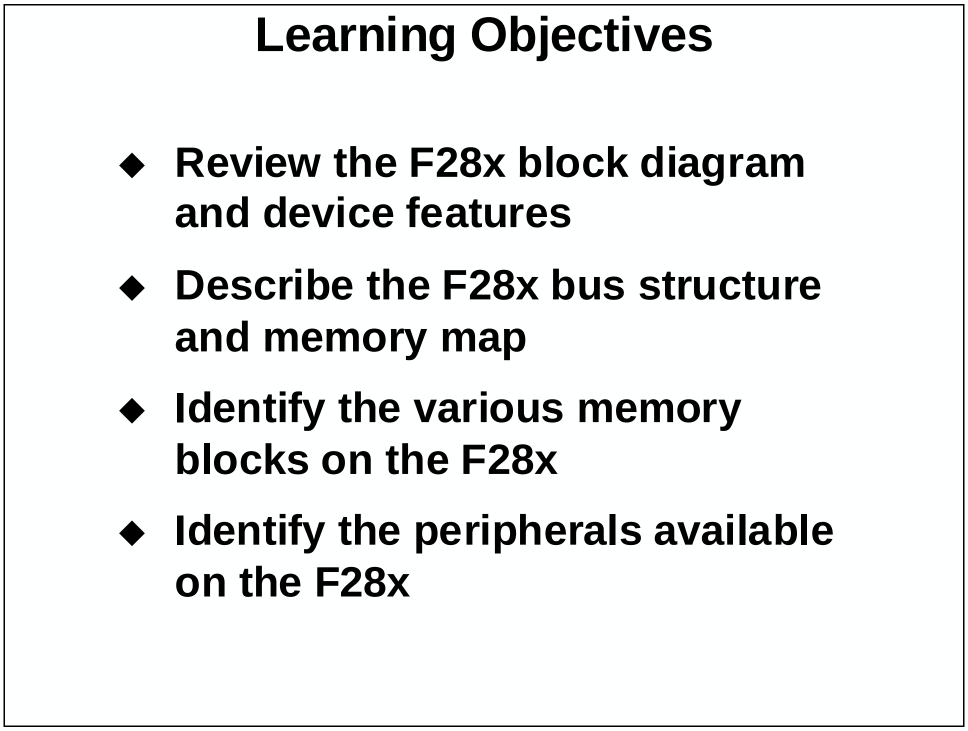
Introduction

This architectural overview introduces the basic architecture of the C2000™ Piccolo™ series of microcontrollers from Texas Instruments. The Piccolo™ series adds a new level of general purpose processing ability unseen in any previous DSP/MCU chips. The C2000™ is ideal for applications combining digital signal processing, microcontroller processing, efficient C code execution, and operating system tasks.

Unless otherwise noted, the terms C28x, F28x and F2803x refer to TMS320F2803x devices throughout the remainder of these notes. For specific details and differences please refer to the device data sheet and user's guide.

Learning Objectives

When this module is complete, you should have a basic understanding of the F28x architecture and how all of its components work together to create a high-end, uniprocessor control system.



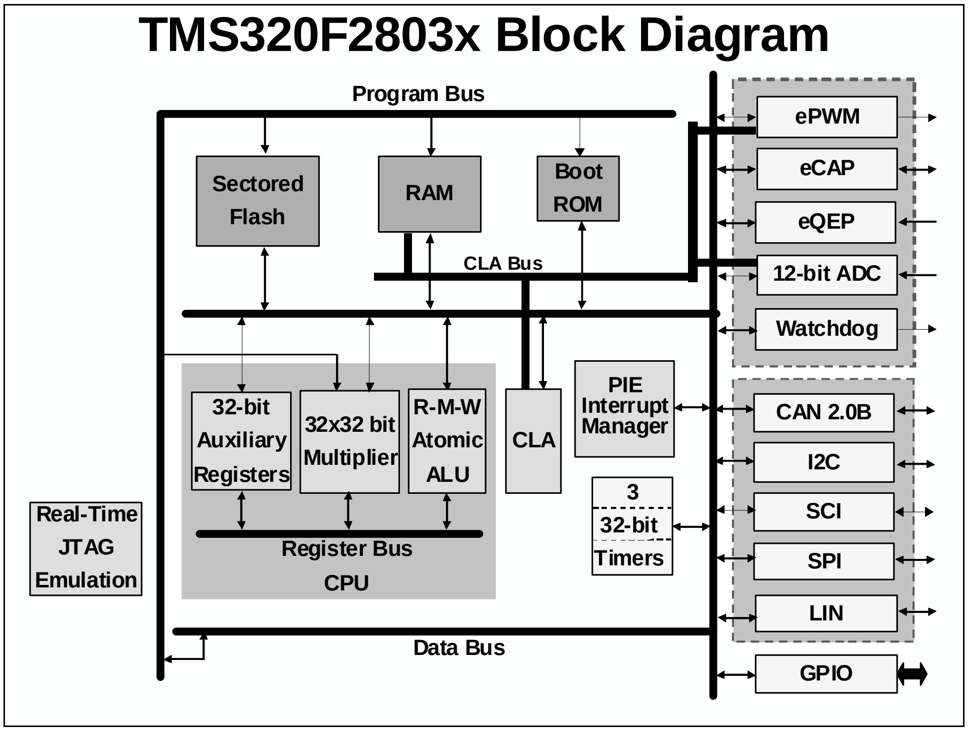
Learning Objectives

- ◆ **Review the F28x block diagram and device features**
- ◆ **Describe the F28x bus structure and memory map**
- ◆ **Identify the various memory blocks on the F28x**
- ◆ **Identify the peripherals available on the F28x**

Module Topics

What is the TMS320C2000™?

The TMS320C2000™ is a 32-bit fixed point microcontroller that specializes in high performance control applications such as, robotics, industrial automation, mass storage devices, lighting, optical networking, power supplies, and other control applications needing a single processor to solve a high performance application.



The F2803x architecture can be divided into 3 functional blocks:

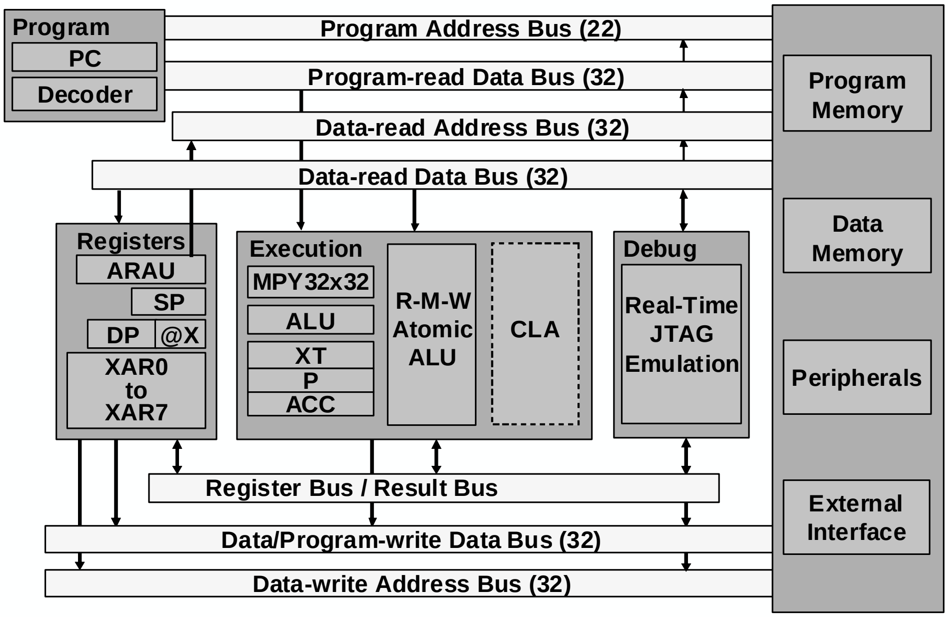
- CPU and busing
- Memory
- Peripherals

TMS320C2000™ Internal Bussing

As with many DSP-type devices, multiple busses are used to move data between the memories and peripherals and the CPU. The F28x memory bus architecture contains:

- A program read bus (22-bit address line and 32-bit data line)
- A data read bus (32-bit address line and 32-bit data line)
- A data write bus (32-bit address line and 32-bit data line)

F28x CPU Internal Bus Structure

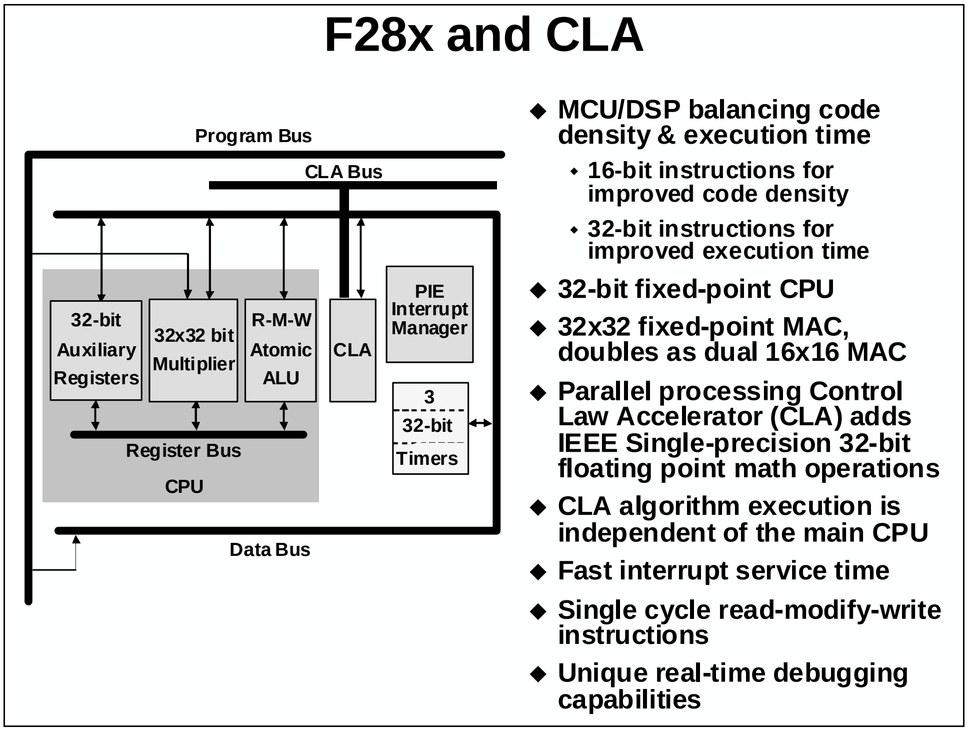


The 32-bit-wide data busses enable single cycle 32-bit operations. This multiple bus architecture, known as a Harvard Bus Architecture enables the F28x to fetch an instruction, read a data value and write a data value in a single cycle. All peripherals and memories are attached to the memory bus and will prioritize memory accesses.

F28x CPU

The F28x is a highly integrated, high performance solution for demanding control applications. The F28x is a cross between a general purpose microcontroller and a digital signal processor, balancing the code density of a RISC processor and the execution speed of a DSP with the architecture, firmware, and development tools of a microcontroller.

The DSP features include a modified Harvard architecture and circular addressing. The RISC features are single-cycle instruction execution, register-to-register operations, and a modified Harvard architecture. The microcontroller features include ease of use through an intuitive instruction set, byte packing and unpacking, and bit manipulation.



F28x and CLA

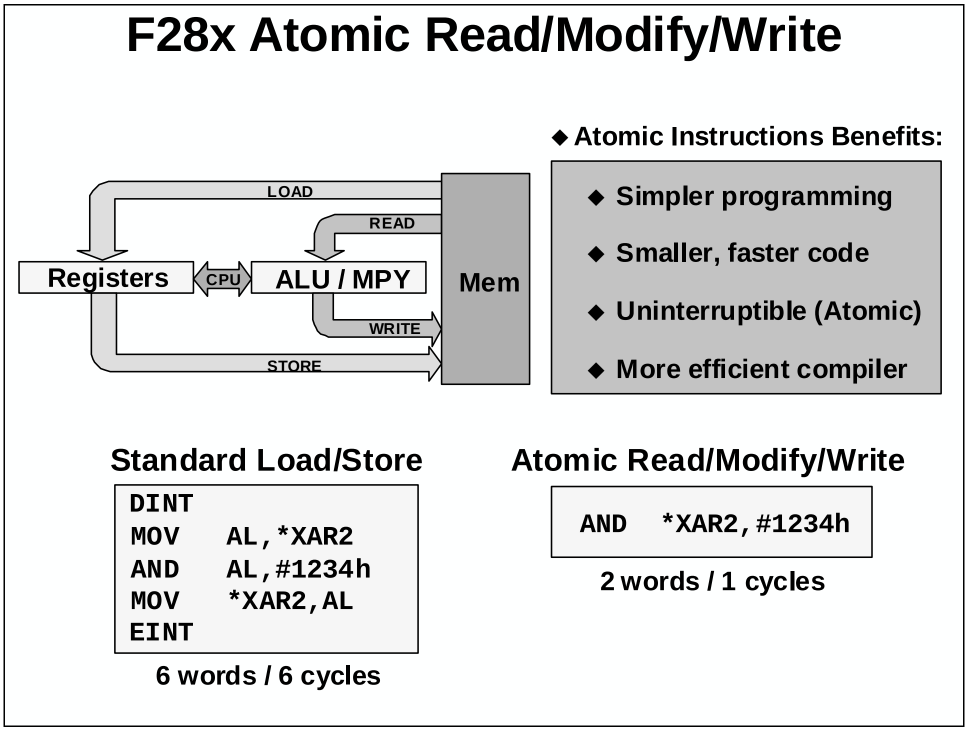
- MCU/DSP balancing code density & execution time
  - 16-bit instructions for improved code density
  - 32-bit instructions for improved execution time
- 32-bit fixed-point CPU
- 32x32 fixed-point MAC, doubles as dual 16x16 MAC
- Parallel processing Control Law Accelerator (CLA) adds IEEE Single-precision 32-bit floating point math operations
- CLA algorithm execution is independent of the main CPU
- Fast interrupt service time
- Single cycle read-modify-write instructions
- Unique real-time debugging capabilities

The F28x design supports an efficient C engine with hardware that allows the C compiler to generate compact code. Multiple busses and an internal register bus allow an efficient and flexible way to operate on the data. The architecture is also supported by powerful addressing modes, which allow the compiler as well as the assembly programmer to generate compact code that is almost one to one corresponded to the C code.

The F28x is as efficient in DSP math tasks as it is in system control tasks. This efficiency removes the need for a second processor in many systems. The 32 x 32-bit MAC capabilities of the F28x and its 64-bit processing capabilities, enable the F28x to efficiently handle higher numerical resolution problems that would otherwise demand a more expensive solution. Along with this is the capability to perform two 16 x 16-bit multiply accumulate instructions simultaneously or Dual MACs (DMAC). Also, some devices feature a floating-point unit.

The, F28x is source code compatible with the 24x/240x devices and previously written code can be reassembled to run on a F28x device, allowing for migration of existing code onto the F28x.

Special Instructions



F28x Atomic Read/Modify/Write

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

Standard Load/Store

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

6 words / 6 cycles

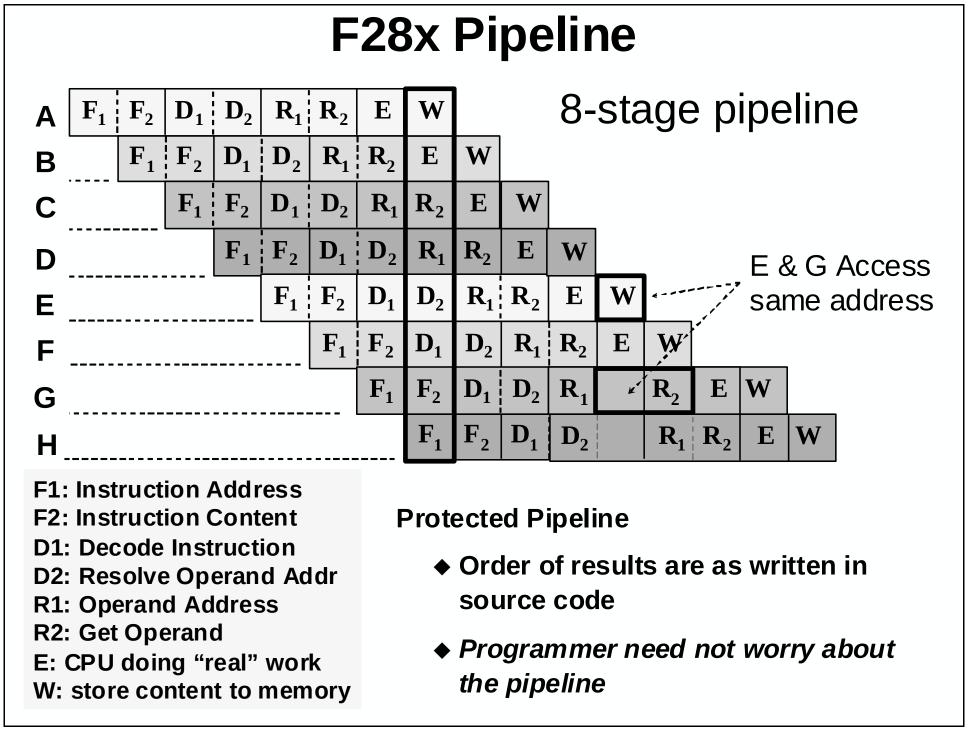
Atomic Read/Modify/Write

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

2 words / 1 cycles

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

Pipeline Advantage



F28x Pipeline

8-stage pipeline

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

E & G Access same address

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

Protected Pipeline

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

The F28x uses a special 8-stage protected pipeline to maximize the throughput. This protected pipeline prevents a write to and a read from the same location from occurring out of order.

This pipelining also enables the F28x to execute at high speeds without resorting to expensive high-speed memories. Special branch-look-ahead hardware minimizes the latency for conditional discontinuities. Special store conditional operations further improve performance.

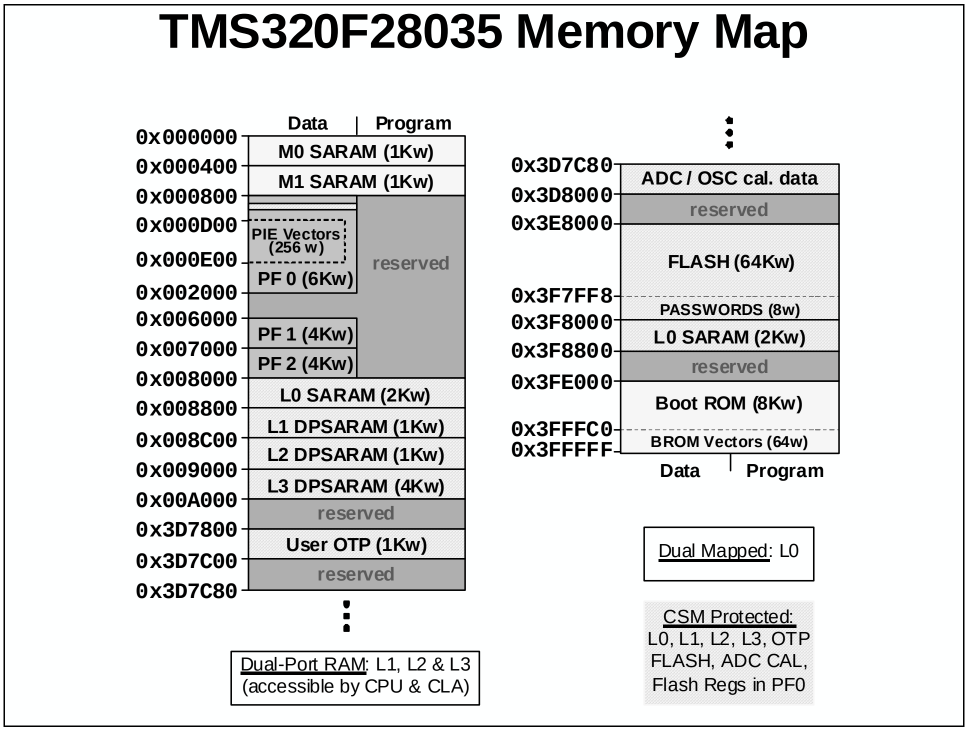
Memory

The memory space on the F28x is divided into program memory and data memory. There are several different types of memory available that can be used as both program memory and data memory. They include the flash memory, single access RAM (SARAM), OTP, and Boot ROM which is factory programmed with boot software routines or standard tables used in math related algorithms.

Memory Map

The F28x CPU contains no memory, but can access memory on chip. The F28x uses 32-bit data addresses and 22-bit program addresses. This allows for a total address reach of 4G words (1 word = 16-bits) in data memory and 4M words in program memory. Memory blocks on all F28x designs are uniformly mapped to both program and data space.

This memory map shows the different blocks of memory available to the program and data space.



TMS320F28035 Memory Map

Address	Data	Program
0x000000	M0 SARAM (1Kw)	
0x000400	M1 SARAM (1Kw)	
0x000800		reserved
0x000D00	PIE Vectors (256 w)	
0x000E00	PF 0 (6Kw)	
0x002000		
0x006000	PF 1 (4Kw)	
0x007000	PF 2 (4Kw)	
0x008000	L0 SARAM (2Kw)	
0x008800	L1 DPSARAM (1Kw)	
0x008C00	L2 DPSARAM (1Kw)	
0x009000	L3 DPSARAM (4Kw)	
0x00A000	reserved	
0x3D7800	User OTP (1Kw)	
0x3D7C00	reserved	
0x3D7C80		

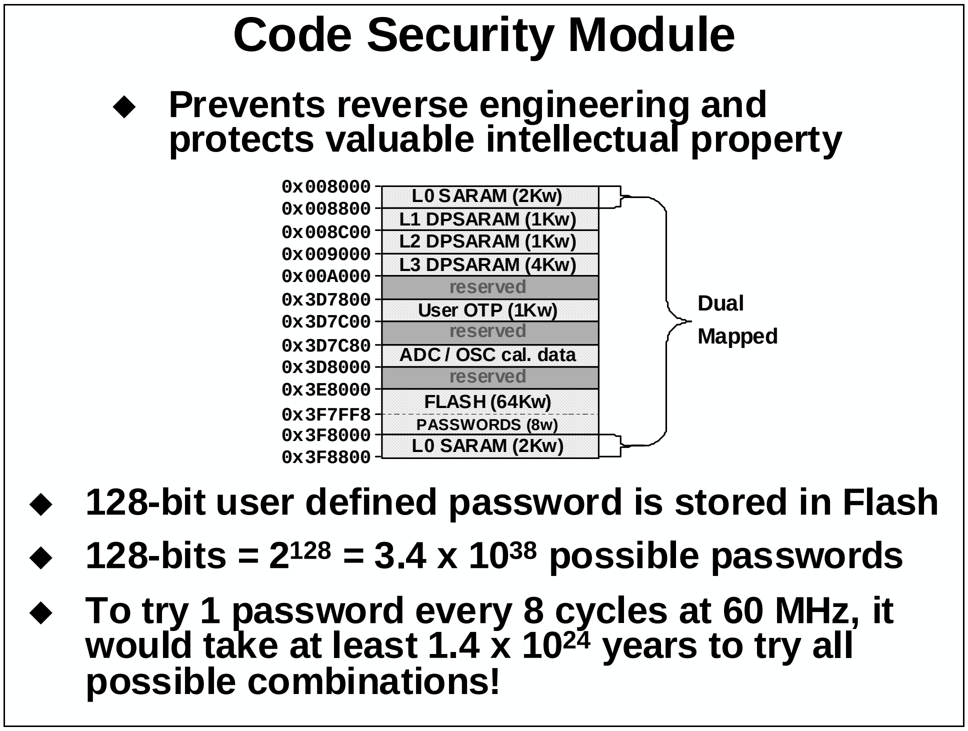
Address	Data	Program
0x3D7C80	ADC / OSC cal. data	
0x3D8000	reserved	
0x3E8000	FLASH (64Kw)	
0x3F7FF8	PASSWORDS (8w)	
0x3F8000	L0 SARAM (2Kw)	
0x3F8800	reserved	
0x3FE000	Boot ROM (8Kw)	
0x3FFFC0	BROM Vectors (64w)	
0x3FFFFF		

Dual-Port RAM: L1, L2 & L3 (accessible by CPU & CLA)

Dual Mapped: L0

CSM Protected:
L0, L1, L2, L3, OTP
FLASH, ADC CAL,
Flash Regs in PF0

Code Security Module (CSM)



Code Security Module

- Prevents reverse engineering and protects valuable intellectual property

Address	Memory
0x008000	L0 SARAM (2Kw)
0x008800	L1 DPSARAM (1Kw)
0x008C00	L2 DPSARAM (1Kw)
0x009000	L3 DPSARAM (4Kw)
0x00A000	reserved
0x3D7800	User OTP (1Kw)
0x3D7C00	reserved
0x3D7C80	ADC / OSC cal. data
0x3D8000	reserved
0x3E8000	FLASH (64Kw)
0x3F7FF8	PASSWORDS (8w)
0x3F8000	L0 SARAM (2Kw)
0x3F8800	

Dual Mapped

- 128-bit user defined password is stored in Flash
- 128-bits = 2^{128} = 3.4 x 10^{38} possible passwords
- To try 1 password every 8 cycles at 60 MHz, it would take at least 1.4 x 10^{24} years to try all possible combinations!

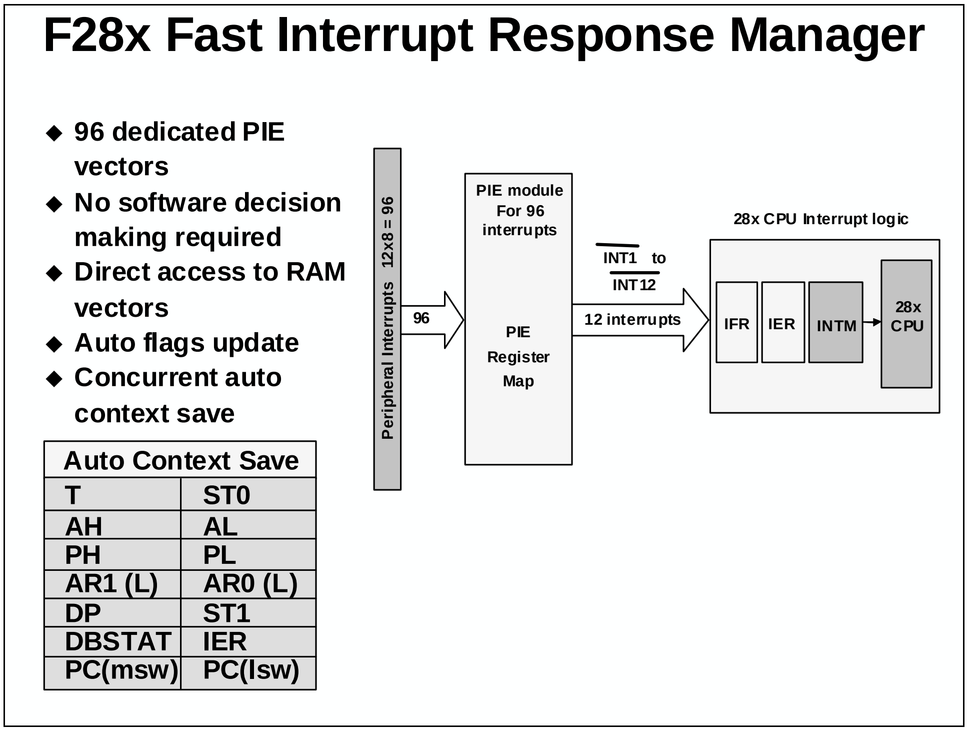
Peripherals

The F28x comes with many built in peripherals optimized to support control applications. These peripherals vary depending on which F28x device you choose.

- ePWM
- eCAP
- eQEP
- Analog-to-Digital Converter
- Watchdog Timer
- CLA
- SPI
- SCI
- I2C
- LIN
- CAN
- GPIO

Fast Interrupt Response

The fast interrupt response, with automatic context save of critical registers, resulting in a device that is capable of servicing many asynchronous events with minimal latency. F28x implements a zero cycle penalty to do 14 registers context saved and restored during an interrupt. This feature helps reduces the interrupt service routine overheads.



F28x Fast Interrupt Response Manager

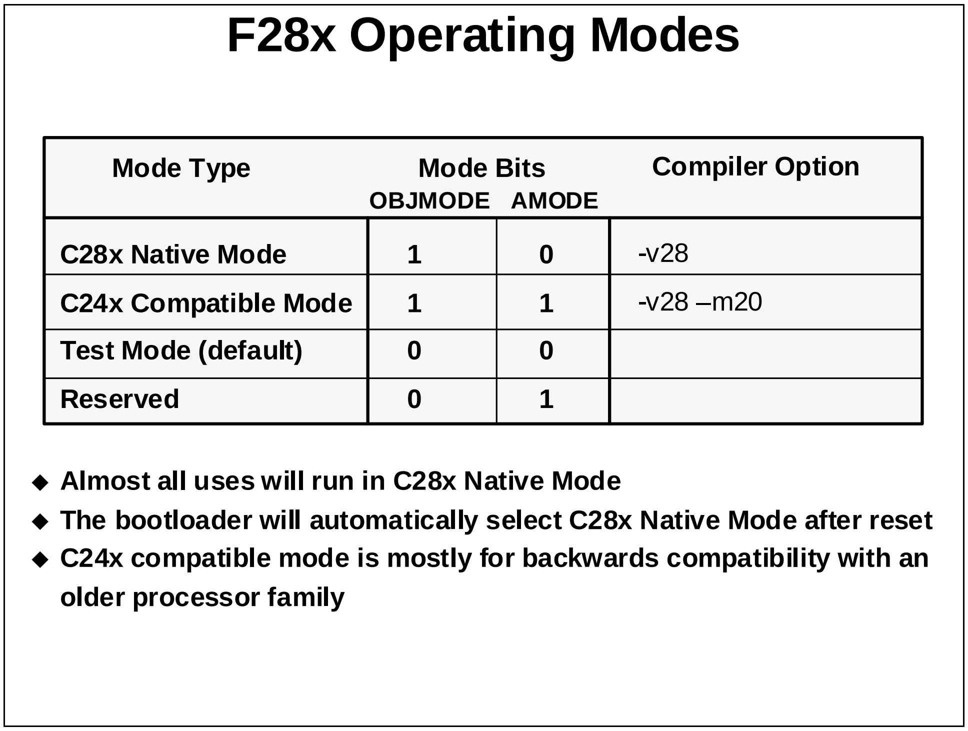
- ◆ **96 dedicated PIE vectors**
- ◆ **No software decision making required**
- ◆ **Direct access to RAM vectors**
- ◆ **Auto flags update**
- ◆ **Concurrent auto context save**

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

Peripheral Interrupts 12x8 = 96
96
PIE module For 96 interrupts
PIE Register Map
$\overline{\text{INT1}}$ to $\overline{\text{INT12}}$
12 interrupts
28x CPU Interrupt logic
IFR
IER
INTM
28x CPU

F28x Mode

The F28x is one of several members of the TMS320 microcontroller family. The F28x is source code compatable with the 24x/240x devices and previously written code can be reassembled to run on a F28x device. This allows for migration of existing code onto the F28x.

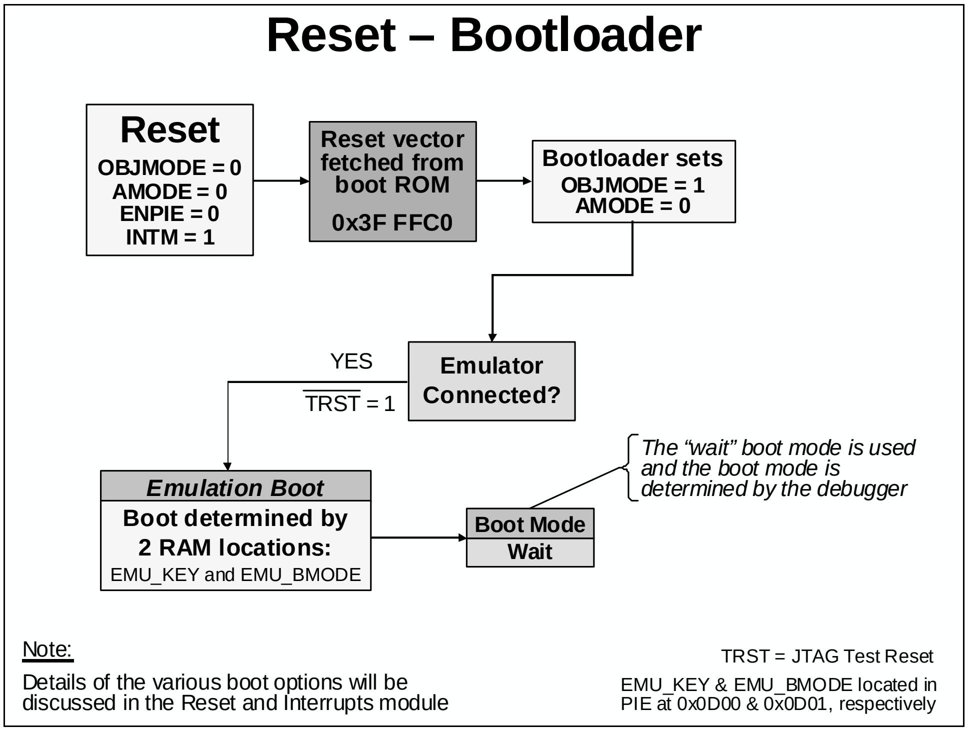


F28x Operating Modes

Mode Type	Mode Bits		Compiler Option
	OBJMODE	AMODE	
C28x Native Mode	1	0	-v28
C24x Compatible Mode	1	1	-v28 –m20
Test Mode (default)	0	0	
Reserved	0	1	

- **Almost all uses will run in C28x Native Mode**
- **The bootloader will automatically select C28x Native Mode after reset**
- **C24x compatible mode is mostly for backwards compatibility with an older processor family**

Reset



Reset – Bootloader

Reset
OBJMODE = 0
AMODE = 0
ENPIE = 0
INTM = 1

Reset vector fetched from boot ROM
0x3F FFC0

Bootloader sets
OBJMODE = 1
AMODE = 0

Emulator Connected?

YES

$\overline{\text{TRST}}$ = 1

Emulation Boot
Boot determined by 2 RAM locations:
EMU_KEY and EMU_BMODE

Boot Mode
Wait

The "wait" boot mode is used and the boot mode is determined by the debugger

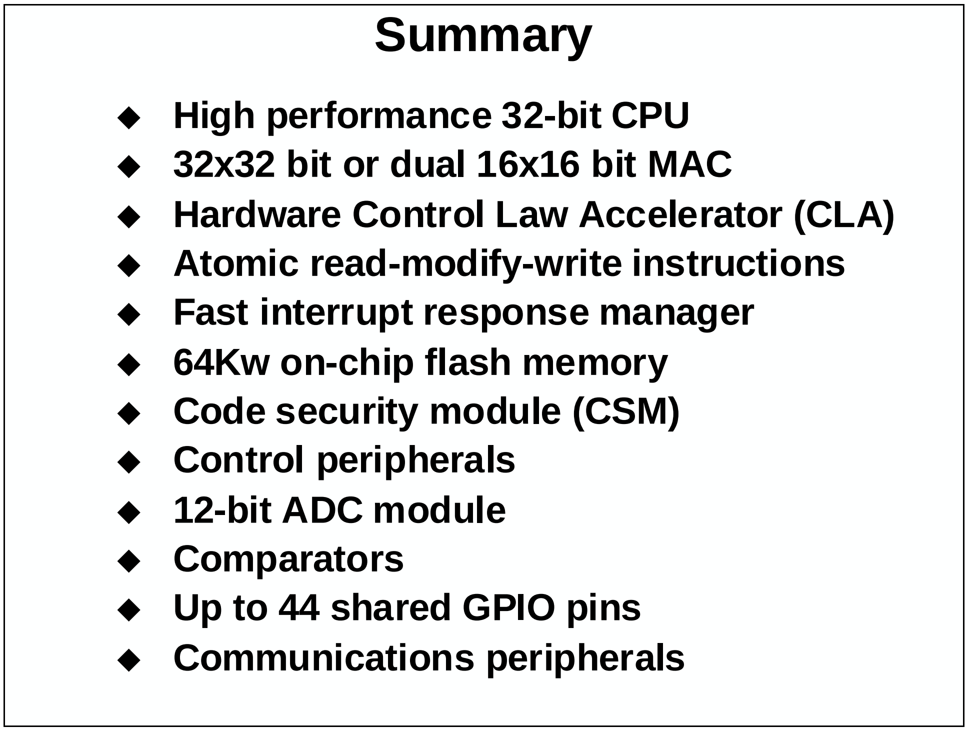
Note:

Details of the various boot options will be discussed in the Reset and Interrupts module

TRST = JTAG Test Reset

EMU_KEY & EMU_BMODE located in PIE at 0x0D00 & 0x0D01, respectively

Summary



Summary

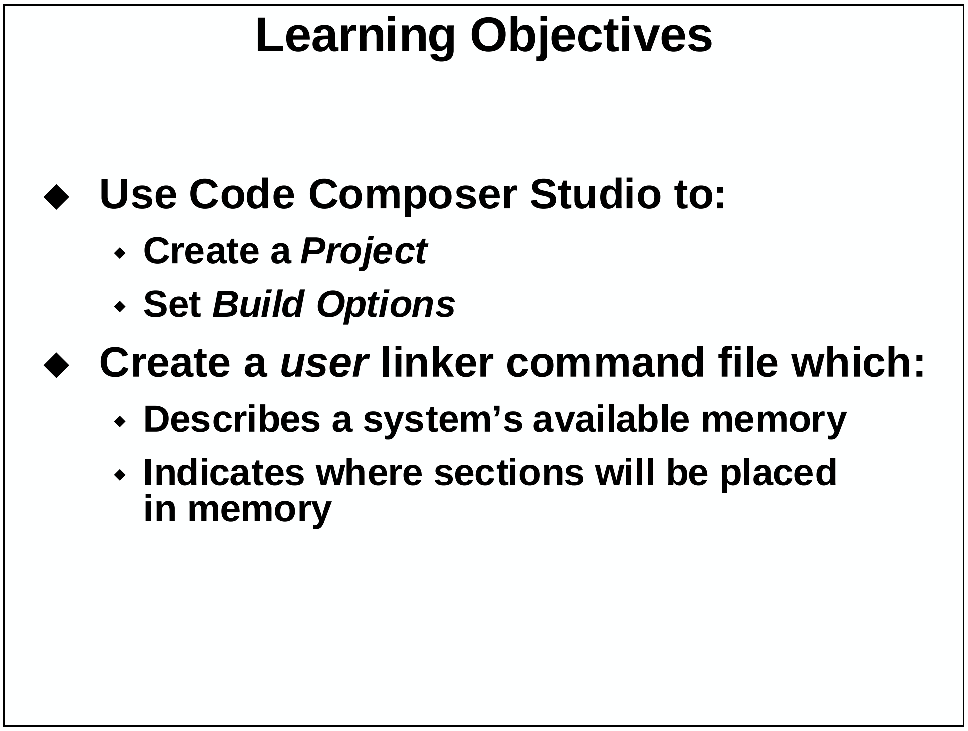
- High performance 32-bit CPU
- 32x32 bit or dual 16x16 bit MAC
- Hardware Control Law Accelerator (CLA)
- Atomic read-modify-write instructions
- Fast interrupt response manager
- 64Kw on-chip flash memory
- Code security module (CSM)
- Control peripherals
- 12-bit ADC module
- Comparators
- Up to 44 shared GPIO pins
- Communications peripherals

Programming Development Environment

Introduction

This module will explain how to use Code Composer Studio (CCS) integrated development environment (IDE) tools to develop a program. Creating projects and setting building options will be covered. Use and the purpose of the linker command file will be described.

Learning Objectives



Learning Objectives

- **Use Code Composer Studio to:**
  - **Create a *Project***
  - **Set *Build Options***
- **Create a *user* linker command file which:**
  - **Describes a system's available memory**
  - **Indicates where sections will be placed in memory**

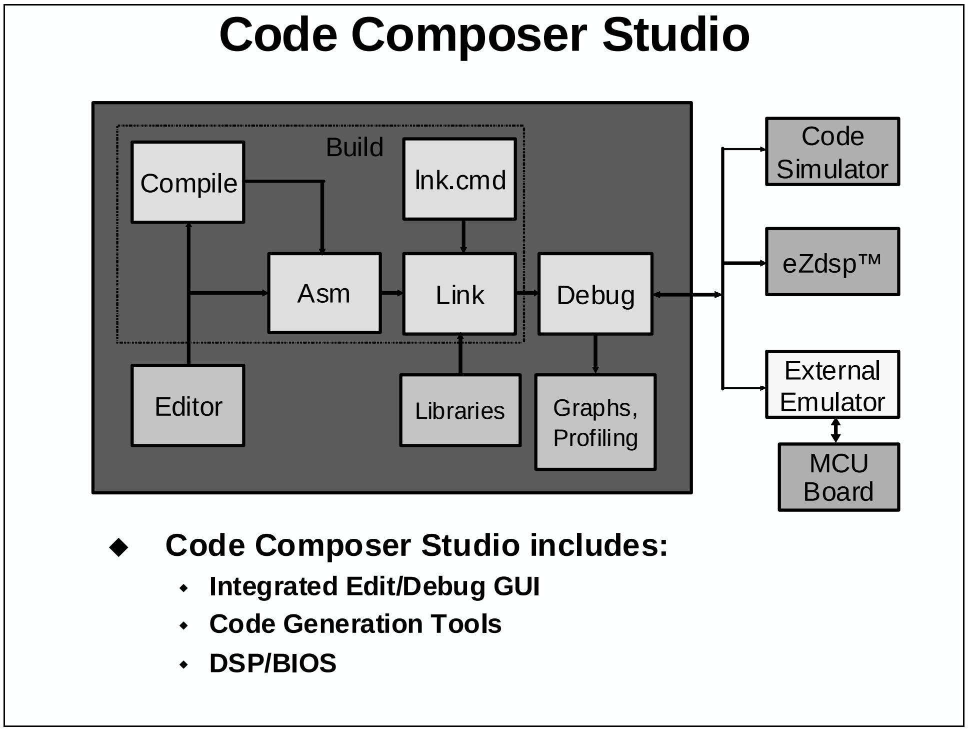
Module Topics

Code Composer Studio

Software Development and COFF Concepts

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

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



Code Composer Studio

- **Code Composer Studio includes:**
  - **Integrated Edit/Debug GUI**
  - **Code Generation Tools**
  - **DSP/BIOS**

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

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

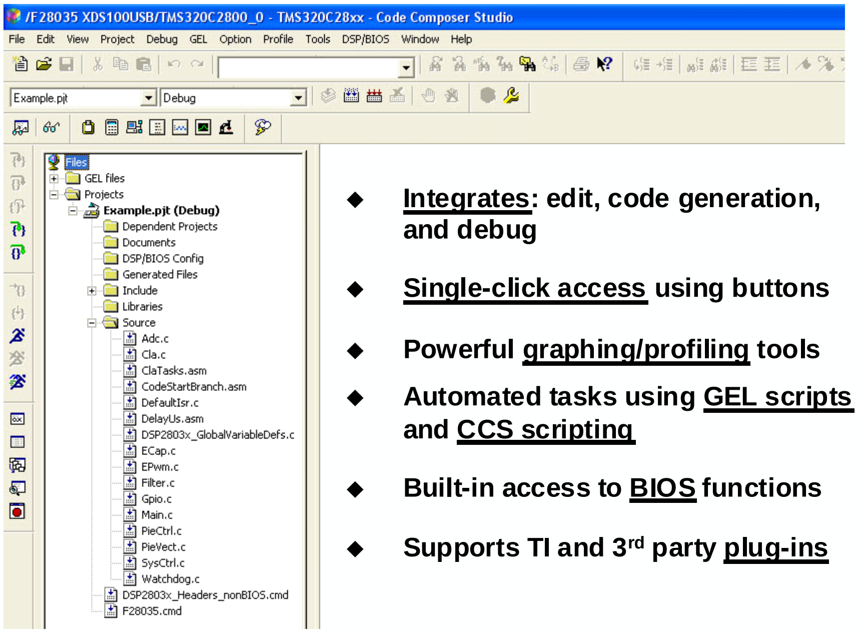
The high level of modularity and portability resulting from this system simplifies the processes of verification, debug and maintenance. The process of COFF development is presented in greater detail in the following paragraphs.

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

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

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

Code Composer Studio: IDE

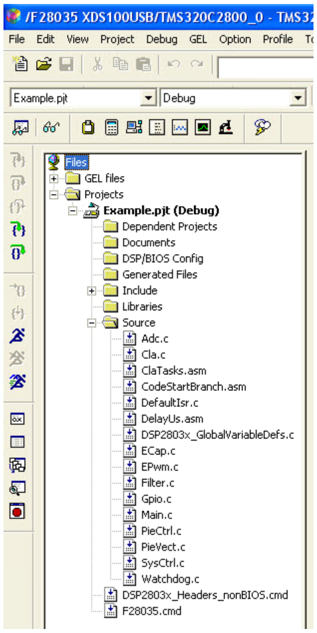


- **Integrates: edit, code generation, and debug**
- **Single-click access using buttons**
- **Powerful graphing/profiling tools**
- **Automated tasks using GEL scripts and CCS scripting**
- **Built-in access to BIOS functions**
- **Supports TI and 3rd party plug-ins**

Projects

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

The CCS Project



Project (.pjt) 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**

The project information is stored in a .PJT file, which is created and maintained by CCS. To create a new project, you need to select the **Project:New**… menu item.

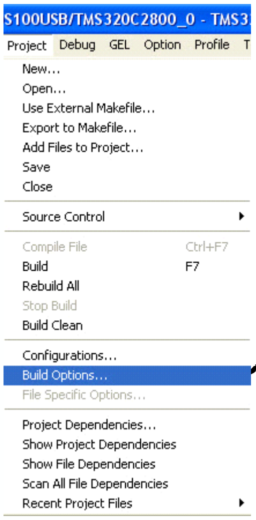
Along with the main **Project** menu, you can also manage open projects using the right-click popup menu. Either of these menus allows you to **Add Files…** to a project. Of course, you can also drag-n-drop files onto the project from Windows Explorer.

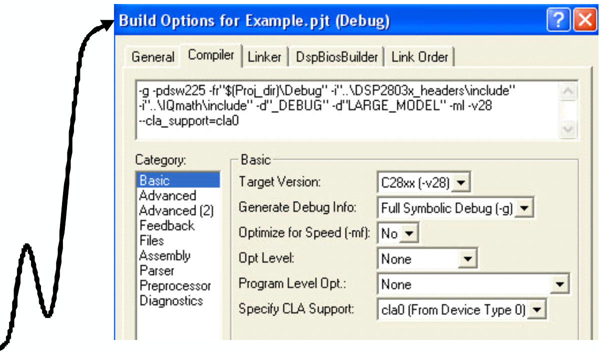
Build Options

Project options direct the code generation tools (i.e. compiler, assembler, linker) to create code according to your system's needs. When you create a new project, CCS creates two sets of build options – called ***Configurations***: one called *Debug*, the other *Release* (you might think of as Optimize).

To make it easier to choose build options, CCS provides a graphical user interface (GUI) for the various compiler options. Here's a sample of the *Debug* configuration options.

Build Options GUI - Compiler





```
-g -pdsw225 -fr"$(Proj_dir)\Debug" -i"..\DSP2803x_headers\include"
-i"..\IQmath\include" -d"_DEBUG" -d"LARGE_MODEL" -ml -v28
--cla_support=cla0
```

Basic

Target Version: C28xx (-v28)

Generate Debug Info: Full Symbolic Debug (-g)

Optimize for Speed (-mf): No

Opt Level: None

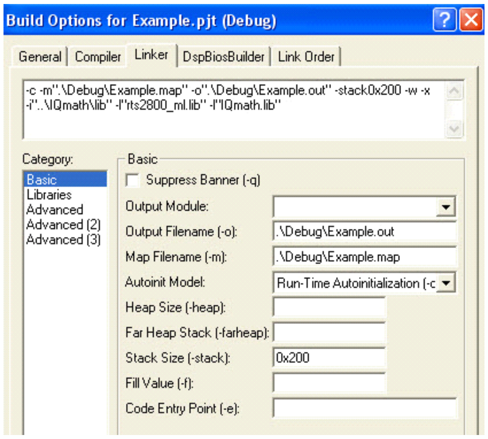
Program Level Opt.: None

Specify CLA Support: cla0 (From Device Type 0)

- **GUI has 8 pages of categories for code generation tools**
- **Controls many aspects of the build process, such as:**
  - **Optimization level**
  - **Target device**
  - **Compiler/assembly/link options**

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

Build Options GUI - Linker



```
-c -m".\Debug\Example.map" -o".\Debug\Example.out" -stack0x200 -w -x
-i"..\IQmath\lib" -l"rts2800_ml.lib" -l"IQmath.lib"
```

Basic

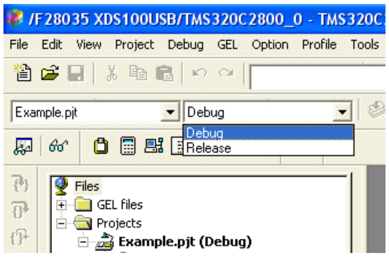
- □ Suppress Banner (-q)
- Output Module:
- Output Filename (-o): .\Debug\Example.out
- Map Filename (-m): .\Debug\Example.map
- Autoinit Model: Run-Time Autoinitialization (-c
- Heap Size (-heap):
- Far Heap Stack (-farheap):
- Stack Size (-stack): 0x200
- Fill Value (-f):
- Code Entry Point (-e):

- GUI has 3 categories for linking
  - Specify various link options
- *.\Debug* means the directory called Debug one level below the .pjt file directory
- *$(Proj_dir)\Debug* is an equivalent expression

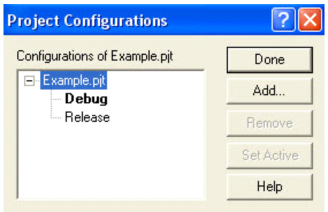
There are many linker options but these four handle all of the basic needs.

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

Default Build Configurations


/F28035 XDS100USB/TMS320C2800_0 - TMS320C
File Edit View Project Debug GEL Option Profile Tools
Example.pjt
Debug
Debug
Release
Files
GEL files
Projects
Example.pjt (Debug)

- For new projects, CCS automatically creates two build configurations:
  - Debug (unoptimized)
  - Release (optimized)
- Use the drop-down menu to quickly select the build configuration



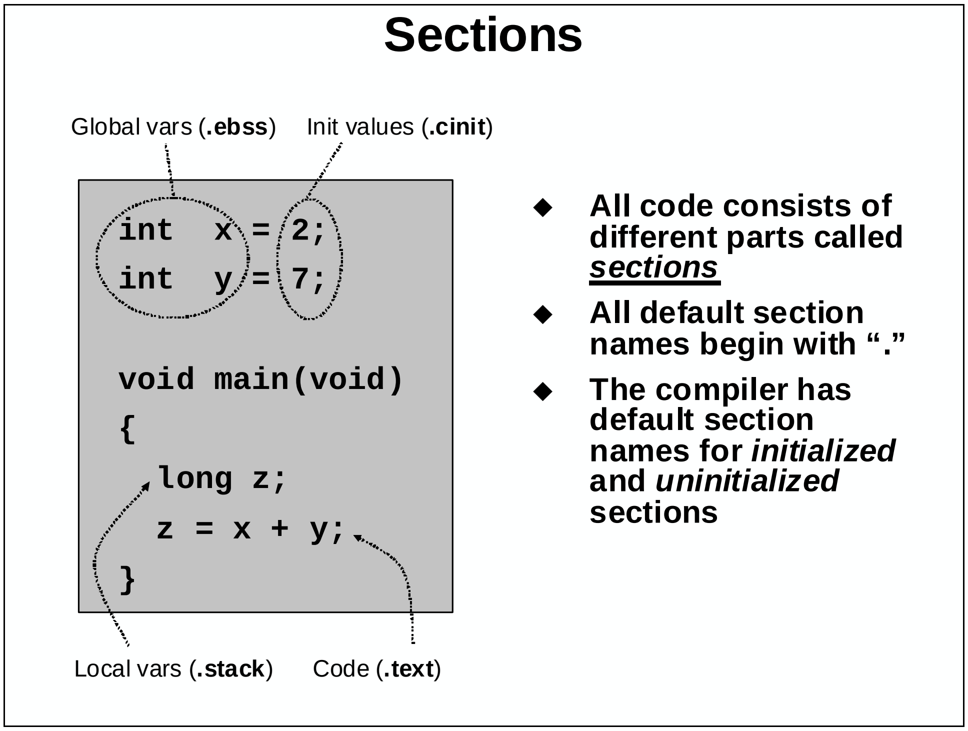
- Add/Remove your own custom build configurations using *Project Configurations*
- Edit a configuration:
  1. Set it active
  2. Modify build options
  3. Save project

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

Creating a Linker Command File

Sections

Looking at a C program, you'll notice it contains both code and different kinds of data (global, local, etc.).



Sections

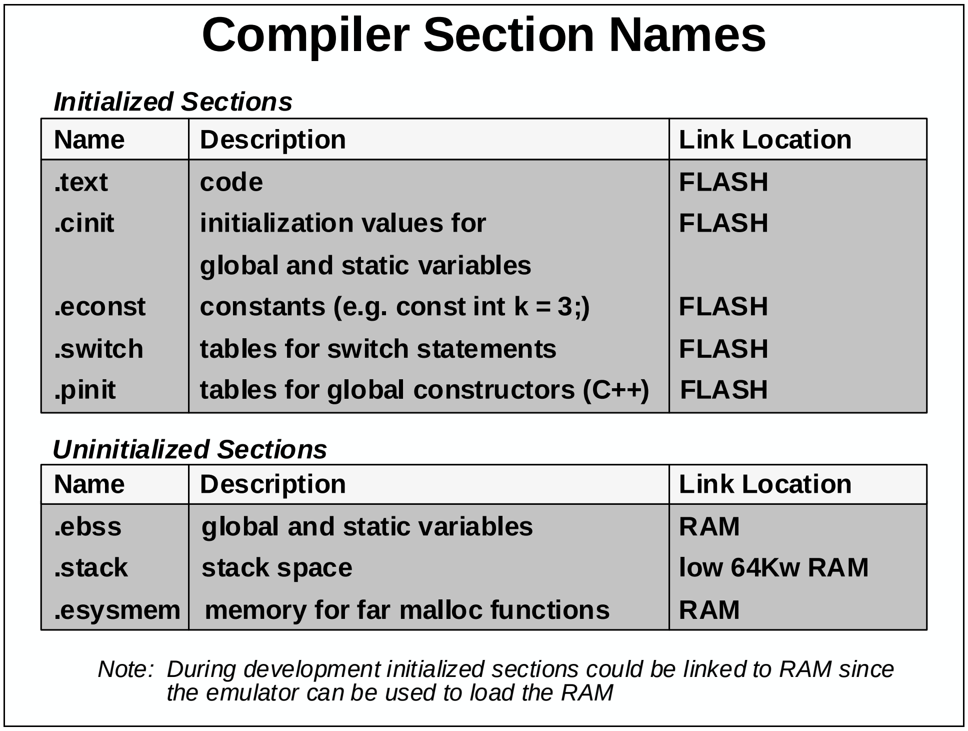
```
int  x = 2;
int  y = 7;

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

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

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

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

Following is a list of the sections that are created by the compiler. Along with their description, we provide the Section Name defined by the compiler.



Compiler Section Names

Initialized Sections

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

Uninitialized Sections

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

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

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

Program Code (.text)

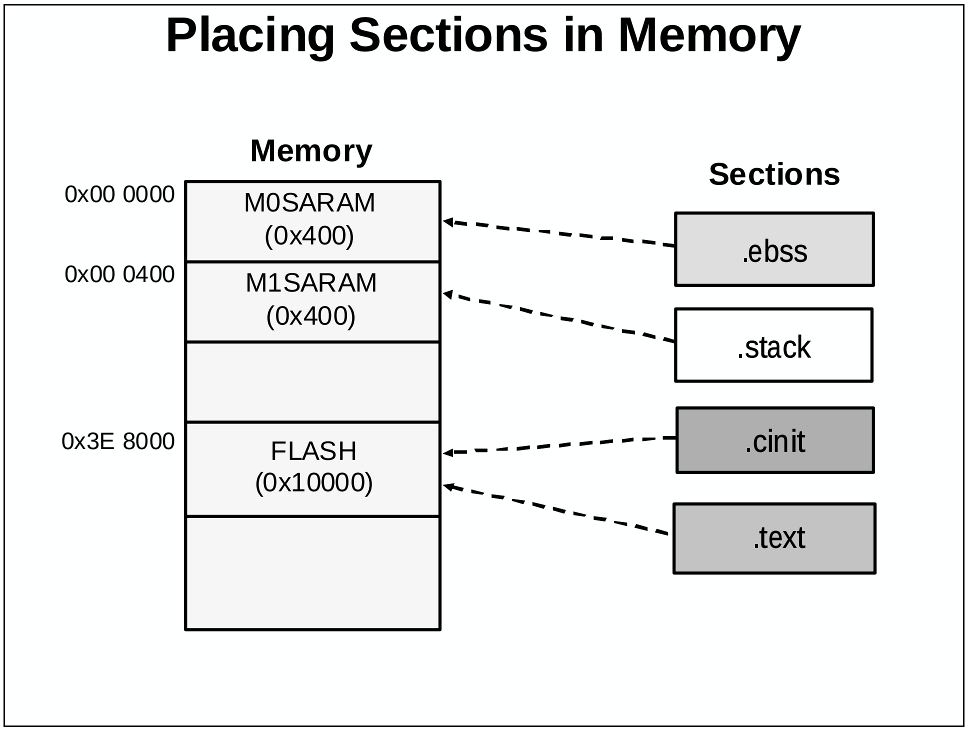
Program code consists of the sequence of instructions used to manipulate data, initialize system settings, etc. Program code must be defined upon system reset (power turn-on). Due to this basic system constraint it is usually necessary to place program code into non-volatile memory, such as FLASH or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

Constants (.cinit – initialized data)

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

Variables (.ebss – uninitialized data)

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



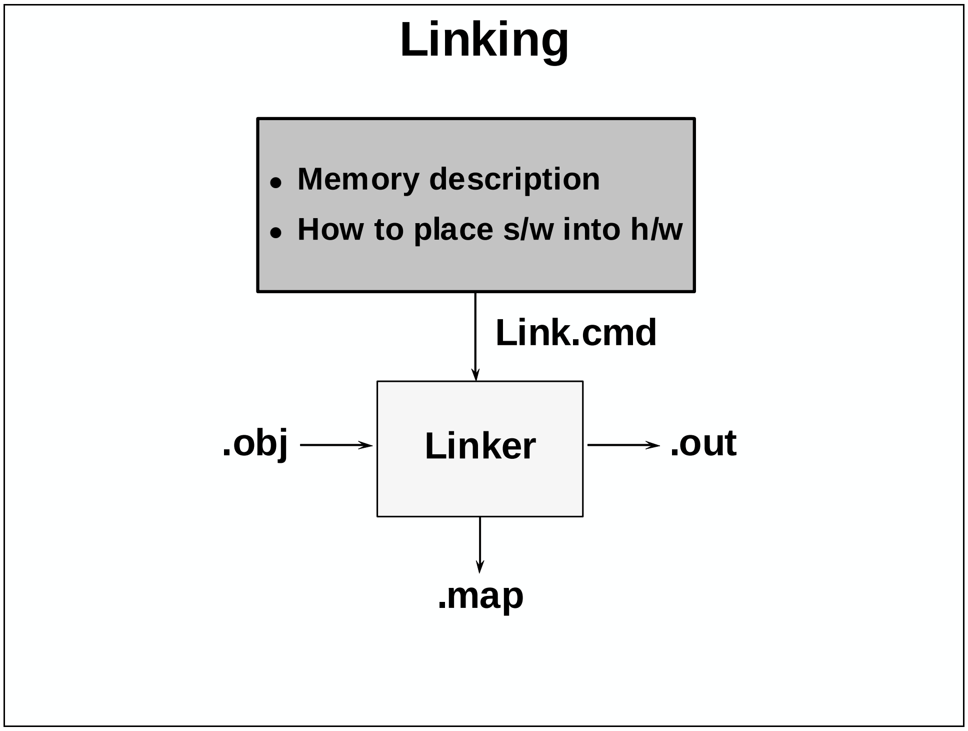
Placing Sections in Memory

Linking code is a three step process:

1. Defining the various regions of memory (on-chip SARAM vs. FLASH vs. External Memory).
2. Describing what sections go into which memory regions
3. Running the linker with “build” or “rebuild”

Linker Command Files (`.cmd`)

The linker concatenates each section from all input files, allocating memory to each section based on its length and location as specified by the MEMORY and SECTIONS commands in the linker command file.



Linking

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

Memory-Map Description

The MEMORY section describes the memory configuration of the target system to the linker.

The format is: *`Name:  origin = 0x????,  length = 0x????`*

For example, if you placed a 64Kw FLASH starting at memory location 0x3E8000, it would read:

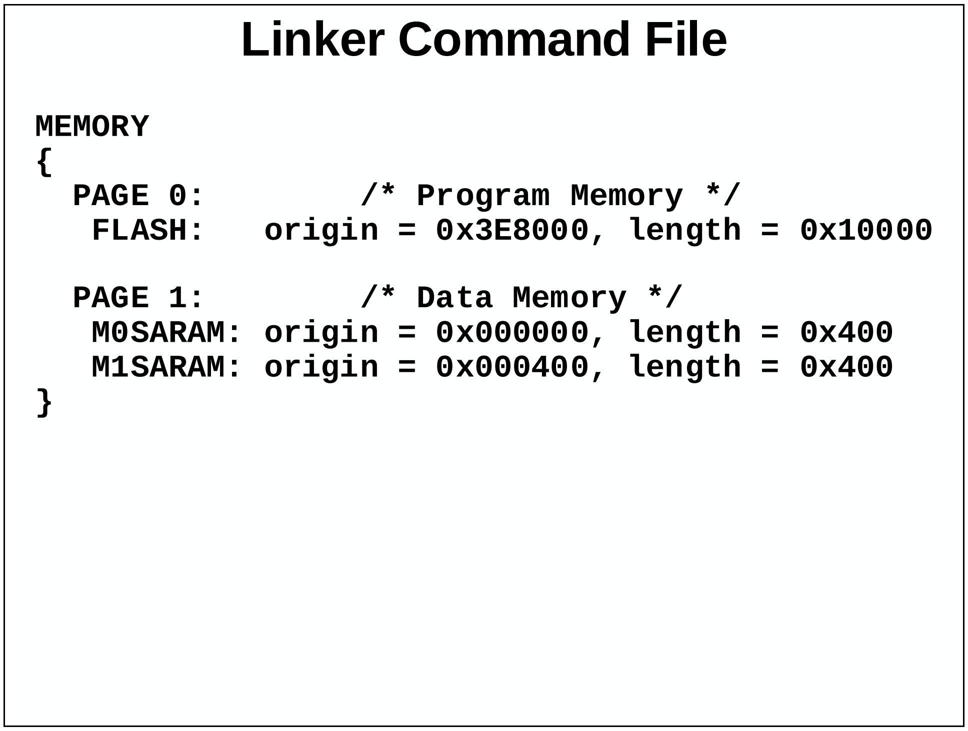
```
MEMORY
{
   FLASH:  origin = 0x3E8000 , length = 0x010000
}
```

Each memory segment is defined using the above format. If you added M0SARAM and M1SARAM, it would look like:

```
MEMORY
{
   M0SARAM:    origin = 0x000000 , length = 0x0400
   M1SARAM:    origin = 0x000400 , length = 0x0400
}
```

Remember that the DSP has two memory maps: *Program*, and *Data*. Therefore, the MEMORY description must describe each of these separately. The loader uses the following syntax to delineate each of these:

Linker Page	TI Definition
Page 0	Program
Page 1	Data



Linker Command File

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

  PAGE 1:        /* Data Memory */
   M0SARAM: origin = 0x000000, length = 0x400
   M1SARAM: origin = 0x000400, length = 0x400
}
```

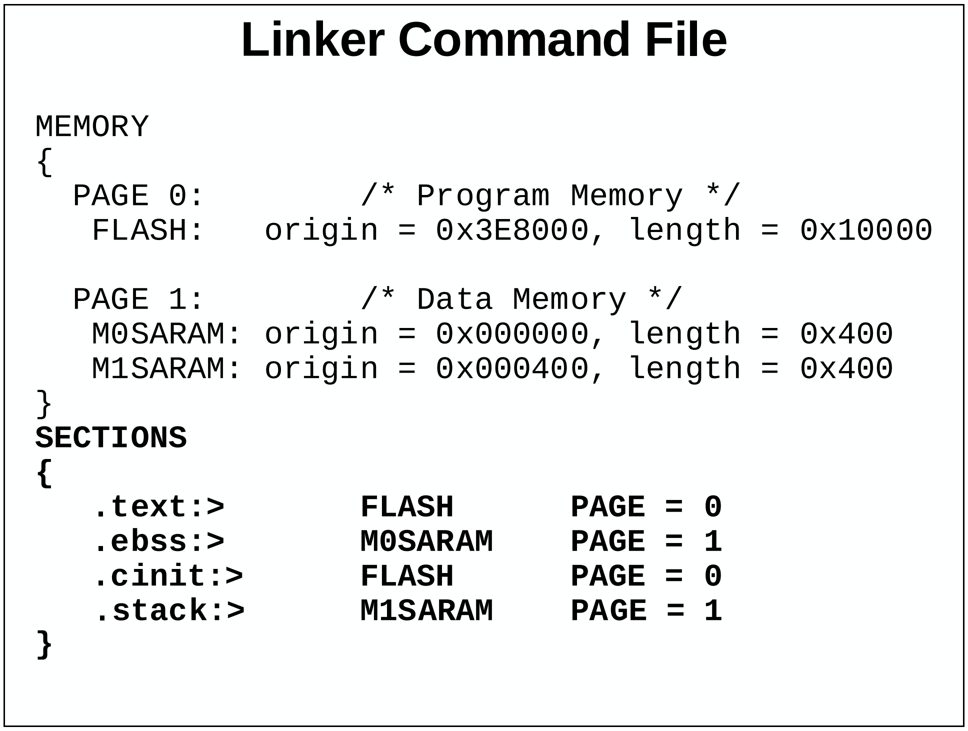
Section Placement

The SECTIONS section will specify how you want the sections to be distributed through memory. The following code is used to link the sections into the memory specified in the previous example:

```
SECTIONS
{
   .text:>  FLASH       PAGE 0
   .ebss:>  M0SARAM     PAGE 1
   .cinit:> FLASH       PAGE 0
   .stack:> M1SARAM     PAGE 1
}
```

The linker will gather all the `code` sections from all the files being linked together. Similarly, it will combine all 'like' sections.

Beginning with the first section listed, the linker will place it into the specified memory segment.

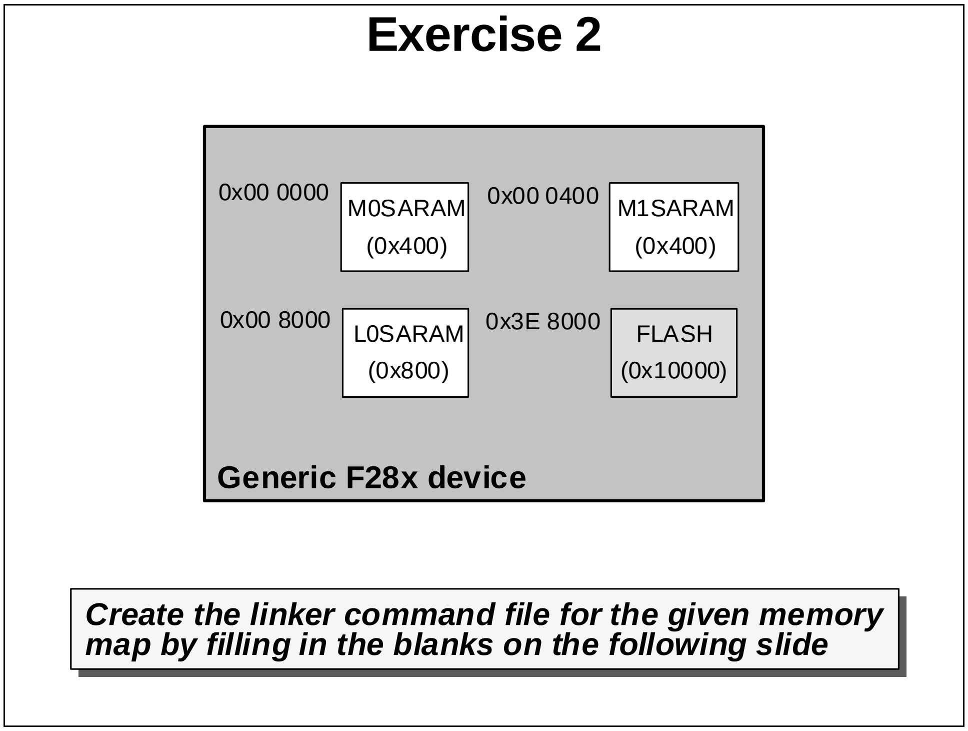


Linker Command File

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

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

Exercise 2

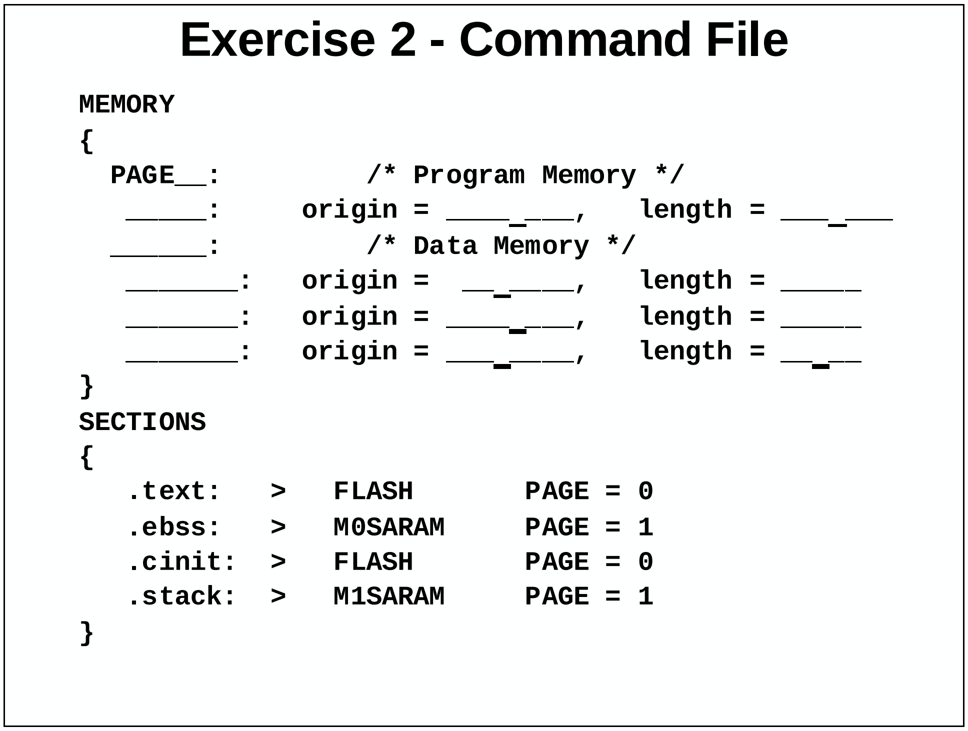
Looking at the following block diagram, and create a linker command file.



Exercise 2

Create the linker command file for the given memory map by filling in the blanks on the following slide

Fill in the blanks:

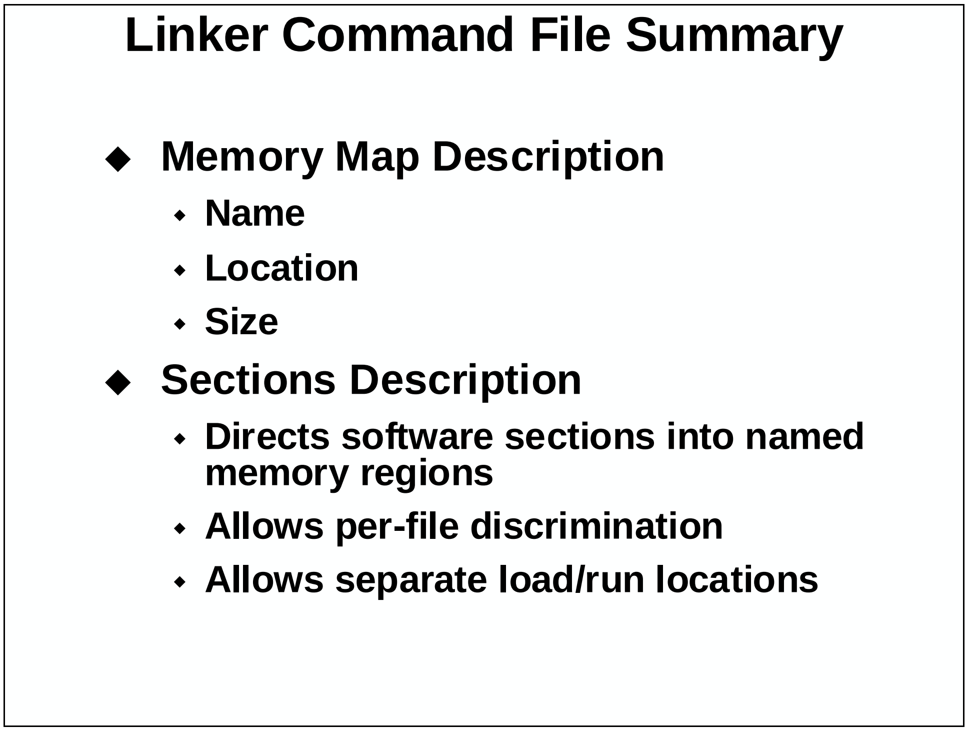


Exercise 2 - Command File

```
MEMORY
{
  PAGE__:         /* Program Memory */
   _____:     origin = ________,   length = _______
  ______:         /* Data Memory */
   _______:   origin =  _______,   length = _____
   _______:   origin = ________,   length = _____
   _______:   origin = ________,   length = _____
}
SECTIONS
{
   .text:   >   FLASH       PAGE = 0
   .ebss:   >   M0SARAM     PAGE = 1
   .cinit:  >   FLASH       PAGE = 0
   .stack:  >   M1SARAM     PAGE = 1
}
```

Summary: Linker Command File

The linker command file (`.cmd`) contains the inputs — commands — for the linker. This information is summarized below:



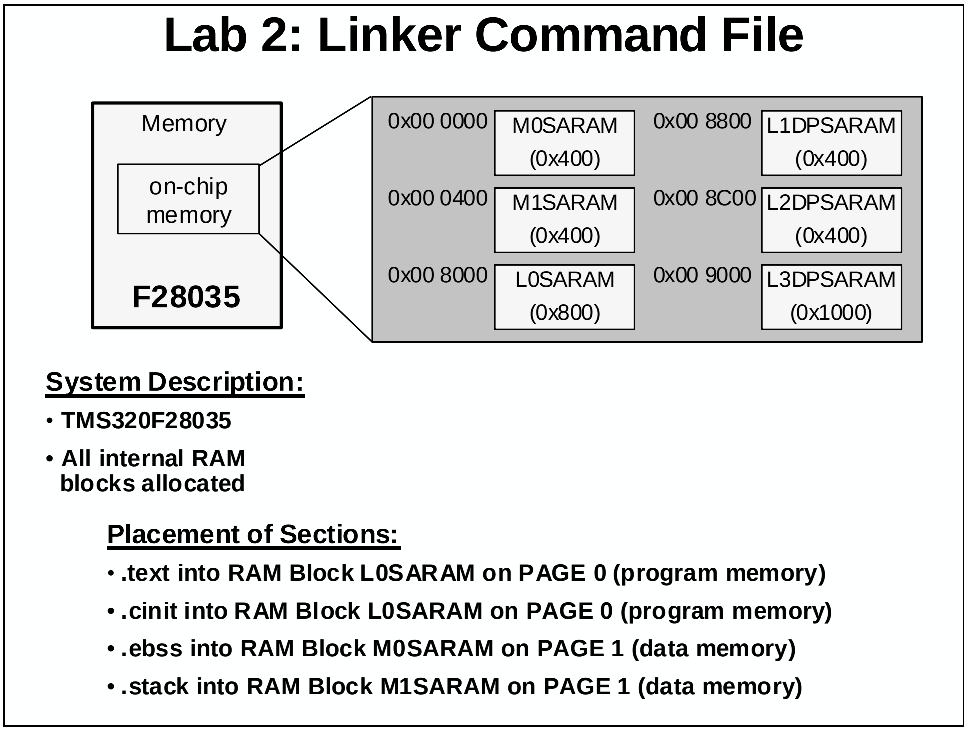
Linker Command File Summary

- **Memory Map Description**
  - **Name**
  - **Location**
  - **Size**
- **Sections Description**
  - **Directs software sections into named memory regions**
  - **Allows per-file discrimination**
  - **Allows separate load/run locations**

Lab 2: Linker Command File

➢ Objective

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



Lab 2: Linker Command File

0x00 0000	M0SARAM (0x400)	0x00 8800	L1DPSARAM (0x400)
0x00 0400	M1SARAM (0x400)	0x00 8C00	L2DPSARAM (0x400)
0x00 8000	L0SARAM (0x800)	0x00 9000	L3DPSARAM (0x1000)

System Description:

- **TMS320F28035**
- **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)**

System Description

- TMS320F28035
- 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

Create a New Project

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

2. Connect to the target.

   Click: `Debug → Connect`

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

3. A *project* contains all the files you will need to develop an executable output file (`.out`) which can be run on the MCU hardware. Let's create a new project for this lab. On the menu bar click:

   `Project → New`

   type `Lab2` in the project name field and make sure the save in location is: `C:\C28x\Labs\Lab2`, then click `Finish`. This will create a *.pjt* file which will invoke all the necessary tools (compiler, assembler, linker) to build your project. It will also create a `debug` folder that will hold immediate output files.

4. Add the C file to the new project. Click:

   `Project → Add Files to Project…`

   and make sure you're looking in `C:\C28x\Labs\Lab2`. Change the "files of type" to view C source files (`*.c`) and select `Lab2.c` and click `OPEN`. This will add the file `Lab2.c` to your newly created project.

5. Add `Lab2.cmd` to the project using the same procedure. This file will be edited during the lab exercise.

6. In the project window on the left, click the plus sign (+) to the left of `Project`. Now, click on the plus sign next to `Lab2.pjt`. Notice that the `Lab2.cmd` file is listed. Click on the plus sign next to `Source` to see the current source file list (i.e. `Lab2.c`).

Project Build Options

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

   Click: `Project → Build Options…`

8. Select the Linker tab. Notice that .out and .map files are being created. The .out file is the executable code that will be loaded into the MCU. The .map file will contain a linker report showing memory usage and section addresses in memory.

9. Set the Stack Size to 0x200.

10. Next, setup the compiler run-time support library. In the Libraries Category, find the `Include Libraries (-l)` box and enter: `rts2800_ml.lib`. Select `OK` and the Build Options window will close.

Edit the Linker Command File - Lab2a.cmd

11. To open and edit `Lab2.cmd`, double click on the filename in the project window.

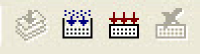
12. Edit the `Memory{}` declaration by describing the system memory shown on the “Lab2: Linker Command File” slide in the objective section of this lab exercise. Place the L0SARAM and L3DPSARAM memory blocks into program memory on page 0. Place the other memory blocks into data memory on page 1.

13. In the `Sections{}` area, notice that a section called .reset has already 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.

14. Place the sections defined on the slide into the appropriate memories via the `Sections{}` area. Save your work and close the file.

Build and Load the Project

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



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

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

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

17. Click the `“Build”` button and watch the tools run in the build window. Check for errors (we have deliberately put an error in `Lab2.c`). When you get an error, scroll the build window at the bottom of the Code Composer Studio screen until you see the error message (in red), and simply double-click the error message. The editor will automatically open the source file containing the error, and position the mouse cursor at the correct code line.

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

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

20. Under `Debug` on the menu bar click `“Go Main”`. This will run through the C-environment initialization routine in the `rts2800_ml.lib` and stop at `main()` in `Lab2.c`.

Debug Enviroment Windows

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

21. Open a *memory window* to view the global variable `“z”`.

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

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

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

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

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

    Click the “Watch Locals” tab and notice that the local variables `x` and `y` are already present. The watch window will always contain the local variables for the code function currently being executed.

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

23. We can also add global variables to the watch window if desired. Let's add the global variable “`z`”.

    Click the “Watch 1” tab at the bottom of the watch window. In the empty box in the “Name” column, type “`z`” and then enter. An ampersand is not used here. The watch window knows you are specifying a symbol.

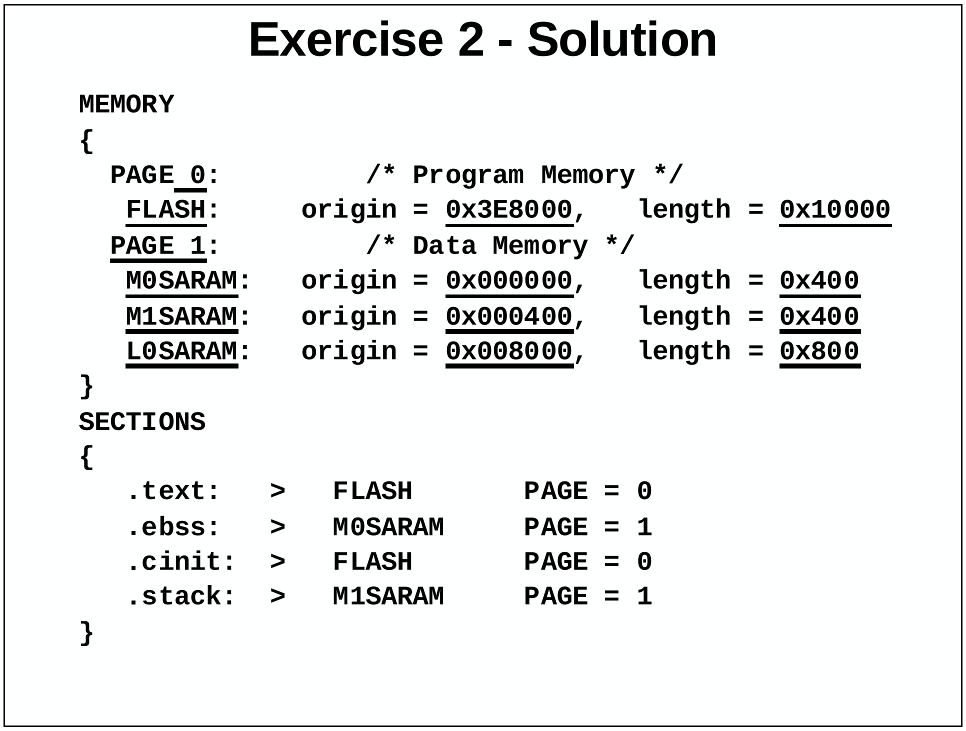
    Check that the watch window and memory window both report the same value for “`z`”. Trying changing the value in one window, and notice that the value also changes in the other window.

Single-stepping the Code

24. Click the “Watch Locals” tab at the bottom of the watch window. Single-step through `main()` by using the <F11> key (or you can use the `Single Step` button on the vertical toolbar). Check to see if the program is working as expected. What is the value for “`z`” when you get to the end of the program?

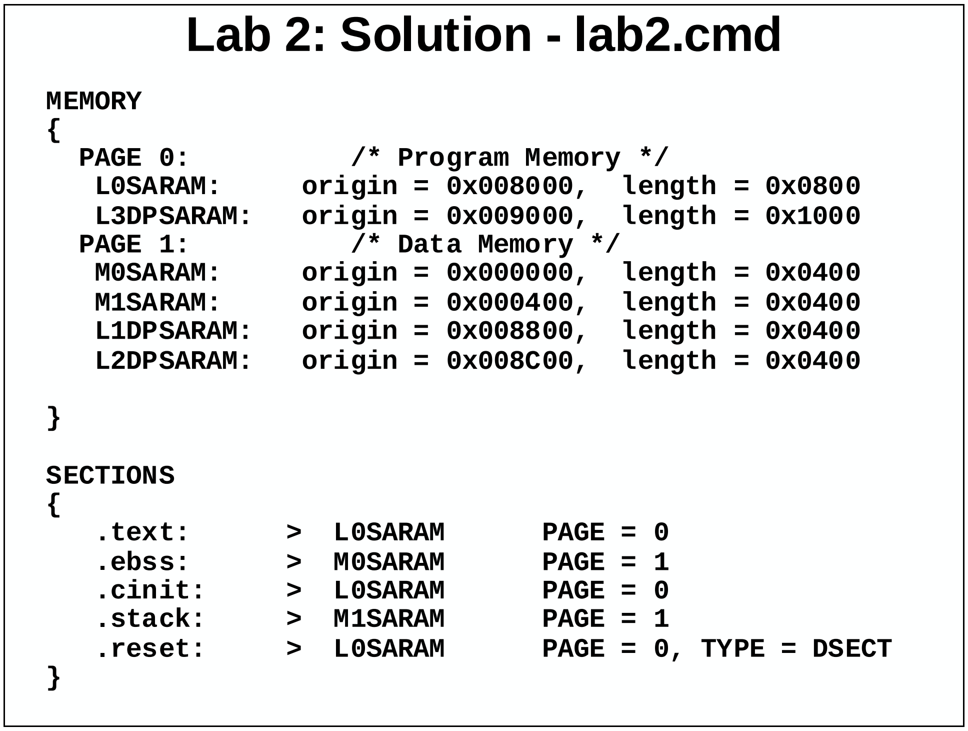
End of Exercise

Solutions



Exercise 2 - Solution

```
MEMORY
{
  PAGE 0:         /* Program Memory */
   FLASH:     origin = 0x3E8000,   length = 0x10000
  PAGE 1:         /* Data Memory */
   M0SARAM:   origin = 0x000000,   length = 0x400
   M1SARAM:   origin = 0x000400,   length = 0x400
   L0SARAM:   origin = 0x008000,   length = 0x800
}
SECTIONS
{
   .text:   >   FLASH       PAGE = 0
   .ebss:   >   M0SARAM     PAGE = 1
   .cinit:  >   FLASH       PAGE = 0
   .stack:  >   M1SARAM     PAGE = 1
}
```



Lab 2: Solution - lab2.cmd

```
MEMORY
{
  PAGE 0:          /* Program Memory */
   L0SARAM:     origin = 0x008000,  length = 0x0800
   L3DPSARAM:   origin = 0x009000,  length = 0x1000
  PAGE 1:          /* Data Memory */
   M0SARAM:     origin = 0x000000,  length = 0x0400
   M1SARAM:     origin = 0x000400,  length = 0x0400
   L1DPSARAM:   origin = 0x008800,  length = 0x0400
   L2DPSARAM:   origin = 0x008C00,  length = 0x0400

}

SECTIONS
{
   .text:      >  L0SARAM      PAGE = 0
   .ebss:      >  M0SARAM      PAGE = 1
   .cinit:     >  L0SARAM      PAGE = 0
   .stack:     >  M1SARAM      PAGE = 1
   .reset:     >  L0SARAM      PAGE = 0, TYPE = DSECT
}
```

Peripherial Registers Header Files

Introduction

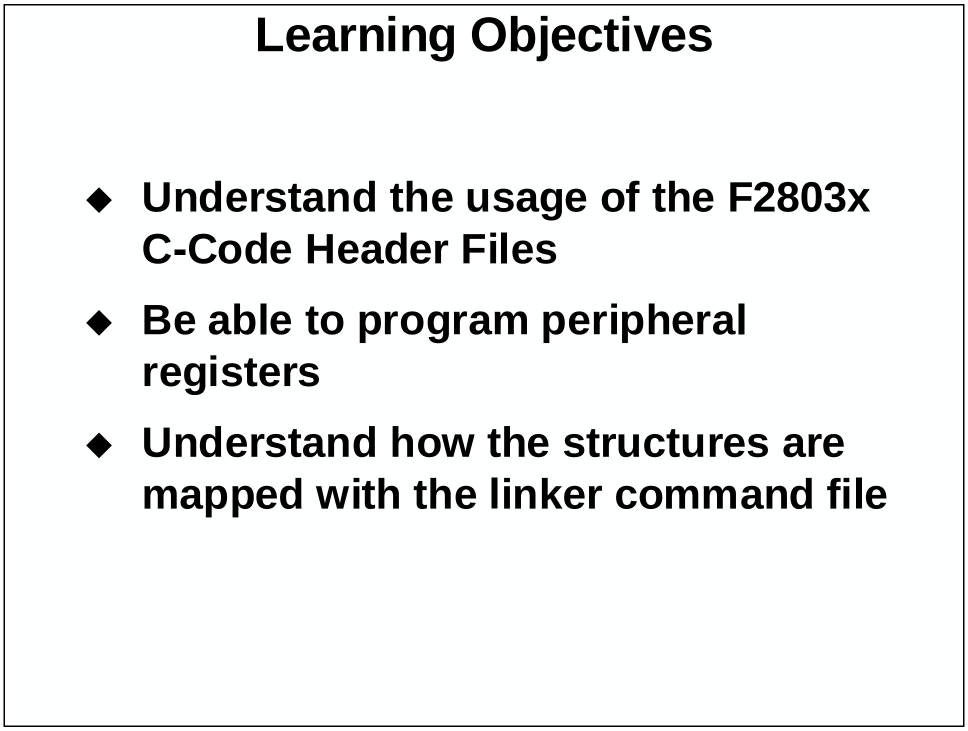
The purpose of the DSP2803x C-code header files is to simplify the programming of the many peripherals on the F28x device. Typically, to program a peripheral the programmer needs to write the appropriate values to the different fields within a control register. In its simplest form, the process consists of writing a hex value (or masking a bit field) to the correct address in memory. But, since this can be a burdensome and repetitive task, the C-code header files were created to make this a less complicated task.

The DSP2803x C-code header files are part of a library consisting of C functions, macros, peripheral structures, and variable definitions. Together, this set of files is known as the 'header files.'

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

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

Learning Objectives

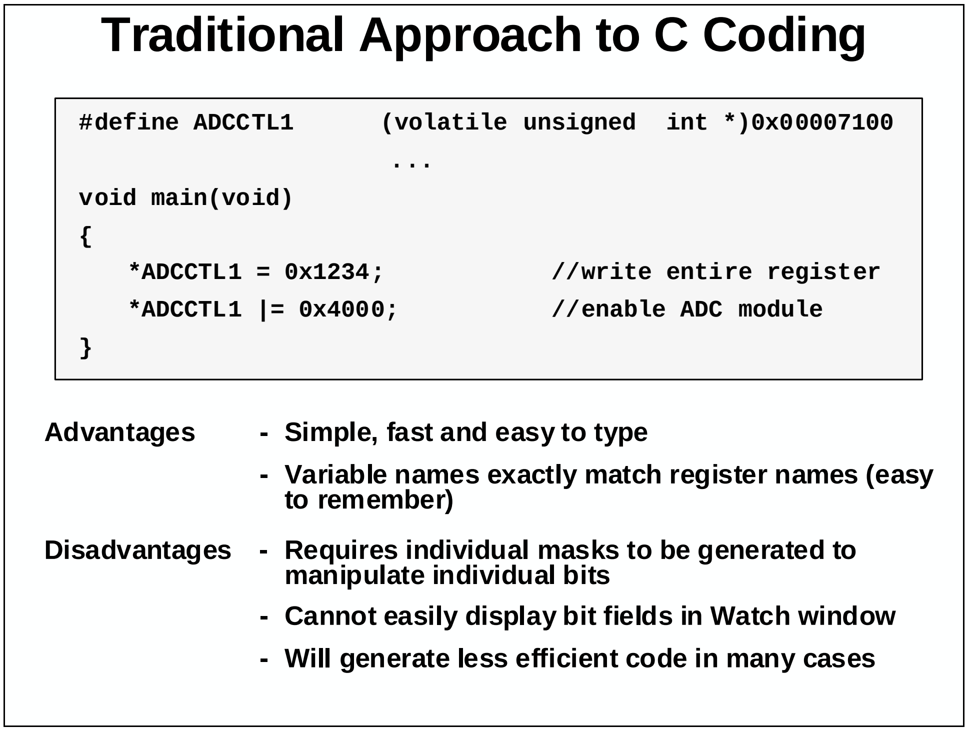


Learning Objectives

- **Understand the usage of the F2803x C-Code Header Files**
- **Be able to program peripheral registers**
- **Understand how the structures are mapped with the linker command file**

Module Topics

Traditional and Structure Approach to C Coding

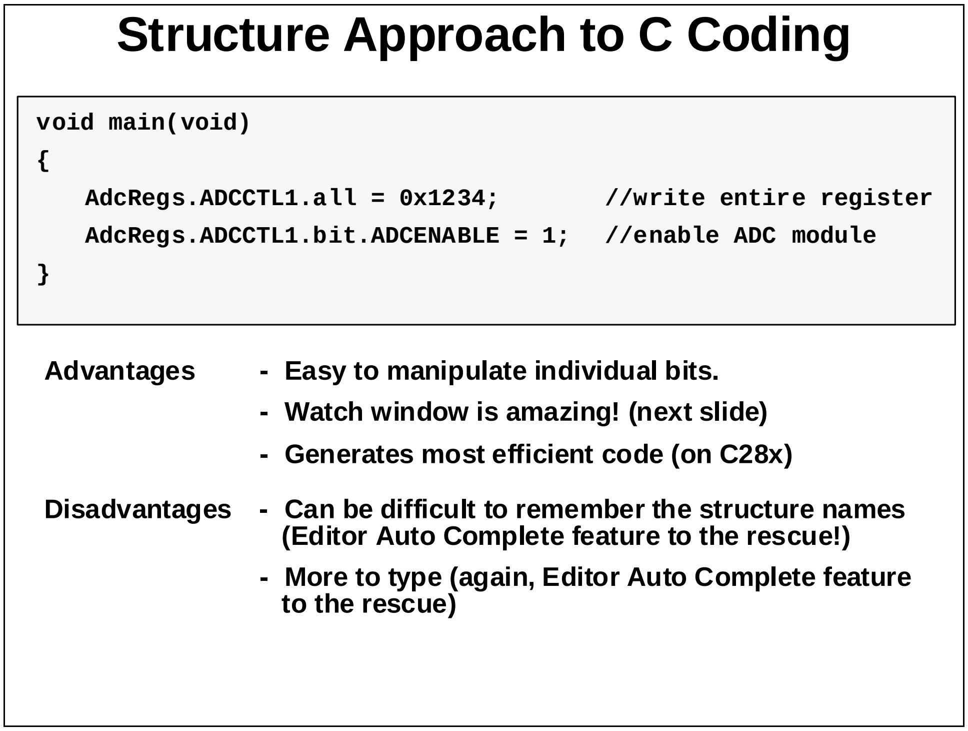


Traditional Approach to C Coding

```
#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 Watch window
- Will generate less efficient code in many cases



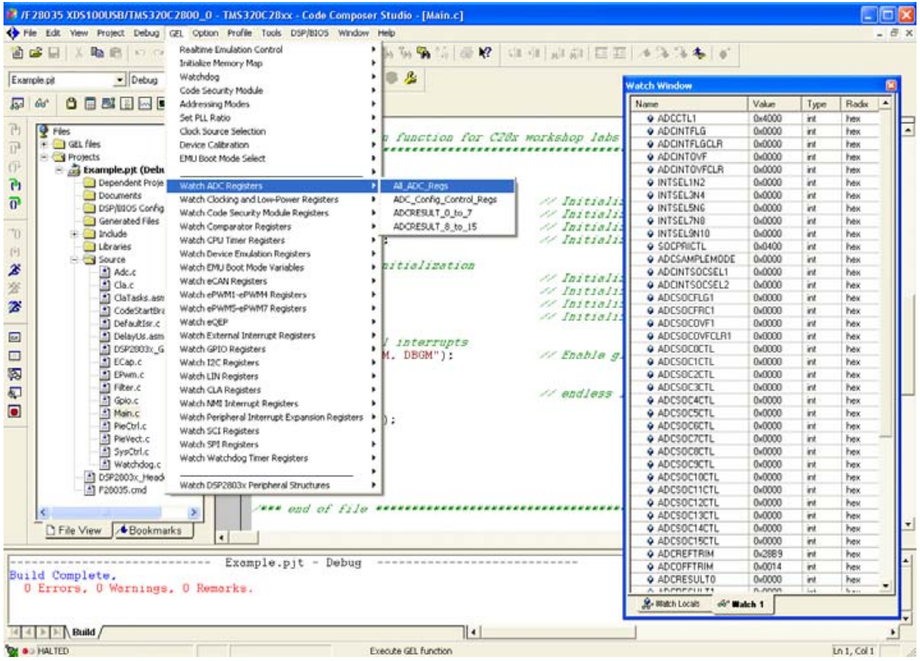
Structure Approach to C Coding

```
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)

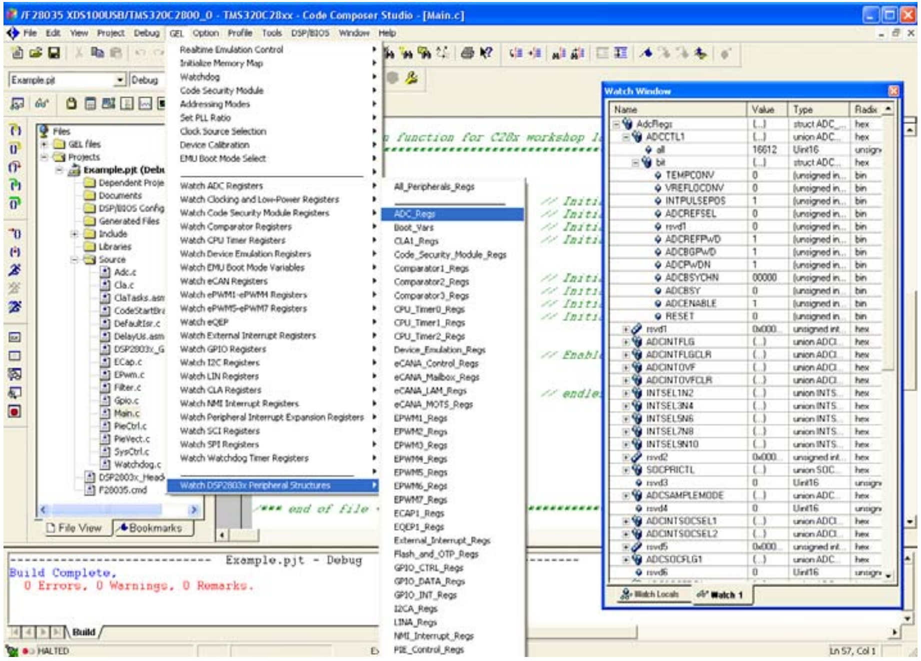
The CCS Watch Window using #define


/F28035 XDS100USB/TMS320C2800_0 - TMS320C28xx - Code Composer Studio - [Main.c]
File Edit View Project Debug GEL Option Profile Tools DSP/BIOS Window Help
Realtime Emulation Control
Initialize Memory Map
Watchdog
Code Security Module
Addressing Modes
Set PLL Ratio
Clock Source Selection
Device Calibration
EMU Boot Mode Select
Watch ADC Registers
Watch Clocking and Low-Power Registers
Watch Code Security Module Registers
Watch Comparator Registers
Watch CPU Timer Registers
Watch Device Emulation Registers
Watch EMU Boot Mode Variables
Watch eCAN Registers
Watch ePWM1-ePWM4 Registers
Watch ePWM5-ePWM7 Registers
Watch eQEP
Watch External Interrupt Registers
Watch GPIO Registers
Watch I2C Registers
Watch LIN Registers
Watch CLA Registers
Watch NMI Interrupt Registers
Watch Peripheral Interrupt Expansion Registers
Watch SCI Registers
Watch SPI Registers
Watch Watchdog Timer Registers
Watch DSP2803x Peripheral Structures
All_ADC_Regs
ADC_Config_Control_Regs
ADCRESULT_0_to_7
ADCRESULT_8_to_15
Example.pjt
Debug
File View
Bookmarks
Build
Watch Window
Watch Locals
Watch 1
HALTED
Execute GEL function
Ln 1, Col 1

Name	Value	Type	Radix
ADCCTL1	0x4000	int	hex
ADCINTFLG	0x0000	int	hex
ADCINTFLGCLR	0x0000	int	hex
ADCINTOVF	0x0000	int	hex
ADCINTOVFCLR	0x0000	int	hex
INTSEL1N2	0x0000	int	hex
INTSEL3N4	0x0000	int	hex
INTSEL5N6	0x0000	int	hex
INTSEL7N8	0x0000	int	hex
INTSEL9N10	0x0000	int	hex
SOCPRICTL	0x0400	int	hex
ADCSAMPLEMODE	0x0000	int	hex
ADCINTSOCSEL1	0x0000	int	hex
ADCINTSOCSEL2	0x0000	int	hex
ADCSOCFLG1	0x0000	int	hex
ADCSOCFRC1	0x0000	int	hex
ADCSOCOVF1	0x0000	int	hex
ADCSOCOVFCLR1	0x0000	int	hex
ADCSOC0CTL	0x0000	int	hex
ADCSOC1CTL	0x0000	int	hex
ADCSOC2CTL	0x0000	int	hex
ADCSOC3CTL	0x0000	int	hex
ADCSOC4CTL	0x0000	int	hex
ADCSOC5CTL	0x0000	int	hex
ADCSOC6CTL	0x0000	int	hex
ADCSOC7CTL	0x0000	int	hex
ADCSOC8CTL	0x0000	int	hex
ADCSOC9CTL	0x0000	int	hex
ADCSOC10CTL	0x0000	int	hex
ADCSOC11CTL	0x0000	int	hex
ADCSOC12CTL	0x0000	int	hex
ADCSOC13CTL	0x0000	int	hex
ADCSOC14CTL	0x0000	int	hex
ADCSOC15CTL	0x0000	int	hex
ADCREFTRIM	0x28B9	int	hex
ADCOFFTRIM	0x0014	int	hex
ADCRESULT0	0x0000	int	hex

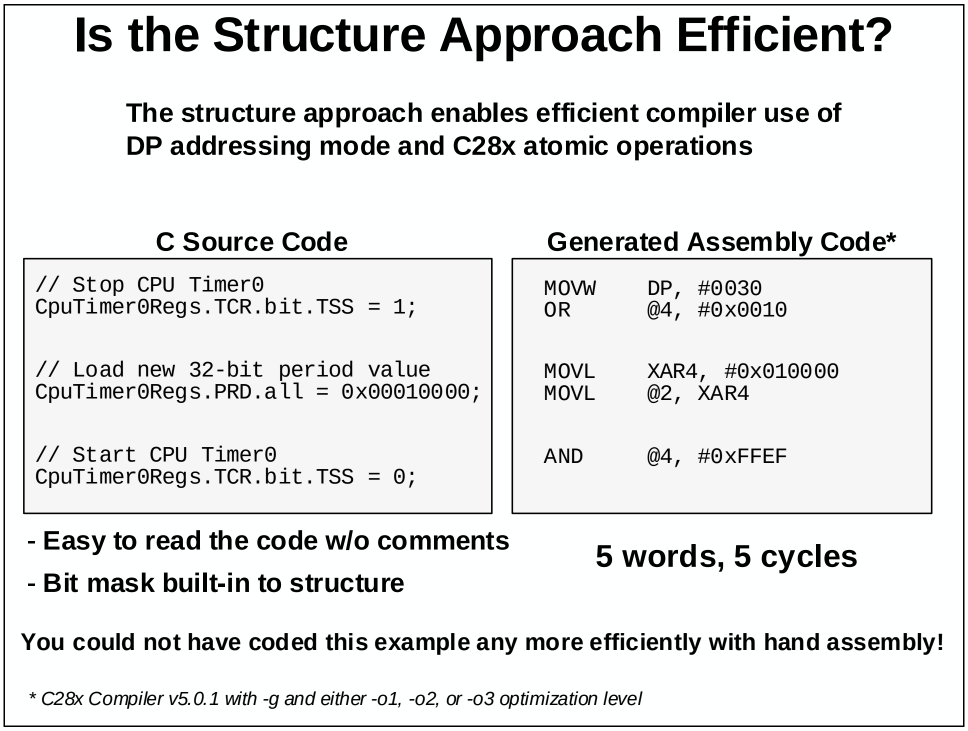
```
------------------------------  Example.pjt - Debug  ------------------------------
Build Complete,
  0 Errors, 0 Warnings, 0 Remarks.
```

The CCS Watch Window using Structures



Name	Value	Type	Radix
AdcRegs	{...}	struct ADC_...	hex
ADCCTL1	{...}	union ADC...	hex
all	16612	Uint16	unsign
bit	{...}	struct ADC...	hex
TEMPCONV	0	(unsigned in...	bin
VREFLOCONV	0	(unsigned in...	bin
INTPULSEPOS	1	(unsigned in...	bin
ADCREFSEL	0	(unsigned in...	bin
rsvd1	0	(unsigned in...	bin
ADCREFPWD	1	(unsigned in...	bin
ADCBGPWD	1	(unsigned in...	bin
ADCPWDN	1	(unsigned in...	bin
ADCBSYCHN	00000	(unsigned in...	bin
ADCBSY	0	(unsigned in...	bin
ADCENABLE	1	(unsigned in...	bin
RESET	0	(unsigned in...	bin
rsvd1	0x000...	unsigned int...	hex
ADCINTFLG	{...}	union ADCI...	hex
ADCINTFLGCLR	{...}	union ADCI...	hex
ADCINTOVF	{...}	union ADCI...	hex
ADCINTOVFCLR	{...}	union ADCI...	hex
INTSEL1N2	{...}	union INTS...	hex
INTSEL3N4	{...}	union INTS...	hex
INTSEL5N6	{...}	union INTS...	hex
INTSEL7N8	{...}	union INTS...	hex
INTSEL9N10	{...}	union INTS...	hex
rsvd2	0x000...	unsigned int...	hex
SOCPRICTL	{...}	union SOC...	hex
rsvd3	0	Uint16	unsign
ADCSAMPLEMODE	{...}	union ADC...	hex
rsvd4	0	Uint16	unsign
ADCINTSOCSEL1	{...}	union ADCI...	hex
ADCINTSOCSEL2	{...}	union ADCI...	hex
rsvd5	0x000...	unsigned int...	hex
ADCSOCFLG1	{...}	union ADC...	hex
rsvd6	0	Uint16	unsign

```
------------------------------  Example.pjt - Debug  ------------------------------
Build Complete,
  0 Errors, 0 Warnings, 0 Remarks.
```



Is the Structure Approach Efficient?

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

C Source Code

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


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


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

Generated Assembly Code*

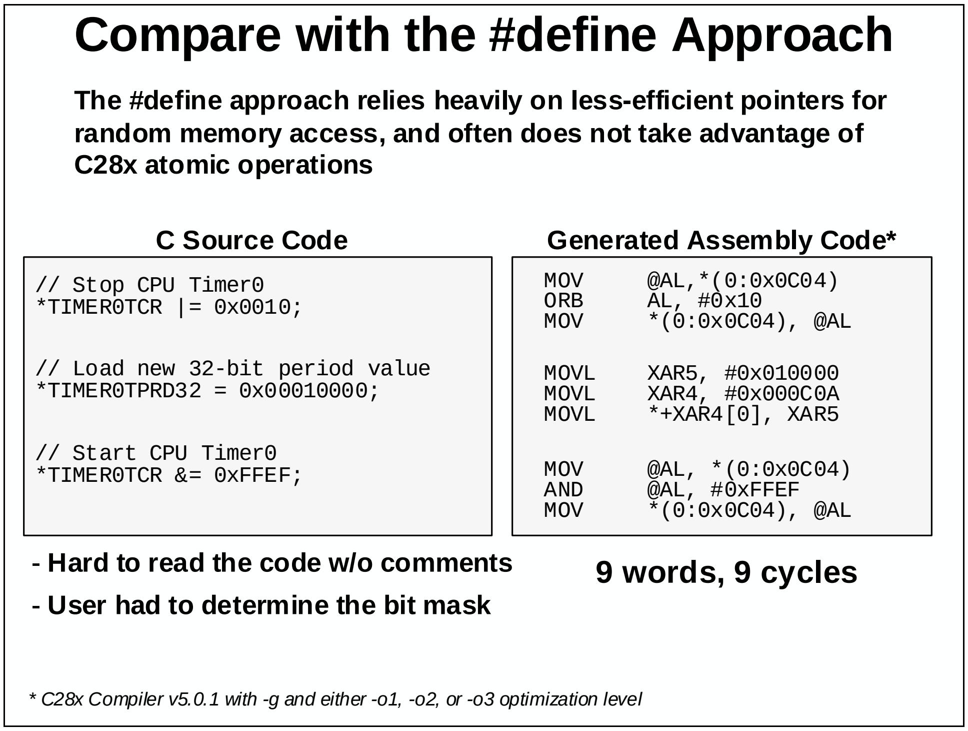
```
MOVW    DP, #0030
OR      @4, #0x0010


MOVL    XAR4, #0x010000
MOVL    @2, XAR4


AND     @4, #0xFFEF
```

- **Easy to read the code w/o comments**
- **Bit mask built-in to structure**

5 words, 5 cycles

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

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



Compare with the #define Approach

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

C Source Code

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


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


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

Generated Assembly Code*

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

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

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

- **Hard to read the code w/o comments**
- **User had to determine the bit mask**

9 words, 9 cycles

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

Naming Conventions

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

Structure Naming Conventions

- **The DSP2803x header files define:**
  - **All of the peripheral structures**
  - **All of the register names**
  - **All of the bit field names**
  - **All of the register addresses**

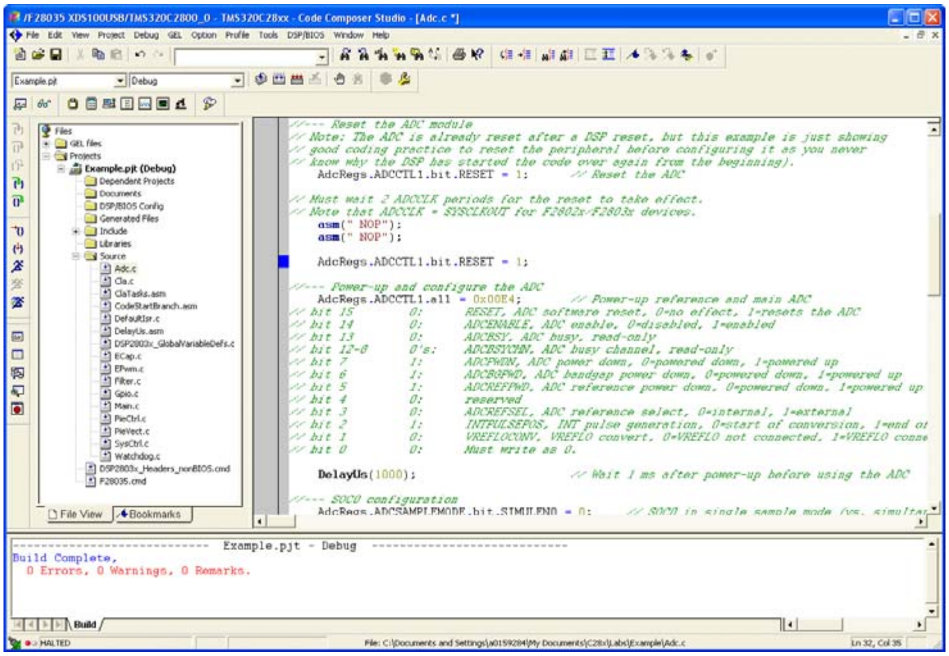
```
PeripheralName.RegisterName.all               // Access full 16 or 32-bit register
PeripheralName.RegisterName.half.LSW          // Access low 16-bits of 32-bit register
PeripheralName.RegisterName.half.MSW          // Access high 16-bits of 32-bit register
PeripheralName.RegisterName.bit.FieldName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

Notes: [1] “PeripheralName” are assigned by TI and found in the DSP2803x 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!

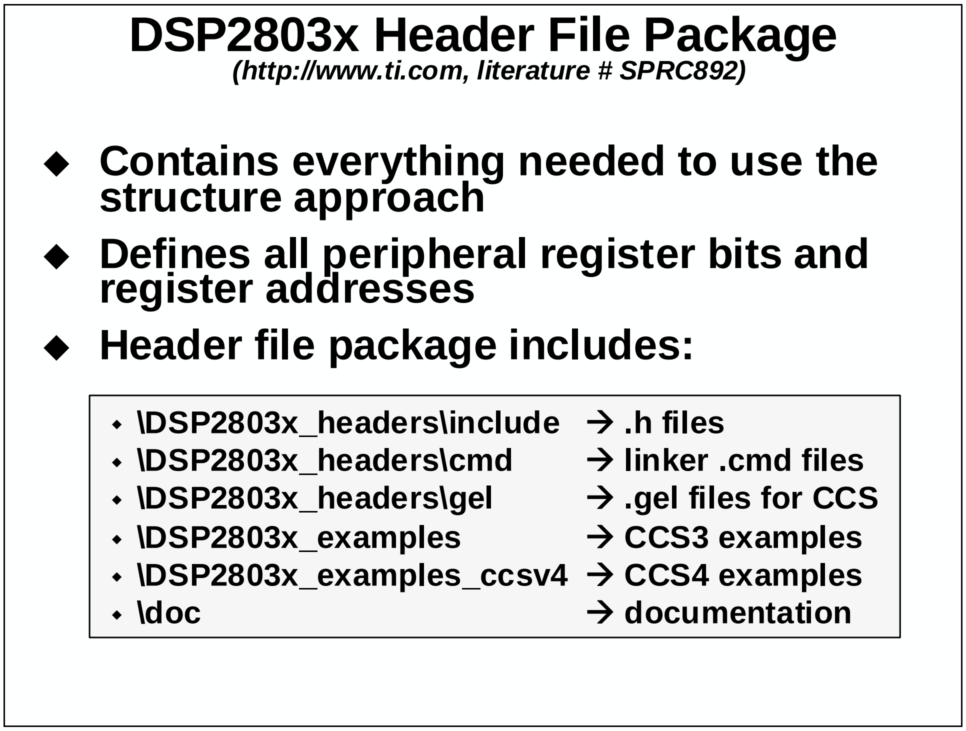


```
//--- Reset the ADC module
// Note: The ADC is already reset after a DSP reset, but this example is just showing
// good coding practice to reset the peripheral before configuring it as you never
// know why the DSP has started the code over again from the beginning).
    AdcRegs.ADCCTL1.bit.RESET = 1;          // Reset the ADC

// Must wait 2 ADCCLK periods for the reset to take effect.
// Note that ADCCLK = SYSCLKOUT for F2802x/F2803x devices.
    asm(" NOP");
    asm(" NOP");

    AdcRegs.ADCCTL1.bit.RESET = 1;

//--- Power-up and configure the ADC
    AdcRegs.ADCCTL1.all = 0x00E4;           // Power-up reference and main ADC
// bit 15        0:      RESET, ADC software reset, 0=no effect, 1=resets the ADC
// bit 14        0:      ADCENABLE, ADC enable, 0=disabled, 1=enabled
// bit 13        0:      ADCBSY, ADC busy, read-only
// bit 12-8      0's:    ADCBSYCHN, ADC busy channel, read-only
// bit 7         1:      ADCPWDN, ADC power down, 0=powered down, 1=powered up
// bit 6         1:      ADCBGPWD, ADC bandgap power down, 0=powered down, 1=powered up
// bit 5         1:      ADCREFPWD, ADC reference power down, 0=powered down, 1=powered up
// bit 4         0:      reserved
// bit 3         0:      ADCREFSEL, ADC reference select, 0=internal, 1=external
// bit 2         1:      INTPULSEPOS, INT pulse generation, 0=start of conversion, 1=end of
// bit 1         0:      VREFLOCONV, VREFLO convert, 0=VREFLO not connected, 1=VREFLO conne
// bit 0         0:      Must write as 0.

    DelayUs(1000);                          // Wait 1 ms after power-up before using the ADC

//--- SOC0 configuration
    AdcRegs.ADCSAMPLEMODE.bit.SIMULEN0 = 0;     // SOC0 in single sample mode (vs. simultar
```

```
-----------------------------  Example.pjt - Debug  -----------------------------
Build Complete,
  0 Errors, 0 Warnings, 0 Remarks.
```

F2803x C-Code Header Files

The C-code header files consists of .h, c source files, linker command files, and other useful example programs, documentations and add-ins for Code Composer Studio.



DSP2803x Header File Package

(http://www.ti.com, literature # SPRC892)

- **Contains everything needed to use the structure approach**
- **Defines all peripheral register bits and register addresses**
- **Header file package includes:**
  - **\DSP2803x_headers\include → .h files**
  - **\DSP2803x_headers\cmd → linker .cmd files**
  - **\DSP2803x_headers\gel → .gel files for CCS**
  - **\DSP2803x_examples → CCS3 examples**
  - **\DSP2803x_examples_ccsv4 → CCS4 examples**
  - **\doc → documentation**

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

A DSP2803x_Peripheral.gel GEL file can provide a pull down menu to load peripheral data structures into a watch window. Code Composer Studio can load a GEL file automatically. To include fuctions to the standard F28035.gel that is part of Code Composer Studio, add:

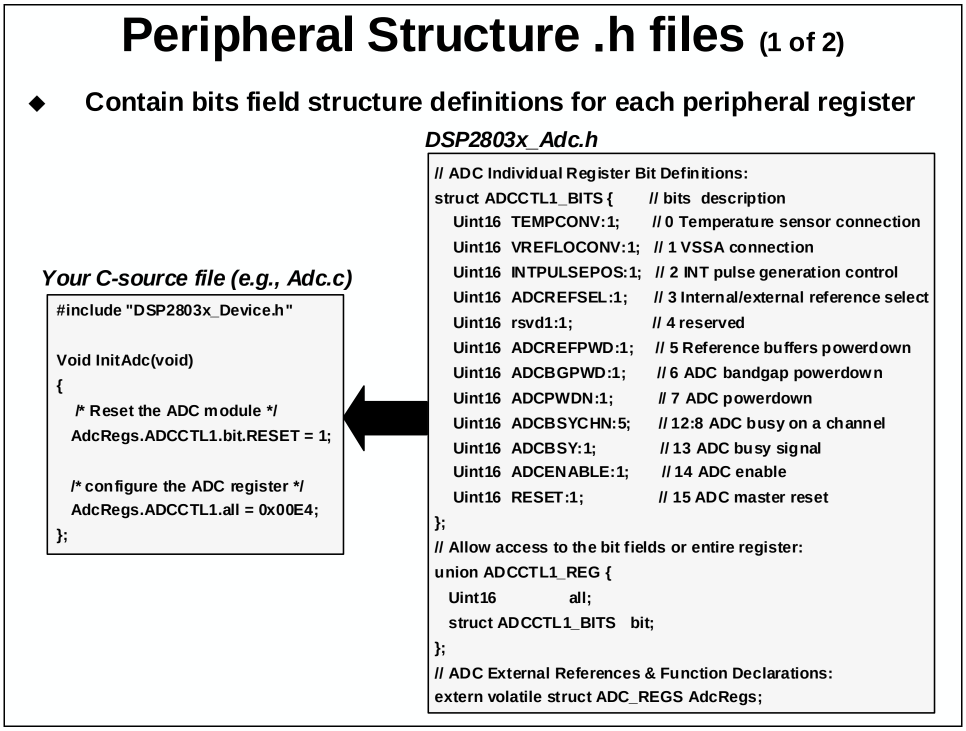
GEL_LoadGel("base_path/gel/DSP2803x_Peripheral.gel")

The GEL file can also be loaded during a Code Composer Studio session by clicking:

```
File → Load GEL…
```

Peripheral Structure .h File

The DSP2803x_Device.h header file is the main include file. By including this file in the .c source code, all of the peripheral specific .h header files are automatically included. Of course, each specific .h header file can be included individually in an application that does not use all the header files, or you can comment out the ones you do not need. (Also includes typedef statements).



Peripheral Structure .h files (1 of 2)

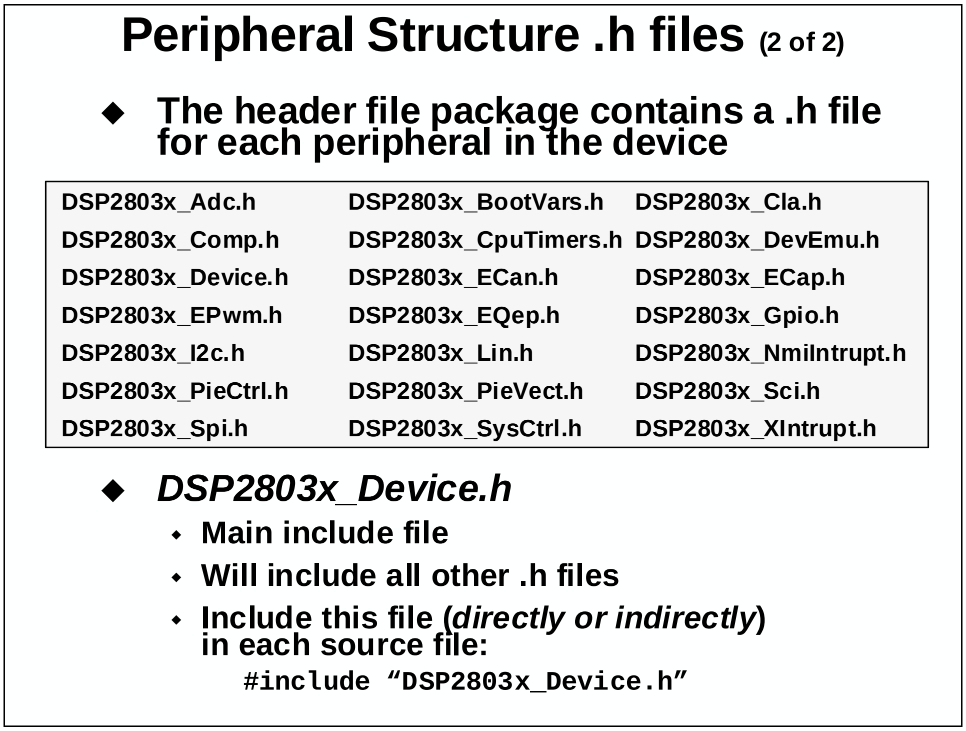
- Contain bits field structure definitions for each peripheral register

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

```
#include "DSP2803x_Device.h"

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

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

DSP2803x_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

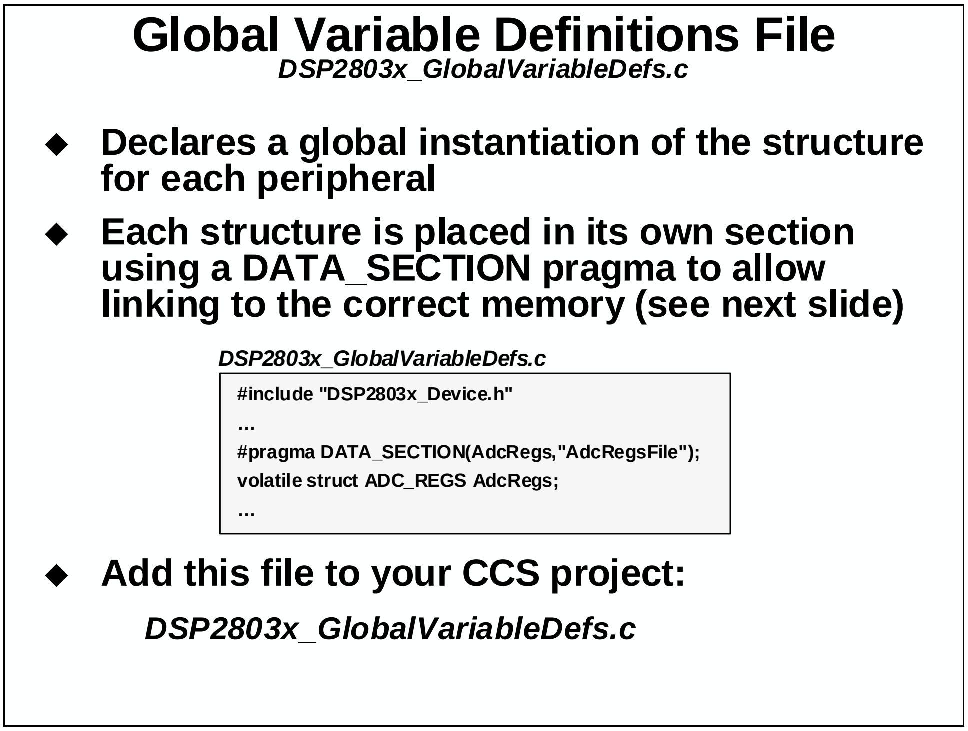
DSP2803x_Adc.h	DSP2803x_BootVars.h	DSP2803x_Cla.h
DSP2803x_Comp.h	DSP2803x_CpuTimers.h	DSP2803x_DevEmu.h
DSP2803x_Device.h	DSP2803x_ECan.h	DSP2803x_ECap.h
DSP2803x_EPwm.h	DSP2803x_EQep.h	DSP2803x_Gpio.h
DSP2803x_I2c.h	DSP2803x_Lin.h	DSP2803x_NmiIntrupt.h
DSP2803x_PieCtrl.h	DSP2803x_PieVect.h	DSP2803x_Sci.h
DSP2803x_Spi.h	DSP2803x_SysCtrl.h	DSP2803x_XIntrupt.h

- *DSP2803x_Device.h*
  - Main include file
  - Will include all other .h files
  - Include this file (*directly or indirectly*) in each source file:

```
#include “DSP2803x_Device.h”
```

Global Variable Definitions File

With DSP2803x_GlobalVariableDefs.c included in the project all the needed variable definitions are globally defined.



Global Variable Definitions File

DSP2803x_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)**

DSP2803x_GlobalVariableDefs.c

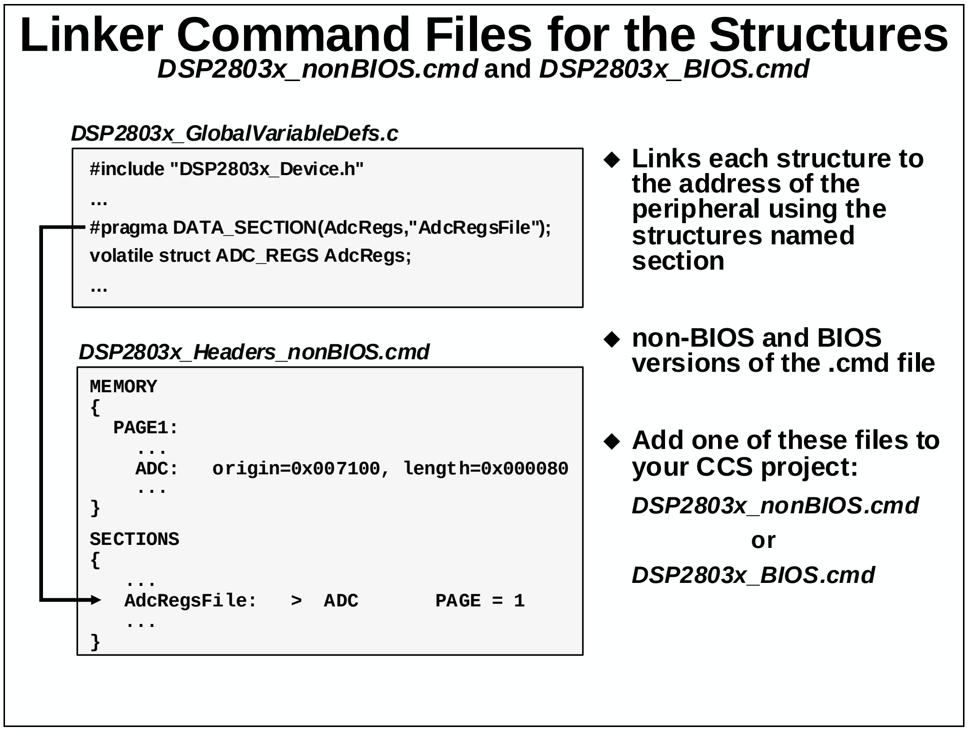
```
#include "DSP2803x_Device.h"
…
#pragma DATA_SECTION(AdcRegs,"AdcRegsFile");
volatile struct ADC_REGS AdcRegs;
…
```

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

  DSP2803x_GlobalVariableDefs.c

Mapping Structures to Memory

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



Linker Command Files for the Structures

DSP2803x_nonBIOS.cmd* and *DSP2803x_BIOS.cmd

DSP2803x_GlobalVariableDefs.c

```
#include "DSP2803x_Device.h"
...
#pragma DATA_SECTION(AdcRegs,"AdcRegsFile");
volatile struct ADC_REGS AdcRegs;
...
```

DSP2803x_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**
- **non-BIOS and BIOS versions of the .cmd file**
- **Add one of these files to your CCS project:**

  DSP2803x_nonBIOS.cmd

  or

  DSP2803x_BIOS.cmd

Linker Command File

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

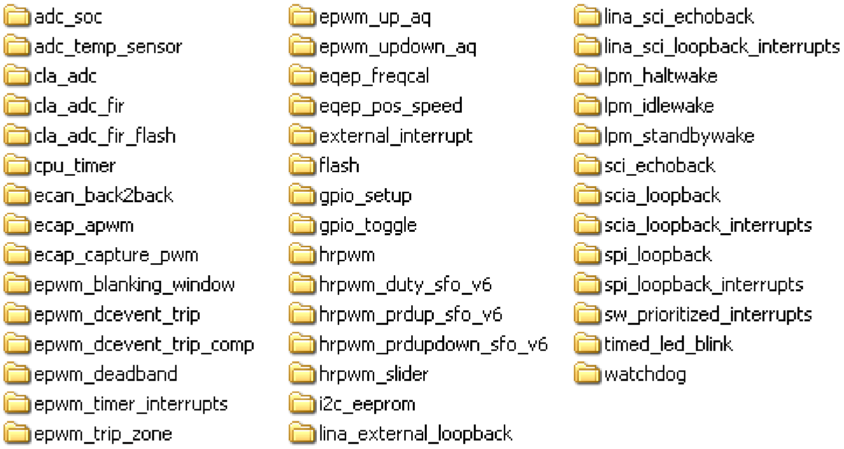
The user can modify their own linker command file, or use a pre-configured linker command file such as F28035.cmd. This file has the peripheral memory regions defined and tied to the individual peripheral.

Peripheral Specific Routines

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

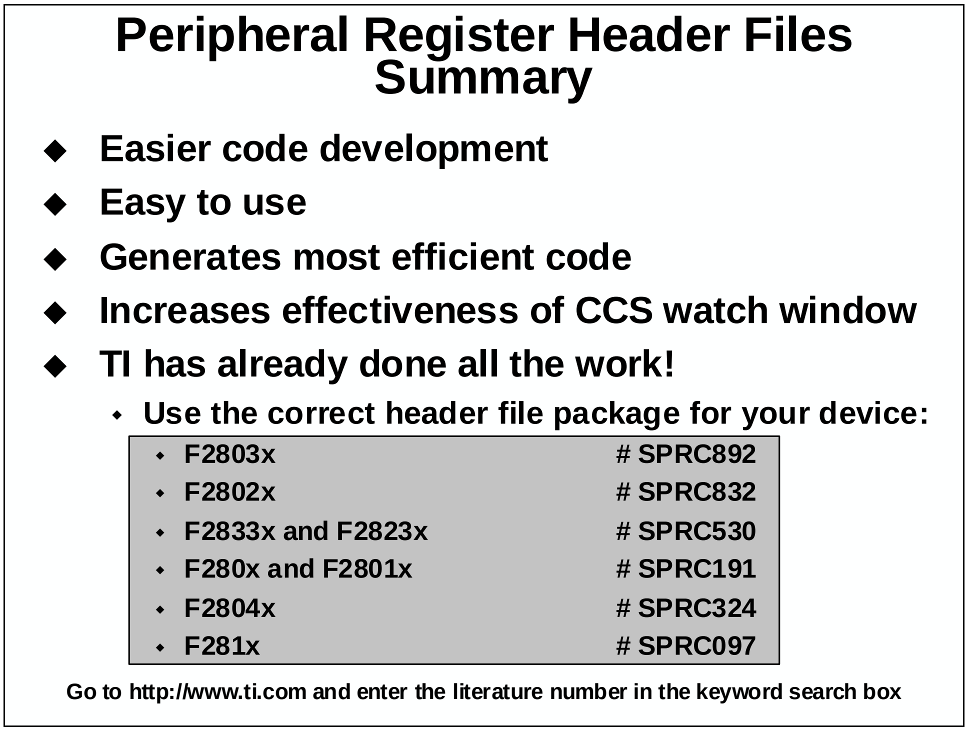
Peripheral Specific Examples

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



adc_soc
adc_temp_sensor
cla_adc
cla_adc_fir
cla_adc_fir_flash
cpu_timer
ecan_back2back
ecap_apwm
ecap_capture_pwm
epwm_blanking_window
epwm_dcevent_trip
epwm_dcevent_trip_comp
epwm_deadband
epwm_timer_interrupts
epwm_trip_zone
epwm_up_aq
epwm_updown_aq
eqep_freqcal
eqep_pos_speed
external_interrupt
flash
gpio_setup
gpio_toggle
hrpwm
hrpwm_duty_sfo_v6
hrpwm_prdup_sfo_v6
hrpwm_prdupdown_sfo_v6
hrpwm_slider
i2c_eeprom
lina_external_loopback
lina_sci_echoback
lina_sci_loopback_interrupts
lpm_haltwake
lpm_idlewake
lpm_standbywake
sci_echoback
scia_loopback
scia_loopback_interrupts
spi_loopback
spi_loopback_interrupts
sw_prioritized_interrupts
timed_led_blink
watchdog

Summary



Peripheral Register Header Files Summary

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

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

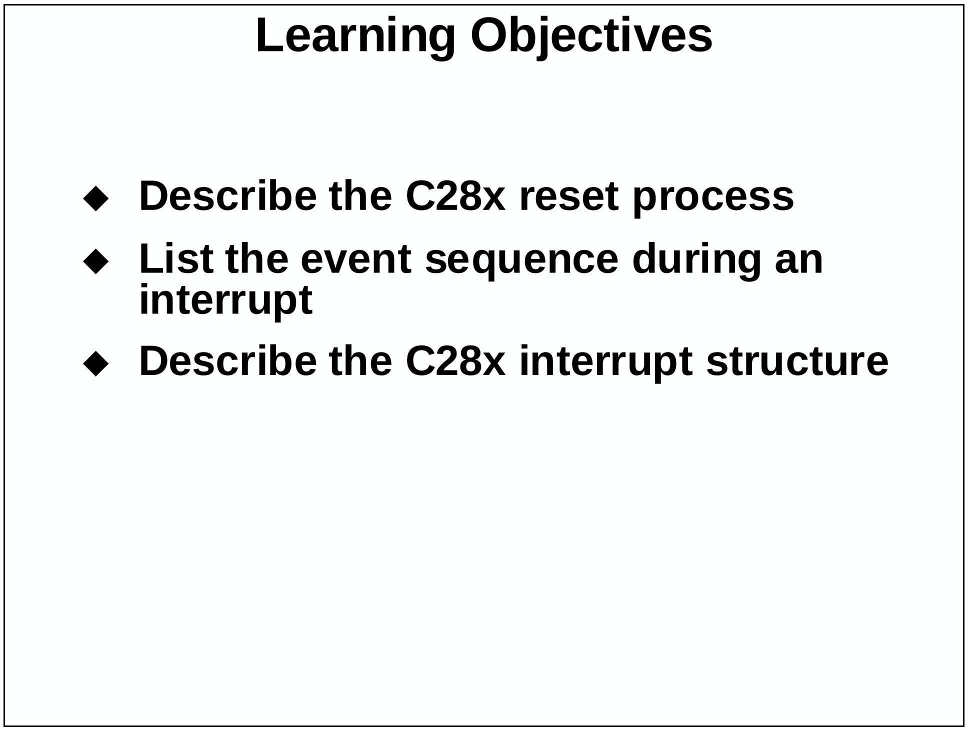
Go to http://www.ti.com and enter the literature number in the keyword search box

Reset and Interrupts

Introduction

This module describes the interrupt process and explains how the Peripheral Interrupt Expansion (PIE) works.

Learning Objectives

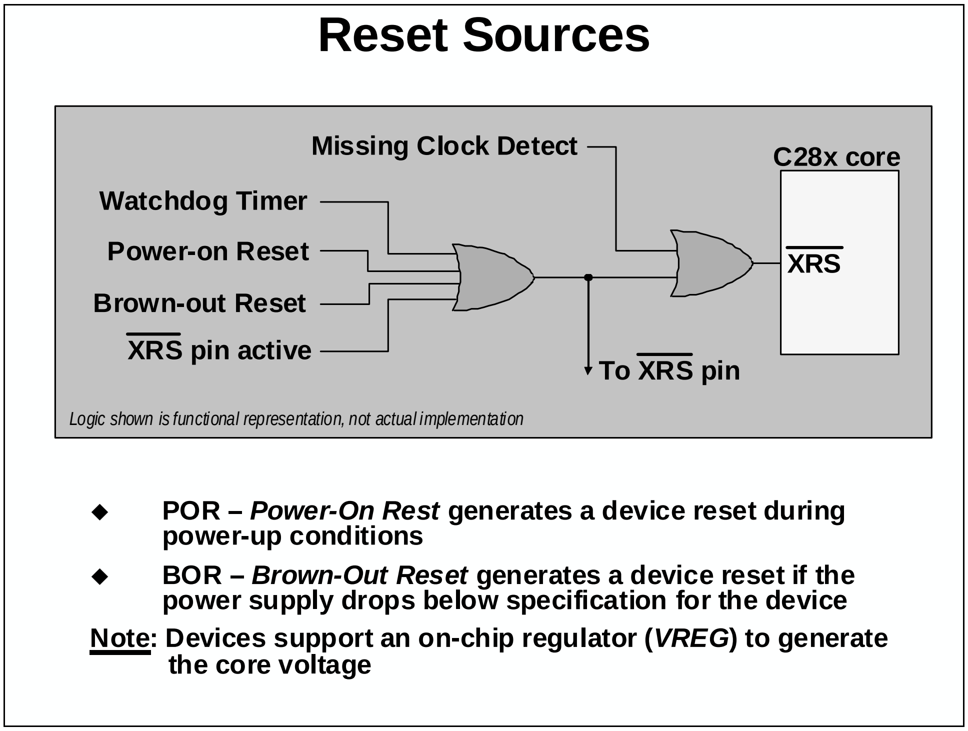


Learning Objectives

- Describe the C28x reset process
- List the event sequence during an interrupt
- Describe the C28x interrupt structure

Module Topics

Reset



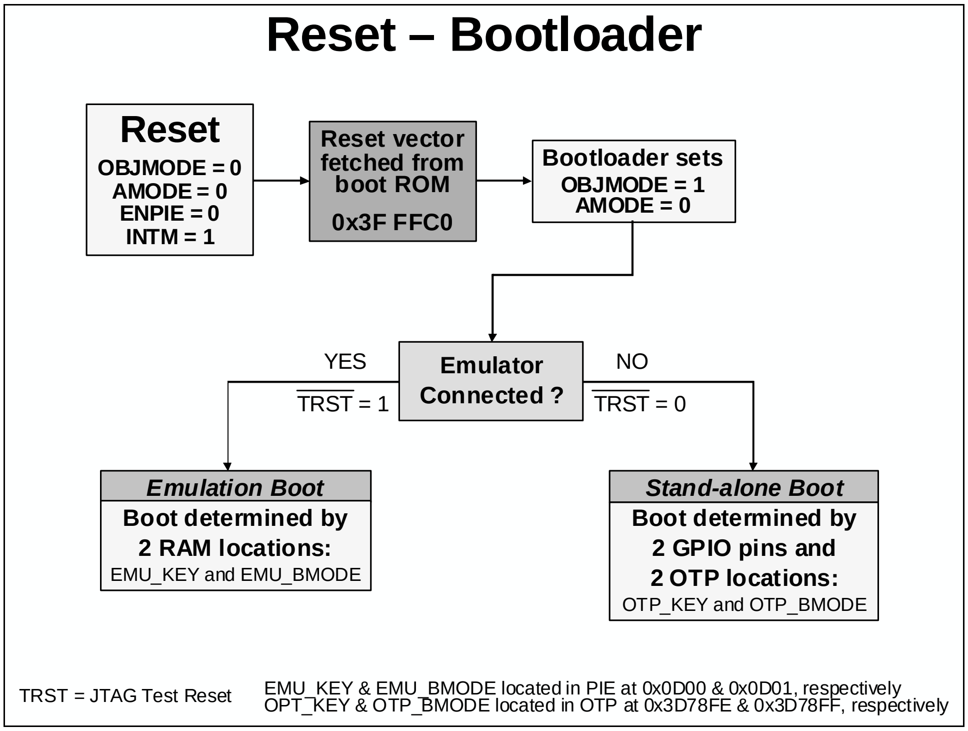
Reset Sources

Logic shown is functional representation, not actual implementation

- **POR – *Power-On Rest* 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 regulator (*VREG*) to generate the core voltage

Reset - Bootloader

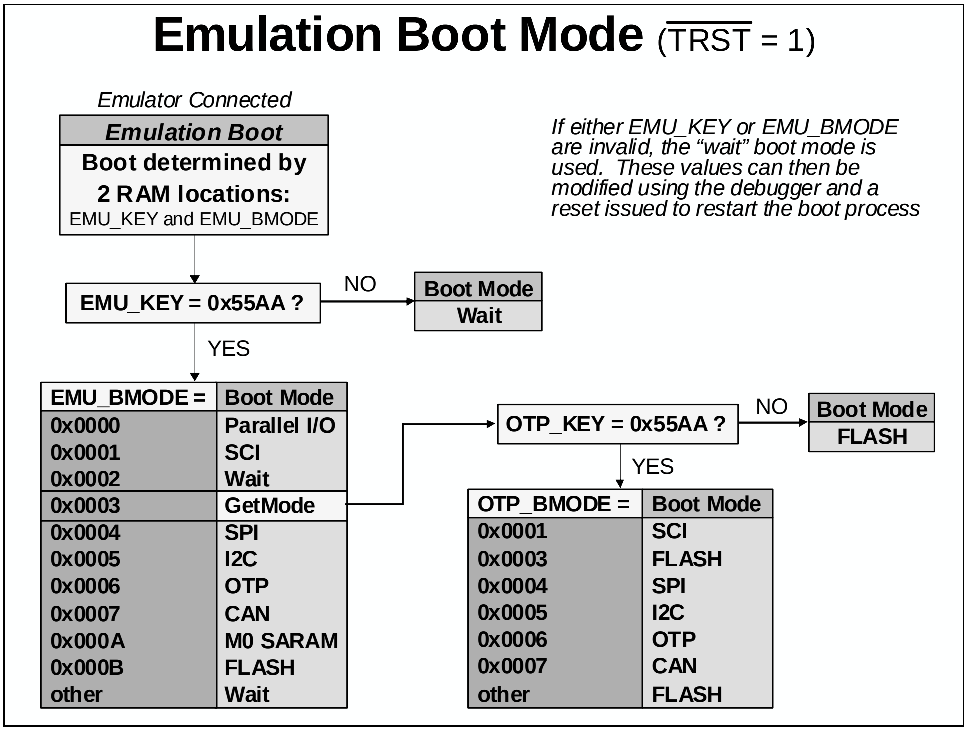


Reset – Bootloader

TRST = JTAG Test Reset

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

Emulation Boot Mode



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

Emulator Connected

Emulation Boot

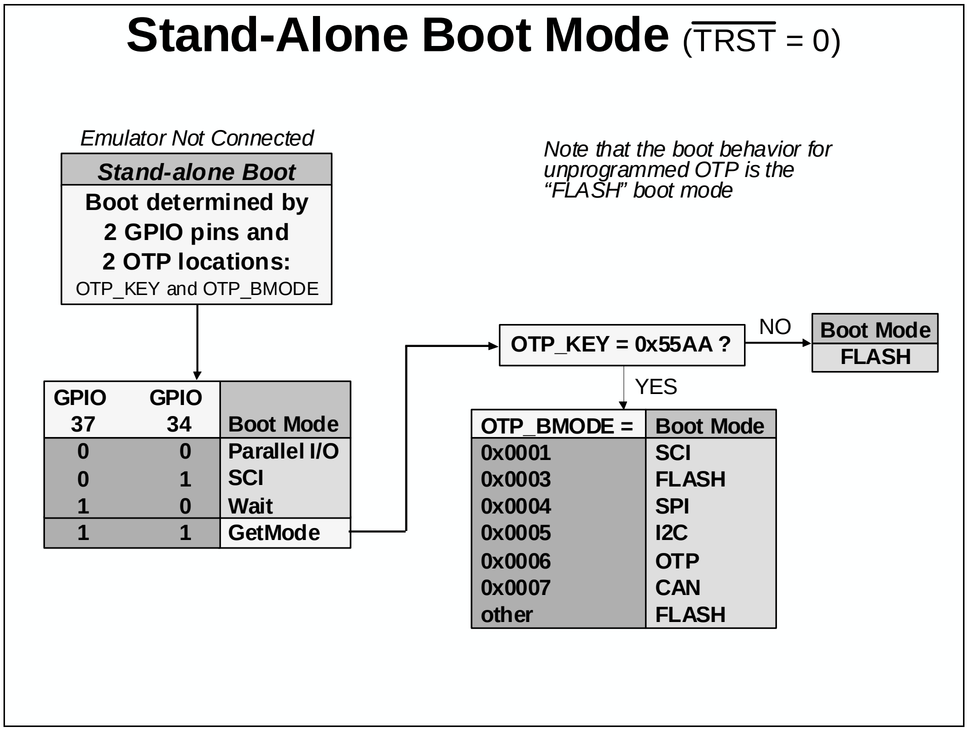
Boot determined by 2 RAM locations:
EMU_KEY and EMU_BMODE

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

EMU_BMODE =	Boot Mode
0x0000	Parallel I/O
0x0001	SCI
0x0002	Wait
0x0003	GetMode
0x0004	SPI
0x0005	I2C
0x0006	OTP
0x0007	CAN
0x000A	M0 SARAM
0x000B	FLASH
other	Wait

OTP_BMODE =	Boot Mode
0x0001	SCI
0x0003	FLASH
0x0004	SPI
0x0005	I2C
0x0006	OTP
0x0007	CAN
other	FLASH

Stand-Alone Boot Mode



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

Emulator Not Connected

Stand-alone Boot

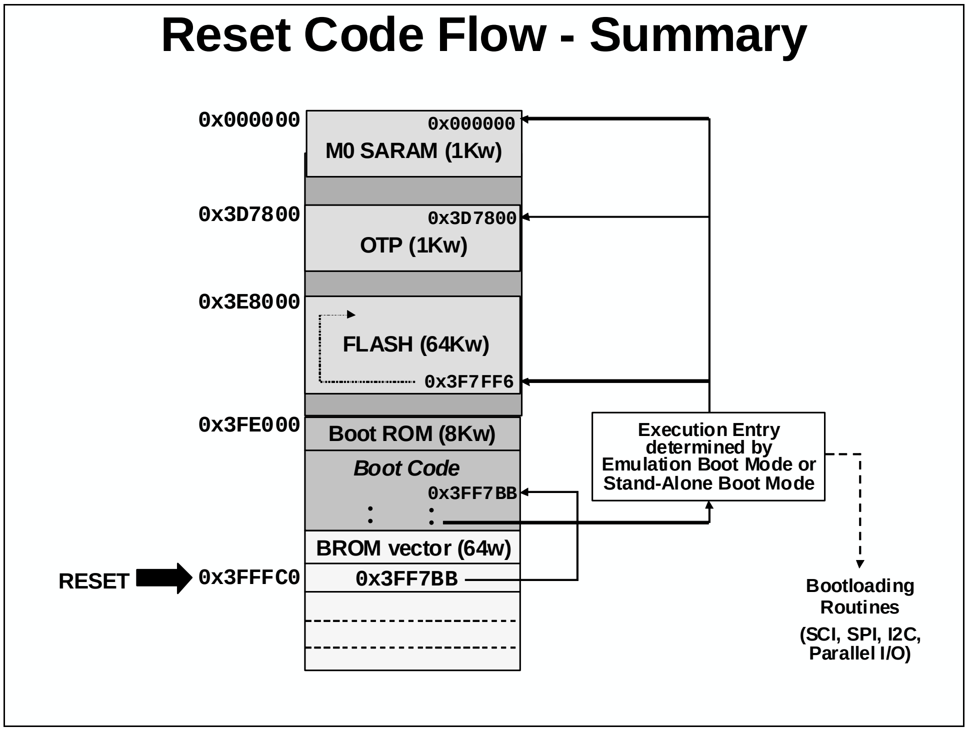
Boot determined by 2 GPIO pins and 2 OTP locations:
OTP_KEY and OTP_BMODE

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

GPIO 37	GPIO 34	Boot Mode
0	0	Parallel I/O
0	1	SCI
1	0	Wait
1	1	GetMode

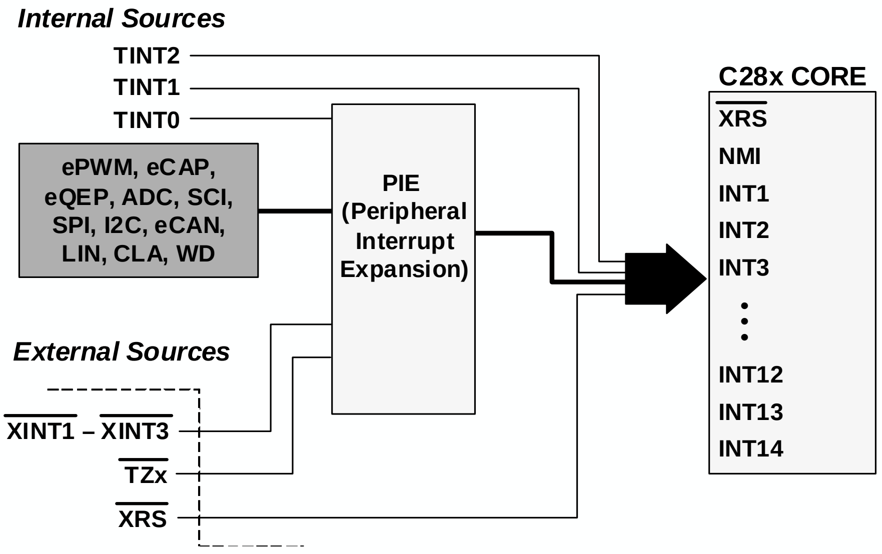
OTP_BMODE =	Boot Mode
0x0001	SCI
0x0003	FLASH
0x0004	SPI
0x0005	I2C
0x0006	OTP
0x0007	CAN
other	FLASH

Reset Code Flow – Summary



Interrupts

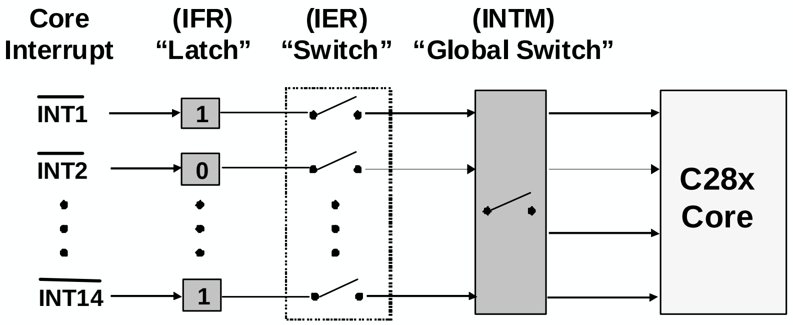
Interrupt Sources



Interrupt Processing

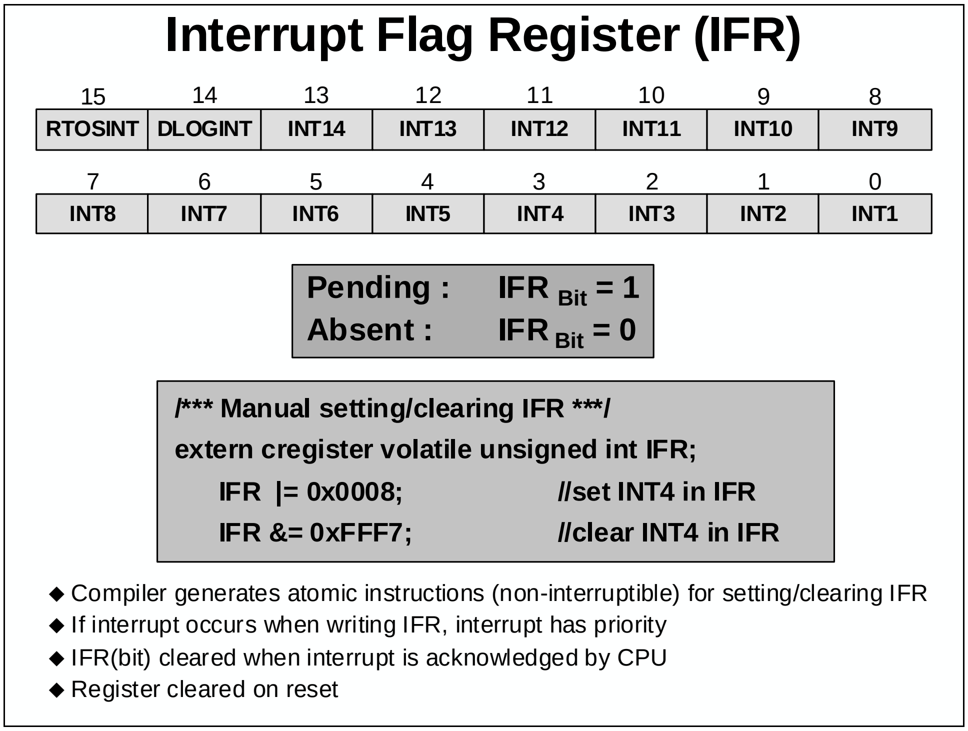
Maskable Interrupt Processing

Conceptual Core Overview



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

Interrupt Flag Register (IFR)



Interrupt Flag Register (IFR)

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

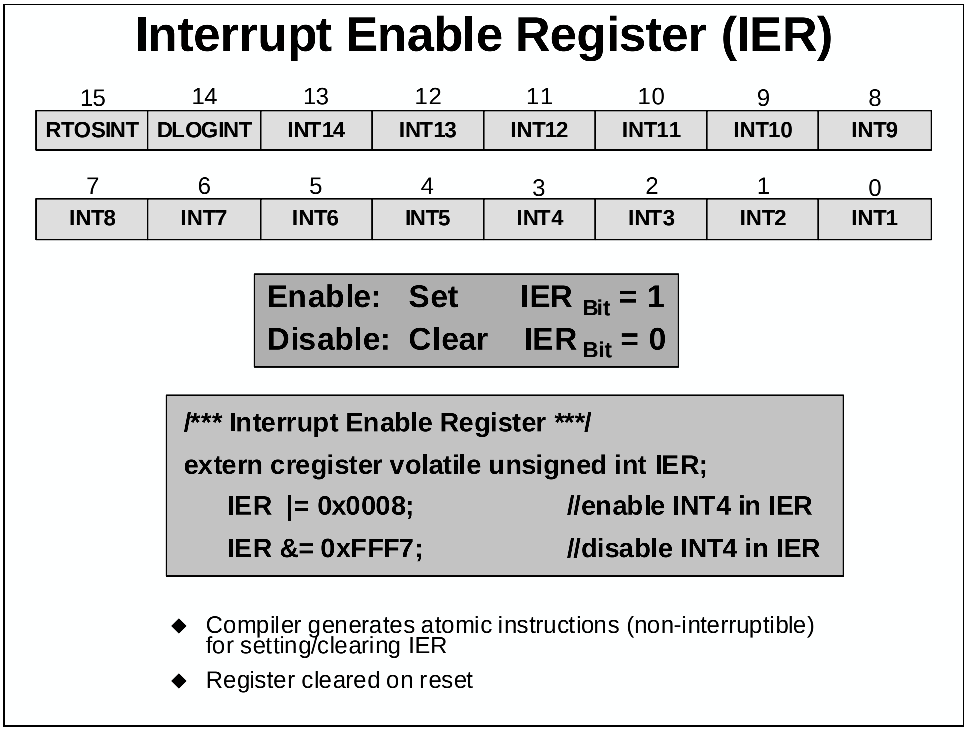
7	6	5	4	3	2	1	0
INT8	INT7	INT6	INT5	INT4	INT3	INT2	INT1

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

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

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

Interrupt Enable Register (IER)



Interrupt Enable Register (IER)

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

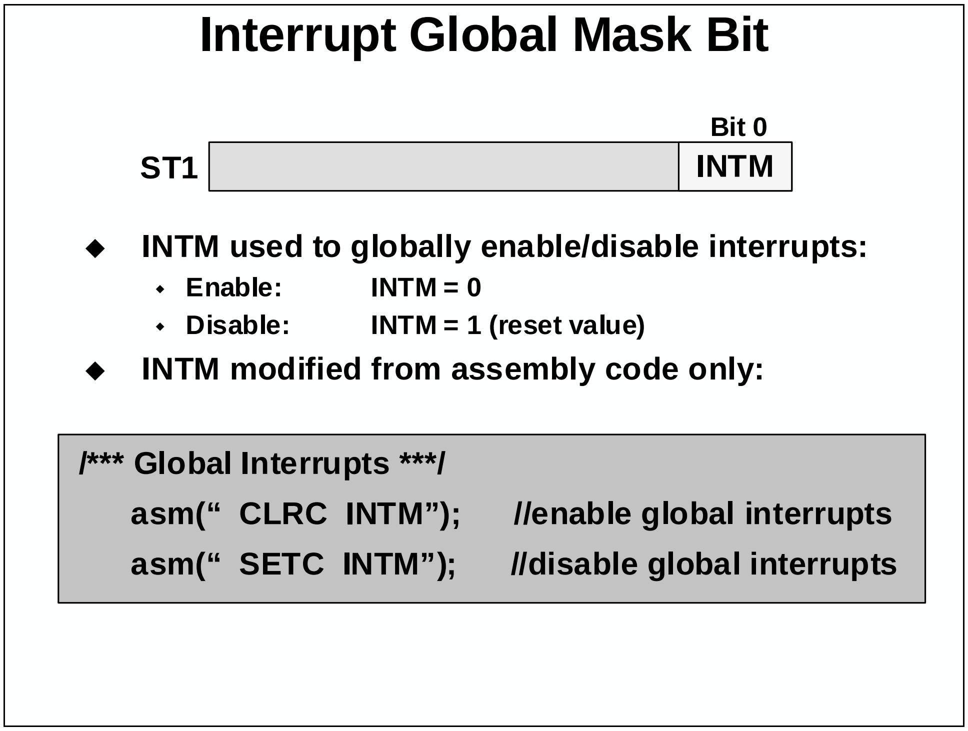
7	6	5	4	3	2	1	0
INT8	INT7	INT6	INT5	INT4	INT3	INT2	INT1

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

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

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

Interrupt Global Mask Bit (INTM)



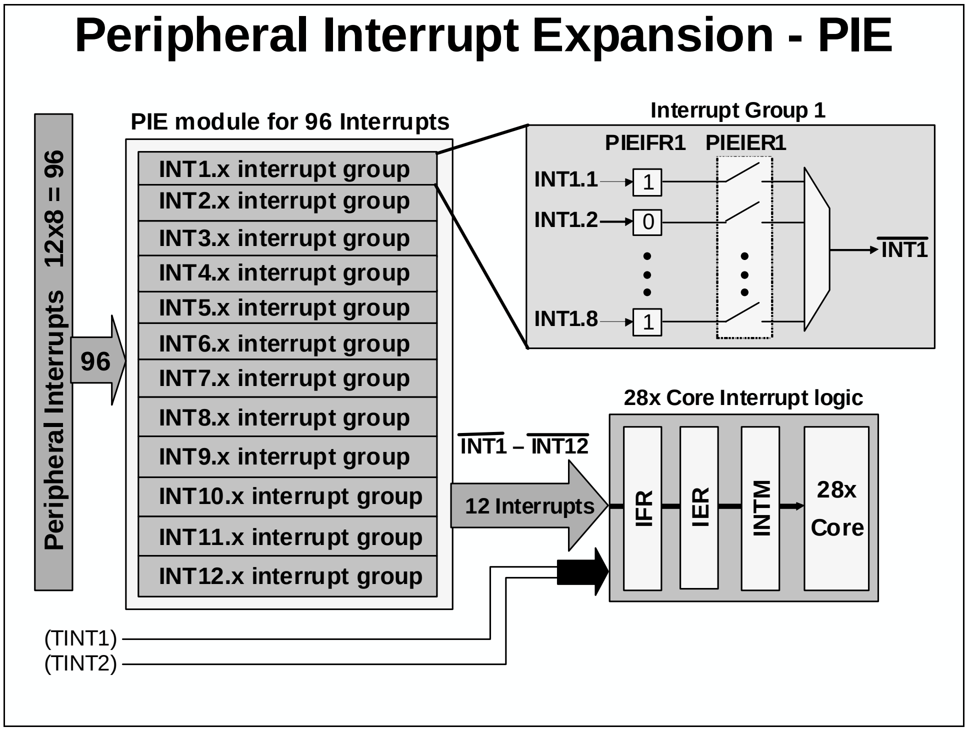
Interrupt Global Mask Bit

	Bit 0
ST1	INTM

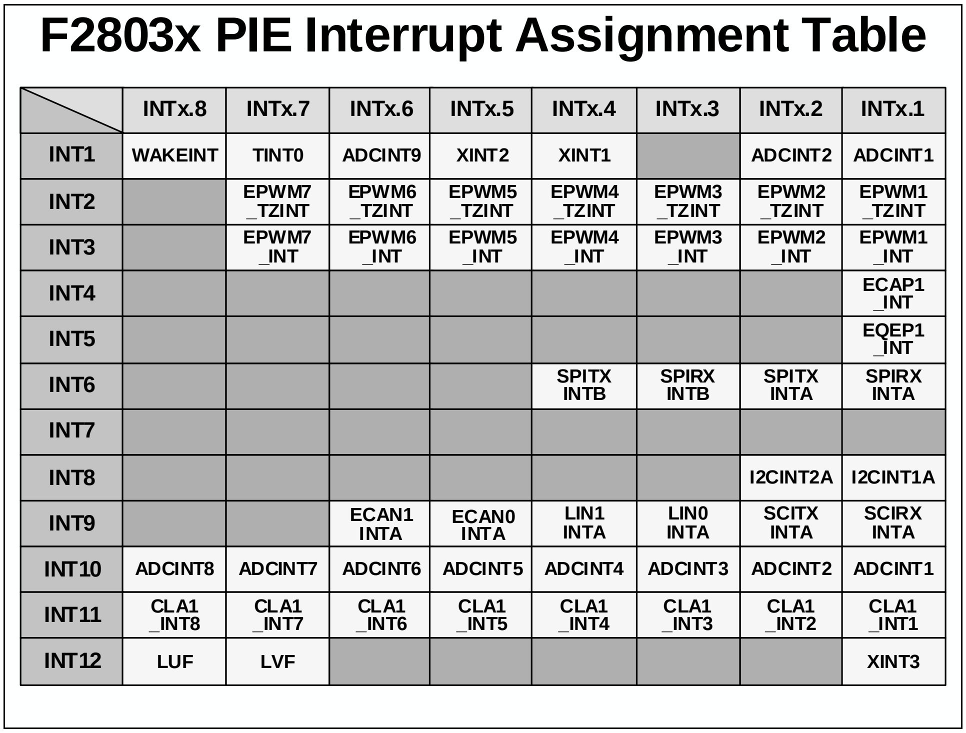
- INTM used to globally enable/disable interrupts:
  - Enable: INTM = 0
  - Disable: INTM = 1 (reset value)
- INTM modified from assembly code only:

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

Peripheral Interrupt Expansion (PIE)

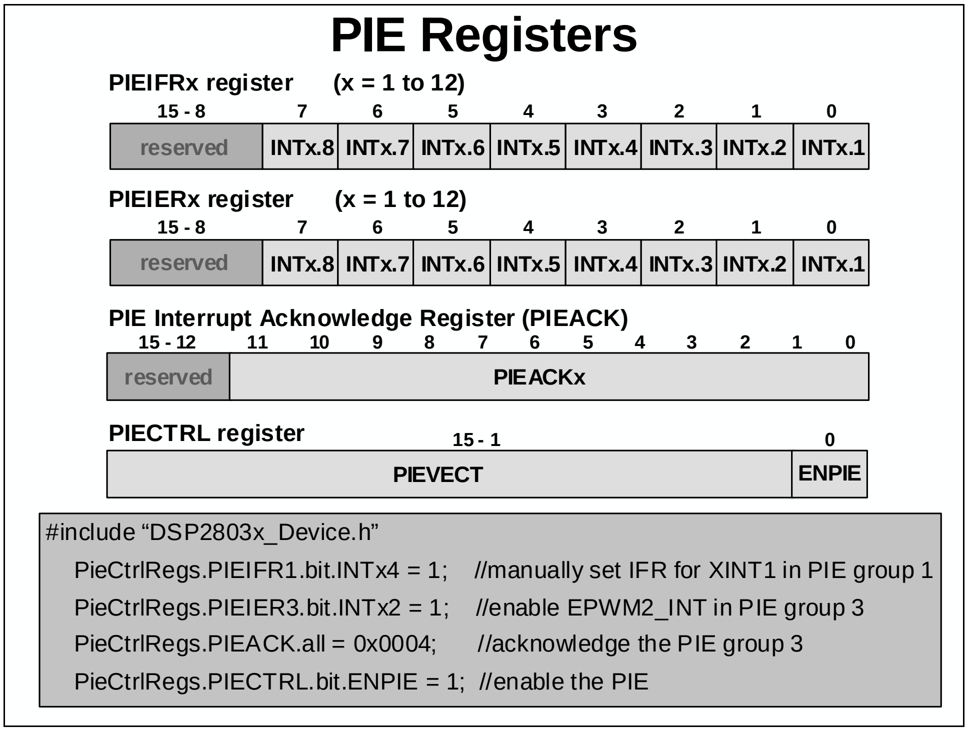


Peripheral Interrupt Expansion - PIE



F2803x 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		EPWM7 _TZINT	EPWM6 _TZINT	EPWM5 _TZINT	EPWM4 _TZINT	EPWM3 _TZINT	EPWM2 _TZINT	EPWM1 _TZINT
INT3		EPWM7 _INT	EPWM6 _INT	EPWM5 _INT	EPWM4 _INT	EPWM3 _INT	EPWM2 _INT	EPWM1 _INT
INT4								ECAP1 _INT
INT5								EQEP1 _INT
INT6					SPITX INTB	SPIRX INTB	SPITX INTA	SPIRX INTA
INT7								
INT8							I2CINT2A	I2CINT1A
INT9			ECAN1 INTA	ECAN0 INTA	LIN1 INTA	LIN0 INTA	SCITX INTA	SCIRX INTA
INT10	ADCINT8	ADCINT7	ADCINT6	ADCINT5	ADCINT4	ADCINT3	ADCINT2	ADCINT1
INT11	CLA1 _INT8	CLA1 _INT7	CLA1 _INT6	CLA1 _INT5	CLA1 _INT4	CLA1 _INT3	CLA1 _INT2	CLA1 _INT1
INT12	LUF	LVF						XINT3



PIE Registers

PIEIFRx register (x = 1 to 12)

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

PIEIERx register (x = 1 to 12)

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

PIE Interrupt Acknowledge Register (PIEACK)

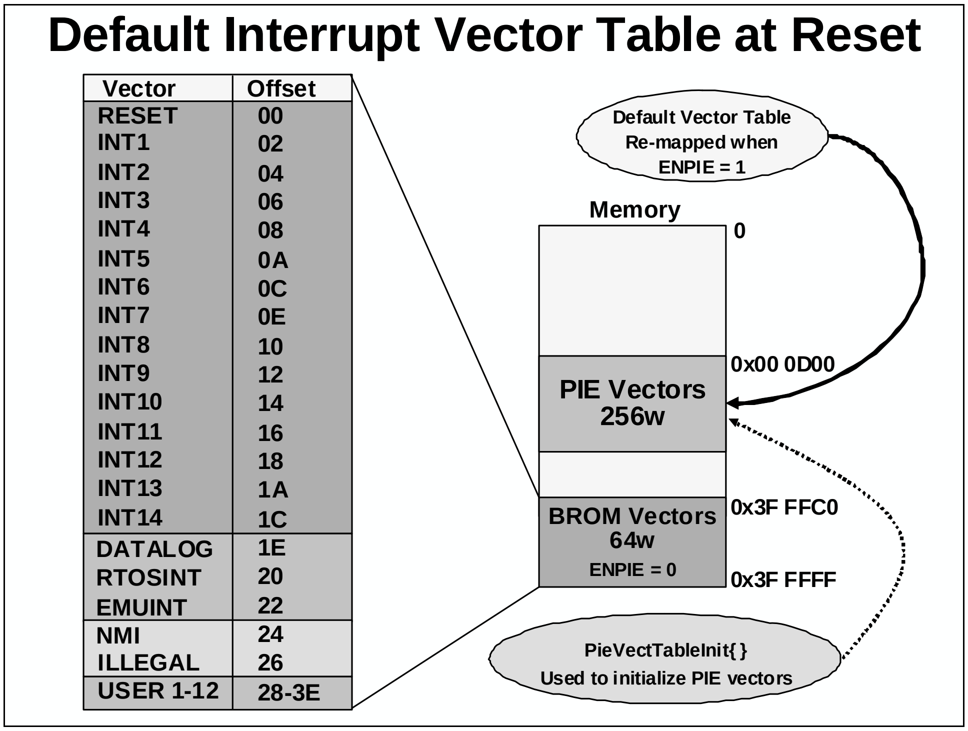
15 - 12	11	10	9	8	7	6	5	4	3	2	1	0
reserved	PIEACKx											

PIECTRL register

15 - 1	0
PIEVECT	ENPIE

```
#include “DSP2803x_Device.h”
    PieCtrlRegs.PIEIFR1.bit.INTx4 = 1;    //manually set IFR for XINT1 in PIE group 1
    PieCtrlRegs.PIEIER3.bit.INTx2 = 1;    //enable EPWM2_INT in PIE group 3
    PieCtrlRegs.PIEACK.all = 0x0004;      //acknowledge the PIE group 3
    PieCtrlRegs.PIECTRL.bit.ENPIE = 1;  //enable the PIE
```

PIE Interrupt Vector Table



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

Default Vector Table
Re-mapped when
ENPIE = 1

Memory

0

0x00 0D00

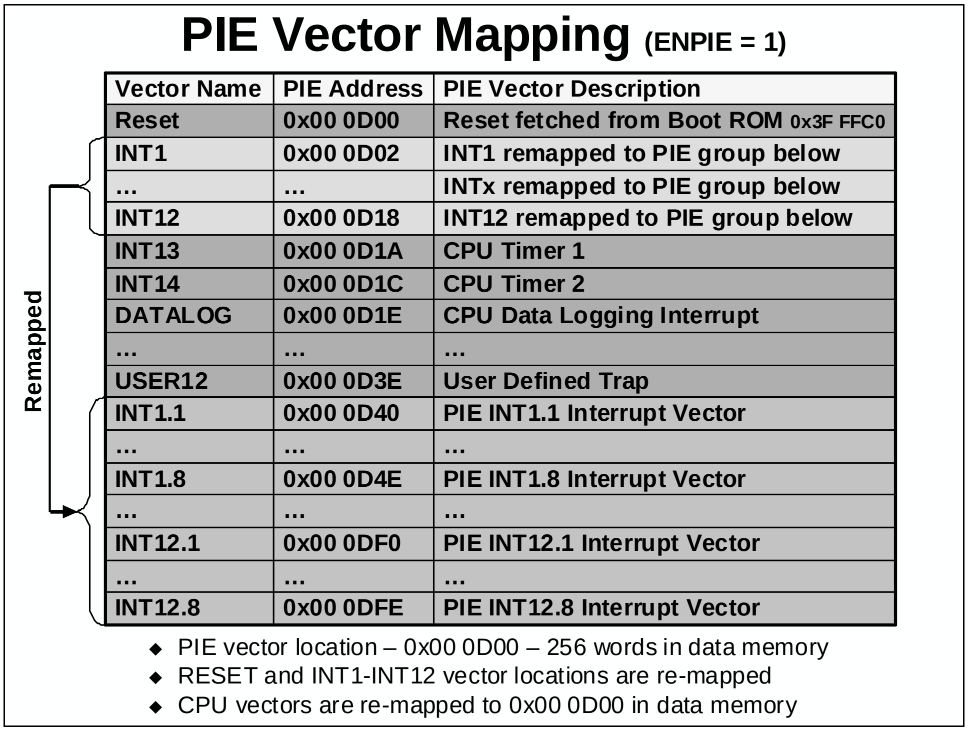
PIE Vectors
256w

BROM Vectors
64w
ENPIE = 0

0x3F FFC0

0x3F FFFF

PieVectTableInit{ }
Used to initialize PIE vectors

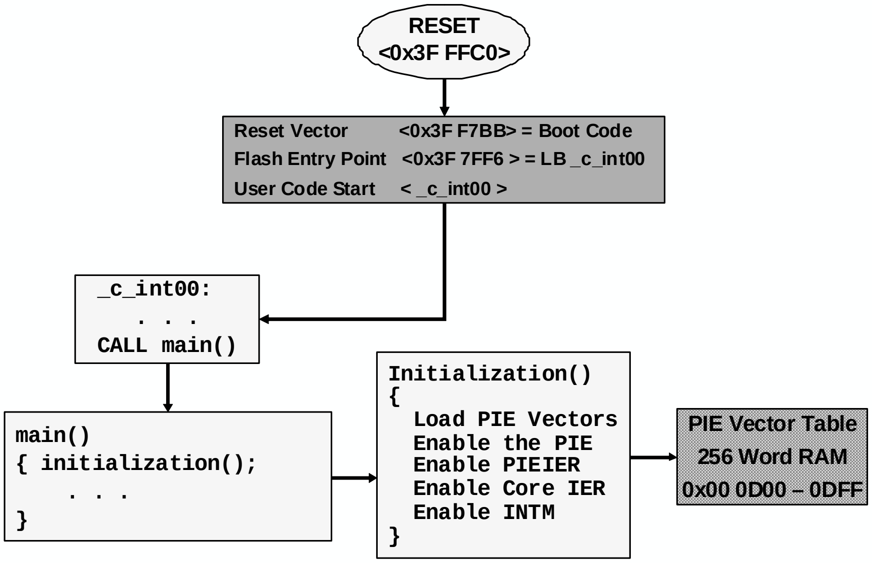


PIE Vector Mapping (ENPIE = 1)

Vector Name	PIE Address	PIE Vector Description
Reset	0x00 0D00	Reset fetched from Boot ROM 0x3F FFC0
INT1	0x00 0D02	INT1 remapped to PIE group below
…	…	INTx remapped to PIE group below
INT12	0x00 0D18	INT12 remapped to PIE group below
INT13	0x00 0D1A	CPU Timer 1
INT14	0x00 0D1C	CPU Timer 2
DATALOG	0x00 0D1E	CPU Data Logging Interrupt
…	…	…
USER12	0x00 0D3E	User Defined Trap
INT1.1	0x00 0D40	PIE INT1.1 Interrupt Vector
…	…	…
INT1.8	0x00 0D4E	PIE INT1.8 Interrupt Vector
…	…	…
INT12.1	0x00 0DF0	PIE INT12.1 Interrupt Vector
…	…	…
INT12.8	0x00 0DFE	PIE INT12.8 Interrupt Vector

- PIE vector location – 0x00 0D00 – 256 words in data memory
- RESET and INT1-INT12 vector locations are re-mapped
- CPU vectors are re-mapped to 0x00 0D00 in data memory

Device Vector Mapping - Summary


RESET
<0x3F FFC0>
Reset Vector <0x3F F7BB> = Boot Code
Flash Entry Point <0x3F 7FF6 > = LB _c_int00
User Code Start < _c_int00 >
PIE Vector Table
256 Word RAM
0x00 0D00 – 0DFF

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

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

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

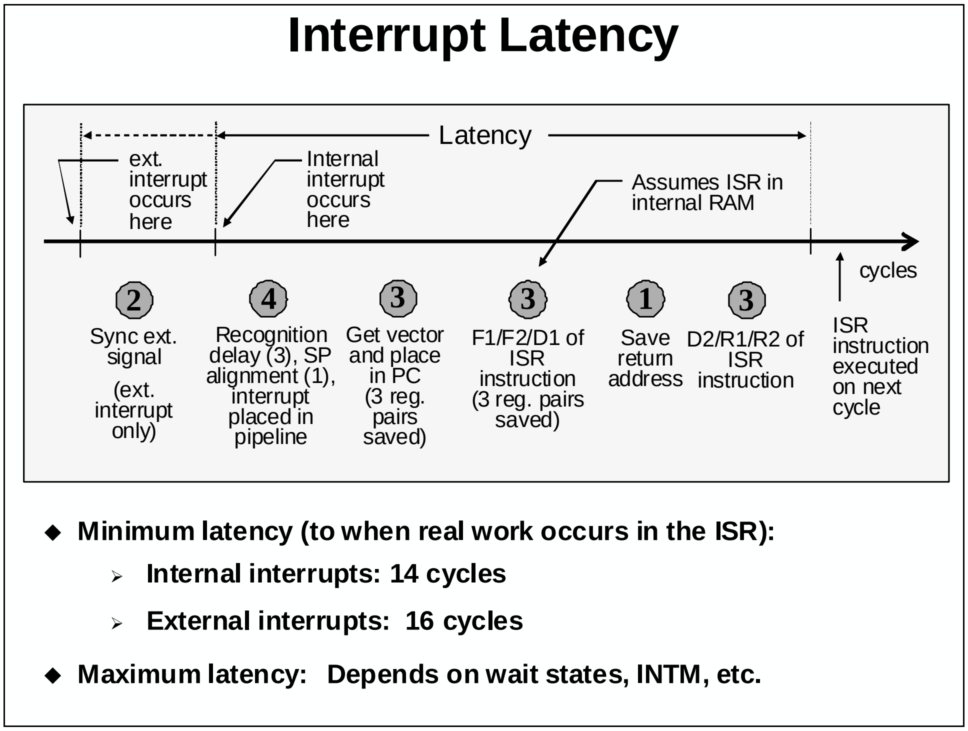
Interrupt Response and Latency

Interrupt Response - Hardware Sequence

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

Note: some actions occur simultaneously, none are interruptible

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



Interrupt Latency

Latency

ext. interrupt occurs here

Internal interrupt occurs here

Assumes ISR in internal RAM

cycles

2 Sync ext. signal (ext. interrupt only)

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

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

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

1 Save return address

3 D2/R1/R2 of ISR instruction

ISR instruction executed on next cycle

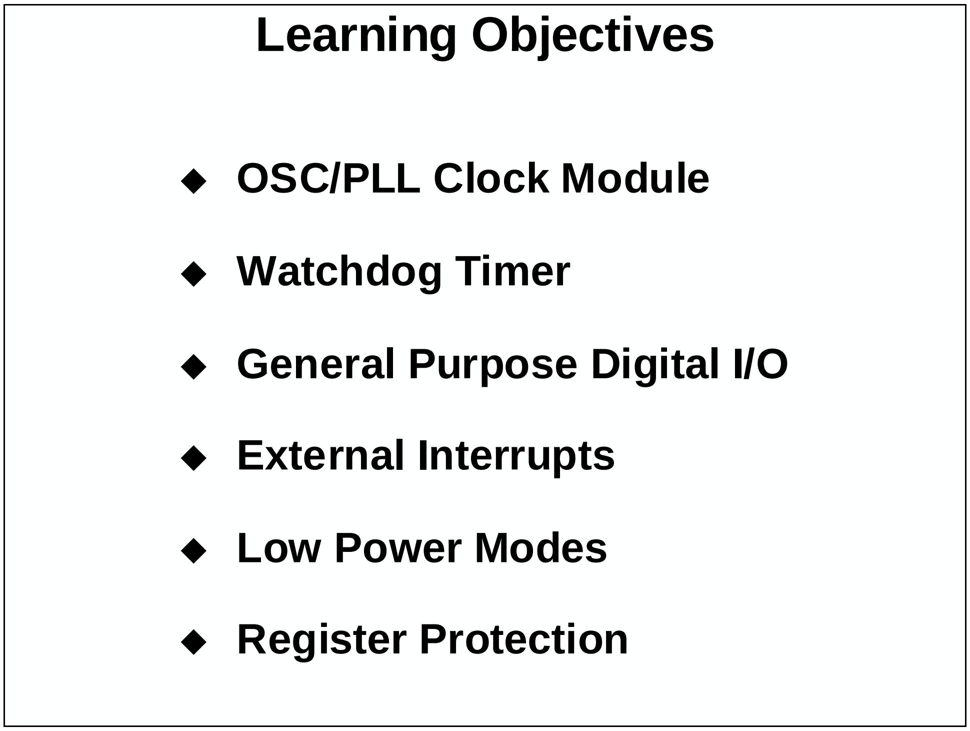
- **Minimum latency (to when real work occurs in the ISR):**
  - **Internal interrupts: 14 cycles**
  - **External interrupts: 16 cycles**
- **Maximum latency: Depends on wait states, INTM, etc.**

System Initialization

Introduction

This module discusses the operation of the OSC/PLL-based clock module and watchdog timer. Also, the general-purpose digital I/O ports, external interrups, various low power modes and the EALLOW protected registers will be covered.

Learning Objectives



Learning Objectives

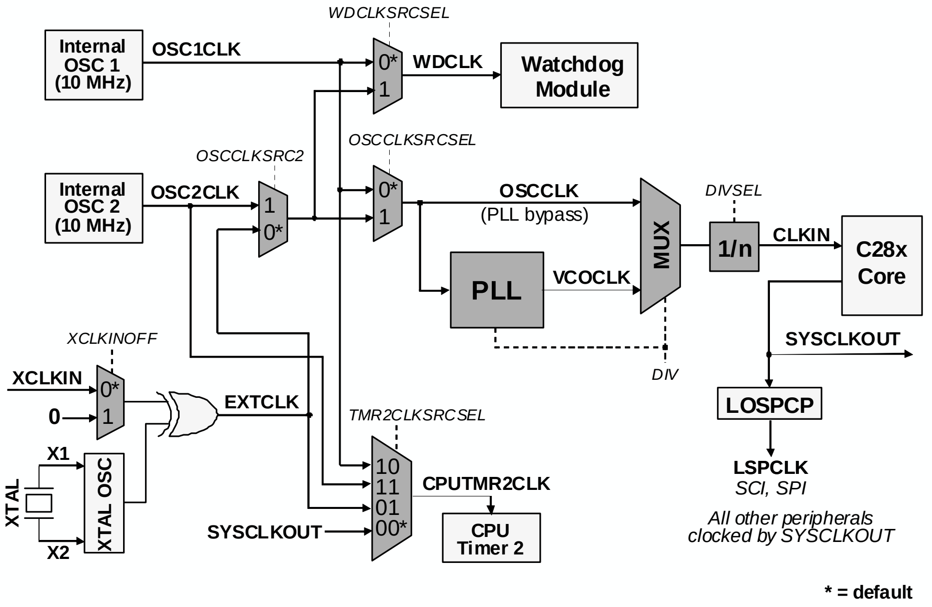
- OSC/PLL Clock Module
- Watchdog Timer
- General Purpose Digital I/O
- External Interrupts
- Low Power Modes
- Register Protection

Module Topics

Oscillator/PLL Clock Module

F2803x Oscillator / PLL Clock Module

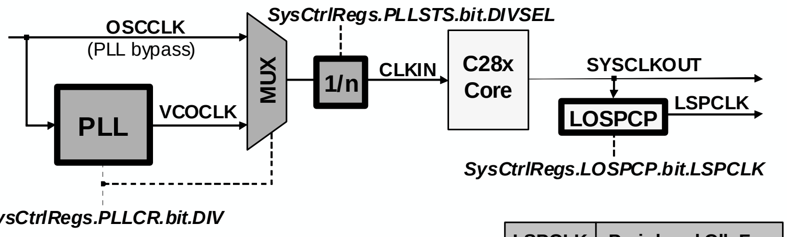
(lab file: SysCtrl.c)



The on-chip oscillator and phase-locked loop (PLL) block provide all the necessary clocking signals for the F2803x devices. The two internal oscillators (INTOSC1 and INTOSC2) need no external components.

F2803x PLL and LOSPCP

(lab file: SysCtrl.c)


OSCCLK
(PLL bypass)
SysCtrlRegs.PLLSTS.bit.DIVSEL
MUX
1/n
CLKIN
C28x Core
SYSCLKOUT
PLL
VCOCLK
LOSPCP
LSPCLK
SysCtrlRegs.LOSPCP.bit.LSPCLK
SysCtrlRegs.PLLCR.bit.DIV

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

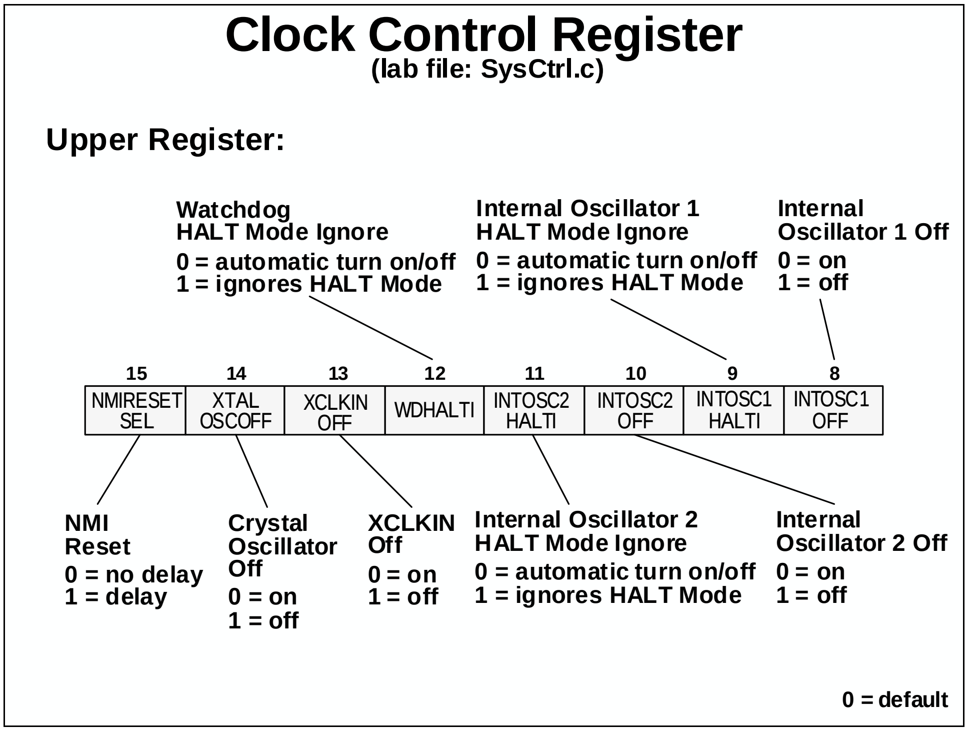
LSPCLK	Peripheral Clk Freq
0 0 0	SYSCLKOUT / 1
0 0 1	SYSCLKOUT / 2
0 1 0	SYSCLKOUT / 4 *
0 1 1	SYSCLKOUT / 6
1 0 0	SYSCLKOUT / 8
1 0 1	SYSCLKOUT / 10
1 1 0	SYSCLKOUT / 12
1 1 1	SYSCLKOUT / 14

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).

The PLL has a 4-bit ratio control to select different CPU clock rates. In addition to the on-chip oscillators, two external modes of operation are supported – crystal operation, and external clock source operation. Crystal operation allows the use of an external crystal/resonator to provide the time base to the device. External clock source operation allows the internal (crystal) oscillator to be bypassed, and the device clocks are generated from an external clock source input on the XCLKIN pin. The C28x core provides a SYSCLKOUT clock signal. This signal is prescaled to provide a clock source for some of the on-chip communication peripherals through the low-speed peripheral clock prescaler. Other peripherals are clocked by SYSCLKOUT and use their own clock prescalers for operation.



Clock Control Register

(lab file: SysCtrl.c)

Upper Register:

15	14	13	12	11	10	9	8
NMIRESET SEL	XTAL OSCOFF	XCLKIN OFF	WDHALTI	INTOSC2 HALTI	INTOSC2 OFF	INTOSC1 HALTI	INTOSC1 OFF

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

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

Internal Oscillator 1 Off
0 = on
1 = off

NMI Reset
0 = no delay
1 = delay

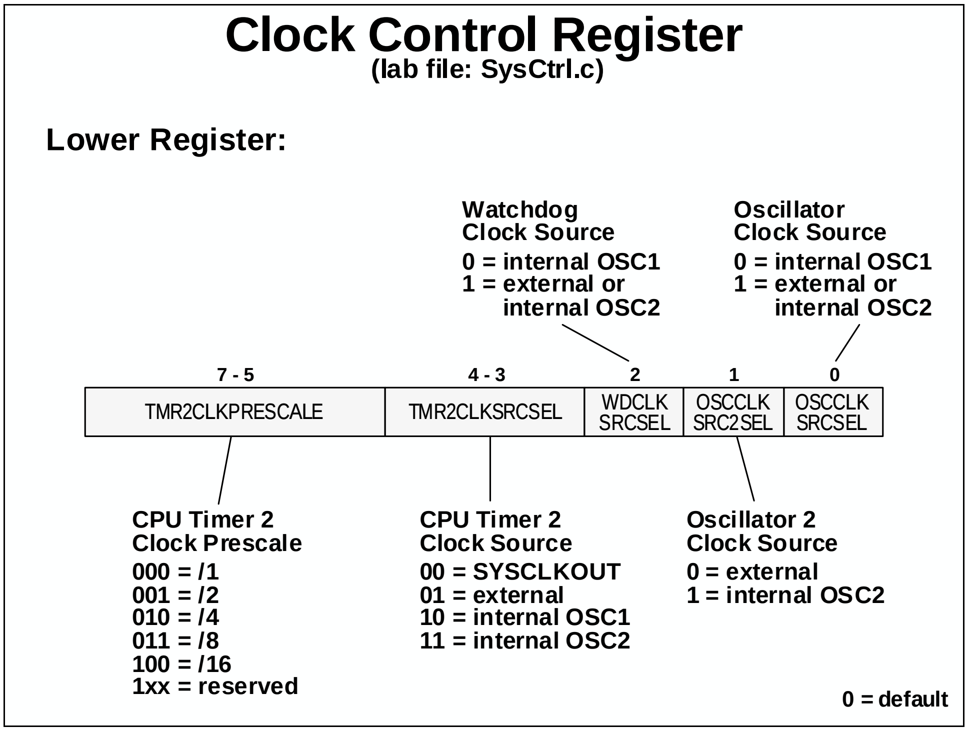
Crystal Oscillator Off
0 = on
1 = off

XCLKIN Off
0 = on
1 = off

Internal Oscillator 2 HALT Mode Ignore
0 = automatic turn on/off
1 = ignores HALT Mode

Internal Oscillator 2 Off
0 = on
1 = off

0 = default



Clock Control Register

(lab file: SysCtrl.c)

Lower Register:

Watchdog Clock Source
0 = internal OSC1
1 = external or internal OSC2

Oscillator Clock Source
0 = internal OSC1
1 = external or internal OSC2

7 - 5	4 - 3	2	1	0
TMR2CLKPRESCALE	TMR2CLKSRCSEL	WDCLK SRCSEL	OSCCLK SRC2SEL	OSCCLK SRCSEL

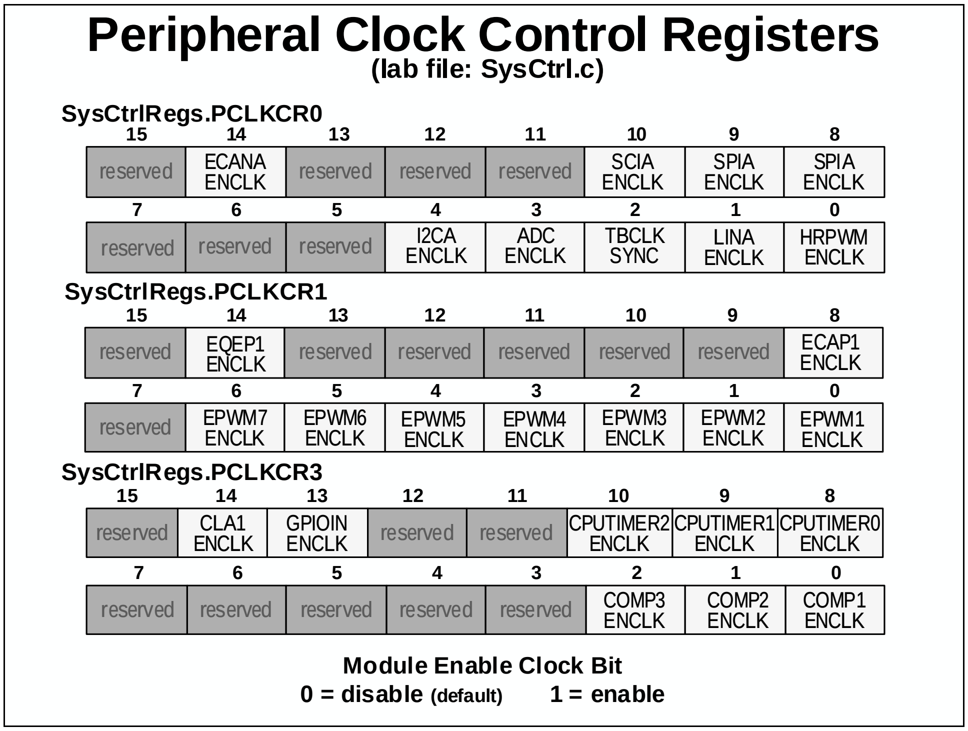
CPU Timer 2 Clock Prescale
000 = /1
001 = /2
010 = /4
011 = /8
100 = /16
1xx = reserved

CPU Timer 2 Clock Source
00 = SYSCLKOUT
01 = external
10 = internal OSC1
11 = internal OSC2

Oscillator 2 Clock Source
0 = external
1 = internal OSC2

0 = default

The peripheral clock control register allows individual peripheral clock signals to be enabled or disabled. If a peripheral is not being used, its clock signal could be disabled, thus reducing power consumption.



Peripheral Clock Control Registers

(lab file: SysCtrl.c)

SysCtrlRegs.PCLKCR0

15	14	13	12	11	10	9	8
reserved	ECANA ENCLK	reserved	reserved	reserved	SCIA ENCLK	SPIA ENCLK	SPIA ENCLK
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
reserved	reserved	reserved	I2CA ENCLK	ADC ENCLK	TBCLK SYNC	LINA ENCLK	HRPWM ENCLK

SysCtrlRegs.PCLKCR1

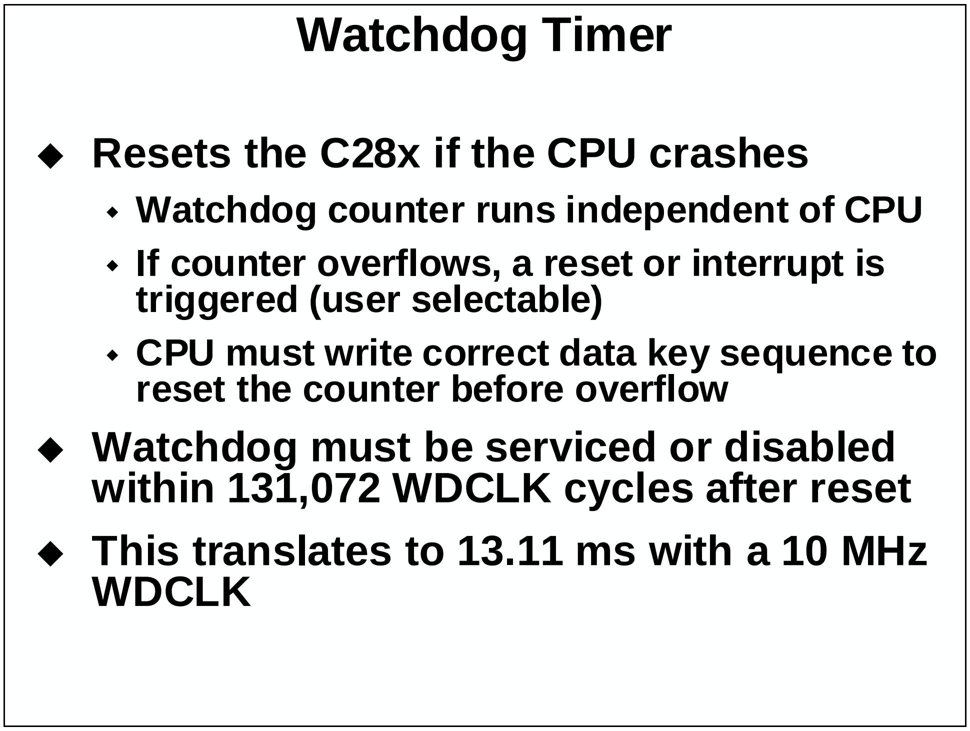
15	14	13	12	11	10	9	8
reserved	EQEP1 ENCLK	reserved	reserved	reserved	reserved	reserved	ECAP1 ENCLK
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
reserved	EPWM7 ENCLK	EPWM6 ENCLK	EPWM5 ENCLK	EPWM4 ENCLK	EPWM3 ENCLK	EPWM2 ENCLK	EPWM1 ENCLK

SysCtrlRegs.PCLKCR3

15	14	13	12	11	10	9	8
reserved	CLA1 ENCLK	GPIOIN ENCLK	reserved	reserved	CPUTIMER2 ENCLK	CPUTIMER1 ENCLK	CPUTIMER0 ENCLK
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
reserved	reserved	reserved	reserved	reserved	COMP3 ENCLK	COMP2 ENCLK	COMP1 ENCLK

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

Watchdog Timer

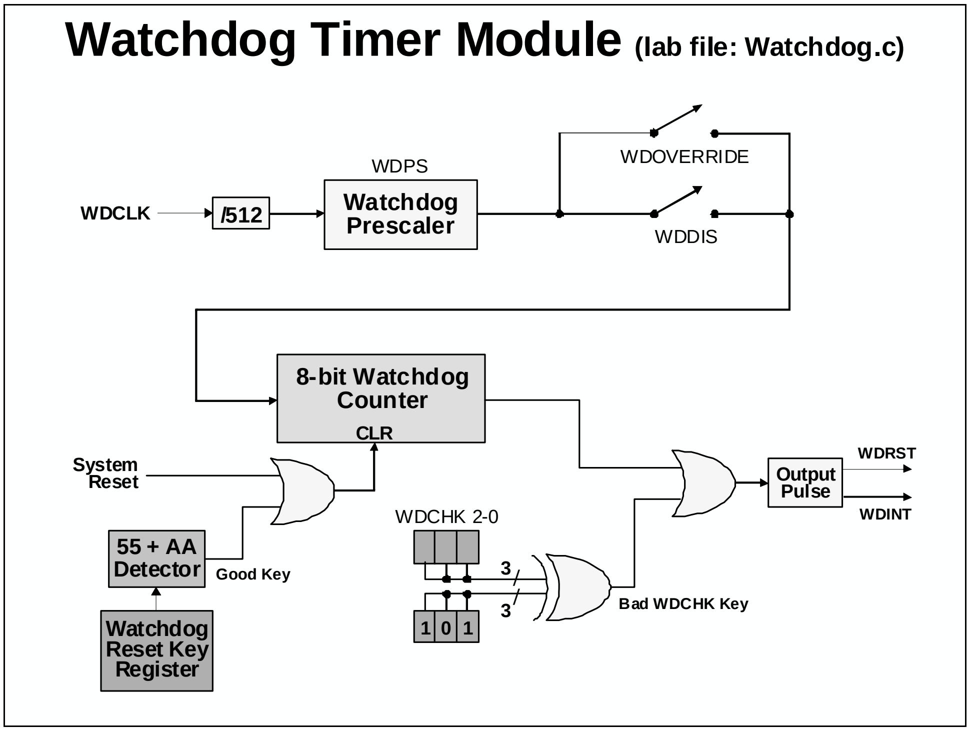


Watchdog Timer

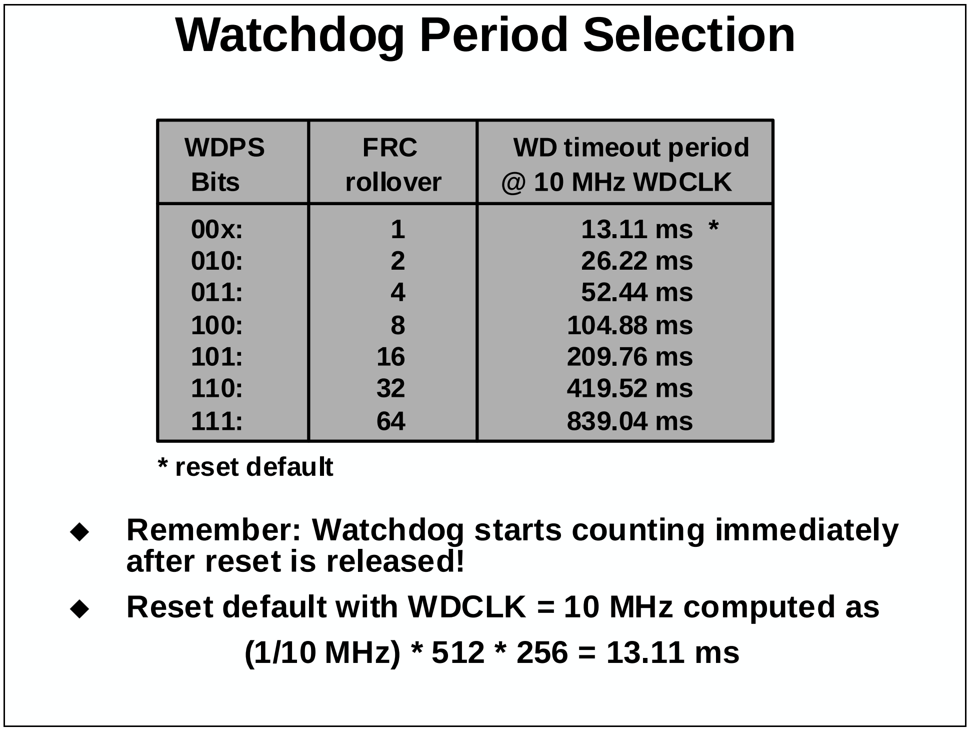
- Resets the C28x if the CPU crashes
  - Watchdog counter runs independent of CPU
  - If counter overflows, a reset or interrupt is triggered (user selectable)
  - CPU must write correct data key sequence to reset the counter before overflow
- Watchdog must be serviced or disabled within 131,072 WDCLK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

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

The watchdog timer is running immediately after system power-up/reset, and must be dealt with by software soon after. Specifically, you have 13.11 ms (for a 60 MHz device) after any reset before a watchdog initiated reset will occur. This translates into 131,072 WDCLK cycles, which is a seemingly tremendous amount! Indeed, this is plenty of time to get the watchdog configured as desired and serviced. A failure of your software to properly handle the watchdog after reset could cause an endless cycle of watchdog initiated resets to occur.



Watchdog Timer Module (lab file: Watchdog.c)



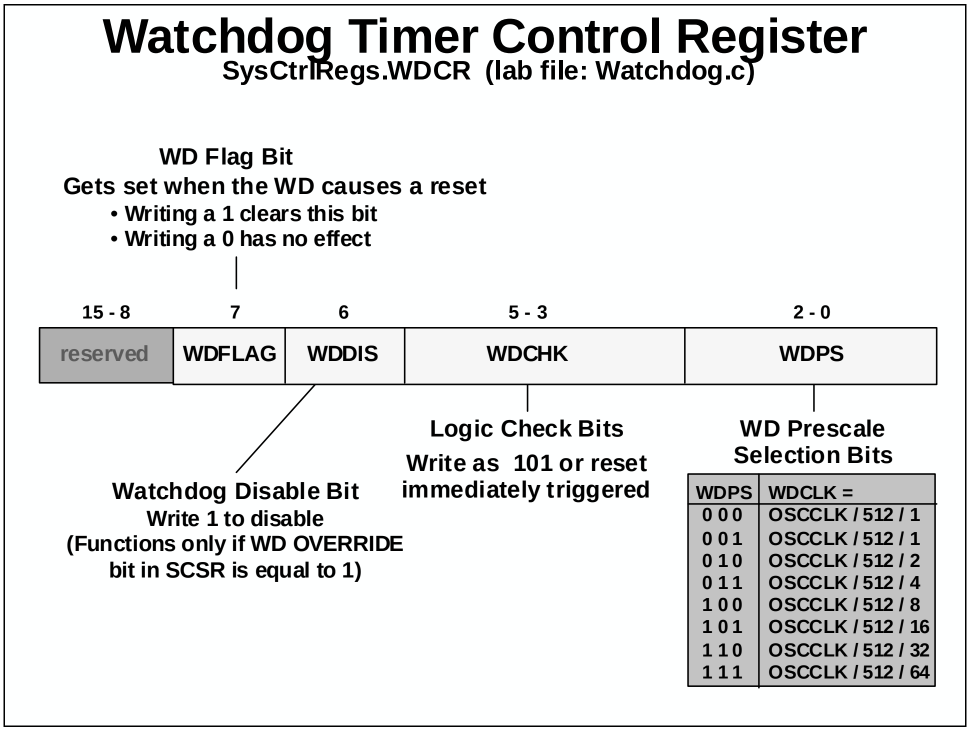
Watchdog Period Selection

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

* reset default

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

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



Watchdog Timer Control Register

SysCtrlRegs.WDCR (lab file: Watchdog.c)

WD Flag Bit
Gets set when the WD causes a reset
- **Writing a 1 clears this bit**
- **Writing a 0 has no effect**

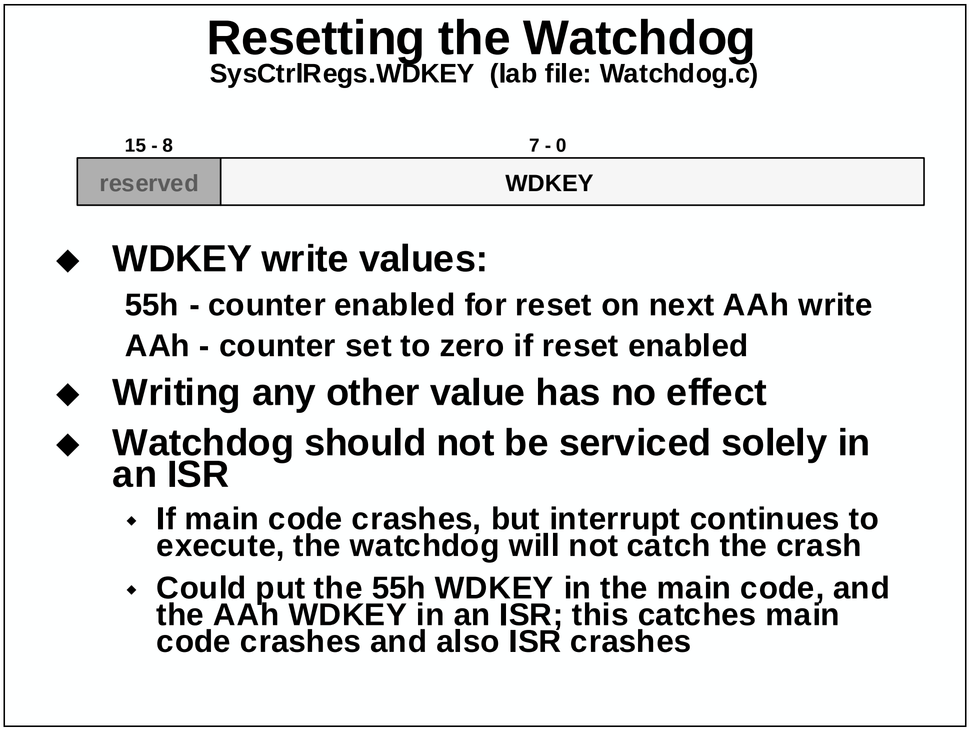
15 - 8	7	6	5 - 3	2 - 0
reserved	WDFLAG	WDDIS	WDCHK	WDPS

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

Logic Check Bits
Write as 101 or reset immediately triggered

WD Prescale Selection Bits

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



Resetting the Watchdog

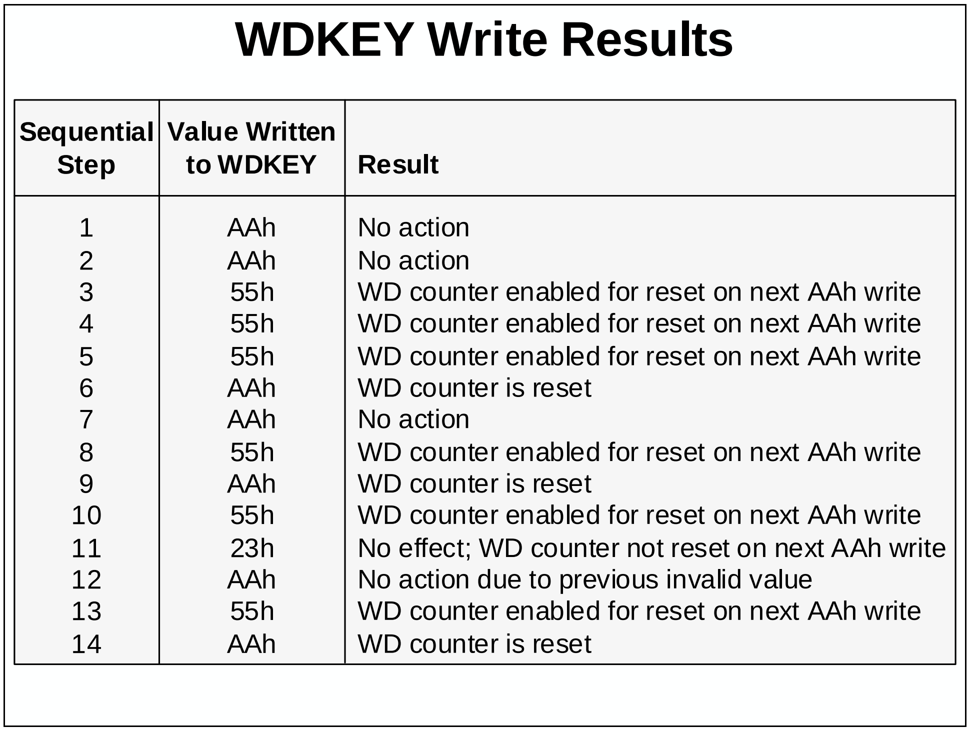
SysCtrlRegs.WDKEY (lab file: Watchdog.c)

15 - 8	7 - 0
reserved	WDKEY

- **WDKEY write values:**

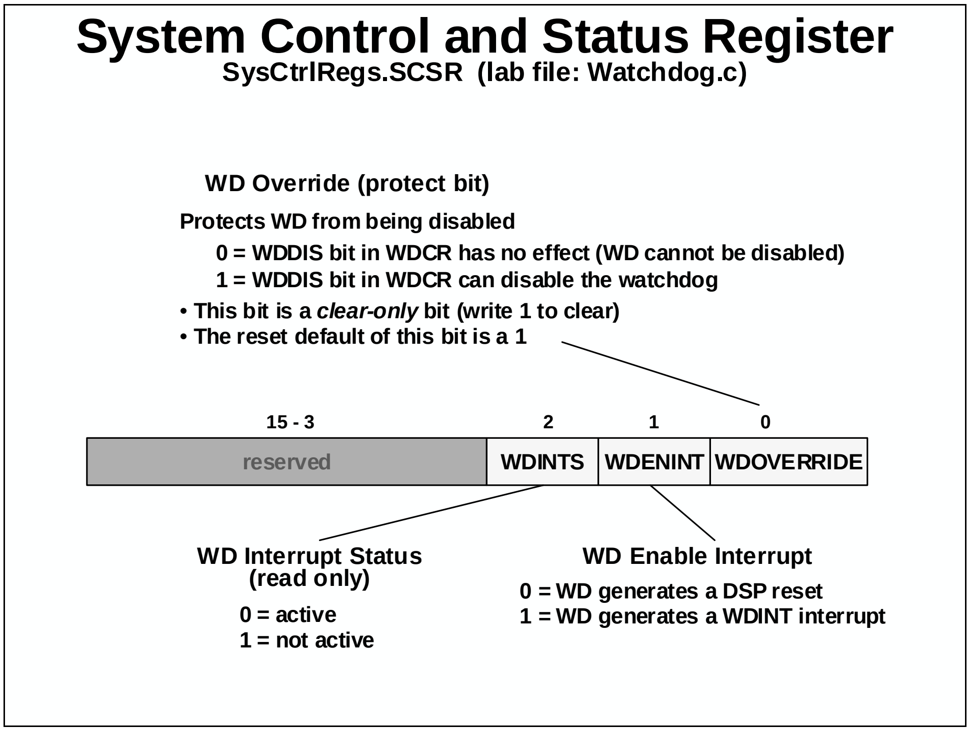
  55h - counter enabled for reset on next AAh write

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



WDKEY Write Results

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



System Control and Status Register

SysCtrlRegs.SCSR (lab file: Watchdog.c)

WD Override (protect bit)

Protects WD from being disabled

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

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

15 - 3	2	1	0
reserved	WDINTS	WDENINT	WDOVERRIDE

WD Interrupt Status (read only)

0 = active
1 = not active

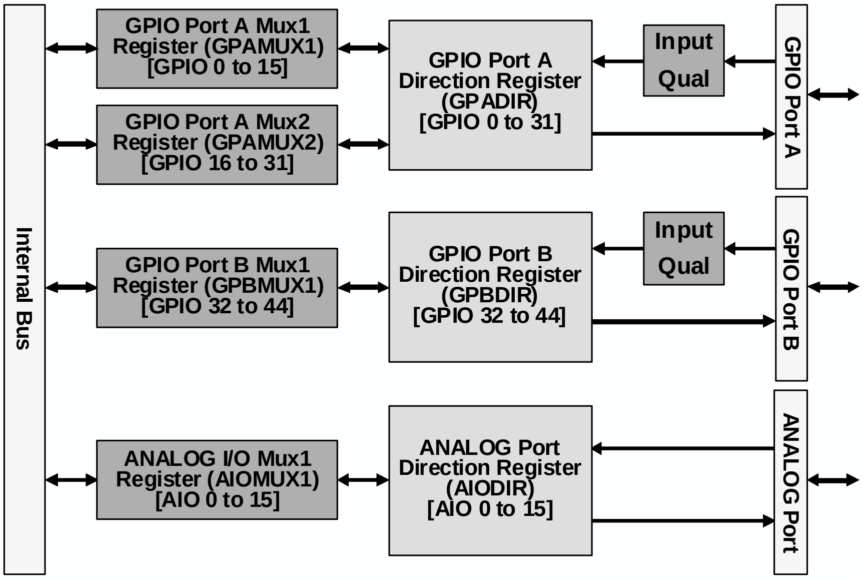
WD Enable Interrupt

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

General-Purpose Digital I/O

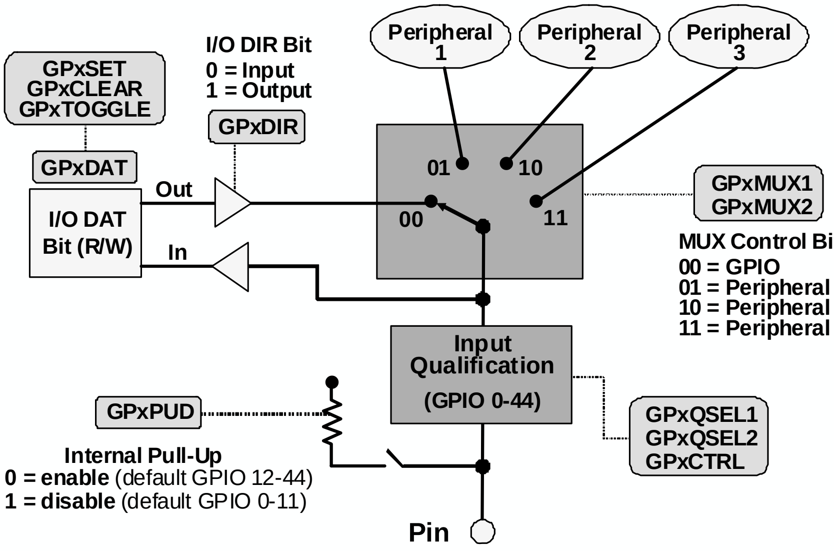
F2803x GPIO Grouping Overview

(lab file: Gpio.c)

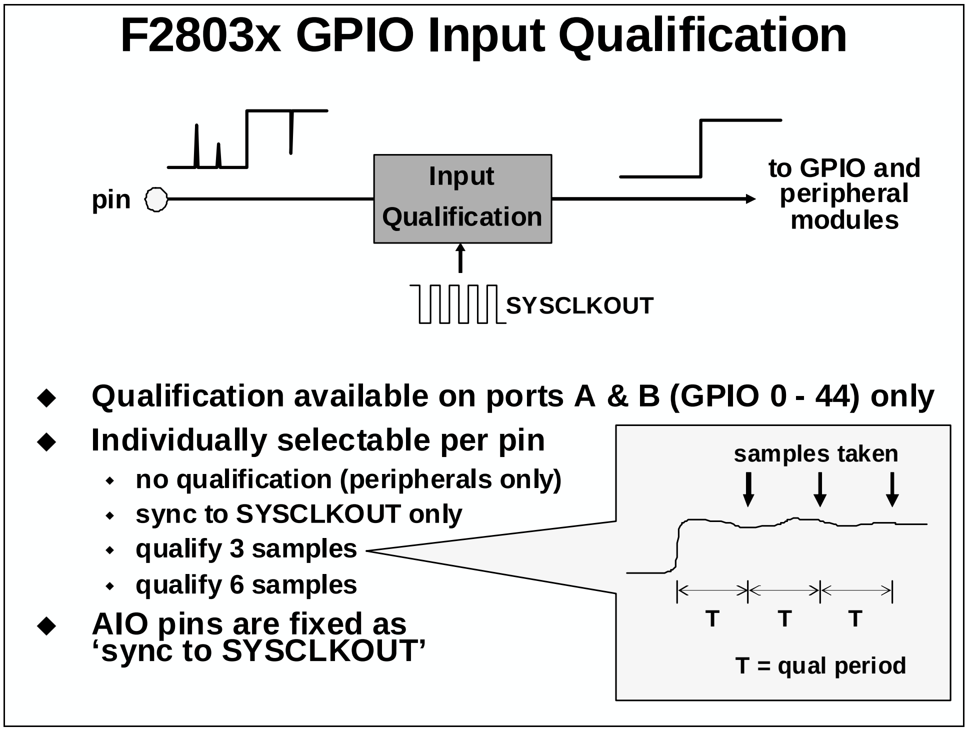

Internal Bus
GPIO Port A Mux1 Register (GPAMUX1) [GPIO 0 to 15]
GPIO Port A Mux2 Register (GPAMUX2) [GPIO 16 to 31]
GPIO Port A Direction Register (GPADIR) [GPIO 0 to 31]
Input Qual
GPIO Port A
GPIO Port B Mux1 Register (GPBMUX1) [GPIO 32 to 44]
GPIO Port B Direction Register (GPBDIR) [GPIO 32 to 44]
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

F2803x GPIO Pin Block Diagram

(lab file: Gpio.c)



** See device datasheet for pin function selection matrices*



F2803x GPIO Input Qualification

pin

Input Qualification

to GPIO and peripheral modules

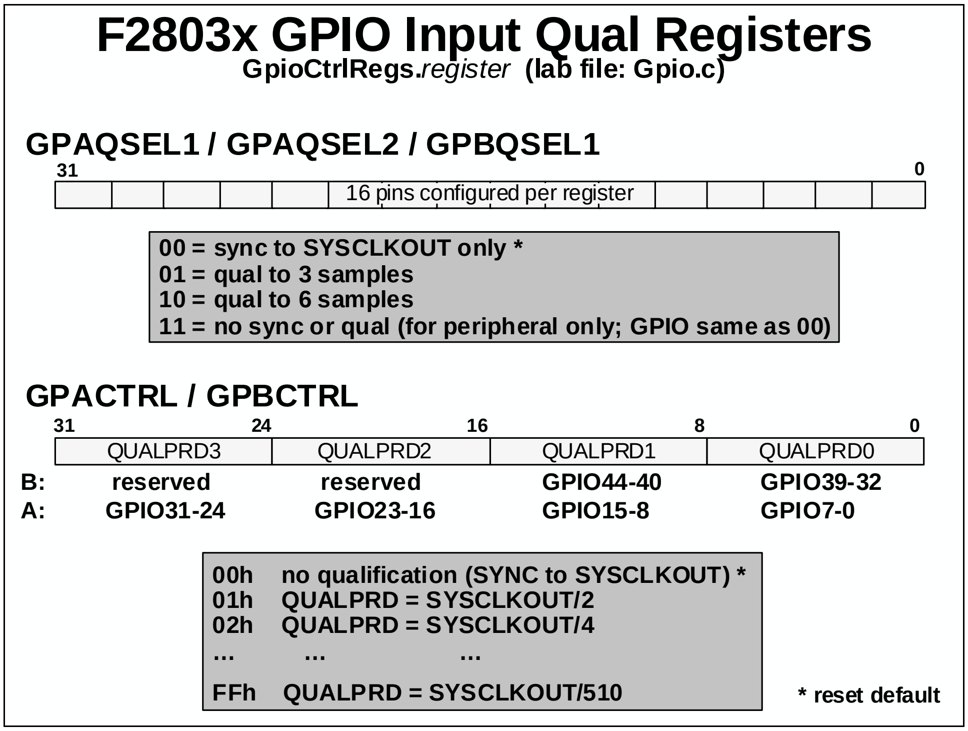
SYSCLKOUT

- Qualification available on ports A & B (GPIO 0 - 44) 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’

samples taken

T T T

T = qual period



F2803x GPIO Input Qual Registers

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

GPAQSEL1 / GPAQSEL2 / GPBQSEL1

31 … 0

16 pins configured per register

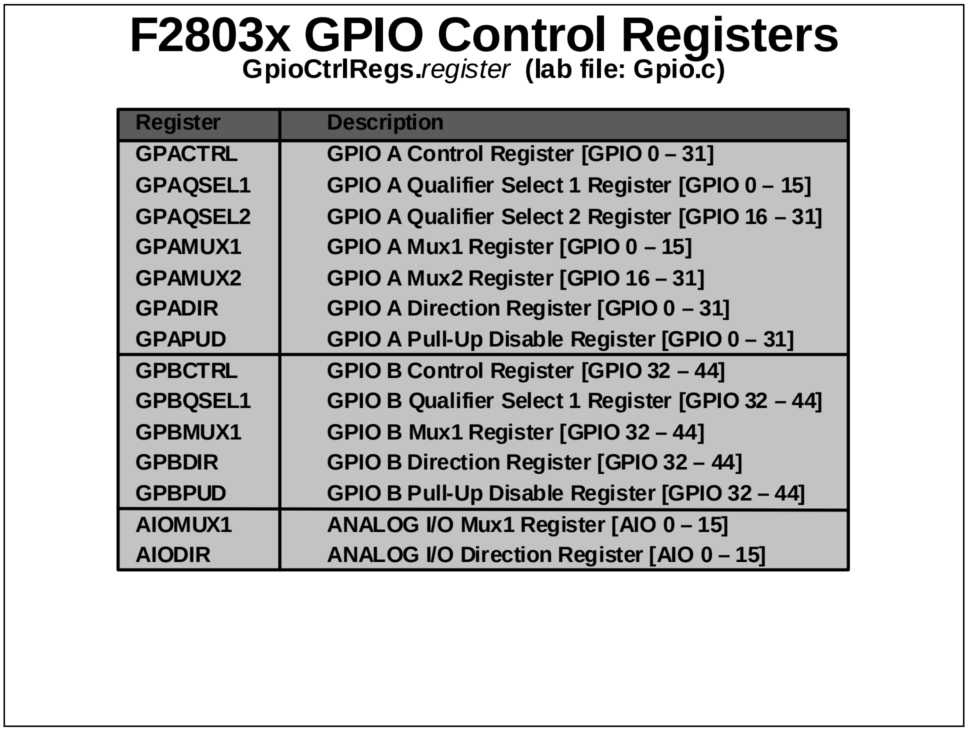
00 = sync to SYSCLKOUT only *
01 = qual to 3 samples
10 = qual to 6 samples
11 = no sync or qual (for peripheral only; GPIO same as 00)

GPACTRL / GPBCTRL

	31	24	16	8	0
	QUALPRD3	QUALPRD2	QUALPRD1	QUALPRD0	
B:	reserved	reserved	GPIO44-40	GPIO39-32	
A:	GPIO31-24	GPIO23-16	GPIO15-8	GPIO7-0	

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

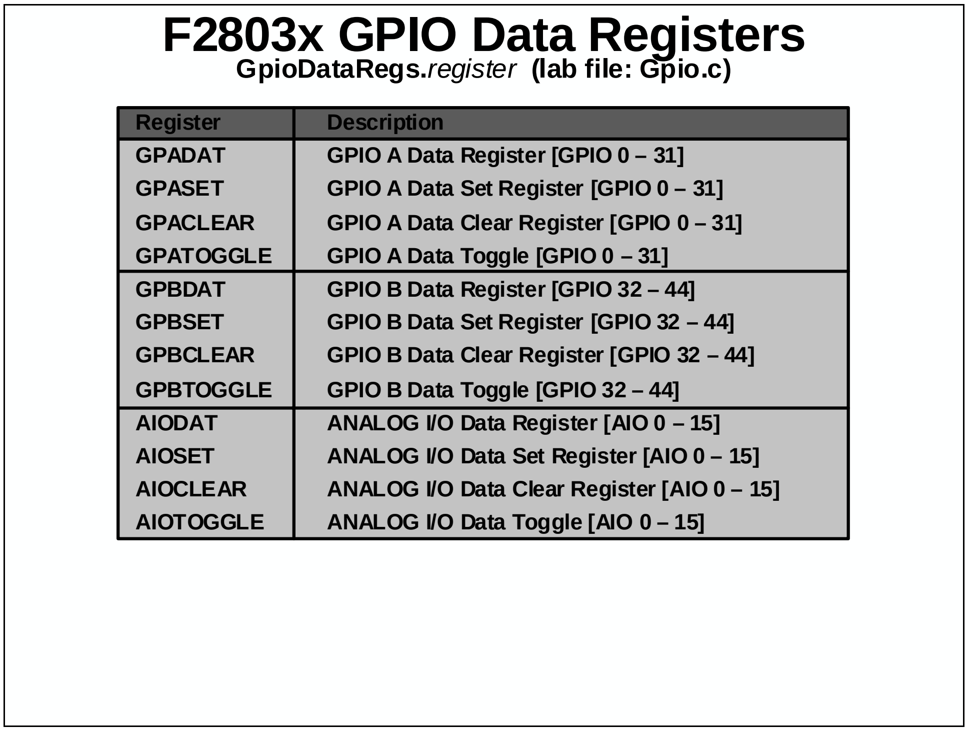
* reset default



F2803x GPIO Control Registers

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

Register	Description
GPACTRL	GPIO A Control Register [GPIO 0 – 31]
GPAQSEL1	GPIO A Qualifier Select 1 Register [GPIO 0 – 15]
GPAQSEL2	GPIO A Qualifier Select 2 Register [GPIO 16 – 31]
GPAMUX1	GPIO A Mux1 Register [GPIO 0 – 15]
GPAMUX2	GPIO A Mux2 Register [GPIO 16 – 31]
GPADIR	GPIO A Direction Register [GPIO 0 – 31]
GPAPUD	GPIO A Pull-Up Disable Register [GPIO 0 – 31]
GPBCTRL	GPIO B Control Register [GPIO 32 – 44]
GPBQSEL1	GPIO B Qualifier Select 1 Register [GPIO 32 – 44]
GPBMUX1	GPIO B Mux1 Register [GPIO 32 – 44]
GPBDIR	GPIO B Direction Register [GPIO 32 – 44]
GPBPUD	GPIO B Pull-Up Disable Register [GPIO 32 – 44]
AIOMUX1	ANALOG I/O Mux1 Register [AIO 0 – 15]
AIODIR	ANALOG I/O Direction Register [AIO 0 – 15]

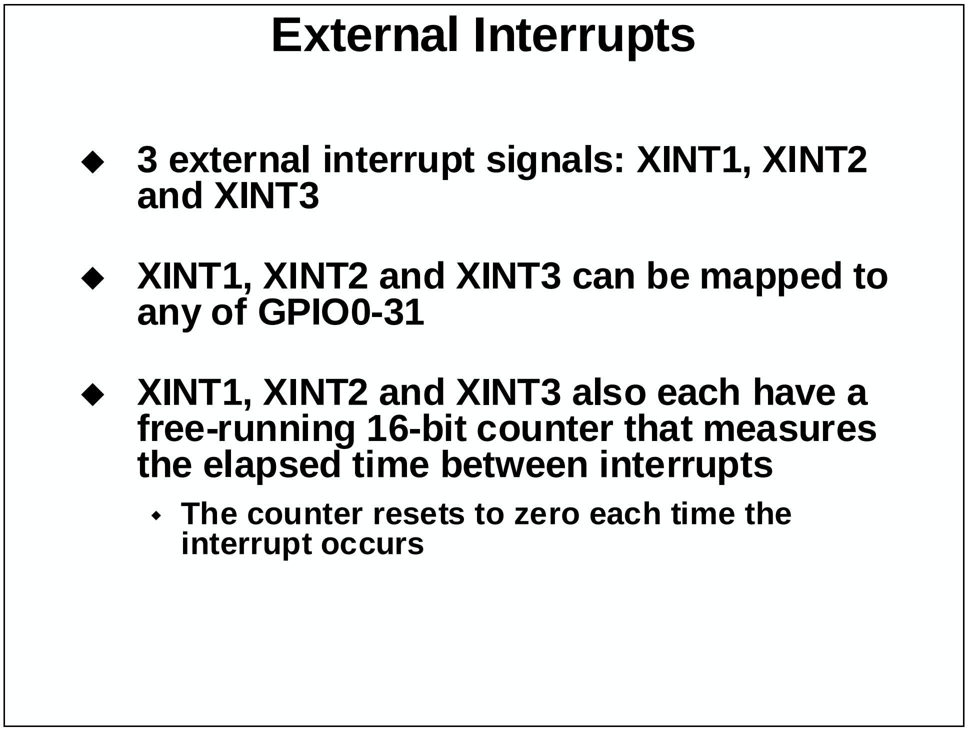


F2803x GPIO Data Registers

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

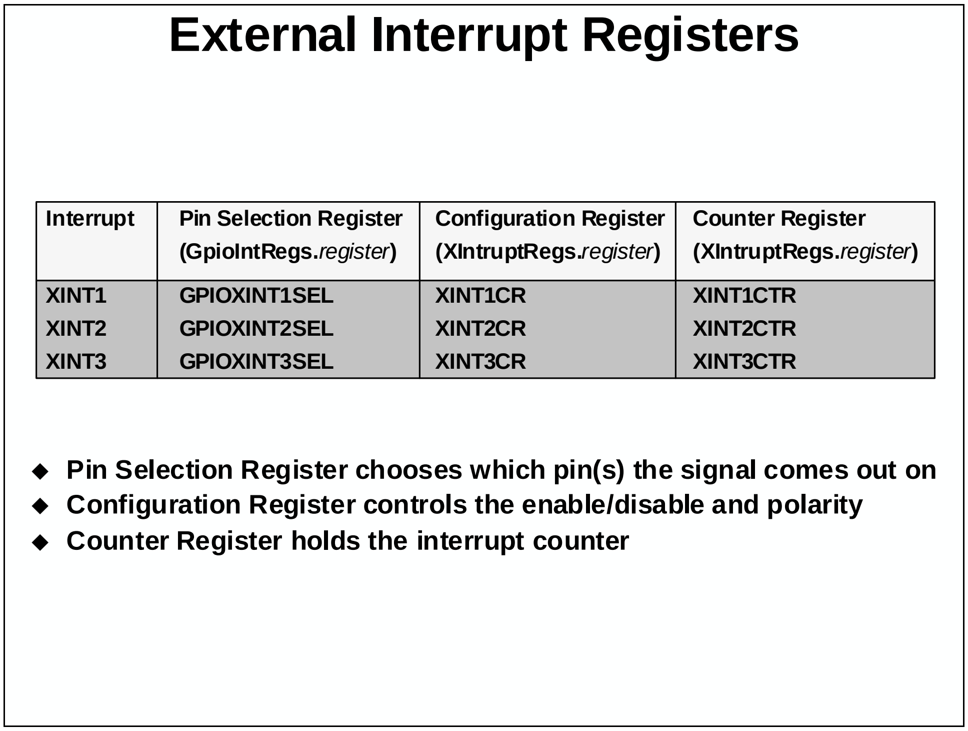
Register	Description
GPADAT	GPIO A Data Register [GPIO 0 – 31]
GPASET	GPIO A Data Set Register [GPIO 0 – 31]
GPACLEAR	GPIO A Data Clear Register [GPIO 0 – 31]
GPATOGGLE	GPIO A Data Toggle [GPIO 0 – 31]
GPBDAT	GPIO B Data Register [GPIO 32 – 44]
GPBSET	GPIO B Data Set Register [GPIO 32 – 44]
GPBCLEAR	GPIO B Data Clear Register [GPIO 32 – 44]
GPBTOGGLE	GPIO B Data Toggle [GPIO 32 – 44]
AIODAT	ANALOG I/O Data Register [AIO 0 – 15]
AIOSET	ANALOG I/O Data Set Register [AIO 0 – 15]
AIOCLEAR	ANALOG I/O Data Clear Register [AIO 0 – 15]
AIOTOGGLE	ANALOG I/O Data Toggle [AIO 0 – 15]

External Interrupts



External Interrupts

- 3 external interrupt signals: XINT1, XINT2 and XINT3
- XINT1, XINT2 and XINT3 can be mapped to any of GPIO0-31
- XINT1, XINT2 and XINT3 also each have a free-running 16-bit counter that measures the elapsed time between interrupts
  - The counter resets to zero each time the interrupt occurs

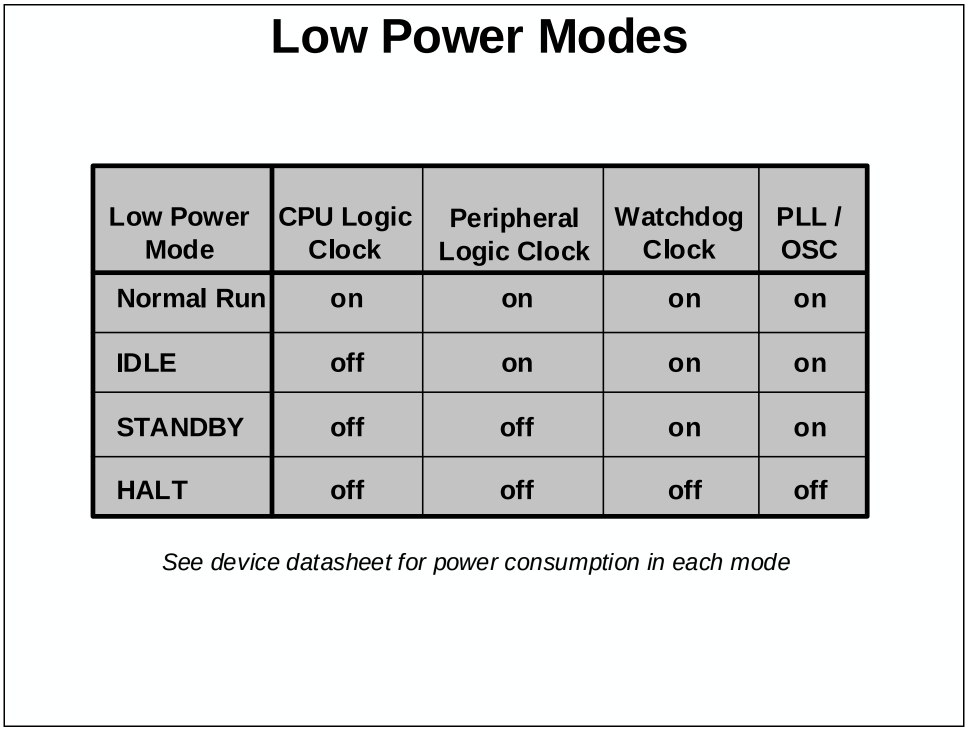


External Interrupt Registers

Interrupt	Pin Selection Register (GpioIntRegs.*register*)	Configuration Register (XIntruptRegs.*register*)	Counter Register (XIntruptRegs.*register*)
XINT1	GPIOXINT1SEL	XINT1CR	XINT1CTR
XINT2	GPIOXINT2SEL	XINT2CR	XINT2CTR
XINT3	GPIOXINT3SEL	XINT3CR	XINT3CTR

- Pin Selection Register chooses which pin(s) the signal comes out on
- Configuration Register controls the enable/disable and polarity
- Counter Register holds the interrupt counter

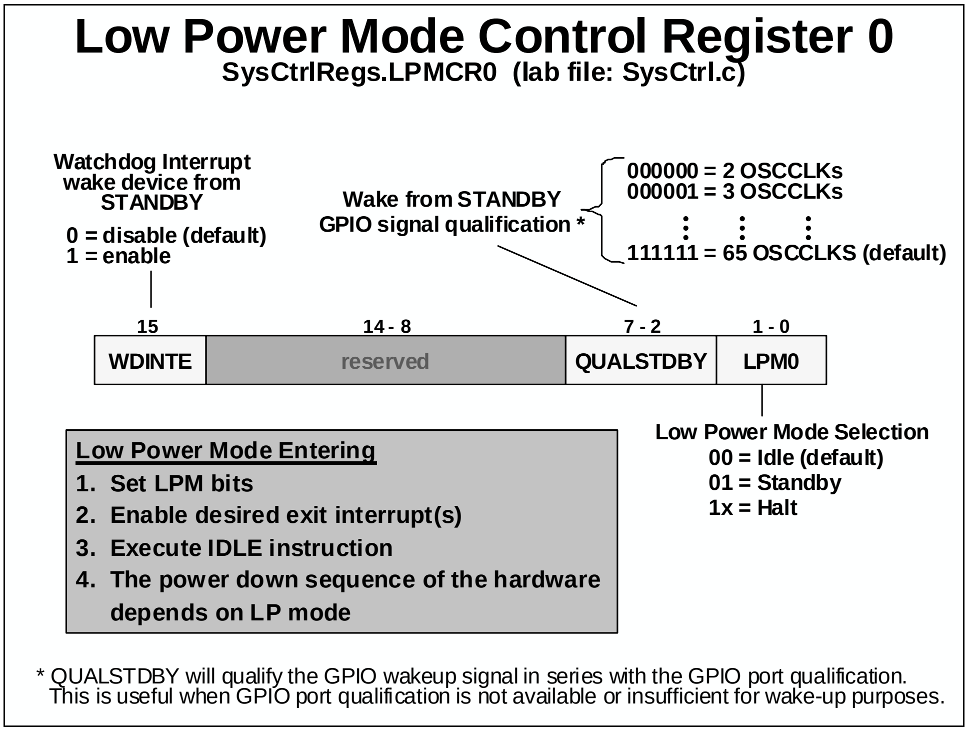
Low Power Modes



Low Power Modes

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

See device datasheet for power consumption in each mode



Low Power Mode Control Register 0

SysCtrlRegs.LPMCR0 (lab file: SysCtrl.c)

15	14 - 8	7 - 2	1 - 0
WDINTE	reserved	QUALSTDBY	LPM0

Watchdog Interrupt wake device from STANDBY

0 = disable (default)
1 = enable

Wake from STANDBY GPIO signal qualification *

000000 = 2 OSCCLKs
000001 = 3 OSCCLKs
⋮ ⋮ ⋮
111111 = 65 OSCCLKS (default)

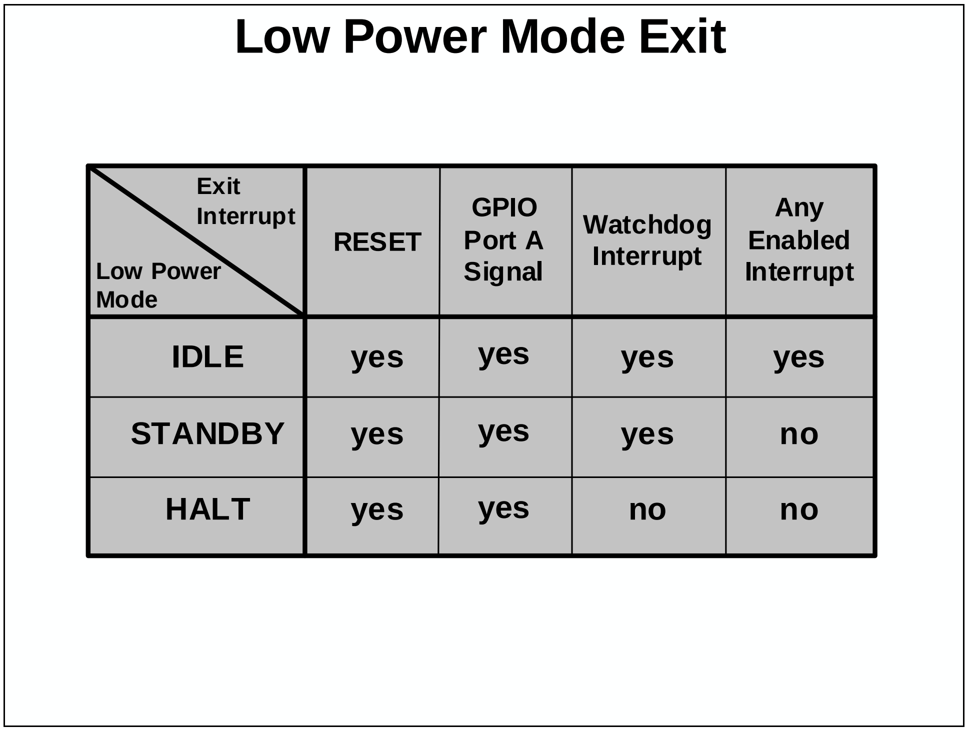
Low Power Mode Selection

00 = Idle (default)
01 = Standby
1x = Halt

Low Power Mode Entering

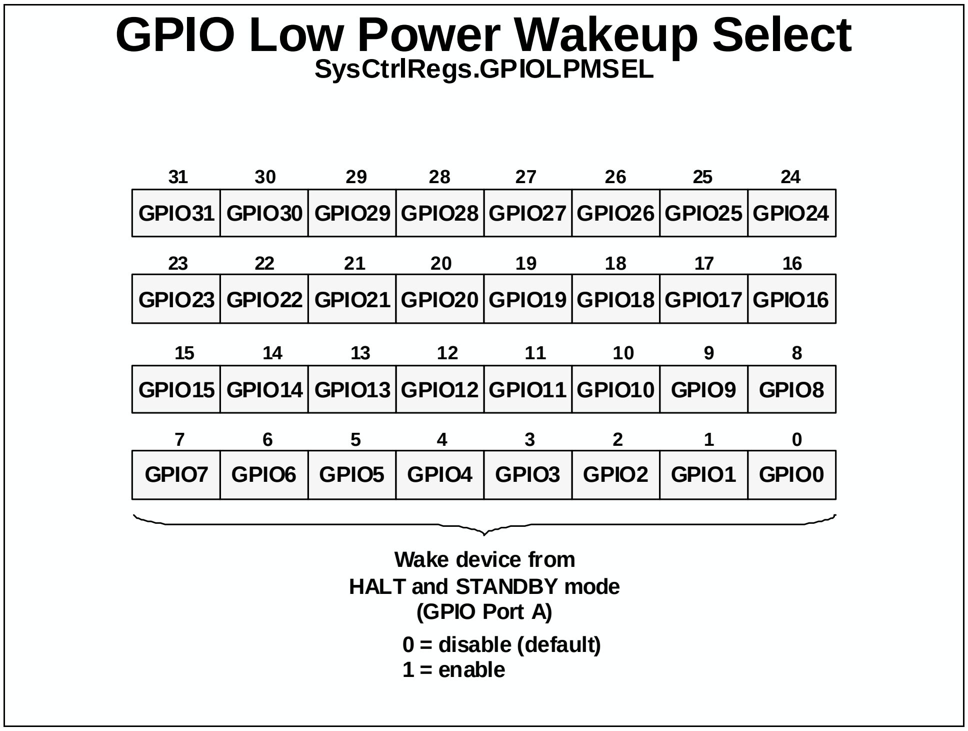
1. Set LPM bits
2. Enable desired exit interrupt(s)
3. Execute IDLE instruction
4. The power down sequence of the hardware depends on LP mode

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



Low Power Mode Exit

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



GPIO Low Power Wakeup Select

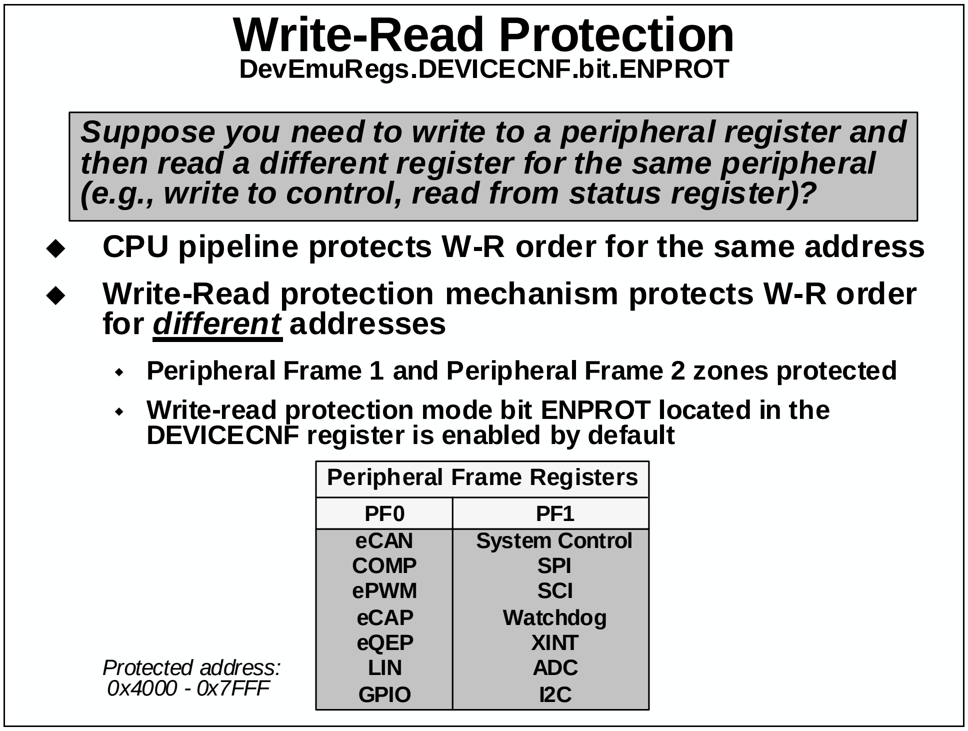
SysCtrlRegs.GPIOLPMSEL

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

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

0 = disable (default)
1 = enable

Register Protection



Write-Read Protection

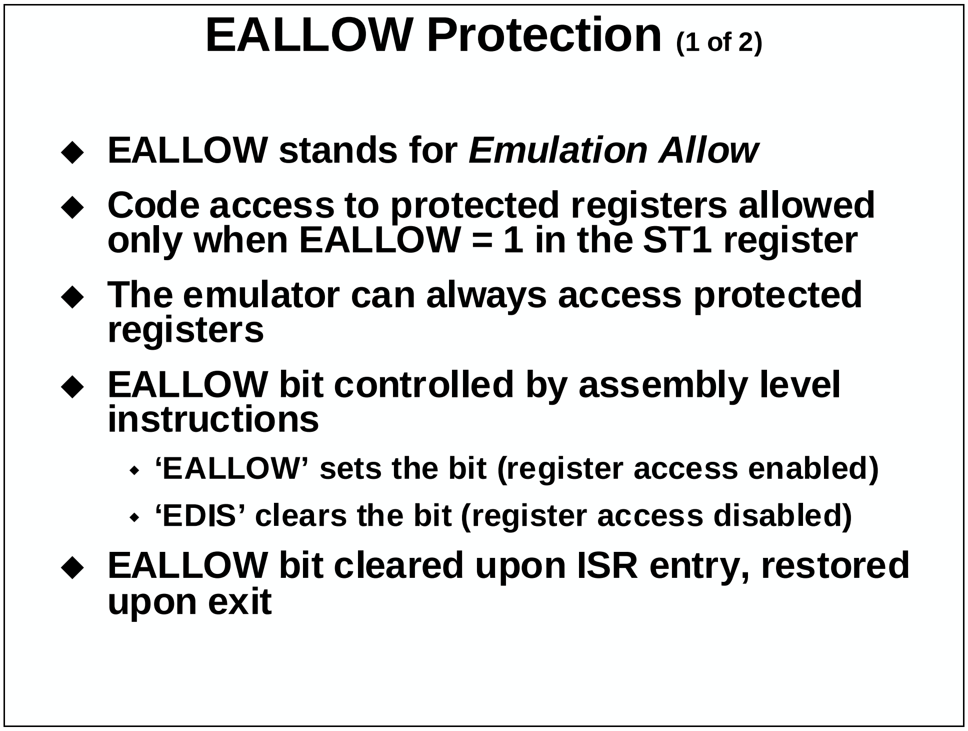
DevEmuRegs.DEVICECNF.bit.ENPROT

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

- **CPU pipeline protects W-R order for the same address**
- **Write-Read protection mechanism protects W-R order for *different* addresses**
  - **Peripheral Frame 1 and Peripheral Frame 2 zones protected**
  - **Write-read protection mode bit ENPROT located in the DEVICECNF register is enabled by default**

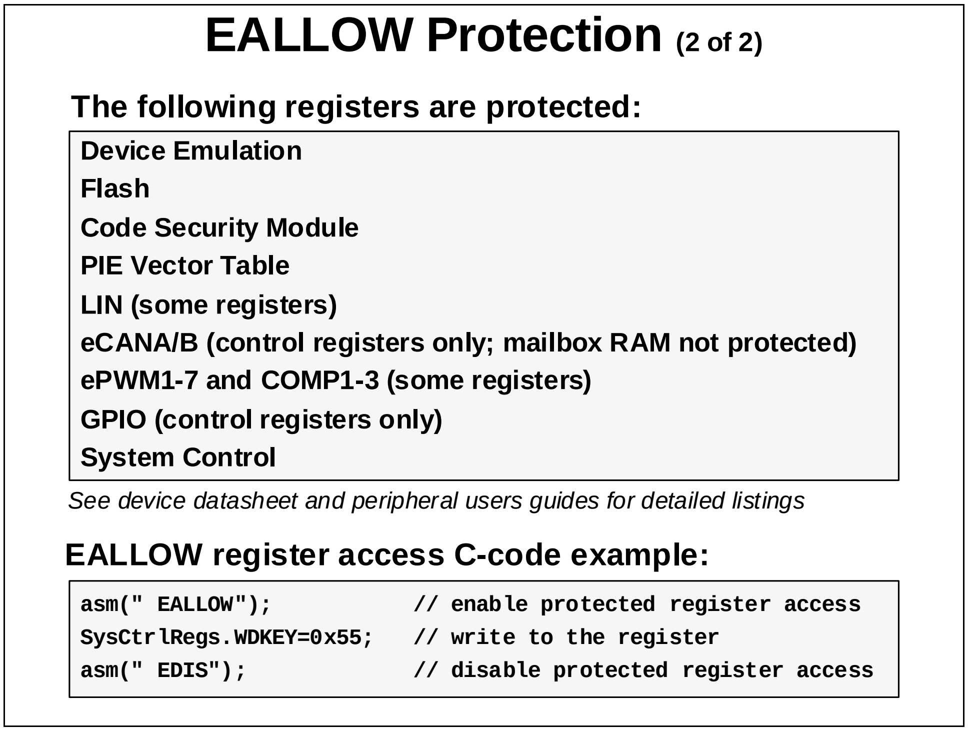
Peripheral Frame Registers	
PF0	**PF1**
eCAN	System Control
COMP	SPI
ePWM	SCI
eCAP	Watchdog
eQEP	XINT
LIN	ADC
GPIO	I2C

Protected address:
0x4000 - 0x7FFF



EALLOW Protection (1 of 2)

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



EALLOW Protection (2 of 2)

The following registers are protected:

Device Emulation
Flash
Code Security Module
PIE Vector Table
LIN (some registers)
eCANA/B (control registers only; mailbox RAM not protected)
ePWM1-7 and COMP1-3 (some registers)
GPIO (control registers only)
System Control

See device datasheet and peripheral users guides for detailed listings

EALLOW register access C-code example:

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

Lab 5: System Initialization

➢ Objective

The objective of this lab is to perform the processor system initialization. Additionally, the peripheral interrupt expansion (PIE) vectors will be initialized and tested using the information discussed in the previous module. This initialization process will be used again in all of the lab exercises throughout this workshop. The system initialization for this lab will consist of the following:

- Setup the clock module – PLL, LOSPCP = /4, low-power modes to default values, enable all module clocks
- Disable the watchdog – clear WD flag, disable watchdog, WD prescale = 1
- Setup watchdog system and 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 a 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 added and tested by using the watchdog to generate an interrupt. This lab will make use of the DSP2803x C-code header files to simplify the programming of the device, as well as take care of the register definitions and addresses. Please review these files, and make use of them in the future, as needed.

➢ Procedure

Create Project File

1. Create a new project called `Lab5.pjt` in `C:\C28x\Labs\Lab5` and add the following files to it:

```
CodeStartBranch.asm
DelayUs.asm
DSP2803x_GlobalVariableDefs.c
DSP2803x_Headers_nonBIOS.cmd
Gpio.c
Lab_5_6_7.cmd
Main_5.c
SysCtrl.c
Watchdog.c
```

   Note that include files, such as `DSP2803x_Device.h` and `Lab.h`, are automatically added at project build time. (Also, `DSP2803x_DefaultIsr.h` is automatically added and will be used with the interrupts in the second part of this lab exercise).

Project Build Options

2. We need to setup the search path to include the peripheral register header files. Click:

   `Project → Build Options…`

   Select the Compiler tab. In the Preprocessor Category, find the `Include Search Path (-i)` box and enter:

   `..\DSP2803x_headers\include`

   This is the path for the header files.

3. Select the Linker tab and set the Stack Size to 0x200.

4. Setup the compiler run-time support library. In the Libraries Category, find the `Include Libraries (-l)` box and enter: `rts2800_ml.lib`. Select `OK` and the Build Options window will close.

Modify Memory Configuration

5. Open and inspect the linker command file `Lab_5_6_7.cmd`. Notice that the user defined section `“codestart”` is being linked to a memory block named `BEGIN_M0`. The codestart section contains code that branches to the code entry point of the project. The bootloader must branch to the codestart section at the end of the boot process. Recall that the "Jump to M0 SARAM" bootloader mode branches to address 0x000000 upon bootloader completion.

   Modify the linker command file `Lab_5_6_7.cmd` to create a new memory block named `BEGIN_M0`: origin = 0x000000, length = 0x0002, in program memory. You will also need to modify the existing memory block `M0SARAM` in data memory to avoid any overlaps with this new memory block.

Setup System Initialization

6. Modify `SysCtrl.c` and `Watchdog.c` to implement the system initialization as described in the objective for this lab.

7. Open and inspect `Gpio.c`. Notice that the shared I/O pins have been set to the GPIO function. Save your work and close the modified files.

Build and Load

8. Click the `“Build”` button and watch the tools run in the build window. The output file should automatically load.

9. Under `Debug` on the menu bar click `“Reset CPU”`.

10. Under GEL on the menu bar click:
    EMU Boot Mode Select → EMU_BOOT_SARAM.
    This has the debugger load values into EMU_KEY and EMU_BMODE so that the bootloader will jump to "M0 SARAM" at 0x000000.

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

Run the Code – Watchdog Reset

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

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

14. Run your code for a few seconds by using the <F5> key, or using the Run button on the vertical toolbar, or using Debug → Run on the menu bar. After a few seconds halt your code by using Shift <F5>, or the Halt button on the vertical toolbar. Where did your code stop? Are the results as expected? If things went as expected, your code should be in the "main loop".

15. Modify the InitWatchdog() function to enable the watchdog (WDCR). This will enable the watchdog to function and cause a reset. Save the file and click the "Build" button.

16. Reset the CPU by performing the following steps:
    Click on Debug → Reset CPU
    Next click Debug → Go Main

17. 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 Cursor..

18. 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 when we invoked the EMU_BOOT_SARAM GEL function earlier in this lab. 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 the previous module.

19. Add the following files to the project:

    ```
    DefaultIsr_5.c
    PieCtrl_5_6_7_8_9_10.c
    PieVect_5_6_7_8_9_10.c
    ```

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

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

    ```
        InitPieCtrl();
    ```

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

    PIE group #:___________ # within group:__________

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

    Then modify `InitWatchdog()` to do the following:
    - Enable the "WAKEINT" interrupt in the PIE (Hint: use the `PieCtrlRegs` structure)
    - Enable the appropriate core interrupt in the IER register

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

24. Open and inspect `DefaultIsr_5.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.

25. Open and inspect `PieCtrl_5_6_7_8_9_10.c`. This file is used to initialize the PIE RAM and enable the PIE. The interrupt vector table located in `PieVect_5_6_7_8_9_10.c` is copied to the PIE RAM to setup the vectors for the interrupts. Close the modified and inspected files.

Build and Load

26. Click the “`Build`” button. Next reset the CPU, and then “`Go Main`”.

Run the Code – Watchdog Interrupt

27. Place the cursor in the `“main loop”` section, right click the mouse key and select `Run To Cursor`.

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

End of Exercise

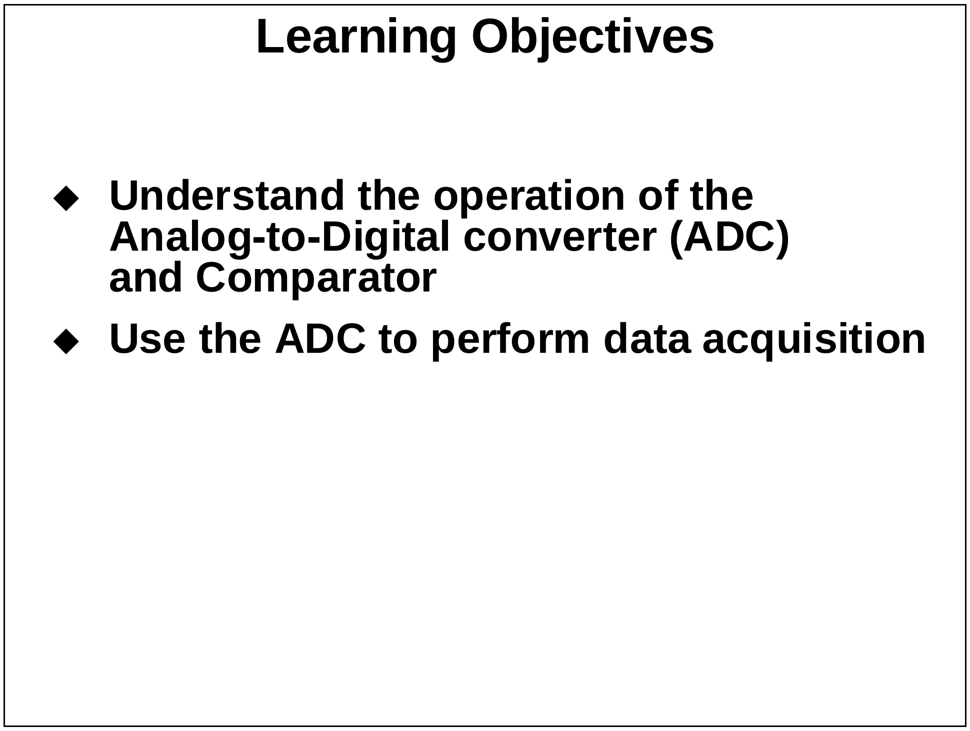
Note: By default, the watchdog timer is enabled out of reset. Code in the file `CodeStartBranch.asm` has been configured to disable the watchdog. This can be important for large C code projects (ask your instructor if this has not already been explained). During this lab exercise, the watchdog was actually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

Analog-to-Digital Converter and Comparator

Introduction

This module explains the operation of the analog-to-digital converter and comparator. The ADC system consists of a 12-bit analog-to-digital converter with up to 16 analog input channels. The analog input channels have a full range analog input of 0 to 3.3 volts or VREFHI/VREFLO ratiometric. Two input analog multiplexers are available, each supporting up to 8 analog input channels. Each multiplexer has its own dedicated sample and hold circuit. Therefore, sequential, as well as simultaneous sampling is supported. The ADC system is start-of-conversion (SOC) based where each independent SOCx (where x = 0 to 15) register configures the trigger source that starts the conversion, the channel to convert, and the acquisition (sample) window size. Up to 16 results registers are used to store the conversion values. Conversion triggers can be performed by an external trigger pin, software, an ePWM or CPU timer interrupt event, or a generated ADCINT1/2 interrupt.

Learning Objectives



Learning Objectives

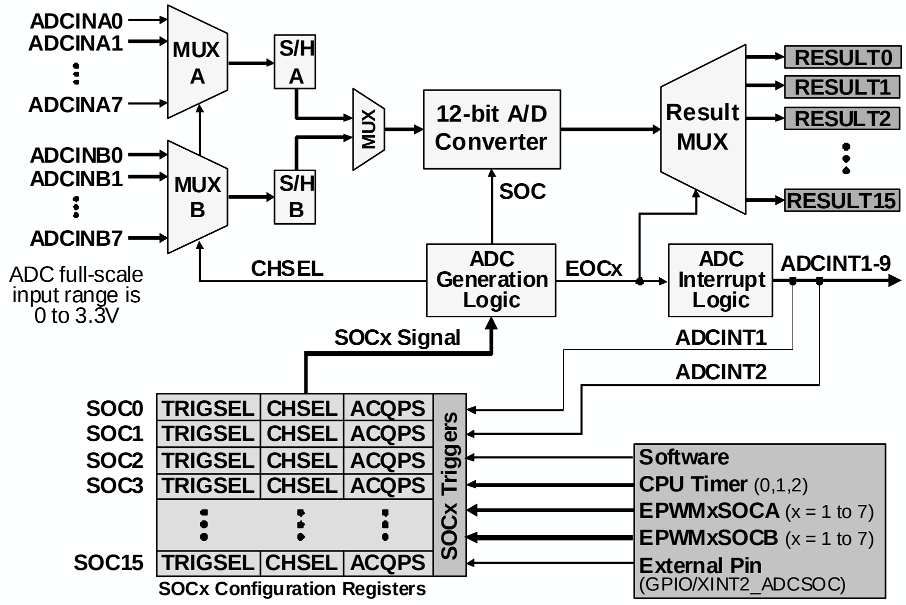
- ◆ **Understand the operation of the Analog-to-Digital converter (ADC) and Comparator**
- ◆ **Use the ADC to perform data acquisition**

Module Topics

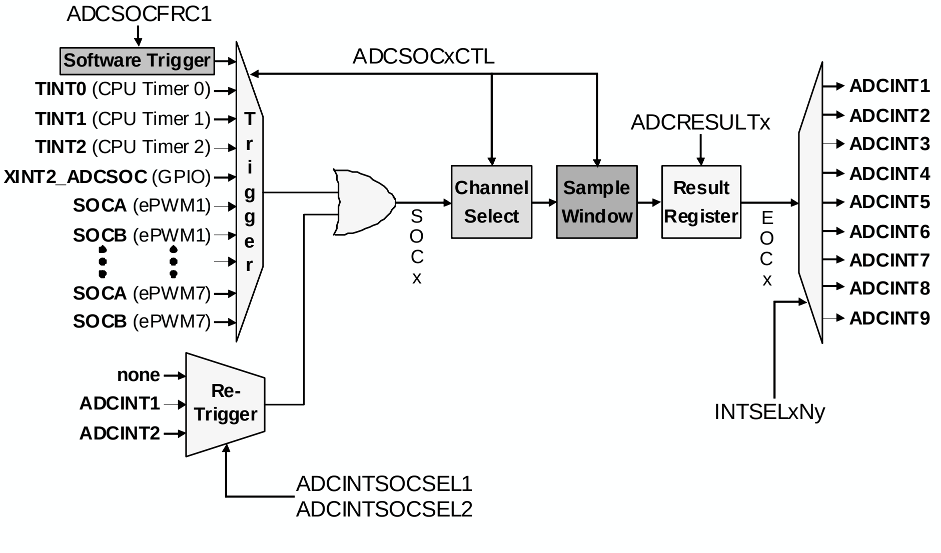
Analog-to-Digital Converter

ADC Block and Functional Diagrams

ADC Module Block Diagram



ADC SOCx Functional Diagram

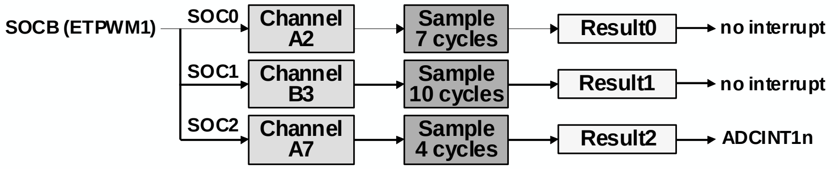


This block diagram is replicated 16 times

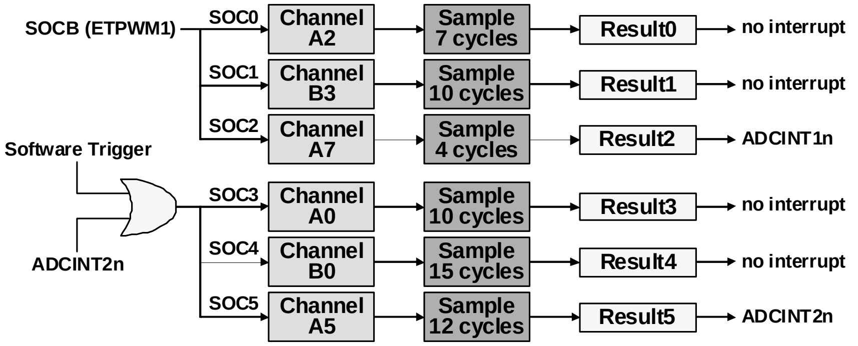
ADC Triggering

Example – ADC Triggering (1 of 2)

Sample A2 → B3 → A7 when ePWM1 SOCB is generated and then generate ADCINT1n:

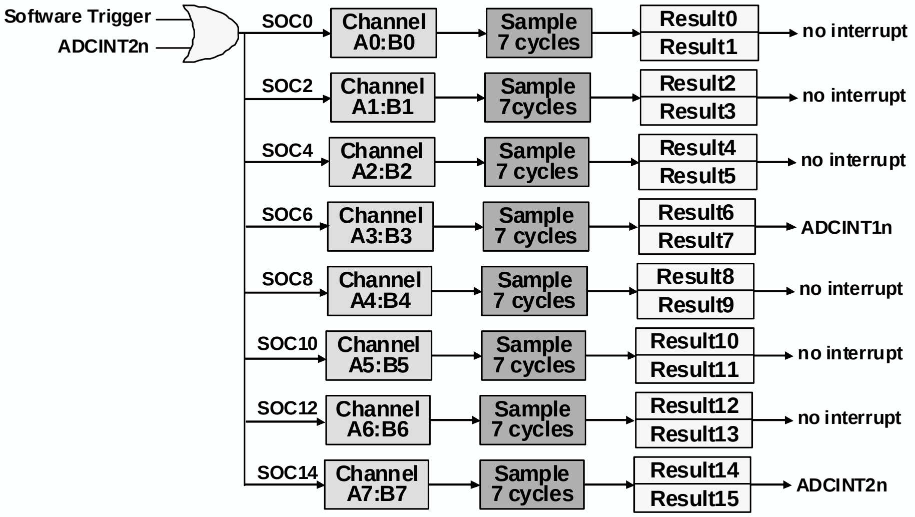


As above, but also sample A0 → B0 → A5 continuously and generate ADCINT2n:

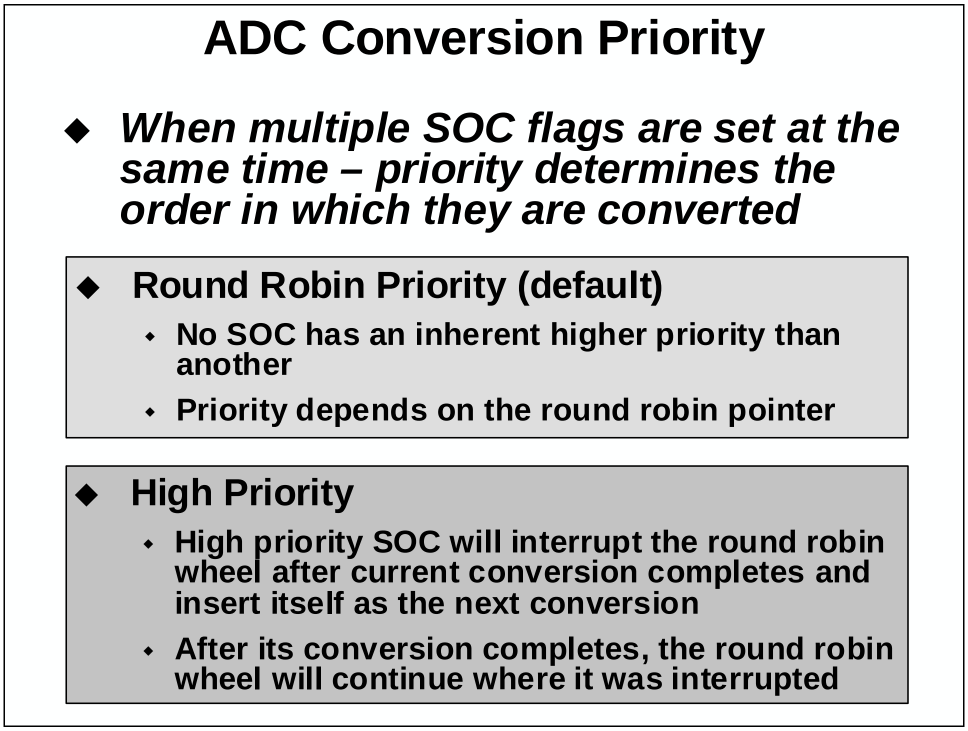


Example – ADC Triggering (2 of 2)

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



ADC Conversion Priority



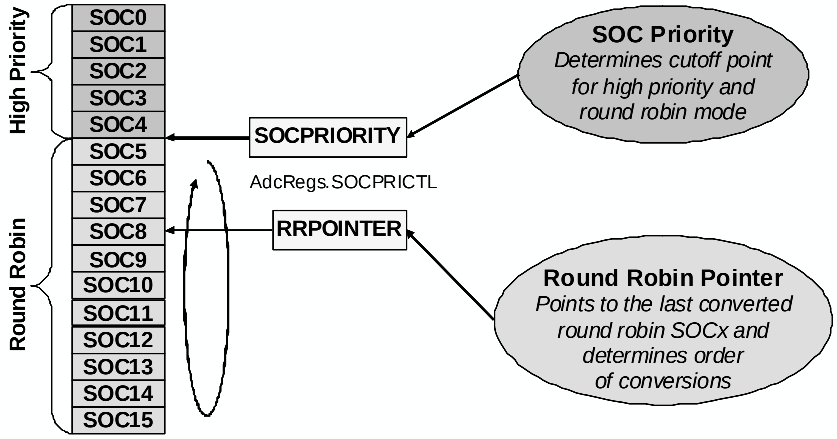
ADC Conversion Priority

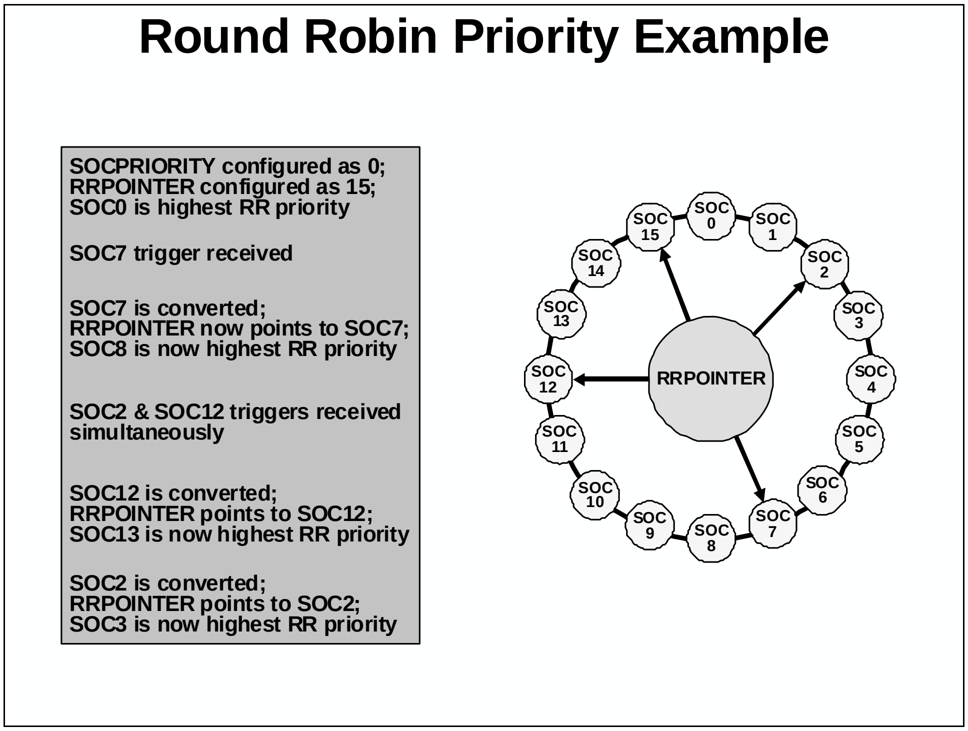
- ***When multiple SOC flags are set at the same time – priority determines the order in which they are converted***

- **Round Robin Priority (default)**
  - **No SOC has an inherent higher priority than another**
  - **Priority depends on the round robin pointer**

- **High Priority**
  - **High priority SOC will interrupt the round robin wheel after current conversion completes and insert itself as the next conversion**
  - **After its conversion completes, the round robin wheel will continue where it was interrupted**

Conversion Priority Functional Diagram





Round Robin Priority Example

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

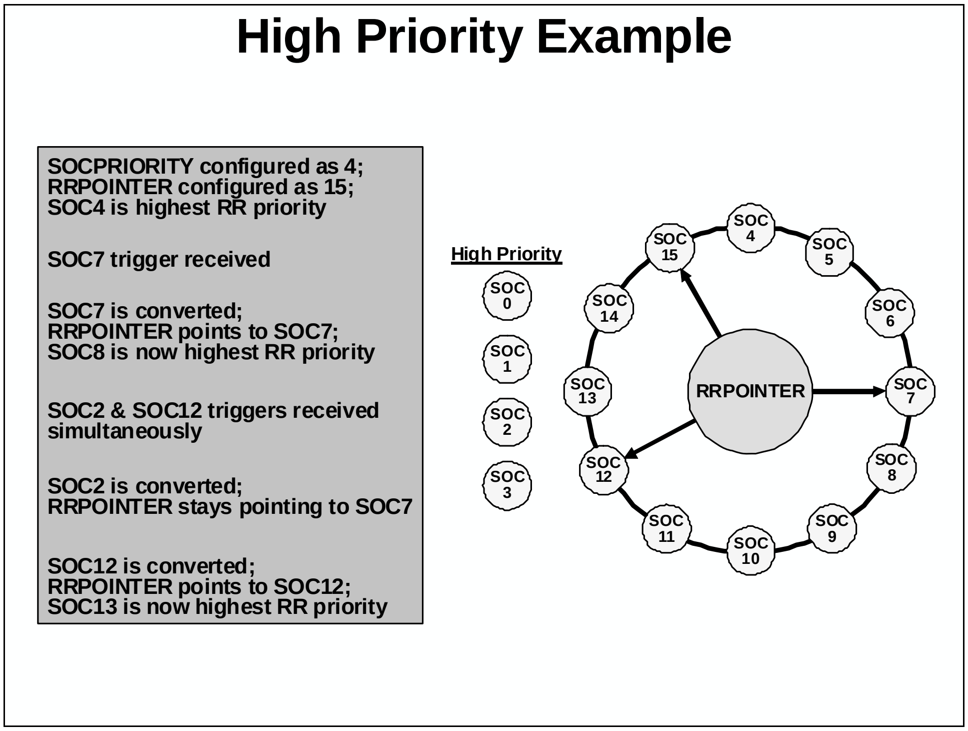
SOC7 trigger received

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

SOC2 & SOC12 triggers received simultaneously

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

SOC2 is converted;
RRPOINTER points to SOC2;
SOC3 is now highest RR priority



High Priority Example

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

SOC7 trigger received

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

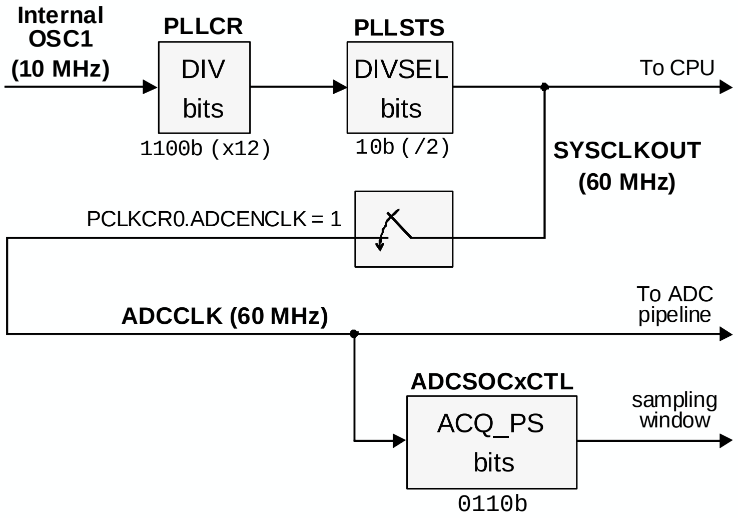
SOC2 & SOC12 triggers received simultaneously

SOC2 is converted;
RRPOINTER stays pointing to SOC7

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

ADC Clock and Timing

ADC Clocking Flow



Internal OSC1 (10 MHz)

PLLCR

DIV bits

1100b (x12)

PLLSTS

DIVSEL bits

10b (/2)

To CPU

SYSCLKOUT (60 MHz)

PCLKCR0.ADCENCLK = 1

ADCCLK (60 MHz)

To ADC pipeline

ADCSOCxCTL

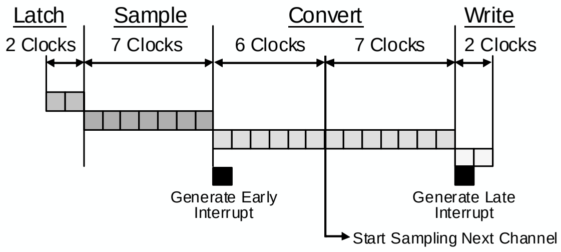
ACQ_PS bits

0110b

sampling window

sampling window = (ACQ_PS + 1)*(1/ADCCLK)

ADC Timing – Sequential Sampling



Latch

2 Clocks

Sample

7 Clocks

Convert

6 Clocks

7 Clocks

Write

2 Clocks

Generate Early Interrupt

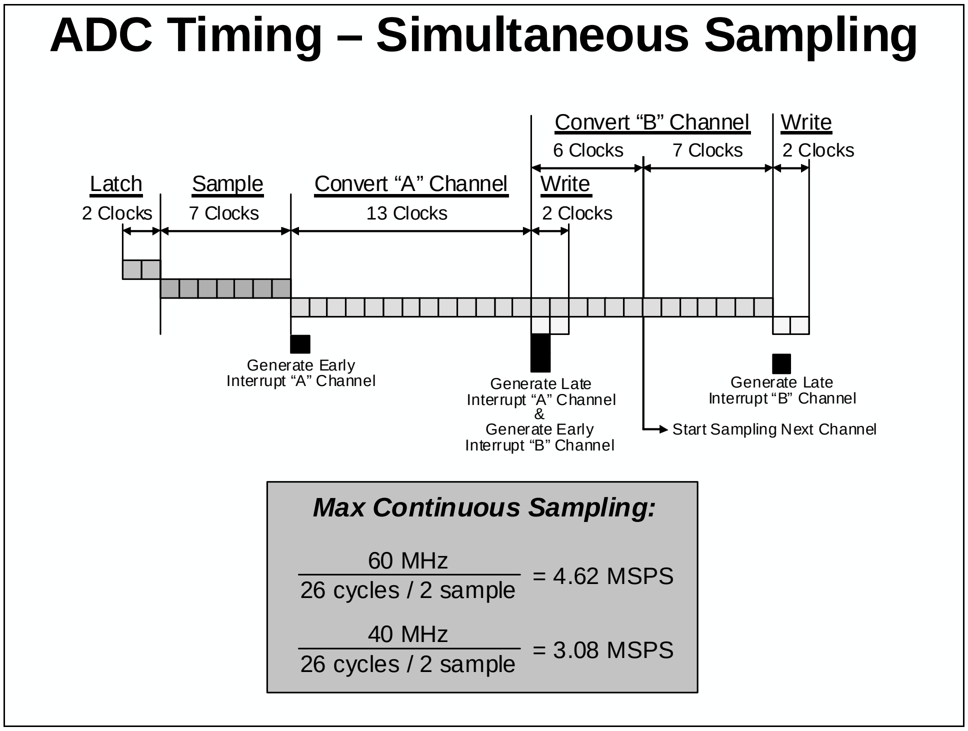
Generate Late Interrupt

Start Sampling Next Channel

Max Continuous Sampling:

$$\frac{60\text{ MHz}}{13\text{ cycles / 1 sample}} = 4.62\text{ MSPS}$$

$$\frac{40\text{ MHz}}{13\text{ cycles / 1 sample}} = 3.08\text{ MSPS}$$



ADC Timing – Simultaneous Sampling

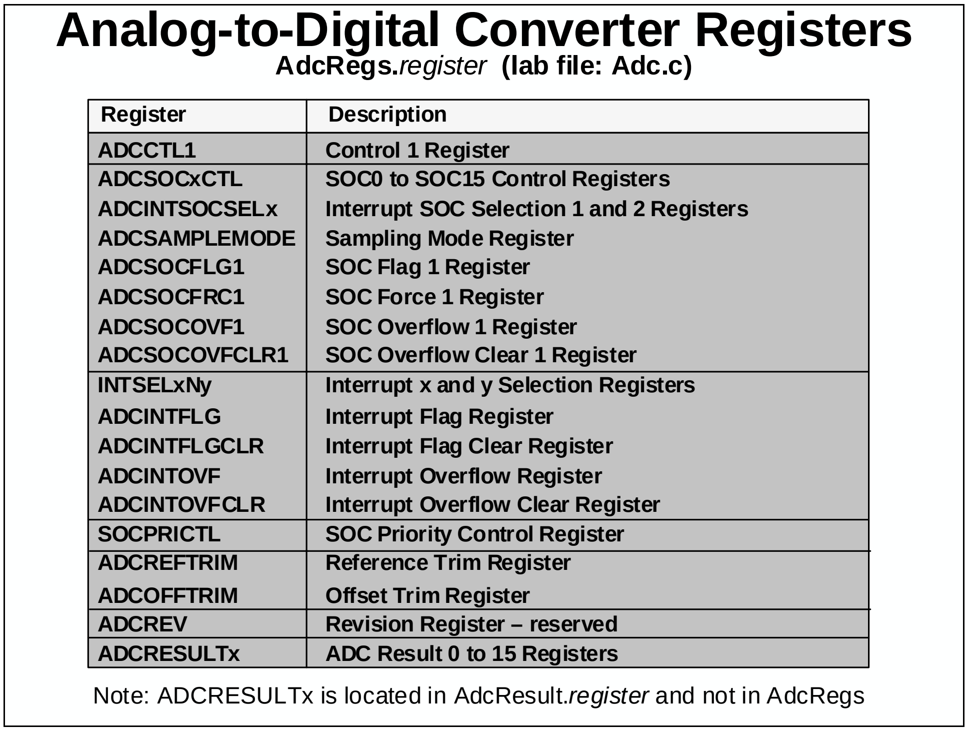
Convert "B" Channel
Write
6 Clocks
7 Clocks
2 Clocks
Latch
Sample
Convert "A" Channel
Write
2 Clocks
7 Clocks
13 Clocks
2 Clocks
Generate Early Interrupt "A" Channel
Generate Late Interrupt "A" Channel & Generate Early Interrupt "B" Channel
Generate Late Interrupt "B" Channel
Start Sampling Next Channel

Max Continuous Sampling:

$$\frac{60\text{ MHz}}{26\text{ cycles / 2 sample}} = 4.62\text{ MSPS}$$

$$\frac{40\text{ MHz}}{26\text{ cycles / 2 sample}} = 3.08\text{ MSPS}$$

ADC Converter Registers



Analog-to-Digital Converter Registers

AdcRegs.*register* (lab file: Adc.c)

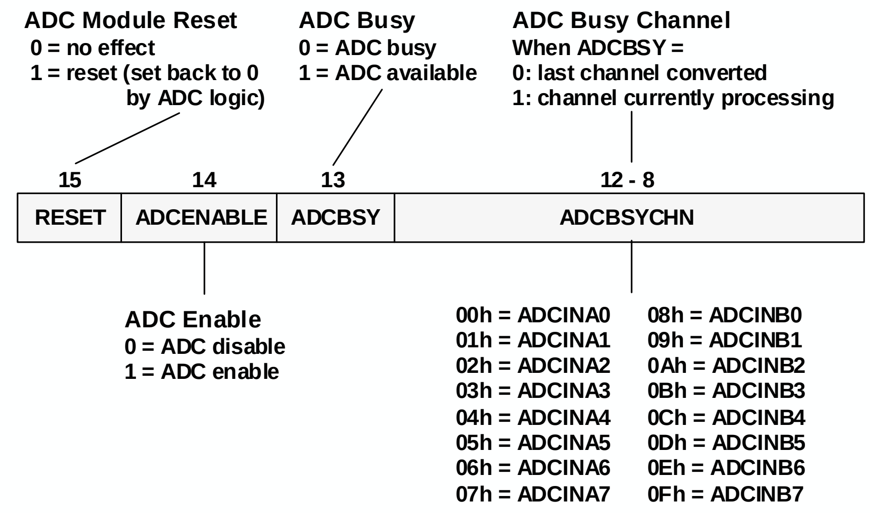
Register	Description
ADCCTL1	**Control 1 Register**
ADCSOCxCTL	**SOC0 to SOC15 Control Registers**
ADCINTSOCSELx	**Interrupt SOC Selection 1 and 2 Registers**
ADCSAMPLEMODE	**Sampling Mode Register**
ADCSOCFLG1	**SOC Flag 1 Register**
ADCSOCFRC1	**SOC Force 1 Register**
ADCSOCOVF1	**SOC Overflow 1 Register**
ADCSOCOVFCLR1	**SOC Overflow Clear 1 Register**
INTSELxNy	**Interrupt x and y Selection Registers**
ADCINTFLG	**Interrupt Flag Register**
ADCINTFLGCLR	**Interrupt Flag Clear Register**
ADCINTOVF	**Interrupt Overflow Register**
ADCINTOVFCLR	**Interrupt Overflow Clear Register**
SOCPRICTL	**SOC Priority Control Register**
ADCREFTRIM	**Reference Trim Register**
ADCOFFTRIM	**Offset Trim Register**
ADCREV	**Revision Register – reserved**
ADCRESULTx	**ADC Result 0 to 15 Registers**

Note: ADCRESULTx is located in AdcResult.*register* and not in AdcRegs

ADC Control Register 1

AdcRegs.ADCCTL1

Upper Register:



ADC Module Reset
0 = no effect
1 = reset (set back to 0 by ADC logic)

ADC Busy
0 = ADC busy
1 = ADC available

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

15	14	13	12 - 8
RESET	ADCENABLE	ADCBSY	ADCBSYCHN

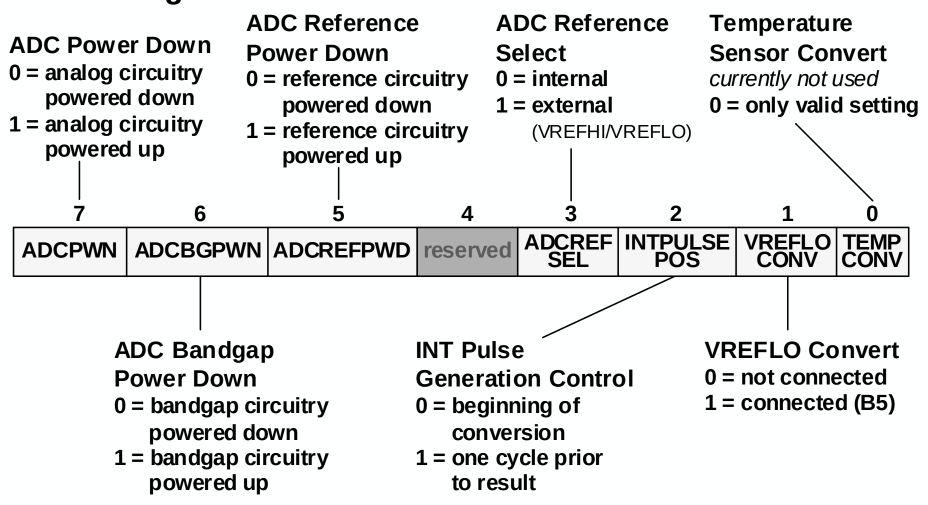
ADC Enable
0 = ADC disable
1 = ADC enable

00h = ADCINA0	08h = ADCINB0
01h = ADCINA1	09h = ADCINB1
02h = ADCINA2	0Ah = ADCINB2
03h = ADCINA3	0Bh = ADCINB3
04h = ADCINA4	0Ch = ADCINB4
05h = ADCINA5	0Dh = ADCINB5
06h = ADCINA6	0Eh = ADCINB6
07h = ADCINA7	0Fh = ADCINB7

ADC Control Register 1

AdcRegs.ADCCTL1

Lower Register:



ADC Power Down
0 = analog circuitry powered down
1 = analog circuitry powered up

ADC Reference Power Down
0 = reference circuitry powered down
1 = reference circuitry powered up

ADC Reference Select
0 = internal
1 = external
(VREFHI/VREFLO)

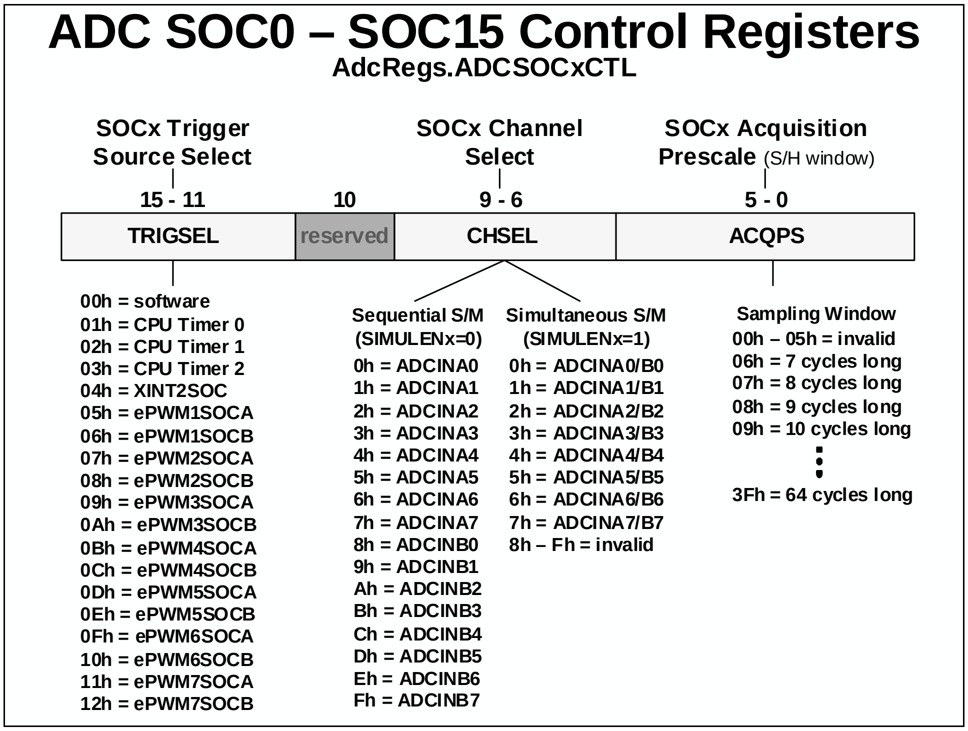
Temperature Sensor Convert
currently not used
0 = only valid setting

7	6	5	4	3	2	1	0
ADCPWN	ADCBGPWN	ADCREFPWD	reserved	ADCREF SEL	INTPULSE POS	VREFLO CONV	TEMP CONV

ADC Bandgap Power Down
0 = bandgap circuitry powered down
1 = bandgap circuitry powered up

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

VREFLO Convert
0 = not connected
1 = connected (B5)



ADC SOC0 – SOC15 Control Registers

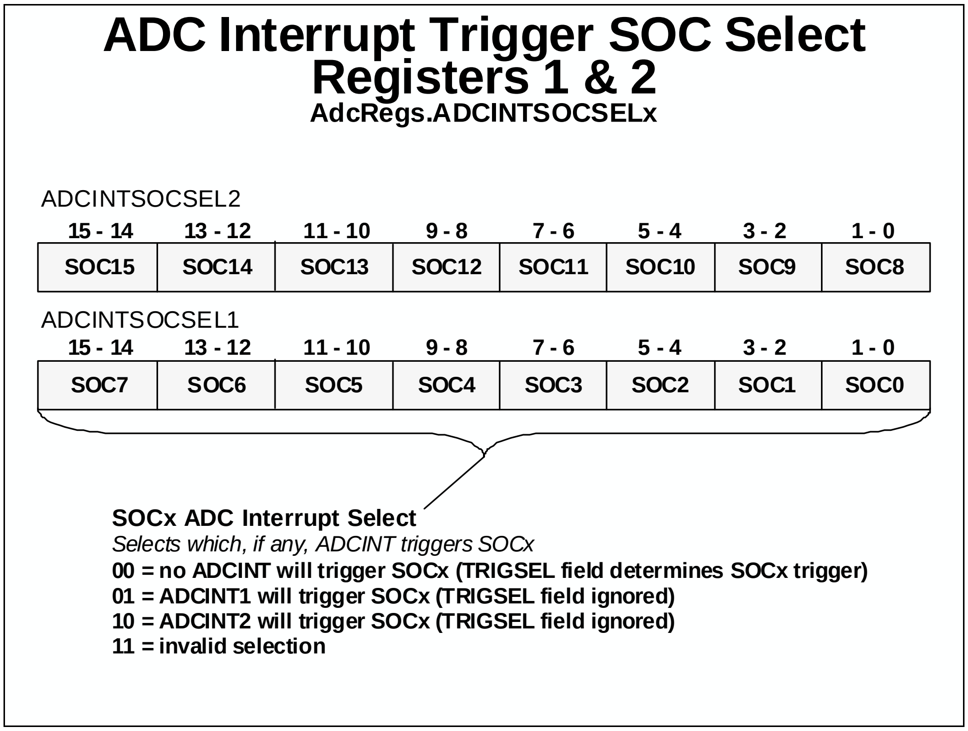
AdcRegs.ADCSOCxCTL

SOCx Trigger Source Select		SOCx Channel Select	SOCx Acquisition Prescale (S/H window)
15 - 11	10	9 - 6	5 - 0
TRIGSEL	reserved	CHSEL	ACQPS

00h = software
01h = CPU Timer 0
02h = CPU Timer 1
03h = CPU Timer 2
04h = XINT2SOC
05h = ePWM1SOCA
06h = ePWM1SOCB
07h = ePWM2SOCA
08h = ePWM2SOCB
09h = ePWM3SOCA
0Ah = ePWM3SOCB
0Bh = ePWM4SOCA
0Ch = ePWM4SOCB
0Dh = ePWM5SOCA
0Eh = ePWM5SOCB
0Fh = ePWM6SOCA
10h = ePWM6SOCB
11h = ePWM7SOCA
12h = ePWM7SOCB

Sequential S/M (SIMULENx=0)	Simultaneous S/M (SIMULENx=1)
0h = ADCINA0	0h = ADCINA0/B0
1h = ADCINA1	1h = ADCINA1/B1
2h = ADCINA2	2h = ADCINA2/B2
3h = ADCINA3	3h = ADCINA3/B3
4h = ADCINA4	4h = ADCINA4/B4
5h = ADCINA5	5h = ADCINA5/B5
6h = ADCINA6	6h = ADCINA6/B6
7h = ADCINA7	7h = ADCINA7/B7
8h = ADCINB0	8h – Fh = invalid
9h = ADCINB1	
Ah = ADCINB2	
Bh = ADCINB3	
Ch = ADCINB4	
Dh = ADCINB5	
Eh = ADCINB6	
Fh = ADCINB7	

Sampling Window
00h – 05h = invalid
06h = 7 cycles long
07h = 8 cycles long
08h = 9 cycles long
09h = 10 cycles long
⋮
3Fh = 64 cycles long



ADC Interrupt Trigger SOC Select Registers 1 & 2

AdcRegs.ADCINTSOCSELx

ADCINTSOCSEL2

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

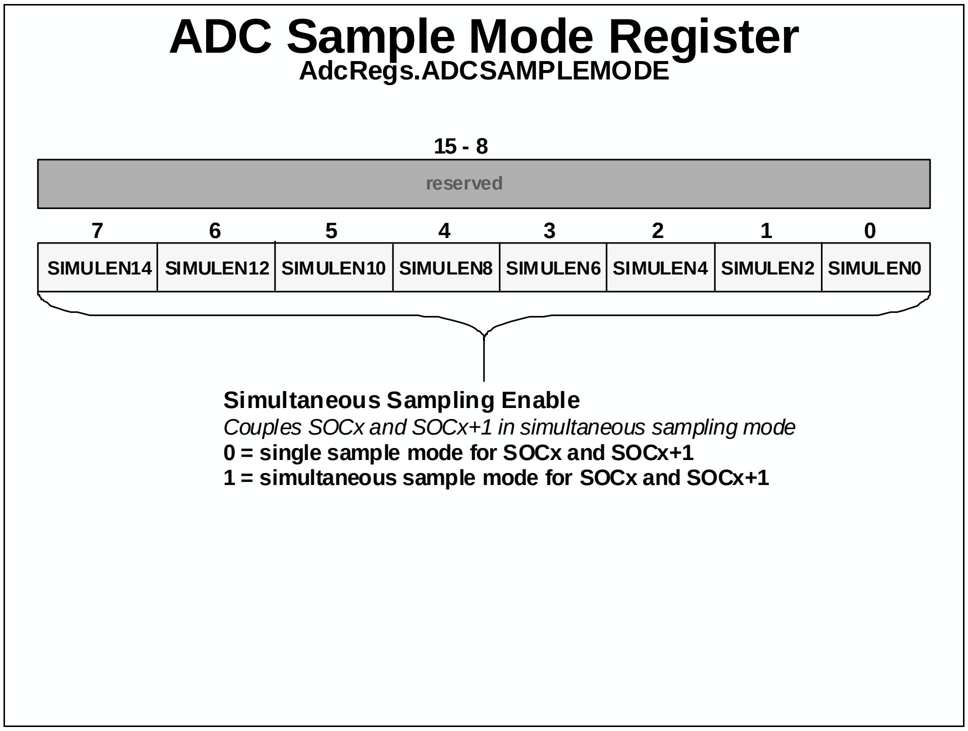
ADCINTSOCSEL1

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

SOCx ADC Interrupt Select

Selects which, if any, ADCINT triggers SOCx

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



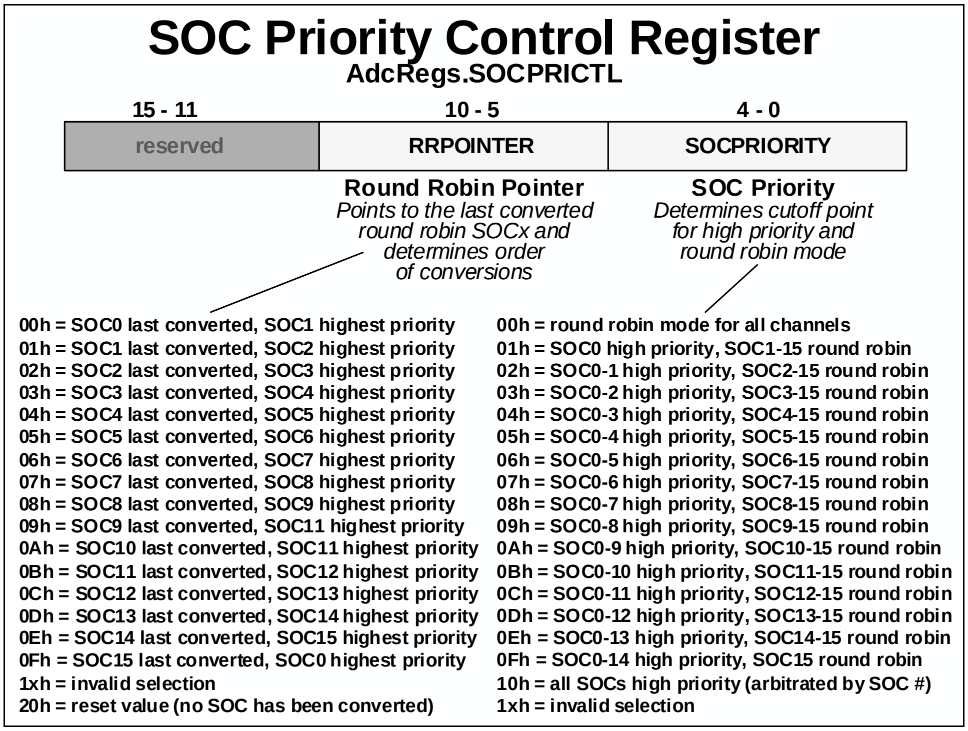
ADC Sample Mode Register

AdcRegs.ADCSAMPLEMODE

15 - 8
reserved

7	6	5	4	3	2	1	0
SIMULEN14	SIMULEN12	SIMULEN10	SIMULEN8	SIMULEN6	SIMULEN4	SIMULEN2	SIMULEN0

Simultaneous Sampling Enable
Couples SOCx and SOCx+1 in simultaneous sampling mode
0 = single sample mode for SOCx and SOCx+1
1 = simultaneous sample mode for SOCx and SOCx+1



SOC Priority Control Register

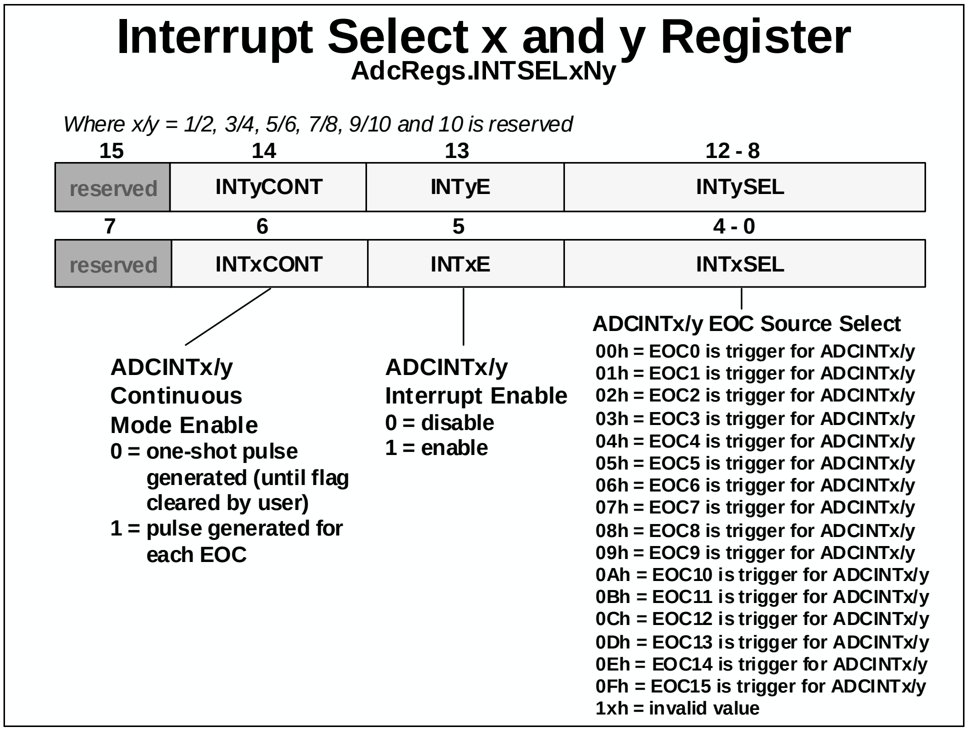
AdcRegs.SOCPRICTL

15 - 11	10 - 5	4 - 0
reserved	RRPOINTER	SOCPRIORITY

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

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

RRPOINTER	SOCPRIORITY
00h = SOC0 last converted, SOC1 highest priority	00h = round robin mode for all channels
01h = SOC1 last converted, SOC2 highest priority	01h = SOC0 high priority, SOC1-15 round robin
02h = SOC2 last converted, SOC3 highest priority	02h = SOC0-1 high priority, SOC2-15 round robin
03h = SOC3 last converted, SOC4 highest priority	03h = SOC0-2 high priority, SOC3-15 round robin
04h = SOC4 last converted, SOC5 highest priority	04h = SOC0-3 high priority, SOC4-15 round robin
05h = SOC5 last converted, SOC6 highest priority	05h = SOC0-4 high priority, SOC5-15 round robin
06h = SOC6 last converted, SOC7 highest priority	06h = SOC0-5 high priority, SOC6-15 round robin
07h = SOC7 last converted, SOC8 highest priority	07h = SOC0-6 high priority, SOC7-15 round robin
08h = SOC8 last converted, SOC9 highest priority	08h = SOC0-7 high priority, SOC8-15 round robin
09h = SOC9 last converted, SOC11 highest priority	09h = SOC0-8 high priority, SOC9-15 round robin
0Ah = SOC10 last converted, SOC11 highest priority	0Ah = SOC0-9 high priority, SOC10-15 round robin
0Bh = SOC11 last converted, SOC12 highest priority	0Bh = SOC0-10 high priority, SOC11-15 round robin
0Ch = SOC12 last converted, SOC13 highest priority	0Ch = SOC0-11 high priority, SOC12-15 round robin
0Dh = SOC13 last converted, SOC14 highest priority	0Dh = SOC0-12 high priority, SOC13-15 round robin
0Eh = SOC14 last converted, SOC15 highest priority	0Eh = SOC0-13 high priority, SOC14-15 round robin
0Fh = SOC15 last converted, SOC0 highest priority	0Fh = SOC0-14 high priority, SOC15 round robin
1xh = invalid selection	10h = all SOCs high priority (arbitrated by SOC #)
20h = reset value (no SOC has been converted)	1xh = invalid selection



Interrupt Select x and y Register

AdcRegs.INTSELxNy

Where x/y = 1/2, 3/4, 5/6, 7/8, 9/10 and 10 is reserved

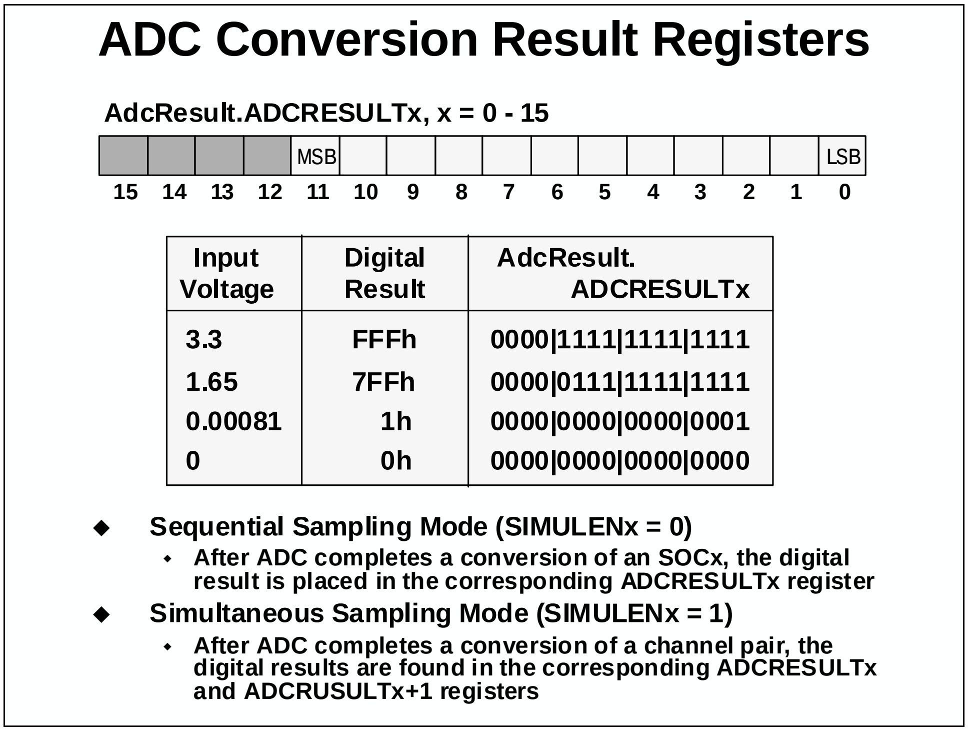
15	14	13	12 - 8
reserved	INTyCONT	INTyE	INTySEL

7	6	5	4 - 0
reserved	INTxCONT	INTxE	INTxSEL

ADCINTx/y Continuous Mode Enable
0 = one-shot pulse generated (until flag cleared by user)
1 = pulse generated for each EOC

ADCINTx/y Interrupt Enable
0 = disable
1 = enable

ADCINTx/y EOC Source Select
00h = EOC0 is trigger for ADCINTx/y
01h = EOC1 is trigger for ADCINTx/y
02h = EOC2 is trigger for ADCINTx/y
03h = EOC3 is trigger for ADCINTx/y
04h = EOC4 is trigger for ADCINTx/y
05h = EOC5 is trigger for ADCINTx/y
06h = EOC6 is trigger for ADCINTx/y
07h = EOC7 is trigger for ADCINTx/y
08h = EOC8 is trigger for ADCINTx/y
09h = EOC9 is trigger for ADCINTx/y
0Ah = EOC10 is trigger for ADCINTx/y
0Bh = EOC11 is trigger for ADCINTx/y
0Ch = EOC12 is trigger for ADCINTx/y
0Dh = EOC13 is trigger for ADCINTx/y
0Eh = EOC14 is trigger for ADCINTx/y
0Fh = EOC15 is trigger for ADCINTx/y
1xh = invalid value



ADC Conversion Result Registers

AdcResult.ADCRESULTx, x = 0 - 15

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

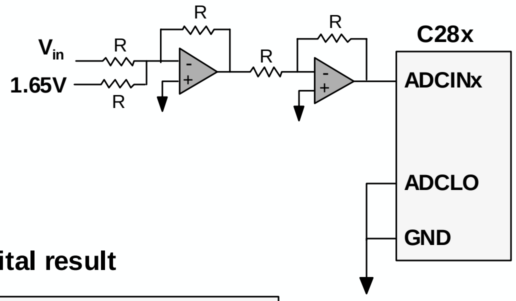
Input Voltage	Digital Result	AdcResult. ADCRESULTx
3.3	FFFh	0000|1111|1111|1111
1.65	7FFh	0000|0111|1111|1111
0.00081	1h	0000|0000|0000|0001
0	0h	0000|0000|0000|0000

- Sequential Sampling Mode (SIMULENx = 0)
  - After ADC completes a conversion of an SOCx, the digital result is placed in the corresponding ADCRESULTx register
- Simultaneous Sampling Mode (SIMULENx = 1)
  - After ADC completes a conversion of a channel pair, the digital results are found in the corresponding ADCRESULTx and ADCRUSULTx+1 registers

How Can We Handle Signed Input Voltages?

Example: -1.65 V ≤ V_{in} ≤ +1.65 V

1) Add 1.65 volts to the analog input



2) Subtract "1.65" from the digital result

```
#include "DSP2803x_Device.h"
#define  offset  0x07FF
void main(void)
{
   int16 value;            // signed

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

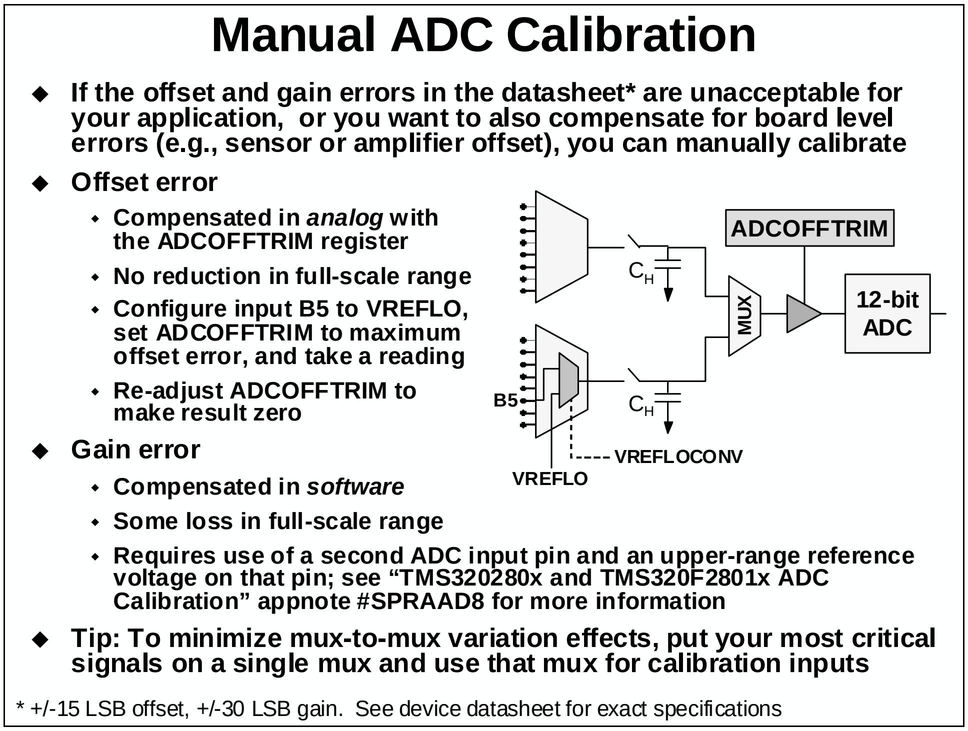
ADC Calibration and Reference

Built-In ADC Calibration

- **TI reserved OTP contains device specific calibration data for the ADC and internal oscillators**
- **The Boot ROM contains a Device_cal() routine that copies the calibration data to their respective registers**
- **Device_cal() must be run to meet the ADC and oscillator specs in the datasheet**
  - **The Bootloader automatically calls Device_cal() such that no action is normally required by the user**
  - **If the bootloader is bypassed (e.g., during development) Device_cal() should be called by the application:**

```
#define Device_cal (void (*)(void))0x3D7C80
void main(void)
{
    (*Device_cal)();       // call Device_cal()
}
```

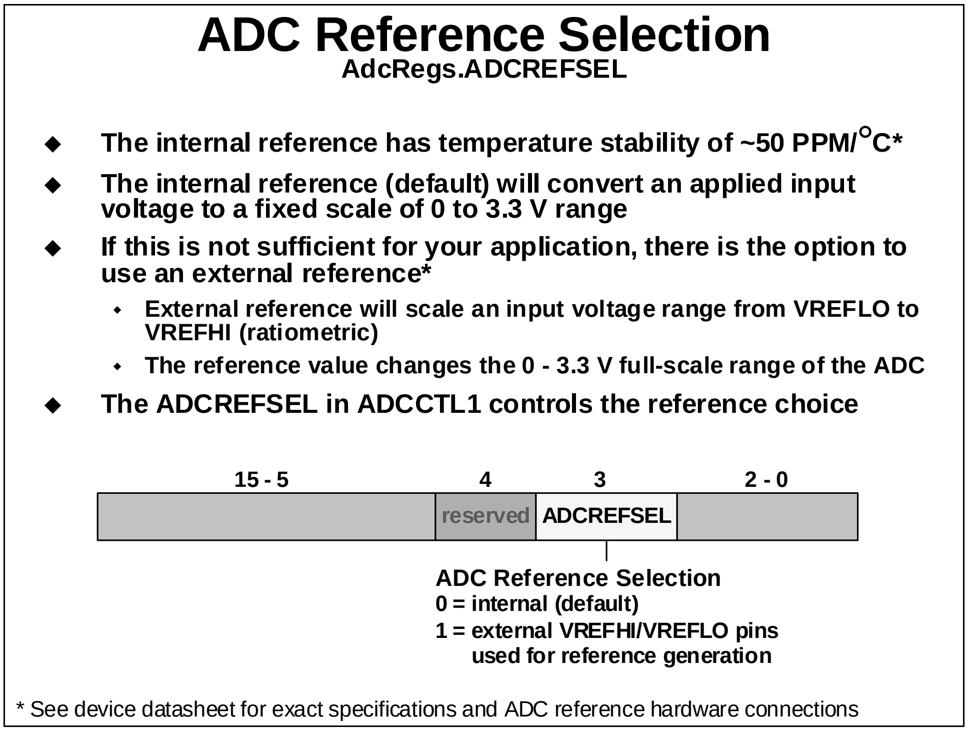
  - **A GEL function using CCS is also available as part of the Peripheral Register Header Files to accomplish this**



Manual ADC Calibration

- If the offset and gain errors in the datasheet* are unacceptable for your application, or you want to also compensate for board level errors (e.g., sensor or amplifier offset), you can manually calibrate
- Offset error
  - Compensated in *analog* with the ADCOFFTRIM register
  - No reduction in full-scale range
  - Configure input B5 to VREFLO, set ADCOFFTRIM to maximum offset error, and take a reading
  - Re-adjust ADCOFFTRIM to make result zero
- Gain error
  - Compensated in *software*
  - Some loss in full-scale range
  - Requires use of a second ADC input pin and an upper-range reference voltage on that pin; see “TMS320280x and TMS320F2801x ADC Calibration” appnote #SPRAAD8 for more information
- Tip: To minimize mux-to-mux variation effects, put your most critical signals on a single mux and use that mux for calibration inputs

* +/-15 LSB offset, +/-30 LSB gain. See device datasheet for exact specifications



ADC Reference Selection

AdcRegs.ADCREFSEL

- The internal reference has temperature stability of ~50 PPM/°C*
- The internal reference (default) will convert an applied input voltage to a fixed scale of 0 to 3.3 V range
- If this is not sufficient for your application, there is the option to use an external reference*
  - External reference will scale an input voltage range from VREFLO to VREFHI (ratiometric)
  - The reference value changes the 0 - 3.3 V full-scale range of the ADC
- The ADCREFSEL in ADCCTL1 controls the reference choice

15 - 5	4	3	2 - 0
	reserved	ADCREFSEL	

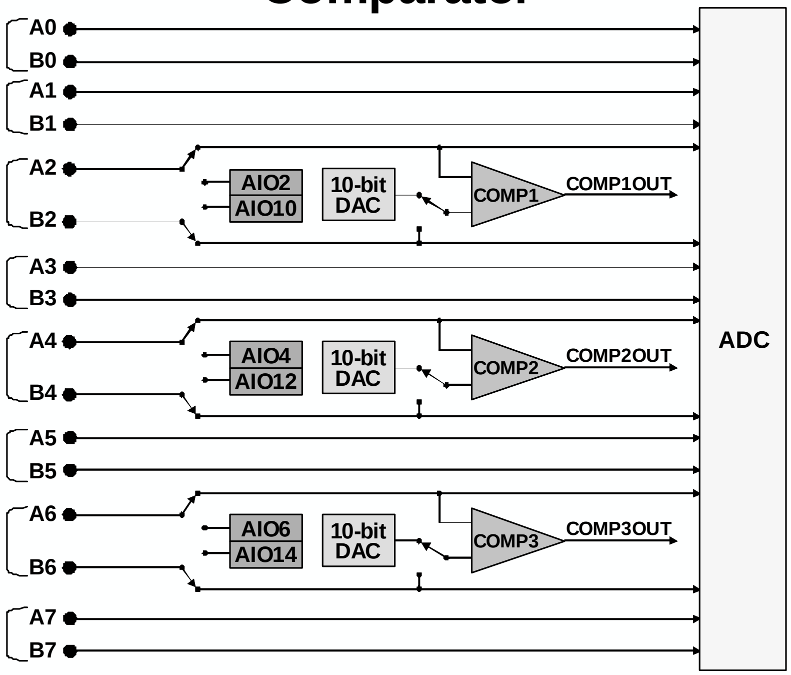
ADC Reference Selection
0 = internal (default)
1 = external VREFHI/VREFLO pins
used for reference generation

* See device datasheet for exact specifications and ADC reference hardware connections

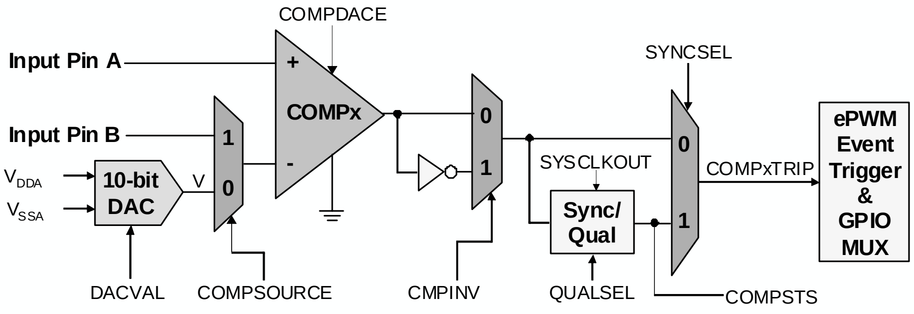
Comparator

Comparator Block Diagram

Comparator



Comparator Block Diagram



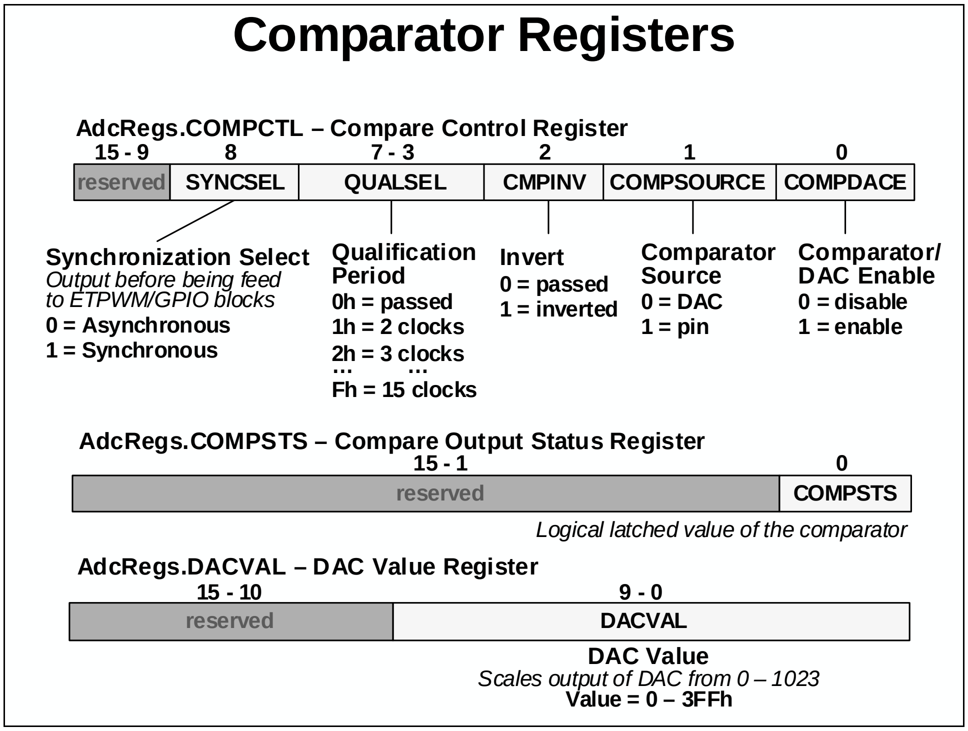
DAC Reference

$$V = \frac{DACVAL * (V_{DDA} - V_{SSA})}{1023}$$

Comparator Truth Table

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

Comparator Registers



Comparator Registers

AdcRegs.COMPCTL – Compare Control Register

15 - 9	8	7 - 3	2	1	0
reserved	SYNCSEL	QUALSEL	CMPINV	COMPSOURCE	COMPDACE

Synchronization Select
Output before being feed to ETPWM/GPIO blocks
0 = Asynchronous
1 = Synchronous

Qualification Period
0h = passed
1h = 2 clocks
2h = 3 clocks
… …
Fh = 15 clocks

Invert
0 = passed
1 = inverted

Comparator Source
0 = DAC
1 = pin

Comparator/ DAC Enable
0 = disable
1 = enable

AdcRegs.COMPSTS – Compare Output Status Register

15 - 1	0
reserved	COMPSTS

Logical latched value of the comparator

AdcRegs.DACVAL – DAC Value Register

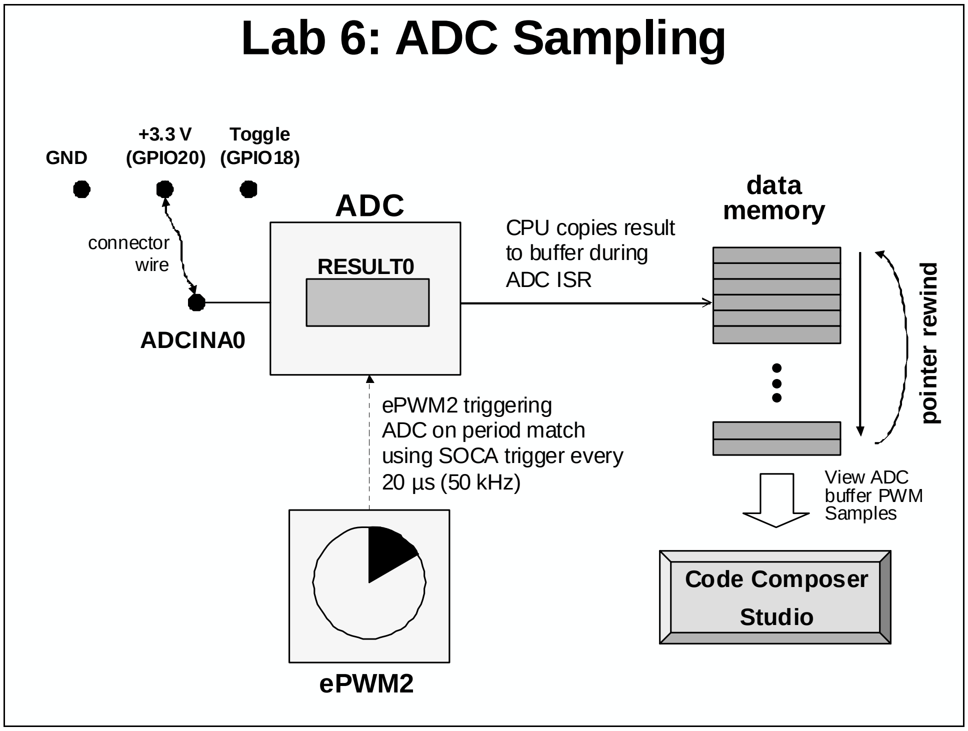
15 - 10	9 - 0
reserved	DACVAL

DAC Value
Scales output of DAC from 0 – 1023
Value = 0 – 3FFh

Lab 6: Analog-to-Digital Converter

➢ Objective

The objective of this lab is to become familiar with the programming and operation of the on-chip analog-to-digital converter. The MCU will be setup to sample a single ADC input channel at a prescribed sampling rate and store the conversion result in a memory buffer. This buffer will operate in a circular fashion, such that new conversion data continuously overwrites older results in the buffer.



Lab 6: ADC Sampling

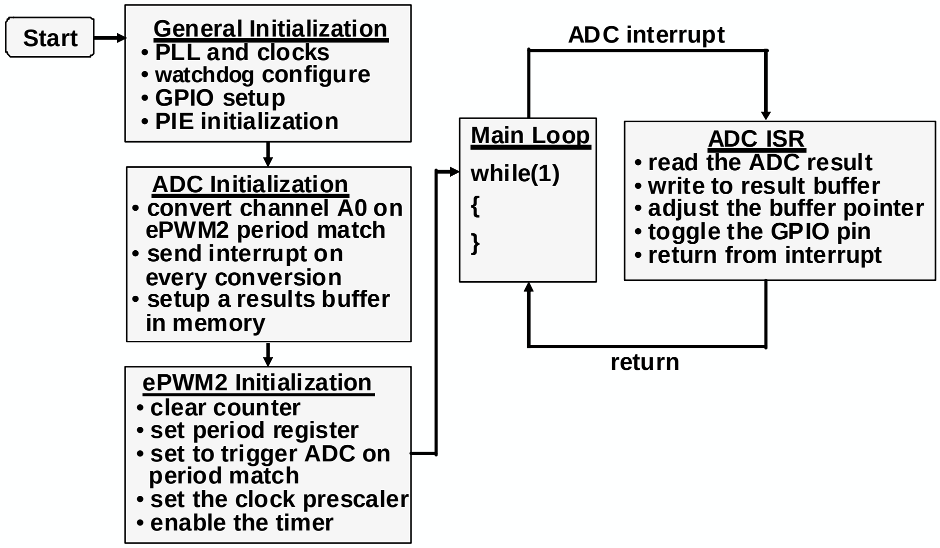
GND
+3.3 V (GPIO20)
Toggle (GPIO18)
connector wire
ADCINA0
ADC
RESULT0
CPU copies result to buffer during ADC ISR
data memory
pointer rewind
ePWM2 triggering ADC on period match using SOCA trigger every 20 µs (50 kHz)
View ADC buffer PWM Samples
Code Composer Studio
ePWM2

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

1. Using software
   a. SOCx bit (where x = 0 to 15) in the ADC SOC Force 1 Register (ADCSOCFRC1) causes a software initiated conversion
2. Automatically triggered on user selectable conditions
   a. CPU Timer 0/1/2 interrupt
   b. ePWMxSOCA / ePWMxSOCB (where x = 1 to 7)
      - ePWM underflow (CTR = 0)
      - ePWM period match (CTR = PRD)
      - ePWM underflow or period match (CTR = 0 or PRD)
      - ePWM compare match (CTRU/D = CMPA/B)
   c. ADC interrupt ADCINT1 or ADCINT2
      - triggers SOCx (where x = 0 to 15) selected by the ADC Interrupt Trigger SOC Select1/2 Register (ADCINTSOCSEL1/2)
3. Externally triggered using a pin
   a. ADCSOC pin (GPIO/XINT2_ADCSOC)

One or more of these methods may be applicable to a particular application. In this lab, we will be using the ADC for data acquisition. Therefore, one of the ePWMs (ePWM2) will be configured to automatically trigger the SOC A signal at the desired sampling rate (ePWM period match CTR = PRD SOC method 2b above). The ADC end-of-conversion interrupt will be used to prompt the CPU to copy the results of the ADC conversion into a results buffer in memory. This buffer pointer will be managed in a circular fashion, such that new conversion results will continuously overwrite older conversion results in the buffer. In order to generate an interesting input signal, the code also alternately toggles a GPIO pin (GPIO18) high and low in the ADC interrupt service routine. The ADC ISR will also toggle LED LD3 on the ControlCARD as a visual indication that the ISR is running. This pin will be connected to the ADC input pin, and sampled. After taking some data, Code Composer Studio will be used to plot the results. A flow chart of the code is shown in the following slide.

Lab 6: Code Flow Diagram



Start

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

ADC Initialization
- convert channel A0 on ePWM2 period match
- send interrupt on every conversion
- setup a results buffer in memory

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

Main Loop

while(1)
{
}

ADC interrupt

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

return

Notes

- Program performs conversion on ADC channel A0 (ADCINA0 pin)
- ADC conversion is set at a 50 kHz sampling rate
- ePWM2 is triggering the ADC on period match using SOCA trigger
- Data is continuously stored in a circular buffer
- GPIO18 pin is also toggled in the ADC ISR
- ADC ISR will also toggle the ControlCARD LED LD3 as a visual indication that it is running

➢ **Procedure**

Project File

1. A project named `Lab6.pjt` has been created for this lab. Open the project by clicking on `Project → Open`… and look in `C:\C28x\Labs\Lab6`. All Build Options have been configured the same as the previous lab. The files used in this lab are:

   `Adc.c`
   `CodeStartBranch.asm`
   `DefaultIsr_6.c`
   `DelayUs.asm`
   `DSP2803x_GlobalVariableDefs.c`
   `DSP2803x_Headers_nonBIOS.cmd`
   `EPwm_6.c`
   `Gpio.c`
   `Lab_5_6_7.cmd`
   `Main_6.c`
   `PieCtrl_5_6_7_8_9_10.c`
   `PieVect_5_6_7_8_9_10.c`
   `SysCtrl.c`
   `Watchdog.c`

Setup ADC Initialization and Enable Core/PIE Interrupts

2. In `Main_6.c` add code to call `InitAdc()` and `InitEPwm()` functions. The `InitEPwm()` function is used to configure ePWM2 to trigger the ADC at a 50 kHz rate. Details about the ePWM and control peripherals will be discussed in the next module.

3. Edit `Adc.c` to implement the ADC initialization as described above in the objective for the lab. Configure SOC0 for single sample mode, with an acquisition sample window of 7 cycles. Don't use the ADCINT to trigger a SOC0, and have all SOCs handled in round-robin mode. Enable ADCINT1 interrupt with EOC0 as the trigger for ADCINT1. Continuously generate an ADCINT1 pulse for each EOC.

4. Using the `"PIE Interrupt Assignment Table"` find the location for the ADC interrupt `"ADCINT1"` (high-priority) and fill in the following information:

   PIE group #:___________ # within group:__________

   This information will be used in the next step.

5. Modify the end of `Adc.c` to do the following:

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

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

Build and Load

7. Save all changes to the files and click the `"Build"` button.

8. Reset the CPU, select `EMU_BOOT_SARAM`, and then `"Go Main"`.

Run the Code

9. In `Main_6.c` place the cursor in the `"main loop"` section, right click on the mouse key and select `Run To Cursor`.

10. Open a memory window to view some of the contents of the ADC results buffer. The address label for the ADC results buffer is *AdcBuf*.

Note: ***Exercise care when connecting any wires, as the power to the USB Docking Station is on, and we do not want to damage the ControlCARD!***

11. Using a connector wire provided, connect the ADCINA0 (pin # ADC-A0) to "GND" (pin # GND) on the Docking Station. Then run the code again, and halt it after a few seconds. Verify that the ADC results buffer contains the expected value of 0x0000.

12. Adjust the connector wire to connect the ADCINA0 (pin # ADC-A0) to "+3.3V" (pin # GPIO-20) on the Docking Station. (Note: pin # GPIO-20 has been set to "1" in Gpio.c). Then run the code again, and halt it after a few seconds. Verify that the ADC results buffer contains the expected value of 0x0FFF.

13. Adjust the connector wire to connect the ADCINA0 (pin # ADC-A0) to GPIO18 (pin # GPIO-18) on the Docking Station. Then run the code again, and halt it after a few seconds. Examine the contents of the ADC results buffer (the contents should be alternating 0x0000 and 0x0FFF values). Are the contents what you expected?

14. Open and setup a graph to plot a 50-point window of the ADC results buffer.
    Click: `View` → `Graph` → `Time/Frequency…` and set the following values:

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

Select `OK` to save the graph options.

15. Recall that the code toggled the GPIO18 pin alternately high and low. (Also, the ADC ISR is toggling the LED LD3 on the ControlCARD as a visual indication that the ISR is running). If you had an oscilloscope available to display GPIO18, you would expect to see a square-wave. Why does Code Composer Studio plot resemble a triangle wave? What is the signal processing term for what is happening here?

16. Recall that the program toggled the GPIO18 pin at a 50 kHz rate. Therefore, a complete cycle (toggle high, then toggle low) occurs at half this rate, or 25 kHz. We therefore expect the period of the waveform to be 40 µs. Confirm this by measuring the period of the triangle wave using the graph (you may want to enlarge the graph window using the mouse). The measurement is best done with the mouse. The lower left-hand corner of the graph window will display the X and Y axis values. Subtract the X-axis values taken over a complete waveform period.

Using Real-time Emulation

Real-time emulation is a special emulation feature that offers two valuable capabilities:

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

B. It allows the user to halt the MCU and step through foreground tasks, while specified interrupts continue to get serviced in the background. This is useful when debugging portions of a realtime system (e.g., serial port receive code) while keeping critical parts of your system operating (e.g., commutation and current loops in motor control).

We will only be utilizing capability #1 above during the workshop. Capability #2 is a particularly advanced feature, and will not be covered in the workshop.

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

    `Debug → Real-time Mode`

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

18. The memory and graph windows displaying *AdcBuf* should still be open. The connector wire between ADCINA0 (pin # ADC-A0) and GPIO18 (pin # GPIO-18) should still be connected. In real-time mode, we would like to have our window continuously refresh. Click:

    `View → Real-time Refresh Options…`

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

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

19. Run the code and watch the windows update in real-time mode. ***Carefully*** remove and replace the connector wire from GPIO18. Are the values updating as expected?

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

21. So far, we have seen data flowing from the MCU to the debugger in realtime. In this step, we will flow data from the debugger to the MCU.
    - Open and inspect `DefaultIsr_6.c`. Notice that the global variable `DEBUG_TOGGLE` is used to control the toggling of the GPIO18 pin. This is the pin being read with the ADC.
    - Highlight `DEBUG_TOGGLE` with the mouse, right click and select “`Add to Watch Window`”. The global variable `DEBUG_TOGGLE` should now be in the watch window with a value of “1”.
    - Run the code in real-time mode and change the value to “0”. Are the results shown in the memory and graph window as expected? Change the value back to “1”. As you can see, we are modifying data memory contents while the processor is running in real-time (i.e., we are not halting the MCU nor interfering with its operation in any way)! When done, fully halt the CPU.

22. Code Composer Studio includes GEL (General Extension Language) functions which automate entering and exiting real-time mode. Four functions are available:
    - `Run_Realtime_with_Reset` *(reset CPU, enter real-time mode, run CPU)*
    - `Run_Realtime_with_Restart` *(restart CPU, enter real-time mode, run CPU)*
    - `Full_Halt` *(exit real-time mode, halt CPU)*
    - `Full_Halt_with_Reset` *(exit real-time mode, halt CPU, reset CPU)*

    These GEL functions can be executed by clicking:

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

    In the remaining lab exercises we will be using the above GEL functions to run and halt the code in real-time mode. If you would like, try repeating the previous step using the following GEL functions:

    `GEL → Realtime Emulation Control → Run_Realtime_with_Reset`

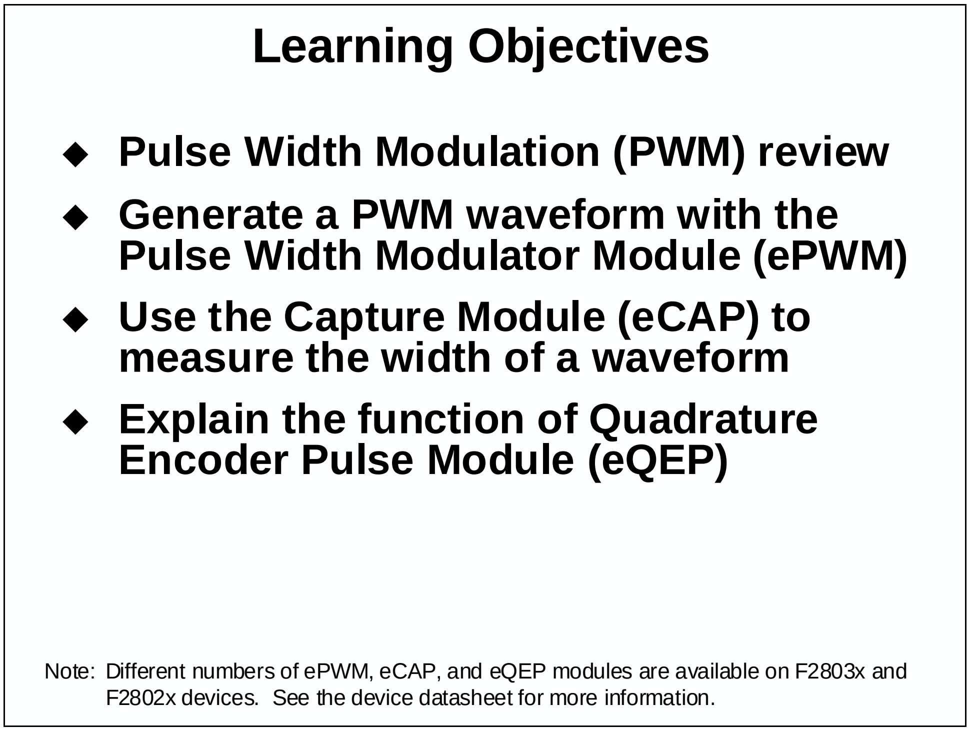
    `GEL → Realtime Emulation Control → Full_Halt`

End of Exercise

Introduction

This module explains how to generate PWM waveforms using the ePWM unit. Also, the eCAP unit, and eQEP unit will be discussed.

Learning Objectives



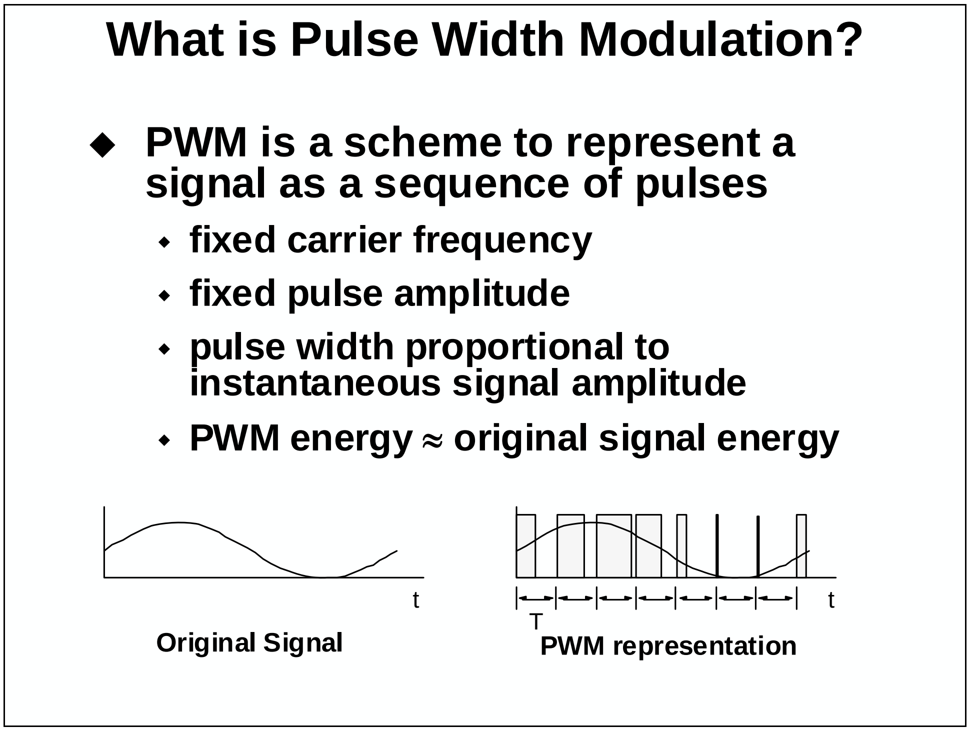
Learning Objectives

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

Note: Different numbers of ePWM, eCAP, and eQEP modules are available on F2803x and F2802x devices. See the device datasheet for more information.

Module Topics

PWM Review



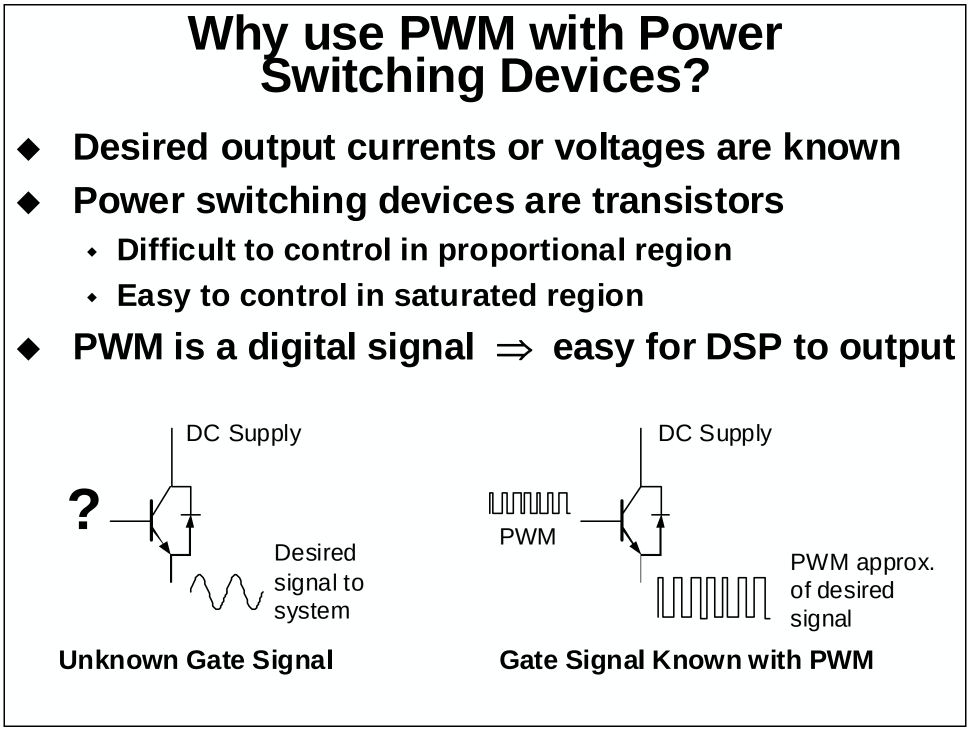
What is Pulse Width Modulation?

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

Original Signal

PWM representation

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



Why use PWM with Power Switching Devices?

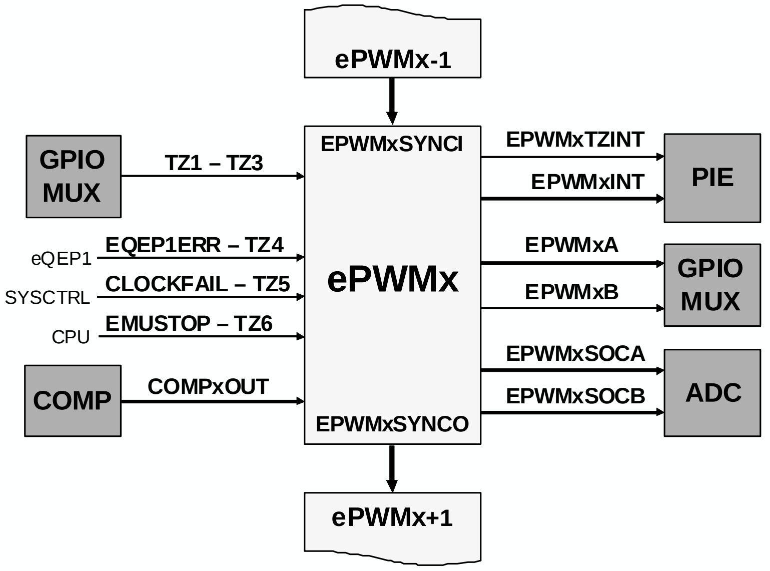
- Desired output currents or voltages are known
- Power switching devices are transistors
  - Difficult to control in proportional region
  - Easy to control in saturated region
- PWM is a digital signal ⇒ easy for DSP to output

Unknown Gate Signal

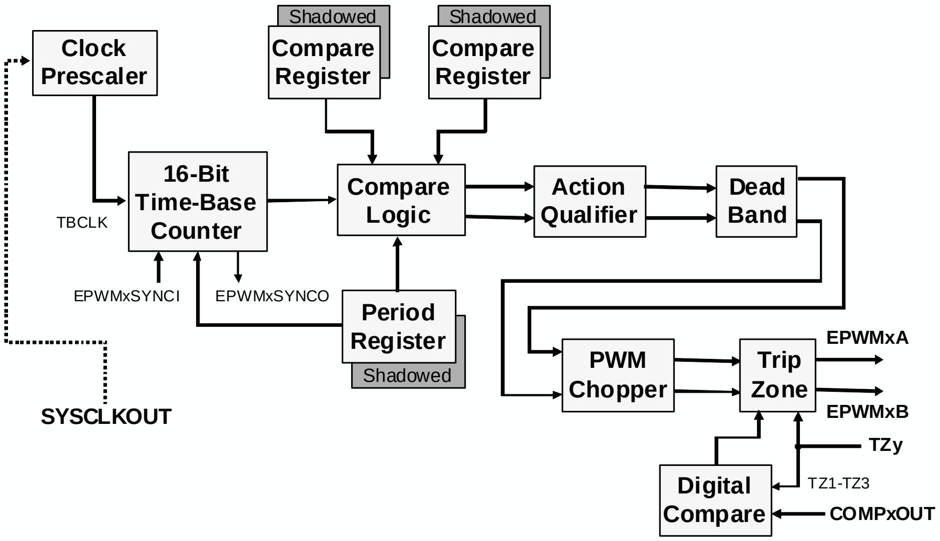
Gate Signal Known with PWM

ePWM

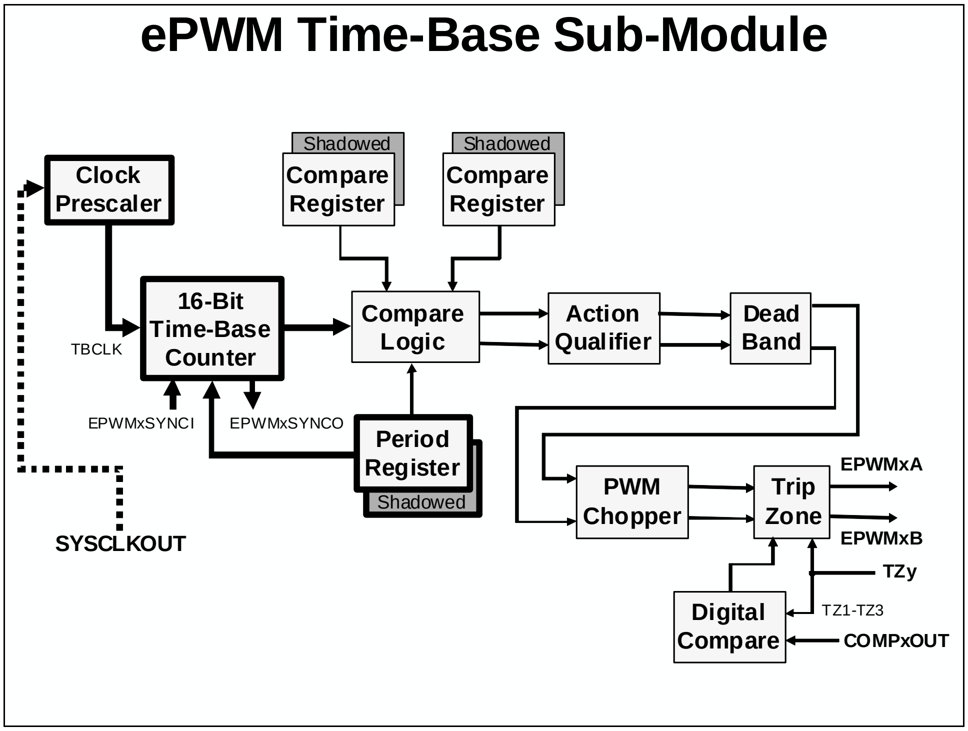
ePWM Module Signals and Connections



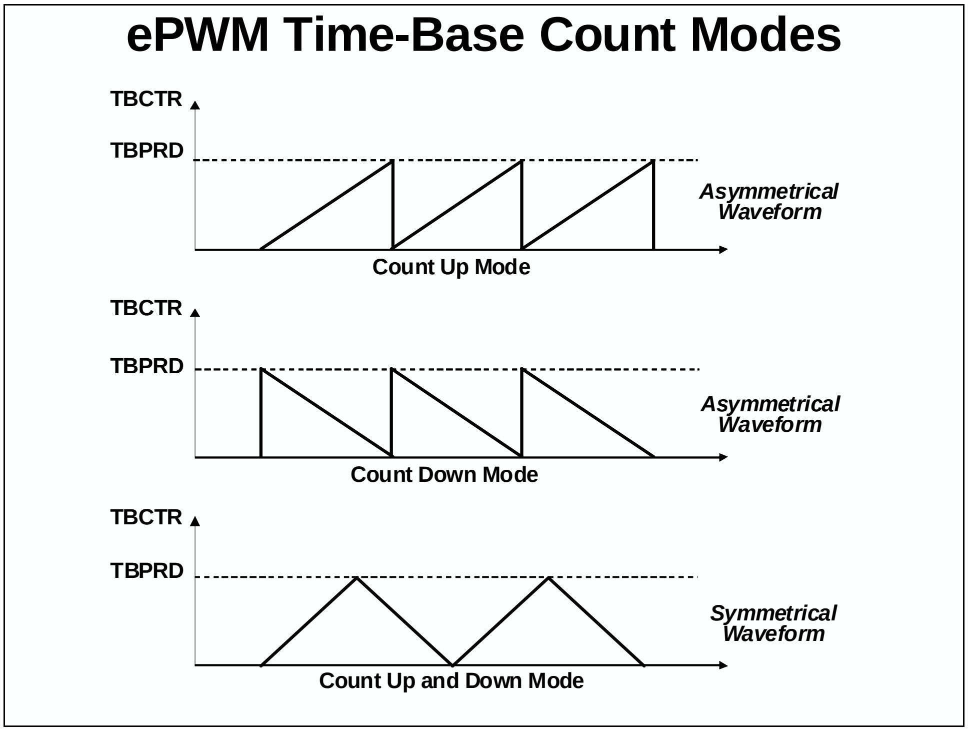
ePWM Block Diagram



ePWM Time-Base Sub-Module

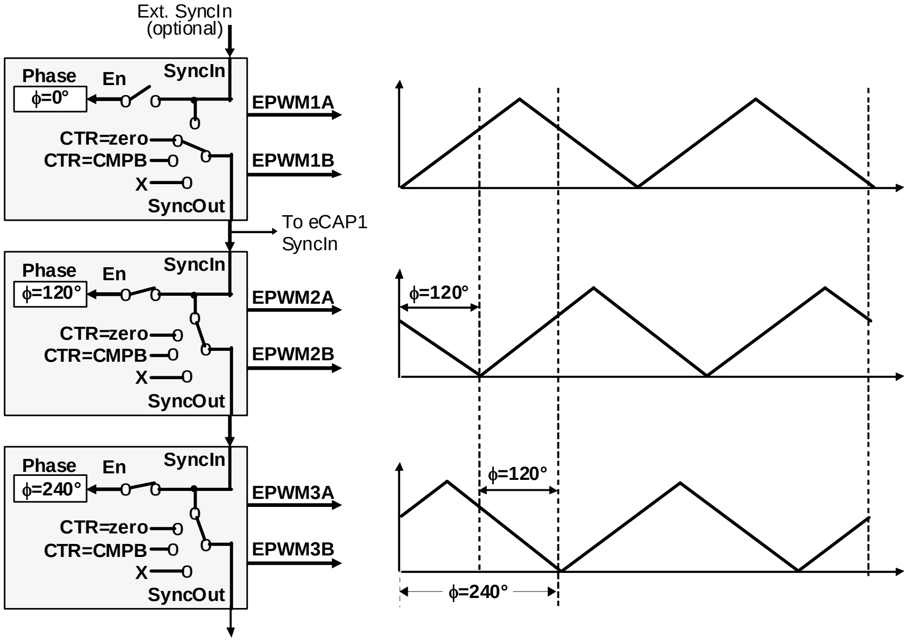


ePWM Time-Base Sub-Module



ePWM Time-Base Count Modes

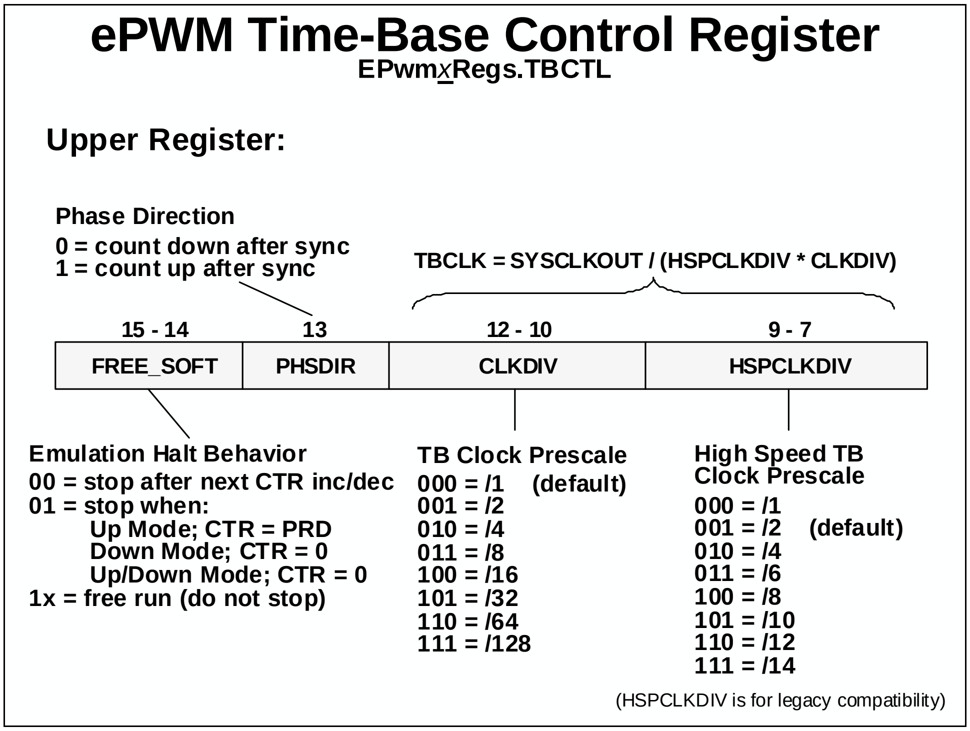
ePWM Phase Synchronization



ePWM Time-Base Sub-Module Registers

(lab file: EPwm.c)

Name	Description	Structure
TBCTL	Time-Base Control	EPwm<u>x</u>Regs.TBCTL.all =
TBSTS	Time-Base Status	EPwm<u>x</u>Regs.TBSTS.all =
TBPHS	Time-Base Phase	EPwm<u>x</u>Regs.TBPHS =
TBCTR	Time-Base Counter	EPwm<u>x</u>Regs.TBCTR =
TBPRD	Time-Base Period	EPwm<u>x</u>Regs.TBPRD =



ePWM Time-Base Control Register

EPwmxRegs.TBCTL

Upper Register:

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

TBCLK = SYSCLKOUT / (HSPCLKDIV * CLKDIV)

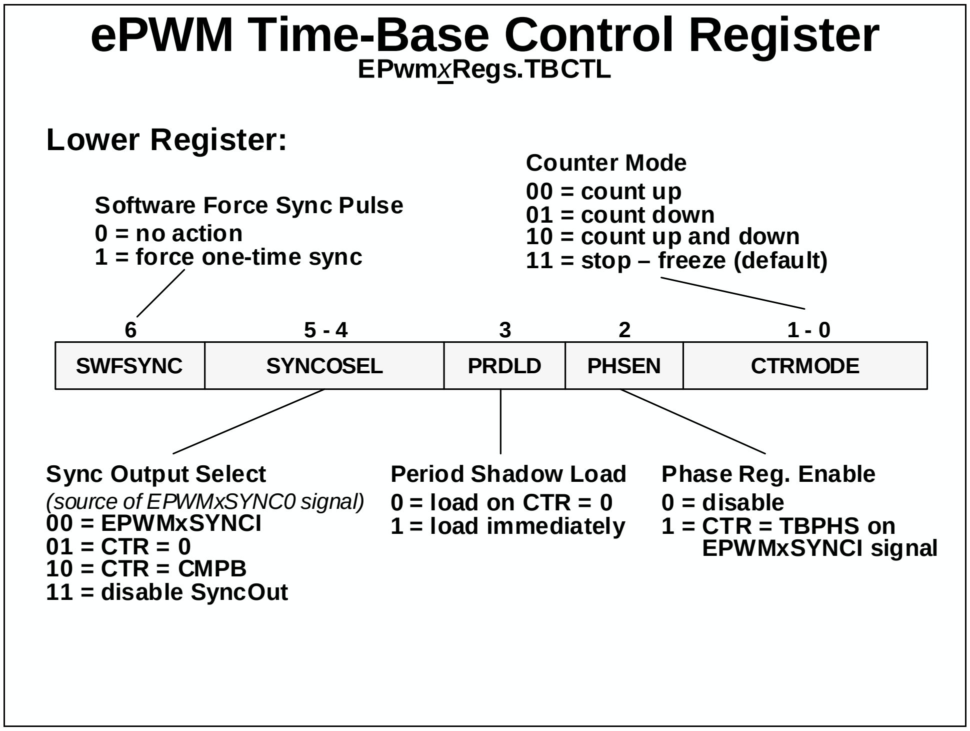
15 - 14	13	12 - 10	9 - 7
FREE_SOFT	PHSDIR	CLKDIV	HSPCLKDIV

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

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

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

(HSPCLKDIV is for legacy compatibility)



ePWM Time-Base Control Register

EPwmxRegs.TBCTL

Lower Register:

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

Software Force Sync Pulse
0 = no action
1 = force one-time sync

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

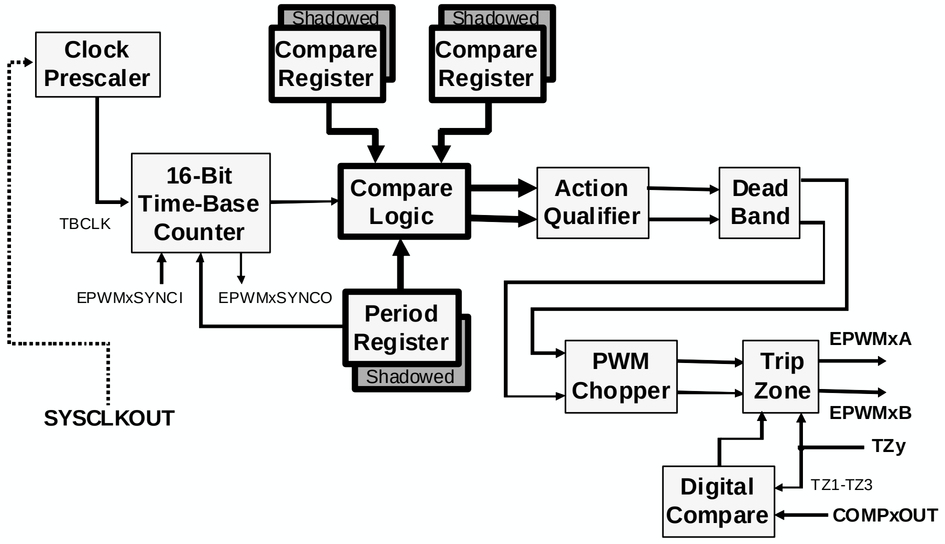
Sync Output Select
(source of EPWMxSYNC0 signal)
00 = EPWMxSYNCI
01 = CTR = 0
10 = CTR = CMPB
11 = disable SyncOut

Period Shadow Load
0 = load on CTR = 0
1 = load immediately

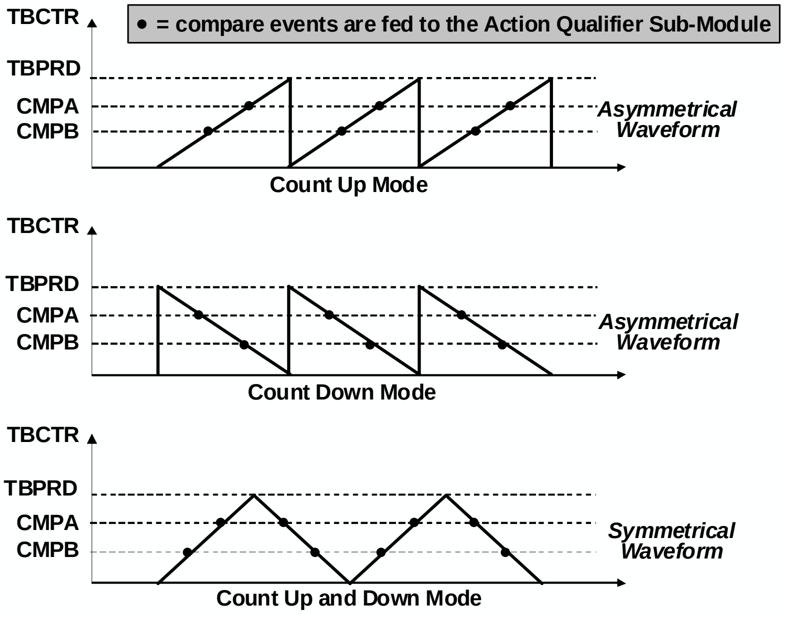
Phase Reg. Enable
0 = disable
1 = CTR = TBPHS on EPWMxSYNCI signal

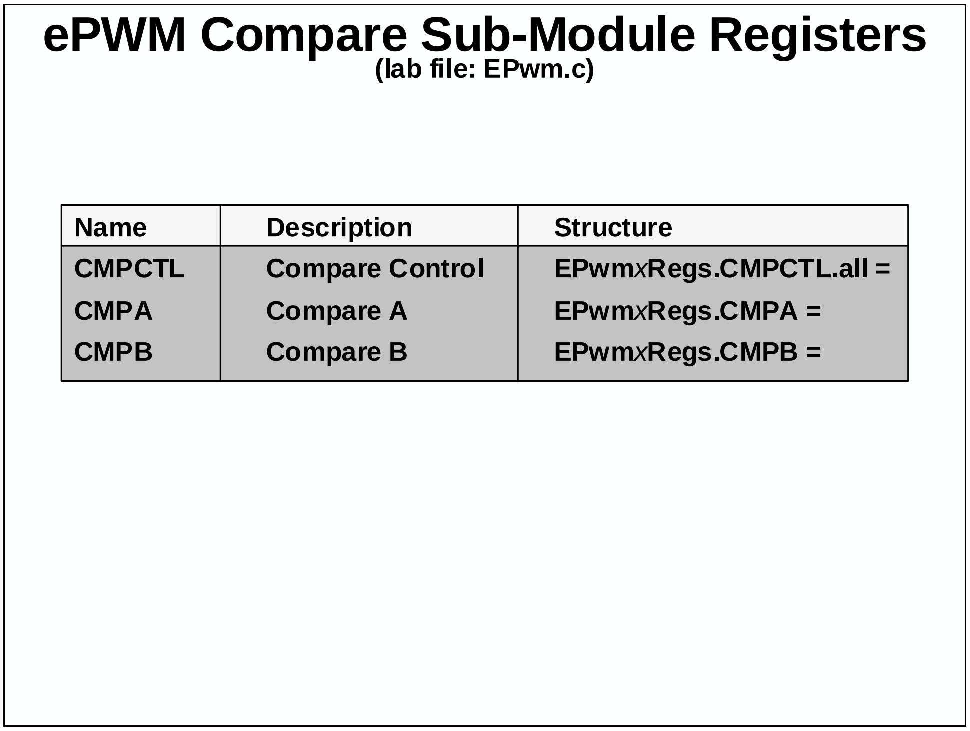
ePWM Compare Sub-Module

ePWM Compare Sub-Module



ePWM Compare Event Waveforms

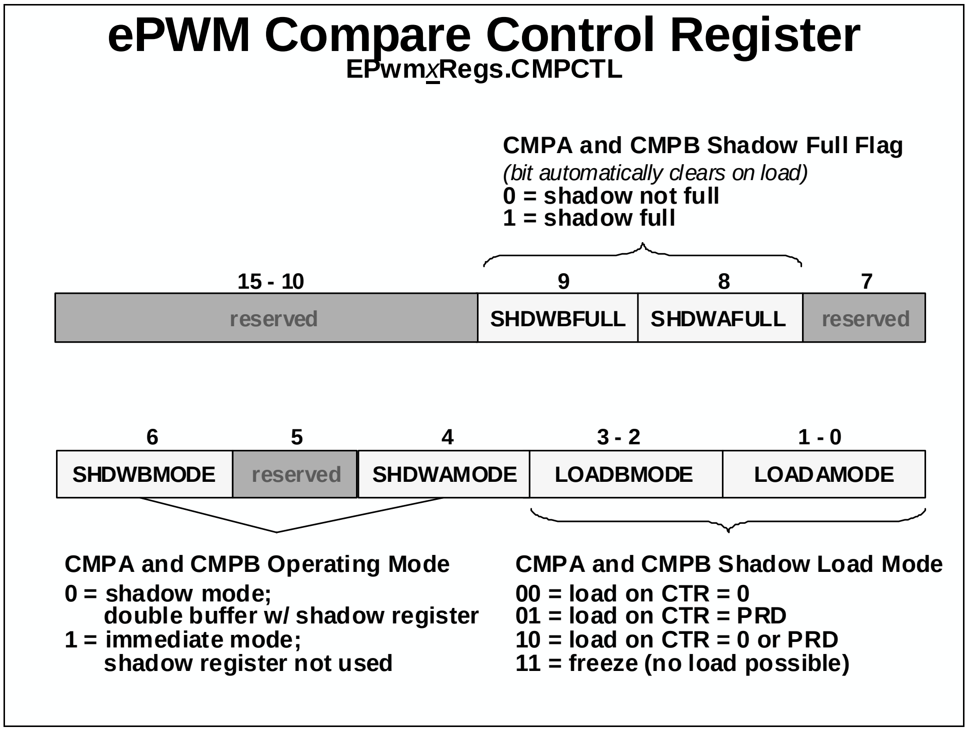




ePWM Compare Sub-Module Registers

(lab file: EPwm.c)

Name	Description	Structure
CMPCTL	Compare Control	EPwm*x*Regs.CMPCTL.all =
CMPA	Compare A	EPwm*x*Regs.CMPA =
CMPB	Compare B	EPwm*x*Regs.CMPB =



ePWM Compare Control Register

EPwm*x*Regs.CMPCTL

CMPA and CMPB Shadow Full Flag
(bit automatically clears on load)
0 = shadow not full
1 = shadow full

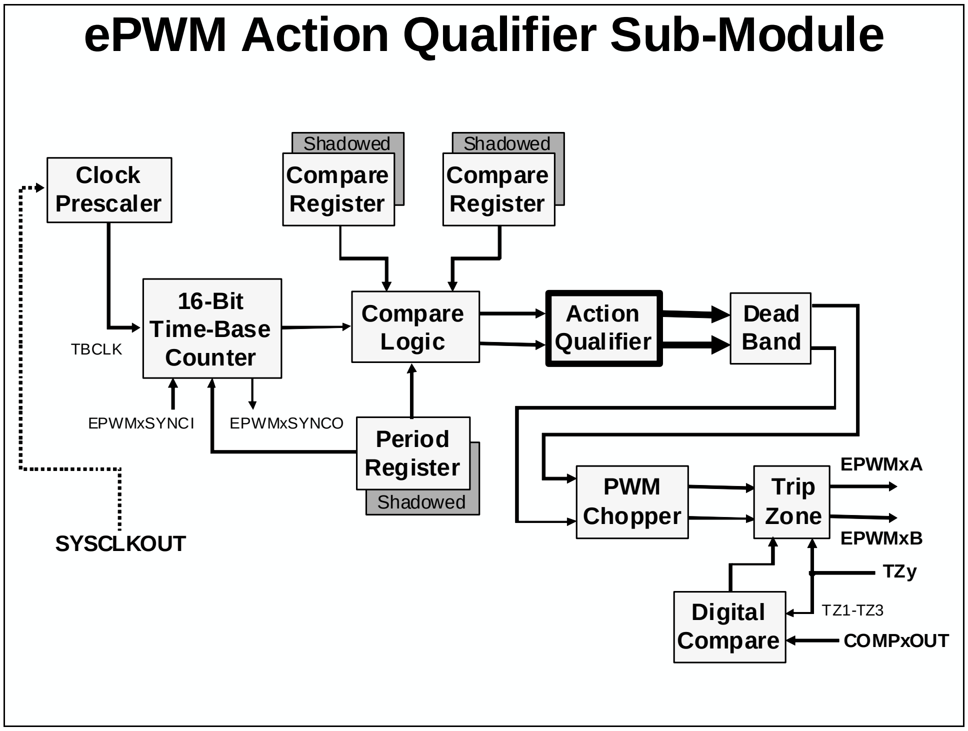
15 - 10	9	8	7
reserved	SHDWBFULL	SHDWAFULL	reserved

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

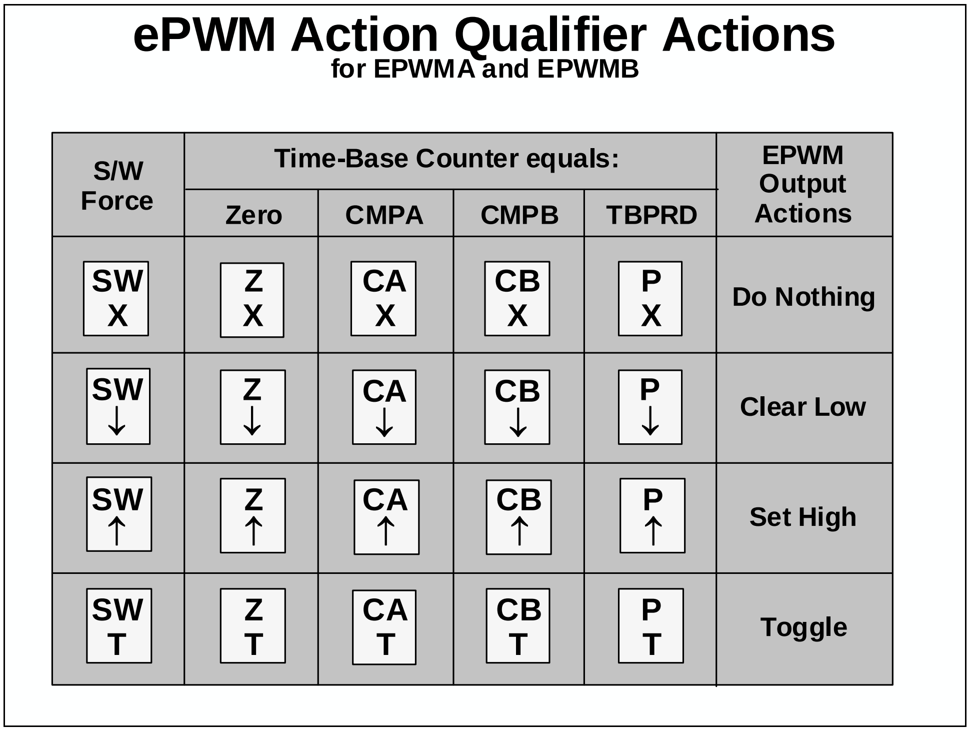
CMPA and CMPB Operating Mode
0 = shadow mode;
double buffer w/ shadow register
1 = immediate mode;
shadow register not used

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

ePWM Action Qualifier Sub-Module



ePWM Action Qualifier Sub-Module



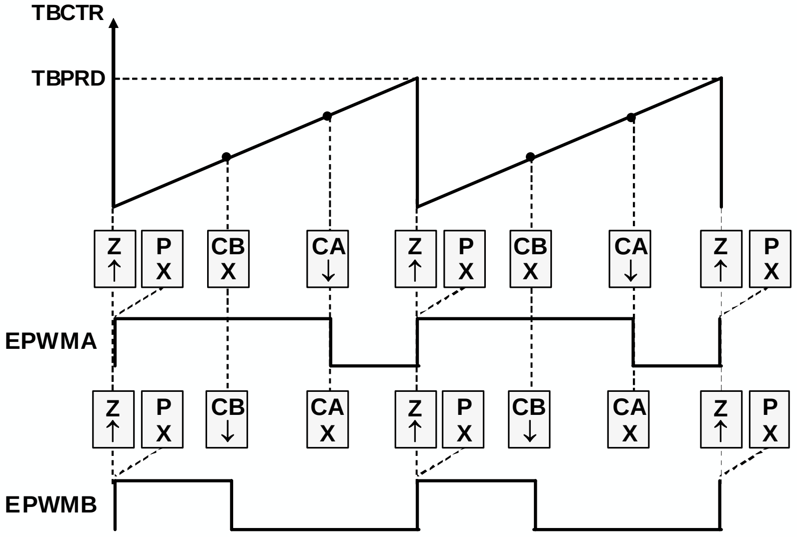
ePWM Action Qualifier Actions

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

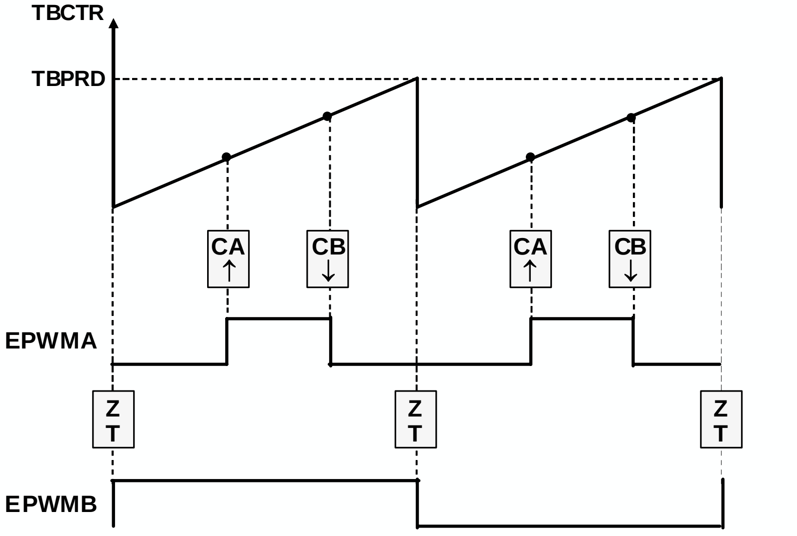
ePWM Count Up Asymmetric Waveform

with Independent Modulation on EPWMA / B



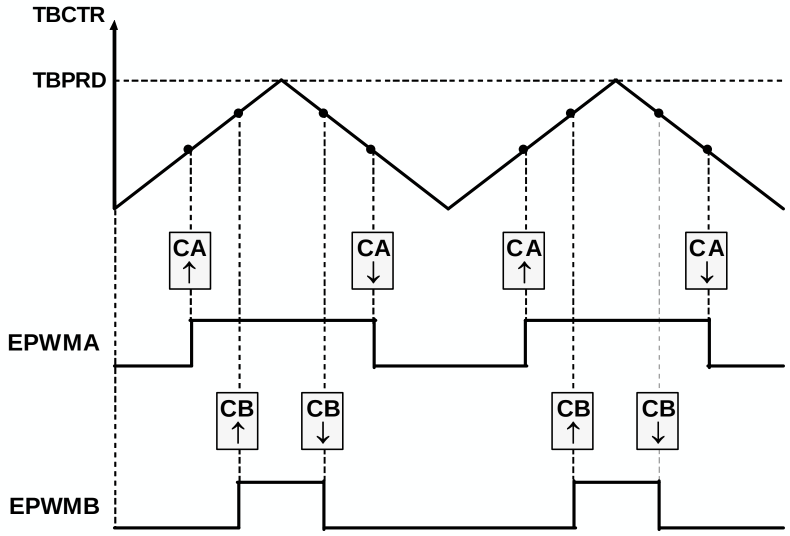
ePWM Count Up Asymmetric Waveform

with Independent Modulation on EPWMA



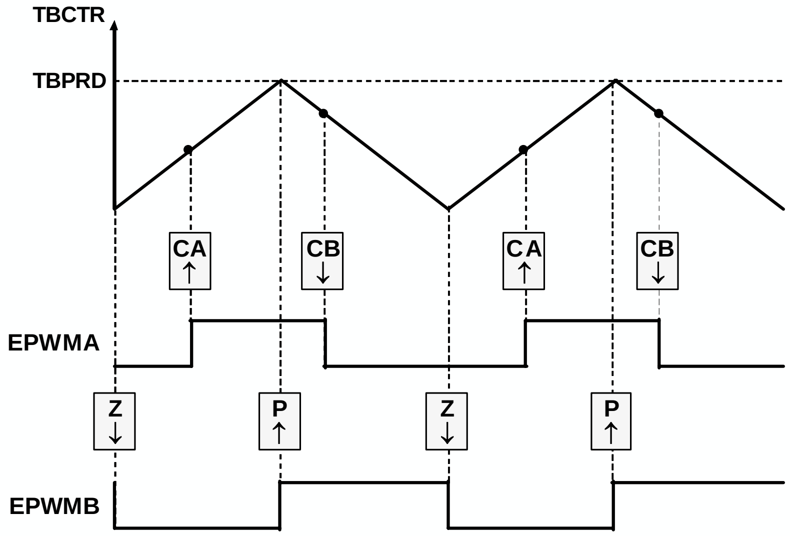
ePWM Count Up-Down Symmetric Waveform

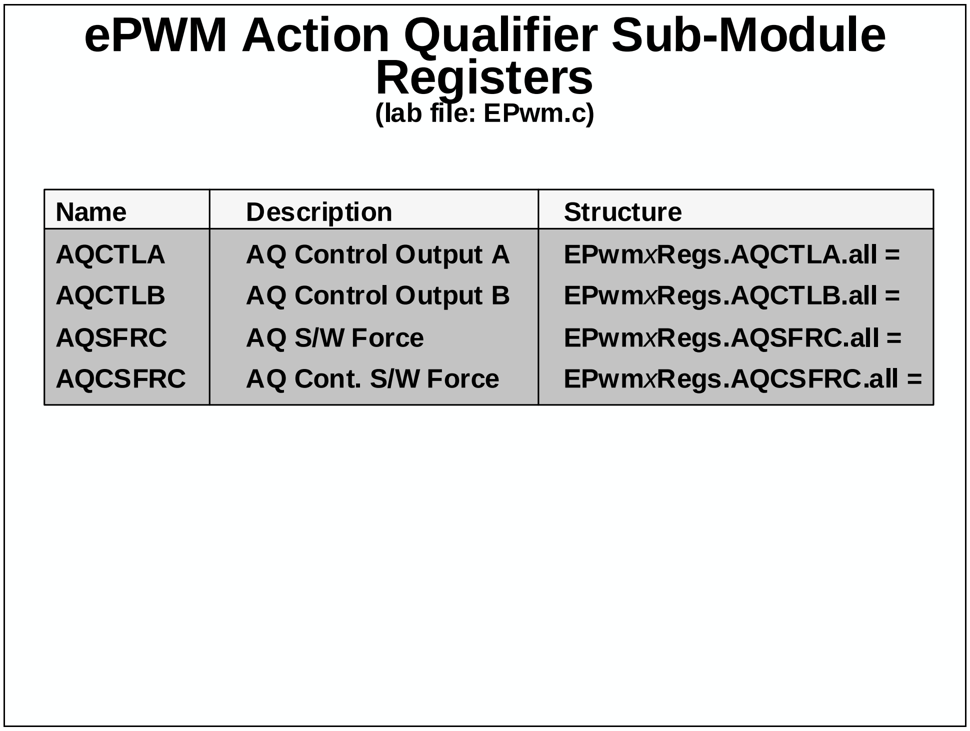
with Independent Modulation on EPWMA / B



ePWM Count Up-Down Symmetric Waveform

with Independent Modulation on EPWMA

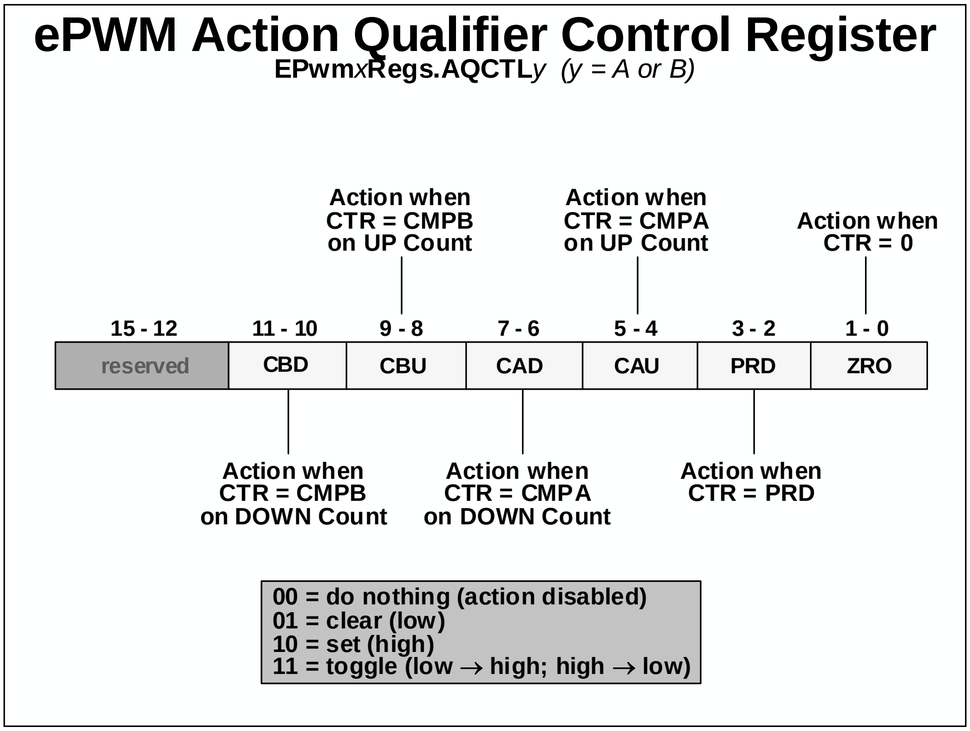




ePWM Action Qualifier Sub-Module Registers

(lab file: EPwm.c)

Name	Description	Structure
AQCTLA	AQ Control Output A	EPwm*x*Regs.AQCTLA.all =
AQCTLB	AQ Control Output B	EPwm*x*Regs.AQCTLB.all =
AQSFRC	AQ S/W Force	EPwm*x*Regs.AQSFRC.all =
AQCSFRC	AQ Cont. S/W Force	EPwm*x*Regs.AQCSFRC.all =



ePWM Action Qualifier Control Register

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

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

CBU: Action when CTR = CMPB on UP Count

CAU: Action when CTR = CMPA on UP Count

ZRO: Action when CTR = 0

CBD: Action when CTR = CMPB on DOWN Count

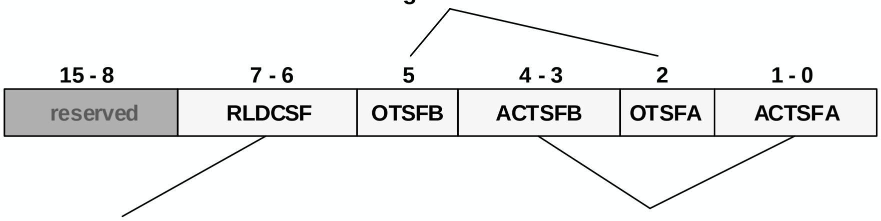
CAD: Action when CTR = CMPA on DOWN Count

PRD: Action when CTR = PRD

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

ePWM Action Qualifier S/W Force Register

EPwm*x*Regs.AQSFRC



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

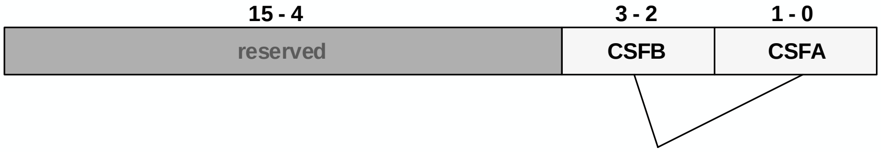
15 - 8	7 - 6	5	4 - 3	2	1 - 0
reserved	RLDCSF	OTSFB	ACTSFB	OTSFA	ACTSFA

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

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

ePWM Action Qualifier Continuous S/W Force Register

EPwm*x*Regs.AQCSFRC



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

Continuous S/W Force on Output B / A
00 = forcing disabled
01 = force continuous low on output
10 = force continuous high on output
11 = forcing disabled

Asymmetric and Symmetric Waveform Generation using the ePWM

PWM switching frequency:

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

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

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

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

PWM resolution:

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

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

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

PWM duty cycle:

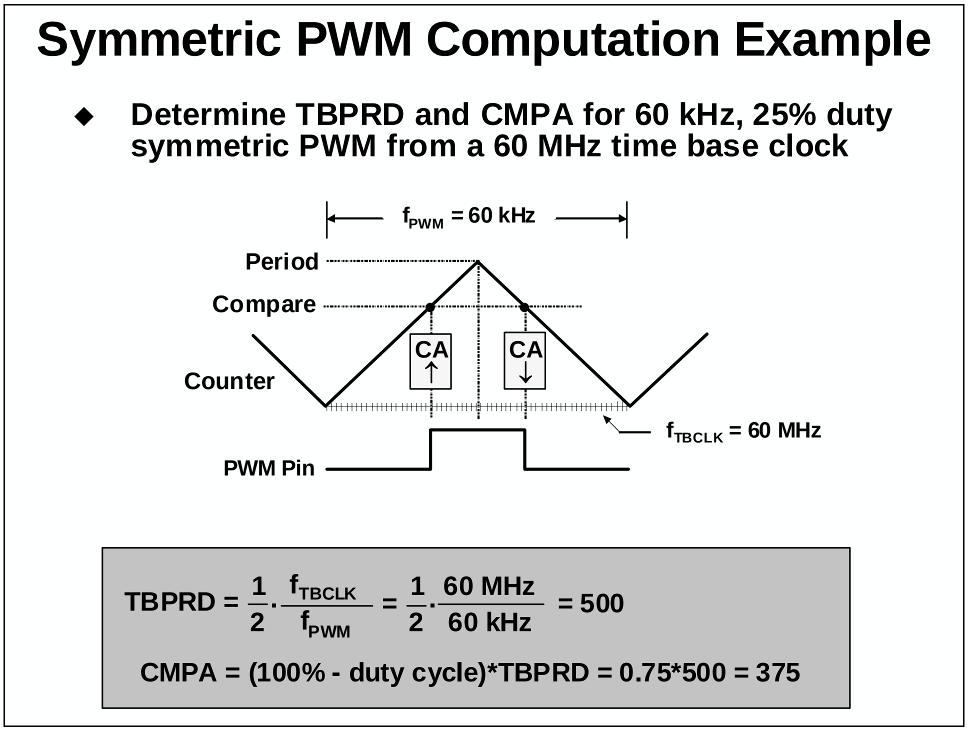
Duty cycle calculations are simple provided one remembers that the PWM signal is initially inactive during any particular timer period, and becomes active after the (first) compare match occurs. The timer compare register should be loaded with the value as follows:

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

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

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

PWM Computation Example

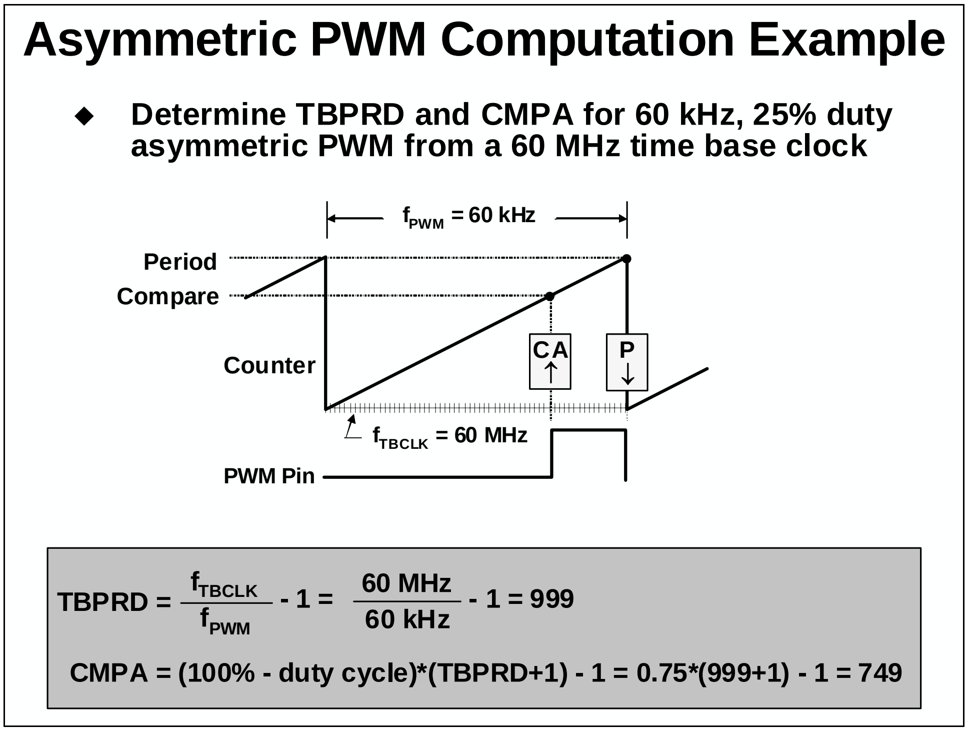


Symmetric PWM Computation Example

- Determine TBPRD and CMPA for 60 kHz, 25% duty symmetric PWM from a 60 MHz time base clock

$$\text{TBPRD} = \frac{1}{2}\cdot\frac{f_{TBCLK}}{f_{PWM}} = \frac{1}{2}\cdot\frac{60\text{ MHz}}{60\text{ kHz}} = 500$$

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



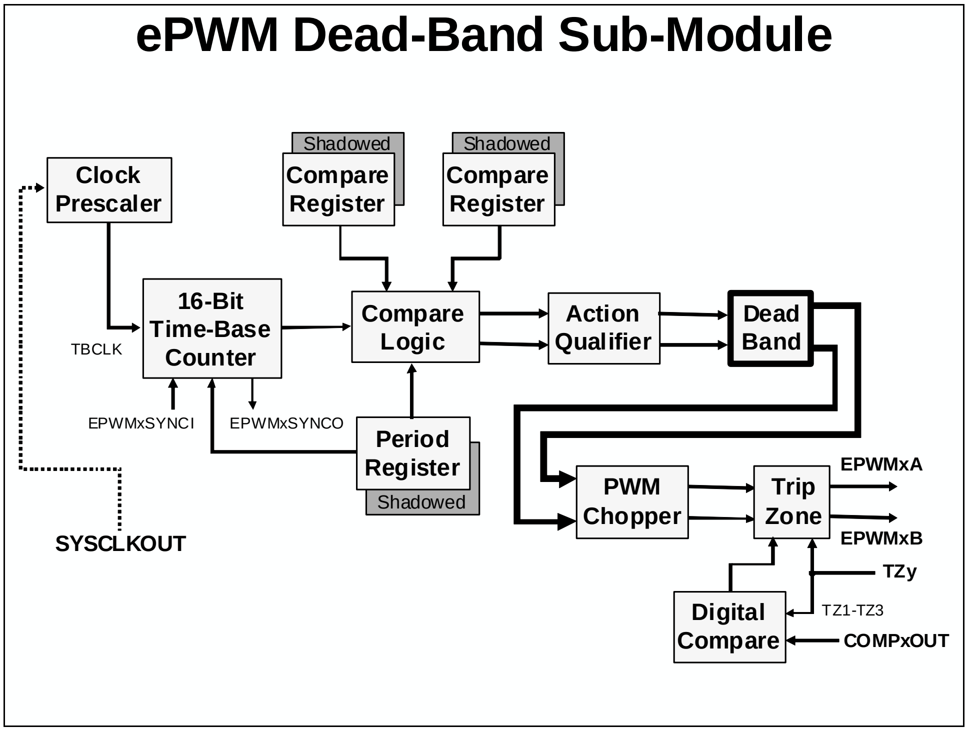
Asymmetric PWM Computation Example

- Determine TBPRD and CMPA for 60 kHz, 25% duty asymmetric PWM from a 60 MHz time base clock

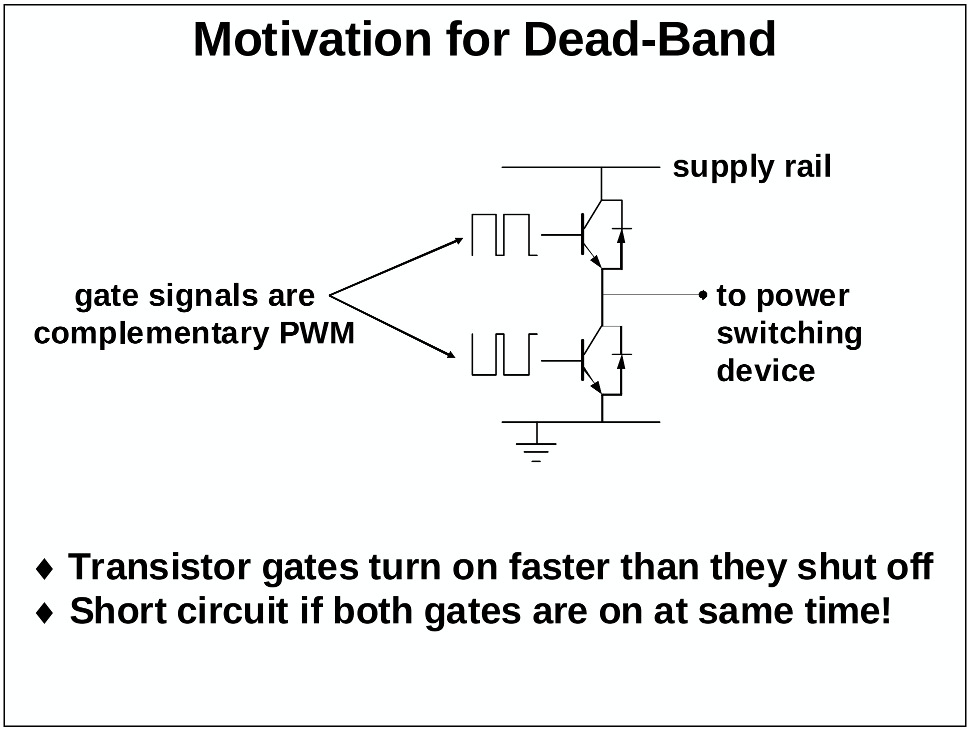
$$\text{TBPRD} = \frac{f_{TBCLK}}{f_{PWM}} - 1 = \frac{60\text{ MHz}}{60\text{ kHz}} - 1 = 999$$

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

ePWM Dead-Band Sub-Module



ePWM Dead-Band Sub-Module

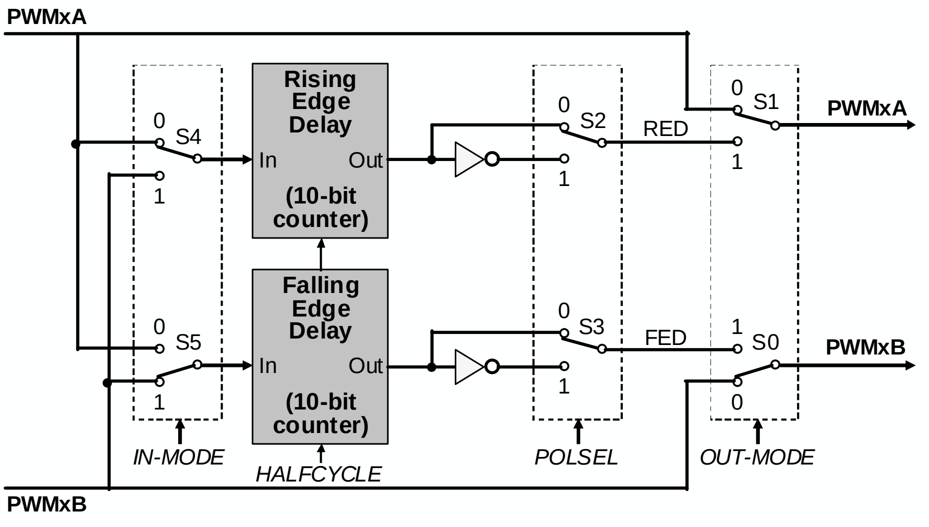


Motivation for Dead-Band

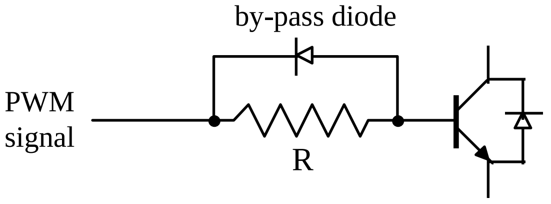
- ♦ Transistor gates turn on faster than they shut off
- ♦ Short circuit if both gates are on at same time!

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

ePWM Dead-Band Block Diagram



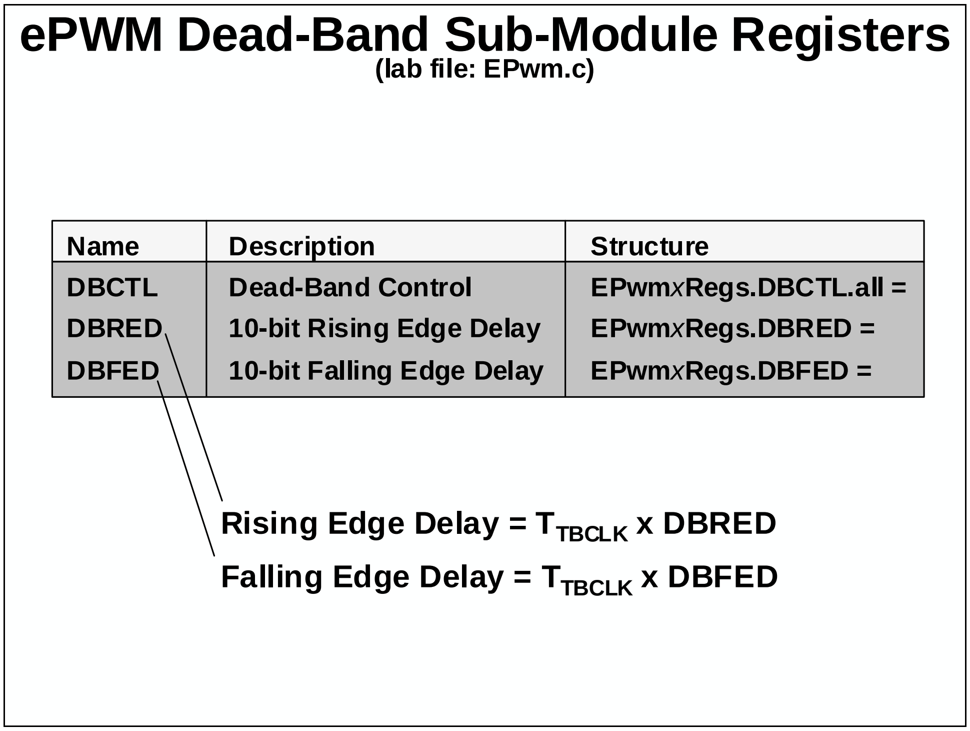
Two basic approaches exist for controlling shoot-through: modify the transistors, or modify the PWM gate signals controlling the transistors. In the first case, the opening time of the transistor gate must be increased so that it (slightly) exceeds the closing time. One way to accomplish this is by adding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



Shoot-through control via power circuit modification

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

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



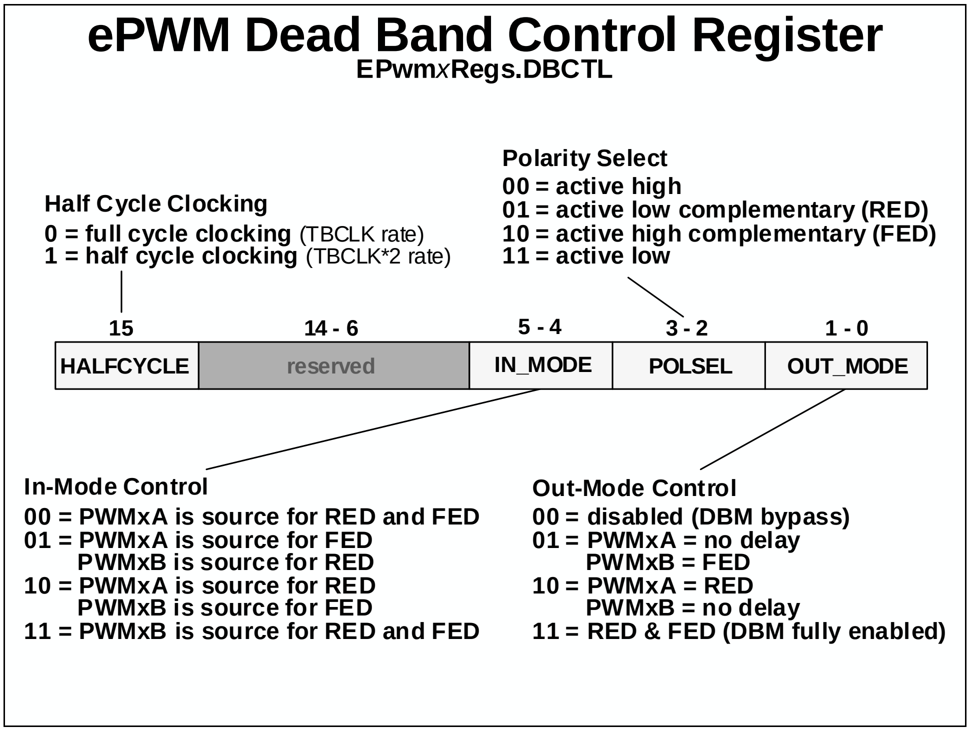
ePWM Dead-Band Sub-Module Registers

(lab file: EPwm.c)

Name	Description	Structure
DBCTL	Dead-Band Control	EPwm*x*Regs.DBCTL.all =
DBRED	10-bit Rising Edge Delay	EPwm*x*Regs.DBRED =
DBFED	10-bit Falling Edge Delay	EPwm*x*Regs.DBFED =

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

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



ePWM Dead Band Control Register

EPwm*x*Regs.DBCTL

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

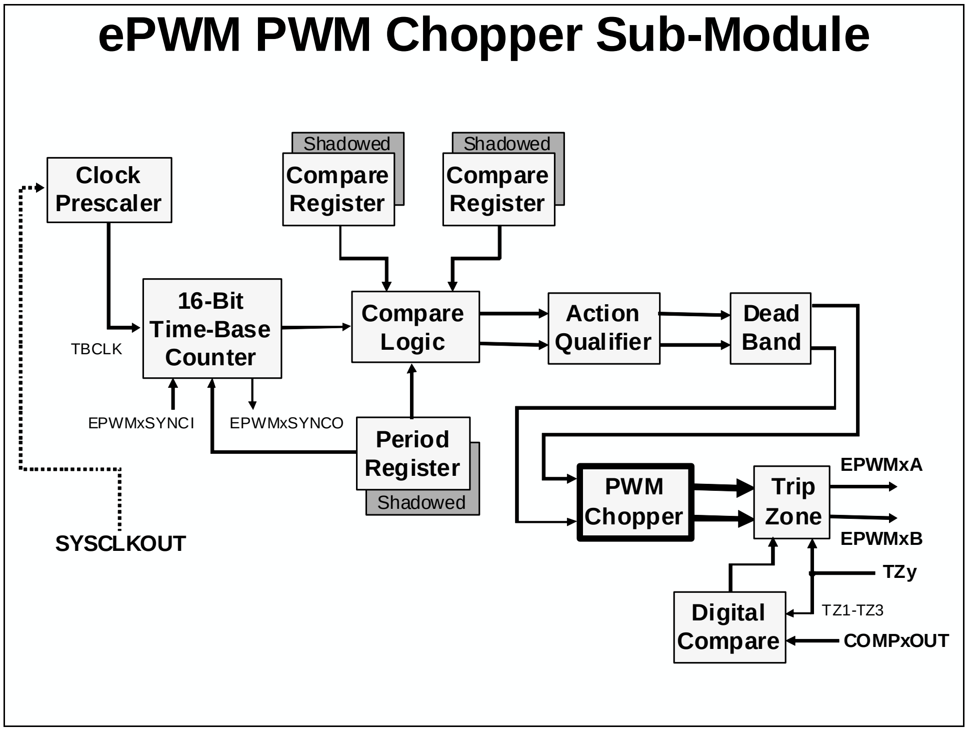
Polarity Select
00 = active high
01 = active low complementary (RED)
10 = active high complementary (FED)
11 = active low

15	14 - 6	5 - 4	3 - 2	1 - 0
HALFCYCLE	reserved	IN_MODE	POLSEL	OUT_MODE

In-Mode Control
00 = PWMxA is source for RED and FED
01 = PWMxA is source for FED
PWMxB is source for RED
10 = PWMxA is source for RED
PWMxB is source for FED
11 = PWMxB is source for RED and FED

Out-Mode Control
00 = disabled (DBM bypass)
01 = PWMxA = no delay
PWMxB = FED
10 = PWMxA = RED
PWMxB = no delay
11 = RED & FED (DBM fully enabled)

ePWM PWM Chopper Sub-Module

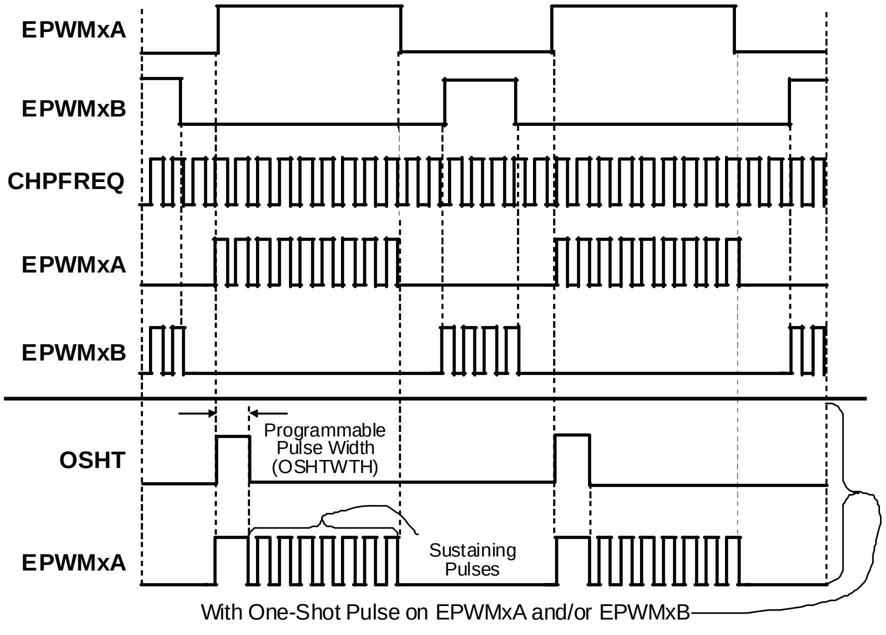


ePWM PWM Chopper Sub-Module

Purpose of the PWM Chopper

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

ePWM Chopper Waveform


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

ePWM Chopper Sub-Module Registers

(lab file: EPwm.c)

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

ePWM Chopper Control Register

EPwm*x*Regs.PCCTL

Chopper Clk Duty Cycle

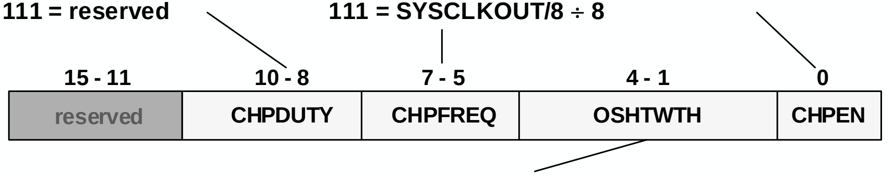
000 = 1/8 (12.5%)
001 = 2/8 (25.0%)
010 = 3/8 (37.5%)
011 = 4/8 (50.0%)
100 = 5/8 (62.5%)
101 = 6/8 (75.0%)
110 = 7/8 (87.5%)
111 = reserved

Chopper Clk Freq.

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

Chopper Enable

0 = disable (bypass)
1 = enable

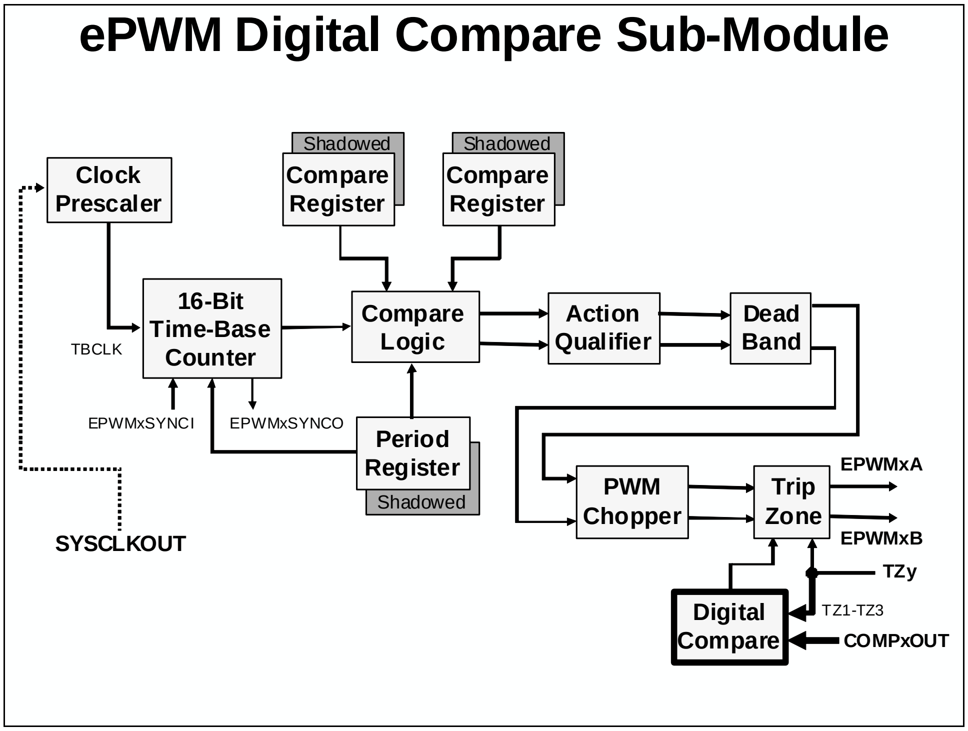


15 - 11	10 - 8	7 - 5	4 - 1	0
reserved	CHPDUTY	CHPFREQ	OSHTWTH	CHPEN

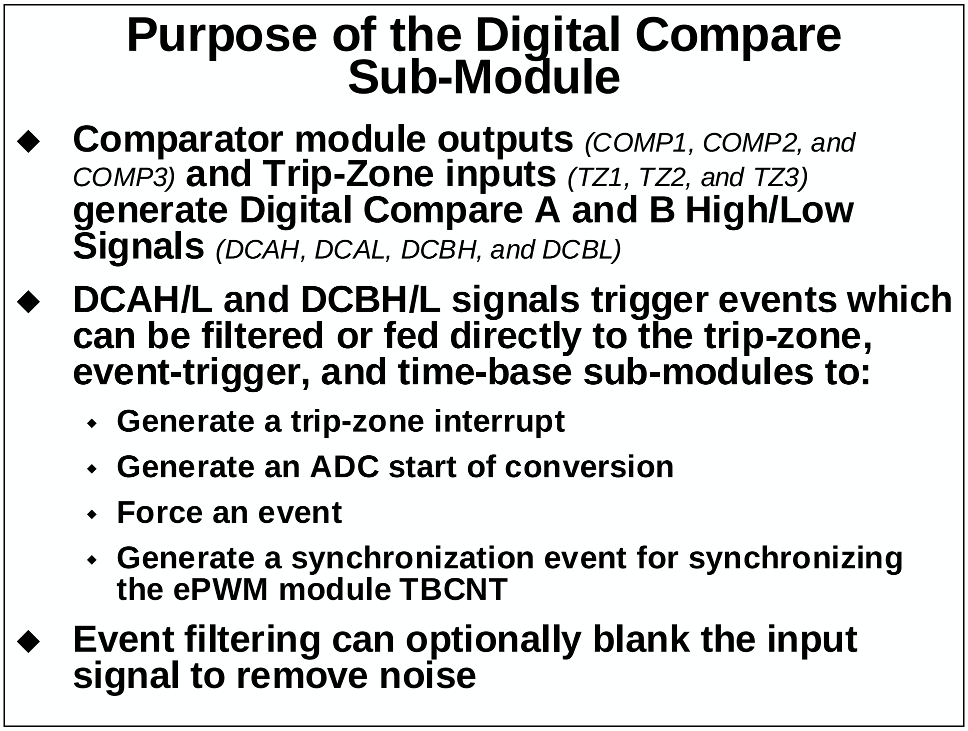
One-Shot Pulse Width

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

ePWM Digital Compare Sub-Module



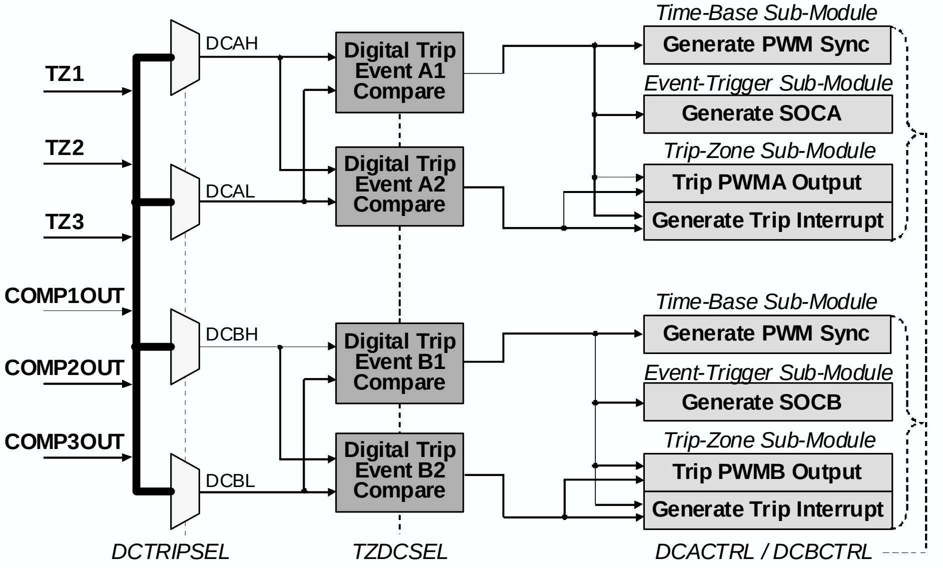
ePWM Digital Compare Sub-Module



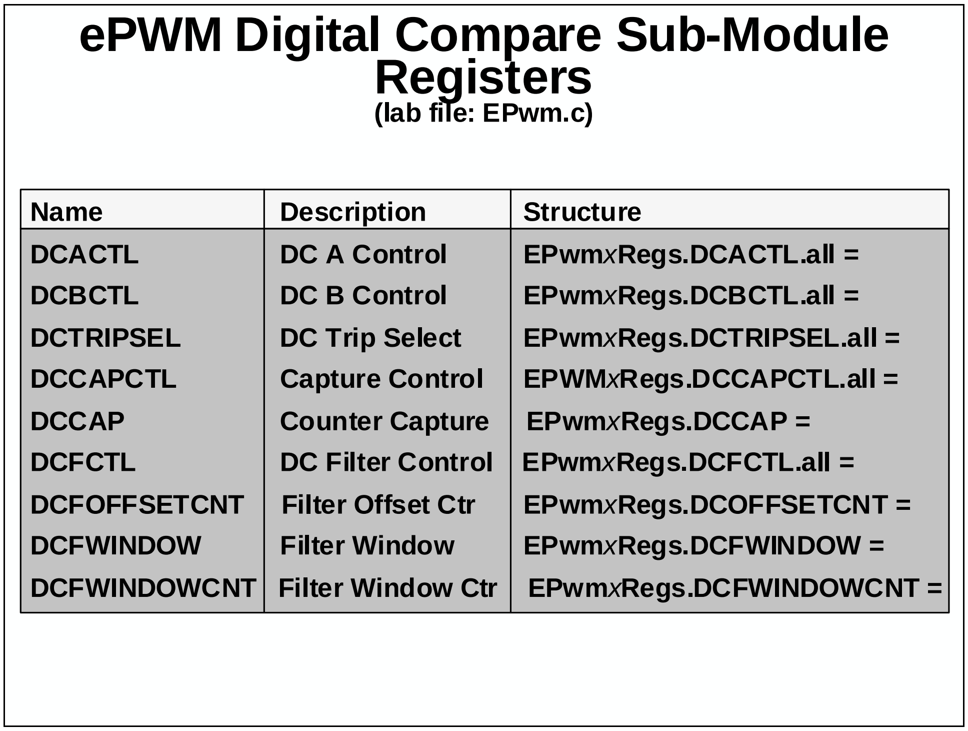
Purpose of the Digital Compare Sub-Module

- **Comparator module outputs** *(COMP1, COMP2, and COMP3)* **and Trip-Zone inputs** *(TZ1, TZ2, and TZ3)* **generate Digital Compare A and B High/Low Signals** *(DCAH, DCAL, DCBH, and DCBL)*
- **DCAH/L and DCBH/L signals trigger events which can be filtered or fed directly to the trip-zone, event-trigger, and time-base sub-modules to:**
  - **Generate a trip-zone interrupt**
  - **Generate an ADC start of conversion**
  - **Force an event**
  - **Generate a synchronization event for synchronizing the ePWM module TBCNT**
- **Event filtering can optionally blank the input signal to remove noise**

Digital Compare Sub-Module Signals



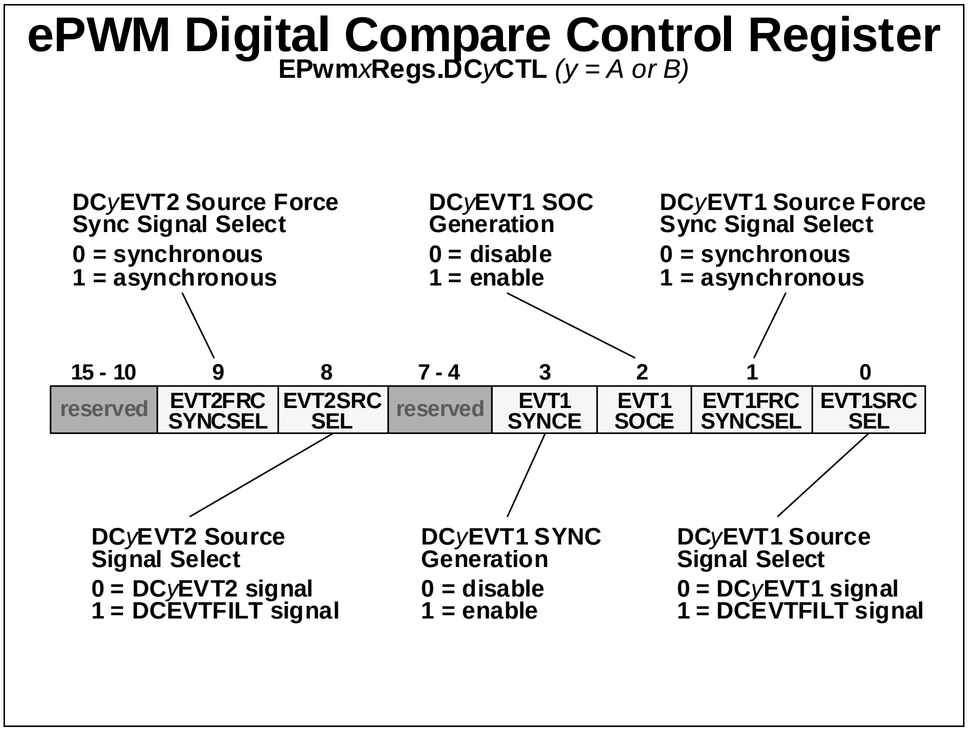
The Digital Compare sub-module compares signals external to the ePWM module to directly generate events which are then feed to the Event-Trigger, Trip-Zone, and Time-Base sub-modules



ePWM Digital Compare Sub-Module Registers

(lab file: EPwm.c)

Name	Description	Structure
DCACTL	DC A Control	EPwm*x*Regs.DCACTL.all =
DCBCTL	DC B Control	EPwm*x*Regs.DCBCTL.all =
DCTRIPSEL	DC Trip Select	EPwm*x*Regs.DCTRIPSEL.all =
DCCAPCTL	Capture Control	EPWM*x*Regs.DCCAPCTL.all =
DCCAP	Counter Capture	EPwm*x*Regs.DCCAP =
DCFCTL	DC Filter Control	EPwm*x*Regs.DCFCTL.all =
DCFOFFSETCNT	Filter Offset Ctr	EPwm*x*Regs.DCOFFSETCNT =
DCFWINDOW	Filter Window	EPwm*x*Regs.DCFWINDOW =
DCFWINDOWCNT	Filter Window Ctr	EPwm*x*Regs.DCFWINDOWCNT =



ePWM Digital Compare Control Register

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

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

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

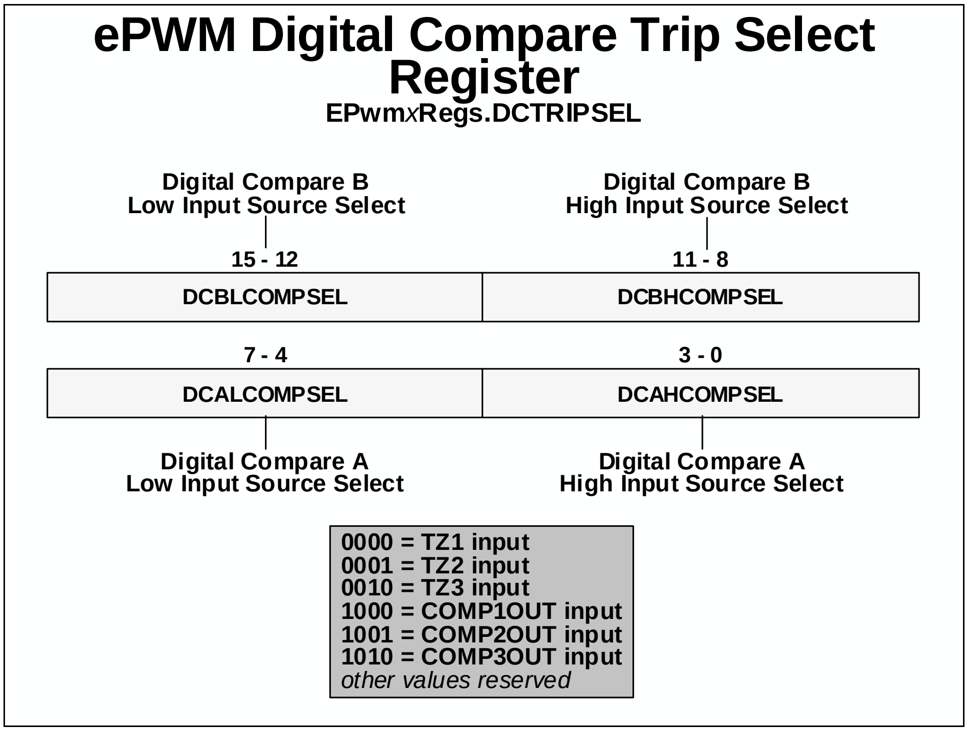
DC*y*EVT1 Source Force Sync Signal Select
0 = synchronous
1 = asynchronous

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

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

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

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



ePWM Digital Compare Trip Select Register

EPwm*x*Regs.DCTRIPSEL

Digital Compare B Low Input Source Select

Digital Compare B High Input Source Select

15 - 12	11 - 8
DCBLCOMPSEL	DCBHCOMPSEL

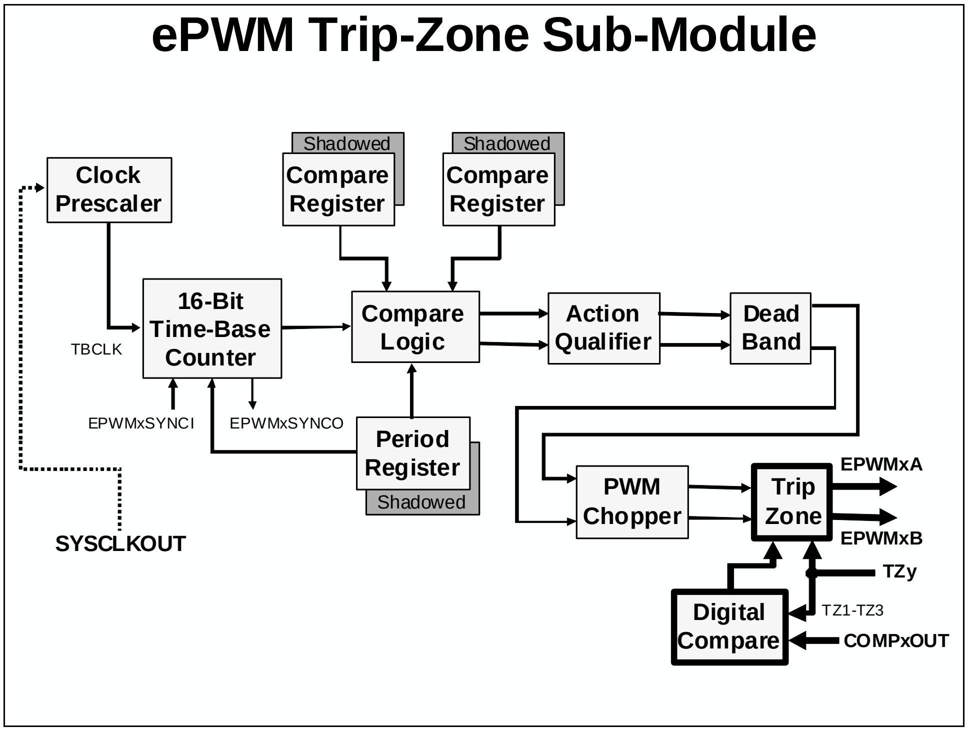
7 - 4	3 - 0
DCALCOMPSEL	DCAHCOMPSEL

Digital Compare A Low Input Source Select

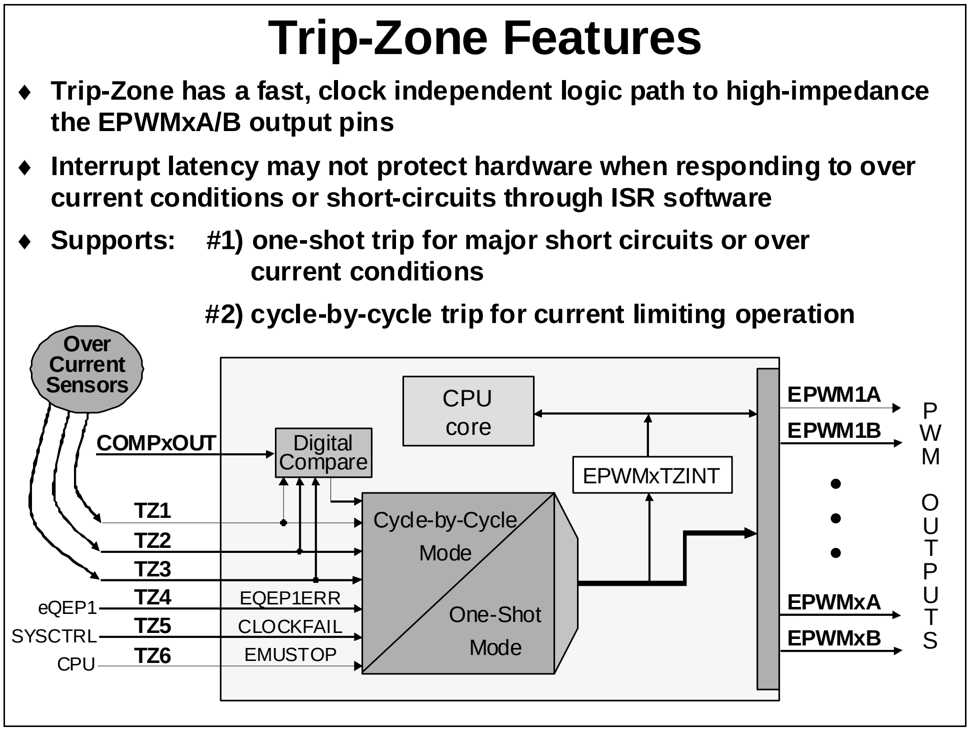
Digital Compare A High Input Source Select

0000 = TZ1 input
0001 = TZ2 input
0010 = TZ3 input
1000 = COMP1OUT input
1001 = COMP2OUT input
1010 = COMP3OUT input
other values reserved

ePWM Trip-Zone Sub-Module



ePWM Trip-Zone Sub-Module

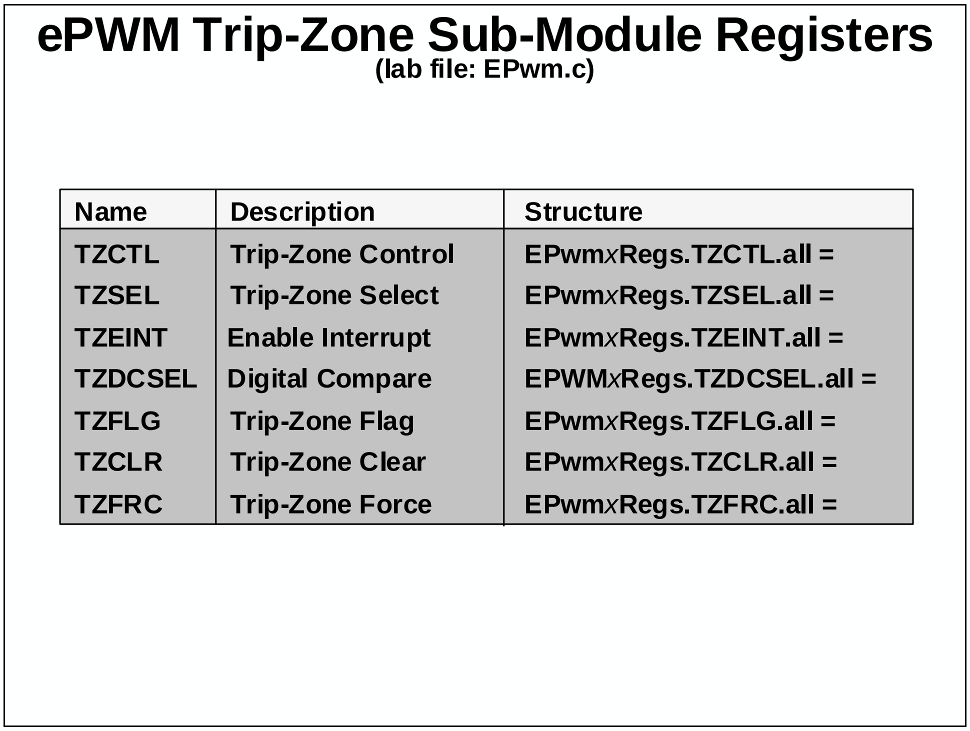


Trip-Zone Features

- ♦ 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

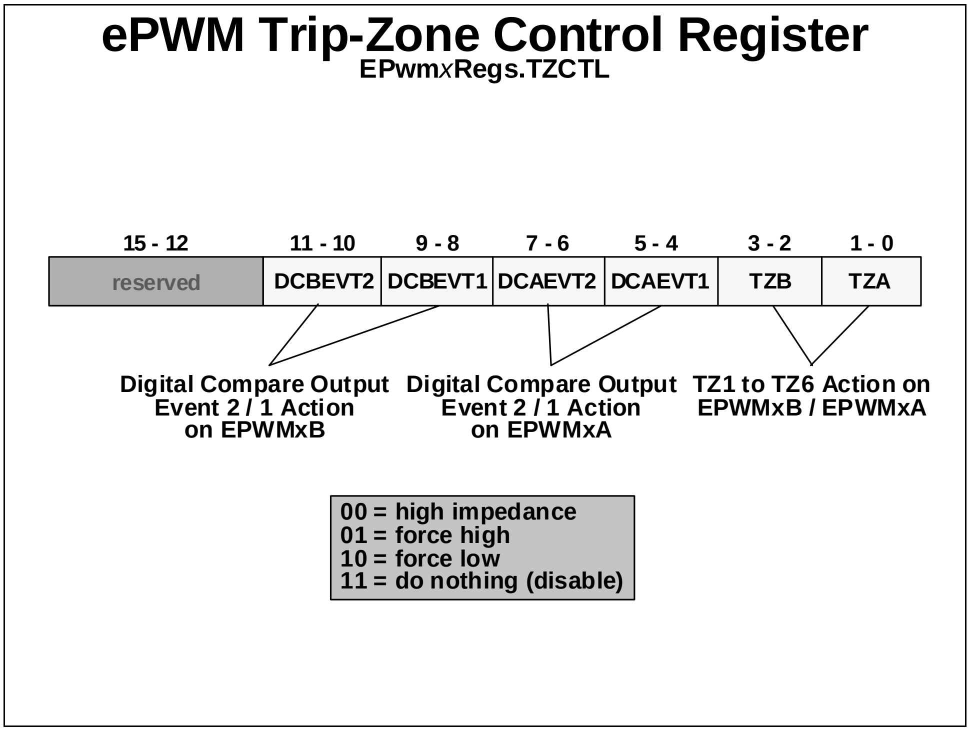
The power drive protection is a safety feature that is provided for the safe operation of systems such as power converters and motor drives. It can be used to inform the monitoring program of motor drive abnormalities such as over-voltage, over-current, and excessive temperature rise. If the power drive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



ePWM Trip-Zone Sub-Module Registers

(lab file: EPwm.c)

Name	Description	Structure
TZCTL	Trip-Zone Control	EPwm*x*Regs.TZCTL.all =
TZSEL	Trip-Zone Select	EPwm*x*Regs.TZSEL.all =
TZEINT	Enable Interrupt	EPwm*x*Regs.TZEINT.all =
TZDCSEL	Digital Compare	EPWM*x*Regs.TZDCSEL.all =
TZFLG	Trip-Zone Flag	EPwm*x*Regs.TZFLG.all =
TZCLR	Trip-Zone Clear	EPwm*x*Regs.TZCLR.all =
TZFRC	Trip-Zone Force	EPwm*x*Regs.TZFRC.all =



ePWM Trip-Zone Control Register

EPwm*x*Regs.TZCTL

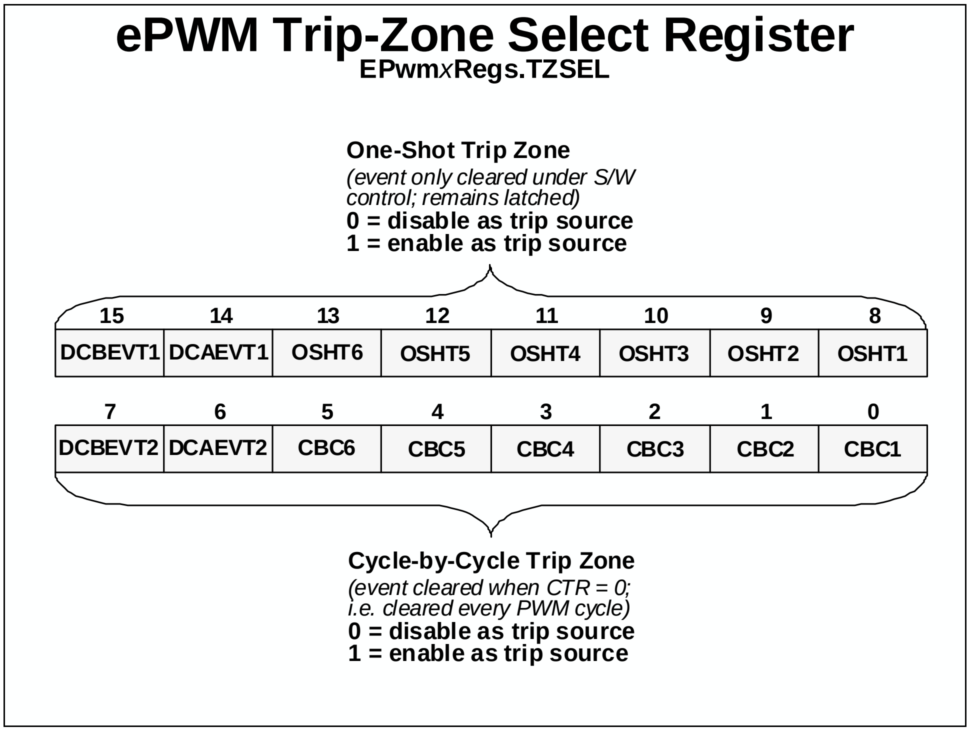
15 - 12	11 - 10	9 - 8	7 - 6	5 - 4	3 - 2	1 - 0
reserved	DCBEVT2	DCBEVT1	DCAEVT2	DCAEVT1	TZB	TZA

Digital Compare Output Event 2 / 1 Action on EPWMxB

Digital Compare Output Event 2 / 1 Action on EPWMxA

TZ1 to TZ6 Action on EPWMxB / EPWMxA

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



ePWM Trip-Zone Select Register

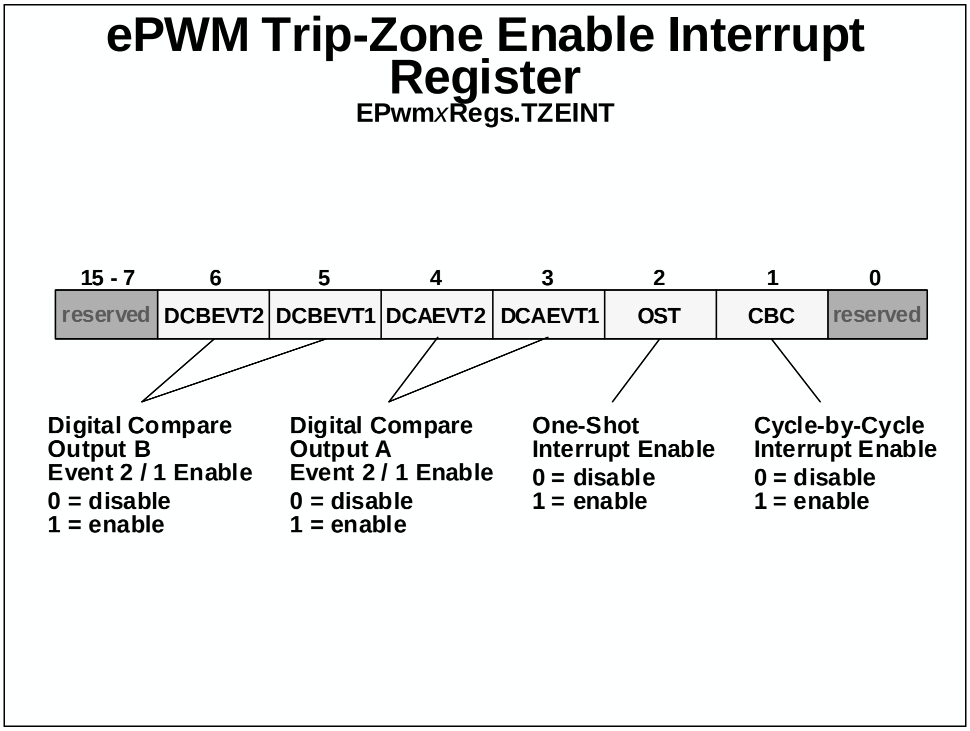
EPwm*x*Regs.TZSEL

One-Shot Trip Zone
(event only cleared under S/W control; remains latched)
0 = disable as trip source
1 = enable as trip source

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

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

Cycle-by-Cycle Trip Zone
(event cleared when CTR = 0; i.e. cleared every PWM cycle)
0 = disable as trip source
1 = enable as trip source



ePWM Trip-Zone Enable Interrupt Register

EPwm*x*Regs.TZEINT

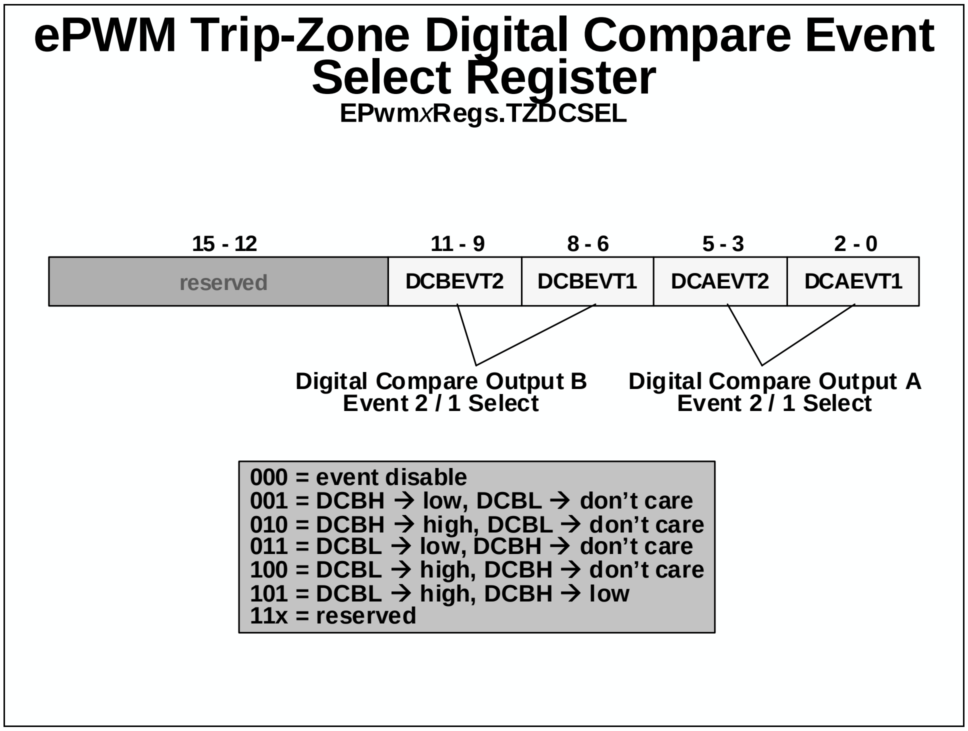
15 - 7	6	5	4	3	2	1	0
reserved	DCBEVT2	DCBEVT1	DCAEVT2	DCAEVT1	OST	CBC	reserved

Digital Compare Output B Event 2 / 1 Enable
0 = disable
1 = enable

Digital Compare Output A Event 2 / 1 Enable
0 = disable
1 = enable

One-Shot Interrupt Enable
0 = disable
1 = enable

Cycle-by-Cycle Interrupt Enable
0 = disable
1 = enable



ePWM Trip-Zone Digital Compare Event Select Register

EPwm*x*Regs.TZDCSEL

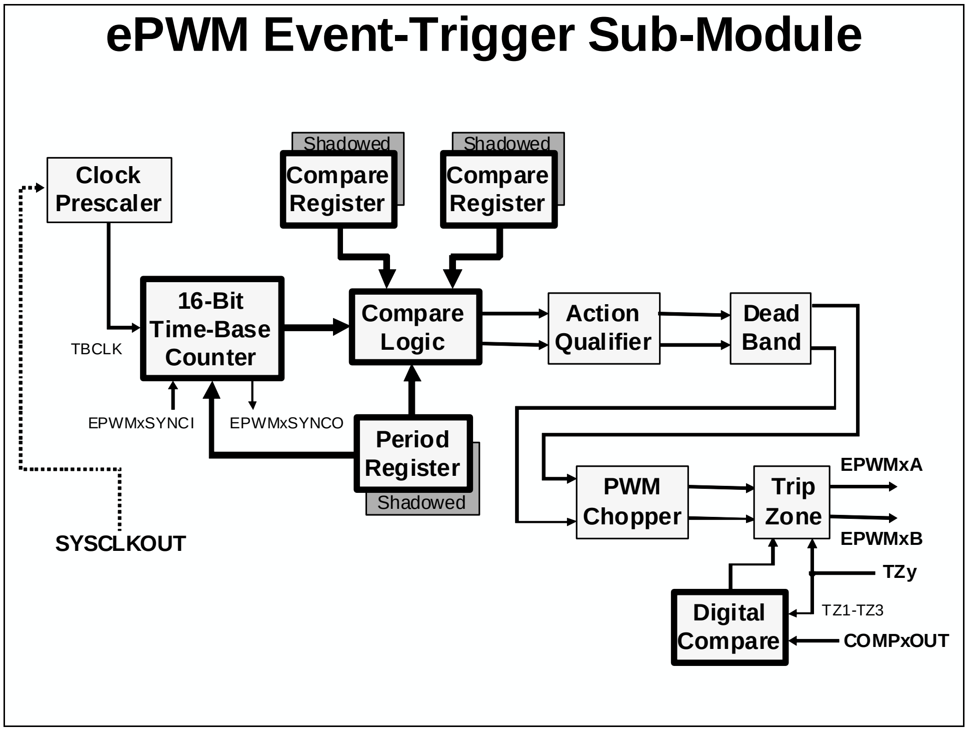
15 - 12	11 - 9	8 - 6	5 - 3	2 - 0
reserved	DCBEVT2	DCBEVT1	DCAEVT2	DCAEVT1

Digital Compare Output B
Event 2 / 1 Select

Digital Compare Output A
Event 2 / 1 Select

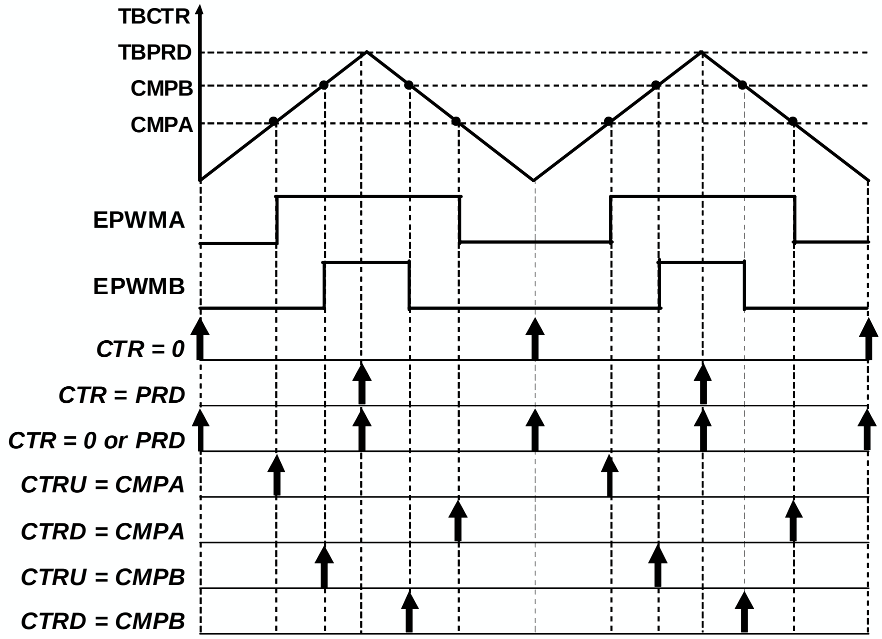
000 = event disable
001 = DCBH → low, DCBL → don't care
010 = DCBH → high, DCBL → don't care
011 = DCBL → low, DCBH → don't care
100 = DCBL → high, DCBH → don't care
101 = DCBL → high, DCBH → low
11x = reserved

ePWM Event-Trigger Sub-Module



ePWM Event-Trigger Sub-Module

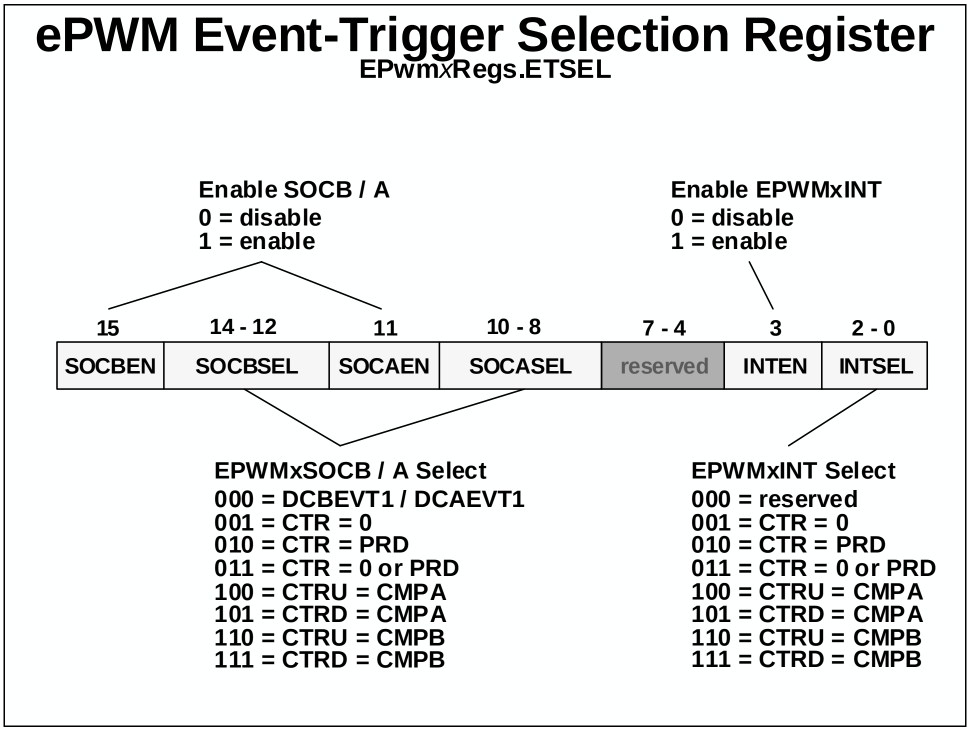
ePWM Event-Trigger Interrupts and SOC



ePWM Event-Trigger Sub-Module Registers

(lab file: EPwm.c)

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



ePWM Event-Trigger Selection Register

EPwm*x*Regs.ETSEL

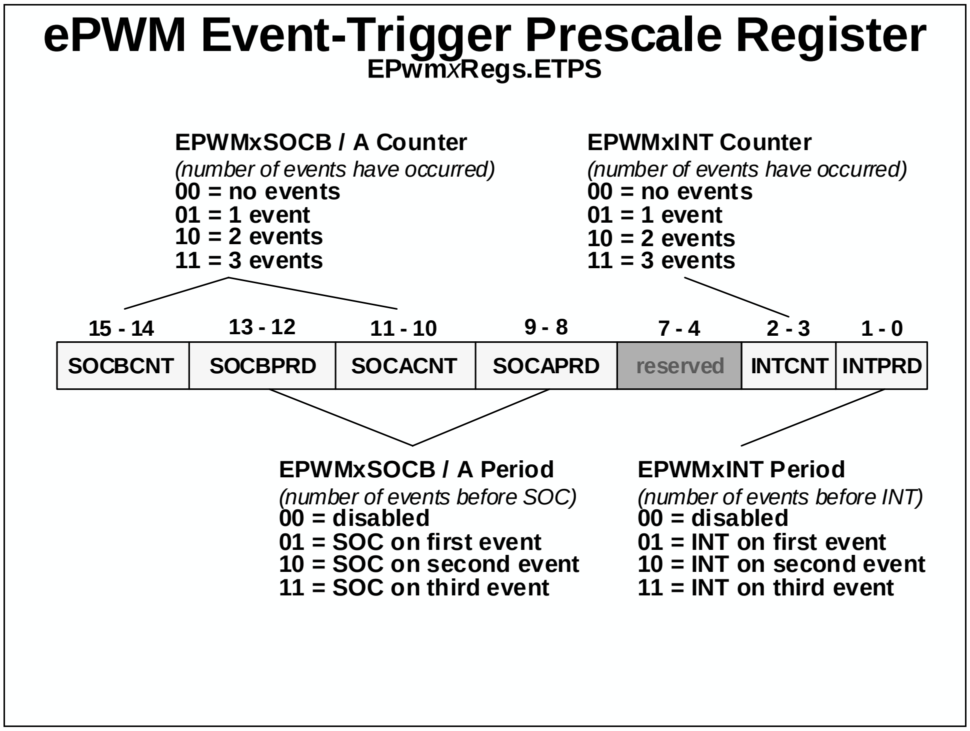
Enable SOCB / A
0 = disable
1 = enable

Enable EPWMxINT
0 = disable
1 = enable

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

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

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



ePWM Event-Trigger Prescale Register

EPwm*x*Regs.ETPS

EPWMxSOCB / A Counter
(number of events have occurred)
00 = no events
01 = 1 event
10 = 2 events
11 = 3 events

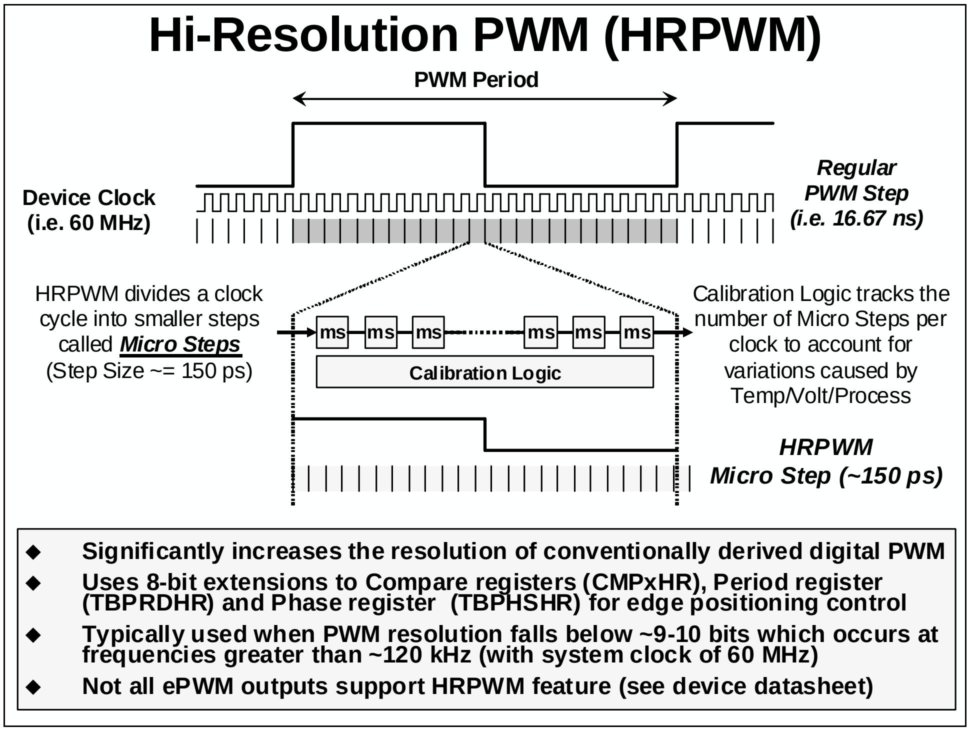
EPWMxINT Counter
(number of events have occurred)
00 = no events
01 = 1 event
10 = 2 events
11 = 3 events

15 - 14	13 - 12	11 - 10	9 - 8	7 - 4	2 - 3	1 - 0
SOCBCNT	SOCBPRD	SOCACNT	SOCAPRD	reserved	INTCNT	INTPRD

EPWMxSOCB / A Period
(number of events before SOC)
00 = disabled
01 = SOC on first event
10 = SOC on second event
11 = SOC on third event

EPWMxINT Period
(number of events before INT)
00 = disabled
01 = INT on first event
10 = INT on second event
11 = INT on third event

Hi-Resolution PWM (HRPWM)



Hi-Resolution PWM (HRPWM)

PWM Period

Device Clock
(i.e. 60 MHz)

Regular
PWM Step
(i.e. 16.67 ns)

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

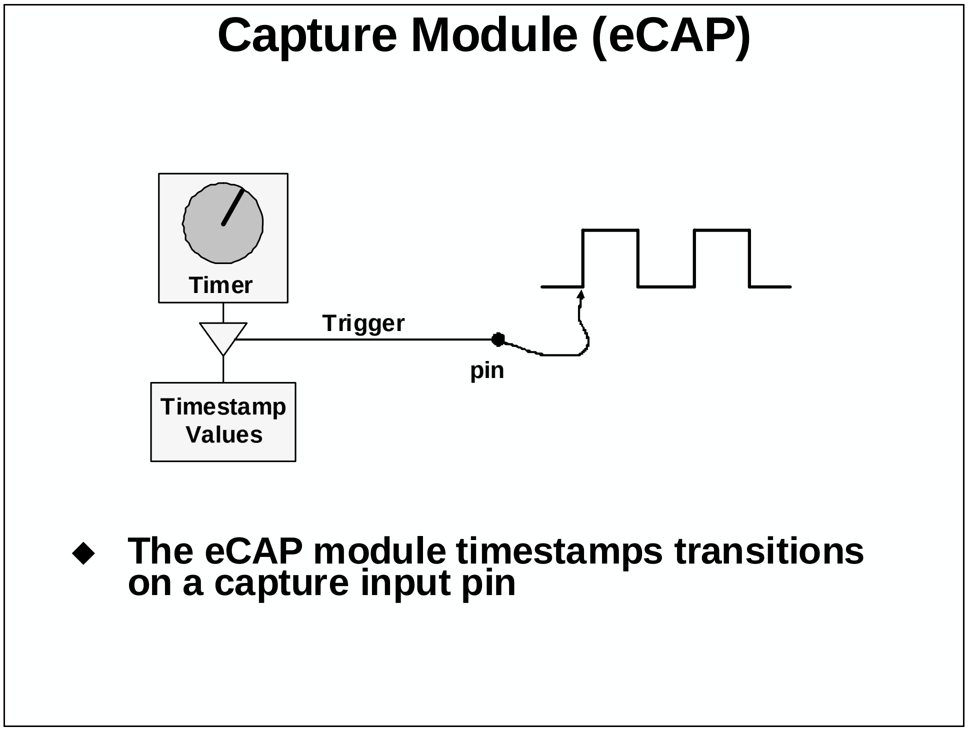
Calibration Logic

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

HRPWM
Micro Step (~150 ps)

- **Significantly increases the resolution of conventionally derived digital PWM**
- **Uses 8-bit extensions to Compare registers (CMPxHR), 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)**

eCAP



Capture Module (eCAP)

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

The capture units allow time-based logging of external TTL signal transitions on the capture input pins. The C28x has up to six capture units.

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

Some Uses for the Capture Module

- **Measure the time width of a pulse**
- **Low speed velocity estimation from incr. encoder:**

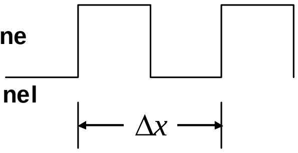
Problem: At low speeds, calculation of speed based on a measured position change at fixed time intervals produces large estimate errors

$$v_k \approx \frac{x_k - x_{k-1}}{\Delta t}$$

Alternative: Estimate the speed using a measured time interval at fixed position intervals

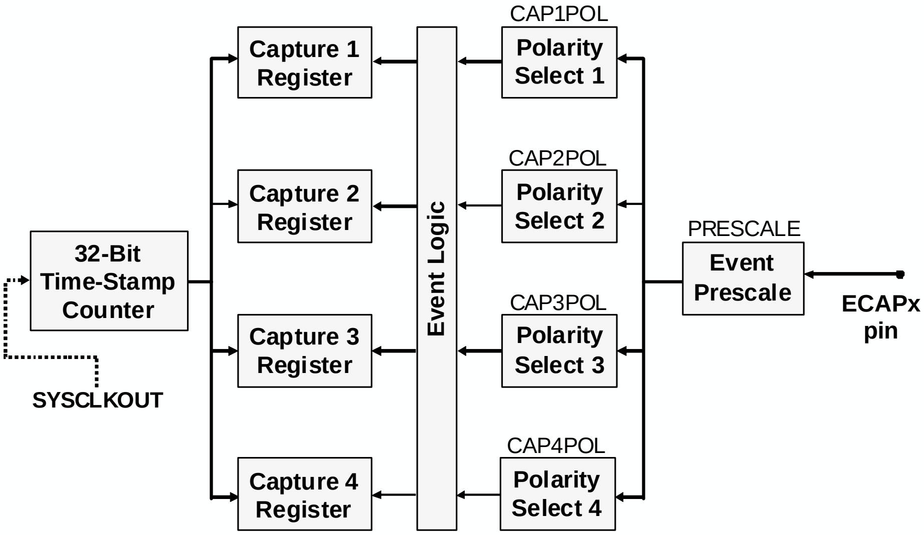
$$v_k \approx \frac{\Delta x}{t_k - t_{k-1}}$$

Signal from one quadrature encoder channel

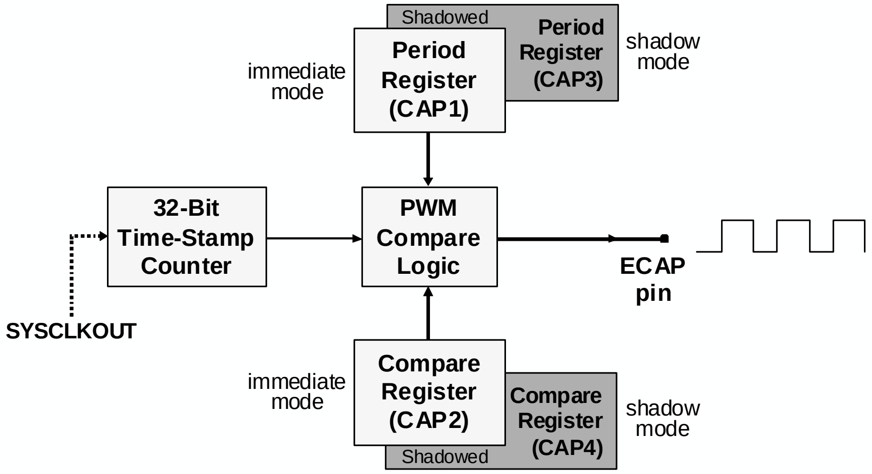


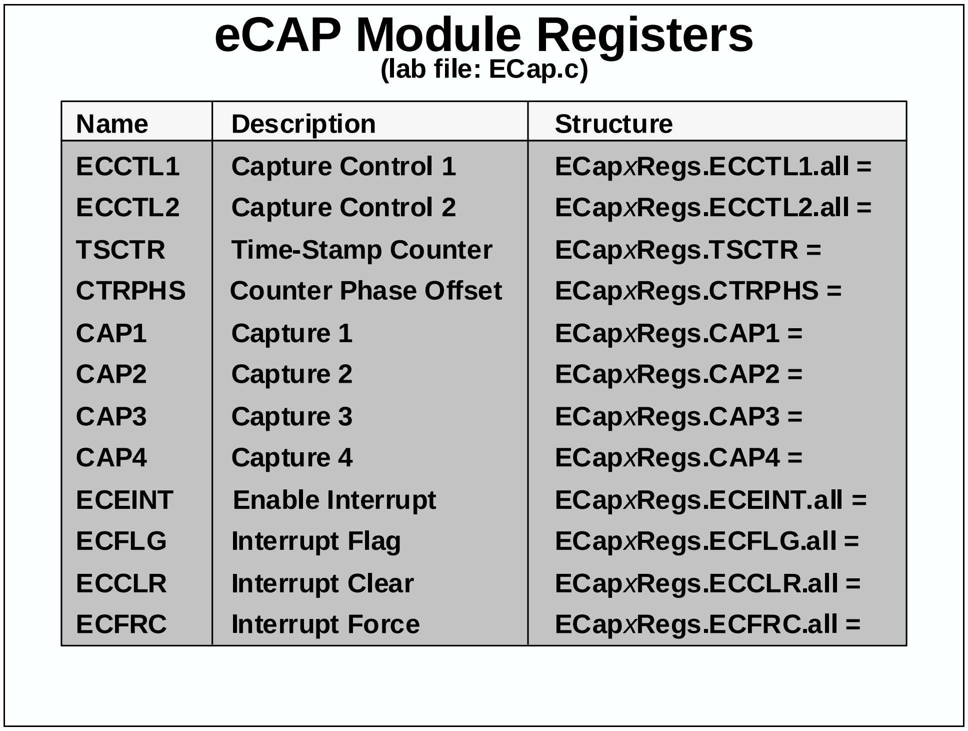
- **Auxiliary PWM generation**

eCAP Module Block Diagram – Capture Mode



eCAP Module Block Diagram – APWM Mode





eCAP Module Registers

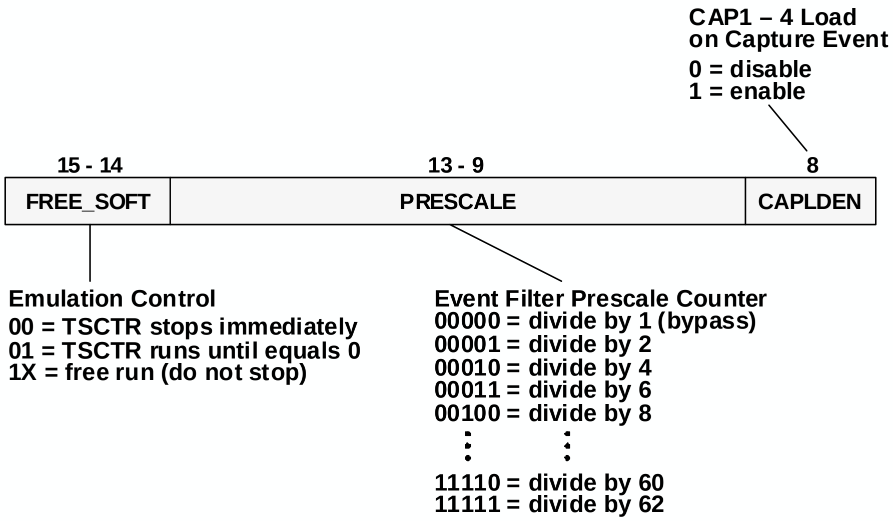
(lab file: ECap.c)

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

eCAP Control Register 1

ECap*x*Regs.ECCTL1

Upper Register:



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

15 - 14	13 - 9	8
FREE_SOFT	PRESCALE	CAPLDEN

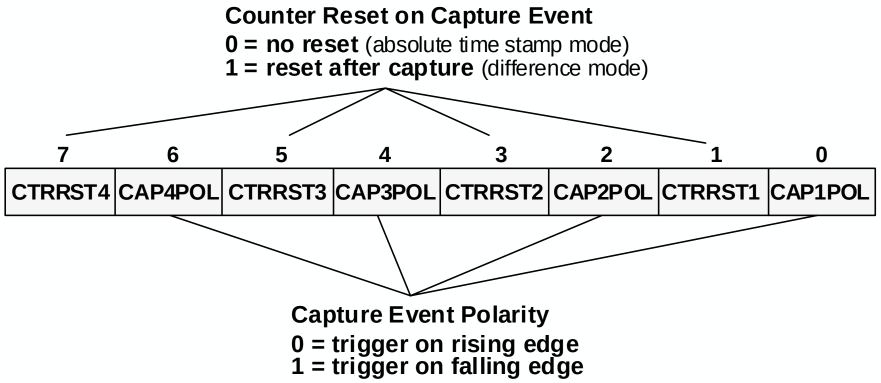
Emulation Control
00 = TSCTR stops immediately
01 = TSCTR runs until equals 0
1X = free run (do not stop)

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

eCAP Control Register 1

ECap*x*Regs.ECCTL1

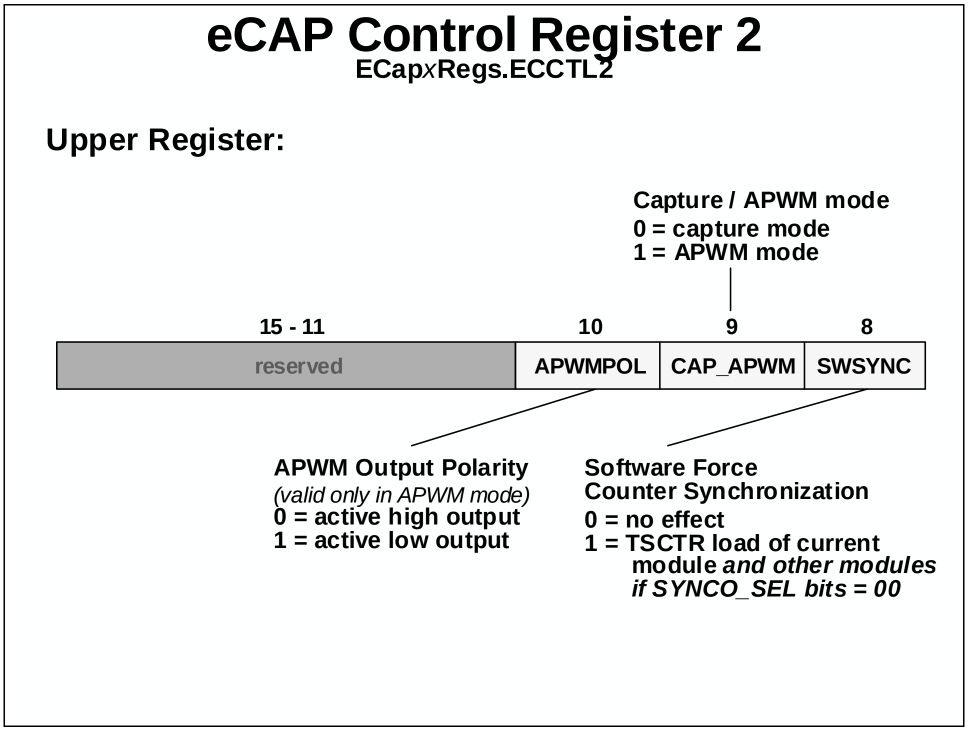
Lower Register:



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

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

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



eCAP Control Register 2

ECap*x*Regs.ECCTL2

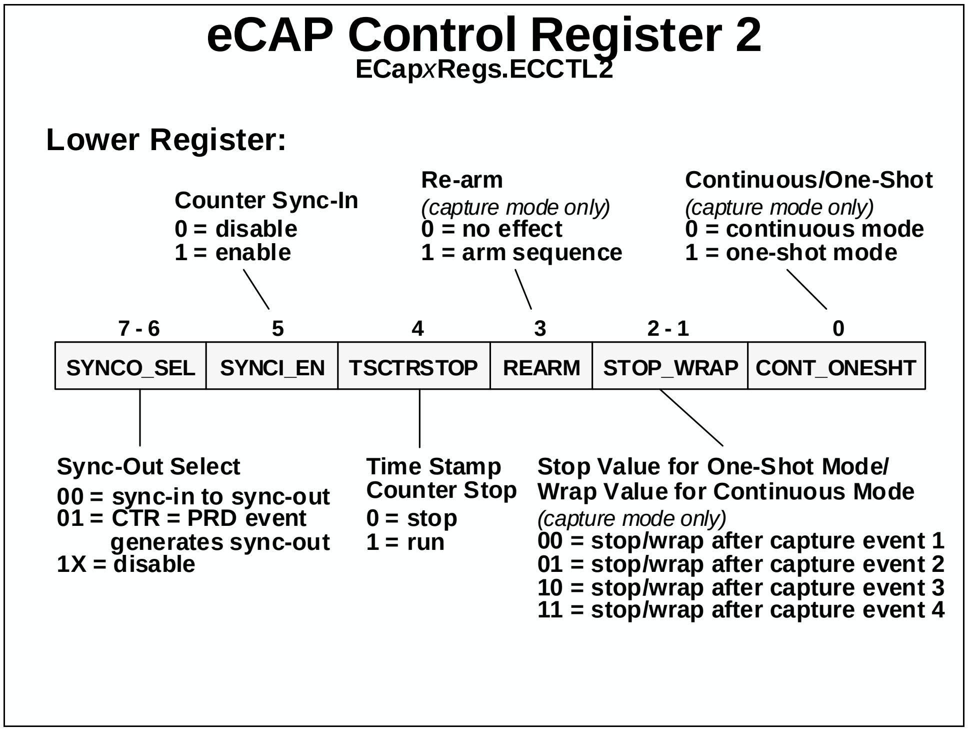
Upper Register:

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

15 - 11	10	9	8
reserved	APWMPOL	CAP_APWM	SWSYNC

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

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



eCAP Control Register 2

ECap*x*Regs.ECCTL2

Lower Register:

Counter Sync-In
0 = disable
1 = enable

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

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

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

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

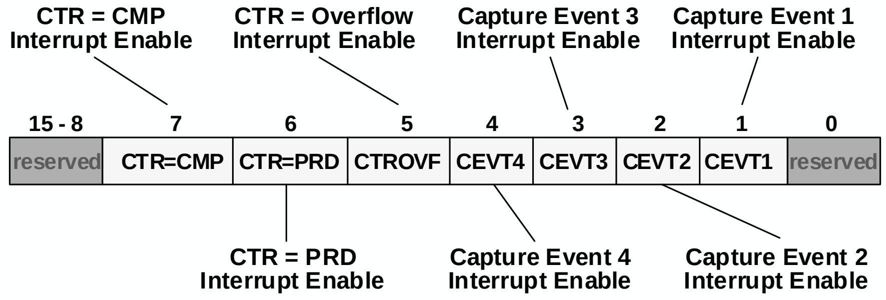
Time Stamp Counter Stop
0 = stop
1 = run

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

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

eCAP Interrupt Enable Register

ECap*x*Regs.ECEINT



CTR = CMP Interrupt Enable

CTR = Overflow Interrupt Enable

Capture Event 3 Interrupt Enable

Capture Event 1 Interrupt Enable

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

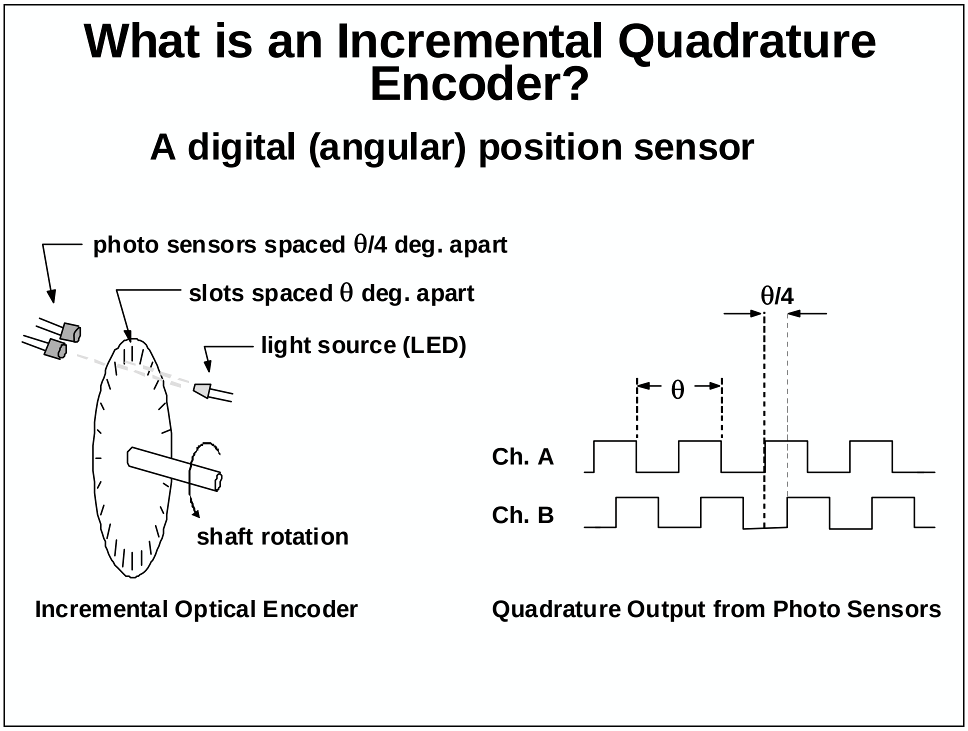
CTR = PRD Interrupt Enable

Capture Event 4 Interrupt Enable

Capture Event 2 Interrupt Enable

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

eQEP



What is an Incremental Quadrature Encoder?

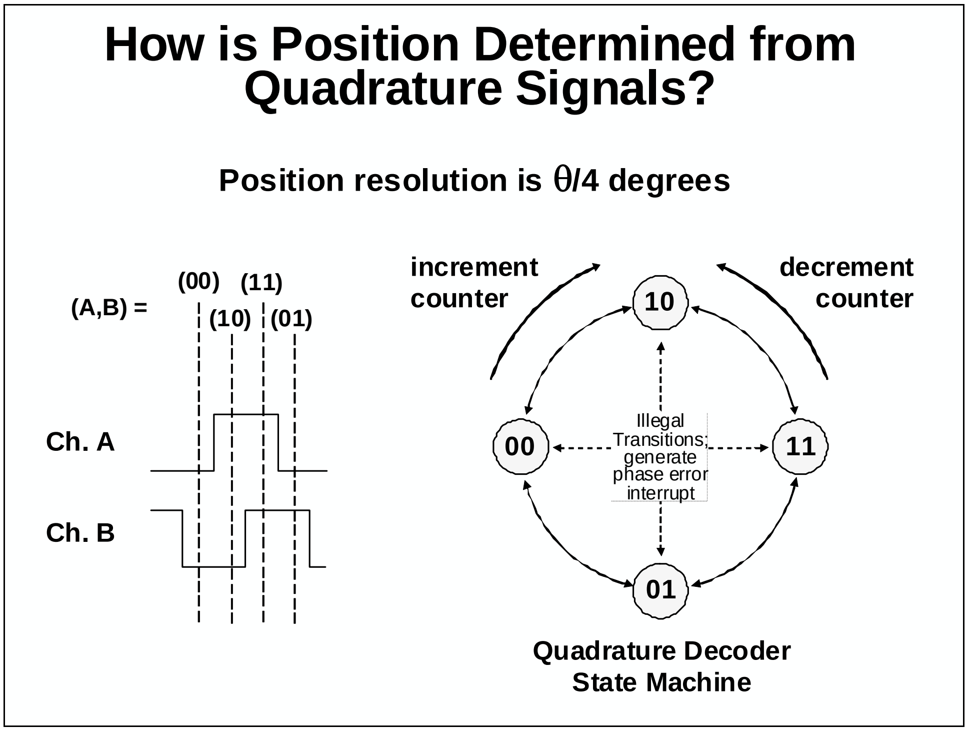
A digital (angular) position sensor

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

Incremental Optical Encoder

Quadrature Output from Photo Sensors

The eQEP circuit, when enabled, decodes and counts the quadrature encoded input pulses. The QEP circuit can be used to interface with an optical encoder to get position and speed information from a rotating machine.

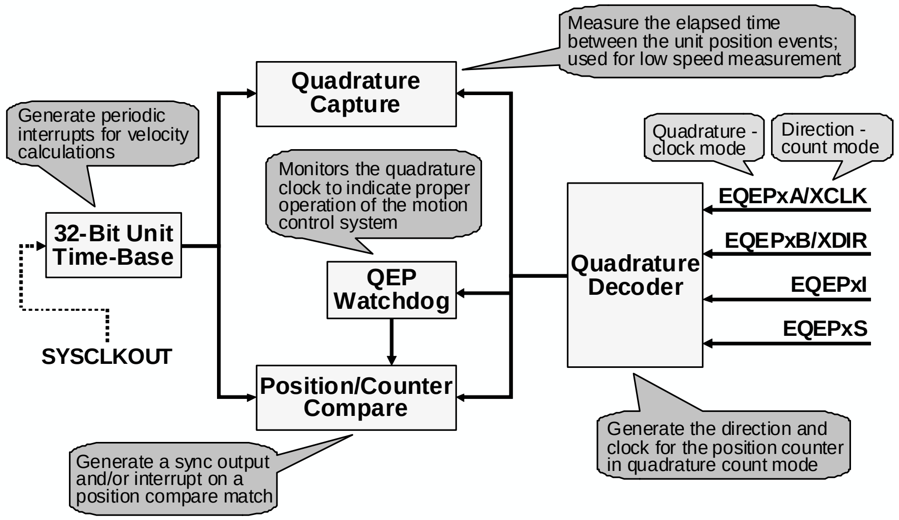


How is Position Determined from Quadrature Signals?

Position resolution is θ/4 degrees

Quadrature Decoder State Machine

eQEP Module Block Diagram



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

Generate periodic interrupts for velocity calculations

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

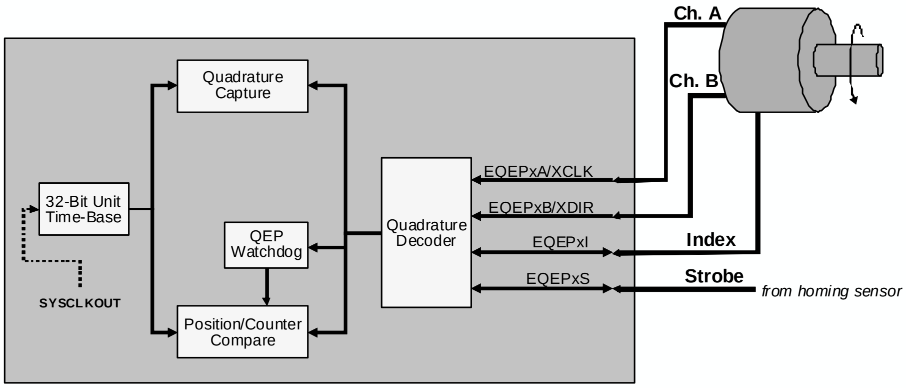
Quadrature - clock mode

Direction - count mode

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

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

eQEP Module Connections

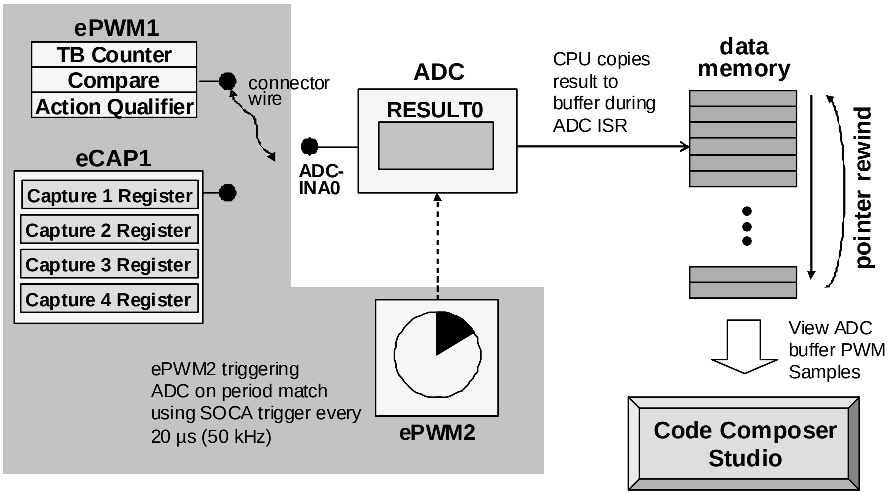


Lab 7: Control Peripherals

➢ Objective

The objective of this lab is to become familiar with the programming and operation of the control peripherals and their interrupts. ePWM1A will be setup to generate a 2 kHz, 25% duty cycle 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. Next, eCAP1 will be setup to detect the rising and falling edges of the waveform. This information will be used to determine the width of the pulse and duty cycle of the waveform. The results of this step will be viewed numerically in a memory window.

Lab 7: Control Peripherals



➢ Procedure

Project File

1. A project named `Lab7.pjt` has been created for this lab. Open the project by clicking on `Project` → `Open…` and look in `C:\C28x\Labs\Lab7`. All Build Options have been configured the same as the previous lab. The files used in this lab are:

```
Adc.c
CodeStartBranch.asm
DefaultIsr_7.c
DelayUs.asm
DSP2833x_GlobalVariableDefs.c
DSP2833x_Headers_nonBIOS.cmd
ECap_7_8_9_10_12.c
EPwm_7_8_9_10_12.c
Gpio.c
Lab_5_6_7.cmd
Main_7.c
PieCtrl_5_6_7_8_9_10.c
PieVect_5_6_7_8_9_10.c
SysCtrl.c
Watchdog.c
```

Setup Shared I/O and ePWM1

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

3. In `EPwm_7_8_9_10_12.c`, setup ePWM1 to implement the PWM waveform as described in the objective for this lab. The following registers need to be modified: TBCTL (set clock prescales to divide-by-1, no software force, sync and phase disabled), TBPRD, CMPA, CMPCTL (load on 0 or PRD), and AQCTLA (set on up count and clear on down count for output A). Software force, deadband, PWM chopper and trip action has been disabled. (Hint – notice the last steps enable the timer count mode and enable the clock to the ePWM module). Either calculate the values for TBPRD and CMPA (as a challenge) or make use of the global variable names and values that have been set using #define in the beginning of `Lab.h` file. Notice that ePWM2 has been initialized earlier in the code for the ADC lab. Save your work.

Build and Load

4. Save all changes to the files and click the `“Build”` button to build and load the project.

Run the Code – PWM Waveform

5. Open a memory window to view some of the contents of the ADC results buffer. The address label for the ADC results buffer is *AdcBuf*. We will be running our code in real-time mode, and will have our window continuously refresh.

6. Using a connector wire provided, connect the PWM1A (pin # GPIO-00) to ADCINA0 (pin # ADC-A0) on the Docking Station.

7. Run the code (real-time mode) using the GEL function: `GEL → Realtime Emulation Control → Run_Realtime_with_Reset`. Watch the window update. Verify that the ADC result buffer contains the updated values.

8. Open and setup a graph to plot a 50-point window of the ADC results buffer. Click: `View → Graph → Time/Frequency…` and set the following values:

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

Select `OK` to save the graph options.

9. The graphical display should show the generated 2 kHz, 25% duty cycle symmetric PWM waveform. The period of a 2 kHz signal is 500 μs. You can confirm this by measuring the period of the waveform using the graph (you may want to enlarge the graph window using the mouse). The measurement is best done with the mouse. The lower left-hand corner of the graph window will display the X and Y-axis values. Subtract the X-axis values taken over a complete waveform period (you can use the PC calculator program found in Microsoft Windows to do this).

Frequency Domain Graphing Feature of Code Composer Studio

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

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

Display Type	FFT Magnitude
Start Address	AdcBuf
Acquisition Buffer Size	50
FFT Framesize	50
DSP Data Type	16-bit unsigned integer
Sampling Rate (Hz)	50000

    Select `OK` to save the graph options.

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

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

Setup eCAP1 to Measure Width of Pulse

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

13. Add the following file to the project:

    `ECap_7_8_9_10_12.c`

Check your files list to make sure the file is there.

14. In `Main_7.c`, add code to call the `InitECap()` function. There are no passed parameters or return values, so the call code is simply:

```
InitECap();
```

15. Edit `Gpio.c` and adjust the shared I/O pin in GPIO5 for the ECAP1 function.

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

17. In `ECap_7_8_9_10_12.c`, setup eCAP1 to calculate PWM_duty and PWM_period. The following registers need to be modified: ECCTL2 (continuous mode, re-arm disable, and sync disable), ECCTL1 (set prescale to divide-by-1, configure capture event polarity without reseting the counter), and ECEINT (enable desired eCAP interrupt).

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

PIE group #:__________ # within group:_________

This information will be used in the next step.

19. Modify the end of `ECap_7_8_9_10_12.c` to do the following:

- Enable the “ECAP1_INT” interrupt in the PIE (Hint: use the `PieCtrlRegs` structure)
- Enable the appropriate core interrupt in the IER register

Build and Load

20. Save all changes to the files and click the `“Build”` button.

Run the Code – Pulse Width Measurement

21. Open a memory window to view the address label *PwmPeriod*. (Type ***&PwmPeriod*** in the address box). The address label *PwmDuty* (address ***&PwmDuty***) should appear in the same memory window.

22. Set the memory window properties format to “32-Bit UnSigned Int”.

23. Using the connector wire provided, connect the PWM1A (pin # GPIO-00) to ECAP1 (pin # GPIO-05) on the Docking Station.

24. Run the code (real-time mode) by using the GEL function: `GEL → Realtime Emulation Control → Run_Realtime_with_Reset`. Notice the values for *PwmDuty* and *PwmPeriod.*

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

Questions:

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

End of Exercise

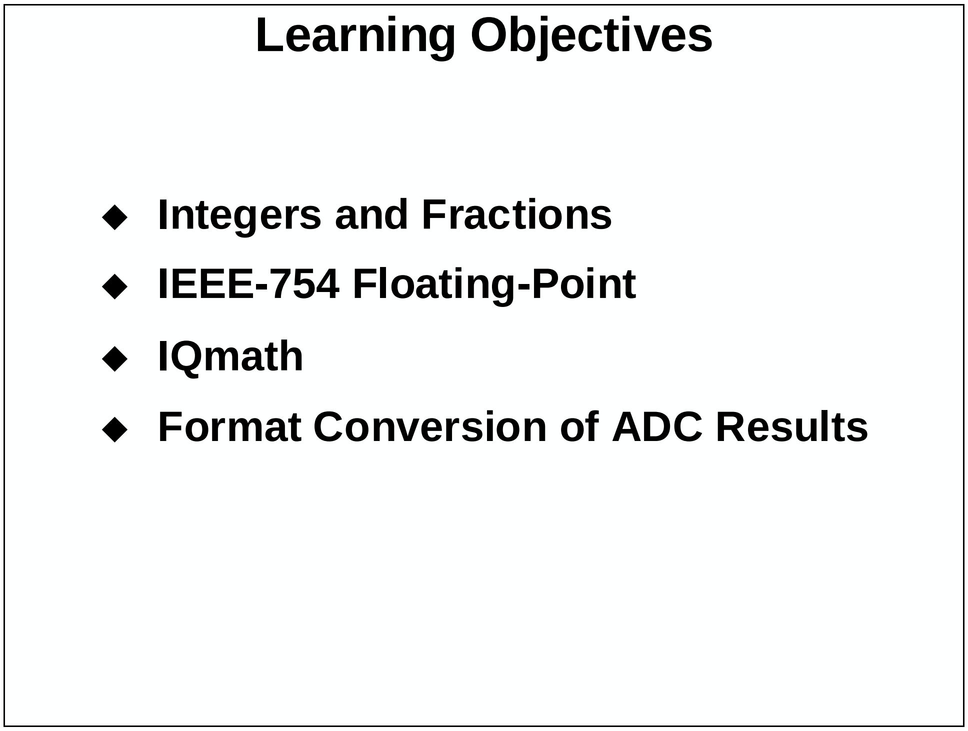
Numerical Concepts

Introduction

In this module, numerical concepts will be explored. One of the first considerations concerns multiplication – how does the user store the results of a multiplication, when the process of multiplication creates results larger than the inputs. A similar concern arises when considering accumulation – especially when long summations are performed. Next, floating-point concepts will be explored and IQmath will be described as a technique for implementing a “virtual floating-point” system to simplify the design process.

The IQmath Library is a collection of highly optimized and high precision mathematical functions used to seamlessly port floating-point algorithms into fixed-point code. These C/C++ routines are typically used in computationally intensive real-time applications where optimal execution speed and high accuracy is needed. By using these routines a user can achieve execution speeds considerable faster than equivalent code written in standard ANSI C language. In addition, by incorporating the ready-to-use high precision functions, the IQmath library can shorten significantly a DSP application development time. (The IQmath user's guide is included in the application zip file, and can be found in the /docs folder once the file is extracted and installed).

Learning Objectives



Learning Objectives

- Integers and Fractions
- IEEE-754 Floating-Point
- IQmath
- Format Conversion of ADC Results

Module Topics

Numbering System Basics

Given the ability to perform arithmetic processes (addition and multiplication) with the C28x, it is important to understand the underlying mathematical issues which come into play. Therefore, we shall examine the numerical concepts which apply to the C28x and, to a large degree, most processors.

Binary Numbers

The binary numbering system is the simplest numbering scheme used in computers, and is the basis for other schemes. Some details about this system are:

- It uses only two values: 1 and 0
- Each binary digit, commonly referred to as a bit, is one “place” in a binary number and represents an increasing power of 2.
- The least significant bit (LSB) is to the right and has the value of 1.
- Values are represented by setting the appropriate 1's in the binary number.
- The number of bits used determines how large a number may be represented.

Examples:

$0110_2 = (0 * 8) + (1 * 4) + (1 * 2) + (0 * 1) = 6_{10}$
$11110_2 = (1 * 16) + (1 * 8) + (1 * 4) + (1 * 2) + (0 * 1) = 30_{10}$

Two's Complement Numbers

Notice that binary numbers can only represent **positive** numbers. Often it is desirable to be able to represent both positive and negative numbers. The two's complement numbering system modifies the binary system to include negative numbers by making the most significant bit (MSB) **negative**. Thus, two's complement numbers:

- Follow the binary progression of simple binary except that the MSB is negative — in addition to its magnitude
- Can have any number of bits — more bits allow larger numbers to be represented

Examples:

$0110_2 = (0 * -8) + (1 * 4) + (1 * 2) + (0 * 1) = 6_{10}$
$11110_2 = (1 * -16) + (1 * 8) + (1 * 4) + (1 * 2) + (0 * 1) = -2_{10}$

The same binary values are used in these examples for two's complement as were used above for binary. Notice that the decimal value is the same when the MSB is 0, but the decimal value is quite different when the MSB is 1.

Two operations are useful in working with two's complement numbers:

- The ability to obtain an additive inverse of a value
- The ability to load small numbers into larger registers (by sign extending)

To load small two's complement numbers into larger registers:

The MSB of the original number must carry to the MSB of the number when represented in the larger register.

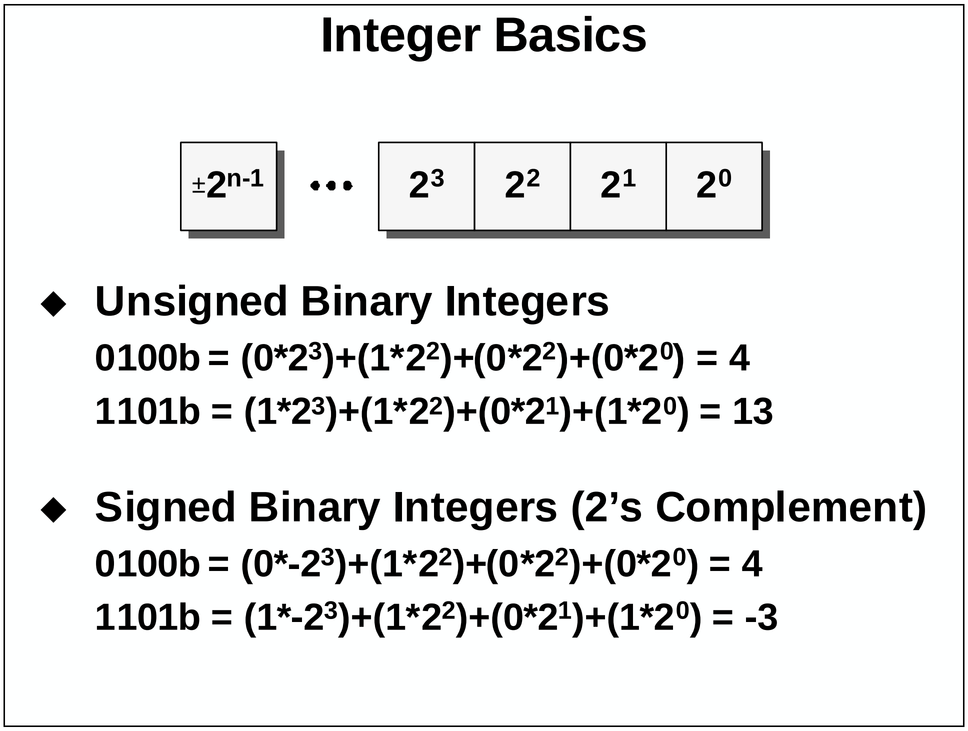
1. Load the small number “right justified” into the larger register.
2. Copy the sign bit (the MSB) of the original number to all unfilled bits to the left in the register (sign extension).

Consider our two previous values, copied into an 8-bit register:

Examples:

Original No.	0 1 1 0_2	= 6_{10}	1 1 1 1 0_2	= -2_{10}
1. Load low	0 1 1 0		1 1 1 1 0	
2. Sign Extend	0 0 0 0 0 1 1 0	= 4 + 2 = 6	1 1 1 1 1 1 1 0	= -128 + 64 + ... + 2 = -2

Integer Basics



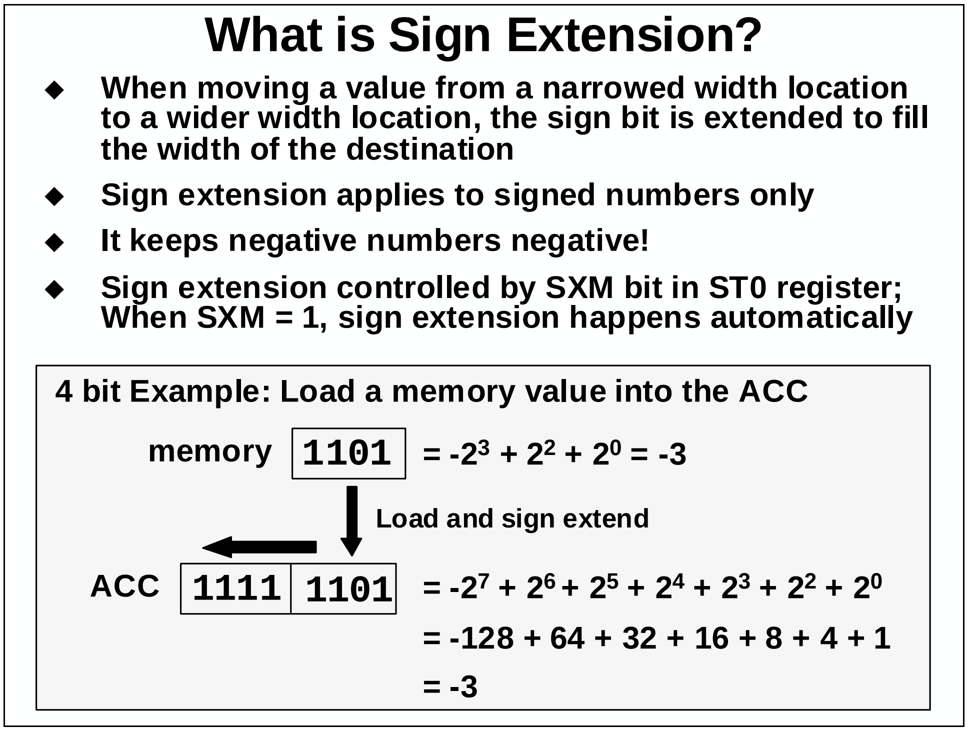
Integer Basics

$\pm 2^{n-1}$ ••• 2^3 2^2 2^1 2^0

- **Unsigned Binary Integers**
  0100b = $(0*2^3)+(1*2^2)+(0*2^2)+(0*2^0) = 4$
  1101b = $(1*2^3)+(1*2^2)+(0*2^1)+(1*2^0) = 13$
- **Signed Binary Integers (2’s Complement)**
  0100b = $(0*-2^3)+(1*2^2)+(0*2^2)+(0*2^0) = 4$
  1101b = $(1*-2^3)+(1*2^2)+(0*2^1)+(1*2^0) = -3$

Sign Extension Mode

The C28x can operate on either unsigned binary or two's complement operands. The “Sign Extension Mode” (SXM) bit, present within a status register of the C28x, identifies whether or not the sign extension process is used when a value is brought into the accumulator. It is good programming practice to always select the desired SXM at the beginning of a module to assure the proper mode.



What is Sign Extension?

- When moving a value from a narrowed width location to a wider width location, the sign bit is extended to fill the width of the destination
- Sign extension applies to signed numbers only
- It keeps negative numbers negative!
- Sign extension controlled by SXM bit in ST0 register; When SXM = 1, sign extension happens automatically

4 bit Example: Load a memory value into the ACC

memory $\boxed{1101}$ $= -2^3 + 2^2 + 2^0 = -3$

Load and sign extend

ACC $\boxed{1111}\boxed{1101}$ $= -2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^0$

$= -128 + 64 + 32 + 16 + 8 + 4 + 1$

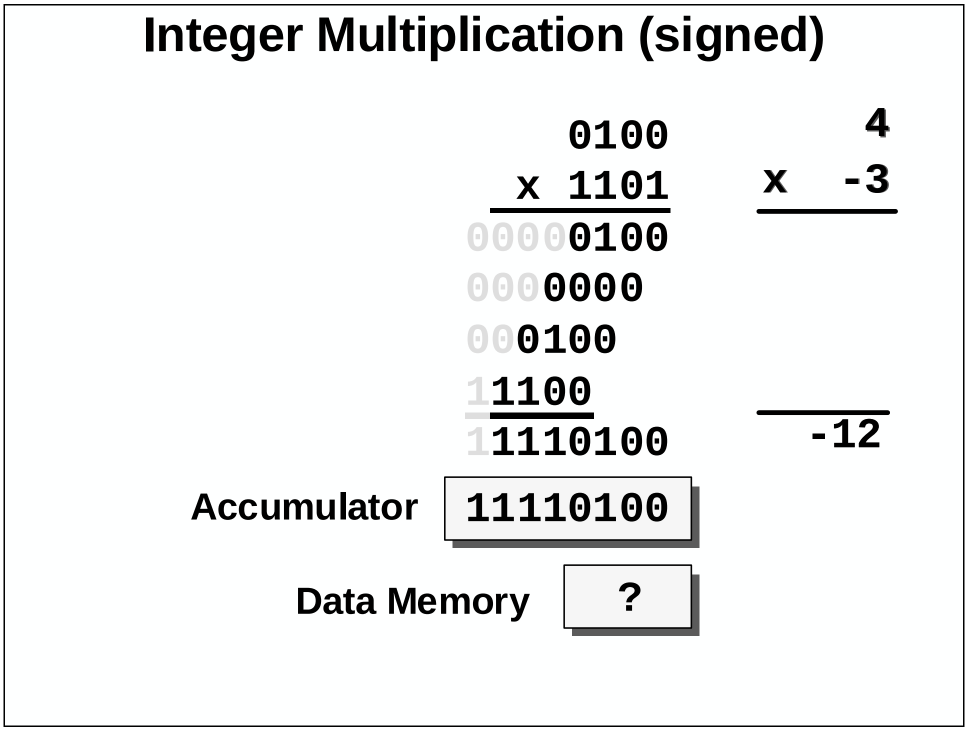
$= -3$

Binary Multiplication

Now that you understand two's complement numbers, consider the process of multiplying two two's complement values. As with “long hand” decimal multiplication, we can perform binary multiplication one “place” at a time, and sum the results together at the end to obtain the total product.

Note: This is not the method the C28x uses in multiplying numbers — it is merely a way of observing how binary numbers work in arithmetic processes.

The C28x uses 16-bit operands and a 32-bit accumulator. For the sake of clarity, consider the example below where we shall investigate the use of 4-bit values and an 8-bit accumulation:



Integer Multiplication (signed)

```
     0100         4
   x 1101      x -3
 00000100
 0000000
 000100
 11100
 11110100       -12

Accumulator  11110100

Data Memory  ?
```

In this example, consider the following:

- What are the two input values, and the expected result?
- Why are the “partial products” shifted left as the calculation continues?
- Why is the final partial product “different” than the others?
- What is the result obtained when adding the partial products?
- How shall this result be loaded into the accumulator?
- How shall we fill the remaining bit? Is this value still the expected one?
- How can the result be stored back to memory? What problems arise?

Note: With two's complement multiplication, the leading "1" in the second multiplicand is a sign bit. If the sign bit is "1", then take the 2's complement of the first multiplicand. Additionally, each partial product must be sign-extended for correct computation.

Note: All of the above questions except the final one are addressed in this module. The last question may have several answers:

- Store the lower accumulator to memory. What problem is apparent using this method in this example?
- Store the upper accumulator back to memory. Wouldn't this create a loss of precision, and a problem in how to interpret the results later?
- Store **both** the upper and lower accumulator to memory. This solves the above problems, but creates some new ones:
  - Extra code space, memory space, and cycle time are used
  - How can the result be used as the input to a subsequent calculation? Is such a condition likely (consider any "feedback" system)?

From this analysis, it is clear that integers do not behave well when multiplied. Might some other type of number system behave better? Is there a number system where the results of a multiplication are bounded?

Binary Fractions

Given the problems associated with integers and multiplication, consider the possibilities of using **fractional** values. Fractions do not grow when multiplied, therefore, they remain representable within a given word size and solve the problem. Given the benefit of fractional multiplication, consider the issues involved with using fractions:

- How are fractions represented in two's complement?
- What issues are involved when multiplying two fractions?

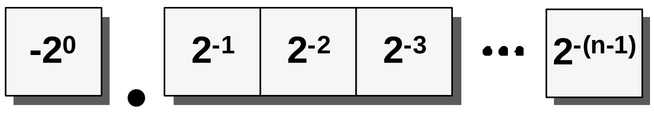
Representing Fractions in Binary

In order to represent both positive and negative values, the two's complement process will again be used. However, in the case of fractions, we will not set the LSB to 1 (as was the case for integers). When one considers that the range of fractions is from -1 to ~+1, and that the only bit which conveys negative information is the MSB, it seems that the MSB must be the “negative ones position.” Since binary representation is based on powers of two, it follows that the next bit would be the “one-halves” position, and that each following bit would have half the magnitude again. Considering, as before, a 4-bit model, we have the representation shown in the following example.

1	.	0	1	1	= -1 + 1/4 + 1/8 = -5/8
-1		1/2	1/4	1/8	

Fraction Basics

Fraction Basics

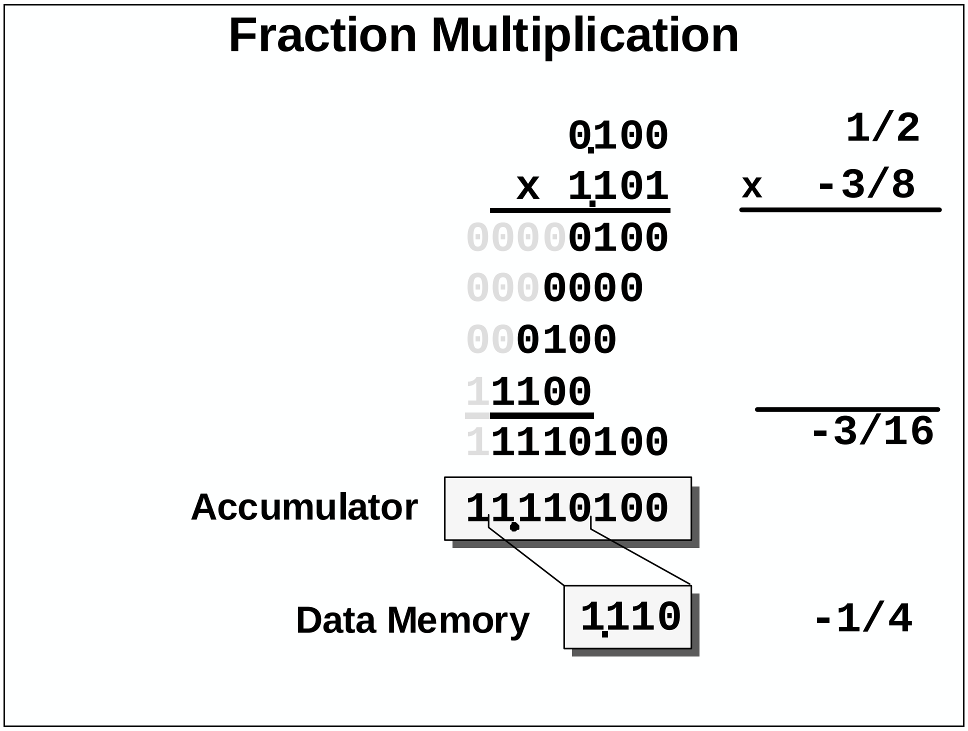


$$1101\text{b} = (1*-2^0)+(1*2^{-1})+(0*2^{-2})+(1*2^{-3})$$
$$= -1 + 1/2 + 1/8$$
$$= -3/8$$

Fractions have the nice property that fraction x fraction = fraction

Multiplying Binary Fractions

When the C28x performs multiplication, the process is identical for all operands, integers or fractions. Therefore, the user must determine how to interpret the results. As before, consider the 4-bit multiply example:



Fraction Multiplication

```
                 0.100          1/2
               x 1.101     x   -3/8
              00000100
              0000000
              000100
              11100
              11110100         -3/16

Accumulator   11.110100

Data Memory      1.110         -1/4
```

As before, consider the following:

- What are the two input values and the expected result?
- As before, “partial products” are shifted left and the final is negative.
- How is the result (obtained when adding the partial products) read?
- How shall this result be loaded into the accumulator?
- How shall we fill the remaining bit? Is this value still the expected one?
- How can the result be stored back to memory? What problems arise?

To “read” the results of the fractional multiply, it is necessary to locate the binary point (the base 2 equivalent of the base 10 decimal point). Start by identifying the location of the binary point in the input values. The MSB is an integer and the next bit is 1/2, therefore, the binary point would be located between them. In our example, therefore, we would have three bits to the right of the binary point in each input value. For ease of description, we can refer to these as “Q3” numbers, where Q refers to the number of places to the right of the point.

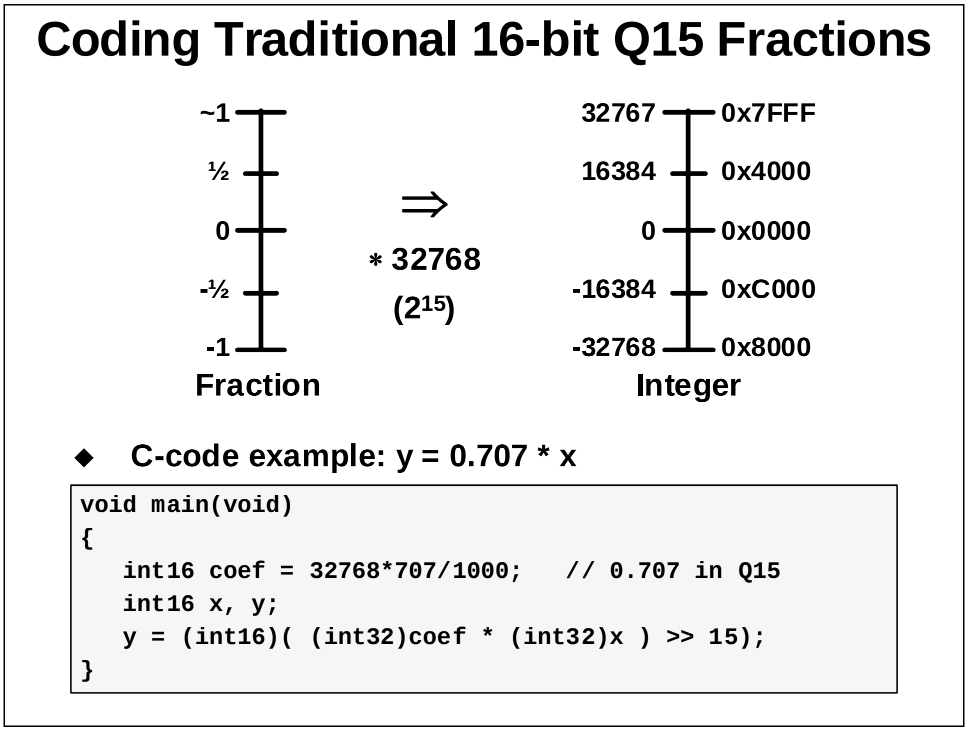
When multiplying numbers, the Q values **add**. Thus, we would (mentally) place a binary point above the sixth LSB. We can now calculate the “Q6” result more readily.

As with integers, the results are loaded low and the MSB is a sign extension of the seventh bit. If this value were loaded into the accumulator, we could store the results back to memory in a variety of ways:

- Store both low and high accumulator values back to memory. This offers maximum detail, but has the same problems as with integer multiply.
- Store only the high (or low) accumulator back to memory. This creates a potential for a memory littered with varying Q-types.
- Store the upper accumulator shifted to the left by 1. This would store values back to memory in the same Q format as the input values, and with equal precision to the inputs. How shall the left shift be performed? Here’s three methods:
  - Explicit shift (C or assembly code)
  - Shift on store (assembly code)
  - Use Product Mode shifter (assembly code)

Fraction Coding

Although COFF tools **recognize** values in integer, hex, binary, and other forms, they **understand** only integer, or non-fractional values. To use fractions within the C28x, it is necessary to describe them as though they were integers. This turns out to be a very simple trick. Consider the following number lines:



Coding Traditional 16-bit Q15 Fractions

Fraction		Integer	
~1	⇒	32767	0x7FFF
½	* 32768	16384	0x4000
0	(2^{15})	0	0x0000
-½		-16384	0xC000
-1		-32768	0x8000

◆ C-code example: y = 0.707 * x

```
void main(void)
{
   int16 coef = 32768*707/1000;   // 0.707 in Q15
   int16 x, y;
   y = (int16)( (int32)coef * (int32)x ) >> 15);
}
```

By multiplying a fraction by 32K (32768), a normalized fraction is created, which can be passed through the COFF tools as an integer. Once in the C28x, the normalized fraction looks and behaves exactly as a fraction. Thus, when using fractional constants in a C28x program, the coder first multiplies the fraction by 32768, and uses the resulting integer (rounded to the nearest whole value) to represent the fraction.

The following is a simple, but effective method for getting fractions past the assembler:

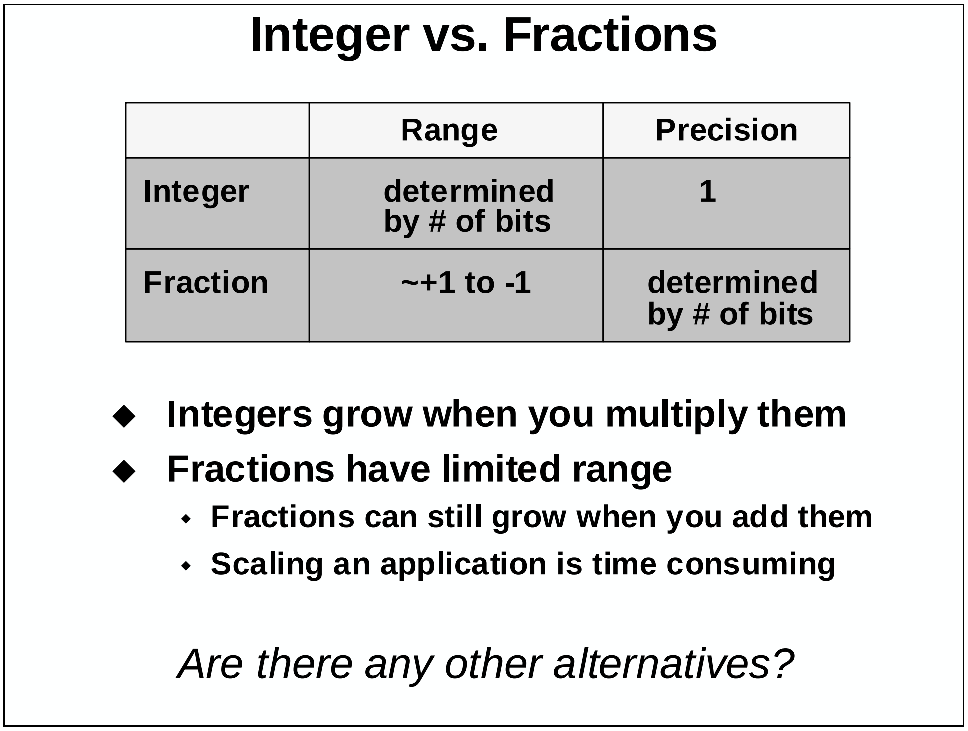
1. Express the fraction as a decimal number (drop the decimal point).
2. Multiply by 32768.
3. Divide by the proper multiple of 10 to restore the decimal position.

➢ **Examples:**

To represent 0.62:	`32768 x   62 / 100`
To represent 0.1405:	`32768 x 1405 / 10000`

This method produces a valid number accurate to 16 bits. You will not need to do the math yourself, and changing values in your code becomes rather simple.

Fractional vs. Integer Representation



Integer vs. Fractions

	Range	Precision
Integer	determined by # of bits	1
Fraction	~+1 to -1	determined by # of bits

- Integers grow when you multiply them
- Fractions have limited range
  - Fractions can still grow when you add them
  - Scaling an application is time consuming

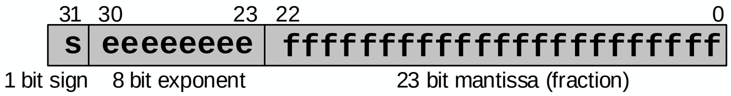
Are there any other alternatives?

The C28x accumulator, a 32-bit register, adds extra range to integer calculations, but this becomes a problem in storing the results back to 16-bit memory.

Conversely, when using fractions, the extra accumulator bits increase precision, which helps minimize accumulative errors. Since any number is accurate (at best) to ± one-half of a LSB, summing two of these values together would yield a worst case result of 1 LSB error. Four summations produce two LSBs of error. By 256 summations, eight LSBs are “noisy.” Since the accumulator holds 32 bits of information, and fractional results are stored from the **high** accumulator, the extra range of the accumulator is a major benefit in noise reduction for long sum-of-products type calculations.

Floating-Point

IEEE-754 Single Precision Floating-Point



Case 1: if $e = 255$ and $f \neq 0$, then $v = \text{NaN}$

Case 2: if $e = 255$ and $f = 0$, then $v = [(-1)^s]*\text{infinity}$

Normalized values → **Case 3:** if $0 < e < 255$, then $v = [(-1)^s]*[2^{(e-127)}]*(1.f)$

Case 4: if $e = 0$ and $f \neq 0$, then $v = [(-1)^s]*[2^{(-126)}]*(0.f)$

Case 5: if $e = 0$ and $f = 0$, then $v = [(-1)^s]*0$

Example: 0x41200000 = 0 | 100 0001 0 | 010 0000 0000 ... 0000 | b

s | e = 130 | f = 2^{-2} = 0.25

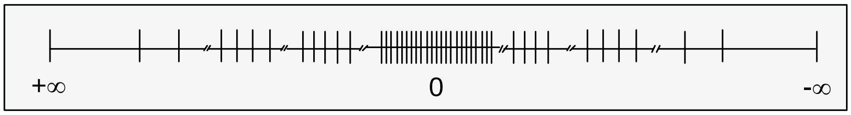
⇒ **Case 3** $v = (-1^0)*2^{(130-127)}*1.25 = 10.0$

Advantage ⇒ Exponent gives large dynamic range

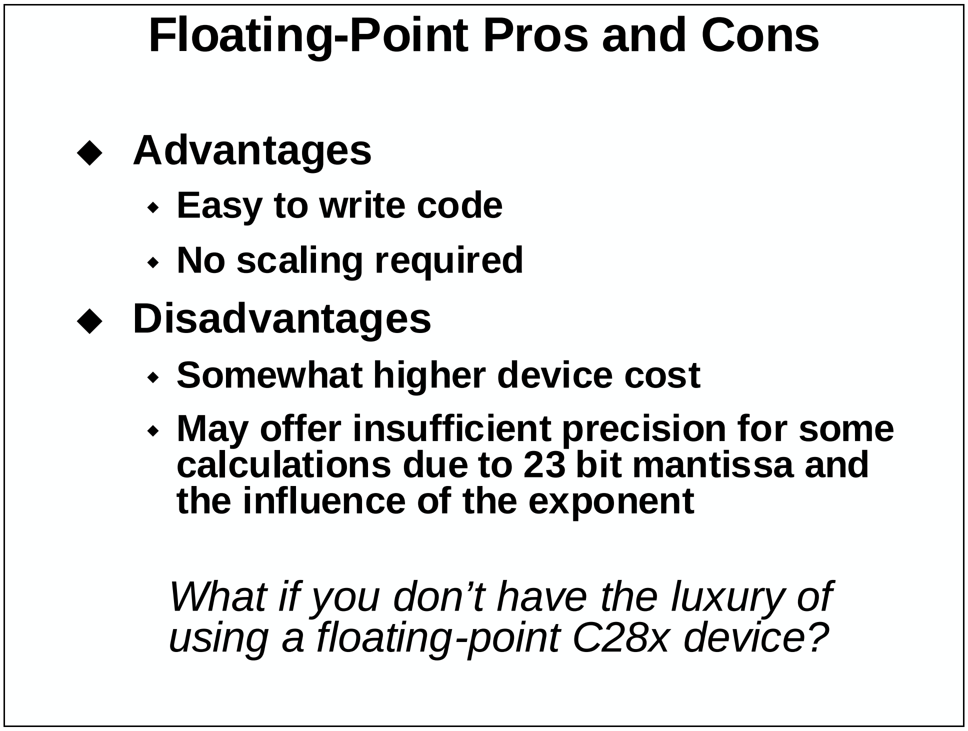
Disadvantage ⇒ Precision of a number depends on its exponent

Number Line Insight

Floating-Point:



- **Non-uniform distribution**
  - **Precision greatest near zero**
  - **Less precision the further you get from zero**



Floating-Point Pros and Cons

- **Advantages**
  - **Easy to write code**
  - **No scaling required**
- **Disadvantages**
  - **Somewhat higher device cost**
  - **May offer insufficient precision for some calculations due to 23 bit mantissa and the influence of the exponent**

What if you don’t have the luxury of using a floating-point C28x device?

IQmath

Implementing complex digital control algorithms on a Digital Signal Processor (DSP), or any other DSP capable processor, typically come across the following issues:

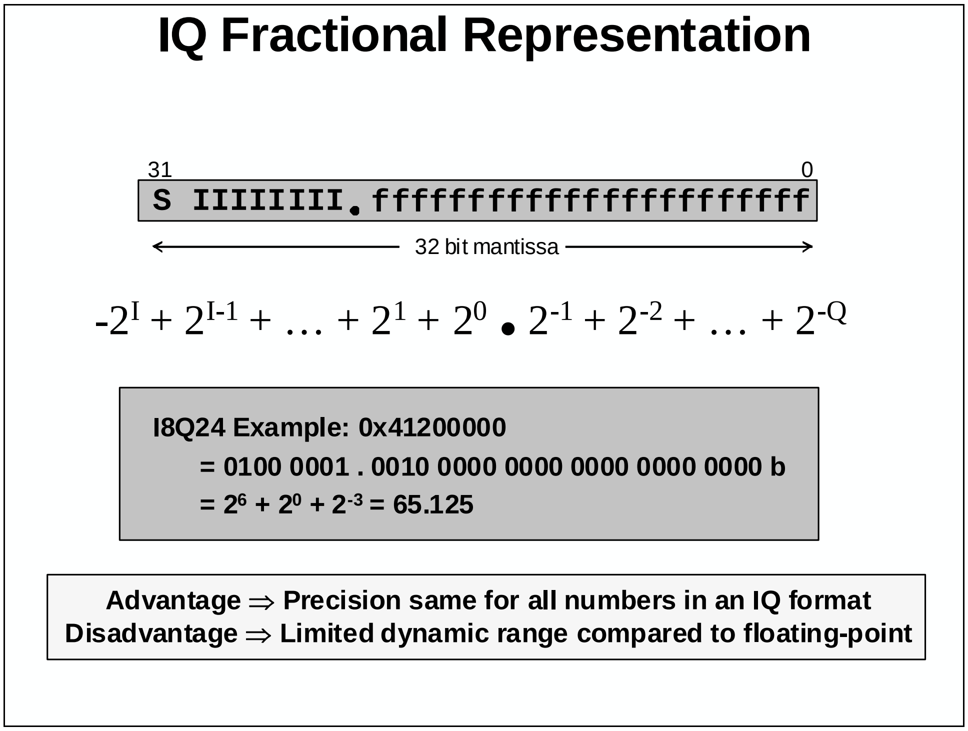
- Algorithms are typically developed using floating-point math
- Floating-point devices are more expensive than fixed-point devices
- Converting floating-point algorithms to a fixed-point device is very time consuming
- Conversion process is one way and therefore backward simulation is not always possible

The design may initially start with a simulation (i.e. MatLab) of a control algorithm, which typically would be written in floating-point math (C or C++). This algorithm can be easily ported to a floating-point device, however because of cost reasons most likely a 16-bit or 32-bit fixed-point device would be used in many target systems.

The effort and skill involved in converting a floating-point algorithm to function using a 16-bit or 32-bit fixed-point device is quite significant. A great deal of time (many days or weeks) would be needed for reformatting, scaling and coding the problem. Additionally, the final implementation typically has little resemblance to the original algorithm. Debugging is not an easy task and the code is not easy to maintain or document.

IQ Fractional Representation

A new approach to fixed-point algorithm development, termed “IQmath”, can greatly simplify the design development task. This approach can also be termed “virtual floating-point” since it looks like floating-point, but it is implemented using fixed-point techniques.



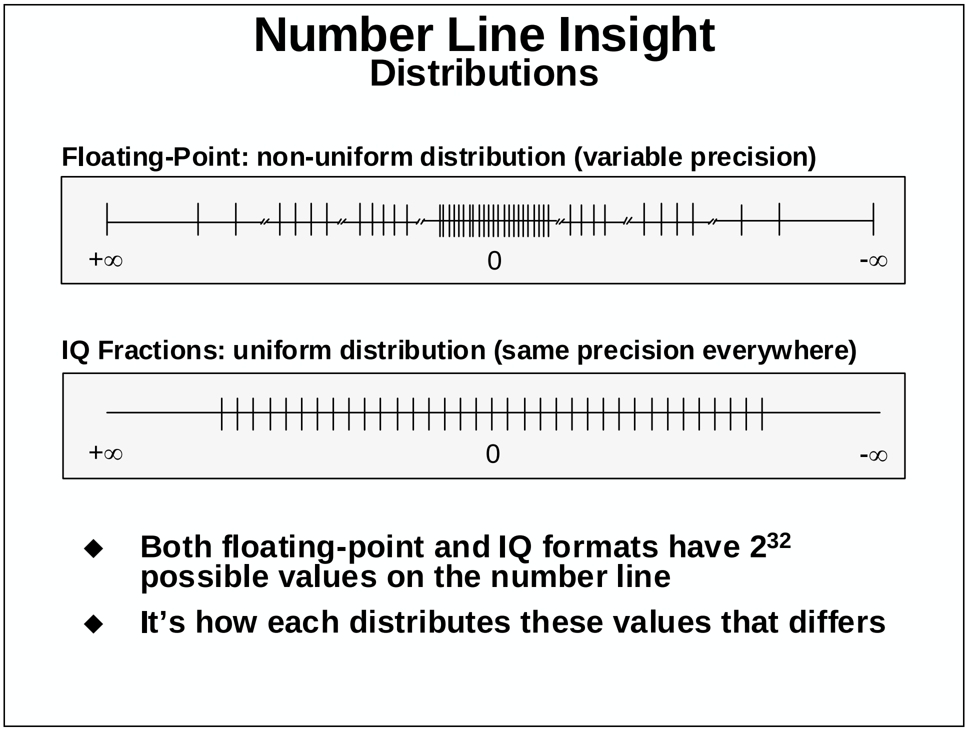
IQ Fractional Representation

$$-2^{I} + 2^{I-1} + \ldots + 2^{1} + 2^{0} \bullet 2^{-1} + 2^{-2} + \ldots + 2^{-Q}$$

I8Q24 Example: 0x41200000
= 0100 0001 . 0010 0000 0000 0000 0000 0000 b
= $2^{6} + 2^{0} + 2^{-3}$ = 65.125

Advantage ⇒ Precision same for all numbers in an IQ format
Disadvantage ⇒ Limited dynamic range compared to floating-point

The IQmath approach enables the seamless portability of code between fixed and floating-point devices. This approach is applicable to many problems that do not require a large dynamic range, such as motor or digital control applications.



Number Line Insight

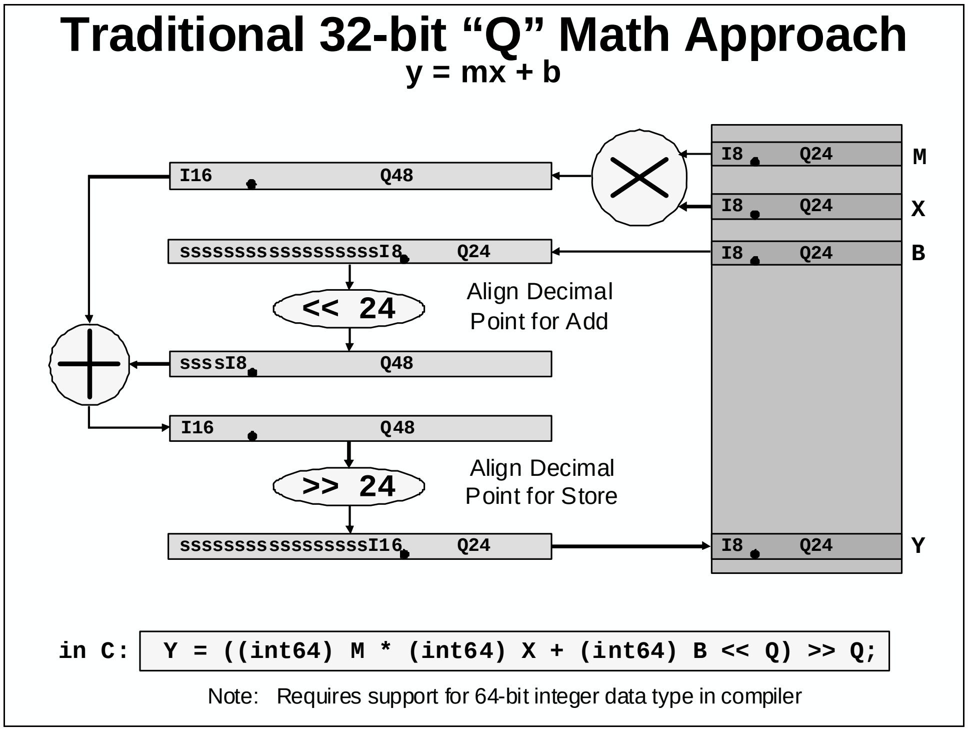
Distributions

Floating-Point: non-uniform distribution (variable precision)

IQ Fractions: uniform distribution (same precision everywhere)

- **Both floating-point and IQ formats have 2^{32} possible values on the number line**
- **It’s how each distributes these values that differs**

Traditional “Q” Math Approach



Traditional 32-bit “Q” Math Approach

y = mx + b

in C:
```
Y = ((int64) M * (int64) X + (int64) B << Q) >> Q;
```

Note: Requires support for 64-bit integer data type in compiler

The traditional approach to performing math operations, using fixed-point numerical techniques can be demonstrated using a simple linear equation example. The floating-point code for a linear equation would be:

```
    float Y, M, X, B;
    Y = M * X + B;
```

For the fixed-point implementation, assume all data is 32-bits, and that the "Q" value, or location of the binary point, is set to 24 fractional bits (Q24). The numerical range and resolution for a 32-bit Q24 number is as follows:

Q value	Min Value	Max Value	Resolution
Q24	$-2^{(32-24)}$ = -128.000 000 00	$2^{(32-24)} - (½)^{24}$ = 127.999 999 94	$(½)^{24}$ = 0.000 000 06

The C code implementation of the linear equation is:

```
    int32 Y, M, X, B; // numbers are all Q24
    Y = ((int64) M * (int64) X + (int64) B << 24) >> 24;
```

Compared to the floating-point representation, it looks quite cumbersome and has little resemblance to the floating-point equation. It is obvious why programmers prefer using floating-point math.

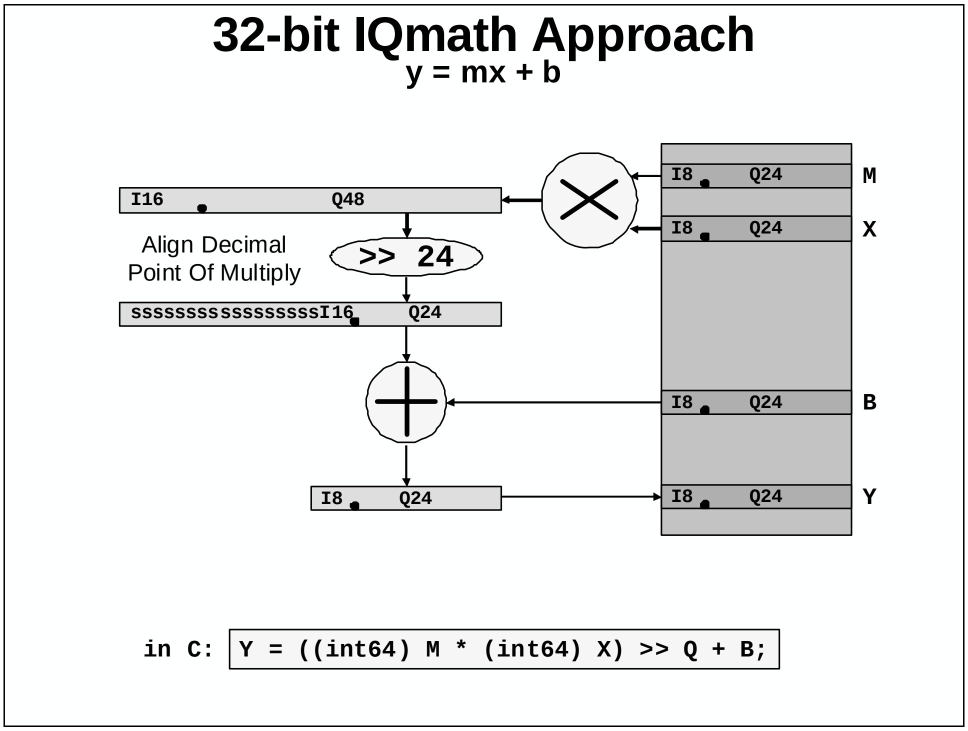
The slide shows the implementation of the equation on a processor containing hardware that can perform a 32x32 bit multiplication, 64-bit addition and 64-bit shifts (logical and arithmetic) efficiently.

The basic approach in traditional fixed-point "Q" math is to align the binary point of the operands that get added to or subtracted from the multiplication result. As shown in the slide, the multiplication of M and X (two Q24 numbers) results in a Q48 value that is stored in a 64-bit register. The value B (Q24) needs to be scaled to a Q48 number before addition to the M*X value (low order bits zero filled, high order bits sign extended). The final result is then scaled back to a Q24 number (arithmetic shift right) before storing into Y (Q24). Many programmers may be familiar with 16-bit fixed-point "Q" math that is in common use. The same example using 16-bit numbers with 15 fractional bits (Q15) would be coded as follows:

```
    int16 Y, M, X, B; // numbers are all Q15
    Y = ((int32) M * (int32) X + (int32) B << 15) >> 15;
```

In both cases, the principal methodology is the same. The binary point of the operands that get added to or subtracted from the multiplication result must be aligned.

IQmath Approach



32-bit IQmath Approach

y = mx + b

in C: `Y = ((int64) M * (int64) X) >> Q + B;`

In the "IQmath" approach, rather then scaling the operands, which get added to or subtracted from the multiplication result, we do the reverse. The multiplication result binary point is scaled back such that it aligns to the operands, which are added to or subtracted from it. The C code implementation of this is given by linear equation below:

```
int32 Y, M, X, B;
Y = ((int64) M * (int64) X) >> 24 + B;
```

The slide shows the implementation of the equation on a processor containing hardware that can perform a 32x32 bit multiply, 32-bit addition/subtraction and 64-bit logical and arithmetic shifts efficiently.

The key advantage of this approach is shown by what can then be done with the C and C++ compiler to simplify the coding of the linear equation example.

Let's take an additional step and create a multiply function in C that performs the following operation:

```
int32 _IQ24mpy(int32 M, int32 X) { return ((int64) M * (int64) X) >> 24; }
```

The linear equation can then be written as follows:

```
Y = _IQ24mpy(M , X) + B;
```

Already we can see a marked improvement in the readability of the linear equation.

Using the operator overloading features of C++, we can overload the multiplication operand "*" such that when a particular data type is encountered, it will automatically implement the scaled multiply operation. Let's define a data type called "iq" and assign the linear variables to this data type:

```
iq Y, M, X, B // numbers are all Q24
```

The overloading of the multiply operand in C++ can be defined as follows:

```
iq operator*(const iq &M, const iq &X){return((int64)M*(int64) X) >> 24;}
```

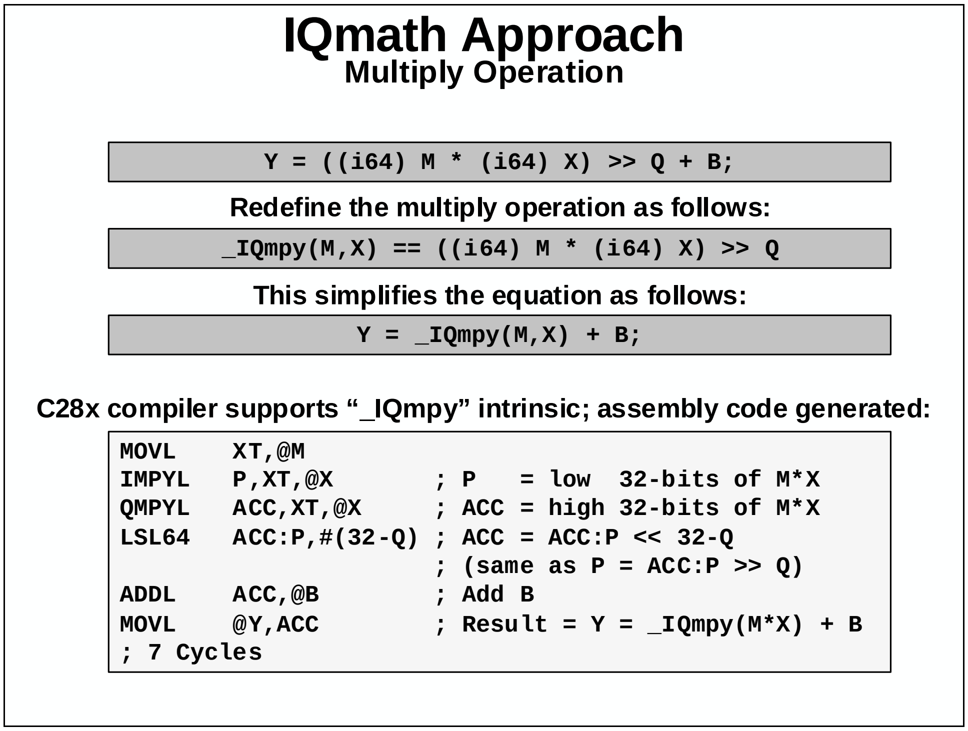
Then the linear equation, in C++, becomes:

```
Y = M * X + B;
```

This final equation looks identical to the floating-point representation. It looks "natural". The four approaches are summarized in the table below:

Math Implementations	**Linear Equation Code**
32-bit floating-point math in C	Y = M * X + B;
32-bit fixed-point "Q" math in C	Y = ((int64) M * (int64) X) + (int64) B << 24) >> 24;
32-bit IQmath in C	Y = _IQ24mpy(M, X) + B;
32-bit IQmath in C++	Y = M * X + B;

Essentially, the mathematical approach of scaling the multiplier operand enables a cleaner and a more "natural" approach to coding fixed-point problems. For want of a better term, we call this approach "IQmath" or can also be described as "virtual floating-point".



IQmath Approach

Multiply Operation

```
Y = ((i64) M * (i64) X) >> Q + B;
```

Redefine the multiply operation as follows:

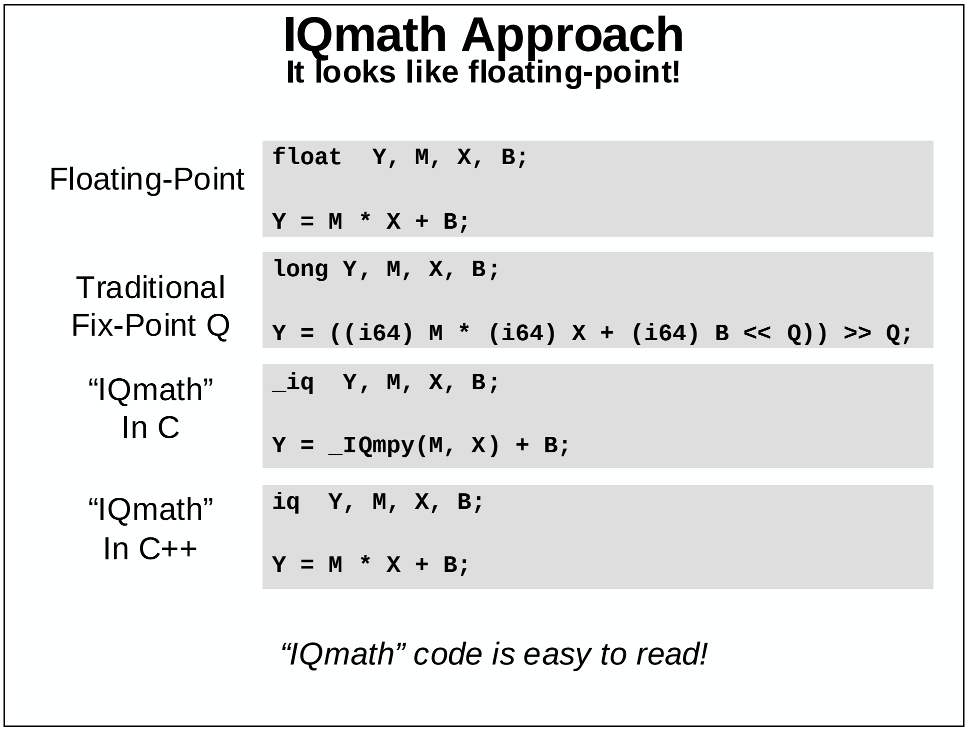
```
_IQmpy(M,X) == ((i64) M * (i64) X) >> Q
```

This simplifies the equation as follows:

```
Y = _IQmpy(M,X) + B;
```

C28x compiler supports "_IQmpy" intrinsic; assembly code generated:

```
MOVL    XT,@M
IMPYL   P,XT,@X       ; P   = low  32-bits of M*X
QMPYL   ACC,XT,@X     ; ACC = high 32-bits of M*X
LSL64   ACC:P,#(32-Q) ; ACC = ACC:P << 32-Q
                      ; (same as P = ACC:P >> Q)
ADDL    ACC,@B        ; Add B
MOVL    @Y,ACC        ; Result = Y = _IQmpy(M*X) + B
; 7 Cycles
```



IQmath Approach

It looks like floating-point!

Floating-Point

```
float  Y, M, X, B;

Y = M * X + B;
```

Traditional
Fix-Point Q

```
long Y, M, X, B;

Y = ((i64) M * (i64) X + (i64) B << Q)) >> Q;
```

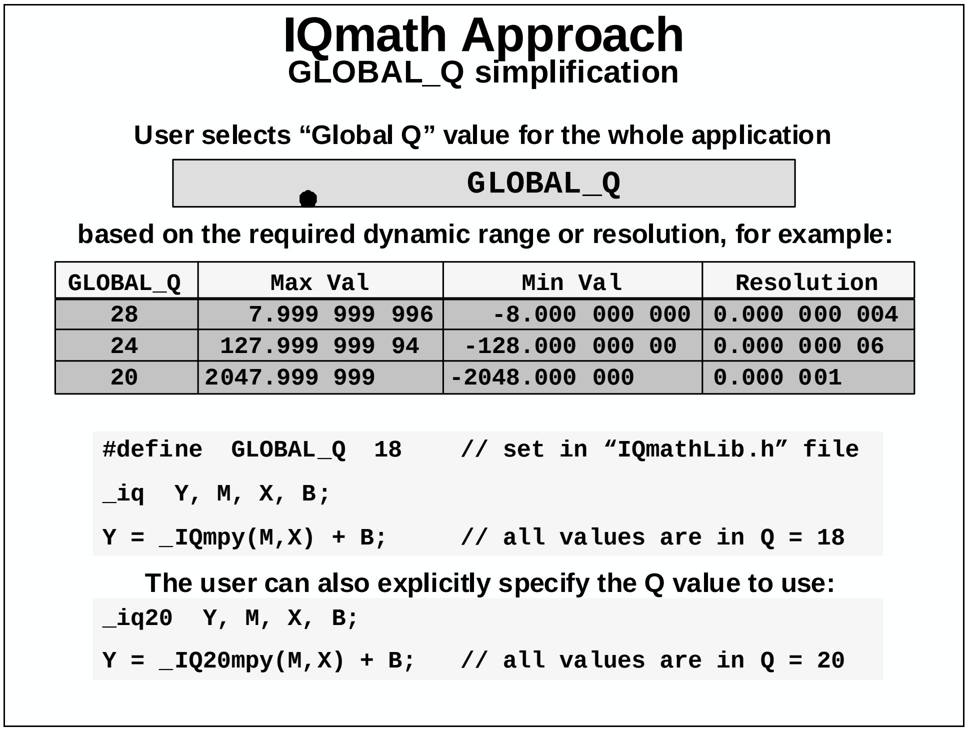
"IQmath"
In C

```
_iq  Y, M, X, B;

Y = _IQmpy(M, X) + B;
```

"IQmath"
In C++

```
iq  Y, M, X, B;

Y = M * X + B;
```

"IQmath" code is easy to read!



IQmath Approach

GLOBAL_Q simplification

User selects "Global Q" value for the whole application

`GLOBAL_Q`

based on the required dynamic range or resolution, for example:

GLOBAL_Q	Max Val	Min Val	Resolution
28	7.999 999 996	-8.000 000 000	0.000 000 004
24	127.999 999 94	-128.000 000 00	0.000 000 06
20	2047.999 999	-2048.000 000	0.000 001

```
#define  GLOBAL_Q  18    // set in "IQmathLib.h" file
_iq  Y, M, X, B;
Y = _IQmpy(M,X) + B;     // all values are in Q = 18
```

The user can also explicitly specify the Q value to use:

```
_iq20  Y, M, X, B;
Y = _IQ20mpy(M,X) + B;   // all values are in Q = 20
```

The basic "IQmath" approach was adopted in the creation of a standard math library for the Texas Instruments TMS320C28x DSP fixed-point processor. This processor contains efficient hardware for performing 32x32 bit multiply, 64-bit shifts (logical and arithmetic) and 32-bit add/subtract operations, which are ideally suited for 32 bit "IQmath".

Some enhancements were made to the basic "IQmath" approach to improve flexibility. They are:

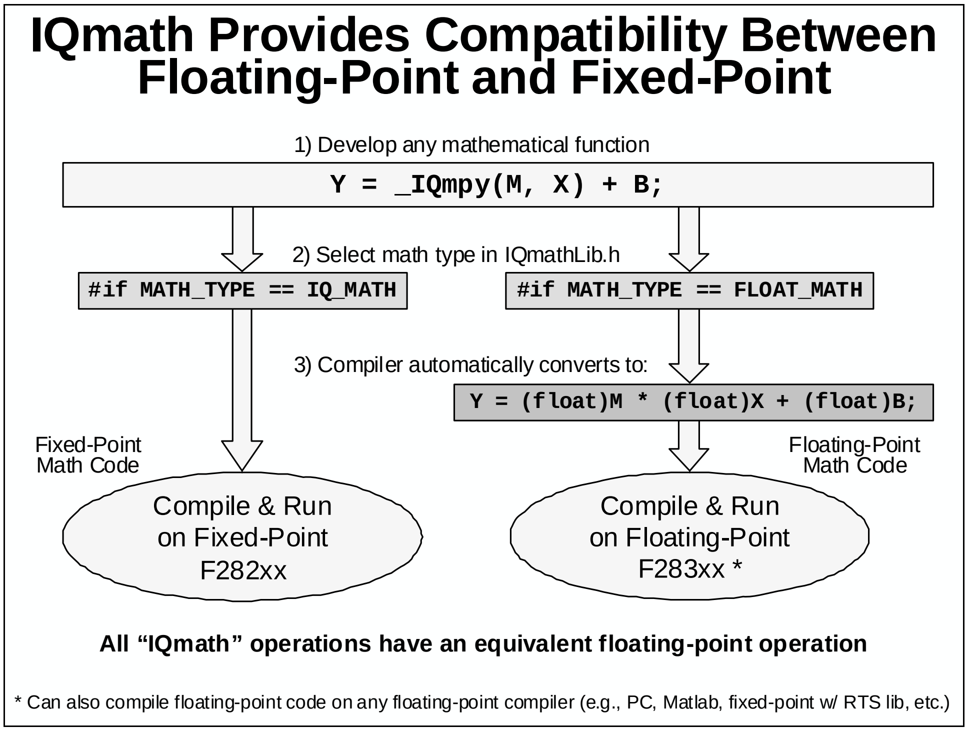
Setting of GLOBAL_Q Parameter Value: Depending on the application, the amount of numerical resolution or dynamic range required may vary. In the linear equation example, we used a Q value of 24 (Q24). There is no reason why any value of Q can't be used. In the "IQmath" library, the user can set a GLOBAL_Q parameter, with a range of 1 to 30 (Q1 to Q30). All functions used in the program will use this GLOBAL_Q value. For example:

```
#define GLOBAL_Q 18
Y = _IQmpy(M, X) + B; // all values use GLOBAL_Q = 18
```

If, for some reason a particular function or equation requires a different resolution, then the user has the option to implicitly specify the Q value for the operation. For example:

```
Y = _IQ23mpy(M,X) + B; // all values use Q23, including B and Y
```

The Q value must be consistent for all expressions in the same line of code.



IQmath Provides Compatibility Between Floating-Point and Fixed-Point

1) Develop any mathematical function

```
Y = _IQmpy(M, X) + B;
```

2) Select math type in IQmathLib.h

```
#if MATH_TYPE == IQ_MATH
```

```
#if MATH_TYPE == FLOAT_MATH
```

3) Compiler automatically converts to:

```
Y = (float)M * (float)X + (float)B;
```

Fixed-Point Math Code
Compile & Run on Fixed-Point F282xx
Floating-Point Math Code
Compile & Run on Floating-Point F283xx *

All “IQmath” operations have an equivalent floating-point operation

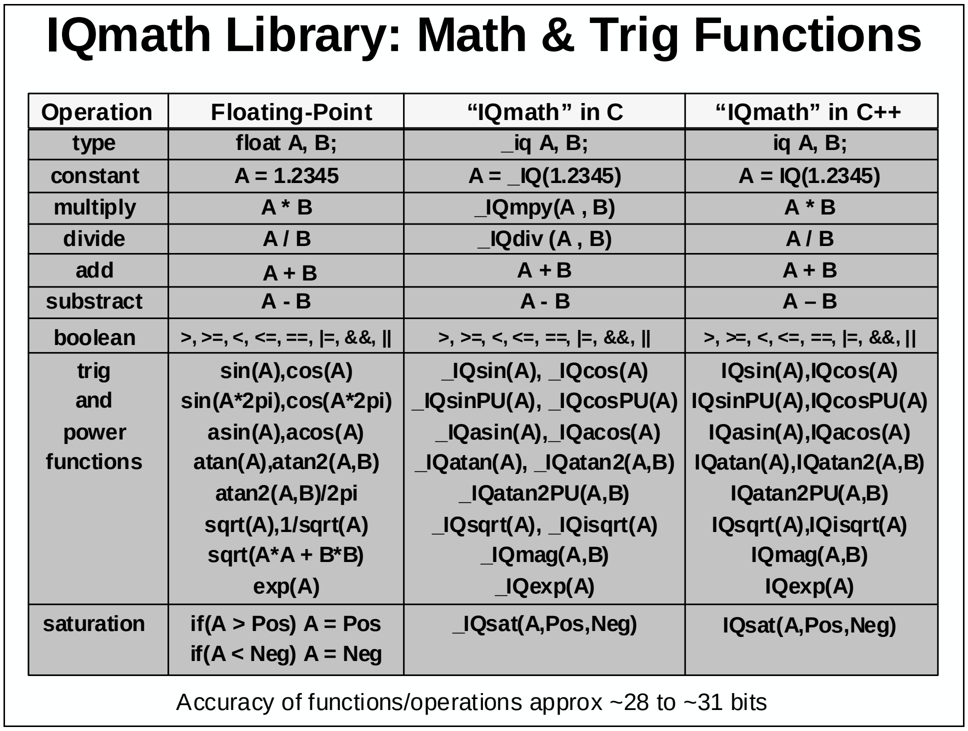
* Can also compile floating-point code on any floating-point compiler (e.g., PC, Matlab, fixed-point w/ RTS lib, etc.)

Selecting FLOAT_MATH or IQ_MATH Mode: As was highlighted in the introduction, we would ideally like to be able to have a single source code that can execute on a floating-point or fixed-point target device simply by recompiling the code. The "IQmath" library supports this by setting a mode, which selects either IQ_MATH or FLOAT_MATH. This operation is performed by simply redefining the function in a header file. For example:

```
#if MATH_TYPE == IQ_MATH
#define _IQmpy(M , X) _IQmpy(M , X)
#elseif MATH_TYPE == FLOAT_MATH
#define _IQmpy(M , X) (float) M * (float) X
#endif
```

Essentially, the programmer writes the code using the "IQmath" library functions and the code can be compiled for floating-point or "IQmath" operations.

IQmath Library



IQmath Library: Math & Trig Functions

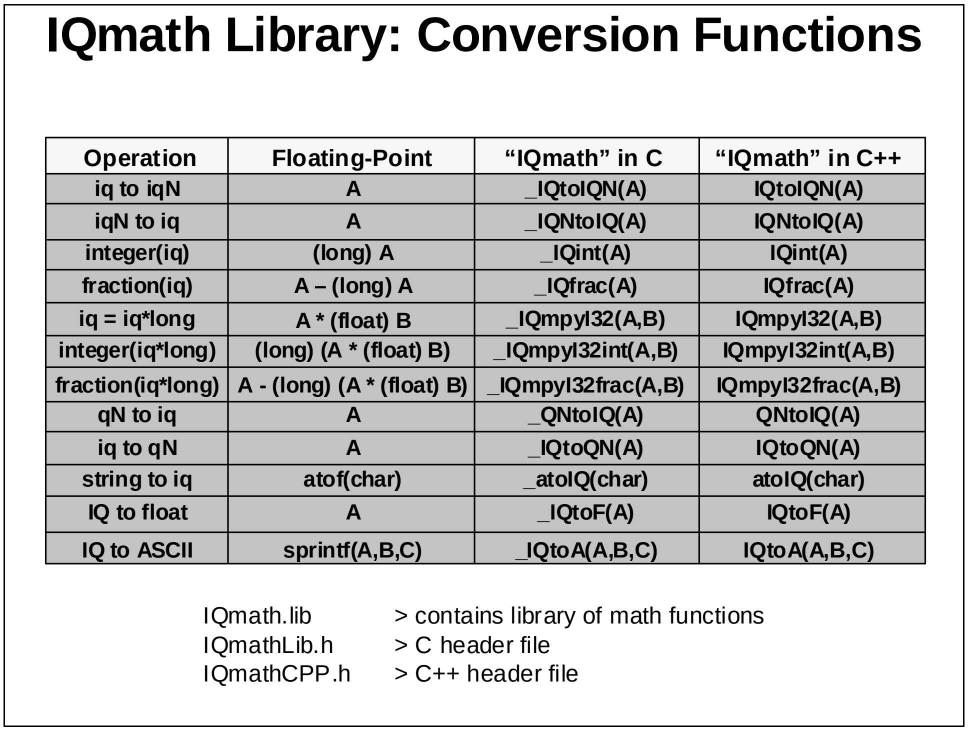
Operation	Floating-Point	"IQmath" in C	"IQmath" in C++
type	float A, B;	_iq A, B;	iq A, B;
constant	A = 1.2345	A = _IQ(1.2345)	A = IQ(1.2345)
multiply	A * B	_IQmpy(A , B)	A * B
divide	A / B	_IQdiv (A , B)	A / B
add	A + B	A + B	A + B
substract	A - B	A - B	A – B
boolean	>, >=, <, <=, ==, |=, &&, ||	>, >=, <, <=, ==, |=, &&, ||	>, >=, <, <=, ==, |=, &&, ||

trig
and
power
functions
	sin(A),cos(A)
	sin(A*2pi),cos(A*2pi)
	asin(A),acos(A)
	atan(A),atan2(A,B)
	atan2(A,B)/2pi
	sqrt(A),1/sqrt(A)
	sqrt(A*A + B*B)
	exp(A)
		_IQsin(A), _IQcos(A)
		_IQsinPU(A), _IQcosPU(A)
		_IQasin(A),_IQacos(A)
		_IQatan(A), _IQatan2(A,B)
		_IQatan2PU(A,B)
		_IQsqrt(A), _IQisqrt(A)
		_IQmag(A,B)
		_IQexp(A)
			IQsin(A),IQcos(A)
			IQsinPU(A),IQcosPU(A)
			IQasin(A),IQacos(A)
			IQatan(A),IQatan2(A,B)
			IQatan2PU(A,B)
			IQsqrt(A),IQisqrt(A)
			IQmag(A,B)
			IQexp(A)

saturation	if(A > Pos) A = Pos
	if(A < Neg) A = Neg
		_IQsat(A,Pos,Neg)	IQsat(A,Pos,Neg)

Accuracy of functions/operations approx ~28 to ~31 bits

Additionally, the "IQmath" library contains DSP library modules for filters (FIR & IIR) and Fast Fourier Transforms (FFT & IFFT).



IQmath Library: Conversion Functions

Operation	Floating-Point	"IQmath" in C	"IQmath" in C++
iq to iqN	A	_IQtoIQN(A)	IQtoIQN(A)
iqN to iq	A	_IQNtoIQ(A)	IQNtoIQ(A)
integer(iq)	(long) A	_IQint(A)	IQint(A)
fraction(iq)	A – (long) A	_IQfrac(A)	IQfrac(A)
iq = iq*long	A * (float) B	_IQmpyI32(A,B)	IQmpyI32(A,B)
integer(iq*long)	(long) (A * (float) B)	_IQmpyI32int(A,B)	IQmpyI32int(A,B)
fraction(iq*long)	A - (long) (A * (float) B)	_IQmpyI32frac(A,B)	IQmpyI32frac(A,B)
qN to iq	A	_QNtoIQ(A)	QNtoIQ(A)
iq to qN	A	_IQtoQN(A)	IQtoQN(A)
string to iq	atof(char)	_atoIQ(char)	atoIQ(char)
IQ to float	A	_IQtoF(A)	IQtoF(A)
IQ to ASCII	sprintf(A,B,C)	_IQtoA(A,B,C)	IQtoA(A,B,C)

IQmath.lib	> contains library of math functions
IQmathLib.h	> C header file
IQmathCPP.h	> C++ header file

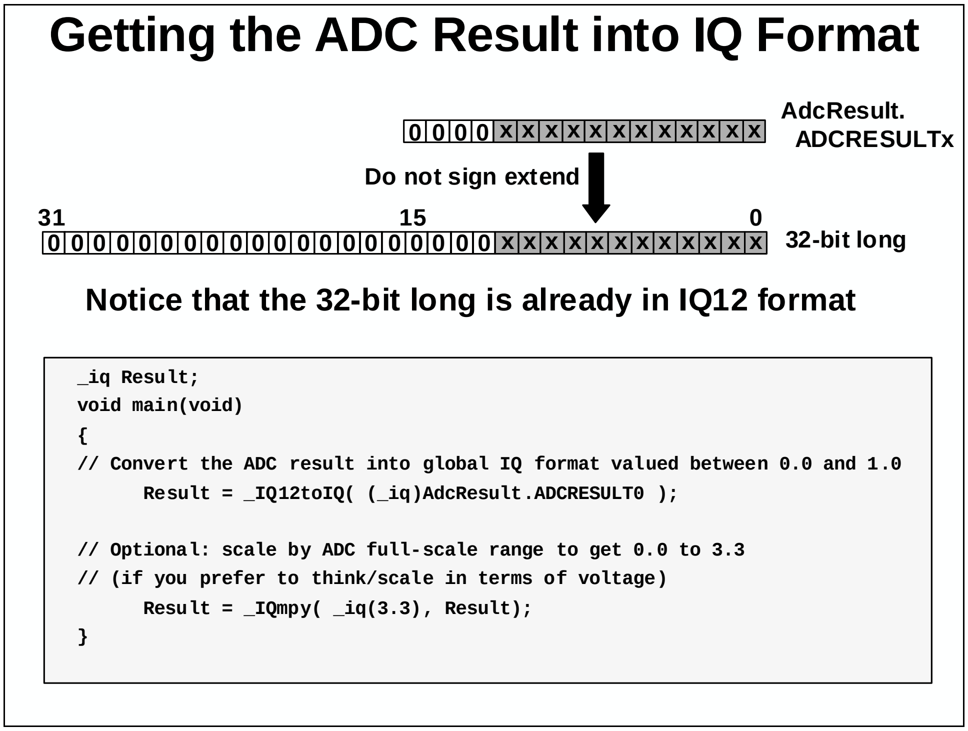
16 vs. 32 Bits

The "IQmath" approach could also be used on 16-bit numbers and for many problems, this is sufficient resolution. However, in many control cases, the user needs to use many different "Q" values to accommodate the limited resolution of a 16-bit number.

With DSP devices like the TMS320C28x processor, which can perform 16-bit and 32-bit math with equal efficiency, the choice becomes more of productivity (time to market). Why bother spending a whole lot of time trying to code using 16-bit numbers when you can simply use 32-bit numbers, pick one value of "Q" that will accommodate all cases and not worry about spending too much time optimizing.

Of course there is a concern on data RAM usage if numbers that could be represented in 16 bits all use 32 bits. This is becoming less of an issue in today's processors because of the finer technology used and the amount of RAM that can be cheaply integrated. However, in many cases, this problem can be mitigated by performing intermediate calculations using 32-bit numbers and converting the input from 16 to 32 bits and converting the output back to 16 bits before storing the final results. In many problems, it is the intermediate calculations that require additional accuracy to avoid quantization problems.

Converting ADC Results into IQ Format



Getting the ADC Result into IQ Format

AdcResult.
ADCRESULTx
0 0 0 0 x x x x x x x x x x x x
Do not sign extend
31
15
0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 x x x x x x x x x x x x
32-bit long

Notice that the 32-bit long is already in IQ12 format

```
_iq Result;
void main(void)
{
// Convert the ADC result into global IQ format valued between 0.0 and 1.0
      Result = _IQ12toIQ( (_iq)AdcResult.ADCRESULT0 );

// Optional: scale by ADC full-scale range to get 0.0 to 3.3
// (if you prefer to think/scale in terms of voltage)
      Result = _IQmpy( _iq(3.3), Result);
}
```

As you may recall, the converted values of the ADC are placed in the lower 12 bits of the ADCRESULT0 register. Before these values are filtered using the IQmath library, they need to to be put into the IQ format as a 32-bit long. For uni-polar ADC inputs (i.e., 0 to 3.3 V inputs), a conversion to global IQ format can be achieved with:

```
IQresult_unipolar = _IQmpy(_IQ(3.3),_IQ12toIQ((_iq) AdcResult.ADCRESULT0));
```

How can we modify the above to recover bi-polar inputs, for example +-1.65 volts? One could do the following to offset the +1.65V analog biasing applied to the ADC input:

```
IQresult_bipolar =
_IQmpy(_IQ(3.3),_IQ12toIQ((_iq) AdcResult.ADCRESULT0)) - _IQ(1.65);
```

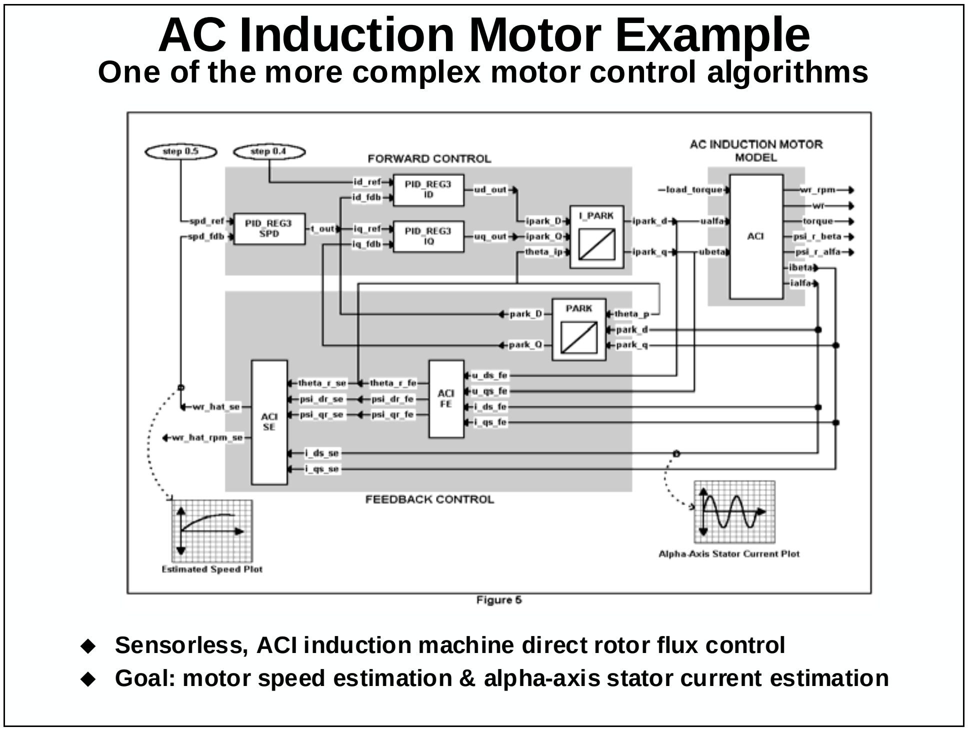
However, one can see that the largest intermediate value the equation above could reach is 3.3. This means that it cannot be used with an IQ data type of IQ30 (IQ30 range is -2 < x < ~2). Since the IQmath library supports IQ types from IQ1 to IQ30, this could be an issue in some applications.

The following clever approach supports IQ types from IQ1 to IQ30:

```
IQresult_bipolar =
_IQmpy(_IQ(1.65),_IQ15toIQ((_iq) ((int16) (AdcResult.ADCRESULT0 ^
0x8000))));
```

The largest intermediate value that this equation could reach is 1.65. Therefore, IQ30 is easily supported.

AC Induction Motor Example



AC Induction Motor Example

One of the more complex motor control algorithms

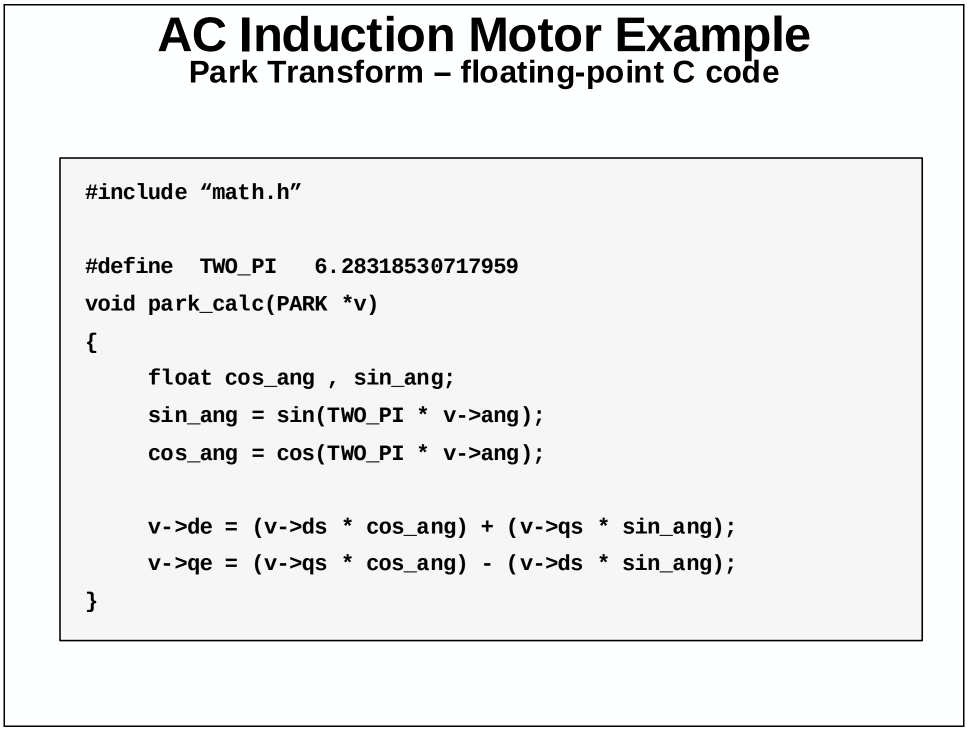
Figure 5

- **Sensorless, ACI induction machine direct rotor flux control**
- **Goal: motor speed estimation & alpha-axis stator current estimation**

The "IQmath" approach is ideally suited for applications where a large numerical dynamic range is not required. Motor control is an example of such an application (audio and communication algorithms are other applications). As an example, the IQmath approach has been applied to the sensor-less direct field control of an AC induction motor. This is probably one of the most challenging motor control problems and as will be shown later, requires numerical accuracy greater then 16-bits in the control calculations.

The above slide is a block diagram representation of the key control blocks and their interconnections. Essentially this system implements a "Forward Control" block for controlling the d-q axis motor current using PID controllers and a "Feedback Control" block using back emf's integration with compensated voltage from current model for estimating rotor flux based on current and voltage measurements. The motor speed is simply estimated from rotor flux differentiation and open-loop slip computation. The system was initially implemented on a "Simulator Test Bench" which uses a simulation of an "AC Induction Motor Model" in place of a real motor. Once working, the system was then tested using a real motor on an appropriate hardware platform.

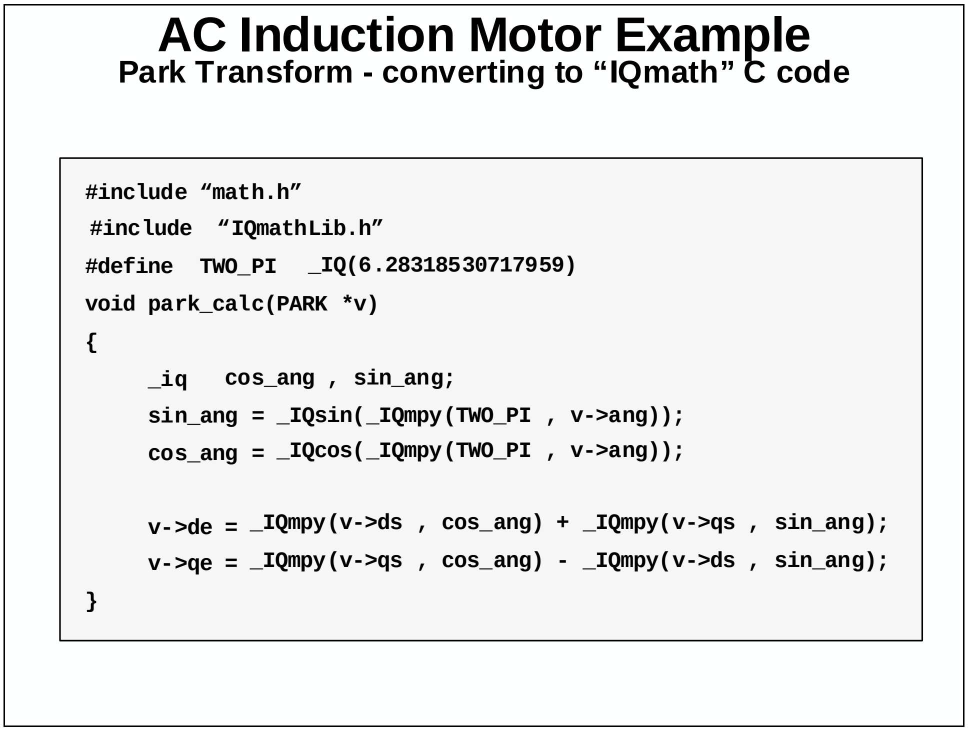
Each individual block shown in the slide exists as a stand-alone C/C++ module, which can be interconnected to form the complete control system. This modular approach allows reusability and portability of the code. The next few slides show the coding of one particular block, PARK Transform, using floating-point and "IQmath" approaches in C:



AC Induction Motor Example

Park Transform – floating-point C code

```
#include “math.h”

#define  TWO_PI   6.28318530717959
void park_calc(PARK *v)
{
     float cos_ang , sin_ang;
     sin_ang = sin(TWO_PI * v->ang);
     cos_ang = cos(TWO_PI * v->ang);

     v->de = (v->ds * cos_ang) + (v->qs * sin_ang);
     v->qe = (v->qs * cos_ang) - (v->ds * sin_ang);
}
```



AC Induction Motor Example

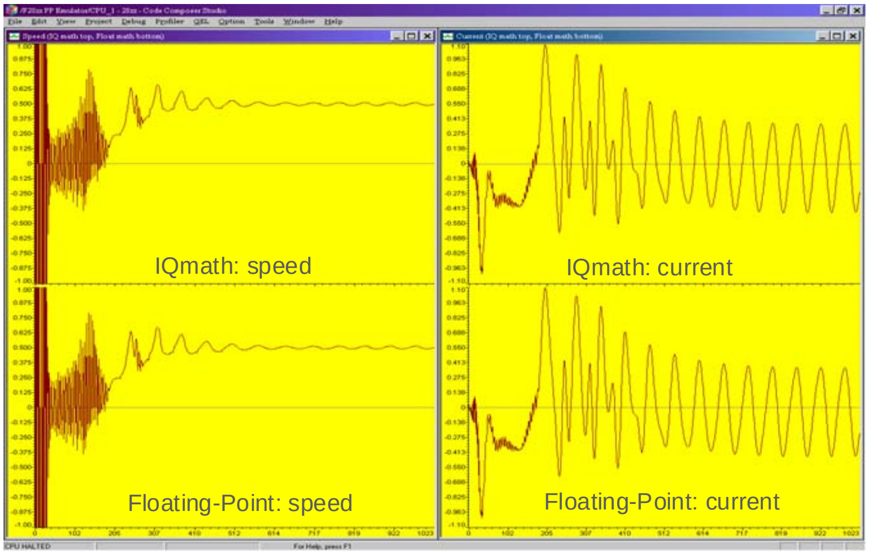
Park Transform - converting to “IQmath” C code

```
#include “math.h”
#include  “IQmathLib.h”
#define  TWO_PI   _IQ(6.28318530717959)
void park_calc(PARK *v)
{
     _iq   cos_ang , sin_ang;
     sin_ang = _IQsin(_IQmpy(TWO_PI , v->ang));
     cos_ang = _IQcos(_IQmpy(TWO_PI , v->ang));

     v->de = _IQmpy(v->ds , cos_ang) + _IQmpy(v->qs , sin_ang);
     v->qe = _IQmpy(v->qs , cos_ang) - _IQmpy(v->ds , sin_ang);
}
```

The complete system was coded using "IQmath". Based on analysis of coefficients in the system, the largest coefficient had a value of 33.3333. This indicated that a minimum dynamic range of 7 bits (+/-64 range) was required. Therefore, this translated to a GLOBAL_Q value of 32-7 = 25 (Q25). Just to be safe, the initial simulation runs were conducted with GLOBAL_Q = 24 (Q24) value. The plots start from a step change in reference speed from 0.0 to 0.5 and 1024 samples are taken.

AC Induction Motor Example

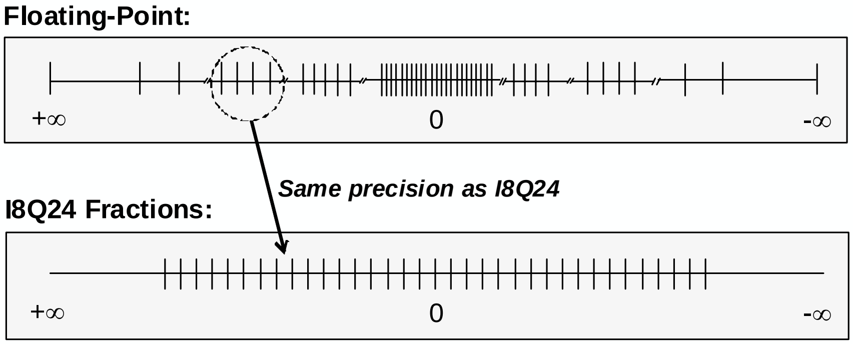
GLOBAL_Q = 24, system stable



The speed eventually settles to the desired reference value and the stator current exhibits a clean and stable oscillation. The block diagram slide shows at which points in the control system the plots are taken from.

What's Happening Here?

Equal Precision in the Computation Region



Floating-Point:

Same precision as I8Q24

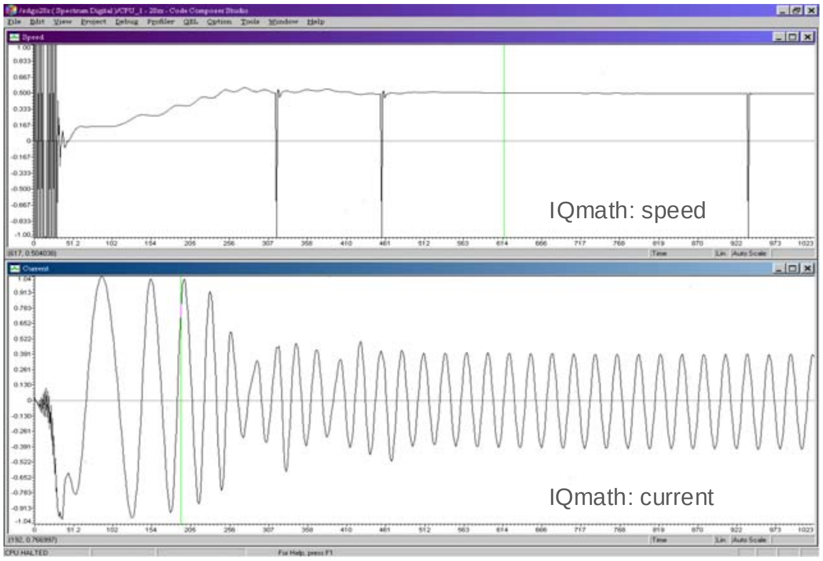
I8Q24 Fractions:

In the region where these particular computations occur, the precision of single-precision floating-point just happens to equal the precision of the I8Q24 format.

So, both produce similar results!

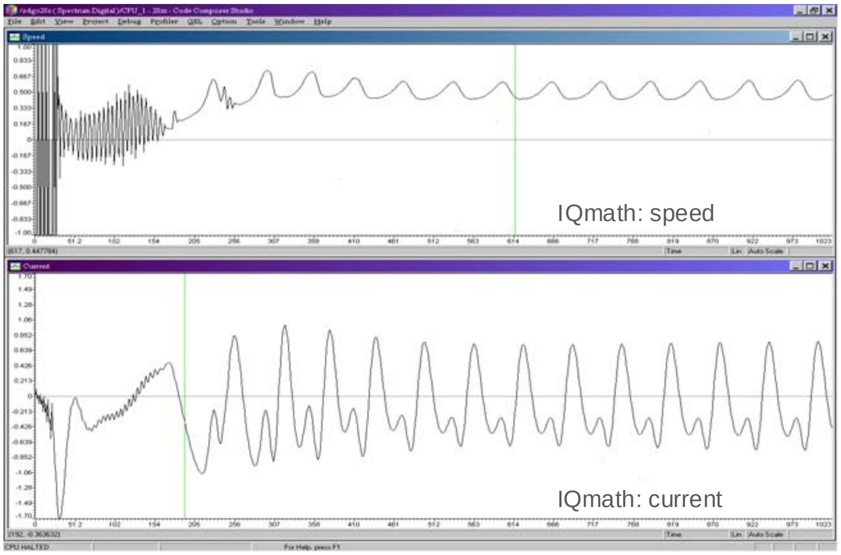
AC Induction Motor Example

GLOBAL_Q = 27, system unstable

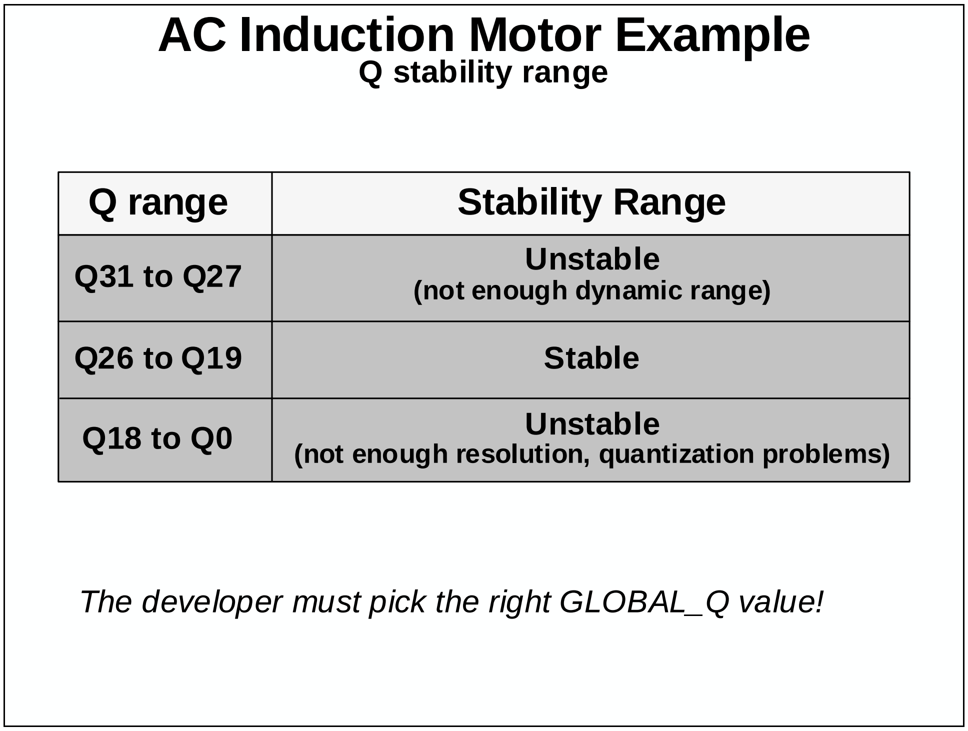


AC Induction Motor Example

GLOBAL_Q = 16, system unstable



With the ability to select the GLOBAL_Q value for all calculations in the "IQmath", an experiment was conducted to see what maximum and minimum Q value the system could tolerate before it became unstable. The results are tabulated in the slide below:



AC Induction Motor Example

Q stability range

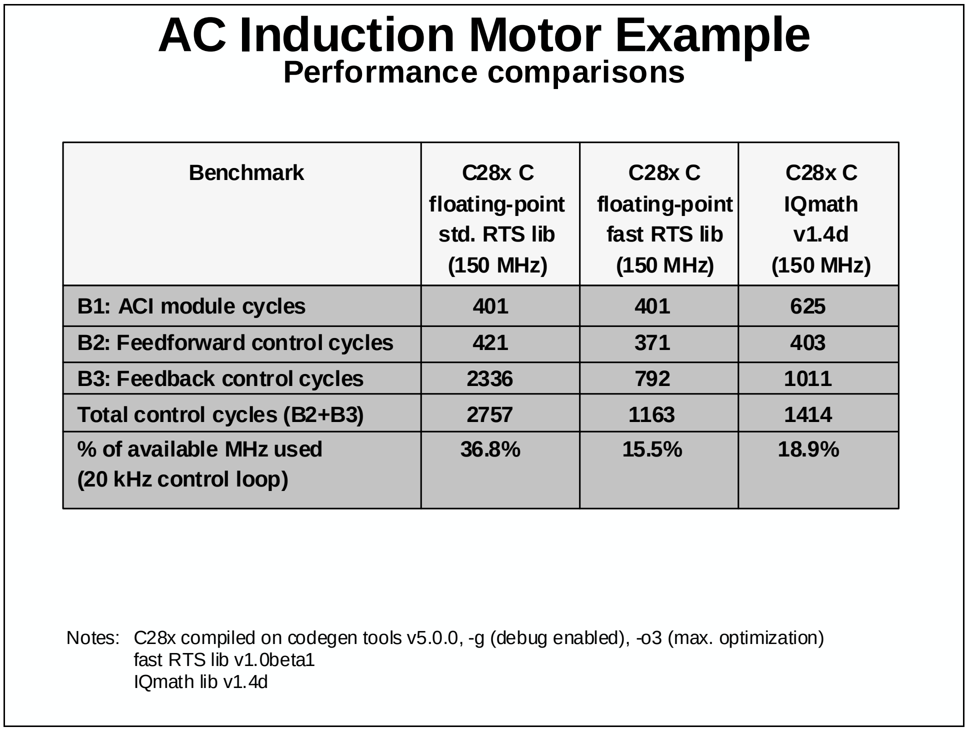
Q range	Stability Range
Q31 to Q27	**Unstable** **(not enough dynamic range)**
Q26 to Q19	**Stable**
Q18 to Q0	**Unstable** **(not enough resolution, quantization problems)**

The developer must pick the right GLOBAL_Q value!

The above indicates that, the AC induction motor system that we simulated requires a minimum of 7 bits of dynamic range (+/-64) and requires a minimum of 19 bits of numerical resolution (+/-0.000002). This confirms our initial analysis that the largest coefficient value being 33.33333 required a minimum dynamic range of 7 bits. As a general guideline, users using IQmath should examine the largest coefficient used in the equations and this would be a good starting point for setting the initial GLOBAL_Q value. Then, through simulation or experimentation, the user can reduce the GLOBAL_Q until the system resolution starts to cause instability or performance degradation. The user then has a maximum and minimum limit and a safe approach is to pick a midpoint.

What the above analysis also confirms is that this particular problem does require some calculations to be performed using greater then 16 bit precision. The above example requires a minimum of 7 + 19 = 26 bits of numerical accuracy for some parts of the calculations. Hence, if one was implementing the AC induction motor control algorithm using a 16 bit fixed-point DSP, it would require the implementation of higher precision math for certain portions. This would take more cycles and programming effort.

The great benefit of using GLOBAL_Q is that the user does not necessarily need to go into details to assign an individual Q for each variable in a whole system, as is typically done in conventional fixed-point programming. This is time consuming work. By using 32-bit resolution and the "IQmath" approach, the user can easily evaluate the overall resolution and quickly implement a typical digital motor control application without quantization problems.



AC Induction Motor Example

Performance comparisons

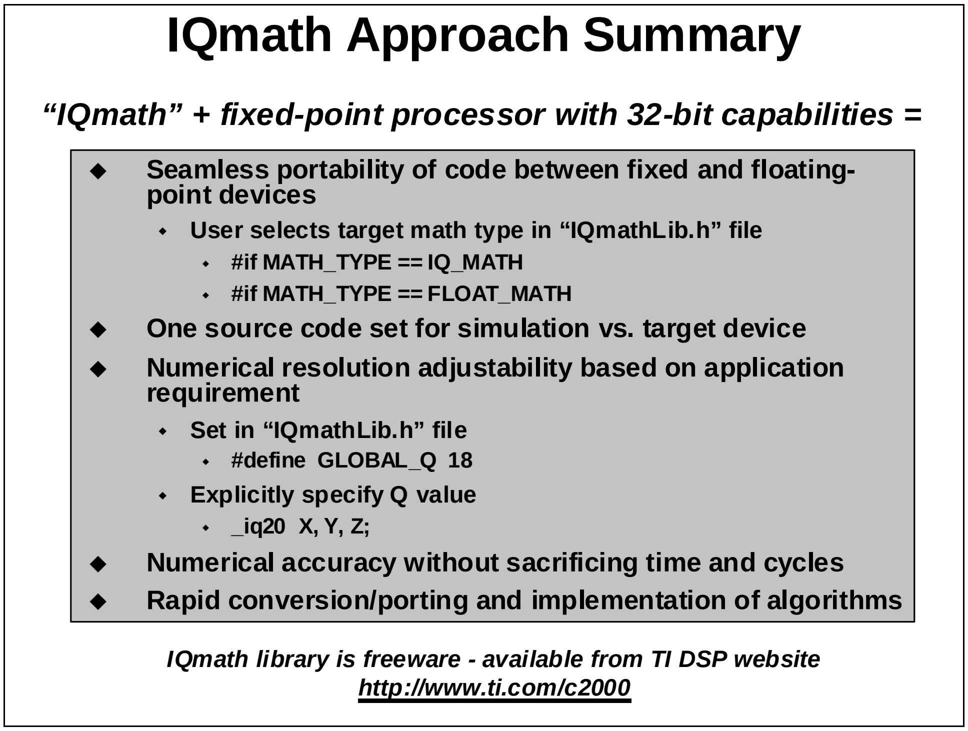
Benchmark	C28x C floating-point std. RTS lib (150 MHz)	C28x C floating-point fast RTS lib (150 MHz)	C28x C IQmath v1.4d (150 MHz)
B1: ACI module cycles	401	401	625
B2: Feedforward control cycles	421	371	403
B3: Feedback control cycles	2336	792	1011
Total control cycles (B2+B3)	2757	1163	1414
% of available MHz used (20 kHz control loop)	36.8%	15.5%	18.9%

Notes: C28x compiled on codegen tools v5.0.0, -g (debug enabled), -o3 (max. optimization)
fast RTS lib v1.0beta1
IQmath lib v1.4d

Using the profiling capabilities of the respective DSP tools, the table above summarizes the number of cycles and code size of the forward and feedback control blocks.

The MIPS used is based on a system sampling frequency of 20 kHz, which is typical of such systems.

IQmath Summary



IQmath Approach Summary

"IQmath" + fixed-point processor with 32-bit capabilities =

- **Seamless portability of code between fixed and floating-point devices**
  - **User selects target math type in "IQmathLib.h" file**
    - **#if MATH_TYPE == IQ_MATH**
    - **#if MATH_TYPE == FLOAT_MATH**
- **One source code set for simulation vs. target device**
- **Numerical resolution adjustability based on application requirement**
  - **Set in "IQmathLib.h" file**
    - **#define GLOBAL_Q 18**
  - **Explicitly specify Q value**
    - **_iq20 X, Y, Z;**
- **Numerical accuracy without sacrificing time and cycles**
- **Rapid conversion/porting and implementation of algorithms**

IQmath library is freeware - available from TI DSP website
http://www.ti.com/c2000

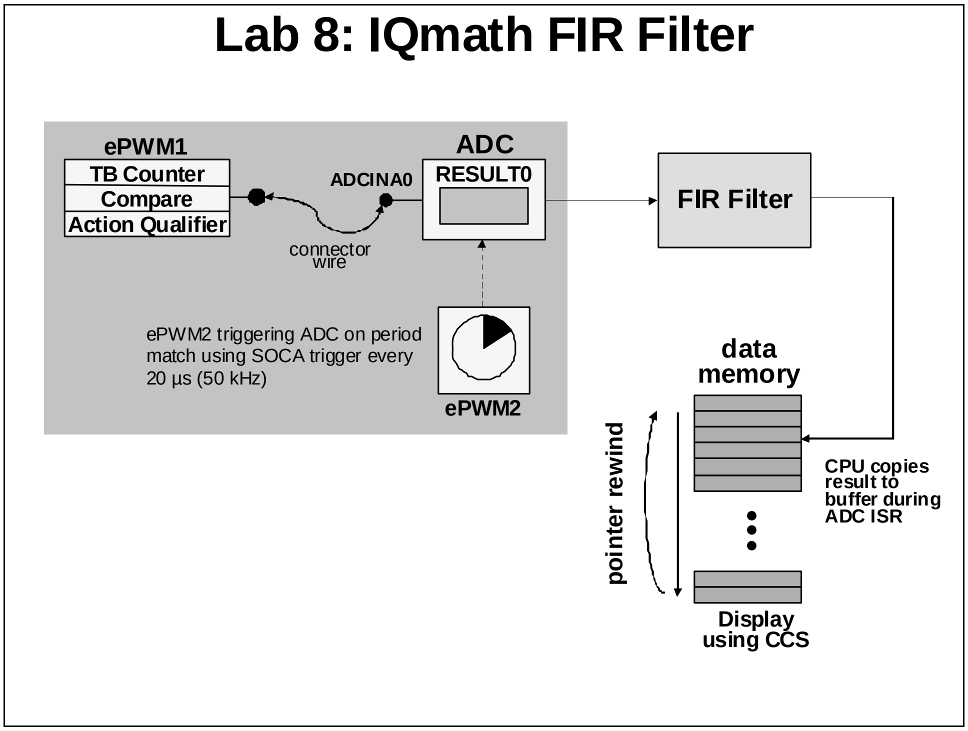
The IQmath approach, matched to a fixed-point processor with 32x32 bit capabilities enables the following:

- Seamless portability of code between fixed and floating-point devices
- Maintenance and support of one source code set from simulation to target device
- Adjustability of numerical resolution (Q value) based on application requirement
- Implementation of systems that may otherwise require floating-point device
- Rapid conversion/porting and implementation of algorithms

Lab 8: IQmath & Floating-Point FIR Filter

➢ Objective

The objective of this lab is to become familiar with IQmath programming. In the previous lab, ePWM1A was setup to generate a 2 kHz, 25% duty cycle symmetric PWM waveform. The waveform was then sampled with the on-chip analog-to-digital converter. In this lab the sampled waveform will be passed through an FIR filter and displayed using the graphing feature of Code Composer Studio. The filter math type is selected in the “IQmathLib.h” file.



Lab 8: IQmath FIR Filter

➢ Procedure

Project File

1. A project named `Lab8.pjt` has been created for this lab. Open the project by clicking on `Project → Open…` and look in `C:\C28x\Labs\Lab8`. All Build Options have been configured the same as the previous lab. The files used in this lab are:

```
Adc.c                            Filter.c
CodeStartBranch.asm              Gpio.c
DefaultIsr_8.c                   Lab_8.cmd
DelayUs.asm                      Main_8.c
DSP2803x_GlobalVariableDefs.c    PieCtrl_5_6_7_8_9_10.c
DSP2803x_Headers_nonBIOS.cmd     PieVect_5_6_7_8_9_10.c
ECap_7_8_9_10_12.c               SysCtrl.c
EPwm_7_8_9_10_12.c               Watchdog.c
```

Project Build Options

2. Setup the include search path to include the IQmath header file. Open the `Build Options` and select the Compiler tab. In the Preprocessor Category, find the `Include Search Path (-i)` box and add to the end of the line (preceeded with a semicolon to append this directory to the existing search path):

   ```
   ;..\IQmath\include
   ```

3. Setup the library search path to include the IQmath library. Select the Linker tab.

   a. In the Libraries Category, find the `Search Path (-i)` box and enter:

   ```
   ..\IQmath\lib
   ```

   b. In the `Include Libraries (-l)` box add to the end of the line (preceeded with a semicolon to append this library to the existing library):

   ```
   ;IQmath.lib
   ```

   Then select `OK` to save the `Build Options`.

Include IQmathLib.h

4. In the CCS project window left click the plus sign (+) to the left of the `Include` folder. Edit `Lab.h` to *uncomment* the line that includes the `IQmathLib.h` header file. Next, in the Function Prototypes section, *uncomment* the function prototype for IQssfir(), the IQ math single-sample FIR filter function. In the Global Variable References section *uncomment* the two _iq references. Save the changes and close the file.

Inspect Lab_8.cmd

5. Open and inspect `Lab_8.cmd`. First, notice that a section called `“IQmath”` is being linked to `L0SARAM`. The IQmath section contains the IQmath library functions (code). Second, notice that a section called `“IQmathTables”` is being linked to the `IQTABLES` with a `TYPE = NOLOAD` modifier after its allocation. The IQmath tables are used by the IQmath library functions. The NOLOAD modifier allows the linker to resolve all addresses in the section, but the section is not actually placed into the `.out` file. This is done because the section is already present in the device ROM (you cannot load data into ROM after the device is manufactured!). The tables were put in the ROM by TI when the device was manufactured. All we need to do is link the section to the addresses where it is known to already reside (the tables are the very first thing in the BOOT ROM, starting at address 0x3FE000). Close the inspected file.

Select a Global IQ value

6. Use `File` → `Open`... to open `c:\C28x\Labs\IQmath\include\IQmathLib.h`. Confirm that the `GLOBAL_Q` type (near beginning of file) is set to a value of 24. If it is not, modify as necessary:

   ```
   #define   GLOBAL_Q       24
   ```

   Recall that this Q type will provide 8 integer bits and 24 fractional bits. Dynamic range is therefore -128 ≤ x < +128, which is sufficient for our purposes in the workshop.

   Notice that the math type is defined as IQmath by:

   ```
   #define   MATH_TYPE      IQ_MATH
   ```

   Close the file.

IQmath Single-Sample FIR Filter

7. Open and inspect `DefaultIsr_8.c`. Notice that the ADCINT_ISR calls the IQmath single-sample FIR filter function, IQssfir(). The filter coefficients have been defined in the beginning of `Main_8.c`.

8. Open and inspect the IQssfir() function in `Filter.c`. This is a simple, non-optimized coding of a basic IQmath single-sample FIR filter. Close the inspected files.

Build and Load

9. Click the “`Build`” button to build and load the project.

Run the Code – Filtered Waveform

10. Open a memory window to view some of the contents of the filtered ADC results buffer. The address label for the filtered ADC results buffer is *AdcBufFiltered*. Set the Format to *16-Bit Unsigned Integer*. We will be running our code in real-time mode, and will have our window continuously refresh.

Note: For the next step, check to be sure that the jumper wire connecting PWM1A (pin # GPIO-00) to ADCINA0 (pin # ADC-A0) is in place on the Docking Station.

---

11. Run the code in real-time mode using the GEL function: `GEL` → `Realtime Emulation Control` → `Run_Realtime_with_Reset`, and watch the memory window update. Verify that the ADC result buffer contains updated values.

12. Open and setup a dual-time graph to plot a 50-point window of the filtered and unfiltered ADC results buffer. Click: `View` → `Graph` → `Time/Frequency`... and set the following values:

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

Select `OK` to save the graph options.

13. The graphical display should show the generated FIR filtered 2 kHz, 25% duty cycle symmetric PWM waveform in the upper display and the unfiltered waveform generated in the previous lab exercise in the lower display. Notice the shape and phase differences between the waveform plots (the filtered curve has rounded edges, and lags the unfiltered plot by several samples). The amplitudes of both plots should run from 0 to 4095.

14. Open and setup two (2) frequency domain plots – one for the filtered and another for the unfiltered ADC results buffer. Click: `View → Graph → Time/Frequency…` and set the following values:

	GRAPH #1	***GRAPH #2***
Display Type	FFT Magnitude	FFT Magnitude
Start Address	AdcBufFiltered	AdcBuf
Acquisition Buffer Size	50	50
FFT Framesize	50	50
DSP Data Type	16-bit unsigned integer	16-bit unsigned integer
Sampling Rate (Hz)	50000	50000

Select `OK` to save the graph options.

15. The graphical displays should show the frequency components of the filtered and unfiltered 2 kHz, 25% duty cycle symmetric PWM waveforms. Notice that the higher frequency components are reduced using the Low-Pass FIR filter in the filtered graph as compared to the unfiltered graph.

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

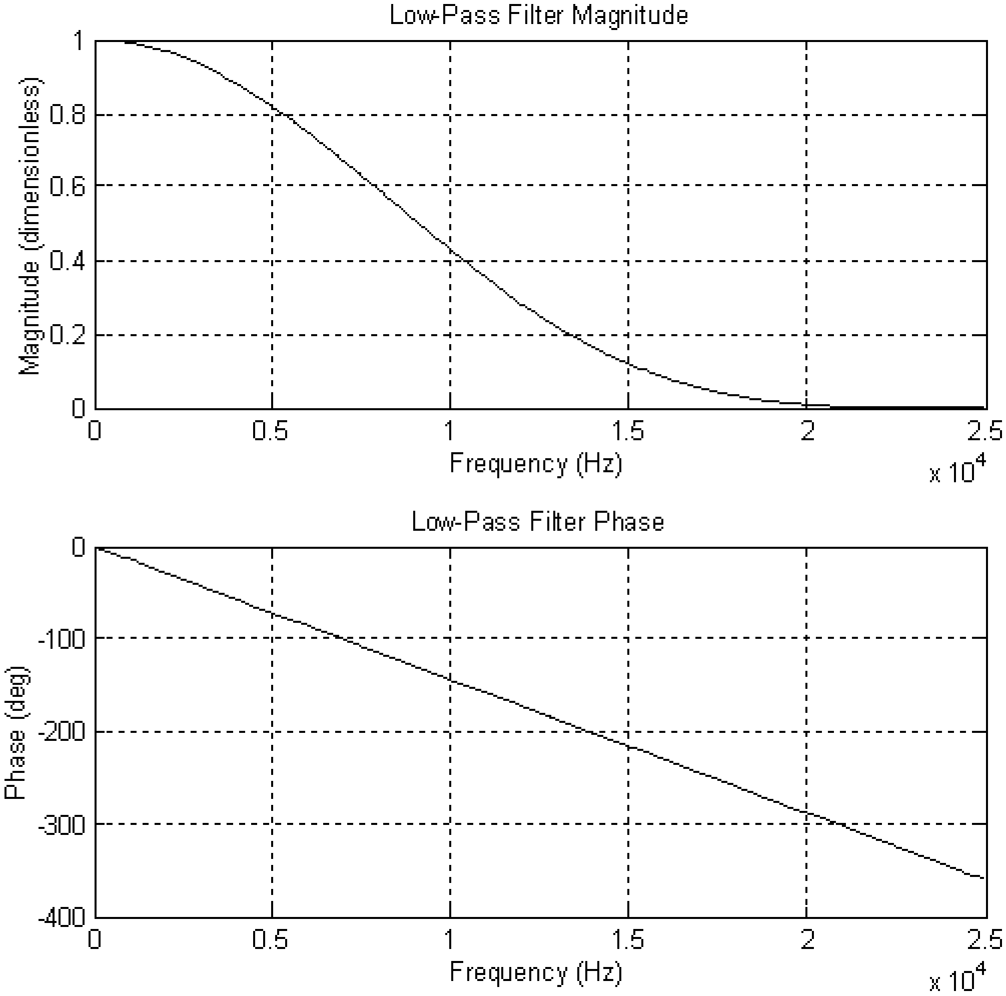
End of Exercise

Lab 8 Reference: Low-Pass FIR Filter

Bode Plot of Digital Low Pass Filter

Coefficients: [1/16, 4/16, 6/16, 4/16, 1/16]

Sample Rate: 50 kHz

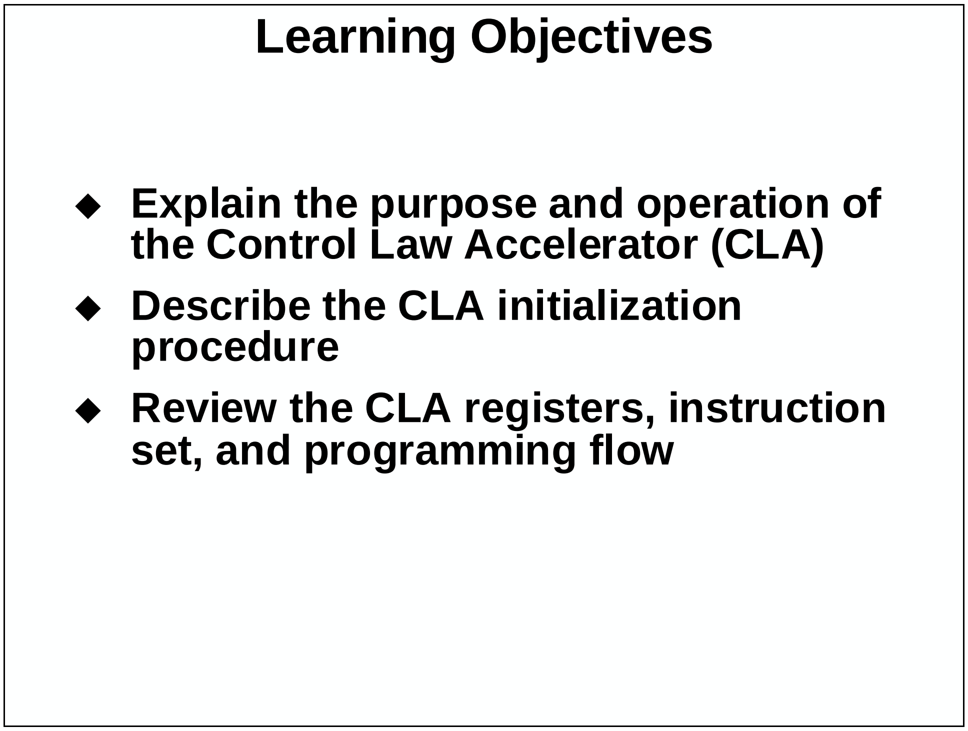


Control Law Accelerator

Introduction

This module explains the operation of the control law accelerator (CLA). The CLA is an independent, fully programmable, 32-bit floating-point math processor that enables concurrent execution into the C28x family. This extends the capabilities of the C28x CPU by adding parallel processing. The CLA has direct access to the ADC result registers, and all ePWM, HRPWM and comparator registers. This allows the CLA to read ADC samples “just-in-time” and significantly reduces the ADC sample to output delay enabling faster system response and higher frequency operation. Utilizing the CLA for time-critical tasks frees up the CPU to perform other system and communication functions concurrently.

Learning Objectives

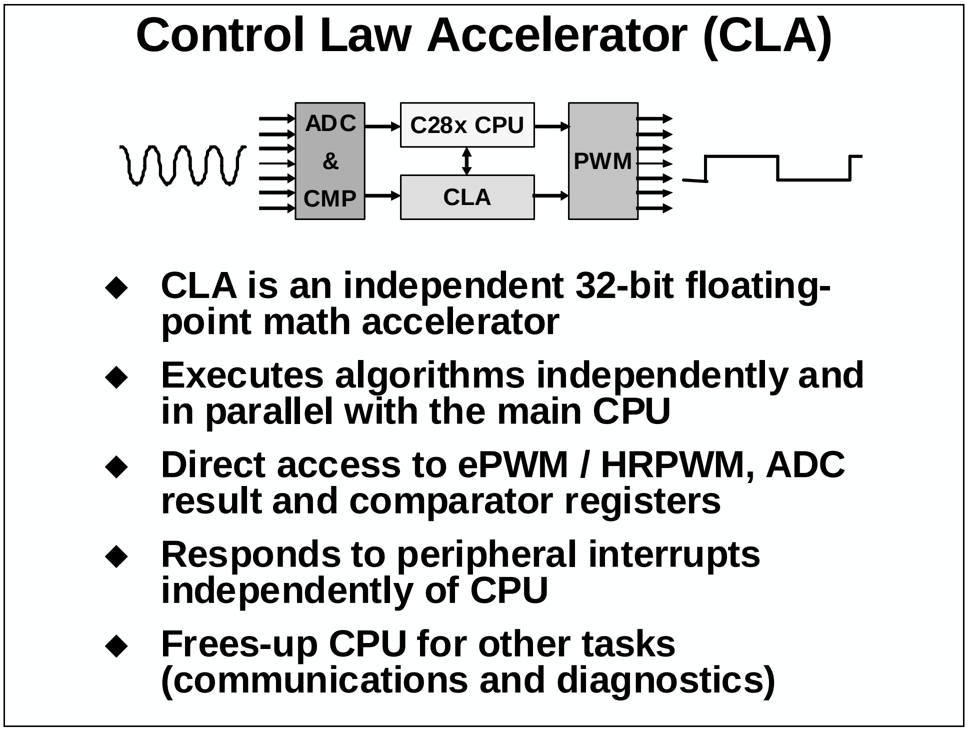


Learning Objectives

- **Explain the purpose and operation of the Control Law Accelerator (CLA)**
- **Describe the CLA initialization procedure**
- **Review the CLA registers, instruction set, and programming flow**

Module Topics

Control Law Accelerator (CLA)

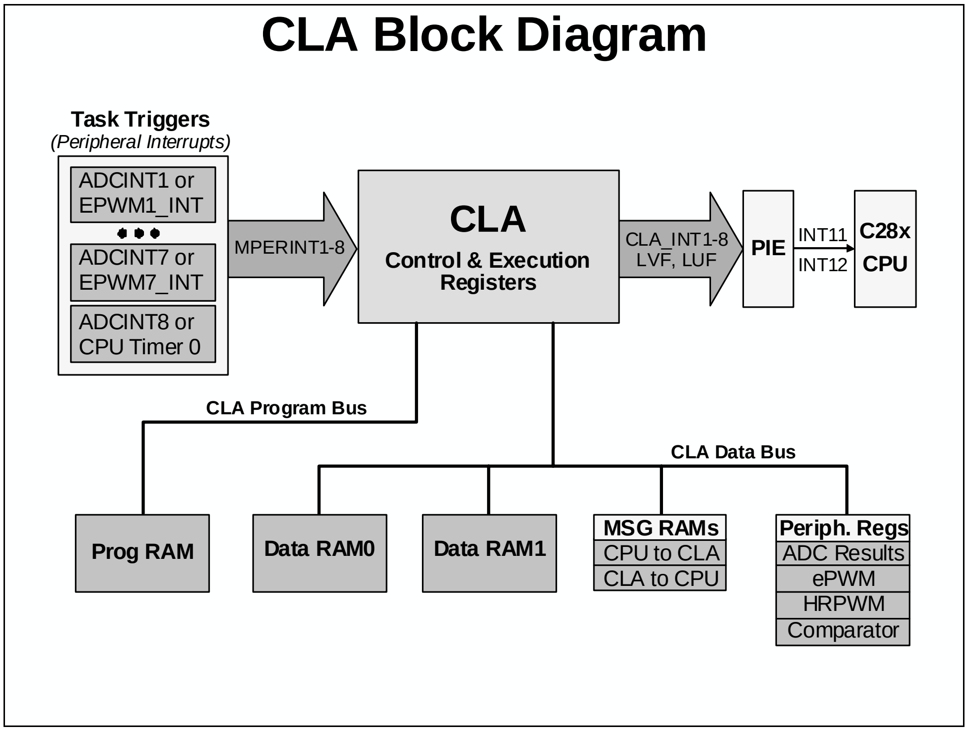


Control Law Accelerator (CLA)

ADC
&
CMP
C28x CPU
CLA
PWM

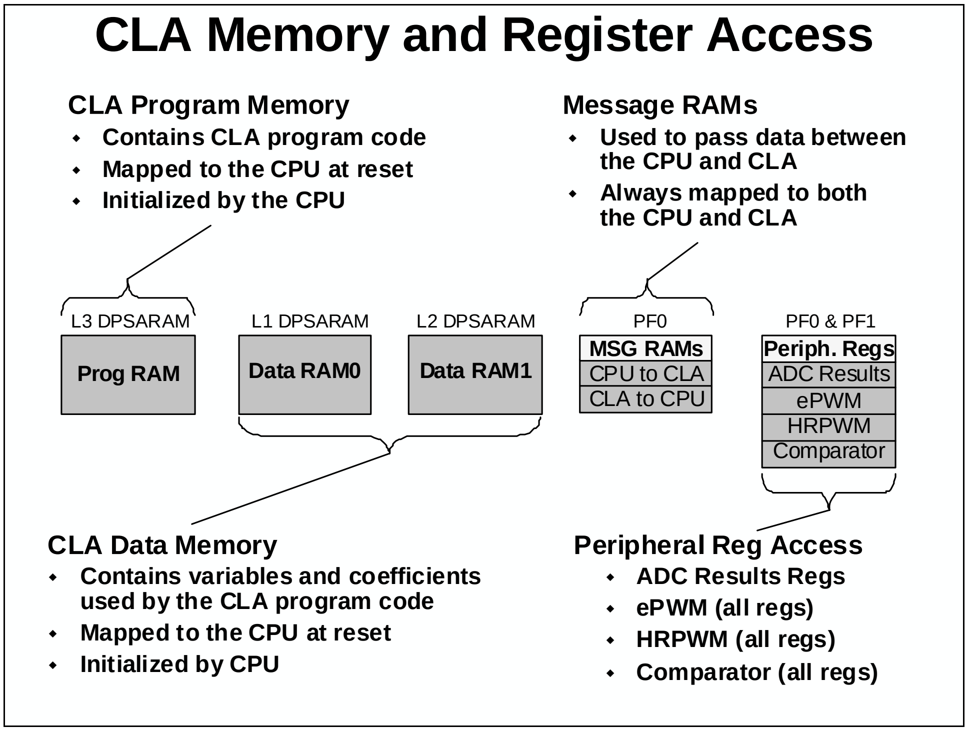
- CLA is an independent 32-bit floating-point math accelerator
- Executes algorithms independently and in parallel with the main CPU
- Direct access to ePWM / HRPWM, ADC result and comparator registers
- Responds to peripheral interrupts independently of CPU
- Frees-up CPU for other tasks (communications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

CLA Block Diagram



CLA Block Diagram

CLA Memory and Register Access



CLA Memory and Register Access

CLA Program Memory

- Contains CLA program code
- Mapped to the CPU at reset
- Initialized by the CPU

Message RAMs

- Used to pass data between the CPU and CLA
- Always mapped to both the CPU and CLA

L3 DPSARAM

Prog RAM

L1 DPSARAM

Data RAM0

L2 DPSARAM

Data RAM1

PF0

MSG RAMs
CPU to CLA
CLA to CPU

PF0 & PF1

Periph. Regs
ADC Results
ePWM
HRPWM
Comparator

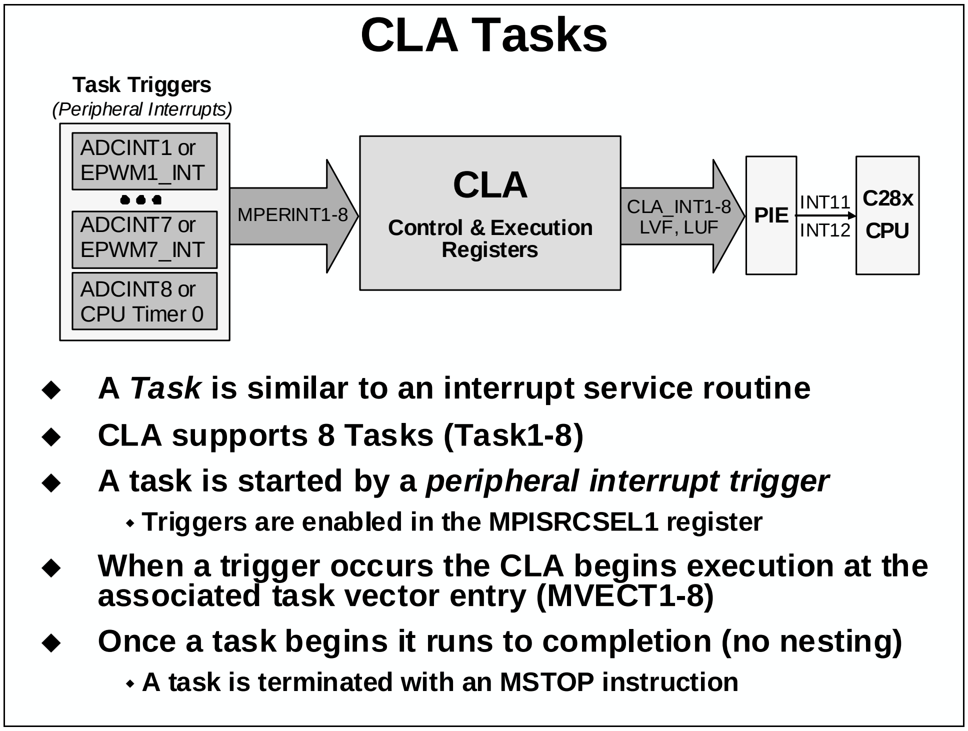
CLA Data Memory

- Contains variables and coefficients used by the CLA program code
- Mapped to the CPU at reset
- Initialized by CPU

Peripheral Reg Access

- ADC Results Regs
- ePWM (all regs)
- HRPWM (all regs)
- Comparator (all regs)

CLA Tasks



CLA Tasks

Task Triggers
(Peripheral Interrupts)

ADCINT1 or EPWM1_INT

•••

ADCINT7 or EPWM7_INT

ADCINT8 or CPU Timer 0

MPERINT1-8

CLA
Control & Execution Registers

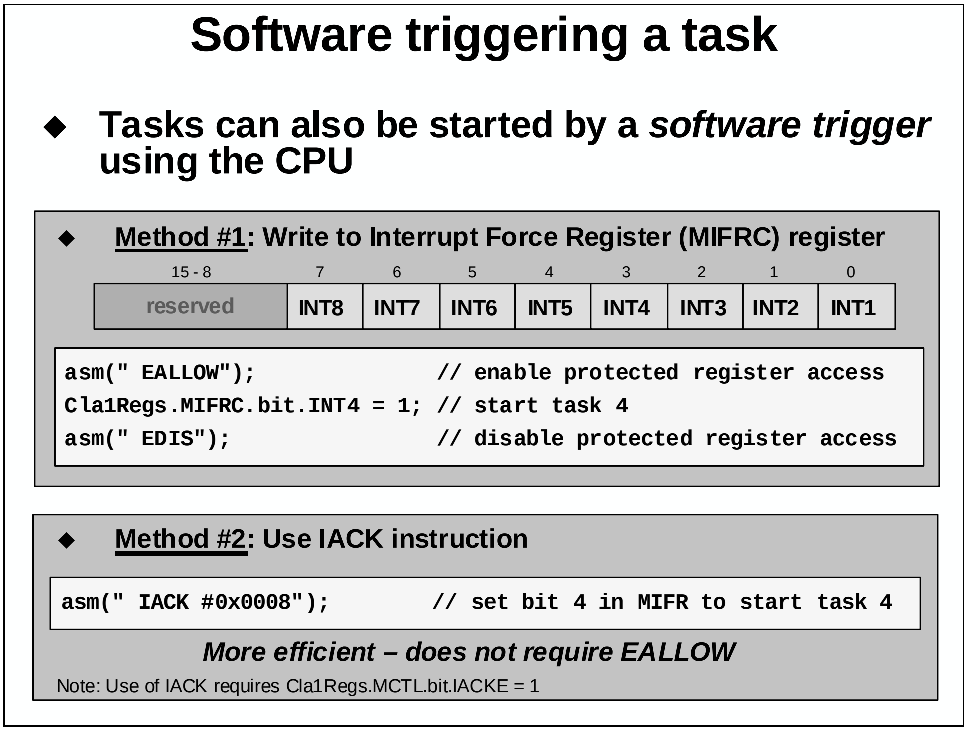
CLA_INT1-8
LVF, LUF

PIE

INT11
INT12

C28x CPU

- **A *Task* is similar to an interrupt service routine**
- **CLA supports 8 Tasks (Task1-8)**
- **A task is started by a *peripheral interrupt trigger***
  - Triggers are enabled in the MPISRCSEL1 register
- **When a trigger occurs the CLA begins execution at the associated task vector entry (MVECT1-8)**
- **Once a task begins it runs to completion (no nesting)**
  - A task is terminated with an MSTOP instruction



Software triggering a task

- Tasks can also be started by a *software trigger* using the CPU

- Method #1: Write to Interrupt Force Register (MIFRC) register

15 - 8	7	6	5	4	3	2	1	0
reserved	INT8	INT7	INT6	INT5	INT4	INT3	INT2	INT1

```
asm(" EALLOW");              // enable protected register access
Cla1Regs.MIFRC.bit.INT4 = 1; // start task 4
asm(" EDIS");                // disable protected register access
```

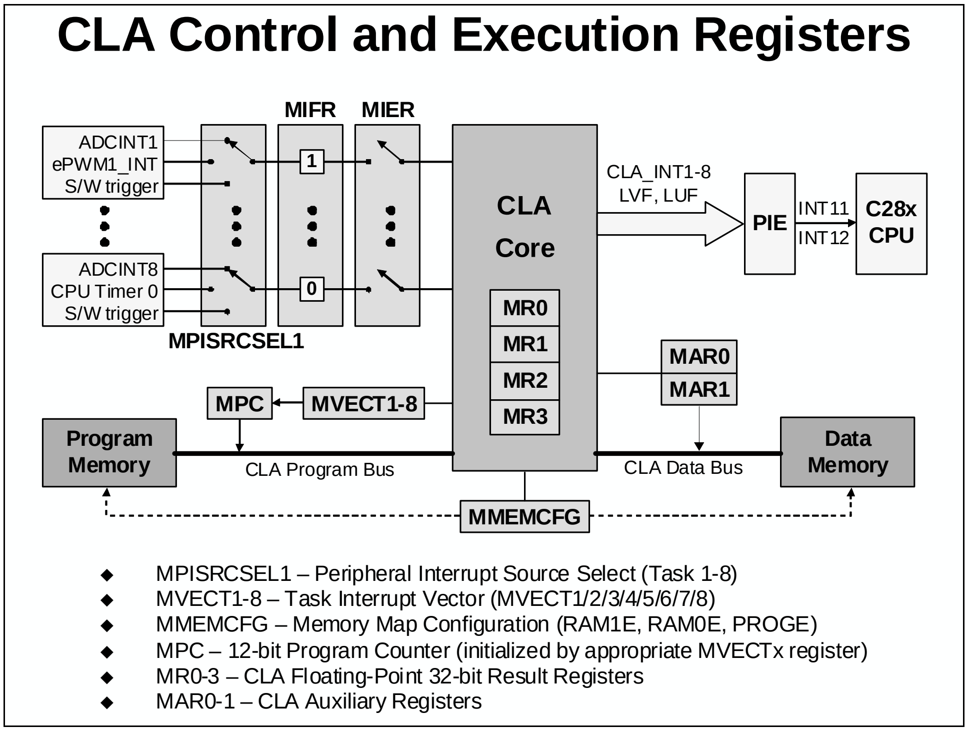
- Method #2: Use IACK instruction

```
asm(" IACK #0x0008");        // set bit 4 in MIFR to start task 4
```

More efficient – does not require EALLOW

Note: Use of IACK requires Cla1Regs.MCTL.bit.IACKE = 1

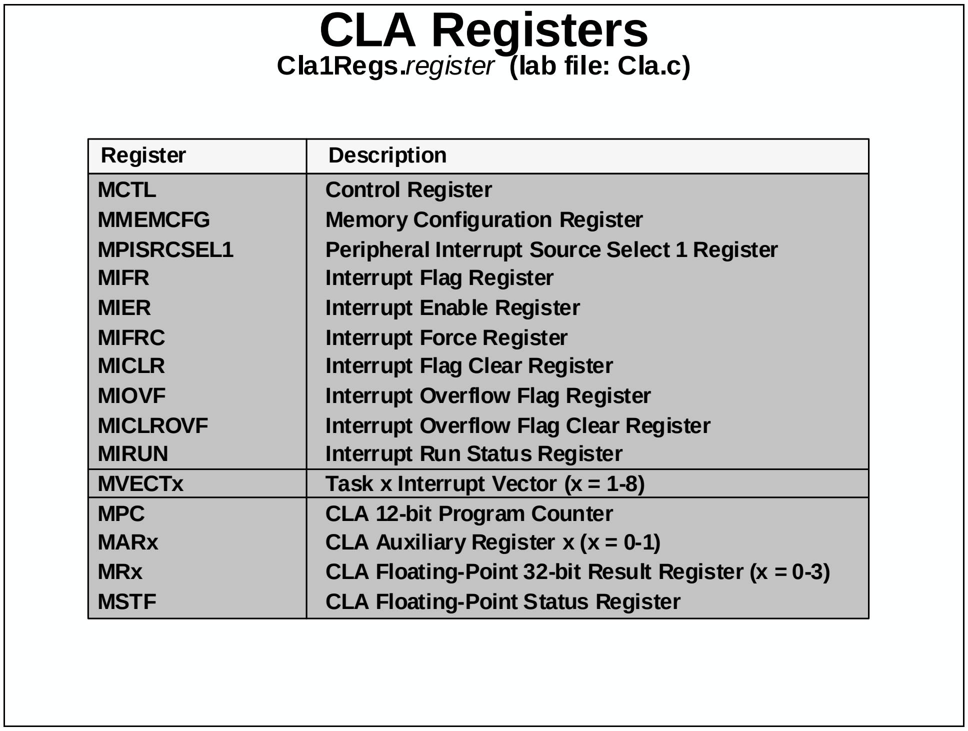
Control and Execution Registers



CLA Control and Execution Registers

- MPISRCSEL1 – Peripheral Interrupt Source Select (Task 1-8)
- MVECT1-8 – Task Interrupt Vector (MVECT1/2/3/4/5/6/7/8)
- MMEMCFG – Memory Map Configuration (RAM1E, RAM0E, PROGE)
- MPC – 12-bit Program Counter (initialized by appropriate MVECTx register)
- MR0-3 – CLA Floating-Point 32-bit Result Registers
- MAR0-1 – CLA Auxiliary Registers

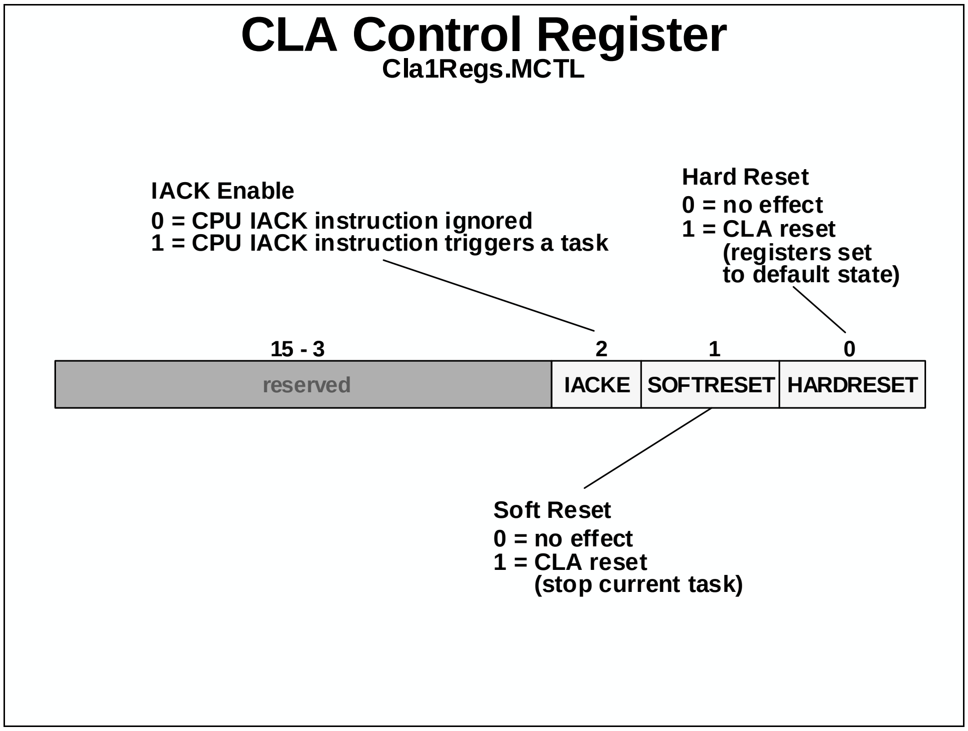
CLA Registers



CLA Registers

Cla1Regs.*register* (lab file: Cla.c)

Register	Description
MCTL	Control Register
MMEMCFG	Memory Configuration Register
MPISRCSEL1	Peripheral Interrupt Source Select 1 Register
MIFR	Interrupt Flag Register
MIER	Interrupt Enable Register
MIFRC	Interrupt Force Register
MICLR	Interrupt Flag Clear Register
MIOVF	Interrupt Overflow Flag Register
MICLROVF	Interrupt Overflow Flag Clear Register
MIRUN	Interrupt Run Status Register
MVECTx	Task x Interrupt Vector (x = 1-8)
MPC	CLA 12-bit Program Counter
MARx	CLA Auxiliary Register x (x = 0-1)
MRx	CLA Floating-Point 32-bit Result Register (x = 0-3)
MSTF	CLA Floating-Point Status Register



CLA Control Register

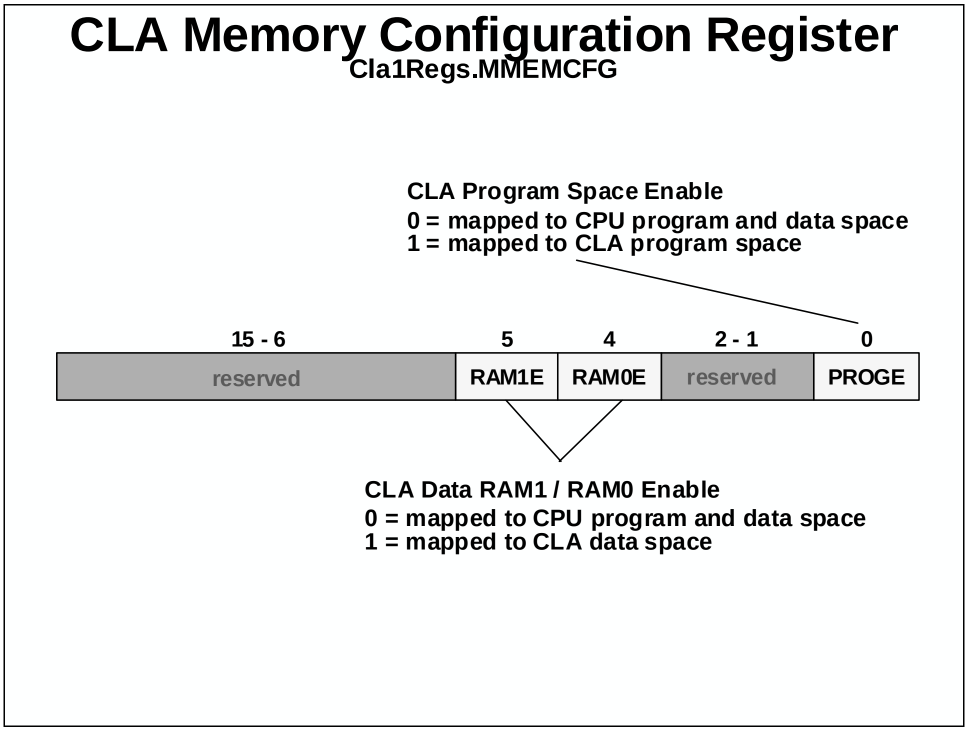
Cla1Regs.MCTL

15 - 3	2	1	0
reserved	IACKE	SOFTRESET	HARDRESET

IACK Enable
0 = CPU IACK instruction ignored
1 = CPU IACK instruction triggers a task

Hard Reset
0 = no effect
1 = CLA reset
(registers set
to default state)

Soft Reset
0 = no effect
1 = CLA reset
(stop current task)



CLA Memory Configuration Register

Cla1Regs.MMEMCFG

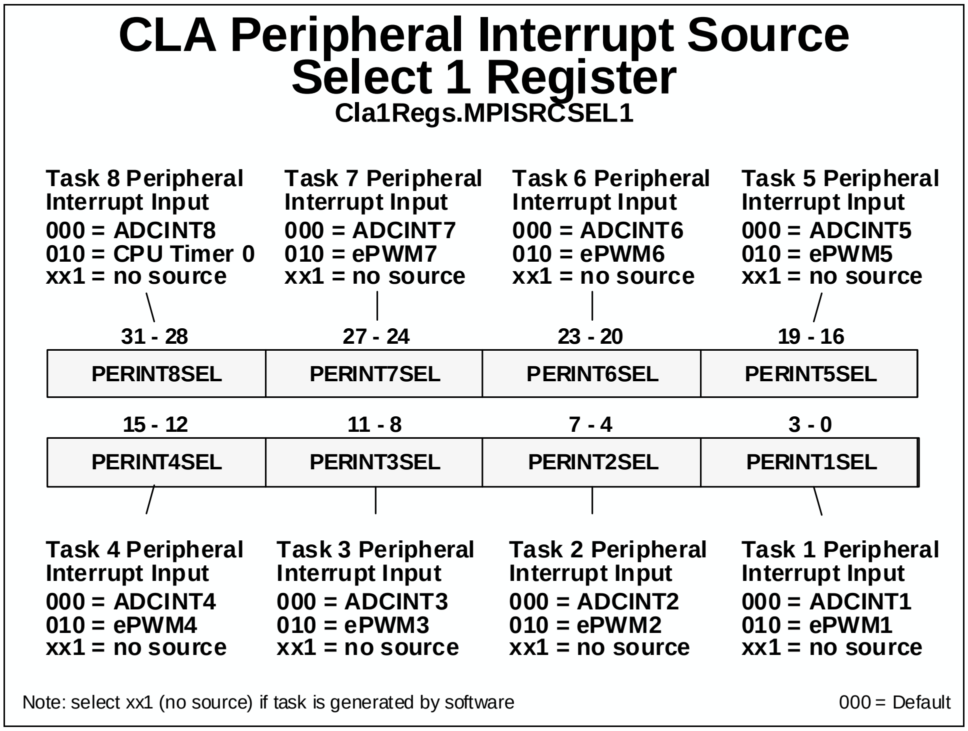
CLA Program Space Enable

0 = mapped to CPU program and data space
1 = mapped to CLA program space

15 - 6	5	4	2 - 1	0
reserved	RAM1E	RAM0E	reserved	PROGE

CLA Data RAM1 / RAM0 Enable

0 = mapped to CPU program and data space
1 = mapped to CLA data space



CLA Peripheral Interrupt Source Select 1 Register

Cla1Regs.MPISRCSEL1

Task 8 Peripheral Interrupt Input
000 = ADCINT8
010 = CPU Timer 0
xx1 = no source

Task 7 Peripheral Interrupt Input
000 = ADCINT7
010 = ePWM7
xx1 = no source

Task 6 Peripheral Interrupt Input
000 = ADCINT6
010 = ePWM6
xx1 = no source

Task 5 Peripheral Interrupt Input
000 = ADCINT5
010 = ePWM5
xx1 = no source

31 - 28	27 - 24	23 - 20	19 - 16
PERINT8SEL	PERINT7SEL	PERINT6SEL	PERINT5SEL

15 - 12	11 - 8	7 - 4	3 - 0
PERINT4SEL	PERINT3SEL	PERINT2SEL	PERINT1SEL

Task 4 Peripheral Interrupt Input
000 = ADCINT4
010 = ePWM4
xx1 = no source

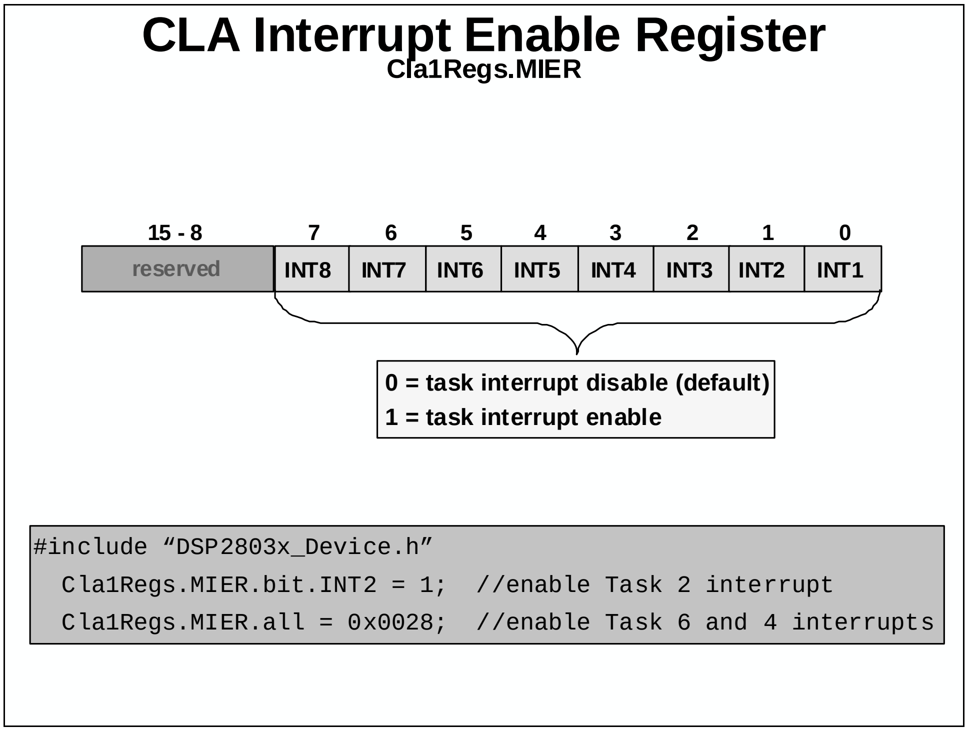
Task 3 Peripheral Interrupt Input
000 = ADCINT3
010 = ePWM3
xx1 = no source

Task 2 Peripheral Interrupt Input
000 = ADCINT2
010 = ePWM2
xx1 = no source

Task 1 Peripheral Interrupt Input
000 = ADCINT1
010 = ePWM1
xx1 = no source

Note: select xx1 (no source) if task is generated by software

000 = Default



CLA Interrupt Enable Register

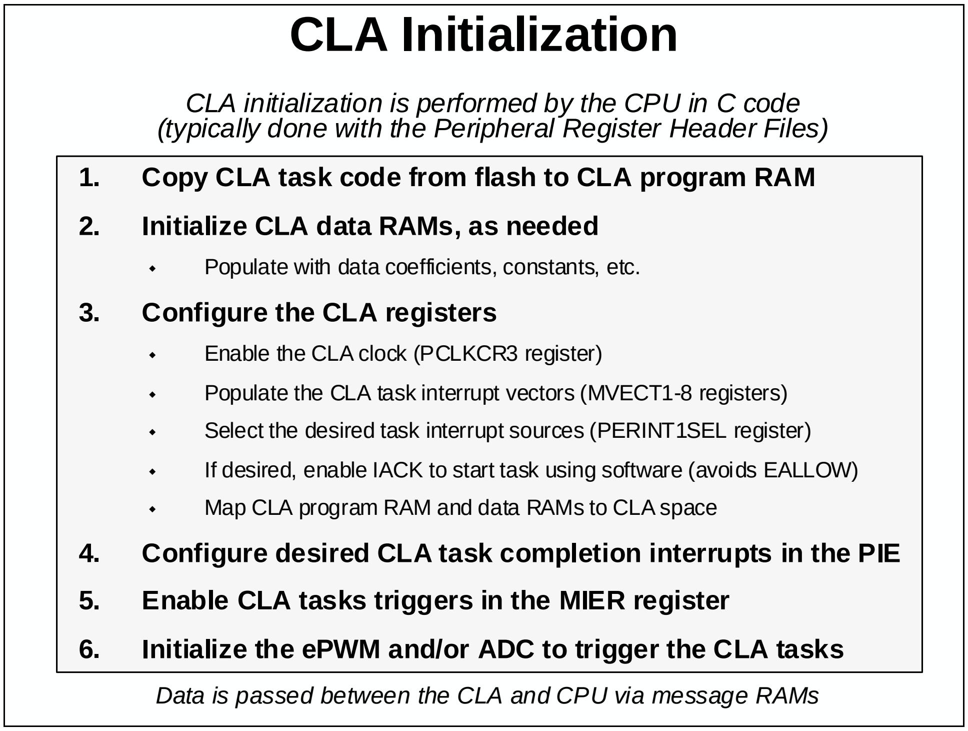
Cla1Regs.MIER

15 - 8	7	6	5	4	3	2	1	0
reserved	INT8	INT7	INT6	INT5	INT4	INT3	INT2	INT1

0 = task interrupt disable (default)
1 = task interrupt enable

```
#include “DSP2803x_Device.h”
  Cla1Regs.MIER.bit.INT2 = 1;  //enable Task 2 interrupt
  Cla1Regs.MIER.all = 0x0028;  //enable Task 6 and 4 interrupts
```

CLA Initialization



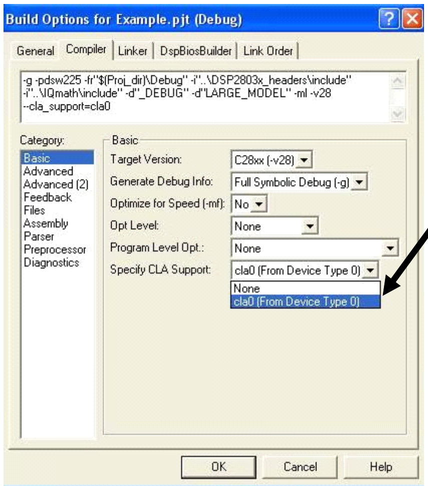
CLA Initialization

CLA initialization is performed by the CPU in C code
(typically done with the Peripheral Register Header Files)

1. **Copy CLA task code from flash to CLA program RAM**
2. **Initialize CLA data RAMs, as needed**
   - Populate with data coefficients, constants, etc.
3. **Configure the CLA registers**
   - Enable the CLA clock (PCLKCR3 register)
   - Populate the CLA task interrupt vectors (MVECT1-8 registers)
   - Select the desired task interrupt sources (PERINT1SEL register)
   - If desired, enable IACK to start task using software (avoids EALLOW)
   - Map CLA program RAM and data RAMs to CLA space
4. **Configure desired CLA task completion interrupts in the PIE**
5. **Enable CLA tasks triggers in the MIER register**
6. **Initialize the ePWM and/or ADC to trigger the CLA tasks**

Data is passed between the CLA and CPU via message RAMs

Enabling CLA Support in CCS



```
-g -pdsw225 -fr"$(Proj_dir)\Debug" -i"..\DSP2803x_headers\include"
-i"..\IQmath\include" -d"_DEBUG" -d"LARGE_MODEL" -ml -v28
--cla_support=cla0
```

Basic

- Target Version: C28xx (-v28)
- Generate Debug Info: Full Symbolic Debug (-g)
- Optimize for Speed (-mf): No
- Opt Level: None
- Program Level Opt.: None
- Specify CLA Support: cla0 (From Device Type 0)

- **Note: You must be using a C28x Piccolo device that has the Control Law Accelerator!**
- **In the project build options, select:** ***'cla0 (From Device Type 0)'***
- **This is required in order to assemble CLA code**
- **CLA support requires codegen tools v5.2.0 or later**

CLA Task Programming

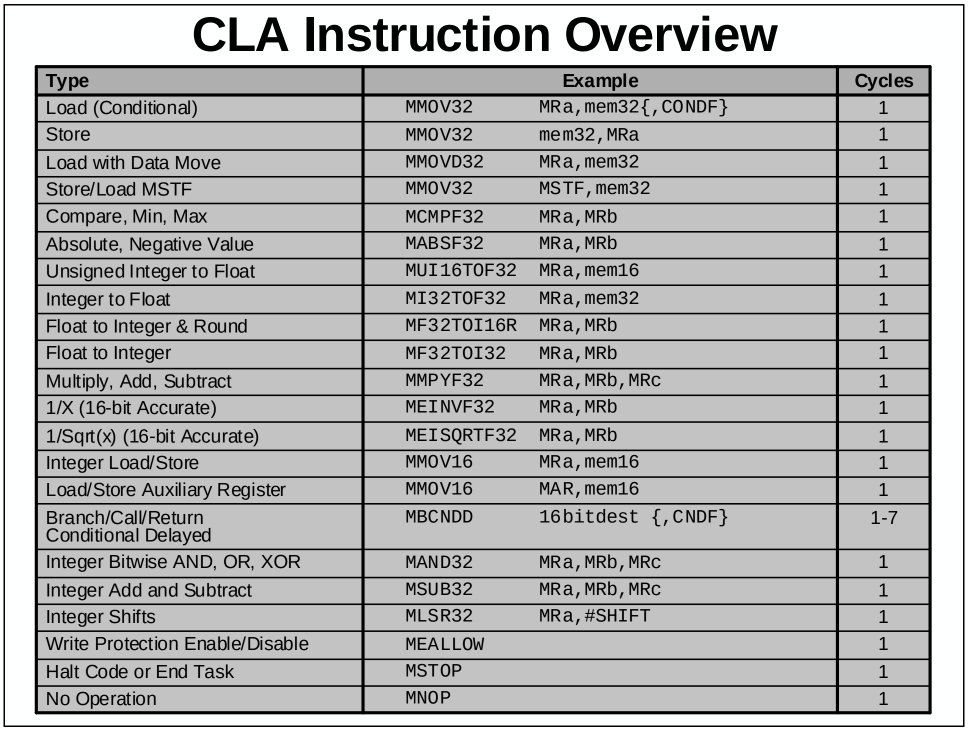
CLA Task Programming

- **CLA tasks are written in assembly code**
- **Same instruction format as the C28x and C28x+FPU**
  - **Destination operand is always on the left**
  - **Same mnemonics as C28x+FPU but with a leading "M"**

```
CPU:               MPY        ACC, T,  loc16
FPU:               MPYF32     R0H, R1H, R2H
CLA:               MMPYF32    MR0, MR1, MR2
```

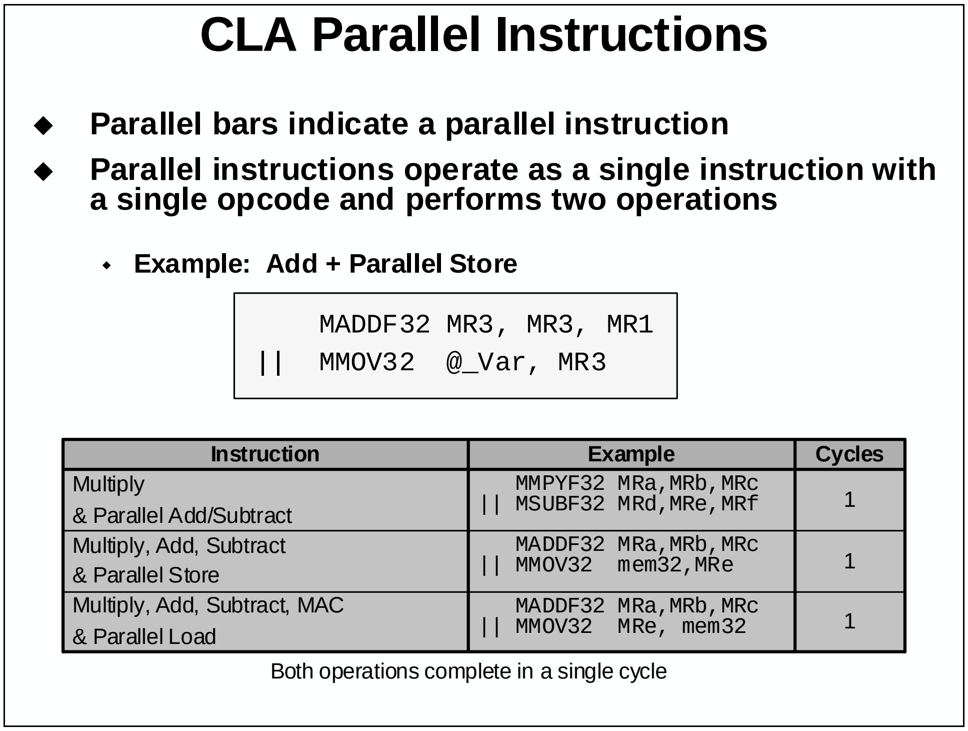
Destination   Source Operands

CLA Instruction Set



CLA Instruction Overview

Type	Example	Cycles
Load (Conditional)	`MMOV32      MRa,mem32{,CONDF}`	1
Store	`MMOV32      mem32,MRa`	1
Load with Data Move	`MMOVD32     MRa,mem32`	1
Store/Load MSTF	`MMOV32      MSTF,mem32`	1
Compare, Min, Max	`MCMPF32     MRa,MRb`	1
Absolute, Negative Value	`MABSF32     MRa,MRb`	1
Unsigned Integer to Float	`MUI16TOF32  MRa,mem16`	1
Integer to Float	`MI32TOF32   MRa,mem32`	1
Float to Integer & Round	`MF32TOI16R  MRa,MRb`	1
Float to Integer	`MF32TOI32   MRa,MRb`	1
Multiply, Add, Subtract	`MMPYF32     MRa,MRb,MRc`	1
1/X (16-bit Accurate)	`MEINVF32    MRa,MRb`	1
1/Sqrt(x) (16-bit Accurate)	`MEISQRTF32  MRa,MRb`	1
Integer Load/Store	`MMOV16      MRa,mem16`	1
Load/Store Auxiliary Register	`MMOV16      MAR,mem16`	1
Branch/Call/Return Conditional Delayed	`MBCNDD      16bitdest {,CNDF}`	1-7
Integer Bitwise AND, OR, XOR	`MAND32      MRa,MRb,MRc`	1
Integer Add and Subtract	`MSUB32      MRa,MRb,MRc`	1
Integer Shifts	`MLSR32      MRa,#SHIFT`	1
Write Protection Enable/Disable	`MEALLOW`	1
Halt Code or End Task	`MSTOP`	1
No Operation	`MNOP`	1



CLA Parallel Instructions

- **Parallel bars indicate a parallel instruction**
- **Parallel instructions operate as a single instruction with a single opcode and performs two operations**
  - **Example: Add + Parallel Store**

```
    MADDF32 MR3, MR3, MR1
||  MMOV32  @_Var, MR3
```

Instruction	Example	Cycles

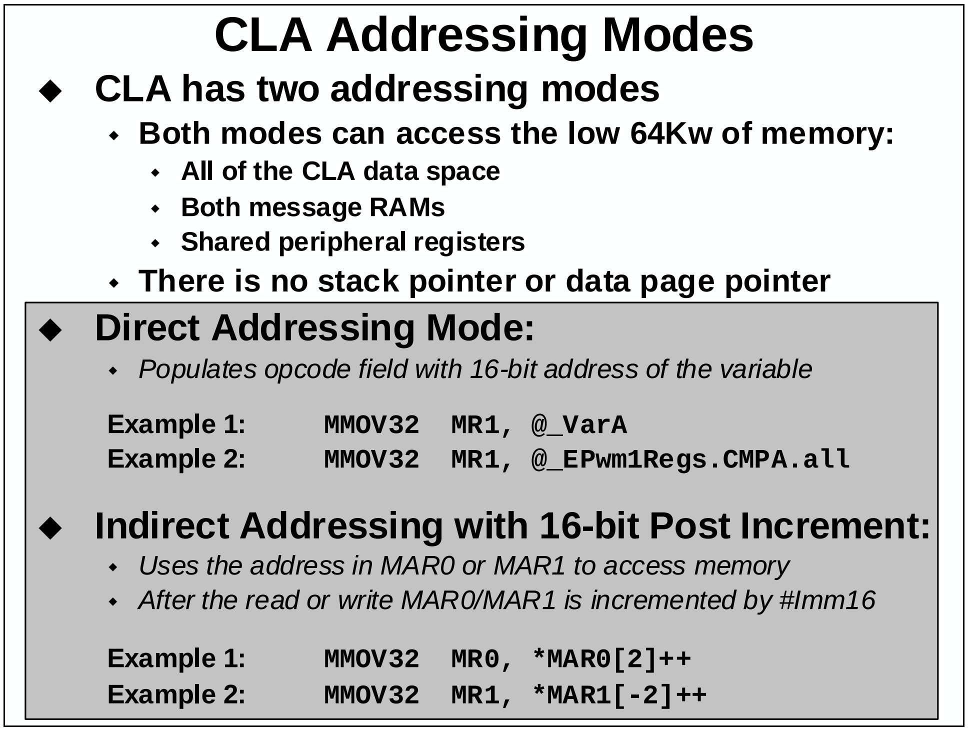
Multiply
& Parallel Add/Subtract
	`   MMPYF32 MRa,MRb,MRc`
	`|| MSUBF32 MRd,MRe,MRf`
		1

Multiply, Add, Subtract
& Parallel Store
	`   MADDF32 MRa,MRb,MRc`
	`|| MMOV32  mem32,MRe`
		1

Multiply, Add, Subtract, MAC
& Parallel Load
	`   MADDF32 MRa,MRb,MRc`
	`|| MMOV32  MRe, mem32`
		1

Both operations complete in a single cycle

CLA Addressing Modes



CLA Addressing Modes

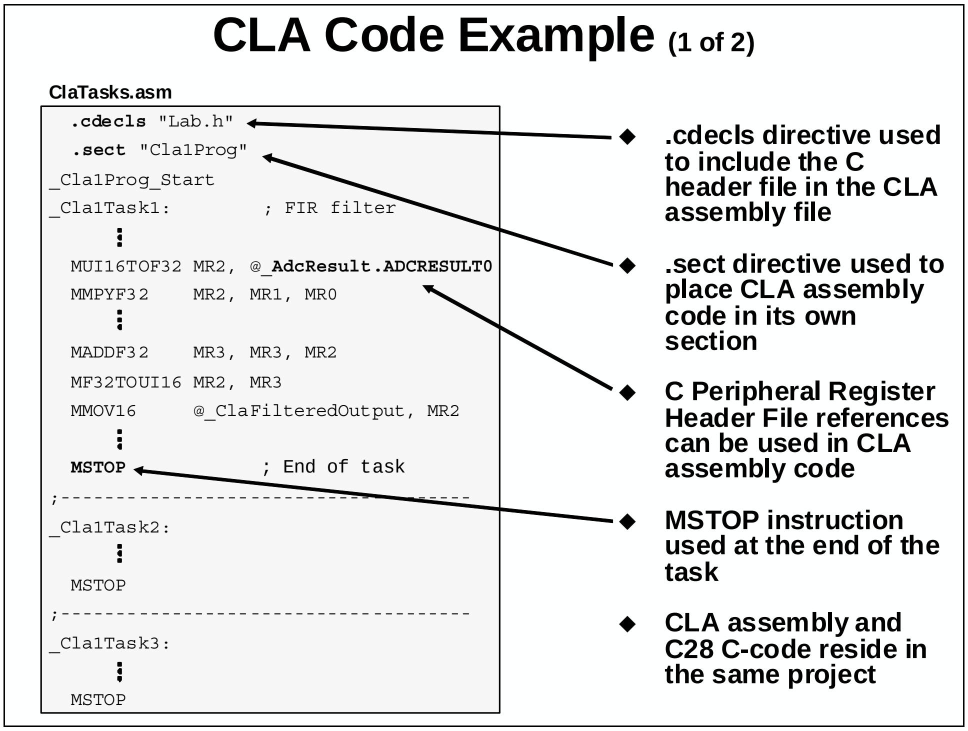
- CLA has two addressing modes
  - Both modes can access the low 64Kw of memory:
    - All of the CLA data space
    - Both message RAMs
    - Shared peripheral registers
  - There is no stack pointer or data page pointer
- Direct Addressing Mode:
  - *Populates opcode field with 16-bit address of the variable*

Example 1: `MMOV32 MR1, @_VarA`
Example 2: `MMOV32 MR1, @_EPwm1Regs.CMPA.all`

- Indirect Addressing with 16-bit Post Increment:
  - *Uses the address in MAR0 or MAR1 to access memory*
  - *After the read or write MAR0/MAR1 is incremented by #Imm16*

Example 1: `MMOV32 MR0, *MAR0[2]++`
Example 2: `MMOV32 MR1, *MAR1[-2]++`

CLA Code Example

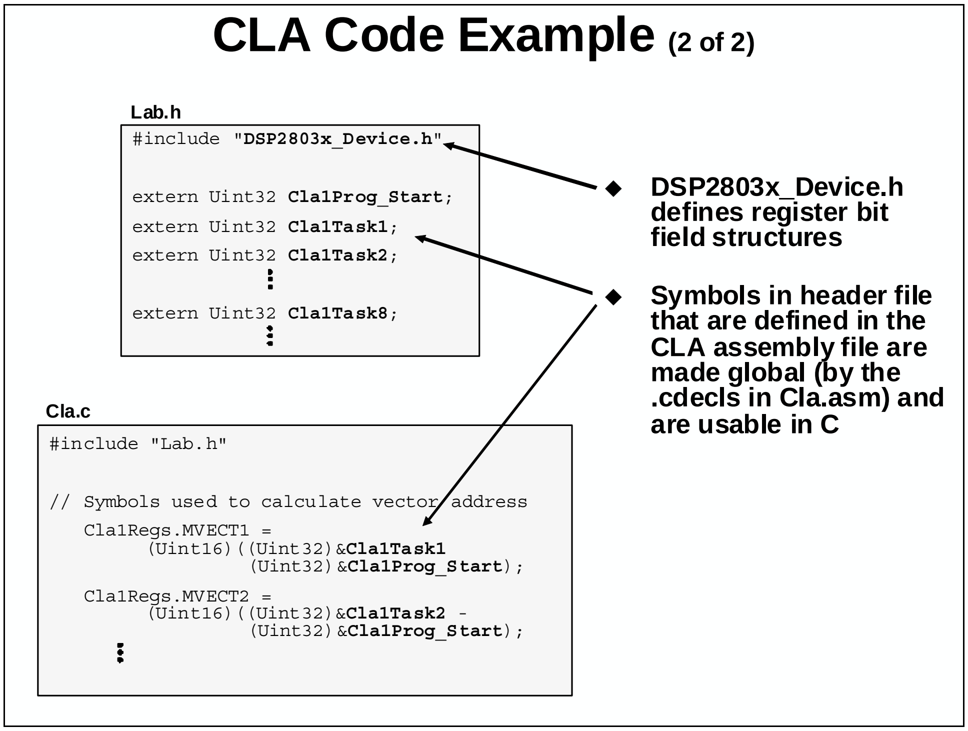


CLA Code Example (1 of 2)

ClaTasks.asm

```
  .cdecls "Lab.h"
  .sect "Cla1Prog"
_Cla1Prog_Start
_Cla1Task1:        ; FIR filter
      ⋮
  MUI16TOF32 MR2, @_AdcResult.ADCRESULT0
  MMPYF32    MR2, MR1, MR0
      ⋮
  MADDF32    MR3, MR3, MR2
  MF32TOUI16 MR2, MR3
  MMOV16     @_ClaFilteredOutput, MR2
      ⋮
  MSTOP              ; End of task
;------------------------------------
_Cla1Task2:
      ⋮
  MSTOP
;------------------------------------
_Cla1Task3:
      ⋮
  MSTOP
```

- .cdecls directive used to include the C header file in the CLA assembly file
- .sect directive used to place CLA assembly code in its own section
- C Peripheral Register Header File references can be used in CLA assembly code
- MSTOP instruction used at the end of the task
- CLA assembly and C28 C-code reside in the same project



CLA Code Example (2 of 2)

Lab.h

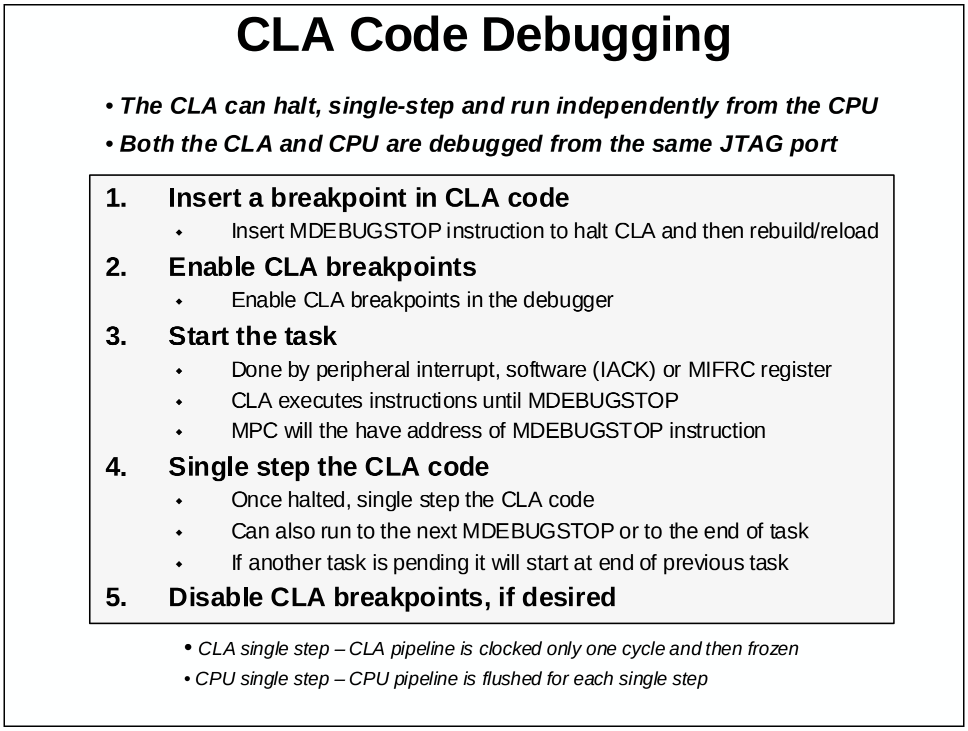
```
#include "DSP2803x_Device.h"

extern Uint32 Cla1Prog_Start;
extern Uint32 Cla1Task1;
extern Uint32 Cla1Task2;
              ⋮
extern Uint32 Cla1Task8;
              ⋮
```

Cla.c

```
#include "Lab.h"

// Symbols used to calculate vector address
   Cla1Regs.MVECT1 =
        (Uint16)((Uint32)&Cla1Task1
                 (Uint32)&Cla1Prog_Start);

   Cla1Regs.MVECT2 =
        (Uint16)((Uint32)&Cla1Task2 -
                 (Uint32)&Cla1Prog_Start);
       ⋮
```

- **DSP2803x_Device.h defines register bit field structures**
- **Symbols in header file that are defined in the CLA assembly file are made global (by the .cdecls in Cla.asm) and are usable in C**

CLA Code Debugging



CLA Code Debugging

- ***The CLA can halt, single-step and run independently from the CPU***
- ***Both the CLA and CPU are debugged from the same JTAG port***

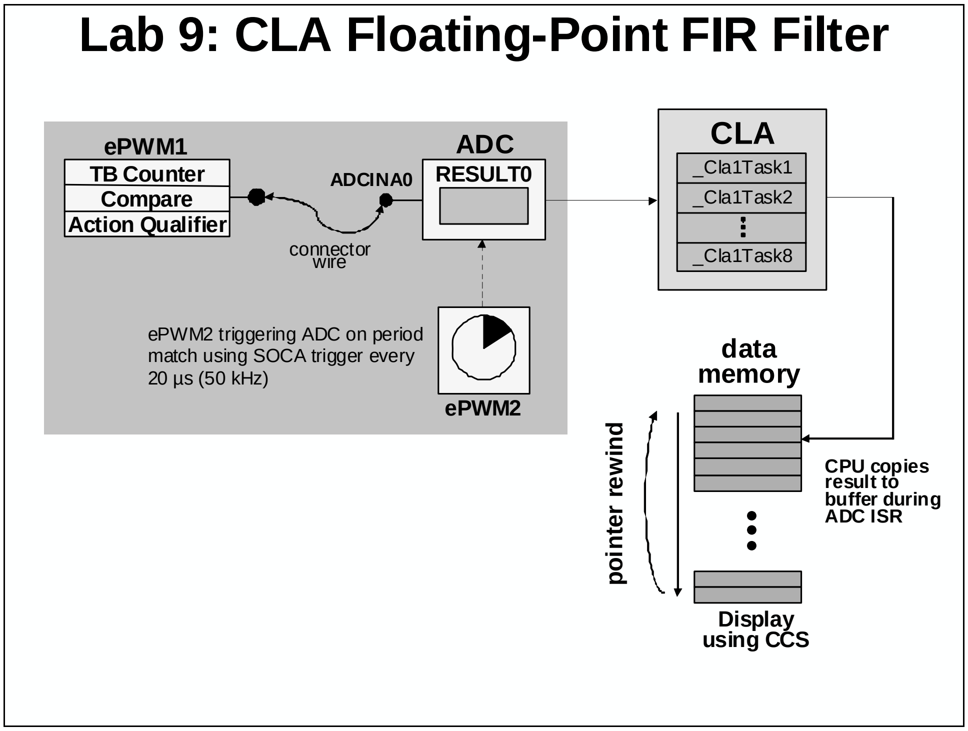
1. **Insert a breakpoint in CLA code**
   - Insert MDEBUGSTOP instruction to halt CLA and then rebuild/reload
2. **Enable CLA breakpoints**
   - Enable CLA breakpoints in the debugger
3. **Start the task**
   - Done by peripheral interrupt, software (IACK) or MIFRC register
   - CLA executes instructions until MDEBUGSTOP
   - MPC will the have address of MDEBUGSTOP instruction
4. **Single step the CLA code**
   - Once halted, single step the CLA code
   - Can also run to the next MDEBUGSTOP or to the end of task
   - If another task is pending it will start at end of previous task
5. **Disable CLA breakpoints, if desired**

- *CLA single step – CLA pipeline is clocked only one cycle and then frozen*
- *CPU single step – CPU pipeline is flushed for each single step*

Lab 9: CLA Floating-Point FIR Filter

- **Objective**

The objective of this lab is to become familiar with operation of the CLA. In the previous lab, the CPU was used to filter the ePWM1A generated 2 kHz, 25% duty cycle symmetric PWM waveform. In this lab, the PWM waveform will be filtered using the CLA. The CLA will directly read the ADC result register and a task will run a low-pass FIR filter on the sampled waveform. The filtered result will be stored in a circular memory buffer. Note that the CLA is operating concurrently with the CPU. As an operational test, the filtered and unfiltered waveforms will be displayed using the graphing feature of Code Composer Studio.


Lab 9: CLA Floating-Point FIR Filter

- **Procedure**

Project File

1. A project named `Lab9.pjt` has been created for this lab. Open the project by clicking on `Project` → `Open…` and look in `C:\C28x\Labs\Lab9`. All Build Options have been configured the same as the previous lab. The files used in this lab are:

```
Adc.c
Cla_9.c
ClaTasks.asm
CodeStartBranch.asm
DefaultIsr_9_10.c
DelayUs.asm
DSP2803x_GlobalVariableDefs.c
DSP2803x_Headers_nonBIOS.cmd
ECap_7_8_9_10_12.c
EPwm_7_8_9_10_12.c
Filter.c
Gpio.c
Lab_9.cmd
Main_9.c
PieCtrl_5_6_7_8_9_10.c
PieVect_5_6_7_8_9_10.c
SysCtrl.c
Watchdog.c
```

Enabling CLA Support in CCS

2. Open the Build Options and select the Compiler tab. In the Basic Category set the Specify CLA Support to cla0 (From Device Type 0). This is needed to assemble CLA code. Then select OK to save the Build Options.

Inspect Lab_9.cmd

3. Open and inspect Lab_9.cmd. Notice that a section called "Cla1Prog" is being linked to L3DPSARAM. This section links the CLA program tasks (assembly code) to the CPU memory space. This memory space will be remapped to the CLA memory space during initialization. Also, notice the two message RAM sections used to pass data between the CPU and CLA.

Setup CLA Initialization

During the CLA initialization, the CPU memory block L3DPSARAM needs to be configured as CLA program memory. This memory space contains the CLA Task routines, which are coded in assembly. The CLA Task 1 has been configured to run an FIR filter. The CLA needs to be configured to start Task 1 on the ADCINT1 interrupt trigger. The next section will setup the PIE interrupt for the CLA.

4. Open ClaTasks.asm and notice that the .cdecls directive is being used to include the C header file in the CLA assembly file. Therefore, we can use the Peripheral Register Header File references in the CLA assembly code. Next, notice Task 1 has been configured to run an FIR filter. Within this code special instructions have been used to convert the ADC result integer (i.e. the filter input) to floating-point and the floating-point filter output back to integer.

5. Edit Cla_9.c to implement the CLA operation as described in the objective for this lab exercise. Configure the L3DPSARM memory block to be mapped to CLA program memory space. Set Task 1 peripheral interrupt source to ADCINT1 and set the other Task peripheral interrupt source inputs to no source. Enable CLA Task 1 interrupt.

6. Open Main_9.c and add a line of code in main() to call the InitCla() function. There are no passed parameters or return values. You just type

   ```
   InitCla();
   ```

   at the desired spot in main().

Setup PIE Interrupt for CLA

Recall that ePWM2 is triggering the ADC at a 50 kHz rate. In the previous lab exercise, the ADC generated an interrupt to the CPU, and the CPU implemented the FIR filter in the ADC ISR. For this lab exercise, the ADC is instead triggering the CLA, and the CLA will directly read the ADC result register and run a task implementing an FIR filter. The CLA will generate an interrupt to the CPU, which will store the filtered results to a circular buffer implemented in the CLA ISR.

7. Edit `Adc.c` to *comment out* the code used to enable ADCINT1 interrupt in PIE group 1. This is no longer being used. The CLA interrupt will be used instead.

8. Using the "PIE Interrupt Assignment Table" find the location for the CLA Task 1 interrupt "CLA1_INT1" and fill in the following information:

   PIE group #:___________ # within group:__________

   This information will be used in the next step.

9. Modify the end of `Cla_9.c` to do the following:

   - Enable the "CLA1_INT1" interrupt in the PIE (Hint: use the `PieCtrlRegs` structure)
   - Enable the appropriate core interrupt in the IER register

10. Open and inspect `DefaultIsr_9_10.c`. Notice that this file contains the CLA interrupt service routine. Save and close all modified files.

Build and Load

11. Click the "`Build`" button to build and load the project.

Run the Code – Test the CLA Operation

Note: For the next step, check to be sure that the jumper wire connecting PWM1A (pin # GPIO-00) to ADCINA0 (pin # ADC-A0) is in place on the Docking Station.

12. Run the code in real-time mode using the GEL function: `GEL → Realtime Emulation Control → Run_Realtime_with_Reset`, and watch the memory window update. Verify that the ADC result buffer contains updated values.

13. Setup a dual-time graph of the filtered and unfiltered ADC results buffer. Click: `View → Graph → Time/Frequency…` and set the following values:

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

14. The graphical display should show the filtered PWM waveform in the upper display and the unfiltered waveform in the lower display. You should see that the results match the previous lab exercise.

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

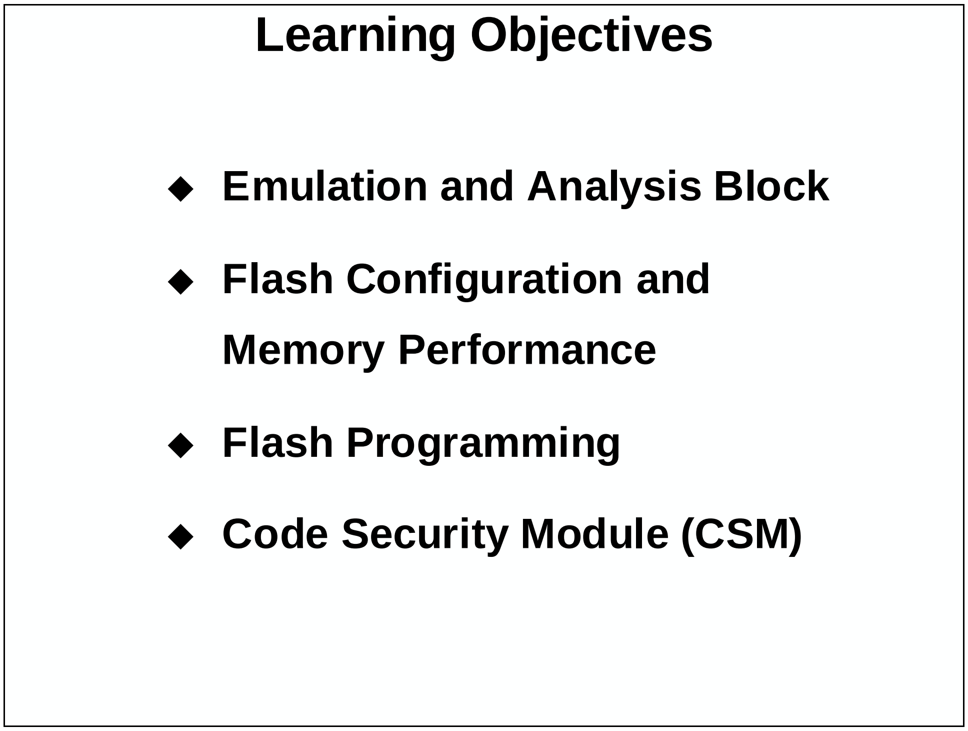
End of Exercise

System Design

Introduction

This module discusses various aspects of system design. Details of the emulation and analysis block along with JTAG will be explored. Flash memory programming and the Code Security Module will be described.

Learning Objectives



Learning Objectives

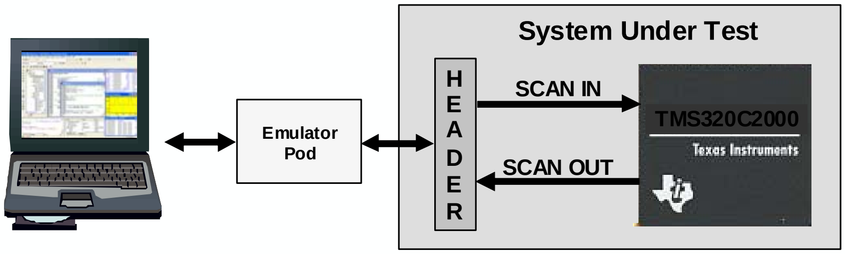
- ◆ Emulation and Analysis Block
- ◆ Flash Configuration and Memory Performance
- ◆ Flash Programming
- ◆ Code Security Module (CSM)

Module Topics

Emulation and Analysis Block

JTAG Emulation System

(based on IEEE 1149.1 Boundary Scan Standard)



Some Available Emulators

XDS510 CLASS -

BlackHawk:	USB2000
Signum System:	JTAGjet-TMS-C2000
Spectrum Digital:	XDS510LC

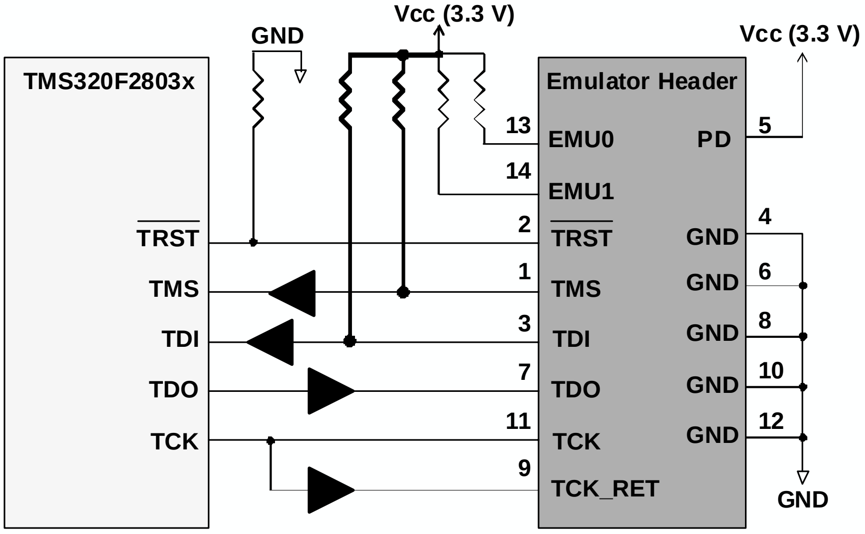
These emulators are C2000 specific, and are much lower cost than emulators that support all TI MCU/DSP platforms (although those can certainly be used)

XDS100 CLASS -

BlackHawk:	USB100
Olimex:	TMS320-JTAG-USB
Spectrum Digital:	XDS100
TI:	TMDSEMU100U-14T

These emulators are much slower than the ones listed above, but are also available at a lower cost than XDS510 class and are NOT C2000 specific

Emulator Connections to the Device



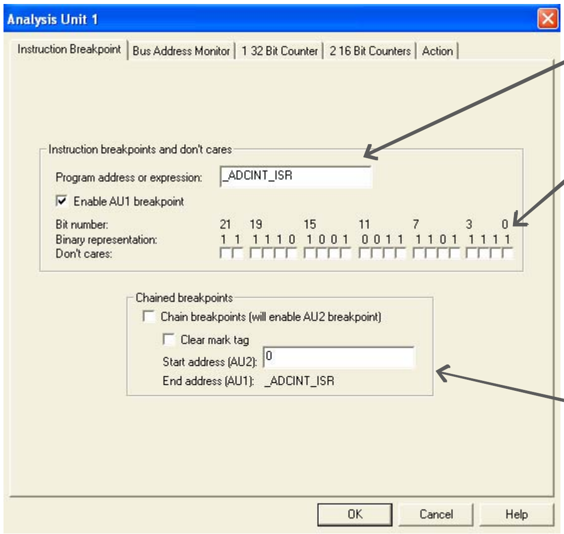
■ = If distance between device and header is greater than 6 inches

On-Chip Emulation Analysis Block: Capabilities

Two hardware analysis units can be configured to provide any one of the following advanced debug features:

Analysis Configuration		Debug Activity
2 Hardware Breakpoints	⇒	Halt on a specified instruction (for debugging in Flash)
2 Address Watchpoints	⇒	A memory location is getting corrupted; halt the processor when any value is written to this location
1 Address Watchpoint with Data	⇒	Halt program execution after a specific value is written to a variable
1 Pair Chained Breakpoints	⇒	Halt on a specified instruction only after some other specific routine has executed

On-Chip Emulation Analysis Block: Hardware Breakpoints



Instruction breakpoints and don't cares

Program address or expression: _ADCINT_ISR

☑ Enable AU1 breakpoint

```
Bit number:             21   19        15        11        7         3      0
Binary representation:  1 1  1 1 1 0  1 0 0 1   0 0 1 1   1 1 0 1   1 1 1 1
Don't cares:            □□   □□□□      □□□□      □□□□      □□□□      □□□□
```

Chained breakpoints

□ Chain breakpoints (will enable AU2 breakpoint)

□ Clear mark tag

Start address (AU2): 0

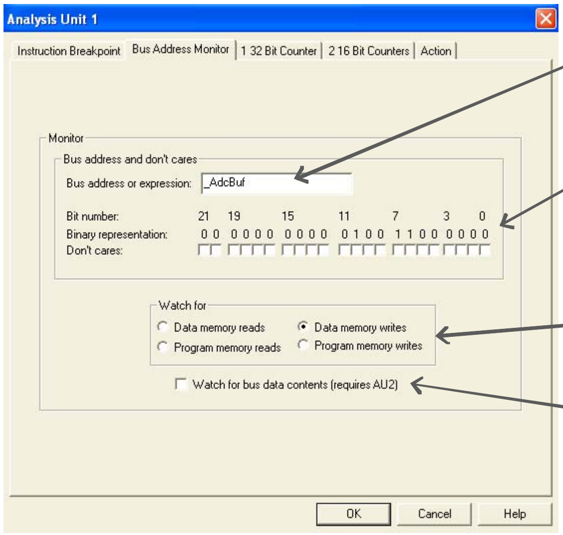
End address (AU1): _ADCINT_ISR

Symbolic or numeric address

Mask value for specifying address ranges

Chained breakpoint selection

On-Chip Emulation Analysis Block: Watchpoints



Monitor

Bus address and don't cares

Bus address or expression: _AdcBuf

Bit number: 21 19 15 11 7 3 0

Binary representation: 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 0 0 0 0 0

Don't cares: □□ □□□□ □□□□ □□□□ □□□□ □□□□

Watch for

○ Data memory reads ● Data memory writes

○ Program memory reads ○ Program memory writes

□ Watch for bus data contents (requires AU2)

Symbolic or numeric address

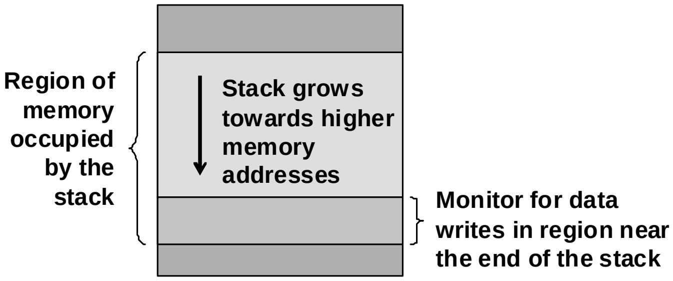
Mask value for specifying address ranges

Bus selection

Address with Data selection

On-Chip Emulation Analysis Block: Online Stack Overflow Detection

- Emulation analysis registers are accessible to code as well!
- Configure a watchpoint to monitor for writes near the end of the stack
- Watchpoint triggers maskable RTOSINT interrupt
- Works with DSP/BIOS and non-DSP/BIOS
  - See TI application report SPRA820 for implementation details



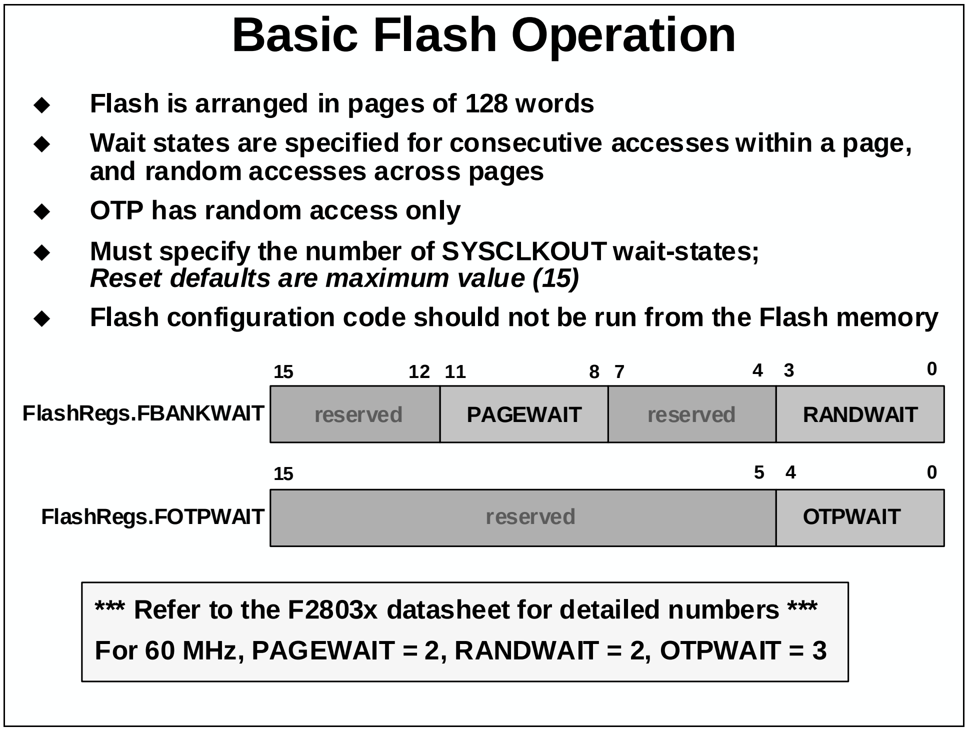
Region of memory occupied by the stack

Stack grows towards higher memory addresses

Monitor for data writes in region near the end of the stack

Data Memory

Flash Configuration and Memory Performance



Basic Flash Operation

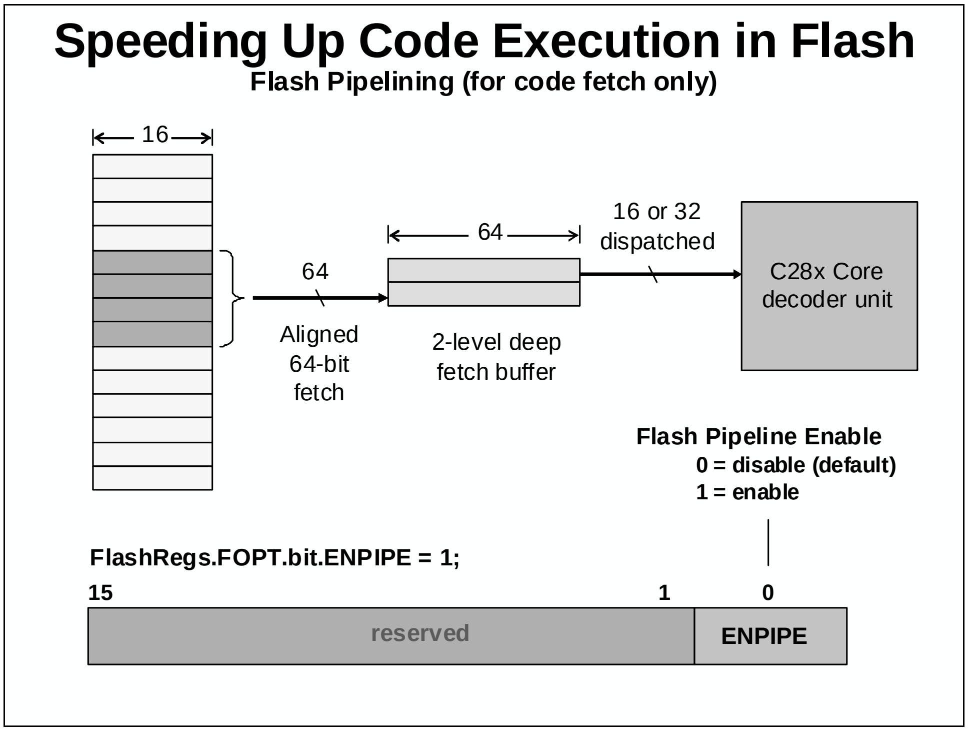
- **Flash is arranged in pages of 128 words**
- **Wait states are specified for consecutive accesses within a page, and random accesses across pages**
- **OTP has random access only**
- **Must specify the number of SYSCLKOUT wait-states;** ***Reset defaults are maximum value (15)***
- **Flash configuration code should not be run from the Flash memory**

	15 - 12	11 - 8	7 - 4	3 - 0
FlashRegs.FBANKWAIT	reserved	**PAGEWAIT**	reserved	**RANDWAIT**

	15 - 5	4 - 0
FlashRegs.FOTPWAIT	reserved	**OTPWAIT**

***** Refer to the F2803x datasheet for detailed numbers *****

For 60 MHz, PAGEWAIT = 2, RANDWAIT = 2, OTPWAIT = 3



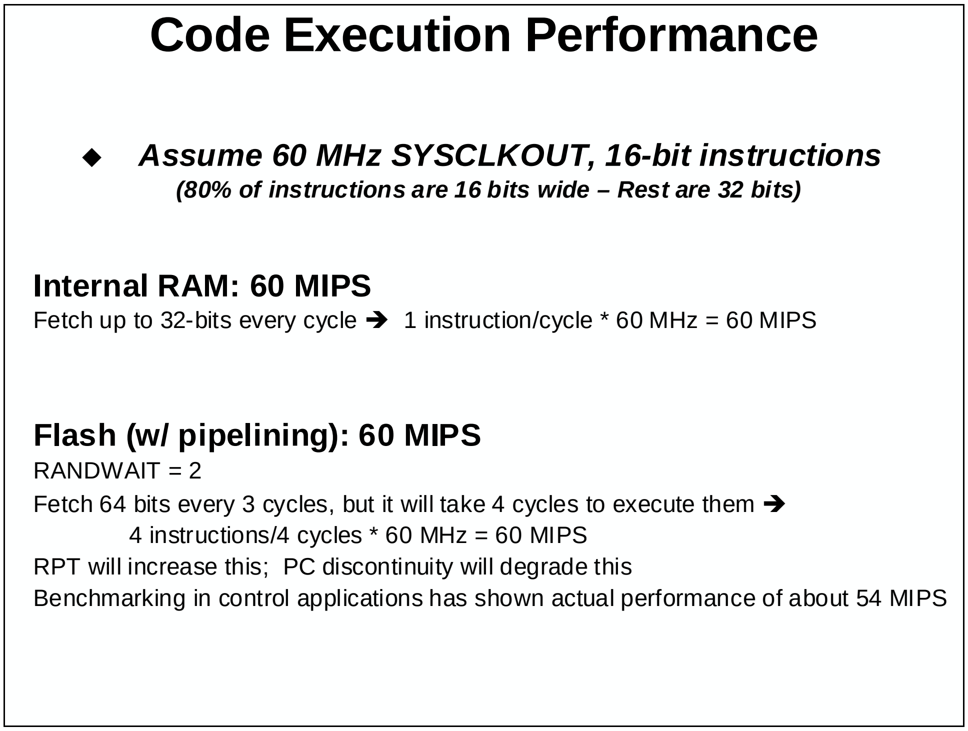
Speeding Up Code Execution in Flash

Flash Pipelining (for code fetch only)

Flash Pipeline Enable
0 = disable (default)
1 = enable

FlashRegs.FOPT.bit.ENPIPE = 1;

15 - 1	0
reserved	**ENPIPE**



Code Execution Performance

- ***Assume 60 MHz SYSCLKOUT, 16-bit instructions***
  (80% of instructions are 16 bits wide – Rest are 32 bits)

Internal RAM: 60 MIPS

Fetch up to 32-bits every cycle ➔ 1 instruction/cycle * 60 MHz = 60 MIPS

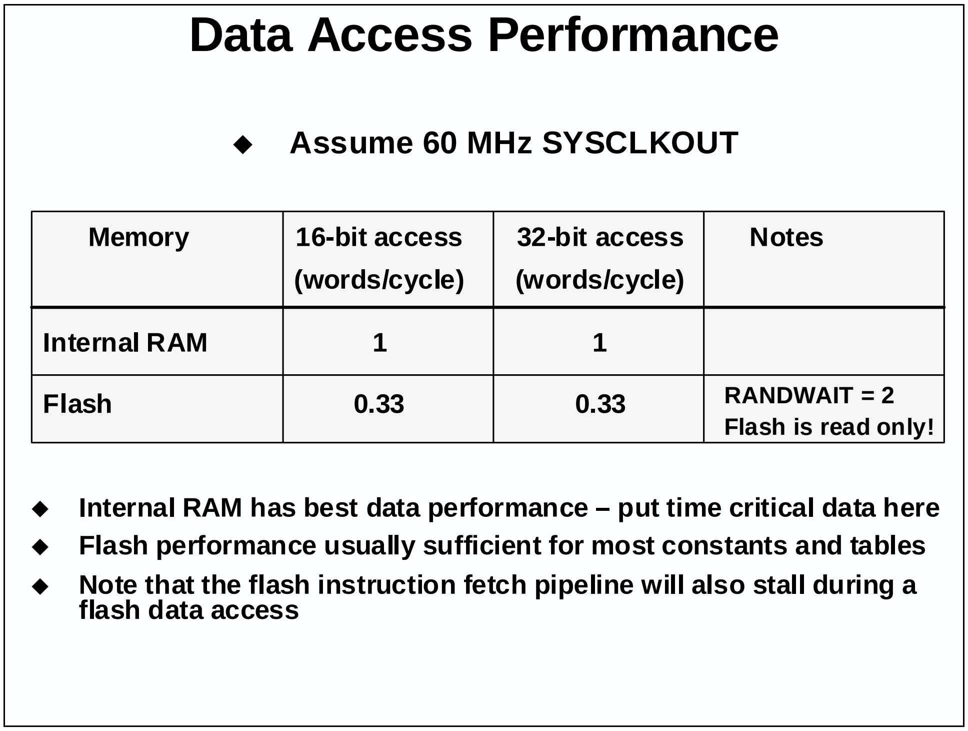
Flash (w/ pipelining): 60 MIPS

RANDWAIT = 2

Fetch 64 bits every 3 cycles, but it will take 4 cycles to execute them ➔
4 instructions/4 cycles * 60 MHz = 60 MIPS

RPT will increase this; PC discontinuity will degrade this

Benchmarking in control applications has shown actual performance of about 54 MIPS

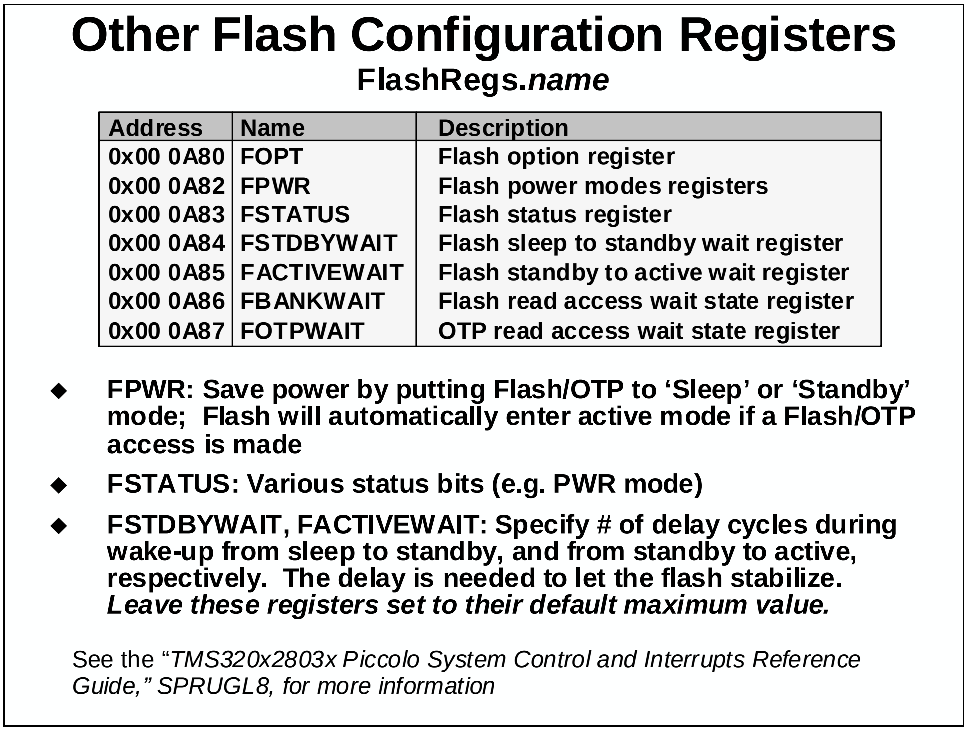


Data Access Performance

- **Assume 60 MHz SYSCLKOUT**

Memory	16-bit access (words/cycle)	32-bit access (words/cycle)	Notes
Internal RAM	1	1	
Flash	0.33	0.33	RANDWAIT = 2 Flash is read only!

- **Internal RAM has best data performance – put time critical data here**
- **Flash performance usually sufficient for most constants and tables**
- **Note that the flash instruction fetch pipeline will also stall during a flash data access**



Other Flash Configuration Registers

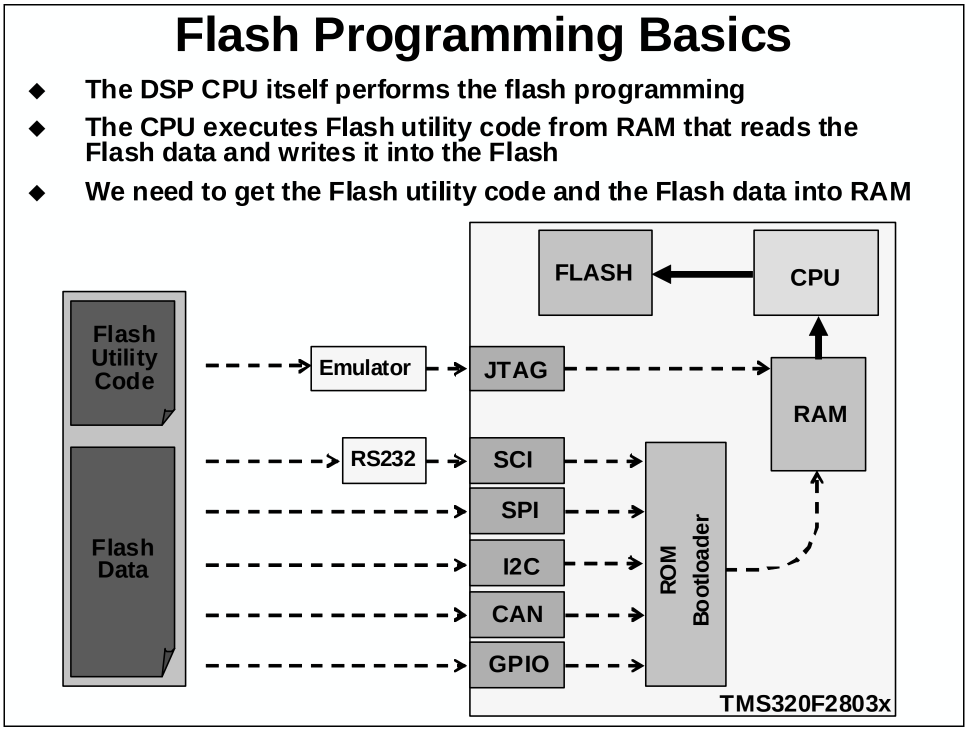
FlashRegs.*name*

Address	Name	Description
0x00 0A80	FOPT	Flash option register
0x00 0A82	FPWR	Flash power modes registers
0x00 0A83	FSTATUS	Flash status register
0x00 0A84	FSTDBYWAIT	Flash sleep to standby wait register
0x00 0A85	FACTIVEWAIT	Flash standby to active wait register
0x00 0A86	FBANKWAIT	Flash read access wait state register
0x00 0A87	FOTPWAIT	OTP read access wait state register

- **FPWR: Save power by putting Flash/OTP to ‘Sleep’ or ‘Standby’ mode; Flash will automatically enter active mode if a Flash/OTP access is made**
- **FSTATUS: Various status bits (e.g. PWR mode)**
- **FSTDBYWAIT, FACTIVEWAIT: Specify # of delay cycles during wake-up from sleep to standby, and from standby to active, respectively. The delay is needed to let the flash stabilize. *Leave these registers set to their default maximum value.***

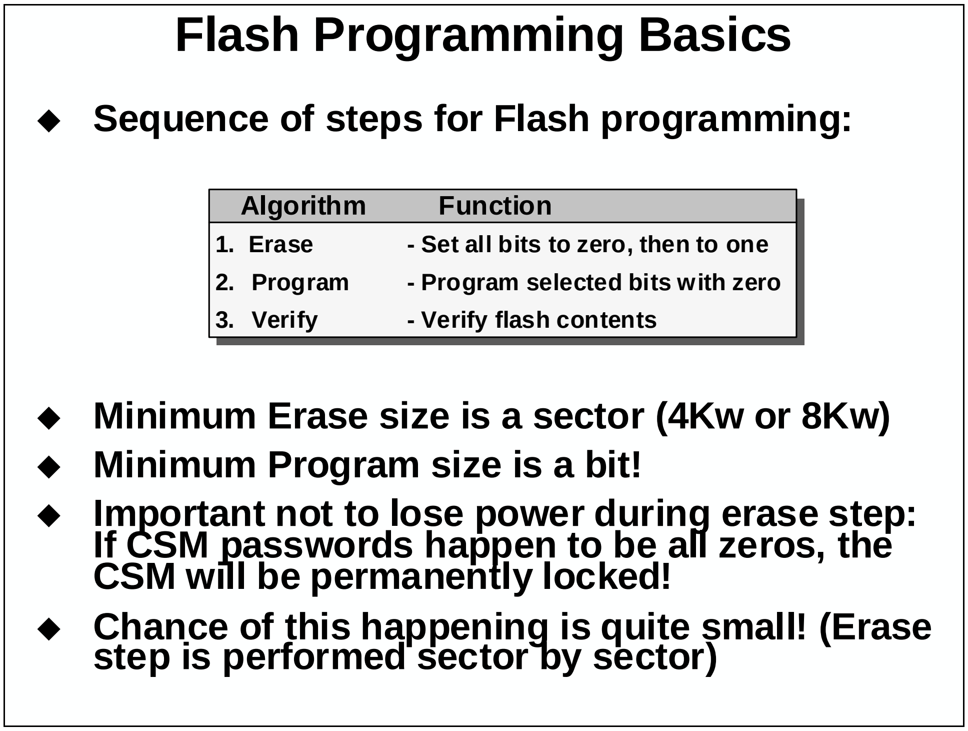
See the “*TMS320x2803x Piccolo System Control and Interrupts Reference Guide,” SPRUGL8, for more information*

Flash Programming



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

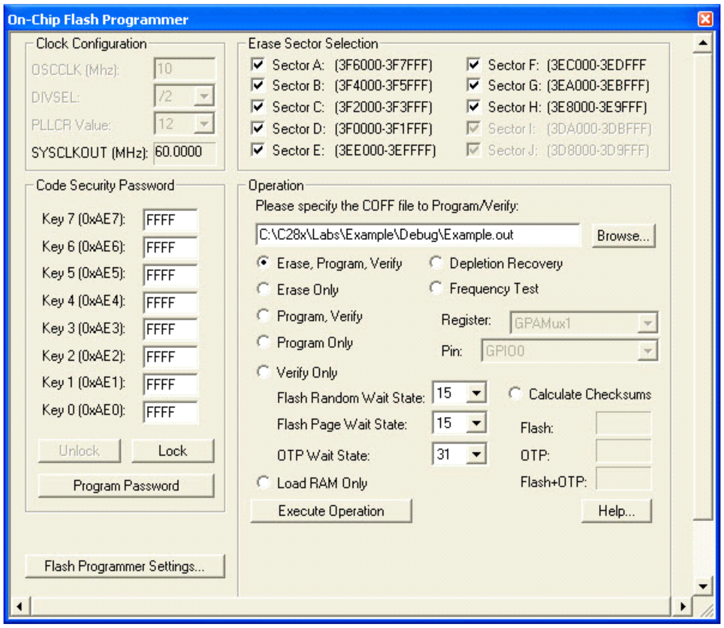
- Minimum Erase size is a sector (4Kw or 8Kw)
- Minimum Program size is a bit!
- Important not to lose power during erase step: If CSM passwords happen to be all zeros, the CSM will be permanently locked!
- Chance of this happening is quite small! (Erase step is performed sector by sector)

Flash Programming Utilities

- JTAG Emulator Based
  - Code Composer Studio Plug-in
  - 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)

Code Composer Studio Flash Plug-In



On-Chip Flash Programmer

Clock Configuration
OSCCLK (Mhz): 10
DIVSEL: /2
PLLCR Value: 12
SYSCLKOUT (MHz): 60.0000

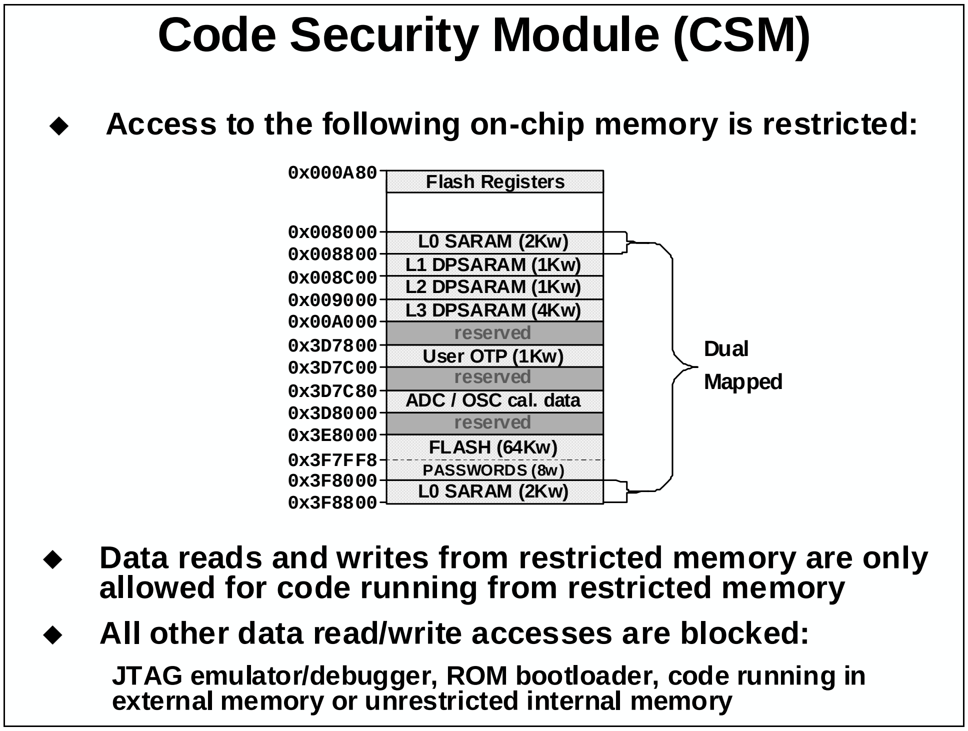
Erase Sector Selection
☑ Sector A: (3F6000-3F7FFF)
☑ Sector B: (3F4000-3F5FFF)
☑ Sector C: (3F2000-3F3FFF)
☑ Sector D: (3F0000-3F1FFF)
☑ Sector E: (3EE000-3EFFFF)
☑ Sector F: (3EC000-3EDFFF)
☑ Sector G: (3EA000-3EBFFF)
☑ Sector H: (3E8000-3E9FFF)
☑ Sector I: (3DA000-3DBFFF)
☑ Sector J: (3D8000-3D9FFF)

Code Security Password
Key 7 (0xAE7): FFFF
Key 6 (0xAE6): FFFF
Key 5 (0xAE5): FFFF
Key 4 (0xAE4): FFFF
Key 3 (0xAE3): FFFF
Key 2 (0xAE2): FFFF
Key 1 (0xAE1): FFFF
Key 0 (0xAE0): FFFF

Operation
Please specify the COFF file to Program/Verify:
C:\C28x\Labs\Example\Debug\Example.out

● Erase, Program, Verify
○ Erase Only
○ Program, Verify
○ Program Only
○ Verify Only
○ Depletion Recovery
○ Frequency Test
Register: GPAMux1
Pin: GPIO0
Flash Random Wait State: 15
Flash Page Wait State: 15
OTP Wait State: 31
○ Calculate Checksums
Flash:
OTP:
Flash+OTP:
○ Load RAM Only

Code Security Module (CSM)



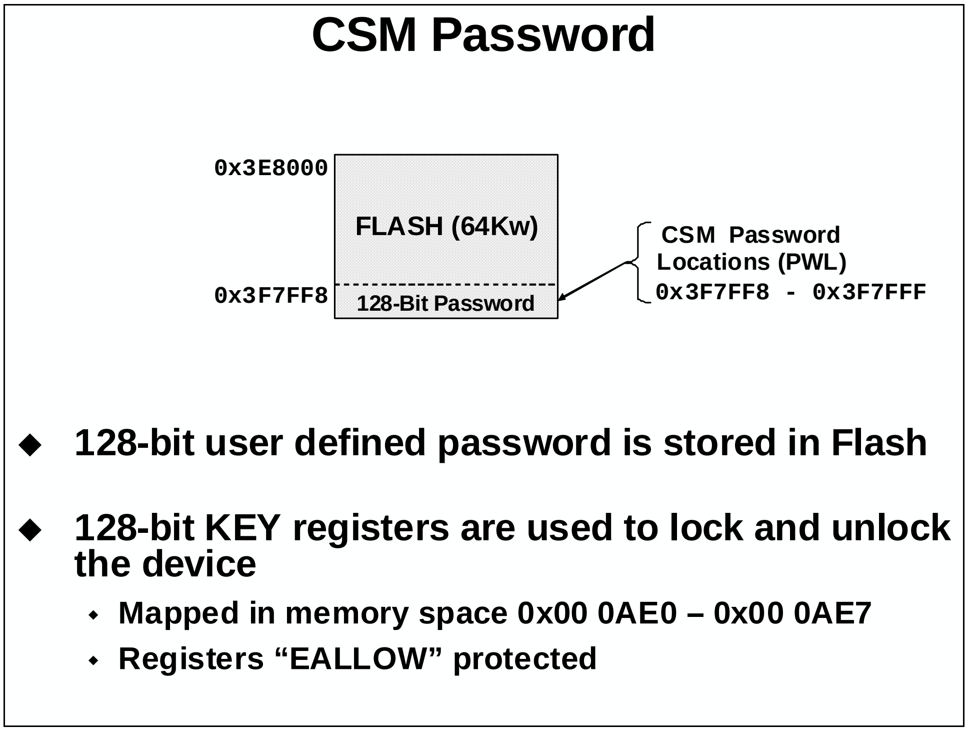
Code Security Module (CSM)

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

0x000A80	Flash Registers
0x008000	L0 SARAM (2Kw)
0x008800	L1 DPSARAM (1Kw)
0x008C00	L2 DPSARAM (1Kw)
0x009000	L3 DPSARAM (4Kw)
0x00A000	reserved
0x3D7800	User OTP (1Kw)
0x3D7C00	reserved
0x3D7C80	ADC / OSC cal. data
0x3D8000	reserved
0x3E8000	FLASH (64Kw)
0x3F7FF8	PASSWORDS (8w)
0x3F8000	L0 SARAM (2Kw)
0x3F8800	

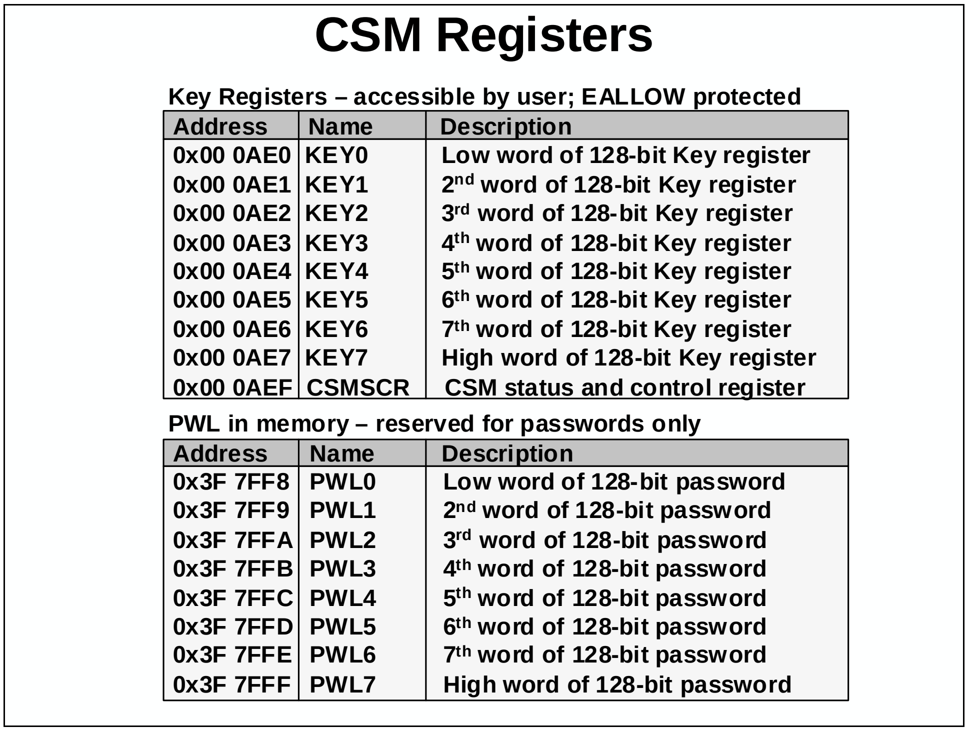
- Data reads and writes from restricted memory are only allowed for code running from restricted memory
- All other data read/write accesses are blocked:

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



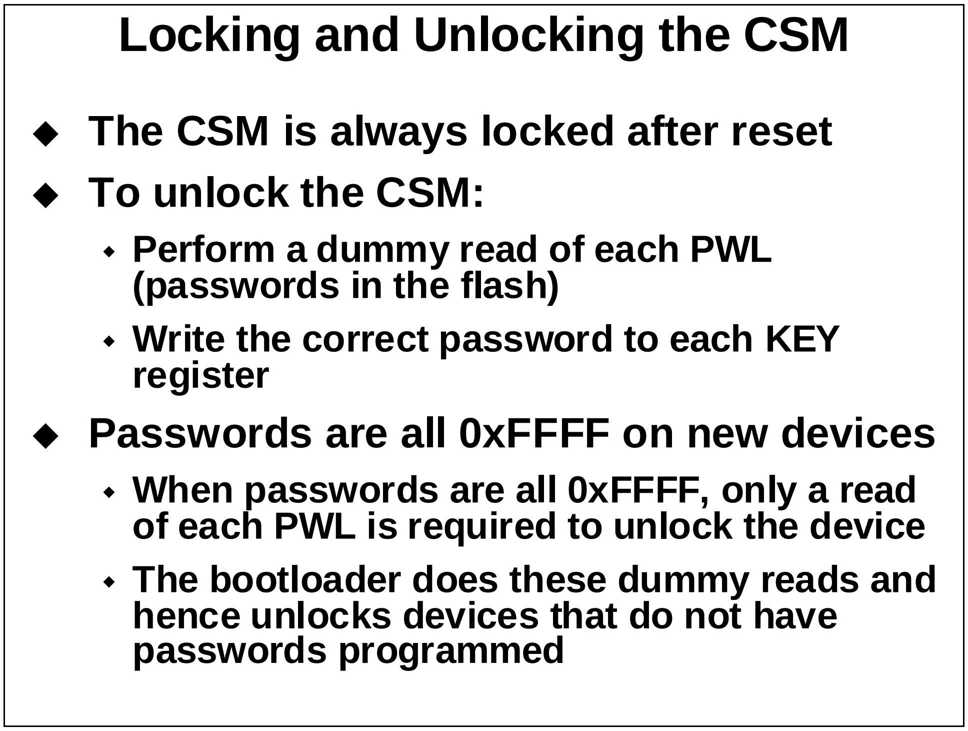
CSM Registers

Key Registers – accessible by user; EALLOW protected

Address	Name	Description
0x00 0AE0	KEY0	Low word of 128-bit Key register
0x00 0AE1	KEY1	2nd word of 128-bit Key register
0x00 0AE2	KEY2	3rd word of 128-bit Key register
0x00 0AE3	KEY3	4th word of 128-bit Key register
0x00 0AE4	KEY4	5th word of 128-bit Key register
0x00 0AE5	KEY5	6th word of 128-bit Key register
0x00 0AE6	KEY6	7th word of 128-bit Key register
0x00 0AE7	KEY7	High word of 128-bit Key register
0x00 0AEF	CSMSCR	CSM status and control register

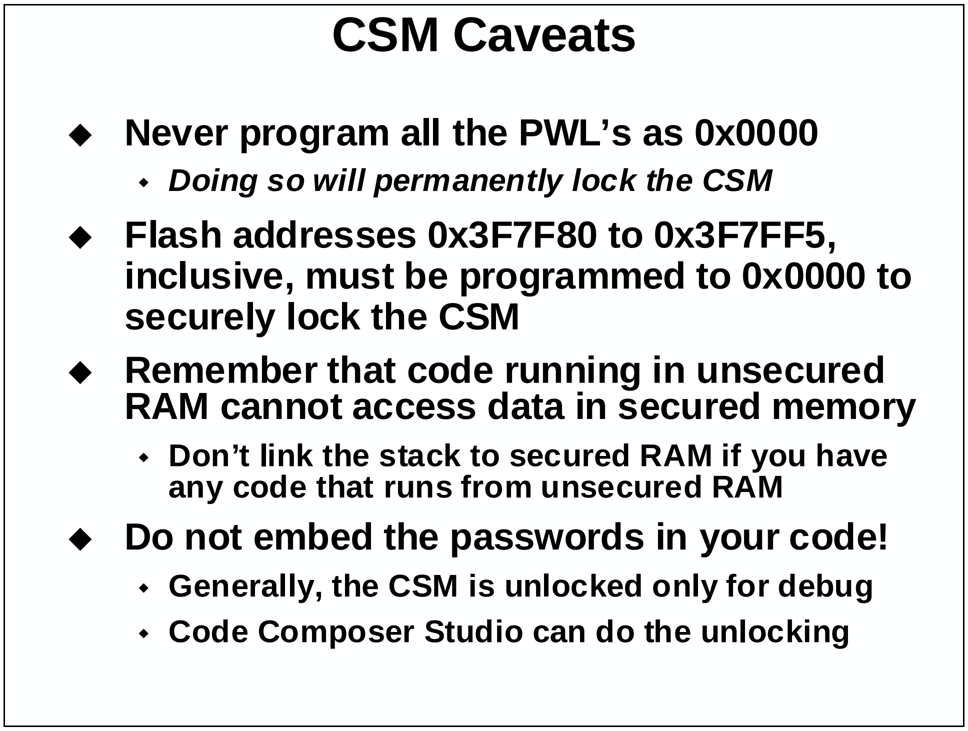
PWL in memory – reserved for passwords only

Address	Name	Description
0x3F 7FF8	PWL0	Low word of 128-bit password
0x3F 7FF9	PWL1	2nd word of 128-bit password
0x3F 7FFA	PWL2	3rd word of 128-bit password
0x3F 7FFB	PWL3	4th word of 128-bit password
0x3F 7FFC	PWL4	5th word of 128-bit password
0x3F 7FFD	PWL5	6th word of 128-bit password
0x3F 7FFE	PWL6	7th word of 128-bit password
0x3F 7FFF	PWL7	High word of 128-bit password



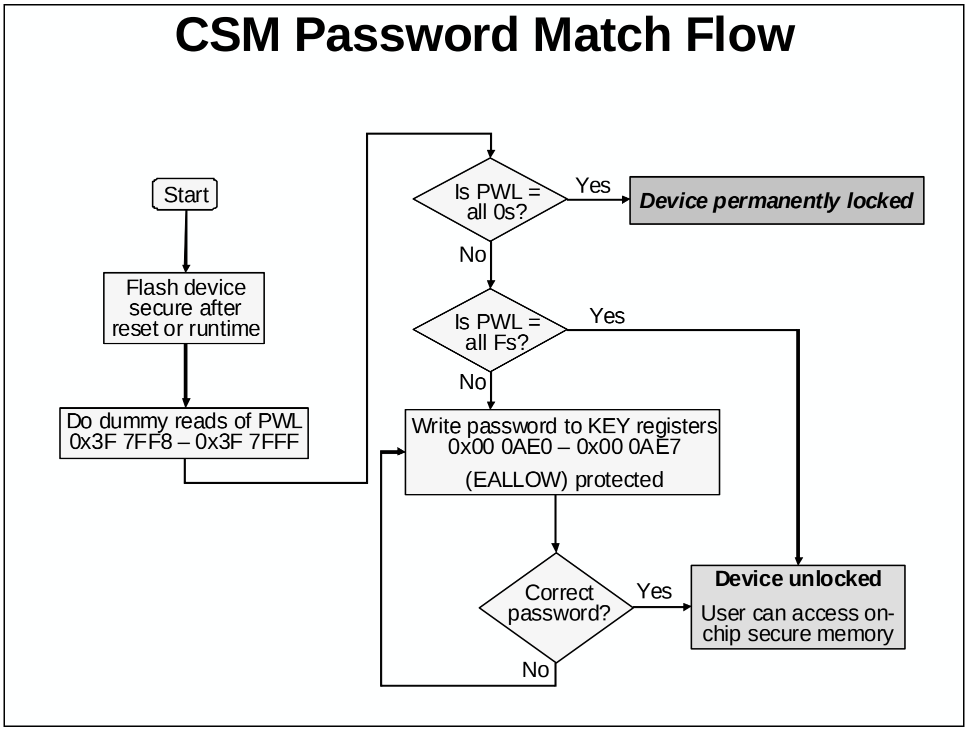
Locking and Unlocking the CSM

- The CSM is always locked after reset
- To unlock the CSM:
  - Perform a dummy read of each PWL (passwords in the flash)
  - Write the correct password to each KEY register
- Passwords are all 0xFFFF on new devices
  - When passwords are all 0xFFFF, only a read of each PWL is required to unlock the device
  - The bootloader does these dummy reads and hence unlocks devices that do not have passwords programmed



CSM Caveats

- **Never program all the PWL’s as 0x0000**
  - ***Doing so will permanently lock the CSM***
- **Flash addresses 0x3F7F80 to 0x3F7FF5, inclusive, must be programmed to 0x0000 to securely lock the CSM**
- **Remember that code running in unsecured RAM cannot access data in secured memory**
  - **Don’t link the stack to secured RAM if you have any code that runs from unsecured RAM**
- **Do not embed the passwords in your code!**
  - **Generally, the CSM is unlocked only for debug**
  - **Code Composer Studio can do the unlocking**



CSM Password Match Flow

Start

Flash device secure after reset or runtime

Do dummy reads of PWL 0x3F 7FF8 – 0x3F 7FFF

Is PWL = all 0s?

Yes

Device permanently locked

No

Is PWL = all Fs?

Yes

No

Write password to KEY registers 0x00 0AE0 – 0x00 0AE7 (EALLOW) protected

Correct password?

Yes

Device unlocked

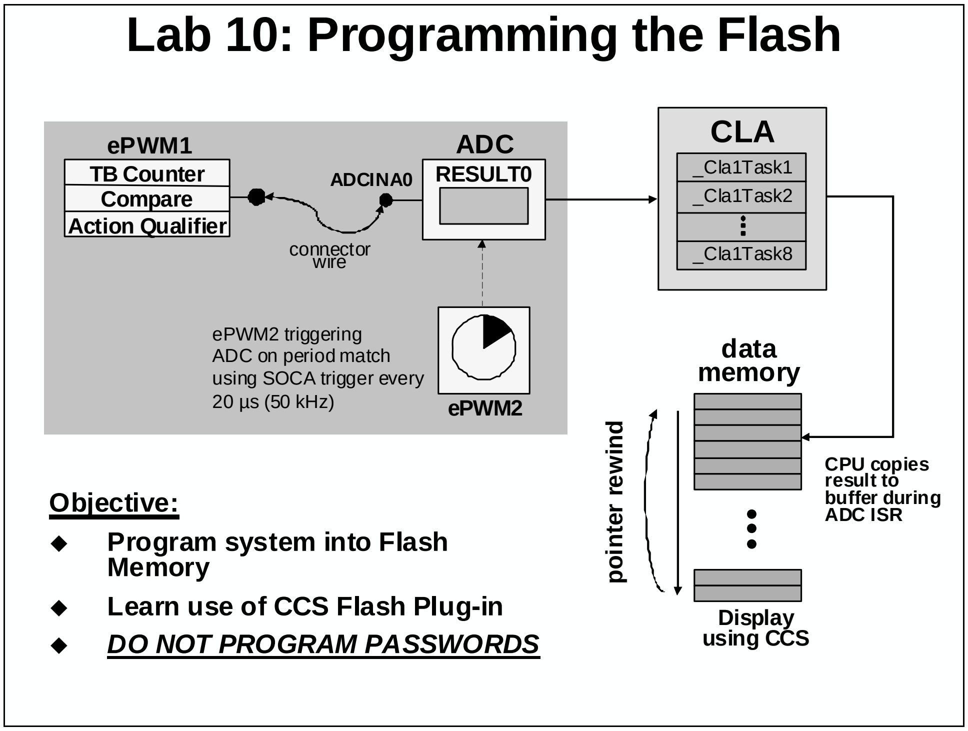
User can access on-chip secure memory

No

Lab 10: Programming the Flash

➢ Objective

The objective of this lab is to program and execute code from the on-chip flash memory. The TMS320F28035 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.



Lab 10: Programming the Flash

Objective:

- ◆ **Program system into Flash Memory**
- ◆ **Learn use of CCS Flash Plug-in**
- ◆ ***DO NOT PROGRAM PASSWORDS***

➢ Procedure

Project File

1. A project named `Lab10.pjt` has been created for this lab. Open the project by clicking on `Project → Open…` and look in `C:\C28x\Labs\Lab10`. All Build Options have been configured the same as the previous lab. The files used in this lab are:

```
Adc.c
Cla_10_12.c
ClaTasks.asm
CodeStartBranch.asm
DefaultIsr_9_10.c
DelayUs.asm
DSP2803x_GlobalVariableDefs.c
DSP2803x_Headers_nonBIOS.cmd
ECap_7_8_9_10_12.c
EPwm_7_8_9_10_12.c
Filter.c
Flash.c
Gpio.c
Lab_10.cmd
Main_10.c
Passwords.asm
PieCtrl_5_6_7_8_9_10.c
PieVect_5_6_7_8_9_10.c
SysCtrl.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 F28035 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.

2. Open and inspect the linker command file `Lab_10.cmd`. Notice that a memory block named `FLASH_ABCDEFGH` has been been created at origin = 0x3E8000, length = 0x00FF80 on Page 0. This flash memory block length has been selected to avoid conflicts with other required flash memory spaces. See the reference slide at the end of this lab exercise for further details showing the address origins and lengths of the various memory blocks used.

3. Edit `Lab_10.cmd` to link the following compiler sections to on-chip flash memory block `FLASH_ABCDEFGH`:

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

4. In `Lab_10.cmd` notice that the section named `“IQmath”` is an initialized section that needs to load to and run from flash. Previously the `“IQmath”` section was linked to `L0SARAM`. Edit `Lab_10.cmd` so that this section is now linked to `FLASH_ABCDEFGH`. Save your work and close the file.

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_5_6_7_8_9_10.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. Add `Flash.c` to the project.

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

8. The `“secureRamFuncs”` section will be linked using the user linker command file `Lab_10.cmd`. Open and inspect `Lab_10.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 size, and run start addresses.

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

9. Open and inspect `Main_10.c`. Notice that the memory copy function memcpy() is being used to copy the section `“secureRamFuncs”`, which contains the initialization function for the flash control registers.

10. Add a line of code in `main()` to call the `InitFlash()` function. There are no passed parameters or return values. You just type

    ```
    InitFlash();
    ```

    at the desired spot in `main()`.

Code Security Module and Passwords

The CSM module provides protection against unwanted copying (i.e. pirating!) of your code from flash, OTP memory, and the L0, L1, L2 and L3 RAM blocks. The CSM uses a 128-bit password made up of 8 individual 16-bit words. They are located in flash at addresses 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`.

11. Add `Passwords.asm` to the project.

12. 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).

13. Open `Lab_10.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 F28035 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`.

14. Open and inspect `CodeStartBranch.asm`. This file creates an initialized section named `“codestart”` that contains a long branch to the C-environment setup routine. This section needs to be linked to a block of memory named `BEGIN_FLASH`.

15. In the earlier lab exercises, the section `“codestart”` was directed to the memory named `BEGIN_M0`. Edit `Lab_10.cmd` so that the section `“codestart”` will be directed to `BEGIN_FLASH`. Save your work and close the opened 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.

Initializing the CLA

Previously, the named section `“Cla1Prog”` containing the CLA program tasks was linked directly to the CPU memory block `L3DPSARAM` for both load and run purposes. At runtime, all the code did was map the `L3DPSARAM` block to the CLA program memory space during CLA initialization. For an embedded application, the CLA program tasks are linked to load to flash and run from RAM. At runtime, the CLA program tasks must be copied from flash to `L3DPSARAM`. The memory copy function *memcpy()* will once again be used to perform the copy. After the copy is performed, the `L3DPSARAM` block will then be mapped to CLA program memory space as was done in the earlier lab.

16. Open and inspect `Lab_10.cmd`. Notice that the named section `“Cla1Prog”` will now load to flash (load address) but will run from `L3DPSARAM` (run address). The linker will also be used to generate symbols for the load start, load size, and run start addresses.

17. Open `Cla_10_12.c` and notice that the memory copy function memcpy() is being used to copy the CLA program code from flash to `L3DPSARAM` using the symbols generated by the linker. Just after the copy the `Cla1Regs` structure is used to configure the `L3DPSARAM` block as CLA program memory space. Close the inspected files.

Build – Lab.out

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

    CCS has a feature that automatically steps over functions without debug information. This can be useful for accelerating the debug process provided that you are not interested in debugging the function that is being stepped-over. While single-stepping in this lab exercise we do not want to step-over any functions. Therefore, select the `“Debug Properties”` tab. *Uncheck* `“Step over functions without debug information when source stepping”`, then click `OK`.

19. Click the `“Build”` button to generate the `Lab.out` file to be used with the CCS Flash Plug-in.

CCS Flash Plug-in

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

    `Tools → F28xx On-Chip Flash Programmer`

21. A Clock Configuration window *may* open. If needed, in the Clock Configuration window set “OSCCLK (MHz):” to 10, “DIVSEL:” to /2, and “PLLCR Value:” to 12. Then click `OK`. In the next Flash Programmer Settings window confirm that the selected DSP device to program is F28035 and all options have been checked. Click `OK`.

22. The CCS Flash Programmer uses the Piccolo™ 10 MHz internal oscillator as the device clock during programming. Confirm the "Clock Configuration" in the upper left corner 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.

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

24. We will not be using the plug-in to program the "Code Security Password". ***Do not modify the Code Security Password fields.*** They should remain as all 0xFFFF.

25. In the "Operation" block, notice that the "COFF file to Program/Verify" field automatically defaults to the current `.out` file. Check to be sure that "Erase, Program, Verify" is selected. We will be using the default wait states, as shown on the slide in this module. The selection for wait-states only affects the verify step, and makes little noticeable difference even if you reduce the wait-states.

26. Click "Execute Operation" to program the flash memory. Watch the programming status update in the plug-in window.

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

Running the Code – Using CCS

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

    `File → Load Symbols → Load Symbols Only…`

    and select `Lab10.out` in the `Debug` folder.

29. Reset the CPU. The program counter should now be at 0x3FF8A1, which is the start of the bootloader in the Boot ROM.

30. Under `GEL` 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 0x3F7FF6.

31. Single-Step <F11> through the bootloader code until you arrive at the beginning of the codestart section in the `CodeStartBranch.asm` file. (Be patient, it will take about 125 single-steps). Notice that we have placed some code in `CodeStartBranch.asm` to give an option to first disable the watchdog, if selected.

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

33. Now do `Debug → Go Main`. The code should stop at the beginning of your `main()` routine. If you got to that point succesfully, it confirms that the flash has been

programmed properly, that the bootloader is properly configured for jump to flash mode, and that the codestart section has been linked to the proper address.

34. You can now RUN the CPU, and you should observe the LED on the ControlCARD 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.

35. HALT the CPU.

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

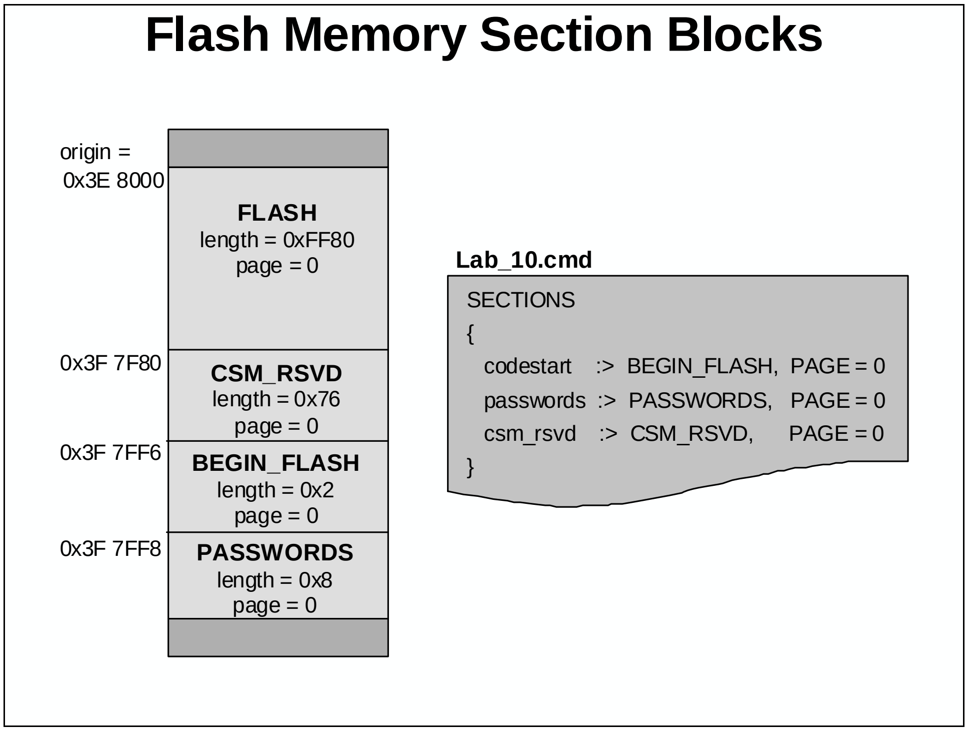
36. Close Code Composer Studio.

37. Disconnect the USB cable (emulator) from the Docking Station (i.e. remove power from the ControlCARD).

38. Re-connect the USB cable to the Docking Station to power the ControlCARD. The LED should be blinking, showing that the code is now running from flash memory.

End of Exercise

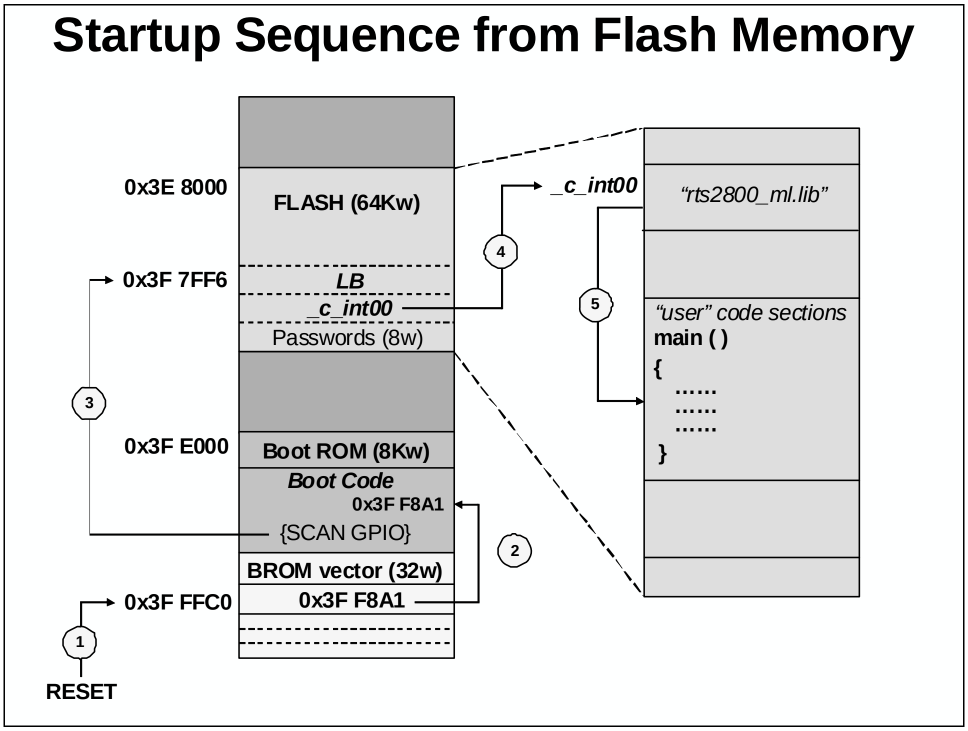
Lab 10 Reference: Programming the Flash



Flash Memory Section Blocks

Lab_10.cmd

```
SECTIONS
{
  codestart   :>  BEGIN_FLASH,  PAGE = 0
  passwords  :>  PASSWORDS,   PAGE = 0
  csm_rsvd    :>  CSM_RSVD,      PAGE = 0
}
```



Startup Sequence from Flash Memory

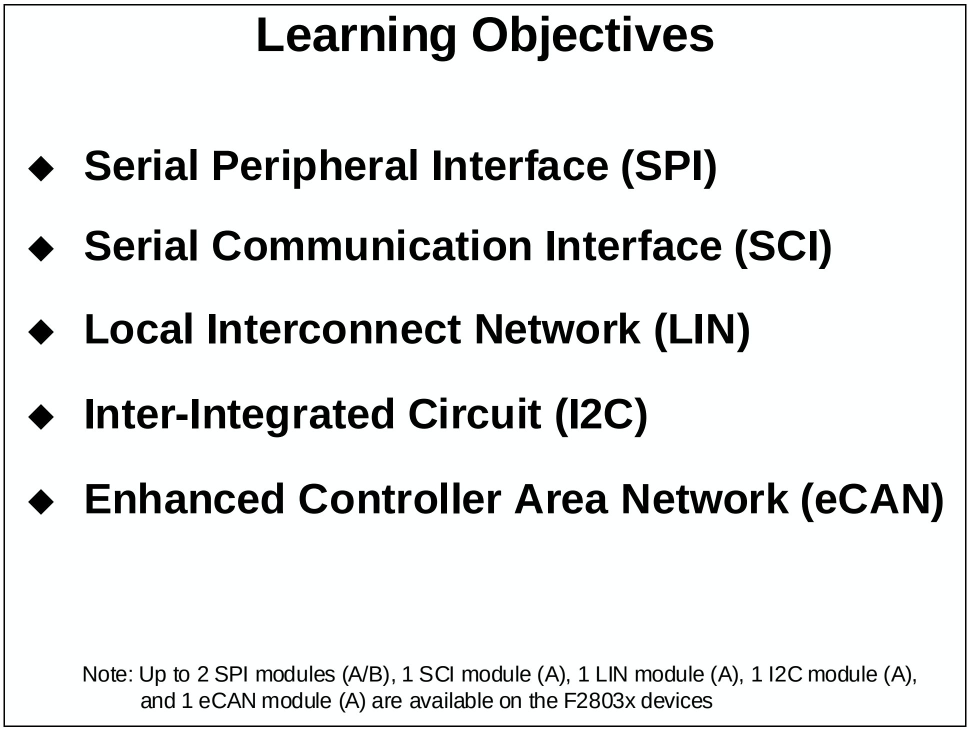
Communications

Introduction

The TMS320C28x contains features that allow several methods of communication and data exchange between the C28x and other devices. Many of the most commonly used communications techniques are presented in this module.

The intent of this module is not to give exhaustive design details of the communication peripherals, but rather to provide an overview of the features and capabilities. Once these features and capabilities are understood, additional information can be obtained from various resources such as documentation, as needed. This module will cover the basic operation of the communication peripherals, as well as some basic terms and how they work.

Learning Objectives



Learning Objectives

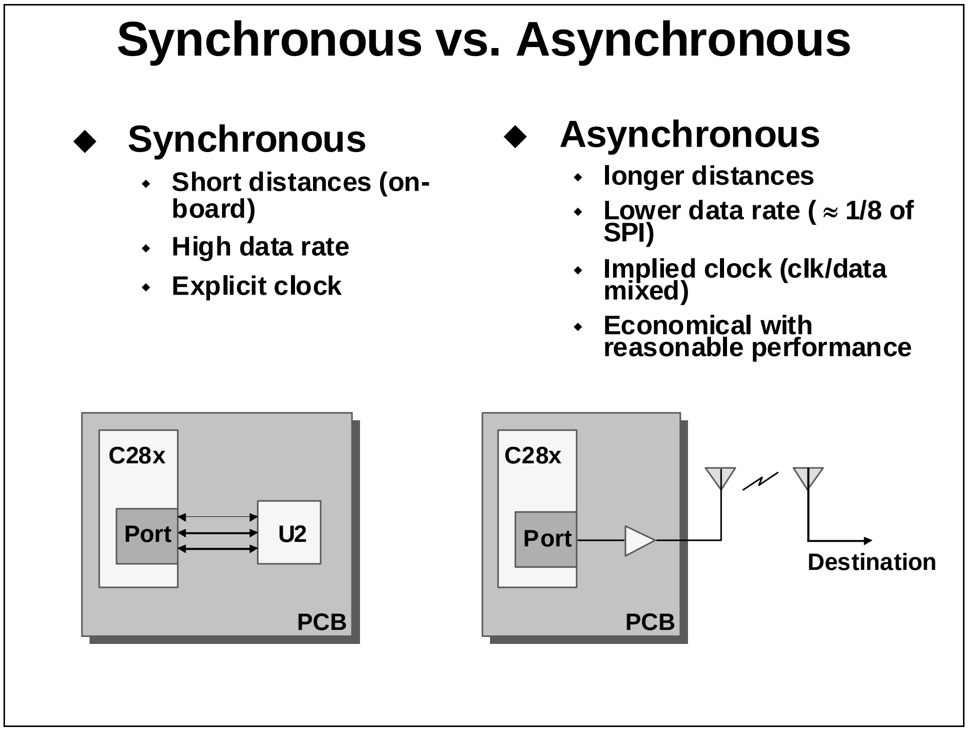
- **Serial Peripheral Interface (SPI)**
- **Serial Communication Interface (SCI)**
- **Local Interconnect Network (LIN)**
- **Inter-Integrated Circuit (I2C)**
- **Enhanced Controller Area Network (eCAN)**

Note: Up to 2 SPI modules (A/B), 1 SCI module (A), 1 LIN module (A), 1 I2C module (A), and 1 eCAN module (A) are available on the F2803x devices

Module Topics

Communications Techniques

Several methods of implementing a TMS320C28x communications system are possible. The method selected for a particular design should reflect the method that meets the required data rate at the lowest cost. Various categories of interface are available and are summarized in the learning objective slide. Each will be described in this module.



Synchronous vs. Asynchronous

- **Synchronous**
  - Short distances (on-board)
  - High data rate
  - Explicit clock
- **Asynchronous**
  - longer distances
  - Lower data rate (≈ 1/8 of SPI)
  - Implied clock (clk/data mixed)
  - Economical with reasonable performance

Serial ports provide a simple, hardware-efficient means of high-level communication between devices. Like the GPIO pins, they may be used in stand-alone or multiprocessing systems.

In a multiprocessing system, they are an excellent choice when both devices have an available serial port and the data rate requirement is relatively low. Serial interface is even more desirable when the devices are physically distant from each other because the inherently low number of wires provides a simpler interconnection.

Serial ports require separate lines to implement, and they do not interfere in any way with the data and address lines of the processor. The only overhead they require is to read/write new words from/to the ports as each word is received/transmitted. This process can be performed as a short interrupt service routine under hardware control, requiring only a few cycles to maintain.

The C28x family of devices have both synchronous and asynchronous serial ports. Detailed features and operation will be described next.

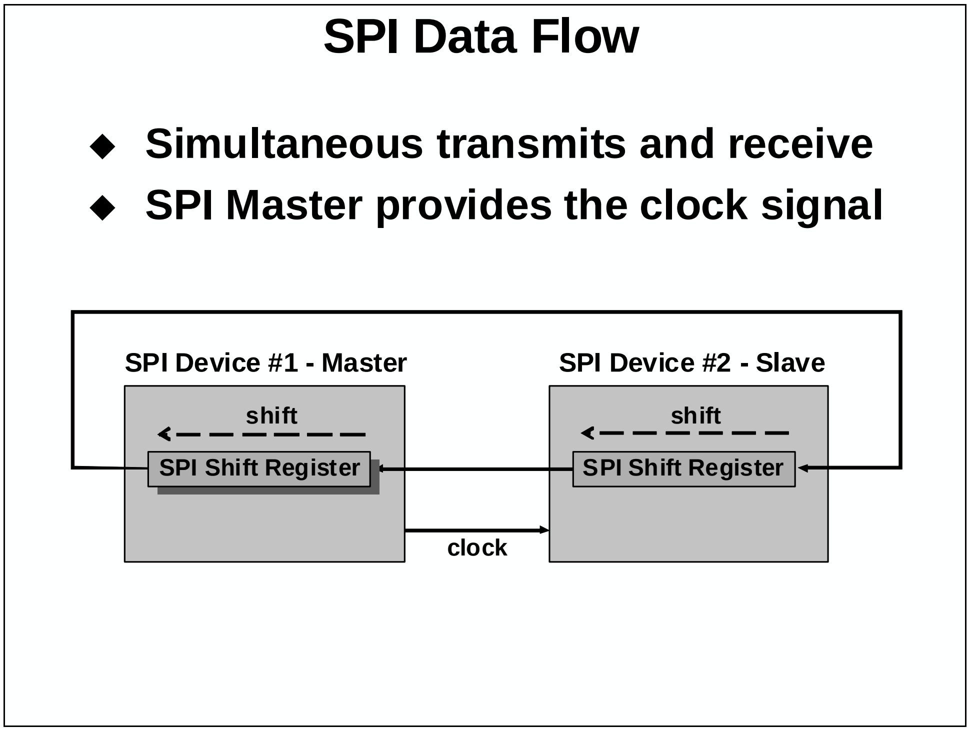
Serial Peripheral Interface (SPI)

The SPI module is a synchronous serial I/O port that shifts a serial bit stream of variable length and data rate between the C28x and other peripheral devices. During data transfers, one SPI device must be configured as the transfer MASTER, and all other devices configured as SLAVES. The master drives the transfer clock signal for all SLAVES on the bus. SPI communications can be implemented in any of three different modes:

- MASTER sends data, SLAVES send dummy data
- MASTER sends data, one SLAVE sends data
- MASTER sends dummy data, one SLAVE sends data

In its simplest form, the SPI can be thought of as a programmable shift register. Data is shifted in and out of the SPI through the SPIDAT register. Data to be transmitted is written directly to the SPIDAT register, and received data is latched into the SPIBUF register for reading by the CPU. This allows for double-buffered receive operation, in that the CPU need not read the current received data from SPIBUF before a new receive operation can be started. However, the CPU must read SPIBUF before the new operation is complete of a receiver overrun error will occur. In addition, double-buffered transmit is not supported: the current transmission must be complete before the next data character is written to SPIDAT or the current transmission will be corrupted.

The Master can initiate a data transfer at any time because it controls the SPICLK signal. The software, however, determines how the Master detects when the Slave is ready to broadcast.

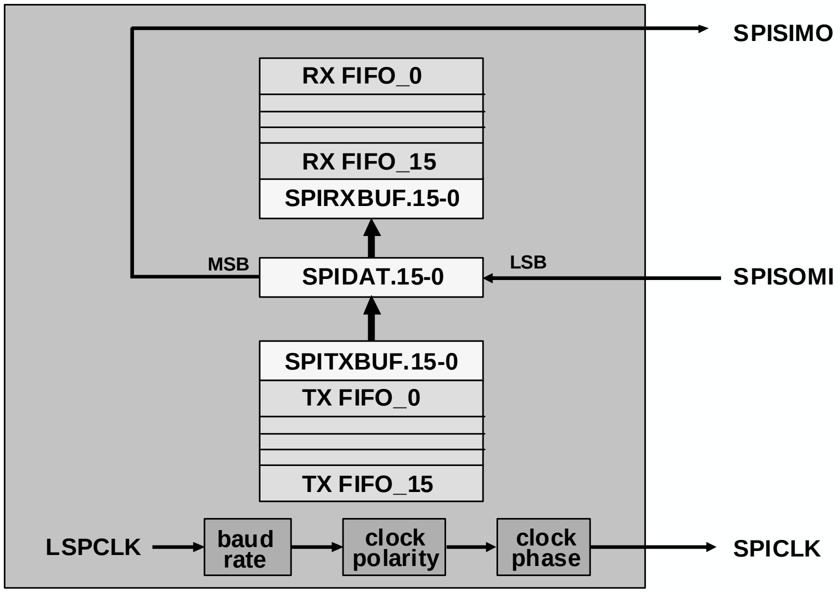


SPI Data Flow

- ◆ Simultaneous transmits and receive
- ◆ SPI Master provides the clock signal

SPI Block Diagram

C28x - SPI Master Mode Shown

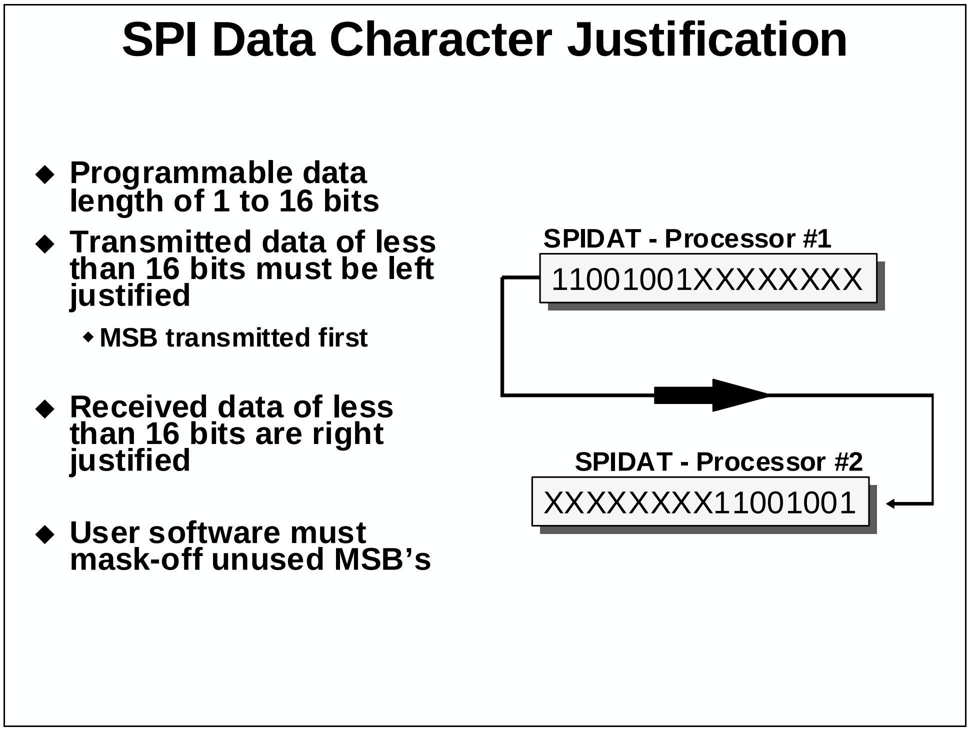


SPI Transmit / Receive Sequence

1. Slave writes data to be sent to its shift register (SPIDAT)
2. Master writes data to be sent to its shift register (SPIDAT or SPITXBUF)
3. Completing Step 2 automatically starts SPICLK signal of the Master
4. MSB of the Master’s shift register (SPIDAT) is shifted out, and LSB of the Slave’s shift register (SPIDAT) is loaded
5. Step 4 is repeated until specified number of bits are transmitted
6. SPIDAT register is copied to SPIRXBUF register
7. SPI INT Flag bit is set to 1
8. An interrupt is asserted if SPI INT ENA bit is set to 1
9. If data is in SPITXBUF (either Slave or Master), it is loaded into SPIDAT and transmission starts again as soon as the Master’s SPIDAT is loaded

Since data is shifted out of the SPIDAT register MSB first, transmission characters of less than 16 bits must be left-justified by the CPU software prior to be written to SPIDAT.

Received data is shifted into SPIDAT from the left, MSB first. However, the entire sixteen bits of SPIDAT is copied into SPIBUF after the character transmission is complete such that received characters of less than 16 bits will be right-justified in SPIBUF. The non-utilized higher significance bits must be masked-off by the CPU software when it interprets the character. For example, a 9 bit character transmission would require masking-off the 7 MSB's.



SPI Data Character Justification

- Programmable data length of 1 to 16 bits
- Transmitted data of less than 16 bits must be left justified
  - MSB transmitted first
- Received data of less than 16 bits are right justified
- User software must mask-off unused MSB's

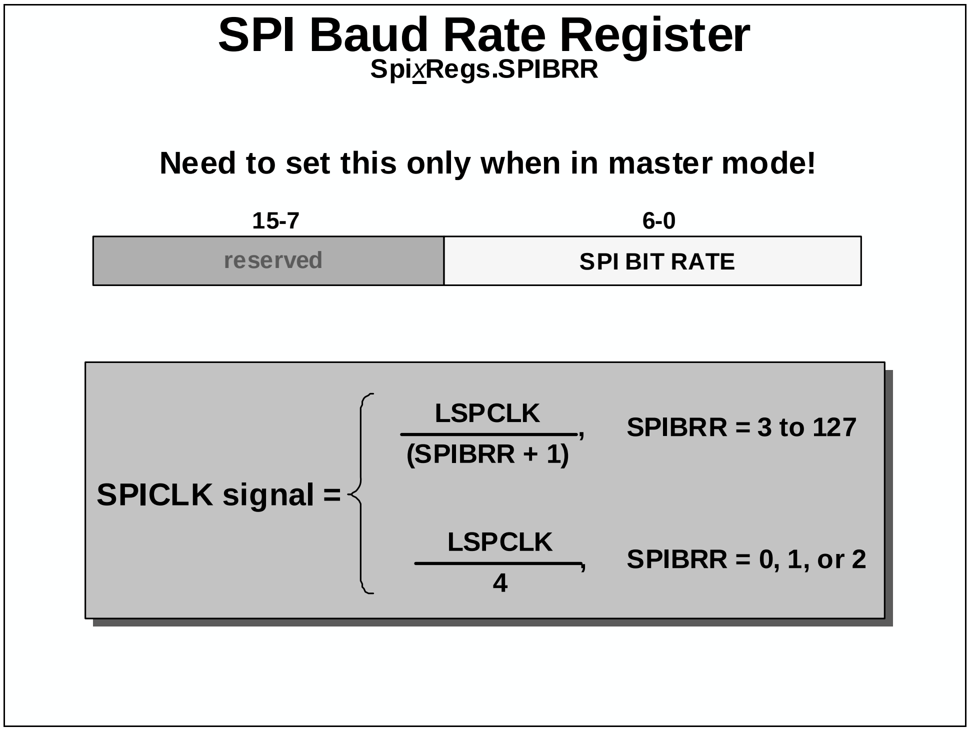
SPIDAT - Processor #1

11001001XXXXXXXX

SPIDAT - Processor #2

XXXXXXXX11001001

SPI Registers



SPI Baud Rate Register

SpixRegs.SPIBRR

Need to set this only when in master mode!

15-7	6-0
reserved	SPI BIT RATE

$$\text{SPICLK signal} = \begin{cases} \dfrac{\text{LSPCLK}}{(\text{SPIBRR} + 1)}, & \text{SPIBRR} = 3 \text{ to } 127 \\ \dfrac{\text{LSPCLK}}{4}, & \text{SPIBRR} = 0, 1, \text{ or } 2 \end{cases}$$

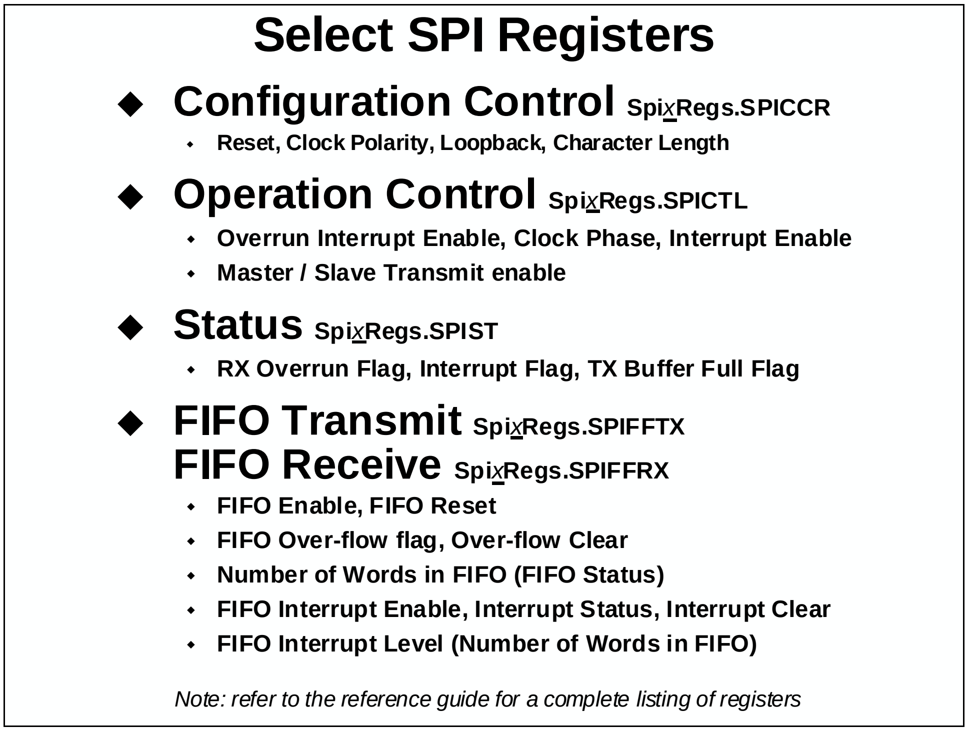
Baud Rate Determination: The Master specifies the communication baud rate using its baud rate register (SPIBRR.6-0):

- For SPIBRR = 3 to 127: SPI Baud Rate = $\dfrac{LSPCLK}{(SPIBRR+1)}$ bits/sec
- For SPIBRR = 0, 1, or 2: SPI Baud Rate = $\dfrac{LSPCLK}{4}$ bits/sec

From the above equations, one can compute

Maximum data rate = 25 Mbps @ 100 MHz

Character Length Determination: The Master and Slave must be configured for the same transmission character length. This is done with bits 0, 1, 2 and 3 of the configuration control register (SPICCR.3-0). These four bits produce a binary number, from which the character length is computed as binary + 1 (e.g. SPICCR.3-0 = `0010` gives a character length of 3).

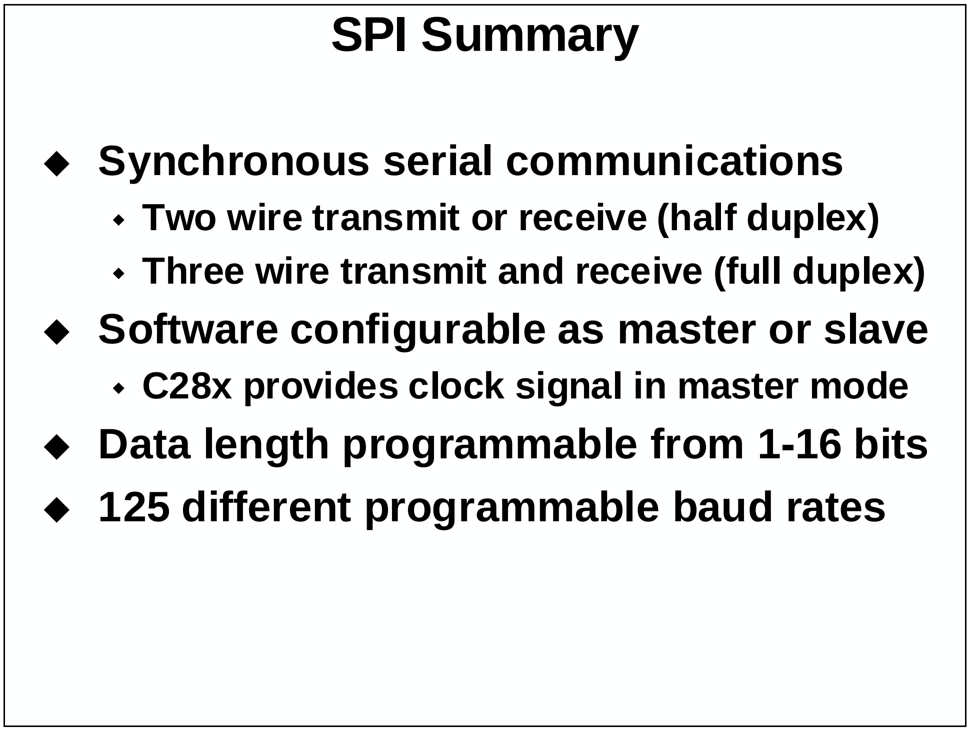


Select SPI Registers

- **Configuration Control** Spi*x*Regs.SPICCR
  - Reset, Clock Polarity, Loopback, Character Length
- **Operation Control** Spi*x*Regs.SPICTL
  - Overrun Interrupt Enable, Clock Phase, Interrupt Enable
  - Master / Slave Transmit enable
- **Status** Spi*x*Regs.SPIST
  - RX Overrun Flag, Interrupt Flag, TX Buffer Full Flag
- **FIFO Transmit** Spi*x*Regs.SPIFFTX
  FIFO Receive Spi*x*Regs.SPIFFRX
  - FIFO Enable, FIFO Reset
  - FIFO Over-flow flag, Over-flow Clear
  - Number of Words in FIFO (FIFO Status)
  - FIFO Interrupt Enable, Interrupt Status, Interrupt Clear
  - FIFO Interrupt Level (Number of Words in FIFO)

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

SPI Summary



SPI Summary

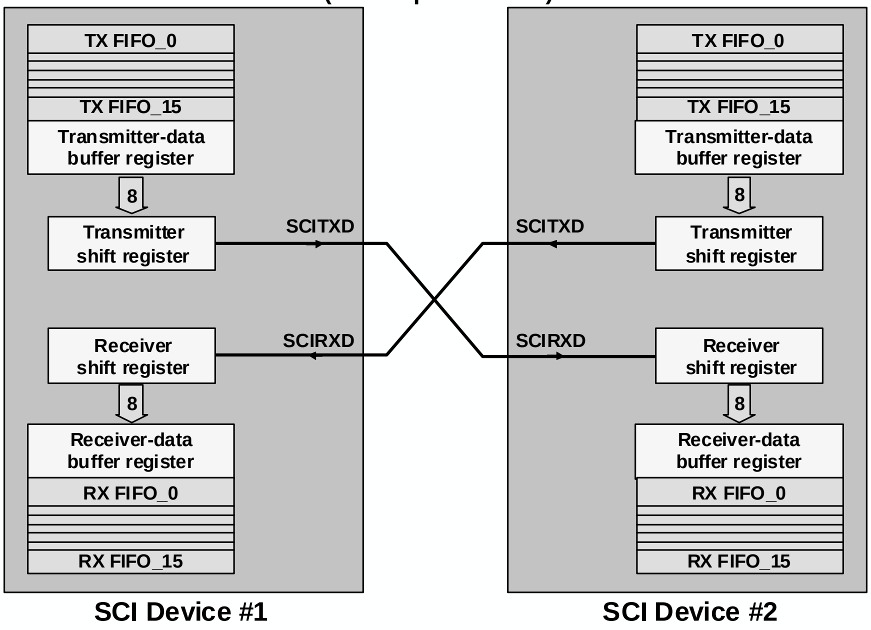
- **Synchronous serial communications**
  - Two wire transmit or receive (half duplex)
  - Three wire transmit and receive (full duplex)
- **Software configurable as master or slave**
  - C28x provides clock signal in master mode
- **Data length programmable from 1-16 bits**
- **125 different programmable baud rates**

Serial Communications Interface (SCI)

The SCI module is a serial I/O port that permits Asynchronous communication between the C28x and other peripheral devices. The SCI transmit and receive registers are both double-buffered to prevent data collisions and allow for efficient CPU usage. In addition, the C28x SCI is a full duplex interface which provides for simultaneous data transmit and receive. Parity checking and data formatting is also designed to be done by the port hardware, further reducing software overhead.

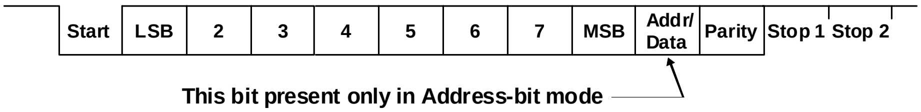
SCI Pin Connections

(Full Duplex Shown)

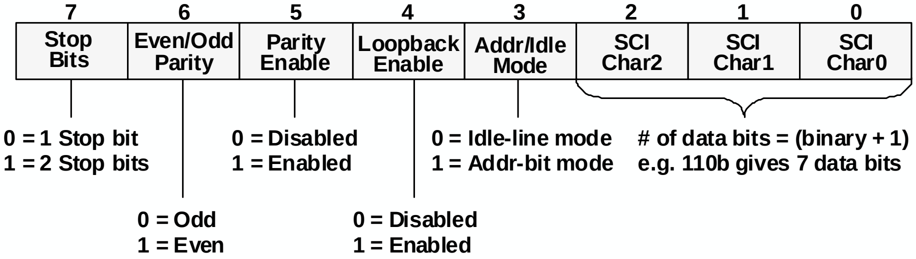


SCI Data Format

NRZ (non-return to zero) format



Communications Control Register (Sci*x*Regs.SCICCR)

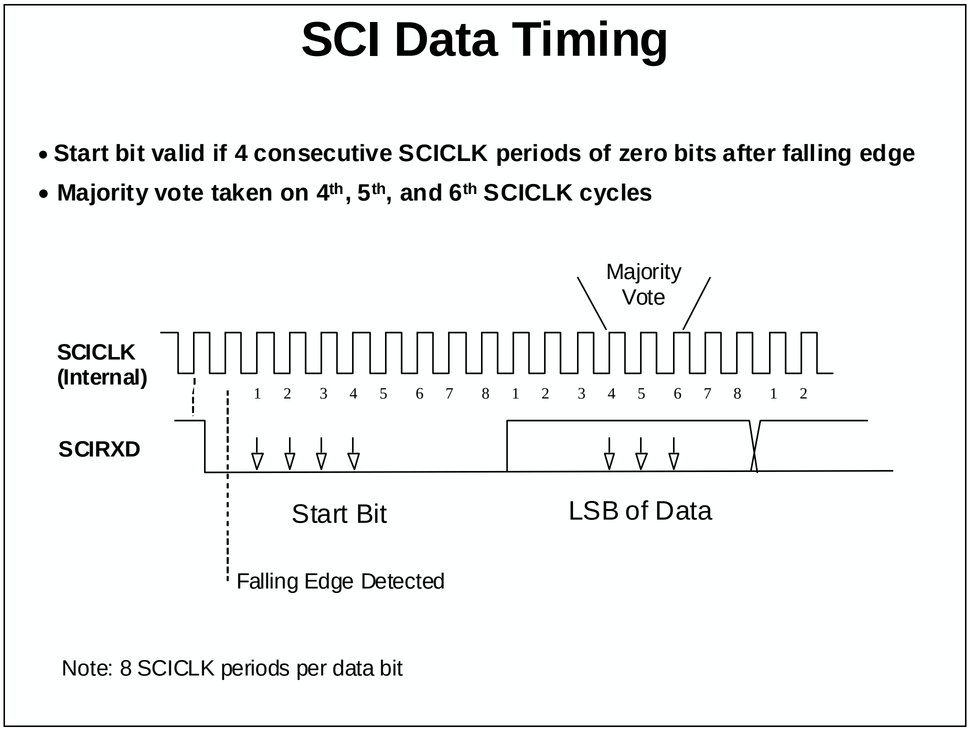


7	6	5	4	3	2	1	0
Stop Bits	Even/Odd Parity	Parity Enable	Loopback Enable	Addr/Idle Mode	SCI Char2	SCI Char1	SCI Char0

The basic unit of data is called a **character** and is 1 to 8 bits in length. Each character of data is formatted with a start bit, 1 or 2 stop bits, an optional parity bit, and an optional address/data bit. A character of data along with its formatting bits is called a **frame**. Frames are organized into groups called blocks. If more than two serial ports exist on the SCI bus, a block of data will usually begin with an address frame which specifies the destination port of the data as determined by the user's protocol.

The start bit is a low bit at the beginning of each frame which marks the beginning of a frame. The SCI uses a NRZ (Non-Return-to-Zero) format which means that in an inactive state the SCIRX and SCITX lines will be held high. Peripherals are expected to pull the SCIRX and SCITX lines to a high level when they are not receiving or transmitting on their respective lines.

When configuring the SCICCR, the SCI port should first be held in an inactive state. This is done using the SW RESET bit of the SCI Control Register 1 (SCICTL1.5). Writing a 0 to this bit initializes and holds the SCI state machines and operating flags at their reset condition. The SCICCR can then be configured. Afterwards, re-enable the SCI port by writing a 1 to the SW RESET bit. At system reset, the SW RESET bit equals 0.

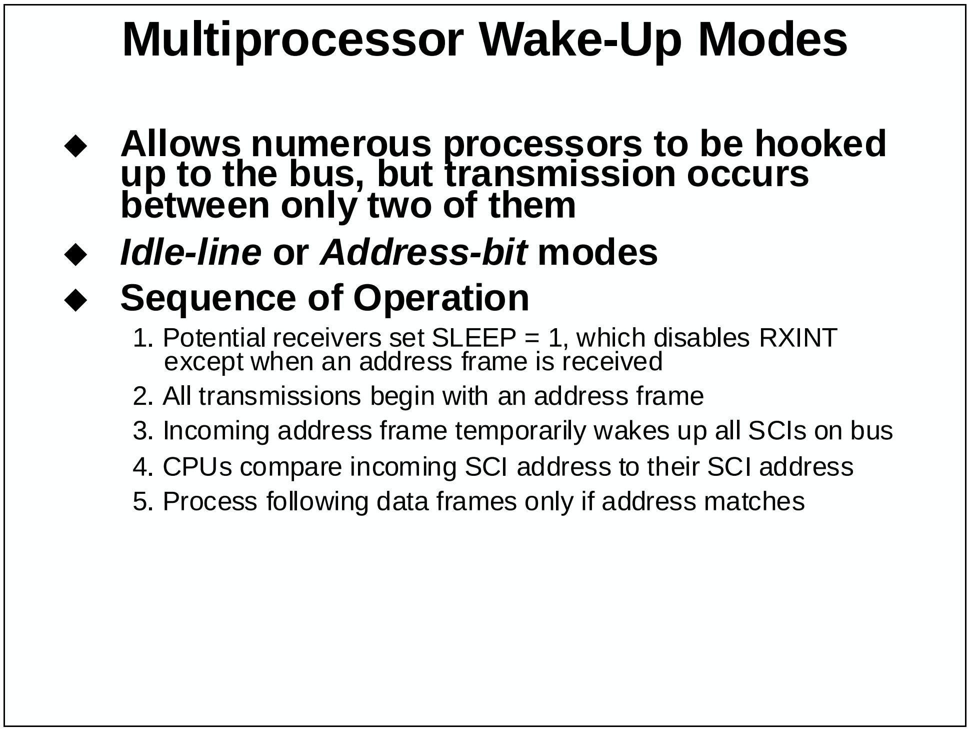


SCI Data Timing

- Start bit valid if 4 consecutive SCICLK periods of zero bits after falling edge
- Majority vote taken on 4th, 5th, and 6th SCICLK cycles

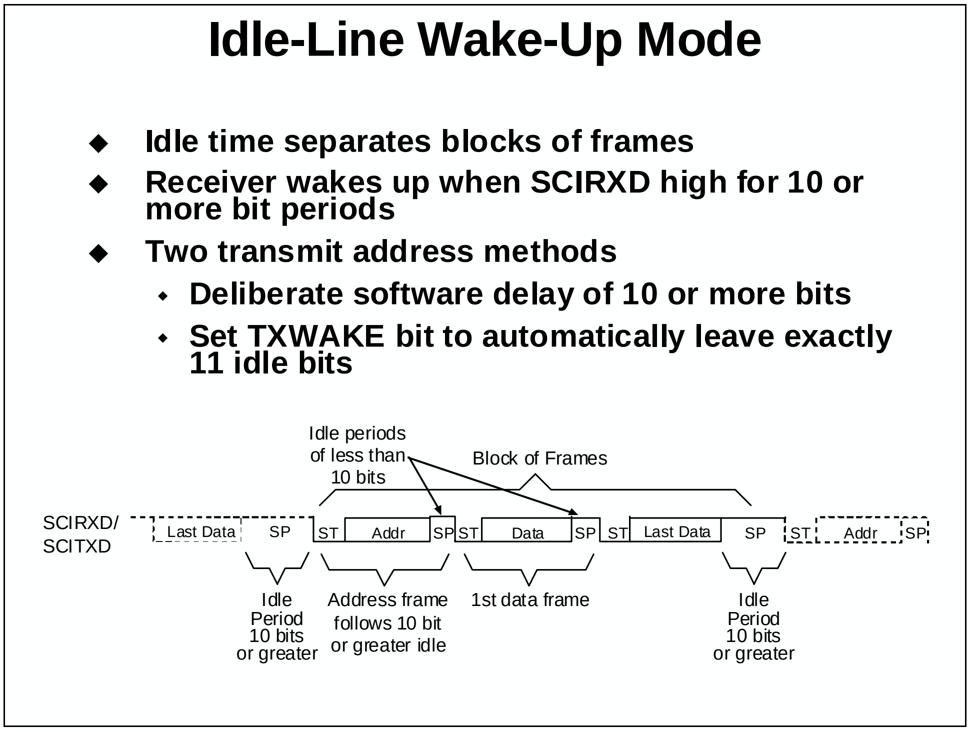
Note: 8 SCICLK periods per data bit

Multiprocessor Wake-Up Modes



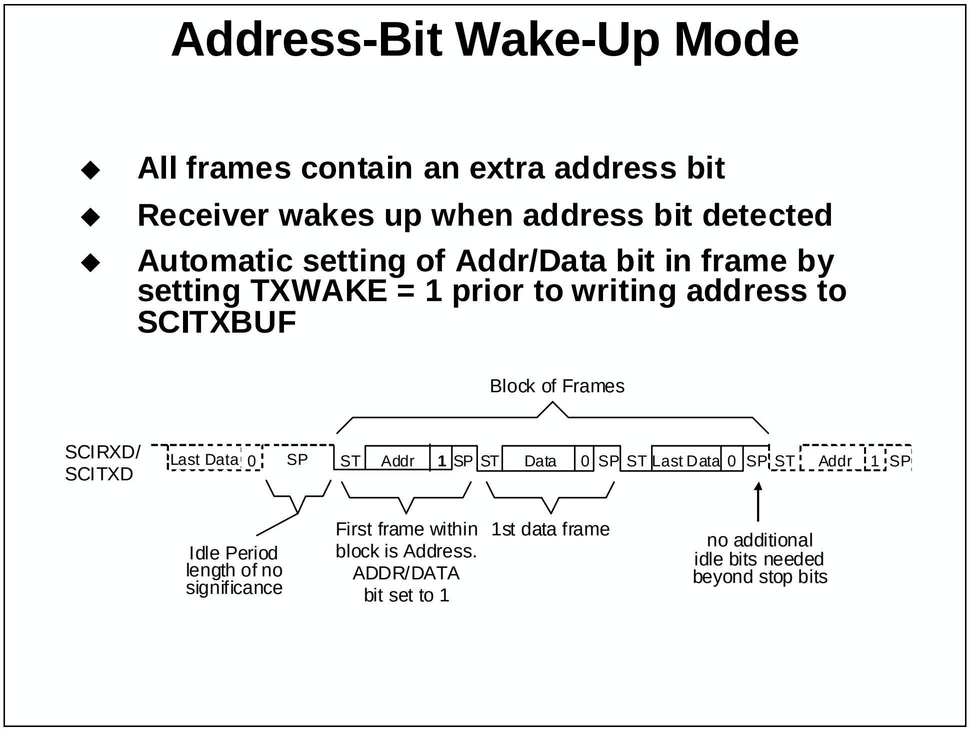
Multiprocessor Wake-Up Modes

- Allows numerous processors to be hooked up to the bus, but transmission occurs between only two of them
- *Idle-line* or *Address-bit* modes
- Sequence of Operation
  1. Potential receivers set SLEEP = 1, which disables RXINT except when an address frame is received
  2. All transmissions begin with an address frame
  3. Incoming address frame temporarily wakes up all SCIs on bus
  4. CPUs compare incoming SCI address to their SCI address
  5. Process following data frames only if address matches



Idle-Line Wake-Up Mode

- Idle time separates blocks of frames
- Receiver wakes up when SCIRXD high for 10 or more bit periods
- Two transmit address methods
  - Deliberate software delay of 10 or more bits
  - Set TXWAKE bit to automatically leave exactly 11 idle bits



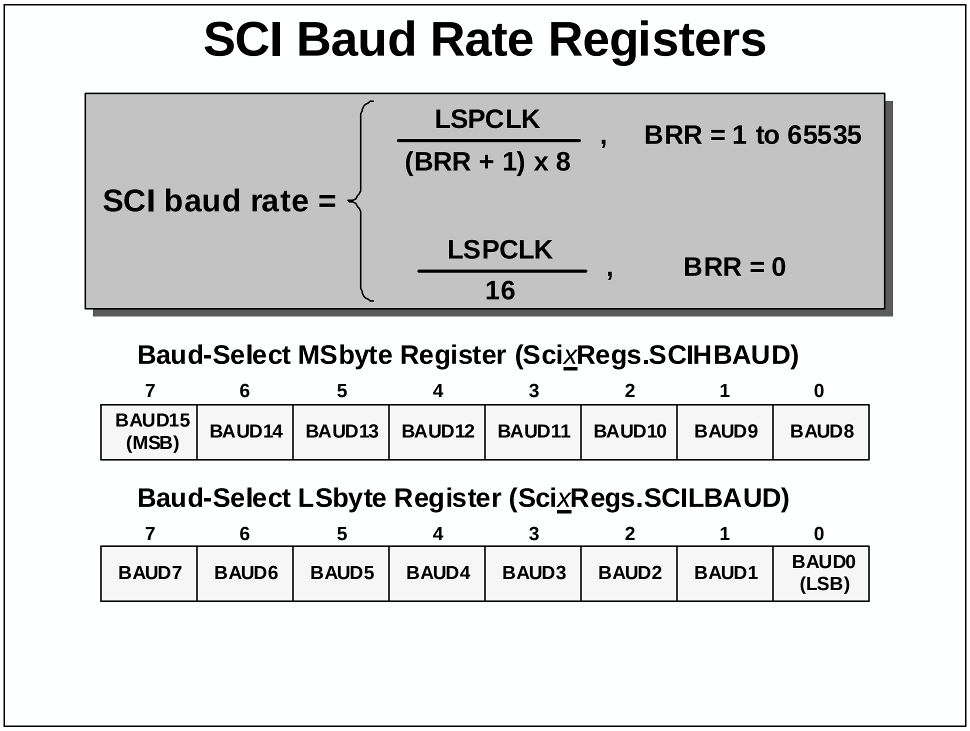
Address-Bit Wake-Up Mode

- All frames contain an extra address bit
- Receiver wakes up when address bit detected
- Automatic setting of Addr/Data bit in frame by setting TXWAKE = 1 prior to writing address to SCITXBUF

The SCI interrupt logic generates interrupt flags when it receives or transmits a complete character as determined by the SCI character length. This provides a convenient and efficient way of timing and controlling the operation of the SCI transmitter and receiver. The interrupt flag for the transmitter is TXRDY (SCICTL2.7), and for the receiver RXRDY (SCIRXST.6). TXRDY is set when a character is transferred to TXSHF and SCITXBUF is ready to receive the next character. In addition, when both the SCIBUF and TXSHF registers are empty, the TX EMPTY flag (SCICTL2.6) is set. When a new character has been received and shifted into SCIRXBUF, the RXRDY flag is set. In addition, the BRKDT flag is set if a break condition occurs. A break condition is where the SCIRXD line remains continuously low for at least ten bits, beginning after a missing stop bit. Each of the above flags can be polled by the CPU to control SCI operations, or interrupts associated with the flags can be enabled by setting the RX/BK INT ENA (SCICTL2.1) and/or the TX INT ENA (SCICTL2.0) bits active high.

Additional flag and interrupt capability exists for other receiver errors. The RX ERROR flag is the logical OR of the break detect (BRKDT), framing error (FE), receiver overrun (OE), and parity error (PE) bits. RX ERROR high indicates that at least one of these four errors has occurred during transmission. This will also send an interrupt request to the CPU if the RX ERR INT ENA (SCICTL1.6) bit is set.

SCI Registers



SCI Baud Rate Registers

$$\text{SCI baud rate} = \begin{cases} \dfrac{\text{LSPCLK}}{(\text{BRR} + 1) \text{ x } 8}, & \text{BRR = 1 to 65535} \\ \dfrac{\text{LSPCLK}}{16}, & \text{BRR = 0} \end{cases}$$

Baud-Select MSbyte Register (Sci<u>x</u>Regs.SCIHBAUD)

7	6	5	4	3	2	1	0
BAUD15 (MSB)	BAUD14	BAUD13	BAUD12	BAUD11	BAUD10	BAUD9	BAUD8

Baud-Select LSbyte Register (Sci<u>x</u>Regs.SCILBAUD)

7	6	5	4	3	2	1	0
BAUD7	BAUD6	BAUD5	BAUD4	BAUD3	BAUD2	BAUD1	BAUD0 (LSB)

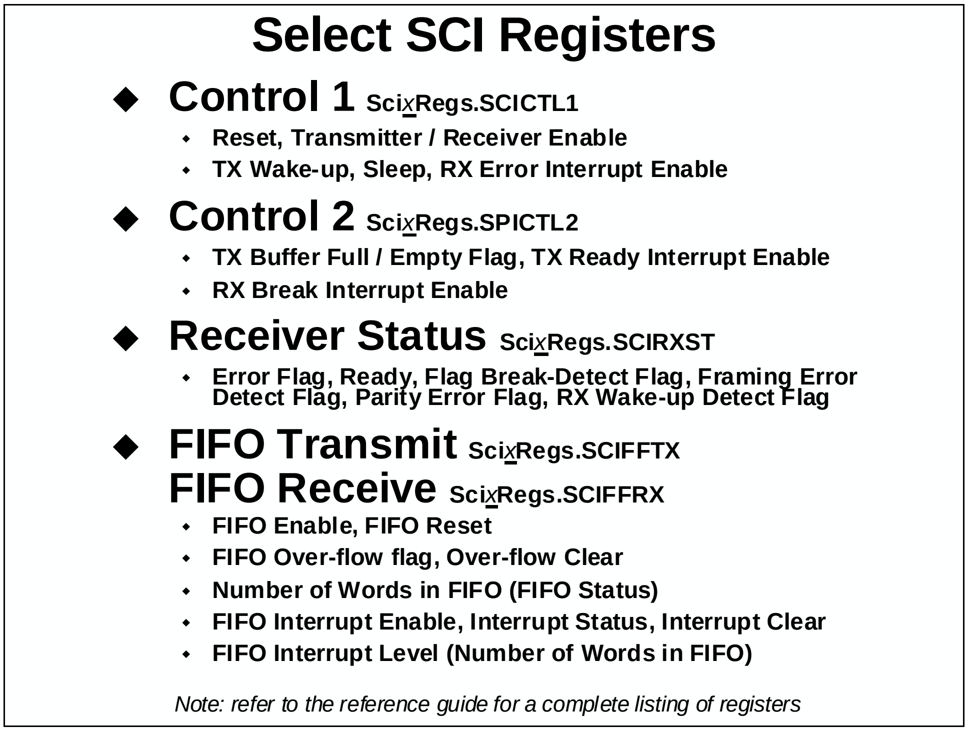
<u>Baud Rate Determination:</u> The values in the baud-select registers (SCIHBAUD and SCILBAUD) concatenate to form a 16 bit number that specifies the baud rate for the SCI.

- For BRR = 1 to 65535: SCI Baud Rate = $\dfrac{LSPCLK}{(BRR+1)\times 8}$ bits/sec

- For BRR = 0: SCI Baud Rate = $\dfrac{LSPCLK}{16}$ bits/sec

Max data rate = 6.25 Mbps @ 100 MHz

Note that the CLKOUT for the SCI module is one-half the CPU clock rate.

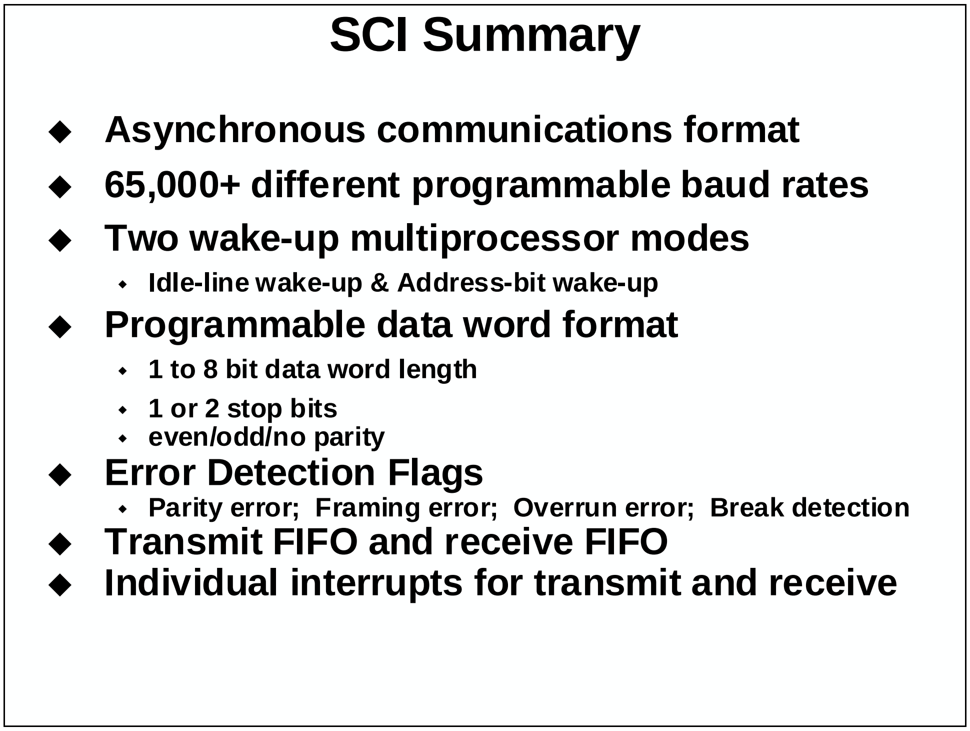


Select SCI Registers

- **Control 1** Sci*x*Regs.SCICTL1
  - Reset, Transmitter / Receiver Enable
  - TX Wake-up, Sleep, RX Error Interrupt Enable
- **Control 2** Sci*x*Regs.SPICTL2
  - TX Buffer Full / Empty Flag, TX Ready Interrupt Enable
  - RX Break Interrupt Enable
- **Receiver Status** Sci*x*Regs.SCIRXST
  - Error Flag, Ready, Flag Break-Detect Flag, Framing Error Detect Flag, Parity Error Flag, RX Wake-up Detect Flag
- **FIFO Transmit** Sci*x*Regs.SCIFFTX
  FIFO Receive Sci*x*Regs.SCIFFRX
  - FIFO Enable, FIFO Reset
  - FIFO Over-flow flag, Over-flow Clear
  - Number of Words in FIFO (FIFO Status)
  - FIFO Interrupt Enable, Interrupt Status, Interrupt Clear
  - FIFO Interrupt Level (Number of Words in FIFO)

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

SCI Summary



SCI Summary

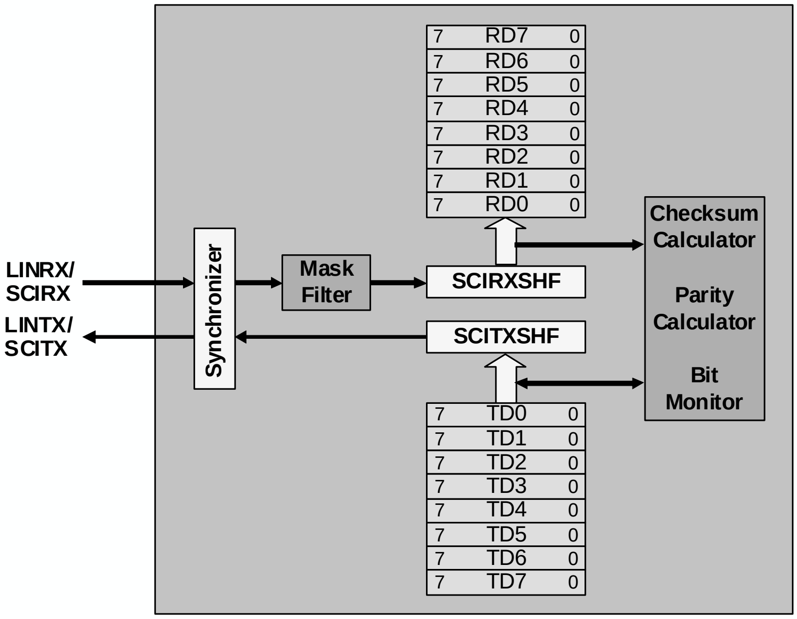
- Asynchronous communications format
- 65,000+ different programmable baud rates
- Two wake-up multiprocessor modes
  - Idle-line wake-up & Address-bit wake-up
- Programmable data word format
  - 1 to 8 bit data word length
  - 1 or 2 stop bits
  - even/odd/no parity
- Error Detection Flags
  - Parity error; Framing error; Overrun error; Break detection
- Transmit FIFO and receive FIFO
- Individual interrupts for transmit and receive

Local Interconnect Network (LIN)

Local Interconnect Network (LIN)

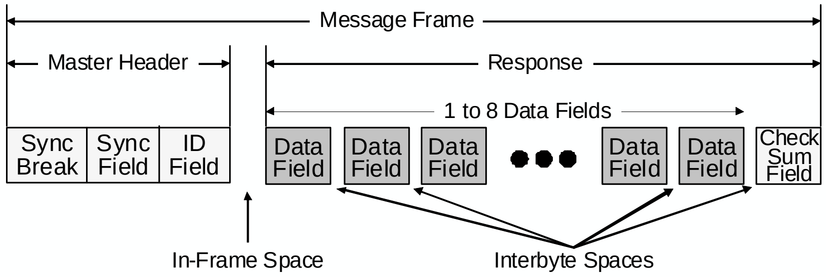
- Compliant to the LIN2.0 protocol Specification Package
- Module based on SCI (core) with added hardware features for LIN compatibility:
  - Error detector
  - Mask filter
  - Synchronizer
  - Multi-buffered receiver/transmitter
- Standard is based on SCI (UART) serial data link format
- Communication concept is single-master/multiple-slave with message identification for multi-cast transmission between any network nodes
- Module can be used in LIN mode or SCI (UART) mode

LIN Block Diagram



LIN Message Frame and Data Timing

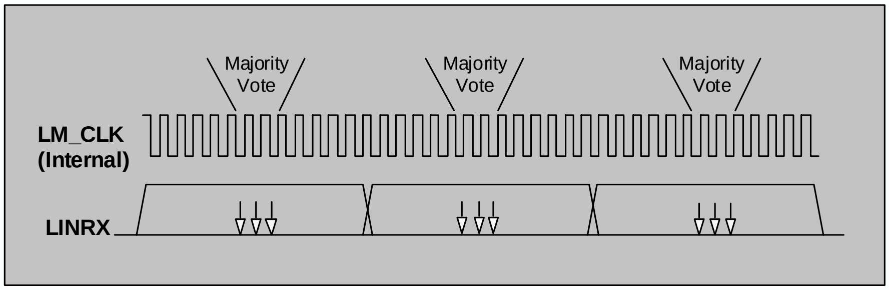
LIN Message Frame



- Sync Break – beginning of a message
- Sync Field – bit rate information
- ID Field – content of a message
- Data Field – consists of 1 data byte, 1 start bit, and 1 stop bit (10 bits total)
- Checksum Field – consists of 1 checksum byte, 1 start bit and 1 stop bit (10 bits total)
- In-Frame & Interbyte Spaces – can be 0

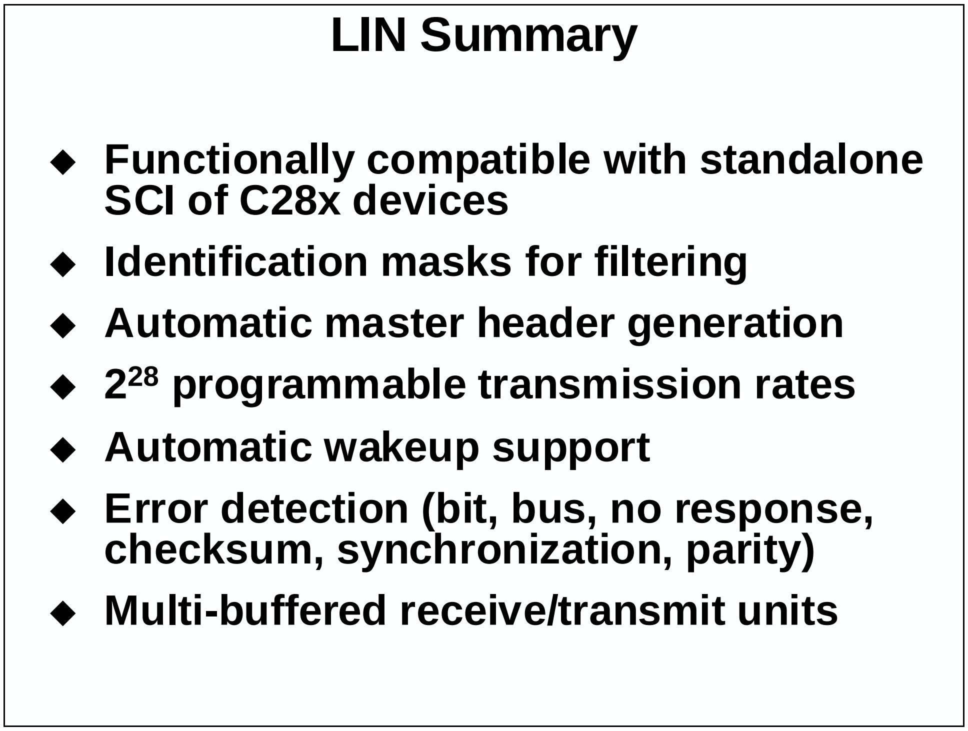
LIN Data Timing

To make a determination of the bit value, 16 samples of each bit are taken with majority vote on samples 8, 9, and 10



- **LIN module is clocked at ½ the CPU clock (SYSCLKOUT)**

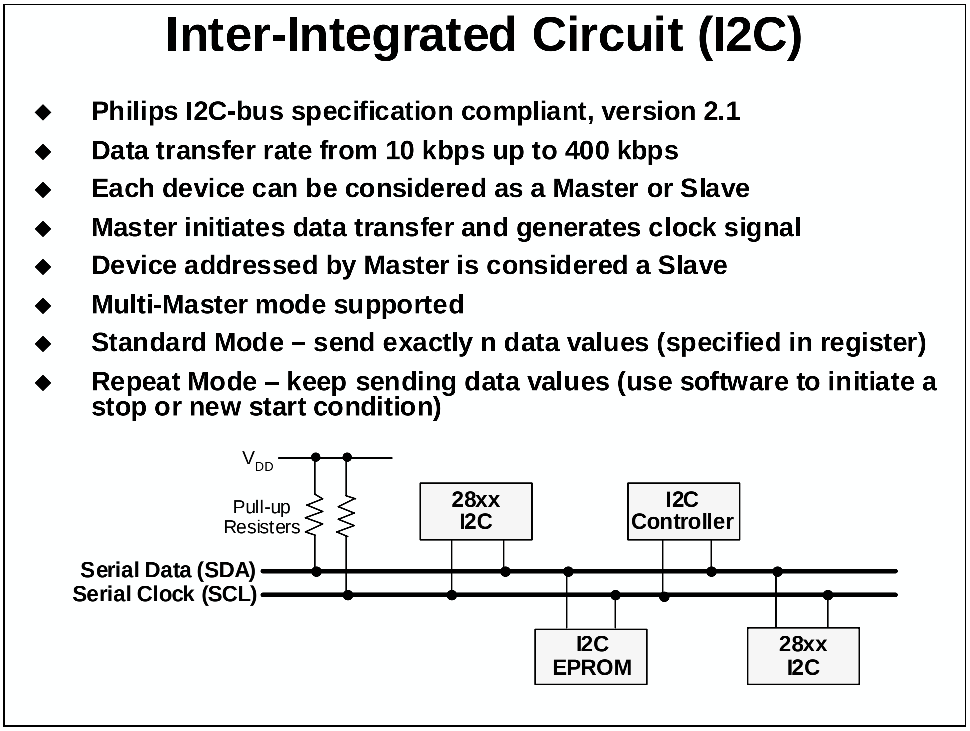
LIN Summary



LIN Summary

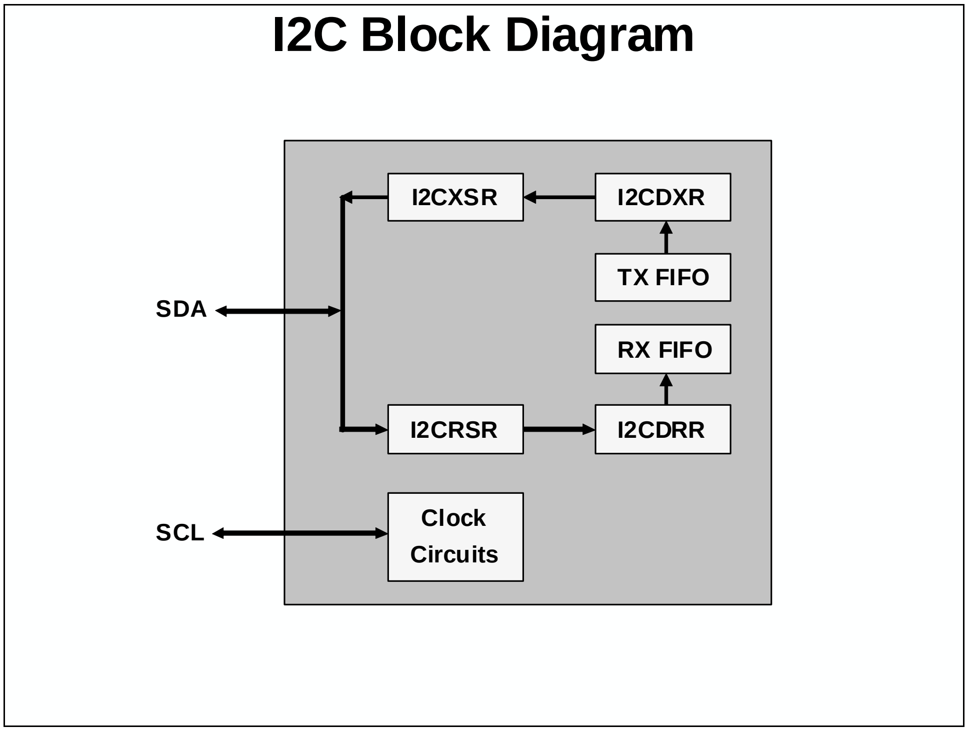
- Functionally compatible with standalone SCI of C28x devices
- Identification masks for filtering
- Automatic master header generation
- 2^{28} programmable transmission rates
- Automatic wakeup support
- Error detection (bit, bus, no response, checksum, synchronization, parity)
- Multi-buffered receive/transmit units

Inter-Integrated Circuit (I2C)



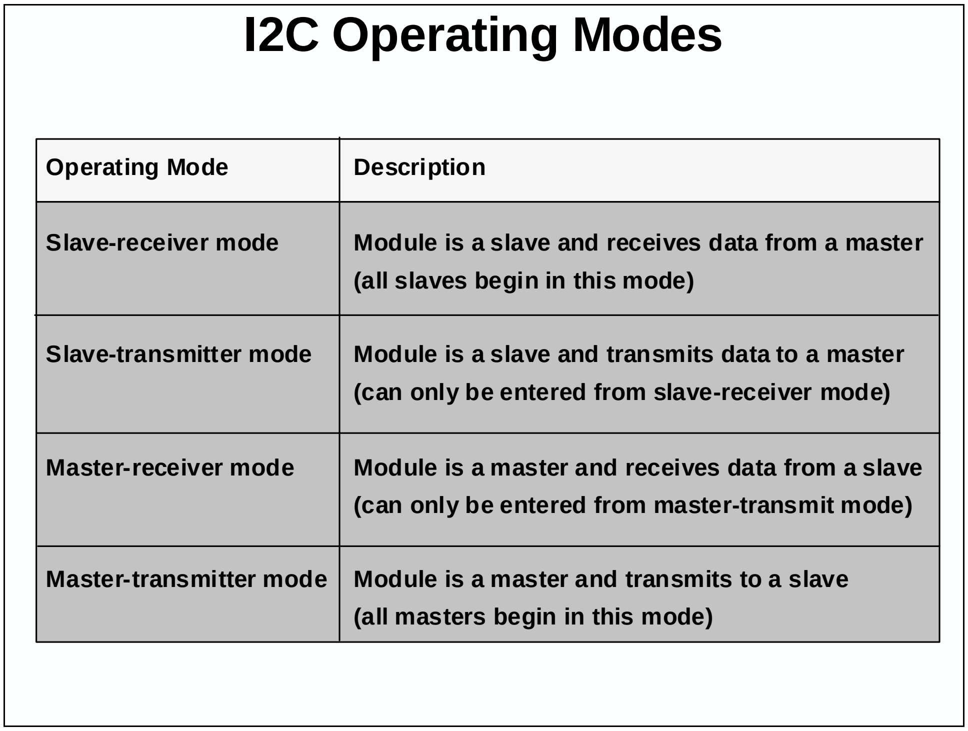
Inter-Integrated Circuit (I2C)

- Philips I2C-bus specification compliant, version 2.1
- Data transfer rate from 10 kbps up to 400 kbps
- Each device can be considered as a Master or Slave
- Master initiates data transfer and generates clock signal
- Device addressed by Master is considered a Slave
- Multi-Master mode supported
- Standard Mode – send exactly n data values (specified in register)
- Repeat Mode – keep sending data values (use software to initiate a stop or new start condition)



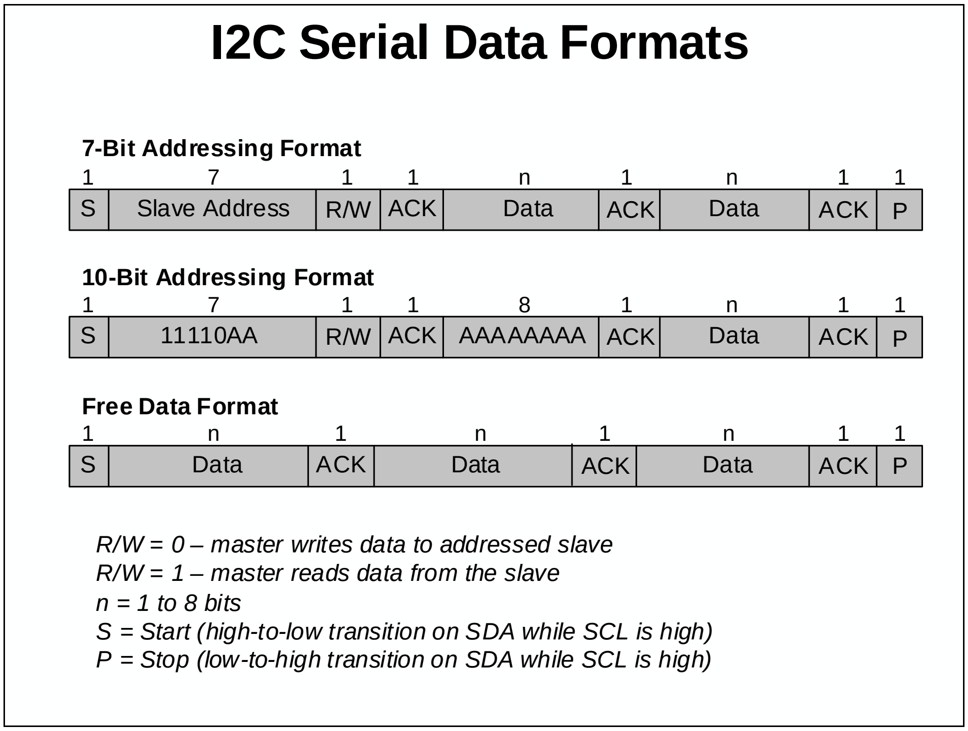
I2C Block Diagram

I2C Operating Modes and Data Formats



I2C Operating Modes

Operating Mode	Description
Slave-receiver mode	**Module is a slave and receives data from a master (all slaves begin in this mode)**
Slave-transmitter mode	**Module is a slave and transmits data to a master (can only be entered from slave-receiver mode)**
Master-receiver mode	**Module is a master and receives data from a slave (can only be entered from master-transmit mode)**
Master-transmitter mode	**Module is a master and transmits to a slave (all masters begin in this mode)**



I2C Serial Data Formats

7-Bit Addressing Format

1	7	1	1	n	1	n	1	1
S	Slave Address	R/W	ACK	Data	ACK	Data	ACK	P

10-Bit Addressing Format

1	7	1	1	8	1	n	1	1
S	11110AA	R/W	ACK	AAAAAAAA	ACK	Data	ACK	P

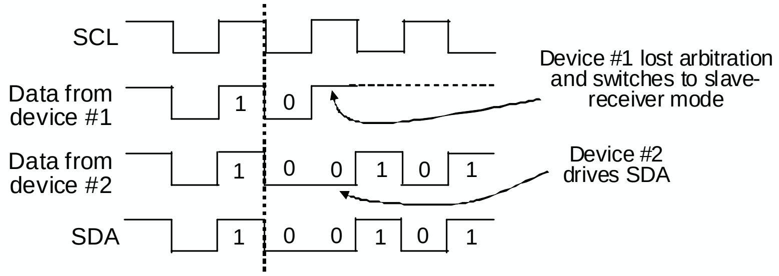
Free Data Format

1	n	1	n	1	n	1	1
S	Data	ACK	Data	ACK	Data	ACK	P

R/W = 0 – master writes data to addressed slave
R/W = 1 – master reads data from the slave
n = 1 to 8 bits
S = Start (high-to-low transition on SDA while SCL is high)
P = Stop (low-to-high transition on SDA while SCL is high)

I2C Arbitration

- **Arbitration procedure invoked if two or more master-transmitters simultaneously start transmission**
  - **Procedure uses data presented on serial data bus (SDA) by competing transmitters**
  - **First master-transmitter which drives SDA high is overruled by another master-transmitter that drives SDA low**
  - **Procedure gives priority to the data stream with the lowest binary value**



Device #1 lost arbitration and switches to slave-receiver mode

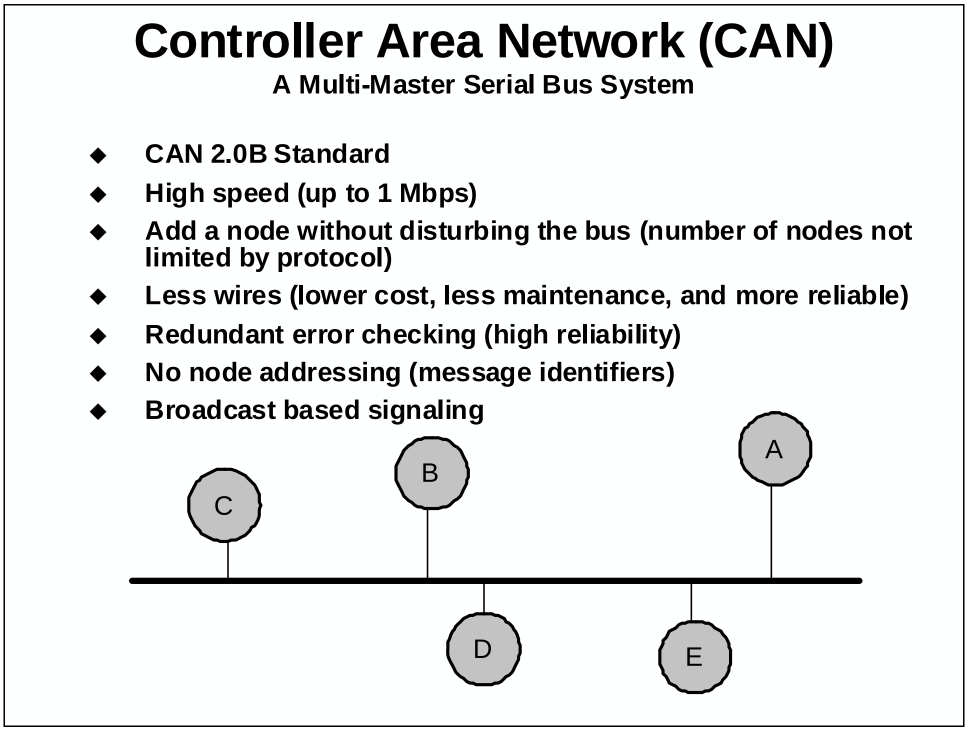
Device #2 drives SDA

I2C Summary

I2C Summary

- **Compliance with Philips I2C-bus specification (version 2.1)**
- **7-bit and 10-bit addressing modes**
- **Configurable 1 to 8 bit data words**
- **Data transfer rate from 10 kbps up to 400 kbps**
- **Transmit FIFO and receive FIFO**

Enhanced Controller Area Network (eCAN)



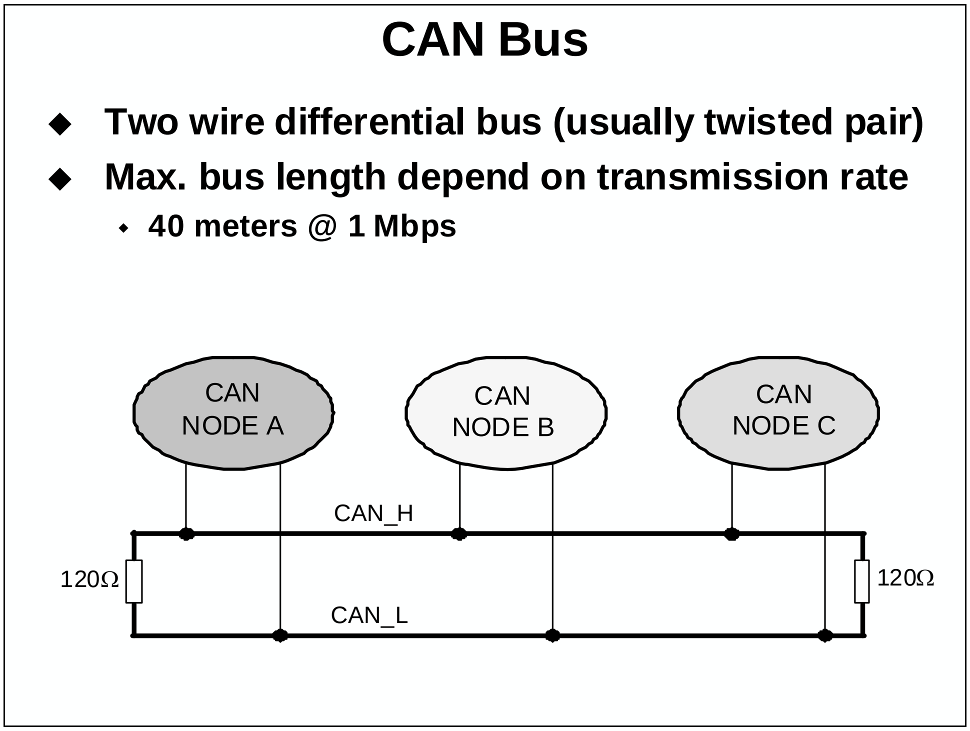
Controller Area Network (CAN)

A Multi-Master Serial Bus System

- **CAN 2.0B Standard**
- **High speed (up to 1 Mbps)**
- **Add a node without disturbing the bus (number of nodes not limited by protocol)**
- **Less wires (lower cost, less maintenance, and more reliable)**
- **Redundant error checking (high reliability)**
- **No node addressing (message identifiers)**
- **Broadcast based signaling**

CAN does not use physical addresses to address stations. Each message is sent with an identifier that is recognized by the different nodes. The identifier has two functions – it is used for message filtering and for message priority. The identifier determines if a transmitted message will be received by CAN modules and determines the priority of the message when two or more nodes want to transmit at the same time.

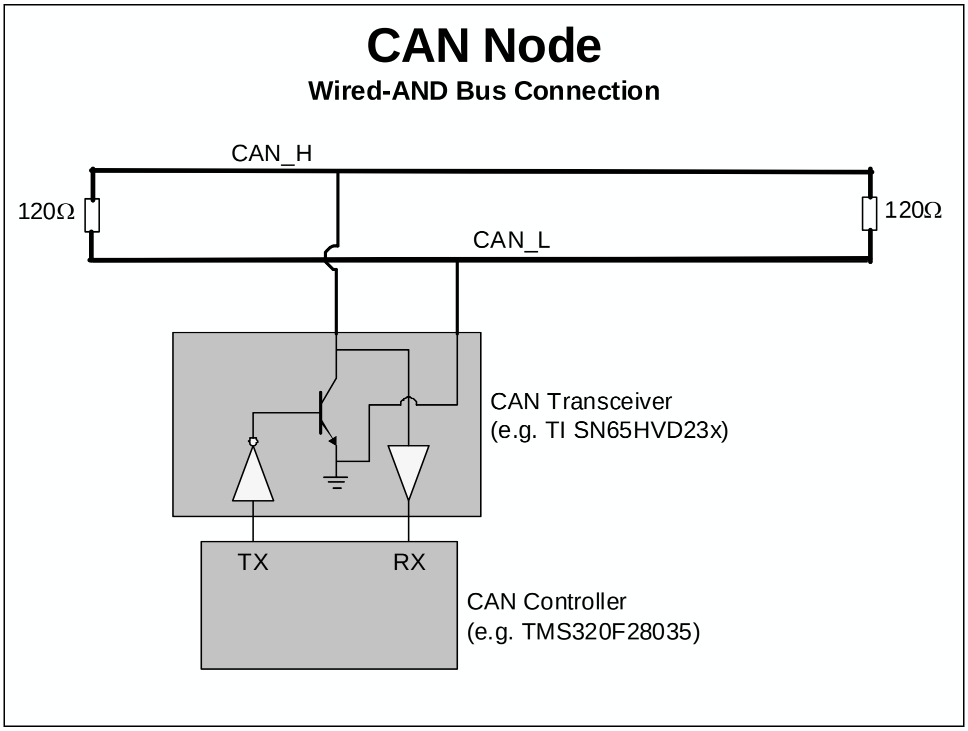
CAN Bus and Node



CAN Bus

- **Two wire differential bus (usually twisted pair)**
- **Max. bus length depend on transmission rate**
  - **40 meters @ 1 Mbps**

The MCU communicates to the CAN Bus using a transceiver. The CAN bus is a twisted pair wire and the transmission rate depends on the bus length. If the bus is less than 40 meters the transmission rate is capable up to 1 Mbit/second.



CAN Node

Wired-AND Bus Connection

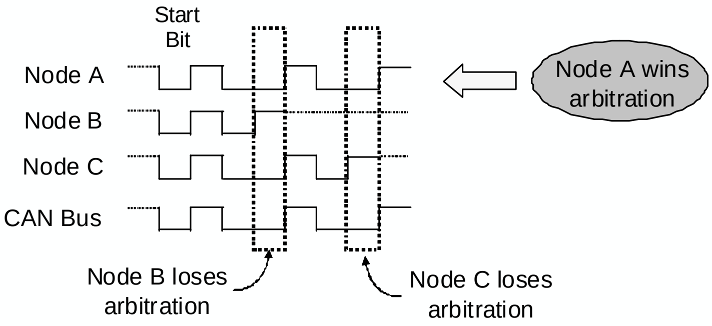
Principles of Operation

Principles of Operation

- Data messages transmitted are identifier based, not address based
- Content of message is labeled by an identifier that is unique throughout the network
  - (e.g. rpm, temperature, position, pressure, etc.)
- All nodes on network receive the message and each performs an acceptance test on the identifier
- If message is relevant, it is processed (received); otherwise it is ignored
- Unique identifier also determines the priority of the message
  - (lower the numerical value of the identifier, the higher the priority)
- When two or more nodes attempt to transmit at the same time, a non-destructive arbitration technique guarantees messages are sent in order of priority and no messages are lost

Non-Destructive Bitwise Arbitration

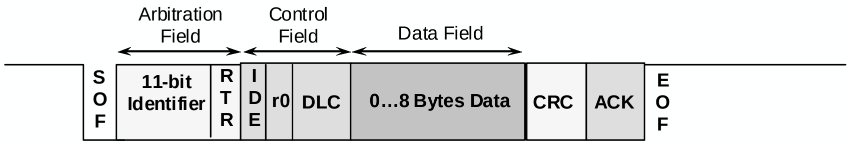
- Bus arbitration resolved via arbitration with wired-AND bus connections
  - Dominate state (logic 0, bus is high)
  - Recessive state (logic 1, bus is low)



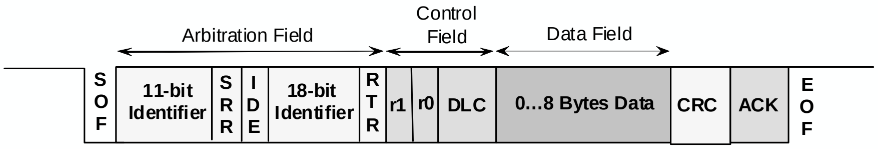
Message Format and Block Diagram

CAN Message Format

- Data is transmitted and received using Message Frames
- 8 byte data payload per message
- Standard and Extended identifier formats
  - Standard Frame: 11-bit Identifier (CAN v2.0A)

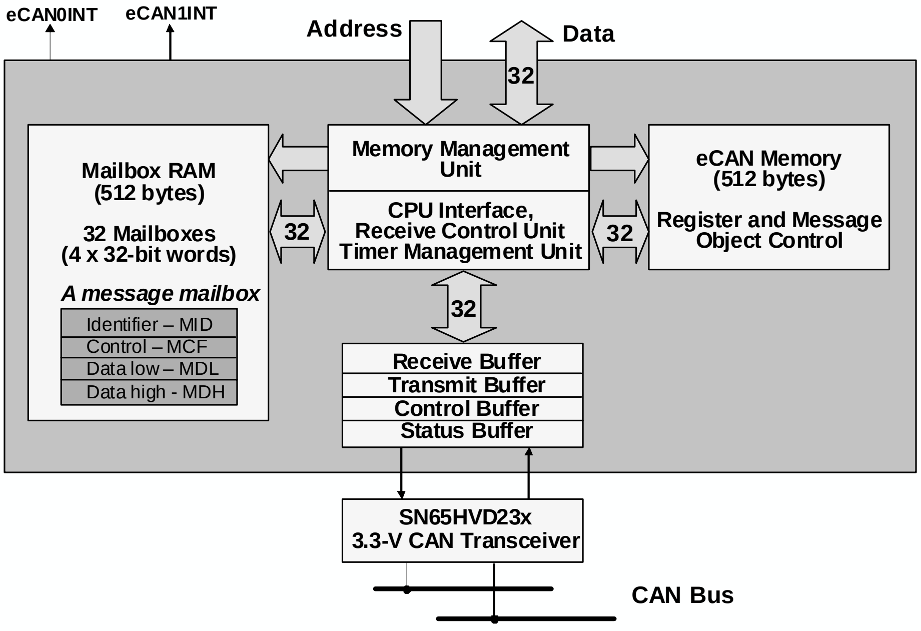


  - Extended Frame: 29-bit Identifier (CAN v2.0B)



The MCU CAN module is a full CAN Controller. It contains a message handler for transmission and reception management, and frame storage. The specification is CAN 2.0B Active – that is, the module can send and accept standard (11-bit identifier) and extended frames (29-bit identifier).

eCAN Block Diagram



The CAN controller module contains 32 mailboxes for objects of 0 to 8-byte data lengths:

- configurable transmit/receive mailboxes
- configurable with standard or extended indentifier

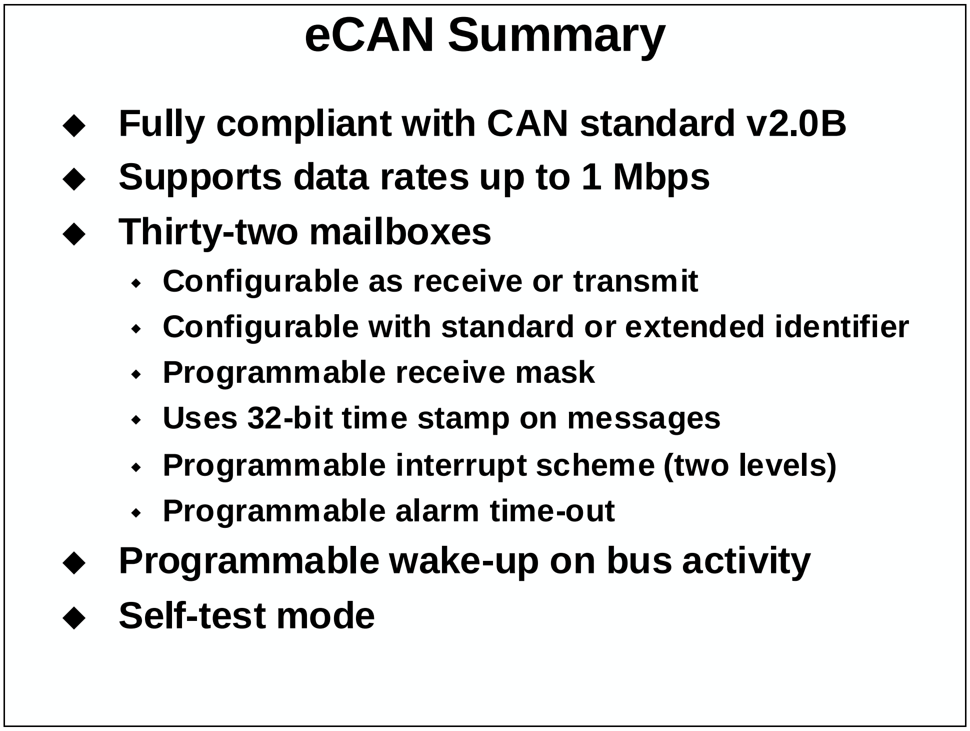
The CAN module mailboxes are divided into several parts:

- MID – contains the identifier of the mailbox
- MCF (Message Control Field) – contains the length of the message (to transmit or receive) and the RTR bit (Remote Transmission Request – used to send remote frames)
- MDL and MDH – contains the data

The CAN module contains registers which are divided into five groups. These registers are located in data memory from 0x006000 to 0x0061FF. The five register groups are:

- Control & Status Registers
- Local Acceptance Masks
- Message Object Time Stamps
- Message Object Timeout
- Mailboxes

eCAN Summary



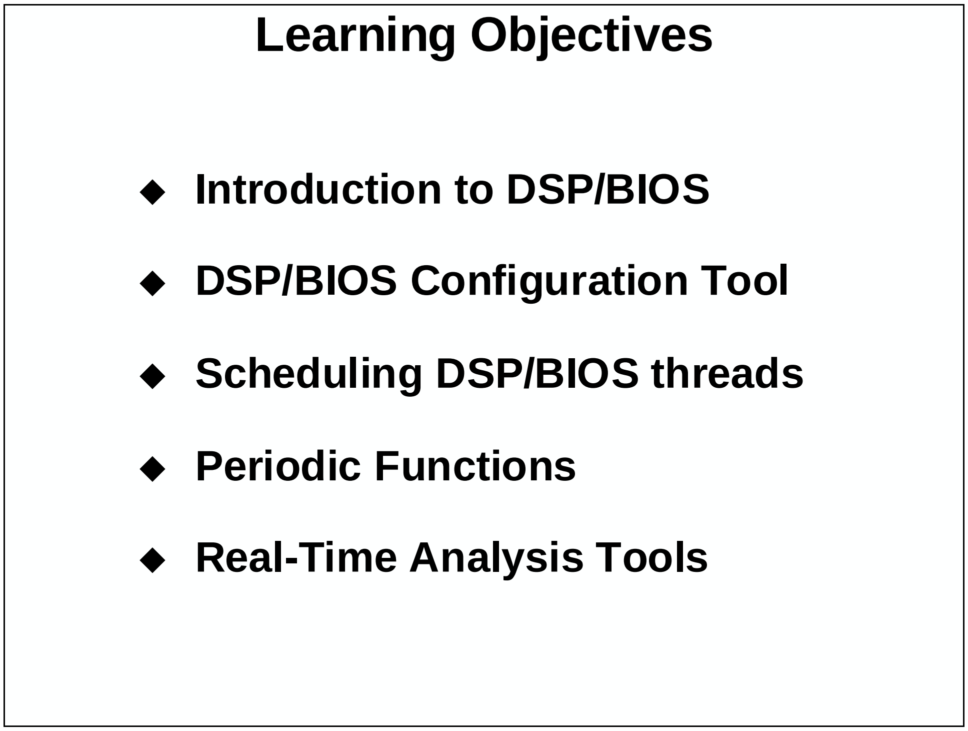
eCAN Summary

- **Fully compliant with CAN standard v2.0B**
- **Supports data rates up to 1 Mbps**
- **Thirty-two mailboxes**
  - **Configurable as receive or transmit**
  - **Configurable with standard or extended identifier**
  - **Programmable receive mask**
  - **Uses 32-bit time stamp on messages**
  - **Programmable interrupt scheme (two levels)**
  - **Programmable alarm time-out**
- **Programmable wake-up on bus activity**
- **Self-test mode**

Introduction

This module discusses the basic features of using DSP/BIOS in a system. Scheduling threads, periodic functions, and the use of real-time analysis tools will be demonstrated, in addition to programming the flash with DSP/BIOS.

Learning Objectives



Learning Objectives

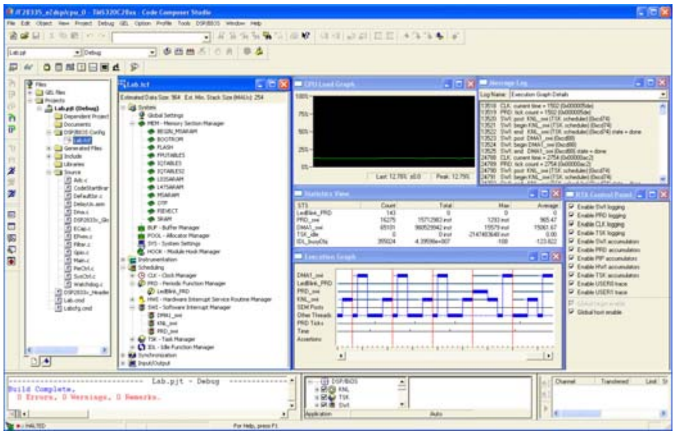
- Introduction to DSP/BIOS
- DSP/BIOS Configuration Tool
- Scheduling DSP/BIOS threads
- Periodic Functions
- Real-Time Analysis Tools

Module Topics

Introduction to DSP/BIOS

What is DSP/BIOS?

- A full-featured, scalable real-time kernel
  - System configuration tools
  - Preemptive multi-threading scheduler
  - Real-time analysis tools



```
------------------------------  Lab.pjt - Debug  --------------
Build Complete,
  0 Errors, 0 Warnings, 0 Remarks.
```

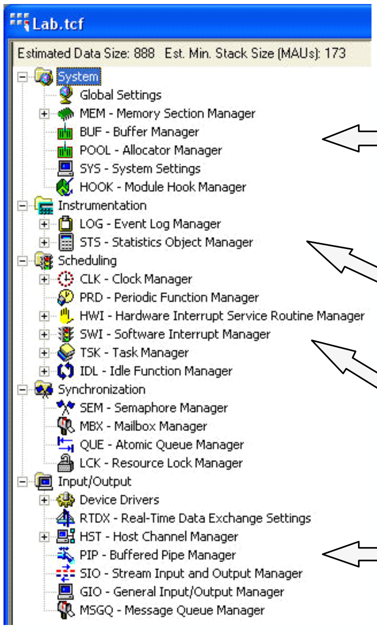
Why Use DSP/BIOS?

- Helps Manage complex system resources
  - *no need to develop or maintain a "home-brew" kernel*
  - *faster time to market*
- Efficient debugging of real-time applications
  - *Real-Time Analysis*
- Create robust applications
  - *industry proven kernel technology*
- Reduce cost of software maintenance
  - *code reuse and standardized software*
- Integrated with Code Composer Studio IDE
  - *requires no runtime license fees*
  - *fully supported by TI*
- Uses minimal Mips and Memory (2-8Kw)
  - *scalable – use only what is needed*
  - *easily fits in limited memory space*

DSP/BIOS Configuration Tool

The *DSP/BIOS Configuration Tool* (often called *Config Tool* or *GUI Tool* or *GUI*) creates and modifies a system file called the Text Configuration File (.tcf). If we talk about using .tcf files, we're also talking about using the *Config Tool*.

DSP/BIOS Configuration Tool (file .tcf)



Lab.tcf

Estimated Data Size: 888 Est. Min. Stack Size (MAUs): 173

- System
  - Global Settings
  - MEM - Memory Section Manager
  - BUF - Buffer Manager
  - POOL - Allocator Manager
  - SYS - System Settings
  - HOOK - Module Hook Manager
- Instrumentation
  - LOG - Event Log Manager
  - STS - Statistics Object Manager
- Scheduling
  - CLK - Clock Manager
  - PRD - Periodic Function Manager
  - HWI - Hardware Interrupt Service Routine Manager
  - SWI - Software Interrupt Manager
  - TSK - Task Manager
  - IDL - Idle Function Manager
- Synchronization
  - SEM - Semaphore Manager
  - MBX - Mailbox Manager
  - QUE - Atomic Queue Manager
  - LCK - Resource Lock Manager
- Input/Output
  - Device Drivers
  - RTDX - Real-Time Data Exchange Settings
  - HST - Host Channel Manager
  - PIP - Buffered Pipe Manager
  - SIO - Stream Input and Output Manager
  - GIO - General Input/Output Manager
  - MSGQ - Message Queue Manager

- ◆ **System Setup Tools**
  - Handles memory configuration (builds .cmd file), run-time support libraries, interrupt vectors, system setup and reset, etc.
- ◆ **Real-Time Analysis Tools**
  - Allows application to run uninterrupted while displaying debug data
- ◆ **Real-Time Scheduler**
  - Preemptive tread manager kernel configures DSP/BIOS scheduling
- ◆ **Real-Time I/O**
  - Allows two way communication between threads or between target and PC host

The GUI (graphical user interface) simplifies system design by:

- Automatically including the appropriate runtime support libraries
- Automatically handles interrupt vectors and system reset
- Handles system memory configuration (builds .cmd file)
- When a .tcf file is saved, the Config Tool generates 5 additional files:

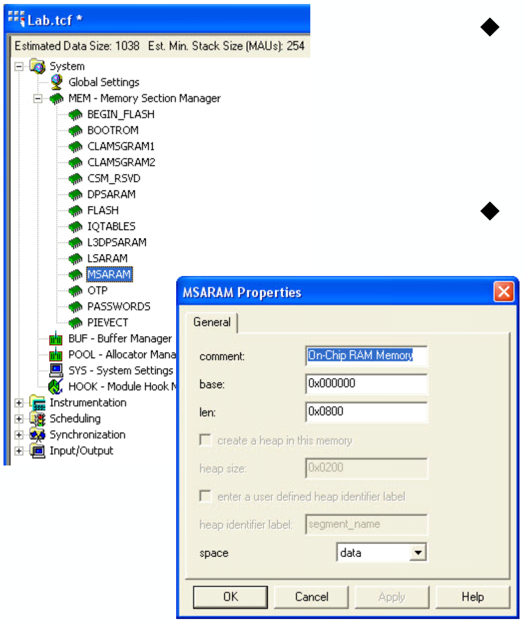
*Filename***.tcf**	Text Configuration File
*Filename***cfg_c.c**	C code created by Config Tool
*Filename***cfg.s28**	ASM code created by Config Tool
*Filename***cfg.cmd**	Linker command file
*Filename***cfg.h**	header file for *cfg_c.c
*Filename***cfg.h28**	header file for *cfg.s28

When you add a .tcf file to your project, CCS automatically adds the C and assembly (.s28) files and the linker command file (.cmd) to the project under the *Generated Files* folder.

1. Creating a New Memory Region (Using MEM)

First, to create a specific memory area, open up the .tcf file, right-click on the Memory Section Manager and select “Insert MEM”. Give this area a unique name and then specify its base and length. Once created, you can place sections into it (shown in the next step).

Memory Section Manager (MEM)



comment: On-Chip RAM Memory

base: 0x000000

len: 0x0800

□ create a heap in this memory

heap size: 0x0200

□ enter a user defined heap identifier label

heap identifier label: segment_name

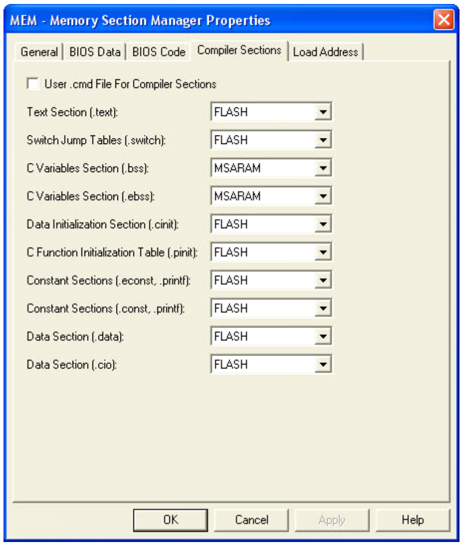
space: data

- Generates the main linker command file for your code project
  - Create memories
  - Place sections
- To create a new memory area:
  - Right-click on MEM and select *insert memory*
  - Enter your choice of a name for the memory
  - Right-click on the memory, and select *Properties*
    - fill in base, length, space

2. Placing Sections – MEM Manager Properties

The configuration tool makes it easy to place sections. The predefined compiler sections that were described earlier each have their own drop-down menu to select one of the memory regions you defined (in step 1).

Memory Section Manager Properties



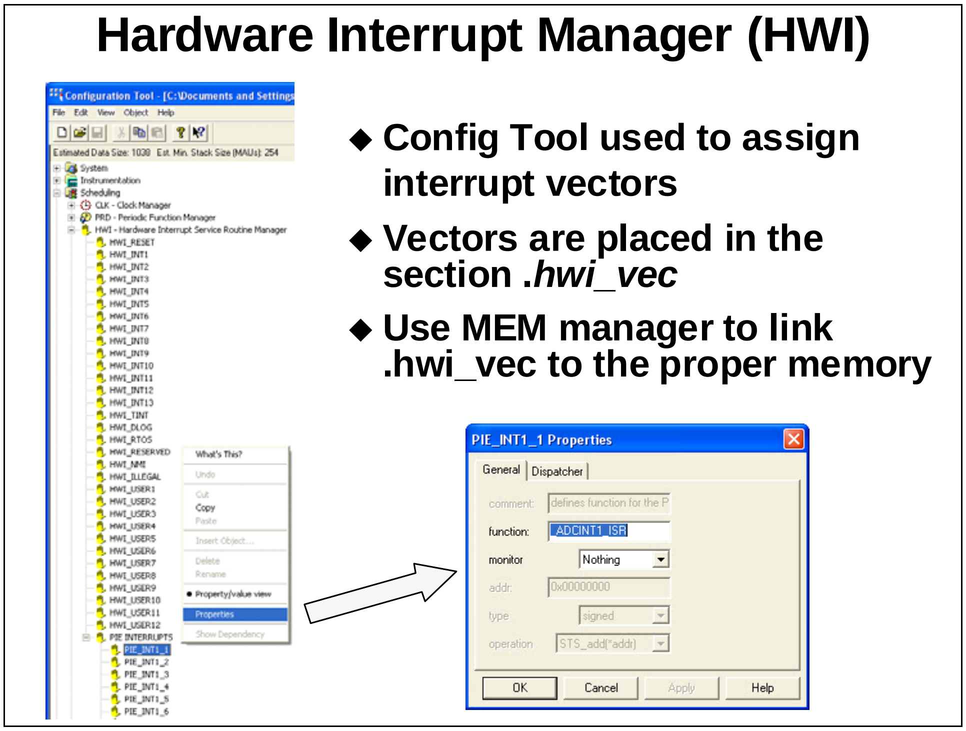
□ User .cmd File For Compiler Sections

Text Section (.text):	FLASH
Switch Jump Tables (.switch):	FLASH
C Variables Section (.bss):	MSARAM
C Variables Section (.ebss):	MSARAM
Data Initialization Section (.cinit):	FLASH
C Function Initialization Table (.pinit):	FLASH
Constant Sections (.econst, .printf):	FLASH
Constant Sections (.const, .printf):	FLASH
Data Section (.data):	FLASH
Data Section (.cio):	FLASH

- **To place a section into a memory area:**
  - **Right-click on MEM and select *Properties***
  - **Select the desired tab (e.g. Compiler)**
  - **Select the memory you would like to link each section to**

3. PIE Interrupts – HWI Interrupts

The configuration tools is also used to assign the interrupt vectors. The vectors are placed into a section named .hwi_vec. The memory manager (MEM) links this section to the proper location in memory.



Hardware Interrupt Manager (HWI)

- ◆ Config Tool used to assign interrupt vectors
- ◆ Vectors are placed in the section *.hwi_vec*
- ◆ Use MEM manager to link .hwi_vec to the proper memory

PIE_INT1_1 Properties

comment: defines function for the P

function: _ADCINT1_ISR

monitor: Nothing

addr: 0x00000000

type: signed

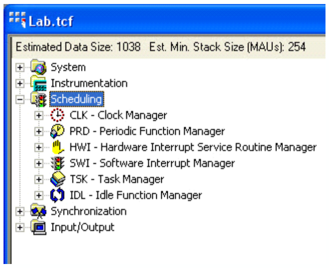
operation: STS_add(*addr)

4. Running the Linker

Creating the Linker Command File (via .tcf)

When you have finished creating memory regions and allocating sections into these memory areas (i.e. when you save the .tcf file), the CCS configuration tool creates five files. One of the files is BIOS's cfg.cmd file — a linker command file.

Files Created by the Configuration Tool

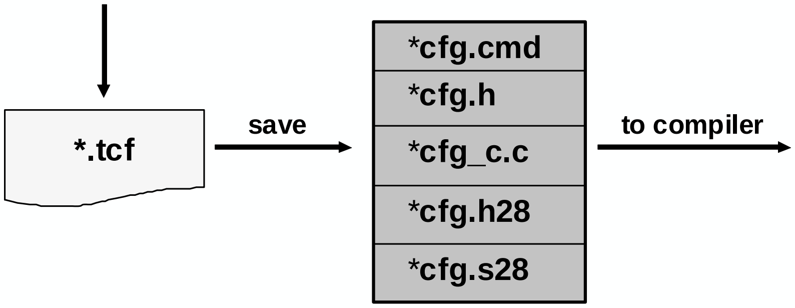


Lab.tcf

Estimated Data Size: 1038 Est. Min. Stack Size (MAUs): 254

- System
- Instrumentation
- Scheduling
  - CLK - Clock Manager
  - PRD - Periodic Function Manager
  - HWI - Hardware Interrupt Service Routine Manager
  - SWI - Software Interrupt Manager
  - TSK - Task Manager
  - IDL - Idle Function Manager
- Synchronization
- Input/Output

- ◆ Config tool generates five different files
- ◆ .cmd file is generated from your MEM settings
- ◆ Vectors put into *cfg_c.c



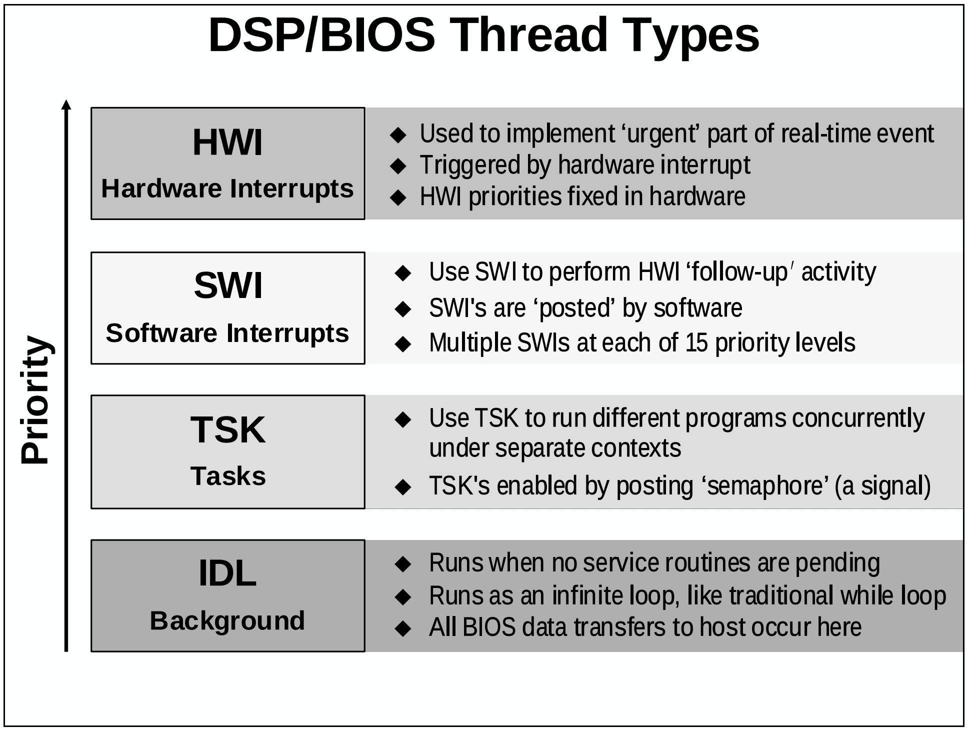
This file contains two main parts, MEMORY and SECTIONS. (Though, if you open and examine it, it's not quite as nicely laid out as shown above.)

Running the Linker

The linker's main purpose is to *link* together various object files. It combines like-named input sections from the various object files and places each new output section at specific locations in memory. In the process, it resolves (provides actual addresses for) all of the symbols described in your code. The linker can create two outputs, the executable (.out) file and a report which describes the results of linking (.map).

Note: The linker gets run automatically when you BUILD or REBUILD your project.

Scheduling DSP/BIOS threads



DSP/BIOS Thread Types

HWI
Hardware Interrupts

- ◆ Used to implement 'urgent' part of real-time event
- ◆ Triggered by hardware interrupt
- ◆ HWI priorities fixed in hardware

SWI
Software Interrupts

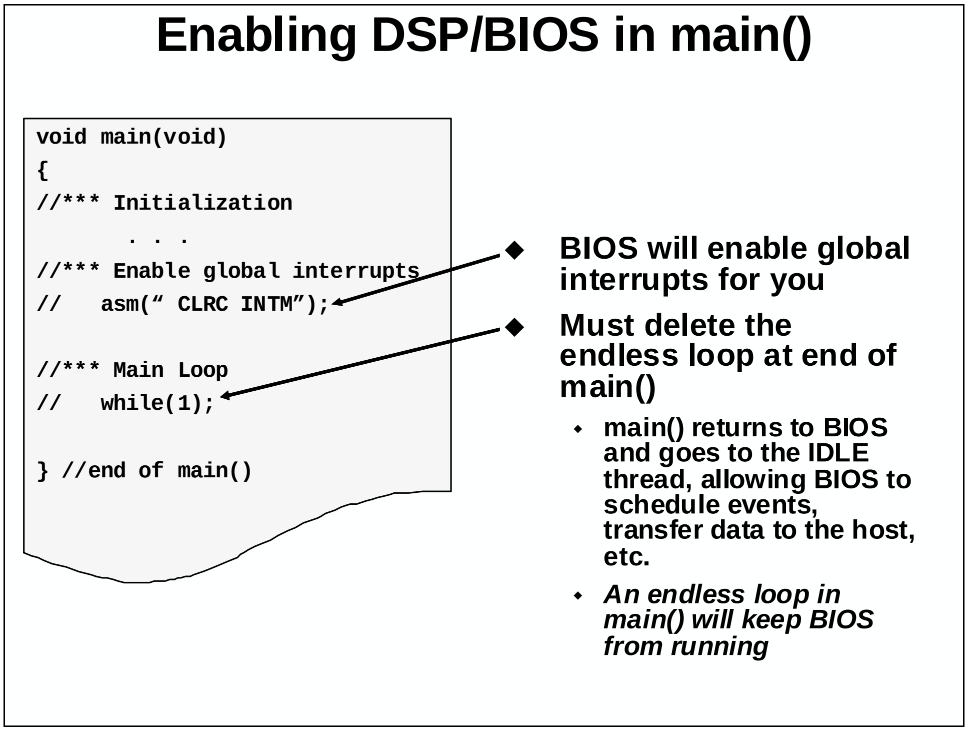
- ◆ Use SWI to perform HWI 'follow-up′ activity
- ◆ SWI's are 'posted' by software
- ◆ Multiple SWIs at each of 15 priority levels

TSK
Tasks

- ◆ Use TSK to run different programs concurrently under separate contexts
- ◆ TSK's enabled by posting 'semaphore' (a signal)

IDL
Background

- ◆ Runs when no service routines are pending
- ◆ Runs as an infinite loop, like traditional while loop
- ◆ All BIOS data transfers to host occur here

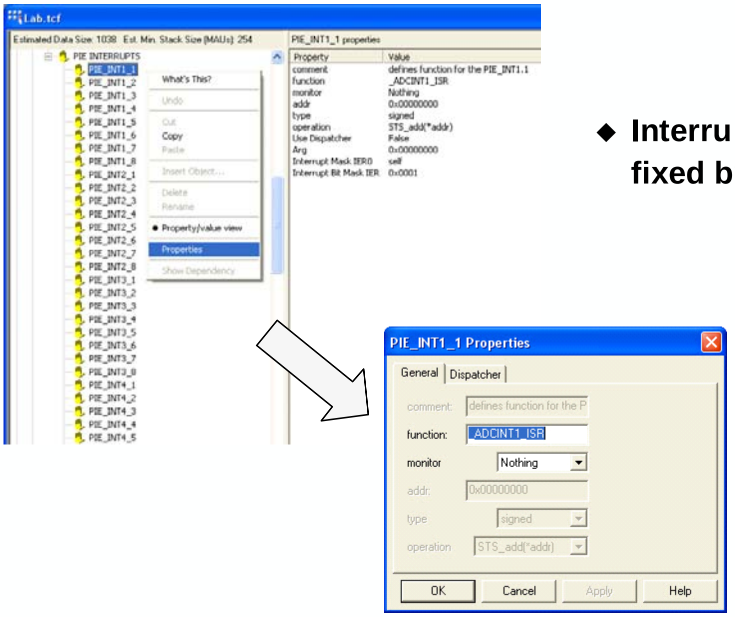


Enabling DSP/BIOS in main()

```
void main(void)
{
//*** Initialization
       . . .
//*** Enable global interrupts
//   asm(" CLRC INTM");

//*** Main Loop
//   while(1);

} //end of main()
```

- ◆ **BIOS will enable global interrupts for you**
- ◆ **Must delete the endless loop at end of main()**
  - **main() returns to BIOS and goes to the IDLE thread, allowing BIOS to schedule events, transfer data to the host, etc.**
  - ***An endless loop in main() will keep BIOS from running***

Using Hardware Interrupts - HWI



Estimated Data Size: 1038 Est. Min. Stack Size (MAUs): 254

PIE_INT1_1 properties

Property	Value
comment	defines function for the PIE_INT1.1
function	_ADCINT1_ISR
monitor	Nothing
addr	0x00000000
type	signed
operation	STS_add(*addr)
Use Dispatcher	False
Arg	0x00000000
Interrupt Mask IER0	self
Interrupt Bit Mask IER	0x0001

PIE_INT1_1 Properties

comment: defines function for the P

function: _ADCINT1_ISR

monitor: Nothing

addr: 0x00000000

type: signed

operation: STS_add(*addr)

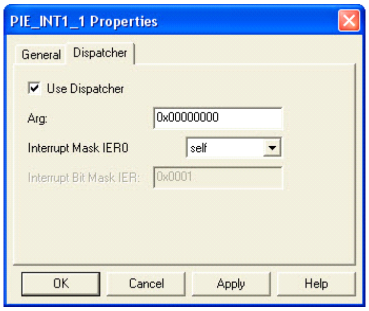
- ◆ Interrupt priority fixed by hardware

The HWI Dispatcher

- ◆ For **non-BIOS** code, use the *interrupt* keyword to declare an ISR
  - ◆ tells the compiler to perform context save/restore

```
interrupt void MyHwi(void)
{
}
```

- ◆ For DSP/BIOS code, use the *Dispatcher* to perform the save/restore
  - ◆ Remove the interrupt keyword from the MyHwi()
  - ◆ Check the "Use Dispatcher" box when you configure the interrupt vector in the DSP/BIOS configuration tool
  - ◆ This is necessary if you want to use any DSP/BIOS functionality inside the ISR



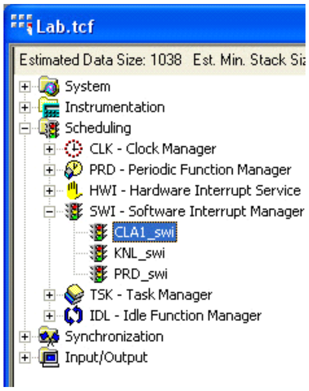
☑ Use Dispatcher

Arg: 0x00000000

Interrupt Mask IER0: self

Interrupt Bit Mask IER: 0x0001

Using Software Interrupts - SWI



Lab.tcf

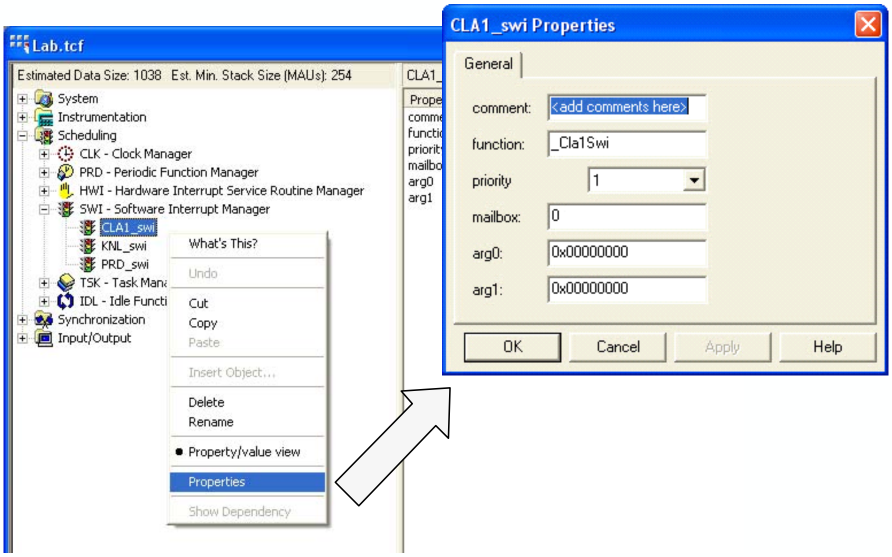
Estimated Data Size: 1038 Est. Min. Stack Siz

- System
- Instrumentation
- Scheduling
  - CLK - Clock Manager
  - PRD - Periodic Function Manager
  - HWI - Hardware Interrupt Service
  - SWI - Software Interrupt Manager
    - CLA1_swi
    - KNL_swi
    - PRD_swi
  - TSK - Task Manager
  - IDL - Idle Function Manager
- Synchronization
- Input/Output

- Make each algorithm an *independent* software interrupt
- SWI scheduling is handled by DSP/BIOS
  - HWI function triggered by hardware
  - SWI function triggered by software

    e.g. a call to SWI_post()
- Why use a SWI?
  - No limitation on number of SWIs, and priorities for SWIs are user-defined
  - SWI can be scheduled by hardware or software event(s)
  - Defer processing from HWI to SWI

SWI Properties



Lab.tcf

Estimated Data Size: 1038 Est. Min. Stack Size (MAUs): 254

- System
- Instrumentation
- Scheduling
  - CLK - Clock Manager
  - PRD - Periodic Function Manager
  - HWI - Hardware Interrupt Service Routine Manager
  - SWI - Software Interrupt Manager
    - CLA1_swi
    - KNL_swi
    - PRD_swi
  - TSK - Task Mana
  - IDL - Idle Functi
- Synchronization
- Input/Output

CLA1_swi Properties

General

comment: <add comments here>

function: _Cla1Swi

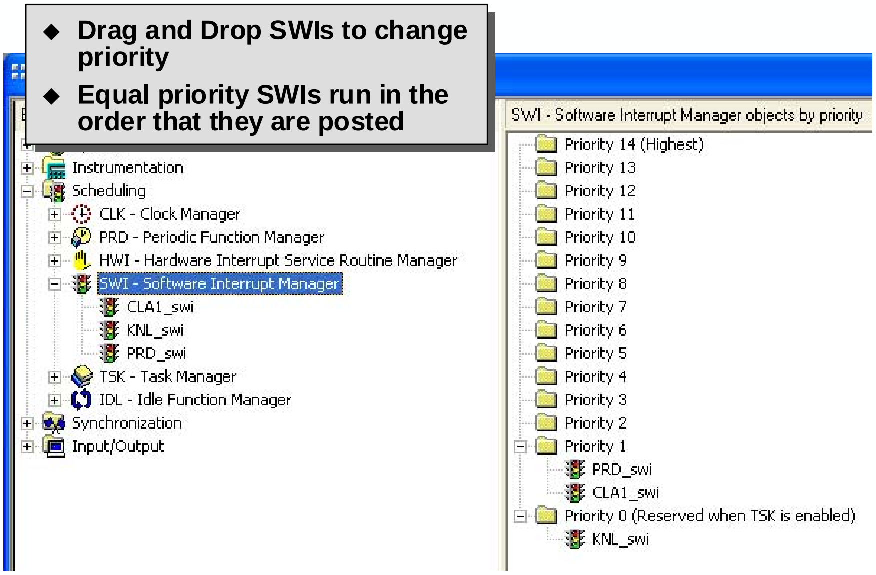
priority: 1

mailbox: 0

arg0: 0x00000000

arg1: 0x00000000

Managing SWI Priority



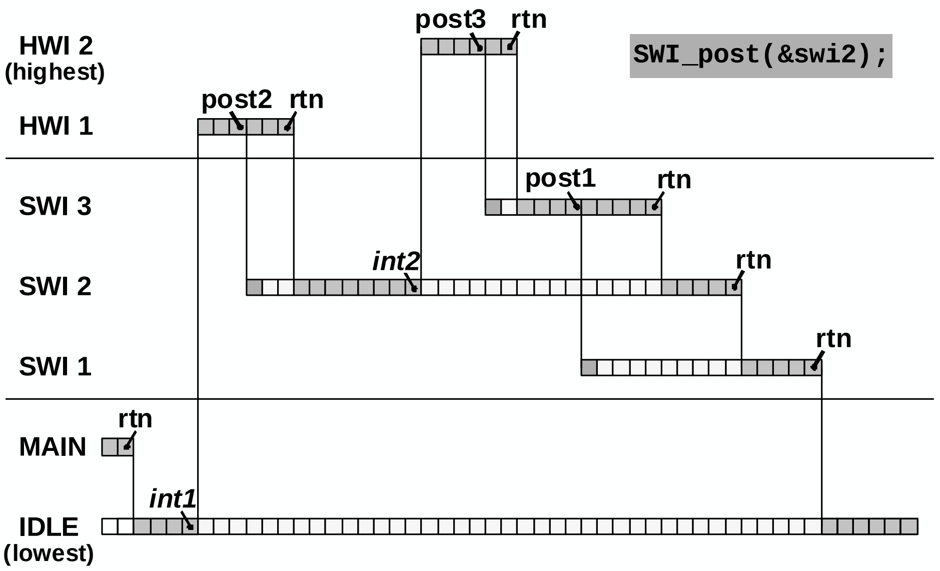
- Drag and Drop SWIs to change priority
- Equal priority SWIs run in the order that they are posted

- Instrumentation
- Scheduling
  - CLK - Clock Manager
  - PRD - Periodic Function Manager
  - HWI - Hardware Interrupt Service Routine Manager
  - SWI - Software Interrupt Manager
    - CLA1_swi
    - KNL_swi
    - PRD_swi
  - TSK - Task Manager
  - IDL - Idle Function Manager
- Synchronization
- Input/Output

SWI - Software Interrupt Manager objects by priority

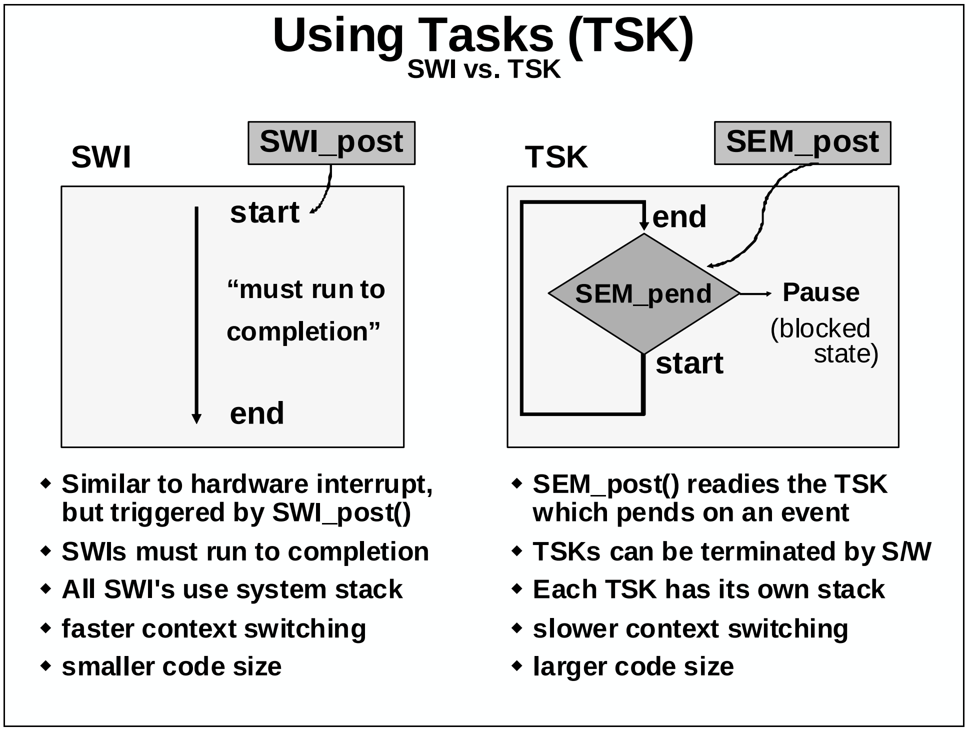
- Priority 14 (Highest)
- Priority 13
- Priority 12
- Priority 11
- Priority 10
- Priority 9
- Priority 8
- Priority 7
- Priority 6
- Priority 5
- Priority 4
- Priority 3
- Priority 2
- Priority 1
  - PRD_swi
  - CLA1_swi
- Priority 0 (Reserved when TSK is enabled)
  - KNL_swi

Priority Based Thread Scheduling



`SWI_post(&swi2);`

User sets the priority...BIOS does the scheduling



Using Tasks (TSK)

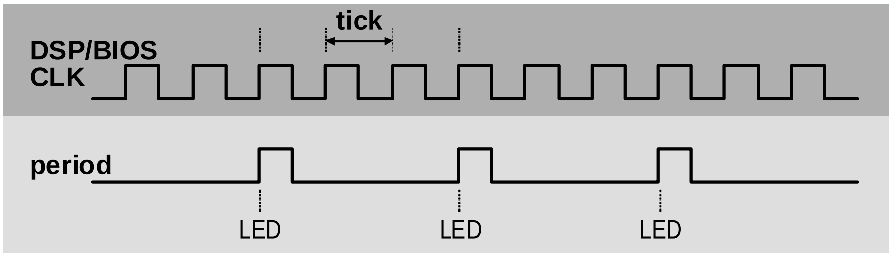
SWI vs. TSK

- Similar to hardware interrupt, but triggered by SWI_post()
- SWIs must run to completion
- All SWI's use system stack
- faster context switching
- smaller code size

- SEM_post() readies the TSK which pends on an event
- TSKs can be terminated by S/W
- Each TSK has its own stack
- slower context switching
- larger code size

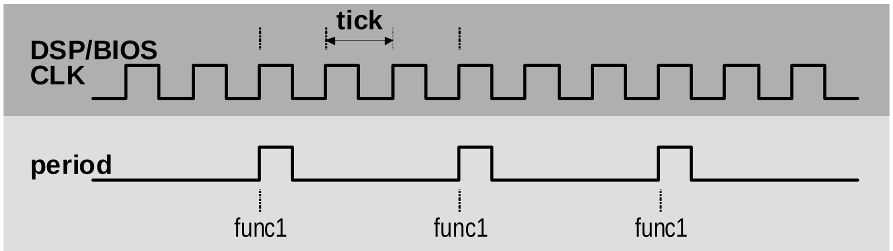
Periodic Functions

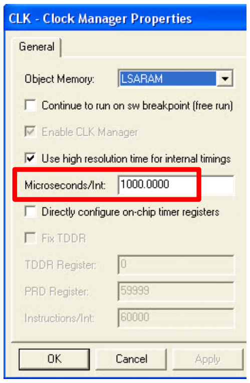
Using Periodic Functions - PRD



- Periodic functions are a special type of SWI that are triggered by DSP/BIOS
- Periodic functions run at a user specified rate:
  - e.g. LED blink requires 0.5 Hz
- Use the CLK Manager to specify the DSP/BIOS CLK rate in microseconds per “tick”
- Use the PRD Manager to specify the period (for the function) in ticks
- Allows multiple periodic functions with different rates

Creating a Periodic Function





CLK - Clock Manager Properties

General

Object Memory: LSARAM

☐ Continue to run on sw breakpoint (free run)

☑ Enable CLK Manager

☑ Use high resolution time for internal timings

Microseconds/Int: 1000.0000

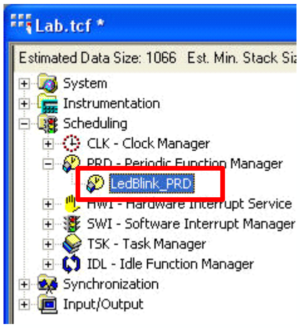
☐ Directly configure on-chip timer registers

☐ Fix TDDR

TDDR Register: 0

PRD Register: 59999

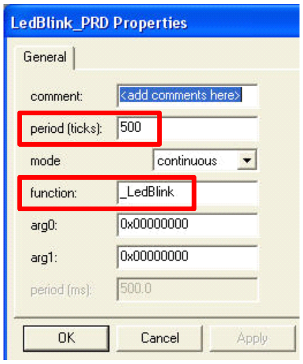
Instructions/Int: 60000



Lab.tcf *

Estimated Data Size: 1066 Est. Min. Stack Si

```
+ System
+ Instrumentation
- Scheduling
    + CLK - Clock Manager
    - PRD - Periodic Function Manager
        LedBlink_PRD
    + HWI - Hardware Interrupt Service
    + SWI - Software Interrupt Manager
    + TSK - Task Manager
    + IDL - Idle Function Manager
+ Synchronization
+ Input/Output
```



LedBlink_PRD Properties

General

comment: <add comments here>

period (ticks): 500

mode: continuous

function: _LedBlink

arg0: 0x00000000

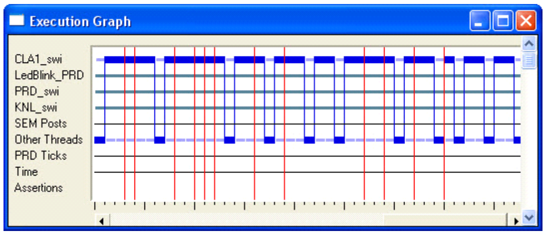
arg1: 0x00000000

period (ms): 500.0

Real-Time Analysis Tools

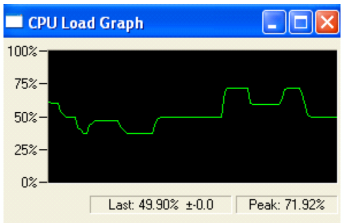
Built-in Real-Time Analysis Tools

- Gather data on target (3-10 CPU cycles)
- Send data during BIOS IDL (100s of cycles)
- Format data on host (1000s of cycles)
- Data gathering does NOT stop target CPU



Execution Graph

- Software logic analyzer
- Debug event timing and priority



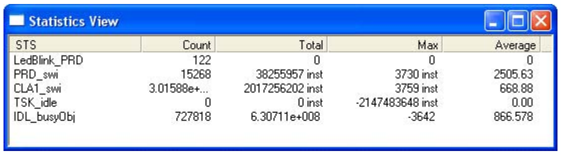
Last: 49.90% ±0.0

Peak: 71.92%

CPU Load Graph

- Shows amount of CPU horsepower being consumed

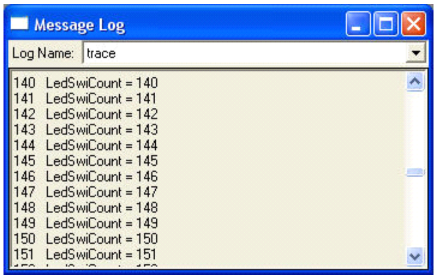
Built-in Real-Time Analysis Tools



STS	Count	Total	Max	Average
LedBlink_PRD	122	0	0	0
PRD_swi	15268	38255957 inst	3730 inst	2505.63
CLA1_swi	3.01588e+...	2017256202 inst	3759 inst	668.88
TSK_idle	0	0 inst	-2147483648 inst	0.00
IDL_busyObj	727818	6.30711e+008	-3642	866.578

Statistics View

- Profile routines w/o halting the CPU



Log Name: trace

```
140  LedSwiCount = 140
141  LedSwiCount = 141
142  LedSwiCount = 142
143  LedSwiCount = 143
144  LedSwiCount = 144
145  LedSwiCount = 145
146  LedSwiCount = 146
147  LedSwiCount = 147
148  LedSwiCount = 148
149  LedSwiCount = 149
150  LedSwiCount = 150
151  LedSwiCount = 151
```

Message LOG

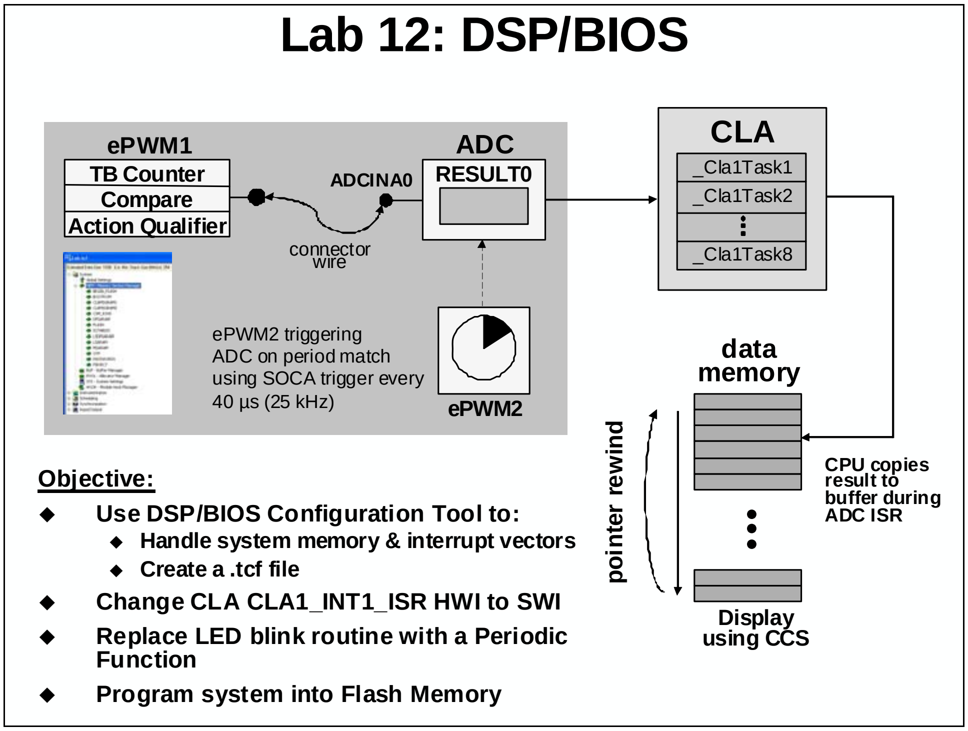
- Send debug msgs to host
- Doesn’t halt the DSP
- Deterministic, low DSP cycle count
- More efficient than traditional printf()

```
LOG_printf(&trace, “LedSwiCount = %u", LedSwiCount++);
```

Lab 12: DSP/BIOS

➢ Objective

The objective of this lab is to become familiar with DSP/BIOS. In this lab exercise, we will make use of the DSP/BIOS configuration tool, implement a software interrupt (SWI) and periodic function (PRD), program the DSP/BIOS project into the flash, and explore the built-in real-time analysis tools. The DSP/BIOS configuration tool creates a text configuration file (*.tcf) and generates a linker command file (*cfg.cmd). This generated linker command file is functionally equivalent to the linker command file previously used. The memory area of the lab linker command file will be deleted; however, part of the sections area will be used to link sections that are not part of DSP/BIOS. In the lab files we will change the CLA HWI (CLA1_INT1_ISR) to a SWI and replace the LED blink routine with a periodic function. The steps required to properly configure the software for execution from internal flash memory will be covered. Features of the real-time analysis tools, such as the CPU Load Graph, Execution Graph, Message Log, and RTA Control Panel will be demonstrated.



Lab 12: DSP/BIOS

Objective:

- Use DSP/BIOS Configuration Tool to:
  - Handle system memory & interrupt vectors
  - Create a .tcf file
- Change CLA CLA1_INT1_ISR HWI to SWI
- Replace LED blink routine with a Periodic Function
- Program system into Flash Memory

➢ Procedure

Project File

1. A project named `Lab12.pjt` has been created for this lab. Open the project by clicking on `Project` → `Open…` and look in `C:\C28x\Labs\Lab12`. All Build Options have been configured the same as the previous lab. The files used in this lab are:

```
Adc.c
Cla_10_12.c
ClaTasks.asm
CodeStartBranch.asm
DefaultIsr_12.c
DelayUs.asm
DSP2803x_GlobalVariableDefs.c
DSP2803x_Headers_BIOS.cmd
ECap_7_8_9_10_12.c
EPwm_7_8_9_10_12.c
Filter.c
Flash.c
Gpio.c
Lab_12.cmd
Main_12.c
Passwords.asm
PieCtrl_12.c
SysCtrl.c
Watchdog.c
```

Edit Lab.h File

2. Edit `Lab.h` to *uncomment* the line that includes the `labcfg.h` header file. This is the DSP/BIOS generated include file, and is needed to allow code to access the DSP/BIOS functions and data structures. Next, *comment out* the line that includes the `“DSP2803x_DefaultIsr.h”` ISR function prototypes. DSP/BIOS will supply its own ISR function prototypes.

3. In our lab setup, we are running the ADC at a 50 kHz interrupt rate. Such a high frequency interrupt would typically be handled directly in the HWI, as SWIs and TSKs have some overhead associated with them and lauching them this frequently can cause very large processing loads on the CPU. DSP/BIOS is flexible in this way. You can have some interrupts processed directly in the HWI, and others delegated to SWIs or TSKs. For purposes of this lab however, we would like to illustrate how to code a SWI. Therefore, we will convert the ADC ISR into a SWI. To reduce the CPU load, we are going to reduce the frequency of the ADC sample rate by half to 25 kHz.

   In `Lab.h` modify the constant definition for the ADC sample rate as follows:

   ```
   #define    ADC_SAMPLE_PERIOD  2399    // 25 KHz sampling
   ```

   Save and close the file.

Remove “rts2800_ml.lib” and Inspect Lab_12.cmd

4. The DSP/BIOS configuration tool supplies its own RTS library. Open the `Build Options` and select the Linker tab. In the Libraries Category, find the `Include Libraries (-l)` box and delete: `rts2800_ml.lib`.

5. Select the Compiler tab. As the project is now configured, we would get a warning at build time stating that the typedef name has already been declared with the same type. This is because it has been defined twice; once in the header files and again in the include file generated by DSP/BIOS. To suppress the warning select Diagnostics Category and find the `Suppress Diagnostic <n> (-pds)`: box. Type in code number `303`. Select `OK` and the `Build Options` window will close.

6. We will be using the DSP/BIOS configuration tool to create a linker command file. Open and inspect `Lab_12.cmd`. Notice that the linker command file does not have a memory

area and includes only a limited sections area. These sections are not part of DSP/BIOS and need to be included in a "user" linker command file. Close the inspected file.

Using the DSP/BIOS Configuration Tool

7. The text configuration file (*.tcf) created by the DSP/BIOS configuration tool controls a wide range of CCS capabilities. The .tcf file will be used to automatically create and perform memory management. Create a new .tcf file for this lab. On the menu bar click:

   `File → New → DSP/BIOS Configuration…`

   A dialog box appears showing a number of available .tcf seed files. The seed files are used to configure many objects specific to the processor and will be invoked as the first item in your own .tcf file. On the C2xxx tab select the **ti.platforms.control28035** template and click OK. A configuration window will open.

8. Save the configuration file by selecting:

   `File → Save As…`

   and name it `Lab.tcf` in `C:\C28x\Labs\Lab12` then click `Save`. Close the configuration window and select `YES` to save changes to `Lab.tcf`.

9. Add the configuration file to the project. Click:

   `Project → Add Files to Project…`

   Make sure you're looking in `C:\C28x\Labs\Lab12`. Change the "files of type" to view All Files (`*.*`) and select `Lab.tcf`. Click `OPEN` to add the file to the project.

10. In the project window left click the plus sign (+) to the left of `DSP/BIOS Config`. Notice that the `Lab.tcf` file is listed.

11. Next, add the generated linker command file `Labcfg.cmd` to the project. After the file has been added you will notice that it is listed under the source files.

Create New Memory Sections Using the TCF File

12. Open the `Lab.tcf` file by double clicking on `Lab.tcf`. In the configuration window, left click the plus sign next to `System` and the plus sign next to `MEM`. By default, the Memory Section Manager has combined the memory space L1, L2 and L3DPSARAM into a single memory block called DPSARAM. It has also combined M0 and M1SARAM into a single memory block called MSARAM.

13. Next, we will add some of the additional memory sections that will be needed for the lab exercises in this module. To add a memory section:

    Right click on `MEM – Memory Section Manager` and select `Insert MEM`. Rename the newly added memory section to `BEGIN_FLASH`. Repeat the process and add the following memory sections: `CLAMSGRAM1`, `CLAMSGRAM2`, `CSM_RSVD`, `IQTABLES`, `L3DPSARAM`, and `PASSWORDS`. *Double check and see that all seven memory sections have been added.*

14. Modify the base addresses, length, and space of each of the memory sections to correspond to the memory mapping shown in the table below. To modify the length, base address, and space of a memory section, right click on the memory in the configuration tool, and select `Properties`.

Memory	**Base**	**Length**	**Space**
BEGIN_FLASH	0x3F 7FF6	0x0002	code
CLAMSGRAM1	0x00 1480	0x0080	data
CLAMSGRAM2	0x00 1500	0x0080	data
CSM_RSVD	0x3F 7F80	0x0076	code
IQTABLES	0x3F E000	0x0B50	code
L3DPSARAM	0x00 9000	0x1000	code
PASSWORDS	0x3F 7FF8	0x0008	code

15. Modify the base addresses, length, and space of each of the memory sections to avoid memory conflicts with the newly added memory sections as shown in the table below.

Memory	**Base**	**Length**	**Space**
BOOTROM	0x3F F27C	0x0D44	code
DPSARAM	0x00 8800	0x0800	data
FLASH	0x3E 8000	0xFF80	code

Link Uninitialized Sections to RAM

16. Right click on `MEM – Memory Section Manager` and select `Properties`. Select the `Compiler Sections` tab and link the following uninitialized sections into the MSARAM memory block via the pull-down boxes.

MSARAM
.bss
.ebss

Link Initialized Sections to Flash

All initialized sections must be linked to the on-chip flash memory. Each initialized section 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.

17. This step assigns the RUN address of those sections that need to run from flash. Using the `MEM – Memory Section Manager` in the DSP/BIOS configuration tool link the following sections to on-chip flash memory via the pull-down boxes:

BIOS Data tab	**BIOS Code tab**	**Compiler Sections tab**
.gblinit	.bios	.text
	.sysinit	.switch
	.hwi	.cinit
	.rtdx_text	.pinit
		.econst / .const
		.data / .cio

18. This step assigns the LOAD address of those sections that need to load to flash. Again using the `MEM – Memory Section Manager` in the DSP/BIOS configuration tool select the **Load Address tab** and check the `“Specify Separate Load Addresses”` box. Then set all entries to the **FLASH** memory block.

19. Click the `BIOS Data` tab and notice that the .stack section has been linked into memory. Click `OK` to close the window.

20. The section named `“IQmath”` is an initialized section that needs to load to and run from flash. This section is not linked using the DSP/BIOS configuration tool (because it is neither a standard compiler section nor a DSP/BIOS generated section). Instead, this section is linked with the user linker command file (`Lab_12.cmd`). Open and inspect `Lab_12.cmd`. Previously the `“IQmath”` section was linked to L0SARAM. Notice that this section is now linked to FLASH.

Set the Stack Size in the TCF File

Recall in the previous lab exercise that the stack size was set using the CCS project Build Options. When using the DSP/BIOS configuration tool, the stack size is instead specified in the .tcf file. First we need to remove the stack size setting from the project Build Options.

21. Click: `Project → Build Options…` and select the Linker tab. Delete the entry of 0x200 in the Stack Size box. Select `OK` to close the Build Options window.

22. Using the `MEM – Memory Section Manager` select the `General` tab. Set the Stack Size to 0x100. The stack size needs to be reduced from 0x200 to 0x100 because of the limited amount of available RAM on the device when using DSP/BIOS. Click `OK` to close the window.

Copying .hwi_vec Section from Flash to RAM

The DSP/BIOS .hwi_vec section contains the interrupt vectors. This section must be loaded to flash (load address) but run from RAM (run address). The code that performs this copy is located in InitPieCtrl(). The linker command file generated by the DSP/BIOS configuration tool generates global symbols that can be accessed by code in order to determine the load address, run address, and length of the .hwi_vec section. The RTS library contains a memory copy function called *memcpy()* which will be used to perform the copy.

23. Open and inspect InitPieCtrl() in `PieCtrl_12.c`. Notice the memcpy() function and the symbols used to initialize (copy) the .hwi_vec section.

Copying the .trcdata Section from Flash to RAM

The DSP/BIOS .trcdata section is used by CCS and DSP/BIOS for certain real-time debugging features. This section must be loaded to flash (load address) but run from RAM (run address). The linker command file generated by the DSP/BIOS configuration tool generates global symbols that can be accessed by code in order to determine the load address, run address, and length of the .trcdata section. The memory copy function *memcpy()* will again be used to perform the copy.

The copying of .trcdata must be performed prior to main(). This is because DSP/BIOS modifies the contents of .trcdata during DSP/BIOS initialization, which also occurs prior to main(). The DSP/BIOS configuration tool provides a user initialization function which will be used to perform the .trcdata section copy prior to both main() and DSP/BIOS initialization.

24. In the DSP/BIOS configuration file (`Lab.tcf`) and select the `Properties` for the `Global Settings`. Check the box `“Call User Init Function”` and enter the `UserInit()` function name with a leading underscore: `_UserInit`. This will cause the function UserInit() to execute prior to main(). Click `OK` to close the window.

25. Open and inspect the file `Main_12.c`. Notice that the function `UserInit()` is used to copy the .trcdata section from its load address to its run address before main().

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.

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

27. Since the DSP/BIOS configuration tool does not know about user defined sections, the `“secureRamFuncs”` section will be linked using the user linker command file `Lab_12.cmd`. Open and inspect `Lab_12.cmd`. The `“secureRamFuncs”` will load to flash (load address) but will run from LSARAM (run address). Also notice that the linker has been asked to generate symbols for the load start, load size, and run start addresses.

28. Open and inspect `Main_12.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. Close all the inspected files.

Setup PIE Vectors for Interrupts in the TCF File

Next, we will setup all of the PIE interrupt vectors that will be needed for the lab exercises in this module. This will include all of the vectors used in the previous lab exercises. (Note: the `PieVect.c` file is not used since DSP/BIOS generates the interrupt vector table).

29. Modify the configuration file `Lab.tcf` to setup the PIE vector for the watchdog interrupt. Click on the plus sign (+) to the left of `Scheduling` and again on the plus sign (+) to the left of `HWI – Hardware Interrupt Service Routine Manager`. Click the plus sign (+) to the left of `PIE INTERRUPTS`. Locate the interrupt entry for the watchdog at `PIE_INT1_8`. Right click, select `Properties`, and type `_WAKEINT_ISR` (with a leading underscore) in the function field. Click `OK` to save.

30. Setup the PIE vector for the ADC interrupt. Locate the interrupt entry for the ADC at `PIE_INT1_1`. Right click, select `Properties`, and type `_ADCINT1_ISR` (with a leading underscore) in the function field. Click `OK` to save.

31. Setup the PIE vector for the ECAP1 interrupt. Locate the interrupt entry for the ECAP1 at `PIE_INT4_1`. Right click, select `Properties`, and type `_ECAP1_INT_ISR` (with a leading underscore) in the function field. Click `OK` to save.

32. Setup the PIE vector for the CLA Task 1 interrupt. Locate the interrupt entry for the CLA Task 1 at `PIE_INT11_1`. Right click, select `Properties`, and type `_CLA1_INT1_ISR` (with a leading underscore) in the function field. Click `OK` to save. Close the configuration window and select `YES` to save changes to `Lab.tcf`.

Prepare main() for DSP/BIOS

33. Open `Main_12.c` and delete the inline assembly code from main() that enables global interrupts. DSP/BIOS will enable global interrupts after main().

34. In `Main_12.c`, remove the endless while() loop from the end of main(). When using DSP/BIOS, you must return from main(). In all DSP/BIOS programs, the main() function should contain all one-time user-defined initialization functions. DSP/BIOS will then take-over control of the software execution. Save and close the file.

Configuring DSP/BIOS Global Settings

35. Open the configuration file `Lab.tcf` and click on the plus sign (+) to the left of `System`. Right click on `Global Settings` and select `Properties`. Confirm that the `“DSP Speed in MHz (CLKOUT)”` field is set to `60` so that it matches the processor speed. Click `OK` to save the value and close the configuration window. This value is used by the CLK manager to calculate the register settings for the on-chip timers and provide the proper time-base for executing CLK functions.

Create a SWI

36. Open `Main_12.c` and notice that at the end of main() two new functions have been added – Cla1Swi() and LedBlink(). We moved part of the CLA1_INT1_ISR() routine from `DefaultIsr_12.c` to this space in `Main_12.c`.

37. Open `DefaultIsr_12.c` and locate the CLA1_INT1_ISR() routine. The entire contents of the CLA1_INT1_ISR() routine was moved to the Cla1Swi() function in `Main_12.c` with the following exceptions:

    - The instruction used to acknowledge the PIE group interrupt

    - The GPIO pin (LED) toggle code

    Comment: In almost all appplications, the PIE group acknowledge code is left in the HWI (rather than move it to a SWI). This allows other interrupts to occur on that PIE group even if the SWI has not yet executed. On the other hand, we are leaving the GPIO toggle code in the HWI just as an example. It illustrates that you can post a SWI and also do additional operations in the HWI. DSP/BIOS is extremely flexible!

38. Delete the `interrupt` key word from the CLA1_INT1_ISR. The interrupt keyword is not used when a HWI is under DSP/BIOS control. A HWI is under DSP/BIOS control when it uses any DSP/BIOS functionality, such as posting a SWI, or calling any DSP/BIOS function or macro.

Post a SWI

39. Still in `DefaultIsr_12.c` add the following SWI_post to the CLA1_INT1_ISR(), just after the structure used to acknowledge the PIE group:

```
SWI_post(&CLA1_swi);          // post a SWI
```

This posts a SWI that will execute the CLA1_swi() code that was moved to the Cla1Swi() function in Main_12.c. In other words, the CLA1 interrupt still executes the same code as before. However, most of that code is now in a posted SWI that DSP/BIOS will execute according to the specified scheduling priorities. Save and close the modified files.

Add the SWI to the TCF File

40. In the configuration file Lab.tcf we need to add and setup the Cla1Swi() SWI. Open Lab.tcf and click on the plus sign (+) to the left of Scheduling and again on the plus sign (+) to the left of SWI – Software Interrupt Manager.

41. Right click on SWI – Software Interrupt Manager and select Insert SWI. Rename SWI0 to CLA1_swi and click OK. This is just an arbitrary name. We want to differentiate the Cla1Swi() function itself (which is nothing but an ordinary C function) from the DSP/BIOS SWI object which we are calling CLA1_swi.

42. Select the Properties for CLA1_swi and type _Cla1Swi (with a leading underscore) in the function field. Click OK. This tells DSP/BIOS that it should run the function Cla1Swi() when it executes the CLA1_swi SWI.

43. We need to have the PIE for the CLA Task 1 interrupt use the dispatcher. The dispatcher will automatically perform the context save and restore, and allow the DSP/BIOS scheduler to have insight into the ISR. You may recall from an earlier lab that the CLA Task 1 interrupt is located at PIE_INT11_1.

    Click on the plus sign (+) to the left of HWI – Hardware Interrupt Service Routine Manager. Click the plus sign (+) to the left of PIE INTERRUPTS. Locate the interrupt entry for the CLA Task 1: PIE_INT11_1. Right click, select Properties, and select the Dispatcher tab. Check the "Use Dispatcher" box and select OK. Close the configuration file and click YES to save changes.

Add a Periodic Function

Recall that an instruction was used in the CLA1_INT1_ISR to toggle the LED on the ControlCARD. This instruction has been moved into a periodic function that will toggle the LED at the same rate.

44. Open DefaultIsr_12.c and locate the CLA1_INT1_ISR routine. Notice that the instruction used to toggle the LED was moved to the LedBlink() function in Main_12.c:

    ```
    GpioDataRegs.GPBTOGGLE.bit.GPIO34 = 1;    // Toggle the pin
    ```

    Also, the code used to implement the interval counter for the LED toggle (i.e., the GPIO32_count++ loop), and the declaration of the GPIO32_count itself from the beginning of CLA1_INT1_ISR() have been deleted. These are no longer needed, as DSP/BIOS will implement the interval counter for us in the periodic function configuration (next step in the lab). Close the inspected files.

45. In the configuration file `Lab.tcf` we need to add and setup the LedBlink_PRD. Open `Lab.tcf` and click on the plus sign (+) to the left of `Scheduling`. Right click on `PRD – Periodic Function Manger` and select `Insert PRD`. Rename `PRD0` to `LedBlink_PRD` and click `OK`.

    Select the `Properties` for `LedBlink_PRD` and type `_LedBlink` (with a leading underscore) in the function field. This tells DSP/BIOS to run the LedBlink() function when it executes the LedBlink_PRD periodic function object.

    Next, in the period (ticks) field type `500`. The default DSP/BIOS system timer increments every 1 millisecond, so what we are doing is telling the DSP/BIOS scheduler to schedule the LedBlink() function to execute every 500 milliseconds. A PRD object is just a special type of SWI which gets scheduled periodically and runs in the context of the SWI level at a specified SWI priority. Click `OK`. Close the configuration file and click `YES` to save changes.

DSP/BIOS – Real-time Analysis Tools

The DSP/BIOS analysis tools complement the CCS environment by enabling real-time program analysis of a DSP/BIOS application. You can visually monitor an MCU application as it runs with essentially no impact on the application's real-time performance. In CCS, the DSP/BIOS realt-time analysis (RTA) tools are found on the DSP/BIOS menu. Unlike traditional debugging, which is external to the executing program, DSP/BIOS program analysis requires that the target program be instrumented with analysis code. By using DSP/BIOS APIs and objects, developers automatically instrument the target for capturing and uploading real-time information to CCS using these tools.

46. In the next few steps the Log Event Manager will be setup to record the occurrence of an event in real-time while the program executes. We will be using `LOG_printf()` to write to a log buffer. The `LOG_printf()` function is a very efficient means of sending a message from the code to the CCS display. Unlike an ordinary C-language printf(), which can consume several hundred CPU cycles to format the data on the MCU before transmission to the CCS host PC, a LOG_printf() transmits the raw data to the host. The host then formats the data and displays it in CCS. This consumes only 10's of cycles rather than 100's of cycles.

    In `Main_12.c` notice the following code at the top of the LedBlink() function just before the instruction used to toggle the LED:

    ```
    static Uint16 LedSwiCount=0;          // used for LOG_printf

    /*** Using LOG_printf() to write to a log buffer ***/

         LOG_printf(&trace, "LedSwiCount = %u", LedSwiCount++);
    ```

    Close the file.

47. In the configuration file `Lab.tcf` we need to add and setup the trace buffer. Open `Lab.tcf` and click on the plus sign (+) to the left of `Instrumentation` and again on the plus sign (+) to the left of `LOG – Event Log Manager`.

48. Right click on `LOG – Event Log Manager` and select `Insert LOG`. Rename `LOG0` to `trace` and click `OK`.

49. Select the `Properties` for `trace` and confirm that the logtype is set to *circular* and the datatype is set to *printf*. Click `OK`. Close the configuration file and click `YES` to save changes.

Build – Lab.out

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

    Next select the `"Debug Properties"` tab and confirm that the `"Step over functions without debug information when source stepping"` is <u>*unchecked*</u>. Then click `OK`.

51. Click the `"Build"` button to generate `Lab.out`.

CCS Flash Plug-in

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

    `Tools → F28xx On-Chip Flash Programmer`

53. A Clock Configuration window *may* open. If needed, in the Clock Configuration window set "OSCCLK (MHz):" to 10, "DIVSEL:" to /2, and "PLLCR Value:" to 12. Then click `OK`. In the next Flash Programmer Settings window confirm that the selected DSP device to program is F28035 and all options have been checked. Click `OK`.

54. The CCS Flash Programmer uses the Piccolo™ 10 MHz internal oscillator as the device clock during programming. Confirm the "Clock Configuration" in the upper left corner 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.

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

56. We will not be using the plug-in to program the "Code Security Password". ***Do not modify the Code Security Password fields.*** They should remain as all 0xFFFF.

57. In the "Operation" block, notice that the "COFF file to Program/Verify" field automatically defaults to the current `.out` file. Check to be sure that "Erase, Program, Verify" is selected. We will be using the default wait states, as shown on the slide in this module. The selection for wait-states only affects the verify step, and makes little noticeable difference even if you reduce the wait-states.

58. Click "Execute Operation" to program the flash memory. Watch the programming status update in the plug-in window.

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

Running the Code – Using CCS

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

    `File → Load Symbols → Load Symbols Only…`

    and select `Lab12.out` in the `Debug` folder.

61. Reset the CPU. The program counter should now be at 0x3FF8A1, which is the start of the bootloader in the Boot ROM.

62. Under `GEL` 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 0x3F7FF6.

63. Single-Step <F11> through the bootloader code until you arrive at the beginning of the codestart section in the `CodeStartBranch.asm` file. (Be patient, it will take about 125 single-steps). Notice that we have placed some code in `CodeStartBranch.asm` to give an option to first disable the watchdog, if selected.

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

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

66. You can now RUN the CPU, and you should observe the LED on the ControlCARD 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.

Run the Code – Real-time Analysis Tools

It will be interesting to investigate the CPU computational burden of the the different pieces of DSP/BIOS real-time analysis tools that we will be using in this lab exercise. The 'CPU Load Graph' feature of DSP/BIOS will provide a quick and easy method for doing this. We will be tabulating these results in the table that follows at various steps throughout the remainder of this lab.

Table 12-1: CPU Computational Burden Results

Case #	Description	CPU Load %
1	CLA processing handled in SWI. LED blink handled in PRD. RTA Global Host Enable disabled.	
2	Case #1 + LOG_printf in SWI.	
3	Case #2 + RTA SWI Logging enabled.	
4	Case #3 + RTA SWI Accumulators enabled.	

67. Open the RTA Control Panel by clicking `DSP/BIOS → RTA Control Panel`. Uncheck ALL of the boxes. This disables most of the realtime analysis tools. We will selectively enable them in the lab.

68. Open the CPU Load Graph by clicking `DSP/BIOS → CPU Load Graph`. The CPU load graph displays the percentage of available CPU computing horsepower that the application is consuming. The CPU may be running ISRs, software interrupts, periodic functions, performing I/O with the host, or running any user routine. When the CPU is not executing user code, it will be idle (in the DSP/BIOS idle thread).

69. Record the value shown in the CPU Load Graph under “Case #1” in Table 12-1.

70. Open the *Message Log*. On the menu bar, click:

    `DSP/BIOS → Message Log`

    The message log dialog box is displaying the commanded LOG_printf() output, i.e. the number of times (count value) that the LedSwi() has executed.

71. Verify that all the check boxes in the RTA Control Panel window are still unchecked. Then, check the box marked “Global Host Enable.” This is the main control switch for most of the RTA tools. We will be selectively enabling the rest of the check boxes in this portion of the exercise.

72. Record the value shown in the CPU Load Graph under “Case #2” in Table 12-1.

73. Open the *Execution Graph*. On the menu bar, click:

    `DSP/BIOS → Execution Graph`

Presently, the execution graph is not displaying anything. This is because we have it disabled in the RTA Control Panel.

In the RTA Control Panel, check the top four boxes to enable logging of all event types to the execution graph. Notice that the Execution Graph is now displaying information about the execution threads being taken by your software. This graph is not based on time, but the activity of events (i.e. when an event happens, such as a SWI or periodic function begins execution). Notice that the execution graph simply records DSP/BIOS CLK events along with other system events (the DSP/BIOS clock periodically triggers the DSP/BIOS scheduler). As a result, the time scale on the execution graph is not linear.

The logging of events to the execution graph consumes CPU cycles, which is why the CPU Load Graph jumped as you enabled logging.

74. Record the value shown in the CPU Load Graph under "Case #3" in Table 12-1.

75. Open the *Statistics View* window. On the menu bar, click:

```
DSP/BIOS → Statistics View
```

Presently, the statistics view window is not changing with the exception of the statistics for the IDL_busyObj row (i.e., the idle loop). This is because we have it disabled in the RTA Control Panel.

In the RTA Control Panel, check the next five boxes (i.e., those with the word "Accumulator" in their description) to enable logging of statistics to the statistics view window. The logging of statistics consumes CPU cycles, which is why the CPU Load Graph jumped as you enabled logging.

76. Record the value shown in the CPU Load Graph under "Case #4" in Table 12-1.

77. Table 12-1 should now be completely filled in. Think about the results.

Note: In this lab exercise only the basic features of DSP/BIOS and the real-time analysis tools have been used. For more information and details, please refer to the DSP/BIOS user's manuals and other DSP/BIOS related training.

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

78. Close Code Composer Studio.

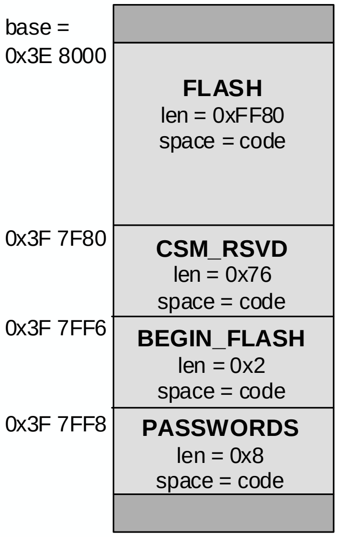
79. Disconnect the USB cable (emulator) from the Docking Station (i.e. remove power from the ControlCARD).

80. Re-connect the USB cable to the Docking Station to power the ControlCARD. The LED should be blinking, showing that the code is now running from flash memory.

End of Exercise

Lab 12 Reference: Programming the Flash

Flash Memory Section Blocks



base =
0x3E 8000

FLASH
len = 0xFF80
space = code

0x3F 7F80

CSM_RSVD
len = 0x76
space = code

0x3F 7FF6

BEGIN_FLASH
len = 0x2
space = code

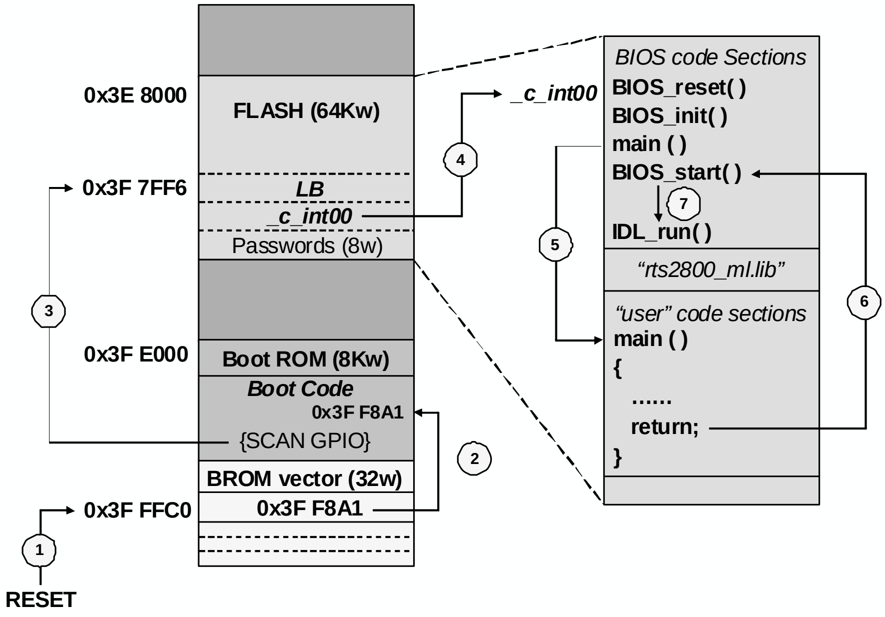
0x3F 7FF8

PASSWORDS
len = 0x8
space = code

Lab_12.cmd

```
SECTIONS
{
  codestart   :>  BEGIN_FLASH,  PAGE = 0
  passwords   :>  PASSWORDS,    PAGE = 0
  csm_rsvd    :>  CSM_RSVD,     PAGE = 0
}
```

BIOS Startup Sequence from Flash Memory



0x3E 8000

FLASH (64Kw)

0x3F 7FF6

LB

_c_int00

Passwords (8w)

0x3F E000

Boot ROM (8Kw)

Boot Code

0x3F F8A1

{SCAN GPIO}

BROM vector (32w)

0x3F FFC0

0x3F F8A1

RESET

1

2

3

4

5

6

7

_c_int00

BIOS code Sections

BIOS_reset( )

BIOS_init( )

main ( )

BIOS_start( )

IDL_run( )

"rts2800_ml.lib"

"user" code sections

```
main ( )
{
  ......
  return;
}
```

Table 12-2: CPU Computational Burden Results (Solution)

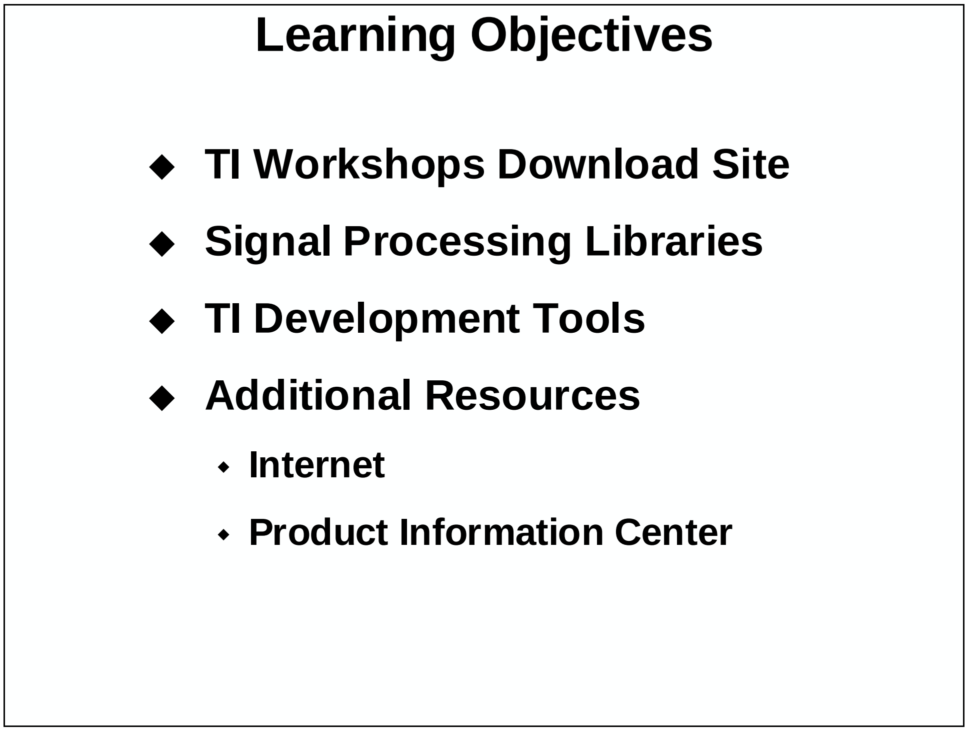
Case #	Description	CPU Load %
1	CLA processing handled in SWI. LED blink handled in PRD. RTA Global Host Enable disabled.	27.5
2	Case #1 + LOG_printf in SWI.	27.5
3	Case #2 + RTA SWI Logging enabled.	37.0
4	Case #3 + RTA SWI Accumulators enabled.	48.6

Development Support

Introduction

This module contains various references to support the development process.

Learning Objectives



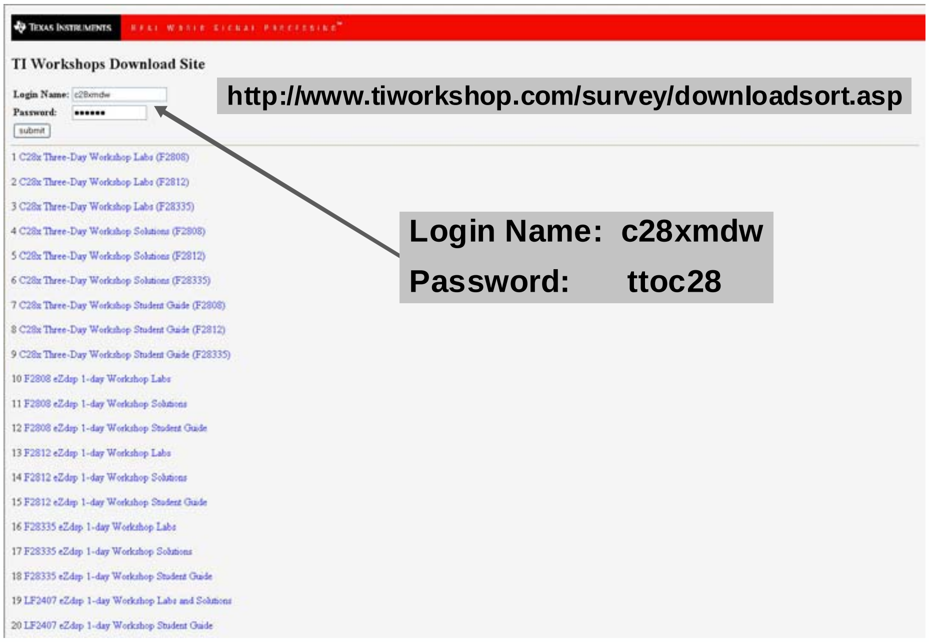
Learning Objectives

- TI Workshops Download Site
- Signal Processing Libraries
- TI Development Tools
- Additional Resources
  - Internet
  - Product Information Center

Module Topics

TI Support Resources

TI Workshops Download Site



TI Workshops Download Site

Login Name: c28xmdw
Password: ••••••

1 C28x Three-Day Workshop Labs (F2808)
2 C28x Three-Day Workshop Labs (F2812)
3 C28x Three-Day Workshop Labs (F28335)
4 C28x Three-Day Workshop Solutions (F2808)
5 C28x Three-Day Workshop Solutions (F2812)
6 C28x Three-Day Workshop Solutions (F28335)
7 C28x Three-Day Workshop Student Guide (F2808)
8 C28x Three-Day Workshop Student Guide (F2812)
9 C28x Three-Day Workshop Student Guide (F28335)
10 F2808 eZdsp 1-day Workshop Labs
11 F2808 eZdsp 1-day Workshop Solutions
12 F2808 eZdsp 1-day Workshop Student Guide
13 F2812 eZdsp 1-day Workshop Labs
14 F2812 eZdsp 1-day Workshop Solutions
15 F2812 eZdsp 1-day Workshop Student Guide
16 F28335 eZdsp 1-day Workshop Labs
17 F28335 eZdsp 1-day Workshop Solutions
18 F28335 eZdsp 1-day Workshop Student Guide
19 LF2407 eZdsp 1-day Workshop Labs and Solutions
20 LF2407 eZdsp 1-day Workshop Student Guide

http://www.tiworkshop.com/survey/downloadsort.asp

Login Name: c28xmdw
Password: ttoc28

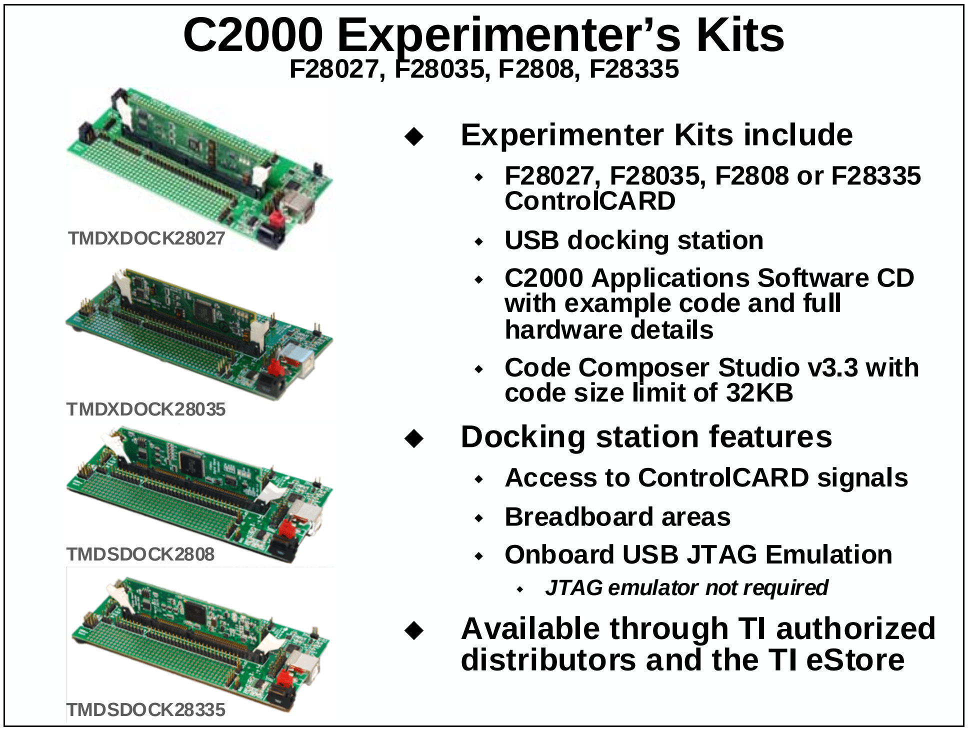
C28x Signal Processing Libraries

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
DSP2803x C/C++ Header Files and Peripheral Examples	SPRC892

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

Experimenter's Kits



C2000 Experimenter's Kits

F28027, F28035, F2808, F28335

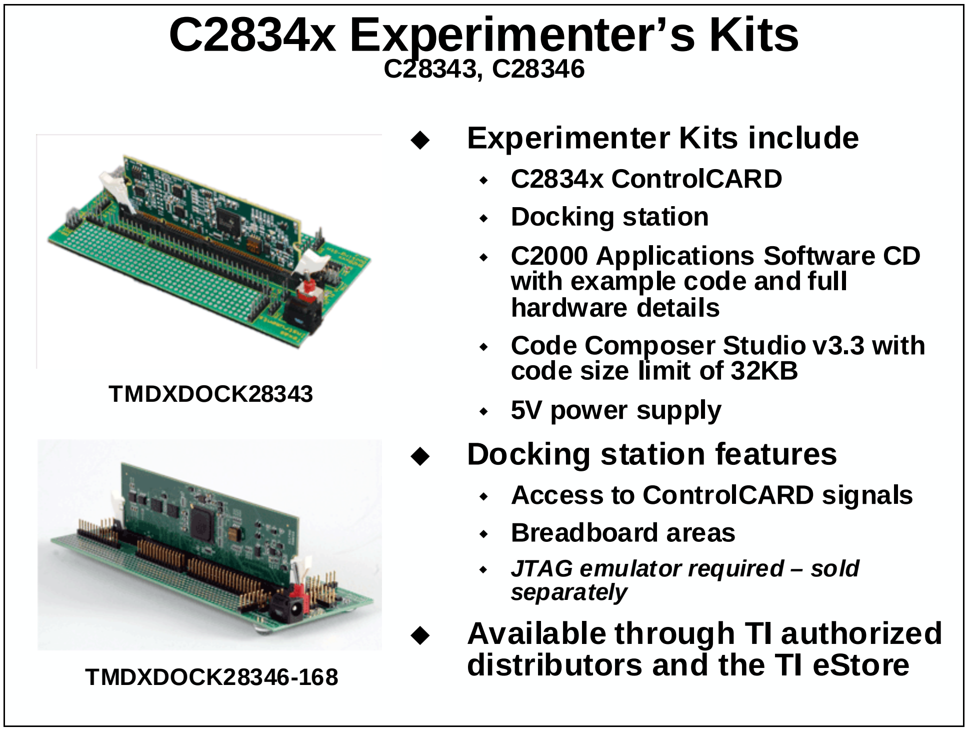
TMDXDOCK28027

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

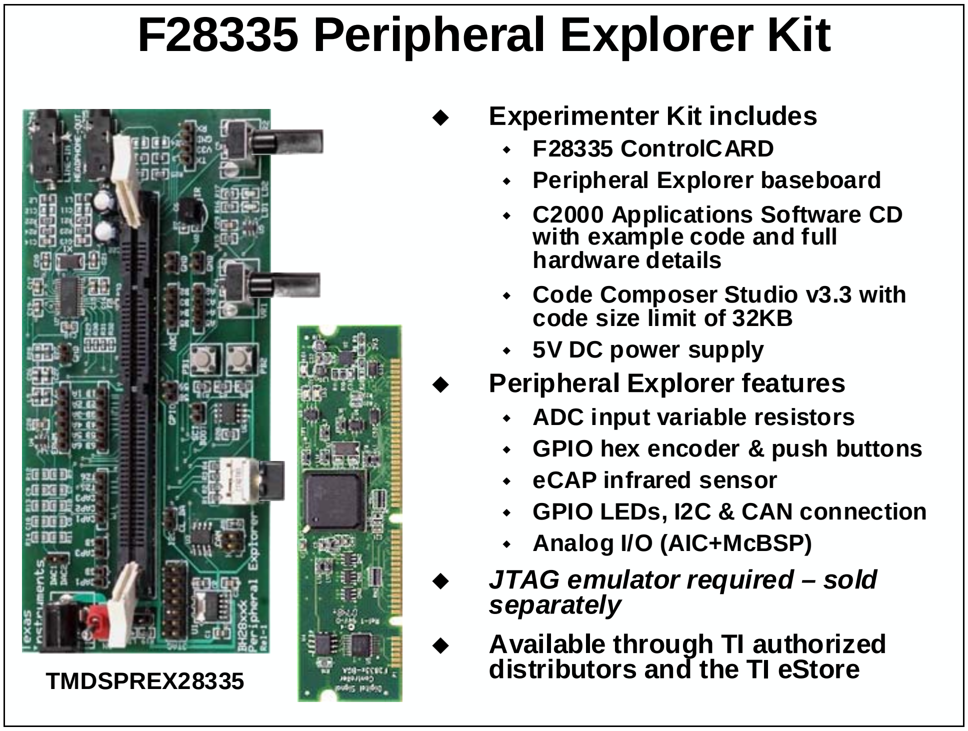
C28343, C28346

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

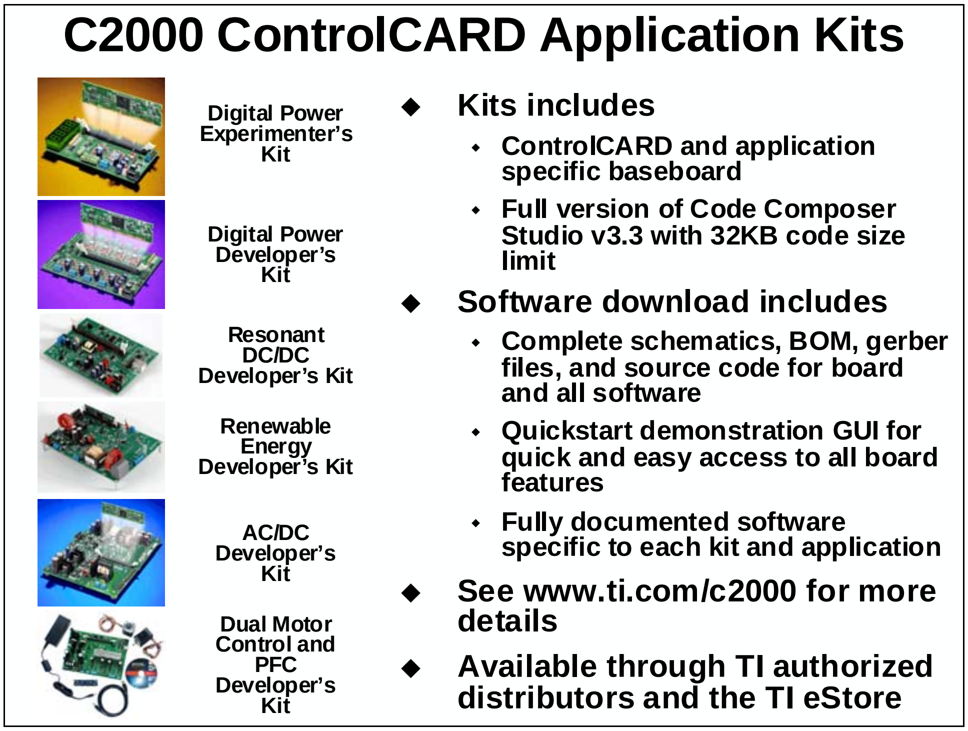


F28335 Peripheral Explorer Kit

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



C2000 ControlCARD Application Kits

Digital Power Experimenter's Kit

Digital Power Developer's Kit

Resonant DC/DC Developer's Kit

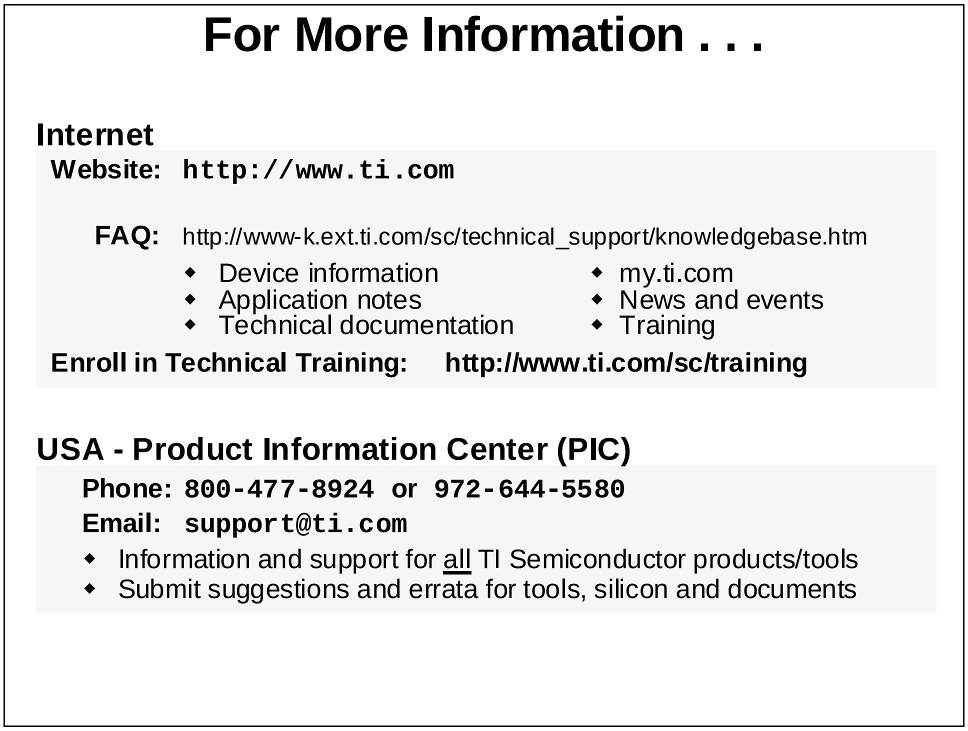
Renewable Energy Developer's Kit

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

Product Information Resources



For More Information . . .

Internet

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

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

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

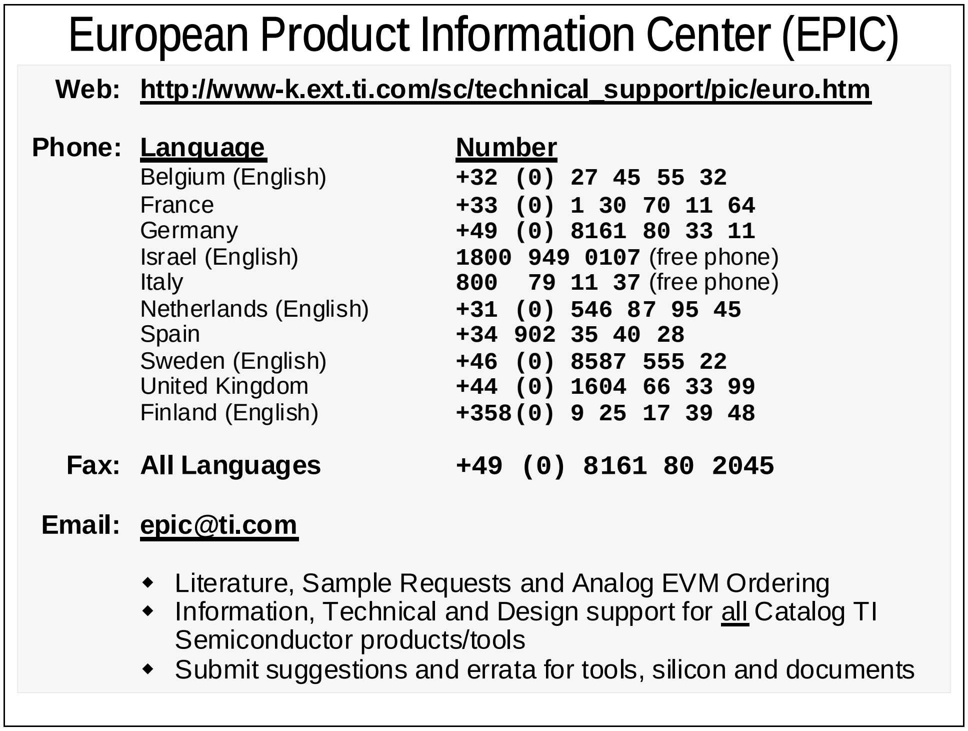
Enroll in Technical Training: **http://www.ti.com/sc/training**

USA - Product Information Center (PIC)

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

Email: **`support@ti.com`**

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



European Product Information Center (EPIC)

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

Phone:	**<u>Language</u>**	**<u>Number</u>**
	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: **<u>epic@ti.com</u>**

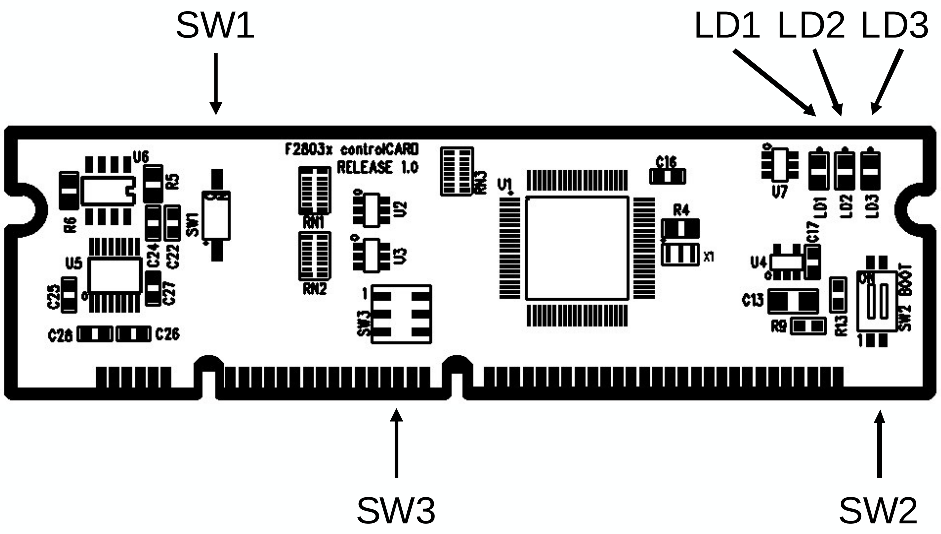
- Literature, Sample Requests and Analog EVM Ordering
- Information, Technical and Design support for <u>all</u> Catalog TI Semiconductor products/tools
- Submit suggestions and errata for tools, silicon and documents

Appendix A – Experimenter’s Kit

Module Topics

F28035 ControlCARD

F28035 PCB Outline (Top View)



LD1 / LD2 / LD3

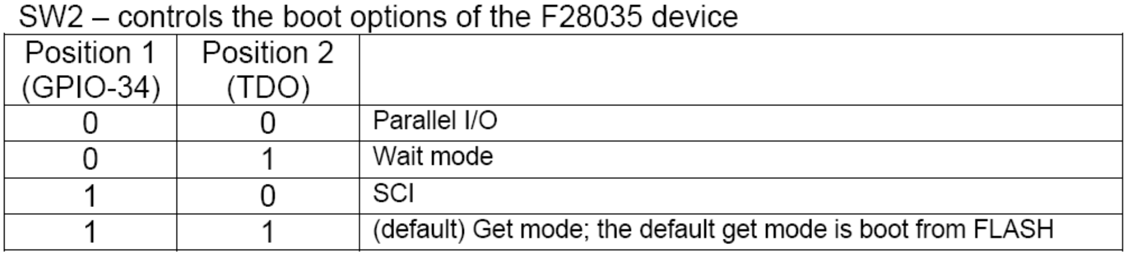
LD1 – Turns on when controlCARD is powered on
LD2 – Controlled by GPIO-31
LD3 – Controlled by GPIO-34

SW1

SW1 – controls whether on-card RS-232 connection is enabled or disabled.

- ON – RS-232 transceiver will be enabled and allow communication through a serial cable via pins 2 and 42 of the DIMM-100 socket. Putting SW1 in the "ON" position will allow the F28035 controlCARD to be card compatible with the F2808, F28044, F28335, and F28027 controlCARDs. GPIO-28 will be stuck as logic high in this position.
- OFF – The default option. SW1 in the "OFF" position allows GPIO-28 to be used as a GPIO. Serial communication is still possible, however an external transceiver such as the FTDI – FT2232D chip.

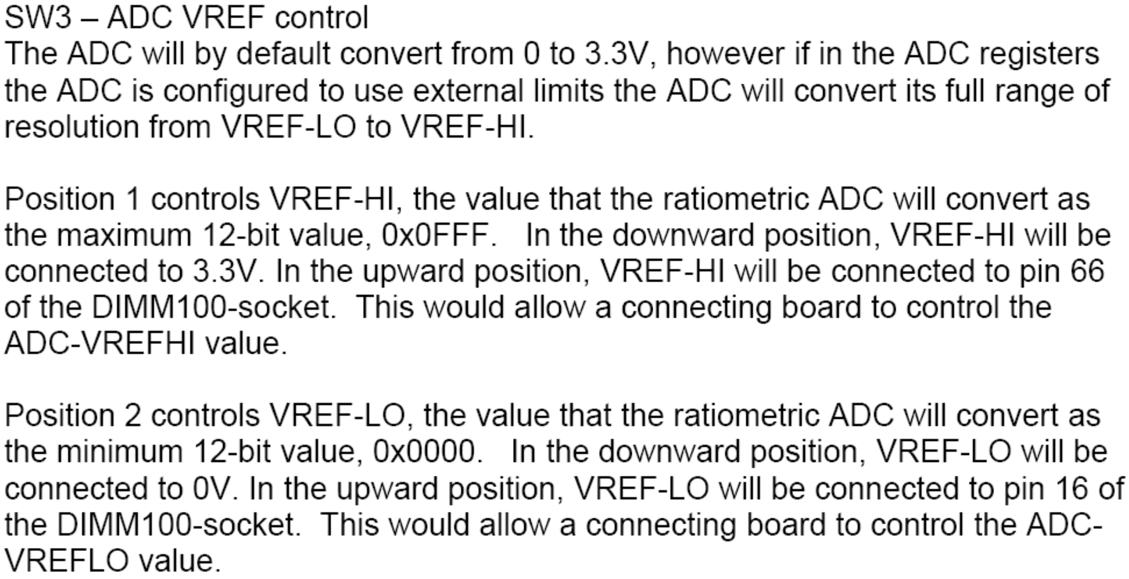
SW2



SW2 – controls the boot options of the F28035 device

Position 1 (GPIO-34)	Position 2 (TDO)	
0	0	Parallel I/O
0	1	Wait mode
1	0	SCI
1	1	(default) Get mode; the default get mode is boot from FLASH

SW3



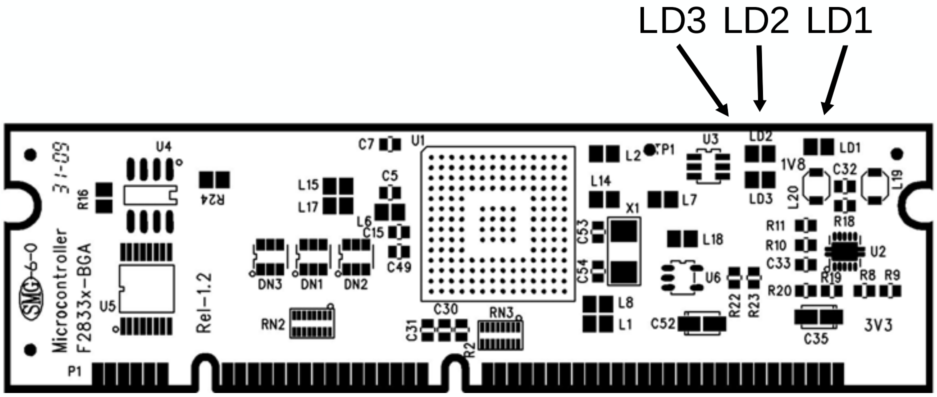
SW3 – ADC VREF control
The ADC will by default convert from 0 to 3.3V, however if in the ADC registers the ADC is configured to use external limits the ADC will convert its full range of resolution from VREF-LO to VREF-HI.

Position 1 controls VREF-HI, the value that the ratiometric ADC will convert as the maximum 12-bit value, 0x0FFF. In the downward position, VREF-HI will be connected to 3.3V. In the upward position, VREF-HI will be connected to pin 66 of the DIMM100-socket. This would allow a connecting board to control the ADC-VREFHI value.

Position 2 controls VREF-LO, the value that the ratiometric ADC will convert as the minimum 12-bit value, 0x0000. In the downward position, VREF-LO will be connected to 0V. In the upward position, VREF-LO will be connected to pin 16 of the DIMM100-socket. This would allow a connecting board to control the ADC-VREFLO value.

F28335 ControlCARD

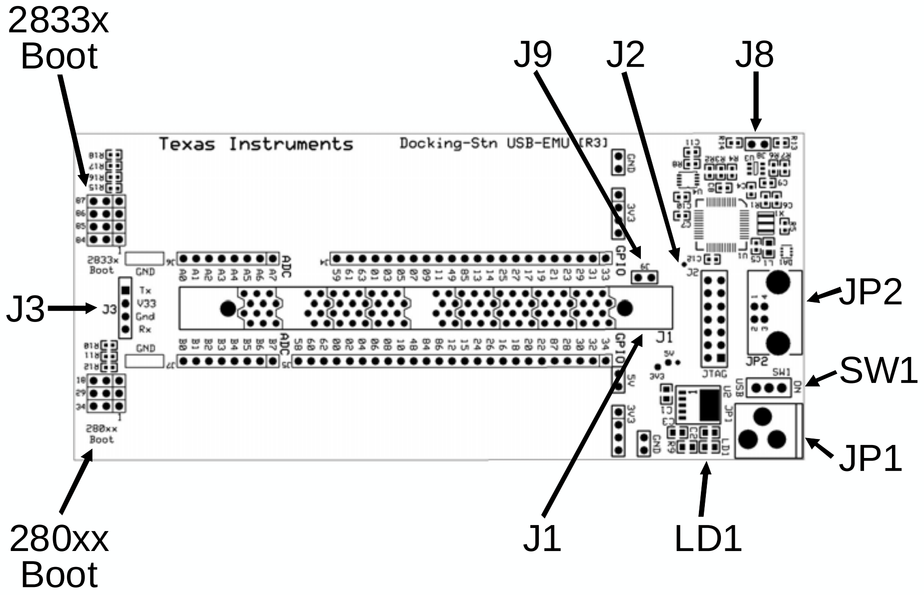
F28335 PCB Outline (Top View)



LD1 / LD2 / LD3

LD1 – Turns on when controlCARD is powered on
LD2 – Controlled by GPIO-31
LD3 – Controlled by GPIO-34

Docking Station



SW1 / LD1

SW1 – USB: Power from USB; ON – Power from JP1

LD1 – Power-On indicator

JP1 / JP2

JP1 – 5.0 V power supply input

JP2 – USB JTAG emulation port

J1 / J2 /J3 / J8 / J9

J1 – ControlCARD 100-pin DIMM socket

J2 – JTAG header connector

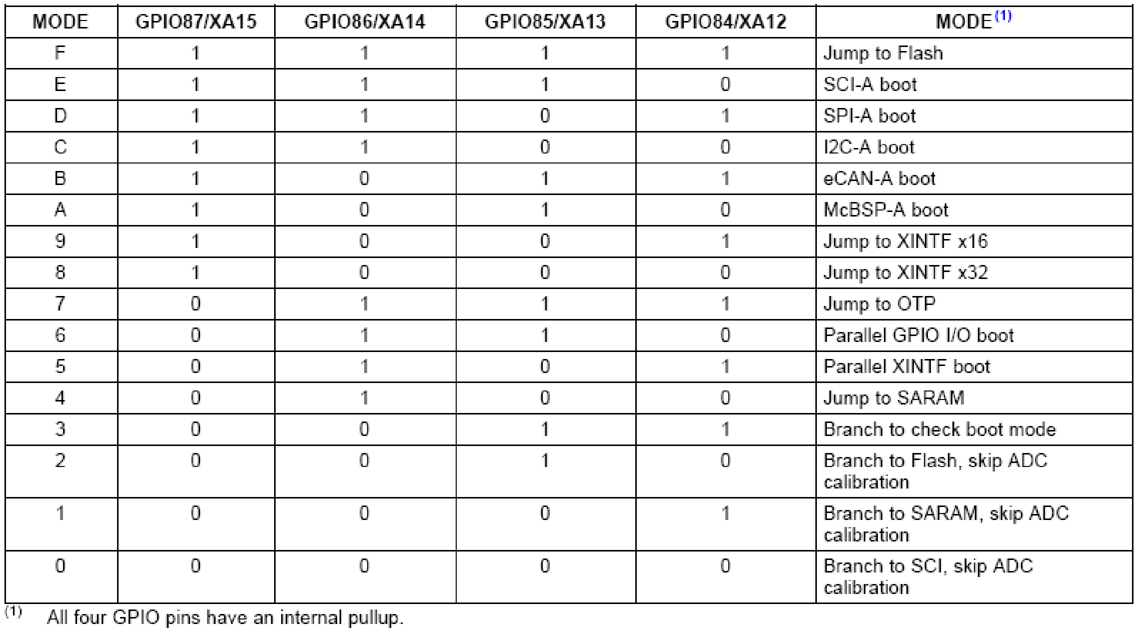
J3 – UART communications header connector

J8 – Internal emulation enable/disable jumper (NO jumper for internal emulation)

J9 – User virtual COM port to C2000 device (Note: ControlCARD would need to be modified to disconnect the C2000 UART connection from header J3)

Note: The internal emulation logic on the Docking Station routes through the FT2232 USB device. By default this device enables the USB connection to perform JTAG communication and in parallel create a virtual serial port (SCI/UART). As shipped, the C2000 device is not connected to the virtual COM port and is instead connected to J3.

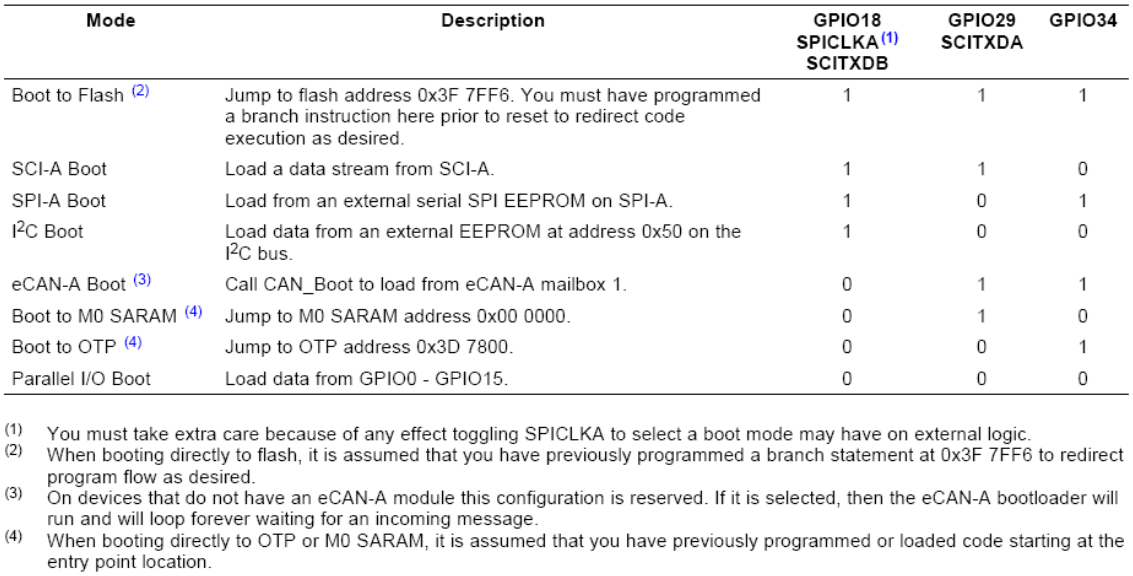
F2833x Boot Mode Selection



MODE	GPIO87/XA15	GPIO86/XA14	GPIO85/XA13	GPIO84/XA12	MODE[1]
F	1	1	1	1	Jump to Flash
E	1	1	1	0	SCI-A boot
D	1	1	0	1	SPI-A boot
C	1	1	0	0	I2C-A boot
B	1	0	1	1	eCAN-A boot
A	1	0	1	0	McBSP-A boot
9	1	0	0	1	Jump to XINTF x16
8	1	0	0	0	Jump to XINTF x32
7	0	1	1	1	Jump to OTP
6	0	1	1	0	Parallel GPIO I/O boot
5	0	1	0	1	Parallel XINTF boot
4	0	1	0	0	Jump to SARAM
3	0	0	1	1	Branch to check boot mode
2	0	0	1	0	Branch to Flash, skip ADC calibration
1	0	0	0	1	Branch to SARAM, skip ADC calibration
0	0	0	0	0	Branch to SCI, skip ADC calibration

[1] All four GPIO pins have an internal pullup.

F280xx Boot Mode Selection



Mode	Description	GPIO18 SPICLKA[1] SCITXDB	GPIO29 SCITXDA	GPIO34
Boot to Flash [2]	Jump to flash address 0x3F 7FF6. You must have programmed a branch instruction here prior to reset to redirect code execution as desired.	1	1	1
SCI-A Boot	Load a data stream from SCI-A.	1	1	0
SPI-A Boot	Load from an external serial SPI EEPROM on SPI-A.	1	0	1
I^{2}C Boot	Load data from an external EEPROM at address 0x50 on the I^{2}C bus.	1	0	0
eCAN-A Boot [3]	Call CAN_Boot to load from eCAN-A mailbox 1.	0	1	1
Boot to M0 SARAM [4]	Jump to M0 SARAM address 0x00 0000.	0	1	0
Boot to OTP [4]	Jump to OTP address 0x3D 7800.	0	0	1
Parallel I/O Boot	Load data from GPIO0 - GPIO15.	0	0	0

[1] You must take extra care because of any effect toggling SPICLKA to select a boot mode may have on external logic.

[2] When booting directly to flash, it is assumed that you have previously programmed a branch statement at 0x3F 7FF6 to redirect program flow as desired.

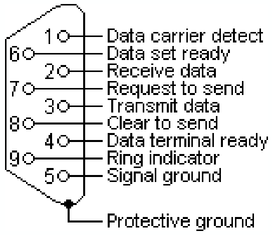
[3] On devices that do not have an eCAN-A module this configuration is reserved. If it is selected, then the eCAN-A bootloader will run and will loop forever waiting for an incoming message.

[4] When booting directly to OTP or M0 SARAM, it is assumed that you have previously programmed or loaded code starting at the entry point location.

J3 – DB-9 to 4-Pin Header Cable

Note: This cable is NOT included with the Experimenter’s Kit and is only shown for reference.

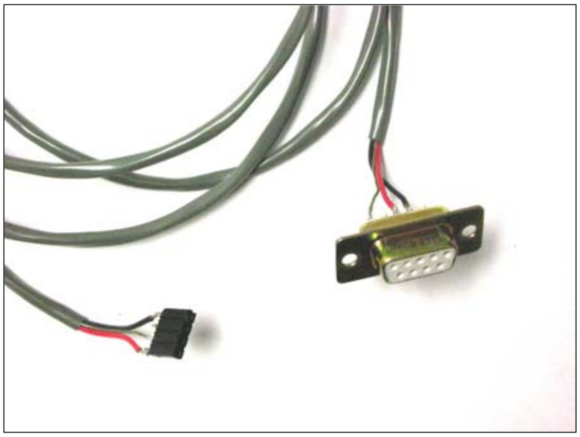
DB-9 Male



Pin-Out Table for Both Ends of the Cable:

DB-9 female	SIL 0.1" female
Pin#	**Pin#**
2 (black)	**1 (TX)**
3 (red)	**4 (RX)**
5 (bare wire)	**3 (GND)**

Note: pin 2 on SIL is a no-connect

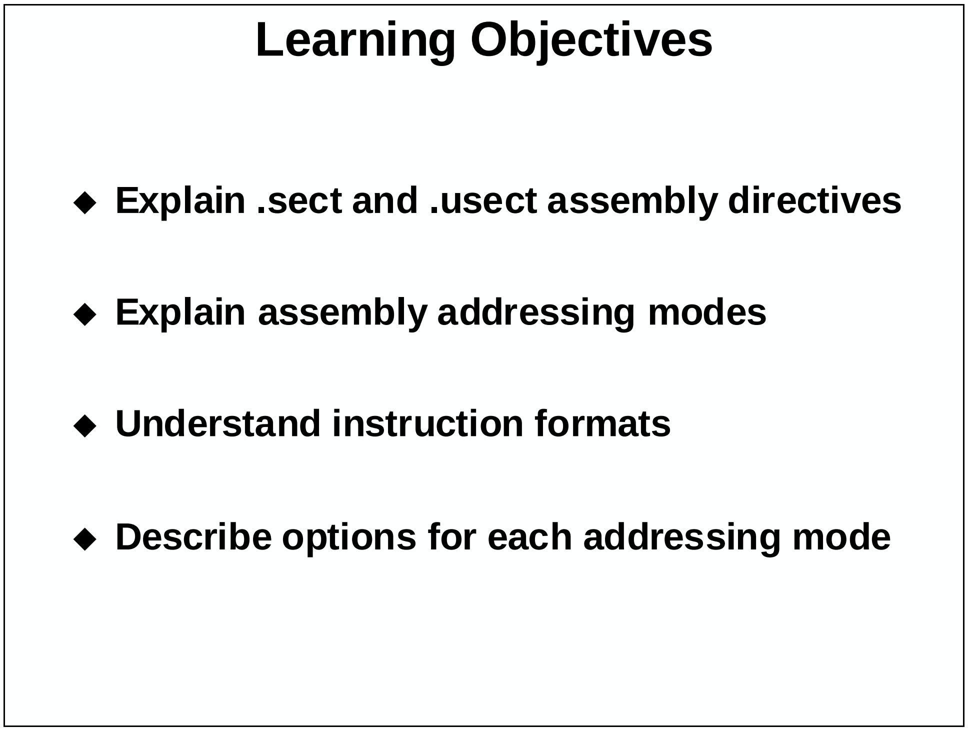


Appendix B – Addressing Modes

Introduction

Appendix B will describe the data addressing modes on the C28x. Immediate addressing allows for constant expressions which are especially useful in the initialization process. Indirect addressing uses auxiliary registers as pointers for accessing organized data in arrays. Direct addressing is used to access general purpose memory. Techniques for managing data pages, relevant to direct addressing will be covered as well. Finally, register addressing allows for interchange between CPU registers.

Learning Objectives

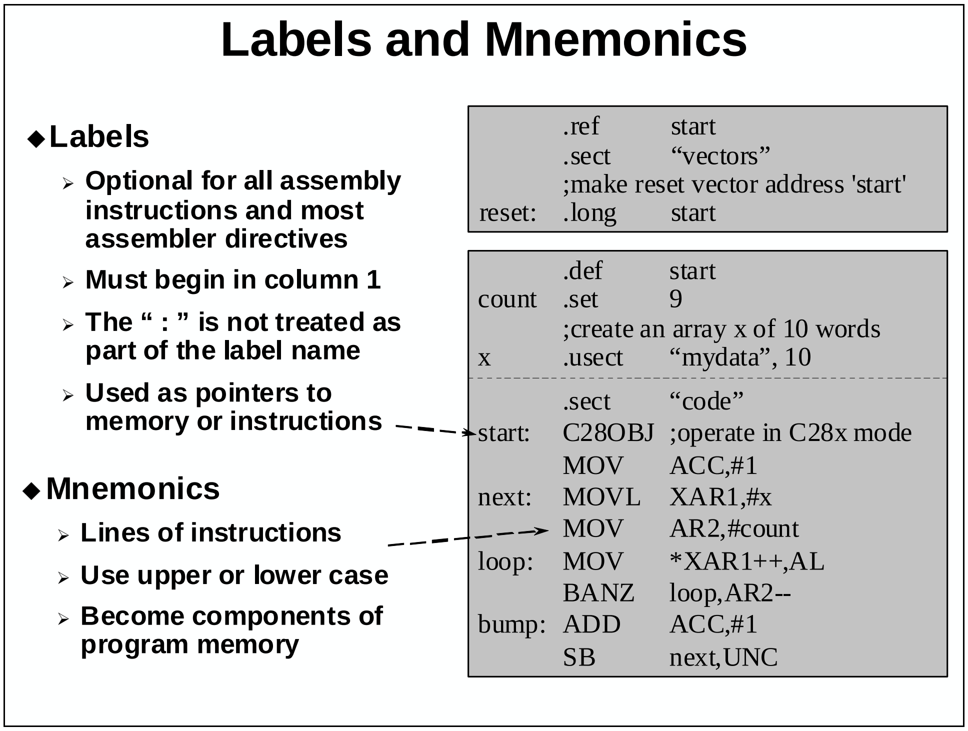


Learning Objectives

- Explain .sect and .usect assembly directives
- Explain assembly addressing modes
- Understand instruction formats
- Describe options for each addressing mode

Module Topics

Labels, Mnemonics and Assembly Directives

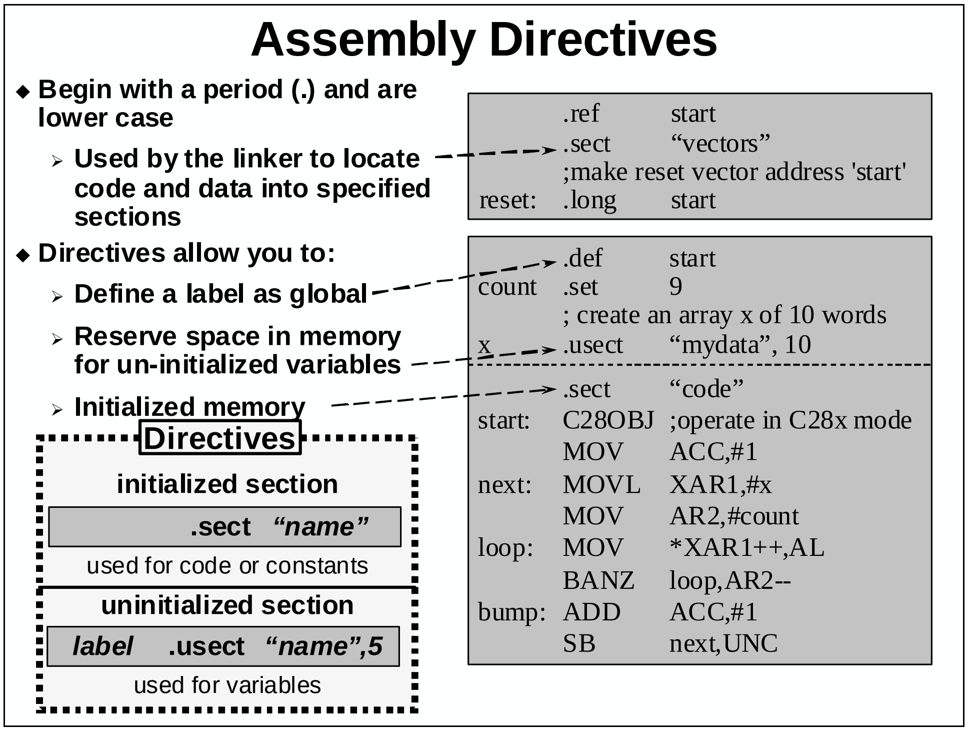


Labels and Mnemonics

- **Labels**
  - **Optional for all assembly instructions and most assembler directives**
  - **Must begin in column 1**
  - **The " : " is not treated as part of the label name**
  - **Used as pointers to memory or instructions**
- **Mnemonics**
  - **Lines of instructions**
  - **Use upper or lower case**
  - **Become components of program memory**

```
          .ref      start
          .sect     "vectors"
          ;make reset vector address 'start'
reset:    .long     start
```

```
          .def      start
count     .set      9
          ;create an array x of 10 words
x         .usect    "mydata", 10
-------------------------------------------------
          .sect     "code"
start:    C28OBJ    ;operate in C28x mode
          MOV       ACC,#1
next:     MOVL      XAR1,#x
          MOV       AR2,#count
loop:     MOV       *XAR1++,AL
          BANZ      loop,AR2--
bump:     ADD       ACC,#1
          SB        next,UNC
```



Assembly Directives

- **Begin with a period (.) and are lower case**
  - **Used by the linker to locate code and data into specified sections**
- **Directives allow you to:**
  - **Define a label as global**
  - **Reserve space in memory for un-initialized variables**
  - **Initialized memory**

Directives

initialized section

.sect *"name"*

used for code or constants

uninitialized section

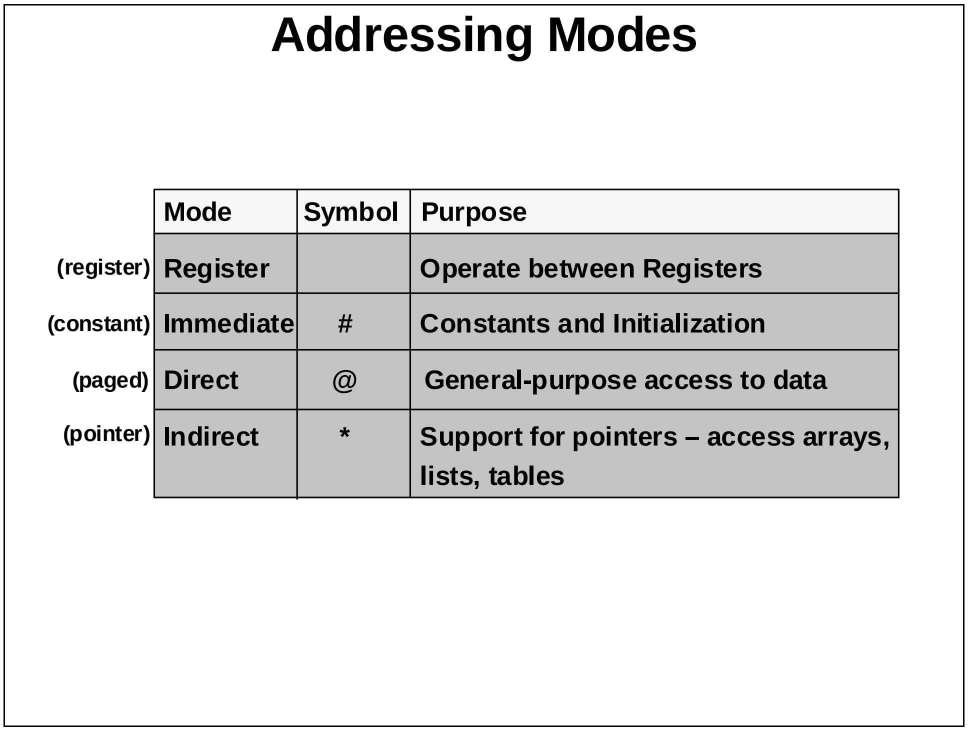
label* .usect *"name",5

used for variables

```
          .ref      start
          .sect     "vectors"
          ;make reset vector address 'start'
reset:    .long     start
```

```
          .def      start
count     .set      9
          ; create an array x of 10 words
x         .usect    "mydata", 10
-------------------------------------------------
          .sect     "code"
start:    C28OBJ    ;operate in C28x mode
          MOV       ACC,#1
next:     MOVL      XAR1,#x
          MOV       AR2,#count
loop:     MOV       *XAR1++,AL
          BANZ      loop,AR2--
bump:     ADD       ACC,#1
          SB        next,UNC
```

Addressing Modes

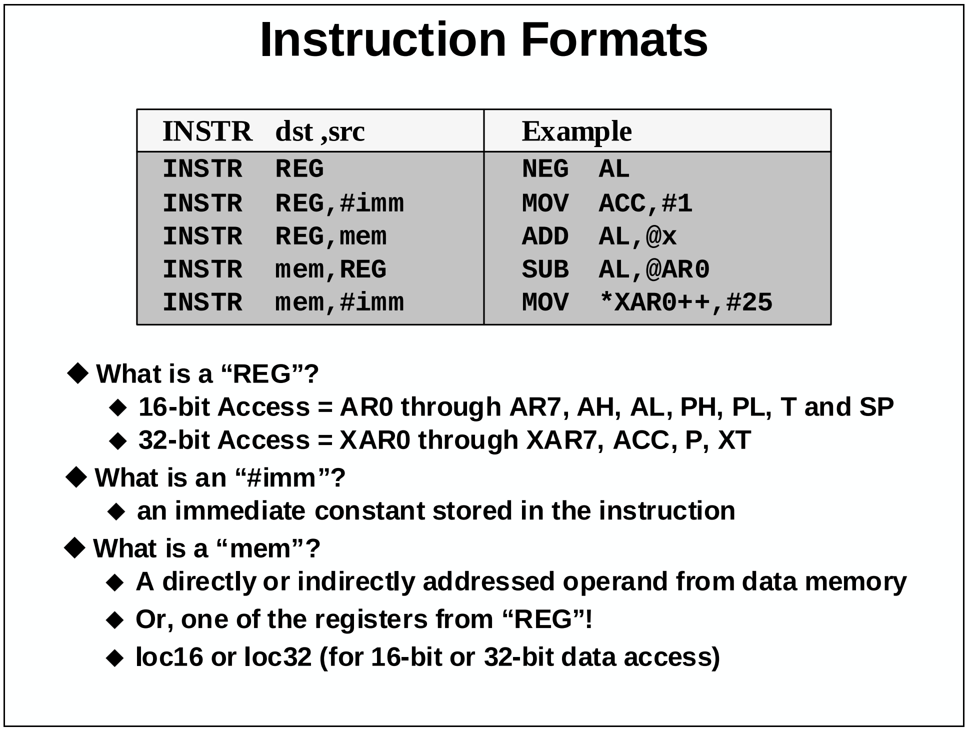


Addressing Modes

	Mode	Symbol	Purpose
(register)	Register		Operate between Registers
(constant)	Immediate	#	Constants and Initialization
(paged)	Direct	@	General-purpose access to data
(pointer)	Indirect	*	Support for pointers – access arrays, lists, tables

Four main categories of addressing modes are available on the C28x. Register addressing mode allows interchange between all CPU registers, convenient for solving intricate equations. Immediate addressing is helpful for expressing constants easily. Direct addressing mode allows information in memory to be accessed. Indirect addressing allows pointer support via dedicated 'auxiliary registers', and includes the ability to index, or increment through a structure. The C28x supports a true software stack, desirable for supporting the needs of the C language and other structured programming environments, and presents a stack-relative addressing mode for efficiently accessing elements from the stack. Paged direct addressing offers general-purpose single cycle memory access, but restricts the user to working in any single desired block of memory at one time.

Instruction Formats



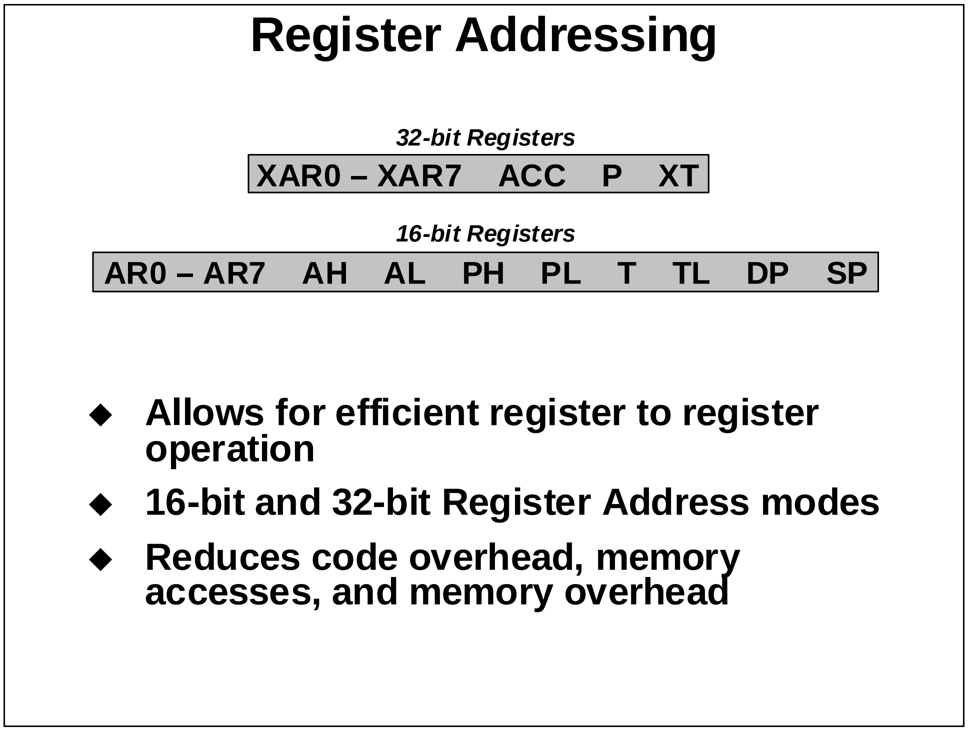
Instruction Formats

INSTR dst ,src	Example
`INSTR  REG`	`NEG  AL`
`INSTR  REG,#imm`	`MOV  ACC,#1`
`INSTR  REG,mem`	`ADD  AL,@x`
`INSTR  mem,REG`	`SUB  AL,@AR0`
`INSTR  mem,#imm`	`MOV  *XAR0++,#25`

- **What is a “REG”?**
  - **16-bit Access = AR0 through AR7, AH, AL, PH, PL, T and SP**
  - **32-bit Access = XAR0 through XAR7, ACC, P, XT**
- **What is an “#imm”?**
  - **an immediate constant stored in the instruction**
- **What is a “mem”?**
  - **A directly or indirectly addressed operand from data memory**
  - **Or, one of the registers from “REG”!**
  - **loc16 or loc32 (for 16-bit or 32-bit data access)**

The C28x follows a convention that uses instruction, destination, then source operand order (INSTR dst, src). Several general formats exist to allow modification of memory or registers based on constants, memory, or register inputs. Different modes are identifiable by their leading characters (# for immediate, * for indirect, and @ for direct). Note that registers or data memory can be selected as a ‘mem’ value.

Register Addressing



Register Addressing

32-bit Registers

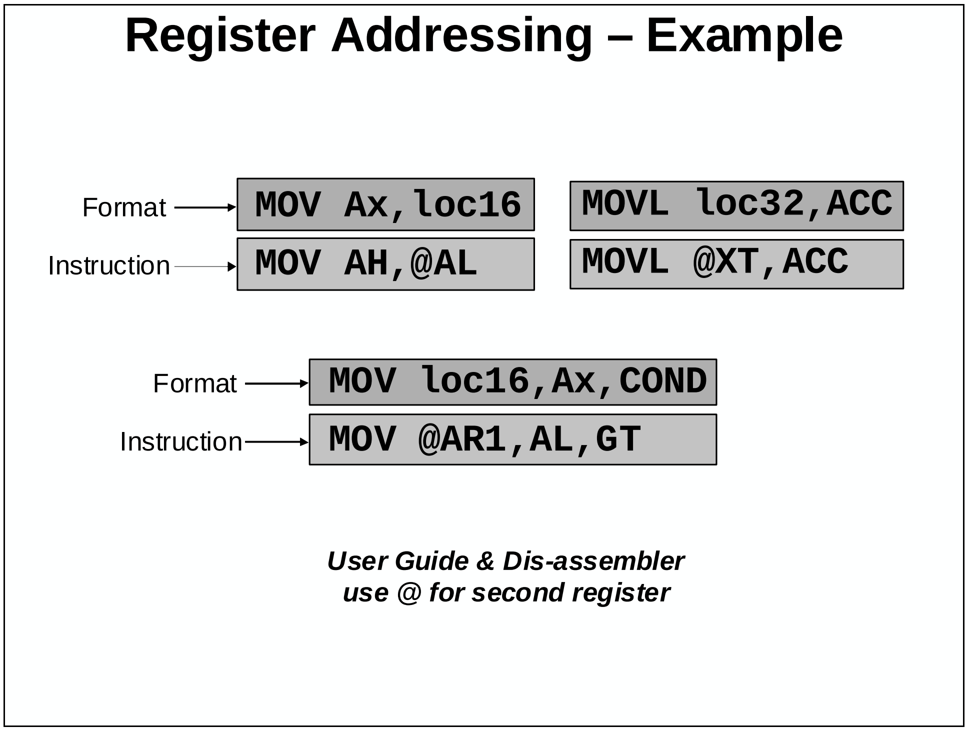
XAR0 – XAR7 ACC P XT

16-bit Registers

AR0 – AR7 AH AL PH PL T TL DP SP

- **Allows for efficient register to register operation**
- **16-bit and 32-bit Register Address modes**
- **Reduces code overhead, memory accesses, and memory overhead**

Register addressing allows the exchange of values between registers, and with certain instructions can be used in conjunction with other addressing modes, yielding a more efficient instruction set. Remember that any 'mem' field allows the use of a register as the operand, and that no special character (such as @, *, or #) need be used to specify the register mode.



Register Addressing – Example

Format → `MOV Ax,loc16` `MOVL loc32,ACC`

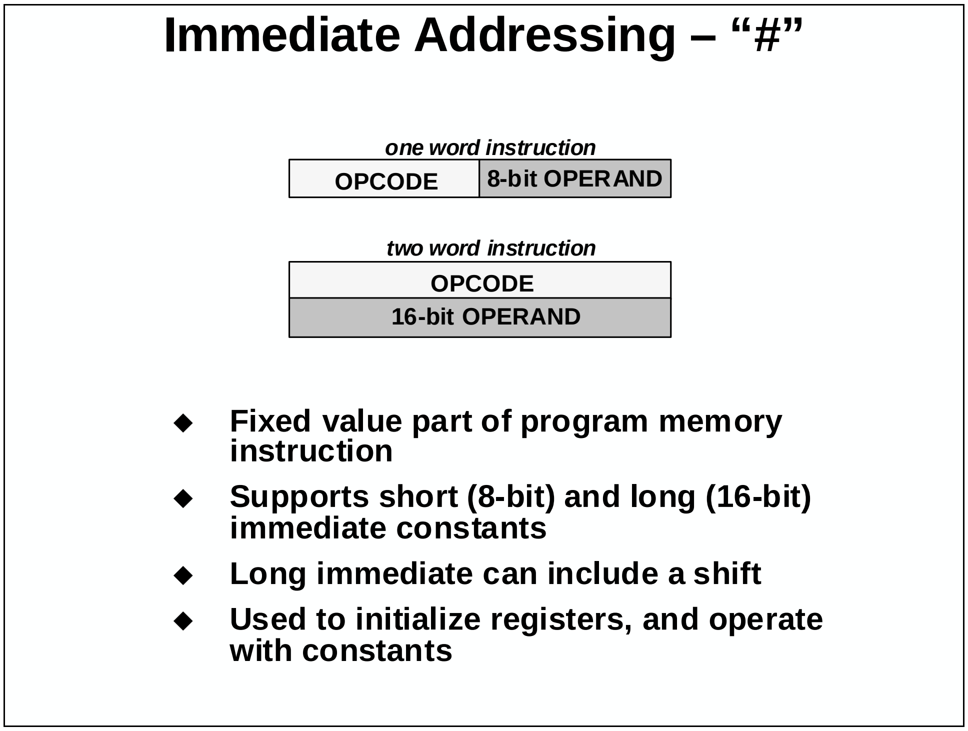
Instruction → `MOV AH,@AL` `MOVL @XT,ACC`

Format → `MOV loc16,Ax,COND`

Instruction → `MOV @AR1,AL,GT`

User Guide & Dis-assembler use @ for second register

Immediate Addressing



Immediate Addressing – “#”

one word instruction

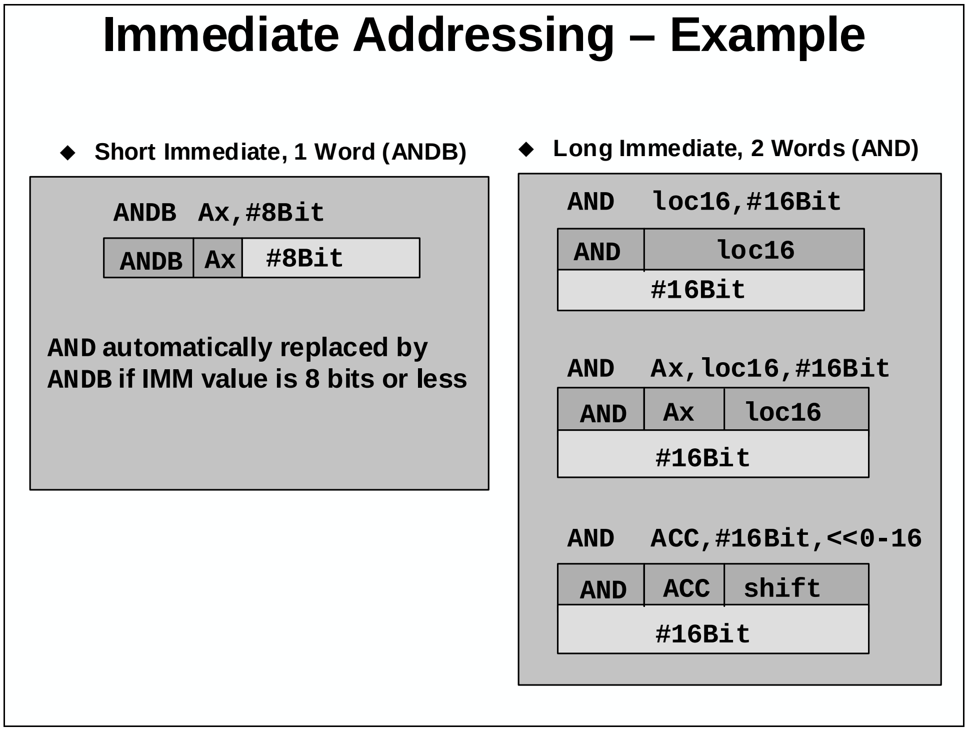
OPCODE	8-bit OPERAND

two word instruction

OPCODE
16-bit OPERAND

- Fixed value part of program memory instruction
- Supports short (8-bit) and long (16-bit) immediate constants
- Long immediate can include a shift
- Used to initialize registers, and operate with constants

Immediate addressing allows the user to specify a constant within an instruction mnemonic. Short immediate are single word, and execute in a single cycle. Long (16-bit) immediate allow full sized values, which become two-word instructions - yet execute in a single instruction cycle.



Immediate Addressing – Example

- Short Immediate, 1 Word (ANDB)

```
ANDB  Ax,#8Bit
```

ANDB	Ax	#8Bit

AND automatically replaced by **ANDB** if IMM value is 8 bits or less

- Long Immediate, 2 Words (AND)

```
AND   loc16,#16Bit
```

AND	loc16
#16Bit	

```
AND   Ax,loc16,#16Bit
```

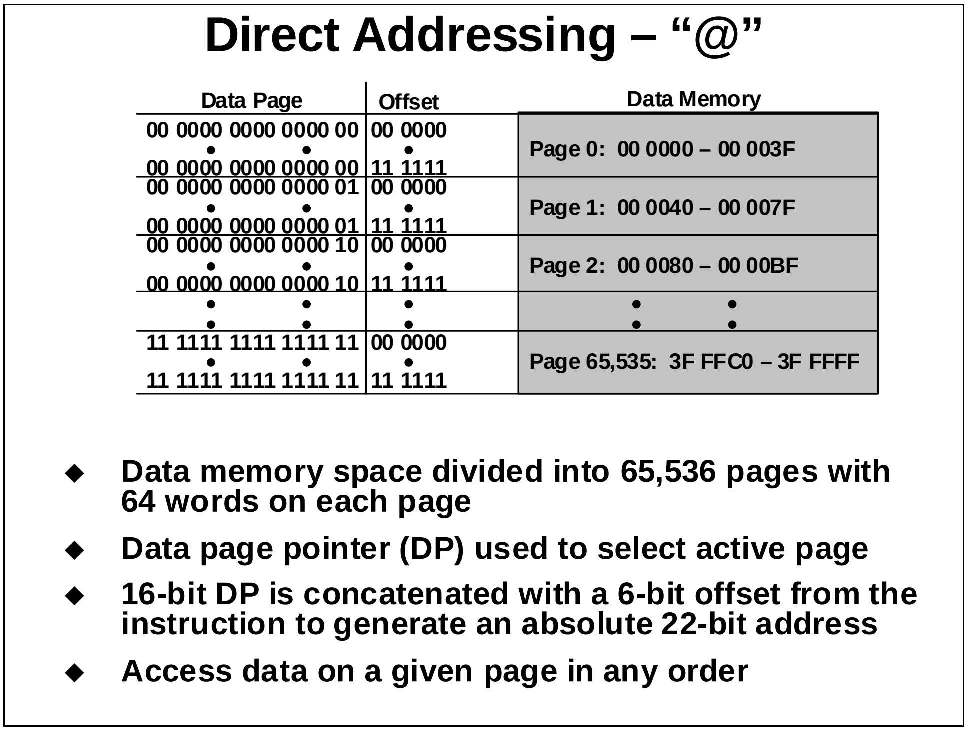
AND	Ax	loc16
#16Bit		

```
AND   ACC,#16Bit,<<0-16
```

AND	ACC	shift
#16Bit		

Direct Addressing

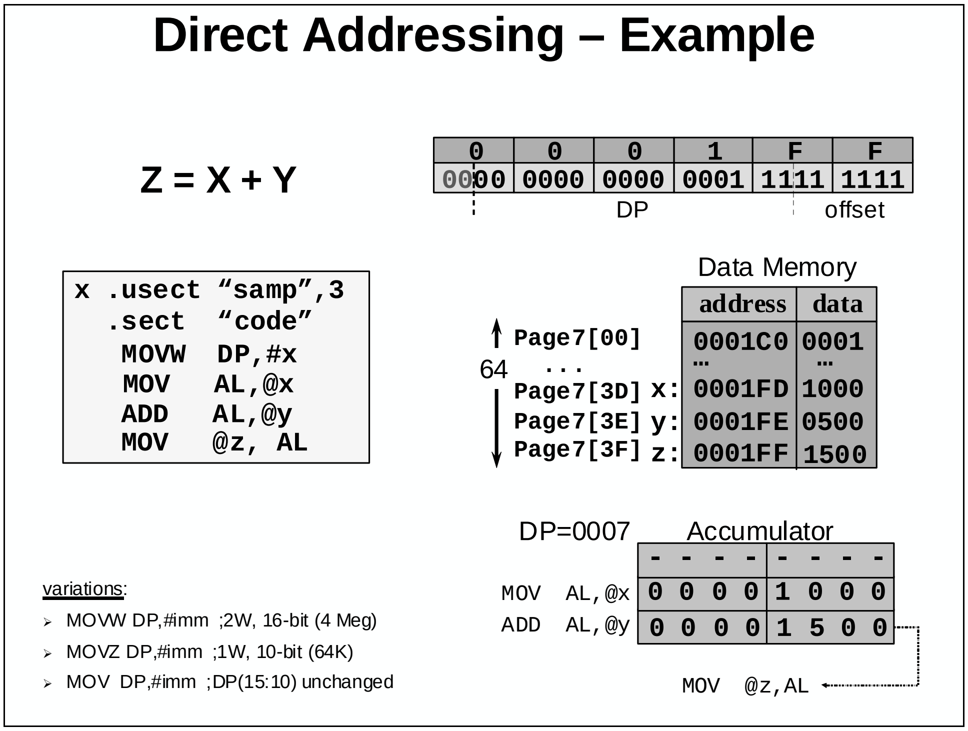
Direct addressing allows for access to the full 4-Meg words space in 64 word “page” groups. As such, a 16-bit Data Page register is used to extend the 6-bit local address in the instruction word. Programmers should note that poor DP management is a key source of programming errors. Paged direct addressing is fast and reliable if the above considerations are followed. The watch operation, recommended for use whenever debugging, extracts the data page and displays it as the base address currently in use for direct addressing.



Direct Addressing – “@”

Data Page	Offset	Data Memory
00 0000 0000 0000 00 • • 00 0000 0000 0000 00	00 0000 • 11 1111	Page 0: 00 0000 – 00 003F
00 0000 0000 0000 01 • • 00 0000 0000 0000 01	00 0000 • 11 1111	Page 1: 00 0040 – 00 007F
00 0000 0000 0000 10 • • 00 0000 0000 0000 10	00 0000 • 11 1111	Page 2: 00 0080 – 00 00BF
• • • •	• •	• • • •
11 1111 1111 1111 11 • • 11 1111 1111 1111 11	00 0000 • 11 1111	Page 65,535: 3F FFC0 – 3F FFFF

- ◆ Data memory space divided into 65,536 pages with 64 words on each page
- ◆ Data page pointer (DP) used to select active page
- ◆ 16-bit DP is concatenated with a 6-bit offset from the instruction to generate an absolute 22-bit address
- ◆ Access data on a given page in any order



Direct Addressing – Example

Z = X + Y

0	0	0	1	F	F
0000	0000	0000	0001	1111	1111

DP offset

```
x .usect "samp",3
  .sect  "code"
   MOVW  DP,#x
   MOV   AL,@x
   ADD   AL,@y
   MOV   @z, AL
```

Data Memory

		address	data
Page7[00]		0001C0	0001
...		…	…
Page7[3D]	x:	0001FD	1000
Page7[3E]	y:	0001FE	0500
Page7[3F]	z:	0001FF	1500

64

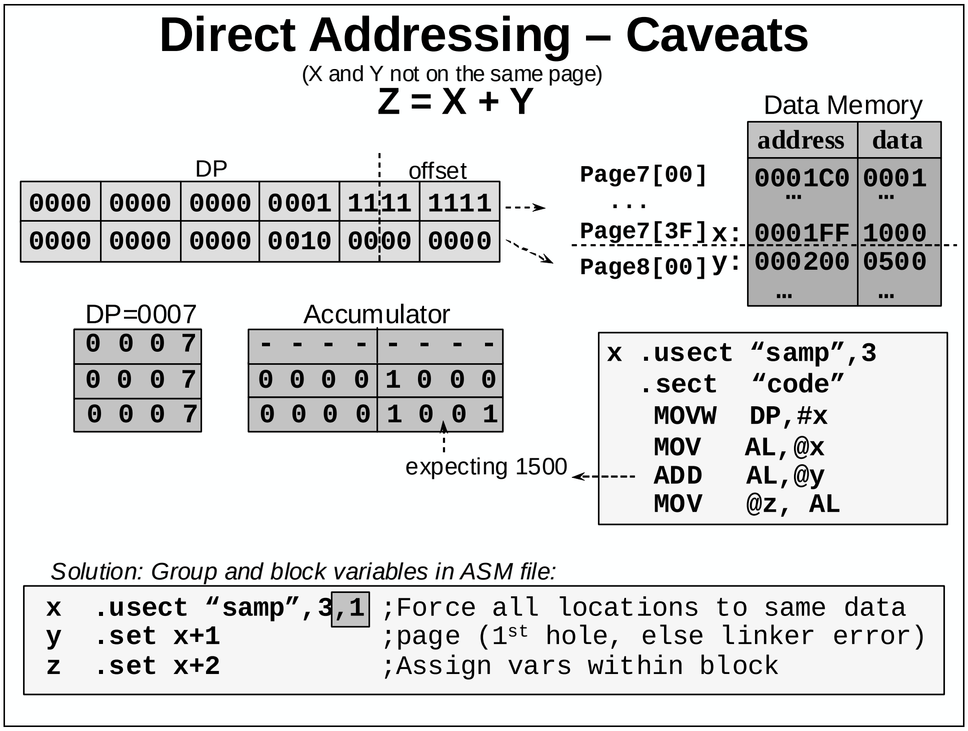
DP=0007 Accumulator

	Accumulator
	- - - - - - - -
MOV AL,@x	0 0 0 0 1 0 0 0
ADD AL,@y	0 0 0 0 1 5 0 0

MOV @z,AL

variations:

- MOVW DP,#imm ;2W, 16-bit (4 Meg)
- MOVZ DP,#imm ;1W, 10-bit (64K)
- MOV DP,#imm ;DP(15:10) unchanged



Direct Addressing – Caveats

(X and Y not on the same page)

Z = X + Y

DP offset

0000	0000	0000	0001	1111	1111
0000	0000	0000	0010	0000	0000

Data Memory

		address	data
Page7[00]		0001C0	0001
...		…	…
Page7[3F]	x:	0001FF	1000
Page8[00]	y:	000200	0500
		…	…

DP=0007	Accumulator
0 0 0 7	- - - - - - - -
0 0 0 7	0 0 0 0 1 0 0 0
0 0 0 7	0 0 0 0 1 0 0 1

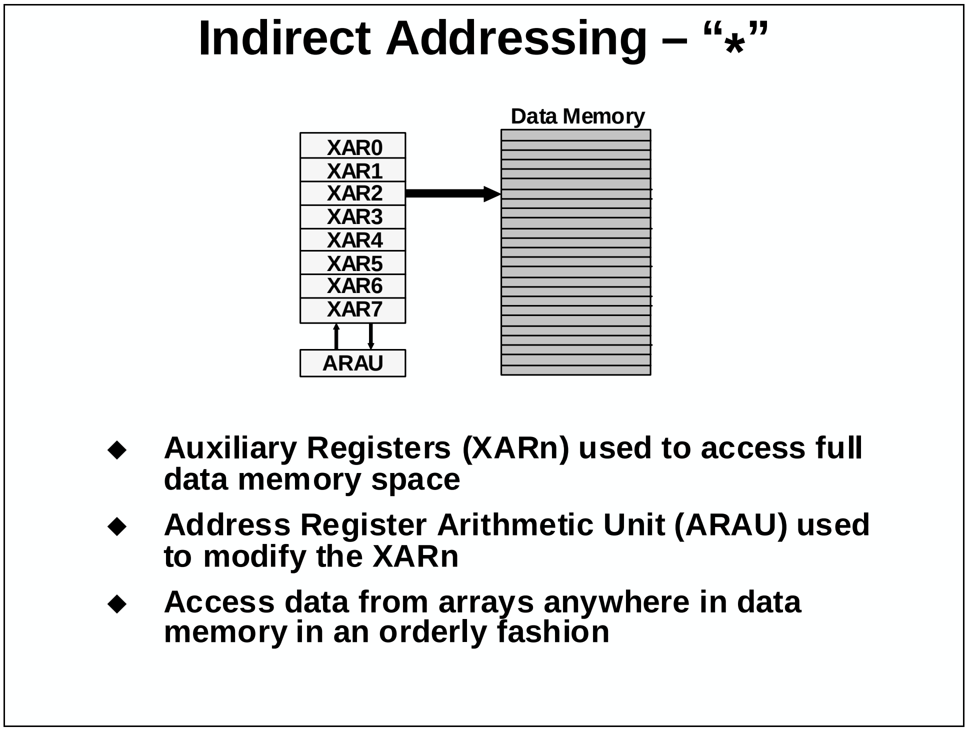
expecting 1500

```
x .usect "samp",3
  .sect  "code"
   MOVW  DP,#x
   MOV   AL,@x
   ADD   AL,@y
   MOV   @z, AL
```

Solution: Group and block variables in ASM file:

```
x   .usect "samp",3,1  ;Force all locations to same data
y   .set x+1           ;page (1st hole, else linker error)
z   .set x+2           ;Assign vars within block
```

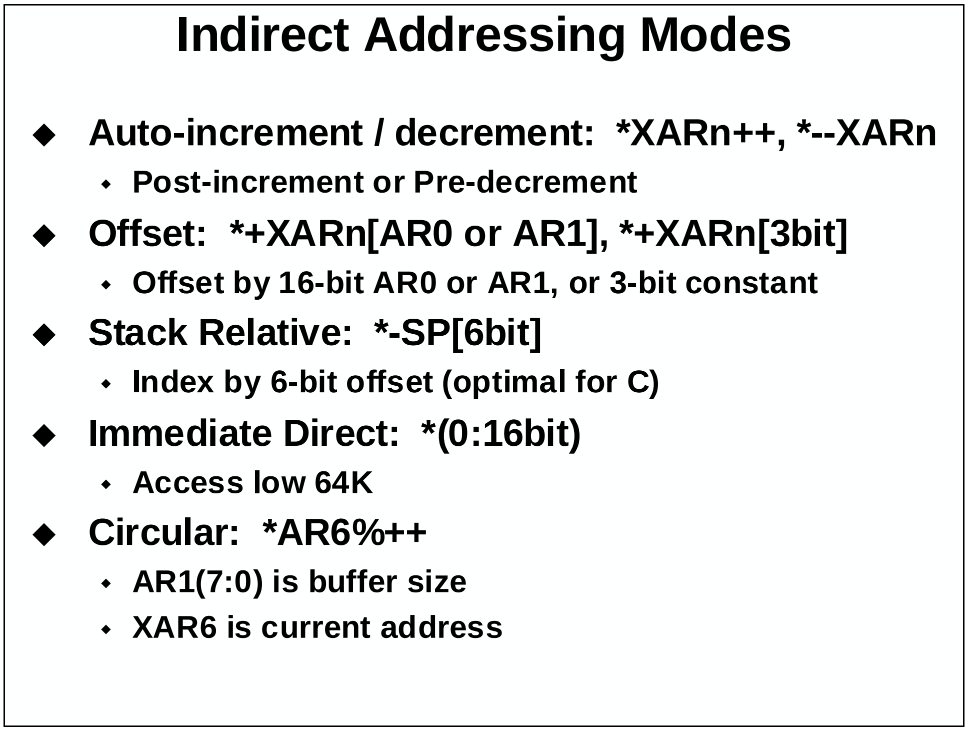
Indirect Addressing



Indirect Addressing – "*"

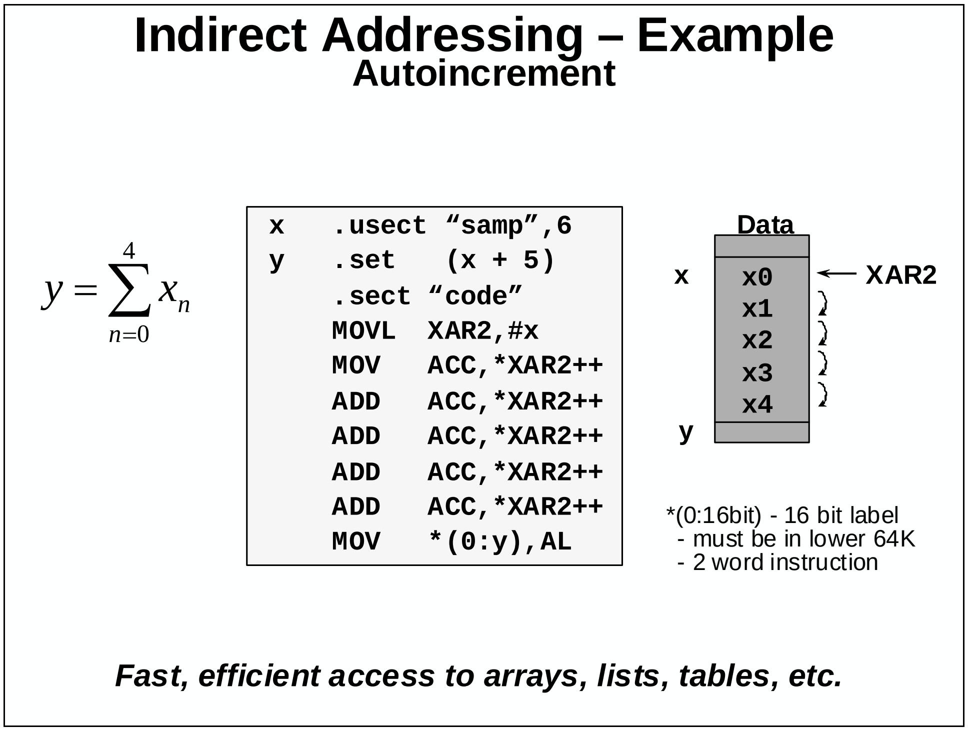
- **Auxiliary Registers (XARn) used to access full data memory space**
- **Address Register Arithmetic Unit (ARAU) used to modify the XARn**
- **Access data from arrays anywhere in data memory in an orderly fashion**

Any of eight hardware pointers (ARs) may be employed to access values from the first 64K of data memory. Auto-increment or decrement is supported at no additional cycle cost. XAR register formats offer larger 32-bit widths, allowing them to access across the full 4-Giga words data space.



Indirect Addressing Modes

- **Auto-increment / decrement: *XARn++, *--XARn**
  - **Post-increment or Pre-decrement**
- **Offset: *+XARn[AR0 or AR1], *+XARn[3bit]**
  - **Offset by 16-bit AR0 or AR1, or 3-bit constant**
- **Stack Relative: *-SP[6bit]**
  - **Index by 6-bit offset (optimal for C)**
- **Immediate Direct: *(0:16bit)**
  - **Access low 64K**
- **Circular: *AR6%++**
  - **AR1(7:0) is buffer size**
  - **XAR6 is current address**



Indirect Addressing – Example
Autoincrement

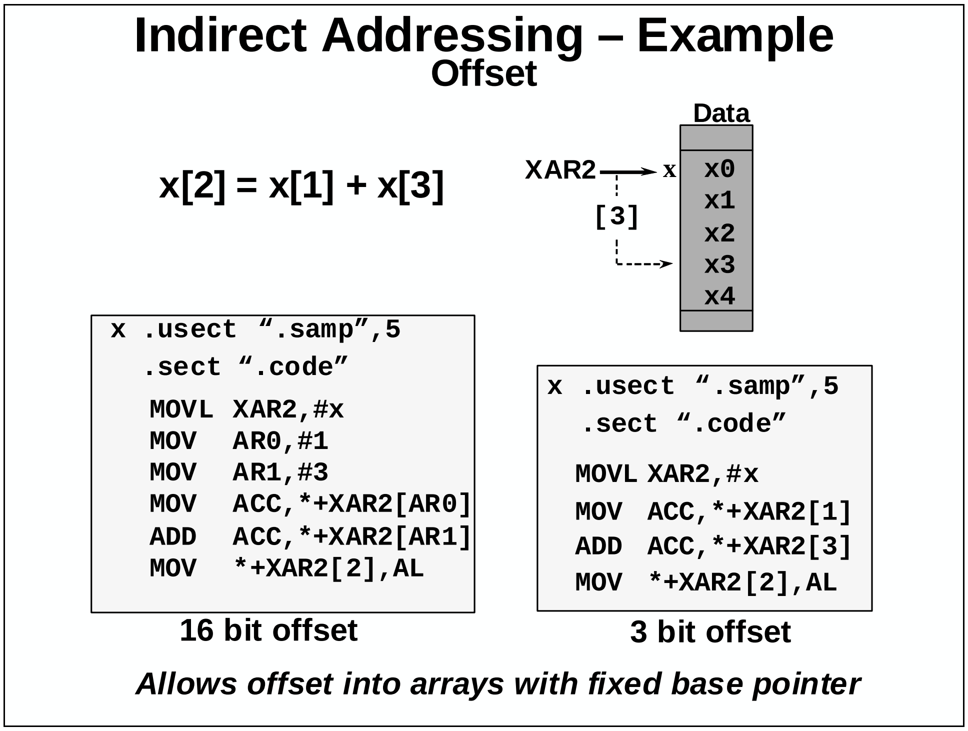
$$y = \sum_{n=0}^{4} x_n$$

```
x   .usect “samp”,6
y   .set   (x + 5)
    .sect “code”
    MOVL  XAR2,#x
    MOV   ACC,*XAR2++
    ADD   ACC,*XAR2++
    ADD   ACC,*XAR2++
    ADD   ACC,*XAR2++
    ADD   ACC,*XAR2++
    MOV   *(0:y),AL
```

*(0:16bit) - 16 bit label
- must be in lower 64K
- 2 word instruction

Fast, efficient access to arrays, lists, tables, etc.

Indexed addressing offers the ability to select operands from within an array without modification to the base pointer. Stack-based operations are handled with a 16-bit Stack Pointer register, which operates over the base 64K of data memory. It offers 6-bit non-destructive indexing to access larger stack-based arrays efficiently.



Indirect Addressing – Example
Offset

x[2] = x[1] + x[3]

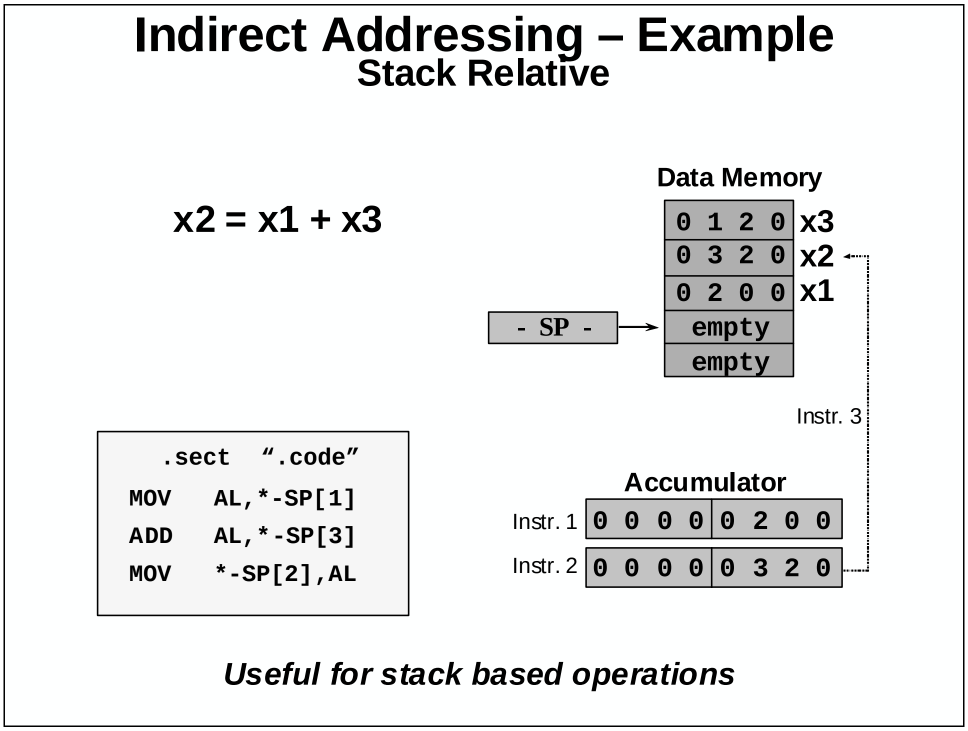
```
x .usect “.samp”,5
  .sect “.code”
   MOVL  XAR2,#x
   MOV   AR0,#1
   MOV   AR1,#3
   MOV   ACC,*+XAR2[AR0]
   ADD   ACC,*+XAR2[AR1]
   MOV   *+XAR2[2],AL
```

16 bit offset

```
x .usect “.samp”,5
  .sect “.code”
  MOVL XAR2,#x
  MOV  ACC,*+XAR2[1]
  ADD  ACC,*+XAR2[3]
  MOV  *+XAR2[2],AL
```

3 bit offset

Allows offset into arrays with fixed base pointer



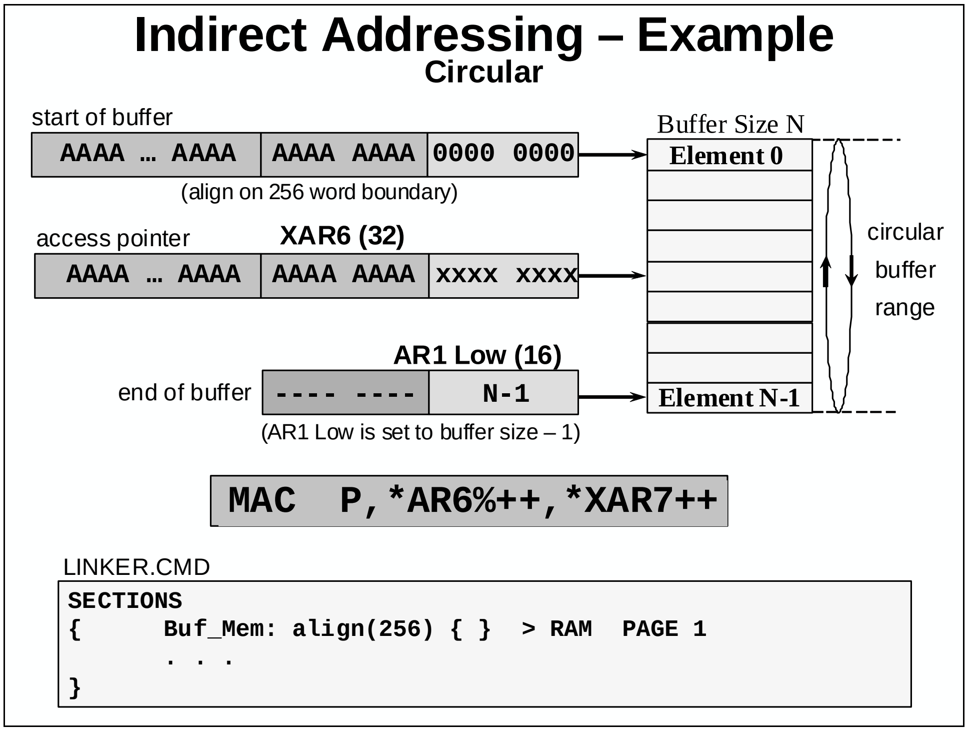
Indirect Addressing – Example

Stack Relative

x2 = x1 + x3

```
     .sect   ".code"
MOV    AL,*-SP[1]
ADD    AL,*-SP[3]
MOV    *-SP[2],AL
```

Useful for stack based operations



Indirect Addressing – Example

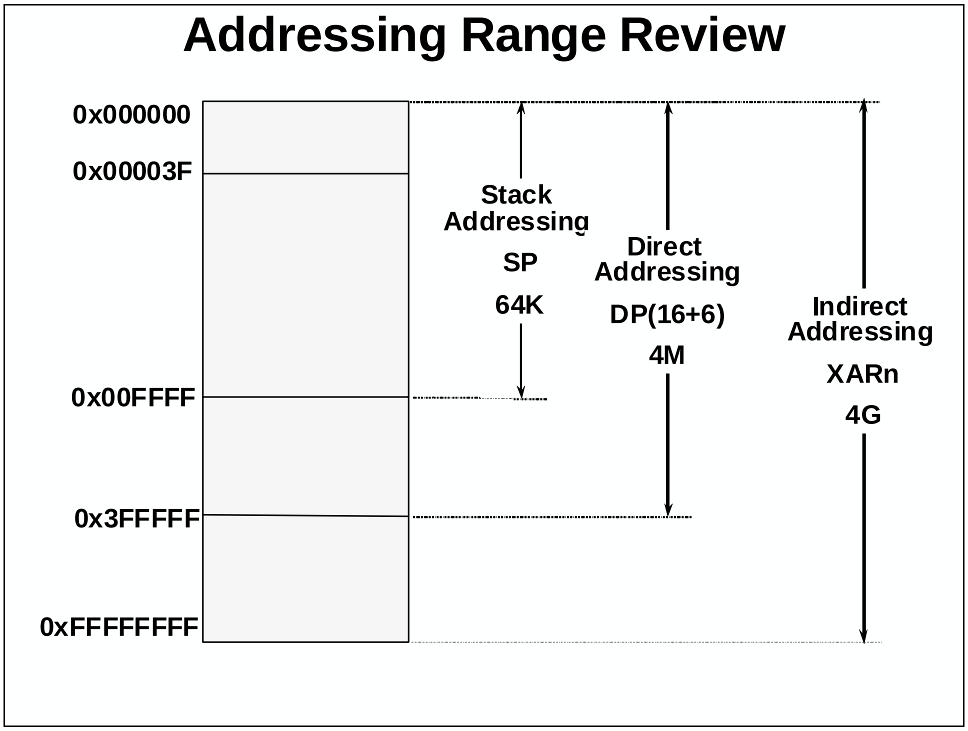
Circular

```
MAC  P,*AR6%++,*XAR7++
```

LINKER.CMD

```
SECTIONS
{      Buf_Mem: align(256) { }  > RAM  PAGE 1
       . . .
}
```

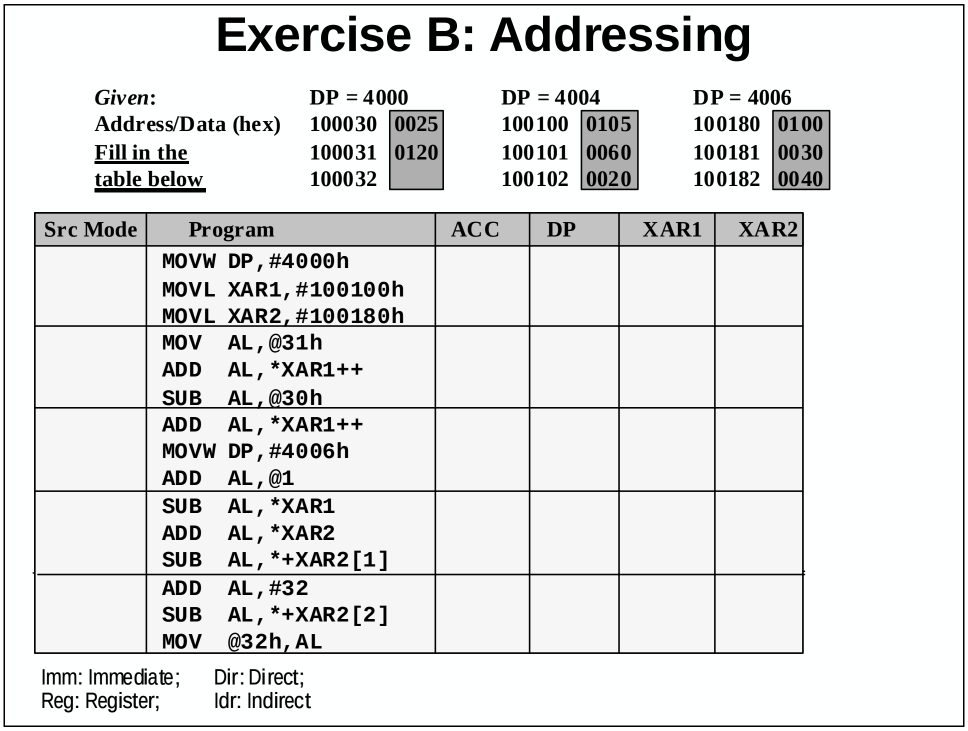
Review


Addressing Range Review

Data memory can be accessed in numerous ways:

- Stack Addressing: allows a range to 64K
- Direct Addressing: Offers a 16-bit DP plus a 6-bit offset, allowing a 4M range
- Indirect Addressing: Offers the full 4G range

Exercise B



Exercise B: Addressing

Given:	DP = 4000		DP = 4004		DP = 4006	
Address/Data (hex)	100030	0025	100100	0105	100180	0100
Fill in the	100031	0120	100101	0060	100181	0030
table below	100032		100102	0020	100182	0040

Src Mode	Program	ACC	DP	XAR1	XAR2
	MOVW DP,#4000h MOVL XAR1,#100100h MOVL XAR2,#100180h				
	MOV AL,@31h ADD AL,*XAR1++ SUB AL,@30h				
	ADD AL,*XAR1++ MOVW DP,#4006h ADD AL,@1				
	SUB AL,*XAR1 ADD AL,*XAR2 SUB AL,*+XAR2[1]				
	ADD AL,#32 SUB AL,*+XAR2[2] MOV @32h,AL				

Imm: Immediate; Dir: Direct;
Reg: Register; Idr: Indirect

In the table above, fill in the values for each of the registers for each of the instructions. Three areas of data memory are displayed at the top of the diagram, showing both their addresses and contents in hexadecimal. Watch out for surprises along the way. First, you should answer the addressing mode for the source operand. Then, fill in the change values as the result of the instruction operation.

Lab B: Addressing

Note: *The lab linker command file is based on the F28035 memory map – modify as needed, if using a different F28xx device memory map.*

- **Objective**

The objective of this lab is to practice and verify the mechanics of addressing. In this process we will expand upon the ASM file from the previous lab to include new functions. Additionally, we learn how to run and observe the operation of code using Code Composer Studio.

In this lab, we will initialize the "vars" arrays allocated in the previous lab with the contents of the "const" table. How is this best accomplished? Consider the process of loading the first "const" value into the accumulator and then storing this value to the first "vars" location, and repeating this process for each of the succeeding values.

- What forms of addressing could be used for this purpose?
- Which addressing mode would be best in this case? Why?
- What problems could arise with using another mode?

- **Procedure**

Copy Files, Create Project File

1. Create a new project called `LabB.pjt` in `C:\C28x\Labs\Appendix\LabB` and add `LabB.asm` and `Lab.cmd` to it. Check your file list to make sure all the files are there. Be sure to setup the Build Options by clicking: `Project → Build Options` on the menu bar. Select the Linker tab. In the middle of the screen select "`No Autoinitialization`" under "`Autoinit Model:`". Enter `start` in the "`Code Entry Point (-e):`" field. Next, select the Compiler tab. Note that "`Full Symbolic Debug (-g)`" under "`Generate Debug Info:`" is selected. Then select `OK` to save the Build Options.

Initialize Allocated RAM Array from ROM Initialization Table

2. Edit `LabB.asm` and modify it to copy *table[9]* to *data[9]* using indirect addressing. (Note: *data[9]* consists of the allocated arrays of *data*, *coeff*, and *result*). Initialize the allocated RAM array from the ROM initialization table:

   - Delete the NOP operations from the "code" section.
   - Initialize pointers to the beginning of the "const" and "vars" arrays.
   - Transfer the first value from "const" to the "vars" array.
   - Repeat the process for all values to be initialized.

   To perform the copy, consider using a load/store method via the accumulator. Which part of an accumulator (low or high) should be used? Use the following when writing your copy routine:

   - use AR1 to hold the address of *table*
   - use AR2 to hold the address of *data*

3. It is good practice to trap the end of the program (i.e. use either `“end:  B end,UNC”` or `“end:  B  start,UNC”`). Save your work.

Build and Load

4. Click the `“Build”` button and watch the tools run in the build window. Debug as necessary. To open up more space, close any open files or windows that you do not need.

5. Load the output file onto the target. Click:

   `File → Load Program…`

   If you wish, right click on the `LabB.asm` source window and select `Mixed Mode` to debug using both source and assembly.

Note: Code Composer Studio can automatically load the output file after a successful build. On the menu bar click: `Option → Customize…` and select the `“Program Load Options”` tab, `check “Load Program After Build”`, then click `OK`.

6. Single-step your routine. While single-stepping, it is helpful to see the values located in *table[9]* and *data[9]* at the same time. Open two memory windows by using the `“View Memory”` button on the vertical toolbar and using the address labels *table* and *data*. Setting the properties filed to “Hex 16 Bit – TI style” will give you more viewable data in the window. Additionally, it is useful to watch the CPU registers. Open the CPU registers by using the `“View → Registers → CPU Registers”`. Deselect `“Allow Docking”` and move/resize the window as needed. Check to see if the program is working as expected.

End of Exercise

OPTIONAL Lab B-C: Array Initialization in C

Note: *The lab linker command file is based on the F28035 memory map – modify as needed, if using a different F28xx device memory map.*

- **Objective**

The objective of this lab is to practice and verify the mechanics of initialization using C. Additionally, we learn how to run and observe the operation of C code using Code Composer Studio. In this lab, we will initialize the `“vars”` arrays with the contents of the `“const”` table.

- **Procedure**

Create Project File

1. In Code Composer Studio create a new project called `LabB-C.pjt` in `C:\C28x\Labs\Appendix\LabB\LabB-C` and add `LabB-C.c` and `Lab.cmd` to it. Check your file list to make sure all the files are there. Open the Build Options and select the Linker tab. Select the “Libraries” Category and enter `rts2800_ml.lib` in the `“Incl. Libraries (-l):”` box. Do not setup any other Build Options. The default values will be used. In Appendix Lab D exercise, we will experiment and explore the various build options when working with C.

Initialize Allocated RAM Array from ROM Initialization Table

2. Edit `LabB-C.c` and modify the “main” routine to copy *table[9]* to the allocated arrays of *data[4]*, *coeff[4]*, and *result[1]*. (Note: *data[9]* consists of the allocated arrays of *data*, *coeff*, and *result*).

Build and Load

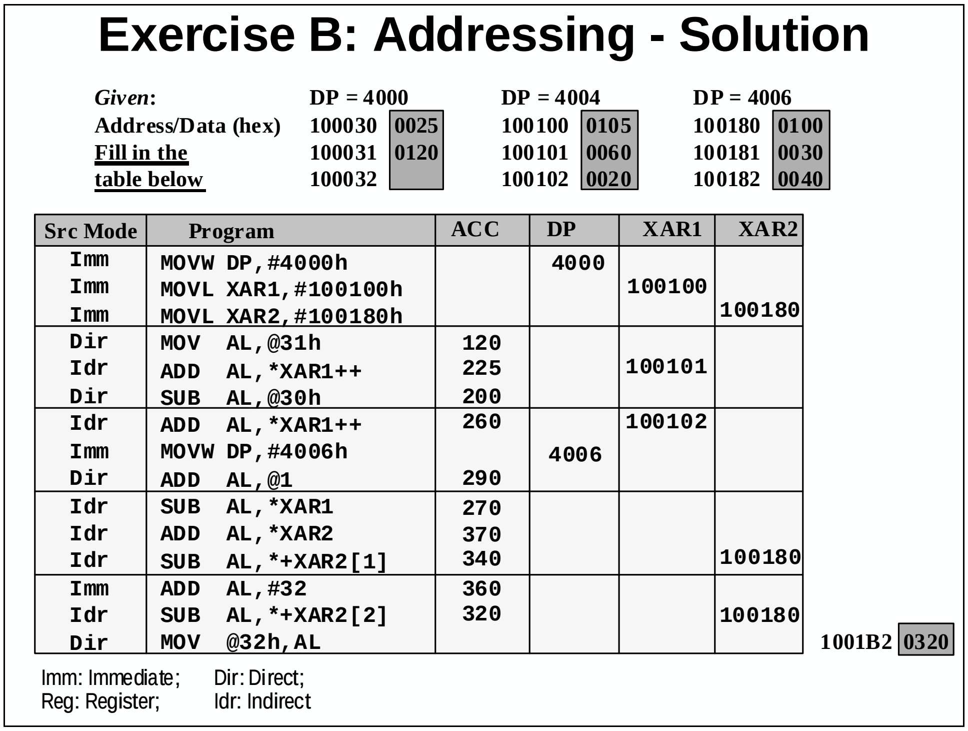
3. Click the `“Build”` button and watch the tools run in the build window. Debug as necessary.

Note: Have Code Composer Studio automatically load the output file after a successful build. On the menu bar click: `Option` → `Customize…` and select the `“Program Load Options”` tab, `check “Load Program After Build”`, then click `OK`.

4. Under `Debug` on the menu bar click `“Go Main”`. Single-step your routine. While single-stepping, it is helpful to see the values located in *table[9]* and *data[9]* at the same time. Open two memory windows by using the `“View Memory”` button on the vertical toolbar and using the address labels *table* and *data*. Setting the properties field to “Hex 16 Bit – TI style” will give you more viewable data in the window. Additionally, you can watch the CPU registers. Open the CPU registers by using the `“View` → `Registers` → `CPU Registers`. Deselect `“Allow Docking”` and move/resize the window as needed. Check to see if the program is working as expected.

End of Exercise

Solutions



Exercise B: Addressing - Solution

Given:	DP = 4000		DP = 4004		DP = 4006	
Address/Data (hex)	100030	0025	100100	0105	100180	0100
Fill in the	100031	0120	100101	0060	100181	0030
table below	100032		100102	0020	100182	0040

Src Mode	Program	ACC	DP	XAR1	XAR2
Imm	`MOVW DP,#4000h`		4000		
Imm	`MOVL XAR1,#100100h`			100100	
Imm	`MOVL XAR2,#100180h`				100180
Dir	`MOV  AL,@31h`	120			
Idr	`ADD  AL,*XAR1++`	225		100101	
Dir	`SUB  AL,@30h`	200			
Idr	`ADD  AL,*XAR1++`	260		100102	
Imm	`MOVW DP,#4006h`		4006		
Dir	`ADD  AL,@1`	290			
Idr	`SUB  AL,*XAR1`	270			
Idr	`ADD  AL,*XAR2`	370			
Idr	`SUB  AL,*+XAR2[1]`	340			100180
Imm	`ADD  AL,#32`	360			
Idr	`SUB  AL,*+XAR2[2]`	320			100180
Dir	`MOV  @32h,AL`				

1001B2 0320

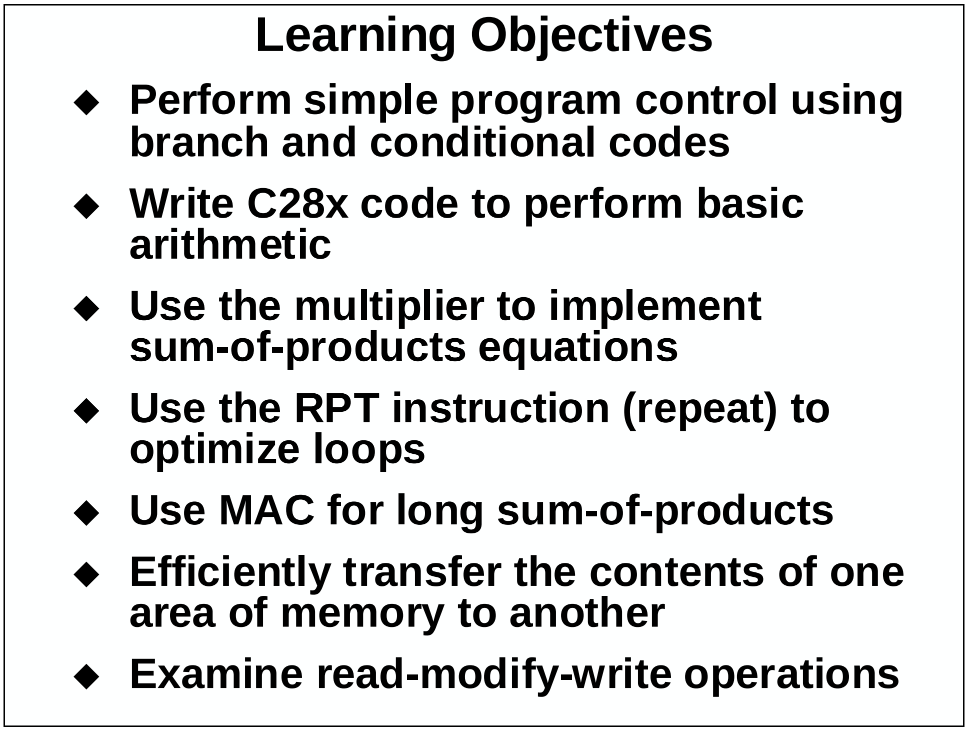
Imm: Immediate; Dir: Direct;
Reg: Register; Idr: Indirect

Appendix C – Assembly Programming

Introduction

Appendix C discusses the details of programming in assembly. It shows you how to use different instructions that further utilize the advantage of the architecture data paths. It gives you the ability to analyze the instruction set and pick the best instruction for the application.

Learning Objectives



Learning Objectives

- Perform simple program control using branch and conditional codes
- Write C28x code to perform basic arithmetic
- Use the multiplier to implement sum-of-products equations
- Use the RPT instruction (repeat) to optimize loops
- Use MAC for long sum-of-products
- Efficiently transfer the contents of one area of memory to another
- Examine read-modify-write operations

Module Topics

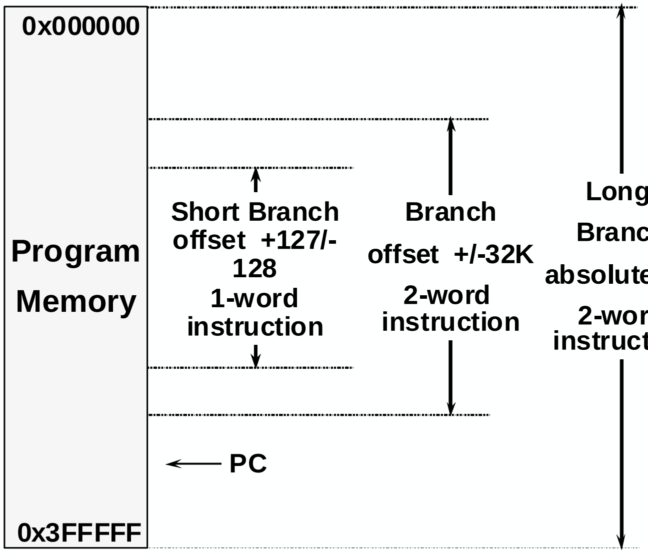
Program Control

The program control logic and program address generation logic work together to provide proper program flow. Normally, the flow of a program is sequential: the CPU executes instructions at consecutive program memory addresses. At times, a discontinuity is required; that is, a program must branch to a nonsequential address and then execute instructions sequentially at that new location. For this purpose, the C28x supports interrupts, branches, calls, returns, and repeats. Proper program flow also requires smooth flow at the instruction level. To meet this need, the C28x has a protected pipeline and an instruction-fetch mechanism that attempts to keep the pipeline full.

Branches

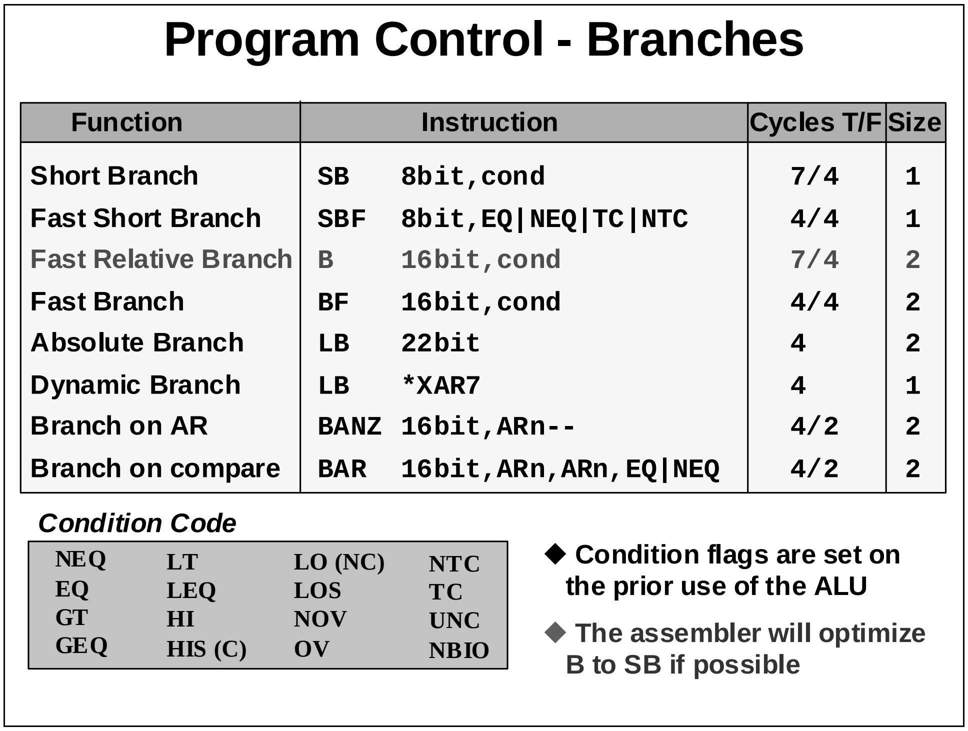
Branch Types and Range

- **3 Branch Types**



The PC can access the entire 4M words (8M bytes) range. Some branching operations offer 8- and 16-bit relative jumps, while long branches, calls, and returns provide a full 22-bit absolute address. Dynamic branching allows a run-time calculated destination. The C28x provides the familiar arithmetic results status bits (Zero, oVerflow, Negative, Carry) plus a Test Control bit which holds the result of a binary test. The states of these bits in various combinations allow a range of signed, unsigned, and binary branching conditions offered.

Program Control Instructions



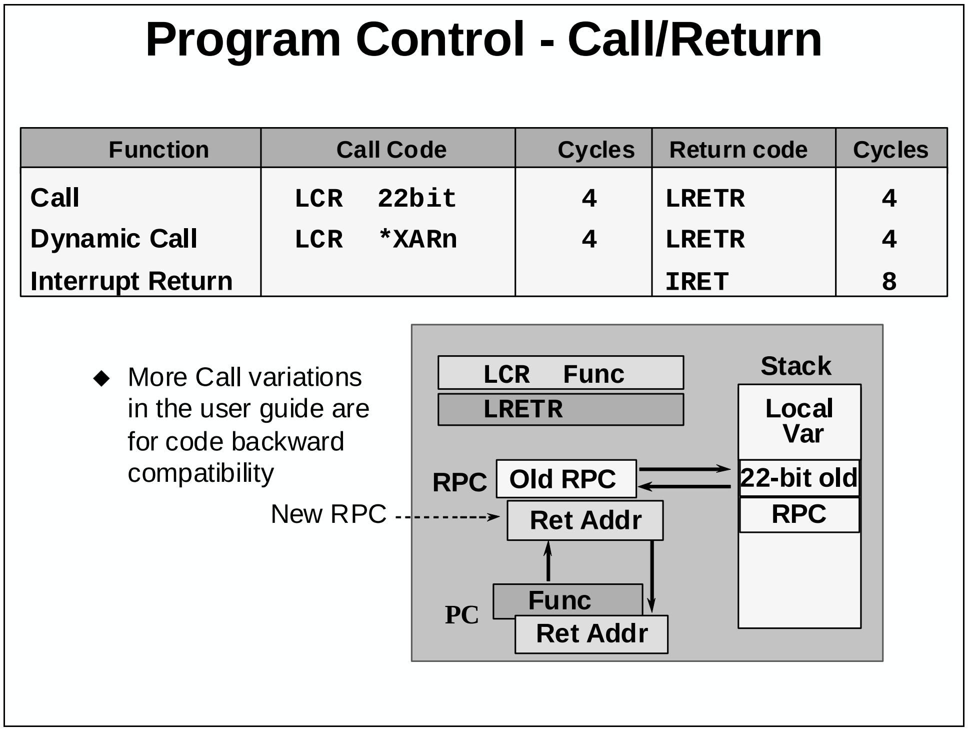
Program Control - Branches

Function	Instruction	Cycles T/F	Size
Short Branch	`SB   8bit,cond`	7/4	1
Fast Short Branch	`SBF  8bit,EQ|NEQ|TC|NTC`	4/4	1
Fast Relative Branch	`B    16bit,cond`	7/4	2
Fast Branch	`BF   16bit,cond`	4/4	2
Absolute Branch	`LB   22bit`	4	2
Dynamic Branch	`LB   *XAR7`	4	1
Branch on AR	`BANZ 16bit,ARn--`	4/2	2
Branch on compare	`BAR  16bit,ARn,ARn,EQ|NEQ`	4/2	2

Condition Code

NEQ	LT	LO (NC)	NTC
EQ	LEQ	LOS	TC
GT	HI	NOV	UNC
GEQ	HIS (C)	OV	NBIO

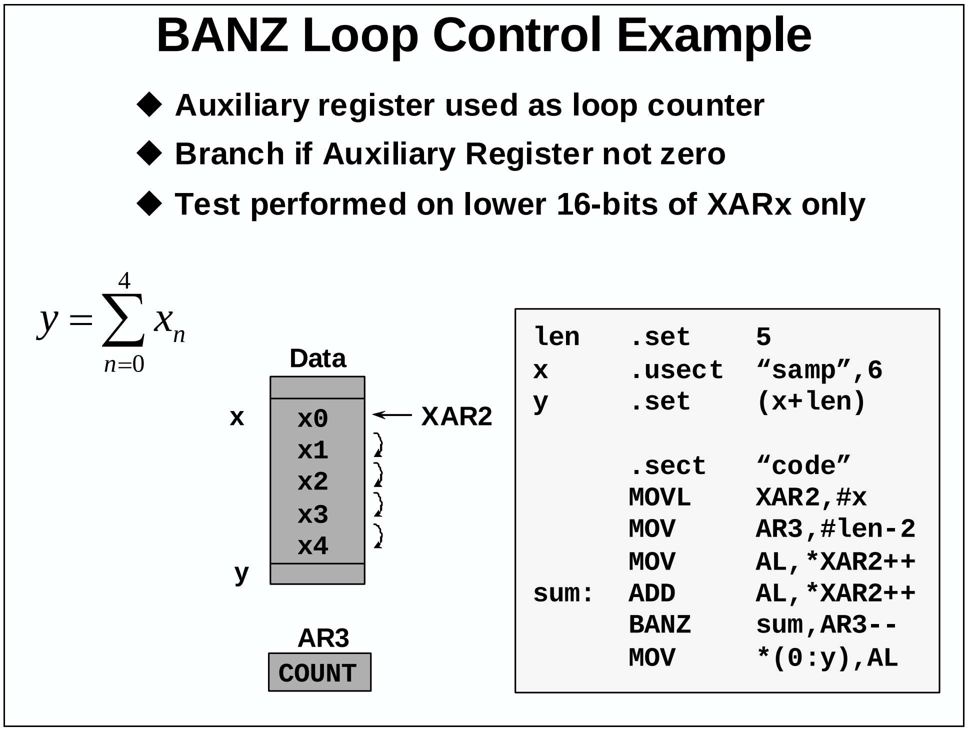
- Condition flags are set on the prior use of the ALU
- The assembler will optimize B to SB if possible



Program Control - Call/Return

Function	Call Code	Cycles	Return code	Cycles
Call	`LCR   22bit`	4	`LRETR`	4
Dynamic Call	`LCR   *XARn`	4	`LRETR`	4
Interrupt Return			`IRET`	8

- More Call variations in the user guide are for code backward compatibility



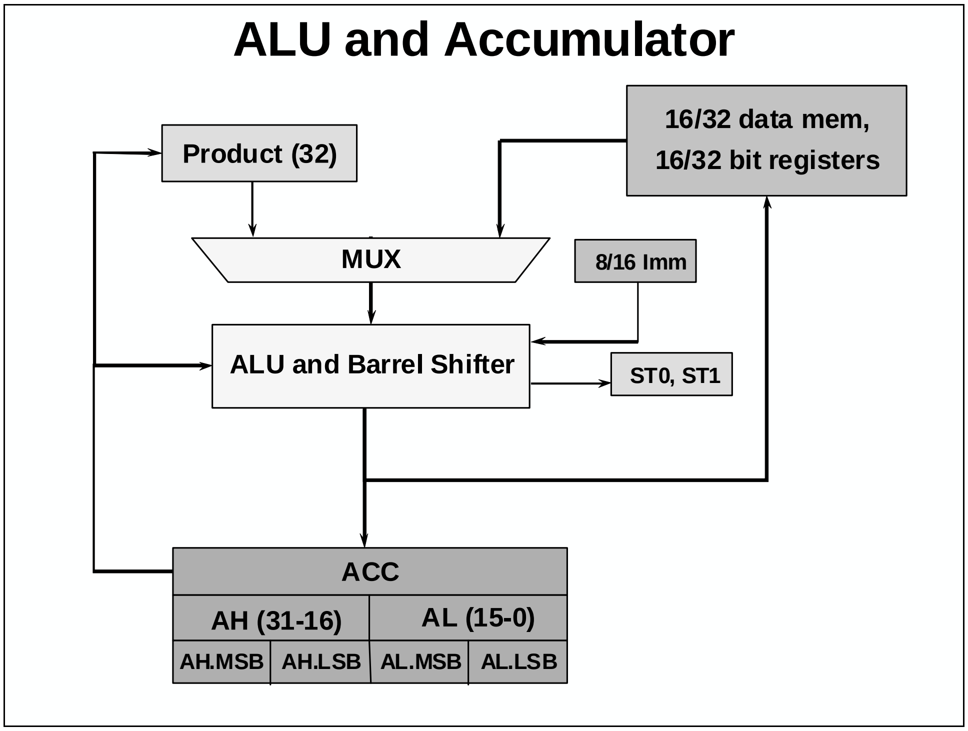
BANZ Loop Control Example

- Auxiliary register used as loop counter
- Branch if Auxiliary Register not zero
- Test performed on lower 16-bits of XARx only

$$y = \sum_{n=0}^{4} x_n$$

```
len     .set     5
x       .usect   “samp”,6
y       .set     (x+len)

        .sect    “code”
        MOVL     XAR2,#x
        MOV      AR3,#len-2
        MOV      AL,*XAR2++
sum:    ADD      AL,*XAR2++
        BANZ     sum,AR3--
        MOV      *(0:y),AL
```

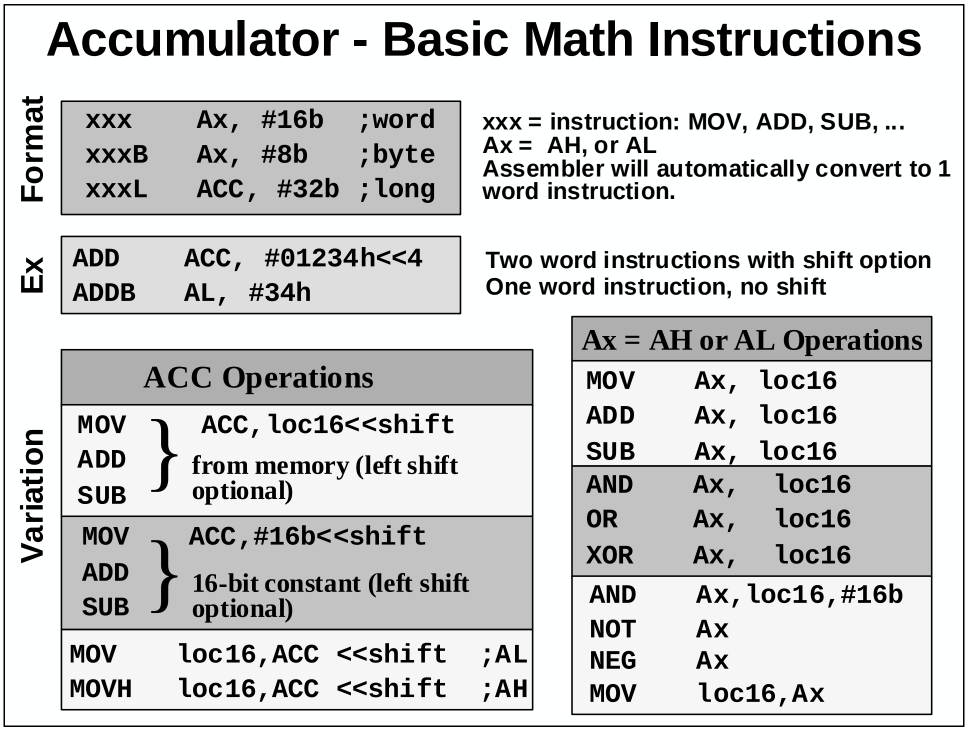
ALU and Accumulator Operations



One of the major components in the execution unit is the Arithmetic-Logical-Unit (ALU). To support the traditional Digital Signal Processing (DSP) operation, the ALU also has the zero cycle barrel shifter and the Accumulator. The enhancement that the C28x has is the additional data paths added form the ALU to all internal CPU registers and data memory. The connection to all internal registers helps the compiler to generate efficient C code. The data path to memory allows the C28x performs single atomic instructions read-modify-write to the memory.

The following slides introduce you to various instructions that use the ALU hardware. Word, byte, and long word 32-bit operation are supported.

Simple Math & Shift



Accumulator - Basic Math Instructions

Format

```
xxx     Ax, #16b   ;word
xxxB    Ax, #8b    ;byte
xxxL    ACC, #32b  ;long
```

xxx = instruction: MOV, ADD, SUB, ...
Ax = AH, or AL
Assembler will automatically convert to 1 word instruction.

Ex

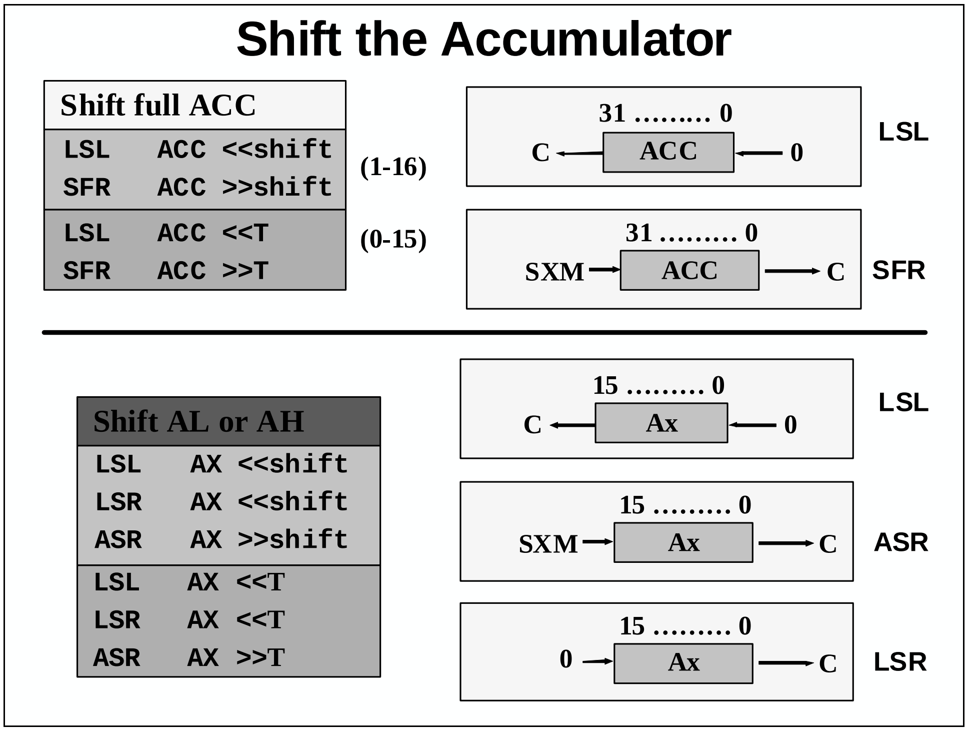
```
ADD     ACC, #01234h<<4
ADDB    AL, #34h
```

Two word instructions with shift option
One word instruction, no shift

Variation

ACC Operations	
MOV ADD SUB	} ACC, loc16<<shift from memory (left shift optional)
MOV ADD SUB	} ACC,#16b<<shift 16-bit constant (left shift optional)
MOV MOVH	loc16,ACC <<shift ;AL loc16,ACC <<shift ;AH

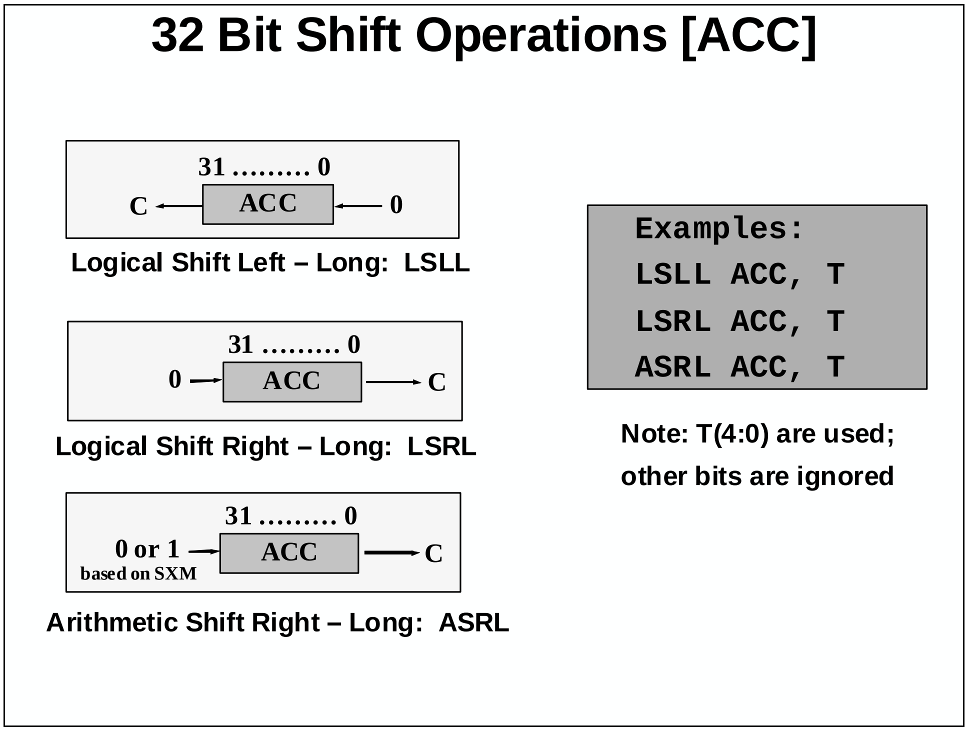
Ax = AH or AL Operations	
MOV	Ax, loc16
ADD	Ax, loc16
SUB	Ax, loc16
AND	Ax, loc16
OR	Ax, loc16
XOR	Ax, loc16
AND	Ax,loc16,#16b
NOT	Ax
NEG	Ax
MOV	loc16,Ax



Shift the Accumulator

Shift full ACC		
LSL	ACC <<shift	(1-16)
SFR	ACC >>shift	
LSL	ACC <<T	(0-15)
SFR	ACC >>T	

Shift AL or AH	
LSL	AX <<shift
LSR	AX <<shift
ASR	AX >>shift
LSL	AX <<T
LSR	AX <<T
ASR	AX >>T



32 Bit Shift Operations [ACC]

Logical Shift Left – Long: LSLL

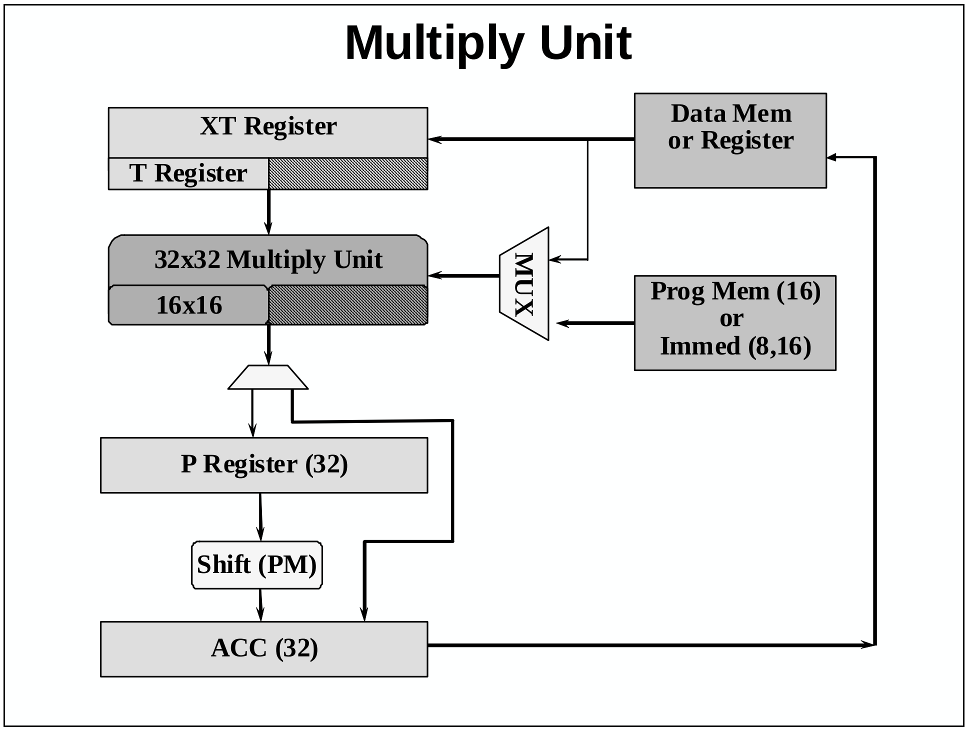
Logical Shift Right – Long: LSRL

Arithmetic Shift Right – Long: ASRL

```
Examples:
LSLL ACC, T
LSRL ACC, T
ASRL ACC, T
```

Note: T(4:0) are used; other bits are ignored

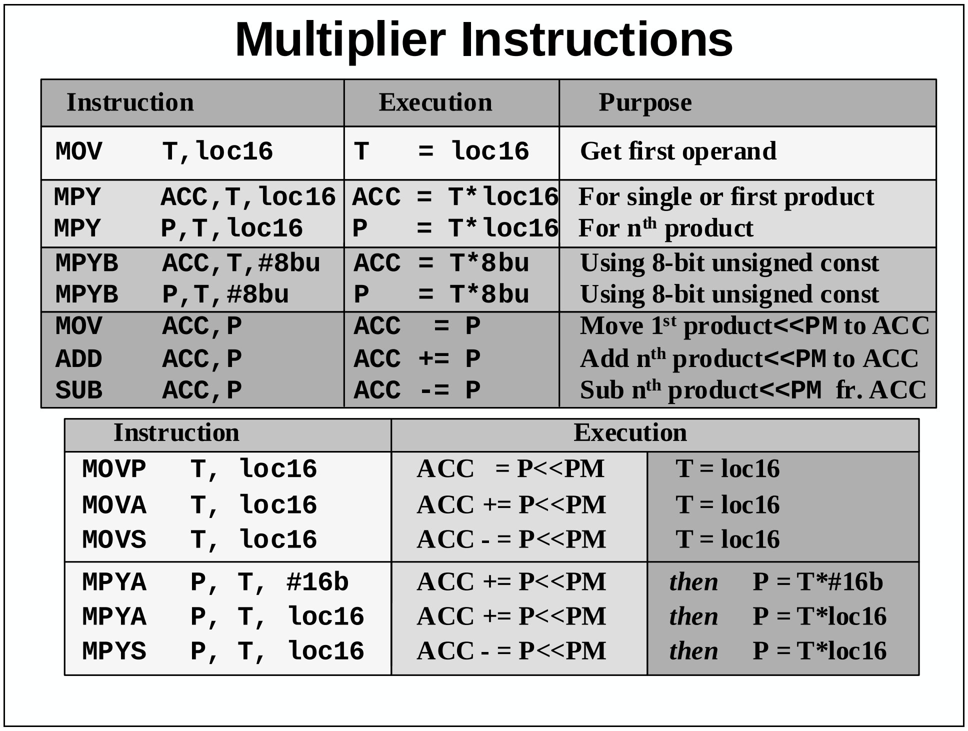
Multiplier



Digital signal processors require many multiply and add math intensive operations. The single cycle multiplier is the second major component in the execution unit. The C28x has the traditional 16-bit-by-16-bit multiplier as previous TI DSP families. In-addition, the C28x has a single cycle 32-bit-by-32-bit multiplier to perform extended precision math operations. The large multiplier allows the C28x to support higher performance control systems requirement while maintaining small or reduce code.

The following slides introduce instructions that use the 16-bit-by-16-bit multiplier and multiply and add (MAC) operations. The 32-bit-by-32-bit multiplication will be covered in the appendix.

Basic Multiplier



Multiplier Instructions

Instruction	Execution	Purpose
`MOV    T,loc16`	`T   = loc16`	Get first operand

`MPY    ACC,T,loc16`
`MPY    P,T,loc16`
	`ACC = T*loc16`
	`P   = T*loc16`
		For single or first product
		For nth product

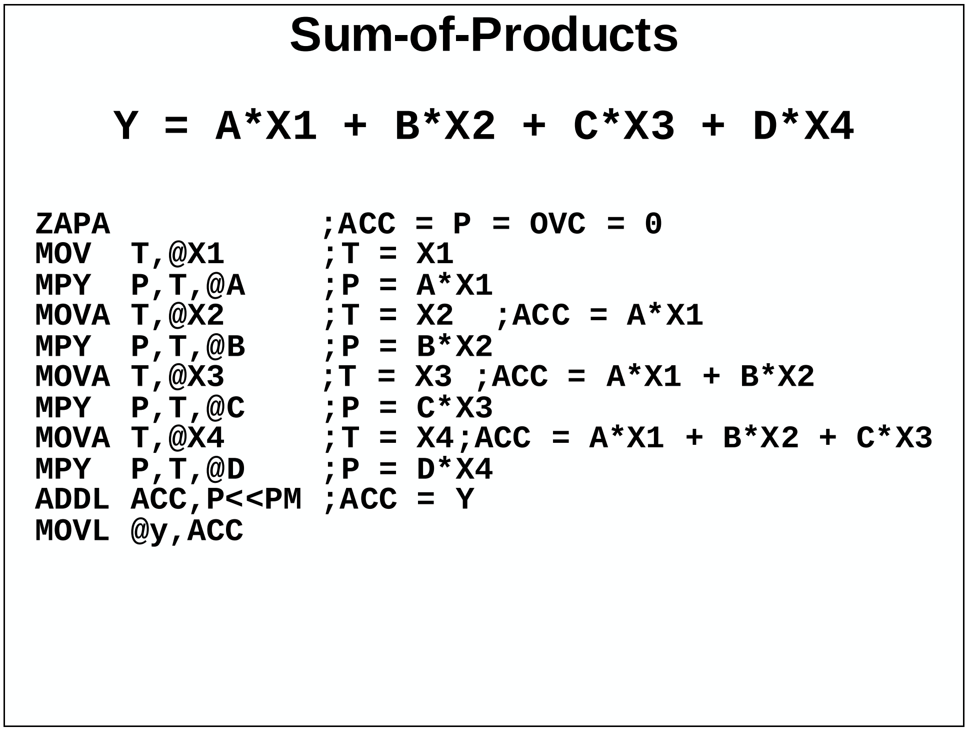
`MPYB   ACC,T,#8bu`
`MPYB   P,T,#8bu`
	`ACC = T*8bu`
	`P   = T*8bu`
		Using 8-bit unsigned const
		Using 8-bit unsigned const

`MOV    ACC,P`
`ADD    ACC,P`
`SUB    ACC,P`
	`ACC  = P`
	`ACC += P`
	`ACC -= P`
		Move 1st product<<PM to ACC
		Add nth product<<PM to ACC
		Sub nth product<<PM fr. ACC

Instruction	Execution	

`MOVP   T, loc16`
`MOVA   T, loc16`
`MOVS   T, loc16`
	ACC = P<<PM
	ACC += P<<PM
	ACC - = P<<PM
		T = loc16
		T = loc16
		T = loc16

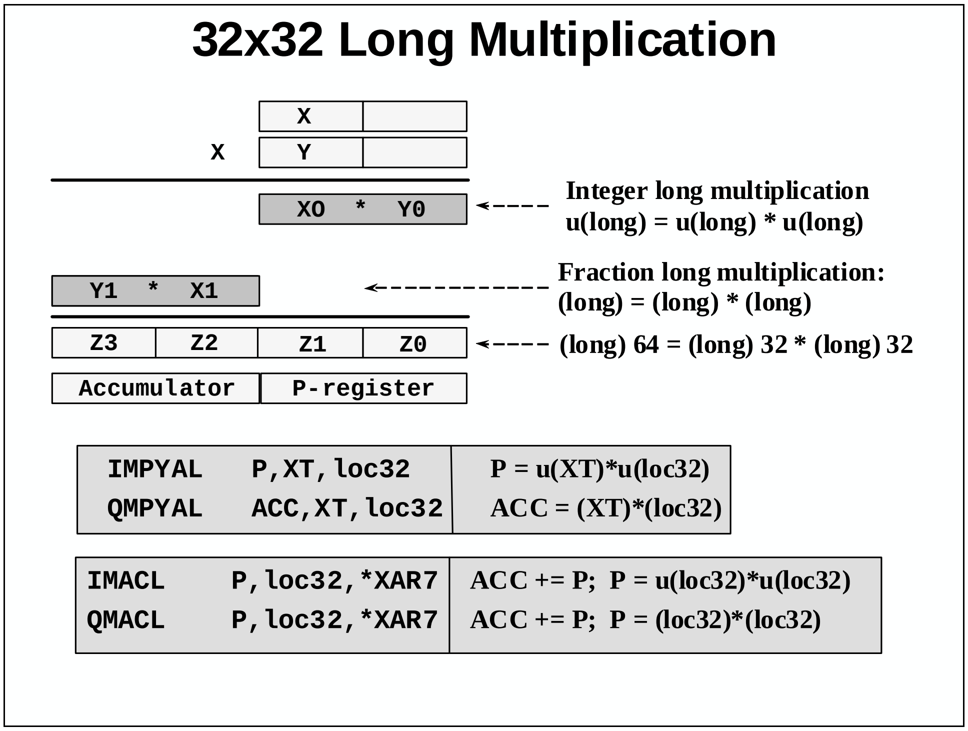
`MPYA   P, T, #16b`
`MPYA   P, T, loc16`
`MPYS   P, T, loc16`
	ACC += P<<PM
	ACC += P<<PM
	ACC - = P<<PM
		then P = T*#16b
		then P = T*loc16
		then P = T*loc16



Sum-of-Products

Y = A*X1 + B*X2 + C*X3 + D*X4

```
ZAPA            ;ACC = P = OVC = 0
MOV  T,@X1      ;T = X1
MPY  P,T,@A     ;P = A*X1
MOVA T,@X2      ;T = X2  ;ACC = A*X1
MPY  P,T,@B     ;P = B*X2
MOVA T,@X3      ;T = X3 ;ACC = A*X1 + B*X2
MPY  P,T,@C     ;P = C*X3
MOVA T,@X4      ;T = X4;ACC = A*X1 + B*X2 + C*X3
MPY  P,T,@D     ;P = D*X4
ADDL ACC,P<<PM ;ACC = Y
MOVL @y,ACC
```



32x32 Long Multiplication

Integer long multiplication
u(long) = u(long) * u(long)

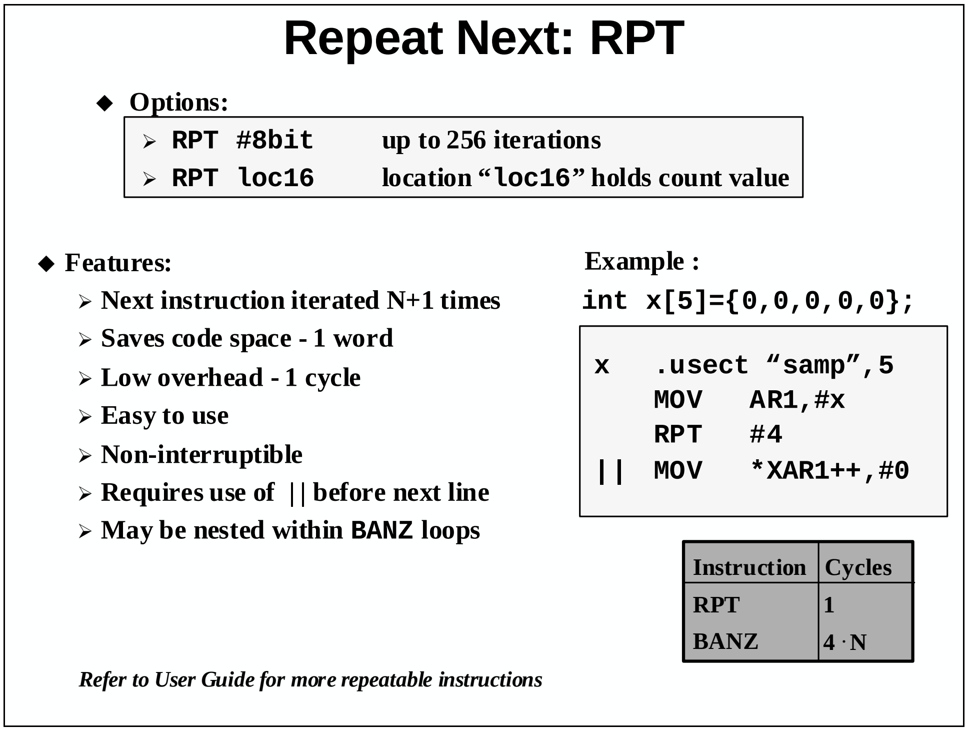
Fraction long multiplication:
(long) = (long) * (long)

(long) 64 = (long) 32 * (long) 32

`IMPYAL P,XT,loc32`	P = u(XT)*u(loc32)
`QMPYAL ACC,XT,loc32`	ACC = (XT)*(loc32)

`IMACL P,loc32,*XAR7`	ACC += P; P = u(loc32)*u(loc32)
`QMACL P,loc32,*XAR7`	ACC += P; P = (loc32)*(loc32)

Repeat Instruction



Repeat Next: RPT

- **Options:**

➢ `RPT #8bit`	up to 256 iterations
➢ `RPT loc16`	location "`loc16`" holds count value

- **Features:**
  - Next instruction iterated N+1 times
  - Saves code space - 1 word
  - Low overhead - 1 cycle
  - Easy to use
  - Non-interruptible
  - Requires use of || before next line
  - May be nested within `BANZ` loops

Example :

```
int x[5]={0,0,0,0,0};
```

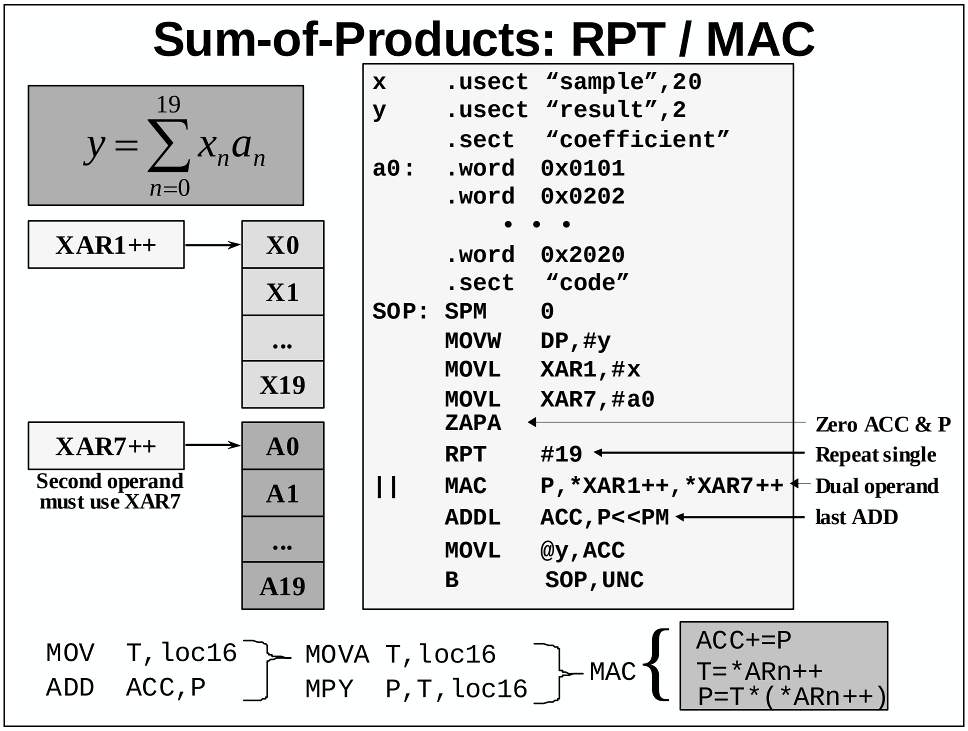
```
x   .usect "samp",5
    MOV   AR1,#x
    RPT   #4
||  MOV   *XAR1++,#0
```

Instruction	Cycles
RPT	1
BANZ	4 · N

Refer to User Guide for more repeatable instructions

Single repeat instruction (RPT) is used to reduce code size and speed up many operations in the DSP application. Some of the most popular operations that use the RPT instruction to perform multiple taps digital filters or perform block of data transfer.

MAC Instruction



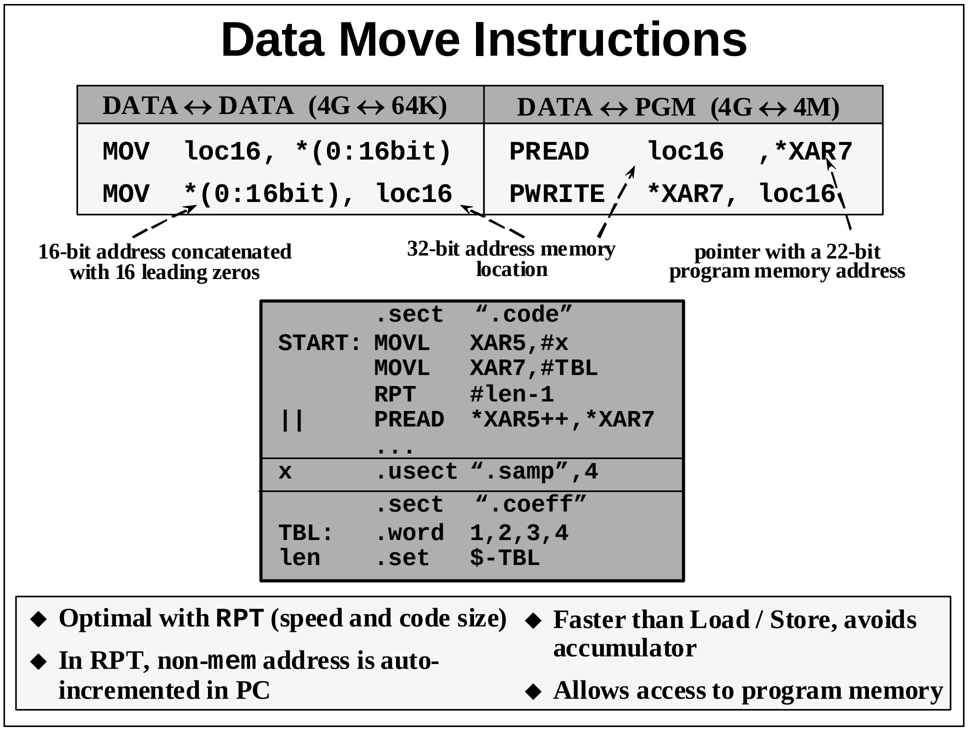
Sum-of-Products: RPT / MAC

$$y = \sum_{n=0}^{19} x_n a_n$$

```
x    .usect “sample”,20
y    .usect “result”,2
     .sect  “coefficient”
a0:  .word  0x0101
     .word  0x0202
          • • •
     .word  0x2020
     .sect  “code”
SOP: SPM    0
     MOVW   DP,#y
     MOVL   XAR1,#x
     MOVL   XAR7,#a0
     ZAPA
     RPT    #19
||   MAC    P,*XAR1++,*XAR7++
     ADDL   ACC,P<<PM
     MOVL   @y,ACC
     B       SOP,UNC
```

```
MOV  T,loc16    MOVA T,loc16       ACC+=P
ADD  ACC,P      MPY  P,T,loc16 MAC T=*ARn++
                                   P=T*(*ARn++)
```

Data Move



Data Move Instructions

DATA ↔ DATA (4G ↔ 64K)	DATA ↔ PGM (4G ↔ 4M)
`MOV loc16, *(0:16bit)`	`PREAD loc16 ,*XAR7`
`MOV *(0:16bit), loc16`	`PWRITE *XAR7, loc16`

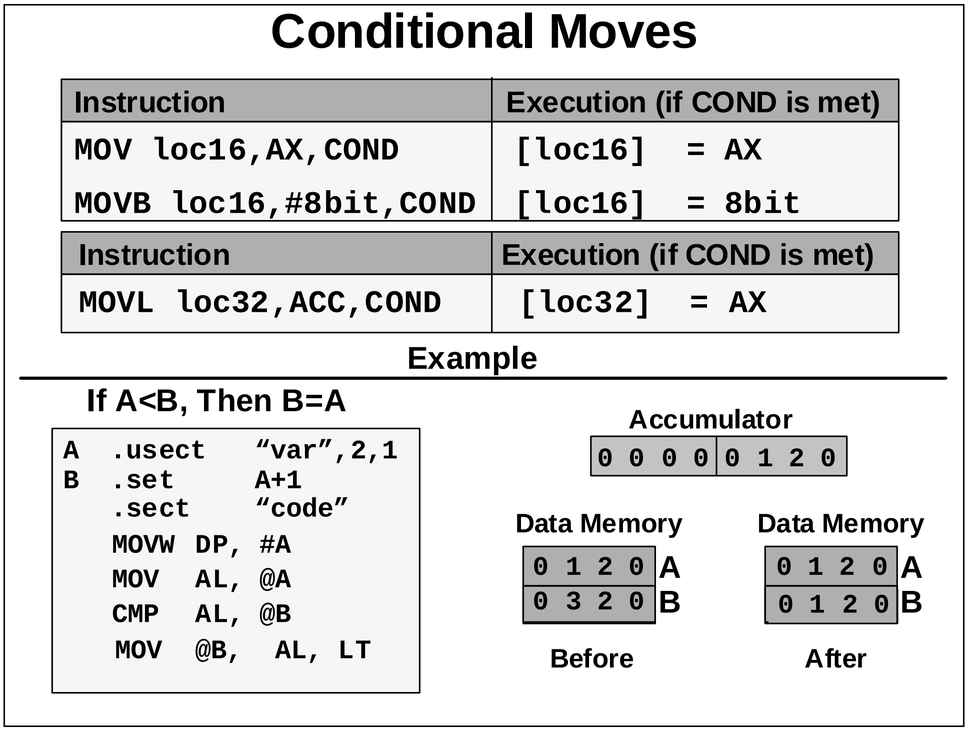
16-bit address concatenated with 16 leading zeros

32-bit address memory location

pointer with a 22-bit program memory address

```
       .sect   ".code"
START: MOVL    XAR5,#x
       MOVL    XAR7,#TBL
       RPT     #len-1
||     PREAD   *XAR5++,*XAR7
       ...
x      .usect ".samp",4
       .sect   ".coeff"
TBL:   .word   1,2,3,4
len    .set    $-TBL
```

- ◆ Optimal with RPT (speed and code size)
- ◆ In RPT, non-mem address is auto-incremented in PC
- ◆ Faster than Load / Store, avoids accumulator
- ◆ Allows access to program memory



Conditional Moves

Instruction	Execution (if COND is met)
`MOV loc16,AX,COND`	`[loc16] = AX`
`MOVB loc16,#8bit,COND`	`[loc16] = 8bit`

Instruction	Execution (if COND is met)
`MOVL loc32,ACC,COND`	`[loc32] = AX`

Example

If A<B, Then B=A

```
A  .usect   "var",2,1
B  .set     A+1
   .sect    "code"
   MOVW DP, #A
   MOV  AL, @A
   CMP  AL, @B
   MOV  @B,  AL, LT
```

Accumulator

0 0 0 0	0 1 2 0

Data Memory		Data Memory	
0 1 2 0	A	0 1 2 0	A
0 3 2 0	B	0 1 2 0	B
Before		After	

The conditional move instruction is an excellent way to avoid a discontinuity (branch or call) based upon a condition code set prior to the instruction. In the above example, the 1st step is to place the contents of A into the accumulator. Once the Ax content is tested, by using the CMP instruction, the conditional move can be executed.

If the specified condition being tested is true, then the location pointed to by the “loc16” addressing mode or the 8–bit zero extended constant will be loaded with the contents of the specified AX register (AH or AL):   if (COND == true) [loc16] = AX or 0:8bit;

Note: Addressing modes are not conditionally executed. Hence, if an addressing mode performs a pre or post modification, it will execute regardless if the condition is true or not. This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Flags and Modes

N - If the condition is true, then after the move, AX is tested for a negative condition. The negative flag bit is set if bit 15 of AX is 1, otherwise it is cleared.

Z - If the condition then after the move, AX is tested for a zero condition. The zero flag bit is set if AX = 0, otherwise it is cleared.

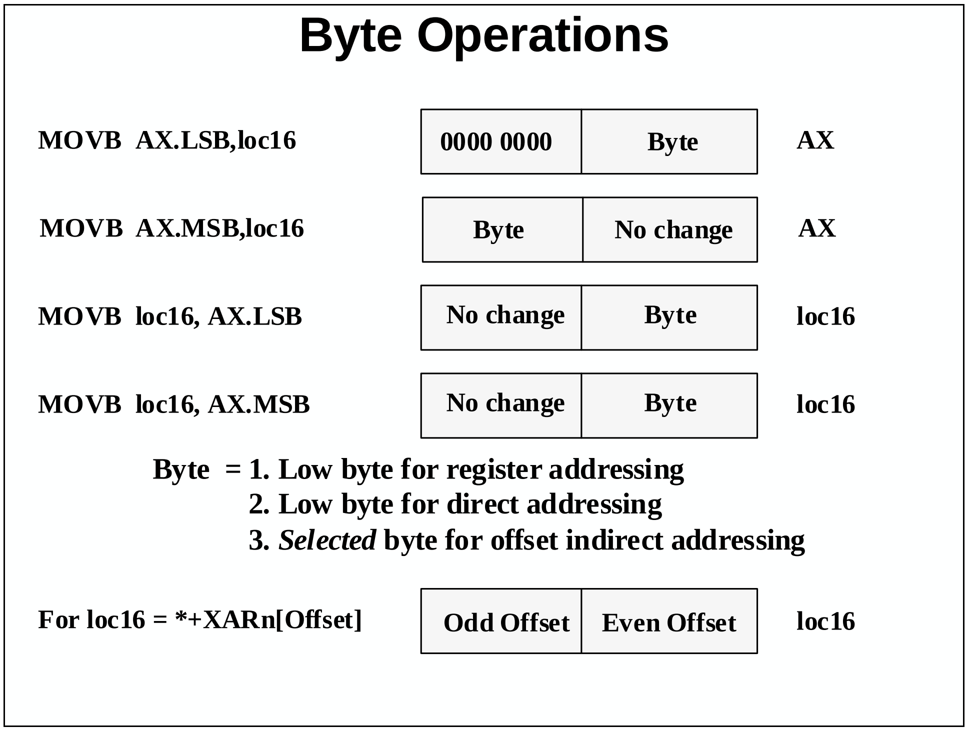
V - If the V flag is tested by the condition, then V is cleared.

C-Example

```
; if ( VarA > 20 )
; VarA = 0;

CMP @VarA,#20 ; Set flags on (VarA – 20)
MOVB @VarA,#0,GT ; Zero VarA if greater then
```

Logical Operations

Byte Operations and Addressing

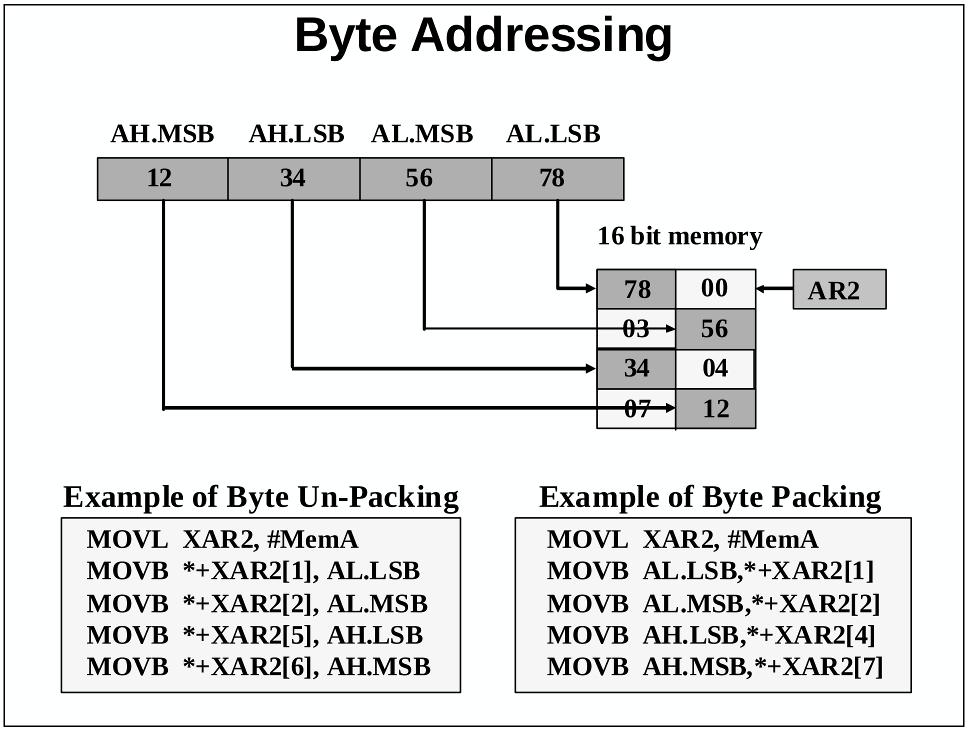


Byte Operations

MOVB AX.LSB,loc16	0000 0000	Byte	AX
MOVB AX.MSB,loc16	Byte	No change	AX
MOVB loc16, AX.LSB	No change	Byte	loc16
MOVB loc16, AX.MSB	No change	Byte	loc16

Byte = 1. Low byte for register addressing
2. Low byte for direct addressing
3. *Selected* byte for offset indirect addressing

For loc16 = *+XARn[Offset]	Odd Offset	Even Offset	loc16



Byte Addressing

78	00
03	56
34	04
07	12

Example of Byte Un-Packing

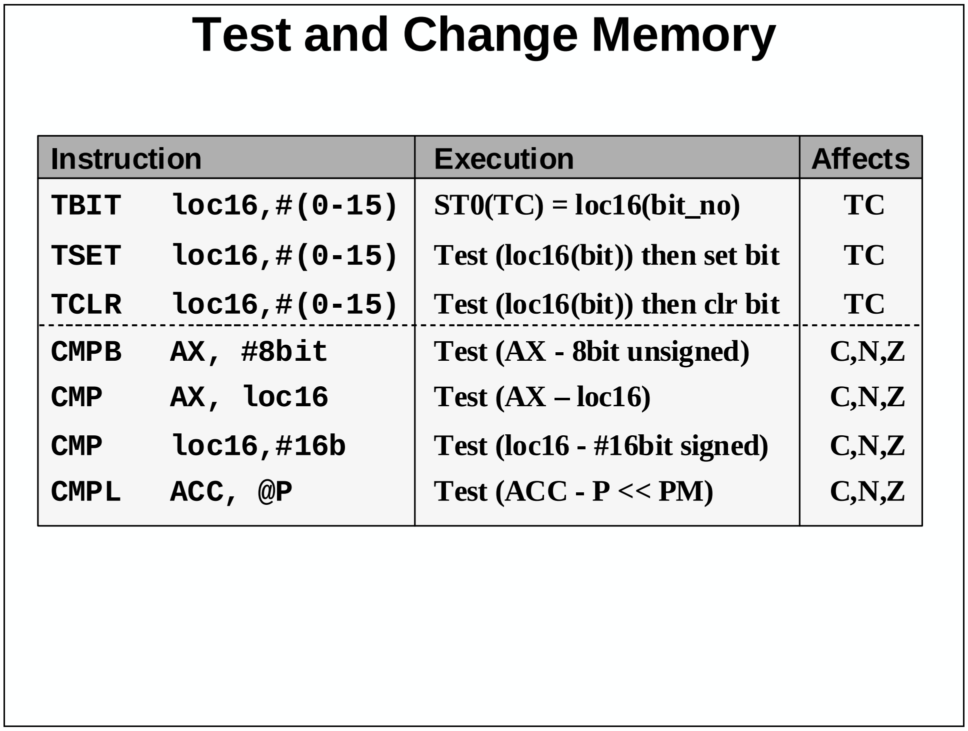
```
MOVL  XAR2, #MemA
MOVB  *+XAR2[1], AL.LSB
MOVB  *+XAR2[2], AL.MSB
MOVB  *+XAR2[5], AH.LSB
MOVB  *+XAR2[6], AH.MSB
```

Example of Byte Packing

```
MOVL  XAR2, #MemA
MOVB  AL.LSB,*+XAR2[1]
MOVB  AL.MSB,*+XAR2[2]
MOVB  AH.LSB,*+XAR2[4]
MOVB  AH.MSB,*+XAR2[7]
```

Test and Change Memory Instructions

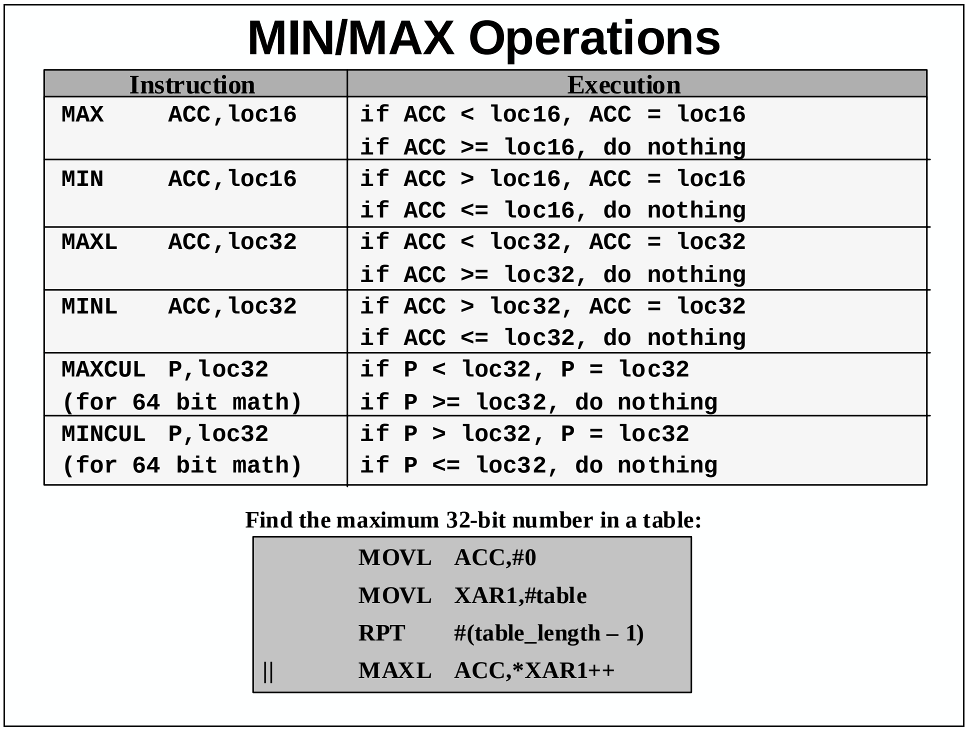
The compare (CMPx) and test (Txxx) instructions allow the ability to test values in memory. The results of these operations can then trigger subsequent conditional branches. The CMPx instruction allows comparison of memory with respect to a specified constant value, while the Txxx instructions allow any single bit to be extracted to the test control (TC) field of status register 0. The contents of the accumulator can also be non-destructively analyzed to establish branching conditions, as seen below.



Test and Change Memory

Instruction	Execution	Affects
`TBIT   loc16,#(0-15)`	ST0(TC) = loc16(bit_no)	TC
`TSET   loc16,#(0-15)`	Test (loc16(bit)) then set bit	TC
`TCLR   loc16,#(0-15)`	Test (loc16(bit)) then clr bit	TC
`CMPB   AX, #8bit`	Test (AX - 8bit unsigned)	C,N,Z
`CMP    AX, loc16`	Test (AX – loc16)	C,N,Z
`CMP    loc16,#16b`	Test (loc16 - #16bit signed)	C,N,Z
`CMPL   ACC, @P`	Test (ACC - P << PM)	C,N,Z

Min/Max Operations



MIN/MAX Operations

Instruction	Execution

`MAX ACC,loc16`	`if ACC < loc16, ACC = loc16`
	`if ACC >= loc16, do nothing`

`MIN ACC,loc16`	`if ACC > loc16, ACC = loc16`
	`if ACC <= loc16, do nothing`

`MAXL ACC,loc32`	`if ACC < loc32, ACC = loc32`
	`if ACC >= loc32, do nothing`

`MINL ACC,loc32`	`if ACC > loc32, ACC = loc32`
	`if ACC <= loc32, do nothing`

`MAXCUL P,loc32`
`(for 64 bit math)`
	`if P < loc32, P = loc32`
	`if P >= loc32, do nothing`

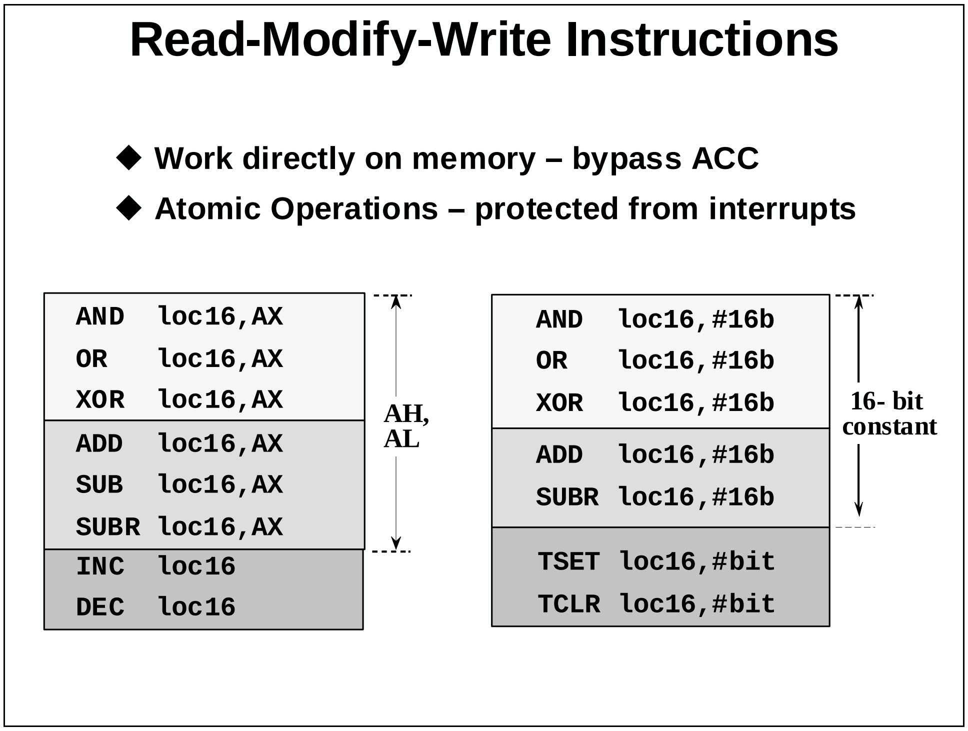
`MINCUL P,loc32`
`(for 64 bit math)`
	`if P > loc32, P = loc32`
	`if P <= loc32, do nothing`

Find the maximum 32-bit number in a table:

```
        MOVL    ACC,#0
        MOVL    XAR1,#table
        RPT     #(table_length – 1)
||      MAXL    ACC,*XAR1++
```

Read Modify Write Operations

The accumulator (ACC) is the main working register for the C28x. It is the destination of all ALU operations except those, which operate directly on memory or registers. The accumulator supports single-cycle move, add, subtract and compare operations from 32-bit-wide data memory. It can also accept the 32-bit result of a multiplication operation. These one or two cycle operations are referred to as read-modify-write operations, or as atomic instructions.

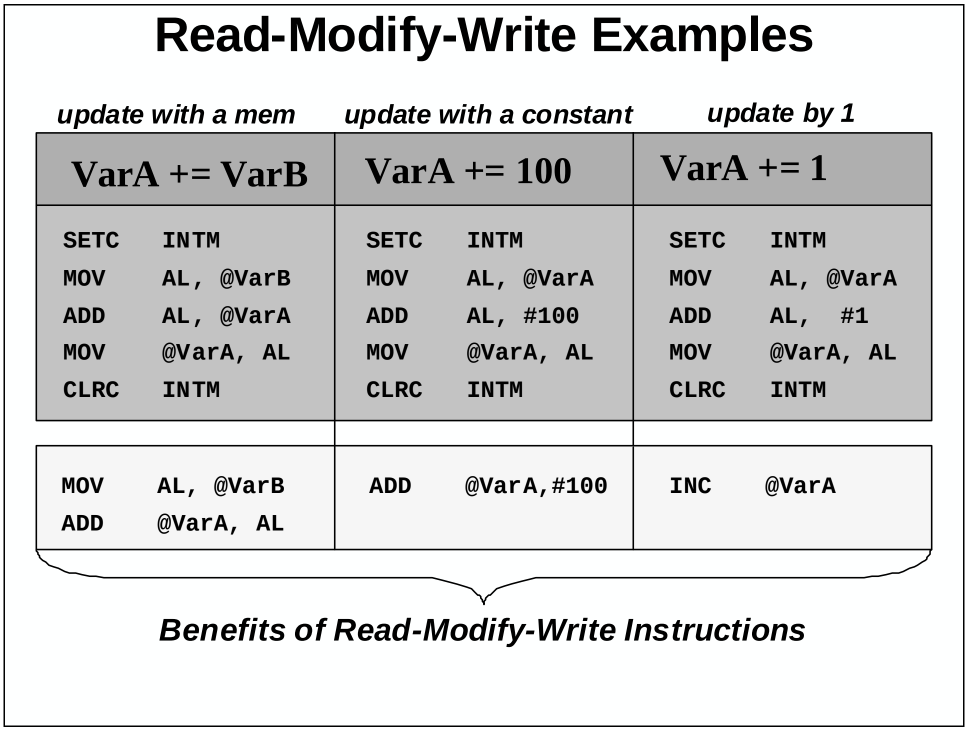


Read-Modify-Write Instructions

- ◆ **Work directly on memory – bypass ACC**
- ◆ **Atomic Operations – protected from interrupts**

```
AND  loc16,AX
OR   loc16,AX
XOR  loc16,AX
ADD  loc16,AX
SUB  loc16,AX
SUBR loc16,AX
INC  loc16
DEC  loc16
```

```
AND  loc16,#16b
OR   loc16,#16b
XOR  loc16,#16b
ADD  loc16,#16b
SUBR loc16,#16b
TSET loc16,#bit
TCLR loc16,#bit
```



Read-Modify-Write Examples

update with a mem	*update with a constant*	*update by 1*
VarA += VarB	**VarA += 100**	**VarA += 1**

`SETC INTM`
`MOV AL, @VarB`
`ADD AL, @VarA`
`MOV @VarA, AL`
`CLRC INTM`
	`SETC INTM`
	`MOV AL, @VarA`
	`ADD AL, #100`
	`MOV @VarA, AL`
	`CLRC INTM`
		`SETC INTM`
		`MOV AL, @VarA`
		`ADD AL, #1`
		`MOV @VarA, AL`
		`CLRC INTM`

`MOV AL, @VarB`
`ADD @VarA, AL`
	`ADD @VarA,#100`	`INC @VarA`

Benefits of Read-Modify-Write Instructions

Lab C: Assembly Programming

Note: *The lab linker command file is based on the F28035 memory map – modify as needed, if using a different F28xx device memory map.*

- **Objective**

The objective of this lab is to practice and verify the mechanics of performing assembly language programming arithmetic on the TMS320C28x. In this exercise, we will expand upon the `.asm` file from the previous lab to include new functions. Code will be added to obtain the sum of the products of the values from each array.

Perform the sum of products using a MAC-based implementation. In a real system application, the *coeff* array may well be constant (values do not change), therefore one can modify the initialization routine to skip the transfer of this arrays, thus reducing the amount of data RAM and cycles required for initialization. Also there is no need to copy the zero to clear the result location. The initialization routine from the previous lab using the load/store operation will be replaced with a looped BANZ implementation.

As in previous lab, consider which addressing modes are optimal for the tasks to be performed. You may perform the lab based on this information alone, or may refer to the following procedure.

- **Procedure**

Copy Files, Create Project File

1. Create a new project called `LabC.pjt` in `C:\C28x\Labs\Appendix\LabC` and add `LabC.asm` and `Lab.cmd` to it. Check your file list to make sure all the files are there. Be sure to setup the Build Options by clicking: `Project` → `Build Options` on the menu bar. Select the Linker tab. In the middle of the screen select `“No Autoinitialization”` under `“Autoinit Model:”`. Enter `start` in the `“Code Entry Point (-e):”` field. Next, select the Compiler tab. Note that `“Full Symbolic Debug (-g)”` under `“Generate Debug Info:”` is selected. Then select `OK` to save the Build Options.

Initialization Routine using BANZ

2. Edit `LabC.asm` and modify it by replacing the initialization routine using the load/store operation with a BANZ process. Remember, it is only necessary to copy the first four values (i.e. initialize the *data* array). Do you still need the *coeff* array in the *vars* section?

3. Save your work. If you would like, you can use Code Composer Studio to verify the correct operation of the block initialization before moving to the next step.

Sum of Products using a RPT/MAC-based Implementation

4. Edit `LabC.asm` to add a RPT/MAC-based implementation to multiply the *coeff* array by the *data* array and storing the final sum-of-product value to *result*.

Build and Load

5. Click the `“Build”` button and watch the tools run in the build window. Debug as necessary. To open up more space, close any open files or windows that you do not need.

6. If the `“Load program after build”` option was not selected in Code Composer Studio, load the output file onto the target. Click: `File → Load Program`…

   If you wish, right click on the source window and select `Mixed Mode` to debug using both source and assembly.

7. Single-step your routine. While single-stepping, open memory windows to see the values located in *table [9]* and *data [9]*. Open the CPU Registers. Check to see if the program is working as expected. Debug and modify, if needed.

Optional Exercise

After completing the above, edit `LabC.asm` and modify it to perform the initialization process using a RTP/PREAD rather than a load/store/BANZ.

End of Exercise

OPTIONAL Lab C-C: Sum-of-Products in C

Note: *The lab linker command file is based on the F28035 memory map – modify as needed, if using a different F28xx device memory map.*

- **Objective**

The objective of this lab is to practice and verify the mechanics of performing C programming arithmetic on the TMS320C28x. The objective will be to add the code necessary to obtain the sum of the products of the n-th values from each array.

- **Procedure**

Create Project File

1. In Code Composer Studio create a new project called `LabC-C.pjt` in `C:\C28x\Labs\Appendix\LabC\LabC-C` and add `LabC-C.c` and `Lab.cmd` to it. Check your file list to make sure all the files are there. Open the Build Options and select the Linker tab. Select the "Libraries" Category and enter `rts2800_ml.lib` in the `"Incl. Libraries (-l):"` box. *Do not* setup any other Build Options. The default values will be used. In Appendix Lab D exercise, we will experiement and explore the various build options when working with C.

Sum of Products using a MAC-based Implementation

2. Edit `LabC-C.c` and modify the "main" routine to perform a MAC-based implementation in C. Since the MAC operation requires one array to be in program memory, the initialization routine can skip the transfer of one of the arrays, thus reducing the amount of data RAM and cycles required for initialization.

Build and Load

3. Click the `"Build"` button and watch the tools run in the build window. Debug as necessary.

Note: Have Code Composer Studio automatically load the output file after a successful build. On the menu bar click: `Option → Customize…` and select the `"Program Load Options"` tab, `check "Load Program After Build"`, then click `OK`.

4. Under `Debug` on the menu bar click `"Go Main"`. Single-step your routine. While single-stepping, open memory windows to see the values located in *table [9]* and *data [9]* . (Note: *data[9]* consists of the allocated arrays of *data*, *coeff*, and *result*). Open the CPU Registers. Check to see if the program is working as expected. Debug and modify, if needed.

End of Exercise

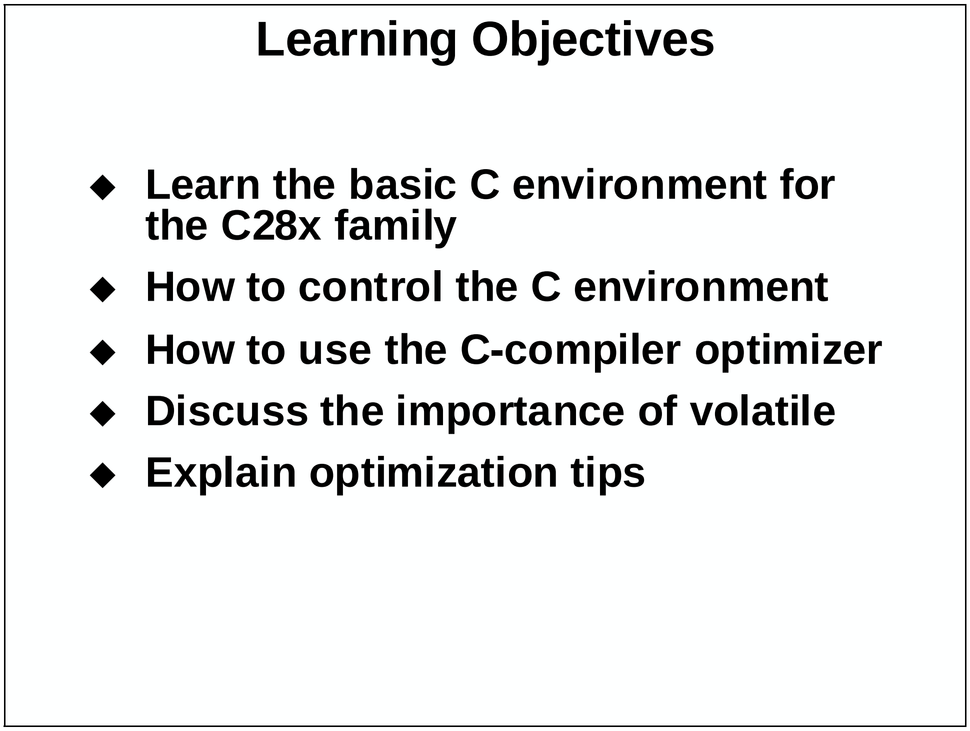
Appendix D – C Programming

Introduction

The C28x architecture, hardware, and compiler have been designed to efficiently support C code programming.

Appendix D will focus on how to program in C for an embedded system. Issues related to programming in C and how C behaves in the C28x environment will be discussed. Also, the C compiler optimization features will be explained.

Learning Objectives

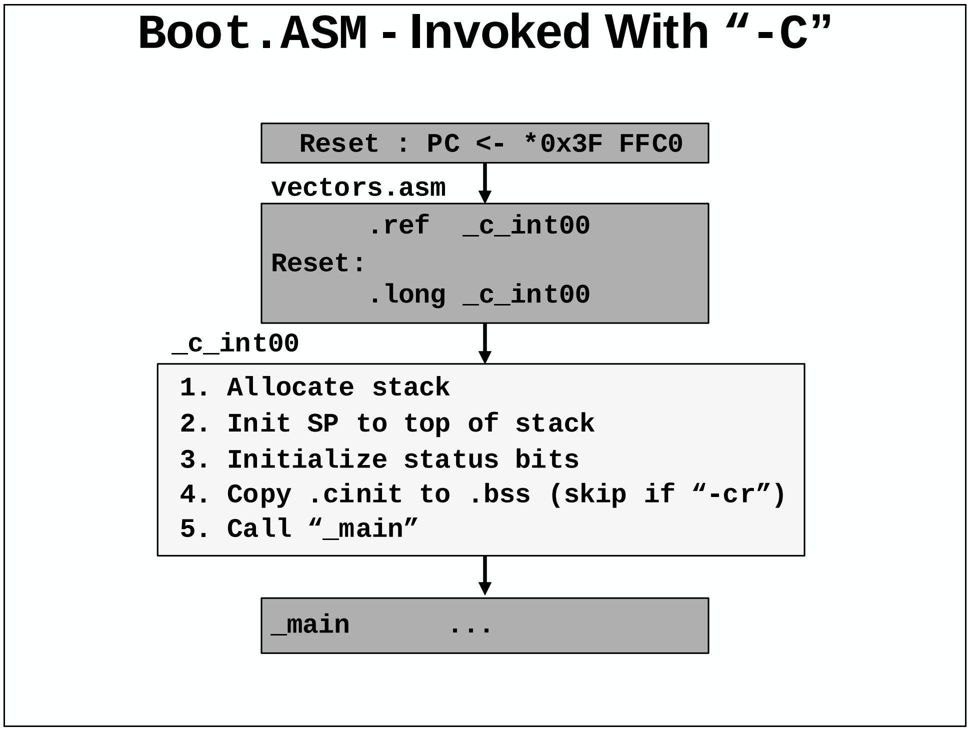


Learning Objectives

- Learn the basic C environment for the C28x family
- How to control the C environment
- How to use the C-compiler optimizer
- Discuss the importance of volatile
- Explain optimization tips

Module Topics

Linking Boot code from RTS2800.lib



Boot.ASM - Invoked With “-C”

```
Reset : PC <- *0x3F FFC0
```

vectors.asm

```
      .ref  _c_int00
Reset:
      .long _c_int00
```

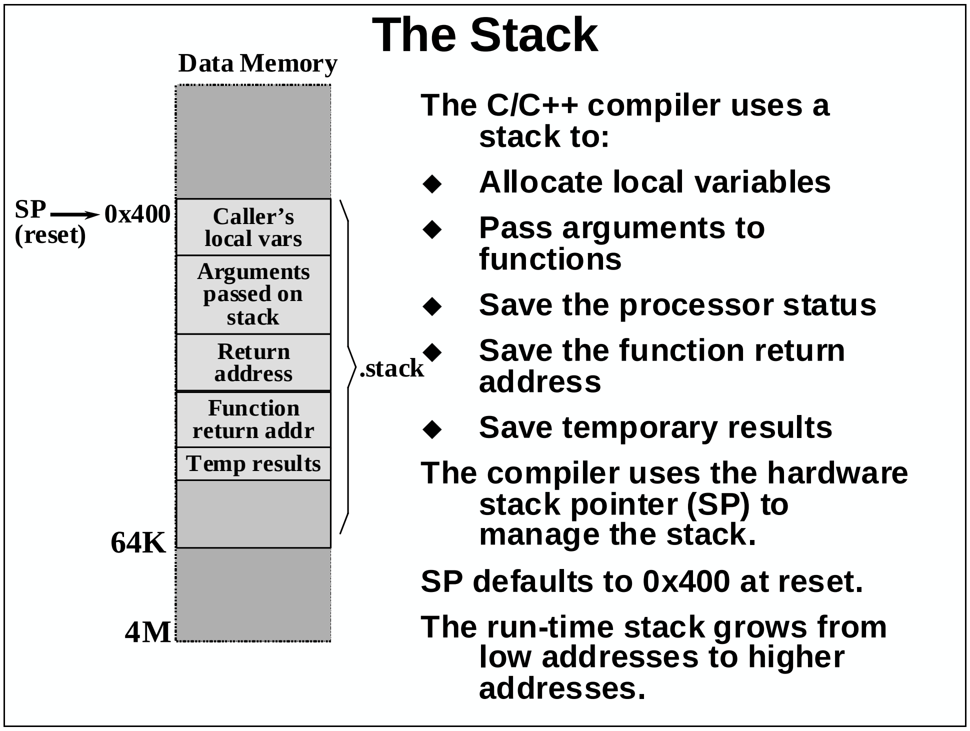
_c_int00

```
1. Allocate stack
2. Init SP to top of stack
3. Initialize status bits
4. Copy .cinit to .bss (skip if “-cr”)
5. Call “_main”
```

```
_main      ...
```

The boot routine is used to establish the environment for C before launching main. The boot routine begins with the label _c_int00 and the reset vector should contain a ".long" to this address to make boot.asm the reset routine. The contents of the boot routine have been extracted and copied on the following page so they may be inspected. Note the various functions performed by the boot routine, including the allocation and setup of the stack, setting of various C-requisite statuses, the initialization of global and static variables, and the call to main. Note that if the link was performed using the "–cr" option instead of the "–c" option that the global/static variable initialization is *not* performed. This is useful on RAM-based C28x systems that were initialized during reset by some external host processor, making transfer of initialization values unnecessary. Later on in this chapter, there is an example on how to do the vectors in C code rather than assembly.

Set up the Stack



The Stack

Data Memory

SP → 0x400 (reset)

Data Memory
Caller's local vars
Arguments passed on stack
Return address
Function return addr
Temp results

.stack

64K

4M

The C/C++ compiler uses a stack to:

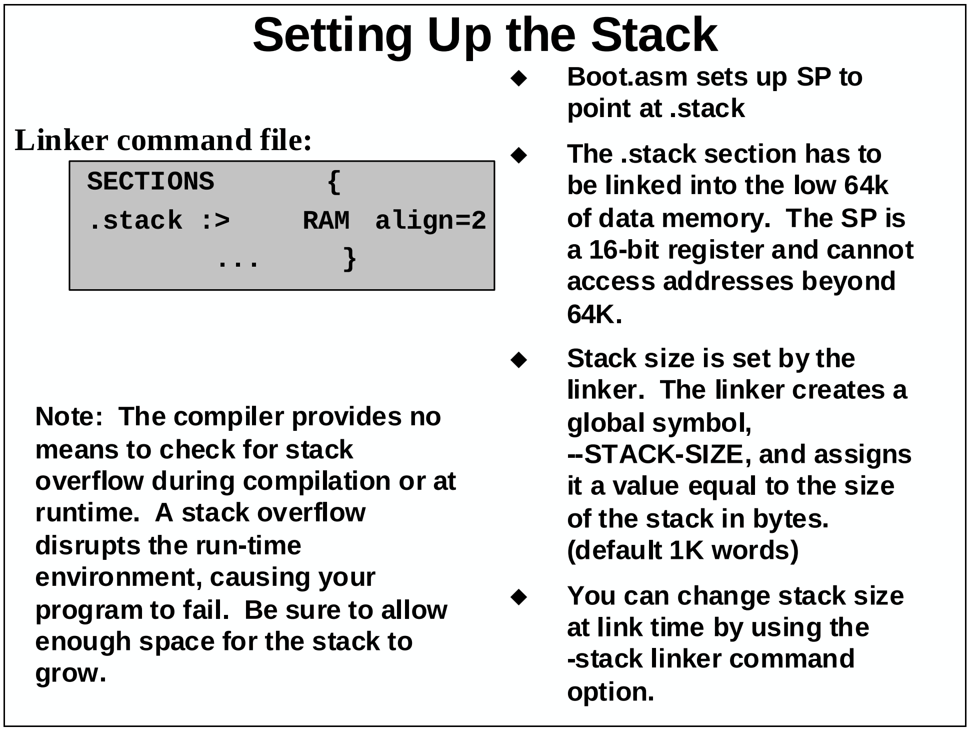
- ◆ Allocate local variables
- ◆ Pass arguments to functions
- ◆ Save the processor status
- ◆ Save the function return address
- ◆ Save temporary results

The compiler uses the hardware stack pointer (SP) to manage the stack.

SP defaults to 0x400 at reset.

The run-time stack grows from low addresses to higher addresses.

The C28x has a 16-bit stack pointer (SP) allowing accesses to the base 64K of memory. The stack grows from low to high memory and always points to the first *unused* location. The compiler uses the hardware stack pointer (SP) to manage the stack. The stack size is set by the linker.



Setting Up the Stack

Linker command file:

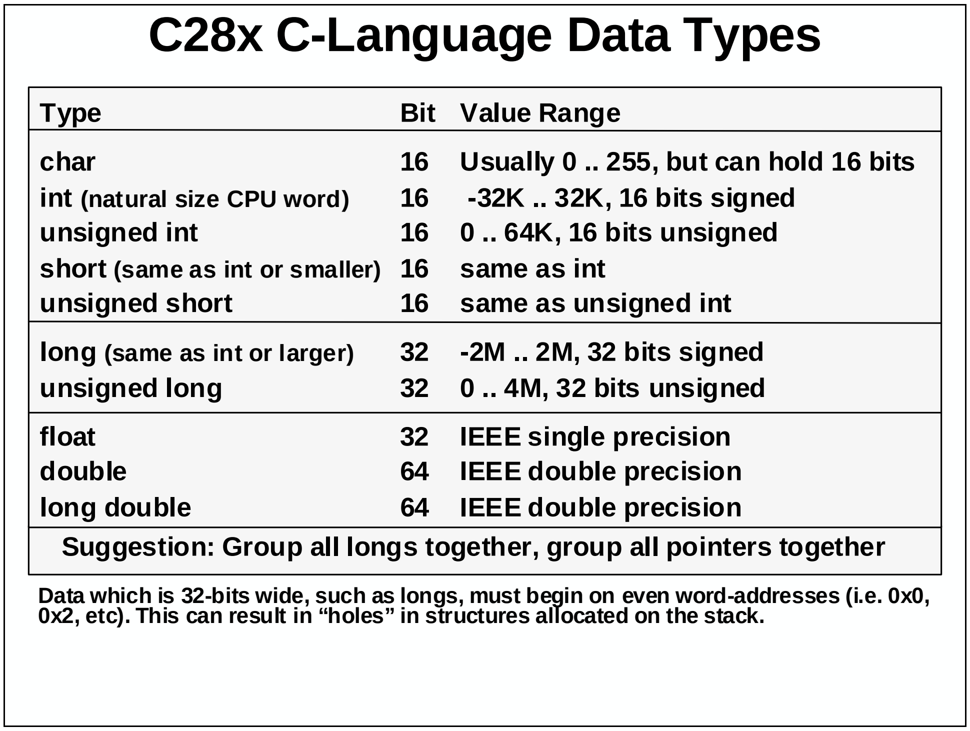
```
SECTIONS       {
.stack :>     RAM  align=2
        ...     }
```

Note: The compiler provides no means to check for stack overflow during compilation or at runtime. A stack overflow disrupts the run-time environment, causing your program to fail. Be sure to allow enough space for the stack to grow.

- ◆ Boot.asm sets up SP to point at .stack
- ◆ The .stack section has to be linked into the low 64k of data memory. The SP is a 16-bit register and cannot access addresses beyond 64K.
- ◆ Stack size is set by the linker. The linker creates a global symbol, --STACK-SIZE, and assigns it a value equal to the size of the stack in bytes. (default 1K words)
- ◆ You can change stack size at link time by using the -stack linker command option.

In order to allocate the stack the linker command file needs to have “align = 2.”

C28x Data Types

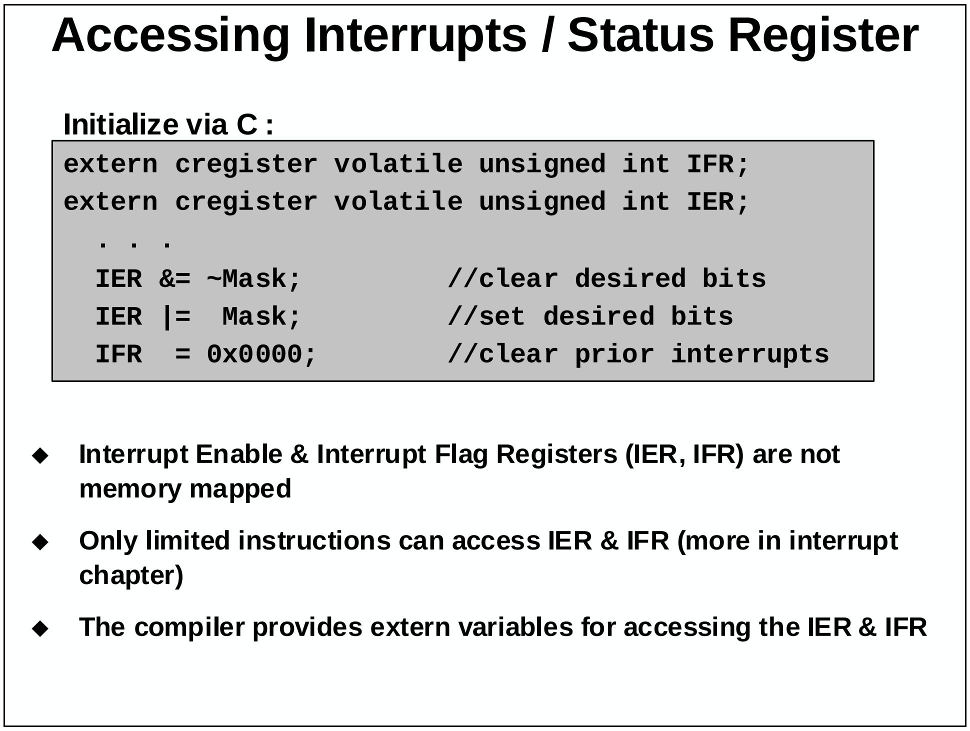


C28x C-Language Data Types

Type	Bit	Value Range
char	16	Usually 0 .. 255, but can hold 16 bits
int (natural size CPU word)	16	-32K .. 32K, 16 bits signed
unsigned int	16	0 .. 64K, 16 bits unsigned
short (same as int or smaller)	16	same as int
unsigned short	16	same as unsigned int
long (same as int or larger)	32	-2M .. 2M, 32 bits signed
unsigned long	32	0 .. 4M, 32 bits unsigned
float	32	IEEE single precision
double	64	IEEE double precision
long double	64	IEEE double precision
Suggestion: Group all longs together, group all pointers together		

Data which is 32-bits wide, such as longs, must begin on even word-addresses (i.e. 0x0, 0x2, etc). This can result in “holes” in structures allocated on the stack.

Accessing Interrupts / Status Register



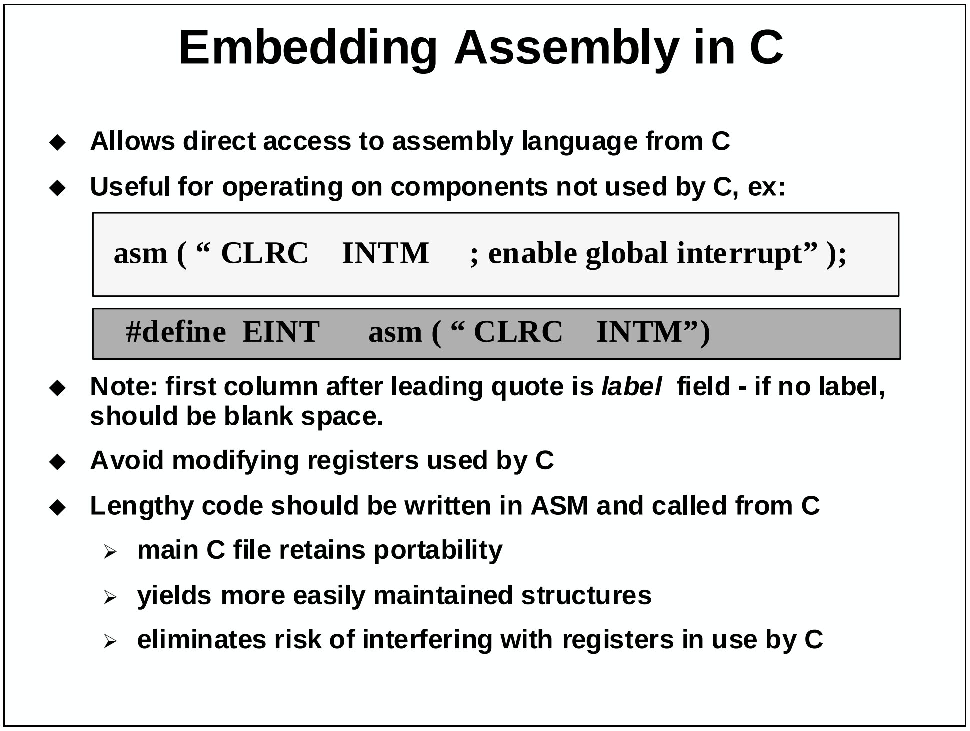
Accessing Interrupts / Status Register

Initialize via C :

```
extern cregister volatile unsigned int IFR;
extern cregister volatile unsigned int IER;
  . . .
  IER &= ~Mask;         //clear desired bits
  IER |=  Mask;         //set desired bits
  IFR  = 0x0000;        //clear prior interrupts
```

- **Interrupt Enable & Interrupt Flag Registers (IER, IFR) are not memory mapped**
- **Only limited instructions can access IER & IFR (more in interrupt chapter)**
- **The compiler provides extern variables for accessing the IER & IFR**

Using Embedded Assembly



Embedding Assembly in C

- **Allows direct access to assembly language from C**
- **Useful for operating on components not used by C, ex:**

```
asm ( “ CLRC    INTM     ; enable global interrupt” );
```

```
#define  EINT      asm ( “ CLRC    INTM”)
```

- **Note: first column after leading quote is *label* field - if no label, should be blank space.**
- **Avoid modifying registers used by C**
- **Lengthy code should be written in ASM and called from C**
  - **main C file retains portability**
  - **yields more easily maintained structures**
  - **eliminates risk of interfering with registers in use by C**

The assembly function allows for C files to contain 28x assembly code. Care should be taken not to modify registers in use by C, and to consider the label field with the assembly function. Also, any significant amounts of assembly code should be written in an assembly file and called from C.

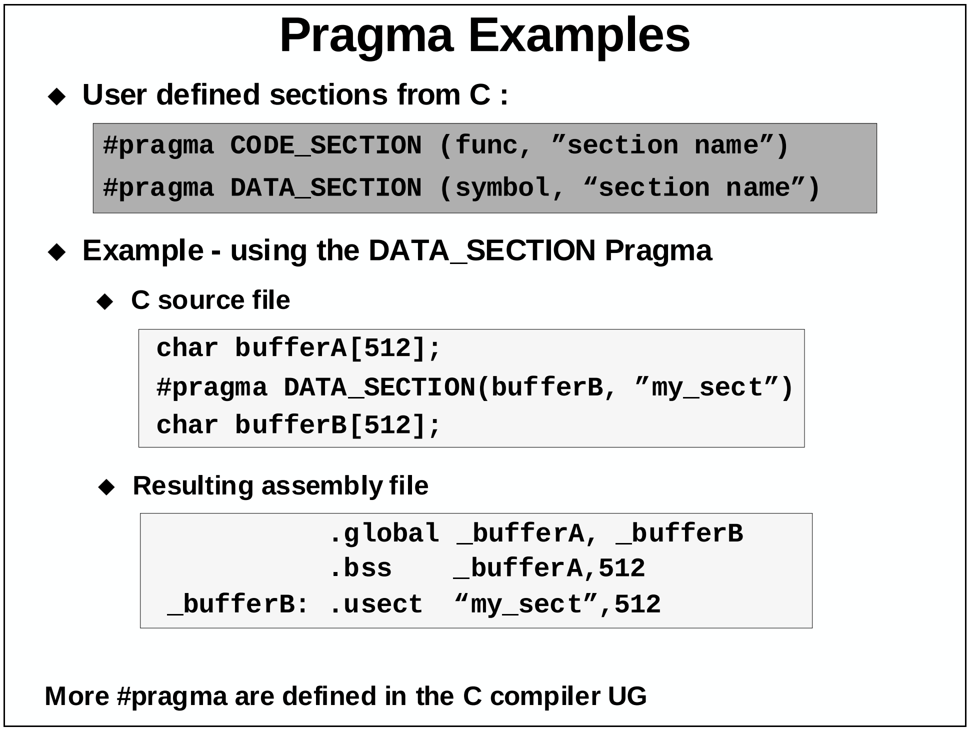
There are two examples in this slide – the first one shows how to embed a single assembly language instruction into the C code flow. The second example shows how to define a C term that will invoke the assembly language instruction.

Using Pragma

Pragma is a preprocessor directive that provides directions to the compiler about how to treat a particular statement.   The following example shows how the DATA_SECTION pragma is used to put a specific buffer into a different section of RAM than other buffers.

The example shows two buffers, bufferA and bufferB. The first buffer, bufferA is treated normally by the C compiler by placing the buffer (512 words) into the ".bss" section. The second, bufferB is specifically directed to go into the “my_sect” portion of data memory. Global variables, normally ".bss", can be redirected as desired.

When using CODE_SECTION, code that is normally linked as ".text", can be identified otherwise by using the code section pragma (like .sect in assembly).



Pragma Examples

- **User defined sections from C :**

```
#pragma CODE_SECTION (func, ”section name”)
#pragma DATA_SECTION (symbol, “section name”)
```

- **Example - using the DATA_SECTION Pragma**
  - **C source file**

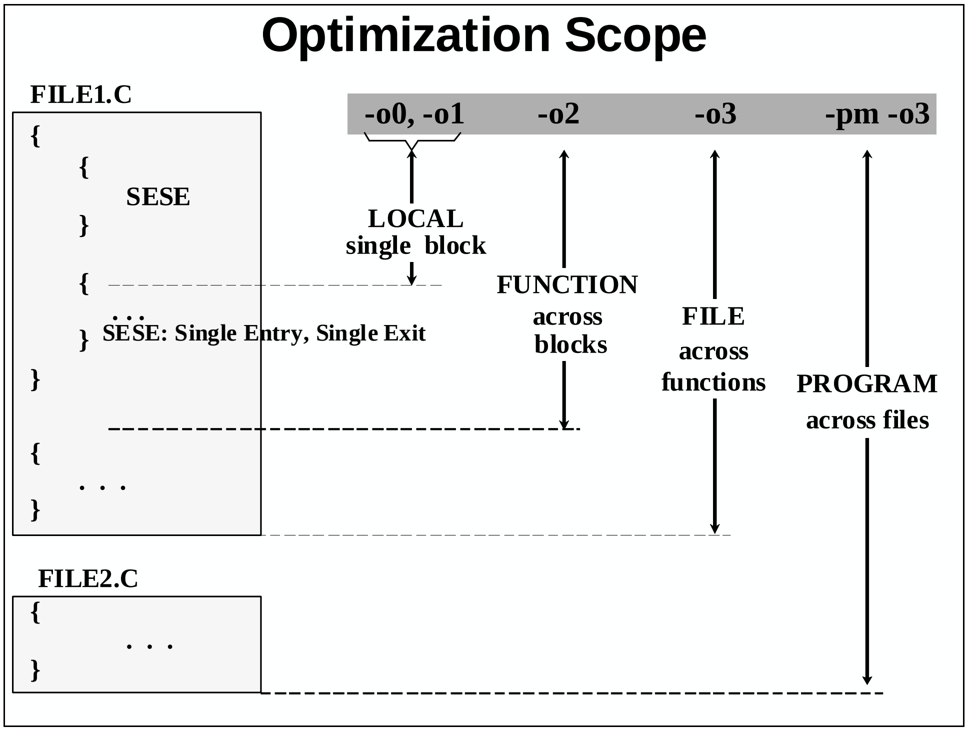
```
char bufferA[512];
#pragma DATA_SECTION(bufferB, ”my_sect”)
char bufferB[512];
```

  - **Resulting assembly file**

```
           .global _bufferA, _bufferB
           .bss    _bufferA,512
_bufferB:  .usect  “my_sect”,512
```

More #pragma are defined in the C compiler UG

Optimization Levels



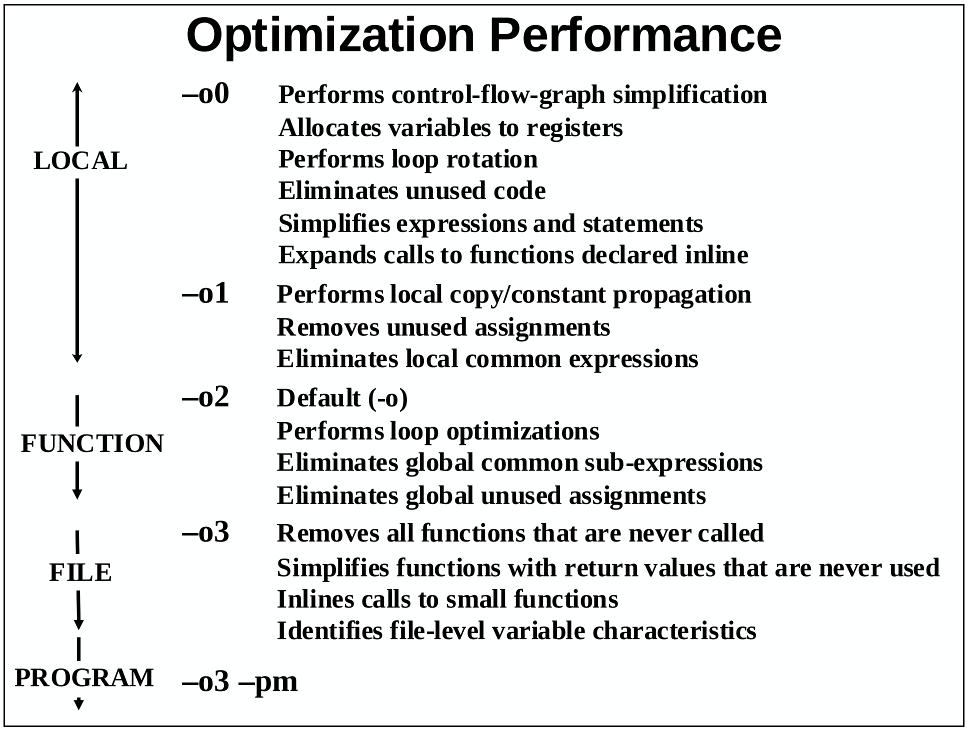
Optimizations fall into 4 categories. This is also a methodology that should be used to invoke the optimizations. It is recommended that optimization be invoked in steps, and that code be verified before advancing to the next step. Intermediate steps offer the gradual transition from fully symbolic to fully optimized compilation. Compiler switched may be invoked in a variety of ways.

Here are 4 steps that could be considered:
1st: use –g
By starting out with –g, you do no optimization at all and keep symbols for debug.

2nd: use –g –o3
The option –o3 might be too big a jump, but it adds the optimizer and keeps symbols.

3rd: use –g –o3 –mn
This is a full optimization, but keeps some symbols

4th: use –o3
Full optimization, symbols are not kept.



Optimization Performance

LOCAL

–o0 Performs control-flow-graph simplification
Allocates variables to registers
Performs loop rotation
Eliminates unused code
Simplifies expressions and statements
Expands calls to functions declared inline

–o1 Performs local copy/constant propagation
Removes unused assignments
Eliminates local common expressions

FUNCTION

–o2 Default (-o)
Performs loop optimizations
Eliminates global common sub-expressions
Eliminates global unused assignments

FILE

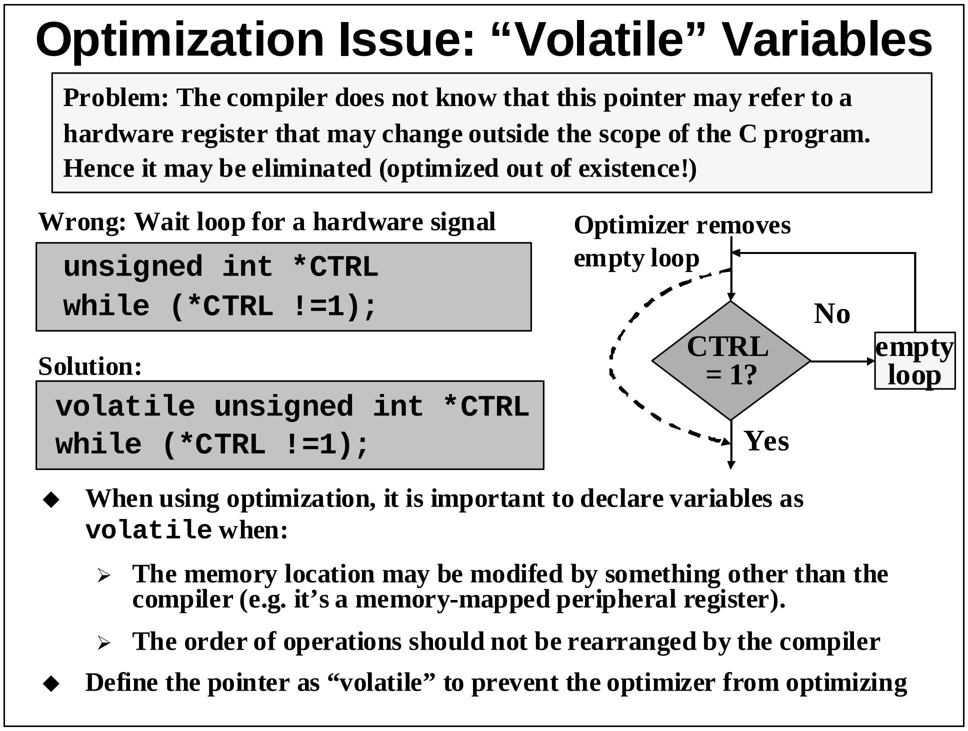
–o3 Removes all functions that are never called
Simplifies functions with return values that are never used
Inlines calls to small functions
Identifies file-level variable characteristics

PROGRAM

–o3 –pm

Optimizer levels zero through three, offer an increasing array of actions, as seen above. Higher levels include all the functions of the lower ones. Increasing optimizer levels also increase the scope of optimization, from considering the elements of single entry, single-exit functions only, through all the elements in a file. The “-pm” option directs the optimizer to view numerous input files as one large single file, so that optimization can be performed across the whole system.

Volatile Usage



Optimization Issue: “Volatile” Variables

Problem: The compiler does not know that this pointer may refer to a hardware register that may change outside the scope of the C program. Hence it may be eliminated (optimized out of existence!)

Wrong: Wait loop for a hardware signal

```
unsigned int *CTRL
while (*CTRL !=1);
```

Solution:

```
volatile unsigned int *CTRL
while (*CTRL !=1);
```

Optimizer removes empty loop
No
CTRL = 1?
empty loop
Yes

- **When using optimization, it is important to declare variables as `volatile` when:**
  - **The memory location may be modifed by something other than the compiler (e.g. it’s a memory-mapped peripheral register).**
  - **The order of operations should not be rearranged by the compiler**
- **Define the pointer as “volatile” to prevent the optimizer from optimizing**

Compiler Advanced Options

To get to these options, go to Project → Build Options in Code Composer Studio.

In the category, pick **Advanced**.

The first thing to notice under advanced options is the **Auto Inlining Threshold**.

- Used with –o3 option

- Functions > size are not auto inlined

Note: To prevent code size increases when using –o3, disable auto inlining with -oi0

The next point we will cover is the **Normal Optimization with Debug (-mn)**.

- Re-enables optimizations disabled by "–g" option (symbolic debug)

- Used for maximum optimization

Note: Some symbolic debug labels will be lost when –mn option is used.

Optimizer should be invoked incrementally:

`-g test`	Symbols kept for debug
`-g -o3 test`	Add optimizer, keep symbols
`-g -o3 -mn test`	More optimize, some symbols
`-o3 test`	Final rev: Full optimize, no symbols

[-mf] : Optimize for speed instead of the default optimization for code size

[-mi] : Avoid RPT instruction. Prevent compiler from generating RPT instruction. RPT instruction is not interruptible

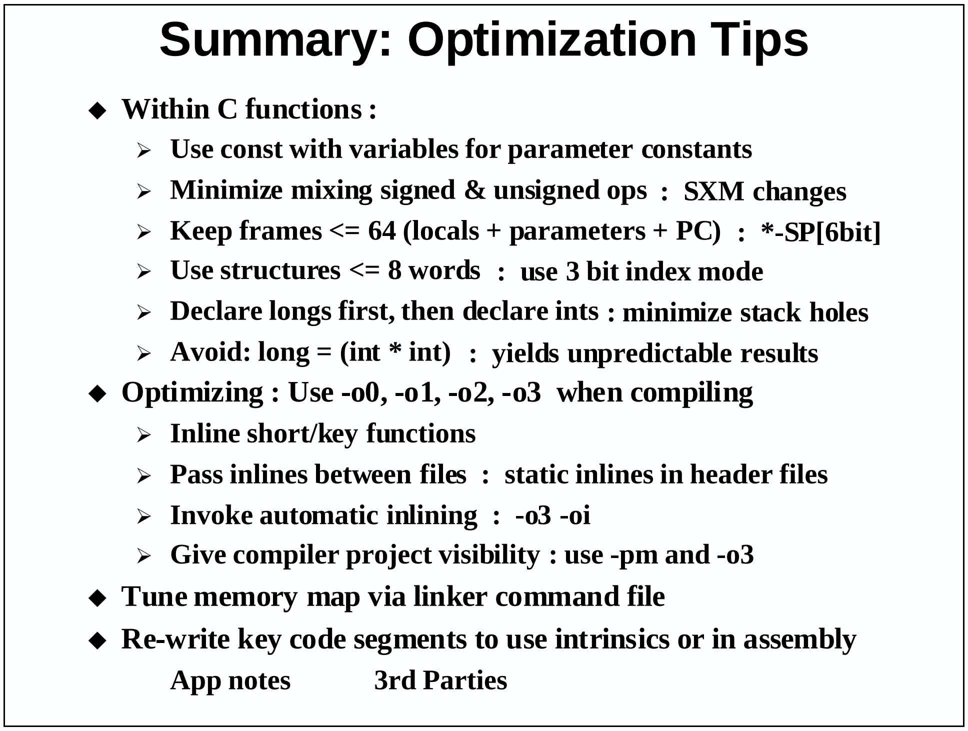
[-mt] : Unified memory model. Use this switch with the unified memory map of the 281x & 280x. Allows compiler to generate the following:

-RPT PREAD for memory copy routines or structure assignments

-MAC instructions

-Improves efficiency of switch tables

Optimization Tips Summary



Summary: Optimization Tips

- **Within C functions :**
  - **Use const with variables for parameter constants**
  - **Minimize mixing signed & unsigned ops : SXM changes**
  - **Keep frames <= 64 (locals + parameters + PC) : *-SP[6bit]**
  - **Use structures <= 8 words : use 3 bit index mode**
  - **Declare longs first, then declare ints : minimize stack holes**
  - **Avoid: long = (int * int) : yields unpredictable results**
- **Optimizing : Use -o0, -o1, -o2, -o3 when compiling**
  - **Inline short/key functions**
  - **Pass inlines between files : static inlines in header files**
  - **Invoke automatic inlining : -o3 -oi**
  - **Give compiler project visibility : use -pm and -o3**
- **Tune memory map via linker command file**
- **Re-write key code segments to use intrinsics or in assembly**

  App notes 3rd Parties

The list above documents the steps that can be taken to achieve increasingly higher coding efficiency. It is recommended that users first get their code to work with no optimization, and then add optimizations until the required performance is obtained.

Lab D: C Optimization

Note: *The lab linker command file is based on the F28035 memory map – modify as needed, if using a different F28xx device memory map.*

- **Objective**

The objective of this lab is to practice and verify the mechanics of optimizing C programs. Using Code Composer Studio profile capabilities, different routines in a project will be benchmarked. This will allow you to analyze the performance of different functions. This lab will highlight the profiler and the clock tools in CCS.

- **Procedure**

Create Project File

1. Create a new project in `C:\C28x\Labs\Appendix\LabD` called `LabD.pjt` and add `LabD.c`, `Lab.cmd`, and `sop-c.c` to it. (Note that `sop-asm.asm` will be used in the next part of the lab, and should not be added now).

2. Setup the Build Options. Select the Linker tab and notice that `“Run-time Autoinitialization”` under `“Autoinit Model:”`is selected. *Do not* enter anything in the `“Code Entry Point (-e):”` field (*leave it blank*). Set the stack size to 0x200. In the Linker options select the “Libraries” Category and enter `rts2800_ml.lib` in the `“Incl. Libraries (-l):”` box. Next, select the Compiler tab. Note that `“Full Symbolic Debug (-g)”` under `“Generate Debug Info:”` in the Basic Category is selected. On the Feedback Category pull down the interlisting options and select `“C and ASM (-ss)”`. On the Assembly Category check the `Keep generated .asm Files (-k), Keep Labels as Symbols (-as)` and `Generate Assembly Listing Files (-al)`. The –as will allow you to see symbols in the memory window and the –al will generate an assembly listing file (.lst file). The listing file has limited uses, but is sometime helpful to view opcode values and instruction sizes. (The .lst file can be viewed with the editor). Both of these options will help with debugging. Then select `OK` to save the Build Options.

Build and Load

3. Click the `“Build”` button and watch the tools run in the build window. Be sure the `“Load program after build”` option is selected in Code Composer Studio. The output file should automatically load. The Program Counter should be pointing to _c_int00 in the Disassembly Window.

Set Up the Profile Session

4. Restart the DSP `(debug → restart)` and then `“Go Main”`. This will run through the C initialization routine in `Boot.asm` and stop at the main routine in `LabD.c`.

5. Set a breakpoint on the NOP in the while(1) loop at the end of main() in `LabD.c`.

6. Set up the profile session by selecting `Profiler → Start New Session`. Enter a session name of your choice (i.e. LabD).

7. In the profiler window, hover the mouse over the icons on the left region of the window and select the icon for `Profile All Functions.` Click on the "`+`" to expand the functions. Record the "Code Size" of the function `sop` C code in the table at the end of this lab. Note: If you do not see a "+" beside the .out file, press "Profile All Functions" on the horizontal tool bar. (You can close the build window to make the profiler window easier to view by right clicking on the build window and selecting "`hide`").

8. Select `F5` or the run icon. Observe the values present in the profiling window. What do the numbers mean? Click on each tab to determine what each displays.

Benchmarking Code

9. Let's benchmark (i.e.count the cycles need by) only a portion of the code. This requires you to set a breakpoint pair on the starting and ending points of the benchmark. Open the file `sop-c.c` and set a breakpoint on the "`for`" statement and the "`return`" statement.

10. In CCS, select `Profile → Setup`. Check "`Profile all Functions and Loops for Total Cycles`" and click "`Enable Profiling`". Then select `Profile → viewer`.

11. Now "`Restart`" the program and then "`Run`" the program. The program should be stopped at the first breakpoint in `sop`. Double click on the clock window to set the clock to zero. Now you are ready to benchmark the code. "Run" to the second breakpoint. The number of cycles are displayed in the viewer window. Record this value in the table at the end of the lab under "C Code - Cycles".

C Optimization

12. To optimize C code to the highest level, we must set up new Build Options for our Project. Select the Compiler tab. In the Basic Category Panel, under "`Opt Level`" select `File (-o3)`. Then select `OK` to save the Build Options.

13. Now "`Rebuild`" the program and then "`Run`" the program. The program should be stopped at the first breakpoint in `sop`. Double click on the clock window to set the clock to zero. Now you are ready to benchmark the code. "Run" to the second breakpoint. The number of cycles are displayed in the clock window. Record this value in the table at the end of the lab under "Optimized C (-o3) - Cycles".

14. Look in your profile window at the code size of sop. Record this value in the table at the end of this lab.

Benchmarking Assembly Code

15. Remove `sop-c.c` from your project and replace it with `sop-asm.asm`. Rebuild and set breakpoints at the beginning and end of the assembly code (`MOVL & LRETR)`.

16. Start a new profile session and set it to profile all functions. Run to the first breakpoint and study the profiler window. Record the code size of the assembly code in the table.

17. Double Click on the clock to reset it. Run to the last breakpoint. Record the number of cycles the assembly code ran.

18. How does assembly, C code, and optimized C code compare on the C28x?

	C Code	Optimized C Code (-o3)	Assembly Code
Code Size			
Cycles			

End of Exercise

OPTIONAL Lab D2: C Callable Assembly

Note: *The lab linker command file is based on the F28035 memory map – modify as needed, if using a different F28xx device memory map.*

- **Objective**

The objective of this lab is to practice and verify the mechanics of implementing a C callable assembly programming. In this lab, a C file will be used to call the sum-of-products (from the previous Appendix LabC exercise) by the "main" routine. Additionally, we will learn how to use Code Composer Studio to configure the C build options and add the run-time support library to the project. As in previous labs, you may perform the lab based on this information alone, or may refer to the following procedure.

- **Procedure**

Copy Files, Create Project File

1. Create a new project in `C:\C28x\Labs\Appendix\LabD2` called `LabD2.pjt` and add `LabD2.c`, `Lab.cmd`, and `sop-c.c` to it.

2. *Do not* add `LabC.asm` to the project (copy of file from Appendix Lab C). It is only placed here for easy access. Parts of this file will be used later during this lab exercise.

3. Setup the Build Options. Select the Linker tab and notice that "`Run-time Autoinitialization`" under "`Autoinit Model:`"is selected. *Do not* enter anything in the "`Code Entry Point (-e):`" field (*leave it blank*). Set the stack size to 0x200. In the Linker options select the "Libraries" Category and enter `rts2800_ml.lib` in the "`Incl. Libraries (-l):`" box. Next, select the Compiler tab. Note that "`Full Symbolic Debug (-g)`" under "`Generate Debug Info:`" in the Basic Category is selected. On the Feedback Category pull down the interlisting options and select "`C and ASM (-ss)`". On the Assembly Category check the `Keep generated .asm Files (-k), Keep Labels as Symbols (-as)` and `Generate Assembly Listing Files (-al)`. The –as will allow you to see symbols in the memory window and the –al will generate an assembly listing file (.lst file). The listing file has limited uses, but is sometime helpful to view opcode values and instruction sizes. (The .lst file can be viewed with the editor). Both of these options will help with debugging. Then select `OK` to save the Build Options.

Build and Load

4. Click the "`Build`" button and watch the tools run in the build window. Be sure the "`Load program after build`" option is selected in Code Composer Studio. The output file should automatically load. The Program Counter should be pointing to _c_int00 in the Disassembly Window.

5. Under `Debug` on the menu bar click "`Go Main`". This will run through the C initialization routine in `Boot.asm` and stop at the main routine in `LabD2.c`.

Verify C Sum of Products Routine

6. Debug using both source and assembly (by right clicking on the window and select `Mixed Mode` or using `View → Mixed Source/ASM`).

7. Open a memory window to view result and data.

8. Single-step through the C code to verify that the C sum-of-products routine produces the results as your assembly version.

Viewing Interlisted Files and Creating Assembly File

9. Using `File → Open` view the `LabD2.asm` and `sop-c.asm` generated files. The compiler adds many items to the generated assembly file, most are not needed in the C-callable assembly file. Some of the unneeded items are .func / .endfunc. .sym, and .line.

10. Look for the _sop function that is generated by the compiler. This code is the basis for the C-callable assembly routine that is developed in this lab. Notice the comments generated by the compiler on which registers are used for passing parameters. Also, notice the C code is kept as comments in the interlisted file.

11. Create a new file (`File → New`, or clicking on the left most button on the horizontal toolbar “New”) and save it as an assembly source file with the name `sop-asm.asm`. Next copy *ONLY* the sum of products function from `LabC.asm` into this file. Add a _sop label to the function and make it visible to the linker (`.def`). Also, be sure to add a .sect directive to place this code in the “code” section. Finally, add the following instruction to the end:

```
LRETR        ; return statement
```

12. Next, we need to add code to initialize the sum-of-products parameters properly, based on the passed parameters. Add the following code to the first few lines after entering the _sop routine: (Note that the two pointers are passed in AR4 and AR5, but one needs to be placed in AR7. The loop counter is the third argument, and it is passed in the accumulator.)

```
MOVL  XAR7,XAR5     ;XAR7 points to coeff [0]

MOV   AR5,AL        ;move n from ACC to AR5 (loop counter)

SUBB  XAR5,#1       ;subtract 1 to make loop counter = n-1
```

Before beginning the MAC loop, add statements to set the sign extension mode, set the SPM to zero, and a ZAPA instruction. Use the same MAC statement as in Lab 4, but use XAR4 in place of XAR2. Make the repeat statement use the passed value of n-1 (i.e. AR5).

```
RPT   AR5           ;repeat next instruction AR5 times
```

Now we need to return the result. To return a value to the calling routine you will need to place your 32-bit value in the ACC. What register is the result currently in? Adjust your code, if necessary.

13. Save the assembly file as `sop-asm.asm`. (*Do not* name it `LabD2.asm` because the compiler has already created with that name from the original `LabD2.c` code).

Defining the Function Prototype as External

14. Note in `LabD2.c` an "extern" modifier is placed in front of the sum-of-products function prototype:

```
extern  int  sop(int*,int*,int); //sop function prototype
```

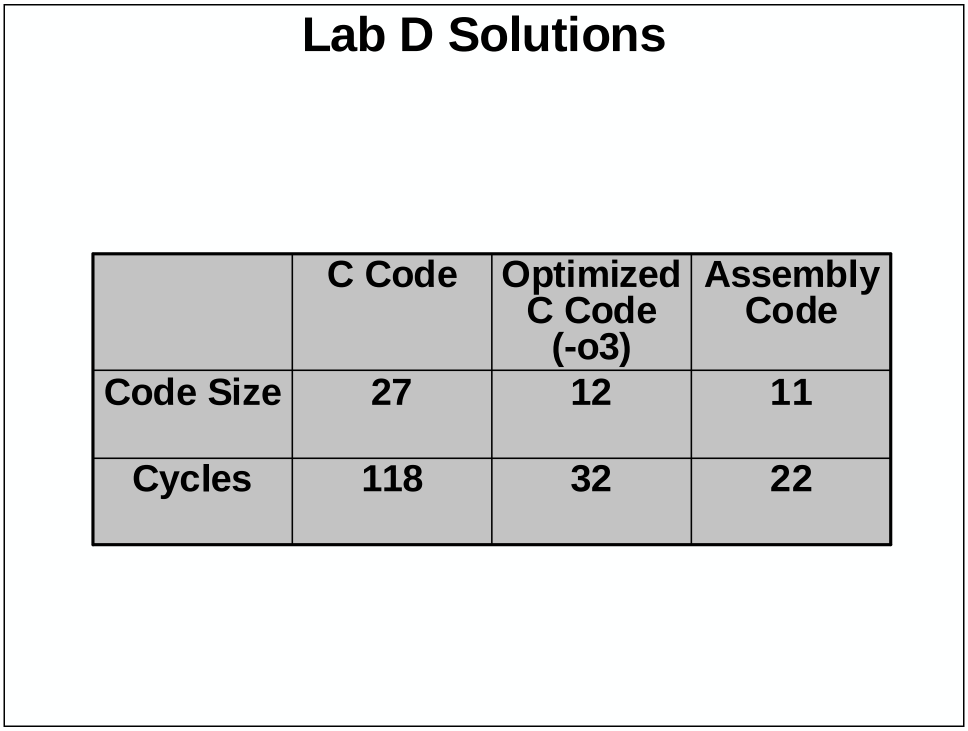
Verify Assembly Sum of Products Routine

15. Remove the `sop-c.c` file from the project and add the new `sop-asm.asm` assembly file to the project.

16. Rebuild and verify that the new assembly sum-of-products routine produces the same results as the C function.

End of Exercise

Solutions



Lab D Solutions

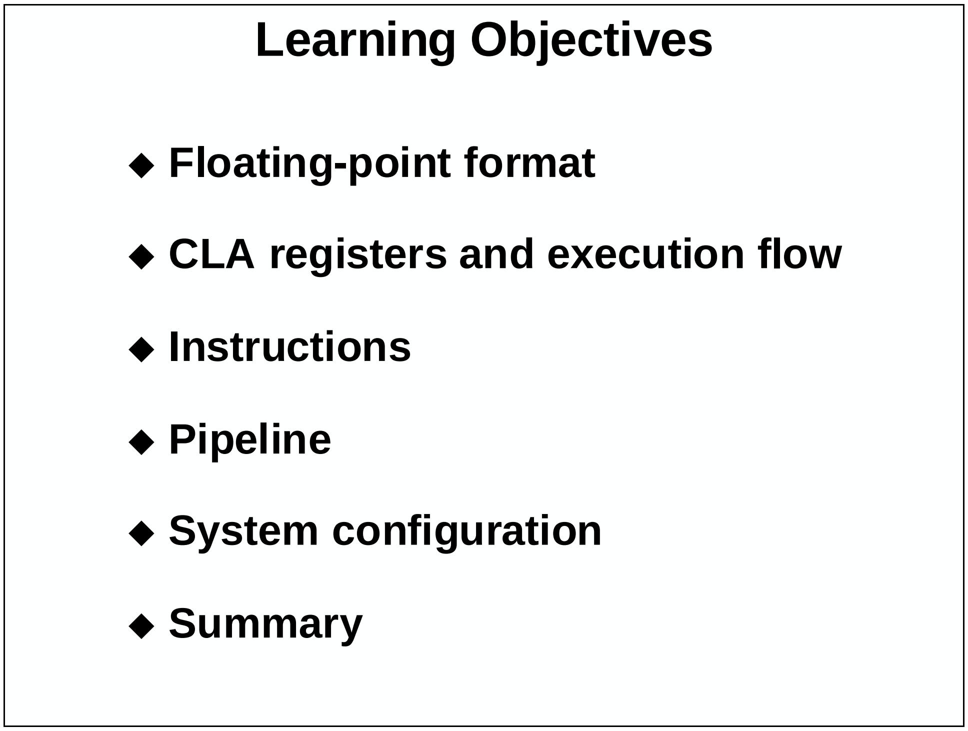
	C Code	Optimized C Code (-o3)	Assembly Code
Code Size	27	12	11
Cycles	118	32	22

Appendix E – Control Law Accelerator

Introduction

Appendix E discusses the details of the Piccolo™ TMS320F2803x Control Law Accelerator (CLA). The floating-point number format and the CLA registers will be discussed. Details of the CLA instruction set and pipeline will be explained. Additionally, system configuration and a comparison to the Delfino™ floating-point unit (FPU) will be covered.

Learning Objectives



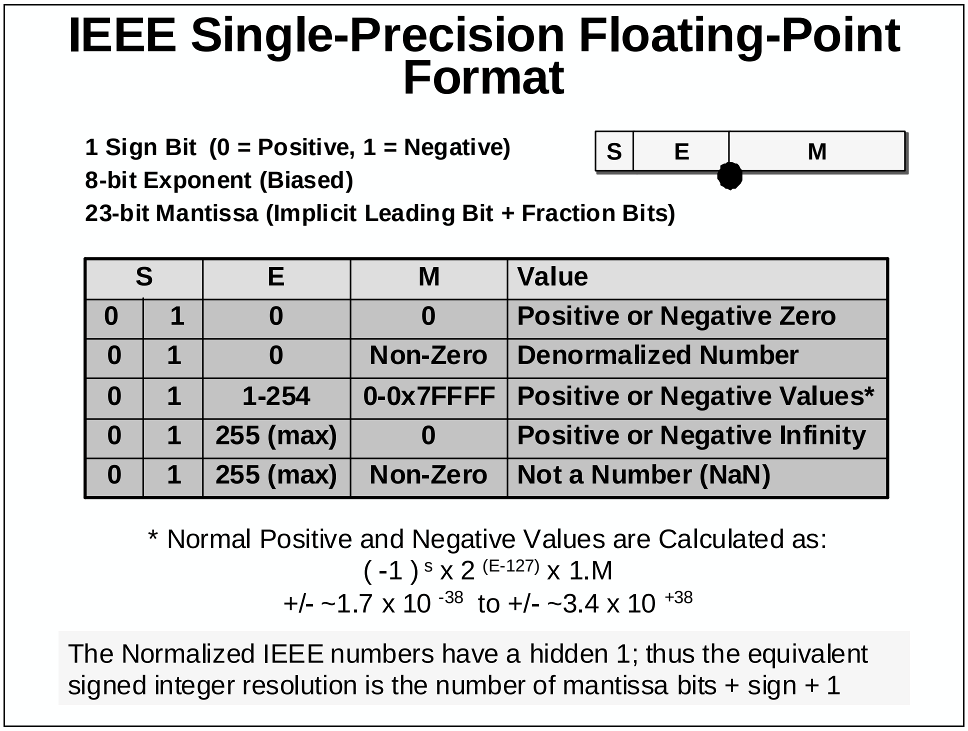
Learning Objectives

- ◆ Floating-point format
- ◆ CLA registers and execution flow
- ◆ Instructions
- ◆ Pipeline
- ◆ System configuration
- ◆ Summary

Module Topics

Control Law Accelerator

Floating-Point Format



IEEE Single-Precision Floating-Point Format

1 Sign Bit (0 = Positive, 1 = Negative)
8-bit Exponent (Biased)
23-bit Mantissa (Implicit Leading Bit + Fraction Bits)

S	E	M

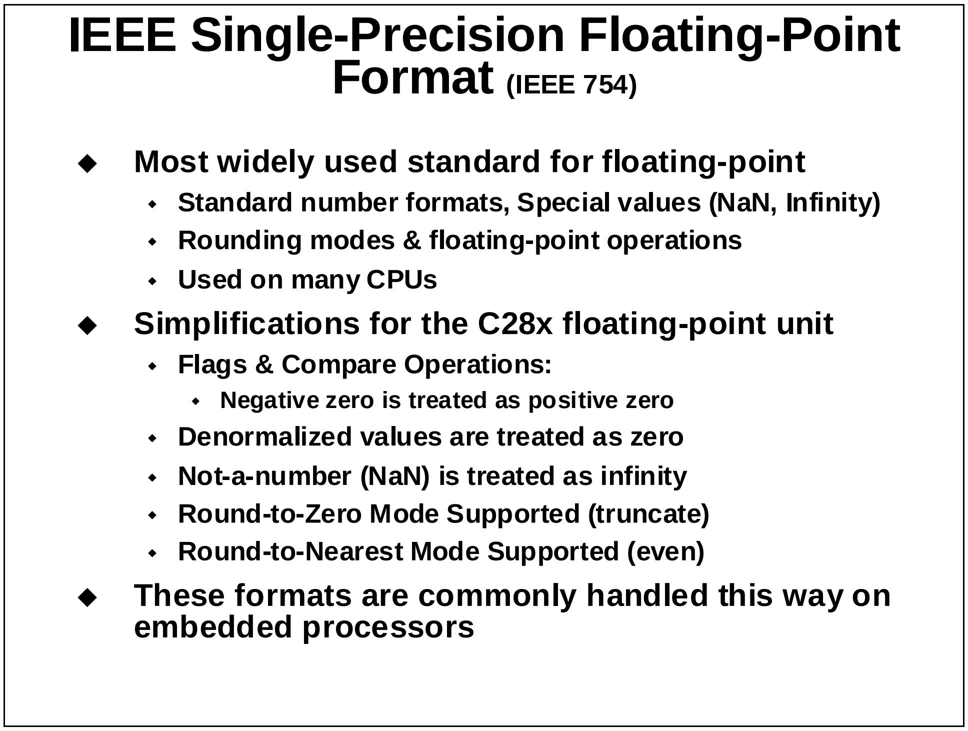
S		E	M	Value
0	1	0	0	Positive or Negative Zero
0	1	0	Non-Zero	Denormalized Number
0	1	1-254	0-0x7FFFF	Positive or Negative Values*
0	1	255 (max)	0	Positive or Negative Infinity
0	1	255 (max)	Non-Zero	Not a Number (NaN)

* Normal Positive and Negative Values are Calculated as:

$$(-1)^{s} \times 2^{(E-127)} \times 1.M$$

$$+/- \sim 1.7 \times 10^{-38} \text{ to } +/- \sim 3.4 \times 10^{+38}$$

The Normalized IEEE numbers have a hidden 1; thus the equivalent signed integer resolution is the number of mantissa bits + sign + 1



IEEE Single-Precision Floating-Point Format (IEEE 754)

- **Most widely used standard for floating-point**
  - **Standard number formats, Special values (NaN, Infinity)**
  - **Rounding modes & floating-point operations**
  - **Used on many CPUs**
- **Simplifications for the C28x floating-point unit**
  - **Flags & Compare Operations:**
    - **Negative zero is treated as positive zero**
  - **Denormalized values are treated as zero**
  - **Not-a-number (NaN) is treated as infinity**
  - **Round-to-Zero Mode Supported (truncate)**
  - **Round-to-Nearest Mode Supported (even)**
- **These formats are commonly handled this way on embedded processors**

CLA Registers and Execution Flow

CLA Register Set

Interrupt/Task Control

MIFR:	Flag
MICLR:	Clear
MIFRC:	Force
MIOVF:	Overflow flag
MICLROVF:	Overflow clear

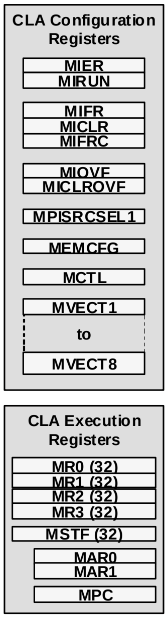
Configuration and Control

MEMCFG:	Memory config
MCTL:	CLA control

Eight Interrupt (Task) Vectors
MVECT1 to MVECT8
Offset from the start of CLA Program Memory to the beginning of the task

Four 32-bit Result Registers
MR0 – MR3

Two 16-bit Auxiliary Registers
MAR0, MAR1
Used for indirect addressing



CLA Configuration Registers:
CSM and EALLOW Protected
Main CPU has Read and Write Access

MIER: Interrupt enable/disable
MIRUN: Which task is running

Interrupt/Task Source Selection
MPISRCSEL1:
Task1: ADCINT1 or EPWM1_INT
Task2: ADCINT2 or EPWM2_INT
....
Task7: ADCINT7 or EPWM7_INT
Task8: ADCINT8 or CPU Timer 0

CLA Execution Registers:
CSM Protected
Main CPU has Read Only Access

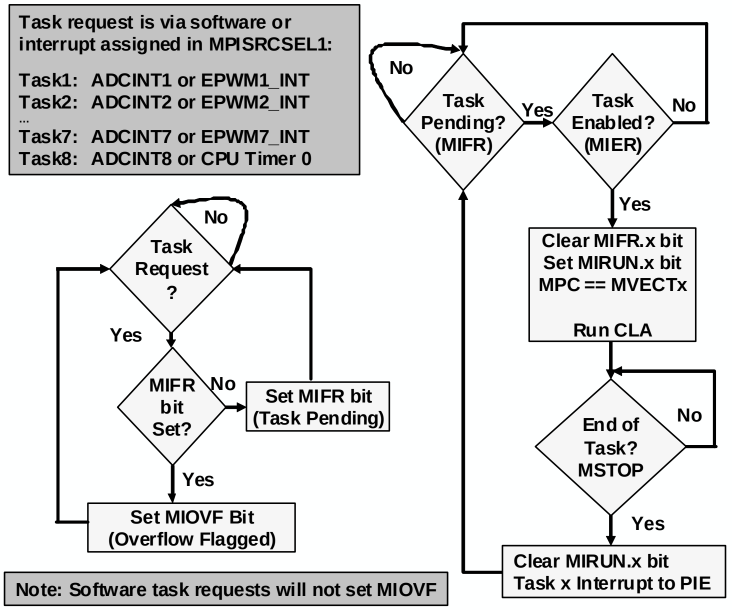
MSTF: Status Register
Zero, negative, overflow, underflow
Rounding mode
RPC: Return PC
MEALLOW

MPC: 12-bit Program Counter
Offset from the start of CLA program memory
Indicates instruction in the D2 phase

CLA Execution Flow

Task request is via software or interrupt assigned in MPISRCSEL1:

Task1: ADCINT1 or EPWM1_INT
Task2: ADCINT2 or EPWM2_INT
...
Task7: ADCINT7 or EPWM7_INT
Task8: ADCINT8 or CPU Timer 0



Note: Software task requests will not set MIOVF

x = Highest priority task both enabled and pending

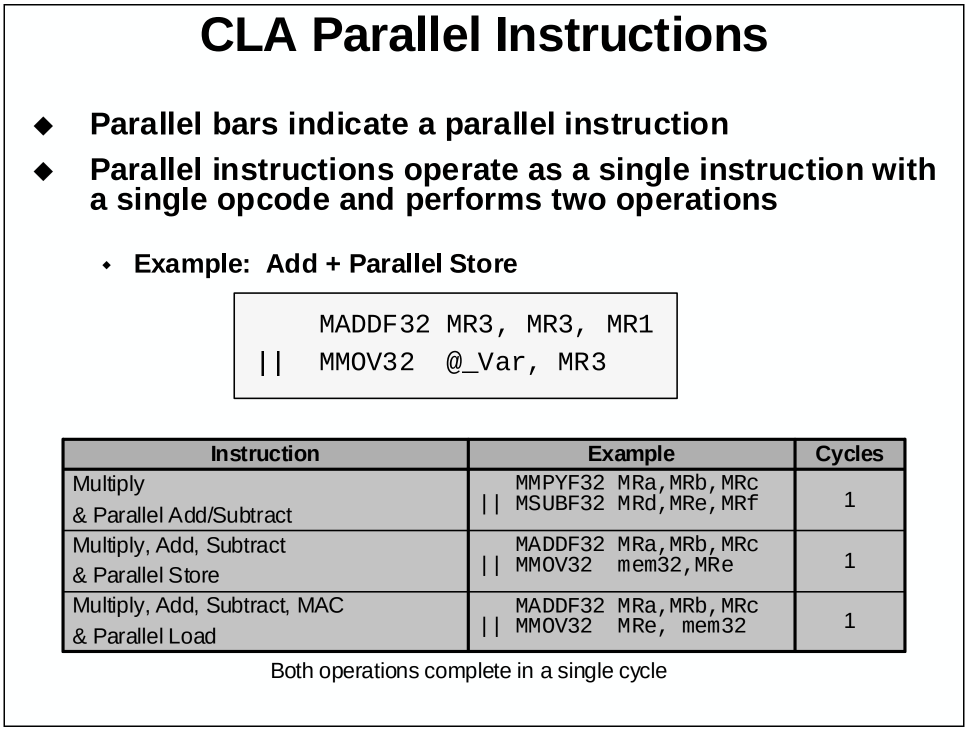
	Priority
Task1:	Highest
...	
Task8:	Lowest

The task runs to completion (No task nesting)

The main CPU continues code execution in parallel with the CLA

When a task completes a task-specific interrupt is sent to the PIE

CLA Instructions



CLA Parallel Instructions

- Parallel bars indicate a parallel instruction
- Parallel instructions operate as a single instruction with a single opcode and performs two operations
  - Example: Add + Parallel Store

```
    MADDF32 MR3, MR3, MR1
||  MMOV32  @_Var, MR3
```

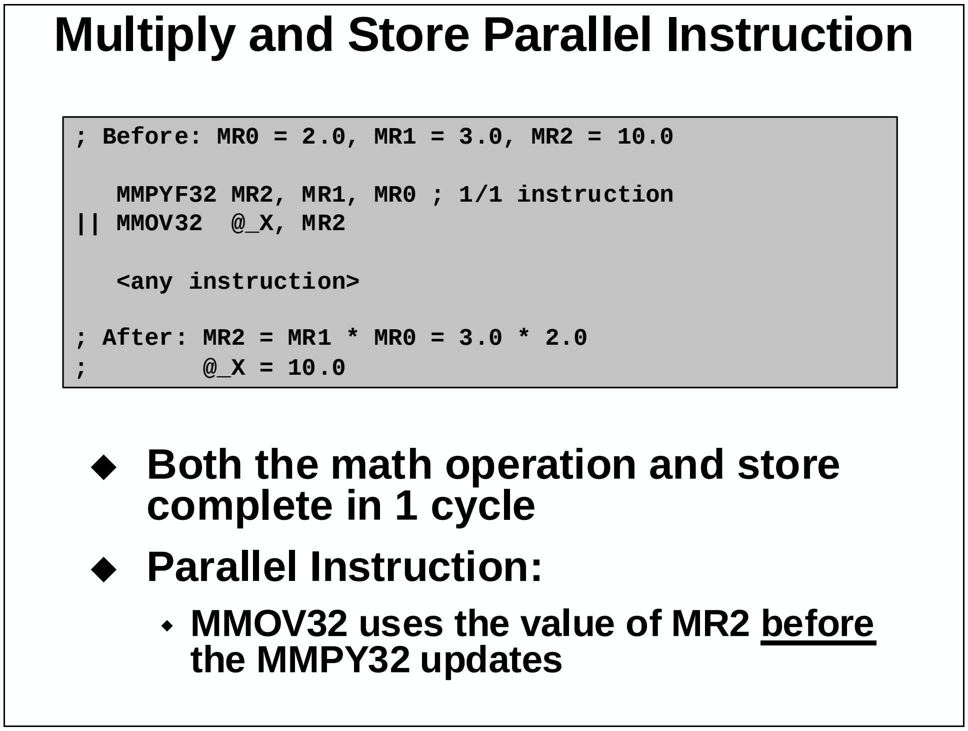
Instruction	Example	Cycles

Multiply
& Parallel Add/Subtract
	`MMPYF32 MRa,MRb,MRc`
	`|| MSUBF32 MRd,MRe,MRf`
		1

Multiply, Add, Subtract
& Parallel Store
	`MADDF32 MRa,MRb,MRc`
	`|| MMOV32  mem32,MRe`
		1

Multiply, Add, Subtract, MAC
& Parallel Load
	`MADDF32 MRa,MRb,MRc`
	`|| MMOV32  MRe, mem32`
		1

Both operations complete in a single cycle

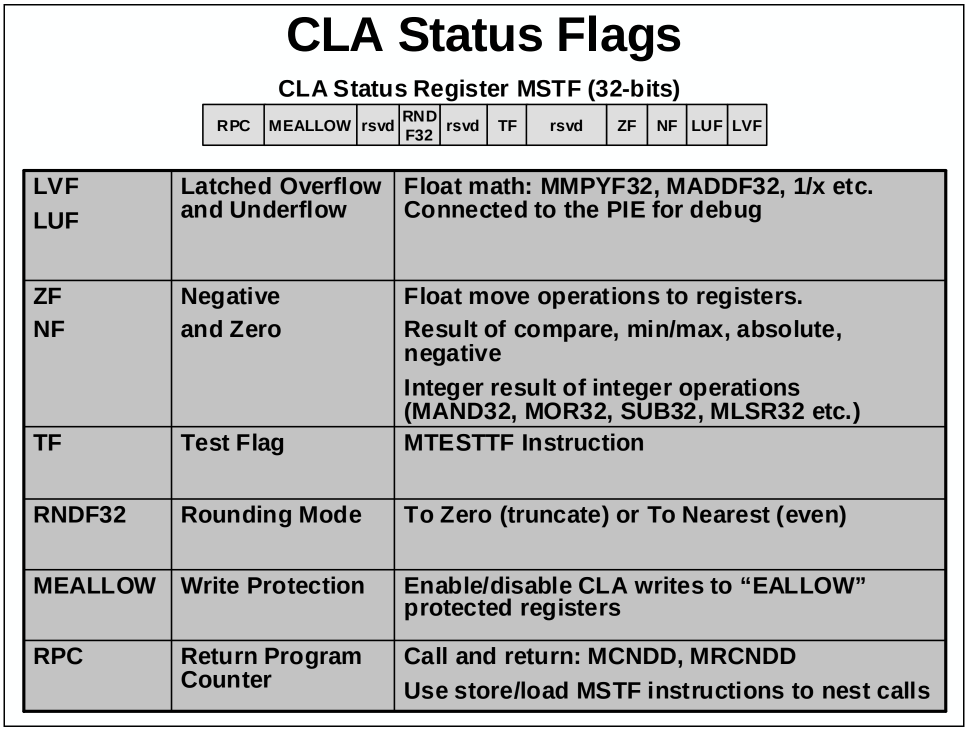


Multiply and Store Parallel Instruction

```
; Before: MR0 = 2.0, MR1 = 3.0, MR2 = 10.0

   MMPYF32 MR2, MR1, MR0 ; 1/1 instruction
|| MMOV32  @_X, MR2

   <any instruction>

; After: MR2 = MR1 * MR0 = 3.0 * 2.0
;        @_X = 10.0
```

- Both the math operation and store complete in 1 cycle
- Parallel Instruction:
  - MMOV32 uses the value of MR2 before the MMPY32 updates

Status Register and Pipeline

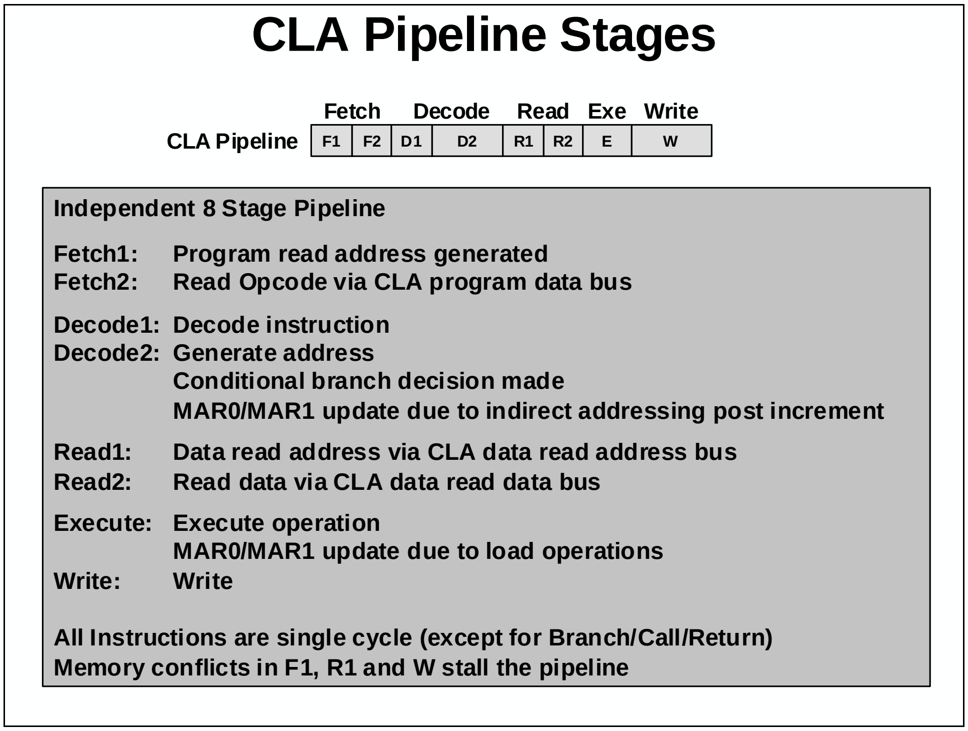


CLA Status Flags

CLA Status Register MSTF (32-bits)

RPC	MEALLOW	rsvd	RND F32	rsvd	TF	rsvd	ZF	NF	LUF	LVF

LVF **LUF**	**Latched Overflow and Underflow**	**Float math: MMPYF32, MADDF32, 1/x etc.** **Connected to the PIE for debug**
ZF **NF**	**Negative** **and Zero**	**Float move operations to registers.** **Result of compare, min/max, absolute, negative** **Integer result of integer operations (MAND32, MOR32, SUB32, MLSR32 etc.)**
TF	**Test Flag**	**MTESTTF Instruction**
RNDF32	**Rounding Mode**	**To Zero (truncate) or To Nearest (even)**
MEALLOW	**Write Protection**	**Enable/disable CLA writes to "EALLOW" protected registers**
RPC	**Return Program Counter**	**Call and return: MCNDD, MRCNDD** **Use store/load MSTF instructions to nest calls**



CLA Pipeline Stages

	Fetch		Decode		Read		Exe	Write
CLA Pipeline	F1	F2	D1	D2	R1	R2	E	W

Independent 8 Stage Pipeline

Fetch1: Program read address generated
Fetch2: Read Opcode via CLA program data bus

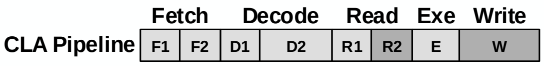
Decode1: Decode instruction
Decode2: Generate address
Conditional branch decision made
MAR0/MAR1 update due to indirect addressing post increment

Read1: Data read address via CLA data read address bus
Read2: Read data via CLA data read data bus

Execute: Execute operation
MAR0/MAR1 update due to load operations
Write: Write

All Instructions are single cycle (except for Branch/Call/Return)
Memory conflicts in F1, R1 and W stall the pipeline

Write Followed-by-Read



	Fetch		Decode		Read		Exe	Write
CLA Pipeline	F1	F2	D1	D2	R1	R2	E	W

```
MMOV32  @_Reg1, MR3          ; Write Reg1
MMOV32  MR0, @_Reg2          ; Read Reg2
```

Due to the pipeline order, the read of Reg2 occurs before the Reg1 write

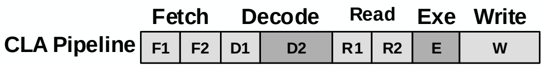
This is only an issue if the location written to can affect the location read
Some peripheral registers
Write to followed by read from the same location

Insert 3 other instructions or MNOPs to allow the write to occur first

Note: This behavior is different for the main C28 CPU:

The C28x CPU protects write followed by read to the same location
Blocks of peripheral registers have write-followed-by read protection

Loading MAR0 and MAR1



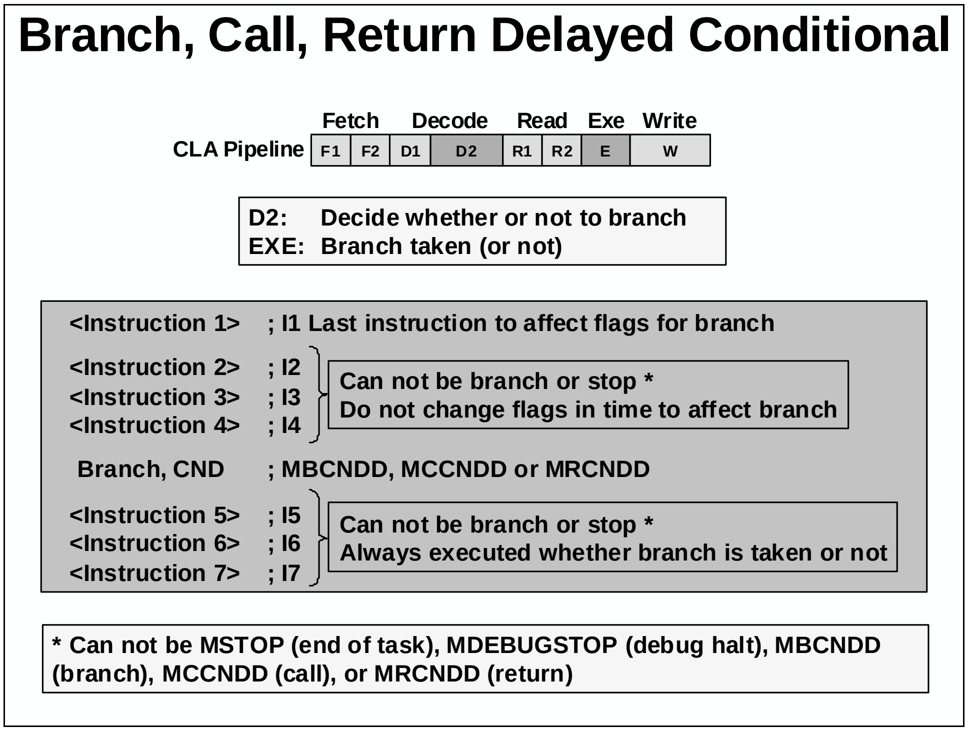
	Fetch		Decode		Read		Exe	Write
CLA Pipeline	F1	F2	D1	D2	R1	R2	E	W

D2: Update to MAR0/MAR1 due to indirect addressing post increment
EXE: Update to MAR0/MAR1 due to load operation

Assume MAR0 is 50 and #_X is 20

```
MMOV16 MAR0, #_X             ; I1 Load MAR0 with 20

MMOV32 MAR1, *MAR0[0]++      ; I2 Uses old MAR0 Value (50)
MMOV32 MAR1, *MAR0[0]++      ; I3 Uses old MAR0 Value (50)

<Instruction 4>              ; I4 Can not use MAR0

MMOV32 MAR1, *MAR0[0]++      ; I5 Uses new MAR0 Value (20)
```

When instruction I1 is in EXE instruction I4 is in D2
If I4 uses MAR0, then a conflict will occur and MAR0 will not be loaded



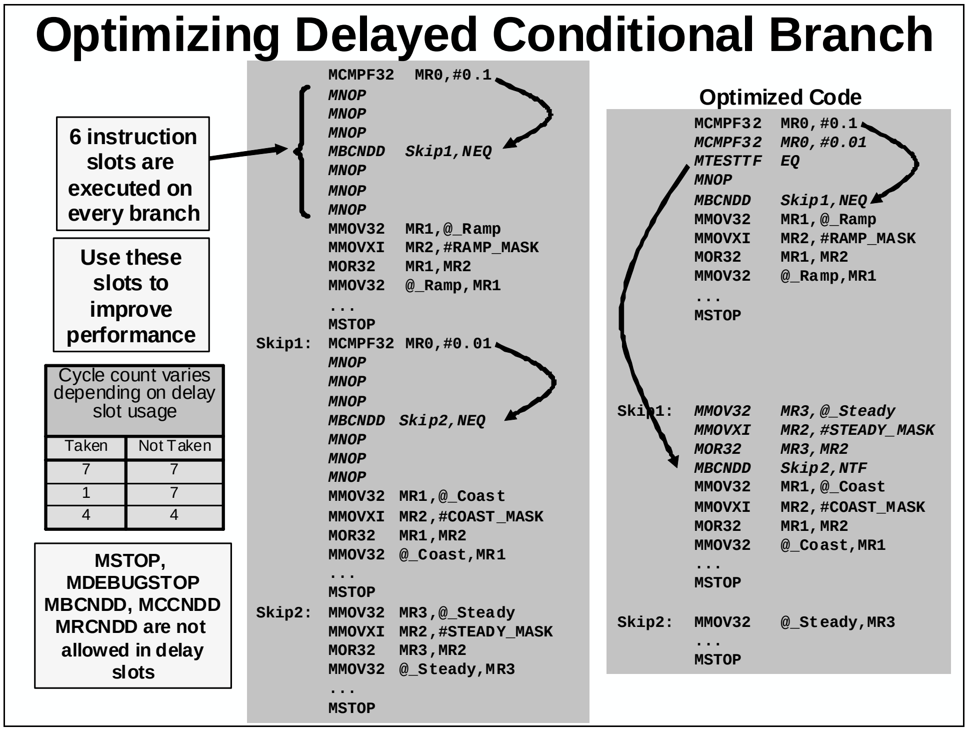
Branch, Call, Return Delayed Conditional

	Fetch		Decode		Read		Exe	Write
CLA Pipeline	F1	F2	D1	D2	R1	R2	E	W

D2: Decide whether or not to branch
EXE: Branch taken (or not)

```
<Instruction 1>   ; I1 Last instruction to affect flags for branch

<Instruction 2>   ; I2  } Can not be branch or stop *
<Instruction 3>   ; I3  } Do not change flags in time to affect branch
<Instruction 4>   ; I4  }

 Branch, CND      ; MBCNDD, MCCNDD or MRCNDD

<Instruction 5>   ; I5  } Can not be branch or stop *
<Instruction 6>   ; I6  } Always executed whether branch is taken or not
<Instruction 7>   ; I7  }
```

* Can not be MSTOP (end of task), MDEBUGSTOP (debug halt), MBCNDD (branch), MCCNDD (call), or MRCNDD (return)



Optimizing Delayed Conditional Branch

6 instruction slots are executed on every branch

Use these slots to improve performance

Cycle count varies depending on delay slot usage	
Taken	Not Taken
7	7
1	7
4	4

MSTOP, MDEBUGSTOP MBCNDD, MCCNDD MRCNDD are not allowed in delay slots

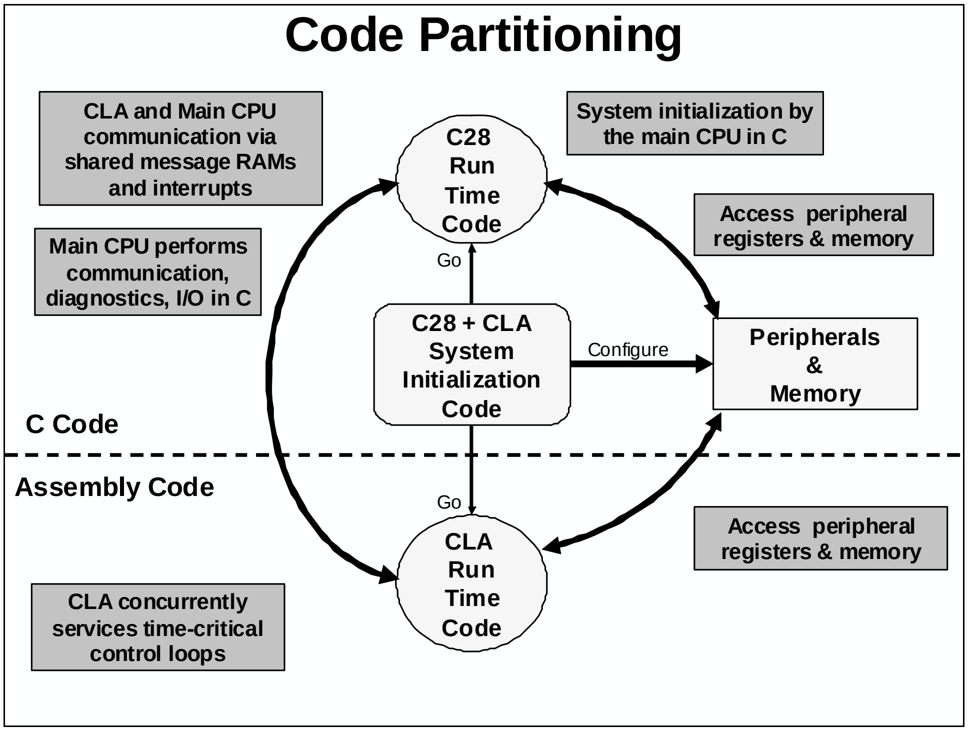
```
        MCMPF32  MR0,#0.1
        MNOP
        MNOP
        MNOP
        MBCNDD   Skip1,NEQ
        MNOP
        MNOP
        MNOP
        MMOV32   MR1,@_Ramp
        MMOVXI   MR2,#RAMP_MASK
        MOR32    MR1,MR2
        MMOV32   @_Ramp,MR1
        ...
        MSTOP
Skip1:  MCMPF32 MR0,#0.01
        MNOP
        MNOP
        MNOP
        MBCNDD  Skip2,NEQ
        MNOP
        MNOP
        MNOP
        MMOV32  MR1,@_Coast
        MMOVXI  MR2,#COAST_MASK
        MOR32   MR1,MR2
        MMOV32  @_Coast,MR1
        ...
        MSTOP
Skip2:  MMOV32  MR3,@_Steady
        MMOVXI  MR2,#STEADY_MASK
        MOR32   MR3,MR2
        MMOV32  @_Steady,MR3
        ...
        MSTOP
```

Optimized Code

```
        MCMPF32   MR0,#0.1
        MCMPF32   MR0,#0.01
        MTESTTF   EQ
        MNOP
        MBCNDD    Skip1,NEQ
        MMOV32    MR1,@_Ramp
        MMOVXI    MR2,#RAMP_MASK
        MOR32     MR1,MR2
        MMOV32    @_Ramp,MR1
        ...
        MSTOP

Skip1:  MMOV32    MR3,@_Steady
        MMOVXI    MR2,#STEADY_MASK
        MOR32     MR3,MR2
        MBCNDD    Skip2,NTF
        MMOV32    MR1,@_Coast
        MMOVXI    MR2,#COAST_MASK
        MOR32     MR1,MR2
        MMOV32    @_Coast,MR1
        ...
        MSTOP

Skip2:  MMOV32    @_Steady,MR3
        ...
        MSTOP
```

CLA System Configuration

Code Partitioning



CLA and Main CPU communication via shared message RAMs and interrupts

Main CPU performs communication, diagnostics, I/O in C

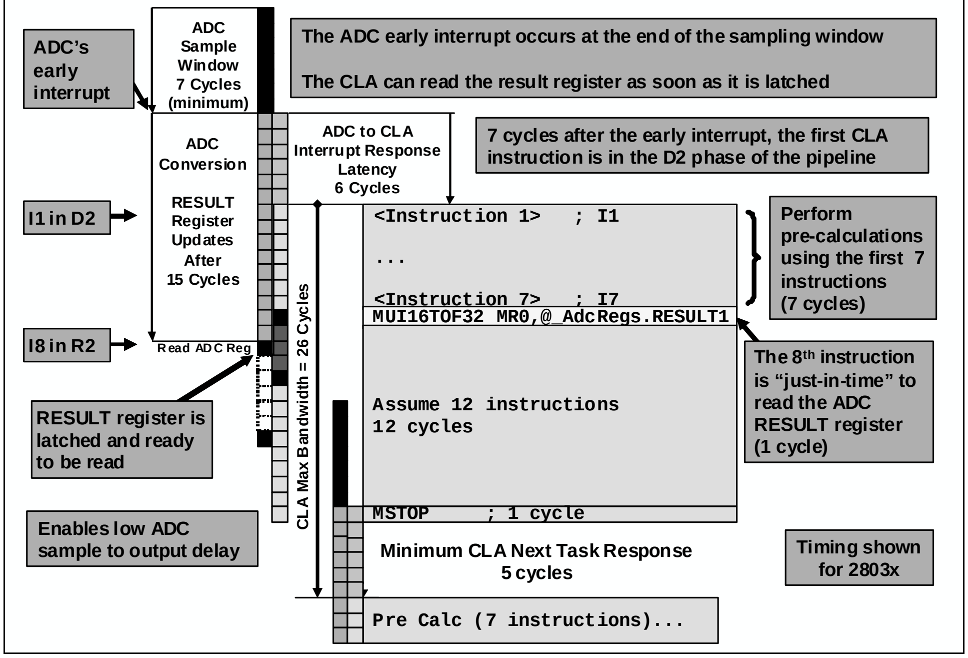
System initialization by the main CPU in C

Access peripheral registers & memory

CLA concurrently services time-critical control loops

Access peripheral registers & memory

"Just in Time" ADC Sampling Using CLA



The ADC early interrupt occurs at the end of the sampling window

The CLA can read the result register as soon as it is latched

7 cycles after the early interrupt, the first CLA instruction is in the D2 phase of the pipeline

Perform pre-calculations using the first 7 instructions (7 cycles)

The 8th instruction is "just-in-time" to read the ADC RESULT register (1 cycle)

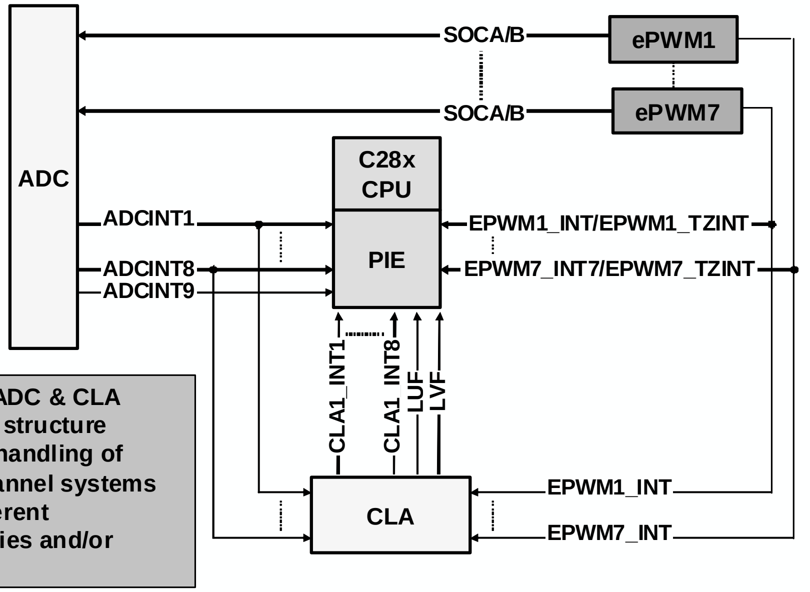
RESULT register is latched and ready to be read

Enables low ADC sample to output delay

Timing shown for 2803x

```
<Instruction 1>   ; I1
...
<Instruction 7>   ; I7
MUI16TOF32 MR0,@_AdcRegs.RESULT1
Assume 12 instructions
12 cycles
MSTOP     ; 1 cycle
Pre Calc (7 instructions)...
```

CLA Interrupts Improved Control Loop Timing



Piccolo ADC & CLA interrupt structure enables handling of multi-channel systems with different frequencies and/or phases

Anatomy of CLA Code

Using a shared C-code header file approach provides easy access to variables and constants in both C28x C and CLA assembly

```
// File: C28x_Project.h

#include “DSP2803x_Device.h”
#include “DSP2803x_Examples.h”
```

```
// File: CLAShared.h

#include “DSP28x_Project.h”
#define PERIOD 100.0
struct PI_CTRL
{
   float  KP;
   float  KI;
   float  I;
   float  Ref;
}
extern struct PI_CTRL PIVars;

extern Uint32 Cla1Prog_Start;
extern Uint32 Cla1Task1;
extern Uint32 Cla1Task2;
etc …
```

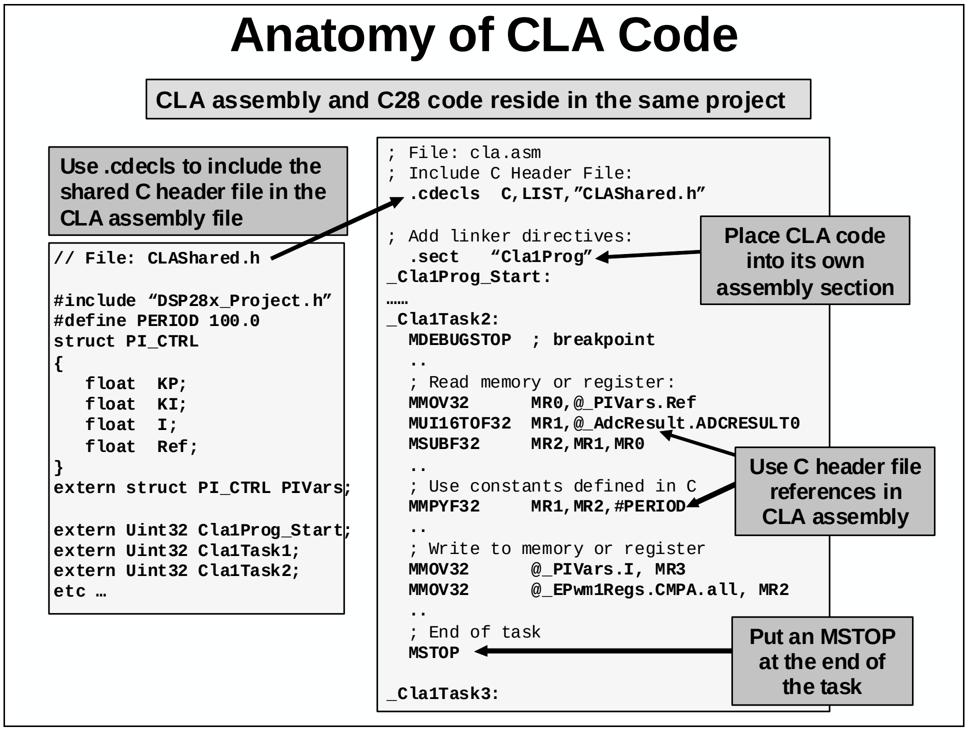
Add symbols defined in CLA assembly to make them global and usable in C

Declare shared constants and variables in C

Include DSP2803x_Device.h to define register bit-field structures

Assign variables to message RAMs or CLA data memory sections using DATA_SECTION pragma

```
// File main.c
#include “CLAShared.h”

#pragma DATA_SECTION(PIVars,"CpuToCla1MsgRAM");
struct PI_CTRL PIVars;
..
// Use Symbols defined in the CLA asm file
 Cla1Regs.MVECT1 = (Uint16) (&Cla1Task1 \
       - &Cla1Prog_Start)*sizeof(Uint32);

// Initialize variables
  PIVars.KP  = 1.234;
  PIVars.KI  = 0.92367;
  PIVars.Ref = 2048.0;
  PIVars.I   = PIVars.KP*PIVars.Ref;
..
// Initialize Peripherals:
Epwm3Regs.PRD = (Uint16) PERIOD;
```



Anatomy of CLA Code

CLA assembly and C28 code reside in the same project

Use .cdecls to include the shared C header file in the CLA assembly file

```
// File: CLAShared.h

#include “DSP28x_Project.h”
#define PERIOD 100.0
struct PI_CTRL
{
   float  KP;
   float  KI;
   float  I;
   float  Ref;
}
extern struct PI_CTRL PIVars;

extern Uint32 Cla1Prog_Start;
extern Uint32 Cla1Task1;
extern Uint32 Cla1Task2;
etc …
```

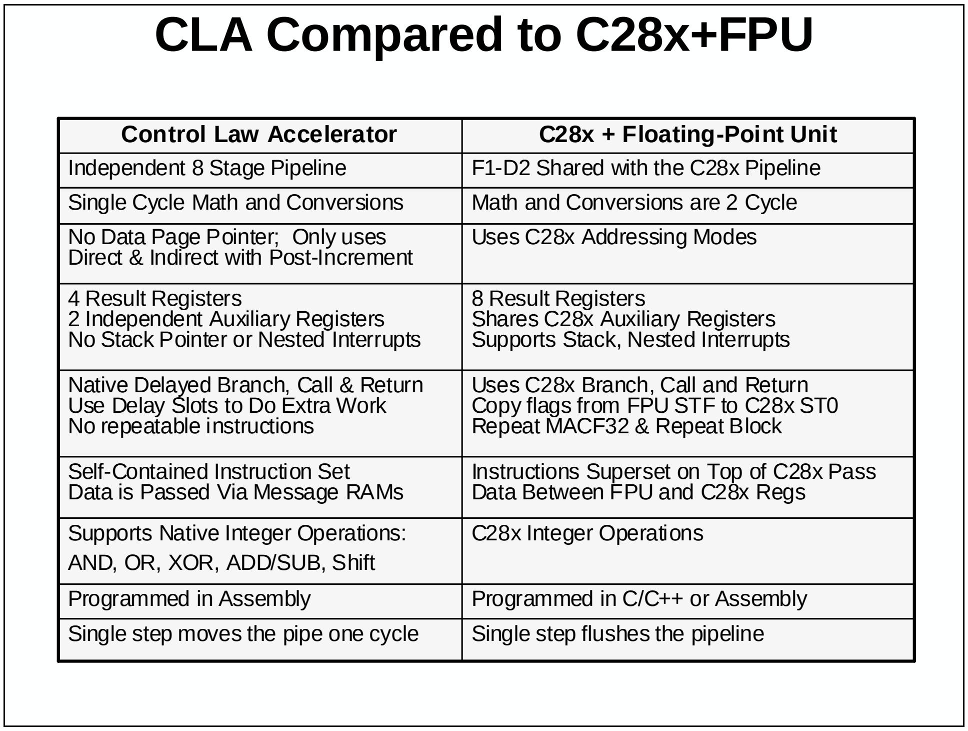
```
; File: cla.asm
; Include C Header File:
  .cdecls  C,LIST,”CLAShared.h”

; Add linker directives:
  .sect   “Cla1Prog”
_Cla1Prog_Start:
……
_Cla1Task2:
  MDEBUGSTOP  ; breakpoint
  ..
  ; Read memory or register:
  MMOV32      MR0,@_PIVars.Ref
  MUI16TOF32  MR1,@_AdcResult.ADCRESULT0
  MSUBF32     MR2,MR1,MR0
  ..
  ; Use constants defined in C
  MMPYF32     MR1,MR2,#PERIOD
  ..
  ; Write to memory or register
  MMOV32      @_PIVars.I, MR3
  MMOV32      @_EPwm1Regs.CMPA.all, MR2
  ..
  ; End of task
  MSTOP

_Cla1Task3:
```

Place CLA code into its own assembly section

Use C header file references in CLA assembly

Put an MSTOP at the end of the task

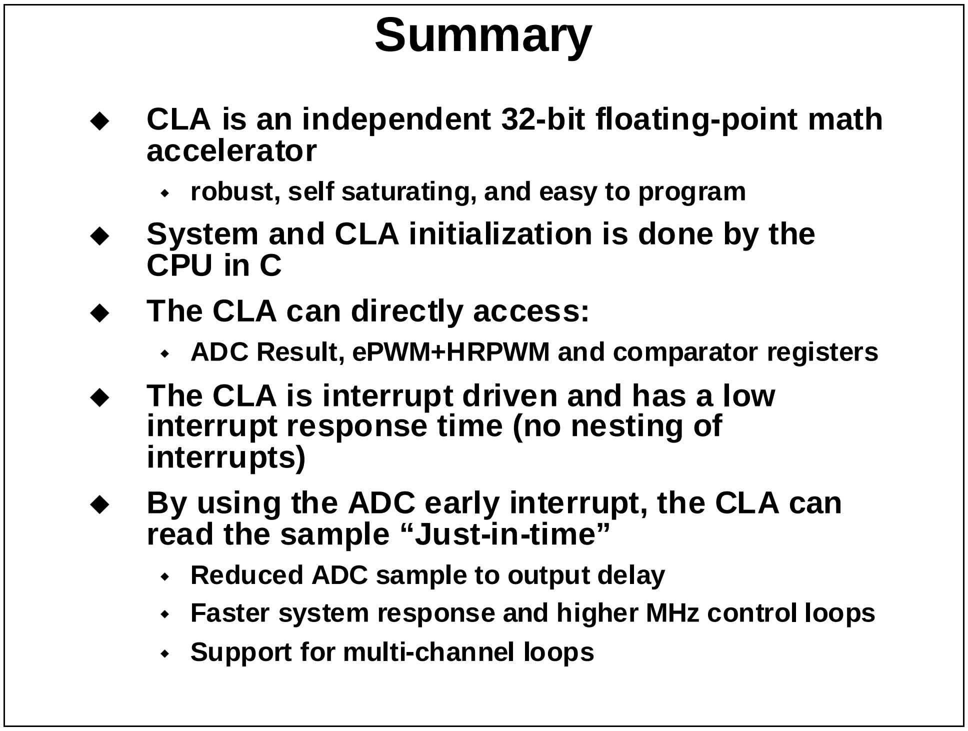
CLA Compared to C28x+FPU



CLA Compared to C28x+FPU

Control Law Accelerator	C28x + Floating-Point Unit
Independent 8 Stage Pipeline	F1-D2 Shared with the C28x Pipeline
Single Cycle Math and Conversions	Math and Conversions are 2 Cycle
No Data Page Pointer; Only uses Direct & Indirect with Post-Increment	Uses C28x Addressing Modes
4 Result Registers 2 Independent Auxiliary Registers No Stack Pointer or Nested Interrupts	8 Result Registers Shares C28x Auxiliary Registers Supports Stack, Nested Interrupts
Native Delayed Branch, Call & Return Use Delay Slots to Do Extra Work No repeatable instructions	Uses C28x Branch, Call and Return Copy flags from FPU STF to C28x ST0 Repeat MACF32 & Repeat Block
Self-Contained Instruction Set Data is Passed Via Message RAMs	Instructions Superset on Top of C28x Pass Data Between FPU and C28x Regs
Supports Native Integer Operations: AND, OR, XOR, ADD/SUB, Shift	C28x Integer Operations
Programmed in Assembly	Programmed in C/C++ or Assembly
Single step moves the pipe one cycle	Single step flushes the pipeline

Summary



Summary

- CLA is an independent 32-bit floating-point math accelerator
  - robust, self saturating, and easy to program
- System and CLA initialization is done by the CPU in C
- The CLA can directly access:
  - ADC Result, ePWM+HRPWM and comparator registers
- The CLA is interrupt driven and has a low interrupt response time (no nesting of interrupts)
- By using the ADC early interrupt, the CLA can read the sample “Just-in-time”
  - Reduced ADC sample to output delay
  - Faster system response and higher MHz control loops
  - Support for multi-channel loops

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

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

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Communications and Telecom	www.ti.com/communications
DSP	dsp.ti.com	Computers and Peripherals	www.ti.com/computers
Clocks and Timers	www.ti.com/clocks	Consumer Electronics	www.ti.com/consumer-apps
Interface	interface.ti.com	Energy	www.ti.com/energy
Logic	logic.ti.com	Industrial	www.ti.com/industrial
Power Mgmt	power.ti.com	Medical	www.ti.com/medical
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
RFID	www.ti-rfid.com	Space, Avionics & Defense	www.ti.com/space-avionics-defense
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Video and Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless-apps

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265