



TMS320C240x DSP Design Workshop

Student Guide

*DSP24
Revision 4.2
February 2002*



Important Notice

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

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

Customers are responsible for their applications using TI components.

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

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

Copyright © 1998, 1999, 2000, 2001, 2002 Texas Instruments Incorporated

Revision History

September 1998 – Revision 1.0

March 1999 – Revision 2.0

April 2000 – Revision 3.0

May 2000 – Revision 3.1

January 2001 – Revision 4.0

June 2001 – Revision 4.1

February 2002 – Revision 4.2

Mailing Address

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

Introduction

TMS320C240x DSP Design Workshop

**Texas Instruments
Technical Training**

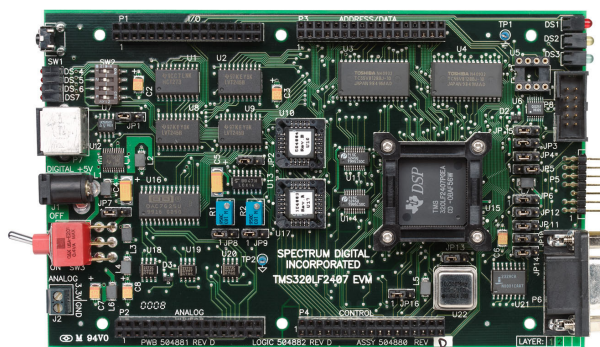
Introductions

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

TMS320C240x Workshop Outline

1. Introduction and Architectural Overview
2. Program Development Tools
3. Addressing Modes
4. Basic Programming Techniques
5. Advanced Programming Techniques
6. Numerical Issues
7. Implementing Algorithms
8. Logical Operations
9. System Initialization
10. Interrupts
11. Analog-to-Digital Converter
12. Event Manager
13. System Design
14. Communications
15. C Compiler
16. Development Support

Spectrum Digital TMS320LF2407 EVM



Introduction and Architectural Overview

Introduction

The TMS320C240x family of Digital Signal Processors combines the enhanced C2xx architectural CPU core with peripherals optimized for digital motor/motion control applications. This module will give an overview to the device architecture and serve as the basic foundation to this workshop.

Unless otherwise noted, the terms TMS320C240x and C240x refer to TMS320C24x, TMS320F24x, TMS320LC240x, and TMS320LF240x throughout the remainder of these notes. For specific details and differences please refer to the device data sheet and user's guide.

Learning Objectives

Learning Objectives

- ◆ Explain basic TMS320C240x block diagram
- ◆ List key features of C240x memory map, bus structures, and peripherals
- ◆ Describe differences among C240x devices

Module Topics

Introduction and Architectural Overview 1-1

Module Topics..... 1-2

TMS320C240x Architecture Basics 1-3

 CPU 1-4

 Program Memory..... 1-5

 Data Memory..... 1-5

Memory Map..... 1-6

Pipeline..... 1-7

Peripherals..... 1-8

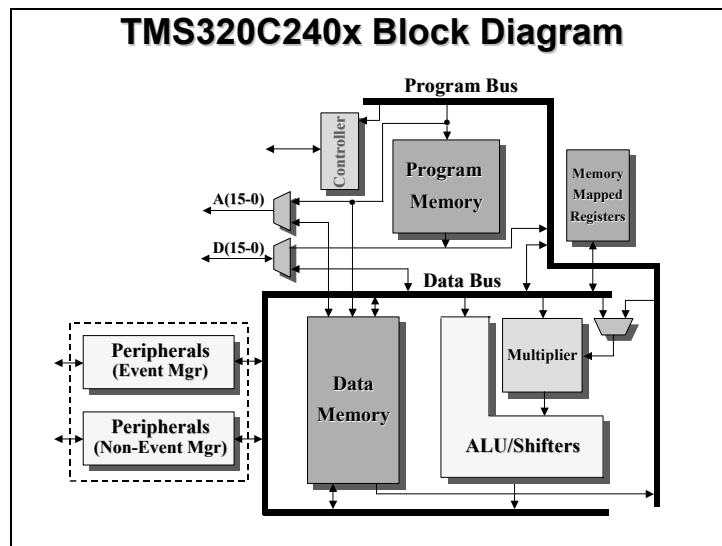
TMS320C240x Instruction Set 1-9

Review.....1-10

TMS320C240x Architecture Basics

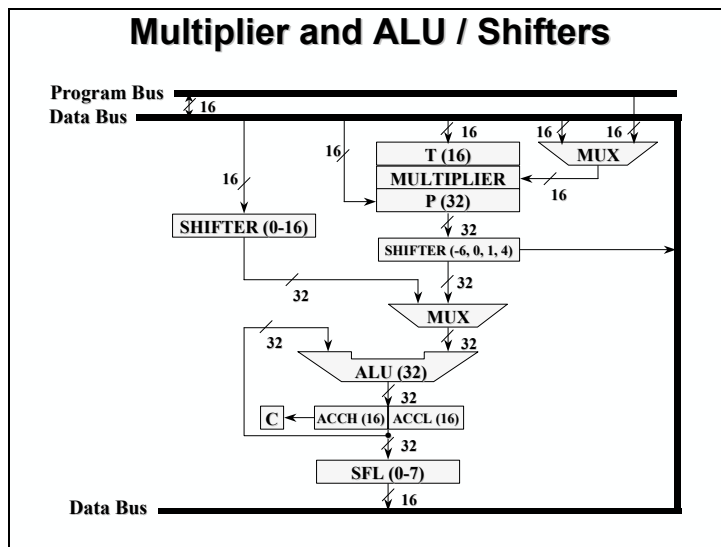
The TMS320C240x uses a *modified Harvard* architecture. These modifications consist of enhancements to the strict Harvard architecture in the form of features from the von Neumann architecture. Those features include the ability to initialize data memory from program memory, and the ability to transfer data memory to program memory. This capability allows the C240x to time-division multiplex its memory between tasks as well as to initialize its data memory with constants (for example, a coefficient) stored in the system's program ROM. This minimizes system cost by eliminating the need for a data ROM and maximizes data memory utilization by allowing dynamic redefinition of data memory's function.

The most important reason for basing the C240x on the Harvard architecture is speed. Separate data and program space allow simultaneous fetching of program instructions and data. In a mathematically intensive application, this effectively doubles algorithm throughput compared to (standard) von Neumann-type processors.



CPU

The Central Processing Unit (CPU) incorporates the multiplier and central arithmetic logic unit (CALU) along with three shifters and several program control registers.



Multiplier

The hardware multiplier is designed to perform a multiply in a single machine cycle. One input comes from the temporary register (T) and the other comes from the data bus or the program bus. The 32-bit result is stored in the product register (P), which is then available to the CALU.

The C240x has multiply/accumulate instructions which execute in a single cycle when used in repeat instructions.

Central Arithmetic Logic Unit (CALU)

The Central Arithmetic Logic Unit (CALU) contains the ALU and three separate shifters. The ALU performs single-cycle logical or arithmetic operations. Results are stored in the 32-bit accumulator (ACC).

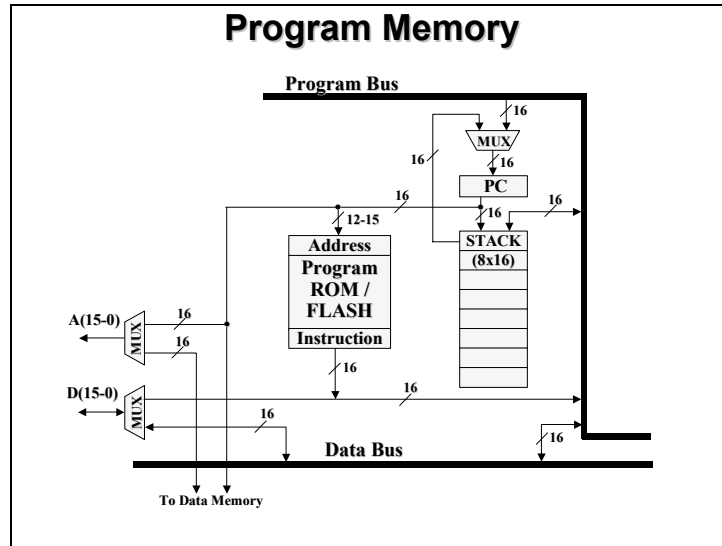
The C240x has a pre-scaling shifter from the data bus to the ALU, a post-scaling barrel shifter from the ACC to the data bus and a product shifter for scaling the multiplier results.

An ALU instruction is always performed as follows:

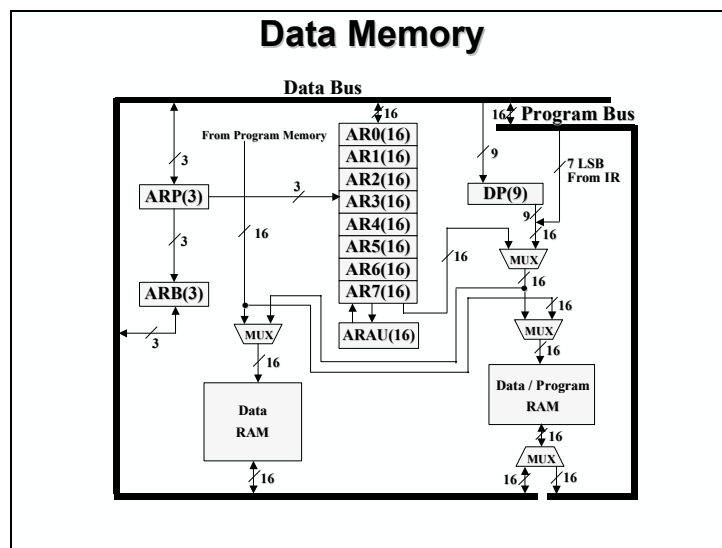
1. Data is read from RAM on the data bus.
2. Data is passed through the scaling shifter and to the ALU where the operation is performed.
3. The result is found in the accumulator.

One input to the ALU is always provided by the accumulator. The other may be transferred from the product register (P) or from the scaling shifter that is loaded from data memory.

Program Memory



Data Memory



Memory Map

The C240x memory space is divided into three regions:

- 64K program
- 64K data
- 64K I/O

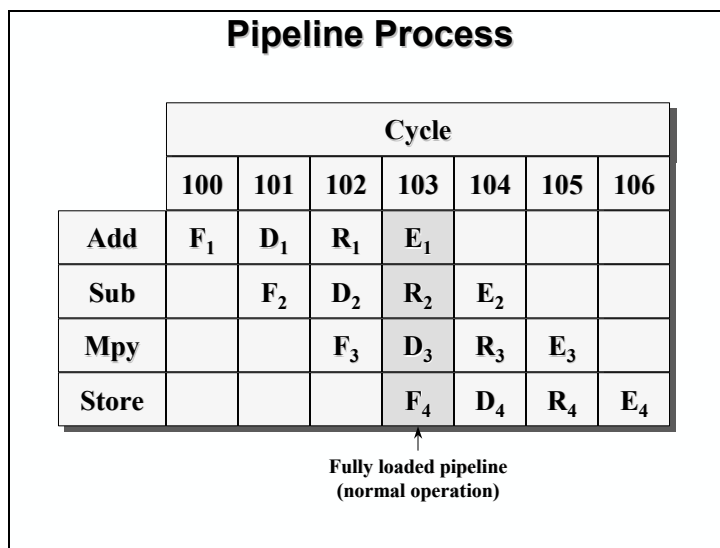
The C240x has a minimum of 544 words of on-chip RAM, which is sometimes referred to as dual-access RAM. It can implement the action of a delay line without the need for “circular buffering” as any other memory would. On reset, this memory is found in data space, but a portion of it may be relocated under software control to program space.

The second on-chip memory is ROM, which is located at the beginning of program space. It may be used or bypassed under control of the MP/ $\overline{\text{MC}}$ signal line. The amount of on-chip non-volatile memory (ROM or flash) varies based on the C240x device.

| TMS320LF2407 Memory Map | | | |
|-------------------------|--|------|--|
| Hex | Program | Hex | Data |
| 0000 | Interrupts | 0000 | Memory-Mapped Registers |
| 0040 | Reserved (code security - LF240xA) | 0060 | On-chip DARAM B2 |
| 0044 | | 0080 | Reserved |
| | On-chip 32K Flash (External if MP/MC = 1) | 0200 | On-chip DARAM B0 (CNF = 0) or Reserved (CNF = 1) |
| 8000 | SARAM (2K) (PON = 1) or External (PON = 0) | 0300 | On-chip DARAM B1 |
| 8800 | External | 0400 | Reserved |
| FE00 | Reserved (CNF = 1) | 0800 | SARAM (2K) (DON = 1) or Reserved (DON = 0) |
| | External (CNF = 0) | 7000 | Non-EV Peripherals |
| FF00 | On-chip DARAM B0 (CNF = 1) or External (CNF = 0) | 7400 | EV Peripherals |
| FFFF | | 7540 | Reserved |
| | | 8000 | External |
| | | FFFF | |

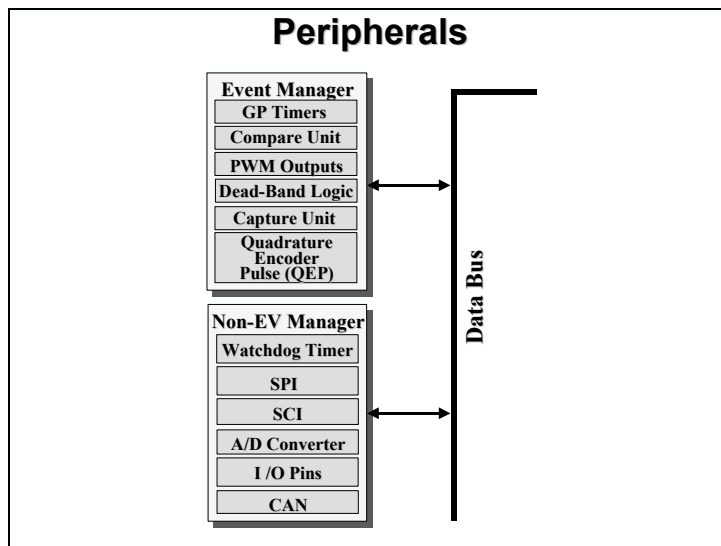
Pipeline

The C240x has an instruction pipeline which results in a more efficient operation of program execution. The instruction fetch-decode-read-execute pipeline phases are independent of each other. Instructions overlap such that, in a given cycle, up to four different instructions can be active, each at a different stage completion.



Peripherals

The C240x devices contain peripherals optimized for motor/motion control applications. For many systems, this could provide a low-cost, high performance solution.



| TMS320C24xx Family | | | | | | | | | |
|--------------------|-------------|-------|--------------|---------------------|-----------|-----|-----|-----|----------|
| TMS320 Devices | ROM / Flash | SARAM | PWM Channels | Compares / Captures | GP Timers | QEP | CAN | SPI | I/O Pins |
| C/F240 | 16K | - | 12 | 9 / 4 | 3 | 2 | No | Yes | 28 |
| F241 | 8K | - | 8 | 5 / 3 | 2 | 2 | Yes | Yes | 26 |
| C242 | 4K | - | 8 | 5 / 3 | 2 | 2 | No | No | 26 |
| F243 | 8K | - | 8 | 5 / 3 | 2 | 2 | Yes | Yes | 32 |
| LF2401 | 8K | 0.5K | 7 | 4 / 1 | 2 | 2 | No | No | 13 |
| LC/F2402 | 6 / 8K | - | 8 | 5 / 3 | 2 | 2 | No | No | 21 |
| LF2403 | 16K | 0.5K | 8 | 5 / 3 | 2 | 2 | Yes | Yes | 21 |
| LC2404 | 16K | 1K | 16 | 10 / 6 | 4 | 4 | No | Yes | 41 |
| LC/F2406 | 32K | 2K | 16 | 10 / 6 | 4 | 4 | Yes | Yes | 41 |
| LF2407 | 32K | 2K | 16 | 10 / 6 | 4 | 4 | Yes | Yes | 41 |

ONLY - C/F240, F243, and F2407 includes an external memory interface

All Devices Include - 544 Words of DARAM, Watchdog Timer, SCI (UART), and ADC

TMS320C240x Instruction Set

The instruction set for the TMS320C240x is identical to that of the TMS320C2xx. This instruction set supports numerically intensive signal processing operations, as well as general-purpose applications. The instruction set is compatible with the TMS320C2x instruction set, and is a subset of the TMS320C5x instruction set.

| TMS320C240x Instruction Set | | | | | | | | | | | |
|-----------------------------------|------|------|--|--|--|---|------|--|--|----------------|------|
| Accumulator Memory Reference | | | | | | Auxiliary Register and Data Page Pointer | | | | Multiply, T, P | |
| ABS | NEG | SUB | | | | ADRK | MAR | | | APAC | MPYA |
| ADD | NORM | SUBB | | | | CMPR | SAR | | | LPH | MPYS |
| ADDC | OR | SUBC | | | | LAR | SBRK | | | LT | MPYU |
| ADDS | ROL | SUBS | | | | LDP | | | | LTA | PAC |
| ADDT | ROR | SUBT | | | | | | | | LTD | SPAC |
| AND | SACH | XOR | | | | | | | | LTP | SPH |
| CMPL | SACL | ZALR | | | | | | | | LTS | SPL |
| LACC | SFL | | | | | | | | | MAC | SPM |
| LACL | SFR | | | | | | | | | MACD | SQRA |
| LACT | | | | | | | | | | MPY | SQRS |
| I/O and Data Memory Operations | | | | | | Control Instructions | | | | Branch | |
| BLDD | OUT | | | | | BIT | POP | | | B | CC |
| BLPD | SPLK | | | | | BITT | POPD | | | BACC | INTR |
| DMOV | TBLR | | | | | CLRC | PSHD | | | BANZ | NMI |
| IN | TBLW | | | | | IDLE | PUSH | | | BCND | RET |
| | | | | | | RPT | SETC | | | CALA | RETC |
| | | | | | | LST | SST | | | CALL | TRAP |
| | | | | | | NOP | | | | | |

C Language Programming

The C compiler is a full implementation of the ANSI standard and outputs assembly language source code. The C compiler supports in-line assembly code; calling assembly language from C and calling C from assembly. Additionally, variables defined in C source can be addressed in the assembly code and vice versa. By allowing a mixture of C and assembly languages, the programmer can use C to achieve faster code development and use assembly language to write those segments that require optimal performance.

C code is relatively efficient on the TMS320, given the availability of an optimizer which may be invoked during compilation. Special internal hardware, such as the eight auxiliary registers, greatly improves the speed of stack and pointer operations.

The module, "C Compiler," covers many of the issues associated with the C compiler for fixed-point devices.

Review

Review

1. Why a modified “Harvard” Architecture?
2. What are the two major buses?
3. Describe on-chip memory resources.
4. Arithmetic Logic Unit - width?
5. Multiplier - input and results width?

Program Development Tools

Introduction

The goal of this module is to understand the basics of writing assembly language programs using the standard Common Object File Format (COFF) tools used by Texas Instruments. This involves understanding the basic structure of the assembly file, along with the basic operation of the assembler and linker.

Learning Objectives

Learning Objectives

- ◆ Describe steps to create executable output files
- ◆ Create an assembly file containing:
 - ◆ Code
 - ◆ Constants (initialized data)
 - ◆ Variables
- ◆ Create a linker command file which:
 - ◆ Describes a system's available memory
 - ◆ Indicates where code and data shall be located
- ◆ Write system reset code

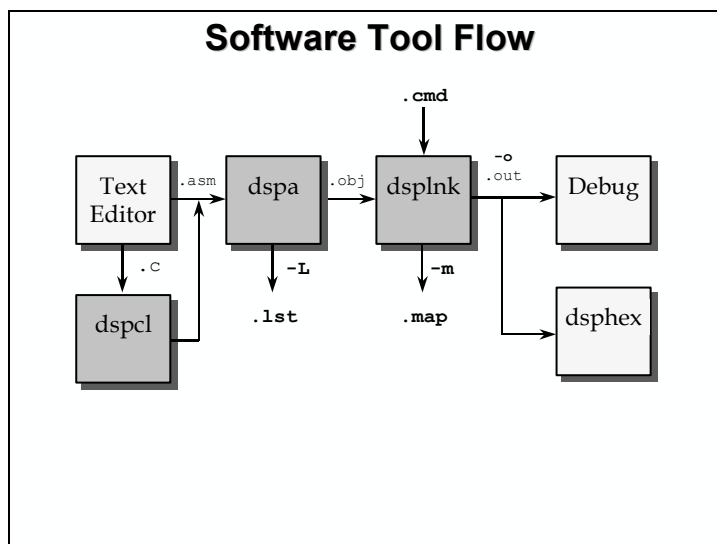
Module Topics

| | |
|--|------------|
| Program Development Tools | 2-1 |
| <i>Module Topics</i> | 2-2 |
| <i>COFF Concepts</i> | 2-3 |
| <i>Assembly Tools</i> | 2-5 |
| Assembler | 2-5 |
| Linker | 2-5 |
| <i>Assembly Conventions</i> | 2-6 |
| <i>Creating the Assembly File</i> | 2-8 |
| Program Code..... | 2-9 |
| Constants (initialized data) | 2-9 |
| Variables (uninitialized data)..... | 2-10 |
| Completed Example | 2-11 |
| Summary | 2-11 |
| <i>Exercise 1</i> | 2-12 |
| <i>Lab 2a</i> | 2-13 |
| <i>Linking Assembly Code</i> | 2-15 |
| <i>Linker Command Files (.cmd)</i> | 2-16 |
| Files - input and output | 2-16 |
| Memory-Map Description | 2-16 |
| Section Placement..... | 2-18 |
| <i>Exercise 2</i> | 2-19 |
| Summary: Linker Command File | 2-20 |
| <i>Lab 2b</i> | 2-21 |
| <i>Multiple Sections</i> | 2-24 |
| Creating Your Own Sections | 2-25 |
| <i>Reset Vector</i> | 2-26 |
| Setting Up a Reset Vector | 2-26 |
| Referencing Labels | 2-28 |
| Linker Command File..... | 2-29 |
| <i>Exercise 3</i> | 2-30 |
| <i>Summary</i> | 2-32 |
| <i>Lab 2c</i> | 2-33 |
| <i>Review</i> | 2-36 |
| Solutions..... | 2-37 |

COFF Concepts

In an effort to standardize the software development process, TI has selected 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 are written in assembly-level mnemonics using any word processor capable of providing a simple ASCII file output. The expected extension of a source file is `.ASM`, which stands for *assembly*.



Next, each of these modules is *assembled* to translate the mnemonic-level code to a binary representation which is recognizable by the TMS320; this results in the object file `.OBJ`. A list, `.LST`, file is optionally produced to document the assembled results.

Numerous modules may be joined to form a complete program. The *linker* is a software tool capable of efficiently allocating the resources available on the TMS320 to each module in the system. The linker is able to refer to a command (`.CMD`) file which identifies all the input and output file names, the resources available on the TMS320, and where the various sections within each module are to go. Outputs of the linking process may include the linked object file (`.OUT`), which runs on the TMS320, and 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.

Assembly Tools

Assembler

The assembler translates assembly-language source code using instruction mnemonics and symbols into relocatable object code.

The most common options are summarized in the table below:

| Option | Action |
|---------|--|
| -v10 | Produce 'C1x code |
| -v25 | Produce 'C2x code |
| -v2xx | Produce 'C2xx code |
| -v50 | Produce 'C5x code |
| -L (-l) | Create listing file (none by default) |
| -s | Put all symbols in .obj file for debug |

When the -L option is specified, the resultant listing file is useful because it contains the output of the assembler and shows any errors and warnings with the associated line of code. The fields in the listing file are shown in the *Fixed-Point Assembly Language Tools User's Guide*.

Linker

The linker maps relocatable software (object-level code) to the users system hardware.

The command file includes:

- System memory configuration
- Allocation of code and data across system memory

Assembler Constants

| Type | Examples |
|-------------------|---|
| Binary | 1110001b or 11111001B |
| Octal | 226q or 572Q |
| Decimal | 1234 or +1234 or -1234 (Default) |
| Hexadecimal | 2A40h or 2A40H or 0FF00h |
| Floating-point | 1.623e-23 (sign and decimal point optional) |
| Character | 'D' |
| Character strings | "this is a string" |

Note: A Hexadecimal constant cannot begin with a letter, start it with "0" instead; for example, use 0FF00h, not FF00h.

Creating the Assembly File

Assembly files consists of three main sections:

- Program Code
- Initialized data (constants)
- Uninitialized data (variables)

These sections are examined in the following assembly file which solves: $z = x + y$

Example: $z = x + y$

Code

| | | |
|---------|------------|-------|
| get x | start LACC | x |
| add y | ADD | y |
| store z | SACL | z |
| loop | SACH | z+1 |
| | B | start |

Constants
x = 2
y = 7

Variables
z

Program Code

Program code consists of the sequence of instructions used to manipulate data. Specific instructions are discussed in detail later in the workshop.

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 chips, such as EPROM's.

The previous example used boldface titles to indicate the different sections of the assembly file. The assembler requires you to 'tag' each section, but its command uses a slightly different syntax.

➤ Define the Code Section

The assembler directive (assembler command) to delineate the code section is `.text`.

```

                .text
start          LACC      x
               ADD      y
               SACL     z
               SACH     z+1
               B         start

```

Constants (initialized data)

Initialized data are those data memory locations defined at reset. They contain 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 EPROM or other non-volatile memory.

➤ Define the Constants Section

Defining initialized data requires two assembler directives:

1. One to define the section: `.data`
2. The second "builds" a constant by reserving a memory location and placing a value into it: `.int`

Our previous example used two integer constants, which are defined using two `.int` directives. The constant's name, called 'label' or 'symbol' is on the left, while the initial value assigned is placed on the right:

```

                .data
x              .int    2
y              .int    7

```

Variables (uninitialized data)

Uninitialized data memory locations can be changed and manipulated by the program code during runtime execution. Unlike program code or constants, 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, and such.

Each variable must be declared with an assembler directive to reserve memory to contain its value. By their nature, no value is assigned by the assembler or linker; instead they must be loaded at runtime by the program code.

➤ Define the Variables Section

The syntax for variables includes the:

- assembler directive (`.bss`)
- symbol (label) on the *right* side
- size of variable

The variable `z` from our earlier example, would be declared:

```
.bss    z,2
```

Declare an array of memory locations called `x` of length five with:

```
.bss    x,5
```

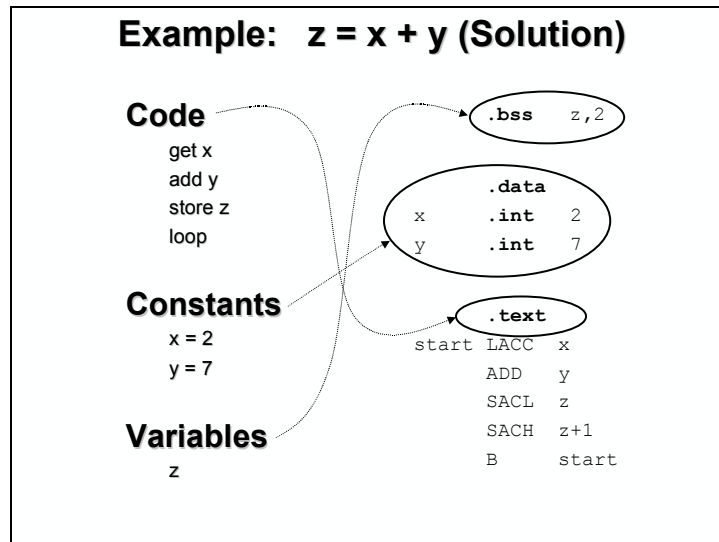
.bss Tips

- ◆ **Only directive with label in the operand field**
- ◆ **Use separate `.bss` statements for each named variable**
- ◆ **Remember `.bss` by thinking:**
 - ◆ **Block** - reserves a block of memory
 - ◆ **Symbol** - beginning at address symbol
 - ◆ **Size** - of the specified size
- ◆ **Example: Create a 5-word array 'x'**

```
.bss    x,5
```


Completed Example

Putting together all of these sections yields:



Summary

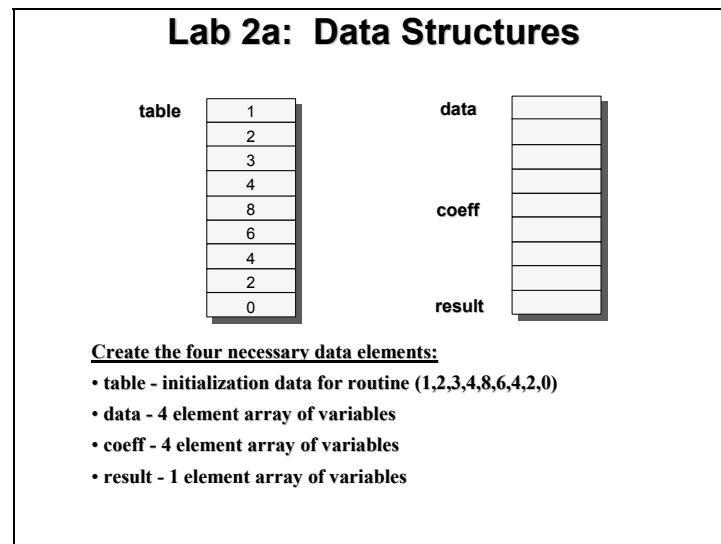
Assembler Directives

| Assembler Directive | Example | Definition |
|---------------------|-----------------|------------------------------------|
| .text | .text | Code to follow |
| .data | .data | Constants to follow |
| .bss | .bss x, 10 | Allocate space for Variables |
| .int .word | A .int 53h, 5Ah | Creates 16-bit integer constant(s) |

Lab 2a

➤ Objective

In Labs 2, 3, and 4 you will create a Sum-of-Products routine, multiplying and summing two arrays of four elements each. Lab 2 takes you through setting up the data structures and sections. Lab 3 practices data addressing by initializing the data and coefficient arrays. Lab 4 is where you will write the actual Sum-of-Products routine.



We begin the process by setting up the sections necessary to perform a sum-of-products.

Create the four necessary data elements:

- table – Initialization data for the routine (1,2,3,4,8,6,4,2,0)
- data – 4 element array of variables
- coeff – 4 element array of variables
- result – 1 element array of variables

Note: For those already comfortable with COFF Assembler syntax and Code Composer, try creating LAB2.ASM to accomplish the objectives above. If you're new to this syntax — or a bit rusty on assembler programming in general — the next page provides a step-by-step procedure.

➤ Procedure

Create a New Project

1. Double click on the Code Composer icon on the desktop. Maximize Code Composer to fill your screen. The menu bar (at the top) lists File ... Help. Note the horizontal tool bar below the menu bar and the vertical tool bar on the left-hand side. The window on the left is the project window and the large right hand window is your workspace.

2. A *project* is all the files you'll need to develop an executable output file (.OUT) which can be run on the target hardware. Let's create a new project for this lab. On the menu bar click:

Project → New

and make sure the "SAVE IN" location is: C:\DSP24\LABS and type LAB2 in the file name window. This will create a *make* file which will invoke all the necessary tools (assembler, linker, compiler) to build your project.

3. Add the assembly file to the new project. Click:

Project → Add Files to Project

and make sure you're looking in C:\DSP24\LABS. Change the "files of type" to view assembly files (.ASM) and select LAB2.ASM and click OPEN. This will add the file LAB2.ASM to your newly created project.

4. Add LAB2.CMD to the project using the same procedure. This will be used during Lab 2b.
5. 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.MAK. Notice that the LAB2.CMD file is listed. Click on Source to see the current source file list (i.e. LAB2.ASM).

Edit LAB2.ASM

6. To open and edit LAB2.ASM, double click on the file in the project window. The code you see in this file is not related to setting up the data structures and sections. It is simply a place holder for use during future labs.
7. Define three arrays in RAM as described on the "Lab 2a: Data Structures" slide by creating uninitialized sections called *data*, *coeff*, and *result*. Refer to the diagram for the sizes.
8. Define an initialized data table section called *table* that contains the nine values shown on the "Lab 2a: Data Structures" slide.
9. Define an initialized program section for code above the beginning label of the code (*start*). Save your changes by clicking the disk on the horizontal tool bar "Save", or on the menu bar click: File → Save

Assemble LAB2.ASM

10. Assemble LAB2.ASM by clicking on the top button on the vertical toolbar (or on the menu bar click: Project → Compile File). When your mouse hovers over this button, you will see the words Compile File. Check for errors. If you get an assembly error, scroll the Build window at the bottom of your screen until you can see the error and simply double-click the error shown in red. Your cursor should now be positioned at the start of the line with the error in your assembly file. We added a little error (in addition to any others you may have made) so you could see how Code Composer reacts. To correct the error, replace label 'strt' with 'start'. Save your changes and compile the file again. When completed (No Errors, No Warnings), you can close the LAB2.ASM edit window.

End of Exercise

Linking Assembly Code

The three sections of an assembly file, discussed earlier, 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

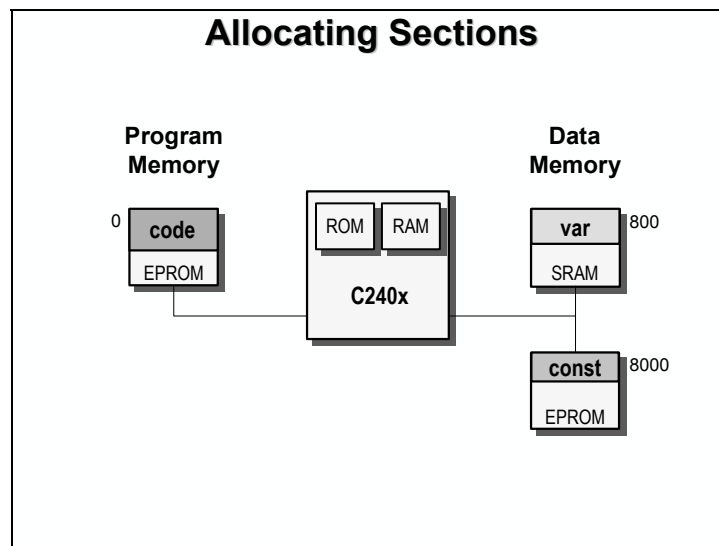
Program code must be defined upon system reset (power turn-on). Due to this basic system constraint it is usually necessary to place program code into non-volatile memory chips, such as EPROM's.

Constants (initialized data)

Initialized data are those data memory locations defined at reset. Similar to program code, constant data is expected to be valid upon reset of the system. It is often found in EPROM or other non-volatile memory.

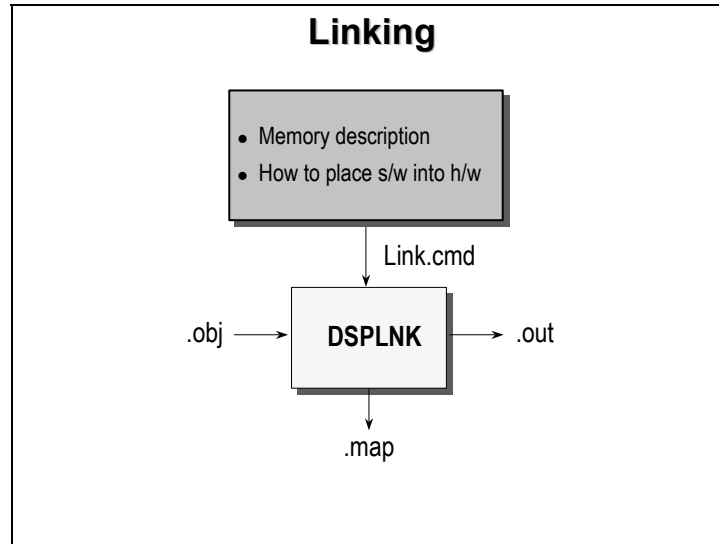
Variables (uninitialized data)

Uninitialized data or variables must reside in volatile memory, such as RAM so they can be modified and updated.



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.



Files - input and output

The two most common linker options are `-o` to specify the output file name, and `-m` to specify a map file name.

Memory-Map Description

Describe the memory configuration of your target system to the linker. Without this specification, the linker might place code or data into memory that doesn't exist.

For example, if you placed an 2K EPROM starting at memory location zero, it would read:

```
MEMORY
{
    NAME:  origin = 0000h , length = 0800h
}
```

You define each memory segment using the above format. If you added a RAM and an ABT646 transceiver, it might look like:

```
MEMORY
{
    EPROM:  origin = 0000h , length = 0800h
    RAM:    origin = 1000h , length = 02000h
    ABT646: origin = 8000h , length = 00001h
}
```

Remember that the C240x processors have three memory maps: *Program*, *Data*, and *I/O*. Therefore, the MEMORY description must describe each of these separately. TI's loader uses the following syntax to delineate each of these:

| Linker Page | TI Definition |
|-------------|---------------|
| Page 0 | Program |
| Page 1 | Data |
| Page 2 | I/O |

Linker Command File

```
MEMORY
{
    PAGE 0:      /* Program */
    EPROM:      org = 0000,    len = 0800h

    PAGE 1:      /* Data */
    SRAM:       org = 0800h,    len = 2000h
    DEEPROM:    org = 8000h,    len = 2000h
}
```

Memory Suggestions

1. Describe each memory resource on the processor (internal RAM and/or ROM)
2. Describe each external memory chip in your system
3. Combine contiguous memory segments, if desired
4. Split any memory segment into multiple segments, if desired
5. Name memory segments with useful names; e.g.:
 - Types of memory chips (EPROM, RAM, EEPROM)
 - Usage (vectors, code, variables)
 - Chip layout names (U1, E2)

Section Placement

You can specify how you want the sections to be distributed through memory. You would use the following code to link the three sections into the memory specified in the former example:

```
SECTIONS
{
    .text:> EPROM
    .data:> DEEPROM
    .bss:> SRAM
}
```

The linker will gather all the `.text` 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.

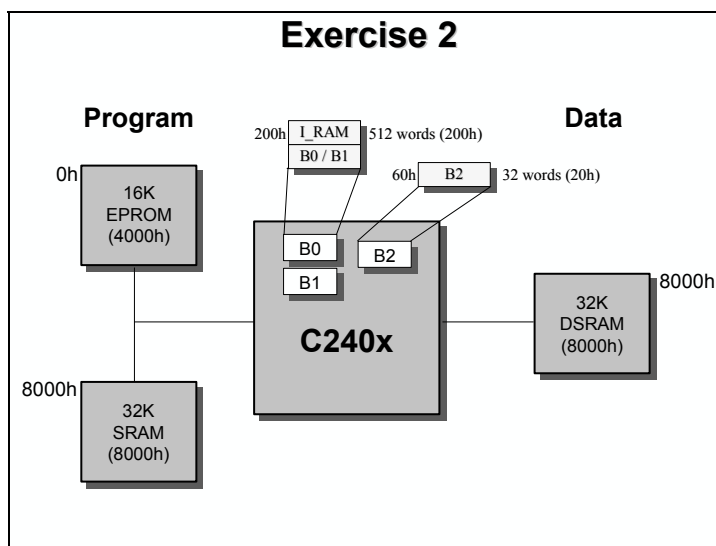
Linker Command File

```
MEMORY
{
    PAGE 0:      /* Program */
        EPROM:   org = 0000h,   len = 0800h

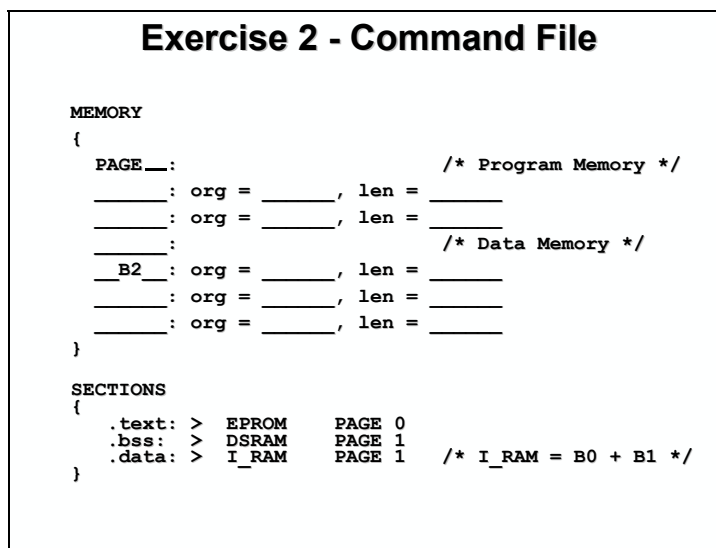
    PAGE 1:      /* Data */
        SRAM:    org = 0800h,   len = 2000h
        DEEPROM: org = 8000h,   len = 2000h
}
SECTIONS
{
    .text:> EPROM      PAGE 0
    .data:> DEEPROM    PAGE 1
    .bss:>  SRAM       PAGE 1
}
```


Exercise 2

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



Fill in the blanks:



The answer should look like:

```
Exercise 2 - Solution

MEMORY
{
    PAGE 0:
        EPROM:  org = 0000h,  len = 4000h
        SRAM:   org = 8000h,  len = 8000h
    PAGE 1:
        B2:     org = 0060h,  len = 0020h
        I_RAM:  org = 0200h,  len = 0200h
        DSRAM:  org = 8000h,  len = 8000h
}

SECTIONS
{
    .text: > EPROM    PAGE 0
    .bss:  > DSRAM    PAGE 1
    .data: > I_RAM    PAGE 1    /* I_RAM = B0 + B1 */
}
```

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

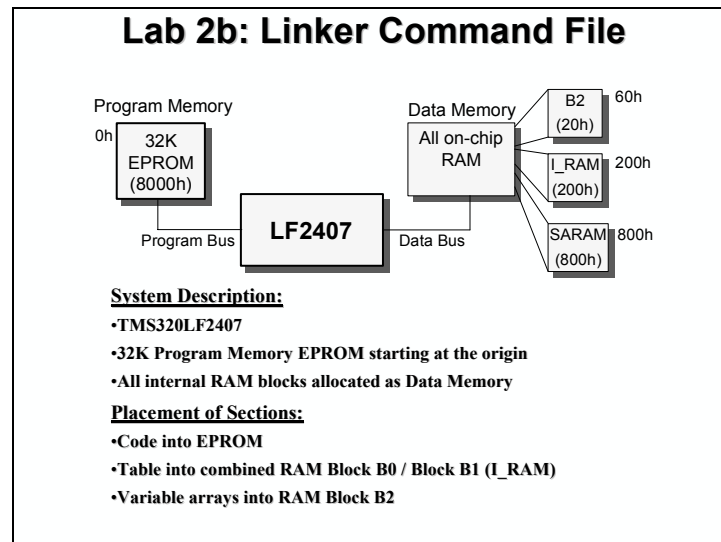
◆ Memory Map Description
    • Name
    • Location
    • Size
    • Attributes

◆ Sections Description
    • Directs software sections into named memory regions
    • Allows per-file discrimination
    • Allows separate load/run locations
```

Lab 2b

➤ Objective

Link your assembly file (LAB2.ASM) into the system described below. Create a linker command file as part of this process.



System Description

- TMS320LF2407
- 32K Program EPROM starting at the origin
- All internal RAM blocks allocated as Data memory

Placement of Sections:

- Code into EPROM
- Table into combined RAM Block B0 / Block B1 (I_RAM)
- Variable arrays into RAM Block B2

➤ Procedure

Edit LAB2.CMD

1. To open and edit LAB2 .CMD, double click on the filename in the project window.
2. Edit the `Memory{}` declaration by describing the system memory shown on the “Lab2b: Linker Command File” slide.
3. Place the sections defined in LAB2 .ASM into the appropriate memories via the `Sections{}` area. Save your work.

Link LAB2

4. Setup the linker options by clicking:

Project → Options

on the menu bar. Select the Linker tab. In the middle of the screen select “No Autoinitialization”. Create a map file by typing LAB2.MAP in the Map Filename [-m] field.

Next, select the Assembler tab. In the middle of the screen check “Enable Source Level Debugging”. This feature will be useful during the next part of the lab.

Then select OK to save the Build Options. To open up more workspace, close any open files that you do not need.

5. The top four buttons on the vertical 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 |

6. Click the “Rebuild All” button and watch the tools run in the build window. Debug as necessary. Right-click on the build window and Hide the build window. Close the LAB2.CMD edit window.
7. Open and inspect LAB2.MAP. Make sure you are looking in C:\DSP24\Labs. This file will show you the results of the link process. Note the addresses for the sections. What address does it indicate for:

.text _____

.data _____

.bss _____

Are the results as expected? Close the .MAP file when you are done.

8. Load the output file onto the target. Click:

File → Load Program...

Make sure you are looking in C:\DSP24\Labs. Select LAB2.OUT and click OPEN.

Note: Code Composer can automatically load the output file after a successful build. On the menu bar click: Option → Program Load... and select: “Load program after build”, then click OK.

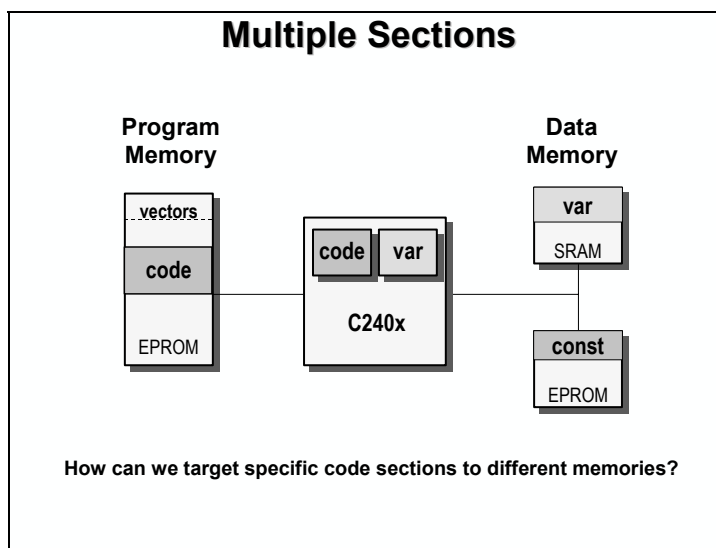
9. If code generation is successful, the Dis-Assembly window will display the LAB2 .ASM source file and a yellow highlight on the line at address “0000h” should appear. This indicates that you are now ready to run.
10. On the menu bar click:
View → Memory
or click the View Memory button on the vertical toolbar. Type “table” into the address to display the contents of the memory starting at label *table*. (Note: Code Composer is case sensitive). You can display as many independent windows as you require. Do you see your initial values in the memory window displaying “table”?
Note that by double-clicking on any location you can edit the contents of the memory location.
11. Close the Dis-Assembly and Memory windows (and source window, if open).

End of Exercise

Multiple Sections

We've spoken of placing sections in memory, but what if we want to put code — or data — in more than one location?

For example, if you wanted to run some code from EPROM and the critical routines from internal memory.



Creating Your Own Sections

Two assembler directives allow you to create and name your own sections. These allow programmers to create multiple constant-data, program-code, or variable-data sections and link them to different memory locations.

| Multiple Sections | | |
|---------------------|--------------------------------------|---|
| Assembler Directive | Example | Description |
| <code>.sect</code> | <code>.sect "vectors"</code> | Creates initialized sections: <ul style="list-style-type: none"> • code • constants |
| <code>.usect</code> | <code>label .usect "name", 23</code> | Creates uninitialized sections: <ul style="list-style-type: none"> • variables |

| | Unnamed Sections | User Named Sections |
|----------------------|--|---|
| Initialized Memory | <code>.text</code> <code>.data</code> | <code>.sect "name"</code> |
| Uninitialized Memory | <code>.bss symbol, size</code> | <code>symbol .usect "name", size</code> |

Bottom Line

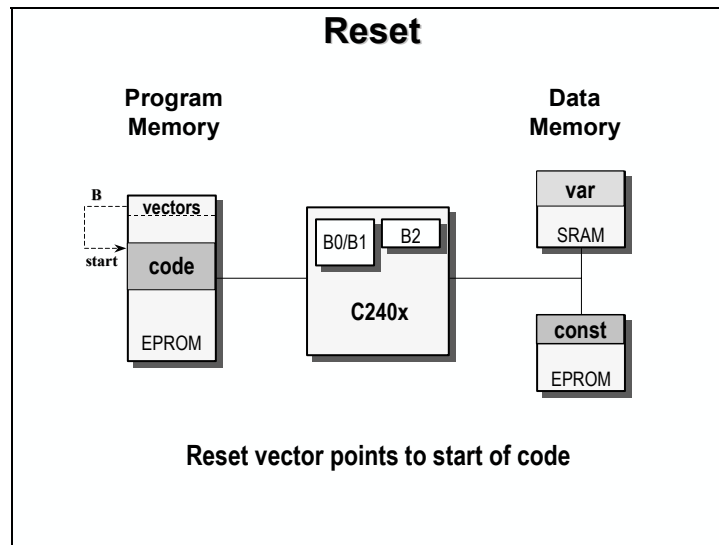
Use `.sect` and `.usect` directives whenever you need to place code or data in specific memory locations.

Reset Vector

Finally, how do we start from the beginning?

All real-world programs begin at RESET. Here's the basic sequence of events when the RESET pin is toggled low by the hardware or debugger:

- Pins and registers are set to specific values (discussed later)
- Program Counter (PC) is set to 0
- Processor starts executing at zero
(we usually put a branch at zero to the beginning of our code)



Setting Up a Reset Vector

1. Determine the starting address (label) of your code.

Note: In this workshop, we use the label `start` at the beginning of our programs, similar to the way the C language uses `main`.

2. Branch to this starting label

```
b          start      ; RESET vector
```


Reset

sum.asm

```

        .text
start   LACC x
        ADD y
        SACL z
        SACH z+1
        B    start

        .data
x       .int 2
y       .int 7
        .bss z, 2

```

label B start

What section should this code be put into?
Why?

3. Locate this code at memory location ZERO.

Reset

sum.asm

```

        .text
start   LACC x
        ADD y
        SACL z
        SACH z+1
        B    start

        .data
x       .int 2
y       .int 7
        .bss z, 2

```

Create new section to enable link directly to location zero.

```

        .sect "vectors"
label B start

```

Where should we put this code?

The optimum method of placing the reset value at location zero is to use the `.sect` directive, as follows:

```

.sect    "vectors"
b        start

```

We could have placed the code into the `.text` section, but it makes it much more difficult to force this code specifically to location 0.

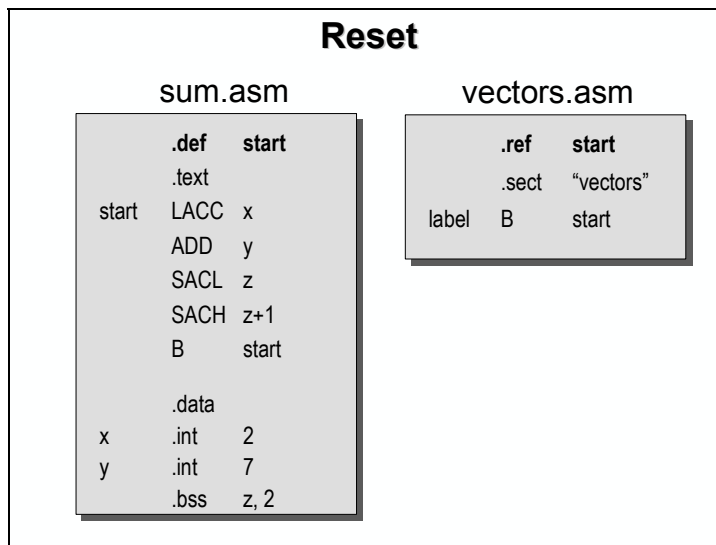
```

.text
b        start    ; RESET vector

```

Referencing Labels

If a label is to be shared between two (or more) files, it must be defined — declared — as global. This is done with the `.def` and `.ref` commands.



- .def** is defined in the current module and used in another module.
- .ref** is referenced in the current module, but defined in another module.
- .global** may be either of the above.

Linker Command File

The linker command file might be modified to:

Linker CMD File with Vectors

```

MEMORY
{
    PAGE 0:
    /* EPROM: org = 0000h, len = 2000h */
    /* VECS: org = 0000h, len = */
    EPROM: org = , len =
    SRAM: org = 8000h, len = 8000h
    PAGE 1:
    /* Data Memory */
    B2: org = 0060h, len = 0020h
    I_RAM: org = 0200h, len = 0200h
    DSRAM: org = 8000h, len = 8000h
}

SECTIONS
{
    .text: > EPROM PAGE 0
    .bss: > DSRAM PAGE 1
    .data: > B2 PAGE 1
    : > PAGE
}

```

Note: It is convenient to create a separate segment of memory to place the vector table. This prevents another section from accidentally being linked ahead of the vectors section.

Exercise 3

Let's put ourselves into the linkers position.

Exercise 3

Procedure

1. Resolve .set expression in mult.asm
2. Using EX3.CMD, fill-in the post-link addresses (left-hand side blanks) of the three ASM files
 - Put an 'X' in any blank that has no address
 - Branch is a 2-word instruction
 - Other instructions are single-word
3. Resolve symbolic references (i.e. replace right-hand side symbols with the corresponding left-hand side address)
4. Link Order: mult.obj sum.obj vectors.obj

Exercise 3: EX3.CMD

```
MEMORY
{
    PAGE 0:
        VECS:  org = 0000h, len = 0040h
        EPROM: org = 0040h, len = 1FC0h
        SRAM:  org = 8000h, len = 8000h
    PAGE 1:
        B2:    org = 0060h, len = 0020h
        I_RAM: org = 0200h, len = 0200h
        DSRAM: org = 8000h, len = 8000h
}

SECTIONS
{
    .text: > EPROM    PAGE 0
    .bss:  > DSRAM    PAGE 1
    .data: > B2       PAGE 1
    vectors: > VECS    PAGE 0
}
```

Exercise 3: mult.asm**mult.asm**

```

____ .ref  z, y
____ .def  c, mult
____K .set  1024
____ .text
____mult LT  z  ____
____ MPY  y  ____
____ ADD  c  ____
____ SACL z  ____
____done B   done ____
____ .data
____c .int  4 * K ____

```

Exercise 3: sum and vectors.asm**sum.asm**

```

____ .ref  c, mult
____ .def  start, z, y
____ .text
____start LACC x ____
____ ADD  c  ____
____ SACL z  ____
____ B    mult ____

____ .data
____x .int  2
____y .int  7
____ .bss z, 1

```

vectors.asm

```

____ .ref  start
____ .sect "vectors"
____ B    start ____

```

Summary

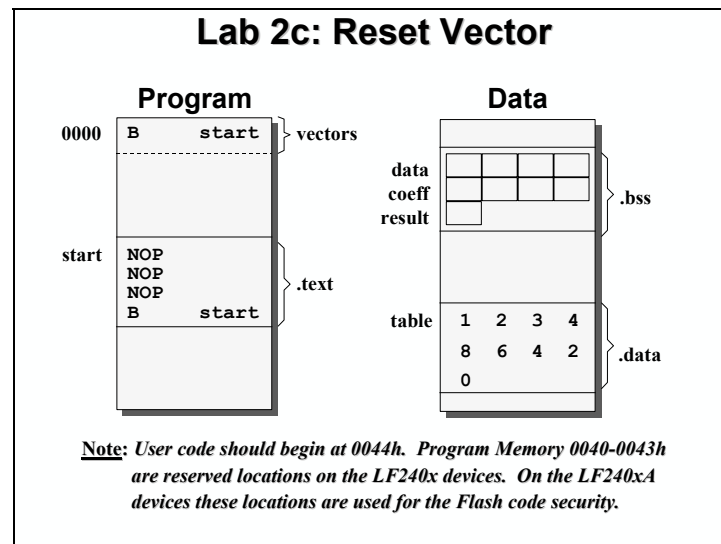
| | Assembler Directives | Definition |
|--------------------|----------------------|---|
| Section Directives | .text | program code section (initialized) |
| | .data | initialized data (constants) section |
| | .sect | user-named initialized section |
| | .bss | variable data section (uninitialized) |
| | .usect | user-named uninitialized section |
| Define Constants | .int | creates integer constant |
| | .word | |
| | .float | creates floating-point constant |
| Global Labels | .def | defines labels globally |
| | .ref | references a global label |
| | .global | declares label global (does the job of both .ref and .def) |
| Misc. | .set | assigns a value (similar to #define in C) |
| | .equ | |
| | .title | prints title at top of every page in listing |
| | .end | the assembler will stop when it finds this or the end-of-file |

Lab 2c

➤ Objective

Finally, Lab 2c asks us to add a reset vector to the system we've been working with. The reset vector resides in external memory (at address 0).

Your challenge is to use the assembly tools to specify the value for the reset vector using “relocatable” assembly coding techniques; i.e., labels. Create a new file to contain your reset vector.



➤ Procedure

Create VECTORS.ASM

1. Create a new file by clicking on the left most button on the horizontal toolbar “New”.
2. Add an initialized code section named “vectors” that contains: B start
Make sure that the label (start) is visible to the linker (.ref). Save your file by clicking on the Save button on the horizontal toolbar. When prompted, save your file and name it “vectors” as type “Assembly Source File” in the C:\DSP24\LABS directory.

Assemble VECTORS.ASM

3. Assemble VECTORS.ASM by clicking on the compile button as you did before to assemble LAB2.ASM. Check for errors before moving on. Save your work after changes.
4. Add VECTORS.ASM to the project using the procedure shown earlier. Close the VECTORS.ASM edit window.

Edit LAB2.ASM

5. To open and edit LAB2.ASM, double click on the file in the project window. Make sure that the label (*start*) is visible to the linker (.def). Save your file by clicking on the Save button on the horizontal toolbar. Close the LAB2.ASM edit window.

Edit LAB2.CMD

6. To open and edit LAB2.CMD, double click on the filename in the project window. Modify the Memory{}, and Sections{} area, as needed. *Note: User code should begin at 0044h. Program Memory 0040- 0043h are reserved locations on the LF240x devices. On the LF240xA devices these locations are used for the Flash code security.* Save your work. Close the LAB2.CMD edit window.

Rebuild All

7. Click the “Rebuild All” button and watch the tools run in the build window. Debug as necessary. Right-click on the build window and Hide the build window.

Note: Code Composer can automatically load the output file after a successful build. On the menu bar click: Option → Program Load... and select: “Load program after build”, then click OK.

8. Open and inspect LAB2.MAP. This file will show you the results of the link process. Note the addresses for the sections. Are the results as expected? Close the .MAP file when done.
9. Reload the output file onto the target. Click: File → Reload Program...
Then reset the DSP by clicking on: Debug → Reset DSP
10. If code generation is successful, the Dis-Assembly window will display the source file and a yellow highlight on “B start” should appear. This indicates that you are now ready to run. To debug using both source and assembly, right click on the VECTORS.ASM window and select Mixed Mode. (This should explain why we checked “Enable Source Level Debugging” on the Assembler tab Build Options during our previous lab).

Running LAB2

Note: Should you experience a problem with Code Composer, quit, then restart, and reload your project and program.

11. Hit the <F8> key or click the single step button on the vertical toolbar repeatedly and single-step through the program. When you are done close the Dis-Assembly window.

Features of Workspace

12. As you can probably tell, the windows in Code Composer can be moved around and resized. Typically, the default window arrangement is not a desirable one. To customize your display,

move the windows around where you want them. You also may want to right click on each window and select: `Float in Main Window`. This will allow each window to be visible when it is active.

12. Right click on the project window and select: `Hide`
13. To see the CPU registers. On the menu bar click:
`View → CPU Registers → CPU Registers`
14. Right click in the CPU Registers window and deselect “Allow Docking”. You can now move and resize the window as you like. Close the CPU Register window. Locate the “Register Window” button on the vertical toolbar, then click it to see if it appears.
15. You can edit the contents of any CPU register by double-clicking on it. Try this with AR7 now. Try typing in both hex and decimal numbers. Note that Code Composer will convert decimal to hex for you.
16. To see the disassembly window, find and click the “View Disassembly” button (at the bottom of the vertical toolbar) or click: `View → Dis-Assembly` on the menu bar.
17. As shown earlier, to open a Memory window, on the menu bar click: `View → Memory` or click the View Memory button on the vertical toolbar.
18. If you’re familiar with the Command Window used by previous TI debuggers, you can add one by clicking: `Tools → Command Window`
on the menu bar. Resize and dock or undock to your liking.
19. At the command line, type: `Step 20 ↵`
and watch the Code Composer actions. You should see the screen update to reflect the results of each individual “step”.
20. Type: `Reset ↵` or click `Debug → Reset DSP` from the menu bar.
Notice that reset takes you back to the reset vector located at 0000h. Restart, on the other hand, will return you to the entry label.
21. You can save your workspace by clicking:
`File → Workspace → Save Workspace`
and selecting a name. Make sure you save it in `C:\DSP24\LABS`. **DO NOT save your new workspace as the “default”**. When you restart Code Composer, you can reload “your” workspace by clicking:
`File → Workspace → Load Workspace`
and select your filename.
You may want to save a “generic” workspace rather than one that opens up a project. Make sure that you close the project before you save this workspace.

End of Exercise

Review

Review

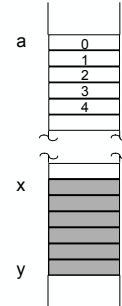
1. List the three assembler directives that are used for initialized sections
2. List the two assembler directives that are used for uninitialized sections
3. List the two parts of the linker command file

Solutions

Exercise 1 - Solution

```
; a = 0,1,2,3,4
; x = input array of length 5
; y = result
```

| | .data | |
|---|-------|-----------|
| a | .int | 0,1,2,3,4 |
| | .bss | x 5 |
| | .bss | y 1 |



Lab 2a: Solution - lab2.asm

```

                .bss    data,4
                .bss    coeff,4
                .bss    result,1

                .data
table          .int     1,2,3,4
               .int     8,6,4,2
               .int     0

                .text
start          nop
               nop
               nop
               nop
               b         start
```

Exercise 2 - Solution

```
MEMORY
{
    PAGE 0:
        EPROM:  org = 0000h,  len = 4000h
        SRAM:   org = 8000h,  len = 8000h
    PAGE 1:
        B2:     org = 0060h,  len = 0020h
        I_RAM:  org = 0200h,  len = 0200h
        DSRAM:  org = 8000h,  len = 8000h
}

SECTIONS
{
    .text: > EPROM    PAGE 0
    .bss:  > DSRAM    PAGE 1
    .data: > I_RAM    PAGE 1    /* I_RAM = B0 + B1 */
}
```

Lab 2b: Solution - lab2.cmd

```
MEMORY
{
    PAGE 0:
        EPROM:  org = 0000h,  len = 8000h

    PAGE 1:
        B2:     org = 0060h,  len = 0020h
        I_RAM:  org = 0200h,  len = 0200h
        SARAM:  org = 0800h,  len = 0800h
}

SECTIONS
{
    .text:      > EPROM    PAGE 0
    .data:      > I_RAM    PAGE 1
    .bss:       > B2      PAGE 1
}
```

Linker CMD File with Vectors

```

MEMORY
{
  PAGE 0:
    /* EPROM: org = 0000h, len = 2000h          */
    /* VECS:  org = 0000h, len = 0040h          */
    EPROM: org = 0040h, len = 1FC0h
    SRAM:  org = 8000h, len = 8000h
    PAGE 1:
      /* Data Memory */
      B2:   org = 0060h, len = 0020h
      I_RAM: org = 0200h, len = 0200h
      DSRAM: org = 8000h, len = 8000h
}

SECTIONS
{
  .text: > EPROM    PAGE 0
  .bss:  > DSRAM    PAGE 1
  .data: > B2       PAGE 1
  vectors: > VECS   PAGE 0
}

```

Exercise 3: Solution - mult.asm

mult.asm

| | | |
|------|-------------|------------|
| x | .ref | z, y |
| x | .def | c, mult |
| x K | .set | 1024 |
| x | .text | |
| 40 | mult LT z | 8000 |
| 41 | MPY y | 62 |
| 42 | ADD c | 60 |
| 43 | SACL z | 8000 |
| 44 | done B done | 44 |
| x | .data | |
| 60 c | .int | 4 * K 4096 |

Exercise 3: Solution - sum and vectors.asm

sum.asm

```

x      .ref  c, mult
x      .def  start, z, y
x      .text
46 start LACC x      61
47      ADD  c      60
48      SACL z      8000
49      B    mult   40

x      .data
61 x    .int  2
62 y    .int  7
8000    .bss  z, 1

```

vectors.asm

```

x      .ref  start
x      .sect  "vectors"
0      B    start 46

```

Lab 2c: Solution - lab2.asm

```

                .def  start

                .bss  data,4
                .bss  coeff,4
                .bss  result,1

                .data
table           .int  1,2,3,4
                .int  8,6,4,2
                .int  0

                .text
start          nop
               nop
               nop
               nop
               b      start

```

Lab 2c: Solution - vectors.asm

```
.ref    start
.sect   "vectors"
b       start
```

Lab 2c: Solution - lab2.cmd

```
MEMORY
{
    PAGE 0:
    VECS:   org = 0000h, len = 0040h
    EPROM:  org = 0044h, len = 7FBCh

    PAGE 1:
    B2:     org = 0060h, len = 0020h
    I_RAM:  org = 0200h, len = 0200h
    SARAM:  org = 0800h, len = 0800h
}

SECTIONS
{
    vectors: > VECS    PAGE 0
    .text:   > EPROM   PAGE 0
    .data:   > I_RAM   PAGE 1
    .bss:    > B2      PAGE 1
}
```

This page is left intentionally blank.

Addressing Modes

Introduction

This module will describe the various addressing modes available on the TMS320C240x. The ability to express constants, especially useful in the initialization process, is called immediate addressing. Direct addressing allows for general purpose memory access. Indirect addressing makes use of auxiliary registers to provide hardware support for various pointer functions, such as accessing data organized in arrays. Techniques for managing data pages, relevant to direct addressing, will be discussed.

Learning Objectives

Learning Objectives

- ◆ Use large and small constants
- ◆ Setup and use direct and indirect addressing modes
- ◆ Identify the best mode for a process
- ◆ Load, store, add and subtract values between memory and the accumulator

Module Topics

| | |
|--------------------------------------|-------------|
| Addressing Modes | 3-1 |
| <i>Module Topics.....</i> | <i>3-2</i> |
| <i>Memory Organization.....</i> | <i>3-3</i> |
| <i>Data Addressing Modes.....</i> | <i>3-4</i> |
| <i>Immediate Addressing.....</i> | <i>3-5</i> |
| <i>Direct Addressing</i> | <i>3-6</i> |
| <i>Indirect Addressing.....</i> | <i>3-10</i> |
| <i>Addressing Modes Review</i> | <i>3-15</i> |
| <i>Addressing Exercise.....</i> | <i>3-16</i> |
| <i>Lab 3: Addressing.....</i> | <i>3-17</i> |
| <i>Review.....</i> | <i>3-19</i> |
| Solutions | 3-20 |

Memory Organization

As was previously noted, the C240x memory map is made up of three 64K ranges for program, data, and I/O memory. A 64K-memory range requires a 16-bit address to uniquely identify each location within the range. The contents of all locations are treated as 16-bit words.

Program memory is generally addressed via 16-bit immediate values such as was seen in the branch instruction (e.g., B 1234h). I/O memory addressing will be presented in Module 7. After a review of the memory structures present on the C240x, this module will concentrate on how to express data values, or operands, as numerical constants or as components of the data memory map.

On-Chip Memory Modules

The C240x devices offer a variety of memory mixes, which include program ROM/Flash, Data RAM, and Dual-purpose RAM.

C240x-based devices may contain ROM which is enabled when the MP/ $\overline{\text{MC}}$ pin is taken low. The ROM, if selected, is present at the lowest addresses of program space.

Data RAM is composed of two blocks of memory located at 0060-007Fh and 0200-03FFh in the Data memory map. All C240x-based devices have these memories in common. An additional feature of these RAMs is that they are “dual access” memories, which are useful in implementing delay lines (and will be demonstrated in Module 7). These dual access RAMs are the most costly of the on-chip memories.

One dual access block, called *RAM Block 0*, is located at 0200-02FFh in Data memory at reset, and may be relocated to FF00-FFFFh in Program memory via the CNF (configure) bit under program control.

LF2407 Memory Map

| TMS320LF2407 Memory Map | | | |
|-------------------------|--|------|--|
| Hex | Program | Hex | Data |
| 0000 | Interrupts | 0000 | Memory-Mapped Registers |
| 0040 | Reserved (code security - LF240xA) | 0060 | On-chip DARAM B2 |
| 0044 | | 0080 | Reserved |
| | On-chip 32K Flash (External if MP/MC = 1) | 0200 | On-chip DARAM B0 (CNF = 0) or Reserved (CNF = 1) |
| 8000 | SARAM (2K) (PON = 1) or External (PON = 0) | 0300 | On-chip DARAM B1 |
| 8800 | External | 0400 | Reserved |
| FE00 | Reserved (CNF = 1) | 0800 | SARAM (2K) (DON = 1) or Reserved (DON = 0) |
| FF00 | On-chip DARAM B0 (CNF = 1) or External (CNF = 0) | 7000 | Non-EV Peripherals |
| FFFF | | 7400 | EV Peripherals |
| | | 7540 | Reserved |
| | | 8000 | External |
| | | FFFF | |

Data Addressing Modes

The C240x instruction set allows several modes for addressing data memory and expressing constants. In this module, we will concentrate on the following:

Data Addressing Modes

| Mode | Purpose |
|-----------------------------|---|
| Immediate (Constant) | Initialize registers, operate with constants |
| Direct (Paged) | Access data on a given page in any order |
| Indirect (Pointer) | Access data from arrays anywhere in data memory in an orderly fashion |

Instructions used in Module

- ◆ Addressing modes in this module will make use of the following instructions:

| | | |
|------|---|---|
| LACC | x | ;Load ACCumulator from <dma> "x" |
| SACL | y | ;Store ACcumulator Low half to <dma> "y" |
| ADD | a | ;ADD to accumulator value from <dma> "a" |
| SUB | b | ;SUBtract from accumulator value from <dma> "b" |

Immediate Addressing

In immediate addressing, the operand is part of the instruction word itself and is identified by the pound (#) symbol.

Small values may be expressed within a single program word, while large values require a second program word (and therefore, a second cycle to execute). Small, or “short,” immediate values are generally limited to 8 bits, but depending on the instruction in which they are used, may range from as small as a 1-bit constant to as large as 13 bits. The *TMS320C240x User’s Guide* presents a table of immediate instruction word widths.

Values which exceed the limit of a short constant become two-word, two-cycle operations on the C240x, but look identical from the programmers perspective, as seen in the example below:

```
ADD #12h           ; 0012h is added to the Acc, 1 cycle
ADD #3456h         ; 3456h is added to the Acc, 2 cycles
```

Long immediate instructions also allow a second operand to be specified: a shift value which can be used to position the 16-bit constant within a 32-bit register. Shift operations will be described and used in later modules.

For a complete list of the operations which support immediate addressing, refer to the *TMS320C240x User’s Guide*.

C240x Immediate Addressing Examples

Supports short and long constants

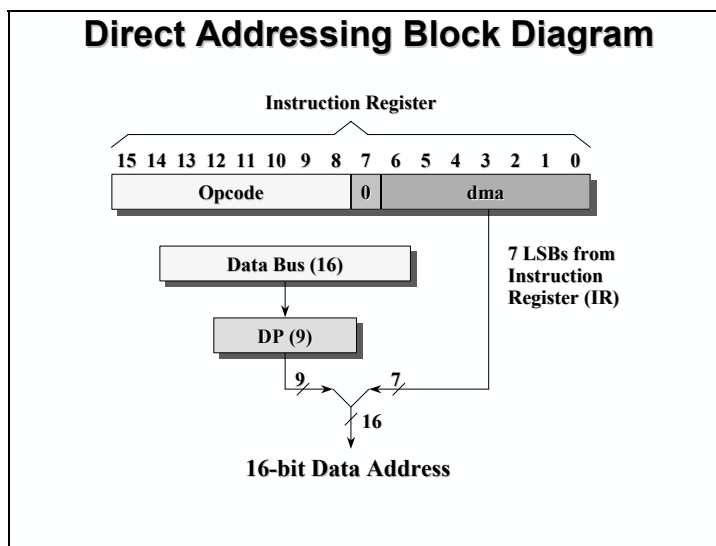
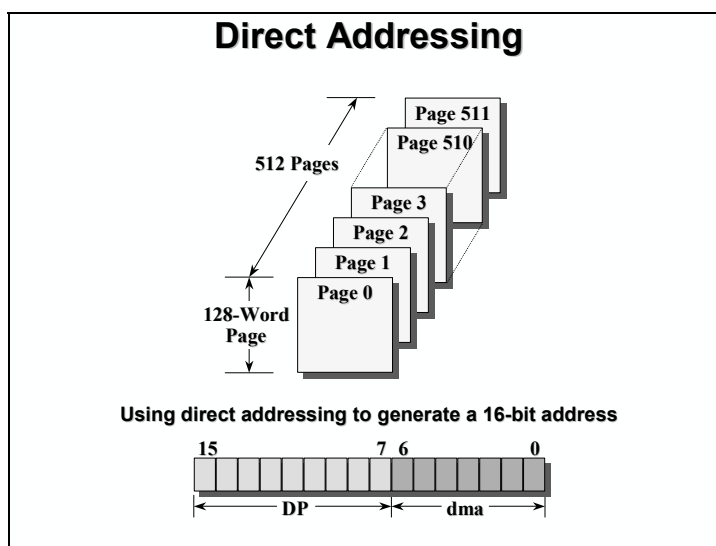
| Code | Program Memory | Words/Cycles |
|------------|----------------|--------------|
| ADD #12h | ADD# 12h | 1/1 |
| ADD #3456h | ADD# 3456h | 2/2 |

An example of the immediate addressing process is shown in the previous figure. Note that the data environment is not used in this type of instruction. A true Harvard machine derives operands from data space only — the ability to pass operands from program space is one of the reasons why the 320 are said to have a “modified” Harvard architecture.

Direct Addressing

Since the C240x has an instruction word size of 16 bits and a data address width of 16 bits, it would seem that attempting to directly use a data memory value would require two cycles; i.e., one to specify an operation, and the second to express the data memory address to be accessed (somewhat like the method used in long immediate addressing). This, however, would yield slow (two-cycle) access for direct addressing, which is undesirable in most systems. Instead, the C240x allows the user to specify an area of memory, and direct-addressed instructions operate within this reduced range of memory. This “paged” memory approach is common to many processors, as a compromise between memory range and speed.

In the C240x, a 9-bit Data Page (DP) register is used to specify the active area of memory. Thus, at any one time, 128 locations, on any one of 512 selectable pages, are active for direct addressing. These 128 locations may be addressed with only 7 bits, making single-cycle operation possible. Since the DP is a programmable register, the entire memory map is accessible 128 locations at a time.



Direct addressing, therefore, begins with initializing the DP, most commonly with the LDP (Load Data Page) instruction. Any number of operations may follow within the selected page.

Direct Addressing Instructions Example

- ◆ Direct addressing begins with initializing the data page pointer, and any number of operations may follow within a selected page

```

.data
x    .int    5
y    .int    4
     .bss    z,1
.text
LDP   #x
LACC  x
ADD   y
SUB   #2

```
- ◆ Accessing values on a different page requires reloading the data page pointer

```

LDP   #z
SACL  z

```
- ◆ How do we guarantee that x and y are in the same data page?

Direct Addressing Considerations

As such, the programmer is advised to group as much data for a given process on a single page, thus minimizing the time spent modifying DP, and maximizing the time spent in actual processing. This presents two more issues. First, as described in Module 2, the COFF tools used at TI are most beneficial when symbolic, as opposed to fixed, addresses are used. How then is the DP to be set, and how does the programmer know when a page boundary is to be crossed?

It turns out that the method for performing direct addressing on a symbolic address is quite simple, as noted in the following example.

```

.bss    x,1           ; allocate 1 (16 bit) location for the variable "x"
LDP     #x            ; load the DP with the page containing "x"
ADD     x             ; add to the acc. the contents of location "x"

```

In the above examples, it is important to note the use of the pound (#) in the LDP operand. Without the pound sign, data page would be loaded from the lower 9 bits of location x on the current page and, since the page has not yet been initialized, an effectively random value will be loaded to DP.

As to the second question raised above, how does the programmer know when a page boundary is going to be crossed? Consider the following example.

```

.bss    x,1           ; allocate a location for x
.bss    y,1           ; allocate a location for y
LDP     #x            ; be on the page that contains x
LACC    x             ; load ACC from loc'n x
ADD     y             ; add from loc'n y to ACC - but, is the page correct?

```

From this segment of code, it is not known that x and y is on the same data page. The following approaches exist for building reliable direct addressed code.

Keeping Variables on Same Data Page

- 1. Place an LDP before every direct operation**
- 2. Force the sections containing the variables / data onto a single page using the link command file**
- 3. Use FILE LIST linker option to split sections into specific files**
- 4. Use the contiguous page switch**
- 5. Use BLOCK or ALIGN linker options**

Note: Refer to the Student Guide for more details on each option

1. Place an LDP before every direct operation. Effective and fail-safe, but wastes processor time and code space.
2. Load `.bss` or `.usect` allocations (via the linker command file) into a memory structure on a single data page. Excellent for smaller sections, inadequate for systems which exceed 128 requests.
3. Use the FILE LIST option in the section declaration within the linker command file. A standard linker command file allocates `.bss` this way:

```
.bss {} : > RAM PAGE 1
```

The braces "{}" may contain a list of files to operate upon. Thus, if a system contained files `f1`, `f2`, and `f3`, and the first two fit on a single page, but the last one needed to be directed to a new page, the command file could be modified to read:

```
.bss {f1,f2} : > RAM1 PAGE 1
.bss {f3} : > RAM2 PAGE 1
```

In this way, `.asm` files could use simple `.bss` allocations, and the linker command file would manage their pagewise continuity.

4. Use the `.bss` Contiguous Page Switch, e.g., `.bss x, 5, 1`. This would force all five locations in the array `x` to reside on a single page. The linker will skip to the next data page if insufficient room is present on the current page for the entire allocation. Skipped locations may be backfilled by later `.bss` allocations which can fit in without being split up.

When using this method, note that the contiguous switch pertains only to the single allocation, thus:

```
.bss x, 5, 1
.bss y, 4, 1
```


may reside on two separate pages. To assure x and y on the same page, consider this approach:

```

        .bss      x, 9, 1
y        .set      x+5

```

By setting x as one large array, and declaring y as a point within the array, the linker will treat both as a single entity, on a single page. **In most cases, this is the optimal method for handling direct addressing.**

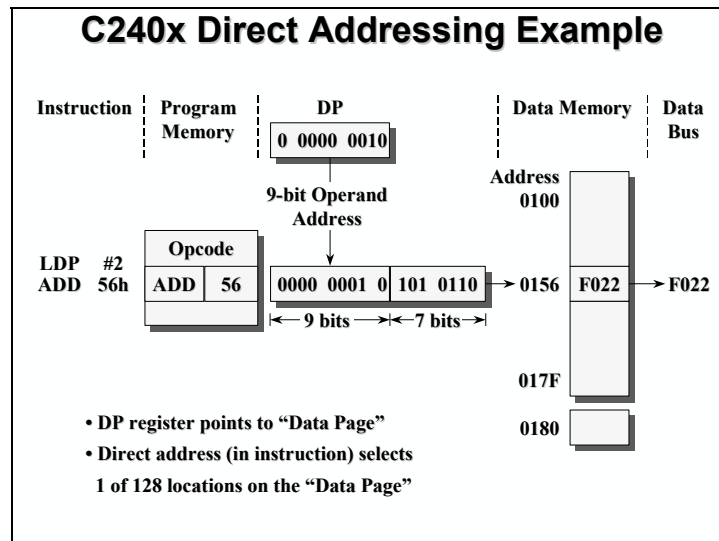
5. Use linker keyword BLOCK or ALIGN in linker command file.

Methods 3 – 5 require the programmer to set the data page when entering a new program. Thereafter, all .bss for the individual program will then be on a single page.

None of these concepts present the single best solution. The programmer should choose the method that yields the desired performance for the given system with the least amount of effort. Simple systems may be served by the earlier suggestions, while more demanding systems will invoke concepts from latter options, or a combination of several.

Direct Addressing Review

An example of direct addressing on the C240x is presented in the following figure.



Indirect Addressing

Indirect addressing is an efficient and powerful way to access data stored in lists, arrays, or other orderly groupings in memory. Unlike direct addressing, the address is not expressed in the instruction word, but instead is located in an Auxiliary Register (AR). The use of an AR provides several benefits. First, ARs are 16-bit registers, and thus may point to any location in the entire data memory map without the aid of the data page register. Additionally, the AR may be automatically incremented or decremented after an operand is read, so that a new datum is being pointed to for use in a later operation. The use of this feature makes the performance of iterative processes fast and easy.

Eight ARs are available on the C240x devices. For indirect addressing to be used, an AR must first be initialized and selected. ARs are initialized via the LAR (Load Aux Register) instruction. Initial selection of the active AR is via the MAR (Modify Aux Register) instruction. For greater performance, subsequent AR selections can be specified within any instruction using indirect addressing.

Indirect Addressing Operands

The use of the asterisk (*) is the indirect operator, indicating the use of the current AR to point to the data value to be used. When the plus sign (+) is added, it represents the auto-increment function, where the current AR is to be incremented by one after the operand is read. Similarly, the use of a minus sign (-) would specify an auto-decrement. Since the C240x devices provide dedicated hardware for implementing auto-increment/-decrement operations, no extra cycle time is required for this operation.

Indirect Addressing Sequence

ADD *+, 1, AR0

- | | |
|---------------|---|
| 1. * | ARP selects AR to address operand |
| 2. 1 | Shift operand left 1-bit optional, 0 default |
| 3. ADD | Operate on shifted operand |
| 4. + | Modify AR value - optional |
| 5. AR0 | New ARP value - optional |

Indirect addressing allows 16-bit registers to be used as pointers to data memory. It is frequently used to operate on arrays of data, and includes built-in hardware to implement several forms of auto-increment or auto-decrement functions (and new AR selection options) within the operation as indicated in the following figure.

C240x Indirect Addressing Options

| Option | Function |
|--------|--|
| * | Does not change current AR |
| ++ | Increments current AR |
| -- | Decrements current AR |
| *0+ | Adds AR0 to current AR |
| *0- | Subtracts AR0 from current AR |
| *BR0+ | Adds AR0 to current AR with reverse carry propagation |
| *BR0- | Subtracts AR0 from current AR with reverse carry propagation |

Indirect Addressing Example

Consider the following example demonstrating the use of indirect addressing to help illustrate the process. Allocate two arrays in memory, x and y, each containing four values. One AR can be initialized to point to x and another to y. To add all these values together, it would first be necessary to make one AR active for summing the x values, and then have the second AR be active for summing the y values. The following code implements this process.

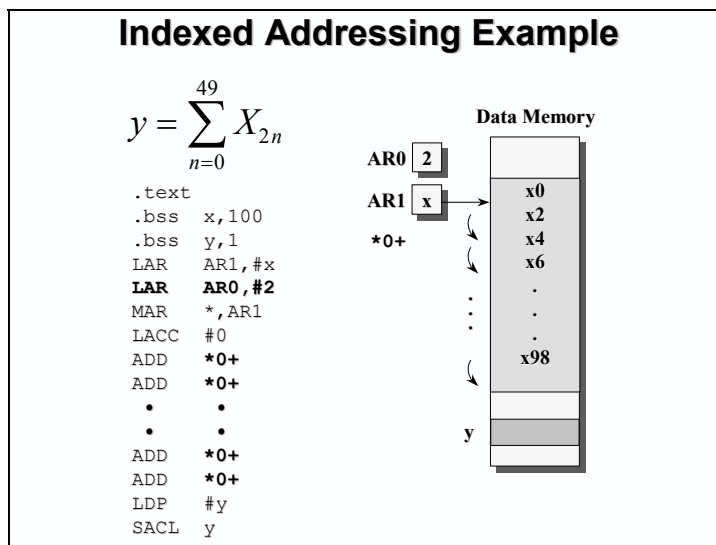
Indirect Addressing Example

```
.bss x,4      ;allocate 4 locations for the "x" array
.bss y,4      ;allocate 4 locations for the "y" array
.text
LAR  AR2,#x   ;load AR2 with the first "x" address
LAR  AR5,#y   ;load AR5 with the first "y" address
MAR  *,AR2    ;LARP AR2
LACC ++       ;load ACC using AR2, then increment AR2
ADD  ++       ;add to ACC using AR2, then increment AR2
ADD  ++       ;add to ACC using AR2, then increment AR2
ADD  *,0,AR5  ;add to ACC using AR2, then make AR5 active
ADD  ++       ;add to ACC value pointed to by AR5 (y),inc AR5
ADD  ++       ;add and increment
ADD  ++       ;add and increment
ADD  ++       ;add and increment
```

Note that when AR5 is made active, a 0 had been specified as the second operand. This is because instructions which operate on the Accumulator offer a shift option in the second operand field. Since no shift was needed in this instance, a 0 (the default value) was inserted to get to the third operand field — the NARP (New ARP) field. From this discussion, it is apparent that a single instruction is capable of finding, shifting, and using an operand; then operating on the current AR; and then selecting a new ARP, as shown in the above figure.

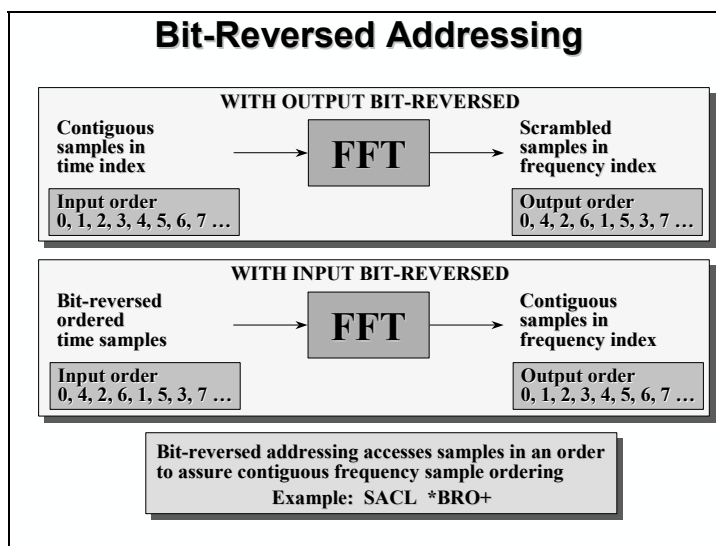
Indexing with Indirect Addressing

Sometimes it is desirable to be able to increment/decrement by values other than one. In these instances, an index register is required to specify the step size to be used. On the C240x, AR0 can be used as either a pointer or an index value on any other pointer. Use of the index during auto-increment/auto-decrement is specified by adding a 0 (shorthand for “AR0”) before the modifier, as in *0+ or *0-.



Bit-Reversed Addressing

Bit-reversed addressing is a very efficient means of addressing in Fast Fourier Transforms (FFTs). The FFT is a process for converting information in the time domain to information in the frequency domain, and relies upon a “butterfly” operation as the core mathematical procedure.



Notice in the figure above that the process causes the outputs to be out of sequence. Reordering the array requires significant amounts of processor time, or the use of external logic. The C240x

has a form of addressing which eliminates the need for address correction, by putting the out-of-order results in their proper locations *during* processing of the FFT. No extra time or hardware is required, allowing a much more efficient FFT implementation. To best understand this addressing mode, note the order of the outputs as binary numbers. At first, they seem random, but notice the pattern that exists if the number is read as if the order of most significant to least significant bit were reversed: it is (read backwards) a simple increment of one, with the carry operations propagating to the right. This addressing operation, sometimes called *reverse carry propagation* is much like a “mirror image” of the normal addressing order. Bit-reversed addressing, as used by the C240x, acts like placing the reversed addresses generated by the FFT in a mirror — they are read normally. To implement this “double mirror image” requires that the array counting value be placed in AR0, and that any other AR be used as the addressing register, with the “*BR0+” or “BR0-” mode selected. The array counting value is simply a 1 in the correct bit position for the increment process. Use the value N/2 for an N-size array.

The actual code to implement the FFT may be found on the TI bulletin board, and will not be discussed here.

One further detail needs to be established for bit-reversed addressing to function properly; i.e., the FFT array must be established at a “0” starting address, or, as it turns out, an address with a sufficient number of LSBs equal to zero. In this example (an 8-point FFT), three LS bits are used in the bit-reversed addressing and the 13 remaining MSBs remain unchanged. Thus, any data memory address with three LS bits at zero would be an acceptable starting point. The need for a specific type of RAM for the FFT calls for the use of a `.usect` in the `.asm` file. Specifying the number of LS zero bits is taken care of in the linker command file with the `align` directive, as indicated in the following figure.

Bit-Reversed Addressing ASM and LINKER CMD Files

ASM File (excerpt)

```

BASE .usect  "fft_array",8 ;allocate 8 locations starting at BASE
      .mmregs                ;allow use of MMRegister names
      .text
LAR    AR0, #0100b ;store to AR0
LAR    AR1,#BASE   ;AR1 points to top of FFT array
MAR    *,AR1       ;AR1 is the active pointer
RPT    #7          ;8 iterations
IN     *BR0+,2     ;read a data sample into array (bit rev'd)

```

LINKER CMD File (excerpt)

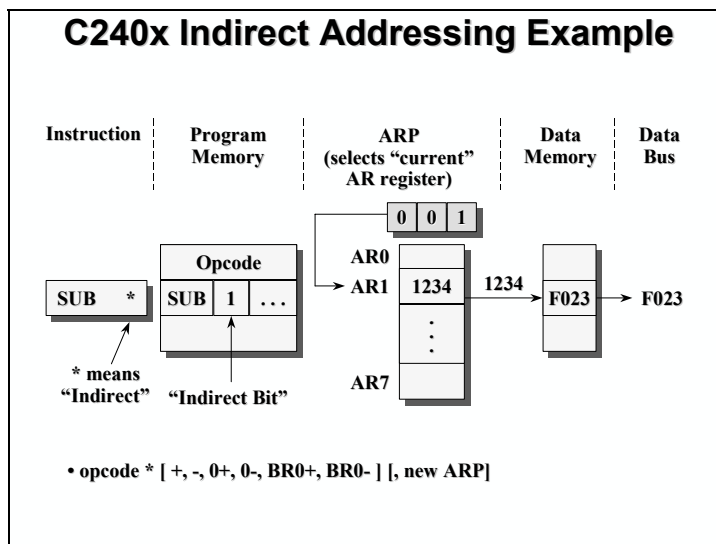
```

SECTIONS {
vectors:      {} >  VECS, PAGE 0
.text:        {} >  ROM, PAGE 0
fft_array    align(8): {} >  RAM, PAGE 1
.bss:         {} >  RAM, PAGE 1
}

```

Indirect Addressing Review

An example of indirect addressing on the C240x is presented in the following figure.



Indexing the Auxiliary Registers

ARP = 101b AR (ARP) = AR5 (AR5) = 0200h (AR0) = 0008h

Successive indexing operations yield the following in AR5:

| | * | *+/*- | *0+/*0- | *BR0+/*BR0- | FFT Resequencing |
|------|------|-------|---------|-------------|------------------|
| AR5 | 0200 | 0200 | 0200 | 0200 | x(0) ← |
| | 0200 | 0201 | 0208 | 0208 | x(8) |
| | 0200 | 0202 | 0210 | 0204 | x(4) |
| | 0200 | 0203 | 0218 | 020C | x(12) |
| | 0200 | 0204 | 0220 | 0202 | x(2) |
| | 0200 | 0205 | 0228 | 020A | x(10) |
| | 0200 | 0206 | 0230 | 0206 | x(6) |
| | 0200 | 0207 | 0238 | 020E | x(14) |
| | 0200 | 0208 | 0240 | 0201 | x(1) |
| | 0200 | 0209 | 0248 | 0209 | x(9) |
| | 0200 | 020A | 0250 | 0205 | x(5) |
| | 0200 | 020B | 0258 | 020D | x(13) |
| | 0200 | 020C | 0260 | 0203 | x(3) |
| | 0200 | 020D | 0268 | 020B | x(11) |
| | 0200 | 020E | 0270 | 0207 | x(7) |
| | 0200 | 020F | 0278 | 020F | x(15) |
| 0200 | 0210 | 0280 | 0200 | | |

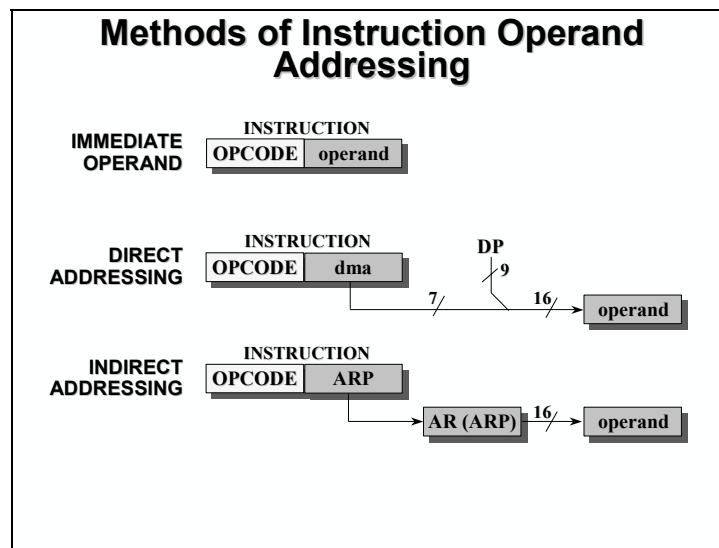
Addressing Modes Review

Three addressing modes have been presented. Each is best suited for different purposes.

Immediate addressing is best for providing constants at initialization time.

Indirect addressing is a powerful addressing mode, best suited to iterative and tabular operations, where the increment/decrement function becomes quite useful.

Direct addressing may be considered the best general-purpose addressing mode. Given an understanding of the paged-memory concept, direct addressing is a very easy-to-use mode, is simpler to debug, and is as fast as indirect addressing in operation (both are single-cycle operations). With direct addressing, no time penalty occurs for operands selected in any order on a given page; however, with indirect addressing, the locations used must be in order.



Addressing Exercise

Before beginning the labs, the details of addressing will be exercised with the segment of code below. You are to read the code on the left, use the memory as defined above, and perform the program as you expect the C240x would. As is often the case in program development, look out for errors.

| Addressing Exercise | | | | | | |
|--|----|-----|-----|-----|-----|-----|
| <div> <div> Address/Data (hex) </div> <div> <div>Block B2</div> <div> DP=0 <div> 60h 10 61h 120 62h </div> </div> </div> <div> <div>Block B0</div> <div> DP=4 <div> 200h 100 201h 60 202h 40 </div> </div> </div> <div> <div>Block B1</div> <div> DP=6 <div> 300h 10 301h 30 302h 60 </div> </div> </div> </div> | | | | | | |
| Program | DP | ARP | AR0 | AR1 | AR2 | ACC |
| LDP #0 | | | | | | |
| MAR *,AR1 | | | | | | |
| LAR AR0,#2 | | | | | | |
| LAR AR1,#200h | | | | | | |
| LAR AR2,#300h | | | | | | |
| LACC 61h | | | | | | 120 |
| ADD ** | | | | | | 220 |
| SUB 60h,1 | | | | | | 200 |
| ADD **,AR2 | | | | | | 260 |
| LDP #6 | | | | | | |
| ADD 1 | | | | | | 290 |
| ADD **,4 | | | | | | 390 |
| SUB **,0,AR1 | | | | | | 360 |
| SUB #32 | | | | | | 340 |
| ADD *0-,0,AR2 | | | | | | 380 |
| SUB *0- | | | | | | 320 |
| SACL 62h | | | | | | |

Lab 3: Addressing

➤ 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.

In this lab, we will initialize the `.bss` arrays allocated in the previous lab with the contents of the `.data` table. How is this best accomplished? Consider the process of loading the first `.data` value into the accumulator and then storing this value to the first `.bss` 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?

Given these considerations, perform the following procedure.

➤ Procedure

Copy Files, Create Make File

1. Using Code Composer, open `LAB2.CMD` in `C:\DSP24\LABS` and save it as `C:\DSP24\LABS\LAB3.CMD`.
2. Open `LAB2.ASM` and save it as `LAB3.ASM`.
3. Create a new project called `LAB3.MAK` and add `LAB3.ASM`, `VECTORS.ASM` and `LAB3.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` → `Options` on the menu bar. Select the Assembler tab. In the middle of the screen check “Enable Source Level Debugging”. Next, select the Linker tab. In the middle of the screen select “No Autoinitialization”. Create a map file named `LAB3.MAP`. Select `OK` to save the Build Options.

Initialize Allocated RAM Array from ROM Initialization Table

4. Edit `LAB3.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 `.text` section.

- Initialize pointers to the beginning of the `.data` and `.bss` arrays.
- Transfer the first value from `.data` to the `.bss` 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`
- setup the appropriate indirect addressing registers

5. It is good practice to trap the end of the program (i.e. use either `"end: B end"` or `"end: B start"`). Save your work.

Build and Load

6. Click the "Rebuild All" 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.
7. Load the output file onto the target. Click:

File → Load Program...

Then reset the DSP by clicking on: Debug → Reset DSP

Right click on the `VECTORS.ASM` window and select `Mixed Mode` to debug using both source and assembly.

Note: Code Composer can automatically load the output file after a successful build. On the menu bar click: Option → Program Load... and select: "Load program after build", then click OK.

8. 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 – TI style" will give you more viewable data in the window. Additionally, it is useful to watch the CPU (and Status) registers. Open the CPU register (and Status) by using the "View CPU Registers". Deselect "Allow Docking" and move/resize the window as needed. Check to see if the program is working as expected.

You might want to use your workspace from the previous lab. Look under File → Recent Workspaces on the menu bar to select your saved workspace quickly. If needed, reload your project.

End of Exercise

Review

Review

1. What is the difference between data memory and program memory?
2. What is direct addressing?
Give an example.
3. What is immediate addressing?
Give an example.
4. What is indirect addressing?
Give two examples.
5. How many auxiliary registers are there?

Solutions

Addressing Exercise - Solution

| <u>Address/Data (hex)</u> | | Block | B2 | | Block | B0 | | Block | B1 |
|---------------------------|--|-------|-----|--|-------|-----|--|-------|----|
| DP=0 | | 60h | 10 | | 200h | 100 | | 300h | 10 |
| | | 61h | 120 | | 201h | 60 | | 301h | 30 |
| | | 62h | | | 202h | 40 | | 302h | 60 |

| Program | DP | ARP | AR0 | AR1 | AR2 | ACC |
|---------------|----|-----|-----|-----|-----|-----|
| LDP #0 | 0 | | | | | |
| MAR *,AR1 | | 1 | | | | |
| LAR AR0,#2 | | | 2 | | | |
| LAR AR1,#200h | | | | 200 | | |
| LAR AR2,#300h | | | | | 300 | |
| LACC 61h | | | | | | 120 |
| ADD ** | | | | 201 | | 220 |
| SUB 60h,1 | | | | 202 | | 200 |
| ADD **,AR2 | | 2 | | | | 260 |
| LDP #6 | 6 | | | | | |
| ADD 1 | | | | | 301 | 290 |
| ADD **,4 | | | | | 302 | 390 |
| SUB **,0,AR1 | | 1 | | | | 360 |
| SUB #32 | | | | 200 | | 340 |
| ADD *0-,0,AR2 | | 2 | | | | 380 |
| SUB *0- | | | | | 300 | |
| SACL 62h | | | | | | 320 |

```

;SOLUTION FILE FOR LAB3.ASM

        .def      start
        .bss      data,4
        .bss      coeff,4
        .bss      result,1

        .data
table:   .int      1,2,3,4
        .int      8,6,4,2
        .int      0

        .text
start:   LAR        AR1,#table      ;AR1 is the source pointer
        LAR        AR2,#data       ;AR2 is the destination pointer
        MAR        *,AR1
        LACC       **,0,AR2        ;1
        SACL       **,0,AR1
        LACC       **,0,AR2        ;2
        SACL       **,0,AR1
        LACC       **,0,AR2        ;3
        SACL       **,0,AR1
        LACC       **,0,AR2        ;4
        SACL       **,0,AR1
        LACC       **,0,AR2        ;5
        SACL       **,0,AR1
        LACC       **,0,AR2        ;6
        SACL       **,0,AR1
        LACC       **,0,AR2        ;7
        SACL       **,0,AR1
        LACC       **,0,AR2        ;8
        SACL       **,0,AR1
        LACC       **,0,AR2        ;9
        SACL       **,0,AR1
        B          start

```

Basic Programming Techniques

Introduction

This module will present three topics: program control, ALU operations, and the use of the multiplier. Program control offers a variety of branch and subroutine call instructions, of which several have conditional operation. Details of the ALU will be examined, and the operation of the multiplier will be explained.

Learning Objectives

Learning Objectives

- ◆ Write C240x code to perform basic arithmetic
- ◆ Perform simple branch, loop control and subroutine operations
- ◆ Use the accumulator to load, store, add, and subtract 16-bit values from data and program memory
- ◆ Use the multiplier to implement sum-of-products equations

Module Topics

| | |
|--|-------------|
| Basic Programming Techniques | 4-1 |
| <i>Module Topics.....</i> | <i>4-2</i> |
| <i>Program Control.....</i> | <i>4-3</i> |
| Branches | 4-3 |
| Subroutines | 4-6 |
| <i>Central Arithmetic Logic Unit</i> | <i>4-8</i> |
| Arithmetic Logic Unit | 4-9 |
| Multiplier..... | 4-11 |
| <i>Lab 4: Basic Programming.....</i> | <i>4-14</i> |
| <i>Review.....</i> | <i>4-16</i> |
| Solutions | 4-16 |

Program Control

The C240x provides several methods of controlling the flow of program execution. Normal program flow is sequential; that is, the processor fetches and executes the next instruction in program memory. When it is necessary to break sequential execution, you can use branches, calls, or traps and soft interrupts. Branches transfer control to any location in program memory. Calls also transfer control to any program memory location, but allow easy return to the original program sequence with the use of the RETURN (RET) instruction. Traps and soft interrupts are special forms of calls.

Branches and calls may be either unconditional, i.e., always taken when encountered; or may be conditional, where the branch or call is made is dependent upon the processor state.

A matrix of the standard program control instructions and conditional codes appears below. The text that follows will explain the details of each item individually.

| Program Control | | | | | |
|-----------------|------------------|------|-------------|------|-------------|
| BRANCH | | CALL | | RET | |
| B | next | CALL | sub | RET | |
| BACC | | CALA | | — | |
| BCND | next,cnd,cnd,... | CC | sub,cnd,... | RETC | cnd,cnd,... |

| Conditional Codes | | | |
|-------------------|---------|-----|---------|
| EQ | ACC = 0 | NEQ | ACC ≠ 0 |
| LT | ACC < 0 | GT | ACC > 0 |
| LEQ | ACC ≤ 0 | GEQ | ACC ≥ 0 |
| C | C = 1 | NC | C = 0 |
| OV | OV = 1 | NOV | OV = 0 |
| TC | TC = 1 | NTC | TC = 0 |
| BIO | BIO = 0 | UNC | uncond. |

Branches

Branch instructions transfer control to any location in program memory. Most branch instructions are two words long and execute in four cycles.

Unconditional Branch

An unconditional branch is always taken. In addition to modifying the program counter, the unconditional branch also provides the programmer the opportunity to use the ARAU to modify the current auxiliary register (AR) and auxiliary register pointer (ARP). In an assembly language program, the unconditional branch instruction is written as shown:

```
[label] B pma [, {ind} [, next ARP]]
```

where: $0000h \leq pma \leq 0FFFFh$ and $0 \leq \text{next ARP} \leq 7$

The first word of the instruction is the branch opcode. The second word is the program memory address (<pma>). When a normal branch instruction is decoded, any ARAU operation specified by indirect addressing is performed. When the instruction is executed, the PC is loaded with the <pma> and program execution begins at the new address specified by the <pma>.

Conditional Branch

Branches can also be made conditional upon the processor state. Conditional branches are coded as follows:

```
[label]    BCND pma, [cond1][,cond2][,...]
```

If all conditions are met, the PC will be loaded with the <pma>. If any condition is false, the PC will be loaded with the address following the branch address and execution will continue.

Conditional branches, when taken, require four cycles to execute. When a conditional branch is not taken, execution occurs in two cycles.

Testing $\overline{\text{BIO}}$ and TC are mutually exclusive. All other combinations of conditions may be specified, although not all combinations are meaningful.

Dynamic Branch

BACC is a one-word instruction which loads the PC from the lower 16 bits of the accumulator. The BACC instruction provides for a branch to a runtime programmable address. This instruction is useful if a program may take one of many branches based upon some condition. An example of this instruction's use is demonstrated in Module 5, Advanced Programming.

Decrement and Branch

BANZ is a special case of conditional branch instruction. It may be used to implement loops which will be executed a number of times. The BANZ instruction is coded as follows:

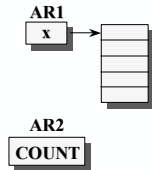
```
[label]    BANZ pma [{ind}[, next ARP]]           ;branch if auxiliary register is
                                                    ;not equal to zero and decrement
                                                    ;auxiliary register
```

The BANZ instruction is conditional on the contents of the current AR before the BANZ instruction is decoded. If the contents of the current AR is not zero, the branch is taken. When the current AR=0, control passes to the next instruction in program memory. The current AR and the ARP are modifiable as noted. If no modification is specified, the AR modification will default to a decrement by one. When the BANZ instruction is used at the end of a loop, N+1 iterations of the loop will be executed if the auxiliary register is initialized to N outside the loop. A caveat regarding the use of the BANZ instruction in the C240x is that specifying a modification other than a decrement by one may result in a loop which will never terminate. (Count value must reach zero to fall out of the loop.)

BANZ Loop Control

BANZ Loop Control Example

$$y = \sum_{n=1}^5 x_n$$



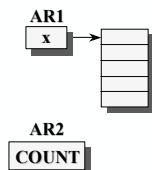
```

.bss x,5
.bss y,1
.text
LDP #y
LAR AR1,#x
LAR AR2,#4
LACL #0
loop: MAR *,AR1
      ADD ** ,0,AR2
      BANZ loop
      SACL y

```

BANZ Loop Control (optimized)

$$y = \sum_{n=1}^5 x_n$$



```

.bss x,5
.bss y,1
.text
LDP #y
LAR AR1,#x
LAR AR2,#4
LACL #0
MAR *,AR1
loop: ADD ** ,0,AR2
      BANZ loop,*-,AR1
      SACL y

```

Subroutines

A subroutine call differs from a branch in that the call is intended to provide a temporary departure from the sequential program execution. Therefore, when the call is executed, a return address is saved at which the program can resume execution after the subroutine is completed. When any of the call instructions are executed, the address of the next instruction following the call is pushed onto the 'C2xx's hardware stack. Like branches, calls can be immediate, and can be either unconditional or conditional.

Calling a Subroutine

The CALL instruction is an unconditional call. Like the unconditional branch instruction, in addition to altering the PC contents, CALL can also modify the AR and ARP via the ARAU. The call instruction is coded as follows:

```
[label]    CALL    pma [, {ind} [, next ARP]]
```

where:

$$0000h \leq pma \leq 0FFFFh$$

$$0 \leq \text{next ARP} \leq 7$$

The address of the instruction to be executed following the subroutine execution is pushed onto the stack. The 'C2xx stack is an eight-level hardware stack which is used to save only return addresses. It is shared by both calls and interrupts. Systems which may exceed eight levels of call/interrupt depth must save the hardware stack to data memory as part of their context save/restore routine. This process is described in detail within the interrupt module in this workshop.

Conditional Call

Like branches, calls can also be conditional. The conditional call instruction is coded as follows:

```
[label]    CC pma, [cond1][,cond2][,...]
```

Conditional calls operate on the same condition codes as BCND and, likewise, will be executed if **all** the selected conditions are true. If **any** of the selected conditions is false, control is passed to the instruction after the CC instruction. The CC instruction is subject to the same limitations as the BCND instruction.

Dynamic Call

The CALA instruction allows calls to a program memory address contained in the accumulator. When the CALA instruction is executed, the return address is pushed onto the stack, and the lower 16 bits of the accumulator are loaded into the PC.

Returning from a Subroutine

As mentioned earlier, the call instructions save a return address at which program execution will resume after the subroutine completes execution. A routine which is entered via any of the call instructions should normally be exited via a return instruction. The 'C2xx provides both unconditional and conditional returns from subroutines.

The RET instruction performs an unconditional return from subroutine. RET is coded as shown below.

```
[label]    RET                ;unconditional return
```

When the RET is executed, the contents of the top of stack are popped to the program counter.

Conditional Return from a Subroutine

The RETC (return conditionally) instruction functions like the RET instruction, except that the return is only taken if specified conditions are met. RETC is coded as shown below.

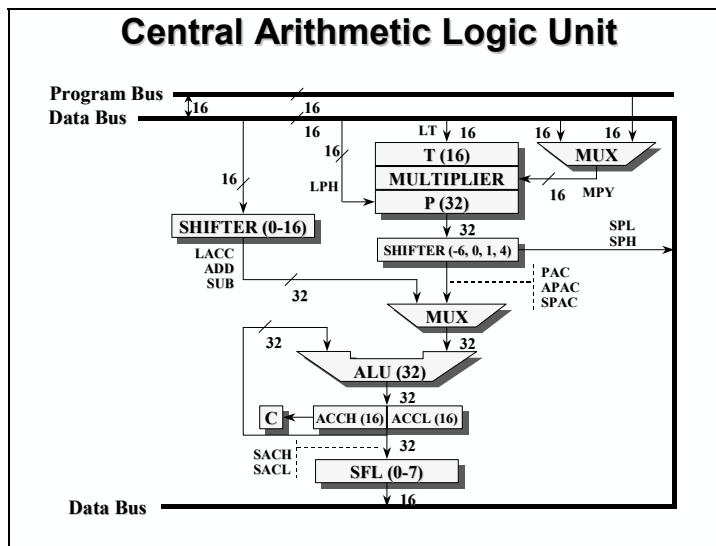
```
[label]    RETC [cond1][,cond2][,...]          ;conditional return
```

The condition codes and operation are identical to BCND.

Central Arithmetic Logic Unit

The Central Arithmetic Logic Unit (CALU) is where values from data and/or program memory may be operated upon arithmetically (addition, subtraction, and multiplication). The registers and functional components of the CALU operate together to allow fast implementation of basic mathematical processes.

In this module, several components of the CALU will be examined, with the objective of being able to understand the workings of these components and to write code which efficiently implements arithmetic equations.



Arithmetic Logic Unit

The Arithmetic Logic Unit (ALU) is a functional block which can perform addition, subtraction, and Boolean functions. The results of the ALU appear at the Accumulator (ACC) register. Inputs to the ALU come from the ACC and either memory (data or program) or a register (Product). The 32-bit ALU and ACC operate on signed or unsigned values of up to 32 bits in a single cycle. The following are examples of instructions often used to operate on the accumulator:

```
LACC    #123                ; ACC = 123 after instruction.
                                ; Old value of ACC is overwritten
ADD     x                    ; ACC = 123 + contents of <dma> "x"
SUB     *                    ; ACC = 123 + @x - contents of <dma>
                                ; pointed to by current AR
```

All three addressing modes shown here may be used in any of these instructions. Additionally, since the ACC is a 32-bit register and the operands from memory are 16-bit, a second operand is permitted after the data operand that indicates the number of bits up from the low accumulator with which to align the LSB of the operand. In this module, no shift will be expressed, thus, the default of zero will place the LSB of the operand at the LSB of the accumulator. The “shift field” and its benefits will be discussed further in Module 6.

| LACC vs. LACL | | |
|---|---|--|
| General Comments | <u>LACC</u> | <u>LACL</u> |
| | <ul style="list-style-type: none"> • 0-16-bit shift allowed • ACC is sign-extended w/sxm=1 | <ul style="list-style-type: none"> • No shift allowed • No sign extension, ACCH=0 |
| Immediate Addr. | <div style="border: 1px solid black; padding: 2px; display: inline-block;"> LACC shift #value </div> <ul style="list-style-type: none"> • Up to 16-bit value • Always 2w/2c | <div style="border: 1px solid black; padding: 2px; display: inline-block;"> LACL #val </div> <ul style="list-style-type: none"> • Up to 8 bits only • Always 1w/1c |
| Dir/Indir Addr. (always 1w/1c) | LACC y, 15 LACC *, 3 | LACL y LACL * |
| <p>Suggestion: Use LACL to load the lower ACC with immediate values of 8 bits or less, or when you do not want sign-extension regardless of the SXM bit setting; otherwise, use LACC</p> | | |

Unlike most instructions using immediate addressing, LACC #const always uses two words (therefore two cycles) regardless of the length of the constant. For example:

```
LACC    #2                  ; two words, two cycles
LACC    #472Ah              ; two words, two cycles
```

both use two words of program space.

If you are loading an 8-bit (or smaller) unsigned value, you can use LACL instead:

```
LACL    #2                  ; one word, one cycle
```

LACL loads the low half of the accumulator with the value (ACCL = value) and clears the high half (ACCH = 0) in a single cycle. This is an excellent time saver, but the user must remember that **LACL does not support negative numbers with normal sign extension through the high accumulator**.

| Using SPLK | | |
|---|------------------|----------------|
| SPLK means Store Parallel Long Immediate | | |
| Problem: Store a 16-bit immediate value to memory | | |
| Solution | Code | Words / Cycles |
| Using LACC | LACC #1234h | 2 / 2 |
| | SACL mem1 | 1 / 1 |
| | | 3 / 3 |
| Using SPLK | SPLK #1234h,mem1 | 2 / 2 |

Benefits:

- SPLK does not use the ACC
- If immediate value is 9-16 bits, you save 1w/1c
- The mnemonic is simpler to read

Note: If immediate value is unsigned and £ 8 bits, you could use LACL instead of LACC and save the extra cycle

Another useful C240x instruction is SPLK (Store Parallel Long Immediate) which stores an immediate value to a memory location of your choice. Consider the following example:

```
LACC    #value           ; load acc with 16-bit immediate value
                          ; (two words, two cycles)
SACL    mem_loc          ; store acc to register (one word,
                          ; one cycle)
```

This process requires a total of three words of program space and a minimum of three cycles. Using SPLK can reduce this operation to two words and two cycles:

```
SPLK    #value, mem_loc  ; store 16-bit value to memory
```

If the value being stored is 8 bits or smaller, there is no advantage to using SPLK vs. LACL/SACL, other than personal preference.

When a calculation is completed, it is often desirable to store the results back to data memory. Since memory is only half as wide as the accumulator, it is convenient to be able to store the ACC to memory in halves by using the following two instructions:

```
SACL    y                ; low 16-bits of ACC written to
                          ; <dma> "y"
SACH    z                ; high 16-bits of ACC written to
                          ; <dma> "z"
```

These instructions may also use the indirect addressing mode. Additionally, it is possible to extract any 16 contiguous bits from the ACC via the output shifter, which is controlled by a second operand field in the SACH and SACL instructions. As noted before, no shifts will be expressed in this module, giving a default of zero. Different cases are shown in Module 6.

Multiplier

Traditionally, multiplication has required many instruction cycles to implement on a microprocessor, slowing the rate at which a multiply-intensive algorithm could be solved. This is especially true in DSP systems which are composed of long sum-of-products type equations. One of the key features of the C240x devices is the presence of dedicated hardware to perform single-cycle multiplication.

The multiplier is fed two 16-bit values and produces a 32-bit result which is placed in a 32-bit product register. The result in the product register is usually passed to the accumulator via a 32-bit bus before another product is formed. In the process of accumulating products, a shift may be preselected. In this module, the shifter will be left at zero shifts (the reset condition). Other shift values and their benefits will be presented in a later module. Multiplication is a signed (two's complement) function, unless the “multiply unsigned” instruction is specified.

In the sections that follow, several methods of performing this process on the C240x will be presented.

Basic Use of the Multiplier

As noted, two values must be fed to the multiplier. In the simplest case, the first operand is routed to the Temporary (T) Register (also referred to as TR or TREG) using the Load Temporary (LT) instruction. LT operands may be specified via direct or indirect (but not immediate) addressing.

The second operand is specified with the Multiply (MPY) instruction via direct, indirect, or immediate addressing (a 13-bit “short” constant). Upon execution of the MPY instruction, a 32-bit product is calculated and placed in the Product (P) Register (also referred to as PR or PREG).

Note that since the TREG and multiplier input are both 16-bit inputs, no shift option is offered with the LT or MPY instructions.

Subsequent multiplications overwrite the previous products; therefore, it is necessary to move the result out of the product register before it is overwritten. Most often, the product is part of a “sum-of-products” calculation, where the product is to be summed with other products, thus, the ability to move the product to the accumulator is desirable. This can be accomplished in several ways by using the following instructions:

| | |
|------|-----------------|
| PAC | ; ACC = P |
| APAC | ; ACC = ACC + P |
| SPAC | ; ACC = ACC - P |

Each of the above instructions operates on a 32-bit value in a single cycle using the dedicated 32-bit path between the P and ACC registers. Given the above instructions, a sum-of-products equation can be solved as shown in the following figure.

Sum-of-Products

$$Y = A \cdot X1 + B \cdot X2 + C \cdot X3 + D \cdot X4$$

```

LT   X1      ;T = X1
MPY  A       ;P = A*X1
PAC                ;ACC = A*X1
LT   X2      ;T = X2
MPY  B       ;P = B*X2
APAC                ;ACC = A*X1 + B*X2
LT   X3      ;T = X3
MPY  C       ;P = C*X3
APAC                ;ACC = A*X1 + B*X2 + C*X3
LT   X4      ;T = X4
MPY  D       ;P = D*X4
APAC                ;ACC = Y
SACL Y

```

Enhanced Use of the Multiplier

As noted, the sum-of-products operation is often the core process in DSP systems, which frequently have their performance indicated by operating speed. In the above example, each “tap” (multiply-accumulate operation) requires three instructions; therefore, a performance measure, or benchmark, of 3 cycles/tap can be expected. Although a DSP is much faster than a standard microprocessor, even faster operation is often desirable. This is made possible by merging the LT and APAC operations, as shown in the figure above in the example of the LTA (Load Temporary and Accumulate) instruction. Three versions of LT with product accumulation are available:

```

LTP      x                ; ACC = P, T = contents of location x,
                        ; or "@x"
LTA      *                ; ACC = ACC+P, T = value pointed to
                        ; by current AR, or "*AR(ARP)"
LTS      *-,AR3           ; ACC = ACC-P, T = *AR(ARP), decrement
                        ; AR(ARP), ARP=3

```

One point to keep in mind is that the load of the “Nth” term is performed in parallel with the accumulation of the previous or “(N-1)th” term. The use of the LTA instruction in the code example in the figure above would yield the code in the following figure.

Enhanced Sum-of-Products

$$Y = A \cdot X1 + B \cdot X2 + C \cdot X3 + D \cdot X4$$

| | | | | |
|------|---|----|------|----|
| | | | LT | X1 |
| | | | MPY | A |
| PAC | } | X2 | LTP | X2 |
| LT | | | MPY | B |
| | | | LTA | X3 |
| | | | MPY | C |
| APAC | } | X4 | LTA | X4 |
| LT | | | MPY | D |
| | | | APAC | |
| | | | SACL | Y |

Using LTA instead of LT and APAC presents no drawbacks (other than potential confusion or error on the part of the programmer); therefore, in most cases, LTA is preferred due to its shorter execution time and reduced code space requirements.

| | | |
|------|----|-----------------------------------|
| | | ;Y = A*X1 + B*X2 + C*X3 + D*X4 |
| LT | X1 | ;T = X1 |
| MPY | A | ;P = A*X1 |
| LTP | X2 | ;ACC = A*X1, T = X2 |
| MPY | B | ;P = B*X2 |
| LTA | X3 | ;ACC = A*X1 + B*X2, T = X3 |
| MPY | C | ;P = C*X3 |
| LTA | X4 | ;ACC = A*X1 + B*X2 + C*X3, T = X4 |
| MPY | D | ;P = D*X4 |
| APAC | | ;ACC = Y |

Lab 4: Basic Programming

➤ Objective

The objective of this lab is to practice and verify the mechanics of performing arithmetic on the TMS320. In this process we will expand upon the .ASM file from the previous lab to include new functions.

The objective will be to add the code necessary to obtain the sum of the products of the n-th values from each array or, expressed mathematically:

$$y = \sum_{n=1}^4 (\text{data}_n * \text{coeff}_n)$$

As in previous labs, 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. Solutions to the procedures are available, if necessary, at the end of the lab.

➤ Procedure

Copy Files, Create Make File

1. Using Code Composer, open LAB3.CMD in C:\DSP24\LABS and save it as C:\DSP24\LABS\LAB4.CMD.
2. Open LAB3.ASM and save it as LAB4.ASM.
3. Create a new project called LAB4.MAK and add LAB4.ASM, VECTORS.ASM and LAB4.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 → Options on the menu bar. Select the Assembler tab. In the middle of the screen check "Enable Source Level Debugging". Next, select the Linker tab. In the middle of the screen select "No Autoinitialization". Create a map file named LAB4.MAP. Select OK to save the Build Options.

Multiply Data Array by Coeff Array and Store Result to Memory

4. Edit LAB4.ASM and modify it to multiply the data array by the coeff array, storing the result to memory.
 - Use a BANTZ loop on the initialization process.
 - Initialize pointers to the beginning of the data and coeff arrays.
 - Multiply the first pair of values and transfer the results to the accumulator.
 - Repeat the process for the remaining pairs. (**Do not use BANTZ on MPY routine.**)
 - Store the result to memory.

Save your work.

Build and Load

5. Click the "Rebuild All" 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, load the output file onto the target. Click: File → Load Program...

Then reset the DSP by clicking on: Debug → Reset DSP

Right click on the `VECTORS.ASM` window and select `Mixed Mode` to debug using both source and assembly.

7. Single-step your routine. While single-stepping, open two memory windows to see the values located in `table[9]` and `data[9]` by using the "View Memory" button on the vertical toolbar and use the address labels `table` and `data`. Setting the properties filed to "Hex – TI style" will give you more viewable data in the window. Additionally, open the CPU register (and Status) by using the "View 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

Review

Review

1. What conditions can you test?
2. How many conditions can specify in a single instruction?
3. What determines a TRUE test of a multiconditional operation?
4. How do you multiply two numbers?
5. How do you repeat a block of code?
6. What is LTS?

Solutions

```
;SOLUTION FILE FOR LAB4.ASM
        .def      start
        .bss      data,4           ;SET UP ARRAYS IN RAM
        .bss      coeff,4
        .bss      result,1

        .data
table:   .int      1,2,3,4         ;SET UP DATA TABLES
        .int      8,6,4,2
        .int      0
length  .set      $-table

        .text
start:   LAR       AR1,#table       ;INIT POINTERS
        LAR       AR2,#data
        LAR       AR3,#length-1   ;SET UP FOR BANZ LOOP
        MAR       *,AR1

loop:    LACC      **+,0,AR2        ;IN-LINE LOAD/STORE TO INIT RAM
        SACL      **+,0,AR3        ;LOAD FROM .data, STORE TO .bss
        BANZ      loop,*-,AR1

sop:     LAR       AR1,#data        ;BEGIN SUM OF PRODUCTS ROUTINE
        LAR       AR2,#coeff
        MAR       *,AR1
        LT        **+,AR2
        MPY       **+,AR1
        LTP       **+,AR2
        MPY       **+,AR1
        LTA       **+,AR2
        MPY       **+,AR1
        APAC
        LDP        #result
        SACL      result
        B         start
```

Advanced Programming Techniques

Introduction

In this module several methods for improving the speed of execution of various operations will be considered. Additionally, several of these methods also offer a reduction in system resources, thus reducing their cost of implementation. These enhancements include the repeat instruction, MAC instruction, and block move operations.

Learning Objectives

Learning Objectives

- ◆ 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

Module Topics

| | |
|--|-------------|
| Advanced Programming Techniques | 5-1 |
| <i>Module Topics.....</i> | <i>5-2</i> |
| <i>Repeat Operations</i> | <i>5-3</i> |
| <i>Advanced Use of the Multiplier</i> | <i>5-5</i> |
| <i>Block Move Operations.....</i> | <i>5-8</i> |
| <i>Lab 5: Advanced Programming Lab.....</i> | <i>5-11</i> |
| <i>Review.....</i> | <i>5-13</i> |
| Solutions..... | 5-14 |

Repeat Operations

In addition to branch and call instructions, the C240x can perform “hardware do-loops”; i.e., the ability to iteratively execute code without the use of branches within the code loop. In the C240x, repeat loops consist of a single line of code.

RPT — Repeat Next Instruction

The RPT instruction is used to direct the C240x to repeat the instruction which follows a specified number of times. RPT has the following characteristics:

- 1-cycle instruction
- Loads RPTC with 8- or 16-bit value
- Following instruction is iterated RPTC+1 times
- Creates “0 overhead loop” on single instruction
- Repeat value is usually specified by a short (8-bit) immediate operand
- Repeat value may also be specified via direct or indirect addressing
- Repeat loops are non-interruptible

The repeat instruction is executed only once — at the beginning of the process — whereas the branch operation has to be performed each iteration. The RPT syntax is:

```
RPT    #<n>                ;execute next instruction (n+1) times,
                           ;only 8 LSB's are used (immediate addressing)
```

A less commonly used version of the repeat instruction which allows the repeat count value to be taken from data memory via direct or indirect addressing is:

```
RPT    <dma>               ;execute next instruction @(dma)+1 times,
                           ;16-bit value used (direct and indirect addressing)
```

The repeat instruction is almost as fast as in-line code, yet uses far less program space for long procedures. For example, the following code would sum all the values from 200 sequential data locations:

```
RPT    #199                ;execute next instruction 200 times
ADD    *+                  ;add from * and increment current AR
```

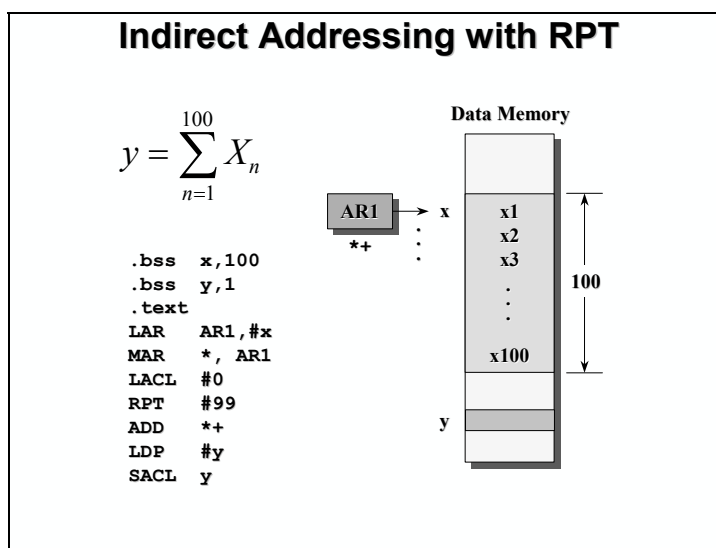
In-line code for the preceding example would take 200 cycles and require 200 lines of code. Using the repeat operation increases the execution time to 201 cycles but uses only two lines of code. The repeat instruction can cause some procedures to run **faster** than in-line code, because, the instruction does not need to be fetched more than once. These instructions include:

| | | | | | |
|------|------|------|-----|------|------|
| TBLR | BLDD | MAC | OUT | SQRA | SUBC |
| TBLW | BLPD | MACD | IN | SQRS | |

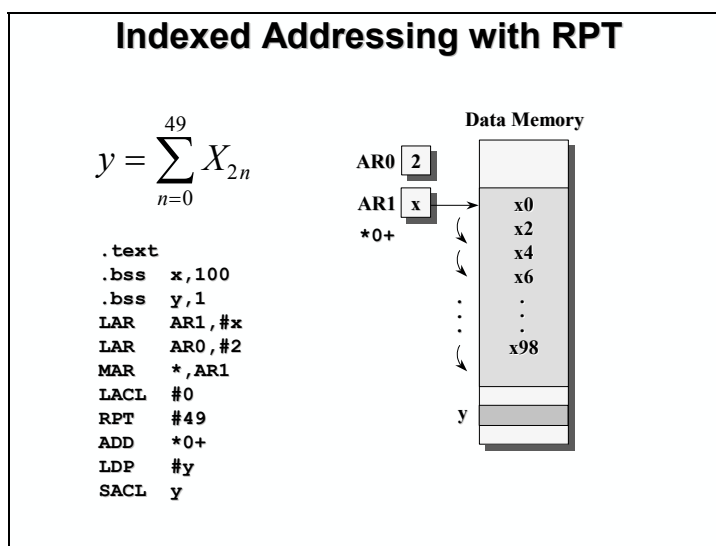
The following instructions *cannot* be used in a repeat operation. If repeated, they will cause improper operation:

- All branch and call operations
- All immediate instructions
- All repeat instructions
- IDLE, TRAP, RET, LACL, SPM

An example of the Repeat Next Instruction (RPT) with indirect addressing is shown below, where a pointer is used to perform a summation of the values in an array.



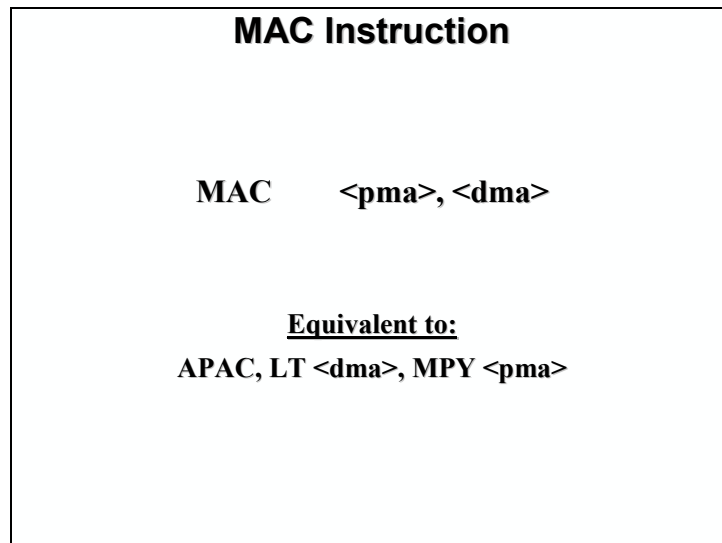
Sometimes it is desirable to be able to increment/decrement by values other than one. In these instances, an index register is required to specify the step size to be used. On the C240x, AR0 can be used as either a pointer or an index value on any other pointer. Use of the index during auto-increment/auto-decrement is specified by adding a 0 (shorthand for “AR0”) before the modifier, as in *0+ or *0-.



Advanced Use of the Multiplier

Since the sum-of-products algorithm is so often encountered in numerical processing, it is desirable to be able to implement a multiply and accumulate function in as little time as possible. In the previous sections, it was seen that this multiply and accumulate function could be performed on the C240x in two or three cycles. Even better performance could be obtained in a system if the multiply and accumulate function could be performed in a single cycle.

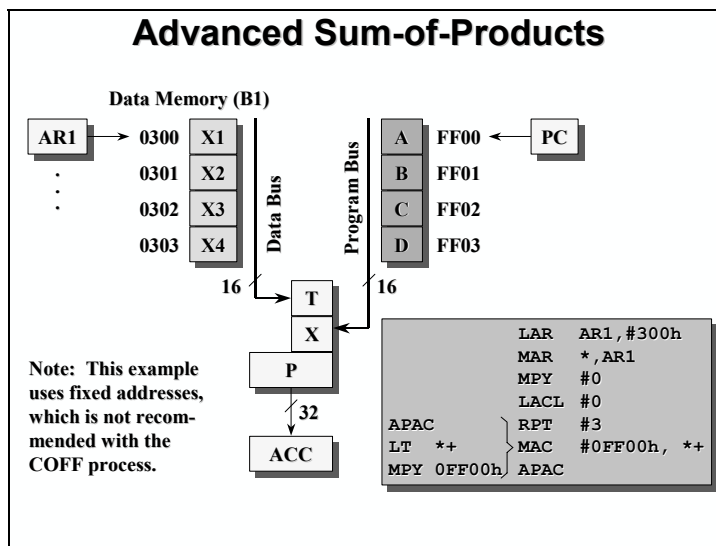
Consider the requirements of a multiply and accumulate function: in a single cycle, two operands must be read and routed to the multiplier, a product formed, and a product accumulated. The LTA instruction performs many of these functions. It accumulates a product and reads an operand from data memory. As such, all that remains is the ability to read a second operand and route it to the multiplier. Unfortunately, the reading of the first operand blocks the use of the data bus. However, because the C240x is a Harvard architecture machine, it has a second bus. This bus (the program bus) could be used to route an operand, but then the operand must reside in program memory for it to be accessed by the program bus. Additionally, the only way to select a location from program memory is via the Program Counter (PC). As such, the normal operation of the program space—fetching instructions—must be temporarily suspended. This operation is performed by the MAC instruction, and requires three cycles to execute due to the offloading of the program counter:



MAC can approach an execution speed of one cycle per tap when operated on by the repeat (RPT) instruction. In the repeat loop, the sharing of the PC between operand and instruction fetching is avoided. This is because the repeated instruction is fetched only once, stored in the “instruction register,” and iterated upon. Thus, the first MAC instruction requires three cycles, but all subsequent MAC instructions within the RPT loop execute in a single cycle. This can be represented schematically and implemented in C240x code as shown in the figure on the next page.

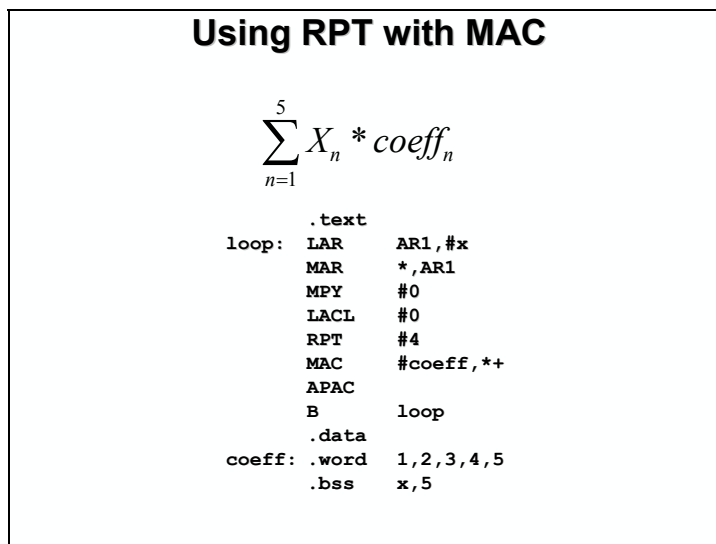
Given that the MAC instruction can be executed in a single cycle, it is sometimes assumed that using MAC is always faster than LTA and MPY. However, when only a few taps are calculated, the overhead (and complexity) in setting up the MAC operation make LTA/MPY the preferred choice. A general guideline is to use MAC when more than 10 taps are to be performed, and to

use LTA/MPY when fewer calculations are required. Additionally, the concept of using a decrement and branch with LTA/MPY is sometimes considered. Since branches require multiple cycles to implement and do not contribute to the numerical solution, they are generally to be avoided in systems where code execution speed is a primary consideration.



In the example in the following figure, RPT is used with the MAC instruction. Note that:

- Five iterations are taken when the RPT argument is set to four.
- MAC is three cycles for the first iteration and single-cycle thereafter.



Other Multiplier Operations

Several other instructions invoke the use of the multiplier. One group provides the ability to perform a “square and accumulate” function. In this function, a single operand is specified and passed to both multiplier inputs. The resulting square of the value is placed in the product register and the prior P value is summed to the Accumulator. This function, which is useful in power functions and other operations, often appears within repeat loops. Another group, the “multiply and add” functions, are the combination of APAC and MPY instructions. These have been useful in the LMS adaptive filters to perform an addition for the LMS update while simultaneously performing a multiply on the filter tap. Here are some examples of these instructions.

```
MPYA  x      ; ACC = ACC+P , P = x*T
MPYS  *      ; ACC = ACC-P , P = T*( *AR(ARP) )
SQRA  y      ; ACC = ACC+P , P = y*y
SQRS  *+     ; ACC = ACC-P , P = square *AR(ARP) , AR(ARP)= AR(ARP)+1
MPYU  n      ; unsigned mpy: P = ABS(T*n)
```

Block Move Operations

Since the C240x operates in two memory spaces, it is sometimes necessary to move an array from one memory space to the other. The most common such example is the initialization of data RAM from program ROM. The C240x can copy to data RAM from ROM located in either data or program space.

Although the initialization values are functionally associated with data, the fact that they need to be located in a ROM-like memory would make it convenient if they could reside in the same ROM devices which might be used to hold the program code. Two “block” instructions are available to facilitate copying arrays.

- BLPD: Program memory to data memory.
- BLDD: Data memory to data memory.

The block move commands are somewhat misnamed, since they perform copy (not move) operations, and they perform a **single** (not a block) memory operation. When coupled with the repeat operation, though, the block instructions can effectively copy an entire array from one place to another.

Block instructions execute in two or three cycles. However, in a repeat loop, only the **first** instruction requires multiple cycles; all subsequent iterations execute in a single cycle.

A block move process requires the specification of three parameters:

- Location of the array to copy from
- Location of the array to copy to
- Number of sequential words in the array

The block instruction specifies the source and destination addresses in a variety of ways. The block size may be specified as the argument of a RPT instruction. Details on each of the block instructions and their usage follow.

```
BLPD    source, destination    ;block move from program memory to data memory
BLDD    source, destination    ;block move from data memory to data memory
```

BLPD — Copying from Program Memory to Data Memory

BLPD is somewhat like the MAC instruction in that two operands are specified:

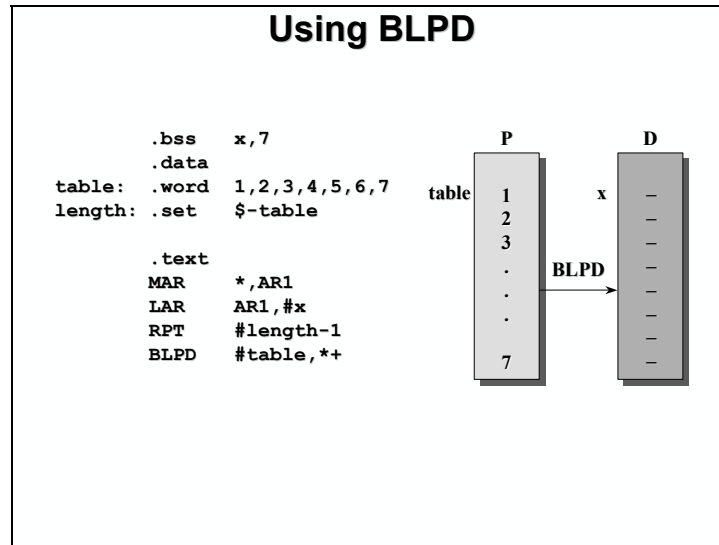
1. An address to an array in program space
2. An address to an array in data space

The <pma> uses an explicit 16-bit address. The <dma> may be specified via direct or indirect addressing, although indirect addressing is almost always used since it can specify copying to incrementing addresses.

As with MAC, since the PC is the only pointer into program space when used in a repeat loop, the <pma> autoincrements after each iteration; the original contents of the PC are restored after the block operation. The hardware stack is not required for MAC or block operations. As noted

before, in a repeat loop, only the first block execution will require multiple cycles; the rest will execute in one cycle for each iteration.

In the example in the following figure, BLPD is used to initialize a block of RAM from a ROM table. Notice the need for the pound sign (#) before the immediate operand `table` in the BLPD instruction.



BLDD — Copying from Data Memory to Data Memory

Sometimes it is beneficial to be able to copy an array from one part of data memory to another. One example would be to move information between on-chip and off-chip memories. The BLDD instruction performs this function in much the same way as BLPD, with the following differences:

- Each operand specifies a <dma>.
- The first operand is the source address and the second is the destination.
- One operand is specified via an immediate address.
- The other operand is specified via direct or indirect addressing.

As with BLPD, execution time is dependent on the locations of the source and destination arrays, and is single cycle during RPT loop iterations.

The following two instructions are rarely used, but maintain code compatibility to earlier TMS320 generations.

TBLW — Copying from Data Memory to Program Memory

The TBLW instruction transfers a word in data memory to program memory. The data-memory address is specified by the instruction, and the program-memory address is specified by the lower 16 bits of the accumulator. A read from data memory is followed by a write to program memory to complete the instruction. When the repeat mode is used, TBLW effectively becomes a single-

cycle instruction, and the program counter that contains the ACCL is incremented once each cycle.

TBLR — Copying from Program Memory to Data Memory

The TBLR instruction transfers a word from a location in program memory to a data-memory location specified by the instruction. The program-memory address is defined by the low-order 16 bits of the accumulator. For this operation, a read from program memory is performed, followed by a write to data memory to complete the instruction. When the repeat mode is used, TBLR effectively becomes a single-cycle instruction, and the program counter that contains the ACCL is incremented once each cycle.

I/O Operations

The C240x has two instructions, IN and OUT, which allow transfers between data memory and I/O devices. These instructions are optimal for use with parallel A/D and D/A converters. The instruction syntax is:

```
IN <dma>, <pa>           ;read the data from peripheral <pa> and store it to
                           ;data memory location <dma>
OUT <dma>, <pa>           ;write the data found at data memory location <dma> to
                           ;peripheral at port address <pa>
```

where: <dma> is a direct or indirect data memory address
 <pa> is a port address value between 0 and 0FFFFh

Note also, that IN and OUT operate only between I/O memory and data memory and the \overline{IS} signal (active low) is active on I/O operations (IN or OUT).

| Data Movement Instruction Summary | | |
|-----------------------------------|---------------------|---|
| <u>Instruction</u> | | <u>Move Data From...</u> |
| BLPD | #<pma>, <dma> | Program to Data |
| BLDD | source, destination | Data to Data |
| TBLW | <dma> | Data to Program (pma specified by ACCL) |
| TBLR | <dma> | Program to Data (pma specified by ACCL) |
| IN | <dma>, PA | I/O to Data (\overline{IS} signal goes low) |
| OUT | <dma>, PA | Data to I/O (\overline{IS} signal goes low) |

Lab 5: Advanced Programming Lab

➤ Objective

The objective of this lab is to practice and verify enhanced programming procedures. In this process we will modify the .ASM file from the previous lab in phases, each designed to yield the same result, but with greater efficiency.

It is recommended that each phase be completed before moving on to the next in order to best observe the effects of each individual modification and (more importantly) to simplify the debugging process by solving one problem at a time. To this end, you will copy LAB4 files to LAB5 files, and make the following modifications:

1. Replace the initialization routine using LACC/SACL/BANZ with a RPT/BLPD process. Consider where the ROM table must now reside for correct operation. Edit LAB5 .CMD and make any necessary changes.
2. After verifying the correct operation of the block initialization, replace the sum of products using LTA and MPY with a MAC-based implementation. Since the MAC operation requires one array to be in program memory, you may wish to modify your initialization routine to skip the transfer of one of the arrays, thus reducing the amount of data RAM and cycles required for initialization.

➤ Procedure

Copy Files, Create Make File

1. Using Code Composer, open LAB4 .CMD in C:\DSP24\LABS and save it as C:\DSP24\LABS\LAB5 .CMD.
2. Open LAB4 .ASM and save it as LAB5 .ASM.
3. Create a new project called LAB5 .MAK and add LAB5 .ASM, VECTORS .ASM and LAB5 .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 → Options on the menu bar. Select the Assembler tab. In the middle of the screen check "Enable Source Level Debugging". Next, select the Linker tab. In the middle of the screen select "No Autoinitialization". Create a map file named LAB5 .MAP. Select OK to save the Build Options.

Initialization Routine using RPT/BLPD

4. Edit LAB5 .ASM and modify it by replacing the initialization routine using LACC/SACL/BANZ with a RPT/BLPD process.
5. Edit LAB5 .CMD and consider where the ROM table must now reside for correct operation.
6. Save your work. If you would like, you can use Code Composer to verify the correct operation of the block initialization before moving to the next step.

Sum of Products using a MAC-based Implementation

7. Edit LAB5.ASM and modify it by replacing the sum of products using LTA and MPY with a MAC-based implementation. Since the MAC operation requires one array to be in program memory, you may wish to modify your initialization routine to skip the transfer of one of the arrays, thus reducing the amount of data RAM and cycles required for initialization.

Build and Load

8. Click the "Rebuild All" 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.
9. If the "Load program after build" option was not selected in Code Composer, load the output file into the target. Click: File → Load Program...

Then reset the DSP by clicking on: Debug → Reset DSP

Right click on the VECTORS.ASM window and select Mixed Mode to debug using both source and assembly.

10. Single-step your routine. While single-stepping, open memory windows to see the values located in *table [9]* and *data [9]*. Open the CPU and Status registers. Check to see if the program is working as expected. Debug and modify, if needed.

End of Exercise

Optional Exercise

After completing the above process, consider the following scenario. If program ROM is in a slow external memory, the performance of the MAC will be reduced since it will be accessing values in a slow memory. The MAC function may still be used in such systems without any performance loss if several additional steps are taken first:

- a. Allocate RAM for the coefficient array using a .usect.
- b. Configure Block 0 over to the data space.
- c. Copy the coefficient array to the new section using another RPT/BLPD.
- d. Configure Block 0 over to program space.
- e. Perform the MAC using the coefficients now located in internal program RAM.
- f. Link the section to the relocatable internal RAM (e.g., Block 0) by modifying the linker command file.

Note: Solutions files for this optional exercise have been provided in the solutions directory. (File names are LAB5C.ASM and LAB5C.CMD).

Review

Loop Ctrl Options (100-term MAC)

| BANZ | | RPT | |
|-----------------|-----------------------|------------|------------|
| LAR | AR1, #x | LAR | AR1, #x |
| LAR | AR2, #a | MAR | *, AR1 |
| LAR | AR0, #99 | MPY | #0 |
| MAR | *, AR1 | LACL | #0 |
| LACL | #0 | RPT | #99 |
| MPY | #0 | MAC | #a, ** |
| loop: APAC | | APAC | |
| LT | **, AR2 | SACL | y |
| MPY | **, AR0 | | |
| BANZ | loop, *- , AR1 | | |
| APAC | | | |
| SACL | y | | |
| cycles | | 9 + N | |
| program control | | 2 | |
| | | 10 + 7N | |
| | | 4N + 1 | |

Review

1. What does the following instruction do?
RPT #16
2. Can we repeat an instruction more than 255 times?
3. Is using the MAC instruction always the most efficient way to multiply numbers?
4. How does the MAC instruction work?
5. What instruction would be useful to initialize the coefficients in an adaptive filter after power-up?

Solutions

```

;SOLUTION FILE FOR LAB5.ASM

.def      start
.bss      data,4           ;SET UP ARRAYS IN RAM
;         ;line not used
.bss      result,1
.data
tableA: .int 1,2,3,4       ;SET UP DATA TABLES
tableB: .int 8,6,4,2
;         ;line not used
length .set ($-tableA)-($-tableB)
.text
start:    LAR  AR0,#data
          MAR *,AR0
          RPT #length-1
          BLPD #tableA,++

          LAR  AR0,#data
          MPY  #0
          LACL #0

          RPT #length-1
          MAC  #tableB,++
          AFAC
          LDP  #result
          SACL result
end        B      end

```

```

/* SOLUTION FILE FOR LAB5.CMD */

MEMORY
{
  PAGE 0:
    VECS:   org = 0000h, len = 0040h
    EPROM:  org = 0044h, len = 7FBCh

    PAGE 1:
    B2:     org = 0060h, len = 0020h
    I_RAM:  org = 0200h, len = 0200h
    SARAM:  org = 0800h, len = 0800h
}

SECTIONS
{
  vectors: > VECS    PAGE 0
  .text:   > EPROM    PAGE 0
  .data:   > EPROM    PAGE 0
  .bss:    > B2       PAGE 1
}

```

Introduction

This module discusses several numerical issues and solutions proposed for handling them. One concerns multiplication, and how to store the results when the process of multiplication creates results larger than the inputs. Another concerns accumulation, especially when long summations are performed. Also, numerical modes and division will be discussed.

Learning Objectives

Learning Objectives

- ◆ Use coding techniques to handle numerical issues associated with fixed-point systems
- ◆ Compare/contrast integer and fractional operations
- ◆ Implement the appropriate overflow mode for a system when adding binary fractions
- ◆ Write code to perform a range of numerical processes with reliable results

Module Topics

| | |
|--|-------------|
| Numerical Issues..... | 6-1 |
| <i>Module Topics.....</i> | <i>6-2</i> |
| <i>Numbering System Basics</i> | <i>6-3</i> |
| Binary Numbers..... | 6-3 |
| Two's Complement Numbers | 6-3 |
| Sign Extension Mode..... | 6-5 |
| <i>Binary Multiplication.....</i> | <i>6-6</i> |
| <i>Binary Fractions</i> | <i>6-8</i> |
| Representing Fractions in Binary | 6-8 |
| Multiplying Binary Fractions | 6-9 |
| <i>Additive Overflow Handling</i> | <i>6-16</i> |
| 1. Establish Headroom via PM Shifter (SPM 3) | 6-16 |
| 2. Overflow Mode..... | 6-16 |
| 3. Test for Overflow..... | 6-17 |
| 4. Overflow Allowed by Design | 6-18 |
| <i>What to do with those “Guard-Bits”</i> | <i>6-19</i> |
| <i>32-Bit Arithmetic.....</i> | <i>6-21</i> |
| <i>Division on the C240x.....</i> | <i>6-22</i> |
| SUBC Instruction | 6-22 |
| Integer Division..... | 6-23 |
| Fractional Division..... | 6-24 |
| <i>Division Summary.....</i> | <i>6-25</i> |
| <i>Lab 6: Exercise</i> | <i>6-26</i> |
| <i>Review.....</i> | <i>6-27</i> |
| Solutions..... | 6-27 |

Numbering System Basics

Given the ability to perform arithmetic processes (addition and multiplication) with the C240x, it is important to understand the underlying mathematical issues which come into play. Therefore, we shall examine the numerical concepts which apply to the C240x 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:

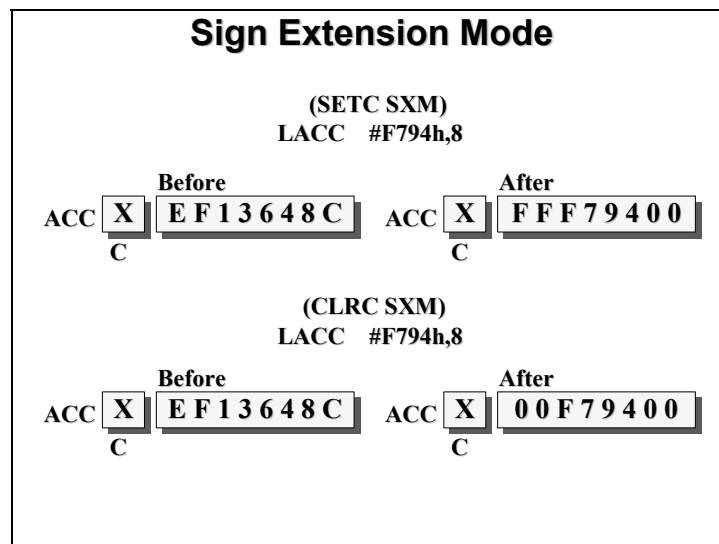
| | | | | |
|----------------|----------------------|-------------------|------------------------|----------------------------|
| Original No. | 0 1 1 0 ₂ | = 6 ₁₀ | 1 1 1 1 0 ₂ | = -2 ₁₀ |
| 1. Load low | 0 1 1 0 | | 1 1 1 1 0 | |
| 2. Sign Extend | 0 0 0 0 0 1 1 0 | = 4 + 2 = 6 | 1 1 1 1 1 1 1 0 | = -128 + 64 + ... + 2 = -2 |

Sign Extension Mode

The C240x can operate on either unsigned binary or two's complement operands. The “Sign Extension Mode” (SXM) bit, present within a status register of the C240x, identifies whether or not the sign extension process is used when a value is brought into the accumulator. The set and clear condition instructions may be used to operate on the SXM bit:

```
SETC    SXM           ; set SXM bit - accumulator operates in two's
                      ; complement mode (invokes sign extension)
CLRC    SXM           ; clear SXM bit - accumulator operates in
                      ; unsigned binary mode
```

On reset, SXM is set. However, it is good programming practice to always select the desired SXM at the beginning of a module to assure the proper mode. As demonstrated, the incorrect setting can yield vastly different results than expected! Only when a system is fully operational should the programmer inspect for, and remove, redundant status selection operations.

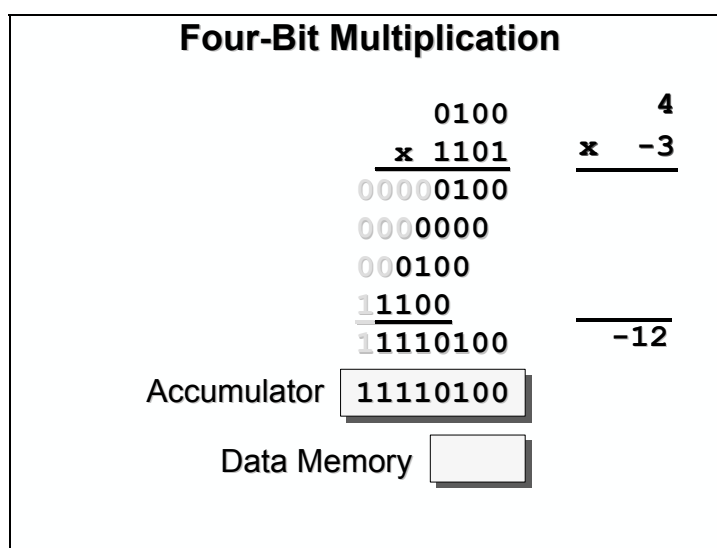


Binary Multiplication

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 C240x uses in multiplying numbers — it is merely a way of observing how binary numbers work in arithmetic processes.

The C240x 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:



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 using the SACL instruction. What problem is apparent using this method in this example?
- Store the upper accumulator back to memory using the SACH instruction. 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 using SACH and SACL. 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

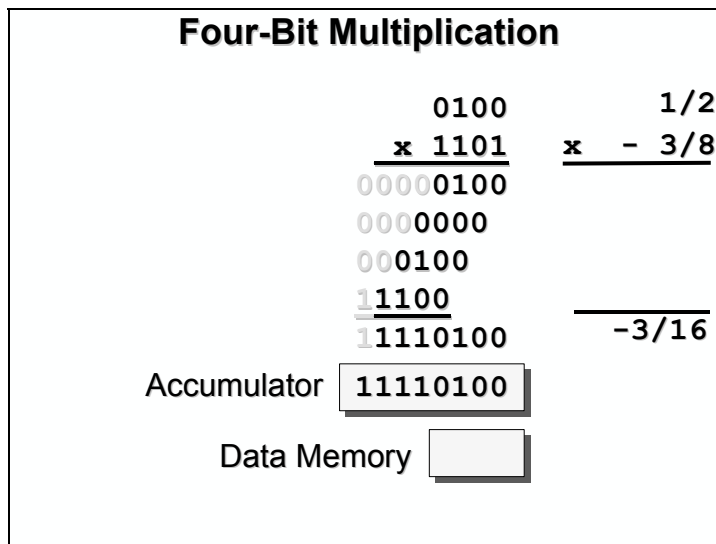
In order to represent both positive and negative values, the two's complement process will again be used. However, in the case of fractions, we will not set the LSB to 1 (as was the case for integers). When one considers that the range of fractions is from -1 to +1, and that the only bit which conveys 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.

$$\begin{array}{|c|} \hline 1 \\ \hline \end{array} . \begin{array}{|c|c|c|} \hline 0 & 1 & 1 \\ \hline \end{array} = -1 + 1/4 + 1/8 = -5/8$$

$\begin{array}{cccc} -1 & & 1/2 & 1/4 & 1/8 \end{array}$

Multiplying Binary Fractions

When the C240x 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:



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 result 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. (SACL, SACH).
- 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
 - Shift on store
 - Use Product Mode shifter

Explicit Shift

As seen in the previous section, the multiplication of two fractional (Q15) operands in the '320 causes a redundant sign bit in the upper accumulator. In order to maintain the greatest possible precision and have results in the same format as source operands, it is necessary to store back to memory the upper accumulator "shifted by one." This could be accomplished in two cycles with the following instructions:

```
; A * B = C

LT   A           ; A and B are Q15 fractions
MPY  B           ; P = A*B   : Q30 notation
PAC                   ; ACC = A*B   in Q30 notation
SFL                   ; ACC = A*B   in Q31 notation
SACH C           ; C = A*B   in Q15 notation (bottom 16 "Q's" lost)
```

Shift on Store

The same result could be passed to C one cycle faster if the output shifters of the accumulator are used with the shift option of the SACH instruction:

```
; A * B = C

LT   A           ; A and B are Q15 fractions
MPY  B           ; P = A*B   : Q30 notation
PAC                   ; Acc = A*B   in Q30 notation
SACH C,1         ; C = A*B   in Q15 notation (bottom 16 "Q's" lost)
```

Yet another method exists for correcting the redundant sign bit of a Q30 number; i.e., using the Product Mode Shifter.

Product Mode Shifter

The Product Mode Shifter, designated as the P-scaler, is located between the P register and the ALU input multiplexer. This shifter can pass the product with no shift, or can perform a left shift of one or four bits, or a right shift of six bits. The shifter is controlled by the contents of the PM field of Status Register 1 (ST1), which can be modified by the SPM (set product shift mode) instruction. The values of PM and the resulting shifts are shown in the following table.

Product Shift Modes

| PM | SHIFT |
|----|---------------|
| 0 | No shift |
| 1 | Left Shift 1 |
| 2 | Left Shift 4 |
| 3 | Right Shift 6 |

By initializing this shifter to provide a shift of one, all accumulations from the product register will be in Q31 notation. The code for this operation follows:

```
SPM 1          ; product mode shift = 1

; A * B = C

LT   A          ; A and B are Q15 fractions
MPY  B          ; P = A*B : Q30 notation
PAC          ; Acc = A*B in Q31 notation
SACH C          ; C = A*B in Q15 notation (bottom 16 "Q's" lost)
```

The PM shifter is set to 0 on reset. It is recommended that the PM shifter be used to correct the Q notation. In this way, the upper accumulator always contains a proper Q15 number, which is easier to evaluate during debugging, and makes equations that sum non-multiplied values with multiplied values somewhat easier to develop.

As an example, the next two segments of code perform the equation of a straight line: $Y = m * X + b$. The first is in Q30 notation and the second is in Q31. Notice the way the value b is summed to the accumulator.

```
; Q30 example
SPM 0          ; Q30 mode
LT   X          ;
MPY  m          ; P = m*X : Q30
PAC          ; Acc = m*X : Q30
ADD  b,15       ; Acc = m*X + b : Q30
SACH Y,1        ; Y = m*X + b : Q15

; Q31 example
SPM 1          ; Q31 mode
LT   X          ;
MPY  m          ; P = m*X : Q30
PAC          ; Acc = m*X : Q31
ADD  b,16       ; Acc = m*X + b : Q31
SACH Y          ; Y = m*X + b : Q15
```

Only two other shift options exist with the Product Mode Shifter: a left shift of four bits and a right shift of six bits.

The four-bit left shift (SPM 2) provides a Q31 accumulator when the MPY immediate (13-bit constant) is used (a Q12).

The final PM shifter value (SPM 3) produces a sign-extended right shift of 6 bits. This is one means of obtaining headroom for summations which may (easily) exceed the range of fractional numbers. In the next section, the concept of additive overflow — including methods for handling it — are considered.

The following figure is a summary of the three methods for correcting redundant sign bit:

| Correcting Redundant Sign Bit | | | |
|--------------------------------------|------|-----|----------------------------|
| ; A * B = C | | | |
| ◆ Explicit Shift | LT | A | ;A and B are Q15 fractions |
| | MPY | B | ;P=A*B : Q30 notation |
| | PAC | | ;ACC=A*B in Q30 notation |
| | SFL | | ;ACC=A*B in Q31 notation |
| | SACH | C | ;C=A*B in Q15 notation |
| ◆ Shift on Store | LT | A | ;A and B are Q15 fractions |
| | MPY | B | ;P=A*B : Q30 notation |
| | PAC | | ;ACC=A*B in Q30 notation |
| | SACH | C,1 | ;C=A*B in Q15 notation |
| | | | |
| ◆ Product Mode Shifter | SPM | 1 | ;product mode shifter = 1 |
| | LT | A | ;A and B are Q15 fractions |
| | MPY | B | ;P=A*B : Q30 notation |
| | PAC | | ;ACC=A*B in Q31 notation |
| | SACH | C | ;C=A*B in Q15 notation |

Fractional vs. Integer Representation

As a final comparison, consider the following about integers vs. fractions:

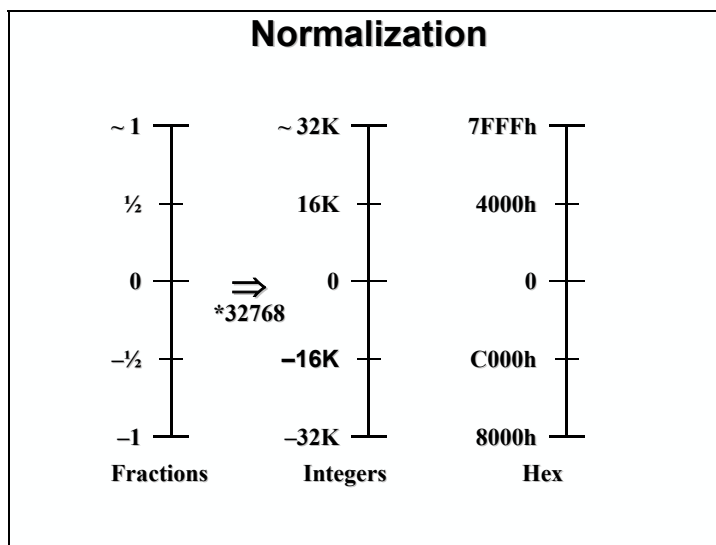
| Fractional vs. Integer | |
|--|--|
| ◆ Range | |
| ◆ Integers have a maximum range determined by the number of bits | |
| ◆ Fractions have a maximum range of ± 1 | |
| ◆ Precision | |
| ◆ Integers have a maximum precision of 1 | |
| ◆ Fractional precision is determined by the number of bits | |

Thus, the C240x 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 \pm 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.

Binary Fractions and the Assembler

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 C240x, 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:



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 C240x, the normalized fraction looks and behaves exactly as a fraction. Thus, when using fractional constants in a C240x 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: $32768 \times 62 / 100$
- To represent 0.1405: $32768 \times 1405 / 10000$

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

Exercise

Exercise - Assembling Fractions

Write the assembly language statement to code the following numbers:

1. Code: 0.14

.int_____

2. Code: 0.123

.int_____

3. Code: -1.0

.int_____

Additive Overflow Handling

In the preceding sections, it was shown that fractions were superior to integers in that the fractions yielded bounded results when multiplied, whereas integers were unbounded — with results growing larger and harder to represent and use.

A second consideration is for addition. While fractions experience “closure” when multiplied, they offer no such guarantees when added. Since many DSP algorithms employ a sum-of-products structure, it is important for the user to be able to perform long summations with reliable results. The C240x offers four mechanisms which aid in resolving this requirement.

1. Establish Headroom via PM Shifter (SPM 3)

As noted in the previous section, the Product Mode (PM) shifter is capable of performing a 6-bit right shift. When one includes the already-present sign and “sign-extension” bits in the product register, a total of eight integer bits will be present in the accumulator when using the SPM 3 option. Thus, a value as large as ± 128 may be represented, which is well beyond the range of a “fractional” (bounded by ± 1) number system.

These ‘extra’ bits are called *headroom* or *guard-bits* since they provide a higher ceiling and guard against overflow. When using this method it would take 128 summations with the maximum input value to overflow the system.

Bottom Line

This system works well in preserving data that would otherwise cause overflow, but it doesn’t keep the data in Q15 format. This makes it inconvenient to use for intermediate operations.

2. Overflow Mode

As noted, it is possible to establish eight integer bits in the accumulator to permit representation of large numbers. However, it is still possible for even greater summations to be generated, which would overflow the accumulator and yield erroneous results. Programmers often (rightfully) consider the “worst case” scenario of adding a 1 to the largest possible positive number. For our 4-bit two’s complement examples, consider the result of:

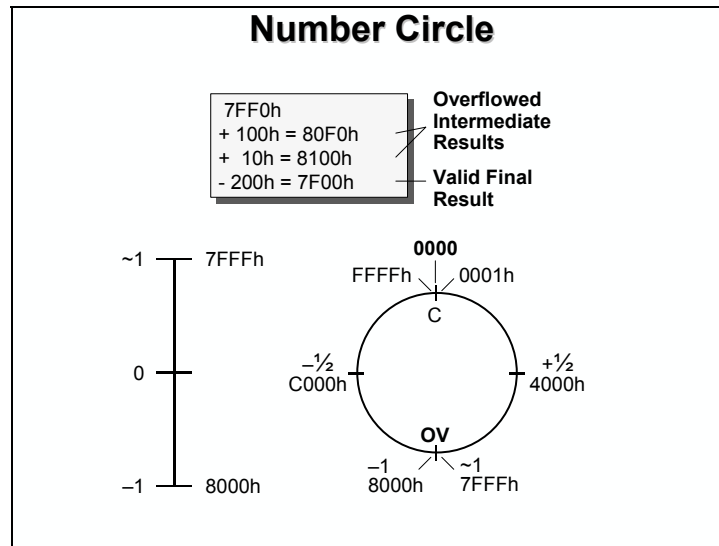
$$0111_2 + 0001_2 = 1000_2 \text{ which produces } 7_{10} + 1_{10} = -8_{10}$$

Obviously, this is not the desired result, but it’s a simple reality for two’s complement numbers. In the case of the C240x, the number of bits is larger (16 bits), but the effect is still the same.

What becomes apparent is that our “number line” used to diagram the range of representable values is actually a number “circle”. The most positive and negative values are adjacent, not opposed, to one another. The C240x can be programmed to handle this process in either of two ways:

- Allow the overflows to occur by issuing the CLRC OVM instruction.
- Impose “saturation” or “clipping” which limits accumulator activity to never crossing the “boundary” between the most positive and most negative value by using the SETC OVM instruction.

Consider these options in performing the calculations noted in the diagram below of a 16-bit integer “number circle” which could be used to represent the C240x's number range:



- What effect does the status of the OVM (overflow mode) bit have on the results obtained in performing the calculations?
- Which mode is better? Under what conditions?

As with the Processor Mode configuration bits, the Overflow Mode is not specified by reset. If desired, it is essential to specify the OVM before performing calculations which may incur overflows on the accumulator. *It is good programming practice to specify all processor status options expected within each module.* In this way, the possibility of a program running with the wrong configurations is reduced — especially valuable during program development. When the program is functioning, you may find that many of these initialization routines are unnecessary, and, if desired, may trim them from the code to improve speed and efficiency.

Bottom Line

This method prevents overflow which can be critical in some systems – especially systems where mechanical damage may occur. Unfortunately, saturating the result at the plus/minus boundary will cause loss of data. This is O.K. for some systems, but intolerable in others.

For systems which cannot tolerate overflow, you can test for overflow – see next.

3. Test for Overflow

Overflow is a term which can be used to describe different things. In the case of the C240x, an overflow (OV) bit in the status register is set whenever the value in the accumulator crosses the boundary between largest positive and negative numbers. An overflow, therefore, indicates that a 33-bit phenomena has occurred. Other details about the overflow bit:

- Overflow is latched — once set it remains high until tested or the processor is reset.
- Overflow is best tested via a conditional branch, e.g., BCND <pma>, OV

- For valid results, it is necessary to clear OV before starting a new calculation. A simple method using the BCND might be as follows:

```
... old calculation ...  
    .  
    .  
NEWCODE    BCND NEWCODE,OV  
           ZAC  
    .  
    .  
... new calculation ...
```

Whether the conditional branch is taken or not, the line NEWCODE will occur next, and the value in the OV will have been tested and, therefore, reset to zero before the “new calculation” begins.

- The OV bit is reset to 0 during a hardware reset.

Bottom Line

This is one of the most common methods for handling overflow: test and, when necessary, re-scale the system. This was adequate when a multiply/accumulate (MAC) took many processor cycles to complete – i.e. the overhead of testing and branching was relatively small.

When using a TMS320 processor, though, the overhead is significant verses our single-cycle MAC. Therefore testing for overflow after each operation or series of operations becomes intolerable.

4. Overflow Allowed by Design

Most systems can be specified to be bounded and linear. That is, we can create a process whose result does not exceed the available number range. Given this constraint, the following processor initialization instructions would be used to achieve best results in the C240x:

```
SETC      SXM           ; allow two's complement  
CLRC      OVM           ; allow intermediate overflows  
SPM       1             ; Q15*Q15 makes a Q31 in accumulator  
           ; or use SACH result,1 to store the answer
```

Bottom Line

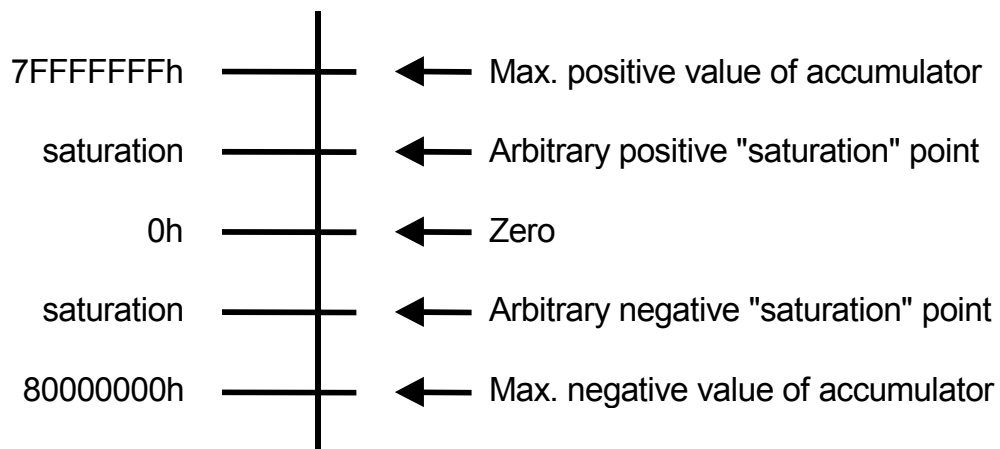
This is the preferred method since it requires no additional processor bandwidth to implement.

Most systems are linearly modeled and their transfer functions can be specified without gain. This allows this method to be used. In those rare instances where a system cannot meet these two conditions, the choice might be Method 1 as outlined next.

What to do with those “Guard-Bits”

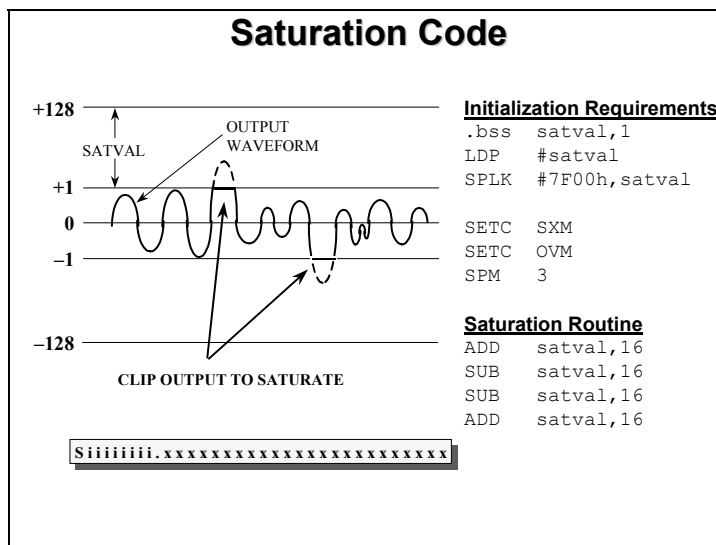
(Putting SPM3 Solution to Work)

In systems which do not meet the conditions of being bounded and/or linear, Method 4 may not achieve the desired results. For example, if a system needs to be able to “clip,” or saturate, as op-amp circuits can, the system is no longer linear, and its configuration must be changed accordingly. Unlike analog circuits, though, when a digital system clips, the output values can be quite inaccurate, as the “number circle” example above demonstrated. It is possible, though, to establish a C240x-based system which will easily and reliably model the clipping process of an analog system. It will provide the C240x sufficient “headroom” (Method 1) to model a linear system during calculations, but output “saturation” values when a certain limit is exceeded. Consider the number line in the following:



Using the product mode shifter (SPM 3), we can scale fractional values to the right by a total of seven bits, allowing a total of eight bits of integer range. If we chose the linear output range as ± 1 (fractional), the headroom — the distance from the saturation limits to the accumulator maxima — will be quite substantial (128 times as large as the linear range). Thus, we have considerable room for modeling signals of great range without overflows within the C240x.

However, we can clip the signal before we send it to an output device by limiting the output to the saturation value whenever the true value exceeds the saturation limit. This process can be performed quite efficiently using the code in the following figure.



This routine works by taking advantage of a bit of algebra, and the fact that the processor is using overflow protection.

By adding the value 7F00 0000 (the difference between the maximum accumulator value and the limits of a fractional number when SPM 3 had been selected) to the accumulator, we can see that values greater than the fractional limit (the positive saturation point) will be forced to 7FFF FFFF.

The second instruction would subtract 7F00 0000 from the accumulator. For values which had **not** exceeded the positive saturation limit, this effectively undoes the effect of the first instruction. But, for values which **had** exceeded the positive saturation limit, the accumulator will be returned to the positive saturation limit value itself.

The second two instructions perform the same process for the negative limit test.

Thus, values between the saturation limits are unchanged, and values in excess of the limits are replaced with the saturation (± 1) values — an effective emulation of clipping which may be passed to an output device — without loss of information within the C240x.

32-Bit Arithmetic

The SACH and SACL operations allow 32-bit values to be stored (in halves) to data memory. These long operands may be used to operate on the accumulator as subsequent input values for loading, addition, and subtraction, using the following instructions

Instructions for 32-bit Arithmetic

```
LACL <dma>
ADDS <dma>
SUBS <dma>
LACC <dma>,16
ADD <dma>,16
SUB <dma>,16
```

Notice that the order of operation on the low half and high half of a long word is not important. This is exemplified by having added in the B term in the reverse order of the other terms.

Example

Use of these instructions is demonstrated by the code for the equation:

32-bit Arithmetic Example

$$D_{32} = A_{32} + B_{32} - C_{32}$$

```
.bss A,2
.bss B,2
.bss C,2
.bss D,2

.text
LACC A,16
ADDS A+1
ADDS B+1
ADD B,16
SUB C,16
SUBS C+1
SACH D
SACL D+1
```

Division on the C240x

The C240x does not have an explicit divide instruction. Instead, the process is broken into a series of subtractions and shifts. An efficient and flexible division routine can be performed with the C240x by the use of the SUBC instruction, which implements one step of the long-division process.

SUBC Instruction

The process of long division on a standard microprocessor is time consuming and requires considerable manipulation to perform each step of the process. The SUBC instruction, effectively a “one-bit-divide,” on the C240x makes this process more efficient.

In each step of the division routine, a test must be done to see if the denominator will “go into” the numerator. After such a subtraction is performed, the result is tested for a positive sign. If positive, the denominator did “go into” the numerator. The subtraction is then taken, and a one is loaded into the low accumulator (after left shifting the accumulator by one) to identify the positive test. If the accumulator tests negative, the denominator did not “go into” the numerator. In this case, the prior accumulator value is restored with a left shift and a zero is loaded at the LSB to identify the failed test. The shifting of the accumulator allows a new bit of the divide to be tested.

After N such conditional subtractions, the N-bit result of the division of two operands is in the lower accumulator with any remainder present in the upper accumulator.

Although not as quick as a single-cycle divide instruction, the division routine based on SUBC is an order of magnitude quicker than performing division without the availability of the “1-bit divide.” Since division is a seldom-used process in DSP, this compromise between silicon cost and performance is, in most cases, quite reasonable.

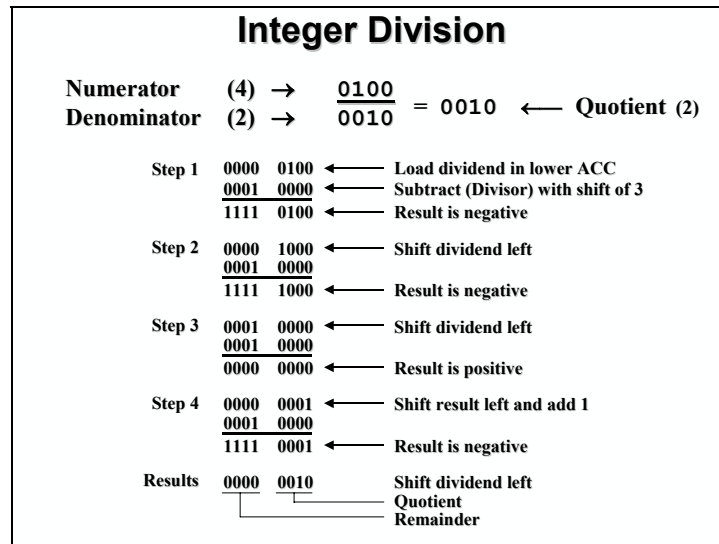
RULES FOR CONDITIONAL SUBTRACTION:

Conditional Subtraction is performed - If results are:

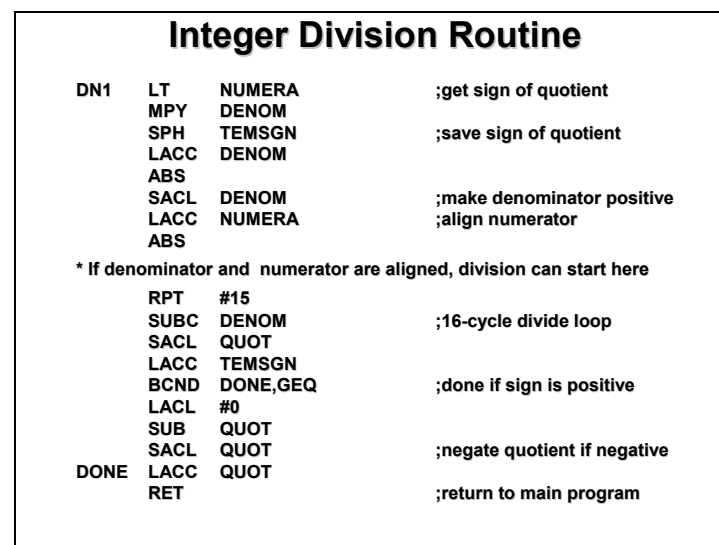
- Negative - then discard results and shift accumulator left by one bit and load “0” in LSB
- Positive - then shift accumulator results left by one bit and load “1” in LSB

Integer Division

The following figure shows a model of integer division using 4-bit operands. The extension to 16-bits is straightforward. Note that the SUBC process is repeated four times in a 4-bit integer system.



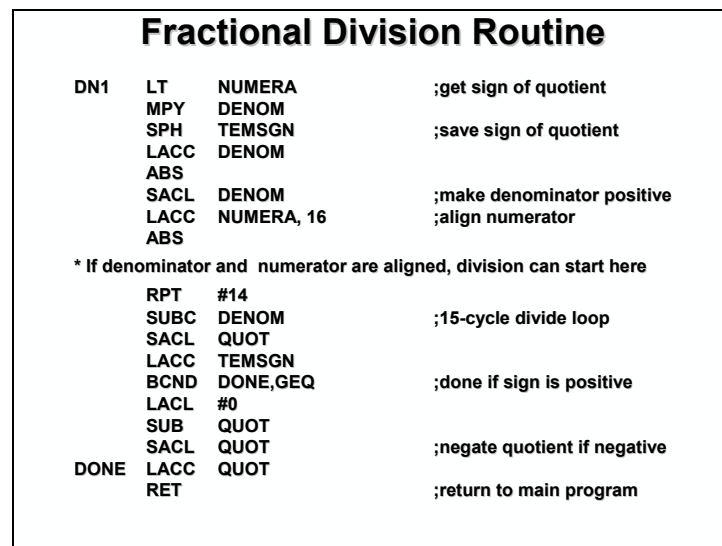
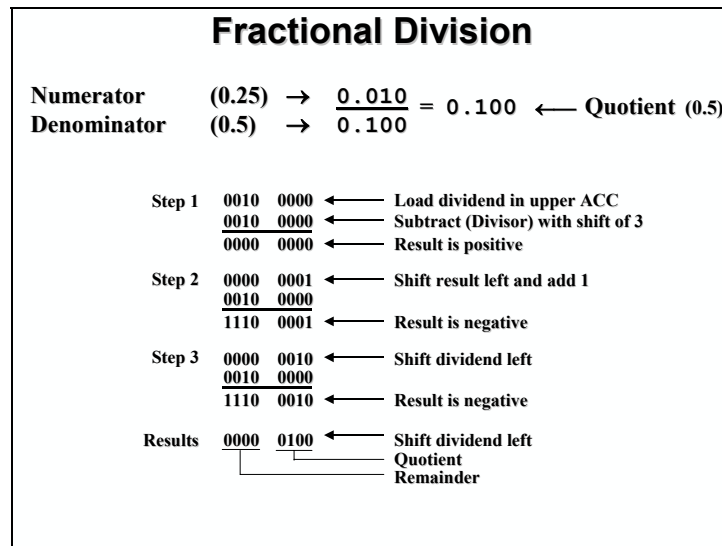
Notice the components that make up the division routine. In the center of the code is the repeat loop and conditional subtract. Since SUBC operates only on unsigned values, the division is preceded by a routine to determine the sign of the result and strip the sign from the input values before the division begins. Once the division concludes, the correct sign is attached to the unsigned result, completing the routine.



Fractional Division

Fractional division is identical to integer division with two exceptions:

- Numerator is loaded into the upper, not lower, accumulator.
- Only N-1 iterations are required for an N-bit fraction. The quotient is in the lower accumulator and the remainder is in the upper accumulator, as shown in the following figure.



Division Summary

Division Summary

$$Y = \frac{\text{NUM}}{\text{DEN}} = \text{QUO} + \text{REM}$$

INTEGERS

1. ACCL = NUM
2. RPT #15
SUBC DEN
3.

| | |
|-----|-----|
| REM | QUO |
|-----|-----|

ACCH ACCL

FRACTIONS

1. ACCH = NUM
2. RPT #14
SUBC DEN
3.

| | |
|-----|-----|
| REM | QUO |
|-----|-----|

ACCH ACCL

Lab 6: Exercise

➤ Objective

The objective of this exercise is to demonstrate your ability to write code which is consistent with the principles of number theory as presented in this module. The previous lab will be modified to use fractional values – Table: (0.1, 0.2, 0.3, 0.4, 0.8, 0.6, 0.4, 0.2, 0.0).

➤ Procedure

Copy Files, Create Make File

1. Using Code Composer, open LAB5.CMD in C:\DSP24\LABS and save it as C:\DSP24\LABS\LAB6.CMD.
2. Open LAB5.ASM and save it as LAB6.ASM.
3. Create a new project called LAB6.MAK and add LAB6.ASM, VECTORS.ASM and LAB6.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 → Options on the menu bar. Select the Assembler tab. In the middle of the screen check “Enable Source Level Debugging”. Next, select the Linker tab. In the middle of the screen select “No Autoinitialization”. Create a map file named LAB6.MAP. Select OK to save the Build Options.

Initialization Data (Table) Fractional Values

4. Edit LAB6.ASM and modify it by changing the initialization data (table) to the following fractional values – table: (0.1, 0.2, 0.3, 0.4, 0.8, 0.6, 0.4, 0.2, 0.0).
5. Modify the .ASM file as needed to perform fractional multiplication. Save your work.

Build and Load

6. Click the “Rebuild All” 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.
7. If the “Load program after build” option was not selected in Code Composer, load the output file into the target. Click: File → Load Program.... Then reset the DSP by clicking on: Debug → Reset DSP. Right click on the VECTORS.ASM window and select Mixed Mode to debug using both source and assembly.
8. Single-step your routine. While single-stepping, open memory windows to see the values located in *table [9]* and *data [9]*. Open the CPU and Status registers. Check to see if the program is working as expected. Debug and modify, if needed.

End of Exercise

Review

Review

1. When using multiplier, what types of numbers are recommended?
2. $Q15 \times Q15 = ?$
3. What is SPM and what are the various shift options used for?
4. How is the OV bit set?
How is the OV bit cleared?
5. How do you store a Q30 back to memory as a Q15?

Solutions

Exercise - Assembling Fractions

Write the assembly language statement to code the following numbers:

1. Code: 0.14

```
.int 32768 * 14 / 100
```

2. Code: 0.123

```
.int 32768 * 123 / 1000
```

3. Code: -1.0

```
.int 32768 * -1
```

```
;SOLUTION FILE FOR LAB6.ASM
        .def      start
        .bss      data,4
        .bss      result,1
Q        .set      32768
        .data
tableA:  .int      Q*1/10,Q*2/10,Q*3/10,Q*4/10
tableB:  .int      Q*8/10,Q*6/10,Q*4/10,Q*2/10
length  .set      ($-tableA)-($-tableB)
        .text
start:   LAR        AR0,#data
        MAR        *,AR0
        RPT        #length-1
        BLPD       #tableA,**
        LAR        AR0,#data
        MPY        #0
        LACL       #0
        SPM        1
        SETC       SXM
        RPT        #length-1
        MAC        #tableB,**
        APAC
        LDP        #result
        SACH       result
end      B          end
```

Implementing Algorithms

Introduction

This module explains basic techniques for implementing algorithms. The main objective is to show how the C240x instructions map directly from a signal flow diagram, difference equations, or a transfer function. Therefore, given a system description, a C240x algorithm can be implemented. In this module FIR and IIR filters are used ONLY as an example. The purpose is not to learn digital filtering, but to use these structures as a learning tool – since they are commonly used and easily understood.

Learning Objectives

Learning Objectives

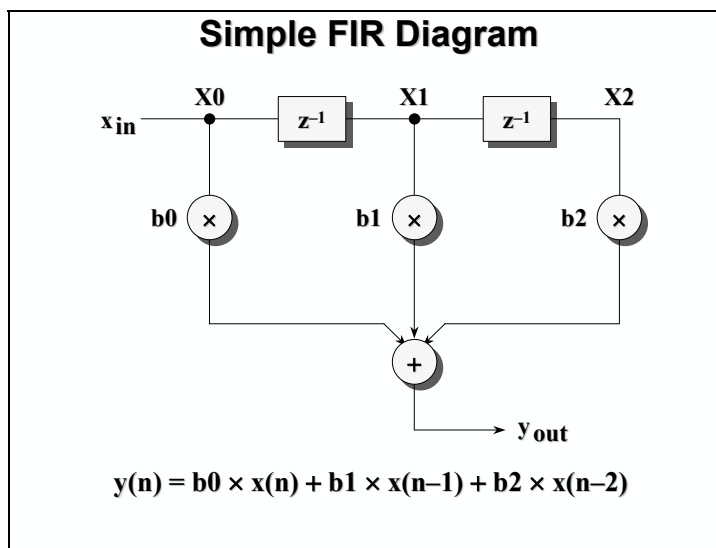
- ◆ Implement unit delay in digital systems
- ◆ Develop TMS320C240x code from signal flow diagrams
- ◆ Show digital filter implementation on the TMS320C240x
- ◆ Improve stability in digital systems

Module Topics

| | |
|--|-------------|
| Implementing Algorithms..... | 7-1 |
| <i>Module Topics.....</i> | <i>7-2</i> |
| <i>FIR Filters.....</i> | <i>7-3</i> |
| <i>Delay Line Implementation.....</i> | <i>7-5</i> |
| <i>FIR Filter Implementation</i> | <i>7-7</i> |
| FIR Filter Using LTD and MPY | 7-8 |
| FIR Filter Using Immediate Coefficients | 7-8 |
| FIR Filter Using MACD..... | 7-9 |
| FIR Filter Implementation Considerations | 7-11 |
| FIR Filter Initialization..... | 7-12 |
| <i>IIR Filters.....</i> | <i>7-13</i> |
| IIR Filter Signal Flow Diagram..... | 7-13 |
| IIR Filter Implementation..... | 7-14 |
| Noise in the IIR Filter..... | 7-15 |
| Input Scaling..... | 7-17 |
| Double APAC..... | 7-18 |
| Break Down of High-Order Systems..... | 7-19 |
| IIR Coding Example..... | 7-19 |
| <i>IIR vs. FIR Filters</i> | <i>7-22</i> |
| <i>Lab 7: Filter Lab.....</i> | <i>7-23</i> |
| <i>Review.....</i> | <i>7-26</i> |
| Solutions..... | 7-27 |

FIR Filters

In this section, the Finite Impulse Response (FIR) filter will be investigated and its concept and implementation will be considered. Several methods of implementing the FIR filter will be presented. This section will conclude with a demonstration of how the FIR filter is derived, how it is coded, and how it operates on an input signal.



FIR Filter Operation

In the above diagram, the input $X(0)$ appears at location $X0$. At this time, $X(0)$ is multiplied by $b0$, and appears at $Yout$. One unit of time later, input Xin has the new value $X(1)$. The value $X(0)$ moves through the time delay and is now at $X1$, making room for $X(1)$ at location $X0$. At this time:

$$Yout = X(1) * b0 + X(0) * b1$$

This process is updated and repeated for each unit of time, and is equated as:

$$Y0 = X0 * b0 + X1 * b1 + X2 * b2$$

What is the relationship between the terms $X(1)$, $X1$, and $X(n-1)$?

FIR Filter Definition

A linear time-invariant system can be described by the difference equation:

$$y(n) = -\sum_{k=1}^N a_k * y(n-k) + \sum_{k=0}^M b_k * x(n-k)$$

A FIR system is one for which the response to a unit sample impulse has finite duration. In a FIR system, all of the a_k terms are zero; therefore, the output of an FIR system can be described by:

$$y(n) = \sum_{k=0}^M b_k * x(n-k)$$

It can be seen that the output $y(n)$ of a FIR system is the weighted sum of the current input $x(n)$ and the previous M input samples, and that the response to a unit impulse will be zero for $n > M$.

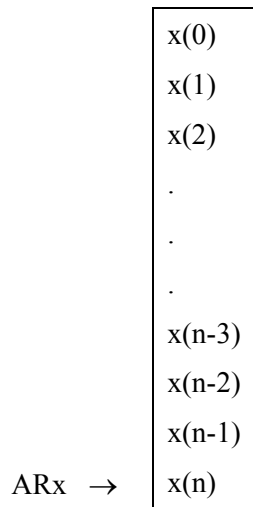
A FIR filter is characterized by unconditional stability and may be designed to produce linear phase response.

Delay Line Implementation

Having looked at the concept of the FIR filter in the previous section, the next challenge will be to develop an efficient model of the FIR filter with the C240x. The FIR is essentially a sum-of-products operating on an array of values maintained in a “delay line.” As you have seen, the sum-of-products arithmetic is simple to implement with the C240x. But how can the delay line be implemented?

The simplest way to implement a delay line in a microprocessor is using a buffer in memory, where an N-tap filter operates on the most recent N samples in the array. Each time the FIR is performed, a new data value is acquired and added to the end of the data list.

In this way, a single auxiliary register can be decremented through the array during the calculation of the FIR and incremented to the next open position in the array to input a new data value before calculating the FIR again.



The advantage of this approach is in the efficiency with which the C240x can maintain the delay line. The AR may be auto-incremented and auto-decremented during the sum-of-products calculation, thus, all of the C240x's processing power can be devoted to performing the calculation rather than managing the delay line.

The disadvantage of the linear buffer is in the use of memory. As each new data value is acquired, another location of memory is needed. In a continuous system, this would require endless amounts of memory — an impractical solution.

Linear Buffer

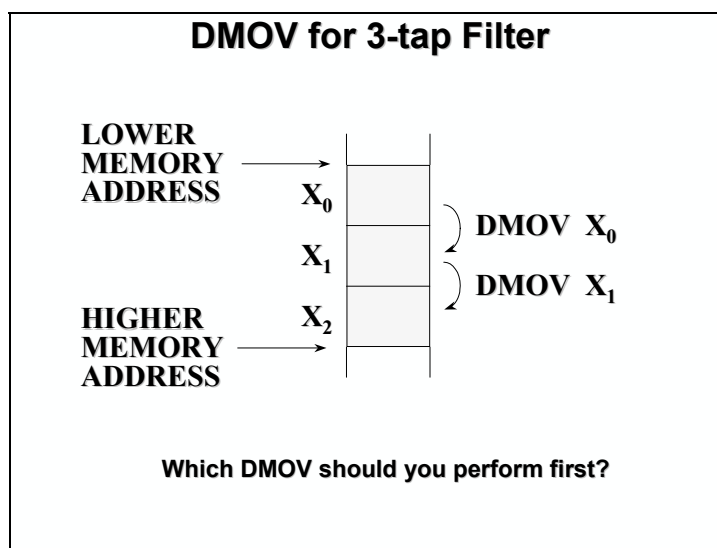
The C240x allows a type of buffering sometimes called a ripple buffer. After each new output is calculated, the data is moved from location to location in a rippling fashion, in much the same way that the signal flow diagram implies.

Using a standard processor, the time required to move data in this fashion would be prohibitive, requiring a load and store operation for each tap of the filter. With the C240x, a single load and store operation is combined in the DMOV instruction:

```
DMOV    <dma>                ;copy from <dma> to <dma+1> in on-chip RAM
                        ;(blocks B0, B1, B2)
```

The value in location <dma> is **copied** to location <dma> + 1. **This instruction affects neither the accumulator nor location <dma>.**

Using our previous 3-tap FIR filter:



The process can be even more efficient since the DMOV instruction can be merged with the LTA (load the temporary register and accumulate) function as LTD:

```
LTD     <dma>                ;load T register with contents of <dma>, accumulate
                        ;previous product, and copy value in location <dma>
                        ;to location <dma>+1
```

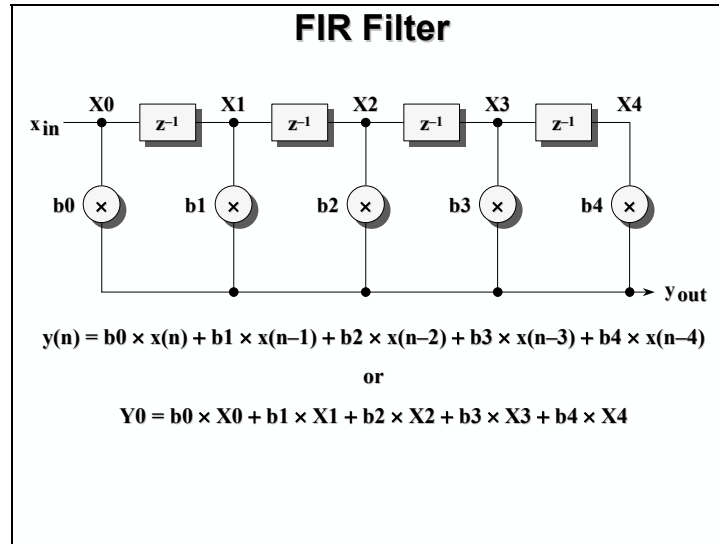
The DMOV function is included in the LTD and MACD instructions. The DMOV instruction works only with on-chip RAM block 0-2 when configured as data memory. If you attempt to use the DMOV on external memory, the location will be read, but no write will be performed.

Lastly, note that when performing operations in a ripple buffer, it is necessary to proceed from the oldest value to the newest. Running from newest to oldest with DMOV would not implement a delay line, but instead would copy the newest value through the entire delay line.

FIR Filter Implementation

The total solution for the FIR filter may now be implemented by combining the DMOV operation and IN/OUT instructions with the sum-of-products equation.

Consider the following signal flow diagram of a fourth-order FIR filter.



The implementation of this filter may be realized in several different ways as described in the following sections.

FIR Filter Using LTD and MPY

As seen earlier, the LTA and MPY instructions can be used to implement a sum-of-products operation. Adding the DMOV operation to the LTA instruction via the LTD instruction handles the rippling of data through the delay line, when the array is accessed from the bottom up. Finally, the inclusion of the IN and OUT instructions allow the filter to send and receive data from ADCs and DACs. The code segments below are the same, with the exception of addressing modes used.

| FIR Filter - Linear Buffer | | | |
|----------------------------|-----------|---------------------|------|
| Direct Addressing | | Indirect Addressing | |
| | LDP #X0 | LDP #X0 | |
| | | MAR *,AR1 | |
| | | LOOP LAR AR2,#B4 | |
| | | LAR AR1,#X4 | |
| START | LACL #0 | LACL #0 | |
| | LT X4 | LT *,AR2 | |
| | MPY B4 | MPY *,AR1 | |
| | LTD X3 | LTD *,AR2 | |
| | MPY B3 | MPY *,AR1 | |
| | LTD X2 | LTD *,AR2 | |
| | MPY B2 | MPY *,AR1 | |
| | LTD X1 | LTD *,AR2 | |
| | MPY B1 | MPY *,AR1 | |
| | LTD X0 | LTD *,AR2 | |
| | MPY B0 | MPY *,AR1 | |
| | APAC | APAC | |
| | SACH Y,1 | SACH Y,1 | |
| | OUT Y,PA0 | OUT Y,PA0 | |
| | IN X0,PA1 | IN X0,PA1 | |
| B | START | B | LOOP |

FIR Filter Using Immediate Coefficients

The above code used coefficients stored in data memory. The MPY instruction can also use **immediate** 13-bit values. This reduces data memory usage from two arrays of values to one, with the coefficient array being “built into” the MPY instruction. As seen below, it is almost identical to the direct addressing code:

| FIR Filter Implementation using LT and Multiply with Constant | | | |
|---|-----------|--|--|
| | LDP #X0 | | |
| START | LACL #0 | | |
| | LT X4 | | |
| | MPY #B4 | | |
| | LTD X3 | | |
| | MPY #B3 | | |
| | LTD X2 | | |
| | MPY #B2 | | |
| | LTD X1 | | |
| | MPY #B1 | | |
| | LTD X0 | | |
| | MPY #B0 | | |
| | APAC | | |
| | SACH Y,4 | | |
| | OUT Y,PA0 | | |
| | IN X0,PA1 | | |
| B | START | | |

FIR Filter Using MACD

As seen earlier, the MAC instruction performs both the LTA and MPY operations. For MAC to operate, one array is located in program memory.

In the FIR filter, the need for the DMOV function is added to MAC in the MACD instruction. Since DMOV operates only to increment the delay line values in data memory, it is necessary to put the data values in data memory, accessed from the bottom up. Coefficient values, therefore, will be located in program memory. Also, since the coefficients are in program space and may be accessed only in incrementing order, they must be listed from last to first to match the order in which the data values will be accessed.

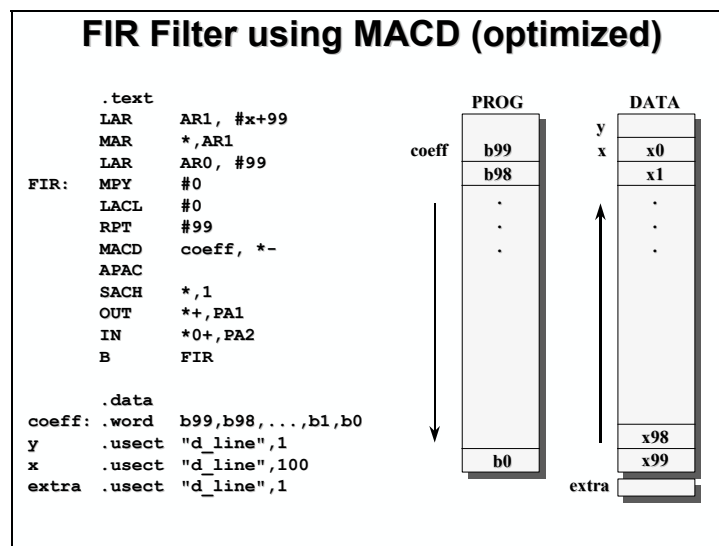
FIR Filter using MACD

```

INIT:  SETC    SXM           ;sign extension mode
        SPM     1           ;q31 accumulator
        CLRC    OVM         ;allow intermediate overflow

        SETC    CNF         ;move B0 from 200h to 0FF00h
        MAR     *,AR1       ;select AR1
FIR:   LAR     AR1,#304      ;AR1 set to bottom of data array
        MPY     #0          ;clear P-register
        LACL    #0          ;clear accumulator
        RPT     #4          ;5 terms (4th-order filter)
        MACD    0FF00h, *-  ;b*x, apac, inc b, dec x
        APAC                    ;accumulate last partial product
        SACH    Y           ;store final result to Y as Q15
        OUT     Y,PA1       ;send results to DAC on PA1
        IN      *,PA0       ;read next input on PA0 to 300h
        B       FIR        ;done, branch and re-do

```



```

.text
LAR   AR1, #x+99           ; point to end of delay line
MAR   *,AR1                ; AR1 active
LAR   AR0, #99             ; array length

FIR:  LACL  #0
      MPY   #0
      RPT   #99             ; 100 iterations
      MACD  coeff, *-
      APAC                  ; final sum
      SACH  *,1             ; store result to Y
      OUT   *+,PA1          ; write to DAC, increment to X0
      IN    *0+,PA2         ; input and reset AR1
      B     FIR             ; loop back

.data
coeff: .word  a99,a98,...,a1,a0 ; store array old to new ...

y      .usect  "d_line",1      ; output value
x      .usect  "d_line",100    ; input array - link to DARAM
extra  .usect  "d_line",1

```


FIR Filter Implementation Considerations

FIR filters can be implemented in several different ways on the C240x. The particular approach used will depend upon the system requirements. The selections allow the user to make tradeoffs between performance, program memory utilization, and data memory utilization.

When you implement an FIR filter using the C240x, you will want to consider the conditions described in the following paragraphs.

Coefficient Storage

Coefficients may be placed either in program or data memory. In program memory, coefficients may be accessed as immediate values or via the MAC (multiply/accumulate) group of instructions. In general, the highest performance will be achieved with coefficients in program memory, because this allows operands to be passed simultaneously to the multiplier via the program and data buses.

In-line or Looped Code

Filters may be implemented using either in-line or looped code. In-line code gives higher performance than looped code, but generally at the cost of increased program memory usage.

Single Repeat

The C240x has hardware support for single-instruction repeats (RPT). The highest performance FIR filters can be achieved using the single-instruction repeat, but this requires that the coefficients be located in program memory. The single-instruction repeat operation cannot be interrupted, which may be either an advantage or disadvantage.

Types of Multiplier Operations

The C240x provides several multiplier operations. Both multiply arguments may be presented to the multiplier from data memory, or one can come from program memory and one from data memory. For long filters, the multiply/accumulate (MAC) group of instructions, which pull operands from both program and data memory, used in repeat mode will give the highest performance. For shorter filters, it may be preferable to use the load-T/multiply (LT/MPY) group of instructions because there is less setup overhead.

FIR Filter Initialization

In review, the general approach to implementing an FIR filter has elements in common with the implementation of any other algorithm. The program must initialize CPU resources, memory to be used, and any registers used.

CPU

You should ensure that CPU configurations, such as product shift mode, sign extension, overflow mode, and memory configuration are set appropriately. Since these configurations are initialized at reset, they may not have to be set explicitly, but take care to ensure that all configurations are as desired.

Memory

The coefficients and sample delay memory must be initialized. Depending on the implementation approach, coefficients may be immediate constants or they may be located in either program memory or data memory. The sample delay memory (input data) may be initialized to all zeros.

Registers

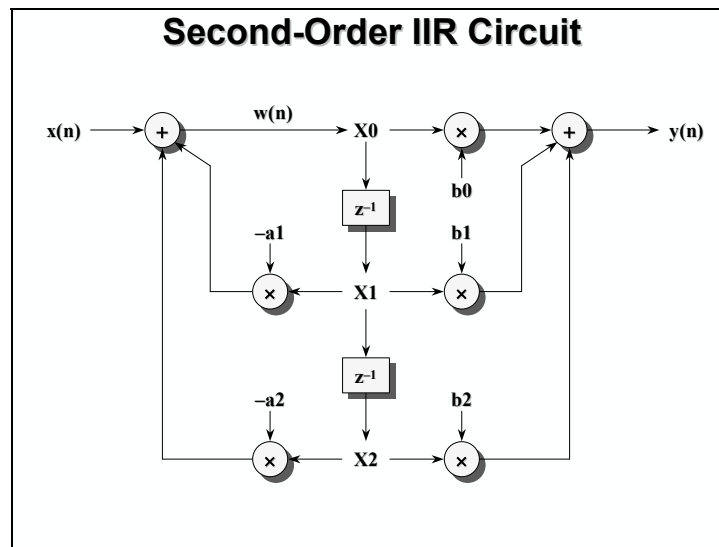
Registers used for addressing or storage of intermediate results may need to be initialized. This includes auxiliary registers and the AR pointer, and may include the accumulator and P registers.

IIR Filters

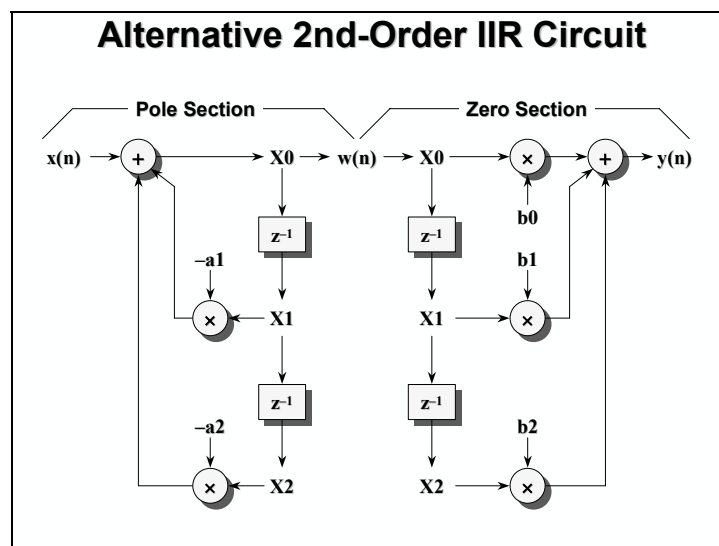
In this section the Infinite Impulse Response (IIR) filter will be investigated. Its concept and implementation will be considered. The standard method of implementing the IIR filter will be presented. This section will conclude with a discussion of noise sources and how to minimize them.

IIR Filter Signal Flow Diagram

The IIR filter is like two joined FIR-type filters. At first glance, the IIR filter signal flow in the following figure is difficult to follow and presents no obvious method of software implementation.



To aid understanding, the complex diagram in the figure above can be envisioned as two simpler signal flow paths: a feedback (pole) section and a feed-forward (zero) section as shown in the next figure.



IIR Filter Implementation

The software implementation of the IIR filter is derived directly from the alternative signal flow diagrams. The first section to code is the feedback section, which will provide the new intermediate term needed by the FIR section. Once this new intermediate term (X_0) is determined, the FIR section can be coded as you saw earlier in this course.

The method for converting a signal flow diagram to 320 code involves working back from the output. The final term, X_0 , is derived from the summation of one input term and two feedback terms. Thus, the summation is to be coded first. A final issue is to remember that the accumulator will hold Q30 numbers as the result of fractional multiplications, and the addition of an input value (a Q15) must take a shift ($30 - 15 = 15$) to align the binary points.

IIR Filter Implementation

```

LDP    #X
SPM    0                                ;no product shift, acc = Q30
IIR    IN    X,PA1                      ;input new sample from PA1, put in Xin
LACC   X,15                            ;load acc with input Xin as Q30 (q15 + s15)
LT     X1                               ;T = X1
MPY    A1                               ;P = X1*A1
LTA    X2                               ;T = X2, acc = Xin + X1*A1
MPY    A2                               ;P = X2*A2
APAC   X0,1                            ;acc = Xin + X1*A1 + X2*A2 (in Q30 format)
SACH   X0,1                            ;store acc (above) to X0 (as Q15 format)
LACL   #0                              ;acc = 0 (clear accumulator)
MPY    B2                               ;P = X2*B2 (last T value reused)
LTD    X1                               ;T = X1, acc = X2*B2, X1 -> X2
MPY    B1                               ;P = X1*B1
LTD    X0                               ;T = X0, acc = X2*B2 + X1*B1, X0 -> X1
MPY    B0                               ;P = X0*B0
APAC   X0,1                            ;acc = X2*B2 + X1*B1 + X0*B0
SACH   Y,1                             ;store acc (above) to Y (as Q15 format)
OUT    Y,PA0                           ;output sample to PA0
B      IIR                             ;loop back

```

IIR Filter Implementation (optimized)

```

LDP    #X
SPM    0                                ;no product shift, acc = Q30
IIR    IN    X,PA1                      ;input new sample from PA1, put in Xin
LACC   X,15                            ;load acc with input Xin as Q30 (q15 + s15)
LT     X1                               ;T = X1
MPY    A1                               ;P = X1*A1
LTA    X2                               ;T = X2, acc = Xin + X1*A1
MPY    A2                               ;P = X2*A2
MPYA   B2                               ;P = X2*B2, acc = Xin + X1*A1 + X2*A2
;      APAC   X0,1                      ;acc = Xin + X1*A1 + X2*A2 (in Q30 format)
SACH   X0,1                            ;store acc (above) to X0 (as Q15 format)
LACL   #0                              ;acc = 0 (clear accumulator)
;      MPY    B2                               ;P = X2*B2 (last T value reused)
LTD    X1                               ;T = X1, acc = X2*B2, X1 -> X2
MPY    B1                               ;P = X1*B1
LTD    X0                               ;T = X0, acc = X2*B2 + X1*B1, X0 -> X1
MPY    B0                               ;P = X0*B0
APAC   X0,1                            ;acc = X2*B2 + X1*B1 + X0*B0
SACH   Y,1                             ;store acc (above) to Y (as Q15 format)
OUT    Y,PA0                           ;output sample to PA0
B      IIR                             ;loop back

```

Noise in the IIR Filter

In this section, the IIR filter will be investigated in greater detail. The focus will be on noise sources and how to minimize them. Most of the issues presented here are also applicable to the FIR filter, but are of less concern with it due to its non-recursive nature. However, the IIR filter uses outputs as part of the next input, so errors can be compounded over time.

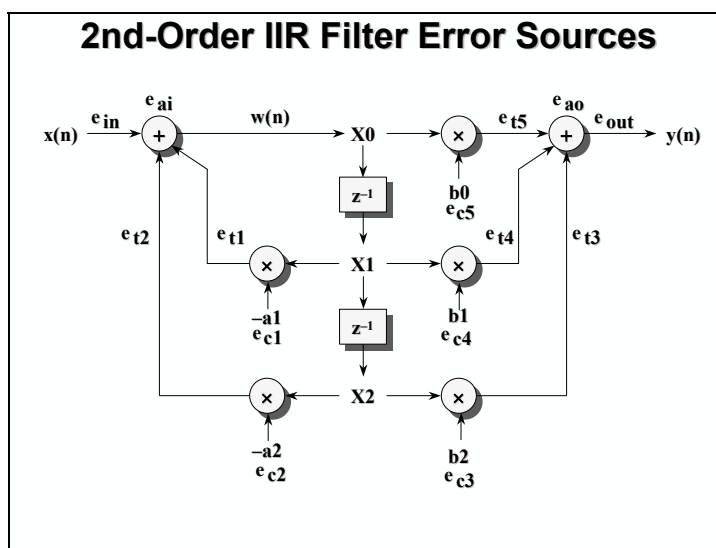
Two types of failures are possible with the IIR filter: overflow and underflow. Overflow occurs when the signal exceeds the system's boundaries. Underflow can occur in the absence of a robust input signal.

To assure the proper operation of the system, these conditions must be guarded against. In so doing, the sources of error must be identified, and the possibilities of catastrophic error eliminated. Guided by this knowledge, a system can be designed to meet the requirements with a high degree of confidence in its reliability.

Error Sources

Numerical calculations present several sources of error, including:

1. **I/O Signal Quantization** — The number of bits of precision available in A/D and D/A converters determines the errors that are attributable to conversion. The dominant factor is in A/D conversion, as input error circulates in and around the filter, while D/A error has no effect upon the filter itself.
2. **Filter Coefficient Quantization** — The result of imprecise coefficient representation is a deviation from the desired frequency response of the system. In most cases 16-bit coefficients suffice. However, in the case of *high-Q* filters with poles near the unit circle, small coefficient errors could result in a pole occurring outside the circle. This condition yields an unstable system.
3. **Uncorrelated Round-Off (Truncation) Noise** — This noise is the result of a system whose outputs are too small to represent with sufficient accuracy. As the number of bits available in the output decreases, the random LSB error becomes increasingly significant.
4. **Correlated Round-Off (Truncation) Noise** — The existence of limit cycles and their worst case example, overflows, is attributable to the effects of accumulated errors. This phenomenon is difficult to model. It is often best dealt with by using highly accurate coefficients and assuring the availability of sufficient dynamic range for input and output signal representation.
5. **Dynamic Range Constraint** — The 16-bit operands used by the TMS320 devices allow for a 96 dB dynamic range. The dynamic range is extended to 192 dB in the math channel (multiplier and accumulator) for intermediate calculations.



To best associate these error phenomena with the IIR filter, consider the diagram in the figure above. Notice that error sources exist at each step of the signal flow.

| | |
|-----------------------|----------------------------------|
| e_{in} | A/D quantization error |
| e_{out} | D/A quantization error |
| $e_{t1} \dots e_{t5}$ | Partial product truncation error |
| e_{ai}, e_{ao} | Accumulation error |
| $e_{c1} \dots e_{c5}$ | Coefficient quantization error |

Assessing the total cumulative error appears complex until you recognize that the internal sources are, or can be made, smaller than the I/O quantization error. If you design the system with care, the error can be considered to be entirely I/O-based and becomes easy to model.

To achieve optimal results from the IIR filter, you are required to maintain the maximum precision possible. Several programming approaches are offered below as solutions to the most commonly encountered problems.

Input Scaling

As noted in the IIR filter noise model, it is desirable to have coefficients as large as possible in order to maintain their precision. In so doing, it is possible to yield a system whose gain is greater than one.

In an earlier discussion, we determined that all operands must be less than unity in order to be representable as fractions. If the input X is bounded by unity, and the maximum gain H of the filter is 7, what maximum output Y must we anticipate? This can be represented by the equation:

$$X * H = Y$$

where:

$$|X| < 1 \quad \text{and} \quad H_{\max} = 7$$

therefore:

$$Y_{\max} = 7$$

Since the value of 7 cannot be represented as a fraction, something must be done with either X or H to allow Y to be representable. Two choices are:

$$Y = X * \frac{H}{7} \quad \text{and} \quad Y = H * \frac{X}{7}$$

At first glance, it would seem that either alternative causes a loss of accuracy, since reducing the X terms or H terms by 7 amounts to almost 3 bits of precision. Since the H array is the cause of the overflow, a designer often will reduce the gain of the filter. In so doing, however, he will also reduce the precision of all the coefficients.

A better method is to scale the input. This is especially easy if the input is from an A/D converter that provides less than 16 bits of data. For example, if a 12-bit A/D were used in this system, an apparent input range of ± 1 would result by connecting the A/D outputs to the 12 most significant bits (MSBs) of the data bus lines on the TMS320 (and grounding the remaining 4 least significant bits (LSBs)). Instead, by connecting these 12 A/D lines to the 12 LSBs of the data bus (and connecting the 4 remaining MSBs of the data bus to the MSB of the A/D — thus creating a hardware sign extension mode), an effective division by 16 occurs on the input range. How much precision is lost in this process? Since there are still 12 bits of precision, no loss of accuracy results, and a representable output is attained.

Even if a 16-bit input is provided, input scaling is possible with no loss of accuracy in an IIR filter. Note in the IIR filter code that the input is first stored to data memory and then loaded to the accumulator in Q30 format, using a shift of 15. If the shift were reduced to 14, the input would be effectively divided by 2. The division by 7 in the above example is best handled by a 3-bit shift (a divide by 8). The division by the next larger power of two allows for a representable output with a loss of less than 1 bit of precision.

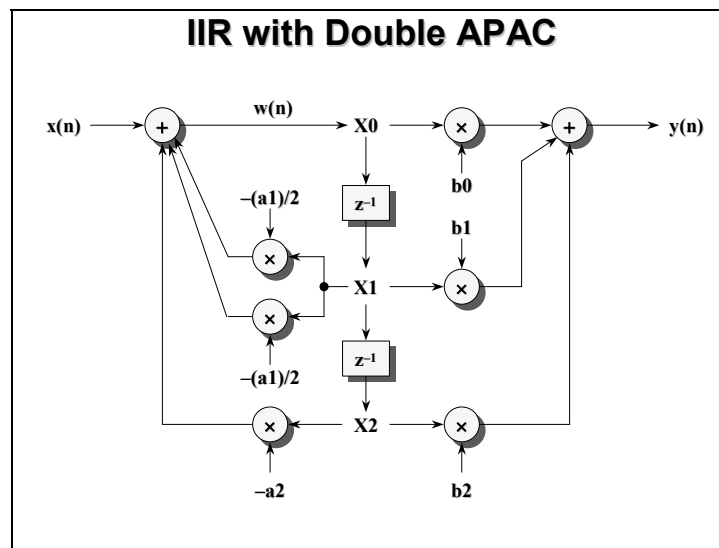
Double APAC

The coefficients derived in many second order IIR filters often include one which exceeds unity. A sample set follows:

$$\begin{array}{ll} b_0 = 0.77 \\ a_1 = 1.93 & b_1 = 0.02 \\ a_2 = 0.83 & b_2 = 0.34 \end{array}$$

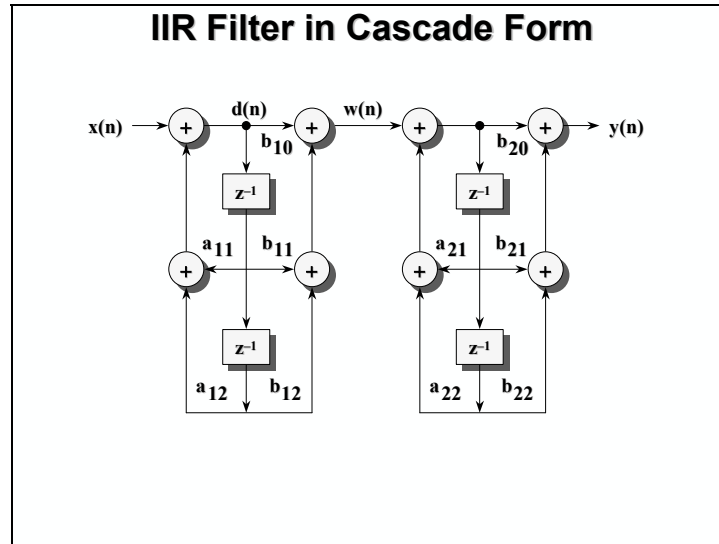
The coefficient a_1 is too large to represent as a fraction. How then can this system be implemented by the TMS320? One solution is to divide all coefficients by 1.93, so that the largest value will then be unity. This approach costs almost one bit in the remaining 5 coefficients, especially undesirable in coefficient b_1 . A second method is to only divide a_1 by the next larger integer value (in this case 2). With no correction, this would alter the performance of the filter. Therefore, the product of $a_2 * X_2$ must be summed to the accumulator the number of times a_1 was scaled down (again, in this case, by 2). This accumulation is easily re-summed by the repeated use of the APAC instruction after the creation of the product $a_2 * X_2$. The extra instruction time is a small price to pay for extra precision in a demanding filter application.

Additionally, the term a_0 is between the feedback summation and X_0 . In this model, a_0 is defined as 1 so the process of multiplying by 1 is skipped and time is saved. However, if you scale down the coefficients, you must remember to scale down the a_0 term.



Break Down of High-Order Systems

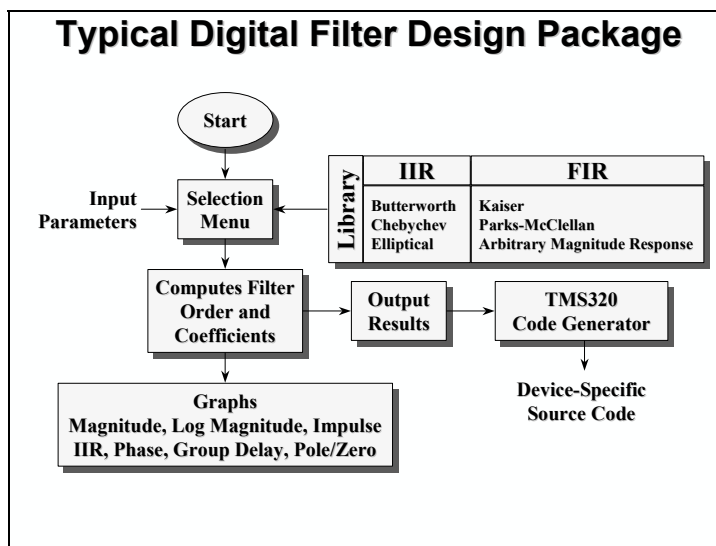
Our discussion has considered the second-order IIR filter. What if the desired performance requires a higher order system? The obvious model for a fourth order IIR filter is one possessing four time delays, and generally is a deeper version of the second-order system. Although conceptually correct, this method is inappropriate in application. The limitations of a 16-bit word size are quickly reached with high-order systems; therefore, it is best to break the high-order system into a series of cascaded second-order sections as shown in the following figure. An additional benefit of this approach is that the code for the IIR presented earlier becomes a standard for all IIR systems.



IIR Coding Example

An example filter design will be presented to demonstrate many of the concepts of IIR filter implementation.

The first step is to define the required performance. Once this is done, the required order of the filter and the value of the coefficients must be determined. This preliminary step can be executed in a variety of ways, given a sufficient background in DSP theory and implementation. For those not possessing vast knowledge of DSP theory, a simple solution exists in various software package capable of assisting in the design of digital filters.



The typical digital filter design package prompts you for details of the system: the sample rate, the desired frequency response, and type of filter to use. The program will then determine the values of the coefficients necessary to meet the given criteria.

Typically, you are offered the option to view and plot the performance of the filter in several ways. If satisfied with the performance, you can then request that the coefficients be incorporated into code which will implement the filter function on the C240x. The created code would be commented and include several of the scaling operations discussed earlier. See the example on the next page.

| IIR Elliptic Lowpass Filter | | | | | |
|--|-----------|-----------|---------|----------|---------|
| INFINITE IMPULSE RESPONSE (IIR) | | | | | |
| ELLIPTIC LOWPASS FILTER | | | | | |
| 16-BIT QUANTIZED COEFFICIENTS | | | | | |
| FILTER ORDER = 4 | | | | | |
| SAMPLING FREQUENCY = 8.000 KILOHERTZ | | | | | |
| I | A(I,1) | A(I,2) | B(I,0) | B(I,1) | B(I,2) |
| 1 | -1.276855 | .513092 | .107407 | .016239 | .107250 |
| 2 | -1.322205 | .885345 | .620819 | -.690979 | .620758 |
| *** CHARACTERISTICS OF DESIGNED FILTER *** | | | | | |
| | BAND 1 | BAND 2 | | | |
| LOWER BAND EDGE | .00000 | 1.20000 | | | |
| UPPER BAND EDGE | 1.00000 | 4.00000 | | | |
| NOMINAL GAIN | 1.00000 | .00000 | | | |
| NOMINAL RIPPLE | .05000 | .05000 | | | |
| MAXIMUM RIPPLE | .04438 | .04287 | | | |
| RIPPLE IN dB | .37714 | -27.35642 | | | |

IIR vs. FIR Filters

The IIR and FIR filters are compared in this section.

IIR Filter

The IIR filter is a good choice in designs where amplitude response is the primary performance criteria. The IIR filter has more affect on the gain over frequency than the FIR for a given filter length and number of instruction cycles. In fact, the IIR filter is 5 to 10 times more efficient than the FIR filter in controlling amplitude response.

FIR Filter

The FIR filter is the preferred choice in systems where phase response is an important parameter. The nature of the FIR filter permits a linear, predictable phase error not possible with an IIR filter.

A second benefit of the FIR filter is stability. An IIR filter contains feedback elements. All recursive systems can be made to fail under certain conditions, therefore they require more care and analysis if they are to be reliably used. The FIR filter, containing no recursive elements, is immune to these problems and presents a simple and sure solution for the programmer.

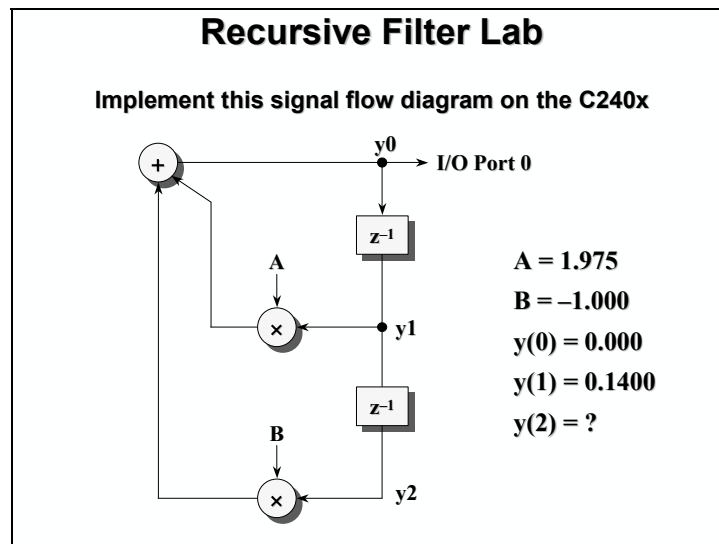
A final factor in favor of using the FIR filter is the greater likelihood of being able to use the MPY with 13-bit constant instruction, and the associated savings of data RAM with its initialization.

Lab 7: Filter Lab

➤ Objective

The objective of this lab is to apply the techniques discussed in Module 7 for implementing signal flow diagrams on the C240x. Use the values for A, B, and the initial conditions noted below.

Provisions have been made using a BANY loop to run the proper number of samples to plot the output using the Code Composer graphing features. Show your results to the instructor for verification.



Notes

1. As you now know, y_0 is the current output value — in this system, it is equal to the weighted sum of two past values of the output $y_1 + y_2$. In this system, the output from times 0 and 1 are provided, allowing your code to begin with the solution to y at time 2.
2. There is no need for a location in memory for y_0 , since it is not an input value in the calculation.
3. You may choose to implement this routine based on this information alone, or you may use the following procedure.

➤ Procedure

Create Make File

1. **NOTE:** LAB7.ASM and LAB7.CMD files have been provided as a starting point for the lab and need to be completed. *DO NOT copy files from Lab 6 to Lab 7.*
2. Create a new project called LAB7.MAK and add LAB7.ASM, VECTORS.ASM and LAB7.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 → Options on the menu bar. Select the

Assembler tab. In the middle of the screen check “Enable Source Level Debugging”. Next, select the Linker tab. In the middle of the screen select “No Autoinitialization”. Create a map file named LAB7.MAP. Select OK to save the Build Options.

Implement Signal Flow Diagram

3. Edit LAB7.ASM and modify it to implement the signal flow diagram on the lab slide.
NOTE: Only edit the “sine loop” section as noted in the comment fields. The initialization section has been completed. Review it to be sure you are using the proper variable names and constants. The code below the “sine loop” should not be modified. It is used to buffer the samples for the graph. Save your work.

Build and Load

4. Click the “Rebuild All” 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. If the “Load program after build” option was not selected in Code Composer (“Option” menu, click on “Program Load...”) load the output file into the target. Click: File → Load Program...

Then reset the DSP by clicking on: Debug → Reset DSP

Right click on the VECTORS.ASM window and select Mixed Mode to debug using both source and assembly.

Setup Graph

6. Open and setup graph. Click: View → Graph → Time/Frequency... and set the following values:

| | |
|-------------------------|-----------------------|
| Start Address | buf_addr |
| Acquisition Buffer Size | 50 |
| Display Data Size | 225 |
| DSP Data Type | 16-bit signed integer |
| Q-value | 15 |

Select OK to save the graph options.

Set Probe Point and Connect to Graph

7. Open LAB7.ASM and place the cursor on the final “B sine” instruction at the end of the program. Right click the mouse key and select Toggle Probe Point. Notice that line is highlighted indicating that the Probe Point has been set.

8. Connect the Probe Point. On the menu bar click: `Debug` → `Probe Points...` and select the Probe Point in the lower part of the window. Next open the "Connect To:" field and select `Graphical Display`. Click on `Add`. Select `OK` to save and close the Probe Point setup window.

Run and View Graph

9. Select the `VECTORS.ASM` window. Run by using the `<F5>` key, or using the `Run` button on the vertical toolbar, or using `Debug` → `Run` on the menu bar. Watch the graph display as it is updated. To Halt, use `Shift <F5>`, or the `Halt` button on the vertical toolbar. Show your results to the instructor for verification.

End of Exercise

Review

Review

1. Why do you move data when implementing a delay line?
2. What instruction is used to move the data in a delay line?
3. What is the single most powerful instruction? Why?
4. How many operations does it perform?

Solutions

; SOLUTION FILE FOR LAB7.ASM - SINE WAVE GENERATOR

```

                .def      start

                .bss      A,4,1          ;all symbols on one page
B               .set      A+1           ;assign all locations off 1st element
Y1              .set      A+2
Y2              .set      A+3

                .data
table:          .int      32768 * 1975/2000    ;A value
                .int      32768 * -1         ;B value
                .int      32768 * 0          ;Y1 value
                .int      32768 * 14/100     ;Y2 value
length:         .set      $-table           ;length of the data table

buf_size        .set      50               ;buffer size for graph
buf_addr        .usect    "buffer",buf_size,1 ;buffer address for graph

WDCR            .set      7029h             ;addr of watchdog cntrl reg

                .text
start:
LDP             #WDCR>>7                  ;set data page
SPLK            #11101000b, WDCR          ;disable the watchdog

LAR             AR1,#A                    ;pointer to values in RAM (.bss)
MAR             *,AR1                     ;AR1 active auxiliary register
RPT             #length-1                ;number of values to transfer
BLPD            #table,*+                 ;block move copy from table (.data)to(.bss)

LDP             #A                        ;load data page pointer for direct addressing
SETC            SXM                       ;specify 2's complement
CLRC            OVM                       ;allow intermediate overflows
SPM             1                         ;Q31 mode accumulator

LAR             AR2,#buf_addr              ;pointer to buffer address for graph
LAR             AR3,#buf_size              ;buffer control loop counter for graph
MAR             *,AR2                     ;AR2 active auxiliary register

sine:           LACL      #0                ;clear accumulator
                LT        Y2                ;T = Y2
                MPY        B                ;P = B*Y2
                LTD        Y1                ;ACC = B*Y2, T = Y1, Y1-->Y2
                MPY        A                ;P = A*Y1/2
                APAC        ;ACC = A*Y1/2 + B*Y2
                APAC        ;ACC = A*Y1 + B*Y2 (double APAC)
                SACH        Y1              ;Y0-->Y1

SACH            *,AR3                      ;buffer samples for graph
BANZ            sine,AR2                   ;branch test for number of graph samples
LAR             AR2,#buf_addr              ;reset buffer addr pointer for graph
LAR             AR3,#buf_size              ;reset buff cntrl loop cntr for graph

B               sine                      ;loop and re-calculate next sine value

```

This page is left intentionally blank.

Logical Operations

Introduction

In addition to performing fast arithmetic, the TMS320C240x is equally well suited for logical operations. In this module, we will discuss the various logical operations available in the Arithmetic Logic Unit (ALU). The ALU may be used to perform any logical operation such as AND, OR, XOR, SHIFT, ROTATE, BIT TEST and COMPARE.

Learning Objectives

Learning Objective

- ◆ Implement ALU logical operations

Module Topics

Logical Operations 8-1

Module Topics..... 8-2

Logical Operations 8-3

Boolean Operations 8-4

Shift and Rotate Instructions..... 8-5

CMPL/NEG/ABS Instructions..... 8-6

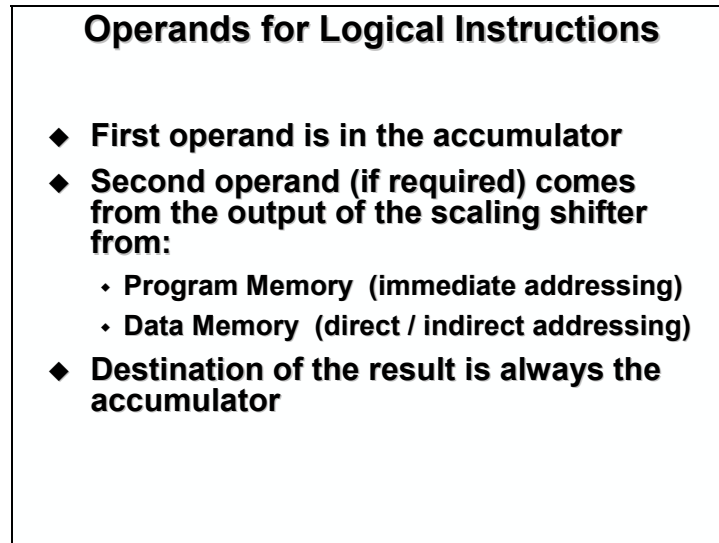
Bit Extraction 8-7

Pointer Comparison..... 8-9

Review.....8-10

Logical Operations

Before we discuss the logical operations available in the ALU, it is useful to consider the location of the source and destination operands. This information is common to all the logical operations and may be summarized as indicated in the following figure.



In other words, the ALU operates on the contents of the accumulator, and puts the result back in the accumulator. The second operand, if required, may come from either program memory (immediate addressing) or data memory (direct or indirect addressing) and may be shifted by the prescaler along the way.

Boolean Operations

The most popular Boolean operators are AND, OR, and EXCLUSIVE OR. As such, the C240x supports numerous ways to perform using each of these. For the accumulator, mask values in memory may be accessed via direct or indirect addressing, or expressed as constants (with an optional shift field). Examples of each of these are presented in the following figure.

| Boolean Operations | | |
|--------------------|--------------|----------------|
| Direct | Indirect | Immediate |
| AND x | AND * | AND #12 |
| OR y | OR *+ | OR #7035 |
| XOR z | XOR *0-, AR2 | XOR #0FFFFh, 5 |

= constant = 2 cycles

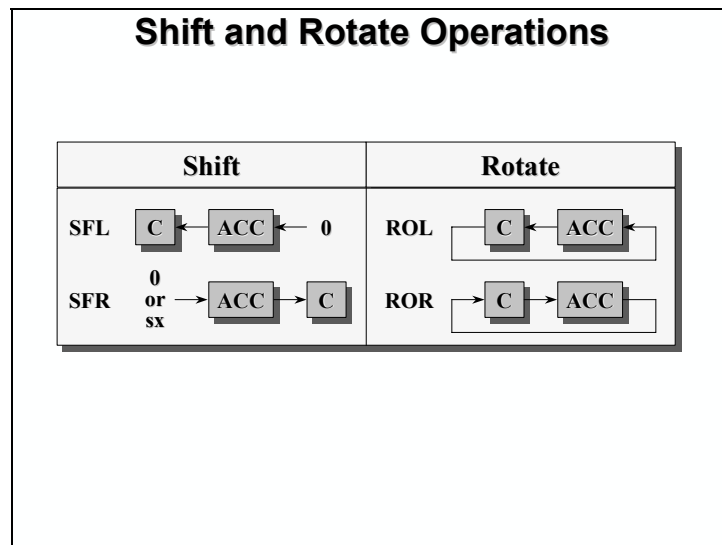
Shift and Rotate Instructions

The ALU supports 32-bit shifts and rotates. For 32-bit operations, the ALU operates on the accumulator.

Single-Bit Shift and Rotate Instructions

The accumulator may be shifted a single bit — either right with the SFR instruction or left with the SFL instruction. In either case, the bit shifted out of the accumulator is shifted into the carry bit. In the case of a left shift, the LSB is zero filled, and the instruction is not affected by SXM. In the case of a right shift, the MSB is either zero filled or sign extended, depending on SXM.

The accumulator may be rotated through the carry bit either right or left with the ROR and ROL instructions. The ROR instruction causes the 32 bits in the accumulator to be shifted one bit right, and the carry bit becomes the MSB, and the LSB becomes the carry bit. The ROL instruction has exactly the opposite effect.



CMPL/NEG/ABS Instructions

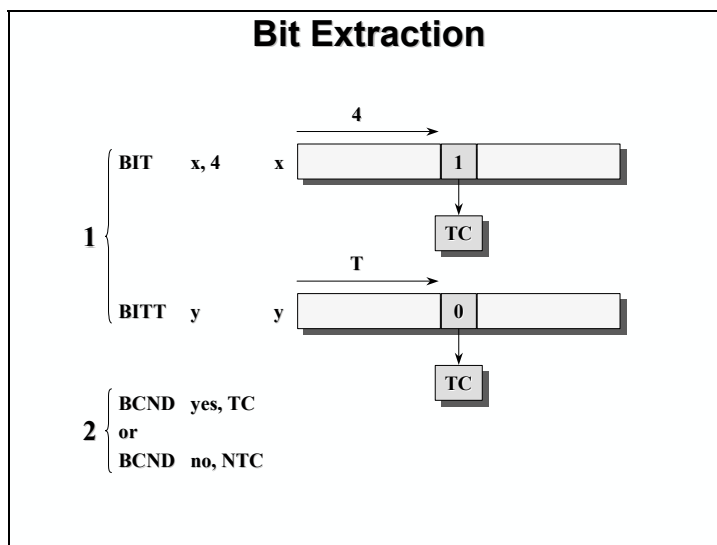
The CMPL instruction takes the one's complement of the accumulator. NEG, on the other hand, takes the two's complement. ABS changes the accumulator to its absolute value.

CMPL / NEG / ABS Operations

| | |
|-------------|------------------------------|
| CMPL | 1's complement of ACC |
| NEG | 2's complement of ACC |
| ABS | absolute value of ACC |

Bit Extraction

The C240x has two special instructions to test the status of a single bit in memory: BIT and BITT. These instructions copy the specified bit (0 - 15) into the TC bit in ST1. Subsequently, it is possible to branch on the contents of the TC bit by using the BCND instruction. The tested bit is specified by a 4-bit code which may be contained directly in the BIT instruction, or specified indirectly via the T register.



Bit Testing – Results placed in Test Control (TC) bit of Status Register ST1. The operation consists of two parts; (1) test the bit, and (2) branch based on the value.

```

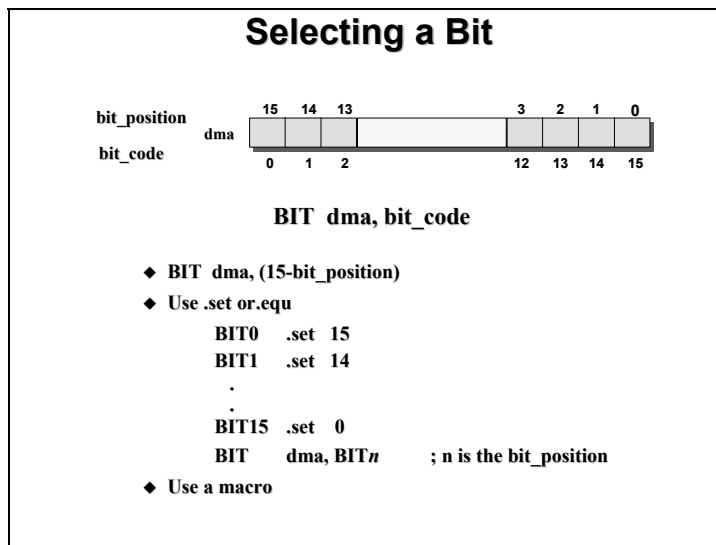
BIT    <dma>, <bit code>    ;copy bit position <bit> of location <dma> to TC

BITT   <dma>                  ;copy bit indicated by the 4 LSB of T register of
                               ;location <dma> to TC

BCND   <pma>, TC              ;branch if TC = 1

BCND   <pma>, NTC             ;branch if TC = 0
  
```

Selecting a Bit



Pointer Comparison

Earlier we described two of the AR0's three functions. The third AR0 function allows AR0 to be compared to the current AR, sometimes referred to as AR(ARP). The CMPR instruction is used with a 2-bit operand that specifies the test condition. The results of the test are placed in the TC (test control) bit, which may be used in a subsequent conditional branch instruction. CMPR is typically used if you desire to terminate the loop with a value other than zero.

| Pointer Comparison | |
|----------------------------|----------------|
| CMPR n | |
| n value | AR(ARP) vs AR0 |
| 0 | = |
| 1 | < |
| 2 | > |
| 3 | ≠ |
| if COMPARE is TRUE, TC = 1 | |

```
CMPR    <cm>                ;compare selected AR with AR0
```

If cm = 00, test if (current AR) = (AR0)

If cm = 01, test if (current AR) < (AR0)

If cm = 10, test if (current AR) > (AR0)

If cm = 11, test if (current AR) ≠ (AR0)

IF TRUE, TC = 1

IF FALSE, TC = 0

```
BCND    <pma>, TC           ;branch if TC = 1
BCND    <pma>, NTC          ;branch if TC = 0
```

Review

New Instructions

| Logical | |
|---------|--------------|
| AND | AND #, shift |
| OR | OR #, shift |
| XOR | XOR #, shift |

| Shift and Rotate | |
|------------------|-----|
| SFL | ROL |
| SFR | ROR |

| Miscellaneous | | |
|---------------|------|-----|
| CMPR | BIT | ABS |
| CMPL | BITT | NEG |

Review

1. What do the following instructions do?
AND *+
BIT x,1
2. How do you use the BIT instruction?
3. List three ALU logical operation instructions.
4. Where do the first and second ALU logical operands come from?

System Initialization

Introduction

This module discusses the operation of the PLL-based clock module and watchdog timer. Also, various low power modes, wait-state generation, and digital I/O ports will be covered.

Learning Objectives

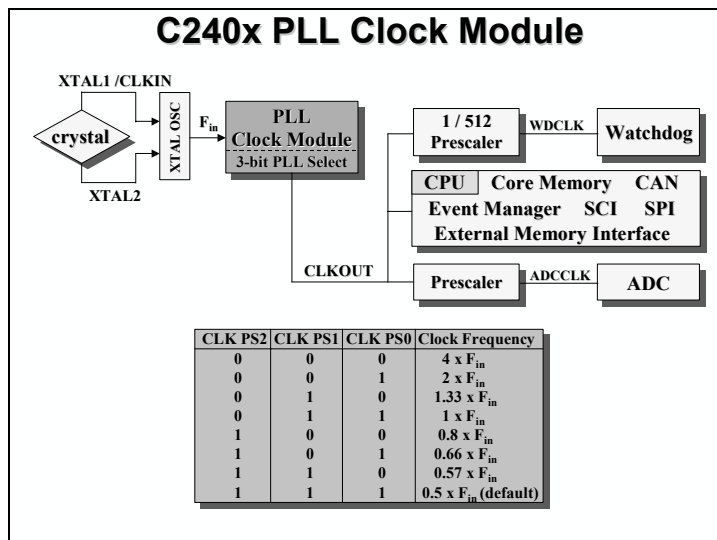
Learning Objectives

- ◆ Describe the PLL Clock Module
- ◆ Configure the Watchdog Timer
- ◆ Explain Low Power Modes
- ◆ Explain Wait-State Generation
- ◆ Describe Digital I/O Ports

Module Topics

| | |
|--|-------------|
| System Initialization..... | 9-1 |
| <i>Module Topics.....</i> | <i>9-2</i> |
| <i>Clock Generation.....</i> | <i>9-3</i> |
| <i>Watchdog Timer.....</i> | <i>9-4</i> |
| <i>System Control and Status Register.....</i> | <i>9-8</i> |
| <i>Power Considerations.....</i> | <i>9-9</i> |
| <i>Wait States.....</i> | <i>9-11</i> |
| Using READY..... | 9-11 |
| Using Software Wait States..... | 9-12 |
| Exercise..... | 9-12 |
| <i>Digital I/O Ports.....</i> | <i>9-13</i> |
| <i>Lab 9: System Initialization.....</i> | <i>9-14</i> |
| <i>Review.....</i> | <i>9-18</i> |
| Solutions..... | 9-18 |

Clock Generation



The PLL clock module provides all the necessary clocking signals for C240x devices. The PLL has a 3-bit ratio control to select different CPU clock rates. Two 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 oscillator to be bypassed, and the device clocks are generated from an external clock source input on the XTAL1/CLKIN pin.

Watchdog Timer

Watchdog Timer

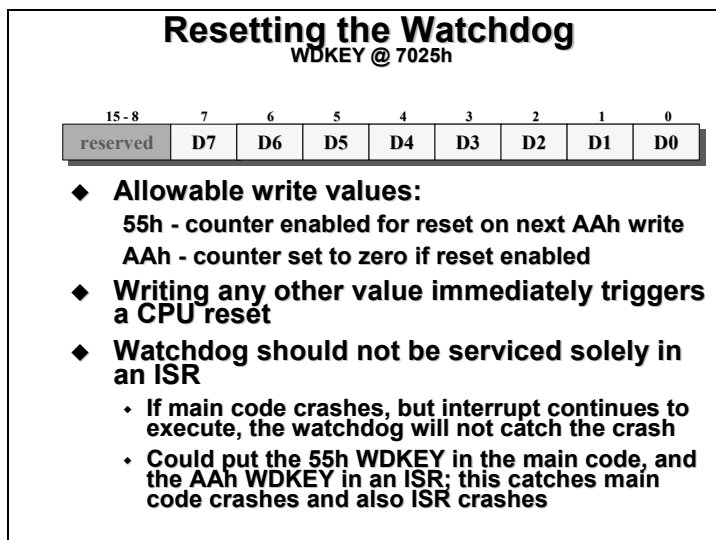
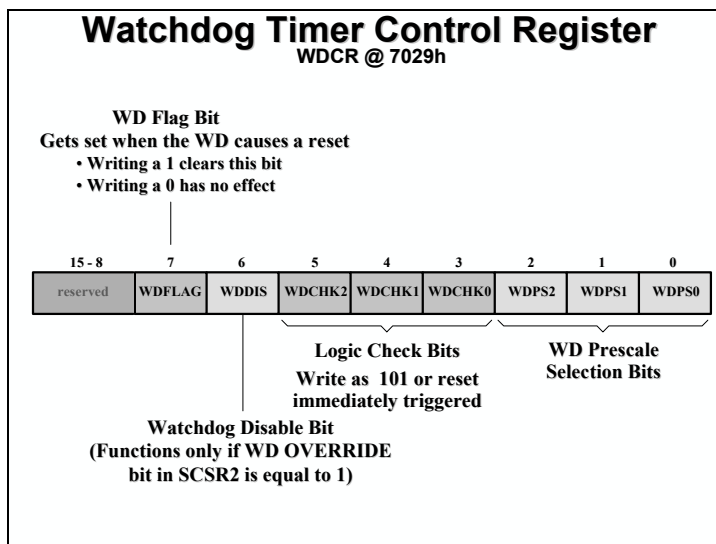
- ◆ **Resets the C240x if the CPU crashes**
 - Watchdog counter runs independent of CPU
 - If counter overflows, reset is triggered
 - CPU must write correct data key sequence to reset the counter before overflow
- ◆ **Watchdog must be serviced (or disabled) within ~3.28ms after reset (40 MHz device)**
- ◆ **This translates into 131,072 instructions!**

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 3.28 ms after any reset before a watchdog initiated reset will occur. For a 25 ns CPU clock, this translates into 131,072 instruction 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.



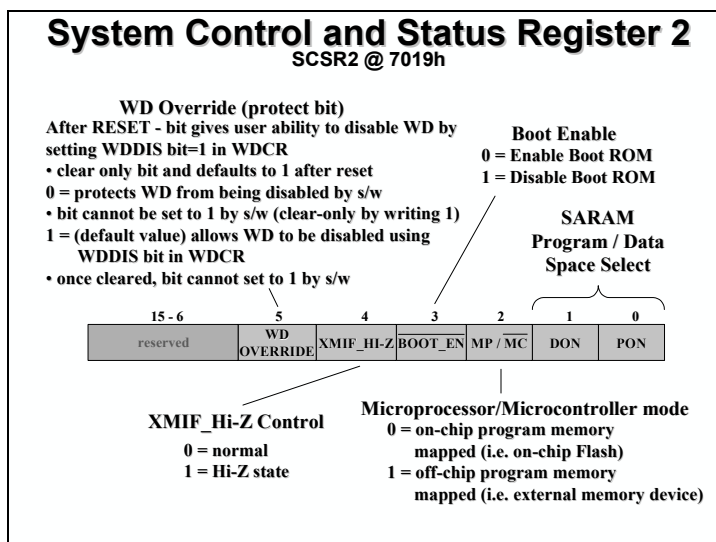
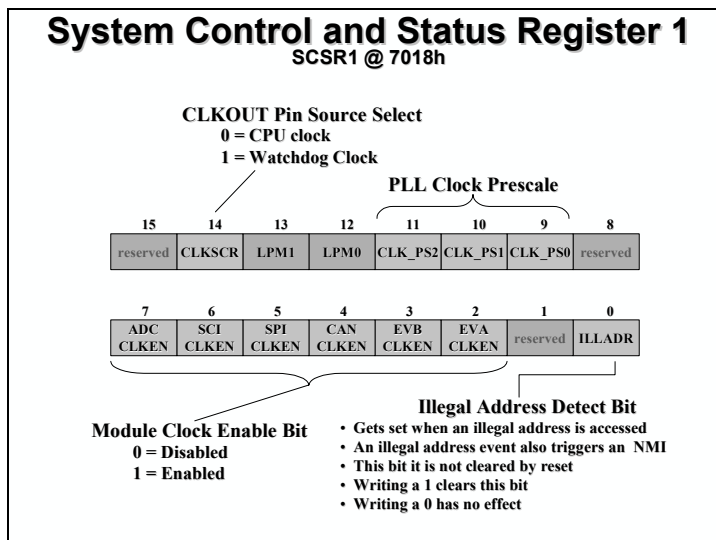
- ◆ **WDPS set to 000 after any CPU reset**
- ◆ **Watchdog starts counting immediately after reset is released**



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 | CPU reset triggered due to improper write value |

System Control and Status Register



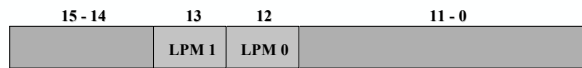
Power Considerations

Low Power Modes

| Low Power Mode | CPU Logic Clock | Peripheral / Interrupt Logic Clock | Watchdog Clock | PLL / OSC | Typical Power |
|----------------|-----------------|------------------------------------|----------------|-----------|---------------|
| Normal Run | on | on | on | on | ~ 90 mA |
| IDLE1 | off | on | on | on | ~ 40 mA |
| IDLE2 | off | off | on | on | ~ 30 mA |
| HALT | off | off | off | off | ~ 10 μ A |

Low Power Mode Entering

SCSR1 @ 7018h



Low Power Mode Selection

00 = Idle 1

01 = Idle 2

1x = Halt

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

Low Power Mode Exit

| Exit Interrupt Low Power Mode | RESET | External or Wake up Interrupts | Peripheral Interrupts |
|--|-------|---|--------------------------|
| Idle 1 | yes | yes | yes |
| Idle 2 | yes | yes | no |
| Halt | yes | no | no |

Note: External include XINTx, PDPINT and NMI

Wake up includes CAN, SPI, SCI and WD

Wait States

The C240x is valued as a high-speed signal processor. However, with high speed often comes the cost of fast, expensive memory devices. In some systems, this can be cost-prohibitive. The C240x can be easily integrated into systems that mix slower and faster memory devices to achieve the optimum cost and performance. This is done via the READY signal or software wait states.

Using READY

The C240x is capable of interfacing to both fast and slow memory in a system by sensing the READY line. Fast memory accesses will assert READY immediately, allowing single-cycle operations, while slow memory accesses will return READY later, permitting sufficient time for the slower device to respond. READY is sampled on the rising edge of CLKOUT1. When READY is high, the current access will be completed. When READY is low, the current access will be extended by one CLKOUT1 cycle.

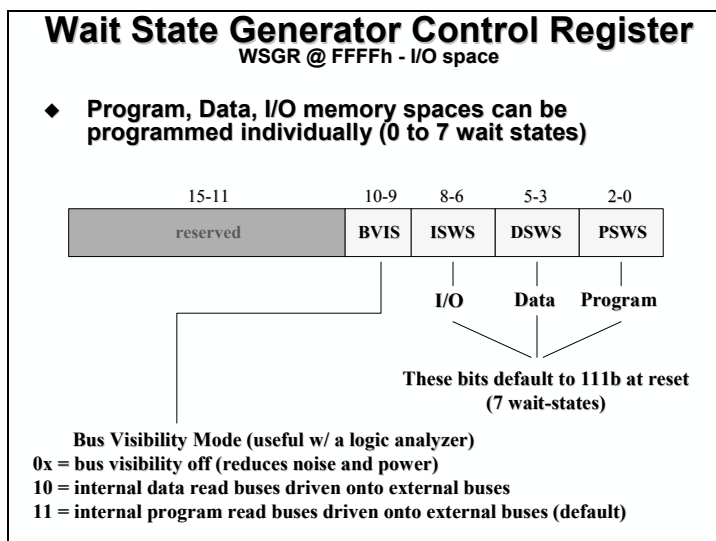
READY timing is also specified relative to address, $\overline{\text{RD}}$, and $\overline{\text{WE}}$. If READY setup and hold times are met for one of these signals—CLKOUT1, $\overline{\text{RD}}$, $\overline{\text{WE}}$ or address lines—then all setup and hold times will be met. Also, for correct device operation, READY must not change after the specified setup time or before the specified hold time.

The READY input pin can be used to extend external read and write cycles. When READY is low, the current access will be extended (the address bus and R/ $\overline{\text{W}}$ will remain stable and $\overline{\text{STRB}}$ will remain active low) until READY goes high. After READY goes active high, the current external interface access completes.

READY is sampled on the rising edge of CLKOUT1. After READY is sampled active high during a read access, the C240x reads data the next time CLKOUT1 falls. After READY is sampled active high during a write access, the C240x finishes writing valid data within a cycle. READY timing is given relative to $\overline{\text{RD}}$, $\overline{\text{WE}}$, CLKOUT1, and address. For a given read or write, if the READY setup and hold times are met relative to one of these signals, the READY timings relative to the other signals will be met also. However, at a minimum, the set-up and hold times for READY relative to one of these signals must be met. If not, proper device operation is not guaranteed.

Using Software Wait States

The software wait-state generator can be used to select 0-7 wait states for program, data, and I/O memory without the use of external hardware on the READY line. Control is programmable via the WSGR register located in I/O space at address FFFFh. **Please note that at Reset, all memory interfaces are set to seven wait states!**



Exercise

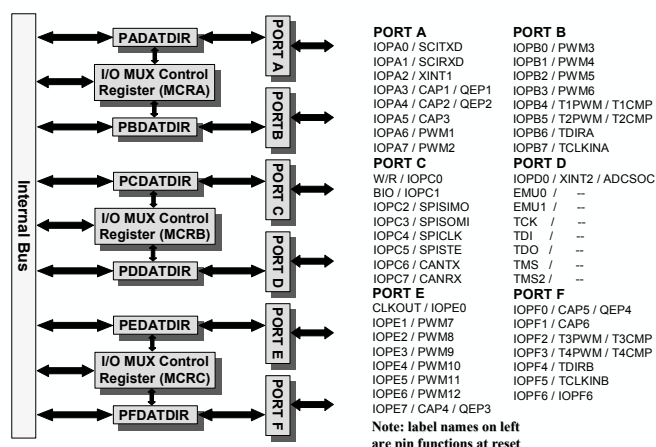
WSGR Exercise

Write the code to set up the following wait states:

- four wait state for program space
- one wait state for data space
- three wait state for I/O space
- clear the BVIS bit (zero)

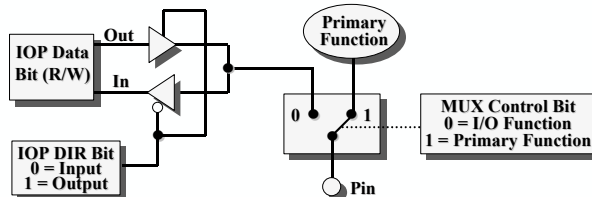
Digital I/O Ports

C240x Digital I/O Pin & Register Structure



Digital I/O Port Registers

| Address | Register | Name |
|---------|----------|--|
| 7090h | MCRA | I/O Mux Control Register A |
| 7092h | MCRB | I/O Mux Control Register B |
| 7094 | MCRC | I/O Mux Control Register C |
| 7098h | PADATDIR | I/O Port A Data and Direction Register |
| 709Ah | PBDATDIR | I/O Port B Data and Direction Register |
| 709Ch | PCDATDIR | I/O Port C Data and Direction Register |
| 709Eh | PDDATDIR | I/O Port D Data and Direction Register |
| 7095h | PEDATDIR | I/O Port E Data and Direction Register |
| 7096h | PFDATDIR | I/O Port F Data and Direction Register |



Lab 9: System Initialization

➤ Objective

The objective of this lab is to perform the processor system initialization by applying the techniques discussed in Module 9. This initialization process will be used again in the Module 11 analog-to-digital converter lab, and the Module 12 event manager lab. The system initialization for this lab will consist of the following:

- Disable the watchdog – clear WD flag, disable watchdog, WDCLK divider = 1
- Setup the clock module – CLKOUT = CPUCLK, IDLE1 low-power, clear ILLADR bit, PLL = x4 mode, enable all module clocks
- Setup control register – DO NOT clear WD OVERRIDE bit, select microprocessor mode, disable Boot ROM, SARAM mapped to Program and Data spaces, XMIF normal mode
- Set wait states for external memory – bus visibility off, 1 w/s for I/O, 0 w/s for Data/Program
- Setup shared I/O pins – set all I/O pins to their reset default function in MCRA, MCRB and MCRC (e.g. a "0" setting, and a "1" setting for emulation functions. Reference the System Peripherals User's Guide)

This lab will also make use of the F2407.h header file to take care of the register definitions and addresses. Please review this file, and make use of it in the future, as needed.

➤ Procedure

Create Make File

1. **NOTE:** LAB9.ASM, VECS_9.ASM and LAB9.CMD files have been provided as a starting point for the lab and need to be completed. *DO NOT copy files from a previous lab.*
2. Create a new project called LAB9.MAK and add LAB9.ASM, VECS_9.ASM and LAB9.CMD to it. Check your file list to make sure all the files are there. (F2407.h will be added automatically during the Build). Be sure to setup the Build Options by clicking: Project → Options on the menu bar. Select the Assembler tab. In the middle of the screen check "Enable Source Level Debugging". Next, select the Linker tab. In the middle of the screen select "No Autoinitialization". Create a map file named LAB9.MAP. Select OK to save the Build Options.

Setup System Initialization

3. Edit LAB9.ASM and modify it to implement the system initialization as described above in the objective for the lab. **NOTE:** Do not edit the "Main loop" section. This section will be used to test the watchdog operation. Save your work.
4. Open and inspect the VECS_9.ASM file. Note that it contains a complete interrupt vector table for the core. Details of the interrupt vector table will be discussed in the Interrupt Module 10. Please review this file, and make use of it in the future, as needed.

Build and Load

5. Click the "Rebuild All" 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 ("Option" menu, click on "Program Load...") load the output file into the target. Click: File → Load Program...

Then reset the DSP by clicking on: Debug → Reset DSP

If you would like to debug using both source and assembly, right click on the VECS_9.ASM window and select Mixed Mode.

Testing Lab9

7. After loading, the VECS_9.ASM window should open with the yellow highlight on "B start". Place the cursor on the "B start" line and set a breakpoint by right clicking the mouse key and select Toggle breakpoint. Notice that line is highlighted indicating that the breakpoint has been set.
8. Single-step your code into the "Main loop" section.
9. 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".
10. Next, in the LAB9.ASM window "comment out" (using ";" or "*" in column one) the code used to disable the watchdog (WDCR). Save the file and click the "Build" button. (Load the output file onto the target if "Load program after build" option was not selected). Then reset the DSP by clicking on: Debug → Reset DSP
11. The VECS_9.ASM window should open with the yellow and pink highlight on "B start" line indicating the breakpoint is still set. Single-step your code into the "Main loop" section. The "commented out" code for the watchdog should be skipped.
12. Run your code by using the <F5> key, or using the Run button on the vertical toolbar, or using Debug → Run on the menu bar. Where did your code stop? Are the results as expected?

End of Exercise

Optional Exercise - Using Hardware Breakpoints

Up to now, the breakpoints you have been setting are called *Software Breakpoints*. When you right click on a line of code and select `Toggle Breakpoint`, what actually happens is that the debugger reads the op-code at the address you clicked on, stores it off someplace nice and safe in the PC, and then replaces the op-code in the DSP with a special undocumented instruction called `ESTOP`. When the DSP executes this instruction, it halts execution and returns control back to the debugger/emulator. Software breakpoints work well when debugging in RAM. But, do you see any problems when you want to debug code that resides in non-volatile memory, such as ROM or FLASH?

When debugging code in non-volatile memory, the debugger cannot insert the `ESTOP` instruction. Instead, we need to use a *Hardware Breakpoint*. The C240x DSP contains special on-chip logic called the *Analysis Block*. You can use the analysis block to monitor the program address bus for a particular value (e.g. the address of the instruction you wish to set the breakpoint at). When the program address bus matches the value you setup, the DSP will halt and control will be returned to the debugger/emulator. In this section of the lab, you will learn how to use a hardware breakpoint in Code Composer.

1. Close Code Composer, and turn off your EVM. Move jumper 6 (JP6) to the 2-3 position. This jumper controls the MP/MC' pin voltage. The 2-3 position maps the on-chip FLASH into the memory map.

IMPORTANT: At the end of the optional part of this lab, step 10 will have you return this jumper back to position 1-2. *If you do not fully complete this optional part of the lab, you must still remember to return the jumper back to the 1-2 position!*

2. The FLASH memory has already been programmed with a small program that sequences the LED bank on the EVM (if you want to see this program, you can open the file `LED_ASM1.ASM` and examine it). Turn on the EVM, and confirm that the program is functioning by observing the LED bank.
3. Start Code Composer. Now RUN the program either by hitting the <F5> key or clicking `Debug` → `Run`. A dialog box will appear saying "Warning! No valid program is currently loaded memory. Do you still want to run?" The reason this box appears is because Code Composer doesn't know you have a valid program in FLASH. Click the "Don't show this dialog box again" and the click the "Yes" button to close the box. You should now see the LED bank sequencing again.
4. Halt the program. In all probability, the disassembly window shows the program halted at program address `0x008B`, which is a branch to itself. This is the main program loop where we loop endlessly waiting for an interrupt to occur. If it is not at this address, you have by chance halted the DSP when it was in the middle of the timer interrupt service routine. If this occurs, Run and Halt the DSP again and it should stop at `0x008B`.
5. In order to do symbolic debug when code is located in FLASH, we need to load the debug symbols into Code Composer. To do this, click `File` → `Load Symbol`, and type in `LED_ASM1.OUT` in the filename box that appears. Notice the change in the disassembly window.

6. We want to set a breakpoint at the beginning of the interrupt service routine that updates the LED's. The code label is `g_isr3`, and can be found at address `0x008D`. First, let's attempt to set a software breakpoint. In the disassembly window, left-click the mouse once at address `0x008D` (the `SST` instruction) to move the cursor to that location, and then right-click the mouse and select "Toggle breakpoint" from the menu. Code Composer should give you an error message that says "Can't set breakpoint." Click "Cancel" to close the error message box. This error message comes from Code Composer not being able to replace the `SST` instruction at address `0x008D` with the `ESTOP` instruction. This shows us that software breakpoints cannot be used in non-volatile (i.e. FLASH) memory.
7. We will now set a hardware breakpoint at this address. Open the breakpoint window, click: Debug → Breakpoints. Select the Breakpoints tab, if it is not already selected.

In the "Breakpoint type" box, select `H/W Break`.

In the "Location" box, type in the address you want to break at: `0x008D` (or you can type the symbol "`g_isr3`").

Click "Add" to add the breakpoint to the breakpoint list. You will see the `HW` breakpoint at `0x008D` added to the breakpoint list in the window. It should have a check mark to its left, which indicates that it is enabled. You might also see an unchecked "`g_isr3`" entry in this window, which is the remnants of our unsuccessful prior attempt to set a software breakpoint at `g_isr3`. Click "OK" to close this window. The DSP will now break when the instruction at that location is *fetched* over the program bus.
8. In the disassembly window you should see `SST` instruction at address `0x008D` highlighted in purple. This indicates that a breakpoint is active at this instruction.
9. Run the program. The program should halt at the `SST` breakpoint. You can step off the breakpoint by hitting `<F8>`, and then do another RUN to convince yourself that the breakpoint is working. You should also see the LED bank advance by one each time you hit RUN.
10. Quit Code Composer, and turn off your EVM. Move EVM jumper 6 (JP6) back to the 1-2 position. **It is important to do this or the remaining labs will not function properly.**

You have now learned how to set a hardware breakpoint in the FLASH using Code Composer debugger. Note that you cannot have more than one hardware breakpoint active at the same time (unlike software breakpoints, where you can have any number active all at once). However, this single hardware breakpoint is very valuable when debugging in ROM or FLASH. One hardware breakpoint is infinitely better than none at all!

Review

Review

- ◆ What are the input clock frequency multiplication factors?
- ◆ How is the Watchdog Timer disabled?
- ◆ How is the Watchdog service interval specified?
- ◆ What power-down modes are available?
- ◆ How many wait states can the software WSGR assign?

Solutions

WSGR Exercise Solution

```
WSGR .set    0ffffh
      .bss    temp,1
      .text
; choose only one of the following LACC statements
      LACC    #0000000000000100b    ; part a only
      LACC    #0000000000001000b    ; part b only
      LACC    #0000000011000000b    ; part c only
      LACC    #0000000000000000b    ; part d only
      LACC    #0000000011001100b    ; part a, b, c, d combined
; write value to WSGR
      LDP     #temp
      SACL    temp
      OUT     temp, WSGR              ; store wait state value to WSGR
```

```

; SOLUTION FILE FOR LAB9.ASM

        .def      start

        .include  f2407.h           ;address definitions

        .bss      temp,1           ;general purpose variable

        .text

start:

;~~~~~
;Disable the watchdog
;~~~~~
        LDP        #DP_PF1         ;set data page

        SPLK       #11101000b, WDCR
* bit 7           1:      clear WD flag
* bit 6           1:      disable the dog
* bit 5-3         101:    must be written as 101
* bit 2-0         000:    WDCLK divider = 1

;~~~~~
;Setup the system control registers
;~~~~~
        LDP        #DP_PF1         ;set data page

        SPLK       #0000000011111101b, SCSR1
;
;          |||||
;          FEDCBA9876543210
* bit 15         0:      reserved
* bit 14         0:      CLKOUT = CPUCLK
* bit 13-12      00:     IDLE1 selected for low-power mode
* bit 11-9       000:    PLL x4 mode
* bit 8          0:      reserved
* bit 7          1:      1 = enable ADC module clock
* bit 6          1:      1 = enable SCI module clock
* bit 5          1:      1 = enable SPI module clock
* bit 4          1:      1 = enable CAN module clock
* bit 3          1:      1 = enable EVB module clock
* bit 2          1:      1 = enable EVA module clock
* bit 1          0:      reserved
* bit 0          1:      clear the ILLADR bit

        SPLK       #0000000000001111b, SCSR2
;
;          |||||
;          FEDCBA9876543210
* bit 15-6       0's:    reserved
* bit 5          0:      DO NOT clear the WD OVERRIDE bit
* bit 4          0:      XMIF_HI-Z, 0=normal mode, 1=Hi-Z'd
* bit 3          1:      1 = disable the BOOT ROM
* bit 2          1:      MP/MC*, 1 = Flash addresses mapped external
* bit 1-0        11:     11 = SARAM mapped to prog and data

;~~~~~
;Set wait states for external memory interface on LF2407 EVM
;~~~~~

```

```

        LDP        #temp        ;set data page

        SPLK       #0000000001000000b, temp
;          |||||
;          FEDCBA9876543210
* bit 15-11      0's:    reserved
* bit 10-9       00:     bus visibility off
* bit 8-6        001:    1 wait-state for I/O space
* bit 5-3        000:    0 wait-state for data space
* bit 2-0        000:    0 wait-state for program space

        OUT        temp, WSGR

;~~~~~
;Setup shared I/O pins
;~~~~~
        LDP        #DP_PF2      ;set data page

        SPLK       #0000000000000000b, MCRA
*          |||||
*          FEDCBA9876543210
* bit 15        0:      0=IOPB7,      1=TCLKINA
* bit 14        0:      0=IOPB6,      1=TDIRA
* bit 13        0:      0=IOPB5,      1=T2PWM/T2CMP
* bit 12        0:      0=IOPB4,      1=T1PWM/T1CMP
* bit 11        0:      0=IOPB3,      1=PWM6
* bit 10        0:      0=IOPB2,      1=PWM5
* bit 9         0:      0=IOPB1,      1=PWM4
* bit 8         0:      0=IOPB0,      1=PWM3
* bit 7         0:      0=IOPA7,      1=PWM2
* bit 6         0:      0=IOPA6,      1=PWM1
* bit 5         0:      0=IOPA5,      1=CAP3
* bit 4         0:      0=IOPA4,      1=CAP2/QEP2
* bit 3         0:      0=IOPA3,      1=CAP1/QEP1
* bit 2         0:      0=IOPA2,      1=XINT1
* bit 1         0:      0=IOPA1,      1=SCIRXD
* bit 0         0:      0=IOPA0,      1=SCITXD

        SPLK       #1111111000000000b, MCRB
*          |||||
*          FEDCBA9876543210
* bit 15        1:      0=reserved,   1=TMS2 (always write as 1)
* bit 14        1:      0=reserved,   1=TMS  (always write as 1)
* bit 13        1:      0=reserved,   1=TD0  (always write as 1)
* bit 12        1:      0=reserved,   1=TDI  (always write as 1)
* bit 11        1:      0=reserved,   1=TCK  (always write as 1)
* bit 10        1:      0=reserved,   1=EMU1 (always write as 1)
* bit 9         1:      0=reserved,   1=EMU0 (always write as 1)
* bit 8         0:      0=IOPD0,      1=XINT2/ADCSOC
* bit 7         0:      0=IOPC7,      1=CANRX
* bit 6         0:      0=IOPC6,      1=CANTX
* bit 5         0:      0=IOPC5,      1=SPISTE
* bit 4         0:      0=IOPC4,      1=SPICLK
* bit 3         0:      0=IOPC3,      1=SPISOMI
* bit 2         0:      0=IOPC2,      1=SPISIMO
* bit 1         0:      0=IOPC1,      1=BIO*
* bit 0         0:      0=IOPC0,      1=W/R*

```



```

        SPLK      #0000000000000000b,MCRC
*          |||||
*          FEDCBA9876543210
* bit 15      0:      reserved
* bit 14      0:      0=IOPF6,      1=IOPF6
* bit 13      0:      0=IOPF5,      1=TCLKINB
* bit 12      0:      0=IOPF4,      1=TDIRB
* bit 11      0:      0=IOPF3,      1=T4PWM/T4CMP
* bit 10      0:      0=IOPF2,      1=T3PWM/T3CMP
* bit 9       0:      0=IOPF1,      1=CAP6
* bit 8       0:      0=IOPF0,      1=CAP5/QEP4
* bit 7       0:      0=IOPE7,      1=CAP4/QEP3
* bit 6       0:      0=IOPE6,      1=PWM12
* bit 5       0:      0=IOPE5,      1=PWM11
* bit 4       0:      0=IOPE4,      1=PWM10
* bit 3       0:      0=IOPE3,      1=PWM9
* bit 2       0:      0=IOPE2,      1=PWM8
* bit 1       0:      0=IOPE1,      1=PWM7
* bit 0       0:      0=IOPE0,      1=CLKOUT

;~~~~~
;Main loop
;~~~~~
loop:      NOP
           B          loop          ;branch to loop

```

```

; SOLUTION FILE FOR VECS_9.ASM

        .ref      start

        .sect     "vectors"

;~~~~~
;Interrupt vector table for core
;~~~~~

int1:      B      start           ;00h reset
int2:      B      int1           ;02h INT1
int3:      B      int2           ;04h INT2
int4:      B      int3           ;06h INT3
int5:      B      int4           ;08h INT4
int6:      B      int5           ;0Ah INT5
int7:      B      int6           ;0Ch INT6
int8:      B      int7           ;0Eh reserved
int9:      B      int8           ;10h INT8 user-defined
int10:     B      int9           ;12h INT9 user-defined
int11:     B      int10          ;14h INT10 user defined
int12:     B      int11          ;16h INT11 user defined
int13:     B      int12          ;18h INT12 user defined
int14:     B      int13          ;1Ah INT13 user defined
int15:     B      int14          ;1Ch INT14 user defined
int16:     B      int15          ;1Eh INT15 user defined
int17:     B      int16          ;20h INT16 user defined
int18:     B      int17          ;22h TRAP
int19:     B      int18          ;24h NMI
int20:     B      int19          ;26h reserved
int21:     B      int20          ;28h INT20 user defined
int22:     B      int21          ;2Ah INT21 user defined
int23:     B      int22          ;2Ch INT22 user defined
int24:     B      int23          ;2Eh INT23 user defined
int25:     B      int24          ;30h INT24 user defined
int26:     B      int25          ;32h INT25 user defined
int27:     B      int26          ;34h INT26 user defined
int28:     B      int27          ;36h INT27 user defined
int29:     B      int28          ;38h INT28 user defined
int30:     B      int29          ;3Ah INT29 user defined
int31:     B      int30          ;3Ch INT30 user defined
int31:     B      int31          ;3Eh INT31 user defined

```

Introduction

This module describes the interrupt structure on the C240x. Additionally, the reset process and methods for managing interrupt latency will be explored.

Learning Objectives

Learning Objectives

- ◆ Describe the C240x reset process and post-reset device state
- ◆ List the event sequence during an interrupt
- ◆ Describe the C240x interrupt structure
- ◆ Examine methods of managing interrupt latency
- ◆ Describe the power drive protection interrupt

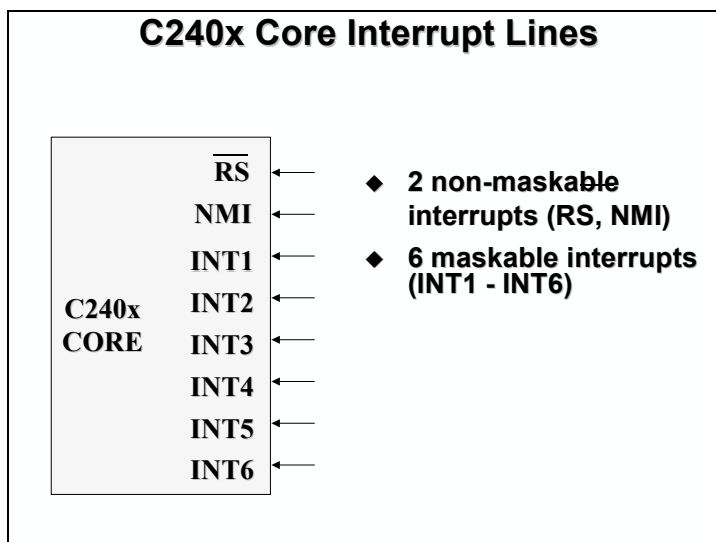
Module Topics

| | |
|---|--------------|
| Interrupts | 10-1 |
| <i>Module Topics.....</i> | <i>10-2</i> |
| <i>Overview</i> | <i>10-3</i> |
| <i>Reset.....</i> | <i>10-4</i> |
| Description of Actions Taken On Reset | 10-4 |
| Reset Initialization Routine | 10-5 |
| <i>Interrupts</i> | <i>10-6</i> |
| Valid Interrupt Signal | 10-7 |
| Interrupt Management - Individual Flags and Masks | 10-8 |
| Interrupt Management - Core Flags and Masks | 10-9 |
| Reading/Writing IFR | 10-9 |
| Interrupt Management - Masks | 10-10 |
| Reading/Writing IMR..... | 10-10 |
| Interrupt Management - Global Switch | 10-11 |
| Interrupt Management — Summary | 10-12 |
| <i>Exercise 10-1</i> | <i>10-13</i> |
| <i>Interrupt Timeline (Hardware).....</i> | <i>10-14</i> |
| Interrupt Hardware Sequence | 10-15 |
| Core Interrupt Vectors | 10-16 |
| <i>Exercise 10-2</i> | <i>10-17</i> |
| <i>Interrupt Service Routine.....</i> | <i>10-18</i> |
| Interrupt Sources | 10-18 |
| Determining the Interrupt Source | 10-19 |
| <i>Interrupt Latency</i> | <i>10-20</i> |
| <i>Context Save and Restore</i> | <i>10-23</i> |
| The Hardware Stack | 10-23 |
| Extending the PC Hardware Stack | 10-24 |
| Status Registers | 10-24 |
| Auxiliary Registers | 10-25 |
| Math Registers..... | 10-25 |
| <i>Managing Latency from Contexting.....</i> | <i>10-26</i> |
| <i>Nesting Interrupts</i> | <i>10-28</i> |
| <i>Return from ISR</i> | <i>10-29</i> |
| <i>Non-Maskable Interrupt.....</i> | <i>10-30</i> |
| <i>Power-Drive Protection Interrupt.....</i> | <i>10-31</i> |
| <i>Software Initiated Interrupts.....</i> | <i>10-32</i> |
| Trap Instruction | 10-32 |
| <i>Review.....</i> | <i>10-33</i> |
| <i>Solutions to Exercise 10-1 and 10-2</i> | <i>10-34</i> |

Overview

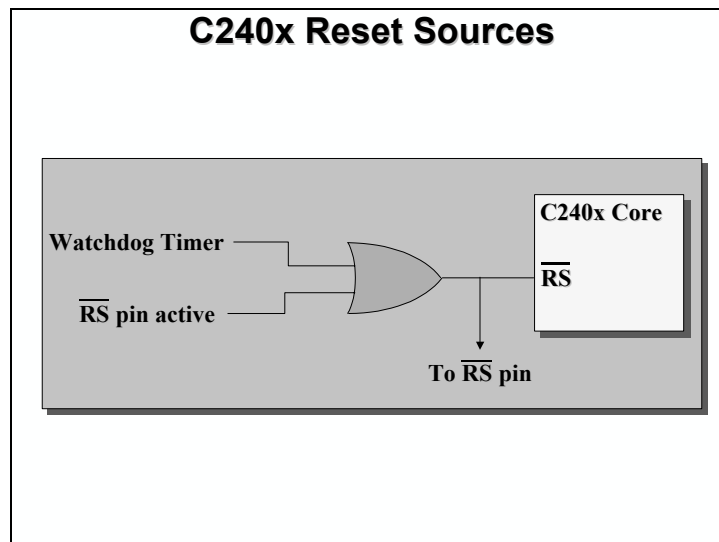
Interrupts provide a means of directing the C240x to suspend its main program in order to respond to a hardware-driven event. This eliminates the need to poll external events via software (unless desired), and improves response time and decrease processor overhead dramatically. Typically, interrupts are generated by devices which need to give or take data from the C240x. Examples of such devices are A/D and D/A converters and other processors. Interrupt may also be used as a signal to inform the C240x when any event of interest has occurred within the system. When the C240x recognizes the interrupt signal, it suspends execution of the main program and begins execution of the code specific to the particular interrupt event. On the C240x, the programmer can dynamically select when interrupts may be taken, and which interrupts will be recognized.

The C240x core of the C240x processor supports six user-maskable interrupts. These interrupts are then fanned out and shared among numerous on-chip peripherals and external pins. Interrupts can be generated by internal or external sources or by software interrupt instructions. A reset function, a non-maskable interrupt, and a power-drive protection interrupt are also supported on all C240x devices.



Reset

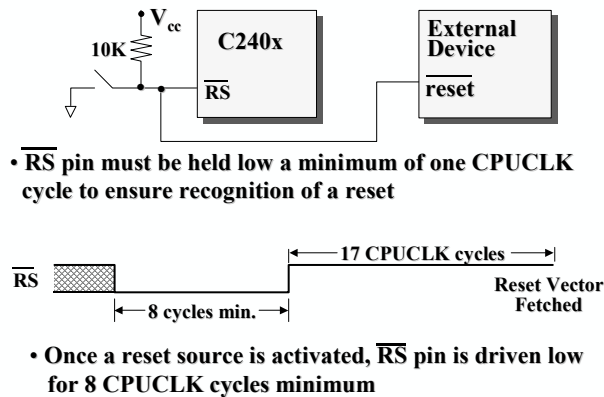
Reset is a non-maskable interrupt that may be initiated by 2 different sources. There is an external reset (\overline{RS}) pin that initiates reset when driven low. Reset may also be asserted by the watchdog timer. Reset is asserted on power-up to allow processing to begin from a known state. Asserting reset when the processor is running causes the C240x to terminate execution and re-establish its initial conditions.



Description of Actions Taken On Reset

1. Any program currently being executed is asynchronously aborted.
2. The Program Counter (PC) begins fetching from program location 0000h.
3. Numerous CPU and peripheral control bits are set in a defined state. See the individual CPU and peripheral register descriptions in the C240x User's Guide and Reference Guide for register initializations.

RS Pin Connections / Timings



Register / Control Bits Initialized at Reset

Status Register 0 & 1 (ST0, ST1)

| | |
|----------|--|
| OV = 0 | Overflow bit cleared |
| INTM = 1 | Disable all maskable interrupts - global |
| CNF = 0 | Block B0 configured as data space |
| SXM = 1 | Sign extension selected |
| XF = 1 | External flag pin set high |
| PM = 0 | Product register shift zero bits |
| C=1 | Carry bit set |

Other Registers

| | |
|--------------|-----------------------------------|
| IFR = 0 | No interrupts pending |
| WSGR = 07FFh | Maximum number of wait states set |

Register / Control bits NOT defined by reset

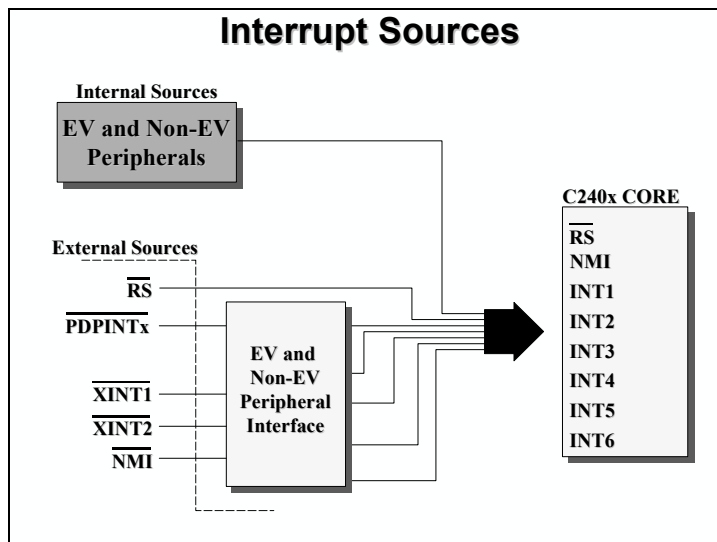
| | |
|-----------|----------------------------|
| ARP | Auxiliary Register Pointer |
| AR0 - AR7 | Auxiliary Registers 0 to 7 |
| DP | Data Page Pointer |
| OVM | Overflow Mode bit |
| ACC | Accumulator |

Note: Any status and control bits not explicitly initialized at reset should be initialized by the application program before they are used.

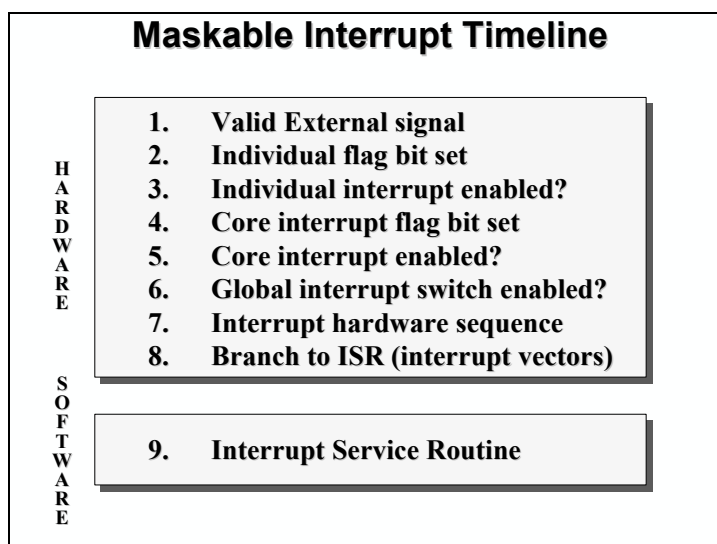
Reset Initialization Routine

Upon reset, the processor is placed into a known state as described previously in this section. It has also been mentioned that many registers are not predefined at reset time. It is left up to the programmer to create an initialization routine to initialize any peripherals and processor operations before executing the main routine.

Interrupts

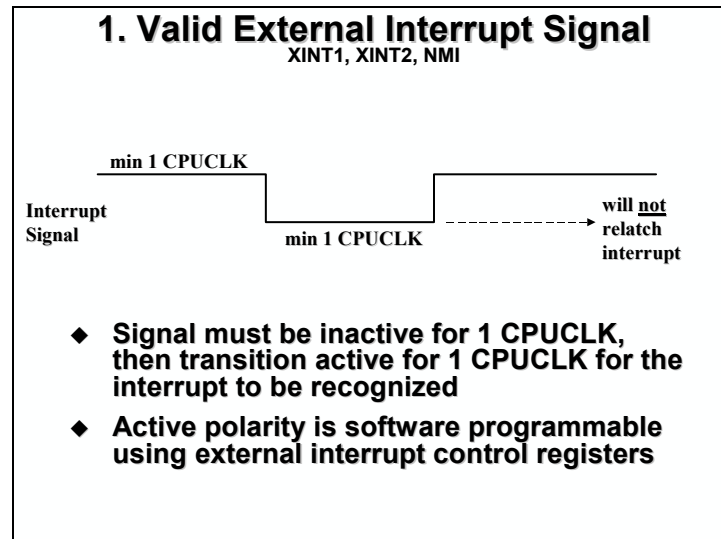


Interrupt processing can be broken down into a series of steps which occur in a serial fashion (that is, if the interrupt is not interrupted!) The figure below shows a conceptual view of how interrupts are processed. The next figure shows a more detailed timeline of events.



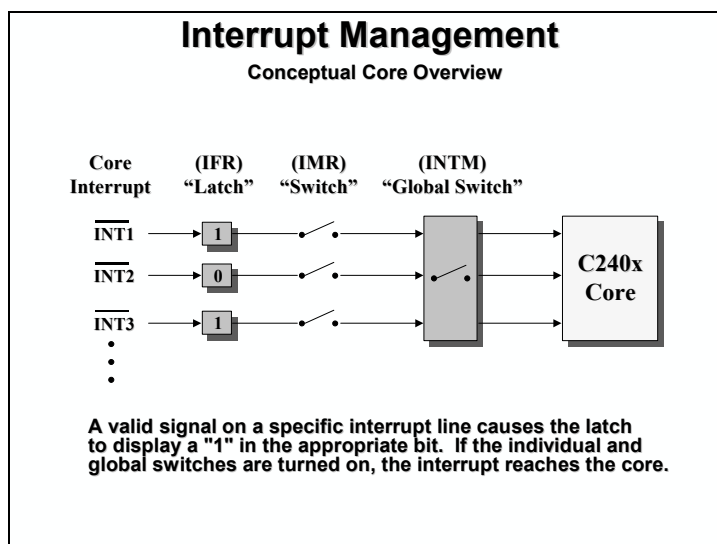
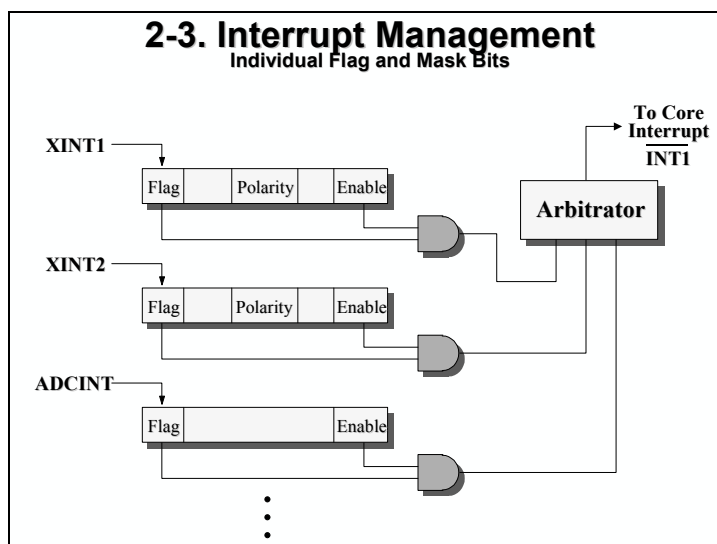
Valid Interrupt Signal

The first step in recognizing an interrupt is receiving a valid signal on an interrupt pin. This signal must meet certain requirements. All external C240x interrupts are falling-edge triggered and must stay active low for at least one cycle. Holding the signal low will not relatch the interrupt.



Interrupt Management - Individual Flags and Masks

Peripheral and externally generated interrupts are arranged in groups and associated with one of the core interrupts, INT1 - INT6. When a valid signal is generated by a peripheral or by an external source, the corresponding individual flag bit is set to 1 in the control register associated with that peripheral or external interrupt[†]. If the individual enable bit is set for that interrupt, an interrupt request will be passed to the arbitration logic, which prioritizes all pending interrupts in its group and then sends the highest priority interrupt request to its associated core interrupt. The individual flag bits are set regardless of the condition of the individual enable bits. The schematic below depicts the signal flow to core INT1.



[†] Peripheral control register information is found in the TMS320C240x Reference Guide under the individual peripheral, while control register addresses for external interrupts (XINT1, XINT2, PDPINT) are listed in the TMS320C240x Data Sheet and described in the TMS320C240x User's Guide.

Interrupt Management - Core Flags and Masks

When an interrupt request is sent to the CPU core by the arbitration logic, the corresponding bit in the Interrupt Flag Register (IFR) is set. Flags are set regardless of the condition of the masks (IMR) or global interrupt switch (INTM). The core flag bits are automatically cleared by the processor after the interrupt is recognized by the CPU.

4. Interrupt Management - Core Flags

IFR @ 0006h

| | | | | | | |
|----------|------|------|------|------|------|------|
| 15 - 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| reserved | INT6 | INT5 | INT4 | INT3 | INT2 | INT1 |

- ◆ IFR indicates when a valid interrupt has occurred
- ◆ "1" is displayed in the corresponding bit

Reading/Writing IFR

To manually clear an IFR flag bit, a logic 1 must be written. The user would only be required to do this if the code was polling for interrupts (processor automatically clears the flag bits when the interrupt is recognized).

Reading/Writing IFR

- ◆ IFR flags are automatically cleared when the interrupt is serviced by the CPU
- ◆ To manually clear a flag, write a 1 to the bit:

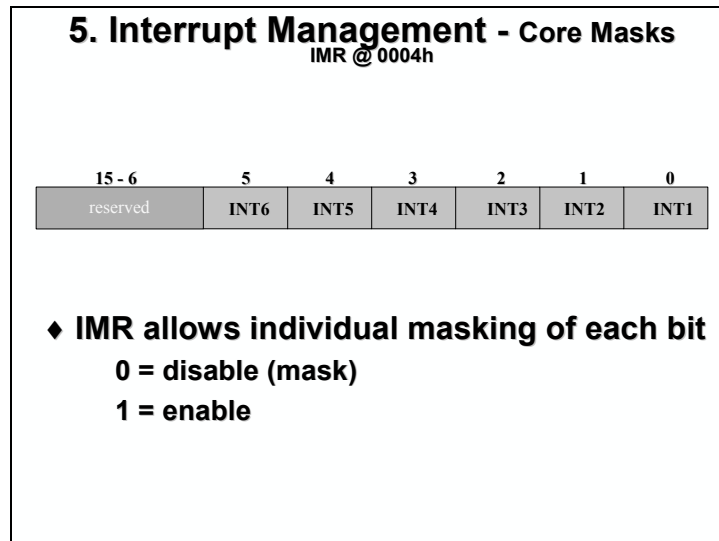
| | | |
|-----|-------|----------|
| IFR | .set | 0006h |
| | .text | |
| | LDP | #0 |
| | LACL | #000001b |
| | SACL | IFR |

Clears $\overline{\text{INT1}}$ Flag

- ◆ Writing 0 to the bit has no effect

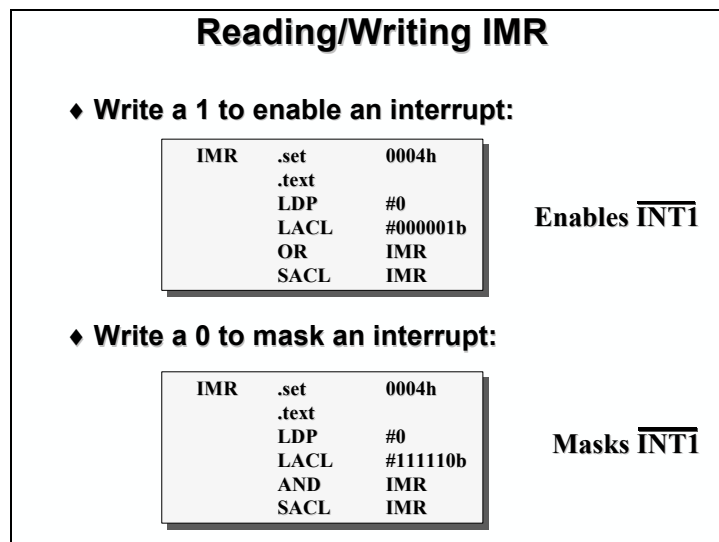
Interrupt Management - Masks

The Interrupt Mask Register (IMR) allows the user to select which core interrupts the C240x should respond to at a given time. A logic 1 written to any mask bit enables the corresponding interrupt. Notice that \overline{RS} and \overline{NMI} cannot be masked.



Reading/Writing IMR

To enable a specific core interrupt, a logic 1 must be written to the corresponding bit in the IMR. Here's one example of enabling and disabling $\overline{INT1}$:



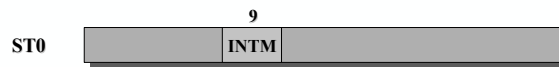
Interrupt Management - Global Switch

The INTM bit (bit 9 in ST0) can be used to globally enable or disable all maskable interrupts. When INTM = 0, all enabled interrupts (as indicated in the IMR register) are allowed. INTM = 1 inhibits these interrupts. Note that the IFR and IMR are not affected by the state of INTM.

The SETC and CLRC instructions are normally used to affect the INTM bit:

- To enable all interrupts need INTM = 0: `CLRC INTM`
- To disable all interrupts need INTM = 1: `SETC INTM`

6. Interrupt Management - Global Switch



♦ INTM bit is global enable/disable of interrupts

0 = enabled `CLRC INTM`

1 = disabled `SETC INTM`

♦ If INTM = 0, all individually enabled interrupts in the IMR register are enabled

Interrupt Management — Summary

The complete interrupt story can now be revealed by observing all four registers related to interrupt operation. For example, if a valid interrupt signal occurred on the XINT1 pin, the following bits would be set:

- Individual flag bit = 1 (for XINT1, this is bit 15 at data address 7070h)

If the following condition were met, an interrupt request would be sent to the arbitration logic:

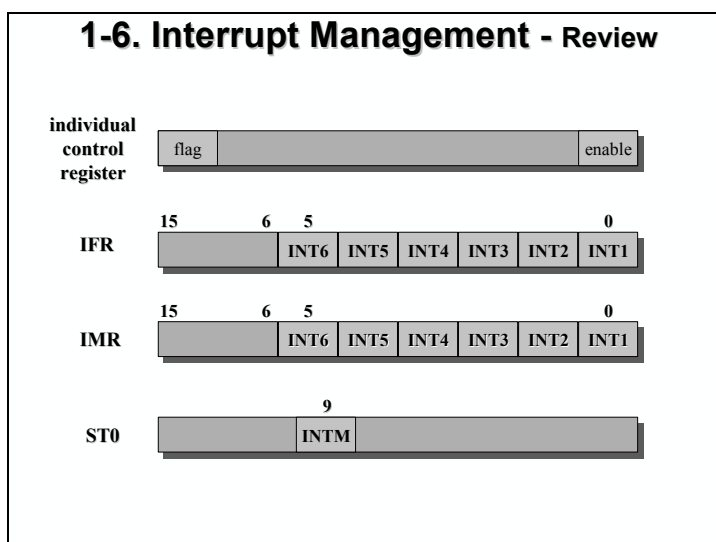
- Individual enable bit = 1 (for XINT1, this is bit 0 at data address 7070h)

In this case, XINT1 is the highest priority interrupt in its arbitration group, so the arbitration logic will always send an interrupt request to the CPU regardless of what other (lower priority) interrupts are pending in its grouping:

- IFR (bit 0) = 1

Finally, if the following conditions were met, the processor would automatically respond to the interrupt:

- IMR (bit 0) = 1
- ST0 (bit 9) = 0



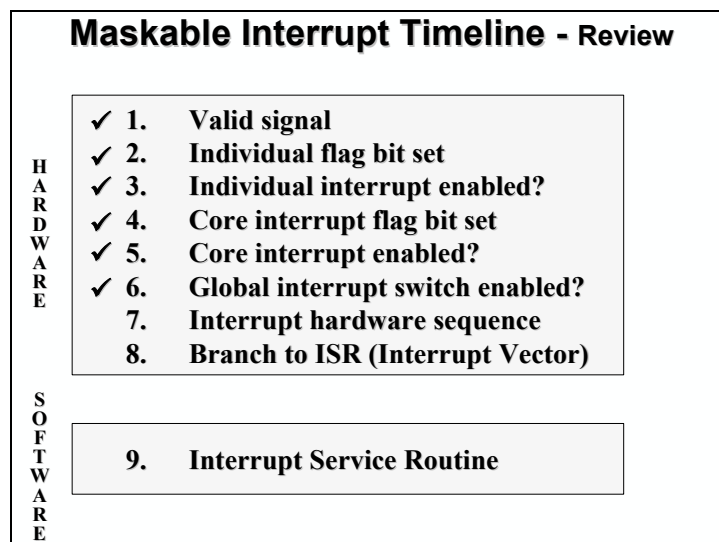
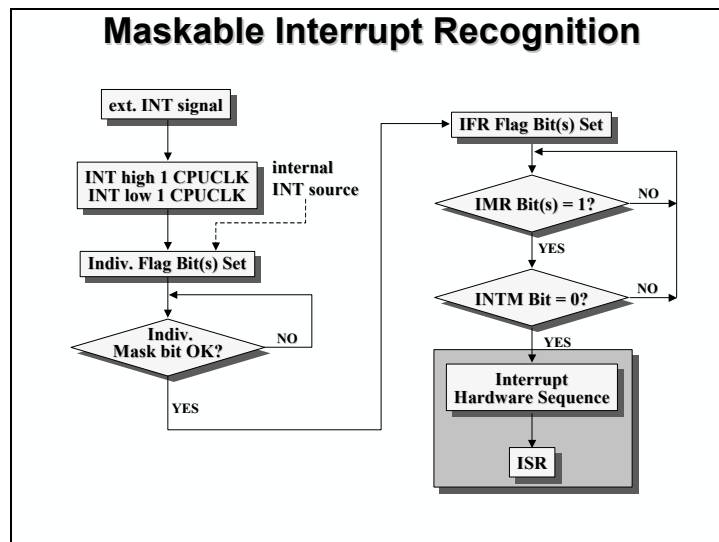
Exercise 10-1

Interrupt Exercise 10-1

At a certain spot in your program, INT2 is known to be disabled.

Write code that clears a potentially pending INT2, and then enables INT2 without affecting the enable/disable status of any other core interrupt.

Interrupt Timeline (Hardware)



Interrupt Hardware Sequence

When a properly enabled interrupt occurs, the processor performs the following actions:

7. Interrupt Hardware Sequence for Maskable Interrupts and NMI

- ◆ First, the current instructions in the pipeline are executed
- ◆ Next, the following actions occur:

| CPU Action | Description |
|--------------|---------------------------------------|
| PC + n → TOS | Push return addr. onto hardware stack |
| Vector → PC | PC loaded with interrupt vector addr. |
| INTM = 1 | Global disable of interrupts |
| IFR bit = 0 | Clears corresponding IFR bit |

Note: Individual control register flag bits must be manually cleared in software. Only IFR flag bits are automatically cleared by the CPU.

If the user desires the interrupt service routine (ISR) to be interruptible, the INTM bit must be set to zero (0) in the ISR (after context save).

Core Interrupt Vectors

After the interrupt has been recognized, the program counter (PC) is loaded with the corresponding address of the interrupt vector as shown in the table below. The user must map a set of branch instructions to the appropriate ISRs at address 0000h in program space using the .cmd file of the linker. Typically, a .sect directive is used:

```
.sect "vectors"
B    reset           ;reset vector
B    INT1_ISR        ;INT1 ISR
B    INT2_ISR        ;INT2 ISR
.
.
.
```

8. Interrupt Vector Locations

| Interrupt | Location (Hex) | Description | Priority * |
|-----------------|----------------|-------------------------|------------|
| \overline{RS} | 0 | Reset | 1 |
| INT1 | 2 | Core Interrupt #1 | 4 |
| INT2 | 4 | Core Interrupt #2 | 5 |
| INT3 | 6 | Core Interrupt #3 | 6 |
| INT4 | 8 | Core Interrupt #4 | 7 |
| INT5 | A | Core Interrupt #5 | 8 |
| INT6 | C | Core Interrupt #6 | 9 |
| TRAP | 22 | Trap Instruction Vector | – |
| NMI | 24 | Nonmaskable Interrupt | 3 |

* Software interrupts such as TRAP do not have a hardware priority

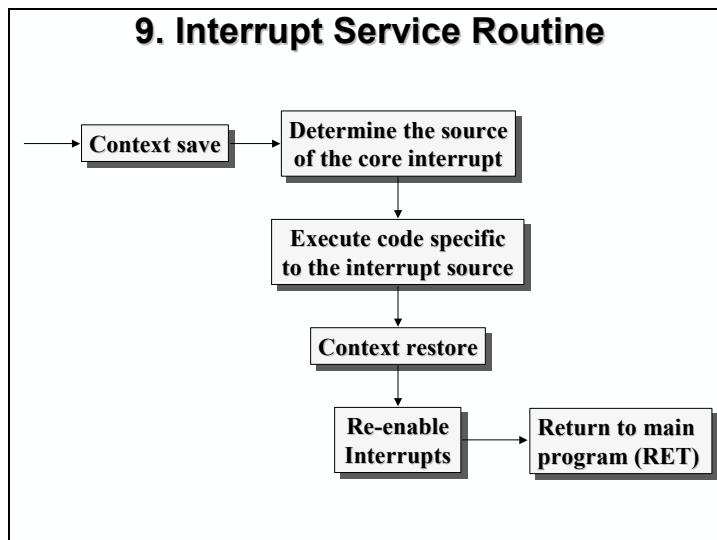
Exercise 10-2

Interrupt Exercise 10-2

Write VECTORS.ASM to include the reset, INT1, and INT3 vectors (reset, ISR1, ISR3). INT2 is not being used.

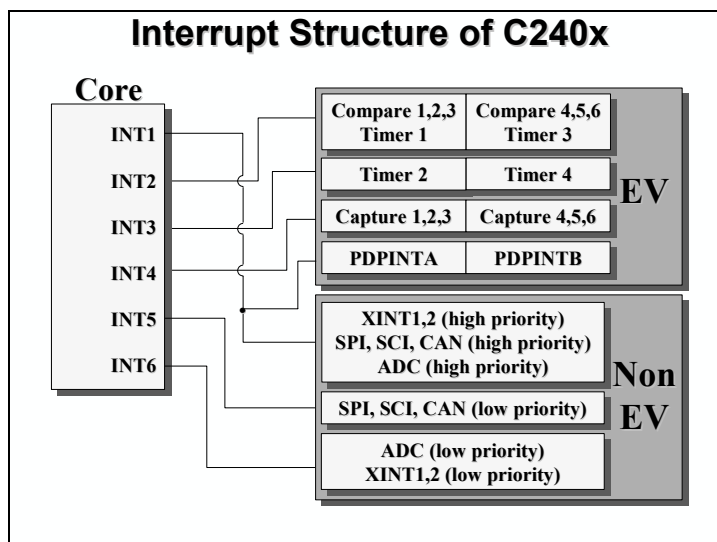
How will you handle an accidental activation of INT2?

Interrupt Service Routine



Interrupt Sources

The C240x core, as previously stated, provides for only six maskable interrupts (INT1 - INT6) plus the reset and $\overline{\text{NMI}}$ interrupts which are non-maskable. The number of peripheral and external interrupts on the C240x number more than six, and therefore the maskable core interrupts must be shared among these sources. Three core interrupts have been dedicated to event-manager peripherals, with the remaining three shared among non-event manager peripherals, the PDPINT (part of event-manager), and all other external interrupt pins.



Determining the Interrupt Source

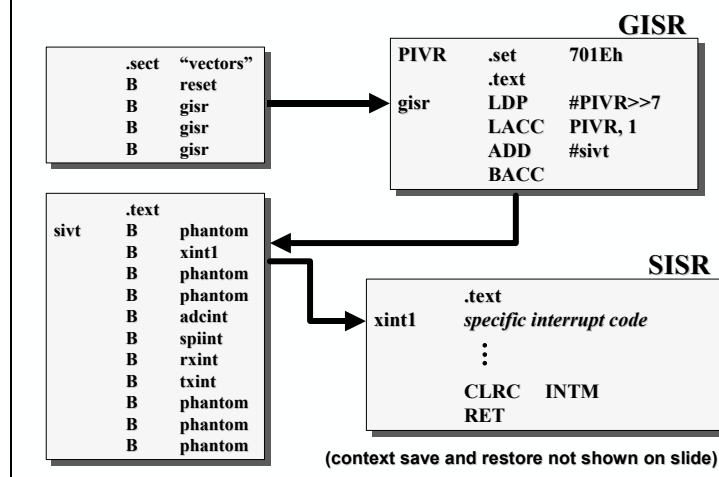
When a peripheral or external interrupt is recognized by the arbitration logic for its group, the interrupt source loads a unique value into the *Peripheral Interrupt Vector Register* (PIVR). This value is used to identify the interrupt source.

Determining Interrupt Source

- ◆ Each peripheral interrupt loads a unique offset value into the *Peripheral Interrupt Vector Registers*
 - location of PIVR @ 701Eh (in data space)

| | | | | | | | |
|--------------------------|-------|--------------------|-------|--------------------|-------|--------------------|-------|
| INT1 (core) | | INT2 (core) | | INT3 (core) | | INT4 (core) | |
| PDPINTA | 0020h | CMP1INT | 0021h | T2PINT | 002Bh | CAP1INT | 0033h |
| PDPINTB | 0019h | CMP2INT | 0022h | T2CINT | 002Ch | CAP2INT | 0034h |
| ADCINT | 0004h | CMP3INT | 0023h | T2UFINT | 002Dh | CAP3INT | 0035h |
| XINT1 | 0001h | T1PINT | 0027h | T2OFINT | 002Eh | CAP4INT | 0036h |
| XINT2 | 0011h | T1CINT | 0028h | T4PINT | 0039h | CAP5INT | 0037h |
| SPIINT | 0005h | T1UFINT | 0029h | T4CINT | 003Ah | CAP6INT | 0038h |
| RXINT | 0006h | T1OFINT | 002Ah | T4UFINT | 003Bh | | |
| CANMBINT | 0040h | CMP4INT | 0024h | T4OFINT | 003Ch | | |
| CANERINT | 0041h | CMP5INT | 0025h | | | INT5 (core) | |
| | | CMP6INT | 0026h | | | SPIINT | 0005h |
| | | T3PINT | 002Fh | INT6 (core) | | RXINT | 0006h |
| Phantom Interrupt Vector | 0000h | T3CINT | 0030h | ADCINT | 0004h | TXINT | 0007h |
| | | T3UFINT | 0031h | XINT1 | 0001h | CANMBINT | 0040h |
| | | T3OFINT | 0032h | XINT2 | 0011h | CANERINT | 0041h |

Basic Approach - (XINT1 example)



Interrupt Latency

Latency is defined as the delay between an interrupt request and the first ISR instruction fetch that is specific to that interrupt. There are several approaches to managing the interrupt latency on the C240x, with worst-case latencies of 20-22 CPU clock cycles, and minimum achievable latencies of 8 cycles. These numbers assume that the instructions already in the pipeline and those executed to determine the interrupt source are all single-cycle and are running out of zero wait-state memory with no external memory bus contention. However, many variables can cause large delays:

- RPT single in pipeline cannot be interrupted
- Processor in $\overline{\text{HOLD}}$
- Program memory wait states
- Global interrupts disabled (INTM = 0)
- Individual interrupt not enabled (mask bit = 0 in IMR, or individual enable bit not set)

Interrupt Latency

- ◆ **Latency is defined as the delay between an interrupt request and the first interrupt specific code fetch**
- ◆ **TMS320C240x Latency Components**
 - **Peripheral interface time (synchronization)**
 - **CPU response time (core latency)**
 - **ISR branching time (ISR latency)**

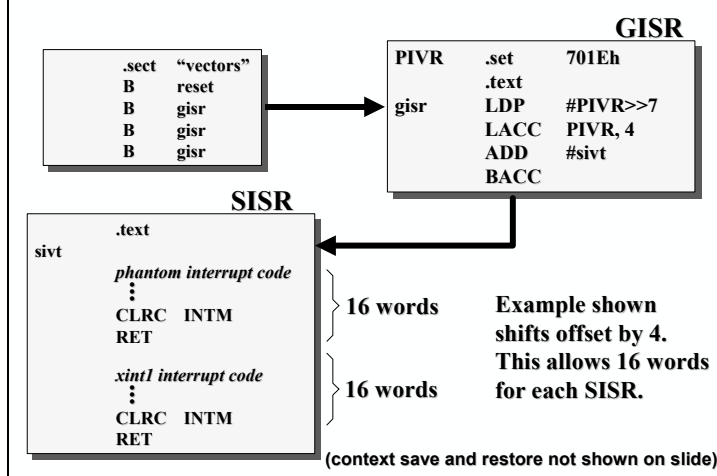
Total Latency

◆ Minimum latency for GISR/SISR procedure:

0 (to 2) cycles for peripheral interface synchronization
 1 cycle for CPU to recognize the interrupt
 3 cycles for CPU to execute already pipelined instructions
 4 cycles to branch to GISR
 8 cycles to execute GISR and branch to SIVT
 4 cycles to branch to SISR

20 (to 22) CPUCLK cycles (excluding context saves)

Reducing Latency (to 16-18 cycles)



Further Reducing Latency (to 8-10 cycles)

```
.sect "vectors"  
B reset  
B isr1  
B isr2  
B isr3
```



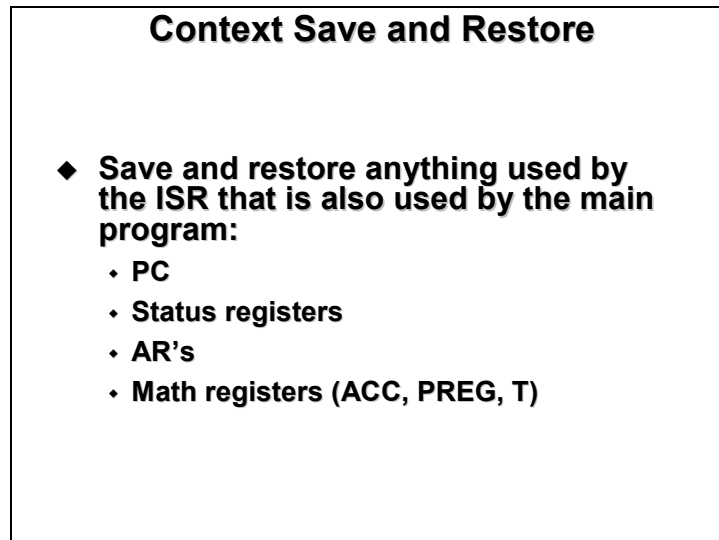
```
isr1 .text  
     specific interrupt code  
     :  
     :  
     CLRC INTM  
     RET
```

- ◆ If only 1 source for each core interrupt is enabled, then interrupt source is immediately known
- ◆ Can branch directly to interrupt specific code

(context save and restore not shown on slide)

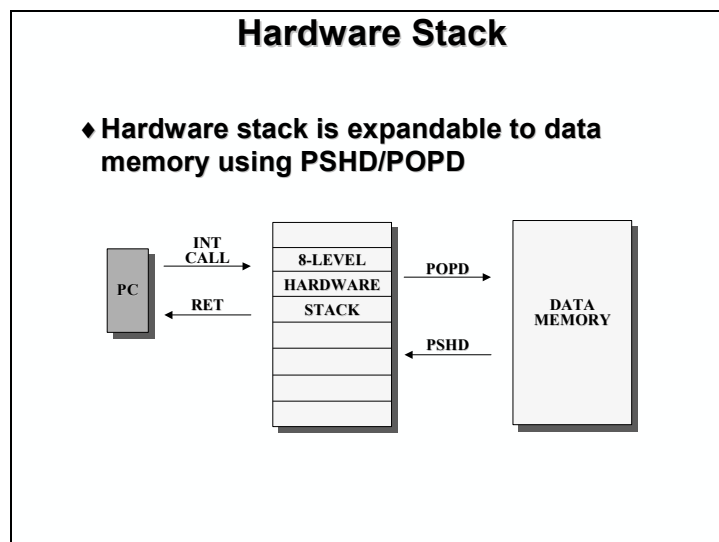
Context Save and Restore

The C240x offers numerous instructions which allow various registers to be saved and restored under program control. The user must determine which registers may be used by both the main and interrupt service routines and must assure that they have been preserved via software control. The following paragraphs describe several operations which facilitate register maintenance.



The Hardware Stack

On interrupt (and when calling subroutines), the program counter is pushed onto the eight-level-deep hardware stack. The C240x directly supports nesting of subroutine calls and interrupt service routines (ISRs) up to eight levels deep. When an ISR begins, the PC is pushed onto the stack and the PC is loaded with the appropriate interrupt vector address. Upon execution of a Return instruction (RET), the PC is reloaded with the address popped from the top of the stack (TOS).



Extending the PC Hardware Stack

Provisions have been made on the C240x to extend the hardware stack, when required, into data memory. The PUSH and POP instructions can off-load the hardware stack to data memory via the accumulator. Two additional instructions, PSHD and POPD, allow the TOS to be directly stored to, and recovered from, data memory. When nesting of interrupts and subroutines beyond eight levels is anticipated, a software stack can be implemented by using the POPD instruction at the beginning of each subroutine to save the PC in data memory. Then, before returning, a PSHD can be used to put the proper value back onto the TOS.

Note: If an overflow of the hardware stack occurs, no warning or error condition is generated. An underflow of the stack would cause the eighth return address to be repeatedly issued.

The return address is automatically pushed onto the eight-level hardware stack at the start of interrupts and subroutines, and is popped back onto the PC with a return instruction (RET). Therefore, systems which do not exceed eight concurrent levels of combined interrupts and subroutines do not need to manage the return address. On systems which cannot make this assumption, the stack can be expanded into data memory with the following instructions:

```
POPD    <dma>          ;Pop top of 8-level stack to location <dma>

PSHD    <dma>          ;Push from location <dma> to top of stack
```

Although all eight levels may be popped from the stack at any time, it is often sufficient for just one level to be off-loaded at the beginning of each interrupt or subroutine (and replaced prior to the completion of the routine).

Status Registers

ST0 and ST1 should be saved whenever an interrupt occurs. They are saved by the following instructions:

```
SST      n, <dma>      ;save status register n to data location <dma>

LST      n, <dma>      ;load status register n from location <dma>
```

These instructions operate differently from most other C240x instructions since they contain the DP and other critical status bits. The following unusual operations should be noted:

- SST with direct addressing is forced to data page 0; current DP is ignored (and unchanged). This allows the user to know which page the status has been stored to, even if DP is changed during the ISR. Indirect saves and all restores function normally.
- LST #0 does not affect the INTM or the ARP.
- LST #1 overwrites ARP with ARB.
- LST may also be used with an immediate value as an initialization operation, noting the above anomalies.

Auxiliary Registers

Auxiliary registers (ARs) should be considered for software contexting. All ARs used by both the interrupt service routine and the main program should be preserved. One way to avoid contexting ARs, however, is to dedicate a few ARs for use only in the interrupt routines and the remainder only for main routines. In this way, main routine ARs would not be used within the ISR, therefore, there would be no need to protect any ARs. When ARs are to be contexted (saved and restored), the following instructions may be used:

```
SAR    ARn,<dma>    ;Save ARn to data memory location <dma>

LAR    ARn,<dma>    ;Restore ARn from location <dma>
```

Note: In these examples, <dma> refers to standard direct or indirect addressing.

Math Registers

The accumulator (ACC) is saved and restored using the following operations:

```
Save:      SACL    <dma1>    ;Store low half of ACC to <dma1>
           SACH    <dma2>    ;Store upper ACC to <dma2>

Restore:   LACL    <dma1>    ;Load ACC from <dma1> and clear upper ACC
           ADD     <dma2>,16  ;Sum <dma2> to hi-ACC
```

In most cases, it is not necessary to preserve the Product (P) and Temporary (T) registers within most ISRs. However, in rare instances, it may be desirable to context these registers. Unlike many registers, these are not directly accessible and require some manipulation using the following operations:

```
Save:      SPM      0        ;PM saved with ST1, changed so PREG
                               ;is not shifted during context save
           SPL      <dma1>    ;Save lo-P to <dma1>
           SPH      <dma2>    ;Save hi-P to <dma2>
           MPY      #1        ;Get T Reg
           SPL      <dma3>    ;Save y to <dma3>

Restore:   LT        <dma1>    ;Put lo-P in T
           MPY      #1        ;Move lo-P from T to P
           LPH      <dma2>    ;Load P (hi) from <dma2>
           LT        <dma3>    ;Load <dma3> to T
```

Saving the P register is unnecessary if multiply instructions are immediately followed by a PAC type operation, since there is a one-cycle latency for an interrupt after any multiply. This allows the product value to be accumulated before the interrupt begins.

Managing Latency from Contexting

Managing Latency From Contexting

- ◆ Can help control latency by contexting in both GISR and SISR
 - Contexting only in GISR would require worst-case save (and increased latency) for all SISR's in group
- ◆ Context restore for both GISR and SISR can be done at end of SISR

| GISR | | | Registers Affected |
|------|-------|----------|--------------------|
| PIVR | .set | 701Eh | |
| | .text | | |
| gisr | LDP | #PIVR>>7 | DP in ST0 |
| | LACC | PIVR, 1 | ACC |
| | ADD | #sivt | C bit in ST1 |
| | BACC | | |

GISR Context Save - Direct Addressing

| | | | |
|--------|--------|---------------|----------------------------------|
| STATUS | .usect | "BLOCKB2", 2 | ;Must be located on data page 0 |
| | .bss | CONTEX1, 5, 1 | ;Located anywhere in data mem |
| PIVR | .set | 701Eh | |
| | .text | | |
| gisr1 | SST | #0, STATUS | ;SST instruction always |
| | SST | #1, STATUS+1 | ; saves to data page 0 |
| | LDP | #CONTEX1 | |
| | SACH | CONTEX1 | ;Save ACCH |
| | SACL | CONTEX1+1 | ;Save ACCL |
| | LDP | #PIVR>>7 | ;Interrupt source identification |
| | LACC | PIVR, 1 | ; code, as before |
| | ADD | #sivt | ; |
| | BACC | | ; |

SISR Context Save/Restore - Dir Addr

```

xint1    .text
         LDP    #CONTEX1
         SAR    AR2, CONTEX1+2    ;Save AR2, for example

         specific interrupt code
         ⋮

         LDP    #CONTEX1
         LAR    AR2,CONTEX1+2    ;Restore AR2
         LACL   CONTEX1+1        ;Restore ACCL w/o sign extension
         ADD    CONTEX1,16       ;Sum in ACCH
         LDP    #0               ;Need DP=0 for status registers
         LST    #1,STATUS+1      ;Restore ST1
         LST    #0,STATUS        ;Restore ST0
         CLRC   INTM             ;Enable interrupts
         RET

```

Note: Order is important for LST instructions

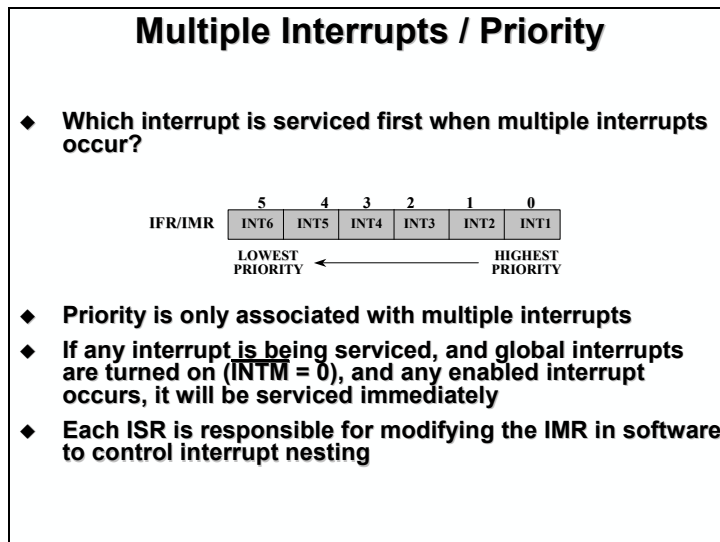
Note: An indirect addressing example of context save/restore is used in the Lab 11.

Nesting Interrupts

There may be some instance when the CPU is handling an ISR when a higher priority interrupt occurs and needs to be serviced before completion of the current ISR. Provisions have been made on the C240x to handle nested interrupts, when necessary, via software control.

Only two additional items must be added to your ISR code to enable nesting of interrupts:

1. Re-prioritizing interrupts via IMR (remember to context save and restore the IMR)
2. Re-enabling INTs via INTM



Return from ISR

Return Instruction

- ◆ **Return (RET) causes the following actions:**
 - **TOS → PC**
 - The return address is loaded back to the PC
 - **Pop stack one level**
 - Top of stack now points to next return address

Non-Maskable Interrupt

The non-maskable interrupt $\overline{\text{NMI}}$, as its name implies, is unaffected by either INTM or the contents of the IMR. $\overline{\text{NMI}}$ is typically used to perform a warm reset on the C240x processor. It may also be used as the input to a system's most time-critical interrupt event. $\overline{\text{NMI}}$ differs from $\overline{\text{RS}}$ in that it does not perform any control bit and register initializations. When $\overline{\text{NMI}}$ is recognized, the C240x completes execution of all instructions in the pipeline, disables interruptability (INTM = 0), and forces the PC to the $\overline{\text{NMI}}$ vector (location 24h). On the C240x devices, an illegal address will generate a $\overline{\text{NMI}}$.

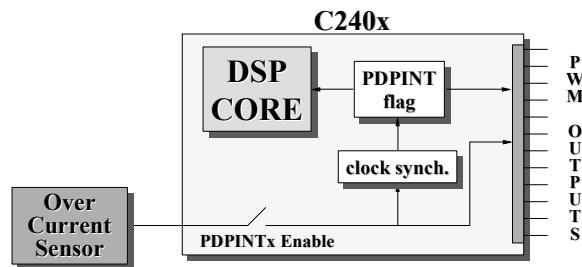
Power-Drive Protection Interrupt

The PDPINT pin provides a special interrupt function for implementing over current protection that differs from using the other external interrupt sources (e.g. XINT1, XINT2). When a valid interrupt signal occurs on the PDPINT pin, two parallel actions take place. The first action is by dedicated logic that automatically high-impedances all PWM outputs in approximately 45-55 ns. This special hardware path avoids the latency that would occur if the high-impedance function were implemented by software in an interrupt service routine. In addition, the hardware path functions independent of the CPU clock, and therefore operates even during a clock failure (e.g. external oscillator disconnect). The second action taken by PDPINT is a regular interrupt request to the CPU through the normal channels (i.e. event manager interrupt arbitration logic, IFR, IMR, INTM).

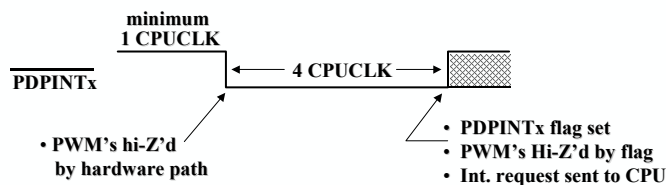
It is important to note that both PDPINT actions can be jointly disabled using the *PDPINT Enable* bit in the *EV Interrupt Mask Register A* (EVIMRA@742Ch). However, only the regular interrupt request action is affected by the IMR and INTM, the PWM high-impedance action will take place regardless of their settings.

Power Drive Protection - PDPINTA & PDPINTB

- ◆ Interrupt latency may not protect hardware when responding to over current through ISR software
- ◆ PDPINTx has a fast, clock independent logic path to high-impedance the PWM output pins (12 ns)



Valid PDPINTx Signal



- ◆ PDPINTx must stay low for 4 CPUCLK cycles or PWM's will not stay hi-Z'd

Software Initiated Interrupts

The C240x also provides facilities for interrupt service routines to be invoked under software control.

| INTR Instruction | | |
|--|-------|---------------------|
| [Label] INTR k ; $0 \leq k \leq 31$ | | |
| INTR automatically disables global interrupts when ISR is serviced (INTM set to 1) | k | Interrupt Location |
| | 0 | RS 0h |
| | 1 | INT1 2h |
| | 2 | INT2 4h |
| | 3 | INT3 6h |
| | 4 | INT4 8h |
| | 5 | INT5 Ah |
| | 6 | INT6 Ch |
| | 17 | TRAP 22h |
| | 18 | NMI 24h |
| | 8-16 | user defined 10-20h |
| | 20-31 | user defined 28-3Eh |

Trap Instruction

The TRAP instruction is a software interrupt that transfers control to program memory address 22h. Unlike the INTR instruction, TRAP does not disable global interrupts when invoked.

The TRAP instruction may be useful for debugging, or may be used to recover from software errors. For example, it may be useful to fill unused locations in program memory with the TRAP instruction, which could be used to stop execution or perform a software reset.

Review

Review

1. What are the interrupt sources?
2. What is the purpose of the IFR, IMR and INTM?
3. What conditions affect interrupt latency?

Solutions to Exercise 10-1 and 10-2

Interrupt Exercise 10-1 Solution

```
IMR    .set    0004h
IFR    .set    0006h
        .text
LDP     #0
LACL    #000010b
SACL    IFR
OR      IMR
SACL    IMR
CLRC    INTM
```

Interrupt Exercise 10-2 Solution

```
.ref     reset, isr1, isr3
.sect    "vectors"

B        reset      ;RS\
B        isr1       ;INT1
RET      ;INT2 1st word
RET      ;INT2 2nd word
B        isr3       ;INT3
```

Note: INT2 can also be handled with "B reset". All unexpected interrupts should be handled with either resetting or RET and ignore the unexpected interrupt. Also, the error could be logged and acted upon - "B phantom".

Analog-to-Digital Converter

Introduction

This module discusses the operation of the analog-to-digital converter (ADC). The system consists of a 10-bit analog-to-digital converter with 16 analog input channels. The analog input channels have a range from 0 to 3.3 volts. Also, the ADC system features a programmable autosequencer which is capable of up to sixteen autoconversions, and includes sixteen individually addressable result registers. The ADC system can be configured to work in dual-sequencer or cascaded mode. Start of conversion (SOC) can be performed by an external trigger, software, or an Event Manager event.

Learning Objectives

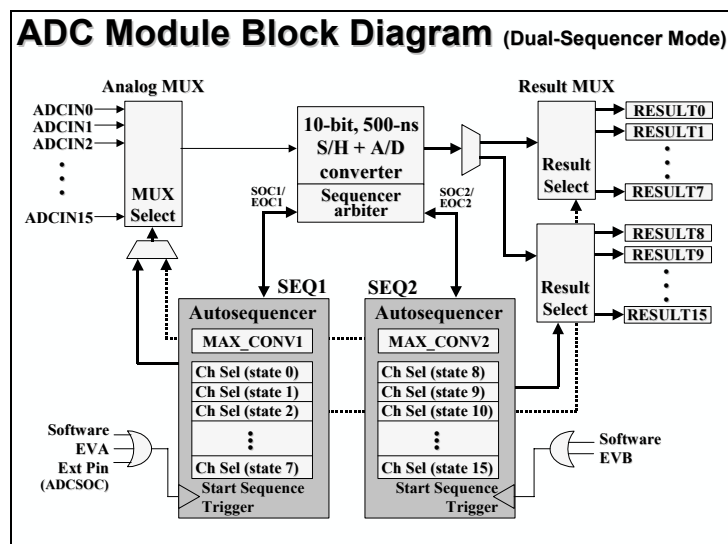
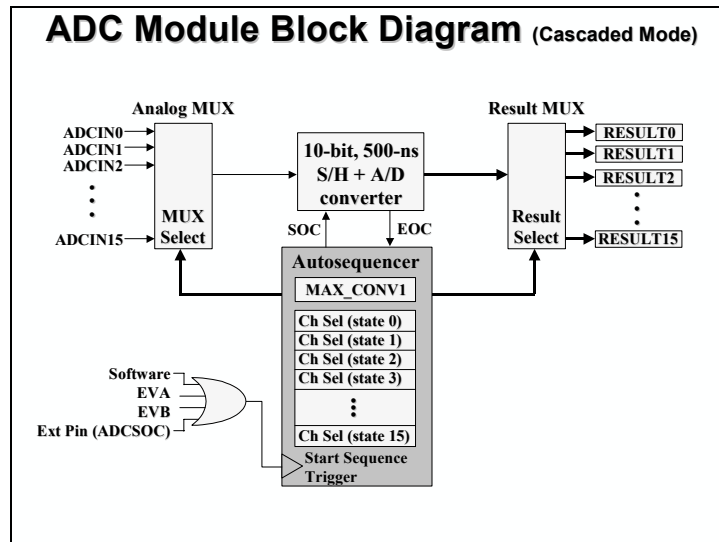
Learning Objectives

- ◆ Understand the operation of Analog-to-Digital converter
- ◆ Use Analog-to-Digital converter on the EVM to capture data

Module Topics

| | |
|--|--------------|
| Analog-to-Digital Converter..... | 11-1 |
| <i>Module Topics.....</i> | <i>11-2</i> |
| <i>Analog-to-Digital Converter.....</i> | <i>11-3</i> |
| Analog-to-Digital Converter Registers..... | 11-4 |
| Numerical Format..... | 11-10 |
| <i>Lab 11: Analog-to-Digital Converter</i> | <i>11-13</i> |
| <i>Review.....</i> | <i>11-18</i> |
| Solutions..... | 11-19 |

Analog-to-Digital Converter



ADC Module

- ◆ 10-bit ADC core with built-in Sample & Hold (S/H)
- ◆ Sixteen multiplexed analog inputs (8 on C2402)
- ◆ Fast conversion time (S/H + conversion) of 500ns
- ◆ Autosequencing capability - up to 16 autoconversions
 - Two independent 8-state sequencers
 - “Dual-sequencer mode”
 - “Cascaded mode”
- ◆ Sixteen individually addressable result registers
- ◆ Multiple trigger sources for start-of-conversion
- ◆ Flexible interrupt control

Analog-to-Digital Converter Registers

Analog-to-Digital Converter Registers

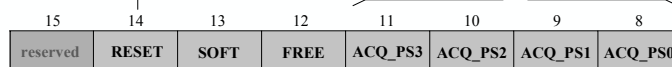
| Register | Address | Description |
|-------------|---------|---|
| ADCTRL1 | 70A0h | ADC Control Register 1 |
| ADCTRL2 | 70A1h | ADC Control Register 2 |
| MAX_CONV | 70A2h | Maximum Conversion Channels Register |
| CHSELSEQ1 | 70A3h | Channel Select Sequencing Control Register 1 |
| CHSELSEQ2 | 70A4h | Channel Select Sequencing Control Register 2 |
| CHSELSEQ3 | 70A5h | Channel Select Sequencing Control Register 3 |
| CHSELSEQ4 | 70A6h | Channel Select Sequencing Control Register 4 |
| AUTO_SEQ_SR | 70A7h | Autosequence Status Register |
| RESULT0 | 70A8h | Conversion Result Buffer Register 0 |
| RESULT1 | 70A9h | Conversion Result Buffer Register 1 |
| RESULT2 | 70AAh | Conversion Result Buffer Register 2 |
| ⋮ | ⋮ | ⋮ |
| RESULT14 | 70B6h | Conversion Result Buffer Register 14 |
| RESULT15 | 70B7h | Conversion Result Buffer Register 15 |
| CALIBRATION | 70B8h | Calibration Result (next conversion correction) |

ADC Control Register 1 - Upper Byte

ADCTRL1 @ 70A0h

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

Acquisition Time Prescale (S/H)
 Value = (binary+1) x 2
 * Time dependent on the "Conversion Clock Prescale" bit (Bit 7 "CPS")



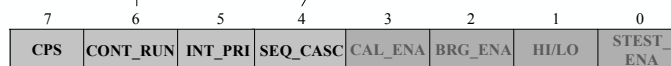
Emulation Halt Behavior
 00 = stop immediately
 10 = stop after current conversion
 x1 = free run (do not stop)

ADC Control Register 1 - Lower Byte

ADCTRL1 @ 70A0h

Continuous Run
 0 = stops after reaching end of sequence
 1 = continuous (starts all over again from "initial state")

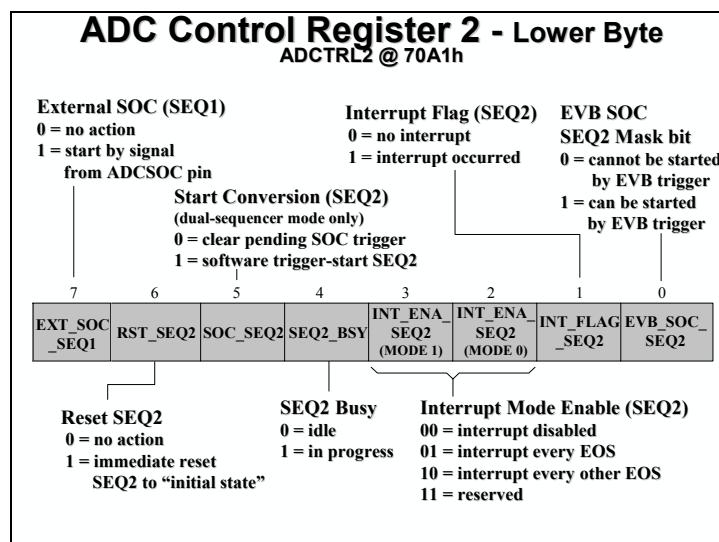
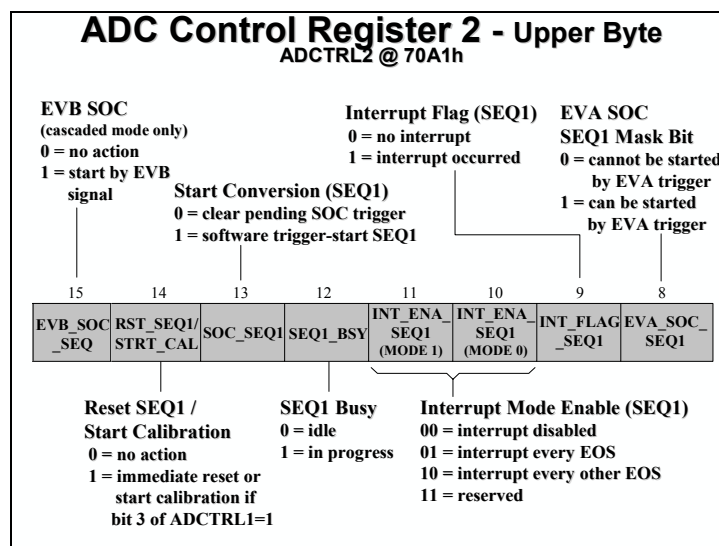
Sequencer Mode
 0 = dual mode
 1 = cascaded mode



Conversion Prescale
 0 = CLK / 1
 1 = CLK / 2

Interrupt Priority
 0 = high
 1 = low

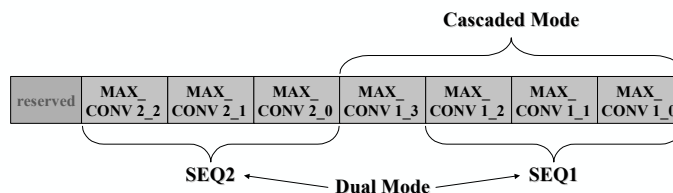
Calibration and Self-Test Functions



Maximum Conversion Channels Register

MAX_CONV @ 70A2h

- ◆ Bit fields define the maximum number of autoconversions (binary+1)



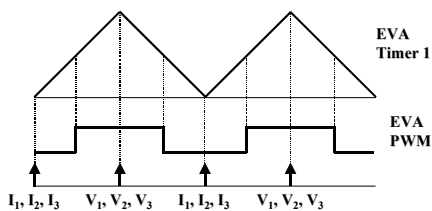
- ◆ Autoconversion session always starts with the “initial state” and continues sequentially until the “end state”, if allowed

| | SEQ1 | SEQ2 | Cascaded |
|---------------|--------|--------|----------|
| Initial state | CONV00 | CONV08 | CONV00 |
| End state | CONV07 | CONV15 | CONV15 |

ADC Input Channel Select Sequencing Control Register

| | Bits 15-12 | Bits 11-8 | Bits 7-4 | Bits 3-0 | |
|-------|------------|-----------|----------|----------|-----------|
| 70A3h | CONV03 | CONV02 | CONV01 | CONV00 | CHSELSEQ1 |
| 70A4h | CONV07 | CONV06 | CONV05 | CONV04 | CHSELSEQ2 |
| 70A5h | CONV11 | CONV10 | CONV09 | CONV08 | CHSELSEQ3 |
| 70A6h | CONV15 | CONV14 | CONV13 | CONV12 | CHSELSEQ4 |

Example - Sequencer “Start/Stop” Operation



System Requirements:

- Three autoconversions (I_1 , I_2 , I_3) off trigger 1 (Timer underflow)
- Three autoconversions (V_1 , V_2 , V_3) off trigger 2 (Timer period)

Event Manager A (EVA) and SEQ1 are used for this example

Example - Sequencer “Start/Stop” Operation Continued

- MAX_CONV1 is set to 2 and Channel Select Sequencing Control Registers are set to:

| Bits → | 15-12 | 11-8 | 7-4 | 3-0 | |
|--------|-------|-------|-------|-------|-----------|
| 70A3h | V_1 | I_3 | I_2 | I_1 | CHSELSEQ1 |
| 70A4h | x | x | V_3 | V_2 | CHSELSEQ2 |

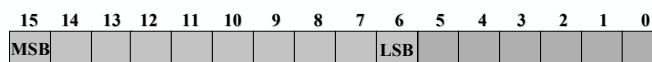
- Once reset and initialized, SEQ1 waits for a trigger
- First trigger three conversions performed: CONV00 (I_1), CONV01 (I_2), CONV02 (I_3)
- MAX_CONV1 value is reset to 2 (unless changed by software)
- SEQ1 waits for second trigger
- Second trigger three conversions performed: CONV03 (V_1), CONV04 (V_2), CONV05 (V_3)
- End of second autoconversion session, ADC Results registers have the following values:

| | | | |
|---------|-------|---------|-------|
| RESULT0 | I_1 | RESULT3 | V_1 |
| RESULT1 | I_2 | RESULT4 | V_2 |
| RESULT2 | I_3 | RESULT5 | V_3 |

- SEQ1 keeps “waiting” at current state for another trigger
- User can reset SEQ1 by software to state CONV00 and repeat same trigger 1, 2 session

ADC Conversion Result Buffer Register

RESULT0 @ 70A8h through RESULT15 @ 70B7h
(Total of 16 Registers)

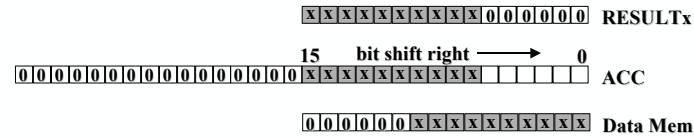


With $V_{\text{REFHI}} = 3.3 \text{ V}$, and $V_{\text{REFLO}} = 0 \text{ V}$, we have:

| <u>analog volts</u> | <u>converted value</u> | <u>RESULTx</u> |
|---------------------|------------------------|---------------------|
| 3.3 | 3FFh | 1111 1111 1100 0000 |
| 1.65 | 1FFh | 0111 1111 1100 0000 |
| 0.00322 | 1h | 0000 0000 0100 0000 |
| 0 | 0h | 0000 0000 0000 0000 |

Numerical Format

Integer Format: How Do We Handle the Shift?



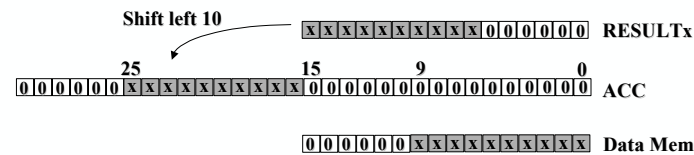
```

RESULT0 .set 70A8h
        .bss value, 1

.text
LDP     #RESULT0>>7
LACL    RESULT0
RPT     #5
SFR
LDP     #value
SACL    value
  
```

6 words, 13 cycles

Integer Format: A Better Solution for Handling the Shift



```

RESULT0 .set 70A8h
        .bss value, 1

.text
CLRC    SXM
LDP     #RESULT0>>7
LACC    RESULT0, 10
LDP     #value
SACH    value
  
```

5 words, 7 cycles

What About Signed Input Voltages?

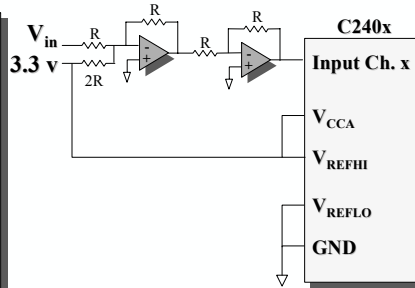
Example: $-1.65 \text{ V} \leq V_{in} \leq +1.65 \text{ V}$

Add 1.65 volts to analog input, then subtract “1.65” from digital result

```

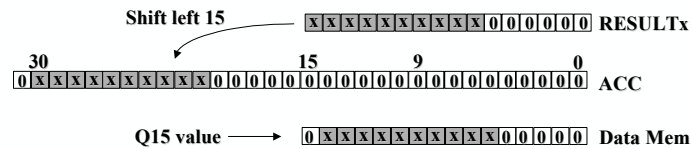
RESULT0 .set 70A8h
        .data
offset .int 1FFh
        .bss value, 1

.text
CLRC    SXM
LDP     #RESULT0>>7
LACC    RESULT0, 10
LDP     #offset
SUB     offset, 16
LDP     #value
SACH    value
  
```



What About Fractional Format?

| Volts | RESULTx | Data Mem | Q15 Fraction |
|---------|---------------------|---------------------|--------------|
| 3.3 | 1111 1111 1100 0000 | 0111 1111 1110 0000 | 0.999023 |
| 1.65 | 0111 1111 1100 0000 | 0011 1111 1110 0000 | 0.499023 |
| 0.00322 | 0000 0000 0100 0000 | 0000 0000 0010 0000 | 0.000977 |
| 0 | 0000 0000 0000 0000 | 0000 0000 0000 0000 | 0.000000 |



Q15 Fractional Format

```

RESULT0 .set 70A8h
        .bss value, 1

.text
CLRC    SXM
LDP     #RESULT0>>7
LACC    RESULT0, 15
LDP     #value
SACH    value
    
```

Unsigned Fractional Code

```

RESULT0 .set 70A8h
        .data
offset  .int 3FE0h
        .bss value, 1

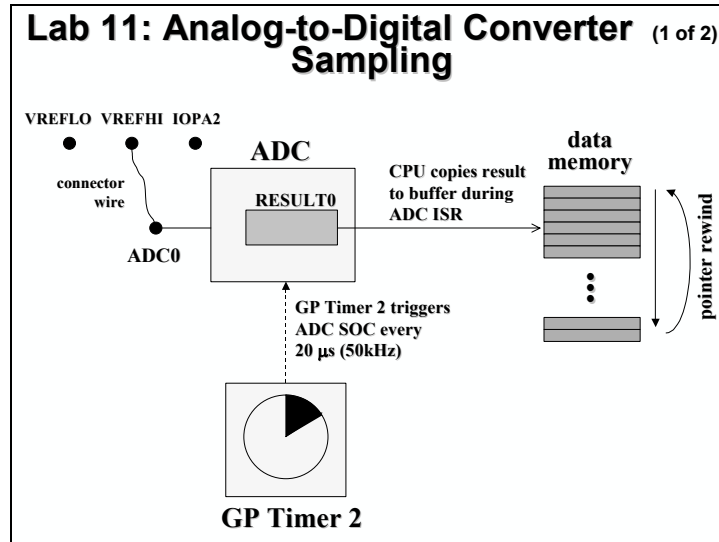
.text
CLRC    SXM
LDP     #RESULT0>>7
LACC    RESULT0, 15
LDP     #offset
SUB     offset, 16
LDP     #value
SACH    value
    
```

Signed Fractional Code

Lab 11: Analog-to-Digital Converter

➤ Objective

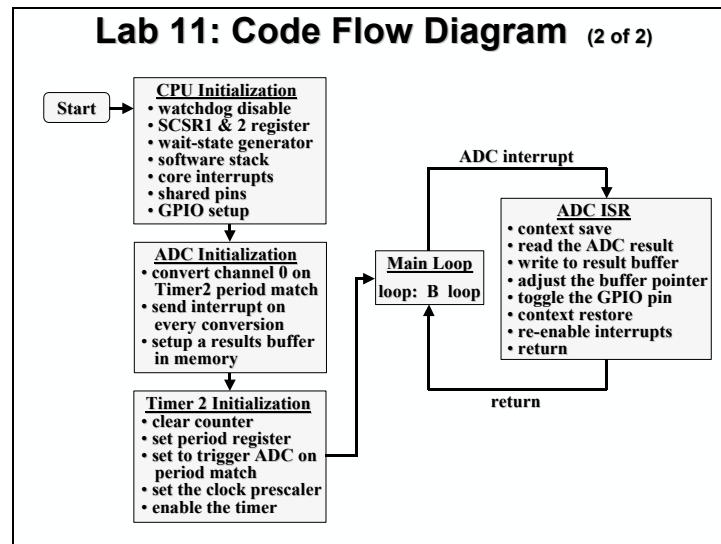
The objective of this lab is to apply the techniques discussed in Module 11 and to become familiar with the programming and operation of the on-chip analog-to-digital converter. The DSP will be setup to sample a single ADC input channel at a prescribed sampling rate and store the conversion result in a buffer in the DSP memory. This buffer will operate in a circular fashion, such that new conversion data continuously overwrites older results in the buffer.



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

1. Using software
 - a. SOC_SEQ1/SOC_SEQ2 bit in ADCTRL2 causes an SOC upon completion of the current conversion (if the ADC is currently idle, an SOC occurs immediately)
2. Automatically triggered on user selectable event manager conditions
 - a. GP Timer 1 or 2 (EVA); 3 or 4 (EVB) underflow (e.g. timer count = 0)
 - b. GP Timer 1 or 2 (EVA); 3 or 4 (EVB) period match
 - c. GP Timer 1 or 2 (EVA); 3 or 4 (EVB) compare match
3. Externally triggered using a pin
 - a. ADCSOC pin
 - b. CAP3 pin - EVA; CAP6 pin - EVB (capture unit #3 / capture unit #6 edge detection)

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 GP timers (GP Timer 2) will be configured to automatically trigger an SOC at the desired sampling rate (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 (IOPA2) high and low in the ADC interrupt service routine. This pin will be connected to the ADC input pin, and sampled. After taking some data, Code Composer will be used to plot the results. A flow chart of the code is shown in the following slide.



Notes

- Program performs conversion on ADC channel 0 (ADCIN0 pin)
- General Purpose Timer 2 is used to auto-trigger the conversions at a 50kHz sampling rate
- Data is continuously stored in a circular buffer
- IOPA2 pin is also toggled in the ADC ISR

➤ Procedure

Create Make File

1. **NOTE:** LAB11.ASM, VECS_11.ASM and LAB11.CMD files have been provided as a starting point for the lab and need to be completed. *DO NOT copy files from a previous lab.*
2. Create a new project called LAB11.MAK and add LAB11.ASM, VECS_11.ASM and LAB11.CMD to it. Check your file list to make sure all the files are there. (F2407.h will be added automatically during the Build). Be sure to setup the Build Options by clicking: Project → Options on the menu bar. Select the Assembler tab. In the middle of the screen check "Enable Source Level Debugging". Next, select the Linker tab. In the middle of the screen select "No Autoinitialization". Create a map file named LAB11.MAP. Select OK to save the Build Options.

Setup ADC Initialization and Enable Core Interrupts

3. Edit LAB11.ASM and modify it to implement the ADC initialization as described above in the objective for the lab. (Hint – in your code first reset ADC module before setting other registers). Also, setup and enable the desired core interrupts. Save your work.

4. Edit `VECS_11.ASM` and modify it to include the vector interrupt (high priority) for the ADC interrupt service routine. Make sure that the labels are visible to the linker. Save your work.

Build and Load

5. Click the "Rebuild All" 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 ("Option" menu, click on "Program Load...") load the output file into the target. Click: File → Load Program...

Then reset the DSP by clicking on: Debug → Reset DSP

If you would like to debug using both source and assembly, right click on the `VECTORS.ASM` window and select `Mixed Mode`.

Testing Lab11

7. After loading, the `VECS_11.ASM` window should open with the yellow highlight on "B start". Single-step your code one time into the `LAB11.ASM` file. Place the cursor at the "loop: B loop" line and set a breakpoint by right clicking the mouse key and select `Toggle breakpoint`. Notice that line is highlighted indicating that the breakpoint has been set. Next, locate the code segment that initializes the ADC results buffer with the value `0x2407`. Notice the effective use of the `RPT` instruction coupled with the `SACL` for this purpose.
8. Open a memory window to view some of the contents of the ADC results buffer. The address label for the ADC results buffer is `adc_buf`.
9. Run the code. It should halt when it hits the breakpoint set at the main loop. Verify that the ADC results buffer has been initialized with the expected value (e.g. `0x2407`).
10. Remove the breakpoint at "loop: B loop" by right clicking the mouse key and select `Toggle breakpoint`.
11. Using the connector wire provided, connect the `ADCIN0` (pin # P2-23) to `VREFLO` "GND" (pin # P2-22) on the EVM. **Exercise care when doing this as the power to the EVM is on, and we do not want to damage the EVM!** 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 `ADCIN0` (pin # P2-23) to `VREFHI` "+3.3V" (pin # P2-21) on the EVM. **Exercise care when doing this as the power to the EVM is on, and we do not want to damage the EVM!** Then run the code again, and halt it after a few seconds. Verify that the ADC results buffer contains the expected value of `0x03FF`.
13. Adjust the connector wire to connect the `ADCIN0` (pin # P2-23) to `IOPA2` (pin # P4-16) on the EVM. **Exercise care when doing this as the power to the EVM is on, and we do not want to damage the EVM!** 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 0x03FF 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 | adc_buf |
| 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 IOPA2 pin alternately high and low. If you had an oscilloscope available to display IOPA2, you would expect to see a square-wave. Why does Code Composer plot resemble a saw-tooth wave? What is the signal processing term for what is happening here?
16. Recall that the program toggled the IOPA2 pin at a 50kHz rate. Therefore, a complete cycle (toggle high, then toggle low) occurs at half this rate, or 25kHz. We therefore expect the period of the waveform to be 40μs. Confirm this by measuring the period of the saw-tooth 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.

End of Exercise

Optional Exercise - Servicing the Watchdog Timer

In the previous portion of the lab, we disabled the watchdog timer at the beginning of the program. In an actual application, it is often desired to keep the watchdog active, and therefore one must periodically service the watchdog by writing the correct key sequence to the WDKEY register. In this section of the lab, you will modify your code to properly service the watchdog timer.

Recall that the watchdog key sequence involves writing the sequence AAh followed by 55h to the WDKEY register. It is generally poor practice to place both writes in an interrupt service routine. In the event of a main code crash, the interrupt can still continue to occur, and the watchdog will continue to be serviced. However, upon completion of the interrupt service routine, a return will occur back to the main crashed code, and the crash will not be caught by the watchdog timer. Therefore, good practice is to place one of the key code writes in your main routine, and the other

key code write in some periodic interrupt service routine. This approach allows the watchdog to catch crashes in both the code foreground and code background.

1. In LAB11.ASM, change the SPLK statement that disables the watchdog so that the watchdog is no longer disabled. Leave the Watchdog Prescale Select bit set so that the WDCLK divider remains equal to 1.
2. The "Main loop" in LAB11.ASM currently consists of an endless loop with a single NOP instruction in it. Replace this NOP with code that will write the 55h keycode value to the WDKEY register.
3. In the ADC interrupt service routine, add code just before the context restore that writes the AAh keycode value to the WDKEY register.

Questions:

Based on the prescale setting for the WDCLK, what is the timeout frequency of the watchdog?

Answer: 228.9 Hz ($30 \text{ MHz} * 1/512 * 1/256$).

At what frequency will the watchdog be serviced?

Answer: 50 kHz

Is this sufficient to ensure that the watchdog will not timeout?

Answer: Yes

4. Rebuild and load the program. Reset the DSP by clicking: Debug → Reset DSP. The VECS_11.ASM window should open with the yellow highlight on "B_start". Set a breakpoint on this instruction.
5. Run the code by either hitting <F5>, or by clicking Debug → Run. The DSP should immediately halt at the "B_start" breakpoint. Do another Run. This time, the DSP should get past the breakpoint and be running as normal. To confirm proper program operation, you can halt the DSP and then repeat step 16 above (graph the adc_buf contents). You should see the same 25kHz saw tooth wave as before.
6. To prove that the watchdog is actually running and being serviced, comment out the AAh write to the WDKEY register that you added in the ADC ISR. Rebuild and load the program, and make sure that the breakpoint is still set on the "B_start" instruction. Now, no matter how many times you do a Run, the DSP will halt at the breakpoint. This is because the watchdog is timing out and causing a system reset.
7. If you plot the ADC buffer contents at this point, you probably still see the 25kHz saw tooth wave. How is this possible when the watchdog is causing the DSP to continually reset?

Review

Review

- ◆ How are “autosequencing conversions” performed on the ADC module?
- ◆ What are the two sequencer modes?
- ◆ How fast is the ADC?
- ◆ How many analog inputs?

Solutions

; SOLUTION FILE FOR LAB11.ASM

```

                .def      start
                .def      adc_isr
                .include f2407.h                ;address definitions

adc_rate        .set      599                ;50kHz sampling rate
adc_buf_len     .set      300                ;ADC results buffer length
stk_len        .set      100                ;stack length

                .bss      temp,1              ;general purpose variable

adc_buf_ptr     .usect    "buffer",1         ;ptr to next free buff addr
adc_buf         .usect    "buffer",adc_buf_len ;reserve space for buffer
stk             .usect    "stack",stk_len     ;reserve space for stack

                .text

start:

;~~~~~
;Disable the watchdog
;~~~~~
                LDP        #DP_PF1            ;set data page

                SPLK       #11101000b, WDCR
* bit 7         1:        clear WD flag
* bit 6         1:        disable the dog
* bit 5-3       101:      must be written as 101
* bit 2-0       000:      WDCLK divider = 1

;~~~~~
;Setup the system control registers
;~~~~~
                LDP        #DP_PF1            ;set data page

                SPLK       #0000000011111101b, SCSR1
;
;                FEDCBA9876543210
* bit 15       0:        reserved
* bit 14       0:        CLKOUT = CPUCLK
* bit 13-12    00:       IDLE1 selected for low-power mode
* bit 11-9     000:      PLL x4 mode
* bit 8        0:        reserved
* bit 7        1:        1 = enable ADC module clock
* bit 6        1:        1 = enable SCI module clock
* bit 5        1:        1 = enable SPI module clock
* bit 4        1:        1 = enable CAN module clock
* bit 3        1:        1 = enable EVB module clock
* bit 2        1:        1 = enable EVA module clock
* bit 1        0:        reserved
* bit 0        1:        clear the ILLADR bit

                SPLK       #0000000000001111b, SCSR2
;
;                FEDCBA9876543210

```

```

* bit 15-6      0's:   reserved
* bit 5         0:     DO NOT clear the WD OVERRIDE bit
* bit 4         0:     XMIF_HI-Z, 0=normal mode, 1=Hi-Z'd
* bit 3         1:     1 = disable the BOOT ROM
* bit 2         1:     MP/MC*, 1 = Flash addresses mapped external
* bit 1-0       11:    11 = SARAM mapped to prog and data

;~~~~~
;Set wait states for external memory interface on LF2407 EVM
;~~~~~
        LDP      #temp      ;set data page

        SPLK     #0000000001000000b, temp
;        |||||
;        FEDCBA9876543210
* bit 15-11     0's:   reserved
* bit 10-9      00:    bus visibility off
* bit 8-6       001:   1 wait-state for I/O space
* bit 5-3       000:   0 wait-state for data space
* bit 2-0       000:   0 wait-state for program space

        OUT      temp, WSGR

;~~~~~
;Setup the software stack
;~~~~~
        LAR      AR1, #stk      ;AR1 is stack pointer
        MAR      *, AR1        ;ARP = AR1

;~~~~~
;Setup the core interrupts
;~~~~~
        LDP      #0h           ;set data page
        SPLK     #111111b, IFR  ;clear any pending interrupts
        SPLK     #000001b, IMR  ;enable desired interrupts

;~~~~~
;Setup shared I/O pins
;~~~~~
        LDP      #DP_PF2       ;set data page

        SPLK     #0000000000000000b, MCRA
*        |||||
*        FEDCBA9876543210
* bit 15        0:      0=IOPB7,      1=TCLKINA
* bit 14        0:      0=IOPB6,      1=TDIRA
* bit 13        0:      0=IOPB5,      1=T2PWM/T2CMP
* bit 12        0:      0=IOPB4,      1=T1PWM/T1CMP
* bit 11        0:      0=IOPB3,      1=PWM6
* bit 10        0:      0=IOPB2,      1=PWM5
* bit 9         0:      0=IOPB1,      1=PWM4
* bit 8         0:      0=IOPB0,      1=PWM3
* bit 7         0:      0=IOPA7,      1=PWM2
* bit 6         0:      0=IOPA6,      1=PWM1
* bit 5         0:      0=IOPA5,      1=CAP3
* bit 4         0:      0=IOPA4,      1=CAP2/QEP2
* bit 3         0:      0=IOPA3,      1=CAP1/QEP1

```



```

* bit 2      0:      0=IOPA2,      1=XINT1
* bit 1      0:      0=IOPA1,      1=SCIRXD
* bit 0      0:      0=IOPA0,      1=SCITXD

        SPLK      #1111111000000000b,MCRB
*          |||||
*          FEDCBA9876543210
* bit 15     1:      0=reserved,    1=TMS2 (always write as 1)
* bit 14     1:      0=reserved,    1=TMS (always write as 1)
* bit 13     1:      0=reserved,    1=TD0 (always write as 1)
* bit 12     1:      0=reserved,    1=TDI (always write as 1)
* bit 11     1:      0=reserved,    1=TCK (always write as 1)
* bit 10     1:      0=reserved,    1=EMU1 (always write as 1)
* bit 9      1:      0=reserved,    1=EMU0 (always write as 1)
* bit 8      0:      0=IOPD0,      1=XINT2/ADCSOC
* bit 7      0:      0=IOPC7,      1=CANRX
* bit 6      0:      0=IOPC6,      1=CANTX
* bit 5      0:      0=IOPC5,      1=SPISTE
* bit 4      0:      0=IOPC4,      1=SPICLK
* bit 3      0:      0=IOPC3,      1=SPISOMI
* bit 2      0:      0=IOPC2,      1=SPISIMO
* bit 1      0:      0=IOPC1,      1=BIO*
* bit 0      0:      0=IOPC0,      1=W/R*

        SPLK      #0000000000000000b,MCRC
*          |||||
*          FEDCBA9876543210
* bit 15     0:      reserved
* bit 14     0:      0=IOPF6,      1=IOPF6
* bit 13     0:      0=IOPF5,      1=TCLKINB
* bit 12     0:      0=IOPF4,      1=TDIRB
* bit 11     0:      0=IOPF3,      1=T4PWM/T4CMP
* bit 10     0:      0=IOPF2,      1=T3PWM/T3CMP
* bit 9      0:      0=IOPF1,      1=CAP6
* bit 8      0:      0=IOPF0,      1=CAP5/QEP4
* bit 7      0:      0=IOPE7,      1=CAP4/QEP3
* bit 6      0:      0=IOPE6,      1=PWM12
* bit 5      0:      0=IOPE5,      1=PWM11
* bit 4      0:      0=IOPE4,      1=PWM10
* bit 3      0:      0=IOPE3,      1=PWM9
* bit 2      0:      0=IOPE2,      1=PWM8
* bit 1      0:      0=IOPE1,      1=PWM7
* bit 0      0:      0=IOPE0,      1=CLKOUT

;~~~~~
;Setup IOPA2 pin for use as output
;~~~~~
        LDP      #DP_PF2          ;set data page
        LACC     PADATDIR          ;ACC = PADATDIR
        OR       #0400h           ;IOPA2 is output
        SACL     PADATDIR          ;write back to GPIO port register

;~~~~~
;Setup the ADC
;~~~~~
        LDP      #DP_PF2          ;set data page

```

```

        SPLK      #0100000000000000b, ADCTRL1
*
*      |||||
*      FEDCBA9876543210
* bit 14      1:      1 = reset ADC module

        SPLK      #0000000000000000b, MAX_CONV
*
*      |||||
*      FEDCBA9876543210
* bit 15-7    0's:    reserved
* bit 6-4     000:    MAX_CONV2 value
* bit 3-0     0000:   MAX_CONV1 value (0 means 1 conversion)

        SPLK      #0000000000000000b, CHSELSEQ1
*
*      |||||
*      FEDCBA9876543210
* bit 15-12   0000:   CONV03 channel
* bit 11-8    0000:   CONV02 channel
* bit 7-4     0000:   CONV01 channel
* bit 3-0     0000:   CONV00 channel (only active conversion)

        SPLK      #0010000000010000b, ADCTRL1
*
*      |||||
*      FEDCBA9876543210
* bit 15      0:      reserved
* bit 14      0:      RESET, 0=no action, 1=reset ADC
* bit 13-12   10:     SOFT and FREE, 10=stop after current conversion
* bit 11-8    0000:   ACQ_Prescaler, 0000 = 1 x Tclk
* bit 7       0:      CPS, 0: Fclk=CPUCCLK/1, 1: Fclk=CPUCCLK/2
* bit 6       0:      CONT_RUN, 0=start/stop mode, 1=continuous run
* bit 5       0:      0=hi priority int, 1=low priority int
* bit 4       1:      0=dual sequencer, 1=cascaded sequencer
* bit 3       0:      0=calibration mode disabled
* bit 2       0:      BRG_ENA, used in calibration mode only
* bit 1       0:      HI/LO, no effect in normal operation mode
* bit 0       0:      0=self-test mode disabled

        SPLK      #0100011100000010b, ADCTRL2
*
*      |||||
*      FEDCBA9876543210
* bit 15      0:      EVB_SOC_SEQ, 0=no action
* bit 14      1:      RST_SEQ1/STRT_CAL, 0=no action
* bit 13      0:      SOC_SEQ1, 0=clear any pending SOC's
* bit 12      0:      SEQ1_BSY, read-only
* bit 11-10   01:     INT_ENA_SEQ1, 01=int on every SEQ1 conv
* bit 9       1:      INT_FLAG_SEQ1, write 1 to clear
* bit 8       1:      EVA_SOC_SEQ1, 1=SEQ1 start from EVA
* bit 7       0:      EXT_SOC_SEQ1, 1=SEQ1 start from ADCSOC pin
* bit 6       0:      RST_SEQ2, 0=no action
* bit 5       0:      SOC_SEQ2, no effect in cascaded mode
* bit 4       0:      SEQ2_BSY, read-only
* bit 3-2     00:     INT_ENA_SEQ2, 00=int disabled
* bit 1       1:      INT_FLAG_SEQ2, write 1 to clear
* bit 0       0:      EVB_SOC_SEQ2, 1=SEQ2 started by EVB

;~~~~~
;Setup the buffer for the ADC results
;~~~~~

```

```

LDP      #adc_buf_ptr      ;set data page
LAR      AR0, #adc_buf      ;pointer to results buffer
SAR      AR0, adc_buf_ptr   ;initialize adc_buf_ptr
MAR      *, AR0             ;ARP = AR0
LACC     #2407h             ;ACC=0x2407
LDP      #temp
SPLK     #adc_buf_len-1, temp
RPT      temp               ;repeat #adc_buf_len-1 times
SACL     *+                 ;initialize the buffer

;~~~~~
;Setup GP Timer2 to trigger an ADC conversion
;~~~~~
LDP      #DP_EVA            ;set data page

SPLK     #0000h, T2CNT      ;clear timer2 counter
SPLK     #adc_rate, T2PR    ;set timer2 period

SPLK     #0000010000000000b, GPTCONA ;init GPTCON register
*        |||||
*        FEDCBA9876543210
* bit 15  0:      reserved
* bit 14  0:      T2STAT - read only
* bit 13  0:      T1STAT - read only
* bit 12-11 00:    reserved
* bit 10-9  10:    T2TOADC, 10 = ADCSOC on period match
* bit 8-7   00:    T1TOADC, 00 = no ADCSOC
* bit 6     0:      1 = enable all timer compare outputs
* bit 5-4   00:    reserved
* bit 3-2   00:    00 = T2PIN forced low
* bit 1-0   00:    00 = T1PIN forced low

SPLK     #0001000001000000b, T2CON ;init T2CON register
*        |||||
*        FEDCBA9876543210
* bit 15-14 00:    stop immediately on emulator suspend
* bit 13     0:      reserved
* bit 12-11 10:    10 = continous-up count mode
* bit 10-8   000:    000 = x/1 prescaler
* bit 7      0:      0 = use own TENABLE bit
* bit 6      1:      1 = enable timer
* bit 5-4   00:    00 = CPUCLK is clock source
* bit 3-2   00:    00 = reload compare reg on underflow
* bit 1      0:      0 = disable timer compare
* bit 0      0:      0 = use own period register

;~~~~~
;Setup the event manager interrupts
;~~~~~
LDP      #DP_EVA            ;set data page
SPLK     #0FFFFh, EVAIFRA ;clear all EVA group A interrupts
SPLK     #0FFFFh, EVAIFRB ;clear all EVA group B interrupts
SPLK     #0FFFFh, EVAIFRC ;clear all EVA group C interrupts
SPLK     #00000h, EVAIMRA ;enabled desired EVA group A interrupts
SPLK     #00000h, EVAIMRB ;enabled desired EVA group B interrupts
SPLK     #00000h, EVAIMRC ;enabled desired EVA group C interrupts

```

```

LDP      #DP_EVB          ;set data page
SPLK     #0FFFFh, EVBIFRA ;clear all EVB group A interrupts
SPLK     #0FFFFh, EVBIFRB ;clear all EVB group B interrupts
SPLK     #0FFFFh, EVBIFRC ;clear all EVB group C interrupts
SPLK     #00000h, EVBIMRA ;enabled desired EVB group A interrupts
SPLK     #00000h, EVBIMRB ;enabled desired EVB group B interrupts
SPLK     #00000h, EVBIMRC ;enabled desired EVB group C interrupts

;~~~~~
;Enable global interrupts
;~~~~~
CLRC     INTM              ;enable global interrupts

;~~~~~
;Main loop
;~~~~~
loop:    NOP
        B          loop      ;branch to loop

*****
*  G E N E R A L  I N T E R R U P T  S E R V I C E  R O U T I N E S  *
*****

;~~~~~
;ADC Interrupt Service Routine
;~~~~~
adc_isr:

;context save
MAR      *,AR1              ;ARP=stack pointer
MAR      *+                 ;skip one stack location
SST      #1, *+             ;save ST1
SST      #0, *+             ;save ST0
SACH     *+                 ;save ACCH
SACL     *+                 ;save ACCL
SAR      AR2, *+            ;save AR2

;clear the INT_FLAG_SEQ1 and read the ADC result
CLRC     SXM
LDP      #DP_PF2            ;set data page
LACC     ADCTRL2             ;read and write ADCTRL2
SACL     ADCTRL2             ;to clear the INT_FLAG_SEQ1
LACC     RESULT0,10          ;read ADC RESULT0

;store the data value to the buffer
LDP      #adc_buf_ptr        ;set data page
LAR      AR2, adc_buf_ptr     ;AR2 points to the buffer
MAR      *, AR2              ;set ARP
SACH     *+                 ;store result
SAR      AR2, adc_buf_ptr     ;store updated pointer

;brute-force the circular buffer
LAR      AR0, #(adc_buf+adc_buf_len-1);AR0 pts to last buff entry
CMPR     2                   ;TC set if AR(ARP) > AR0
BCND     adc_isr1, NTC        ;branch if TC not set

```

```
        SPLK    #adc_buf, adc_buf_ptr        ;re-init the pointer

adc_isr1:

;reset ADC SEQ1 to CONV00 state
        LDP     #DP_PF2                    ;set data page
        LACC    ADCTRL2                    ;read ADCTRL2
        OR      #4000h                      ;set bit 14 (RST_SEQ1/STRT_CAL bit)
        SACL    ADCTRL2                    ;write back to reset SEQ1

;toggle the IOPA2 pin
        LDP     #DP_PF2                    ;set data page
        LACC    PADATDIR                    ;ACC = PADATDIR
        XOR     #0004h                      ;toggle IOPA2 bit
        SACL    PADATDIR                    ;write back to GPIO port register

;context restore
        MAR     *, AR1    ;ARP = AR1
        MAR     *-        ;SP points to last entry
        LAR     AR2, *-    ;restore AR2
        LACL    *-        ;restore ACCL
        ADD     *-,16      ;restore ACCH
        LST     #0, *-    ;restore ST0
        LST     #1, *-    ;restore ST1, de-allocate skipped stack location
        CLRC    INTM      ;re-enable global interrupts
        RET
```

```

; SOLUTION FILE FOR VECS_11.ASM

        .ref      start
        .ref      adc_isr

        .sect     "vectors"

;~~~~~
;Interrupt vector table for core
;~~~~~

int1:    B        start           ;00h reset
int2:    B        adc_isr       ;02h INT1
int3:    B        int2           ;04h INT2
int4:    B        int3           ;06h INT3
int5:    B        int4           ;08h INT4
int6:    B        int5           ;0Ah INT5
int7:    B        int6           ;0Ch INT6
int8:    B        int7           ;0Eh reserved
int9:    B        int8           ;10h INT8  user-defined
int10:   B        int9           ;12h INT9  user-defined
int11:   B        int10          ;14h INT10 user defined
int12:   B        int11          ;16h INT11 user defined
int13:   B        int12          ;18h INT12 user defined
int14:   B        int13          ;1Ah INT13 user defined
int15:   B        int14          ;1Ch INT14 user defined
int16:   B        int15          ;1Eh INT15 user defined
int17:   B        int16          ;20h INT16 user defined
int18:   B        int17          ;22h TRAP
int19:   B        int18          ;24h NMI
int20:   B        int19          ;26h reserved
int21:   B        int20          ;28h INT20 user defined
int22:   B        int21          ;2Ah INT21 user defined
int23:   B        int22          ;2Ch INT22 user defined
int24:   B        int23          ;2Eh INT23 user defined
int25:   B        int24          ;30h INT24 user defined
int26:   B        int25          ;32h INT25 user defined
int27:   B        int26          ;34h INT26 user defined
int28:   B        int27          ;36h INT27 user defined
int29:   B        int28          ;38h INT28 user defined
int30:   B        int29          ;3Ah INT29 user defined
int31:   B        int30          ;3Ch INT30 user defined
int31:   B        int31          ;3Eh INT31 user defined

```

Introduction

This module explains how to generate PWM waveforms using the timers and compare units. Also, the capture units, quadrature encoder pulse circuit, and the hardware deadband units will be discussed. All devices of the C240x family (with the exception of the C2402 and C24x) have two event managers, EVA and EVB. These two event managers are identical to each other in terms of functionality. Register mapping and bit definitions are also identical, with the exception of naming conventions and register addresses. Therefore, for simplicity, only the functionality of EVA will be explained.

Learning Objectives

Learning Objectives

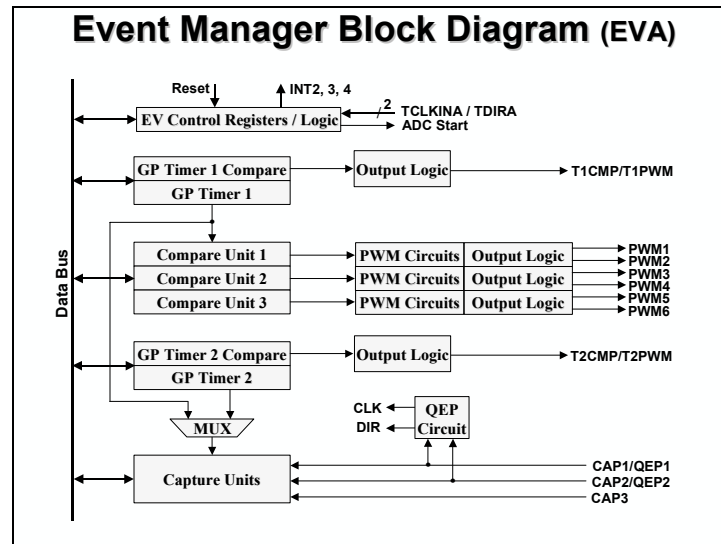
- ◆ **Pulse Width Modulation (PWM) Review**
- ◆ **Generate PWM with the Event Manager:**
 - General-Purpose Timer
 - Compare Units
- ◆ **Explain other Event Manager functions:**
 - Capture Units
 - Quadrature Encoder Pulse (QEP) Circuit

Note: Two identical Event Manager (EVA and EVB) modules are available on some devices. For simplicity, only EVA will be explained.

Module Topics

| | |
|--|--------------|
| Event Manager | 12-1 |
| <i>Module Topics.....</i> | <i>12-2</i> |
| <i>Event Manager.....</i> | <i>12-3</i> |
| <i>PWM Review.....</i> | <i>12-4</i> |
| <i>General-Purpose Timers.....</i> | <i>12-7</i> |
| GP Timer Modes of Operation | 12-8 |
| GP Timer Registers | 12-10 |
| Asymmetric and Symmetric PWM via General Purpose Timer Compares..... | 12-13 |
| GP Timer Compare PWM Exercise | 12-14 |
| <i>Compare Units.....</i> | <i>12-15</i> |
| Compare Unit Registers..... | 12-16 |
| Hardware Dead-Band (Compare Units only) | 12-17 |
| <i>Capture Units.....</i> | <i>12-20</i> |
| Capture Units Registers | 12-22 |
| <i>Quadrature Encoder Pulse (QEP).....</i> | <i>12-24</i> |
| QEP Initialization with GP Timer 2 (EVA)..... | 12-25 |
| <i>Lab 12: Event Manager</i> | <i>12-26</i> |
| <i>Review.....</i> | <i>12-33</i> |
| Solutions | 12-33 |

Event Manager



The Event Manager (EVA and EVB) consist of the following blocks:

General-Purpose Timers

Full Compare Units

Capture Units

Quadrature Encoder Pulse (QEP) circuit

Each block will be discussed in detail in this module. First, pulse width modulation (PWM) will be reviewed.

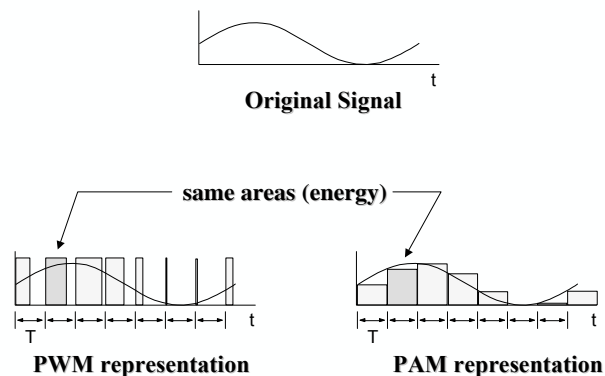
PWM Review

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 \approx original signal energy
- ◆ **Differs from PAM (Pulse Amplitude Mod.)**
 - fixed width, variable amplitude

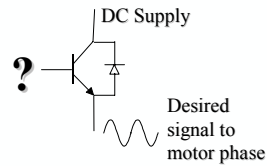
Pulse width modulation (PWM) is a method for representing an analog signal with a digital approximation. The PWM signal consists of a sequence of variable width, constant amplitude pulses which contain the same total energy as the original analog signal. This property is valuable in digital motor control 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.

PWM Signal Representation

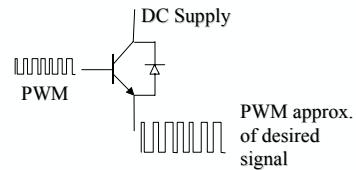


Why Use PWM in Digital Motor Control?

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

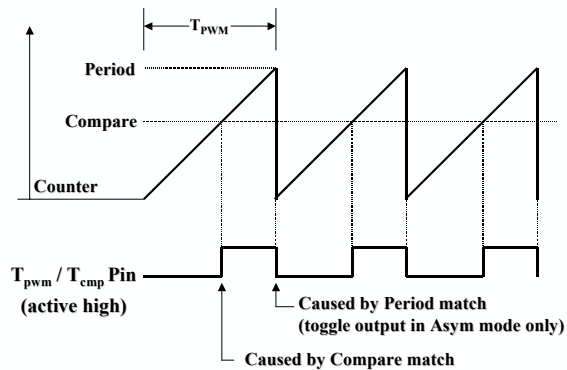


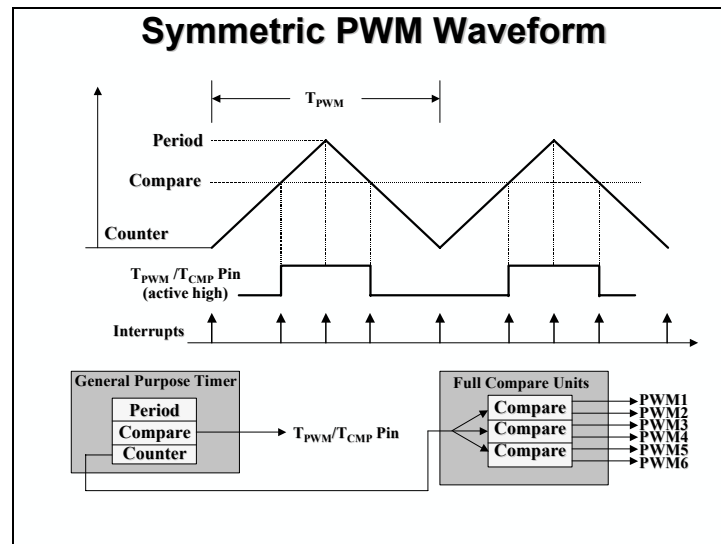
Unknown Gate Signal



Gate Signal Known with PWM

Asymmetric PWM Waveform





Each C240x event managers (EVA and EVB) are capable of generating five independent PWM signals. Up to two different carrier frequencies can be independently selected as follows:

4 PWM @ freq1, 1 PWM @ freq2

Of course, frequency 1 need not differ from frequency 2.

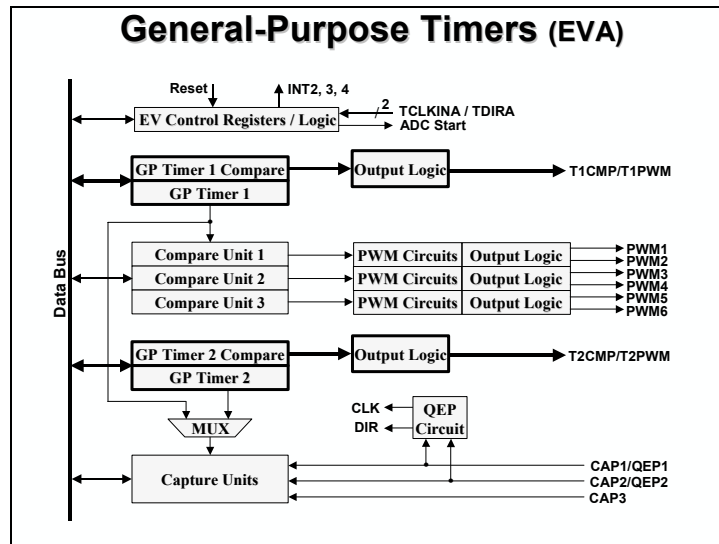
In addition, three additional PWM may be generated which are compliments of three of the above listed five (i.e. full compare units). This forms three complimentary pairs of PWM, and is intended for use as input to a three-phase power converter.

Two different compare types exist on each C240x event managers (EVA and EVB) for generating PWM signals:

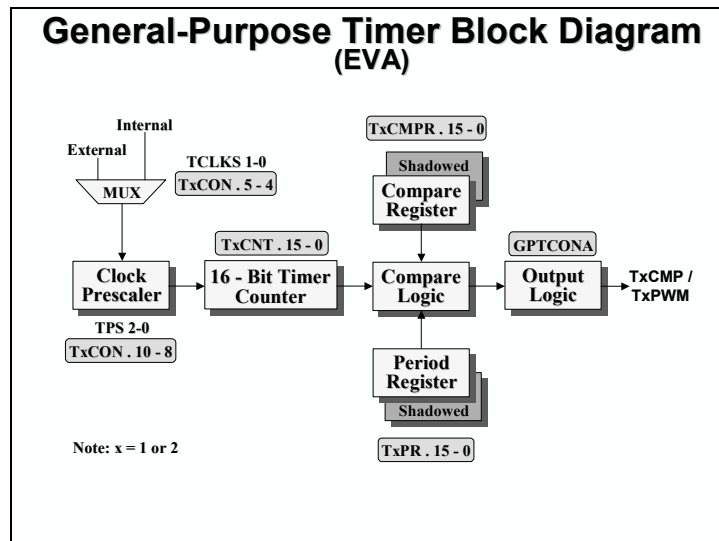
1. General Purpose Timer Compares
2. Compare Units

Each uses one of the general purpose timers for clocking the PWM calculation, and these timers will be covered next.

General-Purpose Timers



The GP Timers provide a time base for the operation of the compare units, and associated PWM circuits to generate PWM outputs. Additionally, they provide a time base for the operation of the quadrature encoder pulse (QEP) circuit (GP Timer 2 for EVA, and GP Timer 4 for EVB only) and the capture units. The GP Timers can also be used to generate a sampling period in a control system.



The TxPR period register holds the user specified counting period. TxPR is automatically loaded from the *period register buffer* on a counter underflow, which is defined as TxCNT=0. This allows for on-the-fly timer period changes. Note that the period register buffer is static in that if no change in the current period value is desired, one is not required to write the same value to the buffer on successive timer cycles.

The clocking signal for each GP Timer can be individually selected as either the internal CPU clock, or the external TCLKINA/B pin. In addition, the QEP outputs can be selected for clocking GP Timers. The external TDIRA/B pin is used to determine the counting direction only when the

timer is in the directional-up/down counting mode. The prescale counter is used to divide the clocking signal down to the desired frequency, when necessary.

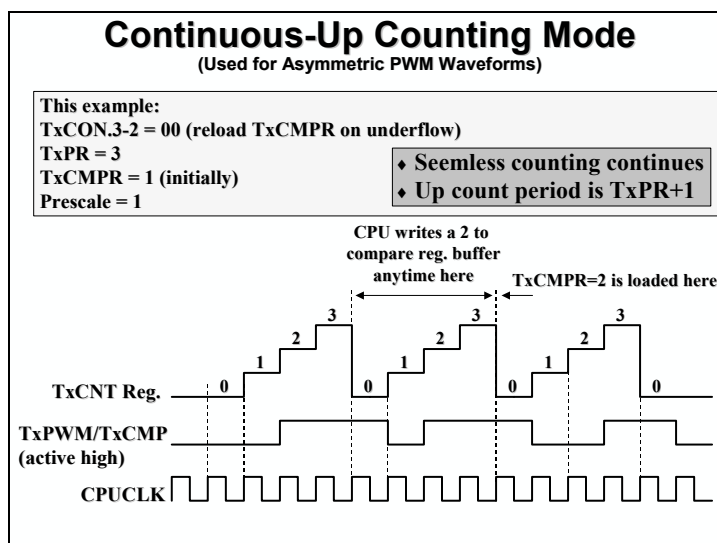
Each GP Timer has its own set of interrupts, all of which are individually maskable:

1. *TxPINT* - period match interrupt. Flag is set one and a half timer clock cycles after the timer counter matches the value in the timer period register.
2. *TxUFINT* - underflow interrupt. Flag is set one and a half timer clock cycles after the timer counter becomes zero.
3. *TxOFINT* - overflow interrupt. Flag is set one and a half timer clock cycles after the timer counter matches 0FFFFh.
4. *TxCINT* – compare match interrupt.

GP Timer Modes of Operation

The C240x event managers (EVA and EVB) each have two General Purpose Timers (GP Timers). Each timer has four different modes of operation. The most simple is the Stop/Hold mode, where the operation of the timer stops and holds at its current state. The timer counter, its compare output, and its prescale counter all remain unchanged in this mode. Two of the remaining three modes are commonly used in PWM generation, the exception being the directional up/down counting mode. This mode allows the use of external signals for both count clocking and direction, and is also employed for recording encoder counts from the on-chip QEP units.

Continuous-Up Counting



The procedure for GP Timer Continuous-Up Counting is as follows:

User sets bit 6 of TxCON register high to initiate counting

Counting begins on next rising clock edge

- 1st count is a “Zero” (no increment)

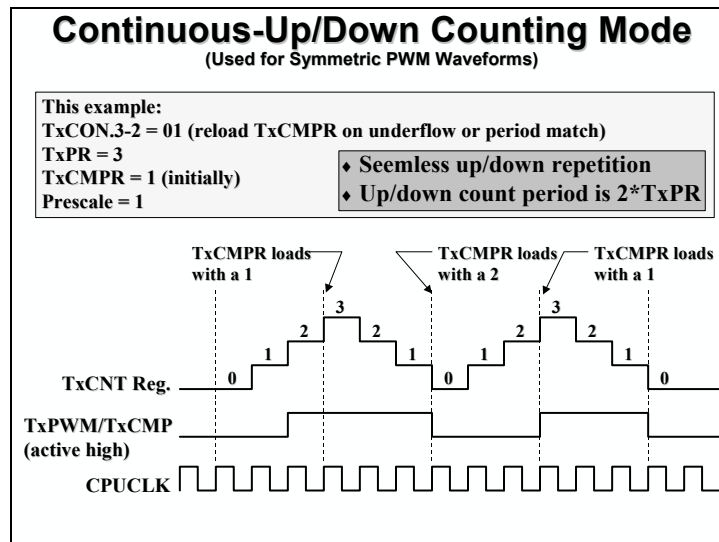
Count up until match with period register

If counter > period counts up to FFFFh

On next rising clock edge:

- Counter resets to zero
- Counting continues seamlessly

Continuous-Up/Down Counting



The procedure for GP Timer Continuous-Up/Down Counting is as follows:

User sets bit 6 of TxCON register high to initiate counting

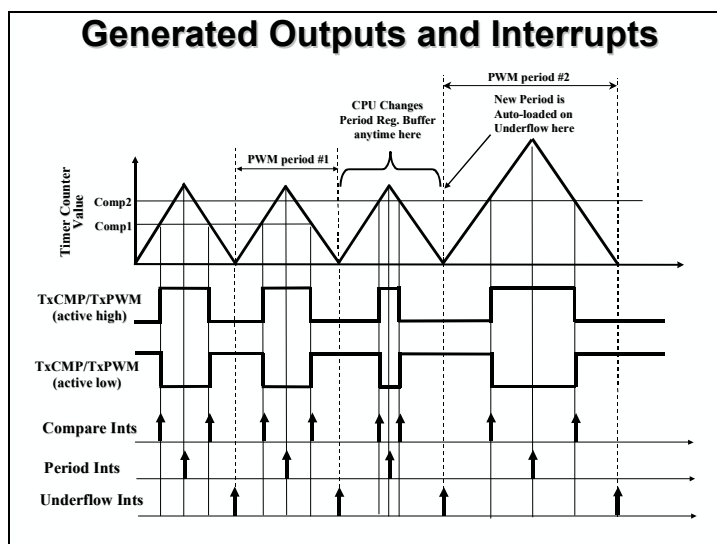
Counting begins on next rising clock edge

- 1st count is a “Zero” (no increment)

Count up until match with period register then backwards to zero

If counter > period counts up to FFFFh then resets to zero and start as if the initial counter value were zero

PWM Outputs and Interrupts



GP Timer Registers

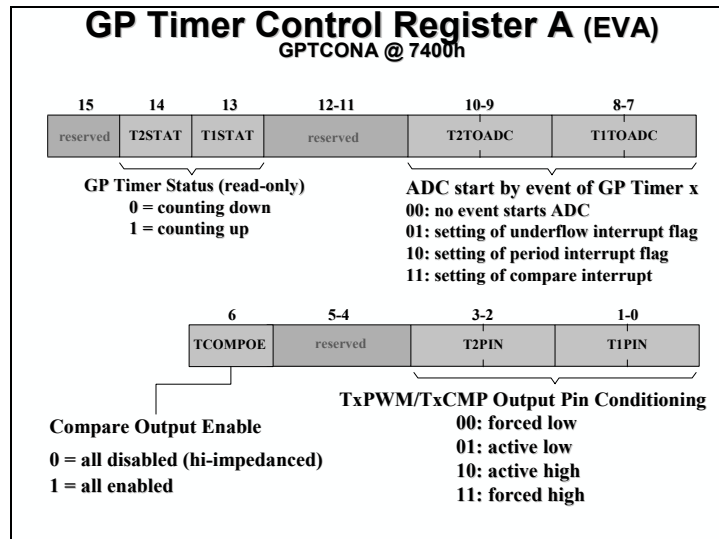
As was the case with the period register, buffering is present for each timer compare register. Software writes a value to the *compare register buffer*, from which the TxCMPCR register is automatically loaded on one of three user selected events:

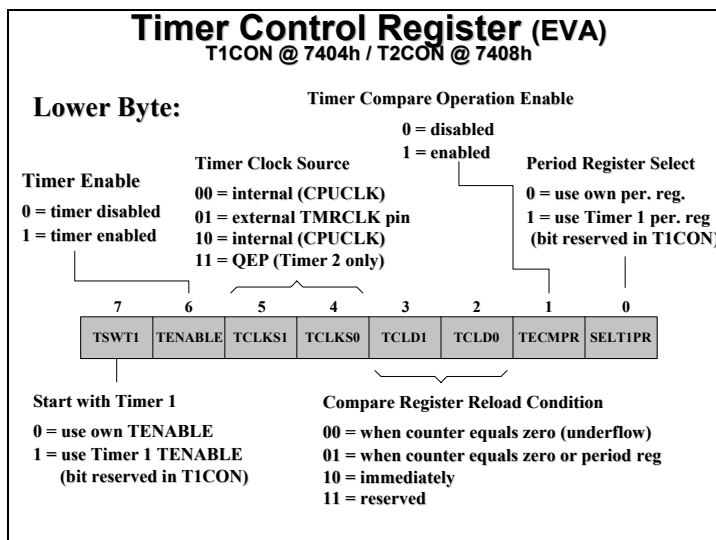
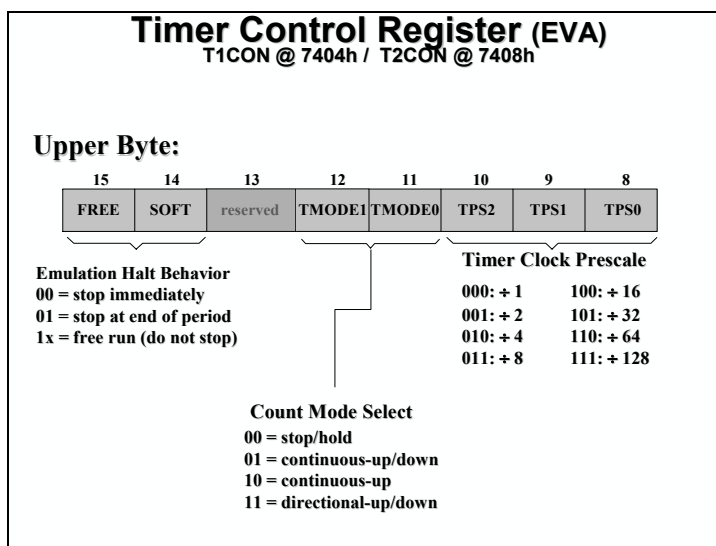
1. timer underflow (TxCNT = 0)
2. timer underflow or period match
3. immediately

The event selection is made using bits 2 and 3 of the TxCON register, and allows for on-the-fly compare value changes. Note that the compare register buffer is static in that if no change in the current compare value is desired, one is not required to write the same value to the buffer on successive timer cycles.

Each GP Timer unit has its own *Symmetric/Asymmetric PWM Waveform Generator*, which as its name implies, is capable of generating two types of PWM. The waveform generator uses the timer compare signal as an input, and outputs a PWM signal to the *Output Logic Unit*. The output logic lets the user select the polarity of the TTL signal on the TxPWM/TxCMP pin (e.g. active high or low) or alternately force the pin either high or low. The selection is made using bits 0-1, and 2-3 of the GPTCONA register for timers 1 and 2 respectively (EVA), and bits 0-1, and 2-3 of the GPTCONB register for timers 3 and 4 respectively (EVB).

| GP Timer Registers | | |
|--------------------|-------------|---------|
| | Register | Address |
| | Description | |
| EVA | GPTCONA | 7400h |
| | T1CNT | 7401h |
| | T1CMPR | 7402h |
| | T1PR | 7403h |
| | T1CON | 7404h |
| | T2CNT | 7405h |
| | T2CMPR | 7406h |
| | T2PR | 7407h |
| EVB | T2CON | 7408h |
| | GPTCONB | 7500h |
| | T3CNT | 7501h |
| | T3CMPR | 7502h |
| | T3PR | 7503h |
| | T3CON | 7504h |
| | T4CNT | 7505h |
| | T4CMPR | 7506h |
| | T4PR | 7507h |
| | T4CON | 7508h |





Asymmetric and Symmetric PWM via General Purpose Timer Compares

PWM switching frequency:

The PWM carrier frequency is determined by the value contained in the period register TxPR, and the frequency of the clocking signal. As an example, suppose it's desired to generate 20 kHz PWM (50 μ s) using a 40 MHz (25 ns) CPU clock to drive the GP Timer with the $\div 1$ prescale option. The value needed in the period register is then:

Asymmetric PWM: period register = $\left(\frac{\text{switching period}}{\text{timer period}} \right) - 1 = \frac{50 \mu\text{s}}{25 \text{ ns}} - 1 = 1999 = 7\text{CFh}$

Symmetric PWM: period register = $\frac{\text{switching period}}{2(\text{timer period})} = \frac{50 \mu\text{s}}{2(25 \text{ ns})} = 1000 = 3\text{E8h}$

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. For the above example:

Asymmetric PWM: approx. 11 bit resolution since $2^{11} = 2048 \approx 1999$

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

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. For the above example, suppose a 75% duty cycle is desired. The timer compare register should be loaded with the value as follows:

Asymmetric PWM: TxCMPR = $(100\% - \text{duty cycle}) * \text{TxPR} = 0.25 * 1999 = 499.75 \approx 1\text{F4h}$

Symmetric PWM: TxCMPR = $(100\% - \text{duty cycle}) * \frac{\text{TxPR}}{2} = 0.25 * \frac{1000}{2} = 125 = 7\text{Dh}$

Note that for symmetric PWM, the desired duty cycle is only achieved if the compare register TxCMPR contains the computed value for both the up-count compare and down-count compare portions of the timer period.

GP Timer Compare PWM Exercise

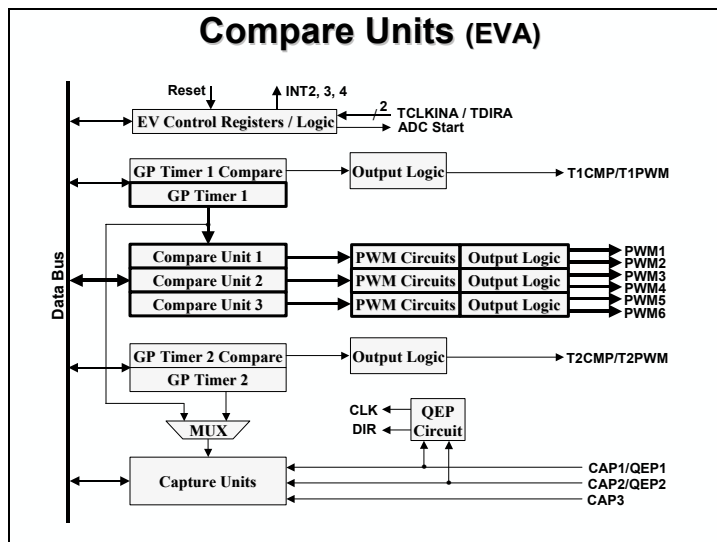
GP Timer Compare PWM Exercise

Symmetric PWM is to be generated as follows:

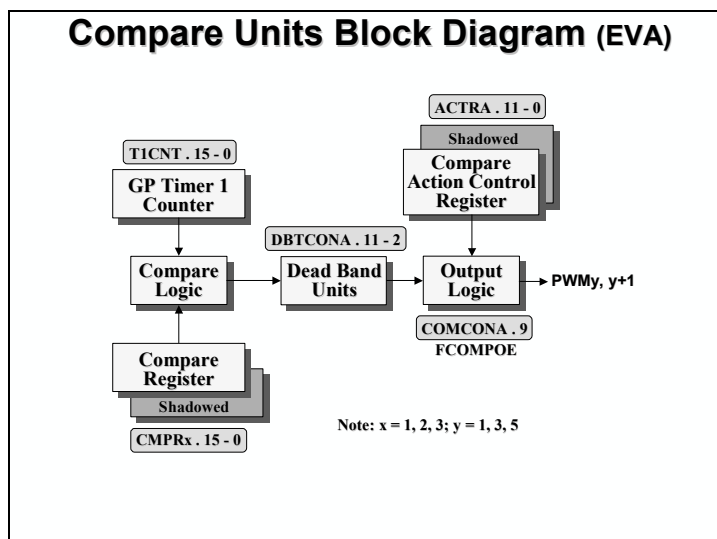
- 20 kHz carrier frequency
- Timer counter clocked by 25 ns CPU clock
- Use the ± 1 prescale option
- 25% duty cycle initially
- Use GP Timer Compare 1 with PWM output active high
- T2PWM/T2CMP pins forced low

1. Determine the initialization values needed in the GPTCONA, TICON, T1PR, and TICMPR registers
2. Write a code segment to initialize and start the PWM

Compare Units



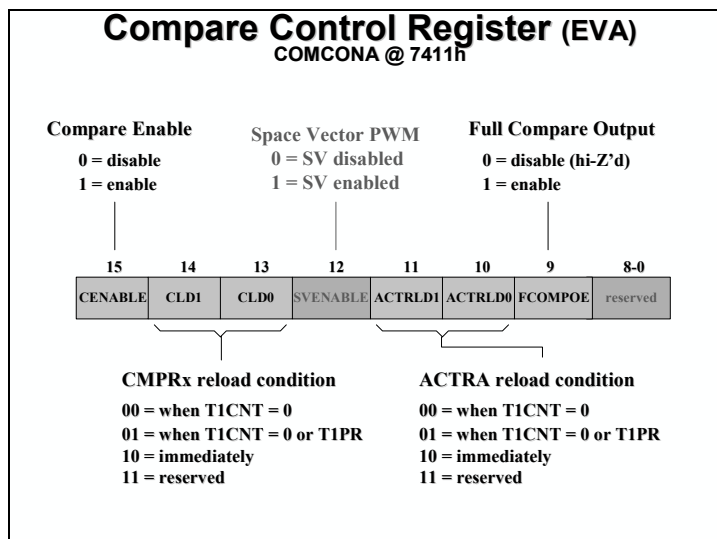
Each event manager (EVA and EVB) has three compare units. Each compare unit has two associated PWM outputs. They have capabilities beyond the GP timer compares, and feature programmable hardware deadband. The time base for the compare units is provided by GP timer 1 for EVA, and GP timer 3 for EVB.

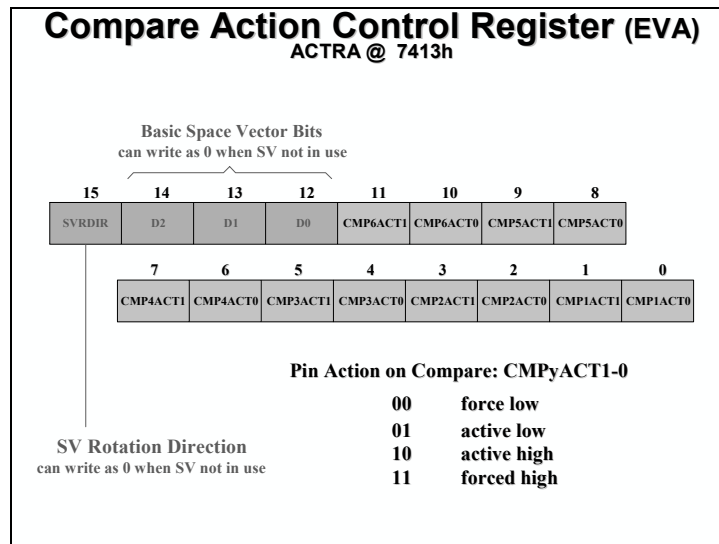


Compare Unit Registers

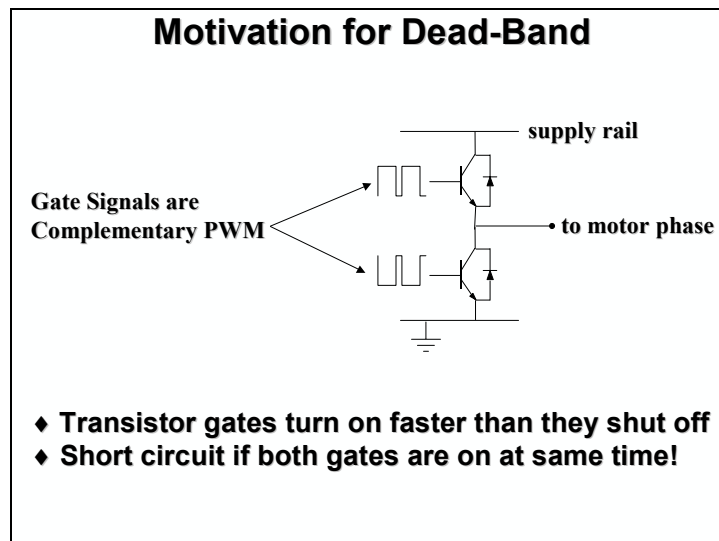
Compare Unit Registers

| | Register | Address | Description |
|-----|----------|---------|------------------------------------|
| EVA | COMCONA | 7411h | Compare Control Register A |
| | ACTRA | 7413h | Compare Action Control Register A |
| | DBTCONA | 7415h | Dead-Band Timer Control Register A |
| | CMPR1 | 7417h | Compare Register 1 |
| | CMPR2 | 7418h | Compare Register 2 |
| | CMPR3 | 7419h | Compare Register 3 |
| EVB | COMCONB | 7511h | Compare Control Register B |
| | ACTRB | 7513h | Compare Action Control Register B |
| | DBTCONB | 7515h | Dead-Band Timer Control Register B |
| | CMPR4 | 7517h | Compare Register 4 |
| | CMPR5 | 7518h | Compare Register 5 |
| | CMPR6 | 7519h | Compare Register 6 |

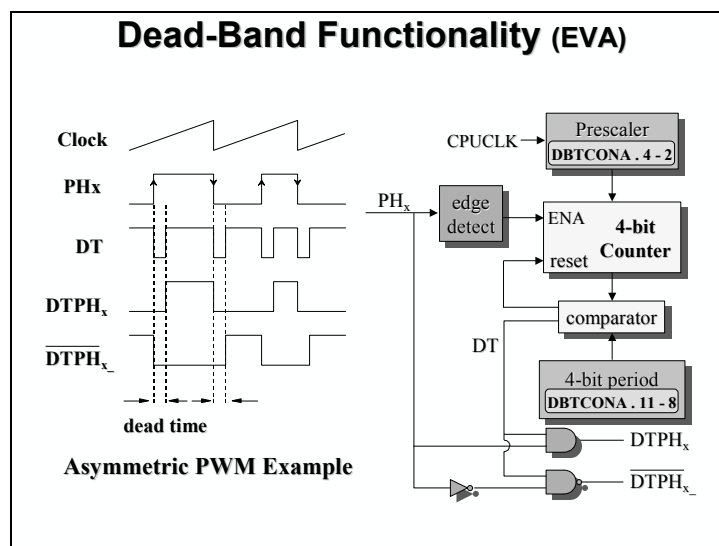




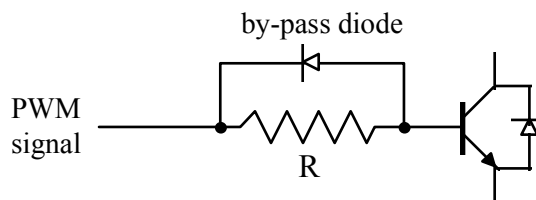
Hardware Dead-Band (Compare Units only)



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.



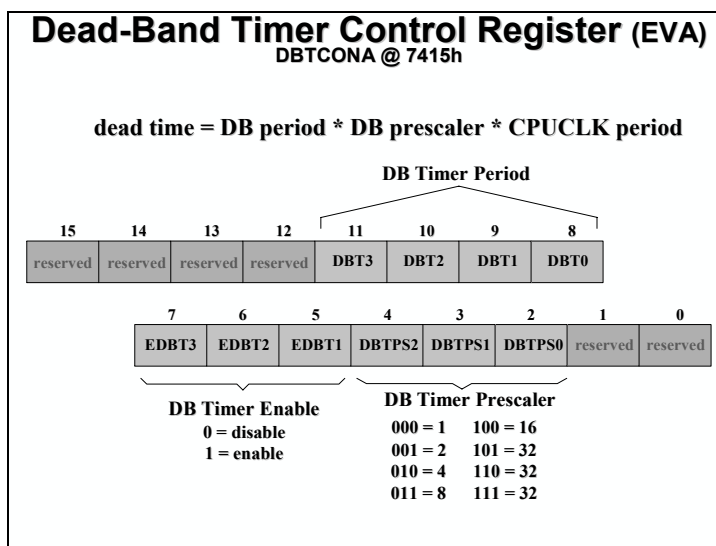
Two basic approaches exist for controlling shoot-through: modify the transistors, or modify the PWM gate signals controlling the transistors. In the first case, the opening time of the transistor gate must be increased so that it (slightly) exceeds the closing time. One way to accomplish this is by adding 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.

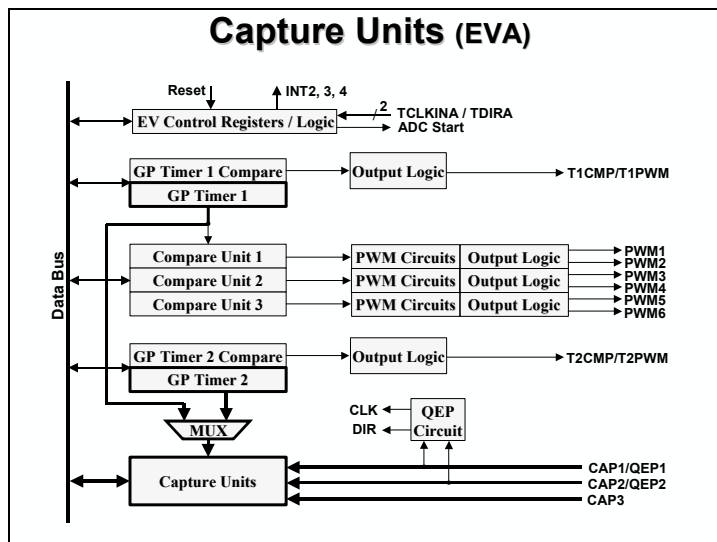
The second approach to shoot-through control separates transitions on complimentary PWM signals with a fixed period of time. This is called dead-band. While it is possible to perform software implementation of dead-band, the C240x offers on-chip hardware for this purpose that requires no additional CPU 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.



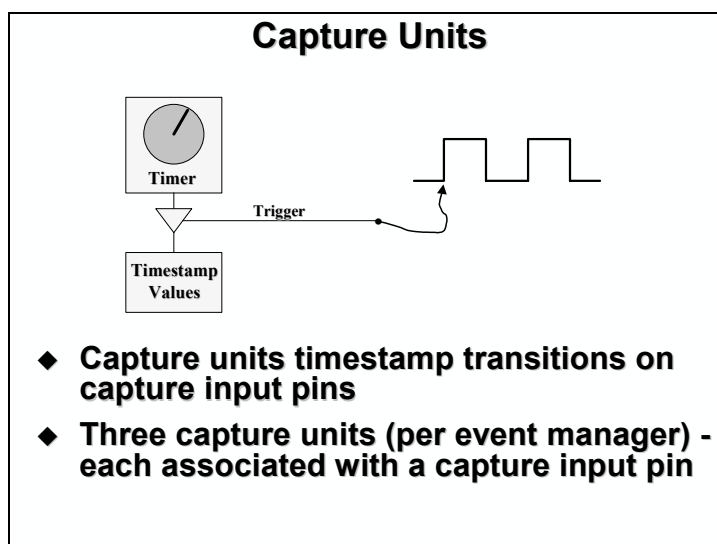
Each compare unit has its own dead-band timer, but shares the clock prescaler unit and the dead-band period with the other compare units. Dead-band can be individually enabled for each compare unit by setting bits 5, 6, and 7 in the DBTCONA register for EVA and DBTCONB for EVB.

The minimum achievable non-zero dead time is one CPU clock cycle (e.g. 50 ns), obtained by choosing the x/1 prescale option, and setting the DB period to 1 (i.e. DBTCONx.11-8 = 0001, where x is A for EVA and x is B for EVB).

Capture Units



Each event manager (EVA and EVB) has three capture units, and each is associated with a capture input pin. Each capture unit can choose GP timer 1 or 2 for EVA, and GP timer 3 or 4 for EVB as its time base. The value of GP timer 1 or 2 (EVA) and GP timer 3 or 4 (EVB) is captured and stored in the corresponding 2-level-deep FIFO stack when a specified transition is detected on a capture input pin.



The capture units allow time-based logging of external TTL signal transitions on the capture input pins. Each C240x event manager (EVA and EVB) has three capture units, two of which are shared with the quadrature encoder interface circuitry (discussed in next section).

Capture unit #3 on EVA and capture unit #6 on EVB 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, falling edges, or both. 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 Units

- ◆ Synchronized ADC start with capture event
- ◆ Measure the time width of a pulse
- ◆ Low speed velocity estimation from incr. encoder:

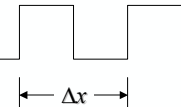
Problem: At low speeds, calculation of speed based on a measured position change at fixed time intervals produces large estimate errors

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

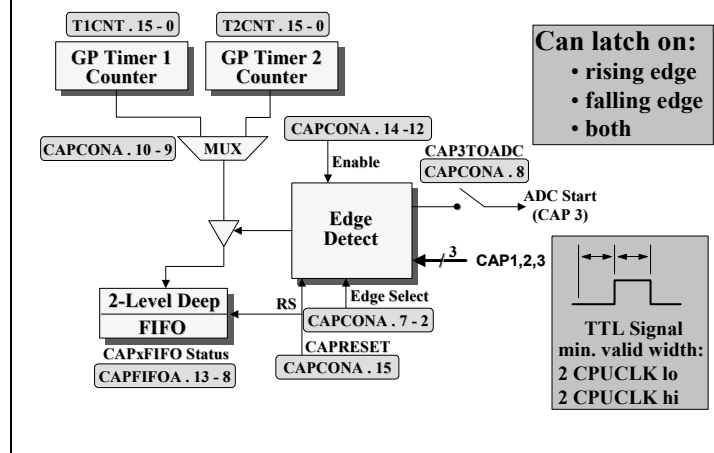
Alternative: Estimate the speed using a measured time interval at fixed position intervals

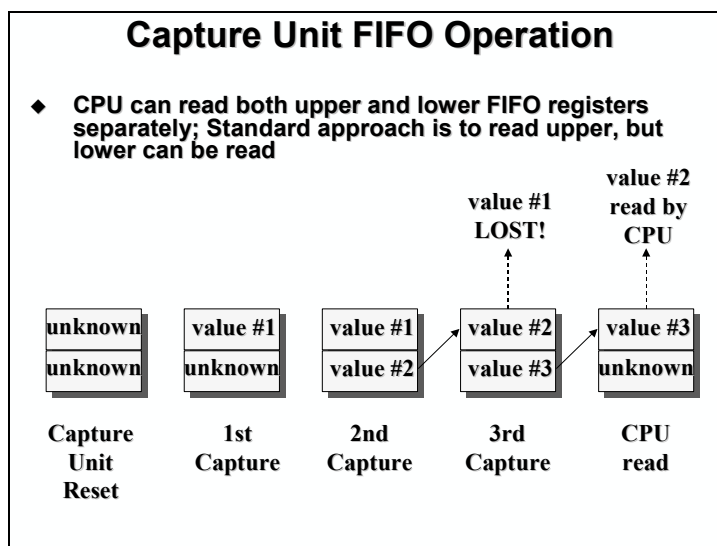
$$v_k \approx \frac{\Delta x}{t_k - t_{k-1}}$$

Signal from one quadrature encoder channel



Capture Units Block Diagram (EVA)





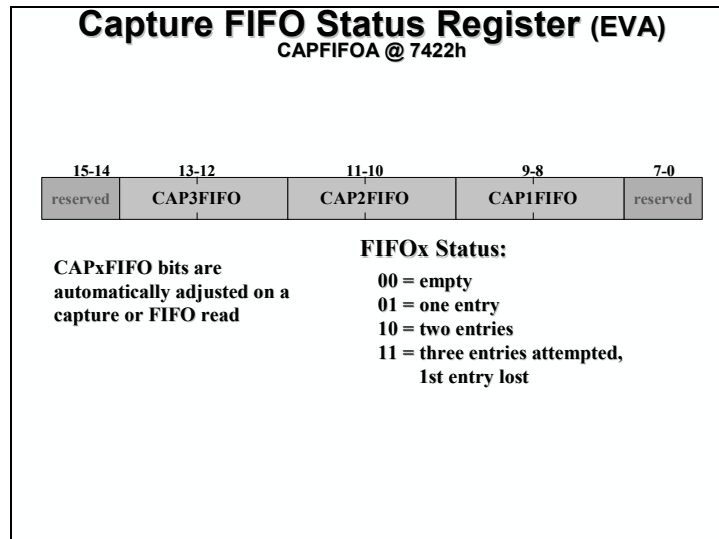
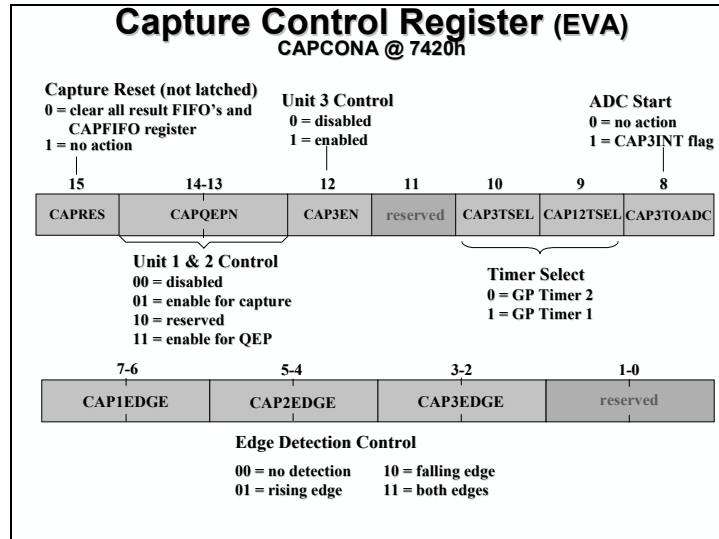
Capture Units Registers

Capture Units Registers

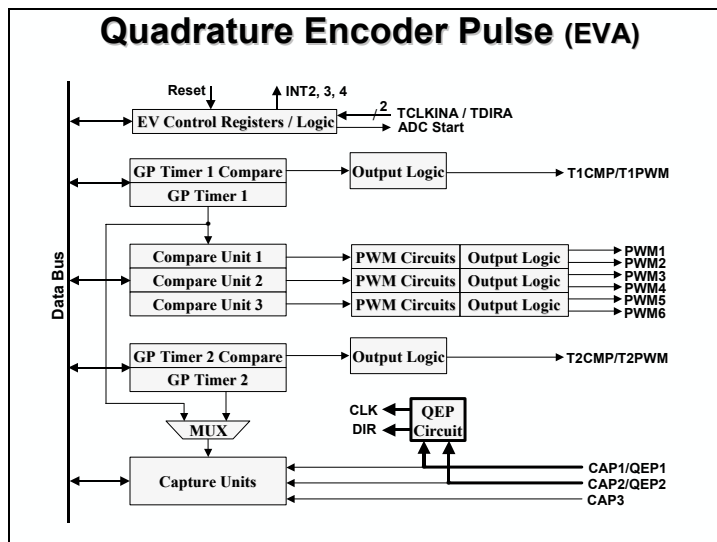
| | Register | Address | Description |
|-----|----------|---------|--------------------------------|
| EVA | CAPCONA | 7420h | Capture Control Register A |
| | CAPFIFOA | 7422h | Capture FIFO Status Register A |
| | CAP1FIFO | 7423h | Two-Level Deep FIFO 1 Stack |
| | CAP2FIFO | 7424h | Two-Level Deep FIFO 2 Stack |
| | CAP3FIFO | 7425h | Two-Level Deep FIFO 3 Stack |
| | CAP1FBOT | 7427h | Bottom Register of FIFO 1 |
| EVB | CAP2FBOT | 7428h | Bottom Register of FIFO 2 |
| | CAP3FBOT | 7429h | Bottom Register of FIFO 3 |
| | CAPCONB | 7520h | Capture Control Register B |
| | CAPFIFOB | 7522h | Capture FIFO Status Register B |
| | CAP4FIFO | 7523h | Two-Level Deep FIFO 4 Stack |
| | CAP5FIFO | 7524h | Two-Level Deep FIFO 5 Stack |
| | CAP6FIFO | 7525h | Two-Level Deep FIFO 6 Stack |
| | CAP4FBOT | 7527h | Bottom Register of FIFO 4 |
| | CAP5FBOT | 7528h | Bottom Register of FIFO 5 |
| | CAP6FBOT | 7529h | Bottom Register of FIFO 6 |

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

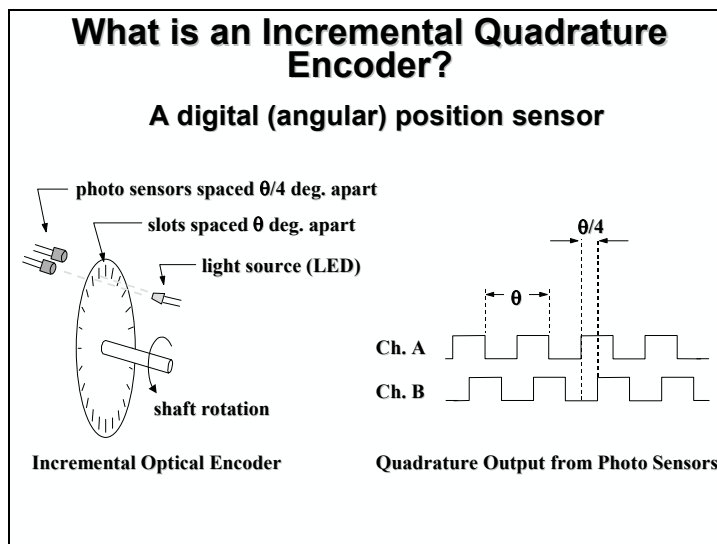
soon as the pulse ends, then the capture interrupt is the best option. However, the capture interrupt will occur after each of the two captures, the first of which will waste a small number of cycles while the CPU is interrupted and then determines that it is indeed only the first capture. If the width of the pulse is not needed as soon as the pulse ends, the CPU can check, as needed, the CAPFIFOA register for EVA and CAPFIFOB register for EVB to see if two captures have occurred, and proceed from there.

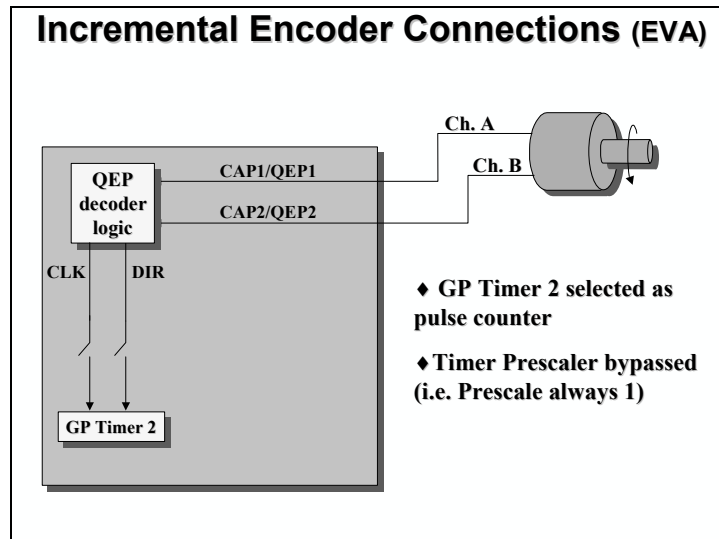
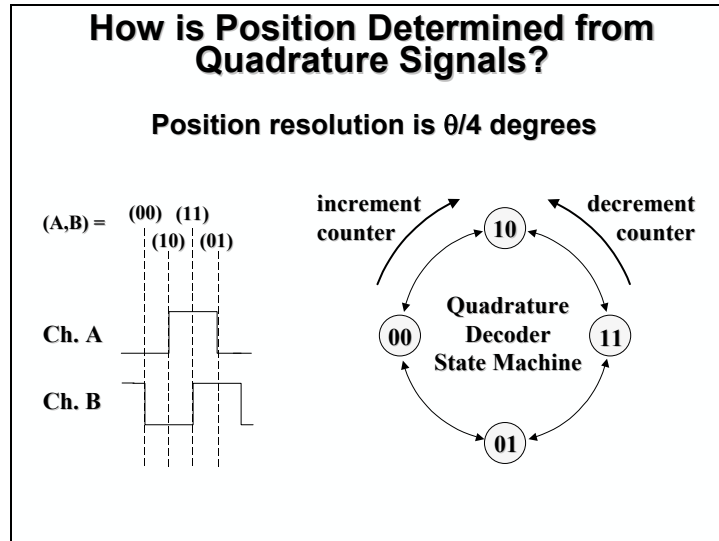


Quadrature Encoder Pulse (QEP)



The QEP circuit, when enabled, decodes and counts the quadrature encoded input pulses on pins CAP1/QEP1 and CAP2/QEP2 in EVA, and CAP4/QEP3 and CAP5/QEP4 in EVB. The QEP circuit can be used to interface with an optical encoder to get position and speed information from a rotating machine. When the QEP circuit is enabled, the capture function on CAP1 and CAP2 in EVA, and CAP4 and CAP5 in EVB pins is disabled. The QEP time base is provided by GP timer 2 in EVA and GP timer 4 in EVB.





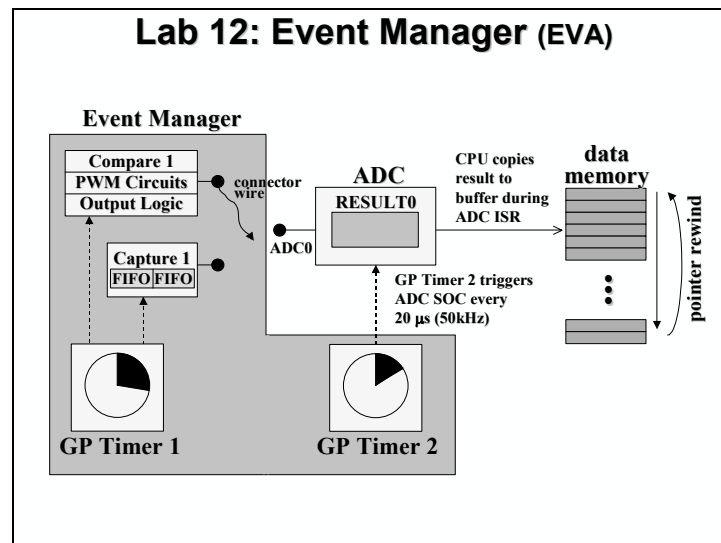
QEP Initialization with GP Timer 2 (EVA)

- Select Timer 2 for Capture number1 and 2
CAPCONA . 9 = 0 (for Timer 2)
- Optionally pre-load Timer 2 counter
TxCNT register
- Select directional-up/down mode for Timer 2
TxCON . 12 – 11 = 11
- Set T2PER register for desired timer rollover
e.g. set the number of ticks per revolution
- Enable QEP operation
CAPCONA . 14 – 13 = 11

Lab 12: Event Manager

➤ Objective

The objective of this lab is to apply the techniques discussed in Module 12 and to become familiar with the programming and operation of the Event Manager (EVA) and its interrupts. General-Purpose Timer 1 and Compare 1 will be setup to generate a 2kHz, 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. Next, Capture Unit 1 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.



➤ Procedure

Create Make File

1. **NOTE:** LAB12.ASM, VECS_12.ASM, CAP_ISR.ASM and LAB12.CMD files have been provided as a starting point for the lab and need to be completed. *DO NOT copy files from a previous lab.*
2. Create a new project called LAB12.MAK and add LAB12.ASM, VECS_12.ASM, CAP_ISR.ASM and LAB12.CMD to it. Check your file list to make sure all the files are there. (F2407.h will be added automatically during the Build). Be sure to setup the Build Options by clicking: Project → Options on the menu bar. Select the Assembler tab. In the middle of the screen check “Enable Source Level Debugging”. Next, select the Linker tab. In the middle of the screen select “No Autoinitialization”. Create a map file named LAB12.MAP. Select OK to save the Build Options.

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

3. Edit LAB12.ASM and adjust the shared I/O pins register MCRA for the PWM1 function. Next, setup the Compare 1 and General-Purpose Timer 1 to implement the PWM waveform

as previously described in the objective for this lab. The following registers need to be modified: T1CON, T1CNT, T1PR, DBTCNA (set deadband units off), CMPR1, ACTRA, COMCONA. (Hint – the last step should be to enable Timer 1). Notice that GPTCONA has been initialized earlier in the code during Timer 2 setup. Save your work.

Build and Load

- Click the “Rebuild All” 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.
- If the “Load program after build” option was not selected in Code Composer (“Option” menu, click on “Program Load...”) load the output file into the target. Click: File → Load Program...

Then reset the DSP by clicking on: Debug → Reset DSP

If you would like to debug using both source and assembly, right click on the VECTORS.ASM window and select Mixed Mode.

Testing Generation of PWM Waveform

- After loading, the VECS_12.ASM window should open with the yellow highlight on “B start”. Single-step your code one time into the LAB12.ASM file.
- Open a memory window to view some of the contents of the ADC results buffer. The address label for the ADC results buffer is *adc_buf*.
- Using the connector wire provided, connect the PWM1 (pin # P1-3) to ADCIN0 (pin # P2-23) on the EVM. **Exercise care when doing this as the power to the EVM is on, and we do not want to damage the EVM!** Then run the code, and halt it after a few seconds. Verify that the ADC results buffer contains updated values.
- 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 | adc_buf |
| 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.

10. The graphical display should show the generated 2kHz, 25% duty cycle symmetric PWM waveform. The period of a 2kHz signal is 500 μ s. 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

11. Code Composer 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 | adc_buf |
| Acquisition Buffer Size | 256 |
| FFT Framesize | 256 |
| DSP Data Type | 16-bit unsigned integer |
| Sampling Rate (Hz) | 50000 |

Select **OK** to save the graph options.

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

Setup Capture Unit 1 to Measure Width of Pulse

13. Edit `LAB12.ASM` and adjust the shared I/O pins register MCRA for the CAP1 function. Next, setup Capture Unit 1 to use Timer 1 as the time base and detect both rising and falling edges. (Hint – when configuring the CAPCONA register, set bit 15 to 0). Then modify the core interrupts (IMR) and event manager interrupts (EVAIMRC) to enable a CAP1 interrupt. Save your work.
14. The file `CAP_ISR.ASM` contains the interrupt service routine capture unit 1, and has been provided to you. Examine and understand the code in `CAP_ISR.ASM`. Notice that the label at the beginning of the code is `"cap_isr"`. Modify the interrupt vector table in `VECS_12.ASM` as needed. Make sure that the labels are visible to the linker in `VECS_12.ASM`. Save your work.

Build and Load

15. 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.
16. If the "Load program after build" option was not selected in Code Composer ("Option" menu, click on "Program Load...") load the output file into the target. Click: File → Load Program...

Then reset the DSP by clicking on: Debug → Reset DSP

If you would like to debug using both source and assembly, right click on the `VECTORS.ASM` window and select Mixed Mode.

Testing Pulse Width Measurement

17. After loading, the `VECS_12.ASM` window should open with the yellow highlight on "B start". Single-step your code one time into the `LAB12.ASM` file.
18. Open a memory window to view the address label `cap_rising`.
19. Using the connector wire provided, connect the PWM1 (pin # P1-3) to CAP1 (pin # P1-21) on the EVM. **Exercise care when doing this as the power to the EVM is on, and we do not want to damage the EVM!** Then run the code, and halt it after a few seconds. Notice the values for `cap_rising` and `cap_falling` (ignore the `cap_duty` value until step 20).

Questions:

- Which GP Timer is being used to clock the PWM1?
- Which GP Timer is being used as the Capture 1 timebase?
- How do the captured values for `cap_rising` and `cap_falling` relate to the compare register setting for PWM1?
- How can the differences be accounted for?

Modify Program for Duty Cycle Measurement

20. In order to accurately compute the duty cycle of the PWM signal, we need to configure Capture Unit 1 to use GP Timer 2 as its timebase, rather than GP Timer 1. GP Timer 1 can not be used for this, since it is being run in continuous up/down-counting mode, whereas GP Timer 2 is running in continuous up-counting mode. (Think of Timer 1 as a clock that runs from 12PM to 6AM, and that runs backwards to 12PM again. It wouldn't be a good clock to use for measuring the length of an event if you simply wanted to subtract the starting time from the ending time!)

Edit `LAB12.ASM` as required so that Capture 1 uses GP Timer 2 as its timebase.

21. We also need increase the period of Timer 2. The width of the active portion of the 25% duty 2kHz PWM is 125 μ sec. However, in `LAB12.ASM`, the line `"adc_rate .set 599"` gives Timer 2 a period of 600 CPUCLK cycles (equivalent to 50kHz, or a 20 μ sec period). It would be difficult to use Timer 2 to measure the duty (To understand why, here is an

analogy. Suppose someone hands you a stopwatch that can record only 1 minute of elapsed time before rolling over, but asks you to use this watch to measure the length of a 1 hour meeting. You would need to manually keep track of how many times the stopwatch had rolled over at the 1 minute mark, which would be inefficient. It would be better to use a watch that could record up to an hour!). It turns out that the easiest thing to do on a 16-bit microprocessor is to use a timer that rolls over at 16-bits. Then, all we need to do is subtract the start time from the end time using 2's complement math, and we get the elapsed time. By rolling over at 16-bits, we do not care if the elapsed time includes a timer rollover (as long as only one such rollover has occurred).

Edit LAB12.ASM and comment out the line `"adc_rate .set 599"` and uncomment the line `"adc_rate .set 0ffffh"`.

Note: This will have the side effect of making the ADC sampling rate only 458Hz. However, we are done using the ADC in this lab, so it is not a concern.

Build and Load

22. 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.
23. If the "Load program after build" option was not selected in Code Composer ("Option" menu, click on "Program Load...") load the output file into the target. Click: File → Load Program...

Then reset the DSP by clicking on: Debug → Reset DSP

Testing Duty Cycle Measurement

24. After loading, the VECS_12.ASM window should open with the yellow highlight on "B start". Single-step your code one time into the LAB12.ASM file.
25. Be sure that the memory window is open to view the address label `cap_duty`.
26. With the wire still connecting the PWM1 (pin # P1-3) to CAP1 (pin # P1-21) on the EVM, run the code again, and halt it after a few seconds. Observe the values for `cap_rising`, `cap_falling`, and `cap_duty` in a memory window. Notice that `cap_duty` is simply the difference between `cap_rising` and `cap_falling` using 16-bit modulo signed math.

Questions:

- What is the value of `cap_duty` in memory?
- How does it compare with the expected value?

End of Exercise

Optional Exercise - Debugging an Illegal Address Problem

Recall that certain addresses in the C240x memory map are *Illegal*. If the DSP attempts to access an illegal address, an NMI interrupt is immediately triggered. This is a failsafe feature that allows the user to handle certain types of software and hardware failures in a controlled manner. During

code development and debug, it is not uncommon to make a coding error and trigger an illegal address (this most commonly happens when you use direct addressing and forget to set the LDP properly!). When this happens, it is desirable to be able to identify the offending code so that it can be corrected. However, you cannot simply step through your code and watch for the NMI to occur since all interrupts (including NMI) are disabled by the debugger in single-step mode. You must Run the code and let the NMI occur. But, once at the NMI routine, how do you figure out where you came from? In this section of the lab, you will learn how to identify the code causing an illegal address NMI.

1. In LAB12.ASM, the main loop currently consists of an endless loop with a single NOP instruction in it:

```
loop:    NOP
        B     loop           ;branch to loop
```

Replace this code with the following (be sure to include the 5 NOP instructions after the SPLK):

```
loop:    NOP
        LDP   #DP_PF1       ;set data page for 7000h - 70FFh
        SPLK  #0, 0000h     ;write to address 0x7000
        NOP
        NOP
        NOP
        NOP
        NOP
        B     loop           ;branch to loop
```

2. Observe in the F2407 datasheet that address 0x7000 in the data space is illegal. Therefore, the above code should trigger an illegal address. Examine VECS_12.ASM and observe that the NMI interrupt vector currently just traps the DSP in an endless loop.
3. Rebuild and load the program. Reset the DSP by clicking: Debug → Reset DSP. The VECS_12.ASM window should open with the yellow highlight on "B start".
4. Run the code. After a few seconds halt your code by using Shift <F5>, or the Halt button on the vertical toolbar. Where did your code stop? If things went as expected, your code should be trapped at the NMI interrupt vector: int18: B int18. This means an illegal address violation occurred, because the only other NMI source is the external NMI pin, and we are not using that pin!

We happen to know which instruction caused the illegal address violation. It was the SPLK statement that we added to the main loop. However, we do not in general deliberately write code to cause an illegal address violation, and therefore we usually will not know what code is erroneously causing the problem. In order to find out, recall that when an interrupt occurs, the DSP automatically places the return program address onto the hardware stack. We could write a

NMI interrupt service routine that uses a POP or POPD instruction to pop off the top entry in the stack. We could then use the debugger to view this value in a memory window. However, Code Composer allows us to view the top of the stack (TOS) in the CPU Register window. Let's view this in Code Composer.

5. Open the CPU window in Code Composer so you can see the contents of the TOS by clicking: View → CPU Registers → CPU Register. Notice the contents of the TOS. This is the return address placed on the hardware stack when the NMI interrupt occurred.
6. Scroll through the Disassembly Window to the address indicated by the TOS. You should find that this address corresponds to the forth NOP after the SPLK statement that causes the illegal address. The return address is not that of the NOP immediately following the SPLK because of the DSP pipeline. The SPLK causes the illegal address violation in the EXECUTE phase of the pipeline. However, the next 3 instructions are already in process in the pipeline (i.e. FETCH, DECODE, and READ pipeline phases), and they continue through to execute before the interrupt is taken. Hence, the forth NOP after the SPLK is the proper return address.

You have now learned how to debug an illegal address problem!

Review

Review

- ◆ Name the 4 GP Timer Counting Modes
- ◆ Which counting mode should be used for asymmetric PWM?
- ◆ Which counting mode should be used for symmetric PWM?
- ◆ What is the purpose of Dead-Band?
- ◆ What is the purpose of the QEP circuit?

Solutions

GP Timer Compare PWM Exercise Solution

GPTCONA = (xxx0000001000010)b = 0042h
 TICON = (xx00100001000010)b = 0842h } (all x's assigned a value of 0)

$$TIPR = \frac{1}{2} \cdot \frac{\text{carrier period}}{\text{timer period}} = \frac{1}{2} \cdot \frac{50 \mu\text{s}}{25 \text{ ns}} = 1000 = 3E8h$$

$$TICMPR = (100\% - \text{duty cycle}) * TIPR = 0.75 * 1000 = 750 = 2EEh$$

CODE SEGMENT (Direct Addressing):

```
LDP #GPTCONA>>7 ;data page for address starting at 7400h
SPLK #0, GPTCONA ;Two TxPWM/TxCMP Hi-Z'd by GPTCONA
SPLK #0, TICON ;Disable timer in TICON
SPLK #0, TICNT ;initialize TICNT to zero
SPLK #3E8h, TIPR ;initialize TIPR
SPLK #2EEh, TICMPR ;initialize TICMPR
SPLK #42h, GPTCONA ;initialize GPTCONA
SPLK #0842h, TICON ;initialize TICON, timer starts
```

```

; SOLUTION FILE FOR LAB12.ASM

                .def      start
                .def      adc_isr

                .include f2407.h                ;address definitions

adc_rate        .set      599                  ;50kHz sampling rate
;adc_rate       .set      0ffffh              ;use for capture computation
of pwm duty

adc_buf_len     .set      300                  ;ADC results buffer length
stk_len         .set      100                 ;stack length

pwm_half_per    .set      7500                ;period/2 for 2kHz symmetric PWM
pwm_duty        .set      5625               ;25% duty cycle

                .bss      temp,1              ;general purpose variable

adc_buf_ptr     .usect    "buffer",1          ;ptr to next free buff addr
adc_buf         .usect    "buffer",adc_buf_len ;reserve space for buffer
stk             .usect    "stack",stk_len     ;reserve space for stack

                .text

start:

;~~~~~
;Disable the watchdog
;~~~~~
                LDP        #DP_PF1            ;set data page

                SPLK        #11101000b, WDCR
* bit 7         1:         clear WD flag
* bit 6         1:         disable the dog
* bit 5-3       101:       must be written as 101
* bit 2-0       000:       WDCLK divider = 1

;~~~~~
;Setup the system control registers
;~~~~~
                LDP        #DP_PF1            ;set data page

                SPLK        #0000000011111101b, SCSR1
;                |
;                FEDCBA9876543210
* bit 15       0:         reserved
* bit 14       0:         CLKOUT = CPUCLK
* bit 13-12    00:        IDLE1 selected for low-power mode
* bit 11-9     000:       PLL x4 mode
* bit 8        0:         reserved
* bit 7        1:         1 = enable ADC module clock
* bit 6        1:         1 = enable SCI module clock
* bit 5        1:         1 = enable SPI module clock
* bit 4        1:         1 = enable CAN module clock
* bit 3        1:         1 = enable EVB module clock
* bit 2        1:         1 = enable EVA module clock
* bit 1        0:         reserved

```



```

* bit 0          1:          clear the ILLADR bit

          SPLK      #00000000000001111b, SCSR2
;              |||||
;              FEDCBA9876543210
* bit 15-6       0's:      reserved
* bit 5          0:          DO NOT clear the WD OVERRIDE bit
* bit 4          0:          XMIF_HI-Z, 0=normal mode, 1=Hi-Z'd
* bit 3          1:          1 = disable the BOOT ROM
* bit 2          1:          MP/MC*, 1 = Flash addresses mapped external
* bit 1-0        11:        11 = SARAM mapped to prog and data

;~~~~~
;Set wait states for external memory interface on LF2407 EVM
;~~~~~
          LDP        #temp          ;set data page

          SPLK      #0000000001000000b, temp
;              |||||
;              FEDCBA9876543210
* bit 15-11      0's:      reserved
* bit 10-9       00:        bus visibility off
* bit 8-6        001:       1 wait-state for I/O space
* bit 5-3        000:       0 wait-state for data space
* bit 2-0        000:       0 wait-state for program space

          OUT        temp, WSGR

;~~~~~
;Setup the software stack
;~~~~~
          LAR        AR1, #stk          ;AR1 is stack pointer
          MAR        *, AR1            ;ARP = AR1

;~~~~~
;Setup the core interrupts
;~~~~~
          LDP        #0h                ;set data page
          SPLK      #111111b, IFR      ;clear any pending interrupts
          SPLK      #001001b, IMR      ;enable desired interrupts

;~~~~~
;Setup shared I/O pins
;~~~~~
          LDP        #DP_PF2            ;set data page

          SPLK      #0000000001001000b, MCRA
*              |||||
*              FEDCBA9876543210
* bit 15         0:          0=IOPB7,      1=TCLKINA
* bit 14         0:          0=IOPB6,      1=TDIRA
* bit 13         0:          0=IOPB5,      1=T2PWM/T2CMP
* bit 12         0:          0=IOPB4,      1=T1PWM/T1CMP
* bit 11         0:          0=IOPB3,      1=PWM6
* bit 10         0:          0=IOPB2,      1=PWM5
* bit 9          0:          0=IOPB1,      1=PWM4
* bit 8          0:          0=IOPB0,      1=PWM3

```

```

* bit 7      0:      0=IOPA7,      1=PWM2
* bit 6      1:      0=IOPA6,      1=PWM1
* bit 5      0:      0=IOPA5,      1=CAP3
* bit 4      0:      0=IOPA4,      1=CAP2/QEP2
* bit 3      1:      0=IOPA3,      1=CAP1/QEP1
* bit 2      0:      0=IOPA2,      1=XINT1
* bit 1      0:      0=IOPA1,      1=SCIRXD
* bit 0      0:      0=IOPA0,      1=SCITXD

        SPLK      #1111111000000000b,MCRB
*
*          |||||
*          FEDCBA9876543210
* bit 15     1:      0=reserved,    1=TMS2 (always write as 1)
* bit 14     1:      0=reserved,    1=TMS (always write as 1)
* bit 13     1:      0=reserved,    1=TD0 (always write as 1)
* bit 12     1:      0=reserved,    1=TDI (always write as 1)
* bit 11     1:      0=reserved,    1=TCK (always write as 1)
* bit 10     1:      0=reserved,    1=EMU1 (always write as 1)
* bit 9      1:      0=reserved,    1=EMU0 (always write as 1)
* bit 8      0:      0=IOPD0,      1=XINT2/ADCSOC
* bit 7      0:      0=IOPC7,      1=CANRX
* bit 6      0:      0=IOPC6,      1=CANTX
* bit 5      0:      0=IOPC5,      1=SPISTE
* bit 4      0:      0=IOPC4,      1=SPICLK
* bit 3      0:      0=IOPC3,      1=SPISOMI
* bit 2      0:      0=IOPC2,      1=SPISIMO
* bit 1      0:      0=IOPC1,      1=BIO*
* bit 0      0:      0=IOPC0,      1=W/R*

        SPLK      #0000000000000000b,MCRB
*
*          |||||
*          FEDCBA9876543210
* bit 15     0:      reserved
* bit 14     0:      0=IOPF6,      1=IOPF6
* bit 13     0:      0=IOPF5,      1=TCLKINB
* bit 12     0:      0=IOPF4,      1=TDIRB
* bit 11     0:      0=IOPF3,      1=T4PWM/T4CMP
* bit 10     0:      0=IOPF2,      1=T3PWM/T3CMP
* bit 9      0:      0=IOPF1,      1=CAP6
* bit 8      0:      0=IOPF0,      1=CAP5/QEP4
* bit 7      0:      0=IOPE7,      1=CAP4/QEP3
* bit 6      0:      0=IOPE6,      1=PWM12
* bit 5      0:      0=IOPE5,      1=PWM11
* bit 4      0:      0=IOPE4,      1=PWM10
* bit 3      0:      0=IOPE3,      1=PWM9
* bit 2      0:      0=IOPE2,      1=PWM8
* bit 1      0:      0=IOPE1,      1=PWM7
* bit 0      0:      0=IOPE0,      1=CLKOUT

;~~~~~
;Setup IOPA2 pin for use as output
;~~~~~
        LDP      #DP_PF2          ;set data page
        LACC     PADATDIR         ;ACC = PADATDIR
        OR       #0400h          ;IOPA2 is output
        SACL     PADATDIR         ;write back to GPIO port register

```

```

;~~~~~
;Setup the ADC
;~~~~~
        LDP      #DP_PF2      ;set data page

        SPLK     #0100000000000000b, ADCTRL1
*          |||||
*          FEDCBA9876543210
* bit 14      1:      1 = reset ADC module

        SPLK     #0000000000000000b, MAX_CONV
*          |||||
*          FEDCBA9876543210
* bit 15-7    0's:    reserved
* bit 6-4     000:    MAX_CONV2 value
* bit 3-0     0000:   MAX_CONV1 value (0 means 1 conversion)

        SPLK     #0000000000000000b, CHSELSEQ1
*          |||||
*          FEDCBA9876543210
* bit 15-12   0000:   CONV03 channel
* bit 11-8    0000:   CONV02 channel
* bit 7-4     0000:   CONV01 channel
* bit 3-0     0000:   CONV00 channel (only active conversion)

        SPLK     #0010000000010000b, ADCTRL1
*          |||||
*          FEDCBA9876543210
* bit 15      0:      reserved
* bit 14      0:      RESET, 0=no action, 1=reset ADC
* bit 13-12   10:     SOFT and FREE, 10=stop after current conversion
* bit 11-8    0000:   ACQ_Prescaler, 0000 = 1 x Tclk
* bit 7       0:      CPS, 0: Fclk=CPUCLK/1, 1: Fclk=CPUCLK/2
* bit 6       0:      CONT_RUN, 0=start/stop mode, 1=continuous run
* bit 5       0:      0=hi priority int, 1=low priority int
* bit 4       1:      0=dual sequencer, 1=cascaded sequencer
* bit 3       0:      0=calibration mode disabled
* bit 2       0:      BRG_ENA, used in calibration mode only
* bit 1       0:      HI/LO, no effect in normal operation mode
* bit 0       0:      0=self-test mode disabled

        SPLK     #0100011100000010b, ADCTRL2
*          |||||
*          FEDCBA9876543210
* bit 15      0:      EVB_SOC_SEQ, 0=no action
* bit 14      1:      RST_SEQ1/STRT_CAL, 0=no action
* bit 13      0:      SOC_SEQ1, 0=clear any pending SOCs
* bit 12      0:      SEQ1_BSY, read-only
* bit 11-10   01:     INT_ENA_SEQ1, 01=int on every SEQ1 conv
* bit 9       1:      INT_FLAG_SEQ1, write 1 to clear
* bit 8       1:      EVA_SOC_SEQ1, 1=SEQ1 start from EVA
* bit 7       0:      EXT_SOC_SEQ1, 1=SEQ1 start from ADCSOC pin
* bit 6       0:      RST_SEQ2, 0=no action
* bit 5       0:      SOC_SEQ2, no effect in cascaded mode
* bit 4       0:      SEQ2_BSY, read-only
* bit 3-2     00:     INT_ENA_SEQ2, 00=int disabled
* bit 1       1:      INT_FLAG_SEQ2, write 1 to clear

```

```

* bit 0          0:          EVB_SOC_SEQ2, 1=SEQ2 started by EVB

;~~~~~
;Setup the buffer for the ADC results
;~~~~~
        LDP      #adc_buf_ptr      ;set data page
        LAR      AR0, #adc_buf      ;pointer to results buffer
        SAR      AR0, adc_buf_ptr   ;initialize adc_buf_ptr
        MAR      *, AR0             ;ARP = AR0
        LACC     #2407h             ;ACC=0x2407
        LDP      #temp
        SPLK     #adc_buf_len-1, temp
        RPT      temp               ;repeat #adc_buf_len-1 times
        SACL     *+                 ;initialize the buffer

;~~~~~
;Setup GP Timer2 to trigger an ADC conversion
;~~~~~
        LDP      #DP_EVA            ;set data page

        SPLK     #0000h, T2CNT      ;clear timer2 counter
        SPLK     #adc_rate, T2PR    ;set timer2 period

        SPLK     #0000010000000000b, GPTCONA ;init GPTCON register
*
*          |||||
*          FEDCBA9876543210
* bit 15      0:          reserved
* bit 14      0:          T2STAT - read only
* bit 13      0:          T1STAT - read only
* bit 12-11   00:        reserved
* bit 10-9    10:        T2TOADC, 10 = ADCSOC on period match
* bit 8-7     00:        T1TOADC, 00 = no ADCSOC
* bit 6       0:          1 = enable all timer compare outputs
* bit 5-4     00:        reserved
* bit 3-2     00:        00 = T2PIN forced low
* bit 1-0     00:        00 = T1PIN forced low

        SPLK     #0001000001000000b, T2CON ;init T2CON register
*
*          |||||
*          FEDCBA9876543210
* bit 15-14   00:        stop immediately on emulator suspend
* bit 13      0:          reserved
* bit 12-11   10:        10 = continous-up count mode
* bit 10-8     000:       000 = x/1 prescaler
* bit 7        0:          0 = use own TENABLE bit
* bit 6        1:          1 = enable timer
* bit 5-4     00:        00 = CPUCLK is clock source
* bit 3-2     00:        00 = reload compare reg on underflow
* bit 1        0:          0 = disable timer compare
* bit 0        0:          0 = use own period register

;~~~~~
;Setup Capture unit 1
;~~~~~
        LDP      #DP_EVA            ;set data page

```

```

        SPLK      #0010001011000000b, CAPCONA      ;init CAPCON register
*
*      |||||
*      FEDCBA9876543210
* bit 15      0:      0 = reset the capture units and registers
* bit 14-13   01:      01 = enable CAP1 and CAP2, QEP disabled
* bit 12      0:      0 = disable CAP3
* bit 11      0:      reserved
* bit 10      0:      0 = CAP3 uses timer2
* bit 9       1:      CAP1 and CAP2 use: 0=timer2, 1=timer1
* bit 8       0:      0 = CAP3 does not start ADC
* bit 7-6     11:      11 = CAP1 detects both rising and falling edges
* bit 5-4     00:      00 = CAP2 no detection
* bit 3-2     00:      00 = CAP3 no detection
* bit 1-0     00:      reserved

;~~~~~
;Setup Full Compare 1 and GP Timer 1 to generate PWM
;~~~~~
        LDP      #DP_EVA                      ;set data page
        SPLK     #0000h, T1CON                 ;disable timer 1
        SPLK     #0000h, T1CNT                 ;clear timer 1 counter
        SPLK     #pwm_half_per, T1PR          ;setup timer 1 period

        SPLK     #0000h, DBTCONA               ;deadband units off
        SPLK     #pwm_duty, CMPR1              ;set PWM1 duty cycle

        SPLK     #0000000000000010b, ACTRA    ;PWM1 set for active high
*
*      |||||
*      FEDCBA9876543210
* bit 15      0:      space vector dir is CCW (don't care)
* bit 14-12   000:     basic space vector is 000 (dont' care)
* bit 11-10   00:      PWM6/IOPB3 pin forced low
* bit 9-8     00:      PWM5/IOPB2 pin forced low
* bit 7-6     00:      PWM4/IOPB1 pin forced low
* bit 5-4     00:      PWM3/IOPB0 pin forced low
* bit 3-2     00:      PWM2/IOPA7 pin forced low
* bit 1-0     10:      PWM1/IOPA6 pin active high

        SPLK     #1000001000000000b, COMCONA  ;configure COMCON
*
*      |||||
*      FEDCBA9876543210
* bit 15      1:      1 = enable full compare operation
* bit 14-13   00:      00 = reload CMPRx regs on timer 1 underflow
* bit 12      0:      0 = space vector disabled
* bit 11-10   00:      00 = reload ACTR on timer 1 underflow
* bit 9       1:      1 = enable PWM pins
* bit 8-0     0's:     reserved

        SPLK     #0000100001000000b, T1CON    ;init T1CON register
*
*      |||||
*      FEDCBA9876543210
* bit 15-14   00:      stop immediately on emulator suspend
* bit 13      0:      reserved
* bit 12-11   01:      01 = continous-up/down count mode
* bit 10-8     000:     000 = x/1 prescaler
* bit 7       0:      0 = use own TENABLE bit
* bit 6       1:      1 = enable timer

```

```

* bit 5-4      00:      00 = CPUCLK is clock source
* bit 3-2      00:      00 = reload compare reg on underflow
* bit 1        0:       0 = disable timer compare
* bit 0        0:       0 = use own period register

;~~~~~
;Setup the event manager interrupts
;~~~~~
        LDP      #DP_EVA          ;set data page
        SPLK     #0FFFFh, EVAIFRA ;clear all EVA group A interrupts
        SPLK     #0FFFFh, EVAIFRB ;clear all EVA group B interrupts
        SPLK     #0FFFFh, EVAIFRC ;clear all EVA group C interrupts
        SPLK     #00000h, EVAIMRA ;enabled desired EVA group A interrupts
        SPLK     #00000h, EVAIMRB ;enabled desired EVA group B interrupts
        SPLK     #00001h, EVAIMRC ;enabled desired EVA group C interrupts

        LDP      #DP_EVB          ;set data page
        SPLK     #0FFFFh, EVBIFRA ;clear all EVB group A interrupts
        SPLK     #0FFFFh, EVBIFRB ;clear all EVB group B interrupts
        SPLK     #0FFFFh, EVBIFRC ;clear all EVB group C interrupts
        SPLK     #00000h, EVBIMRA ;enabled desired EVB group A interrupts
        SPLK     #00000h, EVBIMRB ;enabled desired EVB group B interrupts
        SPLK     #00000h, EVBIMRC ;enabled desired EVB group C interrupts

;~~~~~
;Enable global interrupts
;~~~~~
        CLRC     INTM              ;enable global interrupts

;~~~~~
;Main loop
;~~~~~
loop:    NOP
        B        loop              ;branch to loop

*****
*  G E N E R A L  I N T E R R U P T  S E R V I C E  R O U T I N E S  *
*****

;~~~~~
;ADC Interrupt Service Routine
;~~~~~
adc_isr:

;context save
        MAR      *,AR1              ;ARP=stack pointer
        MAR      *+                  ;skip one stack location
        SST      #1, *+              ;save ST1
        SST      #0, *+              ;save ST0
        SACH     *+                  ;save ACCH
        SACL     *+                  ;save ACCL
        SAR      AR2, *+             ;save AR2

;clear the INT_FLAG_SEQ1 and read the ADC result
        CLRC     SXM

```

```

        LDP      #DP_PF2                ;set data page
        LACC     ADCTRL2                ;read and write ADCTRL2
        SACL     ADCTRL2                ; to clear the INT_FLAG_SEQ1
        LACC     RESULT0,10             ;read ADC RESULT0

;store the data value to the buffer
        LDP      #adc_buf_ptr           ;set data page
        LAR      AR2, adc_buf_ptr       ;AR2 points to the buffer
        MAR      *, AR2                 ;set ARP
        SACH     *+                     ;store result
        SAR      AR2, adc_buf_ptr       ;store updated pointer

;brute-force the circular buffer
        LAR      AR0, #(adc_buf+adc_buf_len-1) ;AR0 points to last
buffer entry
        CMPR     2                      ;TC set if AR(ARP) > AR0
        BCND     adc_isr1, NTC          ;branch if TC not set
        SPLK     #adc_buf, adc_buf_ptr  ;re-init the pointer

adc_isr1:

;reset ADC SEQ1 to CONV00 state
        LDP      #DP_PF2                ;set data page
        LACC     ADCTRL2                ;read ADCTRL2
        OR       #4000h                 ;set bit 14 (RST_SEQ1/STRT_CAL bit)
        SACL     ADCTRL2                ;write back to reset SEQ1

;toggle the IOPA2 pin
        LDP      #DP_PF2                ;set data page
        LACC     PADATDIR               ;ACC = PADATDIR
        XOR      #0004h                 ;toggle IOPA2 bit
        SACL     PADATDIR               ;write back to GPIO port register

;context restore
        MAR      *, AR1                 ;ARP = AR1
        MAR      *-                     ;SP points to last entry
        LAR      AR2, *-                ;restore AR2
        LACL     *-                     ;restore ACCL
        ADD      *-,16                  ;restore ACCH
        LST      #0, *-                 ;restore ST0
        LST      #1, *-                 ;restore ST1, de-allocate skipped stack location
        CLRC     INTM                   ;re-enable global interrupts
        RET

```

```

;~~~~~
; Lab 12: Solution - Capture Unit Interrupt Service Routine
;~~~~~
        .def cap_isr
        .def cap_rising, cap_falling, cap_duty

        .include f2407.h                ;address definitions

        .bss cap_rising,3,1            ;capture rising edge timestamp
cap_falling .set cap_rising+1          ;captured falling edge timestamp
cap_duty    .set cap_rising+2          ;PWM duty cycle computed using cap-
tures

cap_isr:

;context save
        MAR    *,AR1                  ;ARP=stack pointer
        MAR    *+                    ;skip one stack location
        SST    #1, *+                ;save ST1
        SST    #0, *+                ;save ST0
        SACH    *+                    ;save ACCH
        SACL    *+                    ;save ACCL

;clear the CAP1INT flag in EVAIFRC
        LDP     #DP_EVA                ;set data page
        LACC    #001b                 ;ACC = 001b
        SACL    EVAIFRC                ;clear the CAP1INT flag

;read CAP1FIFO
        LACC    CAP1FIFO                ;read the top entry
        LDP     #cap_rising            ;set data page
        SACL    cap_rising             ;store to memory
        LDP     #DP_EVA                ;set data page
        LACC    CAP1FIFO                ;read the 2nd entry
        LDP     #cap_rising            ;set data page
        SACL    cap_falling            ;store to memory
        SETC    SXM                    ;do signed math
        SUB     cap_rising             ;compute the difference
        SACL    cap_duty               ;store to memory

;context restore
        MAR     *, AR1                 ;ARP = AR1
        MAR     *-                     ;SP points to last entry
        LACL    *-                     ;restore ACCL
        ADD     *-,16                  ;restore ACCH
        LST     #0, *-                 ;restore ST0
        LST     #1, *-                 ;restore ST1
        CLRC    INTM                  ;re-enable global interrupts
        RET

```



```

; SOLUTION FILE FOR VECS_12.ASM

        .ref      start
        .ref      adc_isr, cap_isr

        .sect     "vectors"

;~~~~~
;Interrupt vector table for core
;~~~~~

int1:      B      start           ;00h reset
int2:      B      adc_isr        ;02h INT1
int3:      B      int2           ;04h INT2
int4:      B      int3           ;06h INT3
int5:      B      cap_isr       ;08h INT4
int6:      B      int5           ;0Ah INT5
int7:      B      int6           ;0Ch INT6
int8:      B      int7           ;0Eh reserved
int9:      B      int8           ;10h INT8 user-defined
int10:     B      int9           ;12h INT9 user-defined
int11:     B      int10          ;14h INT10 user defined
int12:     B      int11          ;16h INT11 user defined
int13:     B      int12          ;18h INT12 user defined
int14:     B      int13          ;1Ah INT13 user defined
int15:     B      int14          ;1Ch INT14 user defined
int16:     B      int15          ;1Eh INT15 user defined
int17:     B      int16          ;20h INT16 user defined
int18:     B      int17          ;22h TRAP
int19:     B      int18          ;24h NMI
int20:     B      int19          ;26h reserved
int21:     B      int20          ;28h INT20 user defined
int22:     B      int21          ;2Ah INT21 user defined
int23:     B      int22          ;2Ch INT22 user defined
int24:     B      int23          ;2Eh INT23 user defined
int25:     B      int24          ;30h INT24 user defined
int26:     B      int25          ;32h INT25 user defined
int27:     B      int26          ;34h INT26 user defined
int28:     B      int27          ;36h INT27 user defined
int29:     B      int28          ;38h INT28 user defined
int30:     B      int29          ;3Ah INT29 user defined
int31:     B      int30          ;3Ch INT30 user defined
int31:     B      int31          ;3Eh INT31 user defined

```

This page is left intentionally blank.

Introduction

This module describes the features that allow the TMS320C240x to interface to external devices. Additionally, Flash memory, code security and JTAG emulation considerations will be covered.

Learning Objectives

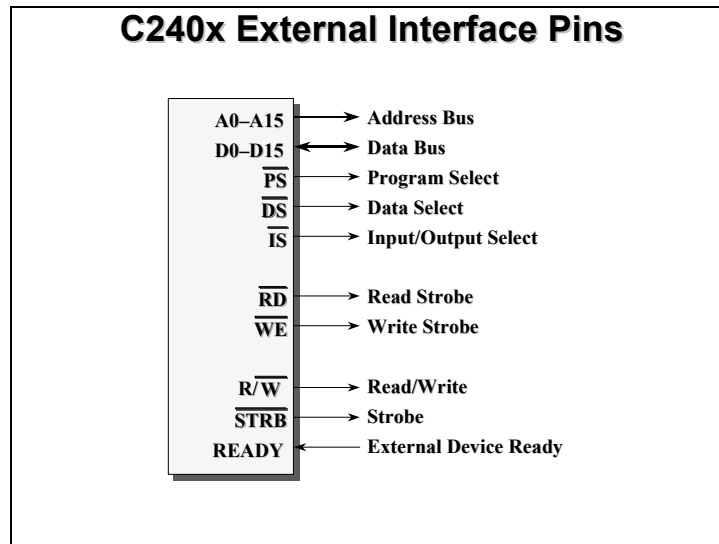
Learning Objectives

- ◆ Describe the purpose of interface pins
- ◆ Identify the key timing for external reads and writes
- ◆ Show connections of the C240x to various memory and peripheral devices
- ◆ Explain programming Flash memory
- ◆ Considerations for JTAG emulation

Module Topics

| | |
|--|--------------|
| System Design | 13-1 |
| <i>Module Topics.....</i> | <i>13-2</i> |
| <i>Interface Pins and Descriptions.....</i> | <i>13-3</i> |
| External Interface Example | 13-4 |
| <i>External Read Timing</i> | <i>13-5</i> |
| Single-Cycle Reads..... | 13-5 |
| Long Memory Access Times..... | 13-5 |
| Read Strobe | 13-6 |
| <i>External Write Timing Issues</i> | <i>13-7</i> |
| External Write Timing..... | 13-7 |
| Minimizing the Write “Slow Down” | 13-8 |
| <i>I/O Memory Space</i> | <i>13-9</i> |
| Parallel I/O..... | 13-9 |
| <i>Memory Timing Summary.....</i> | <i>13-10</i> |
| <i>JTAG Scan-Based Emulation.....</i> | <i>13-11</i> |
| <i>Flash Programming.....</i> | <i>13-13</i> |
| Flash Programming Problem Checklist | 13-16 |
| Code Security Module (CSM) – frequently asked questions..... | 13-18 |
| <i>Review.....</i> | <i>13-21</i> |

Interface Pins and Descriptions



The majority of the interface pins are composed of the data and address buses.

- 16 data lines for 16-bit words allows 96 dB of dynamic range (6 dB/bit)
- 16 address lines provide 64K address range

To specify whether data, program, or I/O will use the external multiplexed data and address buses, one of three control lines is driven low by the C240x.

- Program Fetch — \overline{PS} is driven low
- Data Access — \overline{DS} is driven low
- I/O operation — \overline{IS} is driven low.

\overline{PS} , \overline{DS} , and \overline{IS} may be used for external bank select or chip select.

The \overline{RD} (Read Enable) line is driven active (low) when an external read is required. This signal is active for all external program, data, and I/O reads, and may be connected directly to the \overline{OE} of an external device. This signal is valid for all external program, data, and I/O reads.

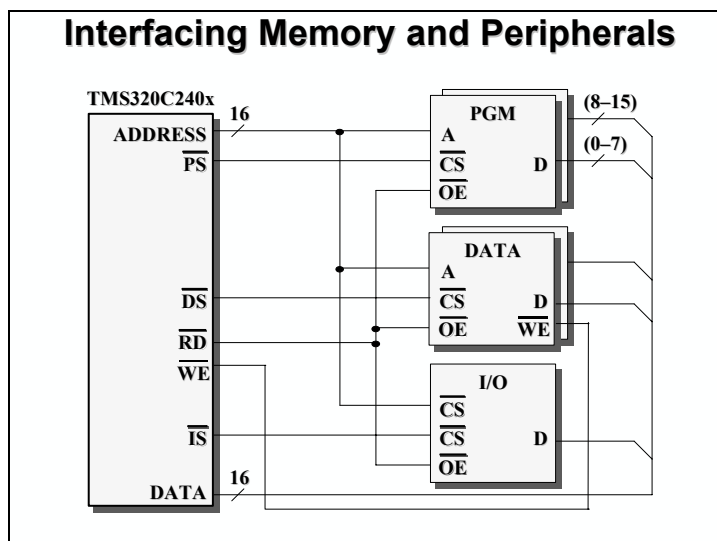
The falling edge of the \overline{WE} (Write Enable) line indicates valid data on the external data bus. The rising edge of \overline{WE} may be used by an external device to latch in valid data. This signal may be connected directly to the \overline{WE} line of external devices. This signal is valid for all external program, data, and I/O writes.

The \overline{STRB} line is used to indicate an active external bus cycle. It is driven low on external read or write cycles, and is valid for all external program, data, and I/O operations.

The $\overline{R/W}$ signal indicates whether an external read (high) or external write (low) is taking place.

The READY line input to the C240x indicates that an external device accessed by the C240x is ready for an external read or write. This signal is generated by external hardware as a C240x input and can be used as an option to create wait states.

External Interface Example



The block diagram above demonstrates the C240x control signals necessary to interface memory and peripherals into each of the memory spaces.

$\overline{\text{PS}}$ — Active low on access to external program space (fetch)

$\overline{\text{DS}}$ — Active low on access to external data space (read)

$\overline{\text{IS}}$ — Active low on I/O operations (IN or OUT)

A15-A0 — Carries address of external access

D15-D0 — Carries data on external access

$\overline{\text{RD}}$ — Active low on external read

$\overline{\text{WE}}$ — Active low on external write

The following signals are used less frequently for standard memory interface.

$\overline{\text{STRB}}$ — Active low on all external accesses (not used in this example)

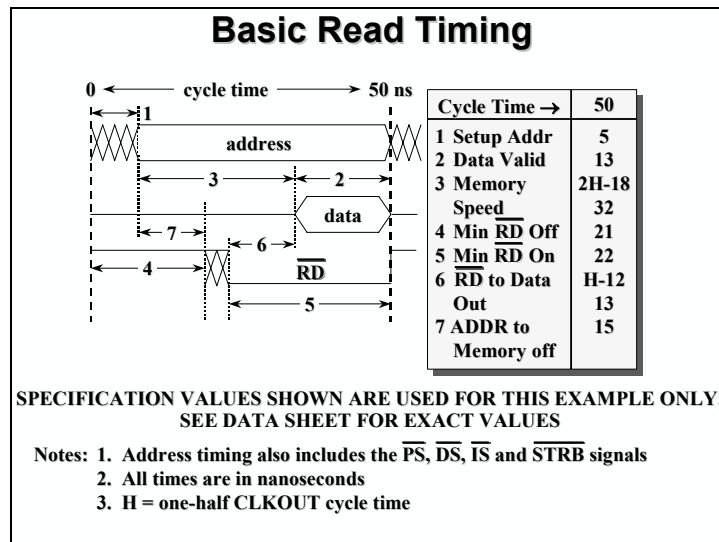
$\text{R}/\overline{\text{W}}$ — High on external read; low on external write (not used in this example)

READY — Used to extend external bus cycle (not used in this example)

External Read Timing

The C240x memory read interface is distinguished by three specific features:

- Fast single-cycle operation
- Approximately two-thirds of the memory cycle is allotted to memory access time.
- Read Strobe ($\overline{\text{RD}}$) line can provide direct glueless interface to the $\overline{\text{OE}}$ input on memories.



Single-Cycle Reads

The ability to perform single-cycle reads allows the C240x to maintain maximum performance during read-intensive DSP algorithms. Single-cycle is defined by the CLKOUT signal that determines the instruction throughput rate on the C240x devices. In the above figure, the period of CLKOUT is 50 ns, as an example.

Long Memory Access Times

The C240x external read interface provides about two-thirds of its cycle to memory access time. This is due to the C240x's ability to generate address and control lines at the start of the access cycle and perform the data read at the end of the access cycle. Even though 60% is greater than the normal 30% of the cycle most processors provide, memory access times still become small due to the short instruction cycle times of the fastest C240x processors. The access time is calculated by subtracting the address setup time and the data hold time from the entire cycle time.

Read Strobe

The C240x memory interface includes a specific read strobe (\overline{RD}) that activates half way through the memory read cycle. It is designed to allow glueless connections to a memory's output enable (\overline{OE}) pin.

Activating a memory's \overline{OE} pin causes the memory's output drivers to activate, thus driving the bus with the data output value. The advantage of using the \overline{OE} feature is to prevent multiple memories from simultaneously driving the bus, while maximizing the memory read access time. That is, at the beginning of a read cycle, the new memory chip specified by the address is selected immediately, but is held off the bus until its \overline{OE} is activated, thus allowing the previous memory ample time to remove itself from the bus. Although \overline{RD} falls late in the cycle, this should not present a problem as \overline{OE} to data valid is usually a very short time.

To ensure proper operation, two critical timings must be met. First, \overline{RD} active (low) to the memory's data valid must occur by the timing in specification 6 in the Basic Read Timing diagram above. Second, all memories within the system must be capable of disabling (turning off) before \overline{RD} active (low).

External Write Timing Issues

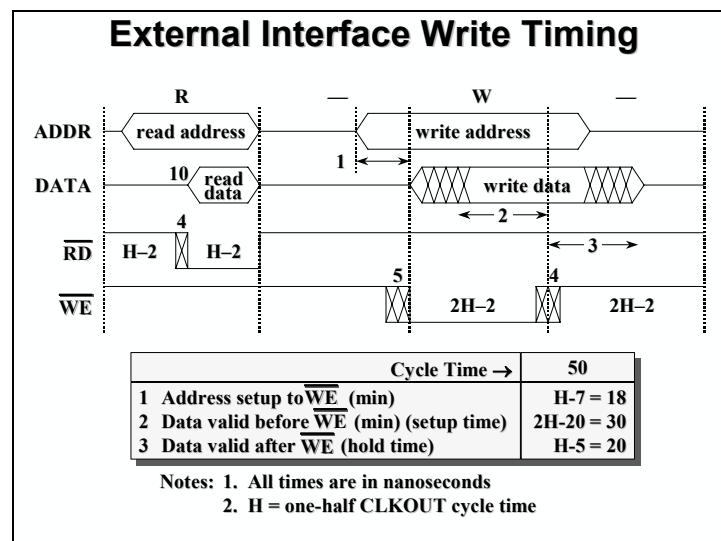
In this section, the write timing will be defined. You will find that a worst-case view shows a write requires three cycles to implement. TI chose this approach to allow for lower-cost systems with minimal performance loss. Given a three-cycle bus action, a valid question would be: "How could TI produce a benefit that outweighs the bus slow down?"

To answer this question, it is necessary to return to the read timing. Given that most processors offer about one-third of a cycle as setup time, it seems that the C240x memory times are twice as wide as you would expect. How is this possible? By performing reads without leading and trailing dead times for address and control signal setup times. This process is satisfactory for reading data, but could introduce errors in external memory if writes progress without inactive transition times. Therefore, the write was given an extra cycle before and after to provide a dead time while addresses are set up.

How valuable was the benefit of relaxed memory timing for full-speed reads? Consider the cost of memory as noted, versus those with times twice as fast. As most hardware designers know, cost rises rapidly when state of the art speeds are approached (as they are here). Next, consider the penalty of a three-cycle write. First: how often are writes performed versus reads? In most systems, the ratio is well below 1:10. From this analysis, you can see that a 10% speed penalty (or less) can yield a 2:1 or better cost savings. Add to that the power dissipation, memory density, and other issues associated with faster memories, and the advantage becomes even greater.

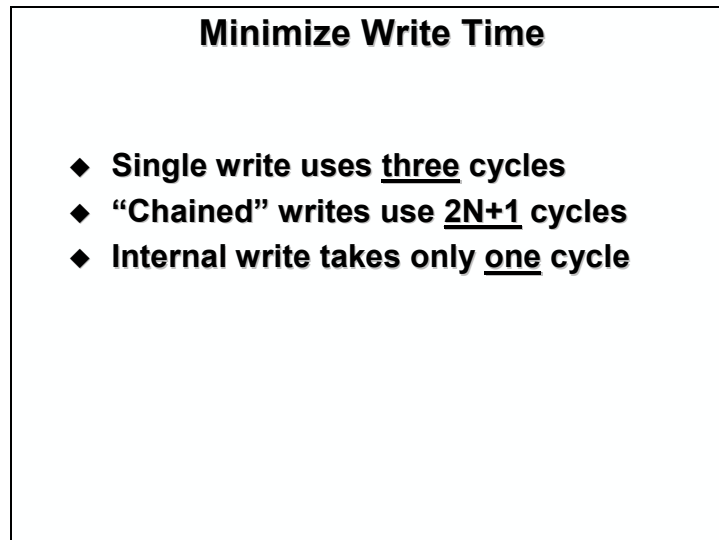
External Write Timing

As noted above, write timings are multicycle. Thus, when compared to read timing, they should be easy for memories to meet. Below is an example of a read followed by a write. Notice the dead (no strobe) cycles before and after the write active cycle.



Minimizing the Write “Slow Down”

So far we have seen that write timing was sacrificed to optimize read timing. This relatively small write penalty allows a relatively large reduction in memory cost. Now, we will consider further ways to reduce the speed penalty of a write. This is summarized in the following diagram.



I/O Memory Space

The I/O space interfaces to 64K external peripheral ports using the same multiplexed 16 address and 16 data bus lines as the program and data space. I/O accesses decode the IS line along with the address to determine which of 65,536 port addresses are selected. Read and write timings for I/O space are identical to those presented earlier for data space reads and writes.

The following example shows how the IN and OUT instructions may be used to access the entire 64K of I/O memory space.

```
IN      ADDR,1      ;read data from an external device
                        ;on port # 1
OUT     ADDR,125     ;write this data to another external
                        ;device on port # 125.
```

Consider the requirements to interface peripherals to general I/O space and memory-mapped I/O space.

- Device can be interfaced to I/O space, requiring the use of IN or OUT instructions.
- If only one I/O device, only the $\overline{\text{IS}}$ line is needed to perform chip select.
- If multiple I/O devices, one or more address lines need to be used in conjunction with $\overline{\text{IS}}$.
- If more address lines (including $\overline{\text{IS}}$) must be used than there are chip selects on the I/O device, external decode logic must be used.
- Some I/O devices have output ($\overline{\text{OE}}$) and write ($\overline{\text{WE}}$) enables. The C240x $\overline{\text{RD}}$ and $\overline{\text{WE}}$ signals often work well with these types of I/O devices.

Parallel I/O

The 64K I/O ports available to the TMS320C240x permit a parallel I/O scheme to be used in large multiprocessor arrays. Latches or FIFOs on each address space could be a mailbox to a different processor. This would create a relatively efficient, but somewhat slow, parallel interface between up to 64K processors. The two penalties in this scheme are:

- Both processors must perform multicycle reads and writes to the latches, whereas a DMA operation allows single-cycle transfers requiring only one processor.
- Each mailbox requires hardware in the form of latches or FIFOs.

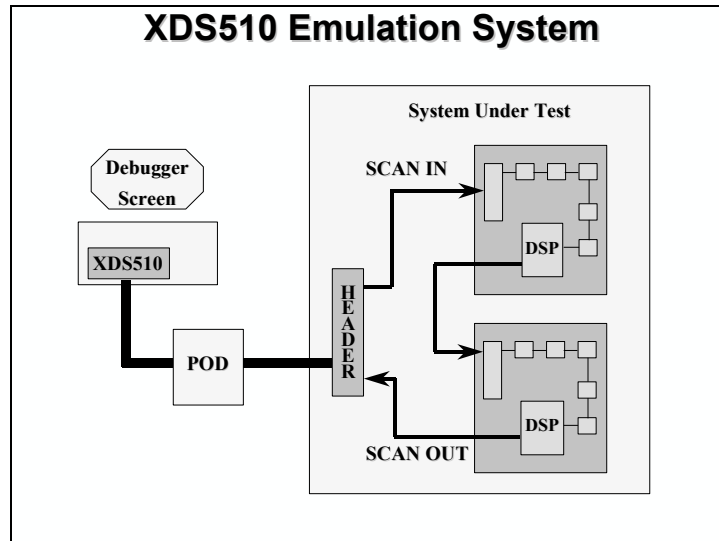
Memory Timing Summary

Memory Timing Summary

- ◆ All internal accesses are single cycle
- ◆ External-read timing is biased for largest memory access times
- ◆ Write-cycle timing (cost/performance tradeoffs):
 - 3 cycles for a single WRITE
 - 2 cycles per WRITE in multiple WRITES
- ◆ Software-generated wait states accommodate slower memories

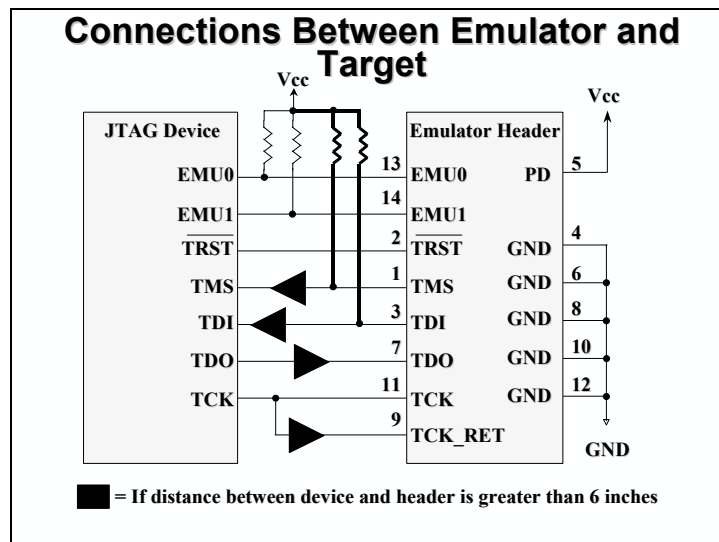
JTAG Scan-Based Emulation

During system test and debug, it is necessary to be able to control and observe registers on the CPU. The term JTAG refers to TI scan-based emulation, which is based on the IEEE 1149.1 standard. JTAG provides the ability to scan information into and out of one or more devices. The scan ring on the C240x includes all internal CPU registers as well as internal memory. This ring allows the user to modify any register or memory location on the device. In order to connect directly to a TI XDS510 emulator, certain design requirements need to be taken into account with the target system.



JTAG target devices support emulation through a dedicated emulation port which is accessed by the emulator. To communicate with the emulator, the target system must have a 14-pin header (two rows of seven pins).

Connections Between Emulator and Target System



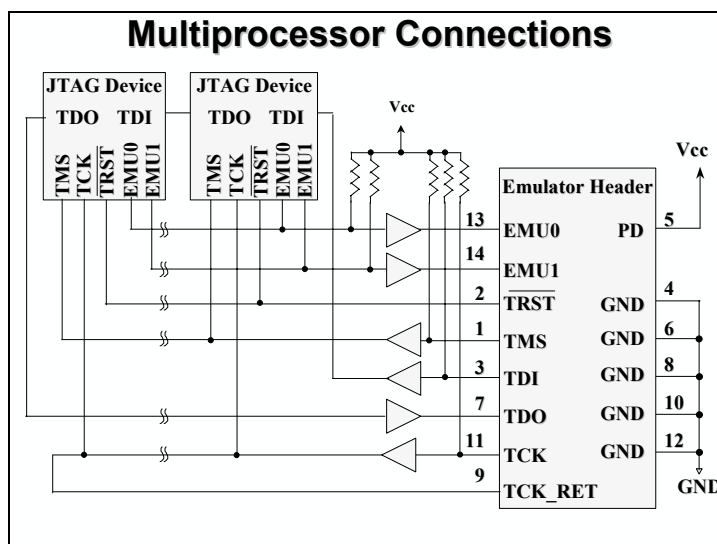
Target-System Clock

It is also possible for the system test clock to be generated by the target system (TCK signal is left unconnected). There are two benefits in having the target system generate the test clock:

- The emulator provides only a single 10.368 MHz test clock. By allowing the target system generate the test clock, it is possible to set the frequency to match the system requirements.
- In some cases, other devices in the system may require a test clock when the emulator is not connected. The system test clock can also serve this purpose.

Configuring Multiple Processors

The figure below shows a typical daisy-chained multiprocessor configuration that meets the minimum requirements of the IEEE 1149.1 specification. The emulation signals are buffered to isolate the processors from the emulator and provide adequate signal drive for the target system.



Flash Programming

The TMS320F240x devices contain on-chip Flash EEPROM (electrically-erasable programmable read-only memory). The embedded Flash memory provides an attractive alternative to masked program ROM. Like ROM, flash memory is nonvolatile, but it has an advantage over ROM: *in-system* reprogrammability.

Embedded Flash memory expands the capabilities of the device in the areas of prototyping, integrated solutions, and field upgradeable designs. The embedded Flash memory also make these devices ideal for highly integrated, low-cost systems.

Flash Programming

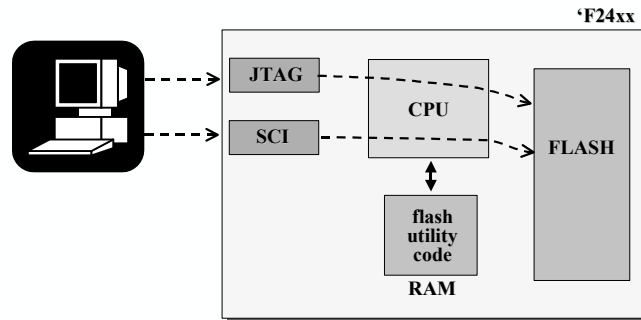
| | |
|---------------------|---------------|
| TMS320LF2407 | - 32Kw |
| TMS320LF2406 | - 32Kw |
| TMS320LF2403 | - 16Kw |
| TMS320LF2402 | - 8Kw |
| TMS320LF2401 | - 8Kw |
| TMS320F243 | - 8Kw |
| TMS320F241 | - 8Kw |
| TMS320F240 | - 16Kw |

Flash Programming Utilities

- ◆ **TI provides PC based flash programming utilities downloadable from our website:**
<http://www.ti.com>
- ◆ **The utilities support programming using**
 - ◆ JTAG - TI XDS510 emulator
 - ◆ JTAG - Spectrum Digital XDS510PP emulator
 - ◆ Serial Loader - PC RS232 link to SCI
- ◆ **The utilities are device specific**
 - (a) F240 (b) F241/3 (c) LF240x (d) LF240x-A

Flash Programming Method

- ◆ The flash programming is performed by the DSP CPU itself. It executes code from on-chip RAM that reads incoming data from the PC, and writes that data to the flash



Getting the Utility into the RAM

- ◆ JTAG: the emulator scans the utility into the RAM
- ◆ Serial Loader: flash resident when the device leaves the TI factory
 - ◆ BIO\ pin is checked by the utility out of reset
 - ◆ BIO\ low - branch to address 40h and start user code
 - ◆ BIO\ hi - execute flash programming utility
 - ◆ Serial utility puts itself back into the last 256 words of flash after programming is complete
 - ◆ If serial utility is not flash resident, you must use an emulator to place it there.

Flash Programming Procedure

- ◆ To program the flash, the following sequence of steps must be performed, in order:

| Algorithm | Function |
|------------|-----------------------------------|
| 1. Clear | - change all bits to zero |
| 2. Erase | - change all bits to one |
| 3. Program | - program selected bits with zero |

- ◆ If these steps are not followed, flash may be driven into depletion (over erased)
 - An error code ERROR114 returned by the Clear or Erase algorithm can indicate depletion
 - Use the *flash-write* algorithm to attempt recovery
 - See the utility documentation for details

Generally, not all bits in Flash have the same amount of charge removed with each erase pulse – some bits may be over erased and are referred to as in depletion mode. The Flash-Write recovers the bits from depletion mode.

Flash Programming

- ◆ **Scan Based Programmer (XDS510)**
 - Supports all flash algorithms (clear, erase, program, and flash-write)
 - Multiple devices must be programmed sequentially
- ◆ **Serial Based Programmer (SCI)**
 - Supports main flash algorithms only (clear, erase, and program)
 - Multiple devices can be programmed in parallel

Flash Programming Problem Checklist

The following checklist is useful for debugging Flash programming problems:

1. Is the Vccp pin directly connected to 5V (without any resistor)?
2. Is the PMT pin directly connected to GND (without any resistor)? (Applicable for 243/241 only).
3. Is the device in MC mode?
4. Does the COFF file (*.out) correctly fit in the flash memory ? (i.e. are there any sections that lie outside the on-chip flash memory range?) If you are able to program the supplied sample file (L8KN.OUT), but are unable to program your file, it suggests a possible problem with your COFF file. Also, do you have any initialized sections in the data space?
5. (Applicable only if plugin is not used) Does BTEST.BAT work correctly? If BTEST itself fails, then there is no point in even trying clear,erase etc. This suggests a possible communication problem between the PC and the DSP via JTAG. This can be further verified by invoking Code composer. If CC does not work, then this communication problem must be fixed first before attempting to use the flash tool.
6. If CC works correctly, then chances are BTEST will work correctly as well. If Clear/Erase/Program operation still fails, can you try a different target board?
7. If you are having problems programming your own target board, try programming (if possible) a commercially available board such as the EVM. This will help identify a board issue versus a tools/emulator issue.
8. Can you try a different DSP sample or different target board?
9. Are you using the most recent version of the flash programming tool? (This can be downloaded from TI or 'Spectrum Digital' website).
10. Is your CLKOUT correct? Monitor the CLKOUT pin using an oscilloscope and make sure your CLKOUT does not exceed the values specified in the datasheet.
11. If your CLKOUT is not the rated maximum CLKOUT of the device (as specified in the datasheet), have you reconfigured the flash programming timing parameters?

Questions applicable to serial port method only:

12. Is the PC's serial port working correctly? This can be easily checked by using a terminal emulation program. Short pins 2 & 3 of the serial port and check if the typed characters are echoed on the screen.
13. Is the communication between the PC and the target board working? This can be checked with the simple diagnostic program PCECHO.asm available in the 'Reference guide'

14. Is the XF/-BOOT_EN pin tied low? (Applicable for 240x/240xA only).

15. If using the LF2407 EVM, make sure you have jumper JP12 in the 1-2 position. This enables the SCI receive signal on the P6 9-pin connector. (Applicable to serial port method only).

16. Are you running Code Composer under Win2000 or Windows ME? Win2000 and Windows ME are not supported by Code Composer v4.1x. Therefore, running the Flash plugin tool may provide unreliable results.

17. Are you using the correct tools ?

For the 'LF240x use the 'LF240x tools. For the 'LF240x -A use the 'LF240x-A tools. For the F241/3 use the F243 tools. For the F240 use the F240 tools.

The plugin for the CC IDE (plugin) version 1.0 supports only the LF240x and LF240x-A parts.

Code Security Module (CSM) – frequently asked questions

What is CSM?

CSM is a security feature incorporated in TMS320Lx240xA DSP controllers. It prevents access/visibility to on-chip Flash/ROM memory (in program space) to unauthorized persons. (i.e. it prevents duplication/reverse engineering of proprietary code).

What do the terms “secure” and “unsecure” mean?

“Secure” means access to on-chip flash/ROM memory is protected. “Unsecure” means access to on-chip flash/ROM memory is not protected. (i.e. the contents of the flash/ROM could be read by any means such as through a debugging tool like Code Composer, for example).

Under what conditions is the device unsecure?

A device is unsecure when the device comes up in the intended application mode in which code is executed from on-chip flash/ROM, **without JTAG connector connected**. (i.e the device is brought up in “microcontroller” mode upon reset with the on-chip ROM bootloader disabled). Note that this is the typical usage of the DSP in an end-product.

Under what conditions is the device secure?

1. When the on-chip ROM bootloader is invoked.
2. When the JTAG connector is connected.
3. When the DSP is powered up in MP mode.
4. When the KEY register values and the PWL values are different.

Can you explain the terms PWL and KEY registers?

PWL stands for “Password locations”. These are memory locations at addresses 40h to 43h in on-chip flash/ROM which store the passwords. PWL is mapped in program memory space. KEY registers are memory locations at addresses 77F0h to 77F3h in on-chip “Peripheral register” space. A device is unsecured by writing the passwords to KEY. KEY is mapped in data memory space.

How do I secure a device?

You secure a device by ensuring the presence of passwords (other than FFFFFFFFFFFFFFFFh or 0000000000000000h) in the PWL.

How do I unsecure a device?

You unsecure a device by executing the following steps:

1. Do a “dummy” read of PWL. The word “dummy” implies that the destination address of this read is insignificant. Only the read of the PWL is important.
2. Write the passwords (stored in the PWL) to the KEY. Of course, you need to know the passwords (stored in the PWL).

Should I program all 64-bits of the password?

For maximum protection, it is advisable to program all 64-bits.

I don’t want to use the CSM. Can I “bypass” it?

There is no way to “bypass” the CSM in TMS320Lx240xA DSP controllers. If code security is not a concern, you can program the “dummy” passwords (FFFFFFFFFFFFFFFFh or 0000000000000000h) in the PWL.

I have programmed the PWL with “dummy” passwords. Do I still need to perform dummy reads of the PWL when I am doing JTAG emulation/debug?”

A dummy read of the PWL is still essential to gain visibility to on-chip flash/ROM. A write to the KEY is not required. In situations where a debugger is used, a read of the PWL by the debugger (in the disassembly window) is sufficient. For example, right-click in the Disassembly window, select Start Address, and enter 0040h in the box. This will unsecure the on-chip ROM/FLASH.

Are there any precautions I should observe while developing code?

During the code development phase, it is a good idea to use the dummy passwords (or stick to a single password).

Should I incorporate routines to “unsecure” the device in my application code?

There is no need to incorporate routines to “unsecure” the device in your application code. Recall that the device comes up as “unsecure” when you power it up in “microcontroller” mode (without JTAG connector connected/ with the on-chip ROM bootloader disabled). Unsecuring is necessary only when you need visibility to ROM/Flash on a currently secured device.

Is CSM applicable to any other memory space?

No, it is applicable only to on-chip Flash/ROM memory.

Should the device be unsecure to run application code?

Yes. The device must be unsecure in order to be able to execute code out of on-chip Flash/ROM memory.

I don’t need code security. Can I store code in PWL also?

This is not advisable. Keeping tab of the password may be difficult, especially if code changes are possible. It is a good practice to define a password section in the project to isolate the PWL from the rest of the code. This forces the user code to begin at 44h and precludes the possibility of code starting from 40h. This practice is especially advantageous when migrating code from LF240x, where code starts at 40h.

**How does the presence of CSM affect flash programming of LF240xA devices?
(or)****I successfully programmed the flash once, but I am unable to do it again. What could be wrong?**

The device must be first unsecured before Clear/Erase/Program (CEP) can be performed. Update the key.asm program with the correct passwords. Assemble and link the program using key.bat. Then run unlock.bat to unsecure the device. You should now be able to clear/erase/program.

After I invoked ‘Code composer’, I couldn’t see my code (programmed in flash) in the disassembly window. I see some “garbage” code instead. What could be wrong?

The device is still in secure mode. In order to be able to view your code in the disassembly window, the device must first be unsecured.

Can you provide me a simple code to unsecure the device?

The following code can be executed from B0 or SARAM:

```
.text

LDP    #00E0h          ; (E0=224) (E0*80=7000)
SPLK   #006Fh, WDCR    ;Disable WD

LDP    #0h             ;Dummy read of the PWL
BLPD   #0040h, 60h     ;update high word
BLPD   #0041h, 60h     ;third word
BLPD   #0042h, 60h     ;second word
BLPD   #0043h, 60h     ;low word

LDP    #0EFh           ;Writing the password
SPLK   #00123h, 77F0h  ;to the KEY registers.
SPLK   #04567h, 77F1h  ;Replace the words shown
SPLK   #089ABh, 77F2h  ;with the appropriate
SPLK   #0CDEFh, 77F3h  ;passwords.

LOOP   B      LOOP
```

I forgot the password I programmed in PWL. Will I be able to reprogram the flash?

No. Not unless you know which COFF file you used to program the flash. It is for this reason you should always store a known value in the PWL during the code development phase.

Are there any restrictions on debug capabilities when secure mode is used ?

No. Once the device is unsecured, the CSM has no impact on debug capabilities.

Are all of the 'Real Time' capabilities still available ?

Yes. CSM does not impact the 'Real Time' capabilities.

Does the addition of Secure Mode require any modifications to the Application Code itself ?

The only requirement is the presence of passwords in the PWL.

Are there any 'bad practices' which should be avoided (which compromise security) ?

Please refer to the “*DOs and DON'Ts to Protect Security Logic*” section in the ‘Reference guide’ for 240xA devices (SPRU357A).

In mass production, can the Flash be programmed, and made secure, in ONE, fast, operation ?

There is no special operation needed “to secure” a device, other than ensuring the presence of passwords in the PWL.

Do the BLPD and TBLR instructions still work when in secure mode ? If so, what prevents a 'Trojan Horse' program...attached to the external bus, from copying from Program to Data space, then allowing data space to be copied to the UART ...or being visible via JTAG ?

No, BLPD and TBLR do not work when the device is in secure mode. Using the external bus implies Microprocessor mode. The device is secured in MP mode. The device will also be secured immediately when the JTAG connector is connected.

Review

Review

- ◆ What signals select program, data or I/O?
- ◆ What happens to the address lines when there is no external bus activity?
- ◆ What is the advantage of slower write cycles?
- ◆ What are the three stages for Flash programming?

This page is left intentionally blank.

Introduction

The TMS320C240x contains features that allow several methods of communication and data exchange between the C240x and other devices. Many of the most commonly used communications techniques are presented in this module.

The intent of this module is not give to 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

- ◆ **Configure the C240x for a communications environment**
- ◆ **Understand SCI, SPI and CAN operation**

Module Topics

Communications.....14-1

Module Topics.....14-2

Communications Techniques14-3

Bit I/O14-4

Serial Ports14-5

 Serial Peripheral Interface (SPI).....14-6

 Serial Communications Interface (SCI).....14-12

Controller Area Network (CAN)14-17

Review.....14-22

Communications Techniques

Several methods of implementing a TMS320C240x 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 following figure. Each will be described in the following sections.

C240x Communication Techniques

- ◆ **Bit I/O**
- ◆ **Serial Peripheral Interface (SPI)**
- ◆ **Serial Communication Interface (SCI)**
- ◆ **Controller Area Network (CAN)**

Bit I/O

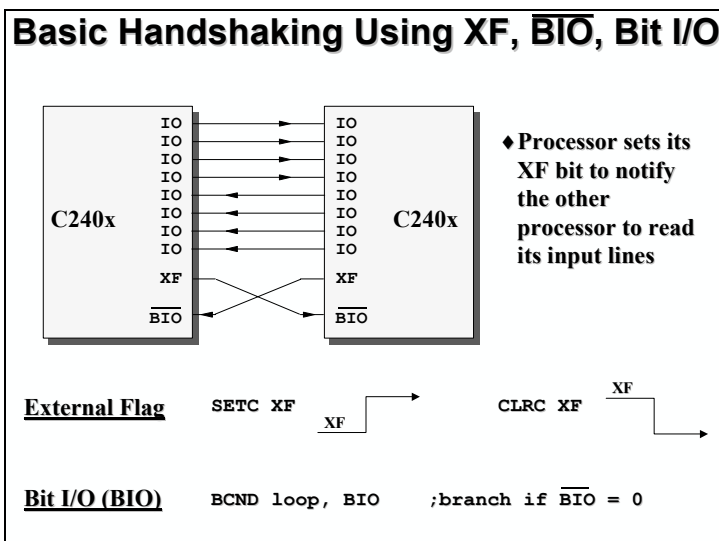
A variety of schemes can be developed for simple bit signaling/transferring between the TMS320C240x and other devices with bit I/O capability. The C240x has up to 32 digital bit I/O pins that are shared with other peripheral functions. All 32 can be configured/accessed via data memory mapped registers. Two of these pins, XF/IOPC2 and $\overline{\text{BIO}}$ /IOPC3 act as bit I/O with their own special access instructions when configured for their shared peripheral functions XF and $\overline{\text{BIO}}$.

```
BCND    <<pma>>,  $\overline{\text{BIO}}$       ;Branch to <pma> when the  $\overline{\text{BIO}}$  line is zero (low)
SETC    XF              ;Set external flag bit; XF bit in status reg = 1
CLRC    XF              ;Clear external flag bit; XF bit in status reg = 0
```

The $\overline{\text{BIO}}$ pin is similar to an interrupt pin in that it senses the state of a single bit, but differs in that it is a software, not hardware, function. Because it is a software function, the programmer must test the $\overline{\text{BIO}}$ pin via the BCND instruction to determine its state. Furthermore, the BCND test reveals only the “current” status of the $\overline{\text{BIO}}$ line, and provides no information about prior status. This differs from an interrupt line which is latched, and will thus hold a high logic state until serviced even if the high signal is removed.

The external flag output pin provides a complementary function to the $\overline{\text{BIO}}$ line, allowing single instruction software control of an output pin. XF is set high on reset.

These pins can be used in a single processor system, or may be used in more complex systems as an efficient means of low-level communications. In a multiprocessor system, they are often combined with other means of data transmission and, for example, may serve the function of a request and acknowledge operation. The use of $\overline{\text{BIO}}$ and XF allow the use of bit signaling without the use of external logic. This reduces board space, design time, and system cost. Also, since the setting or testing of these pins requires only a single cycle, the overhead in software is minimal. Note that the timing of the $\overline{\text{BIO}}$ and XF reflect the latency of the pipeline; that is, they take effect during the third (read) phase of the instruction's implementation.



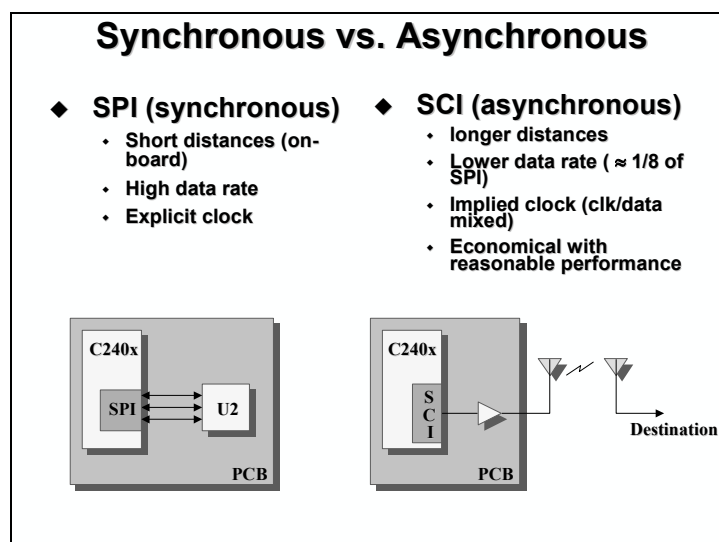
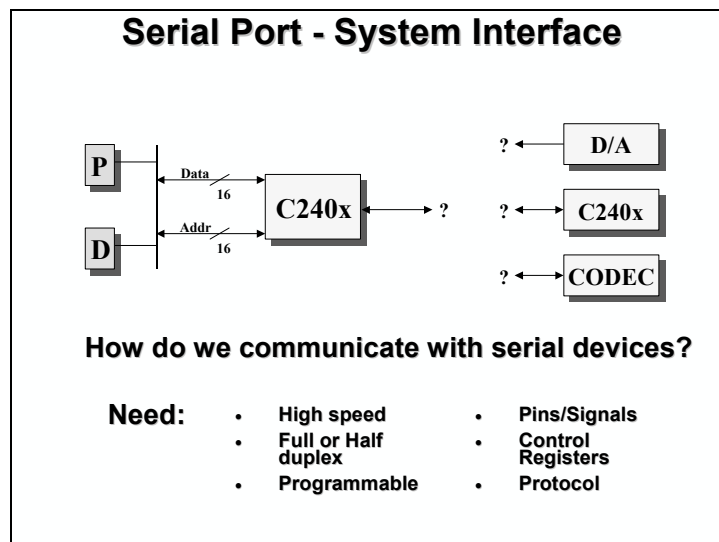
Serial Ports

Serial ports provide a simple, hardware-efficient means of high-level communication between devices. Like the bit I/O 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 C240x family of devices have both a synchronous and asynchronous serial ports. Detailed features and operation are described in the following figures.



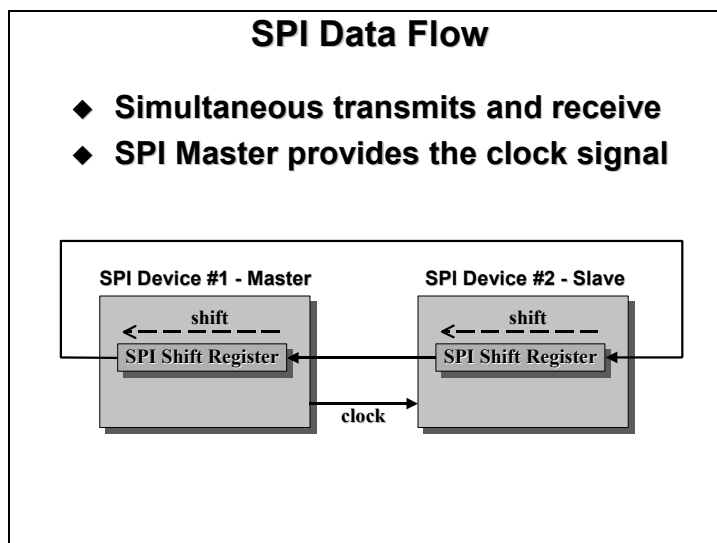
Serial Peripheral Interface (SPI)

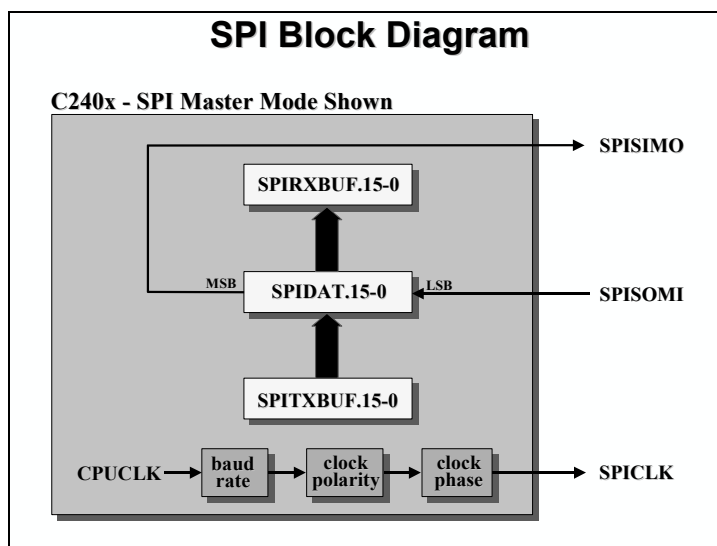
The SPI module is a synchronous serial I/O port that shifts a serial bit stream of variable length and data rate between the C240x and other peripheral devices. During data transfers, one SPI device must be configured as the transfer MASTER, and all other devices configured as SLAVES. The master drives 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 or 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.



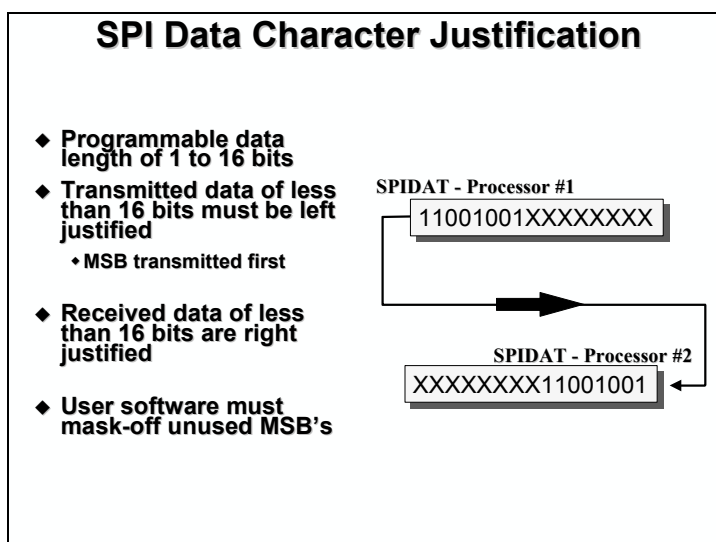


SPI Transmit / Receive Sequence

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

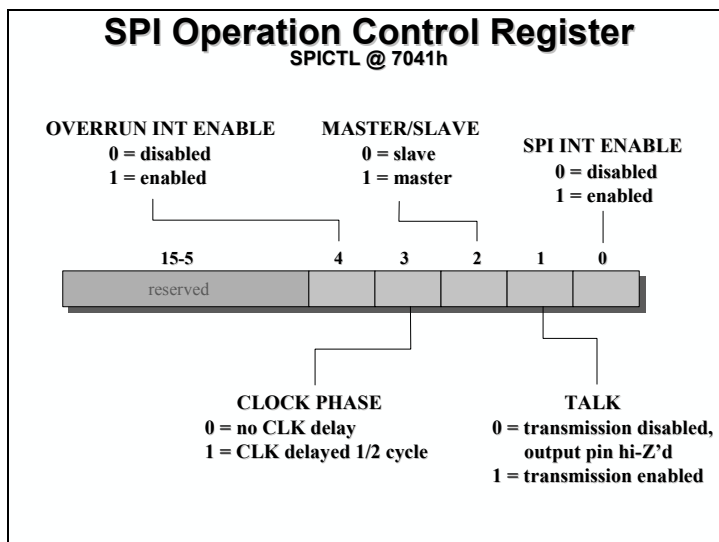
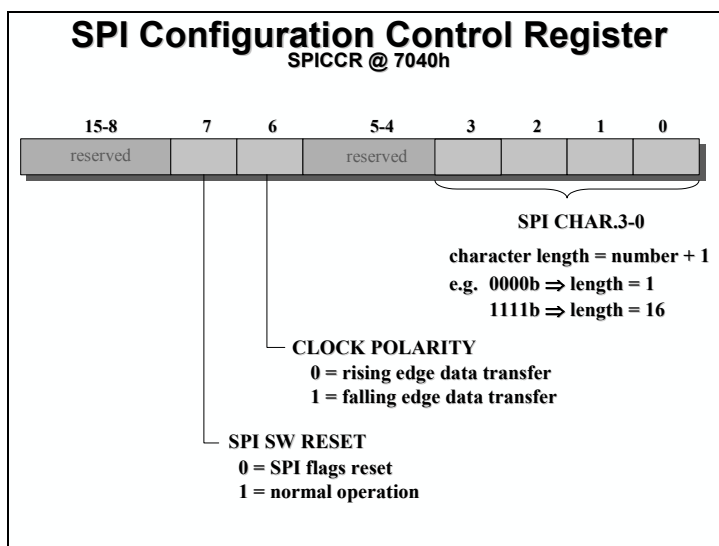
Since data is shifted out of the SPIDAT register MSB first, transmission characters of less than 16 bits must be left-justified by the CPU software prior to be written to SPIDAT.

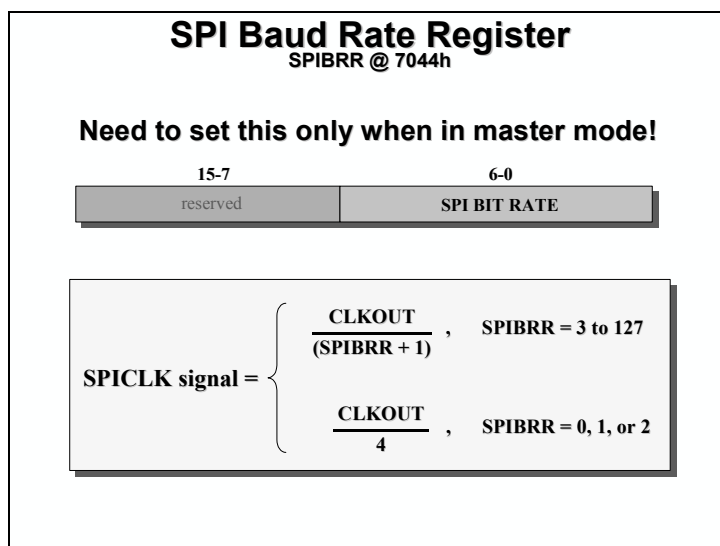
Received data is shifted into SPIDAT from the left, MSB first. However, the entire sixteen bits of SPIDAT is copied into SPIBUF after the character transmission is complete such that received characters of less than 16 bits will be right-justified in SPIBUF. The non-utilized higher significance bits 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 Registers

| Address | Register | Name |
|---------|----------|-------------------------------------|
| 7040h | SPICCR | SPI configuration control register |
| 7041h | SPICTL | SPI operation control register |
| 7042h | SPISTS | SPI status register |
| 7044h | SPIBRR | SPI baud rate register |
| 7047h | SPIRXBUF | SPI serial receive buffer register |
| 7048h | SPITXBUF | SPI serial transmit buffer register |
| 7049h | SPIDAT | SPI serial data register |
| 704Fh | SPIPRI | SPI priority control register |





Baud Rate Determination: The Master specifies the communication baud rate using its baud rate register (SPIBRR.6-0):

- For SPIBRR = 3 to 127:

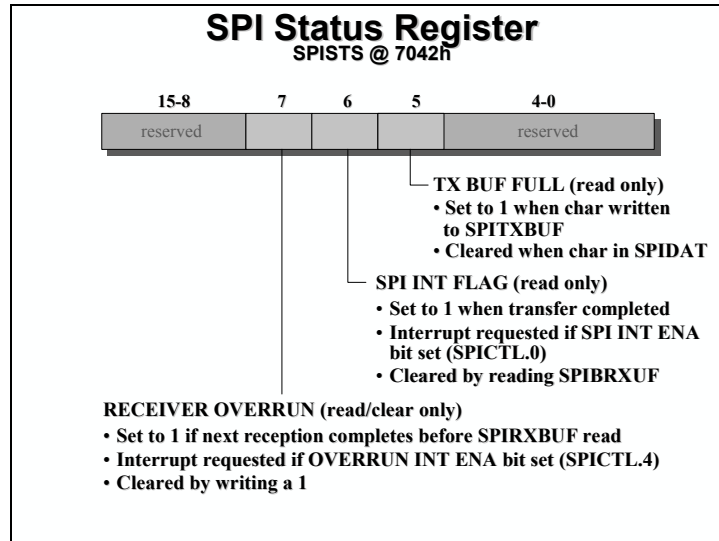
$$\text{SPI Baud Rate} = \frac{CLKOUT}{(SPIBRR + 1)} \text{ bits/sec}$$

- For SPIBRR = 0, 1, or 2:

$$\text{SPI Baud Rate} = \frac{CLKOUT}{4} \text{ bits/sec}$$

From the above equations, one can compute the maximum baud rate as 7.5 Mbps for a 30MHz device.

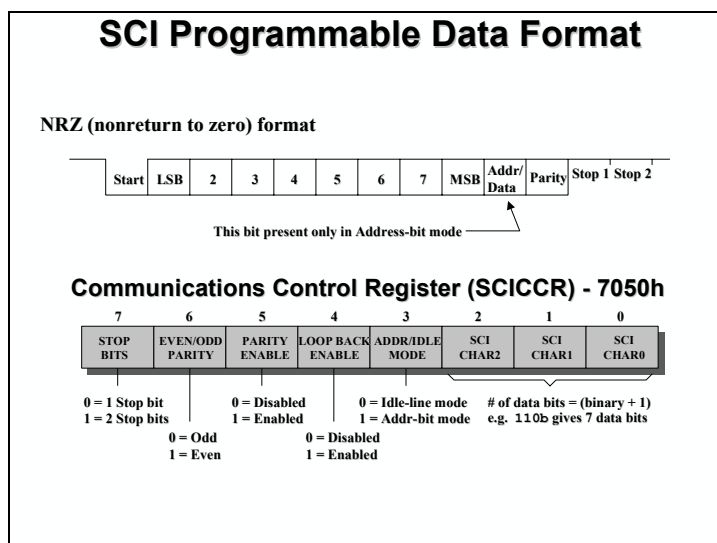
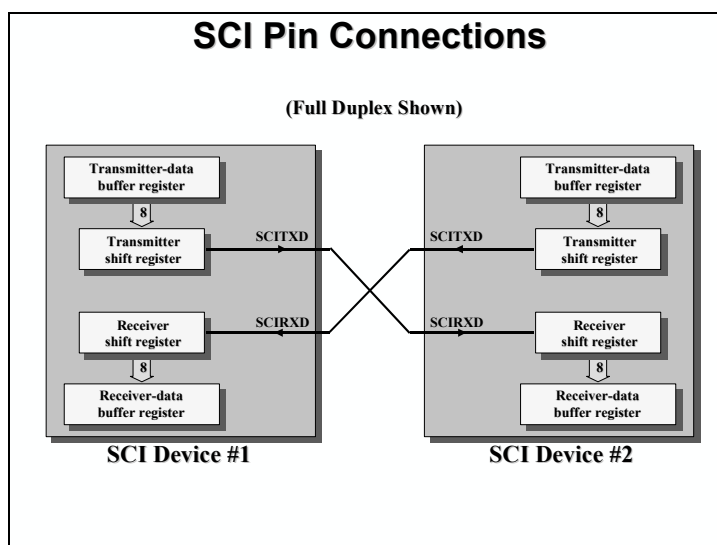
Character Length Determination: The Master and Slave must be configured for the same transmission character length. This is done with bits 0, 1, 2 and 3 of the configuration control register (SPICCR.3-0). These four bits produce a binary number, from which the character length is computed as binary + 1 (e.g. SPICCR.3-0 = 0010 gives a character length of 3).



- SPI Summary**
- ◆ **Provides synchronous serial communications**
 - Two wire transmit or receive (half duplex)
 - Three wire transmit and receive (full duplex)
 - ◆ **Software configurable as master or slave**
 - C240x provides clock signal in master mode
 - ◆ **Data length programmable from 1-16 bits**
 - ◆ **125 different programmable baud rates**
 - Maximum Baud rate of 7.5 Mbps @ 30 MHz CPUCLK

Serial Communications Interface (SCI)

The SCI module is a serial I/O port that permits Asynchronous communication between the C240x 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 C240x 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.



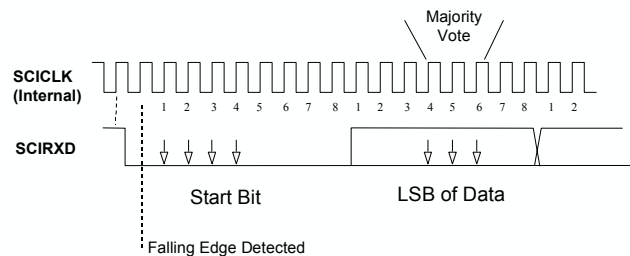
The basic unit of data is called a **character** and is 1 to 8 bits in length. Each character of data is formatted with a start bit, 1 or 2 stop bits, an optional parity bit, and an optional address/data bit. A character of data along with its formatting bits is called a **frame**. Frames are organized 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.

Asynchronous Communication Format

- 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

SCI Baud Rate

$$\text{SCI baud rate} = \begin{cases} \frac{\text{CLKOUT}}{(\text{BRR} + 1) \times 8}, & \text{BRR} = 1 \text{ to } 65535 \\ \frac{\text{CLKOUT}}{16}, & \text{BRR} = 0 \end{cases}$$

Baud-Select MSbyte Register (SCIHBAUD) - 7052h

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|--------------|--------|--------|--------|--------|--------|-------|-------|
| BAUD15 (MSB) | BAUD14 | BAUD13 | BAUD12 | BAUD11 | BAUD10 | BAUD9 | BAUD8 |

Baud-Select LSbyte Register (SCILBAUD) - 7053h

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-------|-------|-------|-------|-------|-------|-------|-------------|
| BAUD7 | BAUD6 | BAUD5 | BAUD4 | BAUD3 | BAUD2 | BAUD1 | BAUD0 (LSB) |

Baud Rate Determination: 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:

$$\text{SCI Baud Rate} = \frac{CLKOUT}{(BRR + 1) \times 8} \text{ bits/sec}$$

- For BRR = 0:

$$\text{SCI Baud Rate} = \frac{CLKOUT}{16} \text{ bits/sec}$$

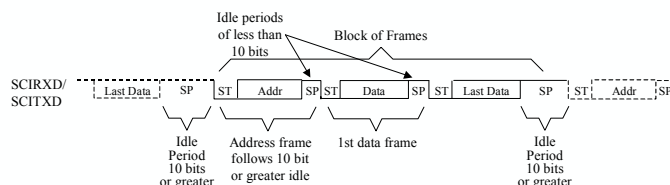
Note that the CLKOUT for the SCI module is one-half the CPU clock rate (e.g. a C240x running at 30 MHz has CLKOUT = 15 MHz for the SCI module). For CLKOUT = 15 MHz, one can compute the maximum baud rates as 1875 kbps.

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

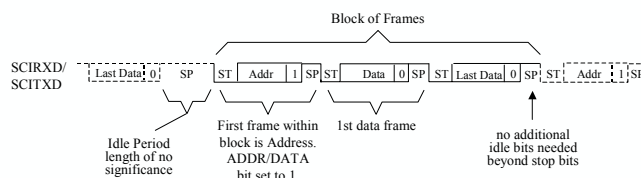
Idle-Line Wake-Up Mode

- ◆ 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 Summary

- ◆ **Asynchronous communications format**
- ◆ **65,000+ different programmable baud rates**
 - Maximum Baud rate of 1875 kbps @ 30 MHz CPUCLK
- ◆ **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
- ◆ **Double-buffered transmit and receive**
- ◆ **Individual interrupts for transmit and receive**

Controller Area Network (CAN)

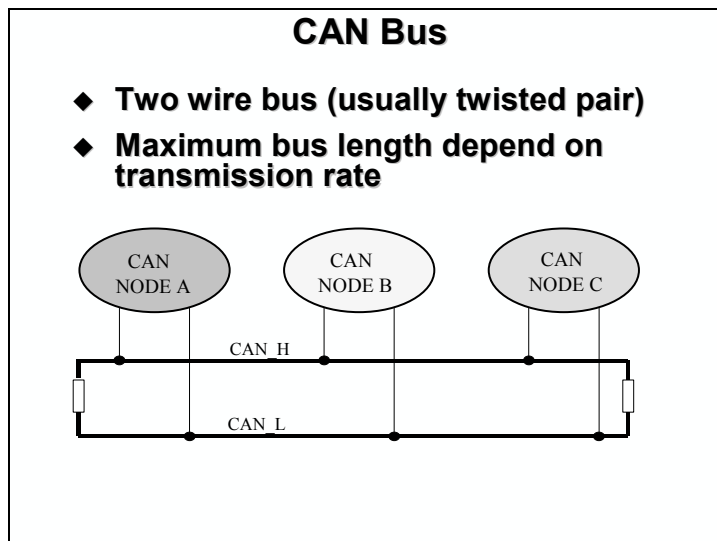
What is CAN?

- ◆ **CAN: Controller Area Network**
- ◆ **Serial bus system**
- ◆ **Broadcast type of bus:**
 - Each node can transmit to all CAN nodes.
- ◆ **Each message contains an identifier:**
 - Message filtering
 - Message priority

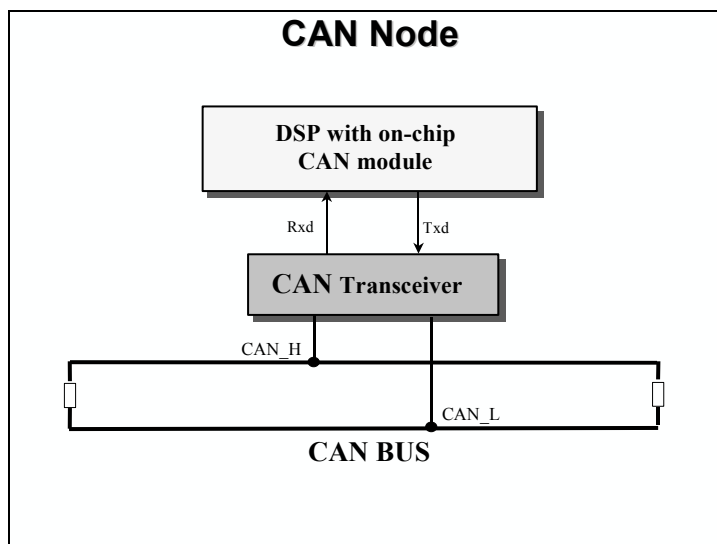
Why CAN?

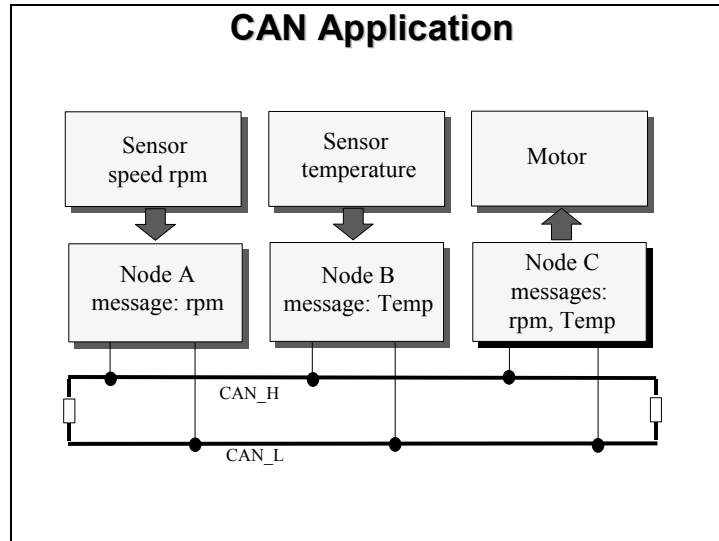
- ◆ **Low cost**
- ◆ **Multi-master network**
- ◆ **Up to 1Mbit/sec (40 m bus length)**
- ◆ **High reliability (error detection and handling)**
- ◆ **International standard (ISO 11898)**
- ◆ **Easy to learn**

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.

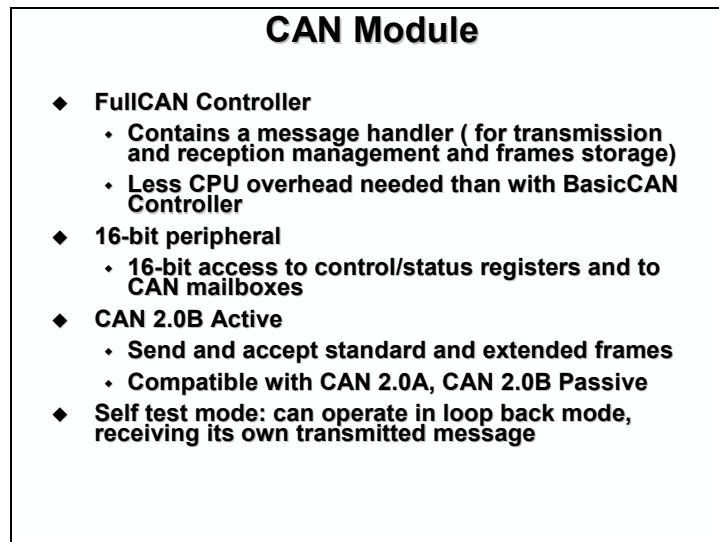


The DSP communicates to the CAN Bus using a transceiver. The CAN bus is a twisted pair wire, and the transmission rate depends on the bus length. If the bus is less than 40 meters the transmission rate is capable up to 1 Mbit/second.



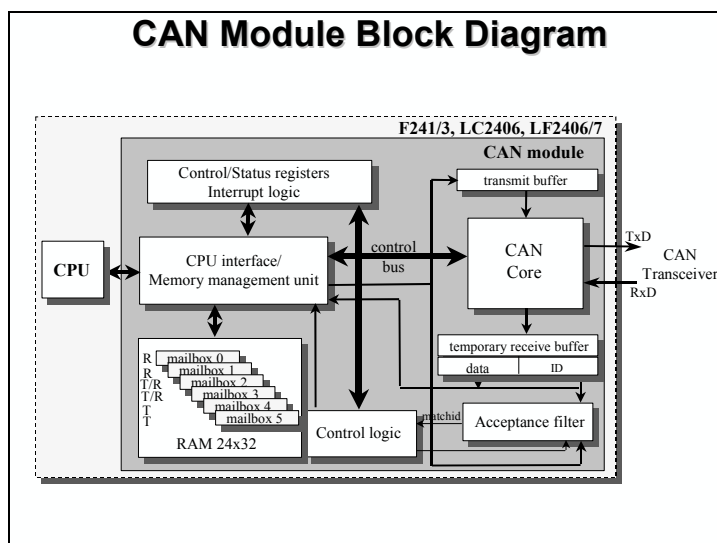
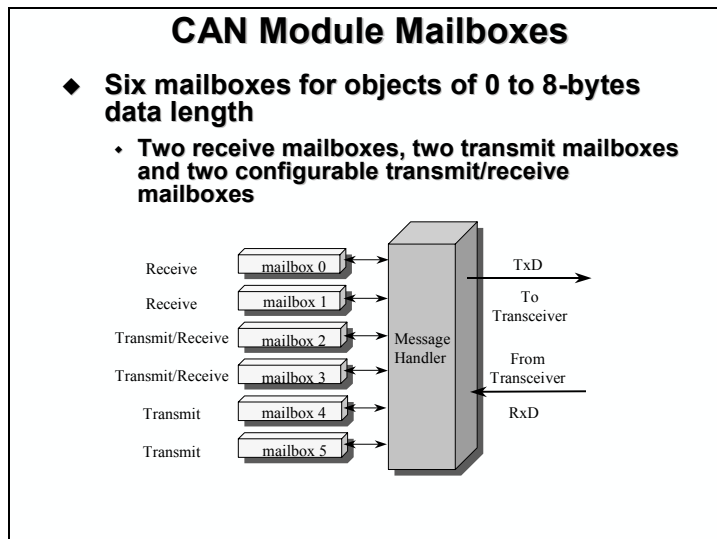


The DSP CAN module is a FullCAN Controller. It contains a message handler for transmission and reception management, and frame storage. The specification is CAN 2.0B Active – that is, the module can send and accept standard (11-bit identifier) and extended frames (29-bit identifier).



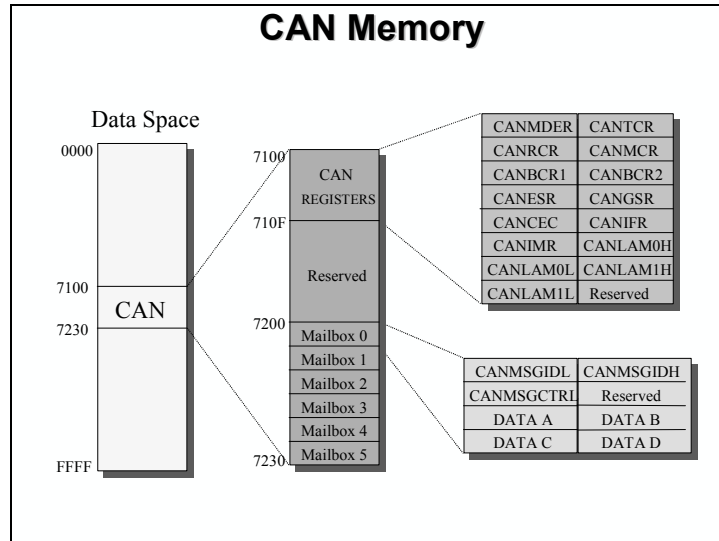
The CAN controller module contains six mailboxes for objects of 0 to 8-byte data lengths:

- two receive mailboxes (mailboxes 0 & 1)
- two transmit mailboxes (mailboxes 4 & 5)
- two configurable transmit/receive mailboxes (mailboxes 2 & 3)



The six CAN module mailboxes are divided into several parts:

- MSGIDL and MSGIDH – contains the identifier of the mailbox
- MSGCTRL (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)
- DATA_A to DATA_D – contains the data which is divided into four words or into eight bytes



The CAN module contains 15 different 16-bit registers which are divided into four groups. These registers are located in data memory from 7100h to 710Fh.

➤ **Controls Registers**

- MDER: Mailbox Direction/Enable Register – used enable/disable mailboxes and configures mailboxes 2 and 3
- TCR: Transmission Control Register – used to transmit messages
- RCR: Receive Control Register – used to receive messages
- MCR: Master Control Register – used to change bit timing configuration, write to the CAN RAM, and configure self test mode
- BCR1 & BCR2: Bit Configuration Register – used to configure bit timing

➤ **Status Registers:**

- ESR: Error Status Register – used to display errors
- GSR: Global Status Register
- CEC: CAN Error Register

➤ **Interrupt Registers:**

- IFR: Interrupt Flag Register
- IMR: Interrupt Mask Register

➤ **Local Acceptance Mask Register:**

- LAM0H & LAM0L: Local Acceptance Mask register for mailboxes 0 & 1
- LAM1H & LAM1L: Local Acceptance Mask register for mailboxes 2 & 3

Review

Review

- ◆ How can the XF, $\overline{\text{BIO}}$ and Bit I/O be used for simple communications?
- ◆ What is the data length on the SPI?
- ◆ Which two multiprocessor modes are available on the SCI?
- ◆ What is the data word format on the SCI?
- ◆ What is CAN?

Introduction

Rarely are today's systems programmed entirely in assembly language or C, most systems combine both to achieve real-time performance with a minimum of development and maintenance time. The focus of this module is to learn how to use the C compiler and control the different optimization levels.

Learning Objectives

Learning Objectives

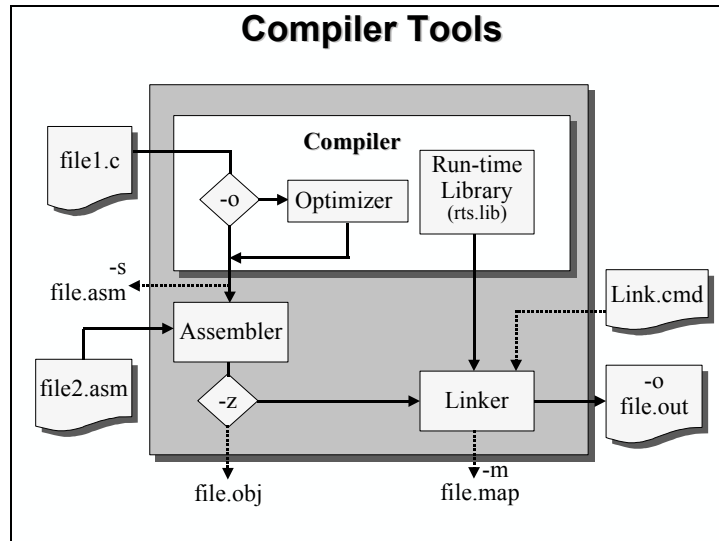
- ◆ Describe the compiler tool flow
- ◆ Describe the C run-time environment and boot process
- ◆ Understand how to call an assembly function from C
- ◆ Describe the Optimizer
- ◆ Understand how to access peripheral registers and use inline assembly

Module Topics

| | |
|--|--------------|
| C Compiler..... | 15-1 |
| <i>Module Topics.....</i> | <i>15-2</i> |
| <i>C Compiler Tools.....</i> | <i>15-3</i> |
| <i>C Runtime Environment.....</i> | <i>15-4</i> |
| Sections Created by the Compiler | 15-4 |
| Linking Code to Specific Hardware | 15-5 |
| <i>Reset in C.....</i> | <i>15-7</i> |
| BOOT.ASM..... | 15-7 |
| C Software Stack | 15-7 |
| Register Usage..... | 15-9 |
| Local Variables and Frames | 15-9 |
| Linking BOOT.ASM | 15-10 |
| <i>Mixing C and Assembly.....</i> | <i>15-12</i> |
| Steps Required for Calling Assembly Functions..... | 15-13 |
| <i>Compiler Optimizations</i> | <i>15-18</i> |
| Optimization Levels | 15-18 |
| Optimization Steps | 15-19 |
| Using the volatile keyword..... | 15-21 |
| <i>Other Items</i> | <i>15-24</i> |
| Accessing Memory and Peripheral Locations | 15-24 |
| asm() Function..... | 15-25 |
| ISR in C..... | 15-26 |

C Compiler Tools

Compiling a C program is a multistep process. It consists of compile, assemble and linking the source files to create an executable output file. The shell program (DSPCL) invoked by Code Composer does all three of these steps for you. The compiler includes an optimizer that allows you to produce highly optimized assembly code. There is also a utility that interlists your original C source statements into the compiler's assembly language output file.



Invoking the Compiler

◆ Task performed by Code Composer -

Compile and Link 'test.c' using 'test.cmd':
`dspcl -v2xx -gs -as -al test.c -z test.cmd`

| Options | Description | Tool |
|---------|--|-----------|
| -v2xx | Create code for TMS320C2xx processor | Compiler |
| -g | Enables symbolic debugging | Compiler |
| -s | Interlist C statements into assembly listing | Compiler |
| -as | Keep labels as symbols (in asm source) | Assembler |
| -al | Make .lst assembly listing file | Assembler |
| -z | Invokes the linker | Linker |

What special items needs to go in the linker command file?

Runtime-support

Some tasks that a C program must perform (such as memory allocation, string conversion, and string searches) are not part of the C language. The ANSI C standard defines a complete set of runtime-support functions that perform these tasks. The TMS320 fixed-point compiler includes a library that contains ANSI standard runtime-support functions. All of the ANSI functions except those that require an underlying operating system (such as I/O and signals) are provided.

C Runtime Environment

The C language insulates you from the hardware more than assembly language does, but this does not mean that you can ignore many of the low-level details that must be considered in an assembly language program. In fact, a thorough understanding of the runtime environment is essential for you to be able to interface assembly language programs with C, or even for obtaining maximum performance from a pure C program.

To understand the C runtime environment, you will need to understand how a C program is organized in the memory map. To do this, we will look at the individual sections that make up a C program. In addition, one of the fundamental concepts of C is that variables local to a function, as well as temporary values needed by the compiler for expression analysis, exist only during the life of the function. This is referred to as the scope of a variable. Two local variables (referred to in C as automatic variables) can have the same name, but be totally independent of each other as long as they are used in different functions. The dynamic allocation of memory for these local variables is managed on the stack, and we will see how this is done.

Sections Created by the Compiler

To comprehend how the compiler utilizes memory, you need to understand the seven sections that are produced by the compiler. These sections can be divided into initialized and uninitialized sections. A program that mixes assembly language routines with C might have a `.data` section and other named sections defined by the assembly language program.

COFF Sections used by the Compiler

| Section | Contents | Type | Link to ... |
|----------------------|---|---------------|--------------|
| <code>.text</code> | code | initialized | prog space |
| <code>.cinit</code> | data tables to initialize global and static variables | initialized | prog space |
| <code>.switch</code> | table for switch statements | initialized | prog space |
| <code>.const</code> | data constants declared by <code>const</code> | initialized | prog space * |
| <code>.bss</code> | global and static variables | uninitialized | data space |
| <code>.stack</code> | C system stack | uninitialized | data space |
| <code>.system</code> | C system heap (used by <code>malloc()</code>) | uninitialized | data space |

* `.const` linking options discussed on upcoming slides

➤ Hints on linker command files:

- Develop one generic linker command file to describe the anticipated systems memory map and placement of sections. Start by copying one from this workshop module.
- Use this command file throughout the development process. Only modify it when integrating with assembly source modules containing user defined section names.
- At the end of development, if necessary, "tweak" placement of the sections to optimize performance.

Linking Code to Specific Hardware

The linker command file provides the means to control the linker. Without it, the source code is linked in a non-optimal fashion using default section locations: placing all code and variables into the lowest memory addresses available which effectively ignores the on-chip memory resources. This creates run-time bus contention between program and data accesses. These bottlenecks reduce overall system throughput. Therefore, the user must create a linker command file which specifies precise placement of sections thus avoiding these bottlenecks.

The structure of linker command files remains the same as described during the COFF Tools module. When linking C code, though, the following considerations must be observed.

Example Linker Command File

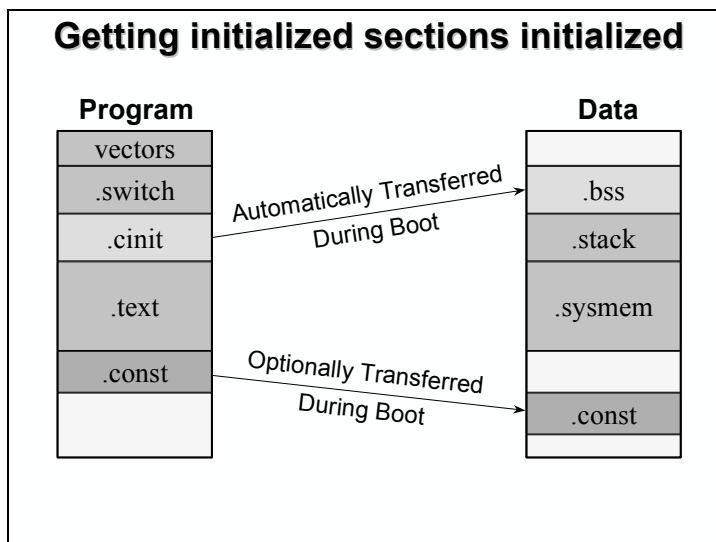
```
MEMORY
{
    PAGE 0:
        VECS:    org = 0000h , len = 0040h
        FLASH:   org = 0040h , len = 1FC0h
    PAGE 1:
        B2:      org = 0060h , len = 0020h
        I_RAM:   org = 0200h , len = 0200h
}

SECTIONS
{
    .text:    > FLASH    PAGE 0 /* code                */
    .switch:  > FLASH    PAGE 0 /* for case stmts */
    .cinit:   > FLASH    PAGE 0 /* global inits    */
    .const:   > FLASH    PAGE 0 /* const int       */
    .vectors: > VECS     PAGE 0 /* vectors         */
    .bss:     > B2       PAGE 1 /* variables       */
    .stack:   > I_RAM    PAGE 1 /* for SP          */
    .system:  > I_RAM    PAGE 1 /* heap, dynamic mem */
    .data:    > I_RAM    PAGE 1 /* for .asm file   */
}
```

1. Set both the stack and heap sizes
(using the `-stack` and `-heap` options)
2. Allocate the seven *sections* produced by the C compiler into memory. These include four initialized (ROM-like) sections and three uninitialized (RAM-like) sections.

Allocating `.const`

Allocating the `.const` section to memory is a bit trickier than the other sections. Remember, `.const` must be initialized at reset (it's ROM-like). It is often expensive and inconvenient to have non-volatile memory (e.g., OTP, EPROM) located in data memory space for the sole purpose of `.const`. Here are some options you can consider when allocating `.const`:



Options for Allocating `.const`

Problem: `.const` is an initialized section that is accessed in the data space

1. Link to the program FLASH, and use the compiler initialization routine (`boot.asm`) to copy it into data RAM during C-environment setup
2. Use static declarations instead of `const` to place constants in the `.bss` section. Initialization automatically performed from `.cinit` section.
3. Use external non-volatile memory in the data space
4. Link to internal or external data RAM. Requires a run-time loader (e.g. debugger)

Options #1 and #2 are most common

Reset in C

BOOT.ASM

The C environment is initialized at reset by running a routine called BOOT.ASM (from RTS2xx.lib). This file *must* be executed since it calls `main`, i.e. your code.

C requires specific registers to be initialized in order to set up the C environment, such as the stack (AR1) and frame (AR0) pointers. Variables in RAM must also be initialized depending on their declarations in C code (e.g. “`int x=25;`”). The C boot routine, which is a function called `c_int0`, is located in the run-time support library (rts2xx.lib) in the file BOOT.ASM and must be linked in with the C object modules.

Setting Up the C Environment: boot.asm

The runtime support library rts2xx.lib contains boot.asm:

1. Creates and initializes the C stack in data memory
2. Creates and initializes the heap
3. Initializes the status registers to compiler expected values
4. Initializes global and static variables
 - Copies values from .cinit to addresses assigned to each variable in the .bss
5. Optionally copies .const from program ROM to data RAM
6. Calls `main()` function to begin running your C program

C Software Stack

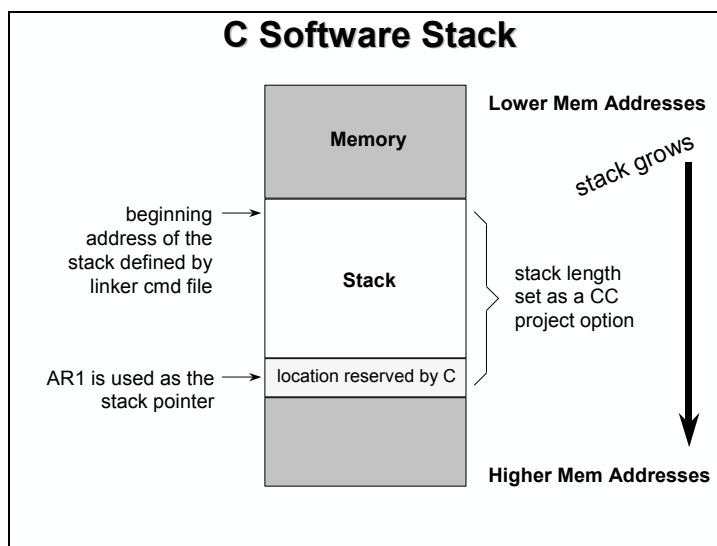
The C compiler builds a C software stack during initialization and contains arguments and local variables during run-time.

The compiler creates the stack in `boot.asm` using:

```
.usect ".stack", __STACK_SIZE
```

The stack size is defined in the linker command file using the `-stack` option. The default size is 1K words. Location of `.stack` is defined in the linker command file similar to other COFF sections.

The compiler uses register AR1 as the Stack Pointer (SP) which is also loaded at run-time by the `boot.asm` C initialization routine. SP indicates the current top-of-stack (TOS) location. A graphical view of this shows:



The C software stack is separate from the fast hardware stack built into each TMS320C2x/2xx/5x device. The hardware stack will only contain up to 8 program counter (PC) values during function calls or when interrupts occur. Due to its small size, the compiler saves the return address from the hardware stack to the C software stack for each function call. The return address is restored to the hardware stack before the function's return occurs. The compiler uses the PSHD and POPD commands to perform the return address context save/restore..

C Software Stack Summary

- The stack grows in memory *from lower to higher* memory addresses
- The SP *usually* points to the first empty location on the stack
- The location of the stack is defined in the linker command file using:
`.stack :> RAM`
- The stack size is defined linker option:
`-stack xxxx`
where *xxxx* is the size (default is 400h, or 1K decimal)

Note: The compiler reserves the first empty location for occasional temporary storage. You must increment the stack pointer when using it for assembly-coded ISR functions.

Register Usage

The following tables describe how the various registers are used by the C compiler. Your called (assembly) function might need to save specific registers as indicated in the table.

Status Register Usage Conventions

| Field | Name | Compiler expects | Compiler may modify |
|-------|----------------------------|------------------|---------------------|
| ARP | Auxillary register pointer | 1 | Yes |
| C | Carry | - | Yes |
| DP | Data Page | - | Yes |
| OV | Overflow | - | Yes |
| OVM | Overflow mode | 0 | No |
| PM | Product shift mode | 0 | No |
| SXM | Sign extension mode | - | Yes |
| TC | Test control bit | - | Yes |

Note: If the user modifies a "Compiler expects" value, this value must be restored by the function

The compiler users guide discusses the settings it applies for the status register fields (ST0, ST1). In your assembly routines, it is always safer to set any status bits to the values you need; better safe than sorry.

Local Variables and Frames

The TMS320 compiler reserves space for local variables on the stack as each function is called. This area reserved by each function is called a *frame*. If many functions are nested, the stack will contain many frames, one right after the other.

Your assembly language function should follow this same usage method for local variable storage. (We'll discuss how this works later.)

Variables within a function that are declared with `.bss` or `.usect` will act as either a global or static variable. Each time the function is called, the variable still exists and will contain any previous value.

Linking BOOT.ASM

In order for your system to run properly from reset, the following two tasks must be accomplished:

1. Setting up the vector table.

How do we run BOOT.ASM?

- 1. Modify vectors.asm so that reset branches to `_c_int0`**

```
.ref _c_int0
.sect "vectors"
reset: B _c_int0
```

- 2. Specify `rts2xx.lib` as a library for your project in Code Composer**

2. Modifying the linker command file:
 - a) Specify linker option `-c` (or `-cr`) tells BOOT.ASM whether or not to initialize global and static variables (`.bss`).
 - b) Link-in 'C2xx run-time support library `rts2xx.lib`
 - c) Link vectors section to reset location.

ROM and RAM Initialization Models (-c and -cr)

When linking a C program, use either the `-c` or `-cr` option. These options tell the compiler whether to initialize global and static variables.

If `-c` is used (which is the default), all global and static variables will be initialized during run-time. The initial value of a variable is allocated in the `.cinit` section and BOOT.ASM copies these values to the proper location in `.bss`. For example:

```
int    x = 25;
```

If `x` is a global variable, the value 25 will be located in the `.cinit` section and will be copied to the address of `x` which is located in the `.bss` section. All globals and statics are initialized in this way during run-time.

If the `-cr` option is used, the initialization from `.cinit` to `.bss` is *not* performed. Some other device – e.g., coprocessor, boot-loader, or debugger – must initialize the RAM prior to code execution. In some systems, this might save initialization time and some program memory space

versus using `-c`. More information regarding `-c` and `-cr` can be found in the *TMS320C2x/2xx/5x Optimizing C Compiler User's Guide*.

Using *BOOT.ASM* to copy *.const* into data memory

To use this option you must extract the *BOOT.ASM* routine and modify the line:

```
CONST_COPY          .set    0  
  
to:                  CONST_COPY      .set    1
```

This tells *BOOT.ASM* to copy the `.const` section from Program ROM to Data RAM at reset.

This option was discussed in the *Allocating .const* section earlier for in this module. For more information see the *TMS320C2x/2xx/5x Optimizing C Compiler Users Guide*.

Link your own system initialization code

If you must perform your own system initialization routines such as: memory/system testing, interrupt initialization, etc., there are three ways to do this:

1. Write a C or assembly function performing your system initialization and call it first thing within `main()`.
2. Point the reset vector to your own initialization routine; then call `_c_init` from your routine.
3. You can replace or modify the *BOOT.ASM* routine by extracting it from the library, making the necessary modifications and then placing it back into the library. However, the boot routine *must* perform its original initialization steps.

Mixing C and Assembly

System developers often would like to work entirely in C code. Its modularity and portability provide a standard software model for maintainable systems. There are times, though, when it makes good sense to use assembly language rather than C; this most often occurs when maximum performance is the utmost need or the functions needing control are beyond the scope of C. Some examples include:

- Interrupts
- Peripherals
- Reset Initialization
- Processor status control

This module focuses on mixing C and Assembly from the standpoint of:
C calling Assembly

Assembly functions could just as easily call C functions. The rules for register usage and argument passing discussed below would simply need to be applied in reverse.

Background

Before we discuss creating and using C-callable assembly language functions, we need to review three important areas concerning how the C environment operates:

- Register Usage by C
- Local Variables and Frames
- C Software Stack

Calling an Assembly Function from C

- ◆ Setup the calling C function
- ◆ Setup the assembly function
- ◆ Upon entry, the assembly function must:
 - Save necessary C environment registers
 - Manage the C stack
 - Access any passed arguments
- ◆ Before returning, the assembly function must:
 - Setup the return value (if one is returned)
 - Restore previous frame and registers

Steps Required for Calling Assembly Functions

There are five steps to creating and using C callable assembly language functions:

1. Setting up C environment
 - Prototype the assembly language function
 - Call the assembly language function
2. Setting up assembly environment
 - Declare the function as a global
 - Define the function name
3. Entering the function
 - Save previous frame and create a new local frame
 - Save any necessary registers or status bits
4. Accessing data
 - Passed arguments
 - Local variables
5. Exiting the function
 - Restoring previous frame and registers
 - Returning a value

1. Set-up C Environment

The function should be prototyped so that the C environment knows how to handle the returned value and check the type of passed arguments. If the function is not prototyped, the C compiler assumes that the function returns an int. The following figure includes a simple C example which declares and calls the assembly language function `func()`.

Setting Up the Calling C Function

- ◆ Add the function prototype
- ◆ Call the assembly language function

```
int func(int, int); /* function prototype */

void main (void)
{   int x = 7;
    int y;

    y = func(x, 5); /* call function */
}
```

2. Set-up Assembly Environment

The function name is simply the label at the beginning of the assembly routine. In order for the linker to resolve any calls to the function, the label for the function must be defined using `.global` or `.def` in the assembly language module where the function is defined. **NOTE: C variables and function names are accessed in assembly language using the variable or function name with a preceding underscore (_).** Continuing the last example, the following figure includes an assembly language code segment that defines the entry point to the function and makes the function name visible to the linker using the `.def` assembler directive.

Setting Up the Assembly Function

- ◆ Declare function name as a global
- ◆ Define the function name using an underscore

```
.def    _func    ;make the label _func
                ;visible to linker

_func:                ;define function's entry point
    nop
    nop
    nop
    ...
```

3. Entering a Function

Saving Necessary C Environment Registers

| Register | Usage | Called Function Must Preserve? |
|-------------|-------------------------|--------------------------------|
| AR0 | Frame Pointer | Yes |
| AR1 | Stack Pointer | Yes |
| AR2 | Local Variable Pointer | No |
| AR3 - AR5 | Expression Analysis | No |
| AR6, AR7 | Register Variables | Yes |
| Accumulator | Expression/Return Value | No |
| P | Expression Analysis | No |
| T | Expression Analysis | No |
| ST0 | Status Register 0 | No* |
| ST1 | Status Register 1 | No* |

* See "Status Register Usage Conventions" slide for required bit settings prior to return from function

Here's a look at the "system" before the C routine calls a function:

Managing the C Stack: Stack before function call is made

```
y = fun(x,5); /* function call */
```

Hardware Stack



AR0 FP →
AR1 SP →

C Stack

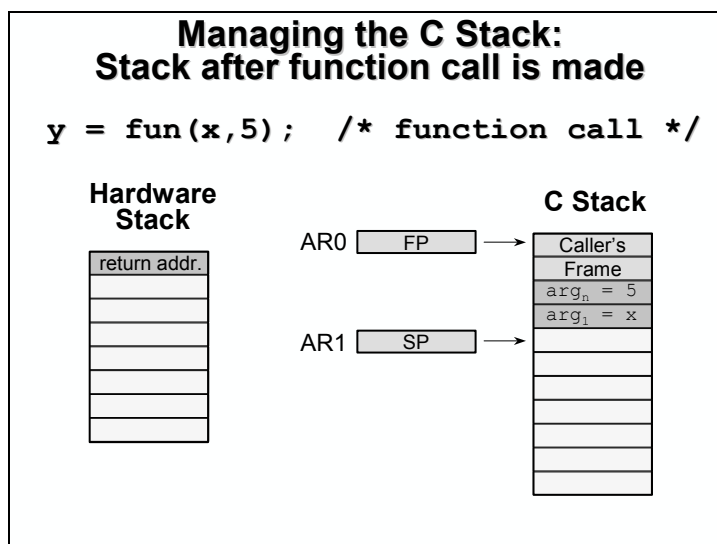


After entering the function, the arguments are available to the called function on the C stack. Therefore it is important to understand how to manage the C stack upon entry to the called function. Here are the conditions when entering the function:

```
func(arg1, arg2, ..., argn)
```

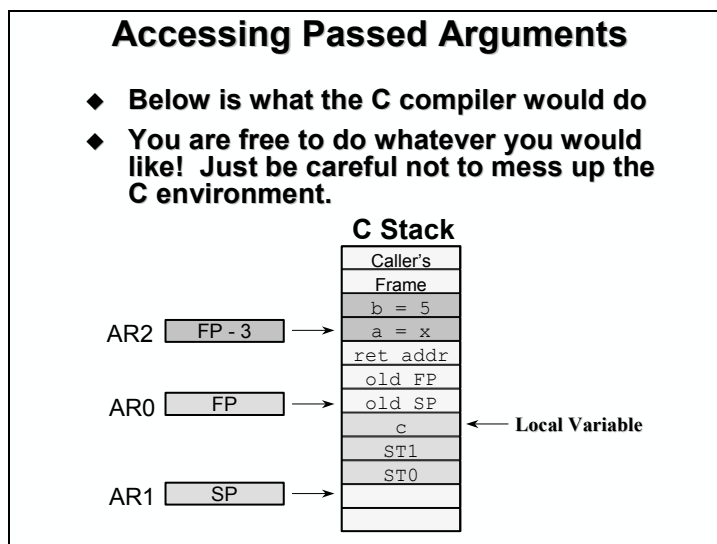
- Stack pointer (AR1) is active (ARP=1)
- Arguments "pushed" on C stack in reverse order
- AR0 used as Frame Pointer; points to previous function's frame
- AR2 to be used for accessing local data
- Return address on hardware stack

Here's a look at the "system" after the C routine calls a function:



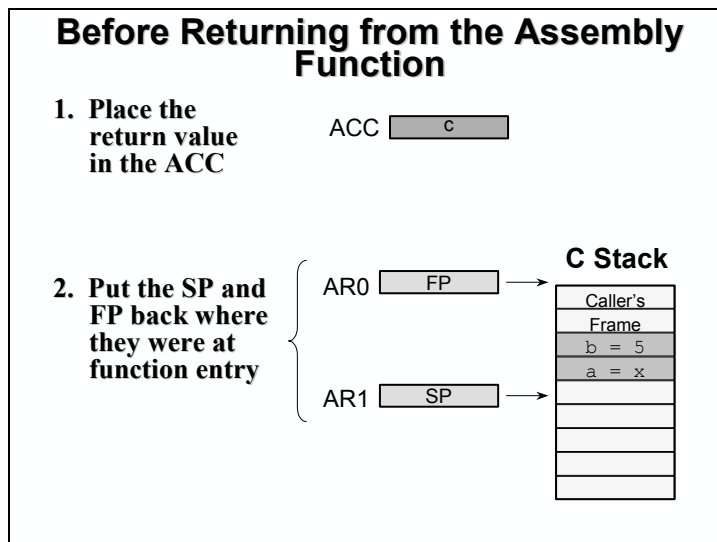
4. Accessing Data

AR2 is typically used to access data, i.e. arguments and local variables. Here we access "a":



5. Exiting the Function

Following completion of the functions, context must be restored. The diagram showing the systems state as the return (RET) is executed follows.



Compiler Optimizations

The C240x compiler uses a wide variety of optimization techniques to improve the execution speed as well as reduce the code size. While optimization techniques are applied throughout the compiler, the separate pass optimizer provides most of the aggressive optimizations that are applied to the source code. The optimizer produces a high level of code motion, or rearranging of the source, and, as a result, it is usually best to develop and debug your program without using the separate pass optimizer. This makes it much easier to compare the execution to the original source file. Once the fundamental program is confirmed to be correct, the optimizer can be used to increase the execution speed and reduce the code size. A full description of the types of optimizations performed by the compiler is listed in the *TMS320 Fixed-Point DSP Optimizing C Compiler User's Guide*.

Optimization Levels

Using the Compiler Optimizer

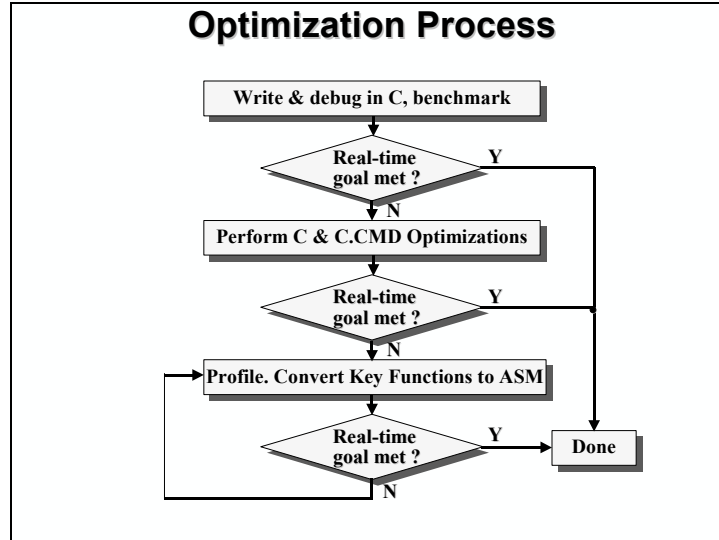
- ◆ **Four levels of optimization**

| LEVEL | OPTIMIZATIONS |
|-------|---------------|
| -o0 | Register |
| -o1 | -o0 + Local |
| -o2 | -o1 + Global |
| -o3 | -o2 + File |
- ◆ **You must specify an optimization level as a CC project option, otherwise the optimizer is not used**
- ◆ **See the Compiler User's Guide for details on what optimizations are performed**

It is usually best to get the program debugged and verified prior to using the optimizer. After optimization, you should again verify correct operation of the program.

When the `-g` option is used to enable symbolic debugging, the compiler is prevented from making the normal optimizations that are done outside the separate optimizing pass. This is done to make the code more readable and to ensure that no rearranging is done by the code generator. If the `-mn` option is included, then the compiler is allowed to generate the identical code that it would have produced without the `-g` option. While the code may be a little less readable, it is generally good procedure to have the compiler produce uniform code whether symbolic debug is enabled or not. In the long run, this may save some time spent in finding problems in your code.

Optimization Steps



Write clear, fast algorithms

In general, clearly written code, using the fastest underlying algorithm is the code the compiler handles best. Code that is easy for a person to understand is usually easier for the compiler to understand as well (global data flow analysis, etc.). Remember, no compiler can fix a slow algorithm!

Use the Optimizer (-o)

The C240x compiler uses a wide variety of optimization techniques to improve code execution speed as well as reduce code size. While optimization techniques are applied throughout the compiler, the separate pass optimizer provides most of the aggressive optimizations that are applied to the source code. These include both general and device specific optimizations.

Optimization is invoked at one of four levels:

| Level | Optimizations |
|----------|---------------|
| -o0 | Register |
| -o1 | -o0 + Local |
| -o2 (-o) | -o1 + Global |
| -o3 | -o2 + File |

➤ How to use the optimizer

At each step below, if the code meets your 'real-time' requirement, leave it alone and move onto the next project. Find the bugs — if any exist — before it's been shuffled around by the optimizer.

How to Use the Optimizer

1. Write, compile, and debug code without using the optimizer: Get the code functioning!
2. Turn on the optimizer and verify code functionality. Debug if necessary.
3. Finally, enable all optimizations

Use Normal Optimization (`-mn`)

The optimization pass of the compiler produces a high level of “code motion”, or rearranging of the source; therefore, the optimized output code may look quite different from the input source code. As a result, *it is usually best to develop and debug the program without using the separate pass optimizer*. This makes it much easier to compare the execution to the original source file during C code debugging.

The TI tools aim to make C source level debugging easier. When symbolic debug is enabled, using the `-g` option, certain code movement optimizations are disabled. This allows the resulting code to be compiled in a more linear, easy to follow, fashion.

Once the fundamental program is confirmed correct, all optimizations can be re-enabled using the `-mn` (normal optimizations) option. This option may bring on problems during debug: tracing code in mixed mode and the call trace back window won't always work. While it's possible to debug code after compiling with `-mn`, some risks still exist.

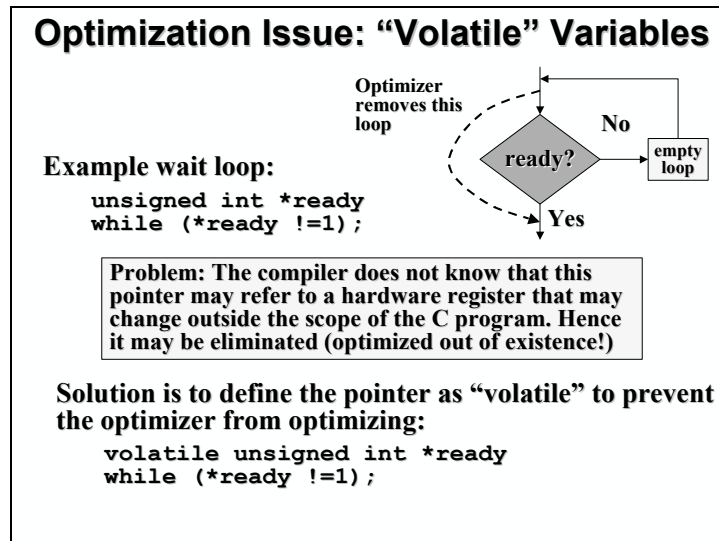
To summarize this:

| Options | What they do ... |
|------------------|---|
| <code>-o</code> | Turns on optimizations |
| <code>-g</code> | Turns on symbolic debugging, but turns off some optimizations |
| <code>-mn</code> | Overrides the optimization disabling effects of <code>-g</code> |

Using the volatile keyword

It is possible that your program will fail to execute correctly when the optimizer is used. This is not an unusual problem with highly optimizing compilers. There is one situation that will almost always cause the optimized code to fail, and if this is understood ahead of time, then the debug process will be greatly simplified.

The optimizer analyzes data flow to avoid memory accesses whenever possible. If you have code that depends on memory accesses occurring exactly as written in the C code, you must use the volatile keyword to identify these accesses. The compiler will not optimize out any references to volatile variables.



In this example, `*ctrl` is a loop-invariant expression; i.e., the expression never changes during execution of the loop. In this case, the optimizer will reduce the loop to a single memory read. This is not what is intended in the case of this busy-waiting loop. This kind of code is very common in a control-type application. To prevent the optimizer from essentially optimizing this busy-waiting loop out of existence, you should use the volatile keyword as shown.

This one type of optimization is the source of most problems that occur during optimization. By being aware of the situation, you can have your optimized code running correctly with much less effort.

Use Function Inlining (`-x`)

The ANSI C feature, function inlining, is another optimization method. When used, the C compiler inserts a copy of each “inlined” function directly into the source code. This saves the overhead of transferring program control to and from the function. Additionally, it allows the compiler to optimize C code across function call boundaries. The tradeoff, inlining usually results in larger code.

Optimization Level 3 automatically invokes function inlining. By default, any function up to ten lines long which is located in the file where it’s called will be inlined. These defaults can be overridden with compiler options.

Additionally, you can specifically force the compiler to inline a function using the *inline* keyword and the `-x` option.

➤ Steps to Inlining Functions

1. Declare function using the *inline* keyword:

```
inline int myfunction(a,b,c) { }
```

2. Invoke the compiler using the `-x` option.

Provide Compiler with Project Visibility

Provide the compiler with as much visibility to the project as possible. If possible, create a single source file by include files. If the project must be distributed over many files, set them up to be included into a single source file. The compiler, operating on a single source file, can optimize over the top of functions — even inlining them — rather than only inside them.

Replace Critical Routines

Replace your critical routines; i.e., those requiring the most time. Once you’ve selected your critical routines, you must replace them with the optimized versions.

Use the Profiler

The *profiler* — included with TI’s debuggers — runs your code and provides a relative summary detailing which code is run most often. Examining the code in this fashion directs you to which routines most require optimization.

Buy or Download Routines

Use hand optimized versions of your C functions. Standard DSP functions — FFT, FIR, IIR, etc. — can receive large performance gains because only assembly can access specific hardware features available on DSPs. For example, it’s hard to argue against assembly when special addressing modes (e.g., bit-reversed) can provide a 10x speed improvement.

Many standard functions are available commercially — from third party developers — or can be found on-line. Good places to look for routines or referrals are:

- TI's DSP Web Site
- TMS320 Development Support Reference Guide
- TMS320 Third Party Guide
- TMS320 Software Cooperative

Even if the specific routine you need is not available, you might find something that could be quickly modified. Also, many tools exist for designing and coding specific routines, such as, filter design packages.

Write in Assembly

When performance is the critical objective and you've exhausted all the other optimization suggestions, you must consider writing the function in assembly. The upcoming section covers various issues associated with writing and mixing assembly code within the C environment.

Other Items

Accessing Memory and Peripheral Locations

The most common reason for accessing memory locations is to read or modify the memory-mapped and peripheral registers. Embedded systems also communicate with external peripherals and message-passing registers using a similar memory mapped I/O strategy. The C240x User's Guide lists the memory-mapped and peripheral registers and their associated addresses.

C Tidbits: Accessing Peripheral Registers

1. Define the address using #define

```
#define WDKEY (volatile unsigned int *)0x7025
```

2. Read and write like any other pointer

```
*WDKEY = 0x0055;
```

Use the *volatile* modifier if the register changes independent of C program control otherwise in some cases the optimizer might remove the line-of-code.

asm() Function

C programs can contain assembly language statements. Inlined assembly statements may be useful to insert assembly comments, modify processor status, or place additional assembly labels into the code.

An assembly language statement is inlined, or inserted, into C code using the `asm()` command:

C Tidbits: Inline Assembly

Must leave at least one space!

```
int x,y;  
asm ("  CLRC INTM  ;enable interrupts");  
y = 15 * x;  
asm ("  SETC INTM  ;disable interrupts");
```

Be aware that the optimizer will not move code over an inline assembly statement

While inlined assembly statements are a powerful tool, be careful not to disrupt the C environment or issue assembler directives. The C compiler does not process inline assembly statements; it merely copies them into the final assembly language file. Also, remember that the first space after the quotation mark (") is column 1. Only labels or a comment are allowed in column 1. In the example above, remember that the mnemonic cannot be in column one.

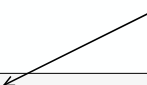
In general, inline assembly statements are most useful for inserting comments or for non-critical register-based tasks from within the C environment.

Note: When the optimizer is enabled, the compiler will most likely produce a great deal of code movement. Inlined assembly statements are only copied into the output file, *after* optimization has taken place, hence the code may move around the fixed `asm()` statements. If the `asm()` statement was placed in a critical position relative to the function of the C code, it may actually execute at the wrong time. This is the real disadvantage of inlined assembly.

ISR in C

C Tidbits: ISR in C

Interrupt keyword



```
interrupt void gisr1()
{
    /* isr C code goes here */
}
```


Introduction

This module describes the various TMS320 development tools and explains where to find more information.

Learning Objectives

Learning Objectives

- ◆ Describe TMS320 development tools
- ◆ Explain where to find more information

Module Topics

Development Support16-1

Module Topics.....16-2

Development Support.....16-3

Development Support

TMS320F/C240x Development Tools

- ◆ **Code Generation Tools**
 - ◆ Assembler, Linker, C-Compiler, Simulator
- ◆ **Emulation Debug Tools**
 - ◆ XDS510 / 510PP, Code Composer
- ◆ **Evaluation / Starter Kits**
 - ◆ Evaluation Modules (EVM), DSP Starter Kit (DSK), Motion Control Kit (MCK)
- ◆ **Flash Programming Utilities**
- ◆ **Other Tools**
 - ◆ DMC Software Library, COFF to DSK Translator, Emulation Porting Kit

Internet

For More Information . . .

Website: www.ti.com

- ◆ Device information
- ◆ Application notes
- ◆ Technical documentation
- ◆ TI & ME
- ◆ News and events
- ◆ Training

Product Information Center (PIC)

Phone: 972-644-5580

Email: sc-infomaster@ti.com

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

Other Resources

Literature Center: 800-477-8924

Software Registration/Upgrades: 972-293-5050

Hardware Repair/Upgrades: 281-274-2285

Enroll in Technical Training: www.ti.com/sc/training
(choose Design Workshops)

This page is left intentionally blank.

Introduction

Space vector PWM refers to a special switching scheme of the six power transistors of a 3-phase power converter. This module explains the details of space vector PWM and how to implement it on the C240x family.

Learning Objectives

Learning Objectives

- ◆ Understand Space Vector PWM

Module Topics

Appendix AA-1

Module Topics..... A-2

Space Vector PWM (full compares only) A-3

Space Vector PWM (full compares only)

What is Space Vector PWM?

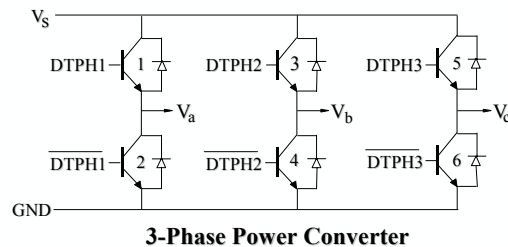
- ◆ Special switching scheme for 3-phase power converters
- ◆ The SV PWM is symmetric on C240x
- ◆ Generates minimum harmonic distortion in motor phase currents
- ◆ More efficient use of supply voltage than sinusoidal PWM modulation

Basic Space Vector Determination (1 of 3)

Only states of transistors 1, 3, & 5 need be determined since 2, 4, & 6 are their respective compliments

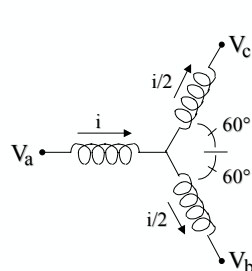
Switching State Notation: (Q5,Q3,Q1)

e.g. (0,0,1) means gate 1 is on, gates 3 & 5 are off

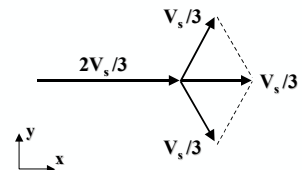


Basic Space Vector Determination (2 of 3)

Example: (Q5,Q3,Q1) = (001) $\Rightarrow V_a=V_s, V_b=V_c=\text{GND}$



Y-Connected Motor Windings
Showing Current Flow

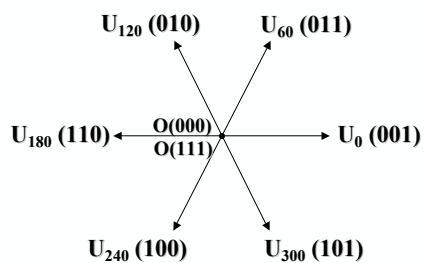


$$x - dir: \frac{2V_s}{3} + 2 \cdot \frac{V_s}{3} \cos(60) = V_s$$

$$y - dir: \frac{V_s}{3} \sin(60) - \frac{V_s}{3} \sin(60) = 0$$

Voltage Drop Vectors

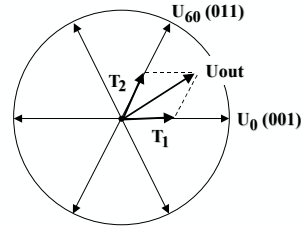
Basic Space Vector Determination (3 of 3)



Basic Space Vectors w/ Switching Patterns

Space Vector PWM Goal

- ◆ Approximate desired voltage drop vector as a linear combination of the basic space vectors
- ◆ Coefficients are duration times



$$U_{out} = \frac{T_1}{T_p} \cdot U_x + \frac{T_2}{T_p} \cdot U_{x+60} + \frac{T_0}{T_p} \cdot [O(000) \text{ or } O(111)]$$

where $T_0 = T_p - T_1 - T_2$ and $T_p :=$ PWM carrier period

Software Responsibilities for SV PWM

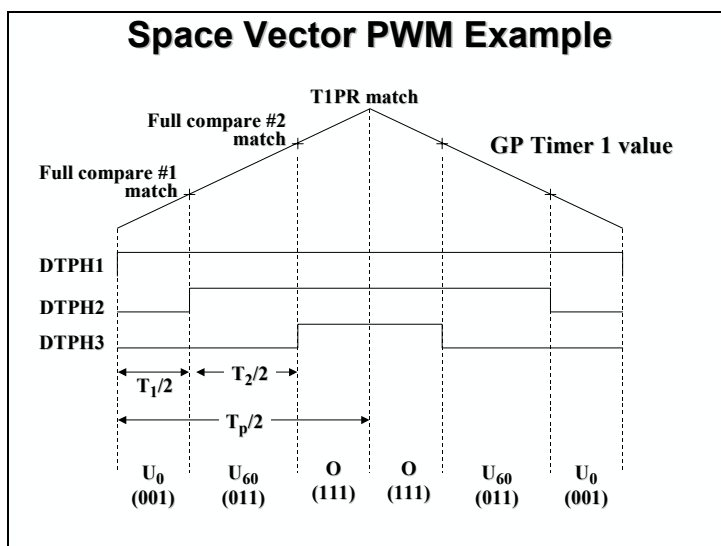
- ◆ Determine the SV sector (i.e. U_x and U_{x+60})
- ◆ Determine the parameters T_1 , T_2
- ◆ Choose desired SV rotation direction
 - Write U_x pattern to ACTR.14-12 and set ACTR.15 = 0
 - OR
 - Write U_{x+60} pattern to ACTR.14-12 and set ACTR.15=1
- ◆ Put $T_1/2$ in CMPR1 register
- ◆ Put $(T_1+T_2)/2$ in CMPR2 register

There are eight basic space vectors for a 3-phase power converter. Six of the vectors have unit magnitude and are positioned at 60 degrees intervals on the unit circle. The remaining two vectors, O(000) and O(111), have zero magnitudes. They represent the fact that if the three phases V_a , V_b , and V_c are all either turned on or turned off, no energy transfer occurs between the motor and power supply. Note that the basic space vectors represent voltage direction. Voltage magnitude is determined by the power supply.

The goal of space vector PWM is to generate the desired voltage vector as a time weighted vector sum of one or more basic space vectors. The effectiveness of this time-averaging stems from the low-pass filter characteristic of motors, which provides the justification for all types of PWM schemes. Consider that if the U_0 switching pattern was active for 100% of each PWM period, then 100% of V_s would be provided to the motor in the U_0 direction all of the time. Alternately, if the U_0 and U_{60} patterns were each active for 50% of each PWM period, vector addition shows the resultant as $0.866V_s$ in a direction exactly half-way between the U_0 and U_{60} vectors. This magnitude turns out to be the worst case in that at least $0.866V_s$ can be obtained in any direction by time-averaging the adjacent space vectors. Now suppose that an average delivered voltage of less than the maximum obtainable was desired. One could use the two zero space vectors O(000) and O(111) to fill out the PWM periods. For example, if the U_0 and U_{60} patterns were each active for 25% of each PWM period with O(111) active for the remaining 50%, then an average voltage of $0.433V_s$ would be provided to the motor in a direction exactly half-way between the U_0 and U_{60} vectors. By increasing the supply source to $1.155V_s$ (i.e. $1.155 = 1/0.866$), from 0% to 100% of V_s could be provided to the motor in any direction.

The C240x automatically decides which one of O(000) and O(111) to use for filling out a PWM period. It chooses the pattern that differs from the previous switching state by only 1 bit (one of these switching states can always be reached by changing just one bit of any U_x pattern. In addition, notice that the switching patterns of any two adjacent space vectors U_x and U_{x+60} also differ by only one active transistor. Because of these relationships and the fact that the C240x implements space vector PWM symmetrically, each of the six power converter transistors will be switched an equal number of times as U_{out} rotates 360 degrees around the unit circle. This distributes switching losses and stresses uniformly over the six transistors.

While symmetric space vector PWM can be implemented using software, doing so requires five different interrupt services during any given PWM period (one at the beginning, and one at each of the four compare matches). Each of these interrupts wastes a significant number of CPU cycles on latency and context saving, and at high PWM carrier frequencies the entire control scheme can become untenable. The hardware implementation of the C240x avoids this problem.



Initializing Space Vector PWM

1. Set ACTR 11-0 to define active states of PWM pins
2. Configure COMCON register
 - Enable space vector operation
 - Put all three full compares in PWM mode
 - Set ACTR reload condition to underflow
 - Set CMPR reload condition to underflow
3. Put GP Timer 1 in continuous-up/down mode to start SV operation
4. Control loop handles Software Responsibilities

This page is left intentionally blank.

Appendix B

Module Topics

Appendix B B-1

Module Topics..... B-2

TMS320LF2407 EVM B-3

 LF2407 EVM Connector / Header and Jumper Diagram B-3

 P1 – Expansion I/O Connector B-4

 P2 – Expansion Analog Connector B-5

 P3 – Expansion Address and Data Connector B-6

 P4 – Expansion Control Connector B-7

 Jumpers..... B-8

F2407.h B-10

P1 – Expansion I/O Connector

| Pin # | Signal | Pin # | Signal |
|-------|-------------------|-------|-------------------|
| 1 | VCC, +5 Volts | 2 | VCC, +5 Volts |
| 3 | PWM1/IOPA6 | 4 | PWM2/IOPA7 |
| 5 | PWM3/IOPB0 | 6 | PWM4/IOPB1 |
| 7 | PWM5/IOPB2 | 8 | PWM6/IOPB3 |
| 9 | PWM7/IOPE1 | 10 | PWM8/IOPE2 |
| 11 | PWM9/IOPE3 | 12 | T1PWM/T1CMP/IOPB4 |
| 13 | T2PWM/T2CMP/IOPB5 | 14 | T3PWM/T3CMP/IOPF2 |
| 15 | * TDIRA/IOPB6 | 16 | * TCLKINA/IOPB7 |
| 17 | GND | 18 | GND |
| 19 | BOOTEN-/XF | 20 | * BIO/IOPC1 |
| 21 | * CAP1/QEP1/IOPA3 | 22 | * CAP2/QEP2/IOPA4 |
| 23 | * CAP3/IOPA5 | 24 | * CAP4/QEP3/IOPE7 |
| 25 | RESERVED | 26 | * PDPINTA- |
| 27 | SCITXD/IOPA0 | 28 | * SCIRXD/IOPA1 |
| 29 | * SPISIMO/IOPC2 | 30 | * SPISOMI/IOPC3 |
| 31 | * SPICLK/IOPC4 | 32 | * SPISTE/IOPC5 |
| 33 | GND | 34 | GND |

* Signal is interfaced through a quick switch to allow 5 volt tolerant inputs.

P2 – Expansion Analog Connector

| Pin # | Signal | Pin # | Signal |
|-------|------------------|-------|---------------------|
| 1 | VCCA, +5V Analog | 2 | VCCA, +5V Analog |
| 3 | TMS2/IOPD7 | 4 | * IOPF6 |
| 5 | ADCIN2 | 6 | ADCIN3 |
| 7 | ADCIN4 | 8 | ADCIN5 |
| 9 | ADCIN6 | 10 | ADCIN7 |
| 11 | ADCIN8 | 12 | ADCIN9 |
| 13 | ADCIN10 | 14 | ADCIN11 |
| 15 | ADCIN12 | 16 | ADCIN13 |
| 17 | AGND | 18 | AGND |
| 19 | ADCIN14 | 20 | ADCIN15 |
| 21 | VREFHI | 22 | VREFLO |
| 23 | ADCIN0 | 24 | ADCIN1 |
| 25 | DACOUT1 | 26 | DACOUT2 |
| 27 | DACOUT3 | 28 | DACOUT4 |
| 29 | RESERVED | 30 | RESERVED |
| 31 | RESERVED | 32 | XINT2-/ADCSOC/IOPD1 |
| 33 | AGND | 34 | AGND |

P3 – Expansion Address and Data Connector

| Pin # | Signal | Pin # | Signal |
|-------|--------|-------|--------|
| 1 | A0 | 2 | A1 |
| 3 | A2 | 4 | A3 |
| 5 | A4 | 6 | A5 |
| 7 | A6 | 8 | A7 |
| 9 | A8 | 10 | A9 |
| 11 | A10 | 12 | A11 |
| 13 | A12 | 14 | A13 |
| 15 | A14 | 16 | A15 |
| 17 | GND | 18 | GND |
| 19 | D0 | 20 | D1 |
| 21 | D2 | 22 | D3 |
| 23 | D4 | 24 | D5 |
| 25 | D6 | 26 | D7 |
| 27 | D8 | 28 | D9 |
| 29 | D10 | 30 | D11 |
| 31 | D12 | 32 | D13 |
| 33 | D14 | 34 | D15 |

P4 – Expansion Control Connector

| Pin # | Signal | Pin # | Signal |
|-------|---------------------|-------|-------------------|
| 1 | VCC, +5 Volts | 2 | VCC, +5 Volts |
| 3 | DS- | 4 | PS- |
| 5 | IS- | 6 | WR-/IOPC0 |
| 7 | WE- | 8 | RD- |
| 9 | STRB- | 10 | R/W- |
| 11 | READY | 12 | PDPINTB- |
| 13 | RS- | 14 | TRGRESET- |
| 15 | * PWM10/IOPE4 | 16 | XINT1-/IOPA2 |
| 17 | GND | 18 | GND |
| 19 | XINT2-/ADCSOC/IOPD1 | 20 | CAP5/QEP4/IOPF0 |
| 21 | CAP6/IOPF1 | 22 | VISOE- |
| 23 | CANTX/IOPC6 | 24 | CANRX/IOPC7 |
| 25 | PWM10/IOPE4 | 26 | PWM11/IOPE5 |
| 27 | PWM12/IOPE6 | 28 | T4PWM/T4CMP/IOPF3 |
| 29 | TDIRB/IOPF4 | 30 | TCLKINB/IOPF5 |
| 31 | Expansion CLKIN | 32 | CLKOUT/IOPE0 |
| 33 | GND | 34 | GND |

Jumpers

JP1 - CAN Termination Select

| Position | Function |
|----------|------------------------------|
| 1-2 | Disable Termination Resistor |
| 2-3 | Enable Termination Resistor |

JP2 - CAN Input Select

| Position | Function |
|----------|---------------------|
| 1-2 | CAN Connector, P7 |
| 2-3 | Expansion Connector |

JP3 - Serial RAM Write Protect Select

| Position | Function |
|----------|-----------------|
| 1-2 | Write enabled |
| 2-3 | Write protected |

JP4 - SPI Port Routing Select

| Position | Function |
|----------|--|
| 1-2 | SPI routed to expansion connector/serial ROM |
| 2-3 | SPI routed to P8 data logging connector |

JP5 - Flash/Watchdog Select

| Position | Function |
|----------|---------------------------|
| 1-2 | Disable Flash Programming |
| 2-3 | Enable Flash Programming |

JP6 - MP/MC Select

| Position | Function |
|----------|--|
| 1-2 | Internal ROM/FLASH disabled (microprocessor mode) |
| 2-3 | Internal ROM/FLASH enabled (microcomputer mode) |

JP7 - Analog Power Select

| Position | Function |
|----------|---|
| 1-2 | Selects digital power for analog logic |
| 2-3 | Selects connector P2 as analog power source |

JP8 - VREF HI Select

| Position | Function |
|----------|----------------------------|
| 1-2 | VCCA (+3.3V VrefH) |
| 2-3 | Trim Pot R1 (0-3.3V VrefH) |

JP9 - VREF LO Select

| Position | Function |
|----------|----------------------------|
| 1-2 | Analog Ground (VrefL) |
| 2-3 | Trim Pot R2 (0-3.3V VrefL) |

JP10 - Host Reset Select

| Position | Function |
|----------|------------------------------------|
| 1-2 | Disabled |
| 2-3 | Reset from P4, pin4 (DTR-) enabled |

JP11 - BIO Hardware Handshaking

| Position | Function |
|----------|--------------------------------|
| 1-2 | Disables P6 RTS- to BIO-/IOPC3 |
| 2-3 | Enables P6 RTS- to BIO-/IOPC3 |

JP12 - SCI Receive Select

| Position | Function |
|----------|----------------------------------|
| 1-2 | Enables P6 RXD to DSP SCIRXD/IO |
| 2-3 | Disables P6 RXD to DSP SCIRXD/IO |

JP13 - Clock Input Select

| Position | Function |
|----------|--|
| 1-2 | Selects Onboard Oscillator |
| 2-3 | Selects Pin 31 on Control connector P4 |

JP14 - DTS/RTS Select

| Position | Function |
|----------|-----------------|
| 1-2 | DTS is selected |
| 2-3 | RTS is selected |

JP15 - SPI/SCI Bootloader Selection

| Position | Function |
|----------|----------|
| 1-2 | Use SPI |
| 2-3 | Use SCI |

JP16 - BOOTEN Select

| Position | Function |
|----------|----------------------|
| 1-2 | Disable boot loading |
| 2-3 | Enables boot loading |

F2407.h

```

;~~~~~
;Filename: f2407.h
;
;Last Modified: 10/24/00
;
;Description: LF2407 DSP register definitions
;~~~~~

;Core registers
IMR                .set 0004h    ;Interrupt mask reg
GREG               .set 0005h    ;Global memory allocation reg
IFR               .set 0006h    ;Interrupt flag reg

;System configuration and interrupt registers
PIRQR0            .set 7010h    ;Peripheral interrupt request reg 0
PIRQR1            .set 7011h    ;Peripheral interrupt request reg 1
PIRQR2            .set 7012h    ;Peripheral interrupt request reg 2
PIACKR0           .set 7014h    ;Peripheral interrupt acknowledge reg 0
PIACKR1           .set 7015h    ;Peripheral interrupt acknowledge reg 1
PIACKR2           .set 7016h    ;Peripheral interrupt acknowledge reg 2
SCSR1             .set 7018h    ;System control & status reg 1
SCSR2             .set 7019h    ;System control & status reg 2
DINR              .set 701Ch    ;Device identification reg
PIVR              .set 701Eh    ;Peripheral interrupt vector reg

;Watchdog timer (WD) registers
WDCNTR            .set 7023h    ;WD counter reg
WDKEY             .set 7025h    ;WD reset key reg
WDCR              .set 7029h    ;WD timer control reg

;Serial Peripheral Interface (SPI) registers
SPICCR           .set 7040h    ;SPI configuration control reg
SPICTL           .set 7041h    ;SPI operation control reg
SPISTS           .set 7042h    ;SPI status reg
SPIBRR           .set 7044h    ;SPI baud rate reg
SPIRXEMU         .set 7046h    ;SPI emulation buffer reg
SPIRXBUF         .set 7047h    ;SPI serial receive buffer reg
SPITXBUF         .set 7048h    ;SPI serial transmit buffer reg
SPIDAT           .set 7049h    ;SPI serial data reg
SPIPRI           .set 704Fh    ;SPI priority control reg

;SCI registers
SCICCR           .set 7050h    ;SCI communication control reg
SCICTL1          .set 7051h    ;SCI control reg 1
SCIHBAUD         .set 7052h    ;SCI baud-select reg, high bits
SCILBAUD         .set 7053h    ;SCI baud-select reg, low bits
SCICTL2          .set 7054h    ;SCI control reg 2
SCIRXST          .set 7055h    ;SCI receiver status reg
SCIRXEMU         .set 7056h    ;SCI emulation data buffer reg
SCIRXBUF         .set 7057h    ;SCI receiver data buffer reg
SCITXBUF         .set 7059h    ;SCI transmit data buffer reg
SCIPRI           .set 705Fh    ;SCI priority control reg

;External interrupt configuration registers
XINT1CR          .set 7070h    ;Ext interrupt 1 config reg

```

```

XINT2CR                .set 7071h    ;Ext interrupt 2 config reg

;Digital I/O registers
MCRA                    .set 7090h    ;I/O mux control reg A
MCRB                    .set 7092h    ;I/O mux control reg B
MCRC                    .set 7094h    ;I/O mux control reg C
PADATDIR                .set 7098h    ;I/O port A data & dir reg
PBDATDIR                .set 709Ah    ;I/O port B data & dir reg
PCDATDIR                .set 709Ch    ;I/O port C data & dir reg
PDDATDIR                .set 709Eh    ;I/O port D data & dir reg
PEDATDIR                .set 7095h    ;I/O port E data & dir reg
PFDATDIR                .set 7096h    ;I/O port F data & dir reg

;Analog-to-Digital Converter (ADC) registers
ADCTRL1                 .set 70A0h    ;ADC control reg 1
ADCTRL2                 .set 70A1h    ;ADC control reg 2
MAX_CONV                .set 70A2h    ;Maximum conversion channels reg
CHSELSEQ1               .set 70A3h    ;Channel select sequencing control reg 1
CHSELSEQ2               .set 70A4h    ;Channel select sequencing control reg 2
CHSELSEQ3               .set 70A5h    ;Channel select sequencing control reg 3
CHSELSEQ4               .set 70A6h    ;Channel select sequencing control reg 4
AUTO_SEQ_SR             .set 70A7h    ;Autosequence status reg
RESULT0                 .set 70A8h    ;Conversion result buffer reg 0
RESULT1                 .set 70A9h    ;Conversion result buffer reg 1
RESULT2                 .set 70AAh    ;Conversion result buffer reg 2
RESULT3                 .set 70ABh    ;Conversion result buffer reg 3
RESULT4                 .set 70ACH    ;Conversion result buffer reg 4
RESULT5                 .set 70ADh    ;Conversion result buffer reg 5
RESULT6                 .set 70AEh    ;Conversion result buffer reg 6
RESULT7                 .set 70AFh    ;Conversion result buffer reg 7
RESULT8                 .set 70B0h    ;Conversion result buffer reg 8
RESULT9                 .set 70B1h    ;Conversion result buffer reg 9
RESULT10                .set 70B2h    ;Conversion result buffer reg 10
RESULT11                .set 70B3h    ;Conversion result buffer reg 11
RESULT12                .set 70B4h    ;Conversion result buffer reg 12
RESULT13                .set 70B5h    ;Conversion result buffer reg 13
RESULT14                .set 70B6h    ;Conversion result buffer reg 14
RESULT15                .set 70B7h    ;Conversion result buffer reg 15
CALIBRATION             .set 70B8h    ;Calibration result reg

;Controller Area Network (CAN) registers
MDER                    .set 7100h    ;CAN mailbox direction/enable reg
TCR                     .set 7101h    ;CAN transmission control reg
RCR                     .set 7102h    ;CAN receive control reg
MCR                     .set 7103h    ;CAN master control reg
BCR2                    .set 7104h    ;CAN bit config reg 2
BCR1                    .set 7105h    ;CAN bit config reg 1
ESR                     .set 7106h    ;CAN error status reg
GSR                     .set 7107h    ;CAN global status reg
CEC                     .set 7108h    ;CAN trans and rcv err counters
CAN_IFR                 .set 7109h    ;CAN interrupt flag reg
CAN_IMR                 .set 710Ah    ;CAN interrupt mask reg
LAM0_H                  .set 710bh    ;CAN local acceptance mask MBX0/1
LAM0_L                  .set 710ch    ;CAN local acceptance mask MBX0/1
LAM1_H                  .set 710dh    ;CAN local acceptance mask MBX2/3
LAM1_L                  .set 710eh    ;CAN local acceptance mask MBX2/3

```

| | | |
|----------------------------------|------------|--|
| MSGID0L | .set 7200h | ;CAN msg ID for mailbox 0 (lo 16 bits) |
| MSGID0H | .set 7201h | ;CAN msg ID for mailbox 0 (hi 16 bits) |
| MSGCTRL0 | .set 7202h | ;CAN RTR and DLC for mailbox 0 |
| MBX0A | .set 7204h | ;CAN 2 of 8 bytes of mailbox 0 |
| MBX0B | .set 7205h | ;CAN 2 of 8 bytes of mailbox 0 |
| MBX0C | .set 7206h | ;CAN 2 of 8 bytes of mailbox 0 |
| MBX0D | .set 7207h | ;CAN 2 of 8 bytes of mailbox 0 |
| | | |
| MSGID1L | .set 7208h | ;CAN msg ID for mailbox 1 (lo 16 bits) |
| MSGID1H | .set 7209h | ;CAN msg ID for mailbox 1 (hi 16 bits) |
| MSGCTRL1 | .set 720Ah | ;CAN RTR and DLC for mailbox 1 |
| MBX1A | .set 720Ch | ;CAN 2 of 8 bytes of mailbox 1 |
| MBX1B | .set 720Dh | ;CAN 2 of 8 bytes of mailbox 1 |
| MBX1C | .set 720Eh | ;CAN 2 of 8 bytes of mailbox 1 |
| MBX1D | .set 720Fh | ;CAN 2 of 8 bytes of mailbox 1 |
| | | |
| MSGID2L | .set 7210h | ;CAN msg ID for mailbox 2 (lo 16 bits) |
| MSGID2H | .set 7211h | ;CAN msg ID for mailbox 2 (hi 16 bits) |
| MSGCTRL2 | .set 7212h | ;CAN RTR and DLC for mailbox 2 |
| MBX2A | .set 7214h | ;CAN 2 of 8 bytes of mailbox 2 |
| MBX2B | .set 7215h | ;CAN 2 of 8 bytes of mailbox 2 |
| MBX2C | .set 7216h | ;CAN 2 of 8 bytes of mailbox 2 |
| MBX2D | .set 7217h | ;CAN 2 of 8 bytes of mailbox 2 |
| | | |
| MSGID3L | .set 7218h | ;CAN msg ID for mailbox 3 (lo 16 bits) |
| MSGID3H | .set 7219h | ;CAN msg ID for mailbox 3 (hi 16 bits) |
| MSGCTRL3 | .set 721Ah | ;CAN RTR and DLC for mailbox 3 |
| MBX3A | .set 721Ch | ;CAN 2 of 8 bytes of mailbox 3 |
| MBX3B | .set 721Dh | ;CAN 2 of 8 bytes of mailbox 3 |
| MBX3C | .set 721Eh | ;CAN 2 of 8 bytes of mailbox 3 |
| MBX3D | .set 721Fh | ;CAN 2 of 8 bytes of mailbox 3 |
| | | |
| MSGID4L | .set 7220h | ;CAN msg ID for mailbox 4 (lo 16 bits) |
| MSGID4H | .set 7221h | ;CAN msg ID for mailbox 4 (hi 16 bits) |
| MSGCTRL4 | .set 7222h | ;CAN RTR and DLC for mailbox 4 |
| MBX4A | .set 7224h | ;CAN 2 of 8 bytes of mailbox 4 |
| MBX4B | .set 7225h | ;CAN 2 of 8 bytes of mailbox 4 |
| MBX4C | .set 7226h | ;CAN 2 of 8 bytes of mailbox 4 |
| MBX4D | .set 7227h | ;CAN 2 of 8 bytes of mailbox 4 |
| | | |
| MSGID5L | .set 7228h | ;CAN msg ID for mailbox 5 (lo 16 bits) |
| MSGID5H | .set 7229h | ;CAN msg ID for mailbox 5 (hi 16 bits) |
| MSGCTRL5 | .set 722Ah | ;CAN RTR and DLC for mailbox 5 |
| MBX5A | .set 722Ch | ;CAN 2 of 8 bytes of mailbox 5 |
| MBX5B | .set 722Dh | ;CAN 2 of 8 bytes of mailbox 5 |
| MBX5C | .set 722Eh | ;CAN 2 of 8 bytes of mailbox 5 |
| MBX5D | .set 722Fh | ;CAN 2 of 8 bytes of mailbox 5 |
| | | |
| ;Event Manager A (EVA) registers | | |
| GPTCONA | .set 7400h | ;GP timer control reg A |
| T1CNT | .set 7401h | ;GP timer 1 counter reg |
| T1CMPR | .set 7402h | ;GP timer 1 compare reg |
| T1PR | .set 7403h | ;GP timer 1 period reg |
| T1CON | .set 7404h | ;GP timer 1 control reg |
| T2CNT | .set 7405h | ;GP timer 2 counter reg |
| T2CMPR | .set 7406h | ;GP timer 2 compare reg |
| T2PR | .set 7407h | ;GP timer 2 period reg |

```

T2CON          .set 7408h    ;GP timer 2 control reg
COMCONA        .set 7411h    ;Compare control reg A
ACTRA          .set 7413h    ;Compare action control reg A
DBTCONA        .set 7415h    ;Dead-band timer control reg A
CMPR1          .set 7417h    ;compare reg 1
CMPR2          .set 7418h    ;compare reg 2
CMPR3          .set 7419h    ;compare reg 3
CAPCONA        .set 7420h    ;Capture control reg A
CAPFIFOA       .set 7422h    ;Capture FIFO status reg A
CAP1FIFO       .set 7423h    ;Capture Channel 1 FIFO top
CAP2FIFO       .set 7424h    ;Capture Channel 2 FIFO top
CAP3FIFO       .set 7425h    ;Capture Channel 3 FIFO top
CAP1FBOT       .set 7427h    ;Bottom reg of capture FIFO stack 1
CAP2FBOT       .set 7427h    ;Bottom reg of capture FIFO stack 2
CAP3FBOT       .set 7427h    ;Bottom reg of capture FIFO stack 3
EVAIMRA        .set 742Ch    ;EVA interrupt mask reg A
EVAIMRB        .set 742Dh    ;EVA interrupt mask reg B
EVAIMRC        .set 742Eh    ;EVA interrupt mask reg C
EVAIFRA        .set 742Fh    ;EVA interrupt flag reg A
EVAIFRB        .set 7430h    ;EVA interrupt flag reg B
EVAIFRC        .set 7431h    ;EVA interrupt flag reg C

;Event Manager B (EVB) registers
GPTCONB        .set 7500h    ;GP timer control reg B
T3CNT          .set 7501h    ;GP timer 3 counter reg
T3CMPR         .set 7502h    ;GP timer 3 compare reg
T3PR           .set 7503h    ;GP timer 3 period reg
T3CON          .set 7504h    ;GP timer 3 control reg
T4CNT          .set 7505h    ;GP timer 4 counter reg
T4CMPR         .set 7506h    ;GP timer 4 compare reg
T4PR           .set 7507h    ;GP timer 4 period reg
T4CON          .set 7508h    ;GP timer 4 control reg
COMCONB        .set 7511h    ;Compare control register B
ACTRB          .set 7513h    ;Compare action control register B
DBTCONB        .set 7515h    ;Dead-band timer control reg B
CMPR4          .set 7517h    ;Compare reg 4
CMPR5          .set 7518h    ;Compare reg 5
CMPR6          .set 7519h    ;Compare reg 6
CAPCONB        .set 7520h    ;Capture control reg B
CAPFIFOB       .set 7522h    ;Capture FIFO status reg B
CAP4FIFO       .set 7523h    ;Capture channel 4 FIFO top
CAP5FIFO       .set 7524h    ;Capture channel 5 FIFO top
CAP6FIFO       .set 7525h    ;Capture channel 6 FIFO top
CAP4FBOT       .set 7527h    ;Bottom reg of capture FIFO stack 4
CAP5FBOT       .set 7527h    ;Bottom reg of capture FIFO stack 5
CAP6FBOT       .set 7527h    ;Bottom reg of capture FIFO stack 6
EVBIMRA        .set 752Ch    ;EVB interrupt mask reg A
EVBIMRB        .set 752Dh    ;EVB interrupt mask reg B
EVBIMRC        .set 752Eh    ;EVB interrupt mask reg C
EVBIFRA        .set 752Fh    ;EVB interrupt flag reg A
EVBIFRB        .set 7530h    ;EVB interrupt flag reg B
EVBIFRC        .set 7531h    ;EVB interrupt flag reg C

;I/O space mapped registers
FCMR           .set 0FF0Fh    ;Flash control mode reg
WSGR           .set 0FFFFh    ;Wait-state generator reg

```

```
;*****
;Other useful definitions below (not addresses) *
;*****

;Data page definitions for LDP instruction
DP_PF1      .set 224      ;sys regs, WD, SPI, SCI, (0x7000-0x707F)
DP_PF2      .set 225      ;ADC, GPIO (0x7080-0x70FF)
DP_CAN1      .set 226      ;CAN control regs (0x7100-0x717F)
DP_CAN2      .set 228      ;CAN mailboxes 1-5 (0x7200-0x727F)
DP_EVA       .set 232      ;Event manager A (0x7400-0x747F)
DP_EVB       .set 234      ;Event manager A (0x7500-0x757F)

;Bit codes for test bit instruction (BIT)
BIT15        .set 0000h    ;Bit code for bit 0
BIT14        .set 0001h    ;Bit code for bit 1
BIT13        .set 0002h    ;Bit code for bit 2
BIT12        .set 0003h    ;Bit code for bit 3
BIT11        .set 0004h    ;Bit code for bit 4
BIT10        .set 0005h    ;Bit code for bit 5
BIT9         .set 0006h    ;Bit code for bit 6
BIT8         .set 0007h    ;Bit code for bit 7
BIT7         .set 0008h    ;Bit code for bit 8
BIT6         .set 0009h    ;Bit code for bit 9
BIT5         .set 000Ah    ;Bit code for bit 10
BIT4         .set 000Bh    ;Bit code for bit 11
BIT3         .set 000Ch    ;Bit code for bit 12
BIT2         .set 000Dh    ;Bit code for bit 13
BIT1         .set 000Eh    ;Bit code for bit 14
BIT0         .set 000Fh    ;Bit code for bit 15
```