Important Notice

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

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

Customers are responsible for their applications using TI components.

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

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


Revision History

October 2001 – Revision 1.0
January 2002 – Revision 2.0
May 2002 – Revision 3.0
June 2002 – Revision 3.1
October 2002 – Revision 4.0
December 2002 – Revision 4.1
July 2003 – Revision 4.2
August 2003 – Revision 4.21
February 2004 – Revision 5.0

Mailing Address

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

- Name
- Company
- Project Responsibilities
- DSP / Microcontroller Experience
- TMS320 DSP Experience
- Hardware / Software - Assembly / C
- Interests
C28x Workshop Outline

1. Architecture Overview
2. Programming Development Environment
3. Peripheral Register Header Files
4. Reset and Interrupts
5. System Initialization
6. Analog-to-Digital Converter
7. Event Manager
8. Numerical Concepts and IQ Math
9. Using DSP/BIOS
10. System Design
11. Communications
12. Support Resources

eZdsp™ F2812 Hardware

JTAG Interface (P1) EXPANSION SRAM Data & Address (P2) 64K x 16
Parallel Port/ JTAG Controller Interface (P3) TMS320F2812 - DSP
Power Connector (P6) +5V I/O Interface (P4/P8/P7) ANALOG Interface (P5/P9)
Introduction

This architecture overview introduces the basic architecture of the TMS320C28x (C28x) series of Digital Signal Processors from Texas Instruments. The C28x series adds a new level of general purpose processing ability unseen in any previous DSP chips. The C28x is ideal for applications combining DSP, standard microcontroller processing, efficient C code execution, or operating system tasks.

Learning Objectives

When this module is complete, you should have a basic understanding of the C28x architecture and how all of its components work together to create a high-end, uniprocessor control system.

<table>
<thead>
<tr>
<th>Learning Objectives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identify the three main components of the C28x</td>
</tr>
<tr>
<td>List the key features of the C28x CPU</td>
</tr>
<tr>
<td>Identify the memory capabilities of the C28x</td>
</tr>
<tr>
<td>Identify the peripherals available on the C28x</td>
</tr>
</tbody>
</table>
Module Topics

Architecture Overview

What is the TMS320C28x?
C28x CPU
  Multiplier, ALU, and Shifters
  TMS320C28x Internal Bussing
  Special Instructions
  Pipeline Advantage

Memory
  Memory Map
  Code Security Module (CSM)
  Peripherals

Fast Interrupt Response

C28 Mode

Reset

Summary
What is the TMS320C28x?

The TMS320C28x is a 32-bit fixed point DSP that specializes in high performance control applications such as, robotics, industrial automation, mass storage devices, lighting, optical networking, power supplies, and other control applications needing a single processor to solve a high performance application.

The C28x architecture can be divided into 3 functional blocks:

- CPU and busing
- Memory
- Peripherals
The C28x is a highly integrated, high performance solution for demanding control applications. The C28x is a cross between a general microcontroller and a digital signal processor, balancing the code density of a RISC chip and the execution speed of a DSP with the architecture, firmware, and development tools of a microcontroller.

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

The C28x design supports an efficient C engine with hardware that allows the C compiler to generate compact code. Multiple busses and an internal register bus allow an efficient and flexible way to operate on the data. The architecture is also supported by powerful addressing modes, which allow the compiler as well as the assembly programmer to generate compact code that is almost one to one corresponded to the C code.

The C28x is as efficient in DSP math tasks as it is in system control tasks that typically are handled by microcontroller devices. This efficiency removes the need for a second processor in many systems.

The C28x is one of several members of the fixed-point generations of digital signal processors (DSPs) in the TMS320 family. The C28x is source-code and object-code compatible with the C27x. In addition, the C28x is source code compatible with the 24x/240x DSP and previously written code can be reassembled to run on a C28x device. This allows for migration of existing code onto the C28x.
Multiplier, ALU, and Shifters

The 32 x 32-bit MAC capabilities of the C28x and its 64-bit processing capabilities, enable the C28x to efficiently handle higher numerical resolution problems that would otherwise demand a more expensive floating-point processor solution. Along with this is the capability to perform two 16 x 16-bit multiply accumulate instructions simultaneously or Dual MACs (DMAC).
TMS320C28x Internal Bussing

As with many DSP type devices, multiple busses are used to move data between the memories and peripherals and the CPU. The C28x memory bus architecture contains:

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

The 32-bit-wide data busses enable single cycle 32-bit operations. This multiple bus architecture, known as a Harvard Bus Architecture enables the C28x to fetch an instruction, read a data value and write a data value in a single cycle. All peripherals and memories are attached to the memory bus and will prioritize memory accesses.
Special Instructions

C28x Atomic Read/Modify/Write

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

Standard Load/Store
- DINT
- MOV AL, *XAR2
- AND AL, #1234h
- MOV *XAR2, AL
- EINT

Atomic Read/Modify/Write
- AND *XAR2, #1234h

2 words / 1 cycles

6 words / 6 cycles

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

The C28x uses a special 8-stage protected pipeline to maximize the throughput. This protected pipeline prevents a write to and a read from the same location from occurring out of order.

This pipelining also enables the C28x to execute at high speeds without resorting to expensive high-speed memories. Special branch-look-ahead hardware minimizes the latency for conditional discontinuities. Special store conditional operations further improve performance.
Memory

The memory space on the C28x is divided into program and data space. There are several different types of memory available that can be used as both program or data space. They include the flash memory, single access RAM (SARAM), expanded SARAM, and Boot ROM which is factory programmed with boot software routines or standard tables used in math related algorithms.

Memory Map

The C28x CPU contains no memory, but can access memory both on and off the chip. The C28x uses 32-bit data addresses and 22-bit program addresses. This allows for a total address reach of 4G words (1 word = 16 bits) in data space and 4M words in program space. Memory blocks on all C28x designs are uniformly mapped to both program and data space.

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

![TMS320F2812 Memory Map](image-url)
Code Security Module (CSM)

- Prevents reverse engineering and protects valuable intellectual property
- 128-bit user defined password is stored in Flash
- 128-bits = $2^{128} = 3.4 \times 10^{38}$ possible passwords
- To try 1 password every 2 cycles at 150 MHz, it would take at least $1.4 \times 10^{23}$ years to try all possible combinations!

Peripherals

The C28x comes with many built in peripherals optimized to support control applications. These peripherals vary depending on which C28x device you choose.

- Event Manager
- Analog-to-Digital Converter
- Watchdog Timer
- SPI
- SCI
- CAN
- McBSP
- GPIO
Fast Interrupt Response

The fast interrupt response, with automatic context save of critical registers, resulting in a device that is capable of servicing many asynchronous events with minimal latency. C28x implements a zero cycle penalty to do 14 registers context saved and restored during an interrupt. This feature helps reduce the interrupt service routine overheads.
C28 Mode

The C28x is one of several members of the fixed-point generations of digital signal processors (DSPs) in the TMS320 family. The C28x is source-code and object-code compatible with the C27x. In addition, the C28x is source code compatible with the 24x/240x DSP and previously written code can be reassembled to run on a C28x device. This allows for migration of existing code onto the C28x.

### C28x / C24x Modes

<table>
<thead>
<tr>
<th>Mode Type</th>
<th>Mode Bits</th>
<th>Compiler Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>C24x Mode</td>
<td>1</td>
<td>-v28 -m20</td>
</tr>
<tr>
<td>C28x Mode</td>
<td>1 0</td>
<td>-v28 (workshop)</td>
</tr>
<tr>
<td>Test Mode (default)</td>
<td>0 0</td>
<td>-v27</td>
</tr>
<tr>
<td>Reserved</td>
<td>0 1</td>
<td></td>
</tr>
</tbody>
</table>

- **C24x source-compatible mode:**
  - Allows you to run C24x source code which has been reassembled using the C28x code generation tools (need new vectors)
- **C28x mode:**
  - Can take advantage of all the C28x native features
Reset

Reset – Bootloader

**Reset**

- OBJMODE=0
- AMODE=0
- ENPIE=0
- VMAP=1

---

**XMPNMC=0**  
(microcomputer mode)

**Reset vector fetched from boot ROM**  
0x3F FFC0

**Bootloader sets**  
OBJMODE = 1  
AMODE = 0

**Boot determined by state of GPIO pins**

**Execution Entry Point**  
H0 SARAM

---

**Note:**  
Details of the various boot options will be discussed in the Reset and Interrupts module.
Summary

- High performance 32-bit DSP
- 32 x 32 bit or dual 16 x 16 bit MAC
- Atomic read-modify-write instructions
- 8-stage fully protected pipeline
- Fast interrupt response manager
- 128Kw on-chip flash memory
- Code security module (CSM)
- Two event managers
- 12-bit ADC module
- 56 shared GPIO pins
- Watchdog timer
- Communications peripherals
Programming Development Environment

Introduction

This module will explain how to use Code Composer Studio (CCS) integrated development environment (IDE) tools to develop a program. Creating projects and setting building options will be covered. Use and the purpose of the linker command file will be described. Additionally, the DSP/BIOS Configuration Tool will be used to handle system memory and system setup.

Learning Objectives

Learning Objectives

- Use Code Composer Studio to:
  - Create a Project
  - Set Build Options
- Create a user linker command file which:
  - Describes a system’s available memory
  - Indicates where sections will be placed in memory
- Use DSP/BIOS Configuration Tool to:
  - Handle system memory and system setup
  - Create a .cdb and .cmd file
Module Topics

Programming Development Environment ................................................................. 2-1

Module Topics ........................................................................................................... 2-2

Code Composer Studio .............................................................................................. 2-3
  Software Development and COFF Concepts .............................................................. 2-3
  Projects ...................................................................................................................... 2-5
  Build Options .......................................................................................................... 2-6

Creating a Linker Command File .............................................................................. 2-9
  Sections ..................................................................................................................... 2-9
  Linker Command Files (.cmd) ................................................................................... 2-12
  Memory-Map Description ......................................................................................... 2-12
  Section Placement .................................................................................................. 2-14

Exercise 2a ................................................................................................................ 2-15
  Summary: Linker Command File ............................................................................ 2-16

Lab 2a: Linker Command File .................................................................................... 2-17

DSP/BIOS Configuration Tool .................................................................................... 2-21

Lab 2b: DSP/BIOS Configuration Tool ....................................................................... 2-26

Solutions .................................................................................................................... 2-30
**Code Composer Studio**

**Software Development and COFF Concepts**

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

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

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

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

The high level of modularity and portability resulting from this system simplifies the processes of verification, debug and maintenance. The process of COFF development is presented in greater detail in the following paragraphs.
The concept of COFF tools is to allow modular development of software independent of hardware concerns. An individual assembly language file is written to perform a single task and may be linked with several other tasks to achieve a more complex total system.

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

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

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

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

![CCS Project](image)

**The CCS Project**

**Project (.pj) files contain:**

- Source files (by reference)
  - Source (C, assembly)
  - Libraries
  - DSP/BIOS configuration
  - Linker command files
- Project settings:
  - Build Options (compiler and assembler)
  - Build configurations
  - DSP/BIOS
  - Linker

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

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

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

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

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

There is a one-to-one relationship between the items in the text box and the GUI check and drop-down box selections. Once you have mastered the various options, you can probably find yourself just typing in the options.
There are many linker options but these four handle all of the basic needs.

- `\-o <filename>` specifies the output (executable) filename.
- `\-m <filename>` creates a map file. This file reports the linker’s results.
- `\-c` tells the compiler to autoinitialize your global and static variables.
- `\-x` tells the compiler to exhaustively read the libraries. Without this option libraries are searched only once, and therefore backwards references may not be resolved.
To help make sense of the many compiler options, TI provides two default sets of options (configurations) in each new project you create. The Release (optimized) configuration invokes the optimizer with –o3 and disables source-level, symbolic debugging by omitting –g (which disables some optimizations to enable debug).
Creating a Linker Command File

Sections

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

Every C program consists of different parts called **sections**.

All default sections names begin with “.”.

The compiler has default sections names for **initialized** and **uninitialized** sections.

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

- Global Variables
- Initial Values for global variables
- Local Variables (i.e. the stack)
- Code (the actual instructions)
Following is a list of the sections that are created by the compiler. Along with their description, we provide the Section Name defined by the compiler.

### Compiler Section Names

#### Initialized Sections

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Link Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text</td>
<td>code</td>
<td>program</td>
</tr>
<tr>
<td>.cinit</td>
<td>initialized global and static variables</td>
<td>program</td>
</tr>
<tr>
<td>.econst</td>
<td>constant data (e.g. const int k = 3;)</td>
<td>data</td>
</tr>
<tr>
<td>.switch</td>
<td>tables for switch statements</td>
<td>prog/low 64K data</td>
</tr>
<tr>
<td>.pinit</td>
<td>tables for global constructors (C++)</td>
<td>program</td>
</tr>
</tbody>
</table>

#### Uninitialized Sections

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Link Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>.ebss</td>
<td>global &amp; static variables</td>
<td>data</td>
</tr>
<tr>
<td>.stack</td>
<td>stack space</td>
<td>low 64K data</td>
</tr>
<tr>
<td>.esysmem</td>
<td>memory for far malloc functions</td>
<td>data</td>
</tr>
</tbody>
</table>

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

#### Program Code (.text)

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

#### Constants (.cinit – initialized data)

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

#### Variables (.ebss – uninitialized data)

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

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

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

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.

The format is: \textit{Name: origin = 0x?????, length = 0x????}

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

```
MEMORY
{  EPROM: origin = 0x0000 , length = 0x0800 }
```

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 = 0x0000 , length = 0x0800
   RAM: origin = 0x1000 , length = 0x02000
   ABT646: origin = 0x8000 , length = 0x00001
}
Remember that the DSP has two memory maps: Program, and Data. Therefore, the MEMORY description must describe each of these separately. The loader uses the following syntax to delineate each of these:

<table>
<thead>
<tr>
<th>Linker Page</th>
<th>TI Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page 0</td>
<td>Program</td>
</tr>
<tr>
<td>Page 1</td>
<td>Data</td>
</tr>
</tbody>
</table>

### Linker Command File

```c
MEMORY
{
    PAGE 0:    /* Program Space */
    FLASH:     org = 0x3D8000,  len = 0x20000

    PAGE 1:    /* Data Space */
    M0SARAM:   org = 0x000000,  len = 0x400
    M1SARAM:   org = 0x000400,  len = 0x400
}
```
**Section Placement**

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

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

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

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

---

**Linker Command File**

```
MEMORY
{
    PAGE 0:    /* Program Space */
        FLASH:    org = 0x3D8000,  len = 0x20000

    PAGE 1:    /* Data Space */
        M0SARAM:  org = 0x000000,  len = 0x400
        M1SARAM:  org = 0x000400,  len = 0x400
}
```

```
SECTIONS
{
    .text:>       FLASH       PAGE 0
    .ebss:>      M0SARAM     PAGE 1
    .cinit:>     FLASH       PAGE 0
    .stack:>    M1SARAM     PAGE 1
}
```
Exercise 2a

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

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

Fill in the blanks:

Exercise 2a - Command File

```
Memory
{
    PAGE__: /* Program Space */
    ______: org = ________, len = ______
    ______: /* Data Space */
    ______: org = ________, len = ______
    ______: org = ________, len = ______
    ______: org = ________, len = ______
    ______: org = ________, len = ______
    ______: org = ________, len = ______
}

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

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

<table>
<thead>
<tr>
<th>Linker Command File Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>◆ Memory Map Description</td>
</tr>
<tr>
<td>• Name</td>
</tr>
<tr>
<td>• Location</td>
</tr>
<tr>
<td>• Size</td>
</tr>
<tr>
<td>◆ Sections Description</td>
</tr>
<tr>
<td>• Directs software sections into named memory regions</td>
</tr>
<tr>
<td>• Allows per-file discrimination</td>
</tr>
<tr>
<td>• Allows separate load/run locations</td>
</tr>
</tbody>
</table>
Lab 2a: Linker Command File

Objective

Create a linker command file and link the C program file (LAB2.C) into the system described below.

System Description:
- TMS320F2812
- All internal RAM blocks allocated

Placement of Sections:
- .text into RAM Block H0SARAM on PAGE 0 (code space)
- .cinit into RAM Block H0SARAM on PAGE 0 (code space)
- .ebss into RAM Block M1SARAM on PAGE 1 (data space)
- .stack into RAM Block M0SARAM on PAGE 1 (data space)

Procedure

Create a New Project

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

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

   Project → New

   type Lab2 in the project name field and make sure the save in location is: 
   C:\C28x\LABS\LAB2. This will create a .pjt file which will invoke all the necessary tools (compiler, assembler, linker) to build your project. It will also create a debug folder that will hold immediate output files.

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

   Project → Add Files to Project…

   and make sure you’re looking in C:\C28x\LABS\LAB2. Change the “files of type” to view C source files (*.c) and select Lab2.c and click OPEN. This will add the file Lab2.c to your newly created project.

4. Add Lab2a.cmd to the project using the same procedure. This file will be edited during the lab exercise.

5. Next, add the compiler run-time support library to the project (C:\ti\c2000\cgtools\lib\rts2800_ml.lib).

6. In the project window on the left click the plus sign (+) to the left of Project. Now, click on the plus sign next to Lab2.pjt. Notice that the Lab2a.cmd file is listed.  
   Click on Source to see the current source file list (i.e. Lab2.c).

**Project Build Options**

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

   Click: Project → Build Options...

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

9. Set the Stack Size to 0x200. Select **OK** and then close the Build Options window.

**Edit the Linker Command File - Lab2a.cmd**

10. To open and edit Lab2a.cmd, double click on the filename in the project window.

11. Edit the Memory{} declaration by describing the system memory shown on the “Lab2a: Linker Command File” slide.
12. In the \texttt{Sections{}} area, notice that a section called \texttt{.reset} has already been allocated. The \texttt{.reset} section is part of the \texttt{rts2800_ml.lib}, and is not needed. By putting the \texttt{TYPE = DSECT} modifier after its allocation, the linker will ignore this section and not allocate it.

13. Place the sections defined on the slide into the appropriate memories via the \texttt{Sections{}} area. Save your work.

\section*{Build and Load the Project}

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

<table>
<thead>
<tr>
<th>Button</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Compile File</td>
<td>Compile, assemble the current open file</td>
</tr>
<tr>
<td>2</td>
<td>Incremental Build</td>
<td>Compile, assemble only changed files, then link</td>
</tr>
<tr>
<td>3</td>
<td>Rebuild All</td>
<td>Compile, assemble all files, then link</td>
</tr>
<tr>
<td>4</td>
<td>Stop Build</td>
<td>Stop code generation</td>
</tr>
</tbody>
</table>

15. Code Composer Studio can automatically load the output file after a successful build. On the menu bar click: Option $\rightarrow$ Customize... and select the “Program Load Options” tab, check “Load Program After Build”, then click OK.

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

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

18. Rebuild the project (there should be no errors this time). The output file should automatically load. The Program Counter should be pointing to \texttt{.Cnt00} in the Disassembly Window.

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

\section*{Debug Environment Windows}

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

20. Open a \textit{memory window} to view the global variable “\texttt{z}”.

Click: View $\rightarrow$ Memory on the menu bar.
Type “&z” into the address field. Note that you must use the ampersand (meaning "address of") when using a symbol in a memory window address box. Also note that Code Composer Studio is case sensitive.

Set the properties format to “Hex – TI style.” This will give you more viewable data in the window.

Click OK to close the window property selection screen. The memory window will now open. You can change the contents of any address in the memory window by double-clicking on its value. This is useful during debug.

21. Open the watch window to view the local variables x and y.

Click: View → Watch Window on the menu bar.

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

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

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

Click the “Watch 1” tab at the bottom of the watch window. In the empty box in the "Name" column, type “z”. Note that you do not use an ampersand here. The watch window knows you are specifying a symbol.

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

**Single-stepping the Code**

23. Single-step through main() by using the <F8> key (or you can use the Single Step button on the vertical toolbar). Check to see if the program is working as expected. What is the value for “z” when you get to the end of the program?

End of Exercise
The DSP/BIOS Configuration Tool (often called Config Tool or GUI Tool or GUI) creates and modifies a system file called the Configuration DataBase (.CDB). If we talk about using CDB files, we’re also talking about using the Config Tool.

The GUI (graphical user interface) simplifies system design by:

- Automatically including the appropriate runtime support libraries
- Automatically handles interrupt vectors and system reset
- Handles system memory configuration (builds CMD file)
- When a CDB file is saved, the Config Tool generates 5 additional files:

<table>
<thead>
<tr>
<th>Filename</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filename.cdb</td>
<td>Configuration Database</td>
</tr>
<tr>
<td>Filename_CFG_c.c</td>
<td>C code created by Config Tool</td>
</tr>
<tr>
<td>Filename_CFG.s28</td>
<td>ASM code created by Config Tool</td>
</tr>
<tr>
<td>Filename_CFG.cmd</td>
<td>Linker commands</td>
</tr>
<tr>
<td>Filename_CFG.h</td>
<td>header file for *cfg_c.c</td>
</tr>
<tr>
<td>Filename_CFG.h28</td>
<td>header file for *cfg.s28</td>
</tr>
</tbody>
</table>

When you add a CDB file to your project, CCS automatically adds the C and assembly (.S28) files to the project under the Generated Files folder. (You must manually add the CMD file, yourself.)
1. Creating a New Memory Region (Using MEM)

First, to create a specific memory area, open up the .CDB file, right-click on the Memory Section Manager and select “Insert MEM”. Give this area a unique name and then specify its base and length. Once created, you can place sections into it (shown in the next step).
2. Placing Sections – MEM Manager Properties

The configuration tool makes it easy to place sections. The predefined compiler sections that were described earlier each have their own drop-down menu to select one of the memory regions you defined (in step 1).

![Memory Section Manager Properties](image)

- To place a section into a memory area:
  - Right-click on MEM and select Properties
  - Select the appropriate tab (e.g. Compiler)
  - Select the memory for each section
3. Running the Linker

Creating the Linker Command File (via .CDB)

When you have finished creating memory regions and allocating sections into these memory areas (i.e. when you save the .CDB file), the CCS configuration tool creates five files. One of the files is BIOS’s cfg.cmd file — a linker command file.

```
MEMORY{
    FLASH:   org = 0x3D8000, len = 0x20000
    H0SARAM: org = 0x3F8000, len = 0x2000
    ... }

SECTIONS{
    .text: > FLASH
    .bss:  > M0SARAM
    ... }
```

This file contains two main parts, MEMORY and SECTIONS. (Though, if you open and examine it, it’s not quite as nicely laid out as shown above.)
Running the Linker

The linker’s main purpose is to link together various object files. It combines like-named input sections from the various object files and places each new output section at specific locations in memory. In the process, it resolves (provides actual addresses for) all of the symbols described in your code.

![Config Tool Linker Command File](image)

- Do not modify appcfg.cmd – your changes will be overwritten during “Build” (or “Rebuild”)

The linker can create two outputs, the executable (.out) file and a report which describes the results of linking (.map).

**Note:** If the graphic above wasn’t clear enough, the linker gets run automatically when you BUILD or REBUILD your project.
Lab 2b: DSP/BIOS Configuration Tool

Objective

Use Code Composer Studio and DSP/BIOS configuration tool to create a configuration database files (*.CDB). The generated linker command file Labcfg.cmd will be then be used with Lab2.c to verify its operation. The memory and sections of a “user” linker command file will be deleted, however, the “user” linker command file will be needed and modified in future labs.

System Description:
- TMS320F2812
- All internal RAM blocks allocated

Placement of Sections:
- .text into RAM Block H0SARAM in code space (PAGE 0)
- .cinit into RAM Block H0SARAM in code space (PAGE 0)
- .ebss into RAM Block M1SARAM in data space (PAGE 1)
- .stack into RAM Block M0SARAM in data space (PAGE 1)

Procedure

Project Lab2.pjt

1. If Code Composer Studio is not running from the previous lab exercise, double click on the CCS 2 (‘C2000) icon on the desktop. Maximize CCS to fill your screen. Then select
Lab 2b: DSP/BIOS Configuration Tool

Lab2.pjt from the Project → Recent Project Files list. This will put you at the proper starting point for this lab exercise.

2. If needed, verify that the project is open in CCS by left clicking the plus sign (+) to the left of Project. Then, click on the plus sign next to Lab2.pjt, as well as the other plus signs to verify all the previous files have been included.

Remove “rts2800_ml.lib” and “Lab2a.cmd” from the Project

3. Highlight the rts2800_ml.lib in the project window and right click, then select “Remove from Project”. The DSP/BIOS Config Tool supplies its own rts library.

4. Remove Lab2a.cmd using the same procedure. We will be using the DSP/BIOS configuration tool to create a linker command file.

Using the DSP/BIOS Configuration Tool

5. The configuration database files (*.cdb), created by the Config Tool, controls a wide range of CCS capabilities. In this lab exercise, the CDB file will be used to automatically create and perform memory management. Create a new CDB file for this lab. On the menu bar click:

File → New → DSP/BIOS Configuration…

A dialog box appears. The CDB files shown in the aforementioned dialog box are called “seed” CDB files. CDB files are used to configure many objects specific to the processor. Select the c28xx.cdb template and click OK.

6. Save the configuration file by selecting:

File → Save As…

and name it Lab.cdb in C:\C28x\LABS\LAB2. Close the configuration window.

7. Add the configuration file to the project. Click:

Project → Add Files to Project…

Make sure you’re looking in C:\C28x\LABS\LAB2. Change the “files of type” to view All Files (*.*) and select Lab.cdb. Click OPEN to add the file to the project.

8. Add the generated linker command file, Labcfg.cmd, to the project using the same procedure.

Create New Memory Sections Using the CDB File

9. Open the Lab.cdb file by left clicking on the plus sign (+) to the left of DSP/BIOS Config and double clicking on the Lab.cdb file. In the configuration window, left click the plus sign next to System and the plus sign next to MEM.

10. By default, the Memory Section Manager has combined the memory space for L0SARAM and L1SARAM into a single memory block called L0SARAM. It has also
combined M0SARAM and M1SARAM into a single memory block called M0SARAM. We want to split these memory sections as shown in the slide for this lab exercise.

Right click on MEM – Memory Section Manager and select Insert MEM. Rename the newly added memory section to L1SARAM. Create a second new memory section, and rename it to M1SARAM.

11. Modify the length and base addresses of each of the memory sections L0SARAM, L1SARAM, M0SARAM, and M1SARAM to correspond to the memory mapping shown in the figure at the beginning of this lab. To modify the length and base address of a memory section, right click on the memory in configuration tool, and select “Properties.” While modifying the length and base address of each section, make sure the “create a heap in memory” box is checked ONLY for L0SARAM. Uncheck this box in the other three memory sections.

12. Right click on MEM – Memory Section Manager and select Properties. Select the Compiler Sections tab and place the sections defined on the slide into the appropriate memories via the pull-down boxes. Click OK to save your work.

Set the Stack Size in the CDB File

13. Recall in the previous lab exercise that the stack size was set using the CCS project Build Options. When using the DSP/BIOS configuration tool, the stack size is instead specified in the CDB file. First we need to remove the stack size setting from the project Build Options.

14. Click: Project → Build Options... and select the Linker tab. Delete the entry for setting the Stack Size to 0x200. Select OK to close the Build Options window.

15. Right click on MEM – Memory Section Manager and select Properties. Select the General tab. Notice that the Stack Size has been set to 0x200 by default, so there is no need to modify this. Click OK to close the window.

Build and Load the Project

16. Click the “Build” button and watch the tools run in the build window. The output file should automatically load. The Program Counter should be pointing to _c_int00 in the Disassembly Window.

17. Under Debug on the menu bar click “Go Main”. This will run through the DSP/BIOS C-environment initialization routine and stop at main() in Lab2.c.

Run the Code

18. We will verify the operation of the code using the same procedure used in Lab2a. Open the watch window and add the global variable z.
19. Next, single-step the routine through to the end. Check to see if the program is working as expected. You should get the same value for "z" as in Lab2a.

End of Exercise
Solutions

**Exercise 2a - Solution**

Memory

```c
{ PAGE 0: /* Program Space */
    FLASH: org = 0x3D8000, len = 0x20000
    PAGE 1: /* Data Space */
    MOSARAM: org = 0x000000, len = 0x400
    M1SARAM: org = 0x000400, len = 0x400
    L0SARAM: org = 0x008000, len = 0x1000
    L1SARAM: org = 0x009000, len = 0x1000
    H0SARAM: org = 0x3F8000, len = 0x20000
}

SECTIONS

```

**Lab 2a: Solution - lab2.cmd**

MEMORY

```c
{ PAGE 0: /* Program Space */
    H0SARAM: org = 0x3F8000, len = 0x2000
    PAGE 1: /* Data Space */
    MOSARAM: org = 0x000000, len = 0x400
    M1SARAM: org = 0x000400, len = 0x400
    L0SARAM: org = 0x008000, len = 0x1000
    L1SARAM: org = 0x009000, len = 0x1000
}

SECTIONS

```

```c
{ .text: > H0SARAM PAGE 0
    .ebss: > M1SARAM PAGE 1
    .cinit: > H0SARAM PAGE 0
    .stack: > MOSARAM PAGE 1
    .reset: > H0SARAM PAGE 0, TYPE = DSECT
}

```
Introduction

The purpose of the F281x C-code header files is to simplify the programming of the many peripherals on the C28x device. Typically, to program a peripheral the programmer needs to write the appropriate values to the different fields within a control register. In its simplest form, the process consists of writing a hex value (or masking a bit field) to the correct address in memory. But, since this can be a burdensome and repetitive task, the C-code header files were created to make this a less complicated task.

The F281x C-code header files are part of a library consisting of C functions, macros, peripheral structures, and variable definitions. Together, this set of files is known as the ‘header files.’ Eventually, the entire collection will be replaced with a CCS add-in called the Chip Support Library (CSL). At that time, there will be a GUI interface used via CCS’s config tool.

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

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

Learning Objectives

- Understand the usage of the F281x C-Code Header Files
- Be able to program peripheral registers
- Understand how peripherals are mapped with linker command file
Module Topics

Peripheral Registers Header Files .......................................................................................................... 3-1

Module Topics........................................................................................................................................ 3-2

Traditional and Structure Approach to C Coding .................................................................................. 3-3

Naming Conventions........................................................................................................................ 3-6

Example of Peripheral Structure .h file .................................................................................................. 3-7

Mapping Structures to Memory ........................................................................................................... 3-8

Linker Command File........................................................................................................................ 3-8

F281x C-Code Header Files ................................................................................................................ 3-9

.h Definition Files........................................................................................................................ 3-10

Global Variable Definition File ......................................................................................................... 3-11

Peripheral Specific Routines............................................................................................................... 3-12

Example Usage Flow ........................................................................................................................ 3-13

Summary ............................................................................................................................................. 3-15
## Traditional and Structure Approach to C Coding

### Traditional Approach to C Coding

```c
#define ADCTRL1      (volatile unsigned int *)0x00007100
#define ADCTRL2      (volatile unsigned int *)0x00007101

void main(void)
{
    *ADCTRL1 = 0x1234;  // write entire register
    *ADCTRL2 |= 0x4000;  // reset sequencer #1
}
```

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

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

### Structure Approach to C Coding

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

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

- **Disadvantages**
  - Can be difficult to remember the structure names (Code Maestro to the rescue!)
  - More to type (again, Code Maestro to the rescue)
The CCS Watch Window using Structures

The CCS Watch Window using #define

3 - 4 C28x - Peripheral Registers Header Files
Is the Structure Approach Efficient?

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

C Source Code

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

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

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

Generated Assembly Code

```assembly
MOVW    DP, #0030
OR      @4, #0x0010
MOVL    XAR4, #0x010000
MOVL    @2, XAR4
AND     @4, #0xffef
```

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

(This example could not have been coded any more efficiently with hand assembly)

Compare with the #define Approach

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

C Source Code

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

// Load new 32-bit period value
*TIMER0TPR32 = 0x00010000;

// Start CPU Timer0
*TIMER0TCR |= 0xffef;
```

Generated Assembly Code

```assembly
MOV     AL,*(0:0x0c04)
ORB     AL, #0x10
MOV     *(0:0x0c04), AL
MOVL    XAR4, #3078
MOVL    XAR5, #65536
MOVL    *+XAR4[0], XAR5
MOV     AL, *(0:0x0c04)
OR      AL, #0xffef
MOV     *(0:0x0c04), AL
```

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

9 Words, 9 cycles
Naming Conventions

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

Structure Naming Conventions

- The DSP281x header files define:
  - All of the peripheral structures
  - All of the register names
  - All of the bit field names
  - All of the register addresses

<table>
<thead>
<tr>
<th>Structure Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PeripheralName.RegisterName.all</td>
<td>Access full 16 or 32-bit register</td>
</tr>
<tr>
<td>PeripheralName.RegisterName.half.LSW</td>
<td>Access low 16-bits of 32-bit register</td>
</tr>
<tr>
<td>PeripheralName.RegisterName.half.MSW</td>
<td>Access high 16-bits of 32-bit register</td>
</tr>
<tr>
<td>PeripheralName.RegisterName.bit.FieldName</td>
<td>Access specified bit fields of register</td>
</tr>
</tbody>
</table>

Notes:
[1] “PeripheralName” are assigned by TI and found in the DSP281x header files. They are a combination of capital and small letters (i.e. CpuTimer0Regs).
[2] “RegisterName” are the same names as used in the data sheet. They are always in capital letters (i.e. TCR, TIM, TPR...).
[3] “FieldName” are the same names as used in the data sheet. They are always in capital letters (i.e. POL, TOG, TSS...).

Code Maestro to the Rescue!

![Code Maestro Screen Shot]

Build Complete:
0 Errors, 0 Warnings, 0 Warnings.

3 - 6 C28x - Peripheral Registers Header Files
Example of Peripheral Structure .h file

Example of Peripheral Structure .h file

Example of Peripheral Structure .h file

Example of Peripheral Structure .h file

Example Adc.h

`/* ADC Individual Register Bit Definitions */
struct ADCCTRL1_BITS {  // bits description
  Uint16  rsvd1:4;              // 3:0 reserved
  Uint16  SEQ_CASC:1;    // 4 Cascaded sequencer mode
  Uint16  rsvd2:1;              // 5 reserved
  Uint16  CONT_RUN:1;   // 6 Continuous run
  Uint16  CPS:1;                // 7 ADC core clock prescaler
  Uint16  ACQ_PS:4;    // 11:8 Acquisition window size
  Uint16  SUSMOD:2;       // 13:12 Emulation suspend mode
  Uint16  RESET:1;    // 14 ADC reset
  Uint16  rsvd3:1;     // 15 reserved
};

/* Allow access to the bit fields or entire register */
union ADCCTRL1_REG {
  Uint16             all;
  struct ADCCTRL1_BITS  bit;
};

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

Example Adc.c or main.c

```
#include  "DSP281x_Device.h"

Void InitAdc(void)
{
  /* Reset the ADC module */
  AdcRegs.ADCTRL1.bit.RESET = 1;
  /* configure the ADC register */
  AdcRegs.ADCTRL1.all = 0x0710;
}
```

C28x - Peripheral Registers Header Files 3 - 7
Mapping Structures to Memory

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

**Example Mapping Structure to Memory**

- **“DATA_SECTION pragma used to assign a unique linker section name to the peripheral structure**
  ```c
  DSP281x_GlobalVariableDefs.C
  #pragma DATA_SECTION(AdcRegs,"AdcRegsFile");
  ```

- **Linker command file maps the unique peripheral section name to the physical memory address**
  ```c
  DSP281x_Headers_BIOS.CMD
  MEMORY
  {
    PAGE1:
    ... 
    ADC: origin=0x007100, length=0x000020
    ...
  }
  SECTIONS
  {
    ...
    AdcRegsFile : > ADC PAGE = 1
    ...
  }
  ```

**Linker Command File**

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

The user can modify their own linker command file, or use the pre-configured linker command files such as EzDSP_RAM_lnk.cmd or F2812.cmd. These files have the peripheral memory regions defined and tied to the individual peripheral.
F281x C-Code Header Files

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

<table>
<thead>
<tr>
<th>DSP281x Header File Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>(<a href="http://www.ti.com">http://www.ti.com</a>, literature # SPRC097)</td>
</tr>
<tr>
<td>◆ Simplifies program of peripherals and other functions</td>
</tr>
<tr>
<td>◆ Takes care of register definitions and addresses</td>
</tr>
<tr>
<td>◆ Header file package consists of:</td>
</tr>
<tr>
<td>- \DSP281x_headers\include ➔ .h files</td>
</tr>
<tr>
<td>- \DSP281x_common\src ➔ .c source files</td>
</tr>
<tr>
<td>- \DSP281x_headers\cmd ➔ linker command files</td>
</tr>
<tr>
<td>- \DSP281x_headers\gel ➔ .gel files for CCS</td>
</tr>
<tr>
<td>- \DSP281x_examples ➔ example programs</td>
</tr>
<tr>
<td>- \doc ➔ documentation</td>
</tr>
</tbody>
</table>

◆ TI has done all of the work for you!

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

A DSP281x_Peripheral.gel GEL file can provide a pull down menu to load peripheral data structures into a watch window. Code Composer Studio can load a GEL file automatically. To include functions to the standard F2812.gel that is part of Code Composer Studio, add:

```c
GEL_LoadGel(“base_path/gel/DSP281x_Peripheral.gel”)
```
The DSP281x_Device.h header file is the main include file. By including this file in the .c source code, all of the peripheral specific .h header files are automatically included. Of course, each specific .h header file can included individually in an application that do not use all the header files, or you can comment out the ones you do not need. (Also includes typedef statements).

<table>
<thead>
<tr>
<th>.h Definition Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>✷ Files found in the</td>
</tr>
<tr>
<td>\DSP281x_headers\include directory</td>
</tr>
<tr>
<td>✷ Define structures and bit fields in peripheral and system registers</td>
</tr>
<tr>
<td>✷ DSP281x_Device.h – main include file will include all other .h files</td>
</tr>
<tr>
<td>• #include “DSP281x_Device.h”</td>
</tr>
<tr>
<td>DSP281x_Device.h</td>
</tr>
<tr>
<td>DSP281x_SysCtrl.h</td>
</tr>
<tr>
<td>DSP281x_Adc.h</td>
</tr>
<tr>
<td>DSP281x_ECan.h</td>
</tr>
<tr>
<td>DSP281x_GPIO.h</td>
</tr>
<tr>
<td>DSP281x_SCI.h</td>
</tr>
<tr>
<td>DSP281x_xINTf.h</td>
</tr>
<tr>
<td>DSP281x_PIEVct.h</td>
</tr>
<tr>
<td>DSP281x_DefaultIsr.h</td>
</tr>
</tbody>
</table>
Global Variable Definition File

With DSP281x_GlobalVariableDefs.c included in the project all the needed variable definitions are globally defined.

Global Variable Definition File

- **DSP281x_GlobalVariableDefs.C**
- Defines all variables to use .h files
- DATA_SECTION pragma used to define data section for each peripheral structure
- Linker will link each structure to the physical address of the peripheral in memory
- Add **DSP281x_GlobalVariableDefs.C** to the Code Composer Studio project
Peripheral Specific Routines

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

Peripheral Specific Routines

- Contains peripheral specific initialization routines and other support functions

<table>
<thead>
<tr>
<th>File Name 1</th>
<th>File Name 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSP281x_SysCtrl.c</td>
<td>DSP281x_Gpio.c</td>
</tr>
<tr>
<td>DSP281x_PiecCtrl.c</td>
<td>DSP281x_Mcbsp.c</td>
</tr>
<tr>
<td>DSP281x_Adc.c</td>
<td>DSP281x_Sci.c</td>
</tr>
<tr>
<td>DSP281x_CpuTimers.c</td>
<td>DSP281x_Spi.c</td>
</tr>
<tr>
<td>DSP281x_ECan.c</td>
<td>DSP281x_Xintf.c</td>
</tr>
<tr>
<td>DSP281x_Ev.c</td>
<td>DSP281x_Xintrupt.c</td>
</tr>
<tr>
<td>DSP281x_PieVect.c</td>
<td>DSP281x_DefaultIsr.c</td>
</tr>
</tbody>
</table>

Workshop lab files based on above files with modifications

Other files included in the packet:

Other Files in Packet

- Linker.cmd files
  - DSP281x_Headers_BIOS.cmd
  - DSP281x_Headers_nonBIOS.cmd
  - Contains memory allocation for all peripheral structure definitions included in C-code header file package

- DSP281x_CodeStartBranch.asm
  - Used to redirect code execution when booting

```
.ref _c_int00
.sect "codestart"
LB _c_int00 ;branch to start of boot.asm in RTS library
```

Workshop lab files based on above files with modifications
Example Usage Flow

// Step 0. Include required header files
// DSP281x_Device.h: device specific definitions #include statements
// for all of the peripheral.h definition files.
// DSP281x_Exception.h is specific for the given example.

#include "DSP281x_Device.h"

// Prototype statements for functions found within this file.
interrupt void cpu_timer0_isr(void);

void main(void)
{

    // Step 1. Initialize System Control registers, PLL, WatchDog, Clocks to
default state:
    // This function is found in the DSP281x_SysCtrl.c file.
    InitSysCtrl();

    // Step 2. Select GPIO for the device or for the specific application:
    // This function is found in the DSP281x_Gpio.c file.
    InitGpio();

    // Step 3. Initialize PIE vector table:
    // The PIE vector table is initialized with pointers to shell Interrupt
    // Service Routines (ISR). The shell routines are found in
    DSP281x_DefaultIsr.c.
    // Insert user specific ISR code in the appropriate shell ISR routine in
    // the DSP281x_DefaultIsr.c file.

    // Disable and clear all CPU interrupts:
    DINT;
    IER = 0x0000;
    IFR = 0x0000;

    // Initialize Pie Control Registers To Default State:
    // This function is found in the DSP281x_PieCtrl.c file.
    InitPieCtrl();

    // Initialize the PIE Vector Table To a Known State:
    // This function is found in DSP281x_PieVect.c.
    // This function populates the PIE vector table with pointers
    // to the shell ISR functions found in DSP281x_DefaultIsr.c.
    InitPieVectTable();

    // Step 4. Initialize all the Device Peripherals to a known state:
    // This function is found in DSP281x_InitPeripherals.c
    InitPeripherals();

    // Step 5. User specific functions, Reassign vectors (optional), Enable Inter-
    // rupts:

    // Initialize CPU Timer 0:
    // > Set Up For 1 Second Interrupt Period
    // > Point To "cpu_timer0_isr" function

    // Reassign CPU-Timer0 ISR.
    // Reassign the PIE vector for TINT0 to point to a different ISR then
    // the shell routine found in DSP281x_DefaultIsr.c.
    // This is done if the user does not want to use the shell ISR routine
    // but instead wants to use their own ISR. This step is optional:
EALLOW;    // This is needed to write to EALLOW protected registers
PieVectTable.TINT0 = &cpu_timer0_isr;
EDIS;      // This is needed to disable write to EALLOW protected registers

// Include application specific functions. This is for this example:

// Configure CPU-Timer 0 to interrupt every second:
ConfigCpuTimer(&CpuTimer0, 100, 1000000);  // 100MHz CPU Freq, 1 second
// Period (in uSeconds)
StartCpuTimer0();

// Enable CPU INT1 which is connected to CPU-Timer 0:
IER |= M_INT1;

// Enable TINT0 in the PIE: Group 1 interrupt 7
PieCtrlRegs.PIEIER1.bit.INTx7 = 1;

// Enable global Interrupts and higher priority real-time debug events:
EINT;     // Enable Global interrupt INTM
ERTM;     // Enable Global realtime interrupt DBGIM

// Step 6. IDLE loop. Just sit and loop forever (optional):
for(;;);

// Step 7. Insert all local Interrupt Service Routines (ISRs) and functions here:
// If local ISRs are used, reassign vector addresses in vector table as shown in Step 5

interrupt void cpu_timer0_isr(void)
{
    CpuTimer0.InterruptCount++;

    // Acknowledge this interrupt to receive more interrupts from group 1
    PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;
}
Summary

Peripheral Register Header Files

Summary

◆ Easier code development
◆ Easy to use
◆ Generates most efficient Code
◆ Increases Effectiveness of CCS Watch Window
◆ TI has already done all the work!
  • Download literature # SPRC097 from www.ti.com
Reset and Interrupts

Introduction

This module describes the interrupt process and explains how the peripheral Interrupt expansion (PIE) works.

Learning Objectives

- Describe the C28x reset process and post-reset device state
- List the event sequence during an interrupt
- Describe the C28x interrupt structure
Module Topics

Reset and Interrupts ........................................................................................................... 4-1

Core Interrupt Lines ........................................................................................................... 4-2
  Reset .......................................................................................................................... 4-3
  Reset - Bootloader ...................................................................................................... 4-5

Interrupt Sources .............................................................................................................. 4-7
  Interrupt Processing ..................................................................................................... 4-7
  Peripheral Interrupt Expansion (PIE) ........................................................................... 4-9
  PIE Interrupt Vector Table .......................................................................................... 4-10
  Interrupt Response and Latency .................................................................................. 4-13
Core Interrupt Lines

- 2 non-maskable interrupts (RS, “selectable” NMI)
- 14 maskable interrupts (INT1 – INT14)

Reset

C28x Reset Sources
Core Interrupt Lines

Register Bits Initialized at Reset

<table>
<thead>
<tr>
<th>Register Bits defined by reset</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 0x3F FFC0</td>
<td></td>
<td>PC loaded with reset vector</td>
</tr>
<tr>
<td>ACC 0x0000 0000</td>
<td></td>
<td>Accumulator cleared</td>
</tr>
<tr>
<td>XAR0 - XAR7 0x0000 0000</td>
<td></td>
<td>Auxiliary Registers</td>
</tr>
<tr>
<td>DP 0x0000</td>
<td></td>
<td>Data Page pointer points to page 0</td>
</tr>
<tr>
<td>P 0x0000 0000</td>
<td></td>
<td>P register cleared</td>
</tr>
<tr>
<td>XT 0x0000 0000</td>
<td></td>
<td>XT register cleared</td>
</tr>
<tr>
<td>SP 0x0400</td>
<td></td>
<td>Stack Pointer to address 0400</td>
</tr>
<tr>
<td>RPC 0x00 0000</td>
<td></td>
<td>Return Program Counter cleared</td>
</tr>
<tr>
<td>IFR 0x0000</td>
<td></td>
<td>no pending interrupts</td>
</tr>
<tr>
<td>IER 0x0000</td>
<td></td>
<td>maskable interrupts disabled</td>
</tr>
<tr>
<td>DBGIER 0x0000</td>
<td></td>
<td>debug interrupts disabled</td>
</tr>
</tbody>
</table>

Control Bits Initialized at Reset

**Status Register 0 (ST0)**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SXM = 0</td>
<td>Sign extension off</td>
</tr>
<tr>
<td>OVM = 0</td>
<td>Overflow mode off</td>
</tr>
<tr>
<td>TC = 0</td>
<td>test/control flag</td>
</tr>
<tr>
<td>C = 0</td>
<td>carry bit</td>
</tr>
<tr>
<td>Z = 0</td>
<td>zero flag</td>
</tr>
<tr>
<td>N = 0</td>
<td>negative flag</td>
</tr>
<tr>
<td>V = 0</td>
<td>overflow bit</td>
</tr>
<tr>
<td>PM = 000</td>
<td>set to left-shift-by-1</td>
</tr>
<tr>
<td>OVC = 000000</td>
<td>overflow counter</td>
</tr>
</tbody>
</table>

**Status Register 1 (ST1)**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTM = 1</td>
<td>Disable all maskable interrupts - global</td>
</tr>
<tr>
<td>DBGM = 1</td>
<td>Emulation access/events disabled</td>
</tr>
<tr>
<td>PAGE0 = 0</td>
<td>Stack addressing mode enabled/Direct addressing disabled</td>
</tr>
<tr>
<td>VMAP = 1</td>
<td>Interrupt vectors mapped to PM 0x3F FFC0 – 0x3F FFFF</td>
</tr>
<tr>
<td>SPA = 0</td>
<td>stack pointer even address alignment status bit</td>
</tr>
<tr>
<td>LOOP = 0</td>
<td>Loop instruction status bit</td>
</tr>
<tr>
<td>EALLOW = 0</td>
<td>emulation access enable bit</td>
</tr>
<tr>
<td>IDLESTAT = 0</td>
<td>Idle instruction status bit</td>
</tr>
<tr>
<td>AMODE = 0</td>
<td>C27x/C28x addressing mode</td>
</tr>
<tr>
<td>OBJMODE = 0</td>
<td>C27x object mode</td>
</tr>
<tr>
<td>M0M1MAP = 1</td>
<td>mapping mode bit</td>
</tr>
<tr>
<td>XF = 0</td>
<td>XF status bit</td>
</tr>
<tr>
<td>ARP = 0</td>
<td>ARP points to AR0</td>
</tr>
</tbody>
</table>
Reset - Bootloader

Reset – Bootloader

<table>
<thead>
<tr>
<th>GPIO pins</th>
<th>F4</th>
<th>F12</th>
<th>F3</th>
<th>F2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

- Jump to FLASH address 0x3F 7FF6
- Jump to H0 SARAM address 0x3F 8000
- Jump to OTP address 0x3D 7800
- Bootload external EEPROM to on-chip memory via SPI port
- Bootload code to on-chip memory via SCI-A port
- Bootload code to on-chip memory via GPIO port B (parallel)

Notes:
- F2810 XMPNMC tied low internal to device
- XMPNMC refers to input signal
- MP/MC is status bit in XINTFCNF2 register
- XMPNMC only sampled at reset

* Boot ROM software configures the device for C28x mode before jump
## Core Interrupt Lines

### Reset Code Flow - Summary

- **0x3D 7800**: OTP (1K)
- **0x3D 8000**: FLASH (128K)
- **0x3F 8000**: H0 SARAM (8K)
- **0x3F 8000**: Boot ROM (4K)
- **0x3F F000**: Boot Code
  - Boot Code Execution Entry Point Determined By GPIO Pins
- **0x3F F000**: Bootloading Routines (SPI, SCI-A, Parallel Load)

**Reset**

**Execution Entry Point Determined By GPIO Pins**

**Bootloading Routines**

**Parallel Load**
Interrupt Sources

Interrupt Sources

Internal Sources

- TINT2
- TINT1
- TINT0

EV and Non-EV Peripherals
- EV, ADC, SPI, SCI, McBSP, CAN

External Sources

- XINT1
- XINT2
- PDPINTx
- RS
- XNMI_XINT13

Interrupt Processing

Maskable Interrupt Processing

Conceptual Core Overview

- Core Interrupt
- (IFR) "Latch"
- (IER) "Switch"
- (INTM) "Global Switch"

- INT1
- INT2
- INT3
- INT12
- INT13
- INT14

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

Interrupt Flag Register (IFR)

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTOSINT</td>
<td>DLOGINT</td>
<td>INT14</td>
<td>INT13</td>
<td>INT12</td>
<td>INT11</td>
<td>INT10</td>
<td>INT9</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>INT8</td>
<td>INT7</td>
<td>INT6</td>
<td>INT5</td>
<td>INT4</td>
<td>INT3</td>
<td>INT2</td>
<td>INT1</td>
</tr>
</tbody>
</table>

Pending: IFR Bit = 1
Absent: IFR Bit = 0

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

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

Interrupt Enable Register (IER)

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTOSINT</td>
<td>DLOGINT</td>
<td>INT14</td>
<td>INT13</td>
<td>INT12</td>
<td>INT11</td>
<td>INT10</td>
<td>INT9</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>INT8</td>
<td>INT7</td>
<td>INT6</td>
<td>INT5</td>
<td>INT4</td>
<td>INT3</td>
<td>INT2</td>
<td>INT1</td>
</tr>
</tbody>
</table>

Enable: Set IER Bit = 1
Disable: Clear IER Bit = 0

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

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

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

```
/*** Global Interrupts ***/
asms(" CLRC INTM"); //enable global interrupts
asms(" SETC INTM"); //disable global interrupts
```

Peripheral Interrupt Expansion (PIE)

```
Peripheral Interrupt Expansion - PIE
```

```
// 28x Core Interrupt logic
```

```
// Peripheral Interrupts
```

```
// Interrupt Group 1
```

```
// PIE module for 96 Interrupts
```

```
// INT13 (TIMT1 / XINT13)
```

```
// INT14 (TINT2)
```

```
// NMI
```

```
// INT1 – INT 12
```

```
// 12 Interrupts
```

```
// 28x Core Interrupt logic
```

```
// 28x Core
```
Interrupt Sources

#PIE Registers

**PIEIFRx register** (x = 1 to 12)

<table>
<thead>
<tr>
<th>15</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>reserved</td>
<td>INTx.8</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>reserved</td>
<td>INTx.7</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>reserved</td>
<td>INTx.6</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>reserved</td>
<td>INTx.5</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>reserved</td>
<td>INTx.4</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>reserved</td>
<td>INTx.3</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>reserved</td>
<td>INTx.2</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>reserved</td>
<td>INTx.1</td>
</tr>
</tbody>
</table>

**PIEIERRx register** (x = 1 to 12)

<table>
<thead>
<tr>
<th>15</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>reserved</td>
<td>INTx.8</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>reserved</td>
<td>INTx.7</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>reserved</td>
<td>INTx.6</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>reserved</td>
<td>INTx.5</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>reserved</td>
<td>INTx.4</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>reserved</td>
<td>INTx.3</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>reserved</td>
<td>INTx.2</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>reserved</td>
<td>INTx.1</td>
</tr>
</tbody>
</table>

**PIE Interrupt Acknowledge Register (PIEACK)**

<table>
<thead>
<tr>
<th>15</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>15</td>
<td>8</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>reserved</td>
<td>PIEACKx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PIECTRL register**

<table>
<thead>
<tr>
<th>15</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>reserved</td>
<td>PIEVECT</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ENPIE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#include “DSP281x_Device.h”

PieCtrlRegs.PIEIFR1.bit.INTx4 = 1; //manually set IFR for XINT1 in PIE group 1
PieCtrlRegs.PIEIER3.bit.INTx5 = 1; //enable CAPINT1 in PIE group 3
PieCtrlRegs.PIEACK.all = 0x0004; //acknowledge the PIE group 3
PieCtrlRegs.PIECTRL.bit.ENPIE = 1; //enable the PIE

**PIE Interrupt Vector Table**

**Default Interrupt Vector Table at Reset**

<table>
<thead>
<tr>
<th>Prio</th>
<th>Vector</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Reset</td>
<td>00</td>
</tr>
<tr>
<td>5</td>
<td>Int 1</td>
<td>02</td>
</tr>
<tr>
<td>6</td>
<td>Int 2</td>
<td>04</td>
</tr>
<tr>
<td>7</td>
<td>Int 3</td>
<td>06</td>
</tr>
<tr>
<td>8</td>
<td>Int 4</td>
<td>08</td>
</tr>
<tr>
<td>9</td>
<td>Int 5</td>
<td>0A</td>
</tr>
<tr>
<td>10</td>
<td>Int 6</td>
<td>0C</td>
</tr>
<tr>
<td>11</td>
<td>Int 7</td>
<td>0E</td>
</tr>
<tr>
<td>12</td>
<td>Int 8</td>
<td>10</td>
</tr>
<tr>
<td>13</td>
<td>Int 9</td>
<td>12</td>
</tr>
<tr>
<td>14</td>
<td>Int 10</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>Int 11</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>Int 12</td>
<td>18</td>
</tr>
<tr>
<td>17</td>
<td>Int 13</td>
<td>1A</td>
</tr>
<tr>
<td>18</td>
<td>Int 14</td>
<td>1C</td>
</tr>
<tr>
<td>19</td>
<td>DlogInt</td>
<td>1E</td>
</tr>
<tr>
<td>2</td>
<td>RtosInt</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>EmulInt</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>NMI</td>
<td>24</td>
</tr>
<tr>
<td>-</td>
<td>Illegal</td>
<td>26</td>
</tr>
<tr>
<td>-</td>
<td>User 1-12</td>
<td>28-3E</td>
</tr>
</tbody>
</table>

Default Vector Table Remapped when ENPIE = 1

Memory

0x00 0D00

0x3F FFC0

0x3F FFFF

PIE Vectors 256 W

BROM Vectors 64 W

PIE vector generated by config Tool
Used to initialize PIE vectors

C28x - Reset and Interrupts
### PIE Vector Mapping (ENPIE = 1)

<table>
<thead>
<tr>
<th>Vector name</th>
<th>PIE vector address</th>
<th>PIE vector Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not used</td>
<td>0x00 0D00</td>
<td>Reset Vector Never Fetched Here</td>
</tr>
<tr>
<td>INT1</td>
<td>0x00 0D02</td>
<td>INT1 re-mapped below</td>
</tr>
<tr>
<td>INT12</td>
<td>0x00 0D18</td>
<td>INT12 re-mapped below</td>
</tr>
<tr>
<td>INT13</td>
<td>0x00 0D1A</td>
<td>XINT1 Interrupt Vector</td>
</tr>
<tr>
<td>INT14</td>
<td>0x00 0D1C</td>
<td>Timer2 - RTOS Vector</td>
</tr>
<tr>
<td>Datalog</td>
<td>0x00 0D1D</td>
<td>Data logging vector</td>
</tr>
<tr>
<td>USER11</td>
<td>0x00 0D3E</td>
<td>User defined TRAP</td>
</tr>
<tr>
<td>INT1.1</td>
<td>0x00 0D40</td>
<td>PIEINT1.1 interrupt vector</td>
</tr>
<tr>
<td>INT12.1</td>
<td>0x00 0D4E</td>
<td>PIEINT12.1 interrupt vector</td>
</tr>
<tr>
<td>INT12.8</td>
<td>0x00 0DF0</td>
<td>PIEINT12.8 interrupt vector</td>
</tr>
</tbody>
</table>

- PIE vector space - 0x00 0D00 – 256 Word memory in Data space
- RESET and INT1-INT12 vector locations are Re-mapped
- CPU vectors are remapped to 0x00 0D00 in Data space

### F281x PIE Interrupt Assignment Table

<table>
<thead>
<tr>
<th>INTx.8</th>
<th>INTx.7</th>
<th>INTx.6</th>
<th>INTx.5</th>
<th>INTx.4</th>
<th>INTx.3</th>
<th>INTx.2</th>
<th>INTx.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT1</td>
<td>WAKEINT</td>
<td>TINT0</td>
<td>ADCINT</td>
<td>XINT2</td>
<td>XINT1</td>
<td>PDPINTB</td>
<td>PDPINTA</td>
</tr>
<tr>
<td>INT2</td>
<td>T1OFINT</td>
<td>T1UFINT</td>
<td>T1CINT</td>
<td>T1PINT</td>
<td>CMP3INT</td>
<td>CMP2INT</td>
<td>CMP1INT</td>
</tr>
<tr>
<td>INT3</td>
<td>CAPINT3</td>
<td>CAPINT2</td>
<td>CAPINT1</td>
<td>T2OFINT</td>
<td>T2UFINT</td>
<td>T2CINT</td>
<td>T2PINT</td>
</tr>
<tr>
<td>INT4</td>
<td>T3OFINT</td>
<td>T3UFINT</td>
<td>T3CINT</td>
<td>T3PINT</td>
<td>CMP6INT</td>
<td>CMP5INT</td>
<td>CMP4INT</td>
</tr>
<tr>
<td>INT5</td>
<td>CAPINT6</td>
<td>CAPINT5</td>
<td>CAPINT4</td>
<td>T4OFINT</td>
<td>T4UFINT</td>
<td>T4CINT</td>
<td>T4PINT</td>
</tr>
<tr>
<td>INT6</td>
<td>MXINT</td>
<td>MRINT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT9</td>
<td>ECAN1INT</td>
<td>ECAN0INT</td>
<td>SCITXINTB</td>
<td>SCIRXINTB</td>
<td>SCITXINTA</td>
<td>SCIRXINTA</td>
<td></td>
</tr>
<tr>
<td>INT10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Interrupt Sources**

### Device Vector Mapping - Summary

**RESET**

**MPNMC = 0 (on-chip ROM memory)**
- Reset Vector <0x3F FFC0> = Boot-ROM Code
- Flash Entry Point <0x3F 7FF6> = LB _c_int00
- User Code Start < _c_int00 >

**Initialization ( )**
- EALLOW
- Load PIE Vectors
- Enable the PIEIER
- Enable PIECTRL
- Enable Core IER
- Enable INTM
- EDIS

**MPNMC = 1 (external memory XINTF)**
- Reset Vector <0x3F FFC0> = _c_int00
- User Code Start < _c_int00 >

---

- _c_int00:
  - . .
  - CALL main()

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

- PIE Vector Table
  - 256 Word RAM
  - 0x00 0D00 ... 0DFF
Interrupt Response and Latency

Interrupt Response - Hardware Sequence

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

Note: some actions occur simultaneously, none are interruptible

Interrupt Latency

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

Introduction

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

Learning Objectives

- OSC/PLL Clock Module
- Watchdog Timer
- Low Power Modes
- General Purpose Digital I/O
- EALLOW Protected Registers
Module Topics

System Initialization.......................................................................................................................... 5-1

Oscillator/PLL Clock Module........................................................................................................ 5-2
Watchdog Timer............................................................................................................................. 5-5
Low Power Modes.......................................................................................................................... 5-9
General-Purpose Digital I/O ........................................................................................................... 5-11
EALLOW Protected Registers........................................................................................................ 5-14
Lab 5: System Initialization ........................................................................................................... 5-15
The OSC/PLL clock module provides all the necessary clocking signals for C28x devices. The PLL has a 4-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 X1/CLKIN pin. The watchdog receives a clock signal from OSCCLK. The C28x core provides a SYSCLKOUT clock signal. This signal is prescaled to provide a clock source for the on-chip peripherals through the high-speed and low-speed peripheral clock prescalers.
The peripheral clock control register allows individual peripheral clock signals to be enabled or disabled. If a peripheral is not being used, its clock signal could be disabled, thus reducing power consumption.
Watchdog Timer

- **Resets the C28x if the CPU crashes**
  - Watchdog counter runs independent of CPU
  - If counter overflows, reset or interrupt is triggered
  - CPU must write correct data key sequence to reset the counter before overflow
- **Watchdog must be serviced (or disabled) within ~4.37ms after reset (30 MHz OSCCLK for 150 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 4.37 ms (for a 150 MHz device) after any reset before a watchdog initiated reset will occur. 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.
Watchdog Timer Module

(OSCCLK) /512

6 - Bit Free - Running Counter

CLR

/64
/32
/16
/8
/4
/2

WDPS

WDCR . 7 - 0

8 - Bit Watchdog Counter

CLR

WDCNTR . 7 - 0

One-Cycle Delay

WDCR . 5 - 3

WDCK 2 - 0

3

3

1

Bad WDCR Key

WDCR . 6

WDDIS

WDCR . 2 = 0

WDPS

WDOVERRIDE

SCSR . 0

WDINT

WDRST

WDFLAG

WDCR . 7

SCSR . 1

WDENINT

55 + AA Detector

Good Key

Bad Key

Watchdog Reset Key Register

WDKEY . 7 - 0

Watchdog Period Selection

<table>
<thead>
<tr>
<th>WDPS Bits</th>
<th>FRC rollover</th>
<th>C28x timeout period @ 150 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>00x:</td>
<td>1</td>
<td>4.37 ms</td>
</tr>
<tr>
<td>010:</td>
<td>2</td>
<td>8.74 ms</td>
</tr>
<tr>
<td>011:</td>
<td>4</td>
<td>17.48 ms</td>
</tr>
<tr>
<td>100:</td>
<td>8</td>
<td>34.96 ms</td>
</tr>
<tr>
<td>101:</td>
<td>16</td>
<td>69.92 ms</td>
</tr>
<tr>
<td>110:</td>
<td>32</td>
<td>139.84 ms</td>
</tr>
<tr>
<td>111:</td>
<td>64</td>
<td>279.68 ms</td>
</tr>
</tbody>
</table>

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

(for OSCCLK = 30 MHz ⇒ (1/30 MHz) * (512*256) = 4.37 ms)
Watchdog Timer Control Register

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

<table>
<thead>
<tr>
<th>15-8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>reserved</td>
<td>WDFLAG</td>
<td>WDDIS</td>
<td>WDCHK2</td>
<td>WDCHK1</td>
<td>WDCHK0</td>
<td>WDPS2</td>
<td>WDPS1</td>
<td>WDPS0</td>
</tr>
</tbody>
</table>

**Logic Check Bits**
- Write as 101 or reset immediately triggered
- Write 1 to disable
  - (Functions only if WD OVERRIDE bit in SCSR is equal to 1)

Resetting the Watchdog

**WDKEY @ 0x007025**

- Allowable write values:
  - 55h - counter enabled for reset on next AAh write
  - AAh - counter set to zero if reset enabled
- Writing any other value immediately triggers a CPU reset
- Watchdog should not be serviced solely in an ISR
  - If main code crashes, but interrupt continues to execute, the watchdog will not catch the crash
  - Could put the 55h WDKEY in the main code, and the AAh WDKEY in an ISR; this catches main code crashes and also ISR crashes
**WDKEY Write Results**

<table>
<thead>
<tr>
<th>Sequential Step</th>
<th>Value Written to WDKEY</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AAh</td>
<td>No action</td>
</tr>
<tr>
<td>2</td>
<td>AAh</td>
<td>No action</td>
</tr>
<tr>
<td>3</td>
<td>55h</td>
<td>WD counter enabled for reset on next AAh write</td>
</tr>
<tr>
<td>4</td>
<td>55h</td>
<td>WD counter enabled for reset on next AAh write</td>
</tr>
<tr>
<td>5</td>
<td>55h</td>
<td>WD counter enabled for reset on next AAh write</td>
</tr>
<tr>
<td>6</td>
<td>AAh</td>
<td>WD counter is reset</td>
</tr>
<tr>
<td>7</td>
<td>AAh</td>
<td>No action</td>
</tr>
<tr>
<td>8</td>
<td>55h</td>
<td>WD counter enabled for reset on next AAh write</td>
</tr>
<tr>
<td>9</td>
<td>AAh</td>
<td>WD counter is reset</td>
</tr>
<tr>
<td>10</td>
<td>55h</td>
<td>WD counter enabled for reset on next AAh write</td>
</tr>
<tr>
<td>11</td>
<td>23h</td>
<td>CPU reset triggered due to improper write value</td>
</tr>
</tbody>
</table>

**System Control and Status Register**

SCSR @ 0x007022 (lab file: SysCtrl.c)

WD Override (protect bit)
After RESET - bit gives user ability to disable WD by setting WDDIS bit=1 in WDCR
* clear only bit and defaults to 1 after reset
* 0 = protects WD from being disabled by s/w
* bit cannot be set to 1 by s/w (clear-only by writing 1)
* 1 = (default value) allows WD to be disabled using WDDIS bit in WDCR
  * once cleared, bit cannot set to 1 by s/w

WD Interrupt Status (read only)
0 = active
1 = not active

WD Enable Interrupt
0 = WD generates a DSP reset
1 = WD generates a WDINT interrupt
Low Power Modes

### Low Power Modes

<table>
<thead>
<tr>
<th>Low Power Mode</th>
<th>CPU Logic Clock</th>
<th>Peripheral Logic Clock</th>
<th>Watchdog Clock</th>
<th>PLL / OSC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal Run</td>
<td>on</td>
<td>on</td>
<td>on</td>
<td>on</td>
</tr>
<tr>
<td>IDLE</td>
<td>off</td>
<td>on</td>
<td>on</td>
<td>on</td>
</tr>
<tr>
<td>STANDBY</td>
<td>off</td>
<td>off</td>
<td>on</td>
<td>on</td>
</tr>
<tr>
<td>HALT</td>
<td>off</td>
<td>off</td>
<td>off</td>
<td>off</td>
</tr>
</tbody>
</table>

#### Low Power Mode Control Register 0
LPMCR0 @ 0x00701E (lab file: SysCtrl.c)

- Qualify before waking from STANDBY mode:
  - 000000 = 2 OSCCLKs
  - 000001 = 3 OSCCLKs
  - \( \vdots \)
  - 111111 = 65 OSCCLKs

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-8</td>
<td>reserved</td>
</tr>
<tr>
<td>7-2</td>
<td>QUALSTDBY</td>
</tr>
<tr>
<td>1</td>
<td>LPM1</td>
</tr>
<tr>
<td>0</td>
<td>LPM0</td>
</tr>
</tbody>
</table>

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

- 00 = Idle
- 01 = Standby
- 1x = Halt
**Low Power Mode Control Register 1**

LPMCR1 @ 0x00701F (lab file: SysCtrl.c)

<table>
<thead>
<tr>
<th>Bit 15</th>
<th>Bit 14</th>
<th>Bit 13</th>
<th>Bit 12</th>
<th>Bit 11</th>
<th>Bit 10</th>
<th>Bit 9</th>
<th>Bit 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>CANRXA</td>
<td>SCIRXB</td>
<td>SCIRXA</td>
<td>C6TRIP</td>
<td>C5TRIP</td>
<td>C4TRIP</td>
<td>C3TRIP</td>
<td>C2TRIP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1TRIP</td>
<td>T4CTRIP</td>
<td>T3CTRIP</td>
<td>T2CTRIP</td>
<td>T1CTRIP</td>
<td>WDINT</td>
<td>XNMI</td>
<td>XINT1</td>
</tr>
</tbody>
</table>

Wake device from STANDBY mode
0 = disable
1 = enable

---

**Low Power Mode Exit**

<table>
<thead>
<tr>
<th>Low Power Mode</th>
<th>Exit Interrupt</th>
<th>RESET</th>
<th>External or Wake up Interrupts</th>
<th>Enabled Peripheral Interrupts</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDLE</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>STANDBY</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>HALT</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

Note: External or Wake up include XINTx, PDPINT, TxCTRIP,
CxTRIP, NMI, CAN, SPI, SCI, WD
## General-Purpose Digital I/O

### F2812 GPIO Pin Assignment

<table>
<thead>
<tr>
<th>GPIO A</th>
<th>GPIO B</th>
<th>GPIO D</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPIOA0 / PWM1</td>
<td>GPIOB0 / PWM7</td>
<td>GPIOD0 / T1CTRIP_PDPINTA</td>
</tr>
<tr>
<td>GPIOA1 / PWM2</td>
<td>GPIOB1 / PWM8</td>
<td>GPIOD1 / T2CTRIP7_EVASOC</td>
</tr>
<tr>
<td>GPIOA2 / PWM3</td>
<td>GPIOB2 / PWM9</td>
<td>GPIOD5 / T3CTRIP_PDPINTB</td>
</tr>
<tr>
<td>GPIOA3 / PWM4</td>
<td>GPIOB3 / PWM10</td>
<td>GPIOD6 / T4CTRIP7_EVBSSOC</td>
</tr>
<tr>
<td>GPIOA4 / PWM5</td>
<td>GPIOB4 / PWM11</td>
<td></td>
</tr>
<tr>
<td>GPIOA5 / PWM6</td>
<td>GPIOB5 / PWM12</td>
<td></td>
</tr>
<tr>
<td>GPIOA6 / T1PWM_T1CMP</td>
<td>GPIOD6 / T1PWM_T1CMP</td>
<td></td>
</tr>
<tr>
<td>GPIOA7 / T2PWM_T2CMP</td>
<td>GPIOD6 / T2PWM_T2CMP</td>
<td></td>
</tr>
<tr>
<td>GPIOA8 / C1QEP1</td>
<td>GPIOD8 / T4PWM_T4CMP</td>
<td></td>
</tr>
<tr>
<td>GPIOA9 / CAP2_QEP2</td>
<td>GPIOD9 / T5PWM_T5CMP</td>
<td></td>
</tr>
<tr>
<td>GPIOA10 / C1TRIP</td>
<td>GPIOD10 / C1TRIP</td>
<td></td>
</tr>
<tr>
<td>GPIOA11 / TDIRA</td>
<td>GPIOD11 / TDIRB</td>
<td></td>
</tr>
<tr>
<td>GPIOA12 / TCLKIN</td>
<td>GPIOD12 / TCLKIN</td>
<td></td>
</tr>
<tr>
<td>GPIOA13 / C1TRIP</td>
<td>GPIOD13 / C1TRIP</td>
<td></td>
</tr>
<tr>
<td>GPIOA14 / C2TRIP</td>
<td>GPIOD14 / C2TRIP</td>
<td></td>
</tr>
<tr>
<td>GPIOA15 / C3TRIP</td>
<td>GPIOD15 / C3TRIP</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GPIO F</th>
<th>GPIO G</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPIOF0 / SPISIMOA</td>
<td>GPIOG4 / SCITXDB</td>
</tr>
<tr>
<td>GPIOF1 / SPISIMIA</td>
<td>GPIOG5 / SCIRXDB</td>
</tr>
<tr>
<td>GPIOF2 / SPICLKA</td>
<td></td>
</tr>
<tr>
<td>GPIOF3 / SPISTEA</td>
<td></td>
</tr>
<tr>
<td>GPIOF4 / SCITXDA</td>
<td></td>
</tr>
<tr>
<td>GPIOF5 / SCIRXDA</td>
<td></td>
</tr>
<tr>
<td>GPIOF6 / CANTXA</td>
<td></td>
</tr>
<tr>
<td>GPIOF7 / CANRXA</td>
<td></td>
</tr>
<tr>
<td>GPIOF8 / MCLKX</td>
<td></td>
</tr>
<tr>
<td>GPIOF9 / MCLKRA</td>
<td></td>
</tr>
<tr>
<td>GPIOF10 / MPSX</td>
<td></td>
</tr>
<tr>
<td>GPIOF11 / MFSR</td>
<td></td>
</tr>
<tr>
<td>GPIOF12 / MDX</td>
<td></td>
</tr>
<tr>
<td>GPIOF13 / MDR</td>
<td></td>
</tr>
<tr>
<td>GPIOF14 / XF</td>
<td></td>
</tr>
</tbody>
</table>

### C28x GPIO Register Structure (lab file: Gpio.c)

- **GPIO A Mux Control Register (GPAMUX)**
- **GPIO A Direction Control Register (GPADIR)**
- **GPIO B Mux Control Register (GPBMUX)**
- **GPIO B Direction Control Register (GPBDIR)**
- **GPIO D Mux Control Register (GPDMUX)**
- **GPIO D Direction Control Register (GPDDIR)**
- **GPIO E Mux Control Register (GPEMUX)**
- **GPIO E Direction Control Register (GPEDIR)**
- **GPIO F Mux Control Register (GPFMUX)**
- **GPIO F Direction Control Register (GPFDIR)**
- **GPIO G Mux Control Register (GPGMUX)**
- **GPIO G Direction Control Register (GPGDIR)**

**Note:** GPIOxx are pin functions at reset. GPIO A, B, D, E include Input Qualification feature.
C28x GPIO Functional Block Diagram

Some digital I/O and peripheral I/O input signals include an Input Qualification feature.

C28x GPIO MUX/DIR Registers (lab file: GPIO.c)

<table>
<thead>
<tr>
<th>Address</th>
<th>Register</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0070C0</td>
<td>GPAMUX</td>
<td>GPIO A Mux Control Register</td>
</tr>
<tr>
<td>0x0070C1</td>
<td>GPADIR</td>
<td>GPIO A Direction Control Register</td>
</tr>
<tr>
<td>0x0070C2</td>
<td>GPAQUAL</td>
<td>GPIO A Input Qualification Control Register</td>
</tr>
<tr>
<td>0x0070C4</td>
<td>GPBMUX</td>
<td>GPIO B Mux Control Register</td>
</tr>
<tr>
<td>0x0070C5</td>
<td>GPBDIR</td>
<td>GPIO B Direction Control Register</td>
</tr>
<tr>
<td>0x0070C6</td>
<td>GPBQUAL</td>
<td>GPIO B Input Qualification Control Register</td>
</tr>
<tr>
<td>0x0070CC</td>
<td>GPDMUX</td>
<td>GPIO D Mux Control Register</td>
</tr>
<tr>
<td>0x0070CD</td>
<td>GPDDR</td>
<td>GPIO D Direction Control Register</td>
</tr>
<tr>
<td>0x0070CE</td>
<td>GPDQUAL</td>
<td>GPIO D Input Qualification Control Register</td>
</tr>
<tr>
<td>0x0070D0</td>
<td>GPEMUX</td>
<td>GPIO E Mux Control Register</td>
</tr>
<tr>
<td>0x0070D1</td>
<td>GPEDIR</td>
<td>GPIO E Direction Control Register</td>
</tr>
<tr>
<td>0x0070D2</td>
<td>GPEQUAL</td>
<td>GPIO E Input Qualification Control Register</td>
</tr>
<tr>
<td>0x0070D4</td>
<td>GPFMUX</td>
<td>GPIO F Mux Control Register</td>
</tr>
<tr>
<td>0x0070D5</td>
<td>GPFDIR</td>
<td>GPIO F Direction Control Register</td>
</tr>
<tr>
<td>0x0070D8</td>
<td>GPGMUX</td>
<td>GPIO G Mux Control Register</td>
</tr>
<tr>
<td>0x0070D9</td>
<td>GPGDIR</td>
<td>GPIO G Direction Control Register</td>
</tr>
</tbody>
</table>
C28x GPIO Data Registers

<table>
<thead>
<tr>
<th>Address</th>
<th>Register</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0070E0</td>
<td>GPADAT</td>
<td>GPIO A Data Register</td>
</tr>
<tr>
<td>0x0070E1</td>
<td>GASET</td>
<td>GPIO A Set Register</td>
</tr>
<tr>
<td>0x0070E2</td>
<td>GACLEAR</td>
<td>GPIO A Clear Register</td>
</tr>
<tr>
<td>0x0070E3</td>
<td>GPAToggle</td>
<td>GPIO A Toggle Register</td>
</tr>
<tr>
<td>0x0070E4</td>
<td>GPBDAT</td>
<td>GPIO B Data Register</td>
</tr>
<tr>
<td>0x0070E5</td>
<td>GPBSET</td>
<td>GPIO B Set Register</td>
</tr>
<tr>
<td>0x0070E6</td>
<td>GPBCLEAR</td>
<td>GPIO B Clear Register</td>
</tr>
<tr>
<td>0x0070E7</td>
<td>GPBTOGGLE</td>
<td>GPIO B Toggle Register</td>
</tr>
<tr>
<td>0x0070EC</td>
<td>GPDDAT</td>
<td>GPIO D Data Register</td>
</tr>
<tr>
<td>0x0070ED</td>
<td>GPDSET</td>
<td>GPIO D Set Register</td>
</tr>
<tr>
<td>0x0070EE</td>
<td>GPDCLEAR</td>
<td>GPIO D Clear Register</td>
</tr>
<tr>
<td>0x0070EF</td>
<td>GPDTOGGLE</td>
<td>GPIO D Toggle Register</td>
</tr>
<tr>
<td>0x0070F0</td>
<td>GPEDAT</td>
<td>GPIO E Data Register</td>
</tr>
<tr>
<td>0x0070F1</td>
<td>GPESET</td>
<td>GPIO E Set Register</td>
</tr>
<tr>
<td>0x0070F2</td>
<td>GPECLEAR</td>
<td>GPIO E Clear Register</td>
</tr>
<tr>
<td>0x0070F3</td>
<td>GPETOGGLE</td>
<td>GPIO E Toggle Register</td>
</tr>
<tr>
<td>0x0070F4</td>
<td>GFDAT</td>
<td>GPIO F Data Register</td>
</tr>
<tr>
<td>0x0070F5</td>
<td>GFSET</td>
<td>GPIO F Set Register</td>
</tr>
<tr>
<td>0x0070F6</td>
<td>GFSCLEAR</td>
<td>GPIO F Clear Register</td>
</tr>
<tr>
<td>0x0070F7</td>
<td>GFPTOGGLE</td>
<td>GPIO F Toggle Register</td>
</tr>
<tr>
<td>0x0070F8</td>
<td>GPEDAT</td>
<td>GPIO G Data Register</td>
</tr>
<tr>
<td>0x0070F9</td>
<td>GPSET</td>
<td>GPIO G Set Register</td>
</tr>
<tr>
<td>0x0070FA</td>
<td>GPCLEAR</td>
<td>GPIO G Clear Register</td>
</tr>
<tr>
<td>0x0070FB</td>
<td>GPETOGGLE</td>
<td>GPIO G Toggle Register</td>
</tr>
</tbody>
</table>
**EALLOW Protected Registers**

**Register Name** | **Address Range** | **size (x16)**
---|---|---
Device Emulation | 0x00 0B80 – 0x00 09FF | 384
FLASH (also protected by CSM) | 0x00 0A00 – 0x00 0ADF | 96
Code Security Module | 0x00 0AE0 – 0x00 0AEF | 16
PIE Vector Table | 0x00 0D00 – 0x00 0DFF | 256
eCAN | 0x00 6000 – 0x00 60FF | 256 (128x32)
System Control | 0x00 7010 – 0x00 702F | 32
GPIO Mux | 0x00 70C0 – 0x00 70DF | 32

while(1) // dummy loop - wait for an interrupt
{
  asm("EALLOW"); // enable EALLOW protected register access
  SysCtrlRegs.WDKEY=0x55; // watchdog enabled for reset on next 0xAA write
  asm("EDIS"); // disable EALLOW protected register access
}

*Note: “SysCtrlRegs.WDKEY=0xAA” is located in an interrupt service routine*

---

**Lab 5: Procedure - System Initialization**

- LAB5 files have been provided as a starting point
- Modify LAB5 files to:
  - **Part 1**
    - Disable the watchdog – clear WD flag, disable watchdog, WD prescale = 1
    - Setup the clock module – PLL = x5, HISPCP = /1, LOSPCP = /4, low-power modes to default values, enable all module clocks
    - Setup control register – DO NOT clear WD OVERRIDE bit, WD generate a DSP reset
    - Setup shared I/O pins – set all GPIO pins to GPIO function (e.g. a "0" setting for GPIO function, and a "1" setting for peripheral function)
  - **Part 2**
    - Initialize peripheral interrupt expansion (PIE) vectors
- Build, debug, and test your code using Code Composer Studio
Lab 5: System Initialization

Objective

The objective of this lab is to perform the processor system initialization by applying the techniques discussed in module 5. Additionally, the peripheral interrupt expansion (PIE) vectors will be initialized and tested using the information discussed in the previous module. This initialization process will be used again in the module 6 analog-to-digital converter lab, and the module 7 event manager lab. The system initialization for this lab will consist of the following:

- Disable the watchdog – clear WD flag, disable watchdog, WD prescale = 1
- Setup the clock module – PLL = x5, HISPCP = /1, LOSPCP = /4, low-power modes to default values, enable all module clocks
- Setup control register – DO NOT clear WD OVERRIDE bit, WD generate a DSP reset
- Setup shared I/O pins – set all GPIO pins to GPIO function (e.g. a "0" setting for GPIO function, and a "1" setting for peripheral function.)

The first part of the lab exercise will setup the system initialization and test the watchdog operation by having the watchdog cause a reset. In the second part of the lab exercise the PIE vectors will be added and tested by using the watchdog to generate an interrupt. This lab will make use of the DSP281x C-code header files to simplify the programming of the device, as well as take care of the register definitions and addresses. Please review these files, and make use of them in the future, as needed.

Procedure

Create Project File

Note: LAB 5 files have been provided as a starting point for the lab and need to be completed. DO NOT copy files from a previous lab.

1. Create a new project called Lab5.pjt in C:\C28x\LABS\LAB5 and add the following files to it:

   Main_5.c
   Lab.cfg.cmd
   Lab.cdb
   User_5_6_7.cmd
   SysCtrl1.c
   Gpio.c
   DSP281x_GlobalVariableDefs.c

   Note that include files, such as DSP281x_Device.h and Lab.h, are automatically added at project build time.
Lab 5: System Initialization

---

**Project Build Options**

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

   Project → Build Options...

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

..\DSP281x_headers\include

This is the path for the header files. Then select OK to save the Build Options.

---

**Modify Memory Configuration**

3. Open and inspect the user linker command file User_5_6_7.cmd. Notice that the section “codestart” is being linked to a memory block named BEGIN_H0. The codestart section contains code that branches to the code entry point of the project. The bootloader must branch to the codestart section at the end of the boot process. Recall that the "Boot to H0" bootloader mode branches to address 0x3F8000 upon bootloader completion.

Modify the configuration file lab.cdb to create a new memory block named BEGIN_H0: base = 0x3F8000, length = 0x0002, space = code. Uncheck the “create a heap in memory” box. You will also need to modify the existing memory block H0SARAM to avoid any overlaps with this new memory block.

---

**Setup System Initialization**

4. Modify SysCtrl.c to implement the system initialization as described in the objective for this lab.

5. Open and inspect Gpio.c. Notice that the shared I/O pins have been set to the GPIO function. (Note: In Main_5.c do not edit the “main loop” section. This section will be used to test the watchdog operation.) Save your work.

---

**Build and Load**

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

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

8. Under Debug on the menu bar click “Go Main”. You should now be at the start of Main().
Run the Code – Watchdog Reset

9. Place the cursor on the first line of code in main() and set a breakpoint by right clicking the mouse key and select Toggle breakpoint. Notice that line is highlighted with a red dot indicating that the breakpoint has been set.

10. Single-step your code into the “main loop” section and watch the lines of code execute. If you don’t want to see each line execute, place the cursor in the “main loop” section (on the asm(“ NOP”); instruction line) and right click the mouse key and select Run To Cursor. This is the same as setting a breakpoint on the selected line, running to that breakpoint, and then removing the breakpoint.

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

12. Modify the InitSysCtrl() function to enable the watchdog (WDCR). This will enable the watchdog to function and cause a reset. Save the file and click the “Build” button. Then reset the DSP by clicking on Debug → Reset CPU. Under Debug on the menu bar click “Go Main”.

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

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

Setup PIE Vector for Watchdog Interrupt

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

15. Add the following files to the project:

   PieCtrl_5_6_7_8_9.c  
   DefaultIsr_5_6_7.c

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

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

   InitPieCtrl();

17. Using the “PIE Interrupt Assignment Table” shown in the previous module find the location for the watchdog interrupt, “WAKEINT”.

C28x - System Initialization
18. Modify `main()` to do the following:
   enable the "WAKEINT" interrupt in the PIE (Hint: use the `PieCtrlRegs` structure)
   enable core INT1 (IER register)
   enable global interrupts (INTM bit)

19. In `SysCtrl.c` modify the system control and status register (SCSR) to cause the
    watchdog to generate a WAKEINT rather than a reset.

20. Open and inspect `DefaultIsr_5_6_7.c`. This file contains interrupt service
    routines. The ISR for WAKEINT has been trapped by an emulation breakpoint contained
    in an inline assembly statement using “ESTOP0”. This gives the same results as placing
    a breakpoint in the ISR. We will run the lab exercise as before, except this time the
    watchdog will generate an interrupt. If the registers have been configured properly, the
    code will be trapped in the ISR.

21. Modify the configuration file `Lab.cdb` to setup the PIE vector for the watchdog
    interrupt. Click on the plus sign (+) to the left of `Scheduling` and again on the plus
    sign (+) to the left of `HWI – Hardware Interrupt Service Routine Manager`. Click the plus sign (+) to the left of `PIE INTERRUPTS`. Locate the
    interrupt location for the watchdog. Right click, select `Properties`, and type
    `_WAKEINT_ISR` (with a leading underscore) in the function field. Click `OK` and save
    all updates.

**Build and Load**

22. Save all changes to the files and click the “Build” button. Then reset the DSP, and
    then “Go Main”.

**Run the Code – Watchdog Interrupt**

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

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

**End of Exercise**

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

Introduction

This module explains the operation of the analog-to-digital converter. The system consists of a 12-bit analog-to-digital converter with 16 analog input channels. The analog input channels have a range from 0 to 3 volts. Two input analog multiplexers are used, each supporting 8 analog input channels. Each multiplexer has its own dedicated sample and hold circuit. Therefore, sequential, as well as simultaneous sampling is supported. Also, the ADC system features programmable auto sequence conversions with 16 results registers. Start of conversion (SOC) can be performed by an external trigger, software, or an Event Manager event.

Learning Objectives

- Understand the operation of Analog-to-Digital converter
- Show how to use the Analog-to-Digital converter to capture data
Module Topics

Analog-to-Digital Converter .............................................................................................................. 6-1

Module Topics .............................................................................................................................. 6-2

Analog-to-Digital Converter ........................................................................................................ 6-3
- Analog-to-Digital Converter Registers .................................................................................. 6-5
- Example – Sequencer “Start/Stop” Operation ......................................................................... 6-10
- ADC Conversion Result Buffer Register ............................................................................... 6-11
- Numerical Format .................................................................................................................... 6-12

Lab 6: Analog-to-Digital Converter ............................................................................................ 6-13
Analog-to-Digital Converter

ADC Module Block Diagram (Cascaded Mode)

ADC Module Block Diagram (Dual-Sequencer mode)
Analog-to-Digital Converter

ADC Module

- 12-bit resolution ADC core
- Sixteen analog inputs (range of 0 to 3V)
- Two analog input multiplexers
  - Up to 8 analog input channels each
- Two sample/hold units (for each input mux)
- Sequential and simultaneous sampling modes
- 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
  - External trigger, S/W, and Event Manager events

F2812 ADC Clocking Example

Important: ADCCLK can be a maximum of 25 MHz!
# Analog-to-Digital Converter Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADCTRL1</td>
<td>0x007100</td>
<td>ADC Control Register 1</td>
</tr>
<tr>
<td>ADCTRL2</td>
<td>0x007101</td>
<td>ADC Control Register 2</td>
</tr>
<tr>
<td>ADCMAXCONV</td>
<td>0x007102</td>
<td>ADC Maximum Conversion Channels Register</td>
</tr>
<tr>
<td>ADCHSELSEQ1</td>
<td>0x007103</td>
<td>ADC Channel Select Sequencing Control Register 1</td>
</tr>
<tr>
<td>ADCHSELSEQ2</td>
<td>0x007104</td>
<td>ADC Channel Select Sequencing Control Register 2</td>
</tr>
<tr>
<td>ADCHSELSEQ3</td>
<td>0x007105</td>
<td>ADC Channel Select Sequencing Control Register 3</td>
</tr>
<tr>
<td>ADCHSELSEQ4</td>
<td>0x007106</td>
<td>ADC Channel Select Sequencing Control Register 4</td>
</tr>
<tr>
<td>ADCASEQSR</td>
<td>0x007107</td>
<td>ADC Autosequence Status Register</td>
</tr>
<tr>
<td>ADCRESULT0</td>
<td>0x007108</td>
<td>ADC Conversion Result Buffer Register 0</td>
</tr>
<tr>
<td>ADCRESULT1</td>
<td>0x007109</td>
<td>ADC Conversion Result Buffer Register 1</td>
</tr>
<tr>
<td>ADCRESULT2</td>
<td>0x00710A</td>
<td>ADC Conversion Result Buffer Register 2</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>ADCRESULT14</td>
<td>0x007116</td>
<td>ADC Conversion Result Buffer Register 14</td>
</tr>
<tr>
<td>ADCRESULT15</td>
<td>0x007117</td>
<td>ADC Conversion Result Buffer Register 15</td>
</tr>
<tr>
<td>ADCTRL3</td>
<td>0x007118</td>
<td>ADC Control Register 3</td>
</tr>
<tr>
<td>ADCST</td>
<td>0x007119</td>
<td>ADC Status and Flag Register</td>
</tr>
</tbody>
</table>

The table lists the registers available in the Analog-to-Digital Converter and their respective addresses and descriptions.
ADC Control Register 1 - Upper Byte
ADCTRL1 @ 0x007100 (lab file: AdcAdc.c)

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

Acquisition Time Prescale (S/H)
Value = (binary+1)
* Time dependent on the “Conversion Clock Prescale” bit (Bit 7 “CPS”)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>reserved</td>
</tr>
<tr>
<td>14</td>
<td>RESET</td>
</tr>
<tr>
<td>13</td>
<td>SUSMOD1</td>
</tr>
<tr>
<td>12</td>
<td>SUSMOD0</td>
</tr>
<tr>
<td>11</td>
<td>ACQ_PS3</td>
</tr>
<tr>
<td>10</td>
<td>ACQ_PS2</td>
</tr>
<tr>
<td>9</td>
<td>ACQ_PS1</td>
</tr>
<tr>
<td>8</td>
<td>ACQ_PS0</td>
</tr>
</tbody>
</table>

Emulation Suspend Mode
00 = [Mode 0] free run (do not stop)
01 = [Mode 1] stop after current sequence
10 = [Mode 2] stop after current conversion
11 = [Mode 3] stop immediately

ADC Control Register 1 - Lower Byte
ADCTRL1 @ 0x007100 (lab file: AdcAdc.c)

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

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>CPS</td>
</tr>
<tr>
<td>6</td>
<td>CONT_RUN</td>
</tr>
<tr>
<td>5</td>
<td>SEQ1_OVRD</td>
</tr>
<tr>
<td>4</td>
<td>SEQ_CASC</td>
</tr>
<tr>
<td>3</td>
<td>reserved</td>
</tr>
<tr>
<td>2</td>
<td>reserved</td>
</tr>
<tr>
<td>1</td>
<td>reserved</td>
</tr>
<tr>
<td>0</td>
<td>reserved</td>
</tr>
</tbody>
</table>

Conversion Prescale
0 = CLK / 1
1 = CLK / 2

Sequencer Override (continuous run mode)
0 = sequencer pointer resets to “initial state” at end of MAX_CONVn
1 = sequencer pointer resets to “initial state” after “end state”
**ADC Control Register 2 - Upper Byte**

ADCTRL2 @ 0x007101 (lab file: AdcAdc.c)

<table>
<thead>
<tr>
<th>Bit 15</th>
<th>Bit 14</th>
<th>Bit 13</th>
<th>Bit 12</th>
<th>Bit 11</th>
<th>Bit 10</th>
<th>Bit 9</th>
<th>Bit 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVB_SOC_SEQ</td>
<td>RST_SEQ1</td>
<td>SOC_SEQ1</td>
<td>reserved</td>
<td>INT_ENA_SEQ1</td>
<td>INT_MOD_SEQ1</td>
<td>reserved</td>
<td>EVA_SOC_SEQ1</td>
</tr>
</tbody>
</table>

- **EVB SOC**
  - (cascaded mode only)
  - 0 = no action
  - 1 = start by EVB signal

- **Start Conversion (SEQ1)**
  - 0 = clear pending SOC trigger
  - 1 = software trigger-start SEQ1

- **Reset SEQ1**
  - 0 = no action
  - 1 = immediate reset

- **Interrupt Enable (SEQ1)**
  - 0 = interrupt disable
  - 1 = interrupt enable

- **Interrupt Mode (SEQ1)**
  - 0 = interrupt every EOS
  - 1 = interrupt every other EOS

**ADC Control Register 2 - Lower Byte**

ADCTRL2 @ 0x007101 (lab file: AdcAdc.c)

<table>
<thead>
<tr>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXT_SOC_SEQ1</td>
<td>RST_SEQ2</td>
<td>SOC_SEQ2</td>
<td>reserved</td>
<td>INT_ENA_SEQ2</td>
<td>INT_MOD_SEQ2</td>
<td>reserved</td>
<td>EVB_SOC_SEQ2</td>
</tr>
</tbody>
</table>

- **External SOC (SEQ1)**
  - 0 = no action
  - 1 = start by signal from ADCSOC pin

- **Start Conversion (SEQ2)**
  - (dual-sequencer mode only)
  - 0 = clear pending SOC trigger
  - 1 = software trigger-start SEQ2

- **Reset SEQ2**
  - 0 = no action
  - 1 = immediate reset

- **Interrupt Enable (SEQ2)**
  - 0 = interrupt disable
  - 1 = interrupt enable

- **Interrupt Mode (SEQ2)**
  - 0 = interrupt every EOS
  - 1 = interrupt every other EOS
### ADC Control Register 3

**ADCTRL3** @ 0x007118  
(lab file: AdcAdc.c)

- **ADC Reference Power Down**
  - 0 = powered down
  - 1 = powered up

- **ADC Bandgap Power Down**
  - 0 = powered down
  - 1 = powered up

- **ADC Power Down (except Bandgap & Ref.)**
  - 0 = powered down
  - 1 = powered up

- **ADC Clock Prescale**
  - BD0 = 0x0
  - BD1 = 0x0
  - BD2 = 0x0
  - BD3 = 0x0

- **SMODE_SEL**
  - 0 = sequential sampling mode
  - 1 = simultaneous sampling mode

- **ADCRFDN**
  - 0 = powered down
  - 1 = powered up

- **ADCBGND**
  - 0 = powered down
  - 1 = powered up

- **ADCPWDN**
  - 0 = powered down
  - 1 = powered up

### Maximum Conversion Channels Register

**ADCMAXCONV** @ 0x007102  
(lab file: AdcAdc.c)

- **Bit fields define the maximum number of autoconversions (binary+1)**

- **Cascaded Mode**
  - Initial state: CONV00
  - End state: CONV15

- **Dual Mode**
  - Initial state: CONV00
  - End state: CONV08

- **SEQ1**
  - Initial state: CONV07
  - End state: CONV15

- **SEQ2**
  - Initial state: CONV00
  - End state: CONV08

- **Autoconversion session always starts with the “initial state” and continues sequentially until the “end state”, if allowed**

---

**6 - 8**  
C28x - Analog-to-Digital Converter
### ADC Input Channel Select Sequencing Control Register

(lab file: Adc.c)

<table>
<thead>
<tr>
<th>Bits 15-12</th>
<th>Bits 11-8</th>
<th>Bits 7-4</th>
<th>Bits 3-0</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONV03</td>
<td>CONV02</td>
<td>CONV01</td>
<td>CONV00</td>
<td>ADCCHSELSEQ1</td>
</tr>
<tr>
<td>CONV07</td>
<td>CONV06</td>
<td>CONV05</td>
<td>CONV04</td>
<td>ADCCHSELSEQ2</td>
</tr>
<tr>
<td>CONV11</td>
<td>CONV10</td>
<td>CONV09</td>
<td>CONV08</td>
<td>ADCCHSELSEQ3</td>
</tr>
<tr>
<td>CONV15</td>
<td>CONV14</td>
<td>CONV13</td>
<td>CONV12</td>
<td>ADCCHSELSEQ4</td>
</tr>
</tbody>
</table>
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 with sequential sampling mode

Example - Sequencer “Start/Stop” Operation

Bits → 15-12 11-8 7-4 3-0
0x007103 \(V_1, I_3, I_2, I_1\) ADCCHSELSEQ1
0x007104 \(x, x, V_3, V_2\) ADCCHSELSEQ2

• 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 auto conversion 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 |

→ User can reset SEQ1 by software to state CONV00 and repeat same trigger 1, 2 session
• SEQ1 keeps “waiting” at current state for another trigger
ADC Conversion Result Buffer Register

<table>
<thead>
<tr>
<th>analog volts</th>
<th>converted value</th>
<th>RESULTx</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.0</td>
<td>FFFh</td>
<td>111111110000</td>
</tr>
<tr>
<td>1.5</td>
<td>7FFh</td>
<td>011111110000</td>
</tr>
<tr>
<td>0.00073</td>
<td>1h</td>
<td>0000000000010000</td>
</tr>
<tr>
<td>0</td>
<td>0h</td>
<td>0000000000000000</td>
</tr>
</tbody>
</table>
Numerical Format

How do we Read the Result? Integer format

\[
\begin{array}{c}
\text{RESULT}_x \\
\text{ACC} \\
\text{Data Mem}
\end{array}
\]

15 bit shift right \[0\]

Example: read RESULT0 register

```
#include "DSP281x_Device.h"

void main(void)
{
    Uint16 value; // unsigned
    value = AdcRegs.ADCRESULT0 >> 4;
}
```

What About Signed Input Voltages?

Example: -1.5 V \(\leq V_{in} \leq +1.5 \text{ V}\)

1) Add 1.5 volts to analog input:

2) Subtract “1.5” from digital result:

```
#include "DSP281x_Device.h"
#define offset 0x07FF

void main(void)
{
    int16 value; // signed
    value = (AdcRegs.ADCRESULT0 >> 4) - offset;
}
```
Lab 6: Analog-to-Digital Converter

Objective

The objective of this lab is to apply the techniques discussed in module 6 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 (GPIOA1) high and low in the ADC interrupt service routine. The ADC ISR will also toggle LED DS2 on the eZdsp™ as a visual indication that the ISR is running. This pin will be connected to the ADC input pin, and sampled. After taking some data, Code Composer Studio will be used to plot the results. A flow chart of the code is shown in the following slide.

![Lab 6: Code Flow Diagram](image)

**Notes**
- Program performs conversion on ADC channel A0 (ADCINA0 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
- GPIOA1 pin is also toggled in the ADC ISR
- ADC ISR will also toggle the eZdsp™ LED DS2 as a visual indication that it is running

➢ Procedure

**Project File**

Note: LAB6 files have been provided as a starting point for the lab and need to be completed. DO NOT copy files from a previous lab.

1. A project named Lab6.pjt has been created for this lab. Open the project by clicking on Project ➔ Open... and look in C:\C28x\LABS\LAB6. All Build Options have been configured like the previous lab. The files used in this lab are:
Setup ADC Initialization and Enable Core/PIE Interrupts

2. In Main_6.c add code to call InitAdc() and InitEv() functions. The InitEv() function is used to configure Timer 2 to trigger the ADC at a 50 kHz rate. Details about the event manager will be discussed in the next module.

3. Edit Adc.c to implement the ADC initialization as described above in the objective for the lab by configuring the following registers: ADCTRL1, ADCTRL2, ADCMAXCONV and ADCCHSELSEQ1. (Set ADC for cascaded sequencer mode, CPS = CLK/1, and acquisition time prescale = 8 * (1/ADCCLK).

4. Using the “PIE Interrupt Assignment Table” find the location for the ADC interrupt, “ADCINT” and fill in the following information:

   PIE group #: ___________ # within group: ______

   This information will be used in the next step.

5. Modify the end of Adc.c to do the following:
   enable the "ADCINT" interrupt in the PIE (Hint: use the PieCtrlRegs structure)
   enable core INT1 (IER register)

6. Open and inspect DefaultIsr_5_6_7.c. This file contains the ADC interrupt service routine.

7. Modify the configuration file lab.cdb to setup the PIE vector for the ADC interrupt. Click on the plus sign (+) to the left of Scheduling and again on the plus sign (+) to the left of HWI - Hardware Interrupt Service Routine Manager. Click the plus sign (+) to the left of PIE INTERRUPTS. Locate the interrupt location for the ADC (use the information from step #4). Right click, select Properties, and type _ADCINT_ISR (with a leading underscore) in the function field. Click OK and save all updates.

Build and Load

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

9. Reset the DSP, and then “Go Main”.

Main_6.c Labcfg.cmd
Lab.cdb DSP281x_Headers_BIOS.cmd
User_5_6_7.cmd CodeStartBranch.asm
SysCtrl.c Gpio.c
DSP281x_GlobalVariableDefs.c PieCtrl_5_6_7_8_9.c
DefaultIsr_5_6_7.c Adc.c
Ev_6.c

C28x - Analog-to-Digital Converter
Run the Code

10. In Main_6.c place the cursor in the “main loop” section, right click on the mouse key and select Run To Cursor.

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

**Note:** For the next set of steps “VREFLO” must be connected to “GND”. Using a connector wire provided, connect “VREFLO” (pin # P9-18) to “GND” (pin # P9-17). **Exercise care when connecting any wires, as the power to the eZdsp™ is on, and we do not want to damage the eZdsp™!** Details of pin assignments can be found in Appendix A.

12. Using another connector wire provided, connect the ADCINA0 (pin # P9-2) to “GND” (pin # P9-1) on the eZdsp™. Then run the code again, and halt it after a few seconds. Verify that the ADC results buffer contains the expected value of 0x0000.

13. Adjust the connector wire to connect the ADCINA0 (pin # P9-2) to “+3.3V” (pin # P2-45) on the eZdsp™. Then run the code again, and halt it after a few seconds. Verify that the ADC results buffer contains the expected value of 0x0FFF.

14. Adjust the connector wire to connect the ADCINA0 (pin # P9-2) to IOPA1 (pin # P8-10) on the eZdsp™. Then run the code again, and halt it after a few seconds. Examine the contents of the ADC results buffer (the contents should be alternating 0x0000 and 0x0FFF values). Are the contents what you expected?

15. 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:
   
<table>
<thead>
<tr>
<th>Start Address</th>
<th>AdcBuf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acquisition Buffer Size</td>
<td>50</td>
</tr>
<tr>
<td>Display Data Size</td>
<td>50</td>
</tr>
<tr>
<td>DSP Data Type</td>
<td>16-bit unsigned integer</td>
</tr>
<tr>
<td>Sampling Rate (Hz)</td>
<td>50000</td>
</tr>
<tr>
<td>Time Display Unit</td>
<td>μs</td>
</tr>
</tbody>
</table>

   Select OK to save the graph options.

16. Recall that the code toggled the IOPA1 pin alternately high and low. (Also, the ADC ISR is toggling the LED DS2 on the eZdsp™ as a visual indication that the ISR is running). If you had an oscilloscope available to display IOPA1, you would expect to see a square-wave. Why does Code Composer Studio plot resemble a saw-tooth wave? What is the signal processing term for what is happening here?
17. Recall that the program toggled the IOPA1 pin at a 50 kHz rate. Therefore, a complete cycle (toggle high, then toggle low) occurs at half this rate, or 25 kHz. We therefore expect the period of the waveform to be 40 µs. Confirm this by measuring the period of the 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.

**Using Real-time Emulation**

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

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

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

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

18. Reset the DSP, and enable real-time mode by selecting:

   Debug → Real-time Mode

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

20. The memory and graph windows displaying `AdcBuf` should still be open. The connector wire between ADCINA0 (pin # P9-2) and IOPA1 (pin # P8-10) should still be connected. In real-time mode, we would like to have our window continuously refresh. Click:

   View → Real-time Refresh Options...

   and check “Global Continuous Refresh”. Alternately, we could have right clicked on each window individually and selected “Continuous Refresh”.

Note: “Global Continuous Refresh” causes all open windows to refresh at the refresh rate. This can be problematic when a large number of windows are open, as bandwidth over the emulation link is limited. Updating too many windows can cause the refresh frequency to bog down. In that case, either close some windows, or disable global refresh and selectively enable “Continuous Refresh” for individual windows of interest instead.
21. Run the code and watch the windows update in real-time mode. Are the values updating as expected?

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

23. So far, we have seen data flowing from the DSP to the debugger in realtime. In this step, we will flow data from the debugger to the DSP.

- Open and inspect DefaultIsr_5_6_7.c. Notice that the global variable DEBUG_TOGGLE is used to control the toggling of the GPIOA1 pin. This is the pin being read with the ADC.
- Highlight DEBUG_TOGGLE with the mouse, right click and select “Add to Watch Window”. The global variable DEBUG_TOGGLE should now be in the watch window with a value of “1”.
- Run the code in real-time mode and change the value to “0”. Are the results shown in the memory and graph window as expected? Change the value back to “1”. As you can see, we are modifying data memory contents while the processor is running in real-time (i.e., we are not halting the DSP nor interfering with its operation in any way)!

End of Exercise
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 C28x family 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

- **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. For simplicity, only EVA will be explained.
# Module Topics

<table>
<thead>
<tr>
<th>Event Manager</th>
<th>7-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module Topics</td>
<td>7-2</td>
</tr>
<tr>
<td>Event Manager</td>
<td>7-3</td>
</tr>
<tr>
<td>PWM Review</td>
<td>7-4</td>
</tr>
<tr>
<td><strong>General-Purpose Timers</strong></td>
<td>7-8</td>
</tr>
<tr>
<td>GP Timer Modes of Operation</td>
<td>7-10</td>
</tr>
<tr>
<td>GP Timer Registers</td>
<td>7-12</td>
</tr>
<tr>
<td>Asymmetric and Symmetric PWM via General Purpose Timer Compares</td>
<td>7-16</td>
</tr>
<tr>
<td>GP Timer Compare PWM Exercise</td>
<td>7-17</td>
</tr>
<tr>
<td><strong>Compare Units</strong></td>
<td>7-18</td>
</tr>
<tr>
<td>Compare Unit Registers</td>
<td>7-19</td>
</tr>
<tr>
<td>Hardware Dead-Band (Compare Units only)</td>
<td>7-22</td>
</tr>
<tr>
<td>Power Drive Protection</td>
<td>7-25</td>
</tr>
<tr>
<td><strong>Capture Units</strong></td>
<td>7-26</td>
</tr>
<tr>
<td>Capture Units Registers</td>
<td>7-29</td>
</tr>
<tr>
<td><strong>Quadrature Encoder Pulse (QEP)</strong></td>
<td>7-32</td>
</tr>
<tr>
<td>QEP Initialization with GP Timer 2 (EVA)</td>
<td>7-34</td>
</tr>
<tr>
<td><strong>Lab 7: Event Manager</strong></td>
<td>7-36</td>
</tr>
<tr>
<td><strong>GP Timer Compare PWM Exercise Solution</strong></td>
<td>7-42</td>
</tr>
</tbody>
</table>
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 ≈ original signal energy
- Differs from PAM (Pulse Amplitude Modulation)
  - 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 Review

---

**PWM Signal Representation**

![Diagram of PWM signal representation](image)

**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 ⇒ easy for DSP to output

---

**C28x - Event Manager**
Asymmetric PWM Waveform

PWM Review

Asymmetric PWM Waveform

Counter

Period

Compare

\( T_{\text{PWM}} \)

\( T_{\text{pwmp}} / T_{\text{cmp}} \) Pin

(active high)

Caused by Period match
(toggle output in Asym mode only)

Caused by Compare match

C28x - Event Manager
Each C28x event manager (EVA and EVB) is 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 PWM signals 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 C28x 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.
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 when the timer counter matches the value in the timer period register.

2. **TxUFINT** - underflow interrupt. Flag is set when the timer counter becomes zero.

3. **TxOFINT** - overflow interrupt. Flag is set when the timer counter matches 0xFFFFh.

**Note:** Maximum frequency for External or QEP is CLKin/4
GP Timer Modes of Operation

The C28x 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 seemlessly
Continuous-Up/Down Counting

Continuous-Up/Down Counting Mode
(Used for Symmetric PWM Waveforms)

This example:
TxCON.3-2 = 01 (reload TxCMPR on underflow or period match)
TxPR = 3
TxCMPR = 1 (initially)
Prescale = 1

♦♦ Seemless up/down repetition
♦ Up/down count period is \(2 \times \text{TxPR}\)

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

Generated 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 TxCMPR 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

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPTCONA</td>
<td>0x007400</td>
<td>General Purpose Timer Control Register A</td>
</tr>
<tr>
<td>T1CNT</td>
<td>0x007401</td>
<td>Timer 1 Counter Register</td>
</tr>
<tr>
<td>T1CMPR</td>
<td>0x007402</td>
<td>Timer 1 Compare Register Buffer</td>
</tr>
<tr>
<td>T1PR</td>
<td>0x007403</td>
<td>Timer 1 Period Register Buffer</td>
</tr>
<tr>
<td>T1CON</td>
<td>0x007404</td>
<td>Timer 1 Control Register</td>
</tr>
<tr>
<td>T2CNT</td>
<td>0x007405</td>
<td>Timer 2 Counter Register</td>
</tr>
<tr>
<td>T2CMPR</td>
<td>0x007406</td>
<td>Timer 2 Compare Register Buffer</td>
</tr>
<tr>
<td>T2PR</td>
<td>0x007407</td>
<td>Timer 2 Period Register Buffer</td>
</tr>
<tr>
<td>T2CON</td>
<td>0x007408</td>
<td>Timer 2 Control Register</td>
</tr>
<tr>
<td>GPTCONB</td>
<td>0x007500</td>
<td>General Purpose Timer Control Register B</td>
</tr>
<tr>
<td>T3CNT</td>
<td>0x007501</td>
<td>Timer 3 Counter Register</td>
</tr>
<tr>
<td>T3CMPR</td>
<td>0x007502</td>
<td>Timer 3 Compare Register Buffer</td>
</tr>
<tr>
<td>T3PR</td>
<td>0x007503</td>
<td>Timer 3 Period Register Buffer</td>
</tr>
<tr>
<td>T3CON</td>
<td>0x007504</td>
<td>Timer 3 Control Register</td>
</tr>
<tr>
<td>T4CNT</td>
<td>0x007505</td>
<td>Timer 4 Counter Register</td>
</tr>
<tr>
<td>T4CMPR</td>
<td>0x007506</td>
<td>Timer 4 Compare Register Buffer</td>
</tr>
<tr>
<td>T4PR</td>
<td>0x007507</td>
<td>Timer 4 Period Register Buffer</td>
</tr>
<tr>
<td>T4CON</td>
<td>0x007508</td>
<td>Timer 4 Control Register</td>
</tr>
</tbody>
</table>

### GP Timer Control Register A (EVA)

**Upper Byte:**

- **Timer 2 Compare Trip Enable**
  - T2CTRIPET
  - 0 = disable
  - 1 = enable

- **Timer 1 Compare Trip Enable**
  - T1CTRIPET
  - 0 = disable
  - 1 = enable

**GP Timer Status (read-only):**

- 0 = counting down
- 1 = counting up

**ADC start by event of GP Timer x**

- 00: no event starts ADC
- 01: setting of underflow interrupt flag
- 10: setting of period interrupt flag
- 11: setting of compare interrupt
### GP Timer Control Register A (EVA)

**GPTCONA @ 0x007400** (lab file: EvEv.c)

**Lower Byte:**

- **Timer 2 Compare Output Enable**
  - T2CMPOE (if EXTCONA[0]=1)
  - 0 = disable (hi-Z)
  - 1 = enable

- **Timer 1 Compare Output Enable**
  - T1CMPOE (if EXTCONA[0]=1)
  - 0 = disable (hi-Z)
  - 1 = enable

- **Compare Output Enable**
  - (reserved when EXTCONA[0]=1)
  - 0 = all disable (hi-Z)
  - 1 = all enable

- **TxPWM/TxCMP Output Pin Conditioning**
  - 00: forced low
  - 01: active low
  - 10: active high
  - 11: forced high

### Timer Control Register (EVA)

**T1CON @ 0x007404 / T2CON @ 0x007408** (lab file: EvEv.c)

**Upper Byte:**

- **Emulation Halt Behavior**
  - 00 = stop immediately
  - 01 = stop at end of period
  - 1x = free run (do not stop)

- **Count Mode Select**
  - 00 = stop/hold
  - 01 = continuous-up/down
  - 10 = continuous-up
  - 11 = directional-up/down

- **Timer Clock Prescale**
  - 000: + 1
  - 001: + 2
  - 010: + 4
  - 011: + 8
  - 100: + 16
  - 101: + 32
  - 110: + 64
  - 111: + 128
General-Purpose Timers

Timer Control Register (EVA)
T1CON @ 0x007404 / T2CON @ 0x007408 (lab file: EvEv.c)

Lower Byte:

- **Timer Enable**
  - 0 = timer disable
  - 1 = timer enable

- **Timer Clock Source**
  - 00 = internal (HSPCLK)
  - 01 = external TCLKIN pin
  - 10 = reserved
  - 11 = QEP

- **Timer Compare Operation Enable**
  - 0 = disable
  - 1 = enable

- **Period Register Select**
  - 0 = use own per. reg.
  - 1 = use Timer 1 per. reg

- **Start with Timer 1**
  - 0 = use own TENABLE
  - 1 = use Timer 1 TENABLE

- **Compare Register Reload Condition**
  - 00 = when counter equals zero (underflow)
  - 01 = when counter equals zero or period reg
  - 10 = immediately
  - 11 = reserved

Extension Control Register A (EVA)
EXTCONA @ 0x007409 (lab file: EvEv.c)

- **QEP Index Enable**
  - 0 = disable
  - 1 = enable

- **Independent Compare Output Enable Mode**
  - 0 = disable
  - 1 = enable

- **EV Start-of-Conversion Output Enable**
  - 0 = disable
  - 1 = enable

- **CAP3/QEPI Index Qualification Mode**
  - 0 = off
  - 1 = on
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. The value needed in the period register is:

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

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

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

PWM resolution:

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

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

Symmetric PWM: approx. 9 bit resolution since \(2^9 = 512 \approx 500\)

PWM duty cycle:

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

Asymmetric PWM: TxCMPR = (100\% - duty cycle) * TxPR

Symmetric PWM: TxCMPR = (100\% - duty cycle) * TxPR

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

Symmetric PWM is to be generated as follows:
- 50 kHz carrier frequency
- Timer counter clocked by 10 ns CPU clock
- Use the \( \div 1 \) prescale option
- 25% duty cycle initially
- Use GP Timer Compare 1 with PWM output active high
- T2PWM/T2CMP pins forced low

Determine the initialization values needed in the GPTCONA, T1CON, T1PR, and T1CMPR registers
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

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMCONA</td>
<td>0x007411</td>
<td>Compare Control Register A</td>
</tr>
<tr>
<td>ACTRA</td>
<td>0x007413</td>
<td>Compare Action Control Register A</td>
</tr>
<tr>
<td>DBTCONA</td>
<td>0x007415</td>
<td>Dead-Band Timer Control Register A</td>
</tr>
<tr>
<td>CMPR1</td>
<td>0x007417</td>
<td>Compare Register 1</td>
</tr>
<tr>
<td>CMPR2</td>
<td>0x007418</td>
<td>Compare Register 2</td>
</tr>
<tr>
<td>CMPR3</td>
<td>0x007419</td>
<td>Compare Register 3</td>
</tr>
<tr>
<td>COMCONB</td>
<td>0x007511</td>
<td>Compare Control Register B</td>
</tr>
<tr>
<td>ACTRB</td>
<td>0x007513</td>
<td>Compare Action Control Register B</td>
</tr>
<tr>
<td>DBTCONB</td>
<td>0x007515</td>
<td>Dead-Band Timer Control Register B</td>
</tr>
<tr>
<td>CMPR4</td>
<td>0x007517</td>
<td>Compare Register 4</td>
</tr>
<tr>
<td>CMPR5</td>
<td>0x007518</td>
<td>Compare Register 5</td>
</tr>
<tr>
<td>CMPR6</td>
<td>0x007519</td>
<td>Compare Register 6</td>
</tr>
<tr>
<td>EXTCONA</td>
<td>0x007409</td>
<td>Extension Control Register A</td>
</tr>
<tr>
<td>EXTCONB</td>
<td>0x007509</td>
<td>Extension Control Register B</td>
</tr>
</tbody>
</table>

### Compare Control Register (EVA)

**COMCONA @ 0x007411**

**Upper Byte:**

- **Compare Enable**: 0 = disable, 1 = enable
- **Space Vector PWM**: 0 = SV disable, 1 = SV enable
- **Full Compare Output Enable**: (reserved when EXTCONA[0]=1)
  - 0 = all disable (high-impedance)
  - 1 = all enable

```
<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>CENABLE</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>CLD1</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>CLD0</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>SVENABLE</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>ACTRLD1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>ACTRLD0</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>FCOMPOE</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>PDPINTA</td>
<td></td>
</tr>
</tbody>
</table>
```

### CMPRx reload condition

- 00 = when T1CNT = 0
- 01 = when T1CNT = 0 or T1PR
- 10 = immediately
- 11 = reserved

### ACTRA reload condition

- 00 = when T1CNT = 0
- 01 = when T1CNT = 0 or T1PR
- 10 = immediately
- 11 = reserved

### PDPINT Status

- 0 = low
- 1 = high
### Compare Control Register (EVA)

**COMCONA @ 0x007411** (lab file: EvEv.c)

#### Lower Byte:

<table>
<thead>
<tr>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESERVED</td>
<td>RESERVED</td>
<td>RESERVED</td>
<td>reserved</td>
<td>reserved</td>
<td>RESERVED</td>
<td>RESERVED</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- **Full Compare 2 Output Enable** (FCMP2OE)
  - (if EXTCONA[0]=1)
  - 0 = disable
  - 1 = enable
- **Full Compare 2 Trip Enable** (C2TRIPE)
  - (if EXTCONA[0]=1)
  - 0 = disable
  - 1 = enable

#### Upper Byte:

- **Full Compare 1 Output Enable** (FCMP1OE)
  - (if EXTCONA[0]=1)
  - 0 = disable
  - 1 = enable
- **Full Compare 1 Trip Enable** (C1TRIPE)
  - (if EXTCONA[0]=1)
  - 0 = disable
  - 1 = enable

### Extension Control Register A (EVA)

**EXTCONA @ 0x007409** (lab file: EvEv.c)

- **QEP Index Enable**
  - 0 = disable
  - 1 = enable
- **Independent Compare Output Enable Mode**
  - 0 = disable
  - 1 = enable
- **EV Start-of-Conversion Output Enable**
  - 0 = disable
  - 1 = enable
- **CAP3/ QEPI Index Qualification Mode**
  - 0 = off
  - 1 = on
### Compare Action Control Register (EVA)

**ACTRA @ 0x007413 (lab file: EvEv.c)**

#### Basic Space Vector Bits
- Can write as 0 when SV not in use

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>SVRDHR</td>
</tr>
<tr>
<td>14</td>
<td>D2</td>
</tr>
<tr>
<td>13</td>
<td>D1</td>
</tr>
<tr>
<td>12</td>
<td>D0</td>
</tr>
<tr>
<td>11</td>
<td>CMP6ACT1</td>
</tr>
<tr>
<td>10</td>
<td>CMP5ACT0</td>
</tr>
<tr>
<td>9</td>
<td>CMP5ACT1</td>
</tr>
<tr>
<td>8</td>
<td>CMP5ACT0</td>
</tr>
<tr>
<td>7</td>
<td>CMP4ACT1</td>
</tr>
<tr>
<td>6</td>
<td>CMP4ACT0</td>
</tr>
<tr>
<td>5</td>
<td>CMP3ACT1</td>
</tr>
<tr>
<td>4</td>
<td>CMP3ACT0</td>
</tr>
<tr>
<td>3</td>
<td>CMP2ACT1</td>
</tr>
<tr>
<td>2</td>
<td>CMP2ACT0</td>
</tr>
<tr>
<td>1</td>
<td>CMP1ACT1</td>
</tr>
<tr>
<td>0</td>
<td>CMP1ACT0</td>
</tr>
</tbody>
</table>

#### Pin Action on Compare: CMPyACT1-0
- 00: Force low
- 01: Active low
- 10: Active high
- 11: Forced high

#### SV Rotation Direction
- Can write as 0 when SV not in use
Hardware Dead-Band (Compare Units only)

Motivation for Dead-Band

Gate Signals are Complementary PWM

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

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

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

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

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).
Power Drive Protection

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

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
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 C28x 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
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)

Can latch on:
- rising edge
- falling edge
- both
Capture Units FIFO Operation

- CPU can read both upper and lower FIFO registers separately; Standard approach is to read upper, but lower can be read.

Capture Unit FIFO Operation Diagram

Capture Units Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAPCONA</td>
<td>0x007420</td>
<td>Capture Control Register A</td>
</tr>
<tr>
<td>CAPFIPOA</td>
<td>0x007422</td>
<td>Capture FIFO Status Register A</td>
</tr>
<tr>
<td>CAP1FIFO</td>
<td>0x007423</td>
<td>Two-Level Deep FIFO 1 Stack</td>
</tr>
<tr>
<td>CAP2FIFO</td>
<td>0x007424</td>
<td>Two-Level Deep FIFO 2 Stack</td>
</tr>
<tr>
<td>CAP3FIFO</td>
<td>0x007425</td>
<td>Two-Level Deep FIFO 3 Stack</td>
</tr>
<tr>
<td>CAP1FBOT</td>
<td>0x007427</td>
<td>Bottom Register of FIFO 1</td>
</tr>
<tr>
<td>CAP2FBOT</td>
<td>0x007428</td>
<td>Bottom Register of FIFO 2</td>
</tr>
<tr>
<td>CAP3FBOT</td>
<td>0x007429</td>
<td>Bottom Register of FIFO 3</td>
</tr>
<tr>
<td>CAPCONB</td>
<td>0x007520</td>
<td>Capture Control Register B</td>
</tr>
<tr>
<td>CAPFIFOB</td>
<td>0x007522</td>
<td>Capture FIFO Status Register B</td>
</tr>
<tr>
<td>CAP4FIFO</td>
<td>0x007523</td>
<td>Two-Level Deep FIFO 4 Stack</td>
</tr>
<tr>
<td>CAP5FIFO</td>
<td>0x007524</td>
<td>Two-Level Deep FIFO 5 Stack</td>
</tr>
<tr>
<td>CAP6FIFO</td>
<td>0x007525</td>
<td>Two-Level Deep FIFO 6 Stack</td>
</tr>
<tr>
<td>CAP4FBOT</td>
<td>0x007527</td>
<td>Bottom Register of FIFO 4</td>
</tr>
<tr>
<td>CAP5FBOT</td>
<td>0x007528</td>
<td>Bottom Register of FIFO 5</td>
</tr>
<tr>
<td>CAP6FBOT</td>
<td>0x007529</td>
<td>Bottom Register of FIFO 6</td>
</tr>
<tr>
<td>EXTCONA</td>
<td>0x007409</td>
<td>Ext. Cntrl Reg.</td>
</tr>
<tr>
<td>EXTCONB</td>
<td>0x007509</td>
<td>Ext. Cntrl Reg.</td>
</tr>
</tbody>
</table>
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 pulsed 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.

**Capture Control Register (EVA)**

```
CAPCONA @ 0x007420 (lab file: EvEv.c)
```

- **Capture Reset (not latched)**
  - 0 = clear all result FIFO's and CAPFIFO register
  - 1 = no action

- **Unit 3 Control**
  - 0 = disable
  - 1 = enable

- **ADC Start**
  - 0 = no action
  - 1 = CAP3INT flag

- **Unit 1 & 2 Control**
  - 00 = disable
  - 01 = enable for capture
  - 10 = reserved
  - 11 = enable for QEP

- **Timer Select**
  - 0 = GP Timer 2
  - 1 = GP Timer 1

- **Edge Detection Control**
  - 00 = no detection
  - 01 = rising edge
  - 10 = falling edge
  - 11 = both edges

7 - 30 C28x - Event Manager
### Capture FIFO Status Register (EVA)

CAPFIFOA @ 0x007422 (lab file: Ev.c)

<table>
<thead>
<tr>
<th>15-14</th>
<th>13-12</th>
<th>11-10</th>
<th>9-8</th>
<th>7-0</th>
</tr>
</thead>
<tbody>
<tr>
<td>reserved</td>
<td>CAP3FIFO</td>
<td>CAP2FIFO</td>
<td>CAP1FIFO</td>
<td>reserved</td>
</tr>
</tbody>
</table>

CAPxFIFO bits are automatically adjusted on a capture or FIFO read.

**FIFOx Status:**
- 00 = empty
- 01 = one entry
- 10 = two entries
- 11 = three entries attempted, 1st entry lost
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.
What is an Incremental Quadrature Encoder?

A digital (angular) position sensor

slots spaced $\theta$ deg. apart

photo sensors spaced $\theta/4$ deg. apart

light source (LED)

Incremental Optical Encoder

Quadrature Output from Photo Sensors

How is Position Determined from Quadrature Signals?

Position resolution is $\theta/4$ degrees.

\[
(A, B) = \begin{cases} (00) & \text{(01)} \\ (10) & (11) \end{cases}
\]

Ch. A

Ch. B

increment counter

decrement counter

Quadrature Decoder State Machine

00

11

01
Quadrature Encoder Pulse (QEP)

Incremental Encoder Connections (EVA)

- GP Timer 2 selected as pulse counter
- Timer Prescaler bypassed (i.e. Prescale always 1)

QEP Initialization with GP Timer 2 (EVA)

- Select Timer 2 for Capture number 1 and 2
  \[ \text{CAPCONA} . 9 = 0 \text{ (for Timer 2)} \]
- Optionally pre-load Timer 2 counter
  \[ \text{TxCNT register} \]
- Select directional-up/down mode for Timer 2
  \[ \text{TXCON} . 12 – 11 = 11 \]
- Set T2PER register for desired timer rollover
  e.g. set the number of ticks per revolution
- Enable QEP operation
  \[ \text{CAPCONA} . 14 – 13 = 11 \]
Extension Control Register A (EVA)
EXTCONA @ 0x007409 (lab file: Ev.c)

- **QEP Index Enable**
  - 0 = disable
  - 1 = enable

- **Independent Compare Output Enable Mode**
  - 0 = disable
  - 1 = enable

- **EV Start-of-Conversion Output Enable**
  - 0 = disable
  - 1 = enable

- **CAP3/QEPI Index Qualification Mode**
  - 0 = off
  - 1 = on

**Bits:**
- 15-4: reserved
- 3: EVSOCE
- 2: QEPIE
- 1: QEPIQUAL
- 0: INDOCE
Lab 7: Event Manager

Objective

The objective of this lab is to apply the techniques discussed in module 7 and become familiar with the programming and operation of the Event Manager and its interrupts. General-Purpose Timer 1 and Compare 1 will be setup to generate a 2 kHz, 25% duty cycle symmetric PWM waveform. The waveform will then be sampled with the on-chip analog-to-digital converter and displayed using the graphing feature of Code Composer Studio. Next, 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

Project File

Note: LAB7 files have been provided as a starting point for the lab and need to be completed. DO NOT copy files from a previous lab.

1. A project named Lab7.pjt has been created for this lab. Open the project by clicking on Project → Open... and look in C:\C28x\LABS\LAB7. All Build Options have been configured like the previous lab. The files used in this lab are:
Main_7.c                Labcfg.cmd
Lab.cdb                DSP281x_Headers_BIOS.cmd
User_5_6_7.cmd         CodeStartBranch.asm
SysCtrl.c              Gpio.c
DSP281x_GlobalVariableDefs.c PieCtrl_5_6_7_8_9.c
DefaultIsr_5_6_7.c     Adc.c
Ev_7_8_9_10.c

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

2. Edit `Gpio.c` and adjust the shared I/O pins in Group A for the PWM1 function.

3. In `Ev_7_8_9_10.c`, setup General-Purpose Timer 1 and Compare 1 to implement the PWM waveform as described in the objective for this lab. The following registers need to be modified: T1CON, T1CNT, T1PR, DBTCONA (set deadband units off), CMPR1, ACTRA, COMCONA. (Hint – the last step should be to enable Timer 1). Make use of the global variable names and values that have been set using #define in the beginning of the `Ev_7_8_9_10.c` file. Notice that GPTCONA has been initialized earlier in the code during Timer 2 setup. Save your work.

**Build and Load**

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

5. Reset the DSP.

**Run the Code – PWM Waveform**

6. Open a memory window to view some of the contents of the ADC results buffer. The address label for the ADC results buffer is `AdcBuf`. We will be running our code in real-time mode, and need to have our window continuously refresh. Enable Real-time Mode and be sure that the Global Continuous Refresh option has been checked.

**Note:** For the next step using the ADC, check to be sure that the jumper wire connecting “VREFLO” (pin # P9-18) to “GND” (pin # P9-17) is still in place on the eZdsp™ from the earlier lab.

7. Using another connector wire provided, connect the PWM1 (pin # P8-9) to ADCIN0 (pin # P9-2) on the eZdsp™.

8. Run the code in real-time mode and watch the window update. Verify that the ADC result buffer contains the updated values.

9. Open and setup a graph to plot a 50-point window of the ADC results buffer.
   Click: View → Graph → Time/Frequency... and set the following values:
Lab 7: Event Manager

<table>
<thead>
<tr>
<th>Start Address</th>
<th>AdcBuf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acquisition Buffer Size</td>
<td>50</td>
</tr>
<tr>
<td>Display Data Size</td>
<td>50</td>
</tr>
<tr>
<td>DSP Data Type</td>
<td>16-bit unsigned integer</td>
</tr>
<tr>
<td>Sampling Rate (Hz)</td>
<td>50000</td>
</tr>
<tr>
<td>Time Display Unit</td>
<td>µs</td>
</tr>
</tbody>
</table>

Select OK to save the graph options.

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

**Frequency Domain Graphing Feature of Code Composer Studio**

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

Click: View ➔ Graph ➔ Time/Frequency... and set the following values:

<table>
<thead>
<tr>
<th>Display Type</th>
<th>FFT Magnitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start Address</td>
<td>AdcBuf</td>
</tr>
<tr>
<td>Acquisition Buffer Size</td>
<td>50</td>
</tr>
<tr>
<td>FFT Framesize</td>
<td>50</td>
</tr>
<tr>
<td>DSP Data Type</td>
<td>16-bit unsigned integer</td>
</tr>
<tr>
<td>Sampling Rate (Hz)</td>
<td>50000</td>
</tr>
</tbody>
</table>

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?
13. Fully halt the DSP (real-time mode) by using Debug → Halt and then unchecking the “Real-time mode”.

**Setup Capture Unit 1 to Measure Width of Pulse**

14. Edit `Gpio.c` and adjust the shared I/O pins in Group A for the CAP1 function.

15. In `Ev_7_8_9_10.c`, setup Capture Unit 1 to use Timer 1 as the time base. Configure to detect both rising and falling edges. (Hint – when configuring the CAPCONA register, set bit 15 to 0).

16. Using the “PIE Interrupt Assignment Table” find the location for the CAP1 interrupt, “CAPINT1” and fill in the following information:

   PIE group #:__________  # within group:__________

   This information will be used in the next step.

17. Modify the end of `Ev_7_8_9_10.c` to do the following:

   enable event manager interrupt (EVAIMRC) (Hint: use the EvaRegs structure)

   enable the “CAP1” interrupt in the PIE (Hint: use the PieCtrlRegs structure)

   enable core INT3 (IER register)

18. Open and inspect the Capture Unit 1 interrupt service routine (CAPINT1_ISR) in the file `DefaultIsr_5_6_7.c`.

19. Modify the configuration file `lab.cdb` to setup the PIE vector for the CAP1 interrupt. Click on the plus sign (+) to the left of Scheduling and again on the plus sign (+) to the left of HWI – Hardware Interrupt Service Routine Manager. Click the plus sign (+) to the left of PIE INTERRUPTS. Locate the interrupt location for the CAP1 (use the information from step #16). Right click, select Properties, and type `_CAPINT1_ISR` (with a leading underscore) in the function field. Click OK and save all updates.

**Build and Load**

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

21. Reset the DSP.

**Run the Code – Pulse Width Measurement**

22. Open a memory window to view the address label `CAP_rising`. (Type `&CAP_rising` in the address box).

23. Using the connector wire provided, connect the PWM1 (pin # P8-9) to CAP1 (pin # P8-6) on the eZdsp™.
24. Run the code in real-time mode. Notice the values for \textit{CAP\_rising} and \textit{CAP\_falling} (ignore the \textit{CAP\_duty} value until step 25). Then fully halt the DSP.

\textbf{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 \textit{CAP\_rising} and \textit{CAP\_falling} relate to the compare register setting for PWM1?
- How can the differences be accounted for?

\textbf{Modify Program for Duty Cycle Measurement}

25. 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 cannot 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 \texttt{Ev\_7\_8\_9\_10.c} as required so that Capture 1 uses GP Timer 2 as its timebase.

26. We also need to increase the period of Timer 2. The width of the active portion of the 25\% duty, 2 kHz PWM, is 125\ µsec. However, in \texttt{Ev\_7\_8\_9\_10.c}, the line "\#define ADC\_sample\_period 2999" gives Timer 2 a period of 3000 SYSCLKOUT cycles (equivalent to 50 kHz, or a 20\ µsec period). It would be difficult to use Timer 2 to measure the duty (To understand why, here is as 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 16bits. Then, all we need to do is subtract the start time from the end time using \texttt{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).

In \texttt{Ev\_7\_8\_9\_10.c} change the "ADC\_sample\_period" to "0xffff".

Note: This will have the side effect of making the ADC sampling rate only 2.3 kHz. However, we are done using the ADC in this lab, so it is not a concern.

\textbf{Build and Load}

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

28. Reset the DSP.
Run the Code – Duty Cycle Measurement

29. Be sure that the memory window is open to view the address label \textit{CAP\_duty}. (Use \&\textit{CAP\_duty} in address box).

30. With the wire still connecting the PWM1 (pin # P8-9) to CAP1 (pin # P8-6) on the eZdsp\textsuperscript{TM}, run (real-time mode) the code again, and fully halt it after a few seconds. Observe the values for \textit{CAP\_rising}, \textit{CAP\_falling}, and \textit{CAP\_duty} in a memory window. Notice that \textit{CAP\_duty} is simply the difference between \textit{CAP\_rising} and \textit{CAP\_falling} using 16-bit modulo signed math.

Questions:
\begin{itemize}
  \item What is the value of \textit{CAP\_duty} in memory?
  \item How does it compare with the expected value?
\end{itemize}

End of Exercise
GP Timer Compare PWM Exercise Solution

GP Timer Compare PWM Exercise Solution

\[
\begin{align*}
\text{GPTCONA} &= (xxx00000010000100)b = 0042h \\
\text{T1CON} &= (xx001000010000100)b = 0842h \quad \text{(all x's assigned a value of 0)} \\
\text{T1PR} &= \frac{1}{2} \cdot \frac{\text{carrier period}}{\text{timer period}} = \frac{1}{2} \cdot \frac{20 \mu s}{10 \text{ ns}} = 1000 = 3E8h \\
\text{T1CMPR} &= (100\% \text{ - duty cycle}) \cdot \text{T1PR} = 0.75 \cdot 1000 = 750 = 2EEh
\end{align*}
\]

Diagram:
- Period
- Compare
- Counter
- Output Pin
- CPU clock = 10 ns
- 50 kHz = 20 µs
- Carrier period
- Timer period
- Duty cycle
- Output pin

50 kHz = 20 µs

CPU clock = 10 ns
Introduction

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

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

Learning Objectives

- Compare/contrast integer and fractional operations
- Discuss fixed-point math development limitations
- Compare/contrast floating-point and IQ representation
- Describe the IQmath approach and the problem it solves
Module Topics

Numerical Concepts & IQmath ................................................................. 8-1

Module Topics.................................................................................. 8-2

Numbering System Basics ............................................................... 8-3
  Binary Numbers ............................................................................. 8-3
  Two's Complement Numbers ....................................................... 8-3
  Sign Extension Mode ................................................................. 8-5

Binary Multiplication......................................................................... 8-6

Binary Fractions................................................................................. 8-8
  Representing Fractions in Binary ................................................ 8-8
  Multiplying Binary Fractions ..................................................... 8-9
  Correcting Redundant Sign Bit ..................................................... 8-10

Fraction Coding................................................................................ 8-11

Fractional vs. Integer Representation............................................ 8-12

IQmath.............................................................................................. 8-13
  Floating-Point Representation ................................................... 8-14
  IQ Fractional Representation .................................................... 8-15
  Traditional “Q” Math Approach ............................................... 8-17
  IQmath Approach ...................................................................... 8-19

IQmath Library................................................................................ 8-24

AC Induction Motor Example ........................................................... 8-26

IQmath Applications and Summary ............................................... 8-33

Converting ADC Results into IQ Format ........................................ 8-35

Lab 8: IQmath FIR Filter................................................................... 8-36
Numbering System Basics

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

Binary Numbers

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

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

Examples:

\[ \begin{align*}
0110_2 &= (0 \times 8) + (1 \times 4) + (1 \times 2) + (0 \times 1) = 6_{10} \\
11110_2 &= (1 \times 16) + (1 \times 8) + (1 \times 4) + (1 \times 2) + (0 \times 1) = 30_{10}
\end{align*} \]

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:

\[ \begin{align*}
0110_2 &= (0 \times -8) + (1 \times 4) + (1 \times 2) + (0 \times 1) = 6_{10} \\
11110_2 &= (1 \times -16) + (1 \times 8) + (1 \times 4) + (1 \times 2) + (0 \times 1) = -2_{10}
\end{align*} \]

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:**

<table>
<thead>
<tr>
<th>Original No.</th>
<th>0 1 1 0₂</th>
<th>= 6₁₀</th>
<th>1 1 1 1 0₂</th>
<th>= -2₁₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Load low</td>
<td>0 1 1 0</td>
<td></td>
<td>1 1 1 0</td>
<td></td>
</tr>
<tr>
<td>2. Sign Extend</td>
<td>0 0 0 0 1 1 0</td>
<td>= 4 + 2 = 6</td>
<td>1 1 1 1 1 0</td>
<td>= -128 + 64 + ... + 2 = -2</td>
</tr>
</tbody>
</table>
Sign Extension Mode

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

What is Sign Extension?

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

4 bit Example: Load a memory value into the ACC

<table>
<thead>
<tr>
<th>memory</th>
<th>1101</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load and sign extend</td>
<td></td>
</tr>
<tr>
<td>ACC</td>
<td>1111 1101</td>
</tr>
</tbody>
</table>

= -2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^0
= -128 + 64 + 32 + 16 + 8 + 4 + 1
= -3
**Binary Multiplication**

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

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

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

![Four-Bit Integer Multiplication](image)

In this example, consider the following:

- What are the two input values, and the expected result?
- Why are the “partial products” shifted left as the calculation continues?
- Why is the final partial product “different” than the others?
- What is the result obtained when adding the partial products?
- How shall this result be loaded into the accumulator?
- How shall we fill the remaining bit? Is this value still the expected one?
- How can the result be stored back to memory? What problems arise?
Note: With two’s complement multiplication, the leading “1” in the second multiplicand is a sign bit. If the sign bit is “1”, then take the 2’s complement of the first multiplicand. Additionally, each partial product must be sign-extended for correct computation.

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

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

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

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

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

Representing Fractions in Binary

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

$$1.011 = -1 + 1/4 + 1/8 = -5/8$$

<table>
<thead>
<tr>
<th>Fractional Position</th>
<th>1/8</th>
<th>1/4</th>
<th>1/2</th>
<th>-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual Value</td>
<td>1/8</td>
<td>1/4</td>
<td>1/2</td>
<td>-1</td>
</tr>
<tr>
<td>Binary Representation</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Multiplying Binary Fractions

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

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.
- 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

**Correcting Redundant Sign Bit**

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>1 1 1 1 0 1 0 0</th>
</tr>
</thead>
</table>

- **Correcting Redundant Sign Bit**
  - IQmath: automatically handled *(next topic)*
  - Q math in “C”, shift in software:
    ```c
    int x, y, z;
    z = ((long)x * (long)y) >> 15;
    ```
**Fraction Coding**

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

![Diagram of how fractions are coded](image)

---

**Example: represent the fraction number 0.707**

```c
void main(void) {
    int coef = 32768 * 707 / 1000;
}
```

---

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

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

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

**Examples:**

- To represent 0.62: $32768 \times 62 \div 100$
- To represent 0.1405: $32768 \times 1405 \div 10000$

This method produces a valid number accurate to 16 bits. You will not need to do the math yourself, and changing values in your code becomes rather simple.
The C28x accumulator, a 32-bit register, adds extra range to integer calculations, but this becomes a problem in storing the results back to 16-bit memory.

Conversely, when using fractions, the extra accumulator bits increase precision, which helps minimize accumulative errors. Since any number is accurate (at best) to ± one-half of a LSB, summing two of these values together would yield a worst case result of 1 LSB error. Four summations produce two LSBs of error. By 256 summations, eight LSBs are “noisy.” Since the accumulator holds 32 bits of information, and fractional results are stored from the high accumulator, the extra range of the accumulator is a major benefit in noise reduction for long sum-of-products type calculations.
Implementing complex digital control algorithms on a Digital Signal Processor (DSP), or any other DSP capable processor, typically come across the following issues:

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

The diagram below illustrates a typical development scenario in use today:

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

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

IEEE Std. 754 Single Precision Floating-Point

<table>
<thead>
<tr>
<th>s</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 bit sign</td>
<td>8 bit exponent</td>
<td>23 bit mantissa (fraction)</td>
</tr>
</tbody>
</table>

- **Case 1:** if e = 255 and f ≠ 0, then v = NaN
- **Case 2:** if e = 255 and f = 0, then v = [(-1)^]infinity
- **Case 3:** if 0 < e < 255, then v = [(-1)^][2^(e-127)][1.f]
- **Case 4:** if e = 0 and f ≠ 0, then v = [(-1)^][2^(-126)][0.f]
- **Case 5:** if e = 0 and f = 0, then v = [(-1)^]0

Advantage ⇒ Exponent gives large dynamic range
Disadvantage ⇒ Precision of a number depends on its exponent

Floating-Point does not Solve Everything!

Example:

\[
x = 10.0 \quad (0x41200000) \\
y = 0.000000238 \quad (0x347F8CF1)
\]

\[
z = 10.000000238 \quad \text{WRONG!}
\]

You cannot represent 10.000000238 with single-precision floating point

\[
0x41200000 = 10.000000000 \\
10.000000238 \Leftarrow \text{can't represent!}
\]

So z gets rounded down to 10.000000000
IQ Fractional Representation

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

![IQ Fractional Representation Diagram](https://example.com/)

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

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

I8Q24 Example:  
\[ x = 10.0 \quad (0x0C000000) \]
\[ + \quad y = 0.000000238 \quad (0x00000002) \]
\[ z = 10.000000238 \quad (0x0C000002) \]

Exact Result (this example)
Traditional “Q” Math Approach

Traditional 16-bit “Q” Math Approach
\[ y = mx + b \]

\[
\begin{array}{c}
\text{M} \\
\times \\
\text{X} \\
\downarrow \\
\text{B} \\
\downarrow \\
\text{Y} \\
\end{array}
\]

Align Binary Point For Add

\[ \text{Y} = ((\text{i32}) \ M \times (\text{i32}) \ X + (\text{i32}) \ B \ll Q) \gg Q; \]

in C:

Traditional 32-bit “Q” Math Approach
\[ y = mx + b \]

\[
\begin{array}{c}
\text{M} \\
\times \\
\text{X} \\
\downarrow \\
\text{B} \\
\downarrow \\
\text{Y} \\
\end{array}
\]

Align Decimal Point For Add

\[ \text{Y} = ((\text{i64}) \ M \times (\text{i64}) \ X + (\text{i64}) \ B \ll Q) \gg Q; \]

in C:

Note: Requires support for 64-bit integer data type in compiler
The traditional approach to performing math operations, using fixed-point numerical techniques can be demonstrated using a simple linear equation example. The floating-point code for a linear equation would be:

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

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

<table>
<thead>
<tr>
<th>Q value</th>
<th>Min Value</th>
<th>Max Value</th>
<th>Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q24</td>
<td>(-2^{32-24} = 128.000 000 00)</td>
<td>(2^{32-24} - (\frac{1}{2})^{24} = 127.999 999 94)</td>
<td>((\frac{1}{2})^{24} = 0.000 000 06)</td>
</tr>
</tbody>
</table>

The C code implementation of the linear equation is:

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

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

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

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

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

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

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

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

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

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

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

The linear equation can then be written as follows:

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

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

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

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

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

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

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

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

<table>
<thead>
<tr>
<th>Math Implementations</th>
<th>Linear Equation Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit floating-point math in C</td>
<td>Y = M * X + B;</td>
</tr>
<tr>
<td>32-bit fixed-point &quot;Q&quot; math in C</td>
<td>Y = ((int64) M * (int64) X) + (int64) B &lt;&lt; 24) &gt;&gt; 24;</td>
</tr>
<tr>
<td>32-bit IQmath in C</td>
<td>Y = _IQ24mpy(M, X) + B;</td>
</tr>
<tr>
<td>32-bit IQmath in C++</td>
<td>Y = M * X + B;</td>
</tr>
</tbody>
</table>

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

Multiply Operation

\[ Y = ((\text{i64}) M * (\text{i64}) X) \gg Q + B; \]

Redefine the multiply operation as follows:

\[ _\text{IQmpy}(M, X) == ((\text{i64}) M * (\text{i64}) X) \gg Q \]

This simplifies the equation as follows:

\[ Y = _\text{IQmpy}(M, X) + B; \]

C28x compiler supports “_IQmpy” intrinsic; assembly code generated:

\[
\begin{align*}
\text{MOVL} & \quad \text{XT,} @\text{M} \\
\text{IMPYL} & \quad \text{P,XT,} @\text{X} ; \quad \text{P} = \text{low 32-bits of M*X} \\
\text{QMPYL} & \quad \text{ACC,XT,} @\text{X} ; \quad \text{ACC} = \text{high 32-bits of M*X} \\
\text{LSL64} & \quad \text{ACC:P,} # (32-Q) ; \quad \text{ACC} = \text{ACC:P} \ll 32-Q \\
\text{ADDL} & \quad \text{ACC,} @\text{B} ; \quad \text{Add B} \\
\text{MOVL} & \quad @\text{Y,ACC} ; \quad \text{Result} = Y = _\text{IQmpy}(M*X) + B
\end{align*}
\]; 7 Cycles

IQmath Approach

It looks like floating-point!

Floating-Point

\[
\begin{align*}
\text{float} & \quad Y, M, X, B; \\
Y & = M * X + B;
\end{align*}
\]

Traditional Fix-Point Q

\[
\begin{align*}
\text{long} & \quad Y, M, X, B; \\
Y & = ((\text{i64}) M * (\text{i64}) X + (\text{i64}) B \ll Q)) \gg Q;
\end{align*}
\]

“IQmath” In C

\[
\begin{align*}
_iq & \quad Y, M, X, B; \\
Y & = _\text{IQmpy}(M, X) + B;
\end{align*}
\]

“IQmath” In C++

\[
\begin{align*}
iq & \quad Y, M, X, B; \\
Y & = M * X + B;
\end{align*}
\]

Taking advantage of operator overloading feature in C++, “IQmath” looks like floating-point math (looks natural!)
**IQmath Approach**

**GLOBAL_Q simplification**

User selects “Global Q” value for the whole application

![GLOBAL_Q](image)

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

<table>
<thead>
<tr>
<th>GLOBAL_Q</th>
<th>Max Val</th>
<th>Min Val</th>
<th>Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>7.999 999 996</td>
<td>-8.000 000 000</td>
<td>0.000 000 004</td>
</tr>
<tr>
<td>24</td>
<td>127.999 999 94</td>
<td>-128.000 000 00</td>
<td>0.000 000 06</td>
</tr>
<tr>
<td>20</td>
<td>2047.999 999 99</td>
<td>-2048.000 000 00</td>
<td>0.000 000 01</td>
</tr>
</tbody>
</table>

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

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

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

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

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

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

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

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

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

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

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

Essentially, the programmer writes the code using the "IQmath" library functions and the code can be compiled for floating-point or "IQmath" operations.
### IQmath Library: math & trig functions (v1.4)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Floating-Point</th>
<th>“IQmath” in C</th>
<th>“IQmath” in C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>float A, B;</td>
<td>_iq A, B;</td>
<td>iq A, B;</td>
</tr>
<tr>
<td>multiply</td>
<td>A * B</td>
<td>_IQmpy(A, B)</td>
<td>A * B</td>
</tr>
<tr>
<td>divide</td>
<td>A / B</td>
<td>_IQdiv(A, B)</td>
<td>A / B</td>
</tr>
<tr>
<td>add</td>
<td>A + B</td>
<td>A + B</td>
<td>A + B</td>
</tr>
<tr>
<td>subtract</td>
<td>A - B</td>
<td>A - B</td>
<td>A - B</td>
</tr>
<tr>
<td>boolean</td>
<td>&gt;, &gt;=, &lt;, &lt;=, ==,</td>
<td>&amp;,&amp;,</td>
<td></td>
</tr>
<tr>
<td>trig functions</td>
<td>sin(A),cos(A)</td>
<td>_IQsin(A), _IQcos(A)</td>
<td>IQsin(A), IQcos(A)</td>
</tr>
<tr>
<td></td>
<td>sin(A<em>2pi),cos(A</em>2pi)</td>
<td>_IQsinPU(A), _IQcosPU(A)</td>
<td>IQsinPU(A), IQcosPU(A)</td>
</tr>
<tr>
<td></td>
<td>atan(A),atan2(A,B)</td>
<td>_IQatan(A), _IQatan2(A,B)</td>
<td>IQatan(A), IQatan2(A,B)</td>
</tr>
<tr>
<td></td>
<td>sqrt(A),1/sqrt(A)</td>
<td>_IQsqrt(A), _IQisqrt(A)</td>
<td>IQsqrt(A), IQisqrt(A)</td>
</tr>
<tr>
<td></td>
<td>sqrt(A<em>A + B</em>B)</td>
<td>_IQmag(A,B)</td>
<td>IQmag(A,B)</td>
</tr>
<tr>
<td>saturation</td>
<td>if(A &gt; Pos) A = Pos</td>
<td>_IQsat(A,Pos,Neg)</td>
<td>IQsat(A,Pos,Neg)</td>
</tr>
<tr>
<td></td>
<td>if(A &lt; Neg) A = Neg</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Accuracy of functions/operations approx ~28 to ~31 bits

Additionally, the "IQmath" library contains DSP library modules for filters (FIR & IIR) and Fast Fourier Transforms (FFT & IFFT).

### IQmath Library: Conversion Functions (v1.4)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Floating-Point</th>
<th>“IQmath” in C</th>
<th>“IQmath” in C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>iq to iqN</td>
<td>A</td>
<td>_IQtoIQN(A)</td>
<td>IQtoIQN(A)</td>
</tr>
<tr>
<td>iqN to iq</td>
<td>A</td>
<td>_IQNtoIQ(A)</td>
<td>IQNtoIQ(A)</td>
</tr>
<tr>
<td>integer(iq)</td>
<td>(long) A</td>
<td>_IQInt(A)</td>
<td>IQInt(A)</td>
</tr>
<tr>
<td>fraction(iq)</td>
<td>A - (long) A</td>
<td>_IQfrac(A)</td>
<td>IQfrac(A)</td>
</tr>
<tr>
<td>iq = iq*long</td>
<td>A * (float) B</td>
<td>_IQmpyI32(A,B)</td>
<td>IQmpyI32(A,B)</td>
</tr>
<tr>
<td>integer(iq*long)</td>
<td>(long) (A * (float) B)</td>
<td>_IQmpyI32int(A,B)</td>
<td>IQmpyI32int(A,B)</td>
</tr>
<tr>
<td>fraction(iq*long)</td>
<td>A - (long) (A * (float) B)</td>
<td>_IQmpyI32frac(A,B)</td>
<td>IQmpyI32frac(A,B)</td>
</tr>
<tr>
<td>qN to iq</td>
<td>A</td>
<td>_QNtoIQ(A)</td>
<td>QNtoIQ(A)</td>
</tr>
<tr>
<td>iq to qN</td>
<td>A</td>
<td>_IQtoQN(A)</td>
<td>IQtoQN(A)</td>
</tr>
<tr>
<td>string to iq</td>
<td>atof(char)</td>
<td>_atofIQ(char)</td>
<td>atofIQ(char)</td>
</tr>
<tr>
<td>IQ to float</td>
<td>A</td>
<td>_IQtoF(A)</td>
<td>IQtoF(A)</td>
</tr>
</tbody>
</table>

IQmath.lib > contains library of math functions
IQmathLib.h > C header file
IQmathCPP.h > C++ header file
16-Bit vs. 32-Bit

The "IQmath" approach could also be used on 16 bit sized numbers and for many problems, this is sufficient resolution. However, in many control cases, the user needs to use many different "Q" values to accommodate the limited resolution of a 16-bit number.

With DSP devices like the TMS320C28x processor, which can perform 16 bit and 32 bit sized math with equal efficiency, the choice becomes more of productivity (time to market). Why bother spending a whole lot of time trying to code using 16 bit numbers when you can simply use 32 bit numbers, pick one value of "Q" that will accommodate all cases and not worry about spending too much time optimizing.

Of course there is a concern on data RAM usage if numbers that could be represented in 16 bits all use 32 bits. This is becoming less of an issue in today's processors because of the finer technology used and the amount of RAM that can be cheaply integrated. However, in many cases, this problem can be mitigated by performing intermediate calculations using 32 bit numbers and converting the input from 16 bit to 32 bit and converting the output from 32 to 16 bit before storing the final results. In many problems, it is the intermediate calculations that require additional accuracy to avoid quantization problems.
The "IQmath" approach is ideally suited for applications where a large numerical dynamic range is not required. Motor control is an example of such an application (audio and communication algorithms are other applications). As an example, the IQmath approach has been applied to the sensor-less direct field control of an AC induction motor. This is probably one of the most challenging motor control problems and as will be shown later, requires numerical accuracy greater then 16-bits in the control calculations.

The above slide is a block diagram representation of the key control blocks and their interconnections. Essentially this system implements a "Forward Control" block for controlling the d-q axis motor current using PID controllers and a "Feedback Control" block using back emf's integration with compensated voltage from current model for estimating rotor flux based on current and voltage measurements. The motor speed is simply estimated from rotor flux differentiation and open-loop slip computation. The system was initially implemented on a "Simulator Test Bench" which uses a simulation of an "AC Induction Motor Model" in place of a real motor. Once working, the system was then tested using a real motor on an appropriate hardware platform.

Each individual block shown in the slide exists as a stand-alone C/C++ module, which can be interconnected to form the complete control system. This modular approach allows reusability and portability of the code.
The next few slides show the coding of one particular block, PARK Transform, using floating-point and "IQmath" approaches in C and C++:

**AC Induction Motor Example**

**Park Transform - floating-point C code**

```c
#include "math.h"
#define TWO_PI 6.28318530717959
void park_calc(PARK *v)
{
    float cos_ang, sin_ang;
    sin_ang = sin(TWO_PI * v->ang);
    cos_ang = cos(TWO_PI * v->ang);
    v->de = (v->ds * cos_ang) + (v->qs * sin_ang);
    v->qe = (v->qs * cos_ang) - (v->ds * sin_ang);
}
```

**AC Induction Motor Example**

**Park Transform - converting to "IQmath" C code**

```c
#include "math.h"
#include "IQmathLib.h"
#define TWO_PI _IQ(6.28318530717959)
void park_calc(PARK *v)
{
    _iq cos_ang, sin_ang;
    sin_ang = _IQsin(_IQmpy(TWO_PI, v->ang));
    cos_ang = _IQcos(_IQmpy(TWO_PI, v->ang));
    v->de = _IQmpy(v->ds, cos_ang) + _IQmpy(v->qs, sin_ang);
    v->qe = _IQmpy(v->qs, cos_ang) - _IQmpy(v->ds, sin_ang);
}
```
As can be seen, the floating-point C and "IQmath" C++ code looks almost identical. It is quite a simple and fast procedure to take any floating-point algorithm and convert it to an "IQmath" algorithm.

The complete system was coded using "IQmath". Based on analysis of coefficients in the system, the largest coefficient had a value of 33.3333. This indicated that a minimum dynamic range of 7 bits (+/-64 range) was required. Therefore, this translated to a GLOBAL_Q value of 32-7 = 25 (Q25). Just to be safe, the initial simulation runs were conducted with GLOBAL_Q = 24 (Q24) value.
The code was compiled and run on the Texas Instruments TMS320C28x fixed-point DSP device (IQ_MATH mode) and on the TMS320C3x floating-point DSP device (FLOAT_MATH mode). The plots of speed and stator current for both devices matched and are shown below.

The plots start from a step change in reference speed from 0.0 to 0.5 and 1024 samples are taken. The speed eventually settles to the desired reference value and the stator current exhibits a clean and stable oscillation. The block diagram slide shows at which points in the control system the plots are taken from.
AC Induction Motor Example

GLOBAL_Q = 27, system unstable

IQmath: speed

IQmath: current

AC Induction Motor Example

GLOBAL_Q = 16, system unstable

IQmath: speed

IQmath: current
With the ability to select the GLOBAL_Q value for all calculations in the "IQmath", an experiment was conducted to see what maximum and minimum Q value the system could tolerate before it became unstable. The results are tabulated in the slide below:

<table>
<thead>
<tr>
<th>Q range</th>
<th>Stability Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q31 to Q27</td>
<td>Unstable (not enough dynamic range)</td>
</tr>
<tr>
<td>Q26 to Q19</td>
<td>Stable</td>
</tr>
<tr>
<td>Q18 to Q0</td>
<td>Unstable (not enough resolution, quantization problems)</td>
</tr>
</tbody>
</table>

The above indicates that, the AC induction motor system that we simulated requires a minimum of 7 bits of dynamic range (+/-64) and requires a minimum of 19 bits of numerical resolution (+/-0.000002). This confirms our initial analysis that the largest coefficient value being 33.33333 required a minimum dynamic range of 7 bits. As a general guideline, users using IQmath should examine the largest coefficient used in the equations and this would be a good starting point for setting the initial GLOBAL_Q value. Then, through simulation or experimentation, the user can reduce the GLOBAL_Q until the system resolution starts to cause instability or performance degradation. The user then has a maximum and minimum limit and a safe approach is to pick a midpoint.

What the above analysis also confirms is that this particular problem does require some calculations to be performed using greater then 16 bit precision. The above example requires a minimum of $7 + 19 = 26$ bits of numerical accuracy for some parts of the calculations. Hence, if one were implementing the AC induction motor control algorithm using a 16 bit fixed-point DSP, it would require the implementation of higher precision math for certain portions. This would take more cycles and programming effort.

The great benefit of using GLOBAL_Q is that the user does not necessarily need to go into details to assign an individual Q for each variable in a whole system, as is typically done in conventional fixed-point programming. This is time consuming work. By using 32-bit resolution and the "IQmath" approach, the user can easily evaluate the overall resolution and quickly implement a typical digital motor control application without quantization problems.
AC Induction Motor Example

Performance comparisons

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>C28x C IQmath (-g) (150MHz)</th>
<th>C3x C float (-g) (75MHz)</th>
<th>C67x C float (-g) (167MHz)</th>
<th>C67x C float (no -g) (167MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1: Forward control cycles</td>
<td>482</td>
<td>308</td>
<td>627</td>
<td>432</td>
</tr>
<tr>
<td>B2: ACI model cycles</td>
<td>564</td>
<td>215</td>
<td>321</td>
<td>183</td>
</tr>
<tr>
<td>B3: Feedback control cycles</td>
<td>1081</td>
<td>829</td>
<td>1295</td>
<td>973</td>
</tr>
<tr>
<td>Key benchmark cycles (B1+B3)</td>
<td>1563</td>
<td>1137</td>
<td>1922</td>
<td>1405</td>
</tr>
<tr>
<td>% of available MHz used</td>
<td>20.9%</td>
<td>30.3%</td>
<td>23.0%</td>
<td>16.8%</td>
</tr>
</tbody>
</table>

Notes: C28x compiled on CGT V3.03, V1.4c IQmath Lib (debug enabled (-g), max opt).
C3x compiled on CGT V5.12, RTS30R Lib (debug enabled (-g), max opt).
C67x compiled on CGT V4.20, RTSFast Lib (debug enabled (-g)/disabled, max opt).
On C67x, when debug is enabled (-g), parallel execution of operations are limited.
On C28x & C3x, turning off debug (no -g) has minimal impact on performance.

Using the profiling capabilities of the respective DSP tools, the table above summarizes the number of cycles and code size of the forward and feedback control blocks.

The MIPS used is based on a system sampling frequency of 20kHz, which is typical of such systems.
IQmath Applications and Summary

Where Is IQmath Applicable?

Anywhere a large dynamic range is not required

Motor Control (PID, State Estimator, Kalman,...)
Servo Control
Modems
Audio (MP3, etc.)
Imaging (JPEG, etc.)
Any application using 16/32-bit fixed-point Q math

Where it is not applicable

Graphical applications (3D rotation, etc.)
When trying to squeeze every last cycle

IQmath Approach Summary

“IQmath” + fixed-point processor with 32-bit capabilities =

- Seamless portability of code between fixed and floating-point devices
  - User selects target math type in “IQmathLib.h” file
    - #if MATH_TYPE == IQ_MATH
    - #if MATH_TYPE == FLOAT_MATH
- One source code set for simulation vs. target device
- Numerical resolution adjustability based on application requirement
  - Set in “IQmathLib.h” file
    - #define GLOBAL_Q 18
  - Explicitly specify Q value
    - _iq20 X, Y, Z;
- Numerical accuracy without sacrificing time and cycles
- Rapid conversion/porting and implementation of algorithms

IQmath library is freeware - available from TI DSP website
http://www.dspvillage.ti.com (follow C2000 DSP links)
The IQmath approach, matched to a fixed-point processor with 32x32 bit capabilities enables the following:

- Seamless portability of code between fixed and floating-point devices
- Maintenance and support of one source code set from simulation to target device
- Adjustability of numerical resolution (Q value) based on application requirement
- Implementation of systems that may otherwise require floating-point device
- Rapid conversion/porting and implementation of algorithms
Converting ADC Results into IQ Format

As you may recall, the converted values of the ADC are placed in the upper 12 bit of the RESULT0 register. Before these values are filtered using the IQmath library, they need to be put into the IQ format as a 32-bit long.

---

**Getting the ADC Result into IQ Format**

![Getting the ADC Result into IQ Format](image)

---

Do not sign extend

Notice that the 32-bit long is already in IQ16 format!

```c
#define AdcFsVoltage   IQ(3.0)   // ADC full scale voltage
#define _iq IQ(1.5)      // IQ16 format

void main(void)
{
    // convert the unsigned 16-bit result to unsigned 32-bit
    temp = _IQ16toIQ( (iq) AdcRegs.ADCRESULT0);

    // convert resulting IQ16 to IQ format
    temp = _IQmpy(_IQ(3.0), temp); // scale by ADC full-scale range (optional)

    Result = _IQmpy(AdcFsVoltage, temp);
}
```

---

For uni-polar ADC inputs (i.e., 0 to 3 V inputs), a conversion to global IQ format can be achieved with:

```
IQresult_unipolar = _IQmpy(_IQ(3.0), _IQ16toIQ((iq) AdcRegs.ADCRESULT0));
```

How can we modify the above to recover bi-polar inputs, for example +/-1.5 volts? One could do the following to offset the +1.5V analog biasing applied to the ADC input:

```
IQresult_bipolar =
    _IQmpy(_IQ(3.0), _IQ16toIQ((iq) AdcRegs.ADCRESULT0)) - _IQ(1.5);
```

However, one can see that the largest intermediate value the equation above could reach is 3.0. This means that it cannot be used with an IQ data type of IQ30 (IQ30 range is -2 < x < ~2). Since the IQmath library supports IQ types from IQ1 to IQ30, this could be an issue in some applications.

The following clever approach supports IQ types from IQ1 to IQ30:

```
IQresult_bipolar =
    _IQmpy(_IQ(1.5), _IQ15toIQ((iq) (int16) (AdcRegs.ADCRESULT0 ^ 0x8000)));
```

The largest intermediate value that this equation could reach is 1.5. Therefore, IQ30 is easily supported.
Lab 8: IQmath FIR Filter

- **Objective**

  The objective of this lab is to apply the techniques discussed in module 8 and to become familiar with IQmath programming. In the previous lab, General-Purpose Timer 1 and Compare 1 from Event Manager A (EVA) were setup to generate a 2 kHz, 25% duty cycle symmetric PWM waveform. The waveform was then sampled with the on-chip analog-to-digital converter. In this lab the sampled waveform will be passed through an IQmath FIR filter and displayed using the graphing feature of Code Composer Studio.

- **Procedure**

  **Project File**

  Note: LAB8 files have been provided as a starting point for the lab and need to be completed. DO NOT copy files from a previous lab.

  1. A project named Lab8.pjt has been created for this lab. Open the project by clicking on Project ➔ Open... and look in C:\C28x\LABS\LAB8. All Build Options have been configured like the previous lab. The files used in this lab are:
Project Build Options

2. Setup the include search path to include the IQmath header file. Open the Build Options and select the Compiler tab. In the Preprocessor Category, find the Include Search Path (-i) box and add to the end of the line (preceded with a semicolon to append this directory to the existing search path):

    ;c:\tidcs\c28\IQmath\cIQmath\include

3. Setup the library search path to include the IQmath library. Open the Build Options and select the linker tab.

   a. In the Basic Category, find the Library Search Path (-i) box and enter:

       c:\tidcs\c28\IQmath\cIQmath\lib

   b. In the Include Libraries (-l) box enter: IQmath.lib

   Then select OK to save the Build Options.

Include IQmathLib.h

4. Open Lab.h and uncomment the line that includes the IQmathLib.h header file. Next, in the Function Prototypes section, uncomment the function prototype for IQssfir(), the IQ math single-sample FIR filter function.

Inspect User_8_9.cmd

5. Open and inspect User_8_9.cmd. First, notice that a section called “IQmath” is being linked to H0SARAM. The IQmath section contains the IQmath library functions (code). Second, notice that a section called “IQmathTables” is being linked to the BOOTROM with a TYPE = NOLOAD modifier after its allocation. The IQmath tables are used by the IQmath library functions. The NOLOAD modifier allows the linker to resolve all addresses in the section, but the section is not actually placed into the .out file. This is done because the section is already present in the device ROM (you cannot load data into ROM after the device is manufactured!). The tables were put in the ROM by TI when the device was manufactured. All we need to do is link the section to the addresses where it is known to already reside (the tables are the very first thing in the BOOT ROM, starting at address 0x3FF000).
Select a Global IQ value

6. Open the file c:\tidcs\c28\IQmath\cIQmath\include\IQmathLib.h. Confirm that the GLOBAL_Q type (near beginning of file) is set to a value of 24. If it is not, modify as necessary:

#define   GLOBAL_Q       24

Recall that this Q type will provide 8 integer bits and 24 fractional bits. Dynamic range is therefore \(-255 \leq x < +255\), which is sufficient for our purposes in the workshop.

IQmath Single-Sample FIR Filter

7. Open and inspect DefaultIsr_8.c. Notice that the ADCINT_ISR calls the IQmath single-sample FIR filter function, IQssfir(). The filter coefficients have been defined in the beginning of the file. Now open and inspect the IQssfir() function in Filter.c. This is a simple, unoptimized coding of a basic IQmath single-sample FIR filter.

Build and Load

8. Click the “Build” button.

9. Reset the DSP.

Run the Code – Filtered Waveform

10. Open a memory window to view some of the contents of the filtered ADC results buffer. The address label for the filtered ADC results buffer is AdcBufFiltered. Set the Q-Value to 24 (which matches the IQ format being used for this variable) and the Format to 32-Bit Signed Int. We will be running our code in real-time mode, and need to have our window continuously refresh. Enable Real-time Mode and be sure that the Global Continuous Refresh option has been checked.

Note: For the next step, check to be sure that the jumper wire connecting “VREFLO” (pin # P9-18) to “GND” (pin # P9-17) and the wire connecting PWM1 (pin # P8-9) to ADCIN0 (pin # P9-2) are still in place on the eZdsp™.

11. Run the code in real-time mode and watch the memory window update. Verify that the ADC result buffer contains updated values.

12. Open and setup a dual time graph to plot a 50-point window of the filtered and unfiltered ADC results buffer. Click: View → Graph → Time/Frequency... and set the following values:
Display Type | Dual Time  
---|---
Start Address – upper display | AdcBufFiltered  
Start Address – lower display | AdcBuf  
Acquisition Buffer Size | 50  
Display Data Size | 50  
DSP Data Type | 32-bit signed integer  
Q-value | 24  
Sampling Rate (Hz) | 50000  
Time Display Unit | µs  

Select OK to save the graph options.

13. The graphical display should show the generated IQmath FIR filtered 2 kHz, 25% duty cycle symmetric PWM waveform in the upper display and the unfiltered waveform generated in the previous lab exercise in the lower display. Notice the shape and phase differences between the waveform plots (the filtered curve has rounded edges, and lags the unfiltered plot by several samples). The amplitudes of both plots should run from 0 to 3.0.

14. Fully halt the DSP (real-time mode) by using Debug → Halt and then unchecking the “Real-time mode”.

End of Exercise
Lab 8 Reference: IQmath FIR Filter

Bode Plot of Digital Low Pass Filter

Coefficients: [1/16, 4/16, 6/16, 4/16, 1/16]

Sample Rate: 50 kHz
Introduction

This module discusses the basic features of using DSP/BIOS in a system. Scheduling threads, periodic functions, and the use of real-time analysis tools will be demonstrated.

Learning Objectives

<table>
<thead>
<tr>
<th>Learning Objectives</th>
</tr>
</thead>
<tbody>
<tr>
<td>◆ Introduction to DSP/BIOS</td>
</tr>
<tr>
<td>◆ Scheduling DSP/BIOS threads</td>
</tr>
<tr>
<td>◆ Periodic Functions</td>
</tr>
<tr>
<td>◆ Real-time Analysis Tools</td>
</tr>
<tr>
<td>◆ DSP/BIOS API modules and summary</td>
</tr>
</tbody>
</table>
Module Topics

Using DSP/BIOS ................................................................................................................................. 9-1

Module Topics ................................................................................................................................. 9-2

Introduction to DSP/BIOS ............................................................................................................... 9-3

Scheduling DSP/BIOS Threads ...................................................................................................... 9-5

Periodic Functions .......................................................................................................................... 9-12

Real-time Analysis Tools .............................................................................................................. 9-13

DSP/BIOS API Module and Summary ............................................................................................. 9-14

Lab 9: DSP/BIOS ............................................................................................................................. 9-15
Introduction to DSP/BIOS

What is DSP/BIOS?

- A full-featured, scalable real-time kernel
  - System configuration tools
  - Preemptive multi-threading scheduler
  - Real-time analysis tools

Why use DSP/BIOS?

- Helps manage complex system resources
- Integrated with Code Composer Studio IDE
  - Requires no runtime license fees
  - Fully supported by TI and is a key component of TI's eXpressDSP™ real-time software technology
- Uses minimal MIPS and memory (2-8Kw)

DSP/BIOS Configuration Tool (.cdb)

- System Setup Tools
  - Handles memory configuration (builds .cmd file), run-time support libraries, interrupt vectors, system setup and reset, etc.
- Real-Time Analysis Tools
  - Allows application to run uninterrupted while displaying debug data
- Real-Time Scheduler
  - Preemptive thread manager kernel
- Real-Time I/O
  - Allows two way communication between threads or between target and PC host
### DSP/BIOS Terminology

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread</td>
<td>a path of program execution</td>
</tr>
<tr>
<td>scheduler</td>
<td>a program that manages threads</td>
</tr>
<tr>
<td>preemption</td>
<td>the act of a higher priority thread interrupting a lower priority thread</td>
</tr>
<tr>
<td>post</td>
<td>an event signal that often makes a thread “ready”</td>
</tr>
<tr>
<td>pend</td>
<td>When a thread waits for an event post</td>
</tr>
<tr>
<td>semaphore</td>
<td>a data object that tracks event occurrences (used by post and pend)</td>
</tr>
</tbody>
</table>
Scheduling DSP/BIOS Threads

Problem: Add a Function to your Code

Existing Function

New Function

- Issues:
  - Do we have enough bandwidth (MIPS)?
  - Will one routine conflict with the other?
  - How is a compound system created?

What are some possible solutions?

Possible Solution Using while Loop

- Call each function from an endless loop within main

Potential Problems:

What if Algorithms run at different rates:
- motor current loop at 20 kHz
- respond to keypad input at 2 Hz

What if one algorithm consumes enough MIPS to force the other algorithm to miss its real-time deadlines / delays its response?

How are these problems typically solved?
Possible Solution Using Interrupts

- An interrupt driven system places each function in its own ISR

<table>
<thead>
<tr>
<th>Function</th>
<th>Period</th>
<th>Compute</th>
<th>CPU Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function 1</td>
<td>0.05 ms</td>
<td>1 µs</td>
<td>2%</td>
</tr>
<tr>
<td>Function 2</td>
<td>500 ms</td>
<td>3 µs</td>
<td>~ 0%</td>
</tr>
</tbody>
</table>

Only one can run at a time
Interrupt is missed...

DSP/BIOS Solution - HWI

- Nested interrupts allow hardware interrupts to preempt each other

- Use DSP/BIOS HWI dispatcher for context save/restore, and allow preemption

- Reasonable approach if you have limited number of interrupts/functions

- Limitation: number of HWI and their priorities are statically determined, only one HWI function for each interrupt
**Scheduling DSP/BIOS Threads**

**HWI Dispatcher for ISRs**

- For non-BIOS code, we use the `interrupt` keyword to declare an ISR
  - tells the compiler to perform context save/restore
  ```c
  interrupt void MyHWI(void)
  {
  }
  ```
- For DSP/BIOS code, the dispatcher will perform the save/restore
  - Remove the `interrupt` keyword from the MyHWI()
  - Check the “Use Dispatcher” box when you configure the interrupt vector in the DSP/BIOS config tools

**Using Software Interrupts - SWI**

- Make each algorithm an independent software interrupt
- SWI scheduling is handled by DSP/BIOS
  - HWI function triggered by hardware
  - SWI function triggered by software e.g. a call to SWI_post()
- Why use a SWI?
  - No limitation on number of SWIs, and priorities for SWIs are user-defined
  - SWI can be scheduled by hardware or software event(s)
  - Defer processing from HWI to SWI
Scheduling DSP/BIOS Threads

SWI Properties

Managing SWI Priority

- Drag and Drop SWIs to change priority
- Equal priority SWIs run in the order that they are posted
Priority Based Thread Scheduling

HWI 2 (highest)

SWI 3

SWI 2

SWI 1

MAIN

IDLE (lowest)

User sets the priority...BIOS does the scheduling

Another Solution – Tasks (TSK)

- DSP/BIOS tasks (TSK) are similar to SWI, but offer additional flexibility
  - SWIs must run to completion
  - TSKs can be terminated by software
- Tradeoffs
  - SWI context switch is faster than TSK
  - TSK module requires more code space
  - TSKs have their own stack
- User preference and system needs usually dictates choice, easy to use both
Scheduling DSP/BIOS Threads

SWIs and TSKs

- Similar to hardware interrupt, but triggered by SWI_post()
- All SWI's share system software stack

SWI

- SWI_post

start

“run to completion”

divided

divided

divided

end

TSK

- SEM_post() readies the TSK which pend on an event
- The event triggers the TSK to run once and pend
- Each TSK has its own stack, which allows them to pause (or be terminated)

- SEM_pend

Pause (blocked state)

DSP/BIOS Thread Types

<table>
<thead>
<tr>
<th>Priority</th>
<th>HWI (Hardware Interrupts)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Used to implement 'urgent' part of real-time event</td>
</tr>
<tr>
<td></td>
<td>Triggered by hardware interrupt</td>
</tr>
<tr>
<td></td>
<td>HWI priorities set by hardware</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Priority</th>
<th>SWI (Software Interrupts)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Use SWI to perform HWI 'follow-up' activity</td>
</tr>
<tr>
<td></td>
<td>SWI's are 'posted' by software</td>
</tr>
<tr>
<td></td>
<td>Multiple SWIs at each of 15 priority levels</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Priority</th>
<th>TSK (Tasks)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Use TSK to run different programs concurrently under separate contexts</td>
</tr>
<tr>
<td></td>
<td>TSK's enabled by posting 'semaphore' (a signal)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Priority</th>
<th>IDL (Background)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Multiple IDL functions</td>
</tr>
<tr>
<td></td>
<td>Runs as an infinite loop, like traditional while loop</td>
</tr>
<tr>
<td></td>
<td>All BIOS data transfers to host occur here</td>
</tr>
</tbody>
</table>
Enabling BIOS – Return from main()

```c
main
{
...  // return to BIOS
}
```

- Must delete the endless while() loop
- `main()` returns to BIOS IDLE thread, allowing BIOS to schedule events, transfer info to host, etc.
- An endless while() loop in `main()` will not allow BIOS to activate
Periodic Functions

Periodic functions run at a specific rate in your system:
- e.g. LED blink requires 0.5 Hz
- Use the CLK Manager to specify the DSP/BIOS CLK rate in microseconds per “tick”
- Use the PRD Manager to specify the period (for the function) in ticks
- Allows multiple periodic functions with different rates

Creating a Periodic Function

![Diagram showing the creation of a periodic function with DSP/BIOS CLK and tick periods for func1, func2, and func3.]
Real-time Analysis Tools

Built-in Real-Time Analysis Tools

- Gather data on target (3-10 CPU cycles)
- Send data during BIOS IDL (100s of cycles)
- Format data on host (1000s of cycles)
- Data gathering does NOT stop target CPU

Execution Graph

- Software logic analyzer
- Debug event timing and priority

CPU Load Graph

- Shows amount of CPU horsepower being consumed

Statistics View

- Profile routines w/o halting the CPU

Message LOG

- Send debug msgs to host
- Doesn’t halt the DSP
- Deterministic, low DSP cycle count
- More efficient than traditional printf()

LOG_printf(&trace, "AdcSwi_count = %u", AdcSwi_count++);
DSP/BIOS API Module and Summary

DSP/BIOS - API Modules

Instrumentation/Real-Time Analysis
- LOG: Message log manager
- STS: Statistics accumulator manager
- TRC: Trace manager
- RTDX: Real-Time Data eXchange manager

Thread Types
- HWI: Hardware interrupt manager
- SWI: Software interrupt manager
- TSK: Multi-tasking manager
- IDL: Idle function & process loop manager

Clock and Periodic Functions
- CLK: System clock manager
- PRD: Periodic function manager

TSK Communication/Synchronization
- SEM: Semaphores manager
- MBX: Mailboxes manager
- LCK: Resource lock manager

Device-Independent Input/Output
- PIP: Data pipe manager
- HST: Host input/output manager
- SIO: Stream I/O manager
- DEV: Device driver interface

Memory and Low-Level Primitives
- MEM: Memory manager
- SYS: System services manager
- QUE: Queue manager
- ATM: Atomic functions
- GBL: Global setting manager

Summary and Benefits of DSP/BIOS

- **Fast time to market**
  - no need to develop or maintain a “home-brew” kernel
- **Efficient debugging of real-time applications**
  - Real-Time Analysis
- **Create robust applications**
  - industry proven kernel technology
- **Reduce cost of software maintenance**
  - code reuse and standardized software
- **Standardized APIs**
  - enable rapid migration across C2000 TMS320 DSPs
- **Small footprint (2-8Kw)**
  - easily fits in limited memory space
- **Set of library functions (scalable)**
  - use only what is needed to minimize code and data size
- **Full featured kernel (extensible)**
  - allows additional OS functions in future
Lab 9: DSP/BIOS

Objective

The objective of this lab is to apply the techniques discussed in module 9 and to become familiar with DSP/BIOS. In this lab exercise, we are going to change the ADCINT_ISR HWI to a SWI. Then, we will replace the LED blink routine with a Periodic Function. Also, some features of the real-time analysis tools will be demonstrated.

Lab 9: DSP/BIOS

Objective:

- Change ADCINT_ISR HWI to SWI
- Replace LED blink routine with a Periodic Function

Procedure

Project File

Note: LAB9 files have been provided as a starting point for the lab and need to be completed. DO NOT copy files from a previous lab.

1. A project named Lab9.pjt has been created for this lab. Open the project by clicking on Project → Open... and look in C:\C28x\LABS\LAB9. All Build Options have been configured like the previous lab. The files used in this lab are:

Note: Due to a bug in CCS and DSP/BIOS, the configuration file name (.cdb) and the project output (.out) and map (.map) names must be the same. Therefore, in this module the Build Options output and map file names will be lab.out and map.out, respectively.
Add a SWI to main.c

2. Open Main_9.c and notice that space has been added at the end of main() for two new functions which will be used in this module – AdcSwi() and LedBlink(). In the next few steps, we will move part of the ADCINT_ISR() routine from DefaultIsr_9_10.c to this space in Main_9.c.

3. Open DefaultIsr_9_10.c and locate the ADCINT_ISR() routine. Move the entire contents of the ADCINT_ISR() routine to the AdcSwi() function in Main_9.c with the following exceptions:

   **DO NOT MOVE:**
   
   • The instruction used to acknowledge the PIE group interrupt.
   
   • The GPIO pin toggle code.
   
   • The LED toggle code.

   Be sure to move the static local variable declaration that is used to index into the ADC buffers.

   *Comment:* In almost all applications, the PIE group acknowledge code is left in the HWI (rather than move it to a SWI). This allows other interrupts to occur on that PIE group even if the SWI has not yet executed. On the other hand, we are leaving the GPIO and LED toggle code in the HWI just as an example. It illustrates that you can post a SWI and also do additional operations in the HWI. DSP/BIOS is extremely flexible!

4. In the beginning of DefaultIsr_9_10.c, move the global variables, definition statements and coefficients for the filter function to the beginning of Main_9.c.

5. Delete the `interrupt` key word from the ADCINT_ISR. The interrupt keyword is not used when a HWI is under DSP/BIOS control. A HWI is under DSP/BIOS control when it uses any DSP/BIOS functionality, such as posting a SWI, or calling any DSP/BIOS function or macro.

6. In Main_9.c remove the in-line assembly code used to enable global interrupts. DSP/BIOS will enable global interrupts after main().

7. In Main_9.c remove the while() loop (the endless loop). When using DSP/BIOS, you must return from main(). DSP/BIOS will then take-over control of the software execution.
### Post a SWI

8. In `DefaultIsr_9_10.c` add the following SWI_post to the ADCINT_ISR(), just after the structure used to acknowledge the PIE group:

```c
SWI_post(&ADC_swif);  // post a SWI
```

This post a SWI that will execute the ADC_swif() code you populated a few steps back in the lab. In other words, the ADC interrupt still executes the same code as before. However, some of that code is now in a posted SWI that DSP/BIOS will execute according to the specified scheduling priorities.

### Add the SWI to the CDB File

9. In the configuration file `Lab.cdb` we need to add and setup the AdcSwif() SWI. Open `Lab.cdb` and click on the plus sign (+) to the left of Scheduling and again on the plus sign (+) to the left of SWI – Software Interrupt Manager.

10. Right click on SWI – Software Interrupt Manager and select Insert SWI. SWI0 will be added. Right-click on it, and rename it to ADC_swif. This is just an arbitrary name. We want to differentiate the AdcSwif() function itself (which is nothing but an ordinary C function) from the DSP/BIOS SWI object which we are calling ADC_swif.

11. Select the Properties for ADC_swif and type _AdcSwif (with a leading underscore) in the function field. Click OK. This tells DSP/BIOS that it should run the function AdcSwif() when it executes the ADC_swif SWI.

12. We need to have the PIE for the ADC interrupt use the dispatcher. The dispatcher will automatically perform the context save and restore, and allow the DSP/BIOS scheduler to have insight into the ISR. You may recall from an earlier lab that the ADC interrupt is located at PIE_INT1_6.

   Click on the plus sign (+) to the left of HWI – Hardware Interrupt Service Routine Manager. Click the plus sign (+) to the left of PIE INTERRUPTS. Locate the interrupt location for the ADC: PIE_INT1_6. Right click, select Properties, and select the Dispatcher tab.

   Now check the “Use Dispatcher” box and select OK. Close the configuration file and click YES to save changes.

### Build, Load, and Reset

13. Click the “Build” button to rebuild and load the project.

14. Reset the DSP.
Run the Code – AdcSwi()

15. We will be running our code in real-time mode, and need to have our window continuously refresh. Enable Real-time Mode and be sure that the Global Continuous Refresh option has been checked.

**Note:** For the next step, check to be sure that the jumper wire connecting “VREFLO” (pin # P9-18) to “GND” (pin # P9-17) and the wire connecting PWM1 (pin # P8-9) to ADCIN0 (pin # P9-2) are still in place on the eZdsp™.

16. Run the code in real-time mode watch the window update. Verify that the ADC result buffer contains updated values.

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

<table>
<thead>
<tr>
<th>Display Type</th>
<th>Dual Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start Address – upper display</td>
<td>AdcBufFiltered</td>
</tr>
<tr>
<td>Start Address – lower display</td>
<td>AdcBuf</td>
</tr>
<tr>
<td>Acquisition Buffer Size</td>
<td>50</td>
</tr>
<tr>
<td>Display Data Size</td>
<td>50</td>
</tr>
<tr>
<td>DSP Data Type</td>
<td>32-bit signed integer</td>
</tr>
<tr>
<td>Q-value</td>
<td>24</td>
</tr>
<tr>
<td>Sampling Rate (Hz)</td>
<td>50000</td>
</tr>
<tr>
<td>Time Display Unit</td>
<td>µs</td>
</tr>
</tbody>
</table>

Select OK to save the graph options.

18. The graphical display should show the generated IQmath FIR filtered 2 kHz, 25% duty cycle symmetric PWM waveform in the upper display and the unfiltered waveform generated in the previous lab exercise in the lower display. The results should be the same as the previous lab, but we are using a SWI rather than a HWI.

19. Fully halt the DSP (real-time mode) by using Debug → Halt and then unchecking the “Real-time mode”.

Add a Periodic Function

Recall that an instruction was used in the ADCINT_ISR to toggle the LED on the eZdsp™. This instruction will be moved into a periodic function that will toggle the LED at the same rate.
20. Open `DefaultIsr_9_10.c` and locate the ADCINT_ISR routine. Move the instruction used to toggle the LED to the LedBlink() function in `Main_9.c`:

```c
GpioDataRegs.GPIOF14.bit.GPIOF14 = 1; // Toggle the pin
```

21. In the configuration file `Lab.cdb` we need to add and setup the LedBlink_PRD. Open `Lab.cdb` and click on the plus sign (+) to the left of Scheduling. Right click on PRD – Periodic Function Manger and select Insert PRD. PRD0 will be added. Right-click on it and rename it to LedBlink_PRD.

22. Select the Properties for LedBlink_PRD and type `_LedBlink` (with a leading underscore) in the function field. This tells DSP/BIOS to run the LedBlink() function when it executes the LedBlink_PRD periodic function object.

Next, in the period (ticks) field type 500. Click OK. Close the configuration file and click YES to save changes.

**Build and Load**

23. Click the “Build” button to rebuild and load the project.

24. Reset the DSP.

**Run the Code – LedBlink**

25. Run the code and check to see if the LED on the eZdsp™ is blinking. When done, halt the code. If you would like, experiment with different period (tick) values and notice that the blink rate changes.

**DSP/BIOS – Real-time Analysis**

The DSP/BIOS analysis tools complement the CCS environment by enabling real-time program analysis of a DSP/BIOS application. You can visually monitor a DSP application as it runs with essentially no impact on the application’s real-time performance. In CCS, the DSP/BIOS analysis tools are found on the DSP/BIOS menu. Unlike traditional debugging, which is external to the executing program, DSP/BIOS program analysis requires that the target program be instrumented with analysis code. By using DSP/BIOS APIs and objects, developers automatically instrument the target for capturing and uploading real-time information to CCS using these tools.

26. Open the **CPU load graph**. On the menu bar click:

```
DSP/BIOS → CPU Load Graph
```

The CPU load graph displays the percentage of available CPU computing horsepower that the application is consuming. The CPU may be running ISRs, software interrupts, periodic functions, performing I/O with the host, or running any user routine. When the CPU is not executing user code, it will be idle (in the DSP/BIOS idle thread). You will notice that we are consuming about 50% of the CPU with our lab code.
27. Next, open the execution graph. On the menu bar click:

DSP/BIOS ➔ Execution Graph

The execution graph is a special graph used to display information about different threads in the system and when they occur relative to the other events. This graph is not based on time, but the activity of events (i.e., when an event happens, such as a SWI or periodic function begins execution). You can enable or disable logging for each of the object types using the RTA Control Panel (DSP/BIOS ➔ RTA Control Panel). Note that the execution graph simply records DSP/BIOS CLK events along with other system events (the DSP/BIOS clock periodically triggers the DSP/BIOS scheduler). As a result, the time scale on the execution graph is not linear.

28. Halt the DSP.

29. In the next few steps the Log Event Manager will be setup to capture an event in real-time while the program executes. We will be using Log_printf() to write to a log buffer. The LOG_printf() function is a very efficient means of sending a message from the code to the CCS display. Unlike an ordinary C-language printf(), which can consume several hundred DSP cycles to format the data on the DSP before transmission to the CCS host PC, a log_printf() transmits the raw data to the host. The host then formats the data and displays it in CCS. This consumes only 10’s of cycles rather than 100’s of cycles.

Add the following to Main_9.c just after the static local variable declaration in AdcSwi():

```c
static Uint32 AdcSwi_count=0;  // used for LOG_printf

/*** Using LOG_printf() to write to a log buffer ***/
   LOG_printf(&trace, "AdcSwi_count = %u", AdcSwi_count++);
```

30. In the configuration file Lab.cdb we need to add and setup the trace buffer. Open Lab.cdb and click on the plus sign (+) to the left of Instrumentation and again on the plus sign (+) to the left of LOG – Event Log Manager.

31. Right click on LOG – Event Log Manager and select Insert LOG. LOG0 will be added. Right-click on it and rename it to trace.

32. Select the Properties for trace and set the logtype to circular and the datatype to printf. Click OK.

33. Since the configuration file was modified, we need to rebuild the project. Click the “Build” button.

34. Reset the DSP.

35. Open the Message Log by clicking:

DSP/BIOS ➔ Message Log
The message log dialog box is displaying the number of times (count value) that AdcSwi() has executed.

36. Run the DSP.

37. Observe the operation of the various windows that are open, and the information that they are conveying in real-time.

**Note:** In this module only the basic features of DSP/BIOS and the real-time analysis tools have been used. For more information and details, please refer to the DSP/BIOS user’s manuals and other DSP/BIOS related training.

**End of Exercise**
Introduction

This module discusses various aspects of system design. Details of the emulation and analysis block along with JTAG will be discussed and the external interface will be explored. Flash memory programming and the Code Security Module will be described.

Learning Objectives

- Emulation and Analysis Block
- External Interface (XINTF)
- Flash Configuration and Memory Performance
- Flash Programming
- Code Security Module (CSM)
Module Topics

System Design .......................................................................................................................................... 10-1

Module Topics ....................................................................................................................................... 10-2

Emulation and Analysis Block .............................................................................................................. 10-3

External Interface (XINTF) .................................................................................................................. 10-7

Flash Configuration and Memory Performance .................................................................................... 10-10

Flash Programming ............................................................................................................................... 10-13

Code Security Module (CSM) .............................................................................................................. 10-15

Lab 10: Programming the Flash ........................................................................................................... 10-19
Emulation and Analysis Block

JTAG Emulation System
(based on IEEE 1149.1 Standard)

Some Available Emulators
- Spectrum Digital XDS510PP+
- Spectrum Digital XDS510USB
- Spectrum Digital PCI
- BlackHawk USB
- DSP Research PCI

eZdsp™ has XDS510PP+ on the board

Connections Between Emulator and Target

= If distance between device and header is greater than 6 inches
On-Chip Emulation Analysis Block: Capabilities

Two hardware analysis units can be configured to provide any one of the following advanced debug features:

<table>
<thead>
<tr>
<th>Analysis Configuration</th>
<th>Debug Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 Hardware Breakpoints</td>
<td>Halt on a specified instruction (for debugging in Flash)</td>
</tr>
<tr>
<td>2 Address Watchpoints</td>
<td>A memory location is getting corrupted. Halt the processor when any value is written to this location</td>
</tr>
<tr>
<td>1 Address Watchpoint with Data</td>
<td>Halt program execution after a specific value is written to a variable</td>
</tr>
<tr>
<td>1 Pair Chained Breakpoints</td>
<td>Halt on a specified instruction only after some other specific routine has executed</td>
</tr>
</tbody>
</table>
On-Chip Emulation Analysis Block: Hardware Breakpoints

- Symbolic or numeric address
- Mask value for specifying address ranges
- Chained breakpoint selection

On-Chip Emulation Analysis Block: Watchpoints

- Symbolic or numeric address
- Mask value for specifying address ranges
- Bus selection
- Address with Data selection
Emulation and Analysis Block

**On-Chip Emulation Analysis Block: Online Stack Overflow Detection**

- Emulation analysis registers are accessible to code as well!
- Configure a watchpoint to monitor for writes near the end of the stack
- Watchpoint triggers maskable RTOSINT interrupt
- Works with DSP/BIOS and non-DSP/BIOS
  - See TI application report SPRA820 for implementation details

![Diagram of stack overflow detection](image-url)
External Interface (XINTF)

TMS320F2812 XINTF Memory Map

F2812 External Interface (XINTF) Signals
F2812 XINTF Timings

- Five external zones: 0, 1, 2, 6, 7
- Each zone has separate read and write timings
- XREADY signal can be used to extend ACTIVE phase

**Read Timing**

- **XZCS**: DSP latches data
- **XRDX**: valid address
- **XRD**: valid data
- **XD**: SRAM valid data

**XINTF Clocking**

- Specify read timing and write timing separately, for each zone:
  - **Lead**: 0-3 XTIMCLK Cycles
  - **Active**: 1-7 XTIMCLK Cycles
  - **Trail**: 0-3 XTIMCLK Cycles

- Each zone has a X2TIMING bit that can double the timing values (both read and write affected)
XINTF Registers
(lab file: Xintf.c)

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Size (x16)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XTIMING0</td>
<td>0x00 0B20</td>
<td>2</td>
<td>XINTF Zone 0 Timing Register</td>
</tr>
<tr>
<td>XTIMING1</td>
<td>0x00 0B22</td>
<td>2</td>
<td>XINTF Zone 1 Timing Register</td>
</tr>
<tr>
<td>XTIMING2</td>
<td>0x00 0B24</td>
<td>2</td>
<td>XINTF Zone 2 Timing Register</td>
</tr>
<tr>
<td>XTIMING6</td>
<td>0x00 0B2C</td>
<td>2</td>
<td>XINTF Zone 6 Timing Register</td>
</tr>
<tr>
<td>XTIMING7</td>
<td>0x00 0B2E</td>
<td>2</td>
<td>XINTF Zone 7 Timing Register</td>
</tr>
<tr>
<td>XINTCNF2</td>
<td>0x00 0B34</td>
<td>2</td>
<td>XINTF Configuration Register</td>
</tr>
<tr>
<td>XBANK</td>
<td>0x00 0B38</td>
<td>2</td>
<td>XINTF Bank Control Register</td>
</tr>
</tbody>
</table>

- **XTIMING**x specifies read (lead, active, trail), write (lead, active, trail), X2TIMING, and use XREADY
- **XINTCNF2** selects fundamental clock XTIMCLK (for lead, active, trail), CLKMODE and CLKOFF
- **XBANK** specifies the number of XTIMCLK cycles to add between two specified zone (bank switching)

Configuring XINTF with Header Files

**XINTCNF2 example:** Timing for all zones based on XTIMCLK = SYSCLKOUT/2

```c
XintfRegs.XINTCNF2.bit.XTIMCLK = 1; // XTIMCLK = SYSCLKOUT/2
XintfRegs.XINTCNF2.bit.CLKOFF = 0; // XCLKOUT is enabled
XintfRegs.XINTCNF2.bit.CLKMODE = 1; // XCLKOUT = XTIMCLK/2
```

**Zone write and read timing example:**

```c
XintfRegs.XTIMING0.bit.XWRLEAD = 1;
XintfRegs.XTIMING0.bit.XWRACTIVE = 2;
XintfRegs.XTIMING0.bit.XWRTRAIL = 0;
XintfRegs.XTIMING0.bit.XRDLEAD = 1;
XintfRegs.XTIMING0.bit.XRDACTIVE = 3;
XintfRegs.XTIMING0.bit.XRDTRAIL = 1;
```

**Bank switching example:** Suppose the external device in zone 7 is slow getting off the bus; Add 3 additional cycles when switching from zone 7 to another zone to avoid bus contention

```c
XintfRegs.XBANK.bit.BANK = 7;
XintfRegs.XBANK.bit.BCYC = 3;
```
Flash Configuration and Memory Performance

Basic Flash Operation

- Flash is arranged in pages of 2048 bits
- Wait states are specified for consecutive accesses without a page, and random accesses across pages
- OTP has random access only
- Must specify the number of SYSCLKOUT wait-states

Reset defaults are maximum values

<table>
<thead>
<tr>
<th>FBANKWAIT @ 0x00 0A86</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 12 11 8 7 4 3 0</td>
</tr>
<tr>
<td>reserved PAGEWAIT reserved RANDWAIT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FOTPWAIT @ 0x00 0A87</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 4 3 0</td>
</tr>
<tr>
<td>reserved OTPWAIT</td>
</tr>
</tbody>
</table>

*** Refer to the F281x datasheet for detailed numbers ***

For 150 MHz, PAGEWAIT = 5, RANDWAIT = 5, OTPWAIT = 8
For 135 MHz, PAGEWAIT = 4, RANDWAIT = 4, OTPWAIT = 8

Speeding Up Code Execution in Flash:
Flash Pipelining (for code fetch only)

- Flash Pipeline Enable
  - 0 = disable (default)
  - 1 = enable

FOPT @ 0x00 0A80

15 1 0
reserved ENPIPE
### Other Flash Configuration Registers

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00 0A80</td>
<td>FOPT</td>
<td>Flash option register</td>
</tr>
<tr>
<td>0x00 0A82</td>
<td>FPWR</td>
<td>Flash power modes registers</td>
</tr>
<tr>
<td>0x00 0A83</td>
<td>FSTATUS</td>
<td>Flash status register</td>
</tr>
<tr>
<td>0x00 0A84</td>
<td>FSTDBYWAIT</td>
<td>Flash sleep to standby wait register</td>
</tr>
<tr>
<td>0x00 0A85</td>
<td>FACTIVEWAIT</td>
<td>Flash standby to active wait register</td>
</tr>
<tr>
<td>0x00 0A86</td>
<td>FBANKWAIT</td>
<td>Flash read access wait state register</td>
</tr>
<tr>
<td>0x00 0A87</td>
<td>FOTPWAIT</td>
<td>OTP read access wait state register</td>
</tr>
</tbody>
</table>

- **FPWR**: Save power by putting Flash/OTP to ‘Sleep’ or ‘Standby’ mode; Flash will automatically enter active mode if a Flash/OTP access is made.
- **FSTATUS**: Various status bits (e.g. PWR mode)
- **FSTDBYWAIT**: Specify number of cycles to wait during wake-up from sleep to standby
- **FACTIVEWAIT**: Specify number of cycles to wait during wake-up from standby to active

Defaults for these registers are often sufficient – See “TMS320F28x DSP System Control and Interrupts Reference Guide,” SPRU078, for more information.

### Code Execution Performance

- **Assume 150 MHz SYSCLKOUT, 16-bit instructions**
  - (80% of instructions are 16 bits wide – Rest are 32 bits)

**Internal RAM**: 150 MIPS true performance

Fetch up to 32-bits every cycle ➔ 1 instruction/cycle * 150 MHz = 150 MIPS

**Flash (w/ pipelining)**: 100 MIPS effective performance

RANDWAIT = 5

Fetch 64 bits every 6 cycles ➔ 4 instructions/6 cycles * 150 MHz = 100 MIPS

RPT will increase this; PC discontinuity will degrade this

**External 10ns SRAM**: 50 MIPS effective performance

XRDLEAD=1, XRDACTIVE=2, XRDTRAIL=0

Fetch 16 bits every 3 cycles ➔ 1 instruction/3 cycles * 150 MHz = 50 MIPS

RPT will increase this

- **Don’t put code in external memory!**
Data Access Performance

- Assume 150 MHz SYSCLKOUT (CPU clock)

<table>
<thead>
<tr>
<th>Memory</th>
<th>16-bit access (words/cycle)</th>
<th>32-bit access (words/cycle)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal RAM</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Flash</td>
<td>0.167</td>
<td>0.167</td>
<td>RANDWAIT = 5 Flash is read only!</td>
</tr>
<tr>
<td>Ext. SRAM (10ns)</td>
<td>0.33</td>
<td>0.167</td>
<td>XRDLEAD = 1, XRDACTIVE = 2, XRDTRAIL = 0</td>
</tr>
</tbody>
</table>

- External RAM can outperform the flash on 16-bit data reads
- Flash performance usually sufficient for most constants and tables
- For cost and power savings, and ease-of-design, use internal RAM for time-critical constants before adding external memory to your system
Flash Programming

Flash Programming Basics

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

Flash Programming Basics

- Sequence of steps for Flash programming:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Erase</td>
<td>- Set all bits to zero, then to one</td>
</tr>
<tr>
<td>2. Program</td>
<td>- Program selected bits with zero</td>
</tr>
<tr>
<td>3. Verify</td>
<td>- Verify flash contents</td>
</tr>
</tbody>
</table>

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

- Code Composer Studio Plug-in (uses JTAG) *
- Serial Flash loader from TI (uses SCI boot) *
- Gang Programmers (use GPIO boot)
  - BP Micro programmer
  - Data I/O programmer
- Build your own custom utility
  - Use a different ROM bootloader method than SCI
  - Embed flash programming into your application
  - Flash API algorithms provided by TI

* Available from TI web at www.ti.com

Code Composer Studio Flash Plug-In

[Image of Code Composer Studio Flash Plug-In interface]
Code Security Module (CSM)

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Memory Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00 8000</td>
<td>LO SARAM (4K)</td>
</tr>
<tr>
<td>0x00 9000</td>
<td>L1 SARAM (4K)</td>
</tr>
<tr>
<td>0x00 A000</td>
<td>reserved</td>
</tr>
<tr>
<td>0x3D 7800</td>
<td>OTP (1K)</td>
</tr>
<tr>
<td>0x3D 7C00</td>
<td>reserved</td>
</tr>
<tr>
<td>0x3D 8000</td>
<td>FLASH (128K)</td>
</tr>
</tbody>
</table>

- Data reads from restricted memory are only allowed for code running from restricted memory

- All other data read accesses are blocked:
  - JTAG emulator/debugger, ROM bootloader, code running in external memory or unrestricted internal memory

CSM Password

- 128-bit user defined password is stored in Flash

- 128-bit Key Register used to lock and unlock the device
  - Mapped in memory space 0x00 0AE0 – 0x00 0AE7
  - Register “EALLOW” protected
### CSM Registers

**Key Registers** – accessible by user; EALLOW protected

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Reset Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00 0AE0</td>
<td>KEY0</td>
<td>0xFFFF</td>
<td>Low word of 128-bit Key register</td>
</tr>
<tr>
<td>0x00 0AE1</td>
<td>KEY1</td>
<td>0xFFFF</td>
<td>2nd word of 128-bit Key register</td>
</tr>
<tr>
<td>0x00 0AE2</td>
<td>KEY2</td>
<td>0xFFFF</td>
<td>3rd word of 128-bit Key register</td>
</tr>
<tr>
<td>0x00 0AE3</td>
<td>KEY3</td>
<td>0xFFFF</td>
<td>4th word of 128-bit Key register</td>
</tr>
<tr>
<td>0x00 0AE4</td>
<td>KEY4</td>
<td>0xFFFF</td>
<td>5th word of 128-bit Key register</td>
</tr>
<tr>
<td>0x00 0AE5</td>
<td>KEY5</td>
<td>0xFFFF</td>
<td>6th word of 128-bit Key register</td>
</tr>
<tr>
<td>0x00 0AE6</td>
<td>KEY6</td>
<td>0xFFFF</td>
<td>7th word of 128-bit Key register</td>
</tr>
<tr>
<td>0x00 0AE7</td>
<td>KEY7</td>
<td>0xFFFF</td>
<td>High word of 128-bit Key register</td>
</tr>
<tr>
<td>0x00 0AEF</td>
<td>CSMSCR</td>
<td>0xFFFF</td>
<td>CSM status and control register</td>
</tr>
</tbody>
</table>

**PWL in memory** – reserved for passwords only

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Reset Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3F 7FF8</td>
<td>PWL0</td>
<td>user defined</td>
<td>Low word of 128-bit password</td>
</tr>
<tr>
<td>0x3F 7FF9</td>
<td>PWL1</td>
<td>user defined</td>
<td>2nd word of 128-bit password</td>
</tr>
<tr>
<td>0x3F 7FFA</td>
<td>PWL2</td>
<td>user defined</td>
<td>3rd word of 128-bit password</td>
</tr>
<tr>
<td>0x3F 7FFB</td>
<td>PWL3</td>
<td>user defined</td>
<td>4th word of 128-bit password</td>
</tr>
<tr>
<td>0x3F 7FFC</td>
<td>PWL4</td>
<td>user defined</td>
<td>5th word of 128-bit password</td>
</tr>
<tr>
<td>0x3F 7FFD</td>
<td>PWL5</td>
<td>user defined</td>
<td>6th word of 128-bit password</td>
</tr>
<tr>
<td>0x3F 7FFE</td>
<td>PWL6</td>
<td>user defined</td>
<td>7th word of 128-bit password</td>
</tr>
<tr>
<td>0x3F 7FFF</td>
<td>PWL7</td>
<td>user defined</td>
<td>High word of 128-bit password</td>
</tr>
</tbody>
</table>

### Locking and Unlocking the CSM

- **The CSM is locked at power-up and reset**
- **To unlock the CSM:**
  - Perform a dummy read of each password in the Flash
  - Write the correct passwords to the key registers
- **New Flash Devices (PWL are all 0xFFFF):**
  - When all passwords are 0xFFFF – only a read of the PWL is required to bring the device into unlocked mode
CSM Caveats

- Never program all the PWL’s as 0x0000
  - *Doing so will permanently lock the CSM*
- Flash addresses 0x3F7F80 to 0x3F7FF5, inclusive, must be programmed to 0x0000 to securely lock the CSM
- Remember that code running in unsecured RAM cannot access data in secured memory
  - Don’t link the stack to secured RAM if you have any code that runs from unsecured RAM
- Do not embed the passwords in your code!
  - Generally, the CSM is unlocked only for debug
  - Code Composer Studio can do the unlocking

CSM Password Match Flow

- Start
- Flash device secure after reset or runtime
  - Do dummy read of PWL 0x3F 7FF8 – 0x3F 7FFF
- Is PWL = all 0s?
  - Yes, Device permanently locked
    - CPU access is limited – device cannot be debugged or reprogrammed
  - No
- Is PWL = all Fs?
  - Yes, Correct password?
    - Yes, Device unlocked
      - User can access on-chip secure memory
    - No
  - No
- Write password to KEY registers 0x00 0AE0 – 0x00 0AE7
  - (EALLOW) protected
- Is PWL = all Fs?
  - Yes, Correct password?
    - Yes, Device unlocked
      - User can access on-chip secure memory
    - No
  - No
Unlocking the CSM:

```c
volatile int *PWL = &CsmPwl.PSWD0; // Pointer to PWL register file
volatile int i, tmp;
for (i = 0; i<8; i++) tmp = *PWL++; // Dummy reads of PWL locations
asm (" EALLOW"); // KEY regs are EALLOW protected
CsmRegs.KEY0 = PASSWORD0; // Write the passwords
CsmRegs.KEY1 = PASSWORD0 // to the Key registers
CsmRegs.KEY2 = PASSWORD2;
CsmRegs.KEY3 = PASSWORD3;
CsmRegs.KEY4 = PASSWORD4;
CsmRegs.KEY5 = PASSWORD5;
CsmRegs.KEY6 = PASSWORD6;
CsmRegs.KEY7 = PASSWORD7;
asm (" EDIS");
```

Locking the CSM:

```c
asm (" EALLOW"); // CSMSCR reg is EALLOW protected
CsmRegs.CSMSCR.bit.FORCESEC = 1; // Set FORCESEC bit
asm (" EDIS");
```
Lab 10: Programming the Flash

Objective

The objective of this lab is to use the techniques discussed in module 10 and program the on-chip flash memory. The TMS320F2812 device has been designed for standalone operation in an embedded system. Using the on-chip flash eliminates the need for external non-volatile memory or a host processor from which to bootload. In this lab, the steps required to properly configure the software for execution from internal flash memory will be covered.

Lab 10: Programming the Flash

Objective:

- Program system into Flash Memory
- Learn use of CCS Flash Plug-in
- DO NOT PROGRAM PASSWORDS

Procedure

Project File

Note: LAB10 files have been provided as a starting point for the lab and need to be completed. DO NOT copy files from a previous lab.

1. A project named Lab10.pjt has been created for this lab. Open the project by clicking on Project → Open... and look in C:\C28x\LABS\LAB10. All Build Options have been configured like the previous lab. The files used in this lab are:

Note: Due to a bug in CCS and DSP/BIOS, the configuration file name (.cdb) and the project output (.out) and map (.map) names must be the same. Therefore, in this module the Build Options output and map file names will be lab.out and map.out, respectively.
Link Initialized Sections to Flash

Initialized sections, such as code and constants, must contain valid values at device power-up. For a stand-alone embedded system with the F2812 device, these initialized sections must be linked to the on-chip flash memory. Note that a stand-alone embedded system must operate without an emulator or debugger in use, and no host processor is used to perform bootloading.

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

2. This step assigns the RUN address of those sections that need to run from flash. Using the memory section manager in the DSP/BIOS configuration tool (Lab.cdb) link the following sections to on-chip flash memory:

<table>
<thead>
<tr>
<th>DSP/BIOS Data tab</th>
<th>DSP/BIOS Code tab</th>
<th>Compiler Sections tab</th>
</tr>
</thead>
<tbody>
<tr>
<td>.gbinit</td>
<td>.bios</td>
<td>.text</td>
</tr>
<tr>
<td></td>
<td>.sysinit</td>
<td>.switch</td>
</tr>
<tr>
<td></td>
<td>.hwi</td>
<td>.cinit</td>
</tr>
<tr>
<td></td>
<td>.rtdx_text</td>
<td>.pinit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.econst/.const</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.data</td>
</tr>
</tbody>
</table>

3. This step assigns the LOAD address of those sections that need to load to flash. Again using the memory section manager in the DSP/BIOS configuration tool (Lab.cdb), select the **Load Address tab** and check the "Specify Separate Load Addresses" box. Then set all entries to the flash memory block.
Lab 10: Programming the Flash

Copying .hwi_vec Section from Flash to RAM

The DSP/BIOS .hwi_vec section contains the interrupt vectors. This section must be loaded to flash (load address) but run from RAM (run address). The code that performs this copy is located in InitPieCtrl(). The DSP/BIOS configuration tool generates global symbols that can be accessed by code in order to determine the load address, run address, and length of the .hwi_vec section. The C-compiler runtime support library contains a memory copy function called memcpy() which will be used to perform the copy.

4. Open and inspect InitPieCtrl() (in PieCtrl_10.c file). Notice the memcpy() function and the symbols used to initialize (copy) the .hwi_vec section.

Copying the .trcdata Section from Flash to RAM

The DSP/BIOS .trcdata section is used by CCS and DSP/BIOS for certain real-time debugging features. This section must be loaded to flash (load address) but run from RAM (run address). The DSP/BIOS configuration tool generates global symbols that can be accessed by code in order to determine the load address, run address, and length of the .trcdata section. The memory copy function memcpy() will again be used to perform the copy.

The copying of .trcdata must be performed prior to main(). This is because DSP/BIOS modifies the contents of .trcdata during DSP/BIOS initialization, which also occurs prior to main(). The DSP/BIOS configuration tool provides a user initialization function which will be used to perform the .trcdata section copy prior to both main() and DSP/BIOS initialization.

5. Open and inspect the file Main_10.c. Notice that the function UserInit() is used to copy the .trcdata section from its load address to its run address before main().

6. Open the DSP/BIOS configuration file (Lab.cdb) and select the Properties for the Global Settings. Check the box “Call User Init Function” and enter the UserInit() function name with a leading underscore: _UserInit.

Initializing the Flash Control Registers

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

7. Add Flash.c to the project.

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

9. Since the DSP/BIOS configuration tool does not know about user defined sections, the “secureRamFuncs” section will be linked using the user linker command file User_10.cmd. Open and inspect User_10.cmd. The “secureRamFuncs” will load to flash (load address) but will run from L1SARAM (run address). Also notice that
the linker has been asked to generate symbols for the load start, load end, and run start addresses.

While not a requirement from a DSP hardware perspective (since the C28x DSP has a unified memory architecture). Code Composer Studio generally prefers code to be linked to program space (and data to be linked to data space). Therefore, notice that for the L1SARAM memory we are linking “secureRamFuncs” to, we are specifying “PAGE = 0” (which is program space). The L1SARAM memory is currently defined in data space in the Lab.cdb, but in the next step we will redefine it to exist in the program space.

10. Using the DSP/BIOS configuration tool (Lab.cdb) modify the entry for L1SARAM so that it is defined in the program space (code).

11. Open and inspect Main_10.c. Notice that the memory copy function memcpy() is being used to copy the section “secureRamFuncs”, which contains the initialization function for the flash control registers.

12. Add a line of code to main() to call the InitFlash() function. There are no passed parameters or return values. You just type

    InitFlash();

    at the desired spot in main().

**Code Security Module and Passwords**

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

The CSM module also requires programming values of 0x0000 into flash addresses 0x3F7F80 through 0x3F7FFF in order to properly secure the CSM. Both tasks will be accomplished using a simple assembly language program Passwords.asm.

13. Add Passwords.asm to your CCS project.

14. Open and inspect Passwords.asm. This file specifies the desired password values **(DO NOT CHANGE THE VALUES FROM 0xFFFF)** and places them in an initialized section named “passwords”. It also creates an initialized section named “csm_rsvd” which contains all 0x0000 values for locations 0x3F7F80 to 0x3F7FFF (length of 0x76).
15. Open User_10.cmd and notice that the initialized sections for “passwords” and “csm_rsvd” are linked to memories named PASSWORDS and CSM_RSVD, respectively.

16. Using the DSP/BIOS configuration tool (Lab.cdb) define memory blocks for PASSWORDS and CSM_RSVD. You will need to setup the MEM Properties for each memory block with the proper base address and length. Set the space to code for both memory blocks. (Uncheck the “create a heap in this memory” box for each block). You may also need to modify the existing flash memory block to avoid conflicts.

**Executing from Flash after Reset**

The F2812 device contains a ROM bootloader that will transfer code execution to the flash after reset. When the boot mode selection pins are set for “Jump to Flash” mode, the bootloader will branch to the instruction located at address 0x3F7FF6 in the flash. An instruction that branches to the beginning of your program needs to be placed at this address. Note that the CSM passwords begin at address 0x3F7FF8. There are exactly two words available to hold this branch instruction, and not coincidentally, a long branch instruction “LB” in assembly code occupies exactly two words. Generally, the branch instruction will branch to the start of the C-environment initialization routine located in the C-compiler runtime support library. The entry symbol for this routine is _c_int00. Recall that C code cannot be executed until this setup routine is run. Therefore, assembly code must be used for the branch. We are using the assembly code file named CodeStartBranch.asm.

17. Open and inspect CodeStartBranch.asm. This file creates an initialized section named “codestart” that contains a long branch to the C-environment setup routine. This section needs to be placed in memory using the DSP/BIOS configuration tool.

18. Using the DSP/BIOS configuration tool (Lab.cdb) define a memory space named BEGIN_FLASH.

19. Setup the MEM Properties with the proper base address, length, and space. (Uncheck the “create a heap in this memory” box). Be sure to avoid memory section conflicts.

20. In the earlier lab exercises, the section “codestart” was directed to the memory named BEGIN_H0. Open and modify User_10.cmd so that the section “codestart” will be directed to BEGIN_FLASH.

21. The eZdsp™ board needs to be configured for “Jump to Flash” bootmode. Move jumper JP7 to position 1-2 to accomplish this. This jumper controls the pullup/down resistor on the GPIOF4 pin, which is one of the pins sampled by the bootloader to determine the bootmode. GPIOF4 alone is sufficient to configure “Jump to Flash” bootmode (see the TMS320F28x DSP Boot ROM Reference Guide, and also the eZdsp F2812 Technical Reference, for more information).
Build – Lab.out

22. At this point we need to build the project, but not have CCS automatically load it since CCS cannot load code into the flash! (the flash must be programmed). On the menu bar click: Option → Customize... and select the “Program Load Options” tab. Uncheck “Load Program After Build”, then click OK.

23. Click the “Build” button to generate the Lab.out file to be used with the CCS Flash Plug-in.

CCS Flash Plug-in

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

   Tools → F28xx On-Chip Flash Programmer

25. Notice that the eZdsp™ board uses a 30 MHz oscillator (located on the board near LEDs DS1 and DS2). Confirm the “Clock Configuration” in the upper left corner has the OSCCLK set to 30 MHz and the PLLCR value is set to 10. Recall that the PLL is divided by two, which gives a SYSCLKOUT of 150 MHz.

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

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

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

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

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

Running the Code – Using CCS

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

   File → Load Symbols → Load Symbols Only...

   and select Lab.out in the Debug folder.

32. Reset the DSP. The program counter should now be at 0x3FFC00, which is the start of the bootloader in the Boot ROM.
33. Single-Step through the bootloader code until you arrive at the beginning of the codestart section in the `CodeStartBranch.asm` file. (Be patient, it will take about 55 single-steps). Notice that we have placed some code in `CodeStartBranch.asm` to give an option to first disable the watchdog, if selected.

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

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

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

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


38. Disconnect the emulator from the eZdsp™ board.

39. Remove the power from the board.

40. Re-connect the power to the board.

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

**End of Exercise**
Lab 10: Programming the Flash
Communications

Introduction

The TMS320C28x contains features that allow several methods of communication and data exchange between the C28x and other devices. Many of the most commonly used communications techniques are presented in this module.

The intent of this module is not 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

- Serial Peripheral Interface (SPI)
- Serial Communication Interface (SCI)
- Multi-Channel Buffered Serial Port (McBSP)
- Enhanced Controller Area Network (eCAN)
Module Topics

Communications................................................................................................................11-1

Module Topics................................................................................................................11-2

Communications Techniques ...............................................................................................11-3

Serial Peripheral Interface (SPI) ............................................................................................11-4
  SPI Registers .....................................................................................................................11-7

Serial Communications Interface (SCI) ................................................................................11-11
  Multiprocessor Wake-Up Modes ......................................................................................11-14

Multi-Channel Buffered Serial Port (McBSP) ......................................................................11-17

Enhanced Controller Area Network (eCAN) ......................................................................11-20
  CAN Bus and Node ............................................................................................................11-21
  Principles of Operation ........................................................................................................11-22
  Message Format and Block Diagram ..................................................................................11-23
Communications Techniques

Several methods of implementing a TMS320C28x communications system are possible. The method selected for a particular design should reflect the method that meets the required data rate at the lowest cost. Various categories of interface are available and are summarized in the learning objective slide. Each will be described in this module.

Synchronous vs. Asynchronous

- **SPI (synchronous)**
  - Short distances (on-board)
  - High data rate
  - Explicit clock

- **SCI (asynchronous)**
  - Longer distances
  - Lower data rate ($\approx 1/8$ of SPI)
  - Implied clock (clk/data mixed)
  - Economical with reasonable performance

Serial ports provide a simple, hardware-efficient means of high-level communication between devices. Like the GPIO pins, they may be used in stand-alone or multiprocessing systems.

In a multiprocessing system, they are an excellent choice when both devices have an available serial port and the data rate requirement is relatively low. Serial interface is even more desirable when the devices are physically distant from each other because the inherently low number of wires provides a simpler interconnection.

Serial ports require separate lines to implement, and they do not interfere in any way with the data and address lines of the processor. The only overhead they require is to read/write new words from/to the ports as each word is received/transmitted. This process can be performed as a short interrupt service routine under hardware control, requiring only a few cycles to maintain.

The C28x family of devices have both a synchronous and asynchronous serial ports. Detailed features and operation will be described next.
Serial Peripheral Interface (SPI)

The SPI module is a synchronous serial I/O port that shifts a serial bit stream of variable length and data rate between the C28x and other peripheral devices. During data transfers, one SPI device must be configured as the transfer MASTER, and all other devices configured as SLAVES. The master drives the transfer clock signal for all SLAVES on the bus. SPI communications can be implemented in any of three different modes:

- MASTER sends data, SLAVES send dummy data
- MASTER sends data, one SLAVE sends data
- MASTER sends dummy data, one SLAVE sends data

In its simplest form, the SPI can be thought of as a programmable shift register. Data is shifted in and out of the SPI through the SPIDAT register. Data to be transmitted is written directly to the SPIDAT register, and received data is latched into the SPIBUF register for reading by the CPU. This allows for double-buffered receive operation, in that the CPU need not read the current received data from SPIBUF before a new receive operation can be started. However, the CPU must read SPIBUF before the new operation is complete 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 Data Flow**

- Simultaneous transmits and receive
- SPI Master provides the clock signal

![SPI Data Flow Diagram]
SPI Transmit / Receive Sequence

1. Slave writes data to be sent to its shift register (SPIDAT)
2. Master writes data to be sent to its shift register (SPIDAT or SPITXBUF)
3. Completing Step 2 automatically starts SPICLK signal of the Master
4. MSB of the Master’s shift register (SPIDAT) is shifted out, and LSB of the Slave’s shift register (SPIDAT) is loaded
5. Step 4 is repeated until specified number of bits are transmitted
6. SPIDAT register is copied to SPIRXBUF register
7. SPI INT Flag bit is set to 1
8. An interrupt is asserted if SPI INT ENA bit is set to 1
9. If data is in SPITXBUF (either Slave or Master), it is loaded into SPIDAT and transmission starts again as soon as the Master’s SPIDAT is loaded
Since data is shifted out of the SPIDAT register MSB first, transmission characters of less than 16 bits must be left-justified by the CPU software prior to be written to SPIDAT.

Received data is shifted into SPIDAT from the left, MSB first. However, the entire sixteen bits of SPIDAT is copied into SPIBUF after the character transmission is complete such that received characters of less than 16 bits will be right-justified in SPIBUF. The non-utilized higher significance bits must be masked-off by the CPU software when it interprets the character. For example, a 9 bit character transmission would require masking-off the 7 MSB’s.

**SPI Data Character Justification**

- **Programmable data length of 1 to 16 bits**
  - Transmitted data of less than 16 bits must be left justified
    - MSB transmitted first
  - Received data of less than 16 bits are right justified
  - User software must mask-off unused MSB’s
### SPI Registers

#### SPI-A Configuration Control Register

**SPICCR** @ 0x007040

<table>
<thead>
<tr>
<th>15-8</th>
<th>7</th>
<th>6</th>
<th>5-4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>reserved</td>
<td></td>
<td></td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **SPI CHAR.3-0**
  - character length = number + 1
  - e.g. 0000b ⇒ length = 1
  - 1111b ⇒ length = 16

- **CLOCK POLARITY**
  - 0 = rising edge data transfer
  - 1 = falling edge data transfer

- **SPI SW RESET**
  - 0 = SPI flags reset
  - 1 = normal operation

#### SPI-A Operation Control Register

**SPICTL** @ 0x007041

<table>
<thead>
<tr>
<th>15-5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **OVERRUN INT ENABLE**
  - 0 = disabled
  - 1 = enabled

- **MASTER/SLAVE**
  - 0 = slave
  - 1 = master

- **SPI INT ENABLE**
  - 0 = disabled
  - 1 = enabled

- **CLOCK PHASE**
  - 0 = no CLK delay
  - 1 = CLK delayed 1/2 cycle

- **TALK**
  - 0 = transmission disabled, output pin hi-Z’d
  - 1 = transmission enabled
SPI-A Baud Rate Register
SPIBRR @ 0x007044

Need to set this only when in master mode!

15-7 6-0
reserved SPI BIT RATE

\[
\text{SPICLK signal} = \begin{cases} 
\frac{\text{LSPCLK}}{(\text{SPIBRR} + 1)}, & \text{SPIBRR = 3 to 127} \\
\frac{\text{LSPCLK}}{4}, & \text{SPIBRR = 0, 1, or 2}
\end{cases}
\]

**Baud Rate Determination:** The Master specifies the communication baud rate using its baud rate register (SPIBRR.6-0):

- For SPIBRR = 3 to 127:
  \[
  \text{SPI Baud Rate} = \frac{\text{CLKOUT}}{(\text{SPIBRR} + 1)} \text{ bits/sec}
  \]

- For SPIBRR = 0, 1, or 2:
  \[
  \text{SPI Baud Rate} = \frac{\text{CLKOUT}}{4} \text{ bits/sec}
  \]

From the above equations, one can compute

Maximum data rate = 37.5 Mbps @ 150 MHz

**Character Length Determination:** The Master and Slave must be configured for the same transmission character length. This is done with bits 0, 1, 2 and 3 of the configuration control register (SPICCR.3-0). These four bits produce a binary number, from which the character length is computed as binary + 1 (e.g. SPICCR.3-0 = 0010 gives a character length of 3).
Serial Peripheral Interface (SPI)

**SPI-A Status Register**
SPISTS @ 0x007042

- **TX BUF FULL** (read only)
  - Set to 1 when char written to SPI TXBUF
  - Cleared when char in SPIDAT
- **SPI INT FLAG** (read only)
  - Set to 1 when transfer completed
  - Interrupt requested if SPI INT ENA bit set (SPICTL.0)
  - Cleared by reading SPIBRXUF
- **RECEIVER OVERRUN** (read/clear only)
  - Set to 1 if next reception completes before SPI RXBUF read
  - Interrupt requested if OVERRUN INT ENA bit set (SPICTL.4)
  - Cleared by writing a 1

**SPI-A FIFO Transmit Register**
SPIFFTX @ 0x00704A

- **SPI FIFO Enhancements**
  - 0 = disable
  - 1 = enable
- **TX FIFO Reset**
  - 0 = reset (pointer to 0)
  - 1 = enable operation
- **TX FIFO Status (read-only)**
  - 00000 = TX FIFO empty
  - 00001 = TX FIFO has 1 word
  - 00010 = TX FIFO has 2 words
  - 00011 = TX FIFO has 3 words
  - ... = TX FIFO has 8 words
  - 10000 = TX FIFO has 16 words
- **TX FIFO Interrupt Flag (read-only)**
  - 0 = not occurred
  - 1 = occurred
- **TX FIFO Interrupt Clear**
  - 0 = no effect
  - 1 = clear
- **TX FIFO Interrupt Enable**
  - 0 = disable
  - 1 = enable
- **TX FIFO Interrupt Level**
  - Interrupt when TXFFST4-0 and TXFFIL4-0 match
SPI-A FIFO Receive Register  
SPIFFRX @ 0x00704B

- RX FIFO Overflow Flag (read-only)  
  0 = no overflow  
  1 = overflow

- RX FIFO Overflow Flag Clear  
  0 = no effect  
  1 = clear

- RX FIFO Status (read-only)  
  00000 = RX FIFO empty  
  00001 = RX FIFO has 1 word  
  00010 = RX FIFO has 2 words  
  00011 = RX FIFO has 3 words  
  ...  
  10000 = RX FIFO has 16 words

- RX FIFO Reset  
  0 = reset (pointer to 0)  
  1 = enable operation

- RX FIFO Interrupt Flag (read-only)  
  0 = not occurred  
  1 = occurred

- RX FIFO Interrupt Flag Clear  
  0 = no effect  
  1 = clear

- RX FIFO Interrupt (on match) Enable  
  0 = disable  
  1 = enable

- RX FIFO Interrupt Level
  Interrupt when RXFFST4-0 and RXFFIL4-0 match

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
  - C28x provides clock signal in master mode

- Data length programmable from 1-16 bits

- 125 different programmable baud rates
Serial Communications Interface (SCI)

The SCI module is a serial I/O port that permits Asynchronous communication between the C28x and other peripheral devices. The SCI transmit and receive registers are both double-buffered to prevent data collisions and allow for efficient CPU usage. In addition, the C28x SCI is a full duplex interface which provides for simultaneous data transmit and receive. Parity checking and data formatting is also designed to be done by the port hardware, further reducing software overhead.
Serial Communications Interface (SCI)

**SCI-A Programmable Data Format**

NRZ (non-return to zero) format

<table>
<thead>
<tr>
<th>Start</th>
<th>LSB</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>MSB</th>
<th>Addr/Data</th>
<th>Parity</th>
<th>Stop 1</th>
<th>Stop 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This bit present only in Address-bit mode

**Communications Control Register (SCICCR) – 0x007050**

<table>
<thead>
<tr>
<th>STOP BITS</th>
<th>EVEN/ODD PARITY</th>
<th>LOOP BACK ENABLE</th>
<th>ADDR/IDLE MODE</th>
<th>SCI CHAR2</th>
<th>SCI CHAR1</th>
<th>SCI CHAR0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 = 1 Stop bit</td>
<td>0 = Disabled</td>
<td>0 = Idle-line mode</td>
<td>0 = Odd</td>
<td>0 = Disabled</td>
<td>0 = Addr-bit mode</td>
<td>0 = Even</td>
</tr>
<tr>
<td>1 = 2 Stop bits</td>
<td>1 = Enabled</td>
<td>1 = Addr-bit mode</td>
<td>1 = Even</td>
<td>1 = Enabled</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

# of data bits = (binary + 1)
e.g. 110b gives 7 data bits

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-A Baud Rate

\[
\text{SCI baud rate} = \begin{cases} 
\frac{\text{LSPCLK}}{(\text{BRR} + 1) \times 8}, & \text{BRR} = 1 \text{ to } 65535 \\
\frac{\text{LSPCLK}}{16}, & \text{BRR} = 0
\end{cases}
\]

Baud-Select MSbyte Register (SCIHBAUD) – 0x007052

Baud-Select LSbyte Register (SCILBAUD) – 0x007053

[SCI-B Baud-Select MSbyte Register (SCIHBAUD) – 0x007752]
[SCI-B Baud-Select LSbyte Register (SCILBAUD) – 0x007753]
**Serial Communications Interface (SCI)**

**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:
  \[
  \text{SCI Baud Rate} = \frac{CLKOUT}{16} \text{ bits/sec}
  \]

Max data rate = 9.3 Mbps @ 150 MHz

Note that the CLKOUT for the SCI module is one-half the CPU clock rate.

**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
Serial Communications Interface (SCI)

**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

```
<table>
<thead>
<tr>
<th>SCIRXD/SCITXD</th>
<th>1st Data</th>
<th>Addr</th>
<th>Data</th>
<th>1st Data</th>
<th>Last Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Idle periods</td>
<td></td>
<td>Addr</td>
<td></td>
<td>1st data</td>
<td></td>
</tr>
<tr>
<td>of less than</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block of Frames</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Addr</td>
<td></td>
<td>1st data</td>
<td></td>
</tr>
<tr>
<td>Idle Period</td>
<td></td>
<td>Addr</td>
<td></td>
<td>1st data</td>
<td></td>
</tr>
<tr>
<td>10 bits or</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>greater</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**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

```
<table>
<thead>
<tr>
<th>SCIRXD/SCITXD</th>
<th>1st Data</th>
<th>Addr</th>
<th>Data</th>
<th>1st Data</th>
<th>Last Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Addr</td>
<td></td>
<td>1st data</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>First frame</td>
<td></td>
<td>Addr</td>
<td></td>
<td>1st data</td>
<td></td>
</tr>
<tr>
<td>within block</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>is Address.</td>
<td></td>
<td>Addr</td>
<td></td>
<td>1st data</td>
<td></td>
</tr>
<tr>
<td>ADDR/DATA bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set to 1</td>
<td></td>
<td>Addr</td>
<td></td>
<td>1st data</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Addr</td>
<td></td>
<td>1st data</td>
<td></td>
</tr>
<tr>
<td>no additional</td>
<td></td>
<td>Addr</td>
<td></td>
<td>1st data</td>
<td></td>
</tr>
<tr>
<td>idle bits</td>
<td></td>
<td>Addr</td>
<td></td>
<td>1st data</td>
<td></td>
</tr>
<tr>
<td>needed</td>
<td></td>
<td>Addr</td>
<td></td>
<td>1st data</td>
<td></td>
</tr>
<tr>
<td>beyond stop</td>
<td></td>
<td>Addr</td>
<td></td>
<td>1st data</td>
<td></td>
</tr>
<tr>
<td>bits</td>
<td></td>
<td>Addr</td>
<td></td>
<td>1st data</td>
<td></td>
</tr>
</tbody>
</table>
```
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
- 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
Multi-Channel Buffered Serial Port (McBSP)

**McBSP Block Diagram**

```
TX FIFO_15  TX FIFO_0
DXR2 TX Buffer  DXR1 TX Buffer

TX FIFO_15  TX FIFO_0
DX FIFO_15  DX FIFO_0

RSR2  RSR1
16  16

RBR2 Register  RBR1 Register
16  16

DRR2 RX Buffer  DRR1 RX Buffer
RX FIFO_15  RX FIFO_0
RX FIFO_15  RX FIFO_0

FSX  CLKX  DX  DR  CLKR  FSR
```

**Definition: Bit and Word**

- “Bit” - one data bit per SP clock period
- “Word” or “channel” contains number of bits (8, 12, 16, 20, 24, 32)
**Multi-Channel Buffered Serial Port (McBSP)**

**Definition: Word and Frame**

- "Frame" - contains one or multiple words
- Number of words per frame: 1-128

![Diagram of Word and Frame](image)

**Multi-Channel Selection**

- Allows multiple channels (words) to be independently selected for transmit and receive (e.g. only enable Ch0, 5, 27 for receive, then process via CPU)
- The McBSP keeps time sync with all channels, but only "listens" or "talks" if the specific channel is enabled (reduces processing/bus overhead)
- Multi-channel mode controlled primarily via two registers:
  - Multi-channel Control Reg (enables Mc-mode)
  - Rec/Xmt Channel Enable Regs (enable/disable channels)
- Up to 128 channels can be enabled/disabled

![Diagram of Multi-Channel Selection](image)
McBSP Summary

- Direct Interface to codecs and other serial devices
- Full-duplex communication
- Independent clocking and framing for transmit and receive
- Internal or external clock and frame sync
- Data size of 8, 12, 16, 20, 24, or 32 bits
- 16 level 32-bit FIFO for transmit data
- 16 level 32-bit FIFO for receive data
- TDM mode - up to 128 channels
- SPI mode

75 Mbits/sec maximum data rate.
Enhanced Controller Area Network (eCAN)

Controller Area Network (CAN)
A Multi-Master Serial Bus System

- CAN 2.0B Standard
- High speed (up to 1 Mbps)
- Add a node without disturbing the bus (number of nodes not limited by protocol)
- Less wires (lower cost, less maintenance, and more reliable)
- Redundant error checking (high reliability)
- No node addressing (message identifiers)
- Broadcast based signaling

CAN does not use physical addresses to address stations. Each message is sent with an identifier that is recognized by the different nodes. The identifier has two functions – it is used for message filtering and for message priority. The identifier determines if a transmitted message will be received by CAN modules and determines the priority of the message when two or more nodes want to transmit at the same time.
CAN Bus and Node

**CAN Bus**

- Two wire differential bus (usually twisted pair)
- Max. bus length depend on transmission rate
  - 40 meters @ 1 Mbps

The 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.

**CAN Node**

Wired-AND Bus Connection

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.
**Principles of Operation**

- Data messages transmitted are identifier based, not address based.
- Content of message is labeled by an identifier that is unique throughout the network.
  - (e.g. rpm, temperature, position, pressure, etc.)
- All nodes on network receive the message and each performs an acceptance test on the identifier.
- If message is relevant, it is processed (received); otherwise it is ignored.
- Unique identifier also determines the priority of the message.
  - (lower the numerical value of the identifier, the higher the priority)
- When two or more nodes attempt to transmit at the same time, a non-destructive arbitration technique guarantees messages are sent in order of priority and no messages are lost.

**Non-Destructive Bitwise Arbitration**

- Bus arbitration resolved via arbitration with wired-AND bus connections.
  - Dominate state (logic 0, bus is high)
  - Recessive state (logic 1, bus is low)

![Diagram of CAN Bus arbitration](image)
Message Format and Block Diagram

CAN Message Format

- Data is transmitted and received using Message Frames
- 8 byte data payload per message
- Standard and Extended identifier formats

- **Standard Frame: 11-bit Identifier (CAN v2.0A)**
  [Diagram of Standard Frame]

- **Extended Frame: 29-bit Identifier (CAN v2.0B)**
  [Diagram of Extended Frame]

The DSP CAN module is a full CAN Controller. It contains a message handler for transmission and reception management, and frame storage. The specification is CAN 2.0B Active – that is, the module can send and accept standard (11-bit identifier) and extended frames (29-bit identifier).

eCAN Block Diagram

[Diagram of eCAN Block Diagram]
The CAN controller module contains 32 mailboxes for objects of 0 to 8-byte data lengths:

- configurable transmit/receive mailboxes
- configurable with standard or extended identifier

The CAN module mailboxes are divided into several parts:

- MID – contains the identifier of the mailbox
- MCF (Message Control Field) – contains the length of the message (to transmit or receive) and the RTR bit (Remote Transmission Request – used to send remote frames)
- MDL and MDH – contains the data

The CAN module contains registers which are divided into five groups. These registers are located in data memory from 0x006000 to 0x0061FF. The five register groups are:

- Control & Status Registers
- Local Acceptance Masks
- Message Object Time Stamps
- Message Object Timeout
- Mailboxes

---

### eCAN Summary

- **Fully CAN protocol compliant, version 2.0B**
- **Supports data rates up to 1 Mbps**
- **Thirty-two mailboxes**
  - Configurable as receive or transmit
  - Configurable with standard or extended identifier
  - Programmable receive mask
  - Uses 32-bit time stamp on messages
  - Programmable interrupt scheme (two levels)
  - Programmable alarm time-out
- **Programmable wake-up on bus activity**
- **Self-test mode**
Introduction

This module contains various references to support the development process.

Learning Objectives

- Signal Processing Libraries
- Additional Resources
  - Internet
  - Product Information Center
Module Topics

Development Support ............................................................................................................ 12-1

Module Topics.................................................................................................................. 12-2

TI Support Resources........................................................................................................... 12-3
TI Support Resources

C28x Signal Processing Libraries

<table>
<thead>
<tr>
<th>Signal Processing Libraries</th>
<th>Literature #</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACI3 3: Simulated Indirect FOC of ACI Motor</td>
<td>SPRC077</td>
</tr>
<tr>
<td>ACI3 4: Real Direct FOC of ACI Motor</td>
<td>SPRC079</td>
</tr>
<tr>
<td>ACI3 4: Simulated Direct FOC of ACI Motor</td>
<td>SPRC078</td>
</tr>
<tr>
<td>Digital Motor Control Library</td>
<td>SPRC080</td>
</tr>
<tr>
<td>DSP Fast Fourier Transform (FFT) Library</td>
<td>SPRC081</td>
</tr>
<tr>
<td>DSP Filter Library</td>
<td>SPRC082</td>
</tr>
<tr>
<td>DSP Fixed-Point Math Library</td>
<td>SPRC085</td>
</tr>
<tr>
<td>DSP IQ Math Library</td>
<td>SPRC087</td>
</tr>
<tr>
<td>DSP Signal Generator Library</td>
<td>SPRC083</td>
</tr>
<tr>
<td>DSP Software Test Bench (STB) Library</td>
<td>SPRC084</td>
</tr>
<tr>
<td>C28x Peripheral Examples in C</td>
<td>SPRC097</td>
</tr>
</tbody>
</table>

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

For More Information . . .

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

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

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

USA - Product Information Center (PIC)
Phone: 800-477-8924 or 972-644-5580
Email: support@ti.com
- Information and support for all TI Semiconductor products/tools
- Submit suggestions and errata for tools, silicon and documents
**European Product Information Center (EPIC)**

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

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

Email: epic@ti.com

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

Module Topics ................................................................................................................................. A-1

eZdsp™ F2812 .................................................................................................................................. A-3
  eZdsp™ F2812 Connector / Header and Pin Diagram ............................................................ A-3
  P2 – Expansion Interface ............................................................................................................ A-4
  P4 / P8 / P7 – I/O Interface ........................................................................................................ A-5
  P5 / P9 – Analog Interface ......................................................................................................... A-7
  eZdsp™ F2812 Jumper Diagram ................................................................................................. A-8
  JP1 – XMP/MCn Select ............................................................................................................... A-8
  JP2 – Flash Programming Voltage Select ................................................................................ A-8
  JP9 – PLL Disable ..................................................................................................................... A-9
  DS1 / DS2 - LEDs .................................................................................................................... A-9
  TP1 / TP2 – Test Points ............................................................................................................. A-10
eZdsp™ F2812

eZdsp™ F2812 Connector / Header and Pin Diagram

![Diagram of eZdsp™ F2812 Connector]

**Table 1: eZdsp™ F2812 Connectors**

<table>
<thead>
<tr>
<th>Connector</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>JTAG Interface</td>
</tr>
<tr>
<td>P2</td>
<td>Expansion</td>
</tr>
<tr>
<td>P3</td>
<td>Parallel Port/JTAG Controller Interface</td>
</tr>
<tr>
<td>P4/P8/P7</td>
<td>I/O Interface</td>
</tr>
<tr>
<td>P5/P9</td>
<td>Analog Interface</td>
</tr>
<tr>
<td>P6</td>
<td>Power Connector</td>
</tr>
</tbody>
</table>
# P2 – Expansion Interface

<table>
<thead>
<tr>
<th>Pin #</th>
<th>Signal</th>
<th>Pin #</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>+5V</td>
<td>2</td>
<td>+5V</td>
</tr>
<tr>
<td>3</td>
<td>XD0</td>
<td>4</td>
<td>XD1</td>
</tr>
<tr>
<td>5</td>
<td>XD2</td>
<td>6</td>
<td>XD3</td>
</tr>
<tr>
<td>7</td>
<td>XD4</td>
<td>8</td>
<td>XD5</td>
</tr>
<tr>
<td>9</td>
<td>XD6</td>
<td>10</td>
<td>XD7</td>
</tr>
<tr>
<td>11</td>
<td>XD8</td>
<td>12</td>
<td>XD9</td>
</tr>
<tr>
<td>13</td>
<td>XD10</td>
<td>14</td>
<td>XD11</td>
</tr>
<tr>
<td>15</td>
<td>XD12</td>
<td>16</td>
<td>XD13</td>
</tr>
<tr>
<td>17</td>
<td>XD14</td>
<td>18</td>
<td>XD15</td>
</tr>
<tr>
<td>19</td>
<td>XA0</td>
<td>20</td>
<td>XA1</td>
</tr>
<tr>
<td>21</td>
<td>XA2</td>
<td>22</td>
<td>XA3</td>
</tr>
<tr>
<td>23</td>
<td>XA4</td>
<td>24</td>
<td>XA5</td>
</tr>
<tr>
<td>25</td>
<td>XA6</td>
<td>26</td>
<td>XA7</td>
</tr>
<tr>
<td>27</td>
<td>XA8</td>
<td>28</td>
<td>XA9</td>
</tr>
<tr>
<td>29</td>
<td>XA10</td>
<td>30</td>
<td>XA11</td>
</tr>
<tr>
<td>31</td>
<td>XA12</td>
<td>32</td>
<td>XA13</td>
</tr>
<tr>
<td>33</td>
<td>XA14</td>
<td>34</td>
<td>XA15</td>
</tr>
<tr>
<td>35</td>
<td>GND</td>
<td>36</td>
<td>GND</td>
</tr>
<tr>
<td>37</td>
<td>XZCS0/AND1n</td>
<td>38</td>
<td>XZCS2n</td>
</tr>
<tr>
<td>39</td>
<td>XREADY</td>
<td>40</td>
<td>10K Pullup</td>
</tr>
<tr>
<td>41</td>
<td>XSnW</td>
<td>42</td>
<td>10K Pullup</td>
</tr>
<tr>
<td>43</td>
<td>XRDn</td>
<td>44</td>
<td>XRDn</td>
</tr>
<tr>
<td>45</td>
<td>+3.3V</td>
<td>46</td>
<td>XMMI/NT13</td>
</tr>
<tr>
<td>47</td>
<td>XRSn/RSn</td>
<td>48</td>
<td>No connect</td>
</tr>
<tr>
<td>49</td>
<td>GND</td>
<td>50</td>
<td>GND</td>
</tr>
<tr>
<td>51</td>
<td>GND</td>
<td>52</td>
<td>GND</td>
</tr>
<tr>
<td>53</td>
<td>XA16</td>
<td>54</td>
<td>XA17</td>
</tr>
<tr>
<td>55</td>
<td>XA18</td>
<td>56</td>
<td>XHOLDN</td>
</tr>
<tr>
<td>57</td>
<td>XHOLDAn</td>
<td>58</td>
<td>No connect</td>
</tr>
<tr>
<td>59</td>
<td>No connect</td>
<td>60</td>
<td>No connect</td>
</tr>
</tbody>
</table>
## P4 / P8 / P7 – I/O Interface

|   | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | P4  |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|-----|
|   | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | P8  |
| 1 | 1 | 3 | 5 | 7 | 9 | 11| 13| 15| 17| 19 | 21| 23 | 25| 27 | 29 | 31 | 33 | 35 | 37 | 39 | P7  |

### Table 3: P4/P8, I/O Connectors

<table>
<thead>
<tr>
<th>P4 Pin #</th>
<th>P4 Signal</th>
<th>P4 Pin #</th>
<th>P8 Signal</th>
<th>P8 Pin #</th>
<th>P8 Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>+5 Volts</td>
<td>1</td>
<td>+5 Volts</td>
<td>2</td>
<td>+5 Volts</td>
</tr>
<tr>
<td>2</td>
<td>XINT2/ADC/SOC</td>
<td>3</td>
<td>SCITXDA</td>
<td>4</td>
<td>SCIRXDA</td>
</tr>
<tr>
<td>3</td>
<td>MCLKXA</td>
<td>5</td>
<td>XINT1n/XB1On</td>
<td>6</td>
<td>CAP1/QEP1</td>
</tr>
<tr>
<td>4</td>
<td>MCLKRA</td>
<td>7</td>
<td>CAP2/QEP2</td>
<td>8</td>
<td>CAP3/QEP1</td>
</tr>
<tr>
<td>5</td>
<td>MFSXA</td>
<td>9</td>
<td>PWM1</td>
<td>10</td>
<td>PWM2</td>
</tr>
<tr>
<td>6</td>
<td>MFSRA</td>
<td>11</td>
<td>PWM3</td>
<td>12</td>
<td>PWM4</td>
</tr>
<tr>
<td>7</td>
<td>MDXA</td>
<td>13</td>
<td>PWM5</td>
<td>14</td>
<td>PWM6</td>
</tr>
<tr>
<td>8</td>
<td>MDRA</td>
<td>15</td>
<td>T1PWM/T1CMP</td>
<td>16</td>
<td>T2PWM/T2CMP</td>
</tr>
<tr>
<td>9</td>
<td>No connect</td>
<td>17</td>
<td>TDIRA</td>
<td>18</td>
<td>TCLKINA</td>
</tr>
<tr>
<td>10</td>
<td>GND</td>
<td>19</td>
<td>GND</td>
<td>20</td>
<td>GND</td>
</tr>
<tr>
<td>11</td>
<td>CAP5/QEP4</td>
<td>21</td>
<td>No connect</td>
<td>22</td>
<td>XINT1n/XB1On</td>
</tr>
<tr>
<td>12</td>
<td>CAP6/QEP12</td>
<td>23</td>
<td>SPI/SOMA</td>
<td>24</td>
<td>SPI/SOHE</td>
</tr>
<tr>
<td>13</td>
<td>T3PWM/T3CMP</td>
<td>25</td>
<td>SPI/CLKA</td>
<td>26</td>
<td>SPI/STEA</td>
</tr>
<tr>
<td>14</td>
<td>T4PWM/T4CMP</td>
<td>27</td>
<td>CANTXA</td>
<td>28</td>
<td>CANRXA</td>
</tr>
<tr>
<td>15</td>
<td>TDIRB</td>
<td>29</td>
<td>XCLKOUT</td>
<td>30</td>
<td>PWM7</td>
</tr>
<tr>
<td>16</td>
<td>TCLKINB</td>
<td>31</td>
<td>PWM8</td>
<td>32</td>
<td>PWM9</td>
</tr>
<tr>
<td>17</td>
<td>XF/XPLLDISn</td>
<td>33</td>
<td>PWM10</td>
<td>34</td>
<td>PWM11</td>
</tr>
<tr>
<td>18</td>
<td>SCITXDB</td>
<td>35</td>
<td>PWM12</td>
<td>36</td>
<td>CAP4/QEP3</td>
</tr>
<tr>
<td>19</td>
<td>SCIRXDB</td>
<td>37</td>
<td>T3CPR/PDPINTAn</td>
<td>38</td>
<td>T3CPR/PDPINTBn</td>
</tr>
<tr>
<td>20</td>
<td>GND</td>
<td>39</td>
<td>GND</td>
<td>40</td>
<td>GND</td>
</tr>
<tr>
<td>P7 Pin #</td>
<td>P7 Signal</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>C1TRIPn</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>C2TRIPn</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C3TRIPn</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>T2CTRIPn/EVASOCn</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>C4TRIPn</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>C5TRIPn</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>C6TRIPn</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>T4CTRIPn/EVBSOCn</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>No connect</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>GND</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# P5 / P9 – Analog Interface

<table>
<thead>
<tr>
<th></th>
<th>P5</th>
<th>ANALOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

## Table 5: P5/P9, Analog Interface Connector

<table>
<thead>
<tr>
<th>P5 Pin #</th>
<th>Signal</th>
<th>P9 Pin #</th>
<th>Signal</th>
<th>P9 Pin #</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ADCINB0</td>
<td>1</td>
<td>GND</td>
<td>2</td>
<td>ADCINA0</td>
</tr>
<tr>
<td>2</td>
<td>ADCINB1</td>
<td>3</td>
<td>GND</td>
<td>4</td>
<td>ADCINA1</td>
</tr>
<tr>
<td>3</td>
<td>ADCINB2</td>
<td>5</td>
<td>GND</td>
<td>6</td>
<td>ADCINA2</td>
</tr>
<tr>
<td>4</td>
<td>ADCINB3</td>
<td>7</td>
<td>GND</td>
<td>8</td>
<td>ADCINA3</td>
</tr>
<tr>
<td>5</td>
<td>ADCINB4</td>
<td>9</td>
<td>GND</td>
<td>10</td>
<td>ADCINA4</td>
</tr>
<tr>
<td>6</td>
<td>ADCINB5</td>
<td>11</td>
<td>GND</td>
<td>12</td>
<td>ADCINA5</td>
</tr>
<tr>
<td>7</td>
<td>ADCINB6</td>
<td>13</td>
<td>GND</td>
<td>14</td>
<td>ADCINA6</td>
</tr>
<tr>
<td>8</td>
<td>ADCINB7</td>
<td>15</td>
<td>GND</td>
<td>16</td>
<td>ADCINA7</td>
</tr>
<tr>
<td>9</td>
<td>ADCREFM</td>
<td>17</td>
<td>GND</td>
<td>18</td>
<td>VREFLO</td>
</tr>
<tr>
<td>10</td>
<td>ADCREFP</td>
<td>19</td>
<td>GND</td>
<td>20</td>
<td>No connect</td>
</tr>
</tbody>
</table>
eZdsp™ F2812 Jumper Diagram

JP1 – XMP/MCn Select

Table 6: JP1, XMP/MCn Select

<table>
<thead>
<tr>
<th>Position</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-2</td>
<td>Microprocessor mode</td>
</tr>
<tr>
<td>2-3 *</td>
<td>Microcomputer mode</td>
</tr>
</tbody>
</table>

* as shipped from factory

JP2 – Flash Programming Voltage Select

Table 7: JP2, Flash Programming Voltage Select

<table>
<thead>
<tr>
<th>Position</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-2 *</td>
<td>Voltage supplied to DSP</td>
</tr>
<tr>
<td>2-3</td>
<td>Voltage not supplied to DSP</td>
</tr>
</tbody>
</table>

* always in this position
Appendix


Table 8: JP7, JP8, JP11, JP12, Boot Mode Select

<table>
<thead>
<tr>
<th>JP7, BOOT3 SCITXDA</th>
<th>JP8, BOOT2 MDXA</th>
<th>JP11, BOOT1 SPISTEA</th>
<th>JP12, BOOT0 SPICLKA</th>
<th>MODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>FLASH</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>SPI</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>SCI</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>H0 *</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>OTP</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>PARALLEL</td>
</tr>
</tbody>
</table>

* factory default

JP9 – PLL Disable

Table 9: JP9, PLL Disable

<table>
<thead>
<tr>
<th>Position</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-2 *</td>
<td>PLL Enabled</td>
</tr>
<tr>
<td>2-3</td>
<td>PLL disabled</td>
</tr>
</tbody>
</table>

* as shipped from the factory

DS1 / DS2 - LEDs

Table 10: LEDs

<table>
<thead>
<tr>
<th>LED #</th>
<th>Color</th>
<th>Controlling Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS1</td>
<td>Green</td>
<td>+5 Volts</td>
</tr>
<tr>
<td>DS2</td>
<td>Green</td>
<td>XF bit (XF high = on)</td>
</tr>
</tbody>
</table>
### TP1 / TP2 – Test Points

**Table 11: Test Points**

<table>
<thead>
<tr>
<th>Test Point</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>TP1</td>
<td>Ground</td>
</tr>
<tr>
<td>TP2</td>
<td>Analog Ground</td>
</tr>
</tbody>
</table>
Appendix B – Addressing Modes

Introduction

Appendix B will describe the data addressing modes on the C28x. Immediate addressing allows for constant expressions which are especially useful in the initialization process. Indirect addressing uses auxiliary registers as pointers for accessing organized data in arrays. Direct addressing is used to access general purpose memory. Techniques for managing data pages, relevant to direct addressing will be covered as well. Finally, register addressing allows for interchange between CPU registers.

Learning Objectives

- Explain .sect and .usect assembly directives
- Explain assembly addressing modes
- Understand instruction formats
- Describe options for each addressing mode
Module Topics

Appendix B – Addressing Modes ................................................................. B-1

Module Topics........................................................................................................ B-2
Labels, Mnemonics and Assembly Directives .................................................... B-3
Addressing Modes............................................................................................. B-4
Instruction Formats .......................................................................................... B-5
Register Addressing......................................................................................... B-6
Immediate Addressing..................................................................................... B-7
Direct Addressing............................................................................................ B-8
Indirect Addressing......................................................................................... B-10
Review.............................................................................................................. B-13
   Exercise B.................................................................................................... B-14
Lab B: Addressing............................................................................................. B-15
   OPTIONAL Lab B-C: Array Initialization in C............................................. B-17
   Solutions.................................................................................................... B-18
Labels, Mnemonics and Assembly Directives

Labels and Mnemonics

- **Labels**
  - Optional for all assembly instructions and most assembler directives
  - Must begin in column 1
  - The " : " is not treated as part of the label name
  - Used as pointers to memory or instructions

- **Mnemonics**
  - Lines of instructions
  - Use upper or lower case
  - Become components of program memory

Assembly Directives

- Begin with a period (.) and are lower case
  - Used by the linker to locate code and data into specified sections

- Directives allow you to:
  - Define a label as global
  - Reserve space in memory for un-initialized variables
  - Initialize memory

- **Directives**
  - Initial memory
    - .sect “name” used for code or constants
    - initialized section
  - uninitialazed section
    - label .usect “name”,5 used for variables
## Addressing Modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Symbol</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Register</td>
<td>Operate between Registers</td>
</tr>
<tr>
<td>Immediate</td>
<td>#</td>
<td>Constants and Initialization</td>
</tr>
<tr>
<td>Direct</td>
<td>@</td>
<td>General-purpose access to data</td>
</tr>
<tr>
<td>Indirect</td>
<td>*</td>
<td>Support for pointers – access arrays, lists, tables</td>
</tr>
</tbody>
</table>

Four main categories of addressing modes are available on the C28x. Register addressing mode allows interchange between all CPU registers, convenient for solving intricate equations. Immediate addressing is helpful for expressing constants easily. Direct addressing mode allows information in memory to be accessed. Indirect addressing allows pointer support via dedicated ‘auxiliary registers’, and includes the ability to index, or increment through a structure. The C28x supports a true software stack, desirable for supporting the needs of the C language and other structured programming environments, and presents a stack-relative addressing mode for efficiently accessing elements from the stack. Paged direct addressing offers general-purpose single cycle memory access, but restricts the user to working in any single desired block of memory at one time.
Instruction Formats

The C28x follows a convention that uses instruction, destination, then source operand order (INSTR dst, src). Several general formats exist to allow modification of memory or registers based on constants, memory, or register inputs. Different modes are identifiable by their leading characters (# for immediate, * for indirect, and @ for direct). Note that registers or data memory can be selected as a ‘mem’ value.

<table>
<thead>
<tr>
<th>INSTR</th>
<th>dst ,src</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSTR</td>
<td>REG</td>
<td>NEG AL</td>
</tr>
<tr>
<td>INSTR</td>
<td>REG,#imm</td>
<td>MOV ACC,#1</td>
</tr>
<tr>
<td>INSTR</td>
<td>REG,mem</td>
<td>ADD AL,@x</td>
</tr>
<tr>
<td>INSTR</td>
<td>mem,REG</td>
<td>SUB AL,@AR0</td>
</tr>
<tr>
<td>INSTR</td>
<td>mem,#imm</td>
<td>MOV *XAR0++,#25</td>
</tr>
</tbody>
</table>

◆ What is a “REG”?  
  ◆ 16-bit Access = AR0 through AR7, AH, AL, PH, PL, T and SP  
  ◆ 32-bit Access = XAR0 through XAR7, ACC, P, XT

◆ What is an “#imm”?  
  ◆ an immediate constant stored in the instruction

◆ What is a “mem”?  
  ◆ A directly or indirectly addressed operand from data memory  
  ◆ Or, one of the registers from “REG”!
  ◆ loc16 or loc32 (for 16-bit or 32-bit data access)
Register addressing allows the exchange of values between registers, and with certain instructions can be used in conjunction with other addressing modes, yielding a more efficient instruction set. Remember that any ‘mem’ field allows the use of a register as the operand, and that no special character (such as @, *, or #) need be used to specify the register mode.

Register addressing - Example

<table>
<thead>
<tr>
<th>Format</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>Ax, loc16</td>
</tr>
<tr>
<td>MOV</td>
<td>AH, @AL</td>
</tr>
<tr>
<td>MOV</td>
<td>loc16, Ax, COND</td>
</tr>
<tr>
<td>MOV</td>
<td>@AR1, AL, GT</td>
</tr>
</tbody>
</table>

User Guide & Dis-assembler
use @ for second register
Immediate Addressing

Immediate Addressing – “#”

- Fixed value part of program memory instruction
- Supports short (8-bit) and long (16-bit) immediate constants
- Long immediate can include a shift
- Used to initialize registers, and operate with constants

Immediate addressing allows the user to specify a constant within an instruction mnemonic. Short immediate are single word, and execute in a single cycle. Long (16-bit) immediate allow full sized values, which become two-word instructions - yet execute in a single instruction cycle.

Immediate Addressing – Example

- Short Immediate, 1 Word (ANDB)

  ANDB Ax,#8Bit
  ANDB Ax #8Bit

  AND automatically replaced by ANDB if IMM value is 8 bits or less

- Long Immediate, 2 Words (AND)

  AND loc16,#16Bit
  AND loc16 #16Bit

  AND Ax,loc16,#16Bit
  AND Ax loc16 #16Bit

  AND ACC,#16Bit,<<0-16
  AND ACC shift #16Bit
Direct Addressing

Direct addressing allows for access to the full 4-Meg words space in 64 word “page” groups. As such, a 16-bit Data Page register is used to extend the 6-bit local address in the instruction word. Programmers should note that poor DP management is a key source of programming errors. Paged direct addressing is fast and reliable if the above considerations are followed. The watch operation, recommended for use whenever debugging, extracts the data page and displays it as the base address currently in use for direct addressing.

![Direct Addressing Table](image)

- Data memory space divided into 65,536 pages with 64 words on each page
- Data page pointer (DP) used to select active page
- 16-bit DP is concatenated with a 6-bit offset from the instruction to generate an absolute 22-bit address
- Access data on a given page in any order
Direct Addressing – Example

\[ Z = X + Y \]

Z = X + Y

Data Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001C0</td>
<td>0001</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0001FD</td>
<td>1000</td>
</tr>
<tr>
<td>0001FE</td>
<td>0500</td>
</tr>
<tr>
<td>0001FF</td>
<td>1500</td>
</tr>
</tbody>
</table>

x .usect "samp", 3
.sect "code"
MOVW DP, #x
MOV AL, @x
ADD AL, @y
MOV @z, AL

movw DP, #x
mov alt, @x
add al, @y
mov @z, al

x .usect "samp", 3
.sect "code"
MOVW DP, #x
MOV AL, @x
ADD AL, @y
MOV @z, AL

Direct Addressing – Caveats

(X and Y not on the same page)

\[ Z = X + Y \]

DP=0007

<table>
<thead>
<tr>
<th>Accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV AL, @x</td>
</tr>
<tr>
<td>0 0 0 0 1 0 0 0</td>
</tr>
</tbody>
</table>

MOV @z, AL

Solution: Group and block variables in ASM file:

x .usect "samp", 3
.sect "code"
MOVW DP, #x
MOV AL, @x
ADD AL, @y
MOV @z, AL

y .set x+1
z .set x+2

Force all locations to same data
page (1st hole, else linker error)
Assign vars within block
Indirect Addressing

Auxiliary Registers (XARn) used to access full data memory space
Address Register Arithmetic Unit (ARAU) used to modify the XARn
Access data from arrays anywhere in data memory in an orderly fashion

Any of eight hardware pointers (ARs) may be employed to access values from the first 64K of data memory. Auto-increment or decrement is supported at no additional cycle cost. XAR register formats offer larger 32-bit widths, allowing them to access across the full 4-Giga words data space.

Indirect Addressing Modes

- **Auto-increment / decrement:** *XARn++, *--XARn
  - Post-increment or Pre-decrement
- **Offset:** *+XARn[AR0 or AR1], *+XARn[3bit]
  - Offset by 16-bit AR0 or AR1, or 3-bit constant
- **Stack Relative:** *-SP[6bit]
  - Index by 6-bit offset (optimal for C)
- **Immediate Direct:** *(0:16bit)
  - Access low 64K
- **Circular:** *AR6%++
  - AR1(7:0) is buffer size
  - XAR6 is current address
Indexed addressing offers the ability to select operands from within an array without modification to the base pointer. Stack-based operations are handled with a 16-bit Stack Pointer register, which operates over the base 64K of data memory. It offers 6-bit non-destructive indexing to access larger stack-based arrays efficiently.
Indirect Addressing

Indirect Addressing – Example
Stack Relative

\[ x_2 = x_1 + x_3 \]

\[
\text{Data Memory}
\begin{array}{c}
0 & 1 & 2 & 0 \\
0 & 3 & 2 & 0 \\
0 & 2 & 0 & 0 \\
\end{array}
\]

- SP -

\[
\text{Accumulator}
\begin{array}{c}
\text{empty} \\
\text{empty} \\
\text{Instr. 2} \quad 0 & 0 & 0 & 0 & 0 & 3 & 2 & 0 \\
\text{Instr. 1} \quad 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 \\
\end{array}
\]

Useful for stack based operations

Indirect Addressing – Example
Circular

start of buffer

\[
\begin{array}{c}
\text{AAAA} \ldots \text{AAAA} \quad \text{AAAA} \quad \text{AAAA} \quad 0000 \quad 0000 \\
\end{array}
\]

(align on 256 word boundary)

access pointer

\[
\begin{array}{c}
\text{XAR6 (32)} \\
\text{AAAA} \ldots \text{AAAA} \\
\text{AR1 Low (16)} \\
\end{array}
\]

end of buffer

\[
\begin{array}{c}
\text{-----} \quad \text{-----} \\
\text{N-1} \\
\end{array}
\]

(Element 0)

Buffer Size N

circular buffer range

(Element N-1)

MAC P,*AR6%++,*XAR7++

LINKER.CMD

SECTIONS

{ Buf_Mem: align(256) } > RAM PAGE 1

...
Data memory can be accessed in numerous ways:
- Stack Addressing: allows a range to 64K
- Direct Addressing: Offers a 16-bit DP plus a 6-bit offset, allowing a 4M range
- Indirect Addressing: Offers the full 4G range
### Exercise B

#### Exercise B: Addressing

<table>
<thead>
<tr>
<th>Given:</th>
<th>DP = 4000</th>
<th>DP = 4004</th>
<th>DP = 4006</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address/Data (hex)</td>
<td>100030</td>
<td>0025</td>
<td>100100</td>
</tr>
<tr>
<td>Fill in the</td>
<td>100031</td>
<td>0120</td>
<td>100101</td>
</tr>
<tr>
<td>table below</td>
<td>100032</td>
<td>0020</td>
<td>100102</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Src Mode</th>
<th>Program</th>
<th>ACC</th>
<th>DP</th>
<th>XAR1</th>
<th>XAR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVW</td>
<td>DP,#4000h</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVL</td>
<td>XAR1,#100100h</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVL</td>
<td>XAR2,#100180h</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| MOV      | AL,#31h |     |    |      |      |
| ADD      | AL,*XAR1++ | |    |      |      |
| SUB      | AL,#30h |     |    |      |      |

| ADD      | AL,*XAR1++ | |    |      |      |
| MOVW     | DP,#4006h | |    |      |      |
| ADD      | AL,#32 |     |    |      |      |

| SUB      | AL,*XAR2 |     |    |      |      |
| SUB      | AL,*XAR2[1] | |    |      |      |

Imm: Immediate; Dir: Direct; Reg: Register; ldr: Indirect

In the table above, fill in the values for each of the registers for each of the instructions. Three areas of data memory are displayed at the top of the diagram, showing both their addresses and contents in hexadecimal. Watch out for surprises along the way. First, you should answer the addressing mode for the source operand. Then, fill in the change values as the result of the instruction operation.
Lab B: Addressing

Objective

The objective of this lab is to practice and verify the mechanics of addressing. In this process we will expand upon the ASM file from the previous lab to include new functions. Additionally, we learn how to run and observe the operation of code using Code Composer Studio.

In this lab, we will initialize the “vars” arrays allocated in the previous lab with the contents of the “const” table. How is this best accomplished? Consider the process of loading the first “const” value into the accumulator and then storing this value to the first “vars” location, and repeating this process for each of the succeeding values.

• What forms of addressing could be used for this purpose?
• Which addressing mode would be best in this case? Why?
• What problems could arise with using another mode?

Procedure

Copy Files, Create Project File

1. Create a new project called LabB.pjt in C:\C28x\Labs\Appendix\LabB and add LabB.asm, and LabB.cmd to it. Check your file list to make sure all the files are there. Be sure to setup the Build Options by clicking: Project → Build Options on the menu bar. Select the Linker tab. In the middle of the screen select “No Autoinitialization” under “Autoinit Model:”. Create a map file by typing .\Debug\LabB.map in the Map Filename [-m] field. Enter start in the “Code Entry Point (-e):” field. Next, select the Compiler tab. Note that “Full Symbolic Debug (-g)” under “Generate Debug Info:” is selected. Then select OK to save the Build Options.

Initialize Allocated RAM Array from ROM Initialization Table

2. Edit LabB.asm and modify it to copy table[9] to data[9] using indirect addressing. (Note: data[9] consists of the allocated arrays of data, coeff, and result). Initialize the allocated RAM array from the ROM initialization table:

   • Delete the NOP operations from the “code” section.
   • Initialize pointers to the beginning of the “const” and “vars” arrays.
   • Transfer the first value from “const” to the “vars” array.
   • Repeat the process for all values to be initialized.

   To perform the copy, consider using a load/store method via the accumulator. Which part of an accumulator (low or high) should be used? Use the following when writing your copy routine:
   - use AR1 to hold the address of table
   - use AR2 to hold the address of data

3. It is good practice to trap the end of the program (i.e. use either “end: B end,UNC” or “end: B start,UNC”). Save your work.
Build and Load

4. Click the “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. Load the output file onto the target. Click:

File → Load Program...

If you wish, right click on the LabB.asm source window and select Mixed Mode to debug using both source and assembly.

**Note:** Code Composer Studio can automatically load the output file after a successful build. On the menu bar click: Option → Customize... and select the “Program Load Options” tab, check “Load Program After Build”, then click OK.

6. Single-step your routine. While single-stepping, it is helpful to see the values located in table[9] and data[9] at the same time. Open two memory windows by using the “View Memory” button on the vertical toolbar and using the address labels table and data. Setting the properties filed to “Hex – TI style” will give you more viewable data in the window. Additionally, it is useful to watch the CPU core (and status) registers. Open the CPU core (and status) registers 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**
OPTIONAL Lab B-C: Array Initialization in C

Objective

The objective of this lab is to practice and verify the mechanics of initialization using C. Additionally, we learn how to run and observe the operation of C code using Code Composer Studio. In this lab, we will initialize the “vars” arrays with the contents of the “const” table.

Procedure

Create Project File

1. In Code Composer Studio create a new project called LabB-C.pjt in C:\C28x\Labs\Appendix\LabB\LabB-C and add LabB-C.c, LabB-C.cmd and C:\ti\c2000\cgtools\lib\rts2800_ml.lib to it. Check your file list to make sure all the files are there. Do not setup any Build Options. The default values will be used. In Appendix Lab D exercise, we will experiment and explore the various build options when working with C.

Initialize Allocated RAM Array from ROM Initialization Table

2. Edit LabB-C.c and modify the “main” routine to copy table[9] to the allocated arrays of data[4], coeff[4], and result[1]. (Note: data[9] consists of the allocated arrays of data, coeff, and result).

Build and Load

3. Click the “Rebuild All” button and watch the tools run in the build window. Debug as necessary.

Note: Have Code Composer Studio automatically load the output file after a successful build. On the menu bar click: Option → Customize… and select the “Program Load Options” tab, check “Load Program After Build”, then click OK.

4. Under Debug on the menu bar click “Go Main”. Single-step your routine. While single-stepping, it is helpful to see the values located in table[9] and data[9] at the same time. Open two memory windows by using the “View Memory” button on the vertical toolbar and using the address labels table and data. Setting the properties filed to “Hex – TI style” will give you more viewable data in the window. Additionally, you can watch the CPU (and Status) registers. Open the CPU core and status registers 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
## Exercise B: Addressing - Solution

### Given:
- **DP = 4000**
- **DP = 4004**
- **DP = 4006**

**Address/Data (hex):**
- 100030
- 0025
- 100100
- 0105
- 100180
- 0100

**Fill in the table below:**
- 100031
- 0120
- 100101
- 0060
- 100181
- 0030

<table>
<thead>
<tr>
<th>Src Mode</th>
<th>Program</th>
<th>ACC</th>
<th>DP</th>
<th>XAR1</th>
<th>XAR2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Imm</strong></td>
<td>MOVW DP,#4000h</td>
<td>4000</td>
<td>100100</td>
<td>100180</td>
<td></td>
</tr>
<tr>
<td><strong>Imm</strong></td>
<td>MOVL XAR1,#100100h</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Imm</strong></td>
<td>MOVL XAR2,#100180h</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dir</strong></td>
<td>MOV AL,@31h</td>
<td>120</td>
<td>225</td>
<td>200</td>
<td></td>
</tr>
<tr>
<td><strong>Idr</strong></td>
<td>ADD AL,*XAR1++</td>
<td>260</td>
<td>100102</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dir</strong></td>
<td>SUB AL,@30h</td>
<td>290</td>
<td>100180</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Idr</strong></td>
<td>ADD AL,*XAR2</td>
<td>270</td>
<td>370</td>
<td>340</td>
<td></td>
</tr>
<tr>
<td><strong>Dir</strong></td>
<td>SUB AL,*+XAR2[1]</td>
<td>360</td>
<td>320</td>
<td>100180</td>
<td></td>
</tr>
<tr>
<td><strong>Imm</strong></td>
<td>ADD AL,#32</td>
<td>1001B2</td>
<td>0320</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Idr</strong></td>
<td>SUB AL,*+XAR2[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dir</strong></td>
<td>MOV @32h,AL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Src Mode:**  
- **Imm:** Immediate;  
- **Reg:** Register;  
- **Dir:** Direct;  
- **Idr:** Indirect

**Program:**
- MOVW DP,#4000h
- MOVL XAR1,#100100h
- MOVL XAR2,#100180h
- MOV AL,@31h
- ADD AL,*XAR1++
- SUB AL,@30h
- ADD AL,*XAR2
- SUB AL,*+XAR2[1]
- ADD AL,#32
- SUB AL,*+XAR2[2]
- MOV @32h,AL
Appendix C – Assembly Programming

Introduction

Appendix C discusses the details of programming in assembly. It shows you how to use different instructions that further utilize the advantage of the architecture data paths. It gives you the ability to analyze the instruction set and pick the best instruction for the application.

Learning Objectives

<table>
<thead>
<tr>
<th>Learning Objectives</th>
</tr>
</thead>
<tbody>
<tr>
<td>◆ Perform simple program control using branch and conditional codes</td>
</tr>
<tr>
<td>◆ Write C28x code to perform basic arithmetic</td>
</tr>
<tr>
<td>◆ Use the multiplier to implement sum-of-products equations</td>
</tr>
<tr>
<td>◆ Use the RPT instruction (repeat) to optimize loops</td>
</tr>
<tr>
<td>◆ Use MAC for long sum-of-products</td>
</tr>
<tr>
<td>◆ Efficiently transfer the contents of one area of memory to another</td>
</tr>
<tr>
<td>◆ Examine read-modify-write operations</td>
</tr>
</tbody>
</table>
Module Topics

Appendix C – Assembly Programming ........................................................................................................C-1

Module Topics ................................................................................................................................................C-2

Program Control ........................................................................................................................................C-3
  Branches ..................................................................................................................................................C-3
  Program Control Instructions ..................................................................................................................C-4

ALU and Accumulator Operations ............................................................................................................C-6
  Simple Math & Shift .................................................................................................................................C-7

Multiplier ....................................................................................................................................................C-9
  Basic Multiplier ....................................................................................................................................C-10
  Repeat Instruction .................................................................................................................................C-11
  MAC Instruction ...................................................................................................................................C-12

Data Move ................................................................................................................................................C-13

Logical Operations ....................................................................................................................................C-15
  Byte Operations and Addressing .............................................................................................................C-15
  Test and Change Memory Instructions ..................................................................................................C-16
  Min/Max Operations ...............................................................................................................................C-17

Read Modify Write Operations ..................................................................................................................C-18

Lab C: Assembly Programming ..................................................................................................................C-20

OPTIONAL Lab C-C: Sum-of-Products in C ...............................................................................................C-22
**Program Control**

The program control logic and program address generation logic work together to provide proper program flow. Normally, the flow of a program is sequential: the CPU executes instructions at consecutive program memory addresses. At times, a discontinuity is required; that is, a program must branch to a nonsequential address and then execute instructions sequentially at that new location. For this purpose, the C28x supports interrupts, branches, calls, returns, and repeats. Proper program flow also requires smooth flow at the instruction level. To meet this need, the C28x has a protected pipeline and an instruction-fetch mechanism that attempts to keep the pipeline full.

**Branches**

The PC can access the entire 4M words (8M bytes) range. Some branching operations offer 8- and 16-bit relative jumps, while long branches, calls, and returns provide a full 22-bit absolute address. Dynamic branching allows a run-time calculated destination. The C28x provides the familiar arithmetic results status bits (Zero, Overflow, Negative, Carry) plus a Test Control bit which holds the result of a binary test. The states of these bits in various combinations allow a range of signed, unsigned, and binary branching conditions offered.
Program Control Instructions

Program Control - Branches

<table>
<thead>
<tr>
<th>Function</th>
<th>Instruction</th>
<th>Cycles T/F</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Branch</td>
<td>SB 8bit,cond</td>
<td>7/4</td>
<td>1</td>
</tr>
<tr>
<td>Fast Short Branch</td>
<td>SBF 8bit,EQ</td>
<td>4/4</td>
<td>1</td>
</tr>
<tr>
<td>Fast Relative Branch</td>
<td>B 16bit,cond</td>
<td>7/4</td>
<td>2</td>
</tr>
<tr>
<td>Fast Branch</td>
<td>BF 16bit,cond</td>
<td>4/4</td>
<td>2</td>
</tr>
<tr>
<td>Absolute Branch</td>
<td>LB 22bit</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Dynamic Branch</td>
<td>LB *XAR7</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Branch on AR</td>
<td>BANZ 16bit,ARn--</td>
<td>4/2</td>
<td>2</td>
</tr>
<tr>
<td>Branch on compare</td>
<td>BAR 16bit,ARn,ARn,EQ</td>
<td>4/2</td>
<td>2</td>
</tr>
</tbody>
</table>

Condition Code

- NEQ
- LT
- EQ
- LEQ
- GT
- HI
- GEQ
- HIS (C)
- LO (NC)
- NTC
- TC
- UN
- NOV
- OV
- NTC
- TC
- UNC
- NBIO

- Condition flags are set on the prior use of the ALU
- The assembler will optimize B to SB if possible

Program Control - Call/Return

<table>
<thead>
<tr>
<th>Function</th>
<th>Call Code</th>
<th>Cycles</th>
<th>Return code</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call</td>
<td>LCR 22bit</td>
<td>4</td>
<td>LRETR</td>
<td>4</td>
</tr>
<tr>
<td>Dynamic Call</td>
<td>LCR *XARn</td>
<td>4</td>
<td>LRETR</td>
<td>4</td>
</tr>
<tr>
<td>Interrupt Return</td>
<td>IRET</td>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- More Call variations in the user guide are for code backward compatibility

Stack

- Local Var
- RPC
- 22-bit old

Ret Addr

RPC

PC

Func

Ret Addr

New RPC

Old RPC

Func

Ret Addr

LCR Func

LRETR

Stack

Local Var

RPC

22-bit old
BANZ Loop Control Example

- Auxiliary register used as loop counter
- Branch if Auxiliary Register not zero
- Test performed on lower 16-bits of XARx only

\[ y = \sum_{n=0}^{4} x_n \]

```
len .set 5
x .usect "samp",6
y .set (x+len)

.sect "code"
MOVL XAR2,#x
MOV AR3,#len
MOV AL,*XAR2++
sum: ADD AL,*XAR2++
BANZ sum,AR3--
MOV *(0:y),AL
```

<table>
<thead>
<tr>
<th>Data</th>
<th>XAR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
</tr>
<tr>
<td>x0</td>
<td></td>
</tr>
<tr>
<td>x1</td>
<td></td>
</tr>
<tr>
<td>x2</td>
<td></td>
</tr>
<tr>
<td>x3</td>
<td></td>
</tr>
<tr>
<td>x4</td>
<td></td>
</tr>
</tbody>
</table>

AR3  COUNT
ALU and Accumulator Operations

One of the major components in the execution unit is the Arithmetic-Logical-Unit (ALU). To support the traditional Digital Signal Processing (DSP) operation, the ALU also has the zero cycle barrel shifter and the Accumulator. The enhancement that the C28x has is the additional data paths added form the ALU to all internal CPU registers and data memory. The connection to all internal registers helps the compiler to generate efficient C code. The data path to memory allows the C28x performs single atomic instructions read-modify-write to the memory.

The following slides introduce you to various instructions that use the ALU hardware. Word, byte, and long word 32-bit operation are supported.
**Simple Math & Shift**

### Accumulator - Basic Math Instructions

<table>
<thead>
<tr>
<th>Format</th>
<th>xxx Ax, #16b ;word</th>
<th>xxxB Ax, #8b ;byte</th>
<th>xxxL ACC, #32b ;long</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Ex</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>ACC, #01234h&lt;&lt;4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDB</td>
<td>AL, #34h</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note**: 
- `xxx =` instruction: MOV, ADD, SUB, ...
- `Ax = AH, or AL` Assembler will automatically convert to 1 word instruction.
- Two word instructions with shift option
- One word instruction, no shift

**Ax = AH or AL Operations**

<table>
<thead>
<tr>
<th>MOV Ax, loc16&gt;&gt;shift</th>
<th>16-bit constant (left shift optional)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD Ax, loc16&gt;01234h</td>
<td>from memory (left shift optional)</td>
</tr>
<tr>
<td>SUB Ax, loc16&lt;34h</td>
<td></td>
</tr>
</tbody>
</table>

**ACC Operations**

| MOV ACC, loc16<<shift | AL
|-----------------------|------------------|
| ADD ACC, loc16<<shift | AL
| SUB ACC, loc16<<shift | AL

**Shift the Accumulator**

**Shift full ACC**

<table>
<thead>
<tr>
<th>LSL ACC &lt;&lt;shift</th>
<th>31 ....... 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SFR ACC &gt;&gt;shift</td>
<td>0</td>
</tr>
<tr>
<td>LSL ACC &lt;&lt;T</td>
<td>31 ....... 0</td>
</tr>
<tr>
<td>SFR ACC &gt;&gt;T</td>
<td>C</td>
</tr>
</tbody>
</table>

**Shift AL or AH**

<table>
<thead>
<tr>
<th>LSL AX &lt;&lt;shift</th>
<th>15 ....... 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSR AX &lt;&lt;shift</td>
<td>C</td>
</tr>
<tr>
<td>ASR AX &gt;&gt;shift</td>
<td>15 ....... 0</td>
</tr>
<tr>
<td>LSL AX &lt;&lt;T</td>
<td>C</td>
</tr>
<tr>
<td>LSR AX &lt;&lt;T</td>
<td>ASR</td>
</tr>
<tr>
<td>ASR AX &gt;&gt;T</td>
<td>LSR</td>
</tr>
</tbody>
</table>

**Format Ex Variation**

- **Shift full ACC**
  - LSL ACC <<shift
  - SFR ACC >>shift
  - LSL ACC <<T
  - SFR ACC >>T

- **Shift AL or AH**
  - LSL AX <<shift
  - LSR AX <<shift
  - ASR AX >>shift
  - LSL AX <<T
  - LSR AX <<T
  - ASR AX >>T
32 Bit Shift Operations [ACC]

Logical Shift Left – Long: LSLL

Logical Shift Right – Long: LSRL

Arithmetic Shift Right – Long: ASRL

Examples:
- LSLL ACC, T
- LSRL ACC, T
- ASRL ACC, T

Note: T(4:0) are used; other bits are ignored

0 or 1
based on SXM

C - 8 C28x - Appendix C - Assembly Programming
Digital signal processors require many multiply and add math intensive operations. The single cycle multiplier is the second major component in the execution unit. The C28x has the traditional 16-bit-by-16-bit multiplier as previous TI DSP families. In addition, the C28x has a single cycle 32-bit-by-32-bit multiplier to perform extended precision math operations. The large multiplier allows the C28x to support higher performance control systems requirement while maintaining small or reduce code.

The following slides introduce instructions that use the 16-bit-by-16-bit multiplier and multiply and add (MAC) operations. The 32-bit-by-32-bit multiplication will be covered in the appendix.
Multiplier

## Basic Multiplier

### Multiplier Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Execution</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV T, loc16</td>
<td>T = loc16</td>
<td>Get first operand</td>
</tr>
<tr>
<td>MPY ACC, T, loc16</td>
<td>ACC = T*loc16</td>
<td>For single or first product</td>
</tr>
<tr>
<td>MPY P, T, loc16</td>
<td>P = T*loc16</td>
<td>For n&lt;sup&gt;th&lt;/sup&gt; product</td>
</tr>
<tr>
<td>MPYB ACC, T, #8bu</td>
<td>ACC = T*8bu</td>
<td>Using 8-bit unsigned const</td>
</tr>
<tr>
<td>MPYB P, T, #8bu</td>
<td>P = T*8bu</td>
<td>Using 8-bit unsigned const</td>
</tr>
<tr>
<td>MOV ACC, P</td>
<td>ACC = P</td>
<td>Move 1&lt;sup&gt;st&lt;/sup&gt; product&lt;&lt;PM to ACC</td>
</tr>
<tr>
<td>ADD ACC, P</td>
<td>ACC += P</td>
<td>Add n&lt;sup&gt;th&lt;/sup&gt; product&lt;&lt;PM to ACC</td>
</tr>
<tr>
<td>SUB ACC, P</td>
<td>ACC -= P</td>
<td>Sub n&lt;sup&gt;th&lt;/sup&gt; product&lt;&lt;PM fr. ACC</td>
</tr>
</tbody>
</table>

### Instruction Execution

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV T, loc16</td>
<td>ACC = P&lt;&lt;PM T = loc16</td>
</tr>
<tr>
<td>MOVA T, loc16</td>
<td>ACC += P&lt;&lt;PM T = loc16</td>
</tr>
<tr>
<td>MOVX T, loc16</td>
<td>ACC - = P&lt;&lt;PM T = loc16</td>
</tr>
<tr>
<td>MPYA P, T, #16b</td>
<td>ACC += P&lt;&lt;PM then P = T*#16b</td>
</tr>
<tr>
<td>MPYA P, T, loc16</td>
<td>ACC += P&lt;&lt;PM then P = T*loc16</td>
</tr>
<tr>
<td>MPYX P, T, loc16</td>
<td>ACC - = P&lt;&lt;PM then P = T*loc16</td>
</tr>
</tbody>
</table>

### Sum-of-Products

\[
Y = A\times X1 + B\times X2 + C\times X3 + D\times X4
\]

1. **ZAPA** ;ACC = P = OVC = 0
2. **MOV T, @X1** ;T = X1
3. **MPY P, T, @A** ;P = A\times X1
4. **MOVA T, @X2** ;T = X2 ;ACC = A\times X1
5. **MPY P, T, @B** ;P = B\times X2
6. **MOVA T, @X3** ;T = X3 ;ACC = A\times X1 + B\times X2
7. **MPY P, T, @C** ;P = C\times X3
8. **MOVA T, @X4** ;T = X4;ACC = A\times X1 + B\times X2 + C\times X3
9. **MPY P, T, @D** ;P = D\times X4
10. **ADDL ACC, P<<PM** ;ACC = Y
11. **MOVL @y, ACC**
### 32x32 Long Multiplication

#### Integer long multiplication
\[ u(\text{long}) = u(\text{long}) \times u(\text{long}) \]

#### Fraction long multiplication:
\[ (\text{long}) = (\text{long}) \times (\text{long}) \]
\[ (\text{long})_{64} = (\text{long})_{32} \times (\text{long})_{32} \]

---

#### Repeat Instruction

**Repeat Next: RPT**

- **Options:**
  - RPT #8bit up to 256 iterations
  - RPT loc16 location “loc16” holds count value

- **Features:**
  - Next instruction iterated N+1 times
  - Saves code space - 1 word
  - Low overhead - 1 cycle
  - Easy to use
  - Non-interruptible
  - Requires use of || before next line
  - May be nested within BANZ loops

**Example:**

```c
int x[5]={0,0,0,0,0};
x .usect "samp",5
MOV AR1,#x
RPT #4
|| MOV *XAR1++,#0
```

Refer to User Guide for more repeatable instructions

---

Refer to User Guide for more repeatable instructions
Single repeat instruction (RPT) is used to reduce code size and speed up many operations in the DSP application. Some of the most popular operations that use the RPT instruction to perform multiple taps digital filters or perform block of data transfer.

**MAC Instruction**

Sum-of-Products: $RPT / MAC$

\[
y = \sum_{n=0}^{19} x_n a_n
\]

- **XAR1++**
  - X0
  - X1
  - ...
  - X19

- **XAR7++**
  - A0
  - A1
  - ...
  - A19

Second operand must use XAR7

- MOV T, loc16
- ADD ACC, P
- MPY P, T, loc16
- MAC

Zero ACC & P
Repeat single
Dual operand
Last ADD

MOV W DP, #y
MOV L XAR1, #x
MOV L XAR7, #a0
ZAPA
RPT #19
MAC P, *XAR1++, *XAR7++
ADDL ACC, P<<PM
ADDL @y, ACC
B SOP, UNC

`x .usect "sample", 20
y .usect "result", 2
sect "coefficient"
a0: .word 0x0101
.word 0x0202
... .word 0x2020
.sect "code"
SOP: .SM 0
MOVW DP, #y
MOV L XAR1, #x
MOV L XAR7, #a0
ZAPA
RPT #19
MAC P, *XAR1++, *XAR7++
ADDL ACC, P<<PM
ADDL @y, ACC
B SOP, UNC

ACC++ = P
T = *ARn++
P = T*(*ARn++)
Data Move

Data Move Instructions

<table>
<thead>
<tr>
<th>DATA ↔ DATA (4G ↔ 64K)</th>
<th>DATA ↔ PGM (4G ↔ 4M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV loc16, *(0:16bit)</td>
<td>PREAD loc16, *XAR7</td>
</tr>
<tr>
<td>MOV *(0:16bit), loc16</td>
<td>PWRITE *XAR7, loc16</td>
</tr>
</tbody>
</table>

- 16-bit address concatenated with 16 leading zeros
- 32-bit address memory location
- Pointer with a 22-bit program memory address

- Faster than Load / Store, avoids accumulator
- Allows access to program memory
- Optimal with RPT (speed and code size)
- In RPT, non-mem address is auto-incremented in PC

Conditional Moves

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Execution (if COND is met)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV loc16, AX, COND</td>
<td>[loc16] = AX</td>
</tr>
<tr>
<td>MOVB loc16, #8bit, COND</td>
<td>[loc16] = 8bit</td>
</tr>
</tbody>
</table>

Example

If A<B, Then B=A

| A  | .usect "var", 2, 1 |
| B  | .set A+1           |
|    | .sect "code"       |
| MOVW DP, #A               |
| MOV AL, @A                |
| CMP AL, @B                |
| MOV @B, AL, LT            |

Accumulator

0 0 0 0 0 1 2 0

Data Memory

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 0</td>
<td>A</td>
</tr>
<tr>
<td>0 3 2 0</td>
<td>B</td>
</tr>
</tbody>
</table>

The conditional move instruction is an excellent way to avoid a discontinuity (branch or call) based upon a condition code set prior to the instruction. In the above example, the 1st step is to
place the contents of A into the accumulator. Once the Ax content is tested, by using the CMP
instruction, the conditional move can be executed.

If the specified condition being tested is true, then the location pointed to by the “loc16” addressing mode or the 8–bit zero extended constant will be loaded with the contents of the specified AX register (AH or AL):

\[
\text{if (COND == true) [loc16] = AX or 0:8bit;}
\]

**Note:** Addressing modes are not conditionally executed. Hence, if an addressing mode performs a pre or post modification, it will execute regardless if the condition is true or not. This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

**Flags and Modes**

**N** - If the condition is true, then after the move, AX is tested for a negative condition. The negative flag bit is set if bit 15 of AX is 1, otherwise it is cleared.

**Z** - If the condition then after the move, AX is tested for a zero condition. The zero flag bit is set if AX = 0, otherwise it is cleared.

**V** - If the V flag is tested by the condition, then V is cleared.

**C-Example**

; if ( VarA > 20 )
; VarA = 0;

CMP @VarA,#20 ; Set flags on (VarA – 20)
MOV @VarA,0,GT ; Zero VarA if greater then
Logical Operations

Byte Operations and Addressing

### Byte Operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>AX</th>
<th>Byte Addressed</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVB AX.LSB, loc16</td>
<td>0000 0000</td>
<td>Byte AX</td>
</tr>
<tr>
<td>MOVB AX.MSB, loc16</td>
<td></td>
<td>No change AX</td>
</tr>
<tr>
<td>MOVB loc16, AX.LSB</td>
<td></td>
<td>No change loc16</td>
</tr>
<tr>
<td>MOVB loc16, AX.MSB</td>
<td></td>
<td>No change loc16</td>
</tr>
</tbody>
</table>

Byte = 1. Low byte for register addressing
2. Low byte for direct addressing
3. Selected byte for offset indirect addressing

For loc16 = *+XARn[Offset]

<table>
<thead>
<tr>
<th>Odd Offset</th>
<th>Even Offset</th>
<th>loc16</th>
</tr>
</thead>
</table>

### Byte Addressing

```
12  34  56  78
```

16 bit memory

```
03  56
34  04
07  12
```

**Example of Byte Un-Packing**

- MOVL XAR2, #MemA
- MOVB *+XAR2[1], AL.LSB
- MOVB *+XAR2[2], AL.MSB
- MOVB *+XAR2[5], AH.LSB
- MOVB *+XAR2[6], AH.MSB

**Example of Byte Packing**

- MOVL XAR2, #MemA
- MOVB AL.LSB, *+XAR2[1]
- MOVB AL.MSB, *+XAR2[2]
- MOVB AH.MSB, *+XAR2[7]
Test and Change Memory Instructions

The compare (CMPx) and test (Txxx) instructions allow the ability to test values in memory. The results of these operations can then trigger subsequent conditional branches. The CMPx instruction allows comparison of memory with respect to a specified constant value, while the Txxx instructions allow any single bit to be extracted to the test control (TC) field of status register 0. The contents of the accumulator can also be non-destructively analyzed to establish branching conditions, as seen below.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Execution</th>
<th>Affects</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBIT loc16,#(0-15)</td>
<td>ST0(TC) = loc16(bit_no)</td>
<td>TC</td>
</tr>
<tr>
<td>TSET loc16,#(0-15)</td>
<td>Test (loc16(bit)) then set bit</td>
<td>TC</td>
</tr>
<tr>
<td>TCLR loc16,#(0-15)</td>
<td>Test (loc16(bit)) then clr bit</td>
<td>TC</td>
</tr>
<tr>
<td>CMPB AX, #8bit</td>
<td>Test (AX - 8bit unsigned)</td>
<td>C,N,Z</td>
</tr>
<tr>
<td>CMP AX, loc16</td>
<td>Test (AX – loc16)</td>
<td>C,N,Z</td>
</tr>
<tr>
<td>CMP loc16,#16b</td>
<td>Test (loc16 - #16bit signed)</td>
<td>C,N,Z</td>
</tr>
<tr>
<td>CMPL ACC, @P</td>
<td>Test (ACC - P &lt;&lt; PM)</td>
<td>C,N,Z</td>
</tr>
</tbody>
</table>
Min/Max Operations

**MIN/MAX Operations**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX ACC,loc16</td>
<td>if ACC &lt; loc16, ACC = loc16</td>
</tr>
<tr>
<td></td>
<td>if ACC &gt;= loc16, do nothing</td>
</tr>
<tr>
<td>MIN ACC,loc16</td>
<td>if ACC &gt; loc16, ACC = loc16</td>
</tr>
<tr>
<td></td>
<td>if ACC &lt;= loc16, do nothing</td>
</tr>
<tr>
<td>MAXL ACC,loc32</td>
<td>if ACC &lt; loc32, ACC = loc32</td>
</tr>
<tr>
<td></td>
<td>if ACC &gt;= loc32, do nothing</td>
</tr>
<tr>
<td>MINL ACC,loc32</td>
<td>if ACC &gt; loc32, ACC = loc32</td>
</tr>
<tr>
<td></td>
<td>if ACC &lt;= loc32, do nothing</td>
</tr>
<tr>
<td>MAXCUL P,loc32 (for 64 bit math)</td>
<td>if P &lt; loc32, P = loc32</td>
</tr>
<tr>
<td></td>
<td>if P &gt;= loc32, do nothing</td>
</tr>
<tr>
<td>MINCUL P,loc32 (for 64 bit math)</td>
<td>if P &gt; loc32, P = loc32</td>
</tr>
<tr>
<td></td>
<td>if P &lt;= loc32, do nothing</td>
</tr>
</tbody>
</table>

Find the maximum 32-bit number in a table:

```
MOVL ACC,#0
MOVL XAR1,#table
RPT #(table_length – 1)
||
MAXL ACC,*XAR1++
```
Read Modify Write Operations

The accumulator (ACC) is the main working register for the C28x. It is the destination of all ALU operations except those, which operate directly on memory or registers. The accumulator supports single-cycle move, add, subtract and compare operations from 32-bit-wide data memory. It can also accept the 32-bit result of a multiplication operation. These one or two cycle operations are referred to as read-modify-write operations, or as atomic instructions.

### Read-Modify-Write Instructions

- **Work directly on memory – bypass ACC**
- **Atomic Operations – protected from interrupts**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND loc16,AX</td>
<td>AND loc16,#16b</td>
<td>16-bit constant</td>
</tr>
<tr>
<td>OR loc16,AX</td>
<td>OR loc16,#16b</td>
<td>16-bit constant</td>
</tr>
<tr>
<td>XOR loc16,AX</td>
<td>XOR loc16,#16b</td>
<td>16-bit constant</td>
</tr>
<tr>
<td>ADD loc16,AX</td>
<td>ADD loc16,#16b</td>
<td>16-bit constant</td>
</tr>
<tr>
<td>SUB loc16,AX</td>
<td>SUB loc16,#16b</td>
<td>16-bit constant</td>
</tr>
<tr>
<td>SUBR loc16,AX</td>
<td>SUBR loc16,#16b</td>
<td>16-bit constant</td>
</tr>
<tr>
<td>INC loc16</td>
<td>TSET loc16,#bit</td>
<td>16-bit constant</td>
</tr>
<tr>
<td>DEC loc16</td>
<td>TCLR loc16,#bit</td>
<td>16-bit constant</td>
</tr>
</tbody>
</table>

- **AH, AL**

---

**C - 18 C28x - Appendix C - Assembly Programming**
### Read-Modify-Write Examples

<table>
<thead>
<tr>
<th>VarA += VarB</th>
<th>VarA += 100</th>
<th>VarA += 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETC INTM</td>
<td>SETC INTM</td>
<td>SETC INTM</td>
</tr>
<tr>
<td>MOV AL, @VarB</td>
<td>MOV AL, @VarA</td>
<td>MOV AL, @VarA</td>
</tr>
<tr>
<td>ADD AL, @VarA</td>
<td>ADD AL, #100</td>
<td>ADD AL, #1</td>
</tr>
<tr>
<td>MOV @VarA, AL</td>
<td>MOV @VarA, AL</td>
<td>MOV @VarA, AL</td>
</tr>
<tr>
<td>CLRC INTM</td>
<td>CLRC INTM</td>
<td>CLRC INTM</td>
</tr>
</tbody>
</table>

- **update with a mem**
- **update with a constant**
- **update by 1**

**Benefits of Read-Modify-Write Instructions**
Lab C: Assembly Programming

Objective

The objective of this lab is to practice and verify the mechanics of performing assembly language programming arithmetic on the TMS320C28x. In this process we will expand upon the .asm file from the previous lab to include new functions. Code will be added to obtain the sum of the products of the values from each array.

Perform the sum of products using a MAC-based implementation. In a real system application, the coeff array may well be constant (values do not change), therefore one can modify the initialization routine to skip the transfer of this arrays, thus reducing the amount of data RAM and cycles required for initialization. Also, there is no need to copy the zero to clear the result location. The initialization routine from the previous lab using the load/store operation will be replaced with a looped BANZ implementation.

As in previous lab, consider which addressing modes are optimal for the tasks to be performed. You may perform the lab based on this information alone, or may refer to the following procedure.

Procedure

Copy Files, Create Project File

1. Create a new project called LabC.pjt in C:\C28x\Labs\Appendix\LabC and add LabC.asm and LabC.cmd to it. Check your file list to make sure all the files are there. Be sure to setup the Build Options by clicking: Project → Build Options on the menu bar. Select the Linker tab. In the middle of the screen select ”No Autoinitialization” under ”Autoinit Model:”. Create a map file by typing .\Debug\LabC.map in the Map Filename [-m] field. Enter start in the ”Code Entry Point (-e):” field. Next, select the Compiler tab. Note that ”Full Symbolic Debug (-g)” under ”Generate Debug Info:” is selected. Then select OK to save the Build Options.

Initialization Routine using BANZ

2. Edit LabC.asm and modify it by replacing the initialization routine using the load/store operation with a BANZ process. Remember, it is only necessary to copy the first four values (i.e. initialize the data array). Do you still need the coeff array in the vars section?

3. Save your work. If you would like, you can use Code Composer Studio to verify the correct operation of the block initialization before moving to the next step.

Sum of Products using a RPT/MAC-based Implementation

4. Edit LabC.asm to add a RPT/MAC-based implementation to multiply the coeff array by the data array and storing the final sum-of-product value to result.
Build and Load

5. Click the “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 Studio, load the output file onto the target. Click: File → Load Program...

If you wish, right click on the source window and select Mixed Mode to debug using both source and assembly.

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

Optional Exercise

After completing the above, edit LabC.asm and modify it to perform the initialization process using a RTP/PREAD rather than a load/store/BANZ.

End of Exercise
OPTIONAL Lab C-C: Sum-of-Products in C

- **Objective**

The objective of this lab is to practice and verify the mechanics of performing C programming arithmetic on the TMS320C28x. The objective will be to add the code necessary to obtain the sum of the products of the n-th values from each array.

- **Procedure**

**Create Project File**

1. In Code Composer Studio create a new project called LabC-C.pjt in C:\C28x\Labs\Appendix\LabC\LabC-C and add LabC-C.c, LabC-C.cmd and C:\ti\c2000\cgtools\lib\rts2800_ml.lib to it. Check your file list to make sure all the files are there. *Do not* setup any Build Options. The default values will be used. In Appendix Lab D exercise, we will experiement and explore the various build options when working with C.

**Sum of Products using a MAC-based Implementation**

2. Edit LabC-C.c and modify the “main” routine to perform a MAC-based implementation in C. Since the MAC operation requires one array to be in program memory, the initialization routine can skip the transfer of one of the arrays, thus reducing the amount of data RAM and cycles required for initialization.

**Build and Load**

3. Click the “Rebuild All” button and watch the tools run in the build window. Debug as necessary.

---

**Note:** Have Code Composer Studio automatically load the output file after a successful build. On the menu bar click: Option Æ Customize... and select the “Program Load Options” tab, check “Load Program After Build”, then click OK.

4. Under Debug on the menu bar click “Go Main”. Single-step your routine. While single-stepping, open memory windows to see the values located in table [9] and data [9]. (Note: data[9] consists of the allocated arrays of data, coeff, and result). Open the CPU Registers. Check to see if the program is working as expected. Debug and modify, if needed.

---

**End of Exercise**
Appendix D – C Programming

Introduction

The C28x architecture, hardware, and compiler has been designed to efficiently support C code programming.

Appendix D will focus on how to program in C for an embedded system. Issues related to programming in C and how C behaves in the C28x environment will be discussed. Also, the C compiler optimization features will be explained.

Learning Objectives

- Learn the basic C environment for the C28x family
- How to control the C environment
- How to use the C-compiler optimizer
- Discuss the importance of volatile
- Explain optimization tips
Module Topics

Appendix D – C Programming

Module Topics..........................................................................................................................D-1

Linking Boot code from RTS2800.lib ..................................................................................D-2
Set up the Stack ..................................................................................................................D-4
C28x Data Types ..............................................................................................................D-5
Accessing Interrupts / Status Register .............................................................................D-6
Using Embedded Assembly ...............................................................................................D-7
Using Pragma .....................................................................................................................D-8
Optimization Levels ..........................................................................................................D-9
  Volatile Usage ...............................................................................................................D-11
  Compiler Advanced Options ..........................................................................................D-12
  Optimization Tips Summary ..........................................................................................D-13
Lab D: C Optimization ......................................................................................................D-14

OPTIONAL Lab D2: C Callable Assembly ..........................................................................D-17
Solutions ............................................................................................................................D-20
The boot routine is used to establish the environment for C before launching main. The boot routine begins with the label _c_int00 and the reset vector should contain a "long" to this address to make boot.asm the reset routine. The contents of the boot routine have been extracted and copied on the following page so they may be inspected. Note the various functions performed by the boot routine, including the allocation and setup of the stack, setting of various C-requisite statuses, the initialization of global and static variables, and the call to main. Note that if the link was performed using the "–cr" option instead of the "–c" option that the global/static variable initialization is not performed. This is useful on RAM-based C28x systems that were initialized during reset by some external host processor, making transfer of initialization values unnecessary. Later on in this chapter, there is an example on how to do the vectors in C code rather than assembly.
Set up the Stack

The Stack

The C/C++ compiler uses a stack to:

- Allocate local variables
- Pass arguments to functions
- Save the processor status
- Save the function return address
- Save temporary results

The compiler uses the hardware stack pointer (SP) to manage the stack.

SP defaults to 0x400 at reset. The run-time stack grows from low addresses to higher addresses.

The C28x has a 16-bit stack pointer (SP) allowing accesses to the base 64K of memory. The stack grows from low to high memory and always points to the first unused location. The compiler uses the hardware stack pointer (SP) to manage the stack. The stack size is set by the linker.

Setting Up the Stack

Linker command file:

```
SECTIONS       {
    .stack :>  RAM  align=2
    ... }
```

Note: The compiler provides no means to check for stack overflow during compilation or at runtime. A stack overflow disrupts the run-time environment, causing your program to fail. Be sure to allow enough space for the stack to grow.

In order to allocate the stack the linker command file needs to have “align = 2.”
# C28x Data Types

## C28x C-Language Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Bit</th>
<th>Value Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>16</td>
<td>Usually 0 .. 255, but can hold 16 bits</td>
</tr>
<tr>
<td>int (natural size CPU word)</td>
<td>16</td>
<td>-32K .. 32K, 16 bits signed</td>
</tr>
<tr>
<td>unsigned int</td>
<td>16</td>
<td>0 .. 64K, 16 bits unsigned</td>
</tr>
<tr>
<td>short (same as int or smaller)</td>
<td>16</td>
<td>same as int</td>
</tr>
<tr>
<td>unsigned short</td>
<td>16</td>
<td>same as unsigned int</td>
</tr>
<tr>
<td>long (same as int or larger)</td>
<td>32</td>
<td>-2M .. 2M, 32 bits signed</td>
</tr>
<tr>
<td>unsigned long</td>
<td>32</td>
<td>0 .. 4M, 32 bits unsigned</td>
</tr>
<tr>
<td>float</td>
<td>32</td>
<td>IEEE single precision</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
<td>IEEE double precision</td>
</tr>
<tr>
<td>long double</td>
<td>64</td>
<td>IEEE double precision</td>
</tr>
</tbody>
</table>

Data which is 32-bits wide, such as longs, must begin on even word-addresses (i.e. 0x0, 0x2, etc). This can result in “holes” in structures allocated on the stack. Suggestion: Group all longs together, group all pointers together
## Accessing Interrupts / Status Register

### Initializing

```c
extern cregister volatile unsigned int IFR;
extern cregister volatile unsigned int IER;

IER &= ~Mask;   //clear desired bits
IER |= Mask;     //set desired bits
IFR = 0x0000;   //clear prior interrupts
```

- Interrupt Enable & Interrupt Flag Registers (IER, IFR) are not memory mapped
- Only limited instructions can access IER & IFR (more in interrupt chapter)
- The compiler provides extern variables for accessing the IER & IFR
Using Embedded Assembly

Embedding Assembly in C

- Allows direct access to assembly language from C
- Useful for operating on components not used by C, ex:

  ```
  asm ( "CLRC INTM ; enable global interrupt”);
  
  #define EINT asmasm ( "CLRC INTM")
  ```

- Note: first column after leading quote is *label* field - if no label, should be blank space.
- Avoid modifying registers used by C
- Lengthy code should be written in ASM and called from C
  - main C file retains portability
  - yields more easily maintained structures
  - eliminates risk of interfering with registers in use by C

The assembly function allows for C files to contain 28x assembly code. Care should be taken not to modify registers in use by C, and to consider the label field with the assembly function. Also, any significant amounts of assembly code should be written in an assembly file and called from C.

There are two examples in this slide – the first one shows how to embed a single assembly language instruction into the C code flow. The second example shows how to define a C term that will invoke the assembly language instruction.
Using Pragma

Pragma is a preprocessor directive that provides directions to the compiler about how to treat a particular statement. The following example shows how the DATA_SECTION pragma is used to put a specific buffer into a different section of RAM than other buffers.

The example shows two buffers, bufferA and bufferB. The first buffer, bufferA is treated normally by the C compiler by placing the buffer (512 words) into the "bss" section. The second, bufferB is specifically directed to go into the “my_sect” portion of data memory. Global variables, normally "bss", can be redirected as desired.

When using CODE_SECTION, code that is normally linked as "text", can be identified otherwise by using the code section pragma (like .sect in assembly).

Pragma Examples

- User defined sections from C:

  ```
  #pragma CODE_SECTION (func, "section name")
  #pragma DATA_SECTION (symbol, "section name")
  ```

- Example - using the DATA_SECTION Pragma

  - C source file

  ```
  char bufferA[512];
  #pragma DATA_SECTION(bufferB, "my_sect")
  char bufferB[512];
  ```

  - Resulting assembly file

  ```
  .global _bufferA, _bufferB
  .bss _bufferA,512
  _bufferB: .usect "my_sect",512
  ```

More #pragma are defined in the C compiler UG
Optimization Levels

Optimizations fall into 4 categories. This is also a methodology that should be used to invoke the optimizations. It is recommended that optimization be invoked in steps, and that code be verified before advancing to the next step. Intermediate steps offer the gradual transition from fully symbolic to fully optimized compilation. Compiler switched may be invoked in a variety of ways.

Here are 4 steps that could be considered:

1st: use –g
   By starting out with –g, you do no optimization at all and keep symbols for debug.

2nd: use –g –o3
   The option –o3 might be too big a jump, but it adds the optimizer and keeps symbols.

3rd: use –g –o3 –mn
   This is a full optimization, but keeps some symbols.

4th: use –o3
   Full optimization, symbols are not kept.
Optimizer levels zero through three, offer an increasing array of actions, as seen above. Higher levels include all the functions of the lower ones. Increasing optimizer levels also increase the scope of optimization, from considering the elements of single entry, single-exit functions only, through all the elements in a file. The “-pm” option directs the optimizer to view numerous input files as one large single file, so that optimization can be performed across the whole system.
Volatile Usage

Optimization Issue: “Volatile” Variables

Problem: The compiler does not know that this pointer may refer to a hardware register that may change outside the scope of the C program. Hence it may be eliminated (optimized out of existence!)

Wrong: Wait loop for a hardware signal

```c
unsigned int *CTRL
while (*CTRL !=1);
```

Solution:

```c
volatile unsigned int *CTRL
while (*CTRL !=1);
```

- When using optimization, it is important to declare variables as `volatile` when:
  - The memory location may be modified by something other than the compiler (e.g. it’s a memory-mapped peripheral register).
  - The order of operations should not be rearranged by the compiler
- Define the pointer as “volatile” to prevent the optimizer from optimizing
Compiler Advanced Options

To get to these options, go to Project → Build Options in Code Composer Studio.

In the category, pick Advanced.

The first thing to notice under advanced options is the **Auto Inlining Threshold**.
- Used with –o3 option
- Functions > size are not auto inlined

**Note:** To prevent code size increases when using –o3, disable auto inlining with -oi0

The next point we will cover is the **Normal Optimization with Debug (-mn)**.
- Re-enables optimizations disabled by “–g” option (symbolic debug)
- Used for maximum optimization

**Note:** Some symbolic debug labels will be lost when –mn option is used.

Optimizer should be invoked incrementally:
- `g test` Symbols kept for debug
- `g -o3 test` Add optimizer, keep symbols
- `g -o3 -mn test` More optimize, some symbols
- `o3 test` Final rev: Full optimize, no symbols

[-mf] : Optimize for speed instead of the default optimization for code size

[-mi] : Avoid RPT instruction. Prevent compiler from generating RPT instruction. RPT instruction is not interruptible

[-mt] : Unified memory model. Use this switch with the unified memory map of the 2810 & 2812. Allows compiler to generate the following:
- RPT PREAD for memory copy routines or structure assignments
- MAC instructions
- Improves efficiency of switch tables
Optimization Levels

Optimization Tips Summary

<table>
<thead>
<tr>
<th>Summary: Optimization Tips</th>
</tr>
</thead>
<tbody>
<tr>
<td>◆ Within C functions:</td>
</tr>
<tr>
<td>➢ Use const with variables for parameter constants</td>
</tr>
<tr>
<td>➢ Minimize mixing signed &amp; unsigned ops: SXM changes</td>
</tr>
<tr>
<td>➢ Keep frames &lt;= 64 (locals + parameters + PC): *-SP[6bit]</td>
</tr>
<tr>
<td>➢ Use structures &lt;= 8 words: use 3 bit index mode</td>
</tr>
<tr>
<td>➢ Declare longs first, then declare ints: minimize stack holes</td>
</tr>
<tr>
<td>➢ Avoid: long = (int * int): yields unpredictable results</td>
</tr>
<tr>
<td>◆ Optimizing: Use -o0, -o1, -o2, -o3 when compiling</td>
</tr>
<tr>
<td>➢ Inline short/key functions</td>
</tr>
<tr>
<td>➢ Pass inlines between files: static inlines in header files</td>
</tr>
<tr>
<td>➢ Invoke automatic inlining: -o3 -oi</td>
</tr>
<tr>
<td>➢ Give compiler project visibility: use -pm and -o3</td>
</tr>
<tr>
<td>◆ Tune memory map via linker command file</td>
</tr>
<tr>
<td>◆ Re-write key code segments to use intrinsics or in assembly</td>
</tr>
</tbody>
</table>

App notes  3rd Parties

The list above documents the steps that can be taken to achieve increasingly higher coding efficiency. It is recommended that users first get their code to work with no optimization, and then add optimizations until the required performance is obtained.
Lab D: C Optimization

**Objective**

The objective of this lab is to practice and verify the mechanics of optimizing C programs. Using Code Composer Studio profile capabilities, different routines in a project will be benchmarked. This will allow you to analyze the performance of different functions. This lab will highlight the profiler and the clock tools in CCS.

**Procedure**

**Create Project File**

1. Create a new project in C:\C28x\Labs\Appendix\LabD called LabD.pjt and add LabD.c, LabD.cmd, and sop-c.c to it. (Note that sop-asm.asm will be used in the next part of the lab, and should not be added now). Also, add the compiler runtime support library to the project (C:\ti\c2000\cgtools\lib\rts2800_ml.lib).

2. Setup the Build Options. Select the Linker tab and in the middle of the screen select “Run-time Autoinitialization” under “Autoinit Model:”. Create a map file by typing \Debug\LabD.map in the Map Filename [-m] field. Do not enter anything in the “Code Entry Point (-e):” field (leave it blank). Set the stack size to 0x400. Next, select the Compiler tab. Note that “Full Symbolic Debug (-g)” under “Generate Debug Info:” in the Basic Category is selected. On the Feedback Category pull down the interlisting options and select “C and ASM (-ss)”. On the Assembly Category check the Keep generated .asm Files (-k), Keep Labels as Symbols(-as) and Generate Assembly Listing Files (-al). The –as will allow you to see symbols in the memory window and the –al will generate an assembly listing file (.lst file). The listing file has limited uses, but is sometime helpful to view opcode values and instruction sizes. (The .lst file can be viewed with the editor). Both of these options will help with debugging. Then select OK to save the Build Options.

**Build and Load**

3. Click the “Rebuild All” button and watch the tools run in the build window. Be sure the “Load program after build” option is selected in Code Composer Studio. The output file should automatically load. The Program Counter should be pointing to _c_int00 in the Disassembly Window.

**Set Up the Profile Session**

4. Restart the DSP (debug \ restart) and then “Go Main”. This will run through the C initialization routine in Boot.asm and stop at the main routine in LabD.c.

5. Set a breakpoint on the NOP in the while(1) loop at the end of main() in LabD.c.
6. Set up the profile session by selecting Profiler ➔ Start New Session. Enter a session name of your choice (i.e. LabD).

7. In the profiler window, hover the mouse over the icons on the left region of the window and select the icon for Profile All Functions. Click on the “+” to expand the functions. Record the “Code Size” of the function sop C code in the table at the end of this lab. Note: If you do not see a “+” beside the .out file, press “Profile All Functions” on the horizontal tool bar. (You can close the build window to make the profiler window easier to view by right clicking on the build window and selecting “hide”).

8. Select F5 or the run icon. Observe the values present in the profiling window. What do the numbers mean? Click on each tab to determine what each displays.

**Benchmarking Code**

9. Let’s benchmark (i.e. count the cycles need by) only a portion of the code. This requires you to set a breakpoint pair on the starting and ending points of the benchmark. Open the file sop-c.c and set a breakpoint on the “for” statement and the “return” statement.

10. In CCS, select profiler ➔ enable clock (must be checked). Then select profiler ➔ view clock.

11. Now “Restart” the program and then “Run” the program. The program should be stopped at the first breakpoint in sop. Double click on the clock window to set the clock to zero. Now you are ready to benchmark the code. “Run” to the second breakpoint. The number of cycles are displayed in the clock window. Record this value in the table at the end of the lab under “C Code - Cycles”.

**C Optimization**

12. To optimize C code to the highest level, we must set up new Build Options for our Project. Select the Compiler tab. In the Basic Category Panel, under “Opt Level” select File (-o3). Then select OK to save the Build Options.

13. Now “Rebuild” the program and then “Run” the program. The program should be stopped at the first breakpoint in sop. Double click on the clock window to set the clock to zero. Now you are ready to benchmark the code. “Run” to the second breakpoint. The number of cycles are displayed in the clock window. Record this value in the table at the end of the lab under “Optimized C (-o3) - Cycles”.

14. Look in your profile window at the code size of sop. Record this value in the table at the end of this lab.

**Benchmarking Assembly Code**

15. Remove sop-c.c from your project and replace it with sop-asm.asm. Rebuild and set breakpoints at the beginning and end of the assembly code (MOVL & LRETR).

16. Start a new profile session and set it to profile all functions. Run to the first breakpoint and study the profiler window. Record the code size of the assembly code in the table.
17. Double Click on the clock to reset it. Run to the last breakpoint. Record the number of cycles the assembly code ran.

18. How does assembly, C code, and optimized C code compare on the C28x?

<table>
<thead>
<tr>
<th></th>
<th>C Code</th>
<th>Optimized C Code (-o3)</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Size</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycles</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

End of Exercise
OPTIONAL Lab D2: C Callable Assembly

Objective

The objective of this lab is to practice and verify the mechanics of implementing a C callable assembly programming. In this lab, a C file will be used to call the sum-of-products (from the previous Appendix LabC exercise) by the “main” routine. Additionally, we will learn how to use Code Composer Studio to configure the C build options and add the run-time support library to the project. As in previous labs, you may perform the lab based on this information alone, or may refer to the following procedure.

Procedure

Copy Files, Create Project File

1. Create a new project in C:\C28x\Labs\Appendix\LabD2 called LabD2.pjt and add LabD2.c, LabD2.cmd, and sop-c.c to it. Also, add the compiler run-time support library to the project (C:\ti\c2000\cgtools\lib\rts2800_ml.lib).

2. Do not add LabC.asm to the project (copy of file from Appendix Lab C). It is only placed here for easy access. Parts of this file will be used later during this lab exercise.

3. Setup the Build Options. Select the Linker tab and in the middle of the screen select “Run-time Autoinitilization” under “Autoinit Model:”. Create a map file by typing \Debug\LabD2.map in the Map Filename [-m] field. Do not enter anything in the “Code Entry Point (-e)” field (leave it blank). Set the stack size to 0x400. Next, select the Compiler tab. Note that “Full Symbolic Debug (-g)” under “Generate Debug Info:” in the Basic Category is selected. On the Feedback Category pull down the interlisting options and select “C and ASM (-ss)”. On the Assembly Category check the Keep generated .asm Files (-k), Keep Labels as Symbols (-as) and Generate Assembly Listing Files (-al). The –as will allow you to see symbols in the memory window and the –al will generate an assembly listing file (.lst file). The listing file has limited uses, but is sometime helpful to view opcode values and instruction sizes. (The .lst file can be viewed with the editor). Both of these options will help with debugging. Then select OK to save the Build Options.

Build and Load

4. Click the “Rebuild All” button and watch the tools run in the build window. Be sure the “Load program after build” option is selected in Code Composer Studio. The output file should automatically load. The Program Counter should be pointing to _c_int00 in the Disassembly Window.

5. Under Debug on the menu bar click “Go Main”. This will run through the C initialization routine in Boot.asm and stop at the main routine in LabD2.c.
Verify C Sum of Products Routine

6. Debug using both source and assembly (by right clicking on the window and select Mixed Mode or using View → Mixed Source/ASM).

7. Open a memory window to view result and data.

8. Single-step through the C code to verify that the C sum-of-products routine produces the results as your assembly version.

Viewing Interlisted Files and Creating Assembly File

9. Using File → Open view the LabD2.asm and sop-c.asm generated files. The compiler adds many items to the generated assembly file, most are not needed in the C-callable assembly file. Some of the unneeded items are .func / .endfunc .sym, and .line.

10. Look for the _sop function that is generated by the compiler. This code is the basis for the C-callable assembly routine that is developed in this lab. Notice the comments generated by the compiler on which registers are used for passing parameters. Also, notice the C code is kept as comments in the interlisted file.

11. Create a new file (File → New, or clicking on the left most button on the horizontal toolbar “New”) and save it as an assembly source file with the name sop-asm.asm. Next copy ONLY the sum of products function from LabC.asm into this file. Add a _sop label to the function and make it visible to the linker (.def). Also, be sure to add a .sect directive to place this code in the “code” section. Finally, add the following instruction to the end:

   LRETR ; return statement

12. Next, we need to add code to initialize the sum-of-products parameters properly, based on the passed parameters. Add the following code to the first few lines after entering the _sop routine: (Note that the two pointers are passed in AR4 and AR5, but one needs to be placed in AR7. The loop counter is the third argument, and it is passed in the accumulator.)

   MOV L XAR7,XAR5 ;XAR7 points to coeff [0]
   MOV AR5,AL ;move n from ACC to AR5 (loop counter)
   SUBB XAR5,#1 ;subtract 1 to make loop counter = n-1

Before beginning the MAC loop, add statements to set the sign extension mode, set the SPM to zero, and a ZAPA instruction. Use the same MAC statement as in Lab 4, but use XAR4 in place of XAR2. Make the repeat statement use the passed value of n-1 (i.e. AR5).

   RPT AR5 ;repeat next instruction AR5 times
Now we need to return the result. To return a value to the calling routine you will need to place your 32-bit value in the ACC. What register is the result currently in? Adjust your code, if necessary.

13. Save the assembly file as sop-asm.asm. (Do not name it LabD2.asm because the compiler has already created with that name from the original LabD2.c code).

Defining the Function Prototype as External

14. Note in LabD2.c an “extern” modifier is placed in front of the sum-of-products function prototype:

   extern int sop(int*,int*,int); //sop function prototype

Verify Assembly Sum of Products Routine

15. Remove the sop-c.c file from the project and add the new sop-asm.asm assembly file to the project.

16. Rebuild and verify that the new assembly sum-of-products routine produces the same results as the C function.

End of Exercise
## Solutions

### Lab D Solutions

<table>
<thead>
<tr>
<th></th>
<th>C Code</th>
<th>Optimized C Code (-o3)</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Size</td>
<td>27</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>Cycles</td>
<td>118</td>
<td>32</td>
<td>22</td>
</tr>
</tbody>
</table>