Important Notice

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

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

Customers are responsible for their applications using TI components.

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

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

Copyright © 2009 Texas Instruments Incorporated

Revision History

September 2009 – Revision 1.0

Mailing Address

Texas Instruments
Training Technical Organization
7839 Churchill Way
M/S 3984
Dallas, Texas 75251-1903
C2000™ Piccolo™ Workshop

Introductions

- Name
- Company
- Project Responsibilities
- DSP / Microcontroller Experience
- TMS320 Processor Experience
- Hardware / Software - Assembly / C
- Interests
C2000™ Piccolo™ Workshop Outline

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

C2000™ Experimenter Kit

Piccolo™ Experimenter Kit

ControlCARD

USB Docking Station
Introduction

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

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

Learning Objectives

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

Learning Objectives

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

## Architecture Overview

<table>
<thead>
<tr>
<th>Module Topics</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is the TMS320C2000™?</td>
<td>1-3</td>
</tr>
<tr>
<td>TMS320C2000™ Internal Bussing</td>
<td>1-4</td>
</tr>
<tr>
<td>F28x CPU</td>
<td>1-5</td>
</tr>
<tr>
<td>Special Instructions</td>
<td>1-6</td>
</tr>
<tr>
<td>Pipeline Advantage</td>
<td>1-7</td>
</tr>
<tr>
<td>Memory</td>
<td>1-8</td>
</tr>
<tr>
<td>Memory Map</td>
<td>1-8</td>
</tr>
<tr>
<td>Code Security Module (CSM)</td>
<td>1-9</td>
</tr>
<tr>
<td>Peripherals</td>
<td>1-9</td>
</tr>
<tr>
<td>Fast Interrupt Response</td>
<td>1-10</td>
</tr>
<tr>
<td>F28x Mode</td>
<td>1-11</td>
</tr>
<tr>
<td>Reset</td>
<td>1-12</td>
</tr>
<tr>
<td>Summary</td>
<td>1-13</td>
</tr>
</tbody>
</table>
What is the TMS320C2000™?

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

The F2803x architecture can be divided into 3 functional blocks:

- CPU and busing
- Memory
- Peripherals
**TMS320C2000™ Internal Bussing**

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

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

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

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

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

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

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

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

**F28x Atomic Read/Modify/Write**

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

---

**Standard Load/Store**

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

6 words / 6 cycles

---

**Atomic Read/Modify/Write**

- AND *XAR2, #1234h

2 words / 1 cycles

---

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

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

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

Memory Map

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

This memory map shows the different blocks of memory available to the program and data space.
**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 8 cycles at 60 MHz, it would take at least $1.4 \times 10^{24}$ years to try all possible combinations!

**Peripherals**

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

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

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

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

### F28x Operating Modes

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

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

Reset – Bootloader

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

Reset vector
OBJMODE = 0
AMODE = 0

Bootloader sets
OBJMODE = 1
AMODE = 0

Emulator
Connected?

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

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

Wait

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

TRST = JTAG Test Reset
EMU_KEY & EMU_BMODE located in PIE at 0x0D00 & 0x0D01, respectively
Summary

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

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

Learning Objectives

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

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

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

Solutions .................................................................................................................... 2-22
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:
- Integrated Edit/Debug GUI
- Code Generation Tools
- DSP/BIOS

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

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

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

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

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

### Code Composer Studio: IDE

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

---

C2000 Piccolo Workshop - Programming Development Environment
Projects

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

The CCS Project

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

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

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

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

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

There is a one-to-one relationship between the items in the text box and the GUI check and dropdown 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.
For new projects, CCS automatically creates two build configurations:

- **Debug** (unoptimized)
- **Release** (optimized)

Use the drop-down menu to quickly select the build configuration.

Add/Remove your own custom build configurations using **Project Configurations**

Edit a configuration:
1. Set it active
2. Modify build options
3. Save project

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

Sections

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

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

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

<table>
<thead>
<tr>
<th>Compiler Section Names</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initialized Sections</strong></td>
</tr>
<tr>
<td>.text</td>
</tr>
<tr>
<td>.cinit</td>
</tr>
<tr>
<td>.econst</td>
</tr>
<tr>
<td>.switch</td>
</tr>
<tr>
<td>.pinit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Uninitialized Sections</strong></th>
<th><strong>Description</strong></th>
<th><strong>Link Location</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>.ebss</td>
<td>global and static variables</td>
<td>RAM</td>
</tr>
<tr>
<td>.stack</td>
<td>stack space</td>
<td>low 64Kw RAM</td>
</tr>
<tr>
<td>.esysmem</td>
<td>memory for far malloc functions</td>
<td>RAM</td>
</tr>
</tbody>
</table>

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

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

**Program Code (.text)**

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

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

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

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

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

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

The format is:  

\[ \text{Name: origin = 0x????, length = 0x????} \]

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

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

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

```
MEMORY
{
  M0SARAM: origin = 0x000000 , length = 0x04000
  M1SARAM: origin = 0x000400 , length = 0x0400
}
```
Remember that the DSP has two memory maps: *Program*, and *Data*. Therefore, the MEMORY description must describe each of these separately. The loader uses the following syntax to delineate each of these:

<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 Memory */
        FLASH: origin = 0x3E8000, length = 0x10000

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

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

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

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

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

```
Linker Command File

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

    PAGE 1:     /* Data Memory */
        M0SARAM: origin = 0x000000, length = 0x400
        M1SARAM: origin = 0x000400, length = 0x400
}
SECTIONS
{
    .text:>     FLASH       PAGE 0
    .ebss:>    M0SARAM     PAGE 1
    .cinit:>   FLASH       PAGE 0
    .stack:>   M1SARAM     PAGE 1
}
```
Exercise 2

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

Fill in the blanks:

Exercise 2 - Command File

```plaintext
MEMORY
{
    PAGE__: /* Program Memory */
    _____: origin = ______, length = _____
    _____: /* Data Memory */
    _____: origin = ______, length = _____
    _____: origin = ______, length = _____
    _____: origin = ______, length = _____
}

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

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

<table>
<thead>
<tr>
<th>Memory Map Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Name</td>
</tr>
<tr>
<td>• Location</td>
</tr>
<tr>
<td>• Size</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sections Description</th>
</tr>
</thead>
<tbody>
<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 2: Linker Command File

Objective

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

Lab 2: Linker Command File

System Description:
- TMS320F28035
- All internal RAM blocks allocated

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

Procedure

Create a New Project

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

Click: **Debug ➔ Connect**

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

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

**Project ➔ New**

type **Lab2** in the project name field and make sure the save in location is:

C:\C28x\Labs\Lab2, then click Finish. This will create a `.pjt` file which will invoke all the necessary tools (compiler, assembler, linker) to build your project. It will also create a `debug` folder that will hold immediate output files.

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

**Project ➔ Add Files to Project...**

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

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

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

**Project Build Options**

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

Click: **Project ➔ Build Options...**

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

9. Set the Stack Size to 0x200.

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

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

11. To open and edit **Lab2.cmd**, double click on the filename in the project window.
12. Edit the `Memory{}` declaration by describing the system memory shown on the “Lab2: Linker Command File” slide in the objective section of this lab exercise. Place the LOSARAM and L3DPSARAM memory blocks into program memory on page 0. Place the other memory blocks into data memory on page 1.

13. In the `Sections{}` area, notice that a section called `.reset` has already been allocated. The `.reset` section is part of the rts2800_ml.lib, and is not needed. By putting the `TYPE = DSECT` modifier after its allocation, the linker will ignore this section and not allocate it.

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

**Build and Load the Project**

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

<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>

16. Code Composer Studio can automatically load the output file after a successful build. On the menu bar click: Option → Customize... and select the “Program/Project/C10” tab, then check “Load Program After Build”.

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

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

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

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

20. Under Debug on the menu bar click “Go Main”. This will run through the C-environment initialization routine in the rts2800_ml.lib and stop at `main()` in Lab2.c.
Lab 2: Linker Command File

**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.

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

   Click: View → Memory on the menu bar.

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

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

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

   Click: View → Watch Window on the menu bar.

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

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

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

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

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

**Single-stepping the Code**

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

   **End of Exercise**
Exercise 2 - Solution

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

Lab 2: Solution - lab2.cmd

```plaintext
MEMORY
{
    PAGE 0: /* Program Memory */
    L0SARAM: origin = 0x008000, length = 0x0800
    L3DPSARAM: origin = 0x009000, length = 0x1000
    PAGE 1: /* Data Memory */
    MOSARAM: origin = 0x000000, length = 0x0400
    M1SARAM: origin = 0x000400, length = 0x0400
    L1DPSARAM: origin = 0x008800, length = 0x0400
    L2DPSARAM: origin = 0x008C00, length = 0x0400
}
SECTIONS
{
    .text: > LOSARAM PAGE = 0
    .ebss: > MOSARAM PAGE = 1
    .cinit: > LOSARAM PAGE = 0
    .stack: > M1SARAM PAGE = 1
    .reset: > LOSARAM PAGE = 0, TYPE = DSECT
}
```
Introduction

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

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

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

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

Learning Objectives

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

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

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

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

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

F2803x C-Code Header Files .............................................................................. 3-7
  Peripheral Structure .h File........................................................................ 3-7
  Global Variable Definitions File ................................................................. 3-9
  Mapping Structures to Memory ................................................................. 3-10
  Linker Command File........................................................................ 3-10
  Peripheral Specific Routines.................................................................... 3-11

Summary ............................................................................................................. 3-12
Traditional and Structure Approach to C Coding

Traditional Approach to C Coding

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

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

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

Structure Approach to C Coding

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

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

Disadvantages
- Can be difficult to remember the structure names (Editor Auto Complete feature to the rescue!)
- More to type (again, Editor Auto Complete feature to the rescue)
Traditional and Structure Approach to C Coding

The CCS Watch Window using #define

The CCS Watch Window using Structures
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, #0xFFE
```

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

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

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

---

Compare with the #define Approach

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

**C Source Code**

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

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

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

**Generated Assembly Code**

```assembly
MOV AL, *(0:0x0C04)
ORB AL, #0x10
MOV *(0:0x0C04), AL
MOVL XAR5, #0x010000
MOVL XAR4, #0x000C0A
MOVL *+XAR4[0], XAR5
MOV AL, *(0:0x0C04)
AND AL, #0xFFEF
MOV *(0:0x0C04), AL
```

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

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

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

Structure Naming Conventions

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

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

Notes:
[1] “PeripheralName” are assigned by TI and found in the DSP2803x header files. They are a combination of capital and small letters (i.e. CpuTimer0Regs).
[2] “RegisterName” are the same names as used in the data sheet. They are always in capital letters (i.e. TCR, TIM, TPR,..).
[3] “FieldName” are the same names as used in the data sheet. They are always in capital letters (i.e. POL, TOG, TSS,..).

Editor Auto Complete to the Rescue!
F2803x C-Code Header Files

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

**DSP2803x Header File Package**
(http://www.ti.com, literature # SPRC892)

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

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

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

```
GEL_LoadGel(“base_path/gel/DSP2803x_Peripheral.gel”)
```

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

File → Load GEL...

**Peripheral Structure .h File**

The DSP2803x_Device.h header file is the main include file. By including this file in the .c source code, all of the peripheral specific .h header files are automatically included. Of course, each specific .h header file can be included individually in an application that does not use all the header files, or you can comment out the ones you do not need. (Also includes typedef statements).
Peripheral Structure .h files (1 of 2)

- Contain bits field structure definitions for each peripheral register

```c
#include "DSP2803x_Device.h"

Void InitAdc(void)
{
    /* Reset the ADC module */
    AdcRegs.ADCCTL1.bit.RESET = 1;
    /* configure the ADC register */
    AdcRegs.ADCCTL1.all = 0x00E4;
};
```

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

```
// ADC Individual Register Bit Definitions:
struct ADCCTL1_BITS {        // bits description
    Uint16  TEMPCONV:1;       // 0 Temperature sensor connection
    Uint16  VREFLOCONV:1;   // 1 VSSA connection
    Uint16  INTPULSEPOS:1;   // 2 INT pulse generation control
    Uint16  ADCREFSEL:1;      // 3 Internal/external reference select
    Uint16  rsvd1:1;                  // 4 reserved
    Uint16  ADCREFPWD:1;     // 5 Reference buffers powerdown
    Uint16  ADCBGPWD:1;       // 6 ADC bandgap powerdown
    Uint16  ADCPWDN:1;          // 7 ADC powerdown
    Uint16  ADCBSYCHN:5;      // 12:8 ADC busy on a channel
    Uint16  ADCBSY:1;              // 13 ADC busy signal
    Uint16  ADENABLE:1;       // 14 ADC enable
    Uint16  RESET:1;                 // 15 ADC master reset
};

// Allow access to the bit fields or entire register:
union ADCCTL1_REG {
    Uint16                all;
    struct ADCCTL1_BITS   bit;
};

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

Peripheral Structure .h files (2 of 2)

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

<table>
<thead>
<tr>
<th>DSP2803x_Adcd.h</th>
<th>DSP2803x_BootVars.h</th>
<th>DSP2803x_Cla.h</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSP2803x_Comp.h</td>
<td>DSP2803x_CpuTimers.h</td>
<td>DSP2803x_DevEmu.h</td>
</tr>
<tr>
<td>DSP2803x_Device.h</td>
<td>DSP2803x_ECan.h</td>
<td>DSP2803x_ECap.h</td>
</tr>
<tr>
<td>DSP2803x_EPwm.h</td>
<td>DSP2803x_EQep.h</td>
<td>DSP2803x_Gpio.h</td>
</tr>
<tr>
<td>DSP2803x_I2c.h</td>
<td>DSP2803x_Lin.h</td>
<td>DSP2803x_Nmilntrupt.h</td>
</tr>
<tr>
<td>DSP2803x_PieCtrl.h</td>
<td>DSP2803x_PieVect.h</td>
<td>DSP2803x_Sci.h</td>
</tr>
<tr>
<td>DSP2803x_Spi.h</td>
<td>DSP2803x_SysCtrl.h</td>
<td>DSP2803x_Xintrupt.h</td>
</tr>
</tbody>
</table>

- **DSP2803x_Device.h**
  - Main include file
  - Will include all other .h files
  - Include this file (directly or indirectly) in each source file:
    
    ```
    #include "DSP2803x_Device.h"
    ```
Global Variable Definitions File

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

Global Variable Definitions File

DSP2803x_GlobalVariableDefs.c

◆ Declares a global instantiation of the structure for each peripheral

◆ Each structure is placed in its own section using a DATA_SECTION pragma to allow linking to the correct memory (see next slide)

DSP2803x_GlobalVariableDefs.c

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

◆ Add this file to your CCS project:

DSP2803x_GlobalVariableDefs.c
Mapping Structures to Memory

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

Linker Command Files for the Structures

DSP2803x_nonBIOS.cmd and DSP2803x_BIOS.cmd

- Links each structure to the address of the peripheral using the structures named section
- non-BIOS and BIOS versions of the .cmd file

Add one of these files to your CCS project:
DSP2803x_nonBIOS.cmd or DSP2803x_BIOS.cmd

Linker Command File

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

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

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

Peripheral Specific Examples

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

<table>
<thead>
<tr>
<th>adi_soc</th>
<th>epwm_up_ac</th>
<th>gina_sc_chochack</th>
</tr>
</thead>
<tbody>
<tr>
<td>adc_temp_sensor</td>
<td>epwm_updown_ac</td>
<td>gina_sc_looback_interrups</td>
</tr>
<tr>
<td>cla_adc</td>
<td>ecep_fregcal</td>
<td>gina_sc_loopback</td>
</tr>
<tr>
<td>cla_adc_fir</td>
<td>ecep_pos_speed</td>
<td>gina_startloopback</td>
</tr>
<tr>
<td>cla_adc_fir_fiash</td>
<td>external_interrupt</td>
<td>gina_standbyloopback</td>
</tr>
<tr>
<td>cpu_timer</td>
<td>flash</td>
<td>gina_stopback</td>
</tr>
<tr>
<td>envr_backback</td>
<td>gpio_setup</td>
<td>gina_loopback</td>
</tr>
<tr>
<td>ecep_capture_pwm</td>
<td>gpio_toggle</td>
<td>gina_loopback_interrups</td>
</tr>
<tr>
<td>epwm_blanking_pwm</td>
<td>hrown</td>
<td>gina_loopback</td>
</tr>
<tr>
<td>epwm_dcevent_trip</td>
<td>hrown_dutysfo_v6</td>
<td>gina_loopback</td>
</tr>
<tr>
<td>epwm_dcevent_trip_comp</td>
<td></td>
<td>gina_loopback</td>
</tr>
<tr>
<td>epwm_deadband</td>
<td>hrown_prdupsfo_v6</td>
<td>gina_loopback</td>
</tr>
<tr>
<td>epwm_dcevent_trip</td>
<td>hrown_prdupsfodosfo_v6</td>
<td>gina_loopback</td>
</tr>
<tr>
<td>epwm_dcevent_trip</td>
<td>hrown_slider</td>
<td>gina_loopback</td>
</tr>
<tr>
<td>epwm_dcevent_trip</td>
<td>i2c_eprom</td>
<td>gina_loopback</td>
</tr>
<tr>
<td>epwm_dcevent_trip</td>
<td>lina_external_loopback</td>
<td>gina_loopback</td>
</tr>
<tr>
<td>epwm_dcevent_trip</td>
<td></td>
<td>gina_loopback</td>
</tr>
</tbody>
</table>
Summary

Peripheral Register Header Files Summary

- Easier code development
- Easy to use
- Generates most efficient code
- Increases effectiveness of CCS watch window
- TI has already done all the work!
  - Use the correct header file package for your device:
    - F2803x # SPRC892
    - F2802x # SPRC832
    - F2833x and F2823x # SPRC530
    - F280x and F2801x # SPRC191
    - F2804x # SPRC324
    - F281x # SPRC097

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

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

Learning Objectives

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

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

        Module Topics......................................................................................... 4-2
        Reset.................................................................................................................. 4-3
        Reset - Bootloader .......................................................................................... 4-3
        Emulation Boot Mode ....................................................................................... 4-4
        Stand-Alone Boot Mode .................................................................................... 4-4
        Reset Code Flow – Summary ........................................................................... 4-5

        Interrupts ............................................................................................................. 4-6
        Interrupt Processing ........................................................................................... 4-6
        Interrupt Flag Register (IFR) ............................................................................. 4-7
        Interrupt Enable Register (IER) ......................................................................... 4-7
        Interrupt Global Mask Bit (INTM) .................................................................... 4-8
        Peripheral Interrupt Expansion (PIE) .................................................................. 4-8
        PIE Interrupt Vector Table ............................................................................... 4-10
        Interrupt Response and Latency ....................................................................... 4-11
## Reset

### Reset Sources

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

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

### Reset - Bootloader

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

- **Reset vector** fetched from boot ROM
  - 0xFFFF FFC0

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

- **Emulator Connected?**
  - YES
  - TRST = 1

- **Stand-alone Boot**
  - Boot determined by 2 GPIO pins and 2 OTP locations:
    - OPT_KEY and OTP_BMODE

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

**Emulator Connected?**

- **YES**
  - TRST = 1
- **NO**
  - TRST = 0

**TRST = JTAG Test Reset**

- EMU_KEY & EMU_BMODE located in RAM at 0x00D00 & 0x00D01, respectively
- OPT_KEY & OPT_BMODE located in OTP at 0x3D78FE & 0x3D78FF, respectively
Emulation Boot Mode

**Emulation Boot Mode** *(TRST = 1)*

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

---

**Stand-Alone Boot Mode**

**Stand-Alone Boot Mode** *(TRST = 0)*

Note that the boot behavior for unprogrammed OTP is the “FLASH” boot mode.
Reset Code Flow – Summary

Execution Entry determined by Emulation Boot Mode or Stand-Alone Boot Mode

Boot Routines
(Sci, SPI, I2C, Parallel I/O)
Interrupts

Interrupt Sources

Internal Sources
- TINT2
- TINT1
- TINT0
- ePWM, eCAP, eQEP, ADC, SCI, SPI, I2C, eCAN, LIN, CLA, WD

External Sources
- XINT1 – XINT3
- TZx
- XRS

C28x CORE
- XRS
- NMI
- INT1
- INT2
- INT3
- ...
- INT12
- INT13
- INT14

Interrupt Processing

Maskable Interrupt Processing
Conceptual Core Overview

<table>
<thead>
<tr>
<th>Core Interrupt</th>
<th>(IFR) “Latch”</th>
<th>(IER) “Switch”</th>
<th>(INTM) “Global Switch”</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT2</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>INT14</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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
Interrupts

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

```c
/* 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

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

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

**Interrupt Global Mask Bit**

<table>
<thead>
<tr>
<th>Bit 0 INTM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

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

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

Peripheral Interrupt Expansion (PIE)

**Peripheral Interrupt Expansion - PIE**

- PIE module for 96 interrupts
- Interrupt Group 1
  - INT1.1
  - INT1.2
  - ...
  - INT1.8
- 28x Core Interrupt logic
  - INT1 – INT12
  - 12 Interrupts
  - 28x Core
Interrupts

F2803x 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>WAKEINT</td>
<td>TINT0</td>
<td>ADCINT9</td>
<td>XINT2</td>
<td>XINT1</td>
<td>ADCINT2</td>
<td>ADCINT1</td>
<td></td>
</tr>
<tr>
<td>INT2</td>
<td>EPWM7_TZINT</td>
<td>EPWM6_TZINT</td>
<td>EPWM5_TZINT</td>
<td>EPWM4_TZINT</td>
<td>EPWM3_TZINT</td>
<td>EPWM2_TZINT</td>
<td>EPWM1_TZINT</td>
</tr>
<tr>
<td>INT3</td>
<td>EPWM7_INT</td>
<td>EPWM6_INT</td>
<td>EPWM5_INT</td>
<td>EPWM4_INT</td>
<td>EPWM3_INT</td>
<td>EPWM2_INT</td>
<td>EPWM1_INT</td>
</tr>
<tr>
<td>INT4</td>
<td>ECAP1_INT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT6</td>
<td>SPI TX_INTB</td>
<td>SPIRX_INTB</td>
<td>SPI TX_INTA</td>
<td>SPIRX_INTA</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>ECAN1_INTA</td>
<td>ECAN0_INTA</td>
<td>LIN1_INTA</td>
<td>LIN0_INTA</td>
<td>SCITX_INTA</td>
<td>SCIRX_INTA</td>
<td></td>
</tr>
<tr>
<td>INT10</td>
<td>ADCINT8</td>
<td>ADCINT7</td>
<td>ADCINT6</td>
<td>ADCINT5</td>
<td>ADCINT4</td>
<td>ADCINT3</td>
<td>ADCINT2</td>
</tr>
<tr>
<td>INT11</td>
<td>CLA1_INT8</td>
<td>CLA1_INT7</td>
<td>CLA1_INT6</td>
<td>CLA1_INT5</td>
<td>CLA1_INT4</td>
<td>CLA1_INT3</td>
<td>CLA1_INT2</td>
</tr>
<tr>
<td>INT12</td>
<td>LUF</td>
<td>LVF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>XINT3</td>
</tr>
</tbody>
</table>

PIE Registers

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

<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>
<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>INTx.8</td>
<td>INTx.7</td>
<td>INTx.6</td>
<td>INTx.5</td>
<td>INTx.4</td>
<td>INTx.3</td>
<td>INTx.2</td>
<td>INTx.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<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>
<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>INTx.8</td>
<td>INTx.7</td>
<td>INTx.6</td>
<td>INTx.5</td>
<td>INTx.4</td>
<td>INTx.3</td>
<td>INTx.2</td>
<td>INTx.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PIE Interrupt Acknowledge Register (PIEACK)**

<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>
<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>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>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PIECTRL register**

<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>
<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>PIEVECT</td>
<td>ENPIE</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></td>
<td></td>
</tr>
</tbody>
</table>

#include "DSP2803x_Device.h"

PieCtrlRegs.PIEIFR1.bit.INTx4 = 1; //manually set IFR for XINT1 in PIE group 1
PieCtrlRegs.PIEIER3.bit.INTx2 = 1; //enable EPWM2_INT in PIE group 3
PieCtrlRegs.PIEACK.all = 0x0004; //acknowledge the PIE group 3
PieCtrlRegs.PIECTRL.bit.ENPIE = 1; //enable the PIE
**PIE Interrupt Vector Table**

### Default Interrupt Vector Table at Reset

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

**PIE Vector Mapping (ENPIE = 1)**

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

Registers:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>ST0</td>
</tr>
<tr>
<td>AH</td>
<td>AL</td>
</tr>
<tr>
<td>PH</td>
<td>PL</td>
</tr>
<tr>
<td>AR1</td>
<td>AR0</td>
</tr>
<tr>
<td>DP</td>
<td>ST1</td>
</tr>
<tr>
<td>DBSTAT</td>
<td>IER</td>
</tr>
<tr>
<td>PC(msw)</td>
<td>PC(lsw)</td>
</tr>
</tbody>
</table>
Interrupts

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, INTM, etc.
Introduction

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

Learning Objectives

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

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

Module Topics ................................................................................ 5-2

Oscillator/PLL Clock Module .......................................................... 5-3

Watchdog Timer ........................................................................... 5-6

General-Purpose Digital I/O ............................................................ 5-10

External Interrupts ........................................................................ 5-13

Low Power Modes ....................................................................... 5-14

Register Protection ....................................................................... 5-16

Lab 5: System Initialization .......................................................... 5-18
Oscillator/PLL Clock Module

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

### F2803x PLL and LOSPCP

<table>
<thead>
<tr>
<th>DIV</th>
<th>CLKN</th>
<th>SysCtrlRegs.PLLSTS.bit.DIVSEL</th>
<th>LSPCLK</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>OSCCLK / n * (PLL bypass)</td>
<td>/4 *</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>OSCCLK x 1 / n</td>
<td>/2</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>OSCCLK x 2 / n</td>
<td>/2</td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>OSCCLK x 3 / n</td>
<td>/2</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>OSCCLK x 4 / n</td>
<td>/2</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>OSCCLK x 5 / n</td>
<td>/2</td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>OSCCLK x 6 / n</td>
<td>/2</td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>OSCCLK x 7 / n</td>
<td>/2</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>OSCCLK x 8 / n</td>
<td>/2</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>OSCCLK x 9 / n</td>
<td>/2</td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>OSCCLK x 10 / n</td>
<td>/2</td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>OSCCLK x 11 / n</td>
<td>/2</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>OSCCLK x 12 / n</td>
<td>/2</td>
<td></td>
</tr>
</tbody>
</table>

**Input Clock Fail Detect Circuitry**

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

An internal device reset will also be issued (XRSn pin not driven).
The PLL has a 4-bit ratio control to select different CPU clock rates. In addition to the on-chip oscillators, two external modes of operation are supported – crystal operation, and external clock source operation. Crystal operation allows the use of an external crystal/resonator to provide the time base to the device. External clock source operation allows the internal (crystal) oscillator to be bypassed, and the device clocks are generated from an external clock source input on the XCLKin pin. The C28x core provides a SYSCLKOUT clock signal. This signal is prescaled to provide a clock source for some of the on-chip communication peripherals through the low-speed peripheral clock prescaler. Other peripherals are clocked by SYSCLKOUT and use their own clock prescalers for operation.
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, a reset or interrupt is triggered (user selectable)
  - CPU must write correct data key sequence to reset the counter before overflow
- Watchdog must be serviced or disabled within 131,072 WDCLK cycles after reset
- This translates to 13.11 ms with a 10 MHz WDCLK

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

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

![Watchdog Timer Module Diagram]

**Watchdog Period Selection**

<table>
<thead>
<tr>
<th>WDPS Bits</th>
<th>FRC rollover</th>
<th>WD timeout period @ 10 MHz WDCLK</th>
</tr>
</thead>
<tbody>
<tr>
<td>00x</td>
<td>1</td>
<td>13.11 ms *</td>
</tr>
<tr>
<td>010</td>
<td>2</td>
<td>26.22 ms</td>
</tr>
<tr>
<td>011</td>
<td>4</td>
<td>52.44 ms</td>
</tr>
<tr>
<td>100</td>
<td>8</td>
<td>104.88 ms</td>
</tr>
<tr>
<td>101</td>
<td>16</td>
<td>209.76 ms</td>
</tr>
<tr>
<td>110</td>
<td>32</td>
<td>419.52 ms</td>
</tr>
<tr>
<td>111</td>
<td>64</td>
<td>839.04 ms</td>
</tr>
</tbody>
</table>

* reset default

- Remember: Watchdog starts counting immediately after reset is released!
- Reset default with WDCLK = 10 MHz computed as (1/10 MHz) * 512 * 256 = 13.11 ms
Watchdog Timer Control Register

SysCtrlRegs.WDCR (lab file: Watchdog.c)

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

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

Resetting the Watchdog
SysCtrlRegs.WDKEY (lab file: Watchdog.c)

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

<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>No effect; WD counter not reset on next AAh write</td>
</tr>
<tr>
<td>12</td>
<td>AAh</td>
<td>No action due to previous invalid value</td>
</tr>
<tr>
<td>13</td>
<td>55h</td>
<td>WD counter enabled for reset on next AAh write</td>
</tr>
<tr>
<td>14</td>
<td>AAh</td>
<td>WD counter is reset</td>
</tr>
</tbody>
</table>

System Control and Status Register

SysCtrlRegs.SCSR (lab file: Watchdog.c)

WD Override (protect bit)
Protects WD from being disabled
- 0 = WDDIS bit in WDCR has no effect (WD cannot be disabled)
- 1 = WDDIS bit in WDCR can disable the watchdog
  - This bit is a clear-only bit (write 1 to clear)
  - The reset default of this bit is a 1

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-3</td>
<td>reserved</td>
</tr>
<tr>
<td>2</td>
<td>WDINTS</td>
</tr>
<tr>
<td>1</td>
<td>WDENINT</td>
</tr>
<tr>
<td>0</td>
<td>WDOVERRIDE</td>
</tr>
</tbody>
</table>

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

WD Enable Interrupt
- 0 = WD generates a DSP reset
- 1 = WD generates a WDINT interrupt
General-Purpose Digital I/O

F2803x GPIO Grouping Overview
(lab file: Gpio.c)

F2803x GPIO Pin Block Diagram
(lab file: Gpio.c)

* See device datasheet for pin function selection matrices
F2803x GPIO Input Qualification

- Qualification available on ports A & B (GPIO 0 - 44) only
- Individually selectable per pin
  - no qualification (peripherals only)
  - sync to SYSCLKOUT only
  - qualify 3 samples
  - qualify 6 samples
- AIO pins are fixed as ‘sync to SYSCLKOUT’

F2803x GPIO Input Qual Registers

GpioCtrlRegs.register (lab file: Gpio.c)

GPAQSEL1 / GPAQSEL2 / GPBQSEL1

<table>
<thead>
<tr>
<th>31</th>
<th>24</th>
<th>16</th>
<th>8</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>sync to SYSCLKOUT only *</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01h</td>
<td>qual to 3 samples</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10h</td>
<td>qual to 6 samples</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11h</td>
<td>no sync or qual (for peripheral only; GPIO same as 00)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

GPACTRL / GPBCTRL

<table>
<thead>
<tr>
<th>31</th>
<th>24</th>
<th>16</th>
<th>8</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>no qualification (SYNC to SYSCLKOUT) *</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01h</td>
<td>QUALPRD = SYSCLKOUT/2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>02h</td>
<td>QUALPRD = SYSCLKOUT/4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FFh</td>
<td>QUALPRD = SYSCLKOUT/510</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* reset default
### F2803x GPIO Control Registers

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

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

### F2803x GPIO Data Registers

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPADAT</td>
<td>GPIO A Data Register [GPIO 0 – 31]</td>
</tr>
<tr>
<td>GPASET</td>
<td>GPIO A Data Set Register [GPIO 0 – 31]</td>
</tr>
<tr>
<td>G Pacifico</td>
<td>GPIO A Data Clear Register [GPIO 0 – 31]</td>
</tr>
<tr>
<td>G Pacifico</td>
<td>GPIO A Data Toggle [GPIO 0 – 31]</td>
</tr>
<tr>
<td>GPBDAT</td>
<td>GPIO B Data Register [GPIO 32 – 44]</td>
</tr>
<tr>
<td>GPBSET</td>
<td>GPIO B Data Set Register [GPIO 32 – 44]</td>
</tr>
<tr>
<td>GPBCLEAR</td>
<td>GPIO B Data Clear Register [GPIO 32 – 44]</td>
</tr>
<tr>
<td>GPBToggle</td>
<td>GPIO B Data Toggle [GPIO 32 – 44]</td>
</tr>
<tr>
<td>AIODAT</td>
<td>ANALOG I/O Data Register [AIO 0 – 15]</td>
</tr>
<tr>
<td>AIOSET</td>
<td>ANALOG I/O Data Set Register [AIO 0 – 15]</td>
</tr>
<tr>
<td>AIOCLEAR</td>
<td>ANALOG I/O Data Clear Register [AIO 0 – 15]</td>
</tr>
<tr>
<td>AIOToggle</td>
<td>ANALOG I/O Data Toggle [AIO 0 – 15]</td>
</tr>
</tbody>
</table>
External Interrupts

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

External Interrupt Registers

<table>
<thead>
<tr>
<th>Interrupt</th>
<th>Pin Selection Register (GpioIntRegs.register)</th>
<th>Configuration Register (XInterruptRegs.register)</th>
<th>Counter Register (XInterruptRegs.register)</th>
</tr>
</thead>
<tbody>
<tr>
<td>XINT1</td>
<td>GPIOXINT1SEL</td>
<td>XINT1CR</td>
<td>XINT1CTR</td>
</tr>
<tr>
<td>XINT2</td>
<td>GPIOXINT2SEL</td>
<td>XINT2CR</td>
<td>XINT2CTR</td>
</tr>
<tr>
<td>XINT3</td>
<td>GPIOXINT3SEL</td>
<td>XINT3CR</td>
<td>XINT3CTR</td>
</tr>
</tbody>
</table>

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

See device datasheet for power consumption in each mode

<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>

Watchdog Interrupt
- wake device from STANDBY
- 0 = disable (default)
- 1 = enable

Watchdog Interrupt
- Wake from STANDBY
- GPIO signal qualification
- 000000 = 2 OSCCLKs
- 000001 = 3 OSCCLKs
- 000011 = 6 OSCCLKS (default)
- 111111 = 65 OSCCLKS (default)

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

Low Power Mode Selection
- 00 = Idle (default)
- 01 = Standby
- 1x = Halt

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

<table>
<thead>
<tr>
<th>Low Power Mode</th>
<th>Exit Interrupt</th>
<th>RESET</th>
<th>GPIO Port A Signal</th>
<th>Watchdog Interrupt</th>
<th>Any Enabled Interrupt</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDLE</td>
<td>yes</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>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>HALT</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

### GPIO Low Power Wakeup Select

SysCtrlRegs.GPIOLPMSEL

- Wake device from HALT and STANDBY mode (GPIO Port A)
- 0 = disable (default)
- 1 = enable
Register Protection

Write-Read Protection

DevEmuReg.DEVICECNF.bit.ENPROT

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

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

<table>
<thead>
<tr>
<th>Peripheral Frame Registers</th>
<th>PF0</th>
<th>PF1</th>
</tr>
</thead>
<tbody>
<tr>
<td>eCAN</td>
<td>System Control</td>
<td></td>
</tr>
<tr>
<td>COMP</td>
<td>SPI</td>
<td></td>
</tr>
<tr>
<td>ePWM</td>
<td>SCI</td>
<td></td>
</tr>
<tr>
<td>eCAP</td>
<td>watchdog</td>
<td></td>
</tr>
<tr>
<td>eQEP</td>
<td>XINT</td>
<td></td>
</tr>
<tr>
<td>LIN</td>
<td>ADC</td>
<td></td>
</tr>
<tr>
<td>GPIO</td>
<td>I2C</td>
<td></td>
</tr>
</tbody>
</table>

Protected address: 0x4000 - 0x7FFF

EALLOW Protection (1 of 2)

- EALLOW stands for **Emulation Allow**
- Code access to protected registers allowed only when EALLOW = 1 in the ST1 register
- The emulator can always access protected registers
- EALLOW bit controlled by assembly level instructions
  - ‘EALLOW’ sets the bit (register access enabled)
  - ‘EDIS’ clears the bit (register access disabled)
- EALLOW bit cleared upon ISR entry, restored upon exit
EALLOW Protection (2 of 2)

The following registers are protected:

<table>
<thead>
<tr>
<th>Device Emulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flash</td>
</tr>
<tr>
<td>Code Security Module</td>
</tr>
<tr>
<td>PIE Vector Table</td>
</tr>
<tr>
<td>LIN (some registers)</td>
</tr>
<tr>
<td>eCAN A/B (control registers only; mailbox RAM not protected)</td>
</tr>
<tr>
<td>ePWM1-7 and COMP1-3 (some registers)</td>
</tr>
<tr>
<td>GPIO (control registers only)</td>
</tr>
<tr>
<td>System Control</td>
</tr>
</tbody>
</table>

See device datasheet and peripheral users guides for detailed listings

EALLOW register access C-code example:

```
asm(" EALLOW");           // enable protected register access
SysCtrlRegs.WDKEY=0x55;    // write to the register
asm(" EDIS");             // disable protected register access
```
Lab 5: System Initialization

Objective

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

- Setup the clock module – PLL, LOSPCP = /4, low-power modes to default values, enable all module clocks
- Disable the watchdog – clear WD flag, disable watchdog, WD prescale = 1
- Setup watchdog system and control register – DO NOT clear WD OVERRIDE bit, WD generate a CPU reset
- Setup shared I/O pins – set all GPIO pins to GPIO function (e.g. a "00" setting for GPIO function, and a “01”, “10”, or “11” setting for a peripheral function.)

The first part of the lab exercise will setup the system initialization and test the watchdog operation by having the watchdog cause a reset. In the second part of the lab exercise the PIE vectors will be added and tested by using the watchdog to generate an interrupt. This lab will make use of the DSP2803x C-code header files to simplify the programming of the device, as well as take care of the register definitions and addresses. Please review these files, and make use of them in the future, as needed.

Procedure

Create Project File

1. Create a new project called Lab5.pjt in C:\C28x\Labs\Lab5 and add the following files to it:
   
   CodeStartBranch.asm       Lab_5_6_7.cmd
   DelayUs.asm               Main_5.c
   DSP2803x_GlobalVariableDefs.c    SysCtrl.c
   DSP2803x_Headers_nonBIOS.cmd  Watchdog.c
   Gpio.c

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

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

   Project → Build Options...

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

   ..\DSP2803x_headers\include

This is the path for the header files.

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

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

Modify Memory Configuration

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

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

Setup System Initialization

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

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

Build and Load

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

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

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

**Run the Code – Watchdog Reset**

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

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

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

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

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

17. Like before, place the cursor in the “main loop” section (on the asm(“ NOP”);
    instruction line) and right click the mouse key and select Run To Cursor.

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

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

19. Add the following files to the project:

   DefaultIsr_5.c
   PieCtrl_5_6_7_8_9_10.c
   PieVect_5_6_7_8_9_10.c

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

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

   ```
   InitPieCtrl();
   ```

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

   PIE group #: __________  # within group: ______

22. Modify `main()` to do the following:

   - Enable global interrupts (INTM bit)

   Then modify `InitWatchdog()` to do the following:

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

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

24. Open and inspect `DefaultIsr_5.c`. This file contains interrupt service routines. The ISR for WAKEINT has been trapped by an emulation breakpoint contained in an inline assembly statement using “ESTOP0”. This gives the same results as placing a breakpoint in the ISR. We will run the lab exercise as before, except this time the watchdog will generate an interrupt. If the registers have been configured properly, the code will be trapped in the ISR.

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

**Build and Load**

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

---

C2000 Piccolo Workshop - System Initialization
Run the Code – Watchdog Interrupt

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

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

End of Exercise

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

Introduction

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

Learning Objectives

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

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

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

Analog-to-Digital Converter .............................................................................................. 6-3
ADC Block and Functional Diagrams .............................................................................. 6-3
ADC Triggering .................................................................................................................. 6-4
ADC Conversion Priority .................................................................................................. 6-5
ADC Clock and Timing ...................................................................................................... 6-7
ADC Converter Registers ................................................................................................. 6-8
ADC Calibration and Reference ......................................................................................... 6-13

Comparator ......................................................................................................................... 6-15
Comparator Block Diagram ............................................................................................... 6-15
Comparator Registers ....................................................................................................... 6-16

Lab 6: Analog-to-Digital Converter .................................................................................. 6-17
Analog-to-Digital Converter

ADC Block and Functional Diagrams

**ADC Module Block Diagram**

**ADC SOCx Functional Diagram**

This block diagram is replicated 16 times
ADC Triggering

Example – ADC Triggering (1 of 2)

Sample $A2 \rightarrow B3 \rightarrow A7$ when ePWM1 SOC is generated and then generate ADCINT1n:

- Channel $A2$: Sample 7 cycles → Result $0$ → no interrupt
- Channel $B3$: Sample 10 cycles → Result $1$ → no interrupt
- Channel $A7$: Sample 4 cycles → Result $2$ → ADCINT1n

As above, but also sample $A0 \rightarrow B0 \rightarrow A5$ continuously and generate ADCINT2n:

- Channel $A2$: Sample 7 cycles → Result $0$ → no interrupt
- Channel $B3$: Sample 10 cycles → Result $1$ → no interrupt
- Channel $A7$: Sample 4 cycles → Result $2$ → ADCINT1n

Example – ADC Triggering (2 of 2)

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

- Channel $A0: B0$: Sample 7 cycles → Result $0$ → no interrupt
- Channel $A1: B1$: Sample 7 cycles → Result $1$ → no interrupt
- Channel $A2: B2$: Sample 7 cycles → Result $2$ → no interrupt
- Channel $A3: B3$: Sample 7 cycles → Result $3$ → no interrupt
- Channel $A4: B4$: Sample 7 cycles → Result $4$ → no interrupt
- Channel $A5: B5$: Sample 7 cycles → Result $5$ → no interrupt
- Channel $A6: B6$: Sample 7 cycles → Result $6$ → ADCINT1n
- Channel $A7: B7$: Sample 7 cycles → Result $7$ → no interrupt
ADC Conversion Priority

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

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

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

Conversion Priority Functional Diagram

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

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

SOCPRORITY configured as 0; RRPONINTER configured as 15; SOC0 is highest RR priority

SOC7 trigger received

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

SOC2 & SOC12 triggers received simultaneously

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

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

---

**High Priority Example**

SOCPRORITY configured as 4; RRPONINTER configured as 15; SOC4 is highest RR priority

SOC7 trigger received

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

SOC2 & SOC12 triggers received simultaneously

SOC2 is converted; RRPONINTER stays pointing to SOC7

SOC12 is converted; RRPONINTER points to SOC12; SOC13 is now highest RR priority
ADC Clock and Timing

ADC Clocking Flow

- Internal OSC1 (10 MHz)
- PLLCR DIV bits: 1100b (x12)
- PLLSTS DIVSEL bits: 10b (/2)
- PCLKRO.ADCENCLK = 1
- ADCCLK (60 MHz)

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

ADC Timing – Sequential Sampling

- Latch: 2 Clocks
- Sample: 7 Clocks
- Convert: 6 Clocks
- Write: 2 Clocks

Max Continuous Sampling:

- 60 MHz: \( \frac{13 \text{ cycles / 1 sample}}{} = 4.62 \text{ MSPS} \)
- 40 MHz: \( \frac{13 \text{ cycles / 1 sample}}{} = 3.08 \text{ MSPS} \)
ADC Timing – Simultaneous Sampling

Max Continuous Sampling:

\[
\begin{align*}
\text{60 MHz} & \quad \text{60 MHz} = 4.62 \text{ MSPS} \\
26 \text{ cycles / 2 sample} & \quad 26 \text{ cycles / 2 sample}
\end{align*}
\]

\[
\begin{align*}
\text{40 MHz} & \quad \text{40 MHz} = 3.08 \text{ MSPS} \\
26 \text{ cycles / 2 sample} & \quad 26 \text{ cycles / 2 sample}
\end{align*}
\]

ADC Converter Registers

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

Note: ADCRESULTx is located in AdcResult.register and not in AdcRegs
ADC Control Register 1
AdcRegs.ADCCTL1

Upper Register:

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

- **ADC Busy**
  - 0 = ADC busy
  - 1 = ADC available

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

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>RESET</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>ADCENABLE</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>ADCBSY</td>
<td></td>
</tr>
<tr>
<td>12-8</td>
<td>ADCBSYCHN</td>
<td></td>
</tr>
</tbody>
</table>

- **ADC Enable**
  - 0 = ADC disable
  - 1 = ADC enable

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

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

- **ADC Reference Select**
  - 0 = internal
  - 1 = external

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

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

- **Temperature Sensor Convert**
  - currently not used
  - 0 = only valid setting

- **VREFLO Convert**
  - 0 = not connected
  - 1 = connected (B5)
ADC SOC0 – SOC15 Control Registers

AdcRegs.ADCSOCxCTL

<table>
<thead>
<tr>
<th>SOCx Trigger Source Select</th>
<th>SOCx Channel Select</th>
<th>SOCx Acquisition Prescale (S/H window)</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 - 11</td>
<td>10</td>
<td>9 - 6</td>
</tr>
<tr>
<td>5 - 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TRIGSEL reserved CHSEL ACQPS

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

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

Sequential S/M (SIMULENx=0)
Simultaneous S/M (SIMULENx=1)

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

ADC Interrupt Trigger SOC Select Registers 1 & 2

AdcRegs.ADCINTSOCSELx

ADCINTSOCSEL2

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

ADCINTSOCSEL1

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

SOCx ADC Interrupt Select
Selects which, if any, ADCINT triggers SOCx
00 = no ADCINT will trigger SOCx (TRIGSEL field determines SOCx trigger)
01 = ADCINT1 will trigger SOCx (TRIGSEL field ignored)
10 = ADCINT2 will trigger SOCx (TRIGSEL field ignored)
11 = invalid selection
**ADC Sample Mode Register**

AdcRegs.ADCSAMPLEMODE

<table>
<thead>
<tr>
<th>15 - 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>reserved</td>
</tr>
</tbody>
</table>

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

**SOC Priority Control Register**

AdcRegs.SOCPRICTL

<table>
<thead>
<tr>
<th>15 - 11</th>
<th>10 - 5</th>
<th>4 - 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>reserved</td>
<td>RRPOINTER</td>
<td>SOC_PRIORITY</td>
</tr>
</tbody>
</table>

- **Round Robin Pointer**
  - Points to the last converted round robin SOCx and determines order of conversions
- **SOC Priority**
  - Determines cutoff point for high priority and round robin mode

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

<table>
<thead>
<tr>
<th>INTy CONT</th>
<th>INTy E</th>
<th>INTy SEL</th>
<th>reserved</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Where \( xy = 1/2, 3/4, 5/6, 7/8, 9/10 \) and 10 is reserved

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

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

ADCINTx/y Interrupt Enable
- \( 0 = \text{disable} \)
- \( 1 = \text{enable} \)

**ADC Conversion Result Registers**

<table>
<thead>
<tr>
<th>Input Voltage</th>
<th>Digital Result</th>
<th>AdcResult. ADCRESULTx</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3</td>
<td>FFFh</td>
<td>0000</td>
</tr>
<tr>
<td>1.65</td>
<td>7FFh</td>
<td>0000</td>
</tr>
<tr>
<td>0.00081</td>
<td>1h</td>
<td>0000</td>
</tr>
<tr>
<td>0</td>
<td>0h</td>
<td>0000</td>
</tr>
</tbody>
</table>

- **Sequential Sampling Mode (SIMULENx = 0)**
  - After ADC completes a conversion of an SOCx, the digital result is placed in the corresponding ADCRESULTx register
- **Simultaneous Sampling Mode (SIMULENx = 1)**
  - After ADC completes a conversion of a channel pair, the digital results are found in the corresponding ADCRESULTx and ADCRESULTx+1 registers
How Can We Handle Signed Input Voltages?

Example: \(-1.65 \, V \leq V_{\text{in}} \leq +1.65 \, V\)

1) Add 1.65 volts to the analog input

\[
\begin{align*}
V_{\text{in}} & \uparrow \\
& 1.65V \\
& \downarrow \\
\end{align*}
\]

2) Subtract “1.65” from the digital result

```c
#include "DSP2803x_Device.h"
#define offset 0x07FF
void main(void) {
    int16 value; // signed
    value = AdcResult.ADCRESULT0 - offset;
}
```

ADC Calibration and Reference

**Built-In ADC Calibration**

- TI reserved OTP contains device specific calibration data for the ADC and internal oscillators
- The Boot ROM contains a Device_cal() routine that copies the calibration data to their respective registers
- Device_cal() must be run to meet the ADC and oscillator specs in the datasheet
  - The Bootloader automatically calls Device_cal() such that no action is normally required by the user
  - If the bootloader is bypassed (e.g., during development) Device_cal() should be called by the application:
    ```c
    #define Device_cal (void (*)(void))0x3D7C80
    void main(void) {
        (*Device_cal)(); // call Device_cal()
    }
    ```
- A GEL function using CCS is also available as part of the Peripheral Register Header Files to accomplish this
**Manual ADC Calibration**

- If the offset and gain errors in the datasheet* are unacceptable for your application, or you want to also compensate for board level errors (e.g., sensor or amplifier offset), you can manually calibrate.

  - **Offset error**
    - Compensated in analog with the ADCOFFTRIM register
    - No reduction in full-scale range
    - Configure input B5 to VREFL0, set ADCOFFTRIM to maximum offset error, and take a reading
    - Re-adjust ADCOFFTRIM to make result zero

  - **Gain error**
    - Compensated in software
    - Some loss in full-scale range
    - Requires use of a second ADC input pin and an upper-range reference voltage on that pin; see "TMS320280x and TMS320F2801x ADC Calibration" appnote #SPRAAD8 for more information

- **Tip:** To minimize mux-to-mux variation effects, put your most critical signals on a single mux and use that mux for calibration inputs.

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

---

**ADC Reference Selection**

\[ \text{AdcRegs.ADCREFSEL} \]

- **The internal reference has temperature stability of ~50 PPM/°C**
- **The internal reference (default) will convert an applied input voltage to a fixed scale of 0 to 3.3 V range**
- **If this is not sufficient for your application, there is the option to use an external reference**
  - External reference will scale an input voltage range from VREFL0 to VREFHI (ratiometric)
  - The reference value changes the 0 - 3.3 V full-scale range of the ADC
- **The ADCREFSEL in ADCCTL1 controls the reference choice**

<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>
<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></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></td>
<td></td>
</tr>
</tbody>
</table>

ADC Reference Selection

0 = internal (default)
1 = external VREFHI/VREFL0 pins used for reference generation

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

Comparator Block Diagram

Comparator

Comparator Block Diagram

Comparator Truth Table

<table>
<thead>
<tr>
<th>Voltages</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Voltage A &lt; Voltage B</td>
<td>0</td>
</tr>
<tr>
<td>Voltage A &gt; Voltage B</td>
<td>1</td>
</tr>
</tbody>
</table>
Comparator Registers

### Comparator Registers

#### AdcRegs.COMPCTL – Compare Control Register

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reserved</td>
<td>15-9</td>
<td>Synchronization Select</td>
</tr>
<tr>
<td>SYNCSEL</td>
<td>8</td>
<td>Output before being fed to ETPWM/GPIO blocks</td>
</tr>
<tr>
<td>QUALSEL</td>
<td>7-3</td>
<td>Qualification Period</td>
</tr>
<tr>
<td>CMPINV</td>
<td>2</td>
<td>Invert</td>
</tr>
<tr>
<td>COMPSOURCE</td>
<td>1</td>
<td>Comparator Source</td>
</tr>
<tr>
<td>COMPDACE</td>
<td>0</td>
<td>Comparator/ DAC Enable</td>
</tr>
</tbody>
</table>

- **Synchronization Select**: 0 = Asynchronous, 1 = Synchronous
- **Qualification Period**: 0h = passed, 1h = 2 clocks, 2h = 3 clocks, ... Fh = 15 clocks
- **Invert**: 0 = passed, 1 = inverted
- **Comparator Source**: 0 = DAC, 1 = pin
- **Comparator/ DAC Enable**: 0 = disable, 1 = enable

#### AdcRegs.COMPSTS – Compare Output Status Register

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reserved</td>
<td>15-9</td>
<td>Logical latched value of the comparator</td>
</tr>
<tr>
<td>COMPSTS</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

#### AdcRegs.DACVAL – DAC Value Register

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reserved</td>
<td>15-10</td>
<td>DAC Value</td>
</tr>
<tr>
<td>DACVAL</td>
<td>9-0</td>
<td>Scales output of DAC from 0 – 1023</td>
</tr>
</tbody>
</table>

- **DAC Value**: Value = 0 – 3FFh
Lab 6: Analog-to-Digital Converter

Objective

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

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

1. Using software
   a. SOCx bit (where x = 0 to 15) in the ADC SOC Force 1 Register (ADCSOCFRC1) causes a software initiated conversion

2. Automatically triggered on user selectable conditions
   a. CPU Timer 0/1/2 interrupt
   b. ePWMxSOCA / ePWMxSOCB (where x = 1 to 7)
      - ePWM underflow (CTR = 0)
      - ePWM period match (CTR = PRD)
      - ePWM underflow or period match (CTR = 0 or PRD)
      - ePWM compare match (CTRU/D = CMPA/B)
   c. ADC interrupt ADCINT1 or ADCINT2
      - triggers SOCx (where x = 0 to 15) selected by the ADC Interrupt Trigger SOC Select1/2 Register (ADCINTSOCSEL1/2)

3. Externally triggered using a pin
   a. ADCSOC pin (GPIO/XINT2_ADCSOC)

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

### Lab 6: Code Flow Diagram

- **Start**
  - General Initialization
    - PLL and clocks
    - watchdog configure
    - GPIO setup
    - PIE initialization
  - ADC Initialization
    - convert channel A0 on ePWM2 period match
    - send interrupt on every conversion
    - setup a results buffer in memory
  - ePWM2 Initialization
    - clear counter
    - set period register
    - set to trigger ADC on period match
    - set the clock prescaler
    - enable the timer
- **Main Loop**
  - while(1)
    - ADC interrupt
    - ADC ISR
      - read the ADC result
      - write to result buffer
      - adjust the buffer pointer
      - toggle the GPIO pin
      - return from interrupt
  - return

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

Project File

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

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

Setup ADC Initialization and Enable Core/PIE Interrupts

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

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

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

   PIE group #:_________    # within group:_________

   This information will be used in the next step.

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

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

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

Build and Load

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

8. Reset the CPU, select EMU_BOOT_SARAM, and then “Go Main“.
Run the Code

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

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

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

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

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

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

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

<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.

15. Recall that the code toggled the GPIO18 pin alternately high and low. (Also, the ADC ISR is toggling the LED LD3 on the ControlCARD as a visual indication that the ISR is running). If you had an oscilloscope available to display GPIO18, you would expect to see a square-wave. Why does Code Composer Studio plot resemble a triangle wave? What is the signal processing term for what is happening here?
16. Recall that the program toggled the GPIO18 pin at a 50 kHz rate. Therefore, a complete cycle (toggle high, then toggle low) occurs at half this rate, or 25 kHz. We therefore expect the period of the waveform to be 40 $\mu$s. Confirm this by measuring the period of the triangle wave using the graph (you may want to enlarge the graph window using the mouse). The measurement is best done with the mouse. The lower left-hand corner of the graph window will display the X and Y axis values. Subtract the X-axis values taken over a complete waveform period.

Using Real-time Emulation

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

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

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

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

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

Debug -> Real-time Mode

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

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

View -> Real-time Refresh Options...

and check “Global Continuous Refresh”. Use the default refresh rate of 100 ms and select OK. Alternately, we could have right clicked on each window individually and selected “Continuous Refresh”.
Note: “Global Continuous Refresh” causes all open windows to refresh at the refresh rate. This can be problematic when a large number of windows are open, as bandwidth over the emulation link is limited. Updating too many windows can cause the refresh frequency to bog down. In that case, either close some windows, or disable global refresh and selectively enable “Continuous Refresh” for individual windows of interest instead.

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

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

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

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

These GEL functions can be executed by clicking:

GEL → Realtime Emulation Control → GEL Function

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

GEL → Realtime Emulation Control → Run_Realtime_with_Reset
GEL → Realtime Emulation Control → Full_Halt

End of Exercise
Introduction

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

Learning Objectives

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

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

Control Peripherals....................................................................................................................................... 7-1

Module Topics.................................................................................................................................................. 7-2

PWM Review.................................................................................................................................................. 7-3

ePWM ........................................................................................................................................................... 7-5
  ePWM Time-Base Sub-Module ...................................................................................................................... 7-6
  ePWM Compare Sub-Module ...................................................................................................................... 7-9
  ePWM Action Qualifier Sub-Module .......................................................................................................... 7-11
  Asymmetric and Symmetric Waveform Generation using the ePWM ....................................................... 7-16
  PWM Computation Example .................................................................................................................... 7-17
  ePWM Dead-Band Sub-Module .................................................................................................................. 7-18
  ePWM PWM Chopper Sub-Module .......................................................................................................... 7-21
  ePWM Digital Compare Sub-Module ....................................................................................................... 7-24
  ePWM Trip-Zone Sub-Module .................................................................................................................... 7-27
  ePWM Event-Trigger Sub-Module ............................................................................................................. 7-30
  Hi-Resolution PWM (HRPWM) ............................................................................................................. 7-33

eCAP............................................................................................................................................................. 7-34

eQEP............................................................................................................................................................. 7-40

Lab 7: Control Peripherals............................................................................................................................... 7-42
Pulse width modulation (PWM) is a method for representing an analog signal with a digital approximation. The PWM signal consists of a sequence of variable width, constant amplitude pulses which contain the same total energy as the original analog signal. This property is valuable in digital motor control as sinusoidal current (energy) can be delivered to the motor using PWM signals applied to the power converter. Although energy is input to the motor in discrete packets, the mechanical inertia of the rotor acts as a smoothing filter. Dynamic motor motion is therefore similar to having applied the sinusoidal currents directly.
Why use PWM with Power Switching Devices?

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

![Diagram](image)
ePWM Time-Base Sub-Module

Clock Prescaler

16-Bit Time-Base Counter

Compare Register

Compare Register

Action Qualifier

Dead Band

Period Register

PWM Chopper

Trip Zone

Digital Compare

ePWM Time-Base Count Modes

Asymmetrical Waveform

Count Up Mode

Asymmetrical Waveform

Count Down Mode

Symmetrical Waveform

Count Up and Down Mode
**ePWM Phase Synchronization**

Ext. SyncIn (optional)

Phase En SyncIn CTR=zero CTR=CMPB X SyncOut

phi=0°

EPWM1A EPWM1B

To eCAP1 SyncIn

phi=120°

EPWM2A EPWM2B

phi=240°

EPWM3A EPWM3B

**ePWM Time-Base Sub-Module Registers**

(lab file: EPwm.c)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBCTL</td>
<td>Time-Base Control</td>
<td>EPwmXRegs.TBCTL.all =</td>
</tr>
<tr>
<td>TBSTS</td>
<td>Time-Base Status</td>
<td>EPwmXRegs.TBSTS.all =</td>
</tr>
<tr>
<td>TBPHS</td>
<td>Time-Base Phase</td>
<td>EPwmXRegs.TBPHS =</td>
</tr>
<tr>
<td>TBCTR</td>
<td>Time-Base Counter</td>
<td>EPwmXRegs.TBCTR =</td>
</tr>
<tr>
<td>TBPRD</td>
<td>Time-Base Period</td>
<td>EPwmXRegs.TBPRD =</td>
</tr>
</tbody>
</table>
The ePWM Time-Base Control Register (EPwmxRegs.TBCTL) is used to control the time base of the ePWM module. It consists of an upper register and a lower register.

### Upper Register:
- **Phase Direction**
  - 0 = count down after sync
  - 1 = count up after sync

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

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

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

- **TBCLK = SYSCLKOUT / (HSPCLKDIV * CLKDIV)**

### Lower Register:
- **Software Force Sync Pulse**
  - 0 = no action
  - 1 = force one-time sync

- **Sync Output Select**
  - 00 = EPWMxSYNCI
  - 01 = CTR = 0
  - 10 = CTR = CMPB
  - 11 = disable SyncOut

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

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

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

(HSPCLKDIV is for legacy compatibility)
ePWM Compare Sub-Module

ePWM Compare Event Waveforms

- = compare events are fed to the Action Qualifier Sub-Module

Count Up Mode

Asymmetrical Waveform

Count Down Mode

Asymmetrical Waveform

Count Up and Down Mode

Symmetrical Waveform
### ePWM Compare Sub-Module Registers

**lab file: EPwm.c**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPCTL</td>
<td>Compare Control</td>
<td>EPwmRegs.CMPCTL.all =</td>
</tr>
<tr>
<td>CMPA</td>
<td>Compare A</td>
<td>EPwmRegs.CMPA =</td>
</tr>
<tr>
<td>CMPB</td>
<td>Compare B</td>
<td>EPwmRegs.CMPB =</td>
</tr>
</tbody>
</table>

### ePWM Compare Control Register

**EPwmRegs.CMPCTL**

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

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

- **CMPA and CMPB Shadow Load Mode**
  - 00 = load on CTR = 0
  - 01 = load on CTR = PRD
  - 10 = load on CTR = 0 or PRD
  - 11 = freeze (no load possible)
ePWM Action Qualifier Sub-Module

![Diagram of ePWM Action Qualifier Sub-Module]

ePWM Action Qualifier Actions for EPWMA and EPWMB

<table>
<thead>
<tr>
<th>S/W Force</th>
<th>Time-Base Counter equals:</th>
<th>EPWM Output Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Zero</td>
<td>CMPA</td>
</tr>
<tr>
<td>SW X</td>
<td>Z</td>
<td>CA</td>
</tr>
<tr>
<td>SW ↓</td>
<td>Z</td>
<td>CA</td>
</tr>
<tr>
<td>SW ↑</td>
<td>Z</td>
<td>CA</td>
</tr>
<tr>
<td>SW T</td>
<td>Z</td>
<td>CA</td>
</tr>
</tbody>
</table>

- Do Nothing
- Clear Low
- Set High
- Toggle
**ePWM Count Up Asymmetric Waveform**

with Independent Modulation on EPWMA / B

```
+-----------------+    +-----------------+
| TBCTR           |    | TBPRD           |
|                 |----|                 |
| EPWMA           |    | EPWMB           |
|                  |    |                  |
|                  |    |                  |
| Z  P  CB  CA    |    | Z  P  CB  CA    |
| ↑    ↓          |    | ↑    ↓          |
| X               |    | X               |
```

**ePWM Count Up Asymmetric Waveform**

with Independent Modulation on EPWMA

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

---

ePWM Count Up-Down Symmetric Waveform
with Independent Modulation on EPWMA
ePWM Action Qualifier Sub-Module Registers

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>AQCTLA</td>
<td>AQ Control Output A</td>
<td>EPwmRegs.AQCTLA.all =</td>
</tr>
<tr>
<td>AQCTLB</td>
<td>AQ Control Output B</td>
<td>EPwmRegs.AQCTLB.all =</td>
</tr>
<tr>
<td>AQSFRG</td>
<td>AQ S/W Force</td>
<td>EPwmRegs.AQSFRG.all =</td>
</tr>
<tr>
<td>AQCSFRG</td>
<td>AQ Cont. S/W Force</td>
<td>EPwmRegs.AQCSFRG.all =</td>
</tr>
</tbody>
</table>

Action Qualifier Control Register

EPwmRegs.AQCTLy \( (y = A \text{ or } B) \)

- **Action when CTR = CMPB on UP Count**
- **Action when CTR = CMPA on UP Count**
- **Action when CTR = 0**

| Bit No. | 
|---------|-----------------------------------|-----------------------------------|-----------------------------------|-----------------------------------|
| 15 - 12 | reserved                          | CBD                              | CBU                              | CAD                              | CAU                              | PRD                              | ZRO                              |

- **Action when CTR = CMPB on DOWN Count**
- **Action when CTR = CMPA on DOWN Count**
- **Action when CTR = PRD**

- **00 = do nothing (action disabled)**
- **01 = clear (low)**
- **10 = set (high)**
- **11 = toggle (low → high; high → low)**
### ePWM Action Qualifier S/W Force Register

**EPwm\[x\]Regs.AQSFRC**

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 - 8</td>
<td>One-Time S/W Force on Output B / A</td>
</tr>
<tr>
<td>7 - 6</td>
<td>0 = no action</td>
</tr>
<tr>
<td>5</td>
<td>1 = single s/w force event</td>
</tr>
</tbody>
</table>

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

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

### ePWM Action Qualifier Continuous S/W Force Register

**EPwm\[x\]Regs.AQCSFRC**

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 - 4</td>
<td>Continuous S/W Force on Output B / A</td>
</tr>
<tr>
<td>3 - 2</td>
<td>00 = forcing disabled</td>
</tr>
<tr>
<td>1 - 0</td>
<td>01 = force continuous low on output</td>
</tr>
<tr>
<td></td>
<td>10 = force continuous high on output</td>
</tr>
<tr>
<td></td>
<td>11 = forcing disabled</td>
</tr>
</tbody>
</table>

---

**C2000 Piccolo Workshop - Control Peripherals** 7 - 15
Asymmetric and Symmetric Waveform Generation using the ePWM

PWM switching frequency:

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

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

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

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: \(\text{TxCMPR} = (100\% - \text{duty cycle}) \times \text{TxPR}\)

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

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

**Symmetric PWM Computation Example**

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

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

\[
\text{CMPA} = (100\% - \text{duty cycle}) \times \text{TBPRD} = 0.75 \times 500 = 375
\]

**Asymmetric PWM Computation Example**

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

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

\[
\text{CMPA} = (100\% - \text{duty cycle}) \times (\text{TBPRD} + 1) - 1 = 0.75 \times (999+1) - 1 = 749
\]
Motivation for Dead-Band

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

### ePWM Dead-Band Sub-Module Registers

*lab file: EPwm.c*

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBCTL</td>
<td>Dead-Band Control</td>
<td>EPwmRegs.DBCTL.all =</td>
</tr>
<tr>
<td>DBRED</td>
<td>10-bit Rising Edge Delay</td>
<td>EPwmRegs.DBRED =</td>
</tr>
<tr>
<td>DBFED</td>
<td>10-bit Falling Edge Delay</td>
<td>EPwmRegs.DBFED =</td>
</tr>
</tbody>
</table>

Rising Edge Delay = $T_{TBCLK} \times DBRED$

Falling Edge Delay = $T_{TBCLK} \times DBFED$
ePWM Dead Band Control Register
EPwmxRegs.DBCTL

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

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

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

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

---

ePWM PWM Chopper Sub-Module

- **Clock Prescaler**
  - TBCLK
  - EPWMxSYNCI
  - EPWMxSYNCO

- **16-Bit Time-Base Counter**

- **Compare Logic**

- **Action Qualifier**

- **Dead Band**

- **PWM Chopper**

- **Trip Zone**
  - EPWMxA
  - EPWMxB

- **Digital Compare**
  - T21-TZ3
  - COMPxOUT
Purpose of the PWM Chopper

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

ePWM Chopper Waveform

- Programmable Pulse Width (OSHTWTH)
- Sustaining Pulses
- With One-Shot Pulse on EPWMxA and/or EPWMxB
**ePWM Chopper Sub-Module Registers**

(lab file: EPwm.c)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCCTL</td>
<td>PWM-Chopper Control</td>
<td>EPwmxRegs.PCCTL.all =</td>
</tr>
</tbody>
</table>

### ePWM Chopper Control Control Register

**EPwmxRegs.PCCTL**

<table>
<thead>
<tr>
<th>Chopper Clk Duty Cycle</th>
<th>Chopper Clk Freq.</th>
<th>Chopper Enable</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 = 1/8 (12.5%)</td>
<td>000 = SYSCLKOUT/8 ÷ 1</td>
<td>0 = disable (bypass)</td>
</tr>
<tr>
<td>001 = 2/8 (25.0%)</td>
<td>001 = SYSCLKOUT/8 ÷ 2</td>
<td>1 = enable</td>
</tr>
<tr>
<td>010 = 3/8 (37.5%)</td>
<td>010 = SYSCLKOUT/8 ÷ 3</td>
<td></td>
</tr>
<tr>
<td>011 = 4/8 (50.0%)</td>
<td>011 = SYSCLKOUT/8 ÷ 4</td>
<td></td>
</tr>
<tr>
<td>100 = 5/8 (62.5%)</td>
<td>100 = SYSCLKOUT/8 ÷ 5</td>
<td></td>
</tr>
<tr>
<td>101 = 6/8 (75.0%)</td>
<td>101 = SYSCLKOUT/8 ÷ 6</td>
<td></td>
</tr>
<tr>
<td>110 = 7/8 (87.5%)</td>
<td>110 = SYSCLKOUT/8 ÷ 7</td>
<td></td>
</tr>
<tr>
<td>111 = reserved</td>
<td>111 = SYSCLKOUT/8 ÷ 8</td>
<td></td>
</tr>
</tbody>
</table>

- **CHPDUTY**: 15 - 11
- **CHPfreq**: 10 - 8
- **OSHTWTH**: 7 - 5
- **CHPEN**: 4 - 1

### One-Shot Pulse Width

<table>
<thead>
<tr>
<th>One-Shot Pulse Width</th>
<th>SYSCLKOUT/8</th>
<th>SYSCLKOUT/8</th>
<th>SYSCLKOUT/8</th>
<th>SYSCLKOUT/8</th>
<th>SYSCLKOUT/8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>1 x</td>
<td>1 x</td>
<td>1 x</td>
<td>1 x</td>
<td>1 x</td>
</tr>
<tr>
<td>0001</td>
<td>2 x</td>
<td>2 x</td>
<td>2 x</td>
<td>2 x</td>
<td>2 x</td>
</tr>
<tr>
<td>0010</td>
<td>3 x</td>
<td>3 x</td>
<td>3 x</td>
<td>3 x</td>
<td>3 x</td>
</tr>
<tr>
<td>0011</td>
<td>4 x</td>
<td>4 x</td>
<td>4 x</td>
<td>4 x</td>
<td>4 x</td>
</tr>
<tr>
<td>0100</td>
<td>5 x</td>
<td>5 x</td>
<td>5 x</td>
<td>5 x</td>
<td>5 x</td>
</tr>
<tr>
<td>0101</td>
<td>6 x</td>
<td>6 x</td>
<td>6 x</td>
<td>6 x</td>
<td>6 x</td>
</tr>
<tr>
<td>0110</td>
<td>7 x</td>
<td>7 x</td>
<td>7 x</td>
<td>7 x</td>
<td>7 x</td>
</tr>
<tr>
<td>0111</td>
<td>8 x</td>
<td>8 x</td>
<td>8 x</td>
<td>8 x</td>
<td>8 x</td>
</tr>
<tr>
<td>1000</td>
<td>9 x</td>
<td>9 x</td>
<td>9 x</td>
<td>9 x</td>
<td>9 x</td>
</tr>
<tr>
<td>1001</td>
<td>10 x</td>
<td>10 x</td>
<td>10 x</td>
<td>10 x</td>
<td>10 x</td>
</tr>
<tr>
<td>1010</td>
<td>11 x</td>
<td>11 x</td>
<td>11 x</td>
<td>11 x</td>
<td>11 x</td>
</tr>
<tr>
<td>1011</td>
<td>12 x</td>
<td>12 x</td>
<td>12 x</td>
<td>12 x</td>
<td>12 x</td>
</tr>
<tr>
<td>1100</td>
<td>13 x</td>
<td>13 x</td>
<td>13 x</td>
<td>13 x</td>
<td>13 x</td>
</tr>
<tr>
<td>1101</td>
<td>14 x</td>
<td>14 x</td>
<td>14 x</td>
<td>14 x</td>
<td>14 x</td>
</tr>
<tr>
<td>1110</td>
<td>15 x</td>
<td>15 x</td>
<td>15 x</td>
<td>15 x</td>
<td>15 x</td>
</tr>
<tr>
<td>1111</td>
<td>16 x</td>
<td>16 x</td>
<td>16 x</td>
<td>16 x</td>
<td>16 x</td>
</tr>
</tbody>
</table>
Purpose of the Digital Compare Sub-Module

- **Comparator module outputs** (COMP1, COMP2, and COMP3) and **Trip-Zone inputs** (TZ1, TZ2, and TZ3) generate Digital Compare A and B High/Low Signals (DCAH, DCAL, DCBH, and DCBL)

- DCAH/L and DCBH/L signals trigger events which can be filtered or fed directly to the trip-zone, event-trigger, and time-base sub-modules to:
  - Generate a trip-zone interrupt
  - Generate an ADC start of conversion
  - Force an event
  - Generate a synchronization event for synchronizing the ePWM module TBCNT

- Event filtering can optionally blank the input signal to remove noise
Digital Compare Sub-Module Signals

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

ePWM Digital Compare Sub-Module Registers

(lab file: EPwm.c)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCACTL</td>
<td>DC A Control</td>
<td>EPwmRegs.DCACTL.all = EPwmRegs.DCACTL.a =</td>
</tr>
<tr>
<td>DCBCTL</td>
<td>DC B Control</td>
<td>EPwmRegs.DCBCTL.all =</td>
</tr>
<tr>
<td>DCTRIPSEL</td>
<td>DC Trip Select</td>
<td>EPwmRegs.DCTRIPSEL.all =</td>
</tr>
<tr>
<td>DCCAPCTL</td>
<td>Capture Control</td>
<td>EPWMRegs.DCCAPCTL.all =</td>
</tr>
<tr>
<td>DCCAP</td>
<td>Counter Capture</td>
<td>EPwmRegs.DCCAP =</td>
</tr>
<tr>
<td>DCFCTL</td>
<td>DC Filter Control</td>
<td>EPwmRegs.DCFCTL.all =</td>
</tr>
<tr>
<td>DCOFFSETCNT</td>
<td>Filter Offset Ctr</td>
<td>EPwmRegs.DCOFFSETCNT =</td>
</tr>
<tr>
<td>DCFWINDOW</td>
<td>Filter Window</td>
<td>EPwmRegs.DCFWINDOW =</td>
</tr>
<tr>
<td>DCFWINDOWCNT</td>
<td>Filter Window Ctr</td>
<td>EPwmRegs.DCFWINDOWCNT =</td>
</tr>
</tbody>
</table>
ePWM Digital Compare Control Register

EPwmxRegs.DCyCTL ($y = A$ or $B$)

- **DC$_y$EVT2 Source Force**
  - Sync Signal Select
  - $0 = $ synchronous
  - $1 = $ asynchronous

- **DC$_y$EVT1 SOC**
  - Generation
  - $0 = $ disable
  - $1 = $ enable

- **DC$_y$EVT1 Source Force**
  - Sync Signal Select
  - $0 = $ synchronous
  - $1 = $ asynchronous

- **DC$_y$EVT2 Source Signal Select**
  - $0 = $ DC$_y$EVT2 signal
  - $1 = $ DCEVTFILT signal

- **DC$_y$EVT1 Signal Select**
  - $0 = $ disable
  - $1 = $ enable

- **DC$_y$EVT1 SOC**
  - Sync Signal Select
  - $0 = $ synchronous
  - $1 = $ asynchronous

- **DC$_y$EVT2 Source Force**
  - Sync Signal Select
  - $0 = $ synchronous
  - $1 = $ asynchronous

- **DC$_y$EVT1 Source Force**
  - Sync Signal Select
  - $0 = $ synchronous
  - $1 = $ asynchronous

- **DC$_y$EVT2 Source Signal Select**
  - $0 = $ DC$_y$EVT2 signal
  - $1 = $ DCEVTFILT signal

- **DC$_y$EVT1 Source Signal Select**
  - $0 = $ disable
  - $1 = $ enable

ePWM Digital Compare Trip Select Register

EPwmxRegs.DCTRIPSEL

- **Digital Compare B**
  - Low Input Source Select
  - High Input Source Select

- **Digital Compare A**
  - Low Input Source Select
  - High Input Source Select

- **DCBLCOMPSEL**
- **DCBHCOMPSEL**

- **DCALCOMPSEL**
- **DCAHCOMPSEL**

- **Digital Compare A**
  - Low Input Source Select
  - High Input Source Select

- **Digital Compare B**
  - Low Input Source Select
  - High Input Source Select

0000 = TZ1 input
0001 = TZ2 input
0010 = TZ3 input
1000 = COMP1OUT input
1001 = COMP2OUT input
1010 = COMP3OUT input
other values reserved
ePWM Trip-Zone Sub-Module

Trip-Zone Features

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

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

### ePWM Trip-Zone Sub-Module Registers

(lab file: EPwm.c)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>TZCTL</td>
<td>Trip-Zone Control</td>
<td>EPwmRegs.TZCTL.all =</td>
</tr>
<tr>
<td>TZSEL</td>
<td>Trip-Zone Select</td>
<td>EPwmRegs.TZSEL.all =</td>
</tr>
<tr>
<td>TZEINT</td>
<td>Enable Interrupt</td>
<td>EPwmRegs.TZEINT.all =</td>
</tr>
<tr>
<td>TZDCSEL</td>
<td>Digital Compare</td>
<td>EPWMxRegs.TZDCSEL.all =</td>
</tr>
<tr>
<td>TZFLG</td>
<td>Trip-Zone Flag</td>
<td>EPwmRegs.TZFLG.all =</td>
</tr>
<tr>
<td>TZCLR</td>
<td>Trip-Zone Clear</td>
<td>EPwmRegs.TZCLR.all =</td>
</tr>
<tr>
<td>TZFRC</td>
<td>Trip-Zone Force</td>
<td>EPwmRegs.TZFRC.all =</td>
</tr>
</tbody>
</table>

### ePWM Trip-Zone Control Register

EPwmxRegs.TZCTL

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-12</td>
<td>reserved</td>
</tr>
<tr>
<td>11-10</td>
<td>DCBEVT2</td>
</tr>
<tr>
<td>9-8</td>
<td>DCBEVT1</td>
</tr>
<tr>
<td>7-6</td>
<td>DCAEV2</td>
</tr>
<tr>
<td>5-4</td>
<td>DCAEV1</td>
</tr>
<tr>
<td>3-2</td>
<td>TZA</td>
</tr>
<tr>
<td>1-0</td>
<td>TZB</td>
</tr>
</tbody>
</table>

Digital Compare Output Event 2 / 1 Action on EPWMxB
Digital Compare Output Event 2 / 1 Action on EPWMxA
TZ1 to TZ6 Action on EPWMxB / EPWMxA

00 = high impedance
01 = force high
10 = force low
11 = do nothing (disable)
**ePWM Trip-Zone Select Register**

EPwmxRegs.TZSEL

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

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

**ePWM Trip-Zone Enable Interrupt Register**

EPwmxRegs.TZEINT

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

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

- **One-Shot Interrupt Enable**
  - 0 = disable
  - 1 = enable

- **Cycle-by-Cycle Interrupt Enable**
  - 0 = disable
  - 1 = enable
**ePWM Trip-Zone Digital Compare Event Select Register**

EPwmxRegs.TZDCSEL

<table>
<thead>
<tr>
<th>15 - 12</th>
<th>11 - 9</th>
<th>8 - 6</th>
<th>5 - 3</th>
<th>2 - 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>reserved</td>
<td>DCBEVT2</td>
<td>DCBEVT1</td>
<td>DCAEVT2</td>
<td>DCAEVT1</td>
</tr>
</tbody>
</table>

Digital Compare Output B Event 2 / 1 Select  
Digital Compare Output A Event 2 / 1 Select

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

---

**ePWM Event-Trigger Sub-Module**
ePWM Event-Trigger Interrupts and SOC

**ePWM Event-Trigger Sub-Module Registers**
(lab file: EPwm.c)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETSEL</td>
<td>Event-Trigger Selection</td>
<td>EPwmxRegs.ETSEL.all =</td>
</tr>
<tr>
<td>ETPS</td>
<td>Event-Trigger Pre-Scale</td>
<td>EPwmxRegs.ETPS.all =</td>
</tr>
<tr>
<td>ETFLG</td>
<td>Event-Trigger Flag</td>
<td>EPwmxRegs.ETFLG.all =</td>
</tr>
<tr>
<td>ETCLR</td>
<td>Event-Trigger Clear</td>
<td>EPwmxRegs.ETCLR.all =</td>
</tr>
<tr>
<td>ETFRC</td>
<td>Event-Trigger Force</td>
<td>EPwmxRegs.ETFRC.all =</td>
</tr>
</tbody>
</table>
**ePWM Event-Trigger Selection Register**

**EPwmxRegs.ETSEL**

<table>
<thead>
<tr>
<th>Bit 15</th>
<th>Bit 14-12</th>
<th>Bit 11</th>
<th>Bit 10-8</th>
<th>Bit 7-4</th>
<th>Bit 3</th>
<th>Bit 2-0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SOCBEN</strong></td>
<td><strong>SOCBSEL</strong></td>
<td><strong>SOCAEN</strong></td>
<td><strong>SOCASEL</strong></td>
<td><strong>reserved</strong></td>
<td><strong>INTEN</strong></td>
<td><strong>INTSEL</strong></td>
</tr>
<tr>
<td>Enable SOCB / A</td>
<td>Enable EPWMxINT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 = disable</td>
<td>0 = disable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 = enable</td>
<td>1 = enable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**EPWMxSOCB / A Select**

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

**EPWMxINT Select**

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

---

**ePWM Event-Trigger Prescale Register**

**EPwmxRegs.ETPS**

<table>
<thead>
<tr>
<th>Bit 15</th>
<th>Bit 14-12</th>
<th>Bit 11</th>
<th>Bit 10-8</th>
<th>Bit 7-4</th>
<th>Bit 2-3</th>
<th>Bit 1-0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SOCBCNT</strong></td>
<td><strong>SOCBPRD</strong></td>
<td><strong>SOCACNT</strong></td>
<td><strong>SOCAPRD</strong></td>
<td><strong>reserved</strong></td>
<td><strong>INTCNT</strong></td>
<td><strong>INTPRD</strong></td>
</tr>
<tr>
<td>EPWMxSOCB / A Counter</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(number of events have occurred)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00 = no events</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 = 1 event</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 = 2 events</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 = 3 events</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

**EPWMxSOCB / A Period**

- 00 = disabled
- 01 = SOC on first event
- 10 = SOC on second event
- 11 = SOC on third event

**EPWMxINT Period**

- 00 = disabled
- 01 = INT on first event
- 10 = INT on second event
- 11 = INT on third event

---

7 - 32  
C2000 Piccolo Workshop - Control Peripherals
Hi-Resolution PWM (HRPWM)

- Significantly increases the resolution of conventionally derived digital PWM
- Uses 8-bit extensions to Compare registers (CMPxHR), Period register (TBPRDHR) and Phase register (TBPHSR) for edge positioning control
- Typically used when PWM resolution falls below ~9-10 bits which occurs at frequencies greater than ~120 kHz (with system clock of 60 MHz)
- Not all ePWM outputs support HRPWM feature (see device datasheet)
The capture units allow time-based logging of external TTL signal transitions on the capture input pins. The C28x has up to six capture units.

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

- **Measure the time width of a pulse**
- **Low speed velocity estimation from incr. encoder:**
  
  Problem: At low speeds, calculation of speed based on a measured position change at fixed time intervals produces large estimate errors

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

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

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

- **Auxiliary PWM generation**

---

**eCAP Module Block Diagram – Capture Mode**

- **Capture 1 Register**
- **Capture 2 Register**
- **Capture 3 Register**
- **Capture 4 Register**
- **Cap1Pol**
- **Cap2Pol**
- **Cap3Pol**
- **Cap4Pol**
- **Event Logic**
- **Event Prescale**
- **PRESCALE**
- **CAP1POL**
- **CAP2POL**
- **CAP3POL**
- **CAP4POL**
- **Polarity Select 1**
- **Polarity Select 2**
- **Polarity Select 3**
- **Polarity Select 4**
- **32-Bit Time-Stamp Counter**
- **SYSLCLKOUT**
- **ECAPx pin**
eCAP Module Block Diagram – APWM Mode

32-Bit Time-Stamp Counter

Period Register (CAP1)

32-Bit

Immediate mode

Shadowed

Period Register (CAP3)

Immediate mode

Shadowed

Compare Logic

Immediate mode

Shadowed

Compare Register (CAP2)

Compare Register (CAP4)

SYSCLKOUT

ECAP pin

eCAP Module Registers
(lab file: ECap.c)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECCTL1</td>
<td>Capture Control 1</td>
<td>ECapxRegs.ECCTL1.all =</td>
</tr>
<tr>
<td>ECCTL2</td>
<td>Capture Control 2</td>
<td>ECapxRegs.ECCTL2.all =</td>
</tr>
<tr>
<td>TSCTR</td>
<td>Time-Stamp Counter</td>
<td>ECapxRegs.TSCTR =</td>
</tr>
<tr>
<td>CTRPHS</td>
<td>Counter Phase Offset</td>
<td>ECapxRegs.CTRPHS =</td>
</tr>
<tr>
<td>CAP1</td>
<td>Capture 1</td>
<td>ECapxRegs.CAP1 =</td>
</tr>
<tr>
<td>CAP2</td>
<td>Capture 2</td>
<td>ECapxRegs.CAP2 =</td>
</tr>
<tr>
<td>CAP3</td>
<td>Capture 3</td>
<td>ECapxRegs.CAP3 =</td>
</tr>
<tr>
<td>CAP4</td>
<td>Capture 4</td>
<td>ECapxRegs.CAP4 =</td>
</tr>
<tr>
<td>ECEINT</td>
<td>Enable Interrupt</td>
<td>ECapxRegs.ECEINT.all =</td>
</tr>
<tr>
<td>ECFLG</td>
<td>Interrupt Flag</td>
<td>ECapxRegs.ECFLG.all =</td>
</tr>
<tr>
<td>ECCLR</td>
<td>Interrupt Clear</td>
<td>ECapxRegs.ECCLR.all =</td>
</tr>
<tr>
<td>ECFRC</td>
<td>Interrupt Force</td>
<td>ECapxRegs.ECFRC.all =</td>
</tr>
</tbody>
</table>
eCAP Control Register 1
ECapRegs.ECCTL1

Upper Register:

- **FREE_SOFT**: 15 - 14
- **PRESCALE**: 13 - 9
- **CAPLDEN**: 8

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

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

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

Lower Register:


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

**Capture Event Polarity**
- 0 = trigger on rising edge
- 1 = trigger on falling edge
### eCAP Control Register 2
**ECap×Regs.ECCTL2**

#### Upper Register:
- **Capture / APWM mode**
  - 0 = capture mode
  - 1 = APWM mode
- **APWM Output Polarity**
  - (valid only in APWM mode)
  - 0 = active high output
  - 1 = active low output
- **Software Force Counter Synchronization**
  - 0 = no effect
  - 1 = TSCTR load of current module and other modules if SYNCO_SEL bits = 00

#### Lower Register:
- **Counter Sync-In**
  - 0 = disable
  - 1 = enable
- **Re-arm**
  - (capture mode only)
  - 0 = no effect
  - 1 = arm sequence
- **Continuous/One-Shot**
  - (capture mode only)
  - 0 = continuous mode
  - 1 = one-shot mode
- **Sync-Out Select**
  - 00 = sync-in to sync-out
  - 01 = CTR = PRD event generates sync-out
  - 1X = disable
- **Time Stamp Counter Stop**
  - 0 = stop
  - 1 = run
- **Stop Value for One-Shot Mode/ Wrap Value for Continuous Mode**
  - (capture mode only)
  - 00 = stop/wrap after capture event 1
  - 01 = stop/wrap after capture event 2
  - 10 = stop/wrap after capture event 3
  - 11 = stop/wrap after capture event 4
The capture unit interrupts offer immediate CPU notification of externally captured events. In situations where this is not required, the interrupts can be masked and flag testing/polling can be used instead. This offers increased flexibility for resource management. For example, consider a servo application where a capture unit is being used for low-speed velocity estimation via a pulsing sensor. The velocity estimate is not used until the next control law calculation is made, which is driven in real-time using a timer interrupt. Upon entering the timer interrupt service routine, software can test the capture interrupt flag bit. If sufficient servo motion has occurred since the last control law calculation, the capture interrupt flag will be set and software can proceed to compute a new velocity estimate. If the flag is not set, then sufficient motion has not occurred and some alternate action would be taken for updating the velocity estimate. As a second example, consider the case where two successive captures are needed before a computation proceeds (e.g. measuring the width of a pulse). If the width of the pulse is needed as soon as the pulse ends, then the capture interrupt is the best option. However, the capture interrupt will occur after each of the two captures, the first of which will waste a small number of cycles while the CPU is interrupted and then determines that it is indeed only the first capture. If the width of the pulse is not needed as soon as the pulse ends, the CPU can check, as needed, the capture registers to see if two captures have occurred, and proceed from there.
What is an Incremental Quadrature Encoder?

A digital (angular) position sensor

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

Incremental Optical Encoder
Quadrature Output from Photo Sensors

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

How is Position Determined from Quadrature Signals?

Position resolution is θ/4 degrees

(A,B) = (00) (11) (10) (01)
Ch. A
Ch. B

Quadrature Decoder State Machine

increment counter
decrement counter

Illegal Transitions; generate phase error interrupt
**eQEP Module Block Diagram**

- **Quadrature Capture**: Monitors the quadrature clock to indicate proper operation of the motion control system.
- **Position/Counter Compare**: Generate the direction and clock for the position counter in quadrature count mode.
- **32-Bit Unit Time-Base**: Generate periodic interrupts for velocity calculations.
- **QEP Watchdog**: Generate a sync output and/or interrupt on a position compare match.
- **SYSCLKOUT**: Measure the elapsed time between the unit position events; used for low speed measurement.

**eQEP Module Connections**

- **Ch. A**: EQEPxA/XCLK, EQEPxB/XDIR, EQEPxI, EQEPxS
- **Ch. B**: EQEPxA/XCLK, EQEPxB/XDIR, EQEPxI, EQEPxS
- **Index**: from homing sensor

**Connections**:
- 32-Bit Unit Time-Base
- SYSCLKOUT
- Strobe
- **Quadrature Capture**
- **QEP Watchdog**
- **Position/Counter Compare**

---

*C2000 Piccolo Workshop - Control Peripherals*
Lab 7: Control Peripherals

Objective

The objective of this lab is to become familiar with the programming and operation of the control peripherals and their interrupts. ePWM1A will be setup to generate a 2 kHz, 25% duty cycle symmetric PWM waveform. The waveform will then be sampled with the on-chip analog-to-digital converter and displayed using the graphing feature of Code Composer Studio. Next, eCAP1 will be setup to detect the rising and falling edges of the waveform. This information will be used to determine the width of the pulse and duty cycle of the waveform. The results of this step will be viewed numerically in a memory window.

Procedure

Project File

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

   - Adc.c
   - CodeStartBranch.asm
   - DefaultIsr_7.c
   - DelayUs.asm
   - DSP2833x_GlobalVariableDefs.c
   - DSP2833x_Headers_nonBIOS.cmd
   - ECap_7_8_9_10_12.c
   - EPwm_7_8_9_10_12.c
   - Gpio.c
   - Lab_5_6_7.cmd
   - Main_7.c
   - PieCtrl_5_6_7_8_9_10.c
   - PieVect_5_6_7_8_9_10.c
   - SysCtrl.c
   - Watchdog.c
Setup Shared I/O and ePWM1

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

3. In EPwm 7_8_9_10_12.c, setup ePWM1 to implement the PWM waveform as described in the objective for this lab. The following registers need to be modified: TBCTL (set clock prescales to divide-by-1, no software force, sync and phase disabled), TBPRD, CMPA, CMPCTL (load on 0 or PRD), and AQCTLA (set on up count and clear on down count for output A). Software force, deadband, PWM chopper and trip action has been disabled. (Hint – notice the last steps enable the timer count mode and enable the clock to the ePWM module). Either calculate the values for TBPRD and CMPA (as a challenge) or make use of the global variable names and values that have been set using #define in the beginning of Lab.h file. Notice that ePWM2 has been initialized earlier in the code for the ADC lab. Save your work.

Build and Load

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

Run the Code – PWM Waveform

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

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

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

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

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

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

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

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

<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.

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

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

**Setup eCAP1 to Measure Width of Pulse**

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

13. Add the following file to the project:

   ECap_7_8_9_10_12.c
Check your files list to make sure the file is there.

14. In \texttt{Main\_7.c}, add code to call the \texttt{InitECap()} function. There are no passed parameters or return values, so the call code is simply:

\begin{verbatim}
InitECap();
\end{verbatim}

15. Edit \texttt{Gpio.c} and adjust the shared I/O pin in GPIO5 for the ECAP1 function.

16. Open and inspect the ECAP1 interrupt service routine (ECAP1\_INT\_ISR) in the file \texttt{DefaultIsr\_7.c}. Notice that \texttt{PwmDuty} is calculated by \texttt{CAP2 – CAP1} (rising to falling edge) and that \texttt{PwmPeriod} is calculated by \texttt{CAP3 – CAP1} (rising to rising edge).

17. In \texttt{ECap\_7\_8\_9\_10\_12.c}, setup ECAP1 to calculate PWM\_duty and PWM\_period. The following registers need to be modified: ECCTL2 (continuous mode, re-arm disable, and sync disable), ECCTL1 (set prescale to divide-by-1, configure capture event polarity without resetting the counter), and ECEINT (enable desired eCAP interrupt).

18. Using the “PIE Interrupt Assignment Table” find the location for the ECAP1 interrupt \texttt{“ECAP1\_INT”} and fill in the following information:

\begin{verbatim}
PIE group #: \hfill  \# within group:
\end{verbatim}

This information will be used in the next step.

19. Modify the end of \texttt{ECap\_7\_8\_9\_10\_12.c} to do the following:

- Enable the “ECAP1\_INT” interrupt in the PIE (Hint: use the \texttt{PieCtrlRegs} structure)
- Enable the appropriate core interrupt in the IER register

**Build and Load**

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

**Run the Code – Pulse Width Measurement**

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

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

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

24. Run the code (real-time mode) by using the GEL function: GEL \rightarrow \texttt{Realtime Emulation Control} \rightarrow \texttt{Run\_Realtime\_with\_Reset}. Notice the values for \texttt{PwmDuty} and \texttt{PwmPeriod}.

25. Fully halt the CPU (real-time mode) by using the GEL function: GEL \rightarrow \texttt{Realtime Emulation Control} \rightarrow \texttt{Full\_Halt}.
Questions:

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

End of Exercise
Numerical Concepts

Introduction

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

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

Learning Objectives

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

### Numerical Concepts

- **Module Topics**
- **Numbering System Basics**
  - Binary Numbers
  - Two's Complement Numbers
  - Integer Basics
  - Sign Extension Mode
- **Binary Multiplication**
- **Binary Fractions**
  - Representing Fractions in Binary
  - Fraction Basics
  - Multiplying Binary Fractions
- **Fraction Coding**
- **Fractional vs. Integer Representation**
- **Floating-Point**
- **IQmath**
  - IQ Fractional Representation
  - Traditional “Q” Math Approach
  - IQmath Approach
- **IQmath Library**
- **Converting ADC Results into IQ Format**
- **AC Induction Motor Example**
- **IQmath Summary**
- **Lab 8: IQmath & Floating-Point FIR Filter**
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:

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

**Examples:**

**Integer Basics**

- **Unsigned Binary Integers**
  
  \[
  \begin{align*}
  0100b &= (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (0 \times 2^0) = 4 \\
  1101b &= (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 13
  \end{align*}
  \]

- **Signed Binary Integers (2’s Complement)**
  
  \[
  \begin{align*}
  0100b &= (0 \times -2^3) + (1 \times 2^2) + (0 \times 2^1) + (0 \times 2^0) = 4 \\
  1101b &= (1 \times -2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = -3
  \end{align*}
  \]
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></td>
<td></td>
</tr>
<tr>
<td>Load and sign extend</td>
<td></td>
</tr>
<tr>
<td>ACC</td>
<td>1111 1101</td>
</tr>
<tr>
<td></td>
<td>1101</td>
</tr>
</tbody>
</table>

= -2³ + 2² + 2⁰ = -3

= -2⁷ + 2⁶ + 2⁵ + 2⁴ + 2³ + 2² + 2⁰

= -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:

<table>
<thead>
<tr>
<th>Integer Multiplication (signed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
</tr>
<tr>
<td>x 1101</td>
</tr>
<tr>
<td>00000100</td>
</tr>
<tr>
<td>00000000</td>
</tr>
<tr>
<td>000100</td>
</tr>
<tr>
<td>11100</td>
</tr>
<tr>
<td>11110100</td>
</tr>
</tbody>
</table>

Accumulator: 11110100

Data Memory:

In this example, consider the following:

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

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.

\[
\begin{array}{c}
1.0111 = -1 + 1/4 + 1/8 = -5/8 \\
-1 \quad 1/2 \quad 1/4 \quad 1/8
\end{array}
\]

Fraction Basics

Fractions have the nice property that
fraction \times fraction = fraction
Multiplying Binary Fractions

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

<table>
<thead>
<tr>
<th>Fraction Multiplication</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
</tr>
<tr>
<td>x 1101</td>
</tr>
<tr>
<td>00000100</td>
</tr>
<tr>
<td>0000000</td>
</tr>
<tr>
<td>000100</td>
</tr>
<tr>
<td>11100</td>
</tr>
<tr>
<td>1110100</td>
</tr>
</tbody>
</table>

Fraction Multiplication

\[
\frac{1}{2} \times -\frac{3}{8} = \frac{-3}{16}
\]

Accumulator 1110100

Data Memory 1110 -1/4

As before, consider the following:

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

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

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

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

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

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

<table>
<thead>
<tr>
<th>Fraction</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>~1</td>
<td>0x7FFF</td>
</tr>
<tr>
<td>½</td>
<td>0x4000</td>
</tr>
<tr>
<td>0</td>
<td>0x0000</td>
</tr>
<tr>
<td>-½</td>
<td>0xC000</td>
</tr>
<tr>
<td>-1</td>
<td>0x8000</td>
</tr>
</tbody>
</table>

* 32768 (2^15)

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

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

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

**Examples:**
- To represent 0.62: 32768 * 62 / 100
- To represent 0.1405: 32768 * 1405 / 10000

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

<table>
<thead>
<tr>
<th>Integer vs. Fractions</th>
<th>Range</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>determined by # of bits</td>
<td>1</td>
</tr>
<tr>
<td>Fraction</td>
<td>~+1 to -1</td>
<td>determined by # of bits</td>
</tr>
</tbody>
</table>

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

Are there any other alternatives?

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

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

IEEE-754 Single Precision Floating-Point

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| s  | e   | f  |

1 bit sign 8 bit exponent 23 bit mantissa (fraction)

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

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

⇒ Case 3

\[ v = (-1)^s \times 2^{130-127} \times 1.25 = 10.0 \]

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

Number Line Insight

Floating-Point:

- Non-uniform distribution
  - Precision greatest near zero
  - Less precision the further you get from zero
Floating-Point Pros and Cons

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

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

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

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

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

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

**IQ Fractional Representation**

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

<table>
<thead>
<tr>
<th>S</th>
<th>IIIIIIIIII...ffffffffff</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32 bit mantissa</td>
<td></td>
</tr>
</tbody>
</table>

\[-2^1 + 2^{-1} - \ldots + 2^1 + 2^0 \cdot 2^{-1} + 2^{-2} + \ldots + 2^{-Q}\]

**I8Q24 Example:** 0x41200000

\[= 0100 0001 . 0010 0000 0000 0000 0000 0000 b\]

\[= 2^6 + 2^0 + 2^{-3} = 65.125\]

**Advantage** ⇒ Precision same for all numbers in an IQ format

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

### Number Line Insight

**Distributions**

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

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

**IQ Fractions:** uniform distribution (same precision everywhere)

### Traditional “Q” Math Approach

**Traditional 32-bit “Q” Math Approach**

$$y = mx + b$$

```c
Y = ((int64) M * (int64) X + (int64) B << Q) >> Q;
```

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

```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:

\[
\text{int32 } Y, M, X, B;
Y = ((\text{int64} \ M \ast \text{int64} \ X) \gg 24 + B;
\]

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

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

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

\[
\text{int32 } _\text{IQ24mpy}(\text{int32 } M, \text{int32 } X) \{ \text{ return } ((\text{int64} \ M \ast \text{int64} \ X) \gg 24; \}
\]

The linear equation can then be written as follows:

\[
Y = _\text{IQ24mpy}(M, X) + B;
\]

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

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

Redefine the multiply operation as follows:

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

This simplifies the equation as follows:

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

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

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

**IQmath Approach**

It looks like floating-point!

Floating-Point

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

Traditional Fix-Point Q

```c
long Y, M, X, B;
Y = ((\text{i64}) M \times (\text{i64}) X + (\text{i64}) B \ll Q)) \gg Q;
```

"IQmath" In C

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

"IQmath" In C++

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

"IQmath" code is easy to read!
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 // 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
_iq20 Y, M, X, B;
Y = _IQ23mpy(M, X) + B; // all values are in Q = 20
```

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:

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

The Q value must be consistent for all expressions in the same line of code.
IQmath Provides Compatibility Between Floating-Point and Fixed-Point

1) Develop any mathematical function

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

2) Select math type in IQmathLib.h

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

3) Compiler automatically converts to:

\[ Y = (float)M \times (float)X + (float)B; \]

Compile & Run on Fixed-Point
F282xx

Compile & Run on Floating-Point
F283xx *

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

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

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

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

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

IQmath Library: Math & Trig Functions

<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>type</td>
<td>float A, B;</td>
<td>_iq A, B;</td>
<td>iq A, B;</td>
</tr>
<tr>
<td>constant</td>
<td>A = 1.2345</td>
<td>A = _IQ(1.2345)</td>
<td>A = IQ(1.2345)</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 and power</td>
<td>sin(A),cos(A)</td>
<td>_IQsin(A), _IQcos(A)</td>
<td>IQsin(A),IQcos(A)</td>
</tr>
<tr>
<td>functions</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>asin(A),acos(A)</td>
<td>_IQasin(A), _IQacos(A)</td>
<td>_IQasin(A),_IQacos(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),sqrt(A)</td>
<td>_IQsqrt(A), _IQsqrt(A)</td>
<td>_IQsqrt(A),_IQsqrt(A)</td>
</tr>
<tr>
<td></td>
<td>exp(A)</td>
<td>_IQexp(A)</td>
<td>IQexp(A)</td>
</tr>
<tr>
<td>saturation</td>
<td>if(A &gt; Pos) A = Pos</td>
<td>_IQsat(A,Pos)</td>
<td>IQsat(A,Pos)</td>
</tr>
<tr>
<td></td>
<td>if(A &lt; Neg) A = Neg</td>
<td>_IQsat(A,Pos,Neg)</td>
<td>IQsat(A,Pos,Neg)</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

<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 = iqN (long)</td>
<td>A * (float) B</td>
<td>_IQmpy32(A,B)</td>
<td>IQmpy32(A,B)</td>
</tr>
<tr>
<td>integer(iq (long))</td>
<td>(long) (A * (float) B)</td>
<td>_IQmpy32int(A,B)</td>
<td>IQmpy32int(A,B)</td>
</tr>
<tr>
<td>fraction(iq (long))</td>
<td>A - (long) (A * (float) B)</td>
<td>_IQmpy32frac(A,B)</td>
<td>IQmpy32frac(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>_atolIQ(char)</td>
<td>atolIQ(char)</td>
</tr>
<tr>
<td>IQ to float</td>
<td>A</td>
<td>_IQtoF(A)</td>
<td>IQtoF(A)</td>
</tr>
<tr>
<td>IQ to ASCII</td>
<td>sprintf(A,B,C)</td>
<td>_IQtoA(A,B,C)</td>
<td>IQtoA(A,B,C)</td>
</tr>
</tbody>
</table>

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

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

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

Of course there is a concern on data RAM usage if numbers that could be represented in 16 bits all use 32 bits. This is becoming less of an issue in today's processors because of the finer technology used and the amount of RAM that can be cheaply integrated. However, in many cases, this problem can be mitigated by performing intermediate calculations using 32-bit numbers and converting the input from 16 to 32 bits and converting the output back to 16 bits before storing the final results. In many problems, it is the intermediate calculations that require additional accuracy to avoid quantization problems.
Converting ADC Results into IQ Format

As you may recall, the converted values of the ADC are placed in the lower 12 bits of the ADCRESULT0 register. Before these values are filtered using the IQmath library, they need to be put into the IQ format as a 32-bit long. For uni-polar ADC inputs (i.e., 0 to 3.3 V inputs), a conversion to global IQ format can be achieved with:

\[
\text{IQresult}_\text{unipolar} = \text{IQ}\text{mpy}_\text{v}(_\text{IQ}(3.3), _\text{IQ}12\text{toIQ}(_{_{\text{iq}}}\text{AdcResult.ADCRESULT0}));
\]

How can we modify the above to recover bi-polar inputs, for example +-1.65 volts? One could do the following to offset the +1.65V analog biasing applied to the ADC input:

\[
\text{IQresult}_\text{bipolar} = \text{IQ}\text{mpy}_\text{v}(_\text{IQ}(3.3), _\text{IQ}12\text{toIQ}(_{_{\text{iq}}}\text{AdcResult.ADCRESULT0}) - _\text{IQ}(1.65);
\]

However, one can see that the largest intermediate value the equation above could reach is 3.3. This means that it cannot be used with an IQ data type of IQ30 (IQ30 range is \(-2 < x < ~2\)). Since the IQmath library supports IQ types from IQ1 to IQ30, this could be an issue in some applications.

The following clever approach supports IQ types from IQ1 to IQ30:

\[
\text{IQresult}_\text{bipolar} = \text{IQ}\text{mpy}_\text{v}(_\text{IQ}(1.65), _\text{IQ}15\text{toIQ}(_{_{\text{iq}}} ((\text{int16}) (\text{AdcResult.ADCRESULT0} ^ 0x8000)))));
\]

The largest intermediate value that this equation could reach is 1.65. Therefore, IQ30 is easily supported.
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 than 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:
The complete system was coded using "IQmath". Based on analysis of coefficients in the system, the largest coefficient had a value of 33.3333. This indicated that a minimum dynamic range of 7 bits (+/-64 range) was required. Therefore, this translated to a GLOBAL_Q value of 32-7 = 25 (Q25). Just to be safe, the initial simulation runs were conducted with GLOBAL_Q = 24 (Q24)
value. The plots start from a step change in reference speed from 0.0 to 0.5 and 1024 samples are taken.

The speed eventually settles to the desired reference value and the stator current exhibits a clean and stable oscillation. The block diagram slide shows at which points in the control system the plots are taken from.

**What’s Happening Here?**

**Equal Precision in the Computation Region**

Floating-Point:

I8Q24 Fractions:

In the region where these particular computations occur, the precision of single-precision floating-point just happens to equal the precision of the I8Q24 format.

So, both produce similar results!
AC Induction Motor Example
GLOBAL_Q = 27, system unstable

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 developer must pick the right GLOBAL_Q value!

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

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

The great benefit of using GLOBAL_Q is that the user does not necessarily need to go into details to assign an individual Q for each variable in a whole system, as is typically done in conventional fixed-point programming. This is time consuming work. By using 32-bit resolution and the "IQmath" approach, the user can easily evaluate the overall resolution and quickly implement a typical digital motor control application without quantization problems.
Using the profiling capabilities of the respective DSP tools, the table above summarizes the number of cycles and code size of the forward and feedback control blocks.

The MIPS used is based on a system sampling frequency of 20 kHz, which is typical of such systems.
The IQmath approach, matched to a fixed-point processor with 32x32 bit capabilities enables the following:

- Seamless portability of code between fixed and floating-point devices
- Maintenance and support of one source code set from simulation to target device
- Adjustability of numerical resolution (Q value) based on application requirement
- Implementation of systems that may otherwise require floating-point device
- Rapid conversion/porting and implementation of algorithms
Lab 8: IQmath & Floating-Point FIR Filter

- Objective

The objective of this lab is to become familiar with IQmath programming. In the previous lab, ePWM1A was setup to generate a 2 kHz, 25% duty cycle symmetric PWM waveform. The waveform was then sampled with the on-chip analog-to-digital converter. In this lab the sampled waveform will be passed through an FIR filter and displayed using the graphing feature of Code Composer Studio. The filter math type is selected in the “IQmathLib.h” file.

- Procedure

Project File

1. A project named Lab8.pjt has been created for this lab. Open the project by clicking on Project ➔ Open... and look in C:\C28x\Labs\Lab8. All Build Options have been configured the same as the previous lab. The files used in this lab are:

   - Adc.c
   - CodeStartBranch.asm
   - DefaultIsr_8.c
   - DelayUs.asm
   - DSP2803x_GlobalVariableDefs.c
   - DSP2803x_Headers_nonBIOS.cmd
   - ECap_7_8_9_10_12.c
   - EPwm_7_8_9_10_12.c
   - Filter.c
   - Gpio.c
   - Lab_8.cmd
   - Main_8.c
   - PieCtrl_5_6_7_8_9_10.c
   - PieVect_5_6_7_8_9_10.c
   - SysCtrl.c
   - Watchdog.c
Project Build Options

2. Setup the include search path to include the IQmath header file. Open the Build Options and select the Compiler tab. In the Preprocessor Category, find the Include Search Path (-i) box and add to the end of the line (preceeded with a semicolon to append this directory to the existing search path):

`;..\IQmath\include`

3. Setup the library search path to include the IQmath library. Select the Linker tab.

   a. In the Libraries Category, find the Search Path (-i) box and enter:

`.;\IQmath\lib`

   b. In the Include Libraries (-l) box add to the end of the line (preceeded with a semicolon to append this library to the existing library):

`;IQmath.lib`

Then select OK to save the Build Options.

Include IQmathLib.h

4. In the CCS project window left click the plus sign (+) to the left of the Include folder. Edit Lab.h to *uncomment* the line that includes the IQmathLib.h header file. Next, in the Function Prototypes section, *uncomment* the function prototype for IQssfir(), the IQ math single-sample FIR filter function. In the Global Variable References section *uncomment* the two _iq references. Save the changes and close the file.

Inspect Lab_8.cmd

5. Open and inspect Lab_8.cmd. First, notice that a section called “IQmath” is being linked to L0SARAM. The IQmath section contains the IQmath library functions (code). Second, notice that a section called “IQmathTables” is being linked to the IQTABLES with a TYPE = NOLOAD modifier after its allocation. The IQmath tables are used by the IQmath library functions. The NOLOAD modifier allows the linker to resolve all addresses in the section, but the section is not actually placed into the .out file. This is done because the section is already present in the device ROM (you cannot load data into ROM after the device is manufactured!). The tables were put in the ROM by TI when the device was manufactured. All we need to do is link the section to the addresses where it is known to already reside (the tables are the very first thing in the BOOT ROM, starting at address 0x3FE000). Close the inspected file.
Select a Global IQ value

6. Use File → Open… to open c:\C28x\Labs\IQmath\include\IQmathLib.h. Confirm that the GLOBAL_Q type (near beginning of file) is set to a value of 24. If it is not, modify as necessary:

```
#define   GLOBAL_Q       24
```

Recall that this Q type will provide 8 integer bits and 24 fractional bits. Dynamic range is therefore $-128 \leq x < +128$, which is sufficient for our purposes in the workshop.

Notice that the math type is defined as IQmath by:

```
#define   MATH_TYPE      IQ_MATH
```

Close the file.

IQmath Single-Sample FIR Filter

7. Open and inspect DefaultIsr_8.c. Notice that the ADCINT_ISR calls the IQmath single-sample FIR filter function, IQssfir(). The filter coefficients have been defined in the beginning of Main_8.c.

8. Open and inspect the IQssfir() function in Filter.c. This is a simple, non-optimized coding of a basic IQmath single-sample FIR filter. Close the inspected files.

Build and Load

9. Click the “Build” button to build and load the project.

Run the Code – Filtered Waveform

10. Open a memory window to view some of the contents of the filtered ADC results buffer. The address label for the filtered ADC results buffer is AdcBufFiltered. Set the Format to 16-Bit Unsigned Integer. We will be running our code in real-time mode, and will have our window continuously refresh.

Note: For the next step, check to be sure that the jumper wire connecting PWM1A (pin # GPIO-00) to ADCINA0 (pin # ADC-A0) is in place on the Docking Station.

11. Run the code in real-time mode using the GEL function: GEL → Realtime Emulation Control → Run_Realtime_with_Reset, and watch the memory window update. Verify that the ADC result buffer contains updated values.

12. Open and setup a dual-time graph to plot a 50-point window of the filtered and unfiltered ADC results buffer. Click: View → Graph → Time/Frequency… and set the following values:
Display Type | Dual Time
--- | ---
Start Address – upper display | AdcBufFiltered
Start Address – lower display | AdcBuf
Acquisition Buffer Size | 50
Display Data Size | 50
DSP Data Type | 16-bit unsigned integer
Sampling Rate (Hz) | 50000
Time Display Unit | μs

Select **OK** to save the graph options.

13. The graphical display should show the generated FIR filtered 2 kHz, 25% duty cycle symmetric PWM waveform in the upper display and the unfiltered waveform generated in the previous lab exercise in the lower display. Notice the shape and phase differences between the waveform plots (the filtered curve has rounded edges, and lags the unfiltered plot by several samples). The amplitudes of both plots should run from 0 to 4095.

14. Open and setup two (2) frequency domain plots – one for the filtered and another for the unfiltered ADC results buffer. Click: View → Graph → Time/Frequency... and set the following values:

<table>
<thead>
<tr>
<th>GRAPH #1</th>
<th>GRAPH #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display Type</td>
<td>FFT Magnitude</td>
</tr>
<tr>
<td>Start Address</td>
<td>AdcBufFiltered</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.

15. The graphical displays should show the frequency components of the filtered and unfiltered 2 kHz, 25% duty cycle symmetric PWM waveforms. Notice that the higher frequency components are reduced using the Low-Pass FIR filter in the filtered graph as compared to the unfiltered graph.
16. Fully halt the CPU (real-time mode) by using the GEL function: GEL → Realtime Emulation Control → Full_Halt.

End of Exercise
Lab 8 Reference: Low-Pass FIR Filter

Bode Plot of Digital Low Pass Filter

Coefficients: \([1/16, 4/16, 6/16, 4/16, 1/16]\)

Sample Rate: 50 kHz
Control Law Accelerator

Introduction

This module explains the operation of the control law accelerator (CLA). The CLA is an independent, fully programmable, 32-bit floating-point math processor that enables concurrent execution into the C28x family. This extends the capabilities of the C28x CPU by adding parallel processing. The CLA has direct access to the ADC result registers, and all ePWM, HRPWM and comparator registers. This allows the CLA to read ADC samples “just-in-time” and significantly reduces the ADC sample to output delay enabling faster system response and higher frequency operation. Utilizing the CLA for time-critical tasks frees up the CPU to perform other system and communication functions concurrently.

Learning Objectives

- Explain the purpose and operation of the Control Law Accelerator (CLA)
- Describe the CLA initialization procedure
- Review the CLA registers, instruction set, and programming flow
Module Topics

Control Law Accelerator ........................................................................................................................................ 9-1

Module Topics .................................................................................................................................................. 9-2

Control Law Accelerator (CLA) .................................................................................................................. 9-3
CLA Block Diagram .......................................................................................................................................... 9-3
CLA Memory and Register Access ................................................................................................................ 9-4
CLA Tasks ....................................................................................................................................................... 9-4
Control and Execution Registers .................................................................................................................. 9-5
CLA Registers .................................................................................................................................................. 9-6
CLA Initialization ............................................................................................................................................. 9-8
CLA Task Programming ............................................................................................................................... 9-9
CLA Instruction Set ........................................................................................................................................ 9-10
CLA Addressing Modes .................................................................................................................................. 9-11
CLA Code Example ....................................................................................................................................... 9-11
CLA Code Debugging ..................................................................................................................................... 9-12

Lab 9: CLA Floating-Point FIR Filter ............................................................................................................. 9-13
Control Law Accelerator (CLA)

CLA is an independent 32-bit floating-point math accelerator
- Executes algorithms independently and in parallel with the main CPU
- Direct access to ePWM / HRPWM, ADC result and comparator registers
- Responds to peripheral interrupts independently of CPU
- Frees-up CPU for other tasks (communications and diagnostics)

CLA Block Diagram

Task Triggers (Peripheral Interrupts)
- ADCINT1 or EPWM1_INT
- ADCINT7 or EPWM7_INT
- ADCINT8 or CPU Timer 0

CLA Control & Execution Registers

CLA Program Bus
- Prog RAM

CLA Data Bus
- Data RAM0
- Data RAM1

MSG RAMs
- CPU to CLA
- CLA to CPU

Periph. Regs
- ADC Results
- ePWM
- HRPWM
- Comparator
CLA Memory and Register Access

CLA Program Memory
- Contains CLA program code
- Mapped to the CPU at reset
- Initialized by the CPU

Message RAMs
- Used to pass data between the CPU and CLA
- Always mapped to both the CPU and CLA

CLA Data Memory
- Contains variables and coefficients used by the CLA program code
- Mapped to the CPU at reset
- Initialized by CPU

Peripheral Reg Access
- ADC Results Regs
- ePWM (all regs)
- HRPWM (all regs)
- Comparator (all regs)

CLA Tasks

- A Task is similar to an interrupt service routine
- CLA supports 8 Tasks (Task1-8)
- A task is started by a peripheral interrupt trigger
  - Triggers are enabled in the MPISRCSEL1 register
- When a trigger occurs the CLA begins execution at the associated task vector entry (MVECT1-8)
- Once a task begins it runs to completion (no nesting)
  - A task is terminated with an MSTOP instruction
Software triggering a task

- Tasks can also be started by a software trigger using the CPU

**Method #1: Write to Interrupt Force Register (MIFRC) register**

```c
asm(" EALLOW"); // enable protected register access
Cla1Regs.MIFRC.bit.INT4 = 1; // start task 4
asm(" EDIS"); // disable protected register access
```

**Method #2: Use IACK instruction**

```c
asm(" IACK #0x0008"); // set bit 4 in MIFR to start task 4
```

More efficient – does not require EALLOW

Note: Use of IACK requires Cla1Regs.MCTL.bit.IACKE = 1

Control and Execution Registers

**CLA Control and Execution Registers**

- MPISRCSEL1 – Peripheral Interrupt Source Select (Task 1-8)
- MVECT1-8 – Task Interrupt Vector (MVECT1/2/3/4/5/6/7/8)
- MMEMCFG – Memory Map Configuration (RAM1E, RAM0E, PROGE)
- MPC – 12-bit Program Counter (initialized by appropriate MVECTx register)
- MR0-3 – CLA Floating-Point 32-bit Result Registers
- MAR0-1 – CLA Auxiliary Registers
## CLA Registers

### Register Description

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCTL</td>
<td>Control Register</td>
</tr>
<tr>
<td>MMEMCFG</td>
<td>Memory Configuration Register</td>
</tr>
<tr>
<td>MIPSRSEL1</td>
<td>Peripheral Interrupt Source Select 1 Register</td>
</tr>
<tr>
<td>MIFR</td>
<td>Interrupt Flag Register</td>
</tr>
<tr>
<td>MIER</td>
<td>Interrupt Enable Register</td>
</tr>
<tr>
<td>MFRC</td>
<td>Interrupt Force Register</td>
</tr>
<tr>
<td>MICLR</td>
<td>Interrupt Flag Clear Register</td>
</tr>
<tr>
<td>MOVF</td>
<td>Interrupt Overflow Flag Register</td>
</tr>
<tr>
<td>MIOVF</td>
<td>Interrupt Overflow Flag Clear Register</td>
</tr>
<tr>
<td>MICLROVF</td>
<td>Interrupt Overflow Flag Clear Register</td>
</tr>
<tr>
<td>MIRUN</td>
<td>Interrupt Run Status Register</td>
</tr>
<tr>
<td>MVECTx</td>
<td>Task x interrupt Vector (x = 1-8)</td>
</tr>
<tr>
<td>MPC</td>
<td>CLA 12-bit Program Counter</td>
</tr>
<tr>
<td>MARx</td>
<td>CLA Auxiliary Register x (x = 0-1)</td>
</tr>
<tr>
<td>MRx</td>
<td>CLA Floating-Point 32-bit Result Register (x = 0-3)</td>
</tr>
<tr>
<td>MSTF</td>
<td>CLA Floating-Point Status Register</td>
</tr>
</tbody>
</table>

### CLA Control Register

**Cla1Regs.MCTL**

- **IACK Enable**
  - 0 = CPU IACK instruction ignored
  - 1 = CPU IACK instruction triggers a task

- **Hard Reset**
  - 0 = no effect
  - 1 = CLA reset (registers set to default state)

- **Soft Reset**
  - 0 = no effect
  - 1 = CLA reset (stop current task)
**CLA Memory Configuration Register**

Cla1Regs.MMEMCFG

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 6</td>
<td>reserved</td>
</tr>
<tr>
<td>5</td>
<td>RAM1E</td>
</tr>
<tr>
<td>4</td>
<td>RAM0E</td>
</tr>
<tr>
<td>2 - 1</td>
<td>reserved</td>
</tr>
<tr>
<td>0</td>
<td>PROGE</td>
</tr>
</tbody>
</table>

**CLA Program Space Enable**

0 = mapped to CPU program and data space
1 = mapped to CLA program space

**CLA Data RAM1 / RAM0 Enable**

0 = mapped to CPU program and data space
1 = mapped to CLA data space

**CLA Peripheral Interrupt Source Select 1 Register**

Cla1Regs.MPISRCSEL1

<table>
<thead>
<tr>
<th>Task 8 Peripheral Interrupt Input</th>
<th>Task 7 Peripheral Interrupt Input</th>
<th>Task 6 Peripheral Interrupt Input</th>
<th>Task 5 Peripheral Interrupt Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 = ADCINT8</td>
<td>000 = ADCINT7</td>
<td>000 = ADCINT6</td>
<td>000 = ADCINT5</td>
</tr>
<tr>
<td>010 = CPU Timer 0 xx1 = no source</td>
<td>010 = ePWM7 xx1 = no source</td>
<td>010 = ePWM6 xx1 = no source</td>
<td>010 = ePWM5 xx1 = no source</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31 - 28</th>
<th>27 - 24</th>
<th>23 - 20</th>
<th>19 - 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>PERINT8SEL</td>
<td>PERINT7SEL</td>
<td>PERINT6SEL</td>
<td>PERINT5SEL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>15 - 12</th>
<th>11 - 8</th>
<th>7 - 4</th>
<th>3 - 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>PERINT4SEL</td>
<td>PERINT3SEL</td>
<td>PERINT2SEL</td>
<td>PERINT1SEL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task 4 Peripheral Interrupt Input</th>
<th>Task 3 Peripheral Interrupt Input</th>
<th>Task 2 Peripheral Interrupt Input</th>
<th>Task 1 Peripheral Interrupt Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 = ADCINT4</td>
<td>000 = ADCINT3</td>
<td>000 = ADCINT2</td>
<td>000 = ADCINT1</td>
</tr>
<tr>
<td>010 = ePWM4 xx1 = no source</td>
<td>010 = ePWM3 xx1 = no source</td>
<td>010 = ePWM2 xx1 = no source</td>
<td>010 = ePWM1 xx1 = no source</td>
</tr>
</tbody>
</table>

Note: select xx1 (no source) if task is generated by software

000 = Default
CLA Interrupt Enable Register

Cla1Regs.MIER

<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>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>

0 = task interrupt disable (default)
1 = task interrupt enable

```
#include "DSP2803x_Device.h"
Cla1Regs.MIER.bit.INT2 = 1;  //enable Task 2 interrupt
Cla1Regs.MIER.all = 0x0028;  //enable Task 6 and 4 interrupts
```

CLA Initialization

CLA initialization is performed by the CPU in C code (typically done with the Peripheral Register Header Files)

1. Copy CLA task code from flash to CLA program RAM
2. Initialize CLA data RAMs, as needed
   - Populate with data coefficients, constants, etc.
3. Configure the CLA registers
   - Enable the CLA clock (PCLKCR3 register)
   - Populate the CLA task interrupt vectors (MVECT1-8 registers)
   - Select the desired task interrupt sources (PERINT1SEL register)
   - If desired, enable IACK to start task using software (avoids EALLOW)
   - Map CLA program RAM and data RAMs to CLA space
4. Configure desired CLA task completion interrupts in the PIE
5. Enable CLA tasks triggers in the MIER register
6. Initialize the ePWM and/or ADC to trigger the CLA tasks

Data is passed between the CLA and CPU via message RAMs
Enabling CLA Support in CCS

- Note: You must be using a C28x Piccolo device that has the Control Law Accelerator!
- In the project build options, select: ‘cla0 (From Device Type 0)’
- This is required in order to assemble CLA code
- CLA support requires codegen tools v5.2.0 or later

CLA Task Programming

CLA tasks are written in assembly code

- Same instruction format as the C28x and C28x+FPU
  - Destination operand is always on the left
  - Same mnemonics as C28x+FPU but with a leading “M”

```
CPU:      MPY    ACC, T,  loc16
FPU:      MPYF32  R0H, R1H, R2H
CLA:      MMPYF32 MR0, MR1, MR2
```

Destination      Source Operands
CLA Instruction Set

CLA Instruction Overview

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load (Conditional)</td>
<td>MMOV32 MRa,mem32,(COND)</td>
<td>1</td>
</tr>
<tr>
<td>Store</td>
<td>MMOV32 mem32,MRa</td>
<td>1</td>
</tr>
<tr>
<td>Load with Data Move</td>
<td>MMOV32 MRa,mem32</td>
<td>1</td>
</tr>
<tr>
<td>Store/Load MSTF</td>
<td>MMOV32 MSTF,mem32</td>
<td>1</td>
</tr>
<tr>
<td>Compare, Min, Max</td>
<td>CMMPF32 MRa,MRb</td>
<td>1</td>
</tr>
<tr>
<td>Absolute, Negative Value</td>
<td>MARSF32 MRa,MRb</td>
<td>1</td>
</tr>
<tr>
<td>Unsigned Integer to Float</td>
<td>MUI16TOF32 MRa,mem16</td>
<td>1</td>
</tr>
<tr>
<td>Integer to Float</td>
<td>M132TOF32 MRa,mem32</td>
<td>1</td>
</tr>
<tr>
<td>Float to Integer &amp; Round</td>
<td>MF32TO116R MRa,MRb</td>
<td>1</td>
</tr>
<tr>
<td>Integer Shift</td>
<td>MBCNDD 16bitdest (,COND)</td>
<td>1-7</td>
</tr>
<tr>
<td>Multiply, Add, Subtract</td>
<td>MMVPF32 MRa,MRb,MRc</td>
<td>1</td>
</tr>
<tr>
<td>1/X (16-bit Accurate)</td>
<td>MBINVF32 MRa,MRb</td>
<td>1</td>
</tr>
<tr>
<td>1/Sqrt(x) (16-bit Accurate)</td>
<td>MBISQRTF32 MRa,MRb</td>
<td>1</td>
</tr>
<tr>
<td>Integer Load/Store Auxiliary Register</td>
<td>MMOV16 MRa,mem16</td>
<td>1</td>
</tr>
<tr>
<td>Conditional Delayed</td>
<td>MBCNDD 16bitdest {,COND}</td>
<td>1</td>
</tr>
<tr>
<td>Integer Bitwise AND, OR, XOR</td>
<td>NAND32 MRa,MRb,MRc</td>
<td>1</td>
</tr>
<tr>
<td>Integer Add and Subtract</td>
<td>MSUB32 MRa,MRb,MRc</td>
<td>1</td>
</tr>
<tr>
<td>Integer Shifts</td>
<td>MLSR32 MRa,#SHIFT</td>
<td>1</td>
</tr>
<tr>
<td>Write Protection Enable/Disable</td>
<td>MEALLOW MRa,MRb</td>
<td>1</td>
</tr>
<tr>
<td>Half Code or End Task</td>
<td>MSTP</td>
<td>1</td>
</tr>
<tr>
<td>No Operation</td>
<td>MNOP</td>
<td>1</td>
</tr>
</tbody>
</table>

CLA Parallel Instructions

- Parallel bars indicate a parallel instruction
- Parallel instructions operate as a single instruction with a single opcode and performs two operations

- Example: Add + Parallel Store

\[
\begin{align*}
\text{MADDF32 } & \text{ MR3, MR3, MR1} \\
\| & \text{ MMOV32 } @_\text{Var}, \text{MR3}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply &amp; Parallel Add/Subtract</td>
<td>MMVPF32 MRa,MRb,MRc</td>
<td></td>
</tr>
<tr>
<td>Multiply, Add, Subtract &amp; Parallel Store</td>
<td>MADDF32 MRa,MRb,MRc</td>
<td></td>
</tr>
<tr>
<td>Multiply, Add, Subtract, MAC &amp; Parallel Load</td>
<td>MADDF32 MRa,MRb,MRc</td>
<td></td>
</tr>
</tbody>
</table>

Both operations complete in a single cycle
CLA Addressing Modes

CLA has two addressing modes

- Both modes can access the low 64Kw of memory:
  - All of the CLA data space
  - Both message RAMs
  - Shared peripheral registers
- There is no stack pointer or data page pointer

Direct Addressing Mode:
- Populates opcode field with 16-bit address of the variable

Example 1:  
```
MMOV32 MR1, @_VarA
```
Example 2:  
```
MMOV32 MR1, @_EPwm1Regs.CMPA.all
```

Indirect Addressing with 16-bit Post Increment:
- Uses the address in MAR0 or MAR1 to access memory
- After the read or write MAR0/MAR1 is incremented by #Imm16

Example 1:  
```
MMOV32 MR0, *MAR0[2]++
```
Example 2:  
```
MMOV32 MR1, *MAR1[-2]++
```

CLA Code Example

```
.cdecls
"Lab.h"
.sect "Cla1Prog"
_CLa1Prog_Start
_CLa1Task1: ; FIR filter
    MPU16TO32 MR2, @AdcResult.ADCRESULT0
    MMPYF32 MR2, MR1, MR0
    MADDF32 MR3, MR3, MR2
    MF32TOUI16 MR2, MR3
    MMOV16 @ClaFilteredOutput, MR2
    MSTOP
; End of task

;-------------------------------------
_CLa1Task2:
    MSTOP
;-------------------------------------
_CLa1Task3:
    MSTOP
```

- .cdecls directive used to include the C header file in the CLA assembly file
- .sect directive used to place CLA assembly code in its own section
- C Peripheral Register Header File references can be used in CLA assembly code
- MSTOP instruction used at the end of the task
- CLA assembly and C28 C-code reside in the same project
CLA Code Example (2 of 2)

```c
#include "DSP2803x_Device.h"

extern Uint32 Cla1Prog_Start;
extern Uint32 Cla1Task1;
extern Uint32 Cla1Task2;
;
extern Uint32 Cla1Task8;

// Symbols used to calculate vector address
Cla1Regs.MVECT1 = (Uint16)((Uint32)&Cla1Task1 - (Uint32)&Cla1Prog_Start);
Cla1Regs.MVECT2 = (Uint16)((Uint32)&Cla1Task2 - (Uint32)&Cla1Prog_Start);

include "Lab.h"
```

CLA Code Debugging

- The CLA can halt, single-step and run independently from the CPU
- Both the CLA and CPU are debugged from the same JTAG port

1. Insert a breakpoint in CLA code
   - Insert MDEBUGSTOP instruction to halt CLA and then rebuild/reload
2. Enable CLA breakpoints
   - Enable CLA breakpoints in the debugger
3. Start the task
   - Done by peripheral interrupt, software (IACK) or MIFRC register
   - CLA executes instructions until MDEBUGSTOP
   - MPC will have the address of MDEBUGSTOP instruction
4. Single step the CLA code
   - Once halted, single step the CLA code
   - Can also run to the next MDEBUGSTOP or to the end of task
   - If another task is pending it will start at end of previous task
5. Disable CLA breakpoints, if desired
   - CLA single step – CLA pipeline is clocked only one cycle and then frozen
   - CPU single step – CPU pipeline is flushed for each single step
Lab 9: CLA Floating-Point FIR Filter

Objective

The objective of this lab is to become familiar with operation of the CLA. In the previous lab, the CPU was used to filter the ePWM1A generated 2 kHz, 25% duty cycle symmetric PWM waveform. In this lab, the PWM waveform will be filtered using the CLA. The CLA will directly read the ADC result register and a task will run a low-pass FIR filter on the sampled waveform. The filtered result will be stored in a circular memory buffer. Note that the CLA is operating concurrently with the CPU. As an operational test, the filtered and unfiltered waveforms will be displayed using the graphing feature of Code Composer Studio.

Procedure

Project File

1. A project named Lab9.pjt has been created for this lab. Open the project by clicking on Project ➔ Open... and look in C:\C28x\Labs\Lab9. All Build Options have been configured the same as the previous lab. The files used in this lab are:
   - Adc.c
   - Cla_9.c
   - ClaTasks.asm
   - CodeStartBranch.asm
   - DefaultIsr_9_10.c
   - DelayUs.asm
   - DSP2803x_GlobalVariableDefs.c
   - DSP2803x_Headers_nonBIOS.cmd
   - ECap_7_8_9_10_12.c
   - EPwm_7_8_9_10_12.c
   - Filter.c
   - Gpio.c
   - Lab_9.cmd
   - Main_9.c
   - PieCtrl_5_6_7_8_9_10.c
   - PieVect_5_6_7_8_9_10.c
   - SysCtrl.c
   - Watchdog.c
Enabling CLA Support in CCS

2. Open the **Build Options** and select the Compiler tab. In the Basic Category set the **Specify CLA Support** to **cla0 (From Device Type 0)**. This is needed to assemble CLA code. Then select **OK** to save the **Build Options**.

Inspect Lab_9.cmd

3. Open and inspect **Lab_9.cmd**. Notice that a section called **“Cla1Prog”** is being linked to **L3DPSARAM**. This section links the CLA program tasks (assembly code) to the CPU memory space. This memory space will be remapped to the CLA memory space during initialization. Also, notice the two message RAM sections used to pass data between the CPU and CLA.

Setup CLA Initialization

During the CLA initialization, the CPU memory block **L3DPSARAM** needs to be configured as CLA program memory. This memory space contains the CLA Task routines, which are coded in assembly. The CLA Task 1 has been configured to run an FIR filter. The CLA needs to be configured to start Task 1 on the ADCINT1 interrupt trigger. The next section will setup the PIE interrupt for the CLA.

4. Open **ClaTasks.asm** and notice that the .cdecls directive is being used to include the C header file in the CLA assembly file. Therefore, we can use the Peripheral Register Header File references in the CLA assembly code. Next, notice Task 1 has been configured to run an FIR filter. Within this code special instructions have been used to convert the ADC result integer (i.e. the filter input) to floating-point and the floating-point filter output back to integer.

5. Edit **Cla_9.c** to implement the CLA operation as described in the objective for this lab exercise. Configure the **L3DPSARM** memory block to be mapped to CLA program memory space. Set Task 1 peripheral interrupt source to ADCINT1 and set the other Task peripheral interrupt source inputs to no source. Enable CLA Task 1 interrupt.

6. Open **Main_9.c** and add a line of code in **main()** to call the **InitCla()** function. There are no passed parameters or return values. You just type

   ```c
   InitCla();
   ```

   at the desired spot in **main()**.

Setup PIE Interrupt for CLA

Recall that ePWM2 is triggering the ADC at a 50 kHz rate. In the previous lab exercise, the ADC generated an interrupt to the CPU, and the CPU implemented the FIR filter in the ADC ISR. For this lab exercise, the ADC is instead triggering the CLA, and the CLA will directly read the ADC result register and run a task implementing an FIR filter. The CLA will generate an interrupt to the CPU, which will store the filtered results to a circular buffer implemented in the CLA ISR.
7. Edit Adc.c to *comment out* the code used to enable ADCINT1 interrupt in PIE group 1. This is no longer being used. The CLA interrupt will be used instead.

8. Using the “PIE Interrupt Assignment Table” find the location for the CLA Task 1 interrupt “CLA1_INT1” and fill in the following information:

   PIE group #:_________  # within group:_________

   This information will be used in the next step.

9. Modify the end of Cla_9.c to do the following:
   - Enable the "CLA1_INT1" interrupt in the PIE (Hint: use the PieCtrlRegs structure)
   - Enable the appropriate core interrupt in the IER register

10. Open and inspect DefaultIsr_9_10.c. Notice that this file contains the CLA interrupt service routine. Save and close all modified files.

### Build and Load

11. Click the “Build” button to build and load the project.

### Run the Code – Test the CLA Operation

**Note:** For the next step, check to be sure that the jumper wire connecting PWM1A (pin # GPIO-00) to ADCINA0 (pin # ADC-A0) is in place on the Docking Station.

12. Run the code in real-time mode using the GEL function: GEL → Realtime Emulation Control → Run Realtime with Reset, and watch the memory window update. Verify that the ADC result buffer contains updated values.

13. Setup a dual-time graph of the filtered and unfiltered ADC results buffer. Click: View → Graph → Time/Frequency... and set the following values:
14. The graphical display should show the filtered PWM waveform in the upper display and the unfiltered waveform in the lower display. You should see that the results match the previous lab exercise.

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

**End of Exercise**
System Design

Introduction

This module discusses various aspects of system design. Details of the emulation and analysis block along with JTAG will be explored. Flash memory programming and the Code Security Module will be described.

Learning Objectives

- Emulation and Analysis Block
- 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

Flash Configuration and Memory Performance ....................................................................................10-6

Flash Programming ...............................................................................................................................10-9

Code Security Module (CSM) ..............................................................................................................10-11

Lab 10: Programming the Flash ...........................................................................................................10-14
Emulation and Analysis Block

JTAG Emulation System
(based on IEEE 1149.1 Boundary Scan Standard)

Some Available Emulators

- **XDS510 CLASS**
  - BlackHawk: USB2000
  - Signum System: JTAGjet-TMS-C2000
  - Spectrum Digital: XDS510LC

- **XDS100 CLASS**
  - BlackHawk: USB100
  - Olimex: TMS320-JTAG-USB
  - Spectrum Digital: XDS100
  - TI: TMDSEMU100U-14T

These emulators are C2000 specific, and are much lower cost than emulators that support all TI MCU/DSP platforms (although those can certainly be used).

These emulators are much slower than the ones listed above, but are also available at a lower cost than XDS510 class and are NOT C2000 specific.

Emulator Connections to the Device

- **TMS320F2803x**
- **Emulator Header**
  - EMU0
  - PD
  - EMU1
- **Emulator Connections**
  - TRST
  - TMS
  - TDI
  - TDO
  - TCK
  - TCK_RET
  - GND
  - Vcc (3.3 V)

= 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**

[Diagram of watchpoint settings]

- Symbolic or numeric address
- Mask value for specifying address ranges
- Bus selection
- Address with Data selection

**On-Chip Emulation Analysis Block: Online Stack Overflow Detection**

- Emulation analysis registers are accessible to code as well!
- Configure a watchpoint to monitor for writes near the end of the stack
- Watchpoint triggers maskable RTOSINT interrupt
- Works with DSP/BIOS and non-DSP/BIOS
  - See TI application report SPRA820 for implementation details

[Diagram of stack overflow detection]

- Region of memory occupied by the stack
- Stack grows towards higher memory addresses
- Monitor for data writes in region near the end of the stack

Data Memory
Flash Configuration and Memory Performance

**Basic Flash Operation**

- Flash is arranged in pages of 128 words
- Wait states are specified for consecutive accesses within a page, and random accesses across pages
- OTP has random access only
- Must specify the number of SYSCLKOUT wait-states; Reset defaults are maximum value (15)
- Flash configuration code should not be run from the Flash memory

Flash configuration code should not be run from the Flash memory.

For 60 MHz, PAGEWAIT = 2, RANDWAIT = 2, OTPWAIT = 3

 *** Refer to the F2803x datasheet for detailed numbers ***

```
16 or 32 dispatched
16 or 32 dispatched
```

2-level deep fetch buffer

C28x Core decoder unit

```
FlashPipeline Enable
0 = disable (default)
1 = enable
```

---

**Speeding Up Code Execution in Flash**

Flash Pipelining (for code fetch only)

```
FlashRegs.FOPT.bit.ENPIPE = 1;
```

---

For 60 MHz, PAGEWAIT = 2, RANDWAIT = 2, OTPWAIT = 3

---

*** Refer to the F2803x datasheet for detailed numbers ***
**Code Execution Performance**

- Assume 60 MHz SYSCLKOUT, 16-bit instructions
  
  (80% of instructions are 16 bits wide – Rest are 32 bits)

**Internal RAM:** 60 MIPS

Fetch up to 32-bits every cycle ➔ 1 instruction/cycle * 60 MHz = 60 MIPS

**Flash (w/ pipelining):** 60 MIPS

RANDWAIT = 2

Fetch 64 bits every 3 cycles, but it will take 4 cycles to execute them ➔

4 instructions/4 cycles * 60 MHz = 60 MIPS

RPT will increase this; PC discontinuity will degrade this

Benchmarking in control applications has shown actual performance of about 54 MIPS

---

**Data Access Performance**

- Assume 60 MHz SYSCLKOUT

<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>
</tbody>
</table>
| Flash         | 0.33                       | 0.33                       | RANDWAIT = 2

Flash is read only!

- Internal RAM has best data performance – put time critical data here
- Flash performance usually sufficient for most constants and tables
- Note that the flash instruction fetch pipeline will also stall during a flash data access
## Other Flash Configuration Registers

<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, FACTIVEWAIT**: Specify # of delay cycles during wake-up from sleep to standby, and from standby to active, respectively. The delay is needed to let the flash stabilize. Leave these registers set to their default maximum value.

See the "TMS320x2803x Piccolo System Control and Interrupts Reference Guide," SPRUGL8, for more information
Flash Programming 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

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

- **JTAG Emulator Based**
  - Code Composer Studio Plug-in
  - BlackHawk Flash utilities (requires Blackhawk emulator)
  - Elprotronic FlashPro2000
  - Spectrum Digital SDFlash JTAG (requires SD emulator)
  - Signum System Flash utilities (requires Signum emulator)

- **SCI Serial Port Bootloader Based**
  - Code-Skin (http://www.code-skin.com)
  - Elprotronic FlashPro2000

- **Production Test/Programming Equipment Based**
  - BP Micro programmer
  - Data I/O programmer

- **Build your own custom utility**
  - Can use any of the ROM bootloader methods
  - Can embed flash programming into your application
  - Flash API algorithms provided by TI

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

---

**Code Composer Studio Flash Plug-In**

![Code Composer Studio Flash Plug-In](image)
Code Security Module (CSM)

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

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

CSM Password

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

CSM Registers

Key Registers – accessible by user; EALLOW protected

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00 0AE0</td>
<td>KEY0</td>
<td>Low word of 128-bit Key register</td>
</tr>
<tr>
<td>0x00 0AE1</td>
<td>KEY1</td>
<td>2nd word of 128-bit Key register</td>
</tr>
<tr>
<td>0x00 0AE2</td>
<td>KEY2</td>
<td>3rd word of 128-bit Key register</td>
</tr>
<tr>
<td>0x00 0AE3</td>
<td>KEY3</td>
<td>4th word of 128-bit Key register</td>
</tr>
<tr>
<td>0x00 0AE4</td>
<td>KEY4</td>
<td>5th word of 128-bit Key register</td>
</tr>
<tr>
<td>0x00 0AE5</td>
<td>KEY5</td>
<td>6th word of 128-bit Key register</td>
</tr>
<tr>
<td>0x00 0AE6</td>
<td>KEY6</td>
<td>7th word of 128-bit Key register</td>
</tr>
<tr>
<td>0x00 0AE7</td>
<td>KEY7</td>
<td>High word of 128-bit Key register</td>
</tr>
<tr>
<td>0x00 0AEF</td>
<td>CSMSCR</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>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3F 7FF8</td>
<td>PWL0</td>
<td>Low word of 128-bit password</td>
</tr>
<tr>
<td>0x3F 7FF9</td>
<td>PWL1</td>
<td>2nd word of 128-bit password</td>
</tr>
<tr>
<td>0x3F 7FFA</td>
<td>PWL2</td>
<td>3rd word of 128-bit password</td>
</tr>
<tr>
<td>0x3F 7FFB</td>
<td>PWL3</td>
<td>4th word of 128-bit password</td>
</tr>
<tr>
<td>0x3F 7FFC</td>
<td>PWL4</td>
<td>5th word of 128-bit password</td>
</tr>
<tr>
<td>0x3F 7FFD</td>
<td>PWL5</td>
<td>6th word of 128-bit password</td>
</tr>
<tr>
<td>0x3F 7FFE</td>
<td>PWL6</td>
<td>7th word of 128-bit password</td>
</tr>
<tr>
<td>0x3F 7FFF</td>
<td>PWL7</td>
<td>High word of 128-bit password</td>
</tr>
</tbody>
</table>

Locking and Unlocking the CSM

◆ The CSM is always locked after reset
◆ To unlock the CSM:
  • Perform a dummy read of each PWL (passwords in the flash)
  • Write the correct password to each KEY register
◆ Passwords are all 0xFFFF on new devices
  • When passwords are all 0xFFFF, only a read of each PWL is required to unlock the device
  • The bootloader does these dummy reads and hence unlocks devices that do not have passwords programmed
CSM Caveats

- Never program all the PWL’s as 0x0000
  - Doing so will permanently lock the CSM
- Flash addresses 0x3F7F80 to 0x3F7FF5, inclusive, must be programmed to 0x0000 to securely lock the CSM
- Remember that code running in unsecured RAM cannot access data in secured memory
  - Don’t link the stack to secured RAM if you have any code that runs from unsecured RAM
- Do not embed the passwords in your code!
  - Generally, the CSM is unlocked only for debug
  - Code Composer Studio can do the unlocking

CSM Password Match Flow

1. Start
2. Flash device secure after reset or runtime
3. Do dummy reads of PWL 0x3F 7FF8 – 0x3F 7FFF
4. Is PWL all 0s?
   - Yes: Device permanently locked
   - No:
     1. Is PWL all Fs?
        - Yes: Write password to KEY registers 0x00 0AE0 – 0x00 0AE7 (EALLOW) protected
        - No: Correct password?
           - Yes: User can access on-chip secure memory
           - No: Device unlocked
Lab 10: Programming the Flash

Objective

The objective of this lab is to program and execute code from the on-chip flash memory. The TMS320F28035 device has been designed for standalone operation in an embedded system. Using the on-chip flash eliminates the need for external non-volatile memory or a host processor from which to bootload. In this lab, the steps required to properly configure the software for execution from internal flash memory will be covered.

Lab 10: Programming the Flash

Objective:
- Program system into Flash Memory
- Learn use of CCS Flash Plug-in
- **DO NOT PROGRAM PASSWORDS**

Procedure

Project File

1. A project named Lab10.pjt has been created for this lab. Open the project by clicking on Project → Open... and look in C:\C28x\Labs\Lab10. All Build Options have been configured the same as the previous lab. The files used in this lab are:

   - Adc.c
   - Cla_10_12.c
   - ClaTasks.asm
   - CodeStartBranch.asm
   - DefaultIsr_9_10.c
   - DelayUs.asm
   - DSP2803x_GlobalVariableDefs.c
   - DSP2803x_Headers_nonBIOS.cmd
   - ECap_7_8_9_10_12.c
   - EPwm_7_8_9_10_12.c
   - Filter.c
   - Flash.c
   - Gpio.c
   - Lab_10.cmd
   - Main_10.c
   - Passwords.asm
   - PieCtrl_5_6_7_8_9_10.c
   - PieVect_5_6_7_8_9_10.c
   - SysCtrl.c
   - Watchdog.c
Link Initialized Sections to Flash

Initialized sections, such as code and constants, must contain valid values at device power-up. Stand-alone operation of an F28035 embedded system means that no emulator is available to initialize the device RAM. Therefore, all initialized sections must be linked to the on-chip flash memory.

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

2. Open and inspect the linker command file Lab_10.cmd. Notice that a memory block named FLASH_ABCDEFGH has been created at origin = 0x3E8000, length = 0x00FF80 on Page 0. This flash memory block length has been selected to avoid conflicts with other required flash memory spaces. See the reference slide at the end of this lab exercise for further details showing the address origins and lengths of the various memory blocks used.

3. Edit Lab_10.cmd to link the following compiler sections to on-chip flash memory block FLASH_ABCDEFGH:

<table>
<thead>
<tr>
<th>Compiler Sections</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text</td>
</tr>
<tr>
<td>.cinit</td>
</tr>
<tr>
<td>.const</td>
</tr>
<tr>
<td>.econst</td>
</tr>
<tr>
<td>.pinit</td>
</tr>
<tr>
<td>.switch</td>
</tr>
</tbody>
</table>

4. In Lab_10.cmd notice that the section named “IQmath” is an initialized section that needs to load to and run from flash. Previously the “IQmath” section was linked to L0SARAM. Edit Lab_10.cmd so that this section is now linked to FLASH_ABCDEFGH. Save your work and close the file.

Copying Interrupt Vectors from Flash to RAM

The interrupt vectors must be located in on-chip flash memory and at power-up needs to be copied to the PIE RAM as part of the device initialization procedure. The code that performs this copy is located in InitPieCtrl(). The C-compiler runtime support library contains a memory copy function called memcpy() which will be used to perform the copy.
Lab 10: Programming the Flash

5. Open and inspect InitPieCtrl() in PieCtrl_5_6_7_8_9_10.c. Notice the memcpy() function used to initialize (copy) the PIE vectors. At the end of the file a structure is used to enable the PIE.

### Initializing the Flash Control Registers

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

6. Add Flash.c to the project.

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

8. The “secureRamFuncs” section will be linked using the user linker command file Lab_10.cmd. Open and inspect Lab_10.cmd. The “secureRamFuncs” will load to flash (load address) but will run from LOSARAM (run address). Also notice that the linker has been asked to generate symbols for the load start, load size, and run start addresses.

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

9. Open and inspect Main_10.c. Notice that the memory copy function memcpy() is being used to copy the section “secureRamFuncs”, which contains the initialization function for the flash control registers.

10. Add a line of code in main() to call the InitFlash() function. There are no passed parameters or return values. You just type

    
    InitFlash();

    at the desired spot in main().

### Code Security Module and Passwords

The CSM module provides protection against unwanted copying (i.e. pirating!) of your code from flash, OTP memory, and the L0, L1, L2 and L3 RAM blocks. The CSM uses a 128-bit password made up of 8 individual 16-bit words. They are located in flash at addresses 0x3F7FF8 to 0x3F7FFF. During this lab, dummy passwords of 0xFFFF will be used – therefore only dummy reads of the password locations are needed to unsecure the CSM. **DO NOT PROGRAM ANY REAL PASSWORDS INTO THE DEVICE.** After development, real passwords are typically
placed in the password locations to protect your code. We will not be using real passwords in the workshop.

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

11. Add Passwords.asm to the project.

12. Open and inspect Passwords.asm. This file specifies the desired password values **(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 0x3F7FF5 (length of 0x76).

13. Open Lab_10.cmd and notice that the initialized sections for “passwords” and “csm_rsvd” are linked to memories named PASSWORDS and CSM_RSVD, respectively.

**Executing from Flash after Reset**

The F28035 device contains a ROM bootloader that will transfer code execution to the flash after reset. When the boot mode selection is set for “Jump to Flash” mode, the bootloader will branch to the instruction located at address 0x3F7FF6 in the flash. An instruction that branches to the beginning of your program needs to be placed at this address. Note that the CSM passwords begin at address 0x3F7FF8. There are exactly two words available to hold this branch instruction, and not coincidentally, a long branch instruction “LB” in assembly code occupies exactly two words. Generally, the branch instruction will branch to the start of the C-environment initialization routine located in the C-compiler runtime support library. The entry symbol for this routine is _c_int00. Recall that C code cannot be executed until this setup routine is run. Therefore, assembly code must be used for the branch. We are using the assembly code file named CodeStartBranch.asm.

14. Open and inspect CodeStartBranch.asm. This file creates an initialized section named “codestart” that contains a long branch to the C-environment setup routine. This section needs to be linked to a block of memory named BEGIN_FLASH.

15. In the earlier lab exercises, the section “codestart” was directed to the memory named BEGIN_M0. Edit Lab_10.cmd so that the section “codestart” will be directed to BEGIN_FLASH. Save your work and close the opened files.

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

Previously, the named section “Cla1Prog” containing the CLA program tasks was linked directly to the CPU memory block L3DPSARAM for both load and run purposes. At runtime, all the code did was map the L3DPSARAM block to the CLA program memory space during CLA initialization. For an embedded application, the CLA program tasks are linked to load to flash and run from RAM. At runtime, the CLA program tasks must be copied from flash to L3DPSARAM. The memory copy function `memcpy()` will once again be used to perform the copy. After the copy is performed, the L3DPSARAM block will then be mapped to CLA program memory space as was done in the earlier lab.

16. Open and inspect `Lab_10.cmd`. Notice that the named section “Cla1Prog” will now load to flash (load address) but will run from L3DPSARAM (run address). The linker will also be used to generate symbols for the load start, load size, and run start addresses.

17. Open `Cla_10_12.c` and notice that the memory copy function `memcpy()` is being used to copy the CLA program code from flash to L3DPSARAM using the symbols generated by the linker. Just after the copy the `Cla1Regs` structure is used to configure the L3DPSARAM block as CLA program memory space. Close the inspected files.

Build – Lab.out

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

CCS has a feature that automatically steps over functions without debug information. This can be useful for accelerating the debug process provided that you are not interested in debugging the function that is being stepped-over. While single-stepping in this lab exercise we do not want to step-over any functions. Therefore, select the “Debug Properties” tab. Uncheck “Step over functions without debug information when source stepping”, then click OK.

19. Click the “Build” button to generate the `Lab.out` file to be used with the CCS Flash Plug-in.

CCS Flash Plug-in

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

   Tools → F28xx On-Chip Flash Programmer

21. A Clock Configuration window may open. If needed, in the Clock Configuration window set “OSCCLK (MHz):” to 10, “DIVSEL:” to /2, and “PLLCR Value:” to 12. Then click OK. In the next Flash Programmer Settings window confirm that the selected DSP device to program is F28035 and all options have been checked. Click OK.
22. The CCS Flash Programmer uses the Piccolo™ 10 MHz internal oscillator as the device clock during programming. Confirm the “Clock Configuration” in the upper left corner has the OSCCLK set to 10 MHz, the DIVSEL set to /2, and the PLLCR value set to 12. Recall that the PLL is divided by two, which gives a SYSCLKOUT of 60 MHz.

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

24. We will not be using the plug-in to program the “Code Security Password”. **Do not modify the Code Security Password fields.** They should remain as all 0xFFFF.

25. In the “Operation” block, notice that the “COFF file to Program/Verify” field automatically defaults to the current .out file. Check to be sure that “Erase, Program, Verify” is selected. We will be using the default wait states, as shown on the slide in this module. The selection for wait-states only affects the verify step, and makes little noticeable difference even if you reduce the wait-states.

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

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

**Running the Code – Using CCS**

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

   File → Load Symbols → Load Symbols Only...

   and select Lab10.out in the Debug folder.

29. Reset the CPU. The program counter should now be at 0x3FF8A1, which is the start of the bootloader in the Boot ROM.

30. Under GEL on the menu bar click:

   EMU Boot Mode Select → EMU_BOOT_FLASH.

   This has the debugger load values into EMU_KEY and EMU_BMODE so that the bootloader will jump to "FLASH" at 0x3F7FF6.

31. Single-Step <F11> through the bootloader code until you arrive at the beginning of the codestart section in the CodeStartBranch.asm file. (Be patient, it will take about 125 single-steps). Notice that we have placed some code in CodeStartBranch.asm to give an option to first disable the watchdog, if selected.

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

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

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

35. HALT the CPU.

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


37. Disconnect the USB cable (emulator) from the Docking Station (i.e. remove power from the ControlCARD).

38. Re-connect the USB cable to the Docking Station to power the ControlCARD. The LED should be blinking, showing that the code is now running from flash memory.

   **End of Exercise**
Lab 10 Reference: Programming the Flash

Flash Memory Section Blocks

<table>
<thead>
<tr>
<th>Origin</th>
<th>Flash Memory Section</th>
<th>Length</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3F 7F80</td>
<td>CSM_RSVD</td>
<td>0x76</td>
<td>0</td>
</tr>
<tr>
<td>0x3F 7FF6</td>
<td>BEGIN_FLASH</td>
<td>0x2</td>
<td>0</td>
</tr>
<tr>
<td>0x3F 7FF8</td>
<td>PASSWORDS</td>
<td>0x8</td>
<td>0</td>
</tr>
</tbody>
</table>

Lab_10.cmd

SECTIONS

{ 
  codestart :> BEGIN_FLASH, PAGE = 0
  passwords :> PASSWORDS, PAGE = 0
  csm_rsvd :> CSM_RSVD, PAGE = 0
}

Startup Sequence from Flash Memory

1. RESET
2. BROM vector (32w)
3. Boot ROM (8Kw)
4. Passwords (8w)
5. "rts2800_ml.lib"
6. "user" code sections
   main ()
   { ....... ....... }

C2000 Piccolo Workshop - System Design
Communications

Introduction

The TMS320C28x contains features that allow several methods of communication and data exchange between the C28x and other devices. Many of the most commonly used communications techniques are presented in this module.

The intent of this module is not to give exhaustive design details of the communication peripherals, but rather to provide an overview of the features and capabilities. Once these features and capabilities are understood, additional information can be obtained from various resources such as documentation, as needed. This module will cover the basic operation of the communication peripherals, as well as some basic terms and how they work.

Learning Objectives

- Serial Peripheral Interface (SPI)
- Serial Communication Interface (SCI)
- Local Interconnect Network (LIN)
- Inter-Integrated Circuit (I2C)
- Enhanced Controller Area Network (eCAN)

Note: Up to 2 SPI modules (A/B), 1 SCI module (A), 1 LIN module (A), 1 I2C module (A), and 1 eCAN module (A) are available on the F283x devices.
# Module Topics

Communications................................................................................................................. 11-1

Module Topics.................................................................................................................. 11-2

Communications Techniques ................................................................................................. 11-3

Serial Peripheral Interface (SPI) .............................................................................................. 11-4
  SPI Registers .................................................................................................................. 11-7
  SPI Summary ................................................................................................................ 11-8

Serial Communications Interface (SCI) .................................................................................... 11-9
  Multiprocessor Wake-Up Modes ..................................................................................... 11-11
  SCI Registers ................................................................................................................. 11-14
  SCI Summary ................................................................................................................ 11-15

Local Interconnect Network (LIN) ........................................................................................ 11-16
  LIN Message Frame and Data Timing ............................................................................. 11-17
  LIN Summary .............................................................................................................. 11-18

Inter-Integrated Circuit (I2C) .................................................................................................. 11-19
  I2C Operating Modes and Data Formats ........................................................................ 11-20
  I2C Summary ................................................................................................................ 11-21

Enhanced Controller Area Network (eCAN) ................................................................. 11-22
  CAN Bus and Node ..................................................................................................... 11-23
  Principles of Operation ............................................................................................... 11-24
  Message Format and Block Diagram ........................................................................... 11-25
  eCAN Summary .......................................................................................................... 11-26
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

<table>
<thead>
<tr>
<th>Synchronous</th>
<th>Asynchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Short distances (on-board)</td>
<td>- longer distances</td>
</tr>
<tr>
<td>- High data rate</td>
<td>- Lower data rate (≈ 1/8 of SPI)</td>
</tr>
<tr>
<td>- Explicit clock</td>
<td>- Implied clock (clk/data mixed)</td>
</tr>
<tr>
<td></td>
<td>- Economical with reasonable performance</td>
</tr>
</tbody>
</table>

Serial ports provide a simple, hardware-efficient means of high-level communication between devices. Like the GPIO pins, they may be used in stand-alone or multiprocessing systems.

In a multiprocessing system, they are an excellent choice when both devices have an available serial port and the data rate requirement is relatively low. Serial interface is even more desirable when the devices are physically distant from each other because the inherently low number of wires provides a simpler interconnection.

Serial ports require separate lines to implement, and they do not interfere in any way with the data and address lines of the processor. The only overhead they require is to read/write new words from/to the ports as each word is received/transmitted. This process can be performed as a short interrupt service routine under hardware control, requiring only a few cycles to maintain.

The C28x family of devices have both synchronous and asynchronous serial ports. Detailed features and operation will be described next.
Serial Peripheral Interface (SPI)

The SPI module is a synchronous serial I/O port that shifts a serial bit stream of variable length and data rate between the C28x and other peripheral devices. During data transfers, one SPI device must be configured as the transfer MASTER, and all other devices configured as SLAVES. The master drives the transfer clock signal for all SLAVES on the bus. SPI communications can be implemented in any of three different modes:

- MASTER sends data, SLAVES send dummy data
- MASTER sends data, one SLAVE sends data
- MASTER sends dummy data, one SLAVE sends data

In its simplest form, the SPI can be thought of as a programmable shift register. Data is shifted in and out of the SPI through the SPIDAT register. Data to be transmitted is written directly to the SPIDAT register, and received data is latched into the SPIBUF register for reading by the CPU. This allows for double-buffered receive operation, in that the CPU need not read the current received data from SPIBUF before a new receive operation can be started. However, the CPU must read SPIBUF before the new operation is complete 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
Serial Peripheral Interface (SPI)

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 being 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 Data Character Justification Diagram]

**SPIDAT - Processor #1**

```
11001001XXXXXXXX
```

**SPIDAT - Processor #2**

```
XXXXXXXXXXX11001001
```
SPI Registers

### SPI Baud Rate Register

*SpixRegs.SPIBRR*

**Need to set this only when in master mode!**

<table>
<thead>
<tr>
<th>15-7</th>
<th>6-0</th>
</tr>
</thead>
<tbody>
<tr>
<td>reserved</td>
<td>SPI BIT RATE</td>
</tr>
</tbody>
</table>

SPICLK signal =

\[
\begin{align*}
\text{SPIBRR} = 3 \text{ to } 127 & : \quad \frac{\text{LSPCLK}}{(\text{SPIBRR} + 1)} \\
\text{SPIBRR} = 0, 1, \text{ or } 2 & : \quad \frac{\text{LSPCLK}}{4}
\end{align*}
\]

Baud Rate Determination: The Master specifies the communication baud rate using its baud rate register (SPIBRR.6-0):

- For SPIBRR = 3 to 127: \( \text{SPI Baud Rate} = \frac{\text{LSPCLK}}{(\text{SPIBRR} + 1)} \) bits/sec
- For SPIBRR = 0, 1, or 2: \( \text{SPI Baud Rate} = \frac{\text{LSPCLK}}{4} \) bits/sec

From the above equations, one can compute

Maximum data rate = 25 Mbps @ 100 MHz

Character Length Determination: The Master and Slave must be configured for the same transmission character length. This is done with bits 0, 1, 2 and 3 of the configuration control register (SPICCR.3-0). These four bits produce a binary number, from which the character length is computed as binary + 1 (e.g. SPICCR.3-0 = 0010 gives a character length of 3).
Select SPI Registers

- **Configuration Control** \( \text{Regs.SPICCR} \)
  - Reset, Clock Polarity, Loopback, Character Length
- **Operation Control** \( \text{Regs.SPICTL} \)
  - Overrun Interrupt Enable, Clock Phase, Interrupt Enable
  - Master / Slave Transmit enable
- **Status** \( \text{Regs.SPIST} \)
  - RX Overrun Flag, Interrupt Flag, TX Buffer Full Flag
- **FIFO Transmit** \( \text{Regs.SPIFFTX} \)
- **FIFO Receive** \( \text{Regs.SPIFFRX} \)
  - FIFO Enable, FIFO Reset
  - FIFO Over-flow flag, Over-flow Clear
  - Number of Words in FIFO (FIFO Status)
  - FIFO Interrupt Enable, Interrupt Status, Interrupt Clear
  - FIFO Interrupt Level (Number of Words in FIFO)

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

SPI Summary

- **Synchronous serial communications**
  - Two wire transmit or receive (half duplex)
  - Three wire transmit and receive (full duplex)
- **Software configurable as master or slave**
  - C28x provides clock signal in master mode
- **Data length programmable from 1-16 bits**
- **125 different programmable baud rates**
Serial Communications Interface (SCI)

The SCI module is a serial I/O port that permits Asynchronous communication between the C28x and other peripheral devices. The SCI transmit and receive registers are both double-buffered to prevent data collisions and allow for efficient CPU usage. In addition, the C28x SCI is a full duplex interface which provides for simultaneous data transmit and receive. Parity checking and data formatting is also designed to be done by the port hardware, further reducing software overhead.

![SCI Pin Connections Diagram](image-url)
Serial Communications Interface (SCI)

SCI 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</th>
<th>Data</th>
<th>Parity</th>
<th>Stop 1</th>
<th>Stop 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>This bit present only in Address-bit mode</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></td>
</tr>
</tbody>
</table>

Communications Control Register (SciRegs.SCICCR)

<table>
<thead>
<tr>
<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>Stop Bits</td>
<td>Even/Odd Parity</td>
<td>Parity Enable</td>
<td>Loopback Enable</td>
<td>Addr/Idle Mode</td>
<td>SCI Char2</td>
<td>SCI Char1</td>
<td>SCI Char0</td>
</tr>
<tr>
<td>0 = 1 Stop bit</td>
<td>0 = Disabled</td>
<td>0 = Idle-line mode</td>
<td># of data bits = (binary + 1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 = 2 Stop bits</td>
<td>1 = Enabled</td>
<td>1 = Addr-bit mode</td>
<td>e.g. 110b gives 7 data bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 = Odd</td>
<td>0 = Disabled</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 = Even</td>
<td>1 = Enabled</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.
Serial Communications Interface (SCI)

**SCI Data Timing**

- Start bit valid if 4 consecutive SCICLK periods of zero bits after falling edge
- Majority vote taken on 4th, 5th, and 6th SCICLK cycles

![SCICLK and SCIRXD timing diagram]

- Note: 8 SCICLK periods per data bit

**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

**Address-Bit Wake-Up Mode**

- All frames contain an extra address bit
- Receiver wakes up when address bit detected
- Automatic setting of Addr/Data bit in frame by setting TXWAKE = 1 prior to writing address to SCITXBUF
The SCI interrupt logic generates interrupt flags when it receives or transmits a complete character as determined by the SCI character length. This provides a convenient and efficient way of timing and controlling the operation of the SCI transmitter and receiver. The interrupt flag for the transmitter is TXRDY (SCICTL2.7), and for the receiver RXRDY (SCIRXST.6). TXRDY is set when a character is transferred to TXSHF and SCITXBUF is ready to receive the next character. In addition, when both the SCIBUF and TXSHF registers are empty, the TX EMPTY flag (SCICTL2.6) is set. When a new character has been received and shifted into SCIRXBUF, the RXRDY flag is set. In addition, the BRKDT flag is set if a break condition occurs. A break condition is where the SCIRXD line remains continuously low for at least ten bits, beginning after a missing stop bit. Each of the above flags can be polled by the CPU to control SCI operations, or interrupts associated with the flags can be enabled by setting the RX/BK INT ENA (SCICTL2.1) and/or the TX INT ENA (SCICTL2.0) bits active high.

Additional flag and interrupt capability exists for other receiver errors. The RX ERROR flag is the logical OR of the break detect (BRKDT), framing error (FE), receiver overrun (OE), and parity error (PE) bits. RX ERROR high indicates that at least one of these four errors has occurred during transmission. This will also send an interrupt request to the CPU if the RX ERR INT ENA (SCICTL1.6) bit is set.
Serial Communications Interface (SCI)

**SCI Registers**

**SCI Baud Rate Registers**

\[
\text{SCI baud rate} = \begin{cases} 
\frac{\text{LSPCLK}}{\text{(BRR + 1) \times 8}}, & \text{BRR = 1 to 65535} \\
\frac{\text{LSPCLK}}{16}, & \text{BRR = 0}
\end{cases}
\]

<table>
<thead>
<tr>
<th>Baud-Select MSbyte Register (SciRegs.SCIHBAUD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAUD15 (MSB) BAUD14 BAUD13 BAUD12 BAUD11 BAUD10 BAUD9 BAUD8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Baud-Select LSbyte Register (SciRegs.SCILBAUD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAUD7 BAUD6 BAUD5 BAUD4 BAUD3 BAUD2 BAUD1 BAUD0 (LSB)</td>
</tr>
</tbody>
</table>

**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: \( \text{SCI Baud Rate} = \frac{\text{LSPCLK}}{(\text{BRR} + 1) \times 8} \) bits/sec
- For BRR = 0: \( \text{SCI Baud Rate} = \frac{\text{LSPCLK}}{16} \) bits/sec

Max data rate = 6.25 Mbps @ 100 MHz

Note that the CLKOUT for the SCI module is one-half the CPU clock rate.
Select SCI Registers

- **Control 1** `SciRegs.SCICTL1`
  - Reset, Transmitter / Receiver Enable
  - TX Wake-up, Sleep, RX Error Interrupt Enable

- **Control 2** `SciRegs.SPICTL2`
  - TX Buffer Full / Empty Flag, TX Ready Interrupt Enable
  - RX Break Interrupt Enable

- **Receiver Status** `SciRegs.SCIRXST`
  - Error Flag, Ready Flag Break-Detect Flag, Framing Error Detect Flag, Parity Error Flag, RX Wake-up Detect Flag

- **FIFO Transmit** `SciRegs.SCIFFTX`

- **FIFO Receive** `SciRegs.SCIFFRX`
  - FIFO Enable, FIFO Reset
  - FIFO Over-flow flag, Over-flow Clear
  - Number of Words in FIFO (FIFO Status)
  - FIFO Interrupt Enable, Interrupt Status, Interrupt Clear
  - FIFO Interrupt Level (Number of Words in FIFO)

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

SCI Summary

- **Asynchronous communications format**
- **65,000+ different programmable baud rates**
- **Two wake-up multiprocessor modes**
  - Idle-line wake-up & Address-bit wake-up
- **Programmable data word format**
  - 1 to 8 bit data word length
  - 1 or 2 stop bits
  - even/odd/no parity
- **Error Detection Flags**
  - Parity error, Framing error, Overrun error, Break detection
- **Transmit FIFO and receive FIFO**
- **Individual interrupts for transmit and receive**
Local Interconnect Network (LIN)

- Compliant to the LIN2.0 protocol Specification Package
- Module based on SCI (core) with added hardware features for LIN compatibility:
  - Error detector
  - Mask filter
  - Synchronizer
  - Multi-buffered receiver/transmitter
- Standard is based on SCI (UART) serial data link format
- Communication concept is single-master/multiple-slave with message identification for multi-cast transmission between any network nodes
- Module can be used in LIN mode or SCI (UART) mode

LIN Block Diagram
LIN Message Frame and Data Timing

**LIN Message Frame**

- **Sync Break** – beginning of a message
- **Sync Field** – bit rate information
- **ID Field** – content of a message
- **Data Field** – consists of 1 data byte, 1 start bit, and 1 stop bit (10 bits total)
- **Checksum Field** – consists of 1 checksum byte, 1 start bit and 1 stop bit (10 bits total)
- **In-Frame & Interbyte Spaces** – can be 0

**LIN Data Timing**

To make a determination of the bit value, 16 samples of each bit are taken with majority vote on samples 8, 9, and 10

- **LIN module** is clocked at ½ the CPU clock (SYSCLKOUT)
LIN Summary

- Functionally compatible with standalone SCI of C28x devices
- Identification masks for filtering
- Automatic master header generation
- $2^{28}$ programmable transmission rates
- Automatic wakeup support
- Error detection (bit, bus, no response, checksum, synchronization, parity)
- Multi-buffered receive/transmit units
Inter-Integrated Circuit (I2C)

- Philips I2C-bus specification compliant, version 2.1
- Data transfer rate from 10 kbps up to 400 kbps
- Each device can be considered as a Master or Slave
- Master initiates data transfer and generates clock signal
- Device addressed by Master is considered a Slave
- Multi-Master mode supported
- Standard Mode – send exactly n data values (specified in register)
- Repeat Mode – keep sending data values (use software to initiate a stop or new start condition)

I2C Block Diagram
I2C Operating Modes and Data Formats

I2C Operating Modes

<table>
<thead>
<tr>
<th>Operating Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slave-receiver mode</td>
<td>Module is a slave and receives data from a master (all slaves begin in this mode)</td>
</tr>
<tr>
<td>Slave-transmitter mode</td>
<td>Module is a slave and transmits data to a master (can only be entered from slave-receiver mode)</td>
</tr>
<tr>
<td>Master-receiver mode</td>
<td>Module is a master and receives data from a slave (can only be entered from master-transmit mode)</td>
</tr>
<tr>
<td>Master-transmitter mode</td>
<td>Module is a master and transmits to a slave (all masters begin in this mode)</td>
</tr>
</tbody>
</table>

I2C Serial Data Formats

7-Bit Addressing Format

```
1 1 1 1 1 1 1 1
S Slave Address R/W ACK Data ACK Data ACK P
```

10-Bit Addressing Format

```
1 1 1 1 1 1 1 1
S 11110AA R/W ACK AAAAAAAA ACK Data ACK P
```

Free Data Format

```
1 1 1 1 1 1 1 1
S Data ACK Data ACK Data ACK P
```

R/W = 0 – master writes data to addressed slave  
R/W = 1 – master reads data from the slave  
n = 1 to 8 bits  
S = Start (high-to-low transition on SDA while SCL is high)  
P = Stop (low-to-high transition on SDA while SCL is high)
I2C Arbitration

- Arbitration procedure invoked if two or more master-transmitters simultaneously start transmission
  - Procedure uses data presented on serial data bus (SDA) by competing transmitters
  - First master-transmitter which drives SDA high is overruled by another master-transmitter that drives SDA low
  - Procedure gives priority to the data stream with the lowest binary value

![Arbitration Diagram]

Device #1 lost arbitration and switches to slave-receiver mode

Device #2 drives SDA

I2C Summary

- Compliance with Philips I2C-bus specification (version 2.1)
- 7-bit and 10-bit addressing modes
- Configurable 1 to 8 bit data words
- Data transfer rate from 10 kbps up to 400 kbps
- Transmit FIFO and receive FIFO
Enhanced Controller Area Network (eCAN)

Controller Area Network (CAN)
A Multi-Master Serial Bus System

- CAN 2.0B Standard
- High speed (up to 1 Mbps)
- Add a node without disturbing the bus (number of nodes not limited by protocol)
- Less wires (lower cost, less maintenance, and more reliable)
- Redundant error checking (high reliability)
- No node addressing (message identifiers)
- Broadcast based signaling

CAN does not use physical addresses to address stations. Each message is sent with an identifier that is recognized by the different nodes. The identifier has two functions – it is used for message filtering and for message priority. The identifier determines if a transmitted message will be received by CAN modules and determines the priority of the message when two or more nodes want to transmit at the same time.
CAN Bus and Node

**CAN Bus**

- Two wire differential bus (usually twisted pair)
- Max. bus length depend on transmission rate
  - 40 meters @ 1 Mbps

The MCU communicates to the CAN Bus using a transceiver. The CAN bus is a twisted pair wire and the transmission rate depends on the bus length. If the bus is less than 40 meters the transmission rate is capable up to 1 Mbit/second.
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 showing arbitration process]

- Node A wins arbitration
- Node B loses arbitration
- Node C loses arbitration
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)**

<table>
<thead>
<tr>
<th>Arbitration Field</th>
<th>Control Field</th>
<th>Data Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>11-bit Identifier</td>
<td>R</td>
</tr>
</tbody>
</table>

**Extended Frame: 29-bit Identifier (CAN v2.0B)**

<table>
<thead>
<tr>
<th>Arbitration Field</th>
<th>Control Field</th>
<th>Data Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>11-bit Identifier</td>
<td>S</td>
</tr>
</tbody>
</table>

The MCU CAN module is a full CAN Controller. It contains a message handler for transmission and reception management, and frame storage. The specification is CAN 2.0B Active – that is, the module can send and accept standard (11-bit identifier) and extended frames (29-bit identifier).

**eCAN Block Diagram**
The CAN controller module contains 32 mailboxes for objects of 0 to 8-byte data lengths:
- configurable transmit/receive mailboxes
- configurable with standard or extended 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**

<table>
<thead>
<tr>
<th>eCAN Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>✷ Fully compliant with CAN standard v2.0B</td>
</tr>
<tr>
<td>✷ Supports data rates up to 1 Mbps</td>
</tr>
<tr>
<td>✷ Thirty-two mailboxes</td>
</tr>
<tr>
<td>• Configurable as receive or transmit</td>
</tr>
<tr>
<td>• Configurable with standard or extended identifier</td>
</tr>
<tr>
<td>• Programmable receive mask</td>
</tr>
<tr>
<td>• Uses 32-bit time stamp on messages</td>
</tr>
<tr>
<td>• Programmable interrupt scheme (two levels)</td>
</tr>
<tr>
<td>• Programmable alarm time-out</td>
</tr>
<tr>
<td>✷ Programmable wake-up on bus activity</td>
</tr>
<tr>
<td>✷ Self-test mode</td>
</tr>
</tbody>
</table>
Introduction

This module discusses the basic features of using DSP/BIOS in a system. Scheduling threads, periodic functions, and the use of real-time analysis tools will be demonstrated, in addition to programming the flash with DSP/BIOS.

Learning Objectives

- Introduction to DSP/BIOS
- DSP/BIOS Configuration Tool
- Scheduling DSP/BIOS threads
- Periodic Functions
- Real-Time Analysis Tools
Module Topics

DSP/BIOS ..................................................................................................................................................12-1

   Module Topics........................................................................................................................................12-2

   Introduction to DSP/BIOS .................................................................12-3

   DSP/BIOS Configuration Tool.............................................................12-4

   Scheduling DSP/BIOS threads...................................................................12-9

   Periodic Functions......................................................................................12-14

   Real-Time Analysis Tools.............................................................................12-15

   Lab 12: DSP/BIOS.......................................................................................12-16
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
  - no need to develop or maintain a “home-brew” kernel
  - faster time to market
- Efficient debugging of real-time applications
  - Real-Time Analysis
- Create robust applications
  - industry proven kernel technology
- Reduce cost of software maintenance
  - code reuse and standardized software
- Integrated with Code Composer Studio IDE
  - requires no runtime license fees
  - fully supported by TI
- Uses minimal Mips and Memory (2-8Kw)
  - scalable – use only what is needed
  - easily fits in limited memory space
DSP/BIOS Configuration Tool

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

**DSP/BIOS Configuration Tool (file .tcf)**

- **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 configures DSP/BIOS scheduling

- **Real-Time I/O**
  - Allows two way communication between threads or between target and PC host

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

- Automatically including the appropriate runtime support libraries
- Automatically handles interrupt vectors and system reset
- Handles system memory configuration (builds .cmd file)
- When a .tcf file is saved, the Config Tool generates 5 additional files:
  
<table>
<thead>
<tr>
<th>Filename</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename.tcf</td>
<td>Text Configuration File</td>
</tr>
<tr>
<td>filenamecfg_c.c</td>
<td>C code created by Config Tool</td>
</tr>
<tr>
<td>filenamecfg.s28</td>
<td>ASM code created by Config Tool</td>
</tr>
<tr>
<td>filenamecfg.cmd</td>
<td>Linker command file</td>
</tr>
<tr>
<td>filenamecfg.h</td>
<td>header file for *cfg_c.c</td>
</tr>
<tr>
<td>filenamecfg.h28</td>
<td>header file for *cfg.s28</td>
</tr>
</tbody>
</table>

When you add a .tcf file to your project, CCS automatically adds the C and assembly (.s28) files and the linker command file (.cmd) to the project under the Generated Files folder.
1. Creating a New Memory Region (Using MEM)

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

Memory Section Manager (MEM)

- Generates the main linker command file for your code project
  - Create memories
  - Place sections
- To create a new memory area:
  - Right-click on MEM and select insert memory
  - Enter your choice of a name for the memory
  - Right-click on the memory, and select Properties
    - fill in base, length, space
2. Placing Sections – MEM Manager Properties

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

Memory Section Manager Properties

- To place a section into a memory area:
  - Right-click on MEM and select Properties
  - Select the desired tab (e.g. Compiler)
  - Select the memory you would like to link each section to
3. PIE Interrupts – HWI Interrupts

The configuration tools is also used to assign the interrupt vectors. The vectors are placed into a section named .hwi_vec. The memory manager (MEM) links this section to the proper location in memory.
4. Running the Linker

Creating the Linker Command File (via .tcf)

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

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

Running the Linker

The linker’s main purpose is to link together various object files. It combines like-named input sections from the various object files and places each new output section at specific locations in memory. In the process, it resolves (provides actual addresses for) all of the symbols described in your code. The linker can create two outputs, the executable (.out) file and a report which describes the results of linking (.map).

Note: The linker gets run automatically when you BUILD or REBUILD your project.
Scheduling DSP/BIOS threads

DSP/BIOS Thread Types

<table>
<thead>
<tr>
<th>Priority</th>
<th>Thread Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HWI</td>
<td>Used to implement 'urgent' part of real-time event</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Triggered by hardware interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HWI priorities fixed in hardware</td>
</tr>
<tr>
<td></td>
<td>SWI</td>
<td>Use SWI to perform HWI ‘follow-up’ activity</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SWI’s are ‘posted’ by software</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Multiple SWIs at each of 15 priority levels</td>
</tr>
<tr>
<td></td>
<td>TSK</td>
<td>Use TSK to run different programs concurrently under separate contexts</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TSK’s enabled by posting ‘semaphore’ (a signal)</td>
</tr>
<tr>
<td></td>
<td>IDL</td>
<td>Runs when no service routines are pending</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Runs as an infinite loop, like traditional while loop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All BIOS data transfers to host occur here</td>
</tr>
</tbody>
</table>

Enabling DSP/BIOS in main()

```c
void main(void)
{
    //*** Initialization
    ...
    //*** Enable global interrupts
    // asm(" CLRC INTM");
    //*** Main Loop
    //   while(1);
}
```

- BIOS will enable global interrupts for you
- Must delete the endless loop at end of main()
- main() returns to BIOS and goes to the IDLE thread, allowing BIOS to schedule events, transfer data to the host, etc.
- An endless loop in main() will keep BIOS from running
Scheduling DSP/BIOS threads

Using Hardware Interrupts - HWI

- Interrupt priority fixed by hardware

The HWI Dispatcher

- For non-BIOS code, use the `interrupt` keyword to declare an ISR
  - tells the compiler to perform context save/restore

```c
interrupt void MyHwi(void)
{
}
```

- For DSP/BIOS code, use the `Dispatcher` to perform the save/restore
  - Remove the interrupt keyword from the `MyHwi()`
  - Check the “Use Dispatcher” box when you configure the interrupt vector in the DSP/BIOS configuration tool
  - This is necessary if you want to use any DSP/BIOS functionality inside the ISR
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

SWI Properties
Scheduling DSP/BIOS threads

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

User sets the priority...BIOS does the scheduling
Using Tasks (TSK)

SWI vs. TSK

- Similar to hardware interrupt, but triggered by SWI_post()
- SWIs must run to completion
- All SWI's use system stack
- faster context switching
- smaller code size

SWI

start
“must run to completion”
end

SEM_post() readies the TSK which pends on an event
- TSKs can be terminated by SW
- Each TSK has its own stack
- slower context switching
- larger code size

Pause
(blocked state)

TSK

start
end

SEM_pend

SEM_post
Periodic Functions

**Using Periodic Functions - PRD**

- Periodic functions are a special type of SWI that are triggered by DSP/BIOS.
- Periodic functions run at a user specified rate:
  - e.g. LED blink requires 0.5 Hz
- Use the CLK Manager to specify the DSP/BIOS CLK rate in microseconds per "tick"
- Use the PRD Manager to specify the period (for the function) in ticks
- Allows multiple periodic functions with different rates

**Creating a Periodic Function**
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

**Built-in Real-Time Analysis Tools**

**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, “LedSwiCount = %u”, LedSwiCount++);
```
Lab 12: DSP/BIOS

Objective

The objective of this lab is to become familiar with DSP/BIOS. In this lab exercise, we will make use of the DSP/BIOS configuration tool, implement a software interrupt (SWI) and periodic function (PRD), program the DSP/BIOS project into the flash, and explore the built-in real-time analysis tools. The DSP/BIOS configuration tool creates a text configuration file (*.tcf) and generates a linker command file (*.cfg.cmd). This generated linker command file is functionally equivalent to the linker command file previously used. The memory area of the lab linker command file will be deleted; however, part of the sections area will be used to link sections that are not part of DSP/BIOS. In the lab files we will change the CLA HWI (CLA1_INT1_ISR) to a SWI and replace the LED blink routine with a periodic function. The steps required to properly configure the software for execution from internal flash memory will be covered. Features of the real-time analysis tools, such as the CPU Load Graph, Execution Graph, Message Log, and RTA Control Panel will be demonstrated.

Procedure

Project File

1. A project named Lab12.pjt has been created for this lab. Open the project by clicking on Project → Open... and look in C:\C28x\Labs\Lab12. All Build Options have been configured the same as the previous lab. The files used in this lab are:
Edit Lab.h File

2. Edit Lab.h to *uncomment* the line that includes the labcfg.h header file. This is the DSP/BIOS generated include file, and is needed to allow code to access the DSP/BIOS functions and data structures. Next, *comment out* the line that includes the “DSP2803x_DefaultIsr.h” ISR function prototypes. DSP/BIOS will supply its own ISR function prototypes.

3. In our lab setup, we are running the ADC at a 50 kHz interrupt rate. Such a high frequency interrupt would typically be handled directly in the HWI, as SWIs and TSKs have some overhead associated with them and launching them this frequently can cause very large processing loads on the CPU. DSP/BIOS is flexible in this way. You can have some interrupts processed directly in the HWI, and others delegated to SWIs or TSKs. For purposes of this lab however, we would like to illustrate how to code a SWI. Therefore, we will convert the ADC ISR into a SWI. To reduce the CPU load, we are going to reduce the frequency of the ADC sample rate by half to 25 kHz.

In Lab.h modify the constant definition for the ADC sample rate as follows:

```c
#define ADC_SAMPLE_PERIOD 2399 // 25 KHz sampling
```

Save and close the file.

Remove “rts2800_ml.lib” and Inspect Lab_12.cmd

4. The DSP/BIOS configuration tool supplies its own RTS library. Open the Build Options and select the Linker tab. In the Libraries Category, find the Include Libraries (-l) box and delete: rts2800_ml.lib.

5. Select the Compiler tab. As the project is now configured, we would get a warning at build time stating that the typedef name has already been declared with the same type. This is because it has been defined twice; once in the header files and again in the include file generated by DSP/BIOS. To suppress the warning select Diagnostics Category and find the Suppress Diagnostic <n> (-pds): box. Type in code number 303. Select OK and the Build Options window will close.

6. We will be using the DSP/BIOS configuration tool to create a linker command file. Open and inspect Lab_12.cmd. Notice that the linker command file does not have a memory
area and includes only a limited sections area. These sections are not part of DSP/BIOS and need to be included in a “user” linker command file. Close the inspected file.

Using the DSP/BIOS Configuration Tool

7. The text configuration file (*.tcf) created by the DSP/BIOS configuration tool controls a wide range of CCS capabilities. The .tcf file will be used to automatically create and perform memory management. Create a new .tcf file for this lab. On the menu bar click:

File ➔ New ➔ DSP/BIOS Configuration…

A dialog box appears showing a number of available .tcf seed files. The seed files are used to configure many objects specific to the processor and will be invoked as the first item in your own .tcf file. On the C2xxx tab select the `ti.platforms.control28035` template and click OK. A configuration window will open.

8. Save the configuration file by selecting:

File ➔ Save As…

and name it `Lab.tcf` in `C:\C28x\Labs\Lab12` then click Save. Close the configuration window and select YES to save changes to `Lab.tcf`.

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

Project ➔ Add Files to Project…

Make sure you’re looking in `C:\C28x\Labs\Lab12`. Change the “files of type” to view All Files (*.*) and select `Lab.tcf`. Click OPEN to add the file to the project.

10. In the project window left click the plus sign (+) to the left of DSP/BIOS Config. Notice that the `Lab.tcf` file is listed.

11. Next, add the generated linker command file `Labcfg.cmd` to the project. After the file has been added you will notice that it is listed under the source files.

Create New Memory Sections Using the TCF File

12. Open the `Lab.tcf` file by double clicking on `Lab.tcf`. In the configuration window, left click the plus sign next to System and the plus sign next to MEM. By default, the Memory Section Manager has combined the memory space L1, L2 and L3DPSARAM into a single memory block called DPSARAM. It has also combined M0 and M1SARAM into a single memory block called MSARAM.

13. Next, we will add some of the additional memory sections that will be needed for the lab exercises in this module. To add a memory section:

Right click on MEM - Memory Section Manager and select Insert MEM. Rename the newly added memory section to BEGIN_FLASH. Repeat the process and add the following memory sections: CLAMSGRAM1, CLAMSGRAM2, CSM_RSVD, IQTABLES, L3DPSARAM, and PASSWORDS. Double check and see that all seven memory sections have been added.
14. Modify the base addresses, length, and space of each of the memory sections to correspond to the memory mapping shown in the table below. To modify the length, base address, and space of a memory section, right click on the memory in the configuration tool, and select Properties.

<table>
<thead>
<tr>
<th>Memory</th>
<th>Base</th>
<th>Length</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN_FLASH</td>
<td>0x3F 7FF6</td>
<td>0x0002</td>
<td>code</td>
</tr>
<tr>
<td>CLAMSGRAM1</td>
<td>0x00 1480</td>
<td>0x0080</td>
<td>data</td>
</tr>
<tr>
<td>CLAMSGRAM2</td>
<td>0x00 1500</td>
<td>0x0080</td>
<td>data</td>
</tr>
<tr>
<td>CSM_RSVD</td>
<td>0x3F 7F80</td>
<td>0x0076</td>
<td>code</td>
</tr>
<tr>
<td>IQTABLES</td>
<td>0x3F E000</td>
<td>0x0B50</td>
<td>code</td>
</tr>
<tr>
<td>L3DPSARAM</td>
<td>0x00 9000</td>
<td>0x1000</td>
<td>code</td>
</tr>
<tr>
<td>PASSWORDS</td>
<td>0x3F 7FF8</td>
<td>0x0008</td>
<td>code</td>
</tr>
</tbody>
</table>

15. Modify the base addresses, length, and space of each of the memory sections to avoid memory conflicts with the newly added memory sections as shown in the table below.

<table>
<thead>
<tr>
<th>Memory</th>
<th>Base</th>
<th>Length</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOTROM</td>
<td>0x3F F27C</td>
<td>0x0D44</td>
<td>code</td>
</tr>
<tr>
<td>DPSARAM</td>
<td>0x00 8800</td>
<td>0x0800</td>
<td>data</td>
</tr>
<tr>
<td>FLASH</td>
<td>0x3E 8000</td>
<td>0xFF80</td>
<td>code</td>
</tr>
</tbody>
</table>

**Link Uninitialized Sections to RAM**

16. Right click on MEM - Memory Section Manager and select Properties. Select the Compiler Sections tab and link the following uninitialized sections into the MSARAM memory block via the pull-down boxes.

<table>
<thead>
<tr>
<th>MSARAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>.bss</td>
</tr>
<tr>
<td>.ebss</td>
</tr>
</tbody>
</table>
Lab 12: DSP/BIOS

Link Initialized Sections to Flash

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

17. This step assigns the RUN address of those sections that need to run from flash. Using the MEM – Memory Section Manager in the DSP/BIOS configuration tool link the following sections to on-chip flash memory via the pull-down boxes:

<table>
<thead>
<tr>
<th>BIOS Data tab</th>
<th>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>.sysinit</td>
<td>.switch</td>
<td></td>
</tr>
<tr>
<td>.hwi</td>
<td>.cinit</td>
<td></td>
</tr>
<tr>
<td>.rtdx_text</td>
<td>.pinit</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>.econst / .const</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.data / .cio</td>
</tr>
</tbody>
</table>

18. This step assigns the LOAD address of those sections that need to load to flash. Again using the MEM – Memory Section Manager in the DSP/BIOS configuration tool select the Load Address tab and check the “Specify Separate Load Addresses” box. Then set all entries to the FLASH memory block.

19. Click the BIOS Data tab and notice that the .stack section has been linked into memory. Click OK to close the window.

20. The section named “IQmath” is an initialized section that needs to load to and run from flash. This section is not linked using the DSP/BIOS configuration tool (because it is neither a standard compiler section nor a DSP/BIOS generated section). Instead, this section is linked with the user linker command file (Lab_12.cmd). Open and inspect Lab_12.cmd. Previously the “IQmath” section was linked to L0SARAM. Notice that this section is now linked to FLASH.
Set the Stack Size in the TCF File

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

21. Click: Project → Build Options… and select the Linker tab. Delete the entry of 0x200 in the Stack Size box. Select OK to close the Build Options window.

22. Using the MEM - Memory Section Manager select the General tab. Set the Stack Size to 0x100. The stack size needs to be reduced from 0x200 to 0x100 because of the limited amount of available RAM on the device when using DSP/BIOS. Click OK to close the window.

Copying .hwi_vec Section from Flash to RAM

The DSP/BIOS .hwi_vec section contains the interrupt vectors. This section must be loaded to flash (load address) but run from RAM (run address). The code that performs this copy is located in InitPieCtrl(). The linker command file generated by the DSP/BIOS configuration tool generates global symbols that can be accessed by code in order to determine the load address, run address, and length of the .hwi_vec section. The RTS library contains a memory copy function called memcpy() which will be used to perform the copy.

23. Open and inspect InitPieCtrl() in PieCtrl_12.c. Notice the memcpy() function and the symbols used to initialize (copy) the .hwi_vec section.

Copying the .trcdata Section from Flash to RAM

The DSP/BIOS .trcdata section is used by CCS and DSP/BIOS for certain real-time debugging features. This section must be loaded to flash (load address) but run from RAM (run address). The linker command file generated by the DSP/BIOS configuration tool generates global symbols that can be accessed by code in order to determine the load address, run address, and length of the .trcdata section. The memory copy function memcpy() will again be used to perform the copy.

The copying of .trcdata must be performed prior to main(). This is because DSP/BIOS modifies the contents of .trcdata during DSP/BIOS initialization, which also occurs prior to main(). The DSP/BIOS configuration tool provides a user initialization function which will be used to perform the .trcdata section copy prior to both main() and DSP/BIOS initialization.

24. In the DSP/BIOS configuration file (Lab.tcf) and select the Properties for the Global Settings. Check the box “Call User Init Function” and enter the UserInit() function name with a leading underscore: _UserInit. This will cause the function UserInit() to execute prior to main(). Click OK to close the window.

25. Open and inspect the file Main_12.c. Notice that the function UserInit() is used to copy the .trcdata section from its load address to its run address before main().
### Initializing the Flash Control Registers

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

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

27. Since the DSP/BIOS configuration tool does not know about user defined sections, the “secureRamFuncs” section will be linked using the user linker command file `Lab_12.cmd`. Open and inspect `Lab_12.cmd`. The “secureRamFuncs” will load to flash (load address) but will run from LSARAM (run address). Also notice that the linker has been asked to generate symbols for the load start, load size, and run start addresses.

28. Open and inspect `Main_12.c`. Notice that the memory copy function `memcpy()` is being used to copy the section ”secureRamFuncs”, which contains the initialization function for the flash control registers. Close all the inspected files.

### Setup PIE Vectors for Interrupts in the TCF File

Next, we will setup all of the PIE interrupt vectors that will be needed for the lab exercises in this module. This will include all of the vectors used in the previous lab exercises. (Note: the `PieVect.c` file is not used since DSP/BIOS generates the interrupt vector table).

29. Modify the configuration file `Lab.tcf` to setup the PIE vector for the watchdog interrupt. Click on the plus sign (+) to the left of `Scheduling` and again on the plus sign (+) to the left of `HWI - Hardware Interrupt Service Routine Manager`. Click the plus sign (+) to the left of `PIE INTERRUPTS`. Locate the interrupt entry for the watchdog at `PIE_INT1_8`. Right click, select `Properties`, and type `_WAKEINT_ISR` (with a leading underscore) in the function field. Click `OK` to save.

30. Setup the PIE vector for the ADC interrupt. Locate the interrupt entry for the ADC at `PIE_INT1_1`. Right click, select `Properties`, and type `_ADCINT1_ISR` (with a leading underscore) in the function field. Click `OK` to save.

31. Setup the PIE vector for the ECAP1 interrupt. Locate the interrupt entry for the ECAP1 at `PIE_INT4_1`. Right click, select `Properties`, and type `_ECAP1_INT_ISR` (with a leading underscore) in the function field. Click `OK` to save.

32. Setup the PIE vector for the CLA Task 1 interrupt. Locate the interrupt entry for the CLA Task 1 at `PIE_INT11_1`. Right click, select `Properties`, and type `_CLA1_INT1_ISR` (with a leading underscore) in the function field. Click `OK` to save. Close the configuration window and select `YES` to save changes to `Lab.tcf`. 
Prepare main() for DSP/BIOS

33. Open Main_12.c and delete the inline assembly code from main() that enables global interrupts. DSP/BIOS will enable global interrupts after main().

34. In Main_12.c, remove the endless while() loop from the end of main(). When using DSP/BIOS, you must return from main(). In all DSP/BIOS programs, the main() function should contain all one-time user-defined initialization functions. DSP/BIOS will then take-over control of the software execution. Save and close the file.

Configuring DSP/BIOS Global Settings

35. Open the configuration file Lab.tcf and click on the plus sign (+) to the left of System. Right click on Global Settings and select Properties. Confirm that the “DSP Speed in MHz (CLKOUT)” field is set to 60 so that it matches the processor speed. Click OK to save the value and close the configuration window. This value is used by the CLK manager to calculate the register settings for the on-chip timers and provide the proper time-base for executing CLK functions.

Create a SWI

36. Open Main_12.c and notice that at the end of main() two new functions have been added – Cla1Swi() and LedBlink(). We moved part of the CLA1_INT1_ISR() routine from DefaultIsr_12.c to this space in Main_12.c.

37. Open DefaultIsr_12.c and locate the CLA1_INT1_ISR() routine. The entire contents of the CLA1_INT1_ISR() routine was moved to the Cla1Swi() function in Main_12.c with the following exceptions:

- The instruction used to acknowledge the PIE group interrupt
- The GPIO pin (LED) toggle code

Comment: In almost all 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 toggle code in the HWI just as an example. It illustrates that you can post a SWI and also do additional operations in the HWI. DSP/BIOS is extremely flexible!

38. Delete the interrupt key word from the CLA1_INT1_ISR. The interrupt keyword is not used when a HWI is under DSP/BIOS control. A HWI is under DSP/BIOS control when it uses any DSP/BIOS functionality, such as posting a SWI, or calling any DSP/BIOS function or macro.

Post a SWI

39. Still in DefaultIsr_12.c add the following SWI_post to the CLA1_INT1_ISR(), just after the structure used to acknowledge the PIE group:
SWI_post(&CLA1_swi);  // post a SWI

This posts a SWI that will execute the CLA1_sw() code that was moved to the
Cla1Sw() function in Main_12.c. In other words, the CLA1 interrupt still executes
the same code as before. However, most of that code is now in a posted SWI that
DSP/BIOS will execute according to the specified scheduling priorities. Save and close
the modified files.

Add the SWI to the TCF File

40. In the configuration file Lab.tcf we need to add and setup the Cla1Sw() SWI. Open
Lab.tcf and click on the plus sign (+) to the left of Scheduling and again on the
plus sign (+) to the left of SWI – Software Interrupt Manager.

41. Right click on SWI – Software Interrupt Manager and select Insert SWI.
Rename SWI0 to CLA1_swi and click OK. This is just an arbitrary name. We want to
differentiate the Cla1Sw() function itself (which is nothing but an ordinary C function)
from the DSP/BIOS SWI object which we are calling CLA1_sw.

42. Select the Properties for CLA1_sw and type _Cla1Swi (with a leading
underscore) in the function field. Click OK. This tells DSP/BIOS that it should run the
function Cla1Sw() when it executes the CLA1_sw SWI.

43. We need to have the PIE for the CLA Task 1 interrupt use the dispatcher. The dispatcher
will automatically perform the context save and restore, and allow the DSP/BIOS
scheduler to have insight into the ISR. You may recall from an earlier lab that the CLA
Task 1 interrupt is located at PIE_INT11_1.

   Click on the plus sign (+) to the left of HWI – Hardware Interrupt Service
   Routine Manager. Click the plus sign (+) to the left of PIE INTERRUPTS. Locate
   the interrupt entry for the CLA Task 1: PIE_INT11_1. Right click, select Properties,
   and select the Dispatcher tab. Check the “Use Dispatcher” box and select OK.
   Close the configuration file and click YES to save changes.

Add a Periodic Function

Recall that an instruction was used in the CLA1_INT1_ISR to toggle the LED on the
ControlCARD. This instruction has been moved into a periodic function that will toggle the LED
at the same rate.

44. Open DefaultIsr_12.c and locate the CLA1_INT1_ISR routine. Notice that the
instruction used to toggle the LED was moved to the LedBlink() function in
Main_12.c:

   GpioDataRegs.GPBTOGGLE.bit.GPIO34 = 1;  // Toggle the pin

Also, the code used to implement the interval counter for the LED toggle (i.e., the
GPIO32_count++ loop), and the declaration of the GPIO32_count itself from the
beginning of CLA1_INT1_ISR() have been deleted. These are no longer needed, as
DSP/BIOS will implement the interval counter for us in the periodic function
configuration (next step in the lab). Close the inspected files.
45. In the configuration file `Lab.tcf` we need to add and setup the LedBlink_PRD. Open `Lab.tcf` and click on the plus sign (+) to the left of Scheduling. Right click on PRD – Periodic Function Manager and select Insert PRD. Rename PRD0 to LedBlink_PRD and click OK.

Select the Properties for LedBlink_PRD and type `_LedBlink` (with a leading underscore) in the function field. This tells DSP/BIOS to run the LedBlink() function when it executes the LedBlink_PRD periodic function object.

Next, in the period (ticks) field type `500`. The default DSP/BIOS system timer increments every 1 millisecond, so what we are doing is telling the DSP/BIOS scheduler to schedule the LedBlink() function to execute every 500 milliseconds. A PRD object is just a special type of SWI which gets scheduled periodically and runs in the context of the SWI level at a specified SWI priority. Click OK. Close the configuration file and click YES to save changes.

DSP/BIOS – Real-time Analysis Tools

The DSP/BIOS analysis tools complement the CCS environment by enabling real-time program analysis of a DSP/BIOS application. You can visually monitor an MCU application as it runs with essentially no impact on the application’s real-time performance. In CCS, the DSP/BIOS real-time analysis (RTA) tools are found on the DSP/BIOS menu. Unlike traditional debugging, which is external to the executing program, DSP/BIOS program analysis requires that the target program be instrumented with analysis code. By using DSP/BIOS APIs and objects, developers automatically instrument the target for capturing and uploading real-time information to CCS using these tools.

46. In the next few steps the Log Event Manager will be setup to record the occurrence of an event in real-time while the program executes. We will be using `LOG_printf()` to write to a log buffer. The `LOG_printf()` function is a very efficient means of sending a message from the code to the CCS display. Unlike an ordinary C-language `printf()`, which can consume several hundred CPU cycles to format the data on the MCU before transmission to the CCS host PC, a `LOG_printf()` transmits the raw data to the host. The host then formats the data and displays it in CCS. This consumes only 10’s of cycles rather than 100’s of cycles.

In `Main_12.c` notice the following code at the top of the LedBlink() function just before the instruction used to toggle the LED:

```c
static Uint16 LedSwiCount=0;          // used for LOG_printf

/*** Using LOG_printf() to write to a log buffer ***/
LOG_printf(&trace, "LedSwiCount = %u", LedSwiCount++);
```

Close the file.

47. In the configuration file `Lab.tcf` we need to add and setup the trace buffer. Open `Lab.tcf` and click on the plus sign (+) to the left of Instrumentation and again on the plus sign (+) to the left of LOG – Event Log Manager.
48. Right click on LOG – Event Log Manager and select Insert LOG. Rename LOG0 to trace and click OK.

49. Select the Properties for trace and confirm that the logtype is set to circular and the datatype is set to printf. Click OK. Close the configuration file and click YES to save changes.

**Build – Lab.out**

50. At this point we need to build the project, but not have CCS automatically load it since CCS cannot load code into the flash (the flash must be programmed)! On the menu bar click: Option → Customize... and select the ”Program/Project CIO” tab and confirm that the “Load Program After Build” is unchecked.

Next select the “Debug Properties” tab and confirm that the “Step over functions without debug information when source stepping” is unchecked. Then click OK.

51. Click the “Build” button to generate Lab.out.

**CCS Flash Plug-in**

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

   Tools → F28xx On-Chip Flash Programmer

53. A Clock Configuration window may open. If needed, in the Clock Configuration window set “OSCCLK (MHz):” to 10, “DIVSEL:” to /2, and “PLLCR Value:” to 12. Then click OK. In the next Flash Programmer Settings window confirm that the selected DSP device to program is F28035 and all options have been checked. Click OK.

54. The CCS Flash Programmer uses the Piccolo™ 10 MHz internal oscillator as the device clock during programming. Confirm the “Clock Configuration” in the upper left corner has the OSCCLK set to 10 MHz, the DIVSEL set to /2, and the PLLCR value set to 12. Recall that the PLL is divided by two, which gives a SYSCLKOUT of 60 MHz.

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

56. We will not be using the plug-in to program the “Code Security Password”. **Do not modify the Code Security Password fields.** They should remain as all 0xFFFF.

57. In the “Operation” block, notice that the “COFF file to Program/Verify” field automatically defaults to the current .out file. Check to be sure that “Erase, Program, Verify” is selected. We will be using the default wait states, as shown on the slide in this module. The selection for wait-states only affects the verify step, and makes little noticeable difference even if you reduce the wait-states.
58. Click “Execute Operation” to program the flash memory. Watch the programming status update in the plug-in window.

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

**Running the Code – Using CCS**

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

   File ➔ Load Symbols ➔ Load Symbols Only...

   and select Lab12.out in the Debug folder.

61. Reset the CPU. The program counter should now be at 0x3FF8A1, which is the start of the bootloader in the Boot ROM.

62. Under GEL on the menu bar click:

   EMU Boot Mode Select ➔ EMU_BOOT_FLASH.

   This has the debugger load values into EMU_KEY and EMU_BMODE so that the bootloader will jump to "FLASH" at 0x3F7FF6.

63. Single-Step <F11> through the bootloader code until you arrive at the beginning of the codestart section in the CodeStartBranch.asm file. (Be patient, it will take about 125 single-steps). Notice that we have placed some code in CodeStartBranch.asm to give an option to first disable the watchdog, if selected.

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

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

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

**Run the Code – Real-time Analysis Tools**

It will be interesting to investigate the CPU computational burden of the the different pieces of DSP/BIOS real-time analysis tools that we will be using in this lab exercise. The ‘CPU Load Graph’ feature of DSP/BIOS will provide a quick and easy method for doing this. We will be tabulating these results in the table that follows at various steps throughout the remainder of this lab.
Table 12-1: CPU Computational Burden Results

<table>
<thead>
<tr>
<th>Case #</th>
<th>Description</th>
<th>CPU Load %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CLA processing handled in SWI. LED blink handled in PRD. RTA Global Host Enable disabled.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Case #1 + LOG_printf in SWI.</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Case #2 + RTA SWI Logging enabled.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Case #3 + RTA SWI Accumulators enabled.</td>
<td></td>
</tr>
</tbody>
</table>

67. Open the RTA Control Panel by clicking DSP/BIOS → RTA Control Panel. Uncheck ALL of the boxes. This disables most of the realtime analysis tools. We will selectively enable them in the lab.

68. Open the CPU Load Graph by clicking DSP/BIOS → CPU Load Graph. The CPU load graph displays the percentage of available CPU computing horsepower that the application is consuming. The CPU may be running ISRs, software interrupts, periodic functions, performing I/O with the host, or running any user routine. When the CPU is not executing user code, it will be idle (in the DSP/BIOS idle thread).

69. Record the value shown in the CPU Load Graph under “Case #1” in Table 12-1.

70. Open the Message Log. On the menu bar, click:

DSP/BIOS → Message Log

The message log dialog box is displaying the commanded LOG_printf() output, i.e. the number of times (count value) that the LedSwi() has executed.

71. Verify that all the check boxes in the RTA Control Panel window are still unchecked. Then, check the box marked “Global Host Enable.” This is the main control switch for most of the RTA tools. We will be selectively enabling the rest of the check boxes in this portion of the exercise.

72. Record the value shown in the CPU Load Graph under “Case #2” in Table 12-1.

73. Open the Execution Graph. On the menu bar, click:

DSP/BIOS → Execution Graph
Presently, the execution graph is not displaying anything. This is because we have it disabled in the RTA Control Panel.

In the RTA Control Panel, check the top four boxes to enable logging of all event types to the execution graph. Notice that the Execution Graph is now displaying information about the execution threads being taken by your software. This graph is not based on time, but the activity of events (i.e. when an event happens, such as a SWI or periodic function begins execution). Notice that the execution graph simply records DSP/BIOS CLK events along with other system events (the DSP/BIOS clock periodically triggers the DSP/BIOS scheduler). As a result, the time scale on the execution graph is not linear.

The logging of events to the execution graph consumes CPU cycles, which is why the CPU Load Graph jumped as you enabled logging.

74. Record the value shown in the CPU Load Graph under “Case #3” in Table 12-1.

75. Open the Statistics View window. On the menu bar, click:

DSP/BIOS → Statistics View

Presently, the statistics view window is not changing with the exception of the statistics for the IDL_busyObj row (i.e., the idle loop). This is because we have it disabled in the RTA Control Panel.

In the RTA Control Panel, check the next five boxes (i.e., those with the word “Accumulator” in their description) to enable logging of statistics to the statistics view window. The logging of statistics consumes CPU cycles, which is why the CPU Load Graph jumped as you enabled logging.

76. Record the value shown in the CPU Load Graph under “Case #4” in Table 12-1.

77. Table 12-1 should now be completely filled in. Think about the results.

Note: In this lab exercise only the basic features of DSP/BIOS and the real-time analysis tools have been used. For more information and details, please refer to the DSP/BIOS user’s manuals and other DSP/BIOS related training.

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

78. Close Code Composer Studio.

79. Disconnect the USB cable (emulator) from the Docking Station (i.e. remove power from the ControlCARD).

80. Re-connect the USB cable to the Docking Station to power the ControlCARD. The LED should be blinking, showing that the code is now running from flash memory.

End of Exercise
Lab 12 Reference: Programming the Flash

Flash Memory Section Blocks

```
<table>
<thead>
<tr>
<th>Base Address</th>
<th>Section</th>
<th>Start Address</th>
<th>End Address</th>
<th>Length</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3E 8000</td>
<td>FLASH</td>
<td>0x3E 8000</td>
<td>0xFFFF80</td>
<td>0xFF80</td>
<td>code</td>
</tr>
<tr>
<td>0x3F 7F80</td>
<td>CSM_RSVD</td>
<td>0x3F 7F80</td>
<td>0x3F 7F80</td>
<td>0x76</td>
<td>code</td>
</tr>
<tr>
<td>0x3F 7FF6</td>
<td>BEGIN_FLASH</td>
<td>0x3F 7FF6</td>
<td>0x3F 7FF8</td>
<td>0x2</td>
<td>code</td>
</tr>
<tr>
<td>0x3F 7FF8</td>
<td>PASSWORDS</td>
<td>0x3F 7FF8</td>
<td>0x3F 7FF8</td>
<td>0x8</td>
<td>code</td>
</tr>
</tbody>
</table>
```

Lab_12.cmd

```
SECTIONS
{
    codestart  :>  BEGIN_FLASH,  PAGE = 0
    passwords  :>  PASSWORDS,   PAGE = 0
    csm_rsvd   :>  CSM_RSVD,    PAGE = 0
}
```

BIOS Startup Sequence from Flash Memory

1. **RESET**
2. Boot ROM (8Kw)
3. Boot Code (0x3F F8A1)
   - (SCAN GPIO)
4. BIOS code Sections
   - BIOS_reset()
   - BIOS_init()
   - main()
   - BIOS_start()
5. IDL_run()
   - "rts2800_ml.lib"
   - "user" code sections
   - main()
   - { 
       ...... return;
   }
6. BIOS code Sections
   - BIOS_reset()
   - BIOS_init()
   - main()
   - BIOS_start()
Table 12-2: CPU Computational Burden Results (Solution)

<table>
<thead>
<tr>
<th>Case #</th>
<th>Description</th>
<th>CPU Load %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CLA processing handled in SWI. LED blink handled in PRD. RTA Global Host Enable disabled.</td>
<td>27.5</td>
</tr>
<tr>
<td>2</td>
<td>Case #1 + LOG_printf in SWI.</td>
<td>27.5</td>
</tr>
<tr>
<td>3</td>
<td>Case #2 + RTA SWI Logging enabled.</td>
<td>37.0</td>
</tr>
<tr>
<td>4</td>
<td>Case #3 + RTA SWI Accumulators enabled.</td>
<td>48.6</td>
</tr>
</tbody>
</table>
Development Support

Introduction

This module contains various references to support the development process.

Learning Objectives

- TI Workshops Download Site
- Signal Processing Libraries
- TI Development Tools
- Additional Resources
  - Internet
  - Product Information Center
Module Topics

Development Support ................................................................. 13-1

Module Topics............................................................................. 13-2

TI Support Resources ................................................................. 13-3
  C28x Signal Processing Libraries ........................................... 13-3
  Experimenter's Kits ................................................................. 13-4
  F28335 Peripheral Explorer Kit ............................................. 13-5
  C2000 ControlCARD Application Kits ................................. 13-5
  Product Information Resources ............................................ 13-6
TI Support Resources

**TI Workshops Download Site**

http://www.tiworkshop.com/survey/downloadsort.asp

- Login Name: c28xmdw
- Password: ttoc28

---

**C28x Signal Processing Libraries**

**C2000 Signal Processing Libraries**

<table>
<thead>
<tr>
<th>Signal Processing Libraries &amp; Applications Software</th>
<th>Literature #</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACI3-1: Control with Constant V/Hz</td>
<td>SPRC194</td>
</tr>
<tr>
<td>ACI3-3: Sensored Indirect Flux Vector Control</td>
<td>SPRC207</td>
</tr>
<tr>
<td>ACI3-3: Sensored Indirect Flux Vector Control (simulation)</td>
<td>SPRC208</td>
</tr>
<tr>
<td>ACI3-4: Sensorless Direct Flux Vector Control</td>
<td>SPRC195</td>
</tr>
<tr>
<td>ACI3-4: Sensorless Direct Flux Vector Control (simulation)</td>
<td>SPRC209</td>
</tr>
<tr>
<td>PMSM3-1: Sensored Field Oriented Control using QEP</td>
<td>SPRC210</td>
</tr>
<tr>
<td>PMSM3-2: Sensorless Field Oriented Control</td>
<td>SPRC197</td>
</tr>
<tr>
<td>PMSM3-3: Sensored Field Oriented Control using Resolver</td>
<td>SPRC211</td>
</tr>
<tr>
<td>PMSM3-4: Sensorless Position Control using QEP</td>
<td>SPRC212</td>
</tr>
<tr>
<td>BLD3C-1: Sensored Trapezoidal Control using Hall Sensors</td>
<td>SPRC213</td>
</tr>
<tr>
<td>BLD3C-2: Sensorless Trapezoidal Drive</td>
<td>SPRC196</td>
</tr>
<tr>
<td>DCMOTOR: Speed &amp; Position Control using QEP without Index</td>
<td>SPRC214</td>
</tr>
<tr>
<td>Digital Motor Control Library (F/C280x)</td>
<td>SPRC215</td>
</tr>
<tr>
<td>Communications Driver Library</td>
<td>SPRC183</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 FPU Fast RTS Library</td>
<td>SPRC664</td>
</tr>
<tr>
<td>DSP2803x C/C++ Header Files and Peripheral Examples</td>
<td>SPRC082</td>
</tr>
</tbody>
</table>

Available from TI Website ⇒ http://www.ti.com/c2000
Experimenter’s Kits

C2000 Experimenter’s Kits
F28027, F28035, F2808, F28335

- Experimenter Kits include
  - F28027, F28035, F2808 or F28335 ControlCARD
  - USB docking station
  - C2000 Applications Software CD with example code and full hardware details
  - Code Composer Studio v3.3 with code size limit of 32KB

- Docking station features
  - Access to ControlCARD signals
  - Breadboard areas
  - Onboard USB JTAG Emulation
    - JTAG emulator not required

- Available through TI authorized distributors and the TI eStore

C2834x Experimenter’s Kits
C28343, C28346

- Experimenter Kits include
  - C2834x ControlCARD
  - Docking station
  - C2000 Applications Software CD with example code and full hardware details
  - Code Composer Studio v3.3 with code size limit of 32KB
  - 5V power supply

- Docking station features
  - Access to ControlCARD signals
  - Breadboard areas
  - JTAG emulator required – sold separately

- Available through TI authorized distributors and the TI eStore
F28335 Peripheral Explorer Kit

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

C2000 ControlCARD Application Kits

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

For More Information . . .

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

FAQ: http://www-k.ext.ti.com/sc/technical_support/knowledgebase.htm
• Device information
• Application notes
• Technical documentation
• my.ti.com
• News and events
• Training

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

USA - Product Information Center (PIC)
Phone: 800-477-8924 or 972-644-5580
Email: support@ti.com
• Information and support for all TI Semiconductor products/tools
• Submit suggestions and errata for tools, silicon and documents

European Product Information Center (EPIC)
Web: http://www-k.ext.ti.com/sc/technical_support/pic/euro.htm

<table>
<thead>
<tr>
<th>Phone: 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

Appendix A – Experimenter's Kit ................................................................. A-1

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

F28035 ControlCARD .................................................................................. A-3
F28035 PCB Outline (Top View) ................................................................. A-3
LD1 / LD2 / LD3 .......................................................................................... A-3
SW1 .............................................................................................................. A-3
SW2 .............................................................................................................. A-4
SW3 .............................................................................................................. A-4

F28335 ControlCARD .................................................................................. A-5
F28335 PCB Outline (Top View) ................................................................. A-5
LD1 / LD2 / LD3 .......................................................................................... A-5

Docking Station ........................................................................................... A-6
SW1 / LD1 .................................................................................................... A-6
JP1 / JP2 ....................................................................................................... A-6
F2833x Boot Mode Selection ........................................................................ A-7
F280xx Boot Mode Selection ....................................................................... A-7
J3 – DB-9 to 4-Pin Header Cable ................................................................. A-8
F28035 ControlCARD

F28035 PCB Outline (Top View)

LD1 / LD2 / LD3

LD1 – Turns on when controlCARD is powered on
LD2 – Controlled by GPIO-31
LD3 – Controlled by GPIO-34

SW1

SW1 – controls whether on-card RS-232 connection is enabled or disabled.
- **ON** – RS-232 transceiver will be enabled and allow communication through a serial cable via pins 2 and 42 of the DIMM-100 socket. Putting SW1 in the “ON” position will allow the F28035 controlCARD to be card compatible with the F2808, F28044, F28335, and F28027 controlCARDs. GPIO-28 will be stuck as logic high in this position.
- **OFF** – The default option. SW1 in the “OFF” position allows GPIO-28 to be used as a GPIO. Serial communication is still possible, however an external transceiver such as the FTDI – FT2232D chip.
SW2

SW2 – controls the boot options of the F28035 device

<table>
<thead>
<tr>
<th>Position 1 (GPIO-34)</th>
<th>Position 2 (TDO)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

SW3

SW3 – ADC VREF control

The ADC will by default convert from 0 to 3.3V, however if in the ADC registers the ADC is configured to use external limits the ADC will convert its full range of resolution from VREF-LO to VREF-HI.

Position 1 controls VREF-HI, the value that the ratiometric ADC will convert as the maximum 12-bit value, 0x0FFF. In the downward position, VREF-HI will be connected to 3.3V. In the upward position, VREF-HI will be connected to pin 66 of the DIMM100-socket. This would allow a connecting board to control the ADC-VREFHI value.

Position 2 controls VREF-LO, the value that the ratiometric ADC will convert as the minimum 12-bit value, 0x0000. In the downward position, VREF-LO will be connected to 0V. In the upward position, VREF-LO will be connected to pin 16 of the DIMM100-socket. This would allow a connecting board to control the ADC-VREFLO value.
F28335 ControlCARD

F28335 PCB Outline (Top View)

LD1 / LD2 / LD3

LD1 – Turns on when controlCARD is powered on
LD2 – Controlled by GPIO-31
LD3 – Controlled by GPIO-34
Docking Station

SW1 / LD1
SW1 – USB: Power from USB; ON – Power from JP1
LD1 – Power-On indicator

JP1 / JP2
JP1 – 5.0 V power supply input
JP2 – USB JTAG emulation port

J1 – ControlCARD 100-pin DIMM socket
J2 – JTAG header connector
J3 – UART communications header connector
J8 – Internal emulation enable/disable jumper (NO jumper for internal emulation)
J9 – User virtual COM port to C2000 device (Note: ControlCARD would need to be modified to disconnect the C2000 UART connection from header J3)
Note: The internal emulation logic on the Docking Station routes through the FT2232 USB device. By default this device enables the USB connection to perform JTAG communication and in parallel create a virtual serial port (SCI/UART). As shipped, the C2000 device is not connected to the virtual COM port and is instead connected to J3.

### F2833x Boot Mode Selection

<table>
<thead>
<tr>
<th>MODE</th>
<th>GPIO87/XA15</th>
<th>GPIO86/XA14</th>
<th>GPIO85/XA13</th>
<th>GPIO84/XA12</th>
<th>MODE(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Jump to Flash</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>SCI-A boot</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>SPI-A boot</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>I2C-A boot</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>eCAN-A boot</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>McDSP-A boot</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Jump to XINTF x16</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Jump to XINTF x32</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Jump to OTP</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Parallel GPIO I/O boot</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Parallel XINTF boot</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Jump to SARAM</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Branch to check boot mode</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Branch to Flash, skip ADC calibration</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Branch to SARAM, skip ADC calibration</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Branch to SCI, skip ADC calibration</td>
</tr>
</tbody>
</table>

(1) All four GPIO pins have an internal pullup.

### F280xx Boot Mode Selection

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
<th>GPIO18 SPICLKA</th>
<th>GPIO29 SCITXDA</th>
<th>GPIO34</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boot to Flash</td>
<td>Jump to flash address 0x3F 7FF8. You must have programmed a branch instruction here prior to reset to redirect code execution as desired.</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SCI-A Boot</td>
<td>Load a data stream from SCI-A.</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>SPI-A Boot</td>
<td>Load from an external serial SPI EEPROM on SPI-A.</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>P2 Boot</td>
<td>Load data from an external EEPROM at address 0x50 on the P2 bus.</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>eCAN-A Boot (3)</td>
<td>Call CAN_Boot to load from eCAN-A mailbox 1.</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Boot to M0 SARAM</td>
<td>Jump to M0 SARAM address 0x00 0000.</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Boot to OTP (4)</td>
<td>Jump to OTP address 0x3D 7800.</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Parallel I/O Boot</td>
<td>Load data from GPIO00 - GPIO15.</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(1) You must take extra care because of any effect toggling SPICLKA to select a boot mode may have on external logic.
(2) When booting directly to flash, it is assumed that you have previously programmed a branch statement at 0x3F 7FF8 to redirect program flow as desired.
(3) On devices that do not have an eCAN-A module this configuration is reserved. If it is selected, then the eCAN-A bootloader will run and will loop forever waiting for an incoming message.
(4) When booting directly to OTP or M0 SARAM, it is assumed that you have previously programmed or loaded code starting at the entry point location.
J3 – DB-9 to 4-Pin Header Cable

Note: This cable is NOT included with the Experimenter’s Kit and is only shown for reference.

Pin-Out Table for Both Ends of the Cable:

**DB-9 female** | **SIL 0.1” female**
---|---
Pin# | Pin#
---|---
2 (black) | 1 (TX)
3 (red) | 4 (RX)
5 (bare wire) | 3 (GND)

Note: pin 2 on SIL is a no-connect
Appendix B – Addressing Modes

Introduction

Appendix B will describe the data addressing modes on the C28x. Immediate addressing allows for constant expressions which are especially useful in the initialization process. Indirect addressing uses auxiliary registers as pointers for accessing organized data in arrays. Direct addressing is used to access general purpose memory. Techniques for managing data pages, relevant to direct addressing will be covered as well. Finally, register addressing allows for interchange between CPU registers.

Learning Objectives

- 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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV ACC,#1</td>
<td>Move Accumulator C to Accumulator 1</td>
</tr>
<tr>
<td>MOVL XAR1,#x</td>
<td>Move Low 16 bits of XAR1 to Accumulator 1</td>
</tr>
<tr>
<td>MOV AR2,#count</td>
<td>Move Accumulator 2 to Accumulator 2</td>
</tr>
<tr>
<td>MOV *XAR1++,AL</td>
<td>Move contents of XAR1 to accumulator 1 and increment</td>
</tr>
<tr>
<td>BANZ loop,AR2--</td>
<td>Branch if not equal to zero to loop and decrement Accumulator 2</td>
</tr>
<tr>
<td>ADD ACC,#1</td>
<td>Add Accumulator 1 to Accumulator C and store in Accumulator C</td>
</tr>
<tr>
<td>SB next,UNC</td>
<td>Store Byte at memory location next and undefined</td>
</tr>
</tbody>
</table>

#### Assembly Directives

- **Directives** allow you to:
  - Define a label as global
  - Reserve space in memory for un-initialized variables
  - Initialize memory

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.sect “name”</td>
<td>Used for code or constants</td>
</tr>
<tr>
<td>.usect “name”,5</td>
<td>Used for variables</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV ACC,#1</td>
<td>Move Accumulator C to Accumulator 1</td>
</tr>
<tr>
<td>MOVL XAR1,#x</td>
<td>Move Low 16 bits of XAR1 to Accumulator 1</td>
</tr>
<tr>
<td>MOV AR2,#count</td>
<td>Move Accumulator 2 to Accumulator 2</td>
</tr>
<tr>
<td>MOV *XAR1++,AL</td>
<td>Move contents of XAR1 to accumulator 1 and increment</td>
</tr>
<tr>
<td>BANZ loop,AR2--</td>
<td>Branch if not equal to zero to loop and decrement Accumulator 2</td>
</tr>
<tr>
<td>ADD ACC,#1</td>
<td>Add Accumulator 1 to Accumulator C and store in Accumulator C</td>
</tr>
<tr>
<td>SB next,UNC</td>
<td>Store Byte at memory location next and undefined</td>
</tr>
</tbody>
</table>
Addressing Modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Symbol</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td></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.
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.

### Instruction Formats

<table>
<thead>
<tr>
<th>INSTR</th>
<th>dst , src</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSTR REG</td>
<td>REG</td>
<td>NEG AL</td>
</tr>
<tr>
<td>INSTR REG,#imm</td>
<td>REG,#imm</td>
<td>MOV ACC,#1</td>
</tr>
<tr>
<td>INSTR REG,mem</td>
<td>mem,REG</td>
<td>ADD AL,@x</td>
</tr>
<tr>
<td>INSTR mem,REG</td>
<td>mem,#imm</td>
<td>SUB AL,@AR0</td>
</tr>
<tr>
<td>INSTR mem,#imm</td>
<td>*XAR0++,#25</td>
<td></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

Register Addressing

32-bit Registers
XAR0 – XAR7  ACC  P  XT

16-bit Registers
AR0 – AR7  AH  AL  PH  PL  T  TL  DP  SP

- Allows for efficient register to register operation
- 16-bit and 32-bit Register Address modes
- Reduces code overhead, memory accesses, and memory overhead

Register addressing allows the exchange of values between registers, and with certain instructions can be used in conjunction with other addressing modes, yielding a more efficient instruction set. Remember that any ‘mem’ field allows the use of a register as the operand, and that no special character (such as @, *, or #) need be used to specify the register mode.

Register Addressing – Example

<table>
<thead>
<tr>
<th>Format</th>
<th>Instruction</th>
<th>Instruction</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MOV Ax,loc16</td>
<td>MOVL loc32,ACC</td>
<td>MOV @XT,ACC</td>
</tr>
<tr>
<td></td>
<td>MOV AH,@AL</td>
<td>MOV @XT,ACC</td>
<td></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)
  
  \[
  \text{ANDB} \ Ax, \#8\text{Bit} \\
  \text{ANDB} \ Ax \ #8\text{Bit}
  \]

  AND automatically replaced by ANDB if IMM value is 8 bits or less

- Long Immediate, 2 Words (AND)
  
  \[
  \text{AND} \ loc16, \#16\text{Bit} \\
  \text{AND} \ loc16, \#16\text{Bit}
  \]

  \[
  \text{AND} \ Ax, \ loc16, \#16\text{Bit} \\
  \text{AND} \ Ax \ #16\text{Bit}
  \]

  \[
  \text{AND} \ ACC, \#16\text{Bit}, \ll<0-16 \\
  \text{AND} \ ACC \ shift \ #16\text{Bit}
  \]
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.

- 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

\[ Z = X + Y \]

```
x .usect "samp", 3
.sect "code"
MOVW DP, #x
MOV AL, @x
ADD AL, @y
MOV @z, AL
```

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>Page7[00]</td>
<td></td>
</tr>
<tr>
<td>0001FE</td>
<td>0500</td>
</tr>
<tr>
<td>Page7[3E]</td>
<td></td>
</tr>
<tr>
<td>0001FF</td>
<td>1500</td>
</tr>
<tr>
<td>Page7[3F]</td>
<td></td>
</tr>
</tbody>
</table>

Variations:
- MOVW DP, #imm ; 2W, 16-bit (4 Meg)
- MOVZ DP, #imm ; 1W, 10-bit (64K)
- MOV DP, #imm ; DP(15:10) unchanged

Direct Addressing – Caveats

(X and Y not on the same page)

```
MOV AL, @x
ADD AL, @y
MOV @z, AL
```

Expected:

```
0000 0000 0000 0010 0000 0000
```

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
```

C2000 Piccolo Workshop - Appendix B - Addressing Modes
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 – Example**

**Autoincrement**

\[
y = \sum_{n=0}^{4} x_n
\]

Indexed addressing offers the ability to select operands from within an array without modification to the base pointer. Stack-based operations are handled with a 16-bit Stack Pointer register, which operates over the base 64K of data memory. It offers 6-bit non-destructive indexing to access larger stack-based arrays efficiently.

**Indirect Addressing – Example**

**Offset**

Indirect Addressing – Example

Stack Relative

\[ x_2 = x_1 + x_3 \]

```
.sect " .code"
MOV AL, *-SP[1]
ADD AL, *-SP[3]
MOV *-SP[2], AL
```

Accumulator

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>2</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Useful for stack based operations

Indirect Addressing – Example

Circular

```
MAC P, *AR6%++, *XAR7++
```

SECTIONS

```
SECTIONS
{ Buf_Mem: align(256) { } > RAM PAGE 1

  . . .
}
```

LINKER.CMD

- Start of buffer
- Access pointer
- End of buffer
- Buffer size N

Circular buffer range

Buffer Size N

Element 0

Element N-1

N-1

AR1 Low (16)

XAR6 (32)

Align on 256 word boundary

AR1 Low is set to buffer size – 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

Given:

<table>
<thead>
<tr>
<th>Address/Data (hex)</th>
<th>DP = 4000</th>
<th>DP = 4004</th>
<th>DP = 4006</th>
</tr>
</thead>
<tbody>
<tr>
<td>100030</td>
<td>0025</td>
<td>100100</td>
<td>1001180</td>
</tr>
<tr>
<td>100031</td>
<td>0120</td>
<td>100101</td>
<td>100181</td>
</tr>
<tr>
<td>100032</td>
<td>0030</td>
<td>100102</td>
<td>100182</td>
</tr>
</tbody>
</table>

Fill in the table below:

<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></td>
<td>MOVW DP,#4000h</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MOV W XAR1,#100100h</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MOV L XAR2,#100180h</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MOV AL,@31h</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ADD AL,*XAR1++</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SUB AL,@30h</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ADD AL,*XAR1++</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MOV AL,#4006h</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ADD AL,@1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SUB AL,*XAR1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ADD AL,*XAR2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SUB AL,*+XAR2[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ADD AL,#32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SUB AL,*+XAR2[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MOV @32h,AL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In the table above, fill in the values for each of the registers for each of the instructions. Three areas of data memory are displayed at the top of the diagram, showing both their addresses and contents in hexadecimal. Watch out for surprises along the way. First, you should answer the addressing mode for the source operand. Then, fill in the change values as the result of the instruction operation.
Lab B: Addressing

Note: The lab linker command file is based on the F28035 memory map – modify as needed, if using a different F28xx device memory map.

Objective

The objective of this lab is to practice and verify the mechanics of addressing. In this process we will expand upon the ASM file from the previous lab to include new functions. Additionally, we learn how to run and observe the operation of code using Code Composer Studio.

In this lab, we will initialize the “vars” arrays allocated in the previous lab with the contents of the “const” table. How is this best accomplished? Consider the process of loading the first “const” value into the accumulator and then storing this value to the first “vars” location, and repeating this process for each of the succeeding values.

- What forms of addressing could be used for this purpose?
- Which addressing mode would be best in this case? Why?
- What problems could arise with using another mode?

Procedure

Copy Files, Create Project File

1. Create a new project called LabB.pjt in C:\C28x\Labs\Appendix\LabB and add LabB.asm and Lab.cmd to it. Check your file list to make sure all the files are there. Be sure to setup the Build Options by clicking: Project → Build Options on the menu bar. Select the Linker tab. In the middle of the screen select “No Autoinitialization” under “Autoinit Model:”. Enter “start” in the “Code Entry Point (-e):” field. Next, select the Compiler tab. Note that “Full Symbolic Debug (-g)” under “Generate Debug Info:” is selected. Then select OK to save the Build Options.

Initialize Allocated RAM Array from ROM Initialization Table

2. Edit LabB.asm and modify it to copy table[9] to data[9] using indirect addressing. (Note: data[9] consists of the allocated arrays of data, coeff, and result). Initialize the allocated RAM array from the ROM initialization table:

- Delete the NOP operations from the “code” section.
- Initialize pointers to the beginning of the “const” and “vars” arrays.
- Transfer the first value from “const” to the “vars” array.
- Repeat the process for all values to be initialized.

To perform the copy, consider using a load/store method via the accumulator. Which part of an accumulator (low or high) should be used? Use the following when writing your copy routine:

- use AR1 to hold the address of table
- use AR2 to hold the address of data
3. It is good practice to trap the end of the program (i.e. use either “end: B end,UNC” or “end: B start,UNC”). Save your work.

**Build and Load**

4. Click the “Build” button and watch the tools run in the build window. Debug as necessary. To open up more space, close any open files or windows that you do not need.

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

   File → Load Program...

   If you wish, right click on the LabB.asm source window and select Mixed Mode to debug using both source and assembly.

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

6. Single-step your routine. While single-stepping, it is helpful to see the values located in table[9] and data[9] at the same time. Open two memory windows by using the “View Memory” button on the vertical toolbar and using the address labels `table` and `data`. Setting the properties filed to “Hex 16 Bit – TI style” will give you more viewable data in the window. Additionally, it is useful to watch the CPU registers. Open the CPU registers by using the “View → Registers → CPU Registers”. Deselect “Allow Docking” and move/resize the window as needed. Check to see if the program is working as expected.

   **End of Exercise**
OPTIONAL Lab B-C: Array Initialization in C

Note: The lab linker command file is based on the F28035 memory map – modify as needed, if using a different F28xx device memory map.

Objective

The objective of this lab is to practice and verify the mechanics of initialization using C. Additionally, we learn how to run and observe the operation of C code using Code Composer Studio. In this lab, we will initialize the “vars” arrays with the contents of the “const” table.

Procedure

Create Project File

1. In Code Composer Studio create a new project called LabB-C.pjt in C:\C28x\Labs\Appendix\LabB\LabB-C and add LabB-C.c and Lab.cmd to it. Check your file list to make sure all the files are there. Open the Build Options and select the Linker tab. Select the “Libraries” Category and enter rts2800_ml.lib in the “Incl. Libraries (-l):” box. Do not setup any other Build Options. The default values will be used. In Appendix Lab D exercise, we will experiment and explore the various build options when working with C.

Initialize Allocated RAM Array from ROM Initialization Table

2. Edit LabB-C.c and modify the “main” routine to copy table[9] to the allocated arrays of data[4], coeff[4], and result[1]. (Note: data[9] consists of the allocated arrays of data, coeff, and result).

Build and Load

3. Click the “Build” button and watch the tools run in the build window. Debug as necessary.

Note: Have Code Composer Studio automatically load the output file after a successful build. On the menu bar click: Option → Customize… and select the “Program Load Options” tab, check “Load Program After Build”, then click OK.

4. Under Debug on the menu bar click “Go Main”. Single-step your routine. While single-stepping, it is helpful to see the values located in table[9] and data[9] at the same time. Open two memory windows by using the “View Memory” button on the vertical toolbar and using the address labels table and data. Setting the properties field to “Hex 16 Bit – TI style” will give you more viewable data in the window. Additionally, you can watch the CPU registers. Open the CPU registers by using the “View → Registers → CPU Registers. Deselect “Allow Docking” and move/resize the window as needed. Check to see if the program is working as expected.

End of Exercise
Exercise B: Addressing - Solution

Given:

Address/Data (hex)  

Given:  

DP = 4000  
DP = 4004  
DP = 4006  

Fill in the table below:  

Table below:  

<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>Imm</td>
<td>MOVW DP, #4000h</td>
<td>4000</td>
<td>100100</td>
<td>100180</td>
<td></td>
</tr>
<tr>
<td>Imm</td>
<td>MOVL XAR1, #100100h</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Imm</td>
<td>MOVL XAR2, #100180h</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dir</td>
<td>MOV AL, @31h</td>
<td>120</td>
<td>100101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Idr</td>
<td>ADD AL, *XAR1++</td>
<td>225</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dir</td>
<td>SUB AL, @30h</td>
<td>200</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Idr</td>
<td>ADD AL, *XAR1++</td>
<td>260</td>
<td>100102</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Imm</td>
<td>MOVL DP, #4006h</td>
<td>4006</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dir</td>
<td>ADD AL, @1</td>
<td>290</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Idr</td>
<td>ADD AL, *XAR2</td>
<td>370</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Idr</td>
<td>SUB AL, *+XAR2 [1]</td>
<td>340</td>
<td>100180</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Imm</td>
<td>ADD AL, #32</td>
<td>360</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Imm: Immediate;  
Dir: Direct;  
Reg: Register;  
Idr: Indirect

Solutions
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

- Perform simple program control using branch and conditional codes
- Write C28x code to perform basic arithmetic
- Use the multiplier to implement sum-of-products equations
- Use the RPT instruction (repeat) to optimize loops
- Use MAC for long sum-of-products
- Efficiently transfer the contents of one area of memory to another
- Examine read-modify-write operations
Module Topics

Appendix C – Assembly Programming

Module Topics ......................................................................................................................................... C-1

Program Control ..................................................................................................................................... C-2
  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

![Branch Types and Range Diagram]

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>NEQ</td>
<td>TC</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>NEQ</td>
<td>4/2</td>
</tr>
</tbody>
</table>

### Condition Code

<table>
<thead>
<tr>
<th>NEQ</th>
<th>LT</th>
<th>EQ</th>
<th>LEQ</th>
<th>GT</th>
<th>HI</th>
<th>NOV</th>
<th>HIS (C)</th>
<th>OV</th>
<th>NBIO</th>
</tr>
</thead>
<tbody>
<tr>
<td>LO</td>
<td>NC</td>
<td>TC</td>
<td></td>
<td>NC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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></td>
<td></td>
<td>IRET</td>
<td>8</td>
</tr>
</tbody>
</table>

- More Call variations in the user guide are for code backward compatibility

Stack

- Local Var
- Var
- 22-bit old
- RPC

RPC

- Old RPC
- New RPC
- Ret Addr
- Func
- Ret Addr
- LCR
- Func
- LRETR

PC

- 22-bit old
- RPC
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

\[ y = \sum_{n=0}^{4} x_n \]

Auxiliary register used as loop counter
Branch if Auxiliary Register not zero
Test performed on lower 16-bits of XARx only

```
len .set 5
x .usect "samp",6
y .set (x+len)

.sect "code"
MOVL XAR2,#x
MOV AR3,#len-2
MOV AL,*XAR2++
sum: ADD AL,*XAR2++
BANZ sum,AR3--
MOV *(0:y),AL
```

Data

<table>
<thead>
<tr>
<th>x0</th>
<th>x1</th>
<th>x2</th>
<th>x3</th>
<th>x4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

XAR2

Y

AR3

COUNT

C2000 Piccolo Workshop - Appendix C - Assembly Programming
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 from 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>Description</th>
</tr>
</thead>
</table>
| xxx Ax, #16b ;word | xxx = instruction: MOV, ADD, SUB, ...
| xxxB Ax, #8b ;byte | Ax = AH, or AL
| xxxL ACC, #32b ;long | Assembler will automatically convert to 1 word instruction.

**Example**

- MOV Ax, loc16
- ADD Ax, loc16
- SUB Ax, loc16
- AND Ax, loc16
- OR Ax, loc16
- XOR Ax, loc16
- AND Ax, loc16,#16b
- NOT Ax
- NEG Ax
- MOV loc16, Ax

**Ax = AH or AL Operations**

| MOV Ax, loc16 |
| ADD Ax, loc16 |
| SUB Ax, loc16 |
| AND Ax, loc16 |
| OR Ax, loc16 |
| XOR Ax, loc16 |
| AND Ax, loc16,#16b |
| NOT Ax |
| NEG Ax |
| MOV loc16, Ax |

**Shift the Accumulator**

**Shift full ACC**

| LSL ACC <<shift | 31 ....... 0 |
| SFR ACC >>shift | C ——— ACC ——— 0 |
| LSL ACC <<T | 31 ....... 0 |
| SFR ACC >>T | SXM ——— ACC ——— C |

**Shift AL or AH**

| LSL AX <<shift | 15 ....... 0 |
| LSR AX <<shift | C ——— Ax ——— 0 |
| ASR AX >>shift | 15 ....... 0 |
| LSL AX <<T | SXM ——— Ax ——— C |
| LSR AX <<T | ASR |
| ASR AX >>T | LSR |
32 Bit Shift Operations [ACC]

Logical Shift Left – Long: LSLL

Logical Shift Right – Long: LSRL

Arithmetic Shift Right – Long: ASRL

Note: T(4:0) are used; other bits are ignored

Examples:
- LSLL ACC, T
- LSRL ACC, T
- ASRL ACC, T
Digital signal processors require many multiply and add math intensive operations. The single cycle multiplier is the second major component in the execution unit. The C28x has the traditional 16-bit-by-16-bit multiplier as previous TI DSP families. In addition, the C28x has a single cycle 32-bit-by-32-bit multiplier to perform extended precision math operations. The large multiplier allows the C28x to support higher performance control systems requirement while maintaining small or reduce code.

The following slides introduce instructions that use the 16-bit-by-16-bit multiplier and multiply and add (MAC) operations. The 32-bit-by-32-bit multiplication will be covered in the appendix.
Basic Multiplier

Multiplier Instructions

<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\textsuperscript{th} 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\textsuperscript{st} product&lt;&lt;PM to ACC</td>
</tr>
<tr>
<td>ADD ACC,P</td>
<td>ACC += P</td>
<td>Add n\textsuperscript{th} product&lt;&lt;PM to ACC</td>
</tr>
<tr>
<td>SUB ACC,P</td>
<td>ACC -= P</td>
<td>Sub n\textsuperscript{th} product&lt;&lt;PM fr. ACC</td>
</tr>
</tbody>
</table>

Instruction | Execution         | Purpose                      |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MPYA P,T,#16b</td>
<td>ACC += P&lt;&lt;PM then P = T*#16b</td>
<td></td>
</tr>
<tr>
<td>MPYA P,T,loc16</td>
<td>ACC += P&lt;&lt;PM then P = T*loc16</td>
<td></td>
</tr>
<tr>
<td>MPYS P,T,loc16</td>
<td>ACC -= P&lt;&lt;PM then P = T*loc16</td>
<td></td>
</tr>
</tbody>
</table>

Sum-of-Products

\[ Y = A\times X_1 + B\times X_2 + C\times X_3 + D\times X_4 \]

```
ZAPA ;ACC = P = OVC = 0
MOV T,@X1 ;T = X1
MPY P,T,@A ;P = A\times X_1
MOVA T,@X2 ;T = X_2 ;ACC = A\times X_1
MPY P,T,@B ;P = B\times X_2
MOVA T,@X3 ;T = X_3 ;ACC = A\times X_1 + B\times X_2
MPY P,T,@C ;P = C\times X_3
MOVA T,@X4 ;T = X_4 ;ACC = A\times X_1 + B\times X_2 + C\times X_3
MPY P,T,@D ;P = D\times X_4
ADDL ACC,P<<PM ;ACC = Y
MOVL @y,ACC
```
32x32 Long Multiplication

\[
\begin{array}{c|c|c}
\hline
X & Y & \multicolumn{1}{c}{X0 \times Y0} \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c}
\hline
Y1 & X1 & \multicolumn{1}{c}{Y1 \times X1} \\
\hline
Z3 & Z2 & Z1 \times Z0 \\
\hline
\end{array}
\]

\[
\begin{array}{c|c}
Accumulator & P-register \\
\hline
\end{array}
\]

---

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:**
```
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
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**

![MAC Instruction Diagram]

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**
- **MOVA T,loc16**
- **MPY P,T,loc16**
- **MAC**
  - 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

Example

<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

\[
\text{If } A < B, \text{ Then } B = A
\]

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>Data Memory</th>
<th>Data Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 2 0</td>
<td>0 1 2 0 A</td>
<td>0 1 2 0 A</td>
</tr>
<tr>
<td></td>
<td>0 3 2 0 B</td>
<td>0 1 2 0 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): 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 B @VarA,#0,GT ; Zero VarA if greater then
Logical Operations

Byte Operations and Addressing

**Byte Operations**

<table>
<thead>
<tr>
<th>MOVB</th>
<th>AX.LSB,loc16</th>
<th>0000 0000</th>
<th>Byte</th>
<th>AX</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVB</td>
<td>AX.MSB,loc16</td>
<td>Byte</td>
<td>No change</td>
<td>AX</td>
</tr>
<tr>
<td>MOVB</td>
<td>loc16, AX.LSB</td>
<td>No change</td>
<td>Byte</td>
<td>loc16</td>
</tr>
<tr>
<td>MOVB</td>
<td>loc16, AX.MSB</td>
<td>No change</td>
<td>Byte</td>
<td>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]  
Odd Offset | Even Offset | loc16

**Byte Addressing**

<table>
<thead>
<tr>
<th>AH.MSB</th>
<th>AH.LSB</th>
<th>AL.MSB</th>
<th>AL.LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>34</td>
<td>56</td>
<td>78</td>
</tr>
</tbody>
</table>

16 bit memory

Example of Byte Un-Packing

- MOVL XAR2, #MemA
- MOVB *+XAR2[1], AL.LSB
- MOVB *+XAR2[2], AL.MSB
- MOVB *+XAR2[5], AH.LSB
- MOVB *+XAR2[6], AH.MSB

Example of Byte Packing

- MOVL XAR2, #MemA
- MOVB AL.LSB,*+XAR2[1]
- MOVB AL.MSB,*+XAR2[2]
- MOVB AH.LSB,*+XAR2[4]
- MOVB AH.MSB,*+XAR2[7]
Test and Change Memory Instructions

The compare (CMPx) and test (Txxx) instructions allow the ability to test values in memory. The results of these operations can then trigger subsequent conditional branches. The CMPx instruction allows comparison of memory with respect to a specified constant value, while the Txxx instructions allow any single bit to be extracted to the test control (TC) field of status register 0. The contents of the accumulator can also be non-destructively analyzed to establish branching conditions, as seen below.

<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>
</table>
| MAX ACC,loc16 | if ACC < loc16, ACC = loc16  
if ACC >= loc16, do nothing |
| MIN ACC,loc16 | if ACC > loc16, ACC = loc16  
if ACC <= loc16, do nothing |
| MAXL ACC,loc32 | if ACC < loc32, ACC = loc32  
if ACC >= loc32, do nothing |
| MINL ACC,loc32 | if ACC > loc32, ACC = loc32  
if ACC <= loc32, do nothing |
| MAXCUL P,loc32 (for 64 bit math) | if P < loc32, P = loc32  
if P >= loc32, do nothing |
| MINCUL P,loc32 (for 64 bit math) | if P > loc32, P = loc32  
if P <= loc32, do nothing |

Find the maximum 32-bit number in a table:

```
MOVL ACC,#0
MOVL XAR1,#table
RPT #(table_length - 1)
|| MAXL ACC,*XAR1++
```
Read Modify Write Operations

The accumulator (ACC) is the main working register for the C28x. It is the destination of all ALU operations except those, which operate directly on memory or registers. The accumulator supports single-cycle move, add, subtract and compare operations from 32-bit-wide data memory. It can also accept the 32-bit result of a multiplication operation. These one or two cycle operations are referred to as read-modify-write operations, or as atomic instructions.

Read-Modify-Write Instructions

- **Work directly on memory – bypass ACC**
- **Atomic Operations – protected from interrupts**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND loc16,AX</td>
<td></td>
<td>AH, AL</td>
</tr>
<tr>
<td>OR loc16,AX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XOR loc16,AX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD loc16,AX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB loc16,AX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBR loc16,AX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INC loc16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC loc16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND loc16,#16b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR loc16,#16b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XOR loc16,#16b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD loc16,#16b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB loc16,#16b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBR loc16,#16b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSET loc16,#bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TCLR loc16,#bit</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## 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, @VarB</td>
<td>MOV AL, @VarB</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>
<tr>
<td>MOV AL, @VarB</td>
<td>ADD @VarA, #100</td>
<td>INC @VarA</td>
</tr>
<tr>
<td>ADD @VarA, AL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Benefits of Read-Modify-Write Instructions**
Lab C: Assembly Programming

Note: The lab linker command file is based on the F28035 memory map – modify as needed, if using a different F28xx device memory map.

- **Objective**

  The objective of this lab is to practice and verify the mechanics of performing assembly language programming arithmetic on the TMS320C28x. In this exercise, we will expand upon the .asm file from the previous lab to include new functions. Code will be added to obtain the sum of the products of the values from each array.

  Perform the sum of products using a MAC-based implementation. In a real system application, the coeff array may well be constant (values do not change), therefore one can modify the initialization routine to skip the transfer of this arrays, thus reducing the amount of data RAM and cycles required for initialization. Also there is no need to copy the zero to clear the result location. The initialization routine from the previous lab using the load/store operation will be replaced with a looped BANZ implementation.

  As in previous lab, consider which addressing modes are optimal for the tasks to be performed. You may perform the lab based on this information alone, or may refer to the following procedure.

- **Procedure**

**Copy Files, Create Project File**

1. Create a new project called LabC.pjt in C:\C28x\Labs\Appendix\LabC and add LabC.asm and Lab.cmd to it. Check your file list to make sure all the files are there. Be sure to setup the Build Options by clicking: Project ➔ Build Options on the menu bar. Select the Linker tab. In the middle of the screen select “No Autoinitialization” under “Autoinit Model:”. Enter start in the “Code Entry Point (-e):” field. Next, select the Compiler tab. Note that “Full Symbolic Debug (-g)” under “Generate Debug Info:” is selected. Then select OK to save the Build Options.

**Initialization Routine using BANZ**

2. Edit LabC.asm and modify it by replacing the initialization routine using the load/store operation with a BANZ process. Remember, it is only necessary to copy the first four values (i.e. initialize the data array). Do you still need the coeff array in the vars section?

3. Save your work. If you would like, you can use Code Composer Studio to verify the correct operation of the block initialization before moving to the next step.
Sum of Products using a RPT/MAC-based Implementation

4. Edit LabC.asm to add a RPT/MAC-based implementation to multiply the coeff array by the data array and storing the final sum-of-product value to result.

Build and Load

5. Click the “Build” button and watch the tools run in the build window. Debug as necessary. To open up more space, close any open files or windows that you do not need.

6. If the “Load program after build” option was not selected in Code Composer Studio, load the output file onto the target. Click: File → Load Program...

   If you wish, right click on the source window and select Mixed Mode to debug using both source and assembly.

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

Optional Exercise

After completing the above, edit LabC.asm and modify it to perform the initialization process using a RTP/PREAD rather than a load/store/BANZ.

End of Exercise
OPTIONAL Lab C-C: Sum-of-Products in C

Note: The lab linker command file is based on the F28035 memory map – modify as needed, if using a different F28xx device memory map.

Objective

The objective of this lab is to practice and verify the mechanics of performing C programming arithmetic on the TMS320C28x. The objective will be to add the code necessary to obtain the sum of the products of the n-th values from each array.

Procedure

Create Project File

1. In Code Composer Studio create a new project called LabC-C.pjt in C:\C28x\Labs\Appendix\LabC\LabC-C and add LabC-C.c and Lab.cmd to it. Check your file list to make sure all the files are there. Open the Build Options and select the Linker tab. Select the “Libraries” Category and enter rts2800_ml.lib in the “Incl. Libraries (-l):” box. Do not setup any other Build Options. The default values will be used. In Appendix Lab D exercise, we will experiment and explore the various build options when working with C.

Sum of Products using a MAC-based Implementation

2. Edit LabC-C.c and modify the “main” routine to perform a MAC-based implementation in C. Since the MAC operation requires one array to be in program memory, the initialization routine can skip the transfer of one of the arrays, thus reducing the amount of data RAM and cycles required for initialization.

Build and Load

3. Click the “Build” button and watch the tools run in the build window. Debug as necessary.

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

4. Under Debug on the menu bar click “Go Main”. Single-step your routine. While single-stepping, open memory windows to see the values located in table [9] and data [9]. (Note: data[9] consists of the allocated arrays of data, coeff, and result). Open the CPU Registers. Check to see if the program is working as expected. Debug and modify, if needed.

End of Exercise
Appendix D – C Programming

Introduction

The C28x architecture, hardware, and compiler have been designed to efficiently support C code programming.

Appendix D will focus on how to program in C for an embedded system. Issues related to programming in C and how C behaves in the C28x environment will be discussed. Also, the C compiler optimization features will be explained.

Learning Objectives

- 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

Linking Boot code from RTS2800.lib

Set up the Stack

C28x Data Types

Accessing Interrupts / Status Register

Using Embedded Assembly

Using Pragma

Optimization Levels

Volatile Usage

Compiler Advanced Options

Optimization Tips Summary

Lab D: C Optimization

OPTIONAL Lab D2: C Callable Assembly

Solutions
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.

Setting Up the Stack

- Boot.asm sets up SP to point at .stack
- The .stack section has to be linked into the low 64k of data memory. The SP is a 16-bit register and cannot access addresses beyond 64K.
- Stack size is set by the linker. The linker creates a global symbol, --STACK-SIZE, and assigns it a value equal to the size of the stack in bytes. (default 1K words)
- You can change stack size at link time by using the -stack linker command option.

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 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.
Accessing Interrupts / Status Register

```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:

  ```c
  asm ( "CLRC  INTM ; enable global interrupt");
  #define EINT asm ( "CLRC  INTM")
  ```

- Note: first column after leading quote is label field - if no label, should be blank space.
- Avoid modifying registers used by C
- Lengthy code should be written in ASM and called from C
  - main C file retains portability
  - yields more easily maintained structures
  - eliminates risk of interfering with registers in use by C

The assembly function allows for C files to contain 28x assembly code. Care should be taken not to modify registers in use by C, and to consider the label field with the assembly function. Also, any significant amounts of assembly code should be written in an assembly file and called from C.

There are two examples in this slide – the first one shows how to embed a single assembly language instruction into the C code flow. The second example shows how to define a C term that will invoke the assembly language instruction.
Using Pragma

Pragma is a preprocessor directive that provides directions to the compiler about how to treat a particular statement. The following example shows how the DATA_SECTION pragma is used to put a specific buffer into a different section of RAM than other buffers.

The example shows two buffers, bufferA and bufferB. The first buffer, bufferA is treated normally by the C compiler by placing the buffer (512 words) into the ".bss" section. The second, bufferB is specifically directed to go into the “my_sect” portion of data memory. Global variables, normally ".bss", can be redirected as desired.

When using CODE_SECTION, code that is normally linked as ".text", can be identified otherwise by using the code section pragma (like .sect in assembly).

Pragma Examples

◆ User defined sections from C:

```c
#pragma CODE_SECTION (func, "section name")
#pragma DATA_SECTION (symbol, "section name")
```

◆ Example - using the DATA_SECTION Pragma

◆ C source file

```c
char bufferA[512];
#pragma DATA_SECTION(bufferB, "my_sect")
char bufferB[512];
```

◆ Resulting assembly file

```assembly
.global _bufferA, _bufferB
.bss _bufferA, 512
_bufferB: .usect "my_sect", 512
```

More #pragma are defined in the C compiler UG
Optimization Levels

Optimization Levels

Optimization Scope

FILE1.C

{  
  {  
    SESE  
  }  
  {  
    SESE: Single Entry, Single Exit  
  }  
  {  
    . .  
  }  
}

FILE2.C

{  
  . .  
}

-00, -01  -02  -03 -pm -03

LOCAL single block  
FUNCTION across blocks  
FILE across functions  
PROGRAM across files

Optimizations fall into 4 categories. This is also a methodology that should be used to invoke the optimizations. It is recommended that optimization be invoked in steps, and that code be verified before advancing to the next step. Intermediate steps offer the gradual transition from fully symbolic to fully optimized compilation. Compiler switched may be invoked in a variety of ways.

Here are 4 steps that could be considered:

1st: use –g
By starting out with –g, you do no optimization at all and keep symbols for debug.

2nd: use –g –o3
The option –o3 might be too big a jump, but it adds the optimizer and keeps symbols.

3rd: use –g –o3 –mn
This is a full optimization, but keeps some symbols.

4th: use –o3
Full optimization, symbols are not kept.
## Optimization Levels

<table>
<thead>
<tr>
<th>LOCAL</th>
<th>FUNCTION</th>
<th>FILE</th>
<th>PROGRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>–o0</td>
<td>Performs control-flow-graph simplification&lt;br&gt;Allocates variables to registers&lt;br&gt;Performs loop rotation&lt;br&gt;Eliminates unused code&lt;br&gt;Simplifies expressions and statements&lt;br&gt;Expands calls to functions declared inline</td>
<td>–o1</td>
<td>Performs local copy/constant propagation&lt;br&gt;Removes unused assignments&lt;br&gt;Eliminates local common expressions</td>
</tr>
<tr>
<td>–o2</td>
<td>Default (-o)&lt;br&gt;Performs loop optimizations&lt;br&gt;Eliminates global common sub-expressions&lt;br&gt;Eliminates global unused assignments</td>
<td>–o3</td>
<td>Removes all functions that are never called&lt;br&gt;Simplifies functions with return values that are never used&lt;br&gt;Inlines calls to small functions&lt;br&gt;Identifies file-level variable characteristics</td>
</tr>
</tbody>
</table>

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.
## Optimization Levels

### Volatile Usage

<table>
<thead>
<tr>
<th>Optimization Issue: “Volatile” Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Problem:</strong> 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!)</td>
</tr>
</tbody>
</table>

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 it out of existence.

![Flowchart showing the decision process for whether to optimize or not based on the `volatile` declaration.](image)
Compiler Advanced Options

To get to these options, go to Project ➔ Build Options in Code Composer Studio.

In the category, pick Advanced.

The first thing to notice under advanced options is the Auto Inlining Threshold.
- Used with –o3 option
- Functions > size are not auto inlined

Note: To prevent code size increases when using –o3, disable auto inlining with -oi0

The next point we will cover is the Normal Optimization with Debug (-mn).
- Re-enables optimizations disabled by “–g” option (symbolic debug)
- Used for maximum optimization

Note: Some symbolic debug labels will be lost when –mn option is used.

Optimizer should be invoked incrementally:

- `-g test` Symbols kept for debug
- `-g -o3 test` Add optimizer, keep symbols
- `-g -o3 -mn test` More optimize, some symbols
- `-o3 test` Final rev: Full optimize, no symbols

- `-mf` : Optimize for speed instead of the default optimization for code size

- `-mi` : Avoid RPT instruction. Prevent compiler from generating RPT instruction. RPT instruction is not interruptible

- `-mt` : Unified memory model. Use this switch with the unified memory map of the 281x & 280x. Allows compiler to generate the following:
  - RPT PREAD for memory copy routines or structure assignments
  - MAC instructions
  - Improves efficiency of switch tables
Optimization Levels

Optimization Tips Summary

Summary: Optimization Tips

- **Within C functions:**
  - Use const with variables for parameter constants
  - Minimize mixing signed & unsigned ops: SXM changes
  - Keep frames ≤ 64 (locals + parameters + PC): *-SP[6bit]*
  - Use structures ≤ 8 words: use 3 bit index mode
  - Declare longs first, then declare ints: minimize stack holes
  - Avoid: long = (int * int): yields unpredictable results

- **Optimizing:** Use -o0, -o1, -o2, -o3 when compiling
  - Inline short/key functions
  - Pass inlines between files: static inlines in header files
  - Invoke automatic inlining: -o3 -oi
  - Give compiler project visibility: use -pm and -o3

- **Tune memory map via linker command file**

- **Re-write key code segments to use intrinsics or in assembly**

App notes 3rd Parties

The list above documents the steps that can be taken to achieve increasingly higher coding efficiency. It is recommended that users first get their code to work with no optimization, and then add optimizations until the required performance is obtained.
Lab D: C Optimization

Note: The lab linker command file is based on the F28035 memory map – modify as needed, if using a different F28xx device memory map.

Objective

The objective of this lab is to practice and verify the mechanics of optimizing C programs. Using Code Composer Studio profile capabilities, different routines in a project will be benchmarked. This will allow you to analyze the performance of different functions. This lab will highlight the profiler and the clock tools in CCS.

Procedure

Create Project File

1. Create a new project in `C:\C28x\Labs\Appendix\LabD` called `LabD.pjt` and add `LabD.c`, `Lab.cmd`, and `sop-c.c` to it. (Note that `sop-asm.asm` will be used in the next part of the lab, and should not be added now).

2. Setup the Build Options. Select the Linker tab and notice that “Run-time Autoinitilization” under “Autoinit Model:” is selected. Do not enter anything in the “Code Entry Point (-e):” field (leave it blank). Set the stack size to 0x200. In the Linker options select the “Libraries” Category and enter `rts2800_ml.lib` in the “Incl. Libraries (-l):” box. Next, select the Compiler tab. Note that “Full Symbolic Debug (-g)” under “Generate Debug Info:” in the Basic Category is selected. On the Feedback Category pull down the interlisting options and select “C and ASM (-ss)”. On the Assembly Category check the Keep generated .asm Files (-k), Keep Labels as Symbols (-as) and Generate Assembly Listing Files (-al). The –as will allow you to see symbols in the memory window and the –al will generate an assembly listing file (.lst file). The listing file has limited uses, but is sometime helpful to view opcode values and instruction sizes. (The .lst file can be viewed with the editor). Both of these options will help with debugging. Then select OK to save the Build Options.

Build and Load

3. Click the “Build” button and watch the tools run in the build window. Be sure the “Load program after build” option is selected in Code Composer Studio. The output file should automatically load. The Program Counter should be pointing to `_c_int00` in the Disassembly Window.

Set Up the Profile Session

4. Restart the DSP (debug → restart) and then “Go Main”. This will run through the C initialization routine in `Boot.asm` and stop at the main routine in `LabD.c`.
5. Set a breakpoint on the NOP in the while(1) loop at the end of main() in LabD.c.

6. Set up the profile session by selecting Profiler → Start New Session. Enter a session name of your choice (i.e. LabD).

7. In the profiler window, hover the mouse over the icons on the left region of the window and select the icon for Profile All Functions. Click on the “+” to expand the functions. Record the “Code Size” of the function sop C code in the table at the end of this lab. Note: If you do not see a “+” beside the .out file, press “Profile All Functions” on the horizontal tool bar. (You can close the build window to make the profiler window easier to view by right clicking on the build window and selecting “hide”).

8. Select F5 or the run icon. Observe the values present in the profiling window. What do the numbers mean? Click on each tab to determine what each displays.

**Benchmarking Code**

9. Let’s benchmark (i.e. count the cycles need by) only a portion of the code. This requires you to set a breakpoint pair on the starting and ending points of the benchmark. Open the file sop-c.c and set a breakpoint on the “for” statement and the “return” statement.

10. In CCS, select Profile → Setup. Check “Profile all Functions and Loops for Total Cycles” and click “Enable Profiling”. Then select Profile → viewer.

11. Now “Restart” the program and then “Run” the program. The program should be stopped at the first breakpoint in sop. Double click on the clock window to set the clock to zero. Now you are ready to benchmark the code. “Run” to the second breakpoint. The number of cycles are displayed in the viewer window. Record this value in the table at the end of the lab under “C Code - Cycles”.

**C Optimization**

12. To optimize C code to the highest level, we must set up new Build Options for our Project. Select the Compiler tab. In the Basic Category Panel, under “Opt Level” select File (-o3). Then select OK to save the Build Options.

13. Now “Rebuild” the program and then “Run” the program. The program should be stopped at the first breakpoint in sop. Double click on the clock window to set the clock to zero. Now you are ready to benchmark the code. “Run” to the second breakpoint. The number of cycles are displayed in the clock window. Record this value in the table at the end of the lab under “Optimized C (-o3) - Cycles”.

14. Look in your profile window at the code size of sop. Record this value in the table at the end of this lab.

**Benchmarking Assembly Code**

15. Remove sop-c.c from your project and replace it with sop-asm.asm. Rebuild and set breakpoints at the beginning and end of the assembly code (MOVL & LRETR).
16. Start a new profile session and set it to profile all functions. Run to the first breakpoint and study the profiler window. Record the code size of the assembly code in the table.

17. Double Click on the clock to reset it. Run to the last breakpoint. Record the number of cycles the assembly code ran.

18. How does assembly, C code, and optimized C code compare on the C28x?

<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

Note: The lab linker command file is based on the F28035 memory map – modify as needed, if using a different F28xx device memory map.

Objective

The objective of this lab is to practice and verify the mechanics of implementing a C callable assembly programming. In this lab, a C file will be used to call the sum-of-products (from the previous Appendix LabC exercise) by the “main” routine. Additionally, we will learn how to use Code Composer Studio to configure the C build options and add the run-time support library to the project. As in previous labs, you may perform the lab based on this information alone, or may refer to the following procedure.

Procedure

Copy Files, Create Project File

1. Create a new project in C:\C28x\Labs\Appendix\LabD2 called LabD2.pjt and add LabD2.c, Lab.cmd, and sop-c.c to it.

2. Do not add LabC.asm to the project (copy of file from Appendix Lab C). It is only placed here for easy access. Parts of this file will be used later during this lab exercise.

3. Setup the Build Options. Select the Linker tab and notice that “Run-time Autoinitilization” under “Autoinit Model:”is selected. Do not enter anything in the “Code Entry Point (-e):” field (leave it blank). Set the stack size to 0x200. In the Linker options select the “Libraries” Category and enter rts2800_ml.lib in the “Incl. Libraries (-l):” box. Next, select the Compiler tab. Note that “Full Symbolic Debug (-g)” under “Generate Debug Info:” in the Basic Category is selected. On the Feedback Category pull down the interlisting options and select “C and ASM (-ss)”. On the Assembly Category check the Keep generated .asm Files (-k), Keep Labels as Symbols (-as) and Generate Assembly Listing Files (-al). The –as will allow you to see symbols in the memory window and the –al will generate an assembly listing file (.lst file). The listing file has limited uses, but is sometime helpful to view opcode values and instruction sizes. (The .lst file can be viewed with the editor). Both of these options will help with debugging. Then select OK to save the Build Options.

Build and Load

4. Click the “Build” button and watch the tools run in the build window. Be sure the “Load program after build” option is selected in Code Composer Studio. The output file should automatically load. The Program Counter should be pointing to _c_int00 in the Disassembly Window.

5. Under Debug on the menu bar click “Go Main”. This will run through the C initialization routine in Boot.asm and stop at the main routine in LabD2.c.
Verify C Sum of Products Routine

6. Debug using both source and assembly (by right clicking on the window and select Mixed Mode or using View → Mixed Source/ASM).

7. Open a memory window to view result and data.

8. Single-step through the C code to verify that the C sum-of-products routine produces the results as your assembly version.

Viewing Interlisted Files and Creating Assembly File

9. Using File → Open view the LabD2.asm and sop-c.asm generated files. The compiler adds many items to the generated assembly file, most are not needed in the C-callable assembly file. Some of the unneeded items are .func / .endfunc .sym, and .line.

10. Look for the _sop function that is generated by the compiler. This code is the basis for the C-callable assembly routine that is developed in this lab. Notice the comments generated by the compiler on which registers are used for passing parameters. Also, notice the C code is kept as comments in the interlisted file.

11. Create a new file (File → New, or clicking on the left most button on the horizontal toolbar “New”) and save it as an assembly source file with the name sop-asm.asm. Next copy ONLY the sum of products function from LabC.asm into this file. Add a _sop label to the function and make it visible to the linker (.def). Also, be sure to add a .sect directive to place this code in the “code” section. Finally, add the following instruction to the end:

   LRETR ; return statement

12. Next, we need to add code to initialize the sum-of-products parameters properly, based on the passed parameters. Add the following code to the first few lines after entering the _sop routine: (Note that the two pointers are passed in AR4 and AR5, but one needs to be placed in AR7. The loop counter is the third argument, and it is passed in the accumulator.)

   MOVL XAR7,XAR5 ;XAR7 points to coeff [0]
   MOV AR5,AL ;move n from ACC to AR5 (loop counter)
   SUBB XAR5,#1 ;subtract 1 to make loop counter = n-1

   Before beginning the MAC loop, add statements to set the sign extension mode, set the SPM to zero, and a ZAPA instruction. Use the same MAC statement as in Lab 4, but use XAR4 in place of XAR2. Make the repeat statement use the passed value of n-1 (i.e. AR5).

   RPT AR5 ;repeat next instruction AR5 times
Now we need to return the result. To return a value to the calling routine you will need to place your 32-bit value in the ACC. What register is the result currently in? Adjust your code, if necessary.

13. Save the assembly file as `sop-asm.asm` (Do not name it `LabD2.asm` because the compiler has already created with that name from the original `LabD2.c` code).

**Defining the Function Prototype as External**

14. Note in `LabD2.c` an “extern” modifier is placed in front of the sum-of-products function prototype:

```
extern int sop(int*,int*,int); //sop function prototype
```

**Verify Assembly Sum of Products Routine**

15. Remove the `sop-c.c` file from the project and add the new `sop-asm.asm` assembly file to the project.

16. Rebuild and verify that the new assembly sum-of-products routine produces the same results as the C function.

**End of Exercise**
## Solutions

### Lab D Solutions

<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>
Appendix E – Control Law Accelerator

Introduction

Appendix E discusses the details of the Piccolo™ TMS320F2803x Control Law Accelerator (CLA). The floating-point number format and the CLA registers will be discussed. Details of the CLA instruction set and pipeline will be explained. Additionally, system configuration and a comparison to the Delfino™ floating-point unit (FPU) will be covered.

Learning Objectives

- Floating-point format
- CLA registers and execution flow
- Instructions
- Pipeline
- System configuration
- Summary
Module Topics

Appendix E – Control Law Accelerator

Module Topics

Control Law Accelerator
Floating-Point Format
CLA Registers and Execution Flow
CLA Instructions
Status Register and Pipeline
CLA System Configuration
CLA Compared to C28x+FPU
Summary
Control Law Accelerator

Floating-Point Format

### IEEE Single-Precision Floating-Point Format

1 Sign Bit (0 = Positive, 1 = Negative)  
8-bit Exponent (Biased)  
23-bit Mantissa (Implicit Leading Bit + Fraction Bits)

<table>
<thead>
<tr>
<th>S</th>
<th>E</th>
<th>M</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Non-Zero</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1-254</td>
<td>0-0x7FFFFF</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>255 (max)</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>255 (max)</td>
<td>Non-Zero</td>
</tr>
</tbody>
</table>

* Normal Positive and Negative Values are Calculated as:

\[
(-1)^s \times 2^{(E-127)} \times 1.M
\]

\[+/- \, 1.7 \times 10^{-38} \text{ to } +/- \, 3.4 \times 10^{+38}\]

The Normalized IEEE numbers have a hidden 1; thus the equivalent signed integer resolution is the number of mantissa bits + sign + 1

---

### IEEE Single-Precision Floating-Point Format (IEEE 754)

- Most widely used standard for floating-point
  - Standard number formats, Special values (NaN, Infinity)
  - Rounding modes & floating-point operations
  - Used on many CPUs

- Simplifications for the C28x floating-point unit
  - Flags & Compare Operations:
    - Negative zero is treated as positive zero
    - Denormalized values are treated as zero
    - Not-a-number (NaN) is treated as infinity
    - Round-to-Zero Mode Supported (truncate)
    - Round-to-Nearest Mode Supported (even)

- These formats are commonly handled this way on embedded processors
CLA Registers and Execution Flow

CLA Register Set

**Interrupt/Task Control**
- MIFR: Flag
- MIFR: Clear
- MIOVF: Overflow flag
- MICLROVF: Overflow clear

**Configuration and Control**
- MEMCFG: Memory config
- MCTL: CLA control

**Eight Interrupt (Task) Vectors**
- MVECT1 to MVECT8
  - Offset from the start of CLA Program Memory to the beginning of the task

**Four 32-bit Result Registers**
- MR0 – MR3

**Two 16-bit Auxiliary Registers**
- MAR0, MAR1
  - Used for indirect addressing

CLA Configuration Registers:
- CSM and EALLOW Protected
  - Main CPU has Read and Write Access

CLA Configuration Registers:
- CSM and EALLOW Protected
  - Main CPU has Read and Write Access

CLA Execution Registers:
- CSR Protected
  - Main CPU has Read Only Access

CLA Configuration Registers:
- CSM and EALLOW Protected
  - Main CPU has Read and Write Access

CLA Configuration Registers:
- CSM and EALLOW Protected
  - Main CPU has Read and Write Access

CLA Execution Registers:
- CSR Protected
  - Main CPU has Read Only Access

CLA Execution Flow

Task request is via software or interrupt assigned in MPRSCSEL1:
- Task1: ADCINT1 or EPWM1_INT
- Task2: ADCINT2 or EPWM2_INT
- Task7: ADCINT7 or EPWM7_INT
- Task8: ADCINT8 or CPU Timer 0

Set MIOVF Bit (Overflow Flagged)
- Yes
- No

Set MIFR Bit (Task Pending)
- Yes
- No

Clear MIFR.x bit
- Yes
- No

Set MIRUN.x bit
- Yes
- No

Run CLA
- Yes
- No

End of Task? MSTOP
- Yes
- No

Clear MIRUN.x bit
- Yes
- No

Task Enabled? (MIER)
- Yes
- No

Priority Task
- Task1: Highest
- Task2: Lower
- Task7: Lowest
- Task8: Lowest

The task runs to completion (No task nesting)

The main CPU continues code execution in parallel with the CLA

When a task completes a task-specific interrupt is sent to the PIE

Note: Software task requests will not set MIOVF
CLA Instructions

CLA Parallel Instructions

- Parallel bars indicate a parallel instruction
- Parallel instructions operate as a single instruction with a single opcode and performs two operations

- Example: Add + Parallel Store

```
MADDF32 MR3, MR3, MR1
|| MMOV32 @_Var, MR3
```

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply &amp; Parallel Add/Subtract</td>
<td><code>MMVF32 MRa, MRb, MRc</code></td>
<td></td>
</tr>
<tr>
<td>Multiply, Add, Subtract &amp; Parallel Store</td>
<td><code>MADDF32 MRa, MRb, MRc</code></td>
<td></td>
</tr>
<tr>
<td>Multiply, Add, Subtract, MAC &amp; Parallel Load</td>
<td><code>MADDF32 MRa, MRb, MRc</code></td>
<td></td>
</tr>
</tbody>
</table>

Both operations complete in a single cycle

Multiply and Store Parallel Instruction

```
; Before: MR0 = 2.0, MR1 = 3.0, MR2 = 10.0

MMVF32 MR2, MR1, MR0 ; 1/1 instruction
|| MMOV32 @_X, MR2

<any instruction>

; After: MR2 = MR1 * MR0 = 3.0 * 2.0
; @_X = 10.0
```

- Both the math operation and store complete in 1 cycle
- Parallel Instruction:
  - `MMOV32` uses the value of MR2 before the `MMVF32` updates
Status Register and Pipeline

### CLA Status Flags

CLA Status Register MSTF (32-bits)

<table>
<thead>
<tr>
<th>LVF</th>
<th>LUF</th>
<th>Latched Overflow and Underflow</th>
<th>Float math: MMPYF32, MADDF32, 1/x etc. Connected to the PIE for debug</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZF</td>
<td>NF</td>
<td>Negative and Zero</td>
<td>Float move operations to registers. Result of compare, min/max, absolute, negative Integer result of integer operations (MÄND32, MOR32, SÜB32, MLSR32 etc.)</td>
</tr>
<tr>
<td>TF</td>
<td></td>
<td>Test Flag</td>
<td>MTESTTF Instruction</td>
</tr>
<tr>
<td>RNDF32</td>
<td></td>
<td>Rounding Mode</td>
<td>To Zero (truncate) or To Nearest (even)</td>
</tr>
<tr>
<td>MEALLOW</td>
<td></td>
<td>Write Protection</td>
<td>Enable/disable CLA writes to “EALLOW” protected registers</td>
</tr>
<tr>
<td>RPC</td>
<td></td>
<td>Return Program Counter</td>
<td>Call and return: MCNDD, MRCNDD Use store/load MSTF instructions to nest calls</td>
</tr>
</tbody>
</table>

### CLA Pipeline Stages

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Decode</th>
<th>Read</th>
<th>Exe</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>F2</td>
<td>D1</td>
<td>D2</td>
<td>R1</td>
</tr>
</tbody>
</table>

Independent 8 Stage Pipeline

- **Fetch1:** Program read address generated
- **Fetch2:** Read Opcode via CLA program data bus
- **Decode1:** Decode instruction
- **Decode2:** Generate address Conditional branch decision made MAR0/MAR1 update due to indirect addressing post increment
- **Read1:** Data read address via CLA data read address bus
- **Read2:** Read data via CLA data read data bus
- **Execute:** Execute operation MAR0/MAR1 update due to load operations
- **Write:** Write

All Instructions are single cycle (except for Branch/Call/Return) Memory conflicts in F1, R1 and W stall the pipeline
### Write Followed-by-Read

Due to the pipeline order, the read of Reg2 occurs before the Reg1 write. This is only an issue if the location written to can affect the location read. Some peripheral registers write to followed by read from the same location. Insert 3 other instructions or MNOPs to allow the write to occur first.

#### Note:
This behavior is different for the main C28 CPU: The C28x CPU protects write followed by read to the same location. Blocks of peripheral registers have write-followed-by-read protection.

### Loading MAR0 and MAR1

Assume MAR0 is 50 and #_X is 20.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Fetch</th>
<th>Decode</th>
<th>Read</th>
<th>Exe</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMOV16 MAR0, #_X</td>
<td>F1</td>
<td>F2</td>
<td>D1</td>
<td>D2</td>
<td>R1</td>
</tr>
<tr>
<td>MMOV32 MAR1, *MAR0[0]++</td>
<td></td>
<td></td>
<td>D2</td>
<td>R1</td>
<td>R2</td>
</tr>
<tr>
<td>MMOV32 MAR1, *MAR0[0]++</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;Instruction 4&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MMOV32 MAR1, *MAR0[0]++</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

When instruction I1 is in EXE instruction I4 is in D2. If I4 uses MAR0, then a conflict will occur and MAR0 will not be loaded.
Branch, Call, Return Delayed Conditional

CLA Pipeline

<table>
<thead>
<tr>
<th>CLA Pipeline</th>
<th>Fetch</th>
<th>Decode</th>
<th>Read</th>
<th>Exe</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F1</td>
<td>F2</td>
<td>D1</td>
<td>D2</td>
<td>R1</td>
</tr>
</tbody>
</table>

D2: Decide whether or not to branch
EXE: Branch taken (or not)

<Instruction 1> ; I1 Last instruction to affect flags for branch
<Instruction 2> ; I2
<Instruction 3> ; I3
<Instruction 4> ; I4
Branch, CND : MBCNDD, MCCNDD or MRCNDD
<Instruction 5> ; I5
<Instruction 6> ; I6
<Instruction 7> ; I7

Can not be branch or stop *
Do not change flags in time to affect branch

Can not be branch or stop *
Always executed whether branch is taken or not

* Can not be MSTOP (end of task), MDEBUGSTOP (debug halt), MBCNDD (branch), MCCNDD (call), or MRCNDD (return)

Optimizing Delayed Conditional Branch

Use these slots to improve performance

6 instruction slots are executed on every branch

Cycle count varies depending on delay slot usage

<table>
<thead>
<tr>
<th>Taken</th>
<th>Not Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>I1</td>
</tr>
<tr>
<td>I2</td>
<td>I2</td>
</tr>
<tr>
<td>I3</td>
<td>I3</td>
</tr>
<tr>
<td>I4</td>
<td>I4</td>
</tr>
</tbody>
</table>

MSTOP, MDEBUGSTOP, MBCNDD, MCCNDD, MRCNDD are not allowed in delay slots

Optimized Code

MCMPPF32 MR0,#0.1
MNOP
MNOP
MNOP
MCNDD Skip1,NEQ
MNOP
MNOP
MMOV32 MR1,#Ramp1
MMOVXI MR2,#RAMP_MASK
MOR32 MR1,MR2
MMOV32 @Ramp,MR1
...
MSTOP

Skip1: MCMPPF32 MR0,#0.01
MNOP
MNOP
MNOP
MCNDD Skip2,NEQ
MNOP
MNOP
MMOV32 MR1,#Coast1
MMOVXI MR2,#COAST_MASK
MOR32 MR1,MR2
MMOV32 @Coast,MR1
...
MSTOP

Skip1: MMOV32 MR3,#Steady
MMOVXI MR2,#STEADY_MASK
MMOV32 MR3,MR2
MSTOP

Skip2: MMOV32 MR3,#Steady
MMOVXI MR2,#STEADY_MASK
MMOV32 MR3,MR2
MSTOP
CLA System Configuration

Code Partitioning

- CLA and Main CPU communication via shared message RAMs and interrupts
- Main CPU performs communication, diagnostics, I/O in C
- CLA concurrently services time-critical control loops
- System initialization by the main CPU in C
- Access peripheral registers & memory

CLA Run Time Code

C28 Run Time Code

C28 + CLA System Initialization Code

C28 Run Time Code

“Just in Time” ADC Sampling Using CLA

ADC's early interrupt

ADC Sample Window 7 Cycles (minimum)

ADC Conversion

RESULT Register Updates After 15 Cycles

ADC to CLA Interrupt Response Latency 6 Cycles

7 cycles after the early interrupt, the first CLA instruction is in the D2 phase of the pipeline

The ADC early interrupt occurs at the end of the sampling window

The CLA can read the result register as soon as it is latched

Perform pre-calculations using the first 7 instructions (7 cycles)

The 8th instruction is “just-in-time” to read the ADC RESULT register (1 cycle)

Timing shown for 2803x

Minimum CLA Next Task Response 5 cycles

Pre Calc (7 instructions)...
CLA Interrupts Improved Control Loop Timing

Piccolo ADC & CLA interrupt structure enables handling of multi-channel systems with different frequencies and/or phases.

Anatomy of CLA Code

Using a shared C-code header file approach provides easy access to variables and constants in both C28x C and CLA assembly.

// File: C28x_Project.h
#include "DSP2833x_Device.h"
#define PERIOD 100.0
struct PI_CTRL
{
  float KP;
  float KI;
  float I;
  float Ref;
} extern struct PI_CTRL PIVars;
extern Uint32 Cla1Prog_Start;
extern Uint32 Cla1Task1;
extern Uint32 Cla1Task2;

// File: CLAShared.h
#include "DSP2833x_Device.h"
#define PERIOD 100.0
struct PI_CTRL
{
  float KP;
  float KI;
  float I;
  float Ref;
} extern struct PI_CTRL PIVars;
extern Uint32 Cla1Prog_Start;
extern Uint32 Cla1Task1;
extern Uint32 Cla1Task2;

// File main.c
#include "CLAShared.h"
#pragma DATA_SECTION(PIVars,"CpuToCla1MsgRAM");
struct PI_CTRL PIVars;

// Use Symbols defined in the CLA ass file
Cla1Regs.MVECT1 = (Uint16) (sCla1Task1 \sCla1Prog_Start)*sizeof(Uint32);

// Initialize variables
PIVars.KP = 1.234;
PIVars.KI = 0.92367;
PIVars.Ref = 2048.0;
PIVars.I = PIVars.KP*PIVars.Ref;

// Initialize Peripherals:
EpwmRegs.MPO = (Uint16) PERIOD;

Declare shared constants and variables in C
Include DSP2803x_Device.h to define register bit-field structures
Assign variables to message RAMs or CLA data memory sections using DATA_SECTION pragma

Add symbols defined in CLA assembly to make them global and usable in C

// Use Symbols defined in the CLA ass file
Cla1Regs.MVECT1 = (Uint16) (sCla1Task1 \sCla1Prog_Start)*sizeof(Uint32);

// Initialize variables
PIVars.KP = 1.234;
PIVars.KI = 0.92367;
PIVars.Ref = 2048.0;
PIVars.I = PIVars.KP*PIVars.Ref;

// Initialize Peripherals:
EpwmRegs.MPO = (Uint16) PERIOD;
### CLA Compared to C28x+FPU

<table>
<thead>
<tr>
<th>Control Law Accelerator</th>
<th>C28x + Floating-Point Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Independent 8 Stage Pipeline</td>
<td>F1-D2 Shared with the C28x Pipeline</td>
</tr>
<tr>
<td>Single Cycle Math and Conversions</td>
<td>Math and Conversions are 2 Cycle</td>
</tr>
<tr>
<td>No Data Page Pointer; Only uses Direct &amp; Indirect with Post-Increment</td>
<td>Uses C28x Addressing Modes</td>
</tr>
<tr>
<td>4 Result Registers</td>
<td>8 Result Registers</td>
</tr>
<tr>
<td>2 Independent Auxiliary Registers</td>
<td>Shares C28x Auxiliary Registers</td>
</tr>
<tr>
<td>No Stack Pointer or Nested Interrupts</td>
<td>Supports Stack, Nested Interrupts</td>
</tr>
<tr>
<td>Native Delayed Branch, Call &amp; Return</td>
<td>Uses C28x Branch, Call and Return</td>
</tr>
<tr>
<td>Use Delay Slots to Do Extra Work</td>
<td>Copy flags from FPU STF to C28x ST0</td>
</tr>
<tr>
<td>No repeatable instructions</td>
<td>Repeat MACF32 &amp; Repeat Block</td>
</tr>
<tr>
<td>Self-Contained Instruction Set</td>
<td>Instructions Superset on Top of C28x Pass</td>
</tr>
<tr>
<td>Data is Passed Via Message RAMs</td>
<td>Data Between FPU and C28x Regs</td>
</tr>
<tr>
<td>Supports Native Integer Operations: AND, OR, XOR, ADD/SUB, Shift</td>
<td>C28x Integer Operations</td>
</tr>
<tr>
<td>Programmed in Assembly</td>
<td>Programmed in C/C++ or Assembly</td>
</tr>
<tr>
<td>Single step moves the pipe one cycle</td>
<td>Single step flushes the pipeline</td>
</tr>
</tbody>
</table>
Summary

- CLA is an independent 32-bit floating-point math accelerator
  - robust, self saturating, and easy to program
- System and CLA initialization is done by the CPU in C
- The CLA can directly access:
  - ADC Result, ePWM+HRPWM and comparator registers
- The CLA is interrupt driven and has a low interrupt response time (no nesting of interrupts)
- By using the ADC early interrupt, the CLA can read the sample “Just-in-time”
  - Reduced ADC sample to output delay
  - Faster system response and higher MHz control loops
  - Support for multi-channel loops
IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI’s terms and conditions of sale supplied at the time of order acknowledgment.

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

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are neither designed nor intended for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use. These TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use. These TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

### Products
- Amplifiers: amplifier.ti.com
- Data Converters: dataconverter.ti.com
- DLP® Products: www.dlp.com
- DSP: dsp.ti.com
- Clocks and Timers: www.ti.com/clocks
- Interface: interface.ti.com
- Logic: logic.ti.com
- Power Mgmt: power.ti.com
- Microcontrollers: microcontroller.ti.com
- RFID: www.ti-rfid.com
- RF/IF and ZigBee® Solutions: www.ti.com/lprf

### Applications
- Audio: www.ti.com/audio
- Automotive: www.ti.com/automotive
- Communications and Telecom: www.ti.com/communications
- Computers and Peripherals: www.ti.com/computers
- Consumer Electronics: www.ti.com/consumer-apps
- Energy: www.ti.com/energy
- Industrial: www.ti.com/industrial
- Medical: www.ti.com/medical
- Security: www.ti.com/security
- Space, Avionics & Defense: www.ti.com/space-avionics-defense
- Video and Imaging: www.ti.com/video
- Wireless: www.ti.com/wireless-apps

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2010, Texas Instruments Incorporated