TMS320F2808™ One-Day Workshop

Workshop Guide and Lab Manual

C28xodw
Revision 4.0
April 2007

Technical Training Organization
Important Notice

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

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

Customers are responsible for their applications using TI components.

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

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

Copyright © 2003 – 2007 Texas Instruments Incorporated

Revision History

February 2003 – Revision 1.0
March 2003 – Revision 1.1
July 2004 – Revision 2.0
August 2005 – Revision 3.0
October 2005 – Revision 3.1
April 2007 – Revision 4.0

Mailing Address

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

Workshop Introduction ...............................................................................................................................4
Architecture Overview .................................................................................................................................7
Programming Development Environment ....................................................................................................10
Code Composer Studio ...............................................................................................................................10
Linking Sections in Memory .......................................................................................................................12
DSP/BIOS Configuration Tool .................................................................................................................13
Lab 1: DSP/BIOS Configuration Tool .......................................................................................................15
Peripheral Register Header Files ..............................................................................................................20
Reset, Interrupts and System Initialization ..................................................................................................26
Reset .....................................................................................................................................................26
Interrupts ..............................................................................................................................................27
Peripheral Interrupt Expansion (PIE) .........................................................................................................29
Oscillator / PLL Clock Module ..................................................................................................................31
Watchdog Timer Module ..........................................................................................................................32
GPIO .....................................................................................................................................................32
Lab 2: System Initialization .......................................................................................................................34
Control Peripherals ..................................................................................................................................38
ADC Module ........................................................................................................................................38
Pulse Width Modulation ..........................................................................................................................40
ePWM ...................................................................................................................................................41
Lab 3: Control Peripherals .........................................................................................................................52
IQ Math Library and DSP/BIOS ..................................................................................................................58
IQ Math Library ....................................................................................................................................58
DSP/BIOS ............................................................................................................................................60
Scheduling DSP/BIOS Threads ....................................................................................................................61
Real-Time Analysis Tools ..........................................................................................................................65
DSP/BIOS API Modules and Summary ....................................................................................................66
Lab 4: DSP/BIOS .....................................................................................................................................67
Flash Programming ................................................................................................................................74
Flash Programming Basics .......................................................................................................................74
Programming Utilities and CCS Plug-in .................................................................................................75
Code Security Module and Passwords ......................................................................................................76
Lab 5: Flash Programming .......................................................................................................................78
The Next Step ........................................................................................................................................85
Development Support ..............................................................................................................................85
Workshop Introduction

TMS320C28x™ 1-Day Workshop

Texas Instruments
Technical Training

eZdsp™ F2808 Starter Kit

eZdsp is a trademark of Spectrum Digital, Inc.
Copyright © 2007 Texas Instruments. All rights reserved.

C28x 1-Day Workshop Outline

◆ Workshop Introduction
◆ Architecture Overview
◆ Programming Development Environment
  • Lab: DSP/BIOS configuration tool
◆ Peripheral Register Header Files
◆ Reset, Interrupts and System Initialization
  • Lab: Watchdog and interrupts
◆ Control Peripherals
  • Lab: Generate and graph a PWM waveform
◆ IQ Math Library and DSP/BIOS
  • Lab: Filter the PWM waveform and use DSP/BIOS
◆ Flash Programming
  • Lab: Run the code from flash memory
◆ The Next Step…
Introductions

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

C2000 Portfolio Expanding with Price/Performance Optimized Derivatives

- High-Precision Control
- High-end Derivatives
- Application specific versions
- Cost optimized versions

DSP / Microcontroller Experience
- TMS320 DSP Experience
- Hardware / Software - Assembly / C
- Interests

Control Performance
- Multi-Function, Appliance & Consumer Control
- 24x™ up to 40 MIPS
- R281x 150 MIPS
- F281x 150 MIPS
- C281x 150 MIPS
- F280xx 100 MIPS
- C280x 100 MIPS

C2800 Portfolio Expanding with Price/Performance Optimized Derivatives
Workshop Introduction

**Broad C28x™ Application Base**

**Solar Inverters**
- Control of laser diode

**Digital Power Supply**
- Provides control, sensing, PFC, and other functions

**Optical Networking**
- LED control

**Industrial Motor Control**
- Motor control

**Other Segments**
- eg. Musical Instruments, HDTV/Displays

**Automotive**

**Medical**

**Non-traditional Motor Control**
- Many new cool applications to come

---

**C2000 Product Portfolio**

<table>
<thead>
<tr>
<th></th>
<th>MIPS</th>
<th>FLASH</th>
<th>ROM</th>
<th>RAM</th>
<th>OTP</th>
<th>PWM</th>
<th>CAP</th>
<th>QEP</th>
<th>ADC</th>
<th>EMIF</th>
<th>SPI</th>
<th>SCI</th>
<th>CAN</th>
<th>I2C</th>
<th>McBSP</th>
<th>GPIO</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2809</td>
<td>100</td>
<td>128K</td>
<td>--</td>
<td>18K</td>
<td>1K</td>
<td>16</td>
<td>4</td>
<td>2</td>
<td>16 ch</td>
<td>--</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>--</td>
<td>35</td>
</tr>
<tr>
<td>F2808</td>
<td>100</td>
<td>64K</td>
<td>--</td>
<td>18K</td>
<td>1K</td>
<td>16</td>
<td>4</td>
<td>2</td>
<td>16 ch</td>
<td>--</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>--</td>
<td>35</td>
</tr>
<tr>
<td>F2806</td>
<td>100</td>
<td>32K</td>
<td>--</td>
<td>10K</td>
<td>1K</td>
<td>16</td>
<td>4</td>
<td>2</td>
<td>16 ch</td>
<td>--</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>--</td>
<td>35</td>
</tr>
<tr>
<td>F2802</td>
<td>100</td>
<td>32K</td>
<td>--</td>
<td>6K</td>
<td>1K</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>16 ch</td>
<td>--</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>--</td>
<td>35</td>
</tr>
<tr>
<td>C2802</td>
<td>100</td>
<td>32K</td>
<td>--</td>
<td>6K</td>
<td>1K</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>16 ch</td>
<td>--</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>--</td>
<td>35</td>
</tr>
<tr>
<td>F2801</td>
<td>100</td>
<td>16K</td>
<td>--</td>
<td>6K</td>
<td>1K</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>16 ch</td>
<td>--</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>--</td>
<td>35</td>
</tr>
<tr>
<td>C2801</td>
<td>100</td>
<td>16K</td>
<td>--</td>
<td>6K</td>
<td>4</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>16 ch</td>
<td>--</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>--</td>
<td>35</td>
</tr>
<tr>
<td>F28016</td>
<td>100</td>
<td>16K</td>
<td>--</td>
<td>6K</td>
<td>1K</td>
<td>8</td>
<td>2</td>
<td>16 ch</td>
<td>--</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>--</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>F28015</td>
<td>100</td>
<td>16K</td>
<td>--</td>
<td>6K</td>
<td>1K</td>
<td>8</td>
<td>2</td>
<td>16 ch</td>
<td>--</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>--</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>F28044</td>
<td>100</td>
<td>64K</td>
<td>--</td>
<td>10K</td>
<td>1K</td>
<td>16</td>
<td>4</td>
<td>2</td>
<td>16 ch</td>
<td>--</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>--</td>
<td>35</td>
</tr>
<tr>
<td>F2812</td>
<td>150</td>
<td>128K</td>
<td>--</td>
<td>18K</td>
<td>1K</td>
<td>16</td>
<td>6</td>
<td>2</td>
<td>16 ch</td>
<td>Y</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>56</td>
</tr>
<tr>
<td>C2812</td>
<td>150</td>
<td>128K</td>
<td>--</td>
<td>128K</td>
<td>18K</td>
<td>16</td>
<td>6</td>
<td>2</td>
<td>16 ch</td>
<td>Y</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>56</td>
</tr>
<tr>
<td>F2811</td>
<td>150</td>
<td>128K</td>
<td>--</td>
<td>128K</td>
<td>18K</td>
<td>16</td>
<td>6</td>
<td>2</td>
<td>16 ch</td>
<td>--</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>56</td>
</tr>
<tr>
<td>C2811</td>
<td>150</td>
<td>128K</td>
<td>--</td>
<td>128K</td>
<td>18K</td>
<td>16</td>
<td>6</td>
<td>2</td>
<td>16 ch</td>
<td>--</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>56</td>
</tr>
<tr>
<td>F2810</td>
<td>150</td>
<td>64K</td>
<td>--</td>
<td>18K</td>
<td>1K</td>
<td>16</td>
<td>6</td>
<td>2</td>
<td>16 ch</td>
<td>--</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>56</td>
</tr>
<tr>
<td>C2810</td>
<td>150</td>
<td>64K</td>
<td>--</td>
<td>18K</td>
<td>1K</td>
<td>16</td>
<td>6</td>
<td>2</td>
<td>16 ch</td>
<td>--</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>56</td>
</tr>
<tr>
<td>R2812</td>
<td>150</td>
<td>20K</td>
<td>--</td>
<td>16</td>
<td>16</td>
<td>6</td>
<td>2</td>
<td>16 ch</td>
<td>Y</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>56</td>
<td></td>
</tr>
<tr>
<td>R2811</td>
<td>150</td>
<td>20K</td>
<td>--</td>
<td>16</td>
<td>16</td>
<td>6</td>
<td>2</td>
<td>16 ch</td>
<td>--</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>56</td>
<td></td>
</tr>
</tbody>
</table>

For further details and information on the C24x family refer to the “DSP Selection Guide” and specific “Data Manuals”
Architecture Overview

C28x Block Diagram

TMS320F2808 Memory Map

TMS320F2808 One-Day Workshop
**C28x Fast Interrupt Response Manager**

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

### Auto Context Save

<table>
<thead>
<tr>
<th>T</th>
<th>ST0</th>
</tr>
</thead>
<tbody>
<tr>
<td>AH</td>
<td>AL</td>
</tr>
<tr>
<td>PH</td>
<td>PL</td>
</tr>
<tr>
<td>AR1 (L)</td>
<td>AR0 (L)</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>

**Reset – Bootloader**

- **Reset**
  - OBJMODE = 0
  - AMODE = 0
  - EN PIE = 0
  - VMAP = 1

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

- Boot determined by state of GPIO pins

Note:
Details of the various boot options will be discussed with the Reset and Interrupts
eZdsp™ F2808 Hardware

- **JTAG Interface (P1)**
- **USB JTAG Controller Interface (J201)**
- **LED (DS1) +5V**
- **Power Connector (P6) +5V**
- **I/O Interface (P8)**
- **Bootloader GPIO Pins**
- **256K Bit Serial EEPROM**
- **20 MHz Clock**
- **LED (DS2) GPIO34**

**On-Chip:**
- 18K RAM
- 64K Flash
- 1K OTP

**TMS320F2808 DSP 100 MIPS**

**ANALOG Interface (P5/P9)**
- **SCl-A (P10)**
- **SCI-B (J10)**
- **eCAN-A (P11)**
- **eCAN-B (J11)**
- **USB JTAG Controller Interface (J201)**

**256K Bit Serial EEPROM**

**TMS320F2808 One-Day Workshop**
Programming Development Environment

Code Composer Studio

Code Composer Studio: IDE

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

The CCS Project

Project (.pjt) files contain:

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

Build Options GUI - Compiler

- 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

Build Options GUI - Linker

- GUI has 2 categories for linking
- Specifies various link options
- “\Debug\” indicates on subfolder level below project (.pjt) location
Linking Sections in Memory

Sections

- Every C program consists of different parts called sections
- All default sections names begin with ".
- The compiler has default sections names for initialized and uninitialized sections

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

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

Global Vars (.ebss) Init vals (.cinit)

(uninitialized) (initialized)

Local vars (.stack) Code (.text)
(uninitialized) (initialized)

Placing Sections in Memory – Linking

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

Link.cmd

`.obj` → Linker → `.out`

`.map`
DSP/BIOS Configuration Tool

DSP/BIOS Configuration Tool (file .cdb)

- MEM handles system memory configuration (builds .cmd file)
- Configures BIOS scheduling, RTA and other BIOS functions
- Automatically handles: run-time support libraries, interrupt vectors, system reset, etc.

Memory Section Manager

- Mem manager allows you to create memory area and place sections
- To create a new memory area:
  - Right-click on MEM and select Insert memory
  - Fill in base, length, space
To place a section into a memory area:
- Right-click on MEM and select Properties
- Select the appropriate tab (e.g. Compiler)
- Select the memory for each section

Config tool generates five different files
- Notice, one of them is the linker command file
- CMD file is generated from your MEM settings
Lab 1: DSP/BIOS Configuration Tool

Objective

Use Code Composer Studio and DSP/BIOS configuration tool to create a configuration database file (*.cdb). The generated linker command file Labcfg.cmd will be then be used with Lab1.c to verify its operation.

System Description:
- TMS320F2808
- All internal RAM blocks allocated

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

Procedure

Open a 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. 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.

2. A project is all the files you will need to develop an executable output file (.out) which can be run on the DSP hardware. A project named Lab1.pjt has been created for this lab. Open the project by clicking:

Project ➔ Open...

and look in C:\C28x\LABS\LAB1. This .pjt file will invoke all the necessary tools (compiler, assembler, linker) to build the project. It will also create a debug folder that will hold immediate output files.

3. In the project window on the left click the plus sign (+) to the left of Project. Now, click on the plus sign next to Lab1.pjt. Click on Source to see the current source file list (i.e. Lab1.c).

4. A test file named Lab1.c has been added to the project. This file will be used in this exercise to demonstrate some features of Code Composer Studio.

**Project Build Options**

5. 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...

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

7. Select OK and close the Build Options window.

**DSP/BIOS Configuration Tool**

8. The configuration database files (*.cdb), created by the Config Tool, controls a wide range of CCS capabilities. In this lab exercise, the CDB file will be used to automatically create and perform memory management.

9. In the project window left click the plus sign (+) to the left of DSP/BIOS Config. Notice that the Lab.cdb file is listed. Then left click the plus sign (+) to the left of Generated Files. The generated linker command file, Labcfg.cmd, is listed and has been automatically added with the configuration file.
Memory Sections Using the CDB File

10. Open the Lab.cdb file by double clicking on the Lab.cdb file. In the configuration window, left click the plus sign next to System and the plus sign next to MEM.

11. Notice that the memory sections have been created as described in the memory mapping shown in the figure at the beginning of this lab exercise.

12. To view the length and base address of a memory section, click on the memory in the configuration tool, and see the “Properties” in the right window.

13. Right click on MEM - Memory Section Manager and select Properties. Select the Compiler Sections tab and notice that .text, .cinit, and .ebss sections have been placed as defined on the lab introduction slide into the appropriate memories via the pull-down boxes. Similarly, the .stack section has been placed into memory using the BIOS Data tab.

14. When using the DSP/BIOS configuration tool, the stack size is specified in the CDB file. Select the General tab. Notice that the Stack Size has been set to 0x200. Click OK to close the window. Close the configuration 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 Load” tab, 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 when a control window is opened”, 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 Lab1.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 DSP/BIO\S C-environment initialization routine and stop at main() in Lab1.c.

**Debug Environme\n
t 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 “\texttt{z}”.

   Click: View \to Memory… on the menu bar.

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

   Set the properties format to “Hex – TI style.” This will give you more viewable data in the window. Click OK to close the window property selection screen. The memory window will now open. You can change the contents of any address in the memory window by double-clicking on its value. This is useful during debug.

22. Open the **watch window** to view the local variables \texttt{x} and \texttt{y}.

   Click: View \to Watch Window on the menu bar.

   Click the “Watch Locals” tab and notice that the local variables \texttt{x} and \texttt{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 “\texttt{z}”.

   Click the “Watch 1” tab at the bottom of the watch window. In the empty box in the "Name" column, type “\texttt{z}”. Note that you do not use an ampersand here. The watch window knows you are specifying a symbol. Check that the watch window and memory window both report the same value for “\texttt{z}”. Trying changing the value in one window, and notice that the value also changes in the other window.
Single-stepping the Code

24. 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
Peripheral Register Header Files

Traditional Approach to C Coding

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

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

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

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

Structure Approach to C Coding

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

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

Disadvantages
- Can be difficult to remember the structure names (Editor Auto Complete feature to the rescue!)
- More to type (again, Editor Auto Complete feature to the rescue)
The CCS Watch Window using #define

The CCS Watch Window using Structures
Peripheral Register Header Files

Structure Naming Conventions

◆ The DSP280x header files define:
  • All of the peripheral structures
  • All of the register names
  • All of the bit field names
  • All of the register addresses

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

Notes:
[1] "PeripheralName" are assigned by TI and found in the DSP280x 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!
DSP280x Header File Package
(http://www.ti.com, literature # SPRC191)

◆ Simplifies program of peripherals and other functions
◆ Takes care of register definitions and addresses
◆ Header file package consists of:
  • \DSP280x_headers\include → .h files
  • \DSP280x_common\src → .c source files
  • \DSP280x_headers\cmd → linker command files
  • \DSP280x_headers\gel → .gel files for CCS
  • \DSP280x_examples → example programs
  • \doc → documentation
◆ TI has done all of the work for you!

.h Definition Files

◆ Files found in the \DSP280x_headers\include directory
◆ Define structures and bit fields in peripheral and system registers
◆ DSP280x_Device.h – main include file will include all other .h files
  • #include “DSP280x_Device.h”

<table>
<thead>
<tr>
<th>DSP280x_Device.h</th>
<th>DSP280x_DevEmu.h</th>
<th>DSP280x_SysCtrl.h</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSP280x_PieCtrl.h</td>
<td>DSP280x_Adcd.h</td>
<td>DSP280x_CpuTimers.h</td>
</tr>
<tr>
<td>DSP280x_ECan.h</td>
<td>DSP280x_ECap.h</td>
<td>DSP280x_EPwm.h</td>
</tr>
<tr>
<td>DSP280x_EQep.h</td>
<td>DSP280x_Gpio.h</td>
<td>DSP280x_I2c.h</td>
</tr>
<tr>
<td>DSP280x_Sci.h</td>
<td>DSP280x_Spi.h</td>
<td>DSP280x_XIntrupt.h</td>
</tr>
<tr>
<td>DSP280x_PieVect.h</td>
<td>DSP280x_DefaultIsr.h</td>
<td></td>
</tr>
</tbody>
</table>
Global Variable Definition File

- **DSP280x_GlobalVariableDefs.C**
- Defines all variables to use .h files
- **DATA_SECTION** pragma used to define data section for each peripheral structure
- Linker will link each structure to the physical address of the peripheral in memory
- Add **DSP280x_GlobalVariableDefs.C** to the Code Composer Studio project

Peripheral Specific Routines

- Contains peripheral specific initialization routines and other support functions

<table>
<thead>
<tr>
<th>Source File</th>
<th>Target File</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSP280x_SysCtrl.c</td>
<td>DSP280x_Gpio.c</td>
</tr>
<tr>
<td>DSP280x_PieCtrl.c</td>
<td>DSP280x_I2c.c</td>
</tr>
<tr>
<td>DSP280x_Ad.c</td>
<td>DSP280x_Sci.c</td>
</tr>
<tr>
<td>DSP280x_CpuTimers.c</td>
<td>DSP280x_Spi.c</td>
</tr>
<tr>
<td>DSP280x_ECan.c</td>
<td>DSP280x_ECap.c</td>
</tr>
<tr>
<td>DSP280x_EPwm.c</td>
<td>DSP280x_EQep.c</td>
</tr>
<tr>
<td>DSP280x_PieVect.c</td>
<td>DSP280x_DefaultIIsr.c</td>
</tr>
</tbody>
</table>

Workshop lab files based on above files with modifications
Other Files in Packet

- Linker.cmd files
  - DSP280x_Headers_BIOS.cmd
  - DSP280x_Headers_nonBIOS.cmd
  - Contains memory allocation for all peripheral structure definitions included in C-code header file package

- DSP280x_CodeStartBranch.asm
  - Used to redirect code execution when booting
    ```assembly
    .ref    __c_int00
    .sect   "codestart"
    LB __c_int00        ;branch to start of boot.asm in RTS library
    ```

Workshop lab files based on above files with modifications

Peripheral Register Header Files
Summary

- Easier code development
- Easy to use
- Generates most efficient Code
- Increases Effectiveness of CCS Watch Window
- TI has already done all the work!
  - Download literature # SPRC191 from www.ti.com
Reset, Interrupts and System Initialization

Reset

C28x Reset Sources

Watchdog Timer  C28x Core
RS pin active

RS
To RS pin

Reset – Bootloader

<table>
<thead>
<tr>
<th>GPIO pins</th>
<th>Bootloader Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>18 29 34</td>
<td></td>
</tr>
<tr>
<td>1 1 1</td>
<td>jump to FLASH address 0x3F 7FF6</td>
</tr>
<tr>
<td>0 1 0</td>
<td>jump to M0 SARAM address 0x00 0000</td>
</tr>
<tr>
<td>0 0 1</td>
<td>jump to OTP address 0x3D 7800</td>
</tr>
<tr>
<td>1 1 0</td>
<td>bootload code to on-chip memory via SCI-A</td>
</tr>
<tr>
<td>1 0 1</td>
<td>bootload external EEPROM to on-chip memory via SPI-A</td>
</tr>
<tr>
<td>1 0 0</td>
<td>bootload external EEPROM to on-chip memory via I2C</td>
</tr>
<tr>
<td>0 1 1</td>
<td>call CAN_Boot to load from eCAN-A mailbox 1</td>
</tr>
<tr>
<td>0 0 0</td>
<td>bootload code to on-chip memory via GPIO port A (parallel)</td>
</tr>
</tbody>
</table>
Reset, Interrupts and System Initialization

Reset Code Flow - Summary

Interrupts

Interrupt Sources
**Maskable Interrupt Processing**

**Conceptual Core Overview**

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

---

**Core Interrupt Registers**

**Interrupt Flag Register (IFR)**

<table>
<thead>
<tr>
<th>Bit</th>
<th>RTOSINT</th>
<th>DLOGINT</th>
<th>INT14</th>
<th>INT13</th>
<th>INT12</th>
<th>INT11</th>
<th>INT10</th>
<th>INT9</th>
<th>INT8</th>
<th>INT7</th>
<th>INT6</th>
<th>INT5</th>
<th>INT4</th>
<th>INT3</th>
<th>INT2</th>
<th>INT1</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</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>
<td></td>
</tr>
<tr>
<td>14</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>
<td></td>
</tr>
<tr>
<td>13</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>
<td></td>
</tr>
<tr>
<td>12</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>
<td></td>
</tr>
<tr>
<td>11</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>
<td></td>
</tr>
<tr>
<td>10</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>
<td></td>
</tr>
<tr>
<td>9</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>
<td></td>
</tr>
<tr>
<td>8</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>
<td></td>
</tr>
</tbody>
</table>

**Interrupt Enable Register (IER)**

<table>
<thead>
<tr>
<th>Bit</th>
<th>RTOSINT</th>
<th>DLOGINT</th>
<th>INT14</th>
<th>INT13</th>
<th>INT12</th>
<th>INT11</th>
<th>INT10</th>
<th>INT9</th>
<th>INT8</th>
<th>INT7</th>
<th>INT6</th>
<th>INT5</th>
<th>INT4</th>
<th>INT3</th>
<th>INT2</th>
<th>INT1</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</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>
<td></td>
</tr>
<tr>
<td>14</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>
<td></td>
</tr>
<tr>
<td>13</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>
<td></td>
</tr>
<tr>
<td>12</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>
<td></td>
</tr>
<tr>
<td>11</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>
<td></td>
</tr>
<tr>
<td>10</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>
<td></td>
</tr>
<tr>
<td>9</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>
<td></td>
</tr>
<tr>
<td>8</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>
<td></td>
</tr>
</tbody>
</table>

**Interrupt Global Mask Bit (INTM)**

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

*** Global Interrupts ***/
asm(" CLRC INTM");  // enable global interrupts
asm(" SETC INTM");  // disable global interrupts
```
Peripheral Interrupt Expansion (PIE)

**Peripheral Interrupt Expansion - PIE**

- **Interrupt Group 1**
  - INTIFR1
  - PIER1
  - INT1.1
  - INT1.2
  - INT1.8

**Peripheral Interrupts**  \(12 \times 8 = 96\)

- INT1.x interrupt group
- INT2.x interrupt group
- INT3.x interrupt group
- INT4.x interrupt group
- INT5.x interrupt group
- INT6.x interrupt group
- INT7.x interrupt group
- INT8.x interrupt group
- INT9.x interrupt group
- INT10.x interrupt group
- INT11.x interrupt group
- INT12.x interrupt group

**INT1.1**

**INT1.2**

**INT1.8**

---

**PIE Registers**

- **PIEIFRx register** \(x = 1\) to \(12\)
  - 15 - 8: reserved
  - 7 - 0: INTx.8 to INTx.1

- **PIEIERx register** \(x = 1\) to \(12\)
  - 15 - 8: reserved
  - 7 - 0: INTx.8 to INTx.1

- **PIE Interrupt Acknowledge Register (PIEACK)**
  - 15 - 12: reserved
  - 11 - 9: reserved
  - 8 - 6: reserved
  - 5 - 3: reserved
  - 2 - 0: reserved

---

```c
#include "DSP280x_Device.h"
PieCtrlRegs.PIEIFR1.bit.INTx4 = 1;  // manually set IFR for XINT1 in PIE group 1
PieCtrlRegs.PIEIER3.bit.INTx5 = 1; // enable EPWM5_INT in PIE group 3
PieCtrlRegs.PIEACK.all = 0x0004;  // acknowledge the PIE group 3
PieCtrlRegs.PIECTRL.bit.ENPIE = 1; // enable the PIE
```
## F280x PIE Interrupt Assignment Table

<table>
<thead>
<tr>
<th>INTx.8</th>
<th>INTx.7</th>
<th>INTx.6</th>
<th>INTx.5</th>
<th>INTx.4</th>
<th>INTx.3</th>
<th>INTx.2</th>
<th>INTx.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT1</td>
<td>WAKEINT</td>
<td>TINT0</td>
<td>ADCINT</td>
<td>XINT2</td>
<td>XINT1</td>
<td>SEQ2INT</td>
<td>SEQ1INT</td>
</tr>
<tr>
<td>INT2</td>
<td>EPWM6</td>
<td>_TZINT</td>
<td>EPWM5</td>
<td>_TZINT</td>
<td>EPWM4</td>
<td>_TZINT</td>
<td>EPWM3</td>
</tr>
<tr>
<td>INT3</td>
<td>EPWM6</td>
<td>_INT</td>
<td>EPWM5</td>
<td>_INT</td>
<td>EPWM4</td>
<td>_INT</td>
<td>EPWM3</td>
</tr>
<tr>
<td>INT4</td>
<td></td>
<td></td>
<td>ECAP4</td>
<td>_INT</td>
<td>ECAP3</td>
<td>_INT</td>
<td>ECAP2</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>SPITXINTD</td>
<td>SPIRXINTD</td>
<td>SPITXINTC</td>
<td>SPIRXINTC</td>
<td>SPITXINTB</td>
<td>SPIRXINTB</td>
<td>SPITXINTA</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 _INTB</td>
<td>ECAN0 _INTB</td>
<td>ECAN1 _INTA</td>
<td>ECAN0 _INTA</td>
<td>SCITXINTB</td>
<td>SCIRXINTB</td>
<td>SCITXINTA</td>
</tr>
<tr>
<td>INT10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Hardware Interrupts

![Hardware Interrupts Diagram]
Oscillator / PLL Clock Module

C280x Oscillator / PLL Clock Module
SysCtrlRegs.PLLCR.bit.DIV (lab file: SysCtrl.c)

- X1 / XCLKIN
- XTAL OSC
- PLL
- OSCCLK
- VCOCLK
- PLLS bit.CLKINDIV

<table>
<thead>
<tr>
<th>DIV</th>
<th>Clock Frequency (CLKIN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>OSCCLK / CLKINDIV (PLL bypass)</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>OSCCLK x 1 / CLKINDIV</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>OSCCLK x 2 / CLKINDIV</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>OSCCLK x 3 / CLKINDIV</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>OSCCLK x 4 / CLKINDIV</td>
</tr>
<tr>
<td>0 1 0 1</td>
<td>OSCCLK x 5 / CLKINDIV</td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>OSCCLK x 6 / CLKINDIV</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>OSCCLK x 7 / CLKINDIV</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>OSCCLK x 8 / CLKINDIV</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>OSCCLK x 9 / CLKINDIV</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>OSCCLK x 10 / CLKINDIV</td>
</tr>
</tbody>
</table>

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

Watchdog Timer

- Resets the C28x if the CPU crashes
  - Watchdog counter runs independent of CPU
  - If counter overflows, reset or interrupt is triggered
  - CPU must write correct data key sequence to reset the counter before overflow
    - 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 must be serviced (or disabled) within ~6.55ms after reset (20 MHz OSCCLK for 100 MHz device)

- This translates into 131,072 instructions!
Watchdog Timer Module

Watchdog Timer Module (lab file: SysCtrl.c)

- 6 - Bit Free - Running Counter
- 8 - Bit Watchdog Counter
- 55 + AA Detector
- Watchdog Reset Key Register

GPIO

C280x GPIO Register Structure (lab file: Gpio.c)

Note: refer to the reference guide for a complete listing of registers and pin assignments
Lab 2: System Initialization

- LAB2 files have been provided
- LAB2 consists of two parts:
  Part 1
  - Test behavior of watchdog when disabled and enabled
  Part 2
  - Initialize peripheral interrupt expansion (PIE) vectors and use watchdog to generate an interrupt
- Modify, build, and test code using Code Composer Studio
Lab 2: System Initialization

- **Objective**

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

- Disable the watchdog – clear WD flag, disable watchdog, WD prescale = 1
- Setup the clock module – PLL = x10/2, HISPCP = /1, LOSPCP = /4, low-power modes to default values, enable all module clocks
- Setup system control register – DO NOT clear WD OVERRIDE bit, WD generate a DSP reset
- Setup shared I/O pins – set all GPIO pins to GPIO function (e.g. a "00" setting for GPIO function, and a “01”, “10”, or “11” setting for peripheral function.)

The first part of the lab exercise will setup the system initialization and test the watchdog operation by having the watchdog cause a reset. In the second part of the lab exercise the PIE vectors will tested by using the watchdog to generate an interrupt. This lab will make use of the DSP280x 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**

**Project File**

**Note:** LAB 2 files have been provided for the lab. DO NOT copy files from a previous lab.

1. A project named Lab2.pjt has been created for this lab. Open the project by clicking on Project → Open... and look in C:\C28x\LABS\LAB2. All Build Options have been configured. The files used in this lab are:

   - Main_2.c
   - Lab.Cdb
   - DSP280x_Headers_BIOS.cmd
   - User_2_3.cmd
   - CodeStartBranch.asm
   - SysCtrl.c
   - Gpio.c
   - DSP280x_GlobalVariableDefs.c
   - PieCtrl_2_3_4.c
   - DefaultIsr_2_3.c

Note that include files, such as DSP280x_Device.h and Lab.h, are automatically added at project build time. (Also, the generated linker command file, Labcfg.cmd, is automatically added with the configuration file).
**Modified Memory Configuration**

2. Open and inspect the user linker command file `User_2_3.cmd`. Notice that the 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 "Boot to M0" bootloader mode branches to address 0x000000 upon bootloader completion.

In the configuration file `lab.cdb` a new memory block has been created and named `BEGIN_M0`: base = 0x000000, length = 0x0002, space = code. Also, the existing memory block `M0SARAM` has been modified to avoid any overlaps with this new memory block.

**System Initialization**

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

4. Open and inspect `Gpio.c`. Notice that the shared I/O pins have been set to the GPIO function, except for GPIO0 which will be used in the next lab exercise. (Note: In `Main_2.c` do not edit the “main loop” section. This section will be used to test the watchdog operation.)

**Build and Load**

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

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

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

**Run the Code – Watchdog Reset**

8. 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.

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

10. 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”.

11. Modify the `InitSysCtrl()` function to enable the watchdog – in `SysCtrl.c` change the WDCR register value to 0x00A8. This will enable the watchdog to function and cause a reset. Save the file and click the “Build” button. Then reset the DSP by clicking on `Debug → Reset CPU`. Under Debug on the menu bar click “Go Main”.


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

**Setup PIE Vector for Watchdog Interrupt**

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

14. Notice that the following files are included in the project:

   ```plaintext
   PieCtrl_2_3_4.c
   DefaultIsr_2_3.c
   ```

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

   ```plaintext
   InitPieCtrl();
   ```

16. Using the “PIE Interrupt Assignment Table” shown in the slides find the location for the watchdog interrupt, “WAKEINT”.

17. In `main()` notice the code used to enable the watchdog interrupt:

   ```plaintext
   enable the "WAKEINT" interrupt in the PIE (using the PieCtrlRegs structure)
   enable core INT1 (IER register)
   enable global interrupts (INTM bit)
   ```

18. Modify the system control and status register (SCSR) to cause the watchdog to generate a WAKEINT rather than a reset – in `SysCtrl.c` change the SCSR register value to 0x0002.

19. Open and inspect `DefaultIsr_2_3.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.
20. In the configuration file Lab.cdb the PIE vector for the watchdog interrupt has been configured. To view the setup, click on the plus sign (+) to the left of Scheduling and again on the plus sign (+) to the left of HWI – Hardware Interrupt Service Routine Manager. Click the plus sign (+) to the left of PIE INTERRUPTS. Locate the interrupt location for the watchdog at PIE_INT1_8. Click and see the “Properties” in the right window. The function field contains _WAKEINT_ISR (with a leading underscore). Close the configuration file.

Build and Load

21. Save all changes to the modified file and click the “Build” button. Then reset the DSP, and then “Go Main”.

Run the Code – Watchdog Interrupt

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

23. 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 SysCtrl.c.
Control Peripherals

ADC Module

ADC Module Block Diagram (Cascaded Mode)

ADC Module Block Diagram (Dual-Sequencer mode)
Control Peripherals

ADC Control Registers (file: Adc.c)

- **ADCTRL1** (ADC Control Register 1)
  - module reset
  - continuous run / stop EOS
  - sequencer mode (cascaded / dual)
  - acquisition time prescale (S/H)

- **ADCTRL2** (ADC Control Register 2)
  - ePWM SOC; start conversion (s/w trigger); ePWM SOC mask bit
  - reset SEQ
  - interrupt enable; interrupt mode: every EOS / every other EOS

- **ADCTRL3** (ADC Control Register 3)
  - ADC clock prescale
  - sampling mode (sequential / simultaneous)

- **ADCMAXCONV** (ADC Maximum Conversion Register)
  - maximum number of autoconversions

- **ADCCHSELSEQx {x=1-4}** (ADC Channel Select Register)
  - Channel select sequencing

- **ADCRESULTx {x=0-15}** (ADC Results Register)

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

What is Pulse Width Modulation?

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

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
ePWM

**ePWM Block Diagram**

- **Clock Prescaler**
- **16-Bit Time-Base Counter**
- **Compare Logic**
- **Action Qualifier**
- **Dead Band**
- **PWM Chopper**
- **Trip Zone**
- **Period Register**
- **Shadowed Compare Register**
- **Shadowed Compare Register**

**Note:** x = 1, 2, 3, 4, 5, or 6

**Note:** Up to 6 ePWM modules, 4 eCAP modules, and 2 eQEP modules are available on the F280x devices. The lower case “e” used with the module name refers to enhanced.

---

**ePWM Time-Base Module**

- **Clock Prescaler**
- **16-Bit Time-Base Counter**
- **Compare Logic**
- **Action Qualifier**
- **Dead Band**
- **PWM Chopper**
- **Trip Zone**
- **Period Register**

**Note:** x = 1, 2, 3, 4, 5, or 6
Control Peripherals

**ePWM Time-Base Count Modes**

- **Count Up Mode**
- **Count Down Mode**
- **Count Up and Down Mode**

**Asymmetrical Waveform**

**Symmetrical Waveform**

**ePWM Phase Synchronization**

- **Phase \( \phi = 0^\circ \)**
- **Phase \( \phi = 120^\circ \)**
- **Phase \( \phi = 240^\circ \)**

- **Ext SyncIn** (optional)

- **SyncIn**
- **SyncOut**

- **EPWM1A**
- **EPWM1B**
- **EPWM2A**
- **EPWM2B**
- **EPWM3A**
- **EPWM3B**
Control Peripherals

16-Bit Time-Base Counter

- TBCTL . 12 - 7
- TBCTR . 15 - 0
- TBPRD . 15 - 0
- CMPA . 15 - 0
- CMPB . 15 - 0
- AQCTLA . 11 - 0
- AQCTLB . 11 - 0
- DBCTL . 4 - 0
- EPWMxA
- EPWMxB
- TZx

Action Qualifier

- Dead Band
- Trip Zone

Period Register

- Shadowed
- CMPA
- CMPB
- Period Register

Compare Logic

- Compare Register
- Shadowed

Shadowed Compare Register

- CMPA . 15 - 0
- CMPB . 15 - 0

Shadowed Period Register

- TBPRD . 15 - 0

Count Up Mode

- Asymmetrical Waveform

Count Down Mode

- Asymmetrical Waveform

Count Up and Down Mode

- Symmetrical Waveform

Note: x = 1, 2, 3, 4, 5, or 6
Control Peripherals

**ePWM Action Qualifier Module**

- Clock Prescaler
- 16-Bit Time-Base Counter
- Compare Logic
- Action Qualifier
- Dead Band
- PWM Chopper
- Trip Zone
- TZW

**ePWM Action Qualifier Actions**

*Note: x = 1, 2, 3, 4, 5, or 6*

<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 CMPA CMPB TBPRD</td>
<td></td>
</tr>
<tr>
<td>SWX</td>
<td>Z CA CX CB P X</td>
<td>Do Nothing</td>
</tr>
<tr>
<td>SW↓</td>
<td>Z CA CB P ↓</td>
<td>Clear Low</td>
</tr>
<tr>
<td>SW↑</td>
<td>Z CA CB P ↑</td>
<td>Set High</td>
</tr>
<tr>
<td>SWT</td>
<td>Z CA CB P T</td>
<td>Toggle</td>
</tr>
</tbody>
</table>
ePWM Count Up Asymmetric Waveform
with Independent Modulation on EPWMA / B

TBCTR
TBPRD

EPWMA
Z ↑ P X CB X CA ↓ Z ↑ P X CB X CA ↓ Z ↑ P X

EPWMB
Z ↑ P X CB ↓ CA X Z ↑ P X CB ↓ CA X Z ↑ P X

ePWM Count Up Asymmetric Waveform
with Independent Modulation on EPWMA

TBCTR
TBPRD

EPWMA
CA ↑ CB ↓ CA ↑ CB ↓ CA ↑ CB ↓

EPWMB
Z T Z T Z T
ePWM Count Up-Down Symmetric Waveform
with Independent Modulation on EPWMA / B

- TBCTR
- TBPRD
- EPWMA
- EPWMB

---

ePWM Count Up-Down Symmetric Waveform
with Independent Modulation on EPWMA

- TBCTR
- TBPRD
- EPWMA
- EPWMB
Control Peripherals

**ePWM Dead-Band Module**

- **Clock Prescaler**
- **16-Bit Time-Base Counter**
- **Compare Logic**
- **Action Qualifier**
- **Dead Band**
- **Period Register**
- **PWM Chopper**
- **Trip Zone**

**Note:** x = 1, 2, 3, 4, 5, or 6

---

**Motivation for Dead-Band**

- Gate Signals are Complementary PWM
- Transistor gates turn on faster than they shut off
- Short circuit if both gates are on at the same time!
**ePWM PWM Chopper Module**

- **16-Bit Time-Base Counter**
- **Compare Logic**
- **Action Qualifier**
- **Dead Band**
- **16-Bit Compare Register**
- **Shadowed Compare Register**
- **Period Register**
- **Shadowed Period Register**
- **TBCTL . 12 - 7**
- **CBCTR . 15 - 0**
- **CMPA . 15 - 0**
- **CMPB . 15 - 0**
- **AQCTLA . 11 - 0**
- **AQCTLB . 11 - 0**
- **DBCTL . 4 - 0**
- **PCCTL . 10 - 0**
- **TZSEL . 15 - 0**
- **EPWMxA SYNCI**
- **EPWMxB SYNCO**
- **EPWMxSYNCI**
- **EPWMxSYNCO**
- **SYSCLKOUT**
- **Note: x = 1, 2, 3, 4, 5, or 6**

**ePWM Chopper Waveform**

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

- **Clock Prescaler**
- **16-Bit Time-Base Counter**
- **Compare Logic**
- **Action Qualifier**
- **Dead Band**
- **PWM Chopper**
- **Trip Zone**

**Note:** $x = 1, 2, 3, 4, 5, \text{ or } 6$

---

**Trip-Zone Module 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
Control Peripherals

**ePWM Event-Trigger Module**

- **Clock Prescaler**
- **16-Bit Time-Base Counter**
- **Compare Logic**
- **ActionQualifier**
- **Dead Band**
- **PWM Chopper**
- **Trip Zone**

**Note:** x = 1, 2, 3, 4, 5, or 6

---

**ePWM Event-Trigger Interrupts and SOC**

- **TBCTR . 15 - 0**
- **TBPRD . 15 - 0**
- **CMPA . 15 - 0**
- **CMPB . 15 - 0**
- **AQCTLA . 11 - 0**
- **AQCTLB . 11 - 0**
- **DBCTL . 4 - 0**
- **EPWMxA**
- **EPWMxB**
- **EPWMxSYNCI**
- **EPWMxSYNCO**
- **TZx**

---

50  TMS320F2808 One-Day Workshop
Hi-Resolution PWM (HRPWM)

- Significantly increases the resolution of conventionally derived digital PWM
- Uses 8-bit extensions to Compare registers (CMPxHR) and Phase register (TBPHSHR) for edge positioning control
- Typically used when PWM resolution falls below ~9-10 bits which occurs at frequencies greater than ~200 kHz (with system clock of 100 MHz)
- Not all ePWM outputs support HRPWM feature (see device data manual)

---

ePWM Control Registers (file: EPwm.c)

- TBCTL  (Time-Base Control)
  - counter mode (up, down, up & down, stop); clock prescale; period shadow load; phase enable/direction; sync select
- CMPCTL  (Compare Control)
  - compare load mode; operating mode (shadow / immediate)
- AQCTLA/B  (Action Qualifier Control Output A/B)
  - action on up/down CTR = CMPA/B, PRD, 0 (nothing/set/clear/toggle)
- DBCTL  (Dead-Band Control)
  - in/out-mode (disable / delay PWMxA/B); polarity select
- PCCTL  (PWM-Chopper Control)
  - enable / disable; chopper CLK freq. & duty cycle; 1-shot pulse width
- TZCTL  (Trip-Zone Control)
  - enable /disable; action (force high / low / high-Z /nothing)
- ETSEL  (Event-Trigger Selection)
  - interrupt & SOCA/B enable / disable; interrupt & SOCA/B select

Note: refer to the reference guide for a complete listing of registers
Lab 3: Control Peripherals

Objective

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

Two ePWM modules have been configured for this lab exercise:

ePWM1A – PWM Generation

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

ePWM2 – ADC Conversion Trigger

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

The software in this exercise configures the ePWM modules and the ADC. It is entirely interrupt driven. The ADC end-of-conversion interrupt will be used to prompt the CPU to copy the results of the ADC conversion into a results buffer in memory. This buffer pointer will be managed in a circular fashion, such that new conversion results will continuously overwrite older conversion.
results in the buffer. The ADC interrupt service routine (ISR) will also toggle LED DS2 on the eZdsp™ as a visual indication that the ISR is running.

Notes

• ePWM1A is used to generate a 2 kHz PWM waveform
• Program performs conversion on ADC channel A0 (ADCINA0 pin)
• ADC conversion is set at a 50 kHz sampling rate
• ePWM2 is triggering the ADC on period match using SOC A trigger
• Data is continuously stored in a circular buffer
• Data is displayed using the graphing feature of Code Composer Studio
• ADC ISR will also toggle the eZdsp™ LED DS2 as a visual indication that it is running

➢ Procedure

Project File

Note: LAB 3 files have been provided for the lab. DO NOT copy files from a previous lab.

1. A project named Lab3.pjt has been created for this lab. Open the project by clicking on Project → Open... and look in C:\C28x\LABS\LAB3. All Build Options have been configured. The files used in this lab are:

   Main_3.c  Labcfg.cmd
   Lab.cdb    DSP280x_Headers_BIOS.cmd
   User_2_3.cmd  CodeStartBranch.asm
   SysCtrl.c  Gpio.c
   DSP280x_GlobalVariableDefs.c  PieCtrl_2_3_4.c
   DefaultIsr_2_3.c  Adc.c
   EPwm.c  DelayUs.asm

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

   Note: DO NOT make any changes to Gpio.c and EPwm.c — ONLY INSPECT

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

Build and Load

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

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

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

Run the Code – PWM Waveform

6. Open a memory window to view some of the contents of the ADC results buffer. To open a memory window click: View → Memory... on the menu bar. The address label for the ADC results buffer is AdcBuf.

Note: Exercise care when connecting any wires, as the power to the eZdsp™ is on, and we do not want to damage the eZdsp™! Details of pin assignments can be found in Appendix A.

7. Using a connector wire provided, connect the PWM1A (pin # P8-9) to ADCINA0 (pin # P9-2) on the eZdsp™.

8. 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. Verify that the ADC result buffer contains the updated values.

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

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

Select OK to save the graph options.

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

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

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

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

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

Select OK to save the graph options.

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

**Using Real-time Emulation**

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

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

   **Debug → Real-time Mode**

14. A message box may appear. Select **YES** to enable debug events. This will set bit 1 (DGBM bit) of status register 1 (ST1) to a “0”. The DGBM is the debug enable mask bit. When the DGBM bit is set to “0”, memory and register values can be passed to the host processor for updating the debugger windows.
15. The memory and graph windows displaying \textit{AdcBuf} should still be open. The connector wire between PWM1A (pin # P8-9) and ADCINA0 (pin # P9-2) should still be connected. In real-time mode, we would like to have our window continuously refresh. Click:

View $\rightarrow$ Real-time Refresh Options...

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

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

16. Run the code and watch the windows update in real-time mode. Are the values updating as expected?

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

**Real-time Mode using GEL Functions**

18. 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 DSP, enter real-time mode, run DSP)
- Run\_Realtime\_with\_Restart (restart DSP, enter real-time mode, run DSP)
- Full\_Halt (exit real-time mode, halt DSP)
- Full\_Halt\_with\_Reset (exit real-time mode, halt DSP, reset DSP)

These GEL functions can be executed by clicking:

GEL $\rightarrow$ Realtime Emulation Control $\rightarrow$ GEL Function

If you would like, try repeating the previous step using the following GEL functions:

GEL $\rightarrow$ Realtime Emulation Control $\rightarrow$ Run\_Realtime\_with\_Reset

GEL $\rightarrow$ Realtime Emulation Control $\rightarrow$ Full\_Halt

**Optional Exercise**

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

**End of Exercise**
IQ Math Library and DSP/BIOS

IQ Math Library

IQmath Library: Floating Point “Ease of Use” on a Fixed Point Machine

- Control algorithms typically start in a Floating-Point format
- The conversion of such algorithms, to run on a fixed-point machine, is a laborious and time-consuming task
- The 32-bit math capabilities of the C28x™ core enable a new C/C++ approach, which makes this task easier and much faster – optimized to take advantage of C28x™ architecture

IQ Fractional Representation

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

A mathematical approach and a set of supporting math libraries that enable the following:
- Reduced time to implement/port/debug math algorithms in C/C++
- Increased numerical resolution of algorithms from 16-bits to 32/64-bits

Typical

Users typically start with a floating point algorithm...

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

and then spend many hours converting to a fixed point algorithm which is not easy to read

```c
int Y, M, X, B; //Q1 to Q15
Y = ((M * X) + B << Q) >> Q);
```

IQmath

IQmath reduces this effort dramatically and the code is easier to read (looks “natural”)

(Using IQmath in C):
```c
_iq Y, M, X, B; //Q1 to Q30
Y = _IQmpy(M, X) + B;
```

(Using IQmath in C++)
```c
iq Y, M, X, B; //Q1 to Q30
Y = M * X + B;
```

IQmath Approach Summary

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

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

IQmath library is freeware - available from TI DSP website
Introduction to DSP/BIOS

What is DSP/BIOS?
- A full-featured, scalable real-time kernel
  - System configuration tools
  - Preemptive multi-threading scheduler
  - Real-time analysis tools

Why use DSP/BIOS?
- Helps manage complex system resources
- Integrated with Code Composer Studio IDE
  - Requires no runtime license fees
  - Fully supported by TI and is a key component of TI's eXpressDSP™ real-time software technology
- Uses minimal MIPS and memory (2-8Kw)

DSP/BIOS Configuration Tool (file .cdb)

- System Setup Tools
  Handles memory configuration (builds .cmd file), run-time support libraries, interrupt vectors, system setup and reset, etc.
- Real-Time Analysis Tools
  Allows application to run uninterrupted while displaying debug data
- Real-Time Scheduler
  Preemptive thread manager kernel
- Real-Time I/O
  Allows two way communication between threads or between target and PC host
Scheduling DSP/BIOS Threads

### DSP/BIOS Thread Types

<table>
<thead>
<tr>
<th>Priority</th>
<th>HWI</th>
<th>Software Interrupts</th>
<th>Tasks</th>
<th>Background</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HWI</td>
<td>SWI</td>
<td>TSK</td>
<td>IDL</td>
</tr>
<tr>
<td></td>
<td>Hardware Interrupts</td>
<td>Software Interrupts</td>
<td>Tasks</td>
<td>Background</td>
</tr>
<tr>
<td></td>
<td>Used to implement 'urgent' part of real-time event</td>
<td>Use SWI to perform HWI 'follow-up' activity</td>
<td>Use TSK to run different programs concurrently under separate contexts</td>
<td>Runs when no service routines are pending</td>
</tr>
<tr>
<td></td>
<td>Triggered by hardware interrupt</td>
<td>SWI's are 'posted' by software</td>
<td>TSK's can be terminated by software</td>
<td>Runs as an infinite loop, like traditional while loop</td>
</tr>
<tr>
<td></td>
<td>HWI priorities fixed in hardware</td>
<td>Multiple SWIs at each of 15 priority levels</td>
<td></td>
<td>All BIOS data transfers to host occur here</td>
</tr>
</tbody>
</table>

### Enabling BIOS – Return from main()

- Must delete the endless while() loop
- main() returns to BIOS and goes to IDLE thread, allowing BIOS to schedule events, transfer information to the host, etc.
- **An endless while() loop in main() will not allow BIOS to activate**

```c
void main(void)
{
    /*************************************************************************
     *** Initialization ***
     . . .
     /*************************************************************************
    /**************************************************************************
    *** Enable global interrupts ***
    /*************************************************************************
    // DSP/BIOS will enable global interrupts
    /**************************************************************************
    *** Main Loop ***
    /*************************************************************************
    // while() loop removed to enable DSP/BIOS
} //end of main()
```
HWI Dispatcher for ISRs

- 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

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
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
Specify the DSP/BIOS CLK rate in microseconds per “tick”

Allows multiple periodic functions with different rates

Specify the period (for the function) in ticks
Real-Time Analysis Tools

Built-in Real-Time Analysis Tools
- Gather data on target (3-10 CPU cycles)
- Send data during BIOS IDL (100s of cycles)
- Format data on host (1000s of cycles)
- Data gathering does NOT stop target CPU

Execution Graph
- Software logic analyzer
- Debug event timing and priority

CPU Load Graph
- Shows amount of CPU horsepower being consumed

Statistics View
- Profile routines w/o halting the CPU

Message LOG
- Send debug msgs to host
- Doesn’t halt the DSP
- Deterministic, low DSP cycle count
- More efficient than traditional printf()

LOG_printf(&trace, "AdcSwi_count = %u", AdcSwi_count++);
DSP/BIOS API Modules and Summary

### DSP/BIOS - API Modules

<table>
<thead>
<tr>
<th>Instrumentation/Real-Time Analysis</th>
<th>TSK Communication/Synchronization</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOG</td>
<td>SEM</td>
</tr>
<tr>
<td>STS</td>
<td>MBX</td>
</tr>
<tr>
<td>TRC</td>
<td>LCK</td>
</tr>
<tr>
<td>RTDX</td>
<td></td>
</tr>
<tr>
<td><strong>Thread Types</strong></td>
<td></td>
</tr>
<tr>
<td>HWI</td>
<td>SEM</td>
</tr>
<tr>
<td>SWI</td>
<td>MBX</td>
</tr>
<tr>
<td>TSK</td>
<td>LCK</td>
</tr>
<tr>
<td>IDL</td>
<td></td>
</tr>
<tr>
<td><strong>Clock and Periodic Functions</strong></td>
<td></td>
</tr>
<tr>
<td>CLK</td>
<td></td>
</tr>
<tr>
<td>PRD</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Device-Independent Input/Output</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>PIP</td>
</tr>
<tr>
<td>HST</td>
</tr>
<tr>
<td>SIO</td>
</tr>
<tr>
<td>DEV</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Memory and Low-Level Primitives</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>MEM</td>
</tr>
<tr>
<td>SYs</td>
</tr>
<tr>
<td>QUE</td>
</tr>
<tr>
<td>ATM</td>
</tr>
<tr>
<td>GBL</td>
</tr>
</tbody>
</table>

### Summary and Benefits of DSP/BIOS

- **Fast time to market**
  - no need to develop or maintain a “home-brew” kernel
- **Efficient debugging of real-time applications**
  - Real-Time Analysis
- **Create robust applications**
  - industry proven kernel technology
- **Reduce cost of software maintenance**
  - code reuse and standardized software
- **Standardized APIs**
  - enable rapid migration across C28x TMS320 DSPs
- **Small footprint (2-8Kw)**
  - easily fits in limited memory space
- **Set of library functions (scalable)**
  - use only what is needed to minimize code and data size
- **Full featured kernel (extensible)**
  - allows additional OS functions in future
Lab 4: DSP/BIOS

Objective

The objective of this lab is to demonstrate the techniques discussed in this module and to become familiar with DSP/BIOS. 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 IQmath FIR filter and displayed using the graphing feature of Code Composer Studio. In this lab exercise, we are going to change the ADCINT_ISR HWI to a SWI. Then, we will replace the LED blink routine with a Periodic Function. Also, some features of the real-time analysis tools will be demonstrated.

Procedure

Project File

1. A project named Lab4.pjt has been created for this lab. Open the project by clicking on Project → Open... and look in C:\C28x1DAY\LABS\LAB4. All Build Options have been configured. The files used in this lab are:
Include IQmathLib.h

2. Open Lab.h which is located in the include folder in the project window. 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. Save the changes.

Add a SWI to main.c

Note: DO NOT make any changes to Main_4.c and DefaultISR_4_5.c – ONLY INSPECT

3. Open Main_4.c and notice that at the end of main() two new functions will be used in this module – AdcSwi() and LedBlink().

4. Open DefaultIsr_4_5.c, and notice that the interrupt key word for the ADCINT_ISR is not used. 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.

5. In Main_4.c notice that the (in-line assembly) code used to enable global interrupts is not used. DSP/BIOS will enable global interrupts after main().

6. Again, in Main_4.c notice that the while() loop (the endless loop) is not used. 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.

Post a SWI

7. In DefaultIsr_4_5.c the following SWI_post has been added to the ADCINT_ISR(), just after the structure used to acknowledge the PIE group:

   SWI_post(&ADC_sw1);  // post a SWI

This will post a SWI that will execute the ADC_sw1() code you examined a few steps back in the lab. In other words, the ADC interrupt still executes the same code as before. However, some of that code is now in a posted SWI that DSP/BIOS will execute according to the specified scheduling priorities.
Add the SWI to the CDB File

8. Open the Lab.cdb file by left clicking on the plus sign (+) to the left of DSP/BIOS Config and double clicking on the Lab.cdb file. In the configuration file Lab.cdb we need to add and setup the AdcSwi() SWI. Click on the plus sign (+) to the left of Scheduling and again on the plus sign (+) to the left of SWI – Software Interrupt Manager.

9. Right click on SWI – Software Interrupt Manager and select Insert SWI. SWI0 will be added. Right-click on it, and rename it to ADC_swi. This is just an arbitrary name. We want to differentiate the AdcSwi() function itself (which is nothing but an ordinary C function) from the DSP/BIOS SWI object which we are calling ADC_swi.

10. Right click on ADC_swi. Select the Properties for ADC_swi and type _AdcSwi (with a leading underscore) in the function field. Click OK. This tells DSP/BIOS that it should run the function AdcSwi() when it executes the ADC_swi SWI.

11. We need to have the PIE for the ADC interrupt use the dispatcher. The dispatcher will automatically perform the context save and restore, and allow the DSP/BIOS scheduler to have insight into the ISR. The ADC interrupt is located at PIE_INT1_6.

   Click on the plus sign (+) to the left of HWI – Hardware Interrupt Service Routine Manager. Click the plus sign (+) to the left of PIE INTERRUPTS. Locate the interrupt location for the ADC: PIE_INT1_6. Right click, select Properties, and select the Dispatcher tab.

   Now check the “Use Dispatcher” box and select OK. Close the configuration file and click YES to save changes.

Build and Load

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

Run the Code – AdcSwi()

13. We will be running our code in real-time mode, and will have our window continuously refresh. Run in Real-time Mode using the GEL function: GEL → Realtime Emulation Control → Run_Realtime_with_Reset.

Note: For the next step, check to be sure that the jumper wire connecting PWM1A (pin # P8-9) to ADCINA0 (pin # P9-2) are still in place on the eZdsp™.

14. 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:
15. The graphical display should show the generated IQmath FIR filtered 2 kHz, 25% duty cycle symmetric PWM waveform in the upper display and the unfiltered waveform generated in the previous lab exercise in the lower display. Notice the shape and phase differences between the waveform plots (the filtered curve has rounded edges, and lags the unfiltered plot by several samples). The amplitudes of both plots should run from 0 to 3.0. The results should be the same as the previous lab, but we are using a SWI rather than a HWI.

16. 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></th>
<th>GRAPH #1</th>
<th>GRAPH #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display Type</td>
<td>FFT Magnitude</td>
<td>FFT Magnitude</td>
</tr>
<tr>
<td>Start Address</td>
<td>AdcBuf</td>
<td>AdcBufFiltered</td>
</tr>
<tr>
<td>Acquisition Buffer Size</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>FFT Framesize</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>DSP Data Type</td>
<td>32-bit signed integer</td>
<td>32-bit signed integer</td>
</tr>
<tr>
<td>Q-value</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>Sampling Rate (Hz)</td>
<td>50000</td>
<td>50000</td>
</tr>
</tbody>
</table>

Select OK to save the graph options.
17. 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 IQmath FIR filter in the filtered graph as compared to the unfiltered graph.

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

**Add a Periodic Function**

Recall that an instruction was used in the ADCINT_ISR to toggle the LED on the eZdsp™. This instruction has been moved into a periodic function that will toggle the LED at the same rate.

19. Open Main_4.c and notice the instruction used to toggle the LED to the LedBlink() function:

   ```c
   GpioDataRegs.GPBTOGGLE.bit.GPIO34 = 1; // Toggle the pin
   ```

20. In the configuration file Lab.cdb we need to add and setup the LedBlink_PRD. Open Lab.cdb and click on the plus sign (+) to the left of Scheduling. Right click on PRD – Periodic Function Manager and select Insert PRD. PRD0 will be added. Right-click on it and rename it to LedBlink_PRD.

21. 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.

**Build and Load**

22. Click the “Build” button to rebuild and load the project.

**Run the Code – LedBlink**

23. Run the code (real-time mode) by using the GEL function: GEL → Realtime Emulation Control → Run_Realtime_with_Reset, and check to see if the LED on the eZdsp™ is blinking.

24. When done, fully halt the DSP (real-time mode) by using the GEL function: GEL → Realtime Emulation Control → Full_Halt. If you would like, experiment with different period (tick) values and notice that the blink rate changes.
DSP/BIOS – Real-time Analysis

The DSP/BIOS analysis tools complement the CCS environment by enabling real-time program analysis of a DSP/BIOS application. You can visually monitor a DSP application as it runs with essentially no impact on the application’s real-time performance. In CCS, the DSP/BIOS analysis tools are found on the DSP/BIOS menu. Unlike traditional debugging, which is external to the executing program, DSP/BIOS program analysis requires that the target program be instrumented with analysis code. By using DSP/BIOS APIs and objects, developers automatically instrument the target for capturing and uploading real-time information to CCS using these tools.

25. Open the CPU load graph. On the menu bar click:

DSP/BIOS → CPU Load Graph

The CPU load graph displays the percentage of available CPU computing horsepower that the application is consuming. The CPU may be running ISRs, software interrupts, periodic functions, performing I/O with the host, or running any user routine. When the CPU is not executing user code, it will be idle (in the DSP/BIOS idle thread).

Run the code (real-time mode) by using the GEL function: GEL → Realtime Emulation Control → Run_Realtime_with_Reset. You will notice that we are consuming about 68% of the CPU with our lab code. (Note: DSP/BIOS realtime analysis tools have some computational overhead. The overhead can be reduced by selectively enabling feature in the RTA Control Panel).

26. Next, open the execution graph. On the menu bar click:

DSP/BIOS → Execution Graph

The execution graph is a special graph used to display information about different threads in the system and when they occur relative to the other events. This graph is not based on time, but the activity of events (i.e. when an event happens, such as a SWI or periodic function begins execution). You can enable or disable logging for each of the object types using the RTA Control Panel (DSP/BIOS → RTA Control Panel). Note that the execution graph simply records DSP/BIOS CLK events along with other system events (the DSP/BIOS clock periodically triggers the DSP/BIOS scheduler). As a result, the time scale on the execution graph is not linear.

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

28. In the next few steps the Log Event Manager will be setup to capture an event in real-time while the program executes. We will be using LOG_printf() to write to a log buffer. The LOG_printf() function is a very efficient means of sending a message from the code to the CCS display. Unlike an ordinary C-language printf(), which can consume several hundred DSP cycles to format the data on the DSP before transmission to the CCS host PC, a log_printf() transmits the raw data to the host. The host then formats the data and displays it in CCS. This consumes only 10’s of cycles rather than 100’s of cycles.
In Main_4.c just after the static local variable declaration in AdcSwi(), *uncomment* the following code used with LOG_printf:

```c
static Uint32 AdcSwi_count=0;          // used for LOG_printf
/*** Using LOG_printf() to write to a log buffer ***/
LOG_printf(&trace, "AdcSwi_count = %u", AdcSwi_count++);
```

29. In the configuration file Lab.cdb we need to add and setup the trace buffer. Open Lab.cdb and click on the plus sign (+) to the left of Instrumentation and again on the plus sign (+) to the left of LOG – Event Log Manager.

30. Right click on LOG – Event Log Manager and select Insert LOG. LOG0 will be added. Right-click on it and rename it to trace.

31. Select the Properties for trace and set the logtype to *circular* and the datatype to *printf*. Click OK. Close the configuration file and click YES to save changes.

32. Since the configuration file was modified, we need to rebuild the project. Click the “Build” button.

33. Run the code (real-time mode) by using the GEL function: GEL → Realtime Emulation Control → Run_Realtime_with_Reset.

34. Open the Message Log by clicking:

DSP/BIOS → Message Log

The message log dialog box is displaying the number of times (count value) that AdcSwi() has executed.

35. Observe the operation of the various windows that are open, and the information that they are conveying in real-time.

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

---

**Note:** In this module only the basic features of DSP/BIOS and the real-time analysis tools have been used. For more information and details, please refer to the DSP/BIOS user’s manuals and other DSP/BIOS related training.

---

**End of Exercise**
Flash Programming Basics

- The DSP CPU itself performs the flash programming
- The CPU executes Flash utility code from RAM that reads the Flash data and writes it into the Flash
- We need to get the Flash utility code and the Flash data into RAM

Flash Programming Basics

Sequence of steps for Flash programming:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Erase</td>
<td>Set all bits to zero, then to one</td>
</tr>
<tr>
<td>2. Program</td>
<td>Program selected bits with zero</td>
</tr>
<tr>
<td>3. Verify</td>
<td>Verify flash contents</td>
</tr>
</tbody>
</table>

- Minimum Erase size is a sector (4Kw/8Kw/16Kw)
- Minimum Program size is a bit!
- Important not to lose power during erase step: If CSM passwords happen to be all zeros, the CSM will be permanently locked!
- Chance of this happening is quite small! (Erase step is performed sector by sector)
Programming Utilities and CCS Plug-in

Flash Programming Utilities

- Code Composer Studio Plug-in (uses JTAG)
- Third-party JTAG utilities
  - SDFlash JTAG from Spectrum Digital (requires SD emulator)
  - Signum System Flash utilities (requires Signum emulator)
- SDFlash Serial utility (uses SCI boot)
- Gang Programmers (use GPIO boot)
  - BP Micro programmer
  - Data I/O programmer
- Build your own custom utility
  - Use a different ROM bootloader method than SCI
  - Embed flash programming into your application
  - Flash API algorithms provided by TI

* TI web has links to all utilities (www.ti.com/c2000)
Code Security Module and Passwords

Code Security Module (CSM)

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

<table>
<thead>
<tr>
<th>Memory Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00 8000</td>
<td>LO SARAM (4Kw)</td>
</tr>
<tr>
<td>0x00 9000</td>
<td>L1 SARAM (4Kw)</td>
</tr>
<tr>
<td>0x00 A000</td>
<td>reserved</td>
</tr>
<tr>
<td>0x00 C000</td>
<td>otp (1Kw)</td>
</tr>
<tr>
<td>0x00 D000</td>
<td>reserved</td>
</tr>
<tr>
<td>0x00 E000</td>
<td>FLASH (64Kw)</td>
</tr>
<tr>
<td>0x00 F000</td>
<td>LO SARAM (4Kw)</td>
</tr>
<tr>
<td>0x3E 8000</td>
<td>L1 SARAM (4Kw)</td>
</tr>
<tr>
<td>0x3E 7FF8</td>
<td>128-bit Password</td>
</tr>
<tr>
<td>0x3F 8000</td>
<td>128-bit Password</td>
</tr>
<tr>
<td>0x3F 9000</td>
<td>128-bit Password</td>
</tr>
</tbody>
</table>

- 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

- Prevents reverse engineering and protects valuable intellectual property.

<table>
<thead>
<tr>
<th>Memory Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3E 8000</td>
<td>FLASH (64Kw)</td>
</tr>
<tr>
<td>0x3E 7FF8</td>
<td>128-bit Password</td>
</tr>
</tbody>
</table>

- 128-bit user defined password is stored in Flash.
- 128-bit Key Register used to lock and unlock the device:
  - Mapped in memory space 0x00 0AE0 – 0x00 0AE7
- 128-bits = $2^{128} = 3.4 \times 10^{38}$ possible passwords.
- To try 1 password every 8 cycles at 100 MHz, it would take at least $8.8 \times 10^{23}$ years to try all possible combinations!
**CSM Password Match Flow**

- **Start**
- Flash device secure after reset or runtime
- Do dummy read of PWL 0x3F 7FF8 – 0x3F 7FFF
- **Is PWL = all 0s?**
  - Yes: **Device permanently locked**
    - CPU access is limited – device cannot be debugged or reprogrammed
  - No: **Is PWL = all Fs?**
    - Yes: Write password to KEY registers 0x00 0AE0 – 0x00 0AE7 (EALLOW) protected
    - No: Correct password?
      - Yes: **Device unlocked**
        - User can access on-chip secure memory
      - No: No
Lab 5: Flash Programming

- **Objective**

The objective of this lab is to demonstrate the techniques discussed in this module and program the on-chip flash memory. The TMS320F2808 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.

- **System Programmed into Flash Memory**
- **Use CCS Flash Plug-in**
- **DO NOT PROGRAM PASSWORDS**

- **Procedure**

**Project File**

1. A project named Lab5.pjt has been created for this lab. Open the project by clicking on Project → Open... and look in C:\C28x1DAY\LABS\LAB5. All Build Options have been configured. The files used in this lab are:

- Main_5.c
- Lab.cdb
- Lab.cfg.cmd
- DSP280x_Headers_BIOS.cmd
- User_5.cmd
- CodeStartBranch.asm
- SysCtrl.c
- Gpio.c
- DSP280x_GlobalVariableDefs.c
- PieCtrl_5.c
- DefaultIsr_4_5.c
- Adc.c
- EPwm.c
- Filter.c
- DelayUs.asm
Link Initialized Sections to Flash

Initialized sections, such as code and constants, must contain valid values at device power-up. For a stand-alone embedded system with the F2808 device, these initialized sections must be linked to the on-chip flash memory. Note that a stand-alone embedded system must operate without an emulator or debugger in use, and no host processor is used to perform bootloading.

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

2. The RUN address of the following sections needs to run from flash. The memory section manager in the DSP/BIOS configuration tool (Lab.cdb) has been set to link the following sections to on-chip flash memory:

<table>
<thead>
<tr>
<th>BIOS Data tab</th>
<th>BIOS Code tab</th>
<th>Compiler Sections tab</th>
</tr>
</thead>
<tbody>
<tr>
<td>.gblinit</td>
<td>.bios</td>
<td>.text</td>
</tr>
<tr>
<td>.sysinit</td>
<td>.switch</td>
<td></td>
</tr>
<tr>
<td>.hwii</td>
<td>.cinit</td>
<td></td>
</tr>
<tr>
<td>.rtdx_text</td>
<td>.pinit</td>
<td>.econst / .const</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.data</td>
</tr>
</tbody>
</table>

3. The LOAD address of the following sections needs to load to flash. Again the memory section manager in the DSP/BIOS configuration tool (Lab.cdb) has been set to specify the load addresses. To view these setting, select the Load Address tab. Notice that the “Specify Separate Load Addresses” box has been checked and all the entries are set to the flash memory block.

Copying .hwi_vec Section from Flash to RAM

The DSP/BIOS .hwi_vec section contains the interrupt vectors. This section must be loaded to flash (load address) but run from RAM (run address). The code that performs this copy is located in InitPieCtrl(). The DSP/BIOS configuration tool generates global symbols that can be accessed by code in order to determine the load address, run address, and length of the .hwi_vec section.
The C-compiler runtime support library contains a memory copy function called `memcpy()` which will be used to perform the copy.

4. Open and inspect `InitPieCtrl()` (in `PieCtrl_5.c` file). Notice the `memcpy()` function and the symbols used to initialize (copy) the `.hwi_vec` section.

### Copying the `.trcdata` Section from Flash to RAM

The DSP/BIOS `.trcdata` section is used by CCS and DSP/BIOS for certain real-time debugging features. This section must be loaded to flash (load address) but run from RAM (run address). The DSP/BIOS configuration tool generates global symbols that can be accessed by code in order to determine the load address, run address, and length of the `.trcdata` section. The memory copy function `memcpy()` will again be used to perform the copy.

The copying of `.trcdata` must be performed prior to `main()`. This is because DSP/BIOS modifies the contents of `.trcdata` during DSP/BIOS initialization, which also occurs prior to `main()`. The DSP/BIOS configuration tool provides a user initialization function which will be used to perform the `.trcdata` section copy prior to both `main()` and DSP/BIOS initialization.

5. Open the DSP/BIOS configuration file (`Lab.cdb`) and select the `Properties` for the `Global Settings`. Notice the box “Call User Init Function” has been checked and the `UserInit()` function name with a leading underscore `_UserInit` is entered. This will cause the function `UserInit()` to execute prior to `main()`.

6. Open and inspect the file `Main_5.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.

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. Since the DSP/BIOS configuration tool does not know about user defined sections, the “secureRamFuncs” section will be linked using the user linker command file `User_5.cmd`. Open and inspect `User_5.cmd`. The “secureRamFuncs” will load to flash (load address) but will run from L1SARAM (run address). Also notice that the linker has been asked to generate symbols for the load start, load end, and run start addresses.

While not a requirement from a DSP hardware perspective (since the C28x DSP has a unified memory architecture), Code Composer Studio generally prefers code to be linked to program space (and data to be linked to data space). Therefore, notice that for the...
L1SARAM memory we are linking “secureRamFuncs” to, we are specifying “PAGE = 0” (which is program space).

9. Open and inspect Main_5.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. The following line of code in main() is used to call the InitFlash() function. Since there are no passed parameters or return values the code is just:

```c
InitFlash();
```

at the required spot in main().

Code Security Module and Passwords

The CSM module provides protection against unwanted copying (i.e. pirating!) of your code from flash, OTP memory, and the L0 and L1 RAM blocks. The CSM uses a 128-bit password made up of 8 individual 16-bit words. They are located in flash at addresses 0x3F7FF8 to 0x3F7FFF. During this lab, dummy passwords of 0xFFFF will be used – therefore only dummy reads of the password locations are needed to unsecure the CSM. **DO NOT PROGRAM ANY REAL PASSWORDS INTO THE DEVICE.** After development, real passwords are typically placed in the password locations to protect your code. We will not be using real passwords in the workshop.

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

11. 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).

12. Open User_5.cmd and notice that the initialized sections for “passwords” and “csm_rsvd” are linked to memories named PASSWORDS and CSM_RSVD, respectively. The DSP/BIOS configuration tool (Lab.cdb) defines memory blocks for PASSWORDS and CSM_RSVD.

Executing from Flash after Reset

The F2808 device contains a ROM bootloader that will transfer code execution to the flash after reset. When the boot mode selection pins are set for “Jump to Flash” mode, the bootloader will branch to the instruction located at address 0x3F7FF6 in the flash. An instruction that branches to the beginning of your program needs to be placed at this address. Note that the CSM passwords begin at address 0x3F7FF8. There are exactly two words available to hold this branch instruction, and not coincidentally, a long branch instruction “LB” in assembly code occupies exactly two words. Generally, the branch instruction will branch to the start of the C-
environment initialization routine located in the C-compiler runtime support library. The entry
symbol for this routine is _c_int00. Recall that C code cannot be executed until this setup routine
is run. Therefore, assembly code must be used for the branch. We are using the assembly code
file named CodeStartBranch.asm.

13. Open and inspect CodeStartBranch.asm. This file creates an initialized section
named “codestart” that contains a long branch to the C-environment setup routine. This section has been placed in memory using the DSP/BIOS configuration tool and the
memory space is named BEGIN_FLASH.

14. In the earlier lab exercises, the section “codestart” was directed to the memory
named BEGIN_M0. Open and inspect User_5.cmd and notice that the section
“codestart” will now be directed to BEGIN_FLASH.

15. The eZdsp™ board needs to be configured for “Jump to Flash” bootmode. Move switch
SW1 positions 1, 2 and 3 to the “open” position to accomplish this. Details of switch
positions can be found in Appendix A. This switch controls the pullup/down resistor on
the GPIO18, GPIO29, and GPIO34 pins, which are the pins sampled by the bootloader to
determine the bootmode. (For additional information on configuring the “Jump to Flash”
bootmode see the TMS320x280x DSP Boot ROM Reference Guide, and also the eZdsp
F2808 Technical Reference).

Build – Lab.out

16. 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 Load” 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.

17. Click the “Build” button to generate the Lab.out file to be used with the CCS Flash
Plug-in.

CCS Flash Plug-in

18. Open the Flash Plug-in tool by clicking :

Tools  ➔ F28xx On-Chip Flash Programmer

19. Notice that the eZdsp™ board uses a 20 MHz oscillator (located on the board near LED
DS2). Confirm the “Clock Configuration” in the upper left corner has the OSCCLK set
to 20 MHz and the PLLCR value is set to 10. Recall that the PLL is divided by two,
which gives a SYSCLKOUT of 100 MHz.
20. 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.

21. We will not be using the plug-in to program the “Code Security Password”. **Do not modify the Code Security Password fields.**

22. 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.

23. Click “Execute Operation” to program the flash memory. Watch the programming status update in the plug-in window.

24. After successfully programming the flash memory, close the programmer window.

**Running the Code – Using CCS**

25. 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 \(\rightarrow\) Load Symbols \(\rightarrow\) Load Symbols Only...

   and select Lab5.out in the Debug folder.

26. Reset the DSP. The program counter should now be at 0x3FFB50, which is the start of the bootloader in the Boot ROM.

27. 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 65 single-steps). Notice that we have placed some code in CodeStartBranch.asm to give an option to first disable the watchdog, if selected.

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

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

30. You can now RUN the DSP, and you should observe the LED on the board blinking. Try resetting the DSP and hitting RUN (without doing all the stepping and the Go Main procedure). The LED should be blinking again.
Running the Code – Stand-alone Operation (No Emulator)


32. Disconnect the emulator (USB cable) from the eZdsp™ board.

33. Remove the power from the board.

34. Re-connect the power to the board.

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

Return Switch SW1 Back to Default Positions

36. Remove the power from the board.

37. Please return the settings of switch SW1 back to the default positions “Jump to M0SARAM” bootmode (see Appendix A for switch position details):

<table>
<thead>
<tr>
<th>Position 3</th>
<th>Position 2</th>
<th>Position 1</th>
<th>Boot Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPIO18</td>
<td>GPIO29</td>
<td>GPIO34</td>
<td></td>
</tr>
<tr>
<td>Closed – 0</td>
<td>Open – 1</td>
<td>Closed – 0</td>
<td>MO SARAM</td>
</tr>
</tbody>
</table>

End of Exercise

Lab 5 Reference: BIOS Startup Sequence from Flash Memory

![BIOS Startup Sequence from Flash Memory Diagram](image)
The Next Step…

Development Support

**Comprehensive Tools and Support Set**

<table>
<thead>
<tr>
<th>Tools</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$F2808\text{ eZdsp}^\text{TM}$ developer’s kit from Spectrum Digital</td>
<td></td>
</tr>
<tr>
<td>Code Composer Studio\text{TM IDE for C2000}\text{TM}</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Software</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Application specific libraries</td>
<td></td>
</tr>
<tr>
<td>Math functions</td>
<td></td>
</tr>
<tr>
<td>Communications drivers</td>
<td></td>
</tr>
<tr>
<td>Pre-bundled system solutions</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Training and Support</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Control developers seminar</td>
<td></td>
</tr>
<tr>
<td>DMC workshop</td>
<td></td>
</tr>
<tr>
<td>One-day technical introduction to C28x™</td>
<td></td>
</tr>
<tr>
<td>Multi-day get started developing C28x™ workshop</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>High-Performance Analog</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Numerous data converter and power management products designed for motor control</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Third Party Network</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Development boards and emulation tools</td>
<td></td>
</tr>
<tr>
<td>Large consultant network</td>
<td></td>
</tr>
<tr>
<td>Increasing range of application software</td>
<td></td>
</tr>
</tbody>
</table>

**Recommended Next Step:**

**F2808\text{ eZdsp}^\text{TM}**

You now have the $F2808\text{ eZdsp}^\text{TM}$, so…

**Start Today!**

The $F2808\text{ eZdsp}^\text{TM}$ integrates the essential key features to allow users to understand the TMS320F2808 characteristics

**Hardware Features**

- TMS320F2808 Digital Signal Processor
- 18 Kwords on-chip RAM
- 64 Kwords on-chip Flash ROM
- 256K bits serial I2C EEPROM memory
- Expansion connectors
- Onboard IEEE 1149.1 JTAG controller
- 5 Volt only operation with supplied adapter
- On board USB JTAG emulation connector
- 2 SCI UART channels / 2 eCAN Channels

**Software**

- Includes TI’s C28xx Code Composer Studio

**Studio Debug Tools**

- Includes Flash programming utilities from Spectrum Digital

**Socketed Device**
Recommended Next Step: Multi-day Training Course

In-depth TMS320F2808 Design and Peripheral Training

C28x Signal Processing Libraries

<table>
<thead>
<tr>
<th>Signal Processing Libraries &amp; Applications Software</th>
<th>Literature #</th>
</tr>
</thead>
<tbody>
<tr>
<td>AG3-1: Control with Constant V/Hz</td>
<td>SPRC194</td>
</tr>
<tr>
<td>AC3-3: Sensored Indirect Flux Vector Control</td>
<td>SPRC207</td>
</tr>
<tr>
<td>AG3-3: Sensored Indirect Flux Vector Control (simulation)</td>
<td>SPRC208</td>
</tr>
<tr>
<td>AG3-4: Sensorless Direct Flux Vector Control</td>
<td>SPRC195</td>
</tr>
<tr>
<td>AG3-4: 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>BLDC3-1: Sensored Trapezoidal Control using Hall Sensors</td>
<td>SPRC213</td>
</tr>
<tr>
<td>BLDC3-2: 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>C280x C/C++ Header Files and Peripheral Examples</td>
<td>SPRC191</td>
</tr>
</tbody>
</table>

Available from TI DSP Website ⇒ http://www.ti.com/c2000
## Customers Are Using C2000™ Products For ...

<table>
<thead>
<tr>
<th>Motor Control</th>
<th>Digital Power</th>
<th>Optical Networking</th>
<th>Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heart/lung machines</td>
<td>Vacuum cleaners</td>
<td>Blood analyzers</td>
<td>Adaptive cruise control</td>
</tr>
<tr>
<td>Human transporters</td>
<td>Vibration control</td>
<td>Data encryption systems</td>
<td>Airbag control</td>
</tr>
<tr>
<td>Industrial drives</td>
<td>Wafer testers</td>
<td>E-meters</td>
<td>Antitheft systems</td>
</tr>
<tr>
<td>Inverters</td>
<td>Washing machines</td>
<td>Gas sensors</td>
<td>... and many more</td>
</tr>
<tr>
<td>Lawn mowers</td>
<td>Windmill control</td>
<td>GPS systems</td>
<td></td>
</tr>
<tr>
<td>Magnetic bearings</td>
<td></td>
<td>Ignition control</td>
<td></td>
</tr>
<tr>
<td>Mass flow control</td>
<td></td>
<td>Induction ovens</td>
<td></td>
</tr>
<tr>
<td>Medical pumps</td>
<td></td>
<td>Park assist systems</td>
<td></td>
</tr>
<tr>
<td>Missile control</td>
<td></td>
<td>Power line mods</td>
<td></td>
</tr>
<tr>
<td>Plotters</td>
<td></td>
<td>Radar control</td>
<td></td>
</tr>
<tr>
<td>Postal sorters</td>
<td></td>
<td>Reactor monitoring</td>
<td></td>
</tr>
<tr>
<td>Power assisted steering</td>
<td></td>
<td>RF ID systems</td>
<td></td>
</tr>
<tr>
<td>Power drills</td>
<td></td>
<td>Spectrum analyzers</td>
<td></td>
</tr>
<tr>
<td>Power generators</td>
<td></td>
<td>Telecom switches</td>
<td></td>
</tr>
<tr>
<td>Printers</td>
<td></td>
<td>Tire pressure sensing</td>
<td></td>
</tr>
<tr>
<td>Refrigerators</td>
<td></td>
<td>Ultrasound scanners</td>
<td></td>
</tr>
<tr>
<td>Robot control</td>
<td></td>
<td>Welding equipment</td>
<td></td>
</tr>
<tr>
<td>Sewing machines</td>
<td></td>
<td>Wireless modems</td>
<td></td>
</tr>
<tr>
<td>Ship propulsion control</td>
<td></td>
<td>Color/light sensors</td>
<td></td>
</tr>
<tr>
<td>Stepper motor control</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Textile machines</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Toy trains</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Treadmills</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### For More Information ...

**Internet**

**Website:** [http://www.ti.com](http://www.ti.com)

**FAQ:** [http://www-k.ext.ti.com/sc/technical_support/knowledgebase.htm](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](http://www.ti.com/sc/training)

**USA - Product Information Center (PIC)**

**Phone:** 800-477-8924 or 972-644-5580

**Email:** support@ti.com

- Information and support for all TI Semiconductor products/tools
- Submit suggestions and errata for tools, silicon and documents
### European Product Information Center (EPIC)

<table>
<thead>
<tr>
<th>Language</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Belgium (English)</td>
<td>+32 (0) 27 45 55 32</td>
</tr>
<tr>
<td>France</td>
<td>+33 (0) 1 30 70 11 64</td>
</tr>
<tr>
<td>Germany</td>
<td>+49 (0) 8161 80 33 11</td>
</tr>
<tr>
<td>Israel (English)</td>
<td>1800 949 0107 (free phone)</td>
</tr>
<tr>
<td>Italy</td>
<td>800 79 11 37 (free phone)</td>
</tr>
<tr>
<td>Netherlands (English)</td>
<td>+31 (0) 546 87 95 45</td>
</tr>
<tr>
<td>Spain</td>
<td>+34 902 35 40 28</td>
</tr>
<tr>
<td>Sweden (English)</td>
<td>+46 (0) 8587 555 22</td>
</tr>
<tr>
<td>United Kingdom</td>
<td>+44 (0) 1604 66 33 99</td>
</tr>
<tr>
<td>Finland (English)</td>
<td>+358 (0) 9 25 17 39 48</td>
</tr>
</tbody>
</table>

**Fax:** All Languages: +49 (0) 8161 80 2045

**Email:** epic@ti.com

- Literature, Sample Requests and Analog EVM Ordering
- Information, Technical and Design support for all Catalog TI Semiconductor products/tools
- Submit suggestions and errata for tools, silicon and documents
Appendix A – eZdsp™ F2808

Note: This appendix only provides a description of the eZdsp™ F2808 interfaces used in this workshop. For a complete description of all features and details, please see the eZdsp™ F2808 USB Technical Reference manual.
Module Topics

Appendix

Module Topics

Appendix .................................................................................................................................................... A-1

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

eZdsp™ F2808 ........................................................................................................................................ A-3
  eZdsp™ F2808 Connector / Header and Pin Diagram ........................................................................ A-3
  P8 – I/O Interface ................................................................................................................................. A-4
  P5 / P9 – Analog Interface .................................................................................................................... A-5
  SW1 – Switch ......................................................................................................................................... A-6
  DS1 / DS2 – LEDs ................................................................................................................................. A-7
  TP1 / TP2 – Test Points ......................................................................................................................... A-7
eZdsp™ F2808

eZdsp™ F2808 Connector / Header and Pin Diagram

Table 1: eZdsp™ F2808 Connectors

<table>
<thead>
<tr>
<th>Connector</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>JTAG Interface</td>
</tr>
<tr>
<td>P8</td>
<td>I/O Interface</td>
</tr>
<tr>
<td>P5/P9</td>
<td>Analog Interface</td>
</tr>
<tr>
<td>P6</td>
<td>Power Connector</td>
</tr>
<tr>
<td>P10</td>
<td>DB-9, RS-232</td>
</tr>
<tr>
<td>P11</td>
<td>DB9, eCAN-A</td>
</tr>
<tr>
<td>J10</td>
<td>2x5 Header, SCIB</td>
</tr>
<tr>
<td>J11</td>
<td>2x5 header, eCAN-B</td>
</tr>
<tr>
<td>J201</td>
<td>USB Controller Interface</td>
</tr>
</tbody>
</table>
## P8 – I/O Interface

<table>
<thead>
<tr>
<th>P8 Pin #</th>
<th>P8 Signal</th>
<th>P8 Pin #</th>
<th>P8 Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>+3.3V/+5V/NC *</td>
<td>2</td>
<td>+3.3V/+5V/NC *</td>
</tr>
<tr>
<td>3</td>
<td>MUX_GPIO029</td>
<td>4</td>
<td>MUX_GPIO028</td>
</tr>
<tr>
<td>5</td>
<td>GPIO14</td>
<td>6</td>
<td>GPIO20</td>
</tr>
<tr>
<td>7</td>
<td>GPIO21</td>
<td>8</td>
<td>GPIO23</td>
</tr>
<tr>
<td>9</td>
<td>GPIO00</td>
<td>10</td>
<td>GPIO31</td>
</tr>
<tr>
<td>11</td>
<td>GPIO22</td>
<td>12</td>
<td>GPIO34</td>
</tr>
<tr>
<td>13</td>
<td>GPIO4</td>
<td>14</td>
<td>GPIO35</td>
</tr>
<tr>
<td>15</td>
<td>GPIO27</td>
<td>16</td>
<td>GPIO36</td>
</tr>
<tr>
<td>17</td>
<td>GPIO13</td>
<td>18</td>
<td>GPIO38</td>
</tr>
<tr>
<td>19</td>
<td>GND</td>
<td>20</td>
<td>GND</td>
</tr>
<tr>
<td>21</td>
<td>GPIO7</td>
<td>22</td>
<td>GPIO10</td>
</tr>
<tr>
<td>23</td>
<td>GPIO16</td>
<td>24</td>
<td>GPIO10</td>
</tr>
<tr>
<td>25</td>
<td>GPIO18</td>
<td>26</td>
<td>GPIO10</td>
</tr>
<tr>
<td>27</td>
<td>MUX_GPIO081</td>
<td>28</td>
<td>MUX_GPIO080</td>
</tr>
<tr>
<td>29</td>
<td>MUX_GPIO11</td>
<td>30</td>
<td>MUX_GPIO08</td>
</tr>
<tr>
<td>31</td>
<td>MUX_GPIO09</td>
<td>32</td>
<td>MUX_GPIO10</td>
</tr>
<tr>
<td>33</td>
<td>GPIO22_GPIO24</td>
<td>34</td>
<td>GPIO25</td>
</tr>
<tr>
<td>35</td>
<td>GPIO26</td>
<td>36</td>
<td>GPIO32</td>
</tr>
<tr>
<td>37</td>
<td>GPIO12</td>
<td>38</td>
<td>GPIO33</td>
</tr>
<tr>
<td>39</td>
<td>GND</td>
<td>40</td>
<td>GND</td>
</tr>
</tbody>
</table>

* Default is No Conn (NC). User can jumper to +3.3V or +5V on backside of ezdsp with JP4.
Appendix

P5 / P9 – Analog Interface

Table 3: P5/P9, Analog Interface Connector

<table>
<thead>
<tr>
<th>P5 Pin #</th>
<th>Signal</th>
<th>P9 Pin #</th>
<th>Signal</th>
<th>P9 Pin #</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ADCINB0</td>
<td>1</td>
<td>GND</td>
<td>2</td>
<td>ADCINA0</td>
</tr>
<tr>
<td>2</td>
<td>ADCINB1</td>
<td>3</td>
<td>GND</td>
<td>4</td>
<td>ADCINA1</td>
</tr>
<tr>
<td>3</td>
<td>ADCINB2</td>
<td>5</td>
<td>GND</td>
<td>6</td>
<td>ADCINA2</td>
</tr>
<tr>
<td>4</td>
<td>ADCINB3</td>
<td>7</td>
<td>GND</td>
<td>8</td>
<td>ADCINA3</td>
</tr>
<tr>
<td>5</td>
<td>ADCINB4</td>
<td>9</td>
<td>GND</td>
<td>10</td>
<td>ADCINA4</td>
</tr>
<tr>
<td>6</td>
<td>ADCINB5</td>
<td>11</td>
<td>GND</td>
<td>12</td>
<td>ADCINA5</td>
</tr>
<tr>
<td>7</td>
<td>ADCINB6</td>
<td>13</td>
<td>GND</td>
<td>14</td>
<td>ADCINA6</td>
</tr>
<tr>
<td>8</td>
<td>ADCINB7</td>
<td>15</td>
<td>GND</td>
<td>16</td>
<td>ADCINA7</td>
</tr>
<tr>
<td>9</td>
<td>ADCREFM</td>
<td>17</td>
<td>GND</td>
<td>18</td>
<td>VREFLO *</td>
</tr>
<tr>
<td>10</td>
<td>ADCREFP</td>
<td>19</td>
<td>GND</td>
<td>20</td>
<td>No connect</td>
</tr>
</tbody>
</table>

* Connect VREFLO to AGND or VREFL0 of target system for proper ADC operation.
### SW1 – Switch

<table>
<thead>
<tr>
<th>Position</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-3</td>
<td>Boot Mode Select</td>
</tr>
<tr>
<td>4</td>
<td>Serial EEPROM WP</td>
</tr>
<tr>
<td>5</td>
<td>Serial EEPROM Pullups</td>
</tr>
<tr>
<td>6</td>
<td>Serial EEPROM - A1</td>
</tr>
</tbody>
</table>
Table 5: SW1, Positions 1-3

<table>
<thead>
<tr>
<th>Position 3 GPIO18</th>
<th>Position 2 GPIO29</th>
<th>Position 1 GPIO34</th>
<th>Boot Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open-1</td>
<td>Open-1</td>
<td>Open-1</td>
<td>Flash</td>
</tr>
<tr>
<td>Open-1</td>
<td>Open-1</td>
<td>Closed-0</td>
<td>SCI-A</td>
</tr>
<tr>
<td>Open-1</td>
<td>Closed-0</td>
<td>Open-1</td>
<td>SPI-A</td>
</tr>
<tr>
<td>Open-1</td>
<td>Closed-0</td>
<td>Closed-0</td>
<td>I²C-A</td>
</tr>
<tr>
<td>Closed-0</td>
<td>Open-1</td>
<td>Open-1</td>
<td>eCAN-A</td>
</tr>
<tr>
<td>Closed-0</td>
<td>Open-1</td>
<td>Closed-0</td>
<td>M0 SARAM *</td>
</tr>
<tr>
<td>Closed-0</td>
<td>Closed-0</td>
<td>Open-1</td>
<td>OTP</td>
</tr>
<tr>
<td>Closed-0</td>
<td>Closed-0</td>
<td>Closed-0</td>
<td>I/O</td>
</tr>
</tbody>
</table>

* factory default

DS1 / DS2 – LEDs

Table 6: LEDs

<table>
<thead>
<tr>
<th>LED #</th>
<th>Color</th>
<th>Controlling Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS1</td>
<td>Green</td>
<td>+5 Volts</td>
</tr>
<tr>
<td>DS2</td>
<td>Green</td>
<td>GPIO34 bit (GPIO34 high = on)</td>
</tr>
</tbody>
</table>

J1 / J2 – Test Points

Table 7: Test Points

<table>
<thead>
<tr>
<th>Test Point</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>J1</td>
<td>Analog Ground</td>
</tr>
<tr>
<td>J2</td>
<td>Ground</td>
</tr>
</tbody>
</table>