Important Notice

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

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

Customers are responsible for their applications using TI components.

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

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

Copyright © 2009 – 2014 Texas Instruments Incorporated

Revision History

April 2009 – Revision 1.0
October 2009 – Revision 1.1
June 2010 – Revision 2.0
December 2010 – Revision 2.1
October 2011 – Revision 3.0
February 2012 – Revision 4.0
January 2013 – Revision 4.1
January 2014 – Revision 4.2

Mailing Address

Texas Instruments
Training Technical Organization
6500 Chase Oaks Blvd Building 2
M/S 8437
Plano, Texas 75023
Workshop Topics

Workshop Topics ................................................................................................................................. 3
Workshop Introduction .......................................................................................................................... 4
Architecture Overview ............................................................................................................................ 11
Programming Development Environment ............................................................................................ 16
  Code Composer Studio ......................................................................................................................... 16
  Linking Sections in Memory .................................................................................................................. 20
Lab 1: Linker Command File .................................................................................................................. 24
Peripheral Register Header Files .......................................................................................................... 29
Reset, Interrupts and System Initialization ............................................................................................ 39
  Reset .................................................................................................................................................... 39
  Interrupts ............................................................................................................................................ 44
  Peripheral Interrupt Expansion (PIE) .................................................................................................... 47
  Oscillator / PLL Clock Module ............................................................................................................ 52
  Watchdog Timer Module ..................................................................................................................... 54
  GPIO .................................................................................................................................................... 56
Lab 2: System Initialization ..................................................................................................................... 60
Control Peripherals ............................................................................................................................... 65
  ADC Module ....................................................................................................................................... 65
  Pulse Width Modulation ...................................................................................................................... 69
  ePWM ............................................................................................................................................... 71
  eCAP ................................................................................................................................................ 87
  eQEP ................................................................................................................................................ 89
Lab 3: Control Peripherals ....................................................................................................................... 92
Flash Programming ............................................................................................................................... 98
  Flash Programming Basics ................................................................................................................ 98
  Programming Utilities and CCS Flash Programmer .......................................................................... 100
  Code Security Module and Password ............................................................................................... 101
Lab 4: Programming the Flash ............................................................................................................. 104
The Next Step ....................................................................................................................................... 111
  Training ............................................................................................................................................ 111
  controlSUITE ................................................................................................................................ 112
  Development Tools ............................................................................................................................. 113
  C2000 Workshop Download Wiki .................................................................................................... 116
  Development Support ......................................................................................................................... 117
Notes: .................................................................................................................................................... 118
The objective of this workshop is to gain a fully understand and a complete working knowledge of the C2000 microcontroller. This will be accomplished through detailed presentations and hands-on lab exercises.

The workshop will start with the basic topics and progress to more advanced topics in a logical flow such that each topic and lab exercise builds on the previous one presented. At the end of the workshop, you should be confident in applying the skills learned in your product design.
The outline for this workshop starts with an introduction covering the materials required and provides a brief overview of the C2000 product family. Next, we will look at the architectural details and discuss the block diagram, memory map, and peripherals. From there, we will move to the programming development environment and gain the basic skills needed to use Code Composer Studio, as well as learn about a linker command file.

Then we will have a lab exercise to practice the skills learned thus far. We will follow this with the peripheral register header files. At this point, we have covered the basic foundation. Next, we will explore the device and learn about reset, interrupts, and system initialization. In this lab exercise, we will use the watchdog to generate a reset and interrupt, reinforcing what we have just learned.

Then we will cover the control peripherals, which includes the ADC, ePWM, eCAP, and eQEP. In this lab exercise, we will generate a PWM waveform, then feed it into the ADC, and graph it using Code Composer Studio. Additionally, we will learn about the real-time mode emulation features.

The next topic will be programming the flash. In this lab exercise, we will develop a complete embedded system using the code from the previous lab exercises. Finally, in the “Next Step”, we will discuss where you can find more information, allowing you to continue learning and exploring on your own.
The materials required for this workshop are available using the links shown at the top of this slide. Please be sure that you have the F28069 controlSTICK kit. The included jumper wire will be needed for the lab exercises. Make sure that you have all of the software installed. The lab directions are written based on the version of Code Composer Studio as shown on this slide. The workshop installer will automatically install the lab files, solution files, workshop manual, and documentation.
The Texas Instruments embedded processing portfolio covers a wide range of embedded processors – from microcontrollers, to ARM-based processors, to digital signal processors. Our microcontrollers include the 16-bit ultra-low power MSP430 microcontroller family and, the 32-bit real-time C2000 microcontroller family. Our ARM-based processors include the 32-bit ARM Cortex M3 Stellaris microcontroller family, the ARM Cortex A8 Sitara microprocessor family, and it also includes the OMAP devices, which incorporates a C6000 and ARM processor. Our digital signal processors include the C6000 DaVinci, as well as multicore DSPs with up to 24 billion multiply accumulates per second. Additionally, there's an ultra-low power C5000 digital signal processor family.
The C2000 family, which this workshop is based on, includes the C28x Piccolo, C28x Delfino, and C28x + Cortex-M3 product lines. The Piccolo product line ranges in performance from 40 MHz to 90 MHz and has options for a floating point unit, control law accelerator coprocessor, a Viterbi complex math CRC unit, and USB. The Delfino product line ranges in performance from 100 MHz to 300 MHz and features a floating point unit and some devices with an external memory interface. The C28x + Cortex-M3 product line consists of a dual-system architecture, incorporating a C28X CPU with performance up to 150 MHz and an ARM Cortex M3 CPU with performance up to 100 MHz. In addition to floating point capabilities, it has a Viterbi complex math CRC unit. The three product lines provide you with over 150 devices to choose from and maintain software compatibility.
The C2000 product family has a very broad application base with target markets in motor control, lighting, smart grid and power line communications, automotive, digital power, and renewable energy.

<table>
<thead>
<tr>
<th>C2000 Delfino / Piccolo Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Clock</strong></td>
</tr>
<tr>
<td><strong>Flash / RAM</strong></td>
</tr>
<tr>
<td><strong>On-chip Oscillators</strong></td>
</tr>
<tr>
<td><strong>VREG / POR / BOR</strong></td>
</tr>
<tr>
<td><strong>Watchdog Timer</strong></td>
</tr>
<tr>
<td><strong>12-bit ADC</strong></td>
</tr>
<tr>
<td><strong>Analog COMP w/ DAC</strong></td>
</tr>
<tr>
<td><strong>FPU</strong></td>
</tr>
<tr>
<td><strong>6-Channel DMA</strong></td>
</tr>
<tr>
<td><strong>CLA</strong></td>
</tr>
<tr>
<td><strong>VCU</strong></td>
</tr>
<tr>
<td><strong>ePWM / HR ePWM</strong></td>
</tr>
<tr>
<td><strong>eCAP / HR eCAP</strong></td>
</tr>
<tr>
<td><strong>eQEP</strong></td>
</tr>
<tr>
<td><strong>SCI / SPI / I2C</strong></td>
</tr>
<tr>
<td><strong>LIN</strong></td>
</tr>
<tr>
<td><strong>McBSP</strong></td>
</tr>
<tr>
<td><strong>USB</strong></td>
</tr>
<tr>
<td><strong>External Interface</strong></td>
</tr>
</tbody>
</table>

When comparing the Delfino and Piccolo product lines, you will notice that the Piccolo F2806x
devices share many features with the Delfino product line. The Delfino product line is shown in the table by the F2833x column; therefore, the F28069, being the most feature-rich Piccolo device, was chosen as the platform for this workshop. The knowledge learned from this device will be applicable to all C2000 product lines.

The development tool for this workshop will be the TMS320F28069 controlSTICK kit. This controlSTICK is a self-contained system that plugs into a free USB port on your computer. The USB port provides power, as well as communicates to the onboard JTAG emulation controller. LED LD1 illuminates when the board is powered, and LED LD2 is connected to GPIO34. We will be using this LED as a visual indicator during the lab exercises. Some of the I/O lines from the F28069 device are pinned out to the peripheral header. We will be using the included jumper wire to connect various I/O lines on this header.
This block diagram represents an overview of all device features and is not specific to any one device. The F28069 device is designed around a multibus architecture, also known as a modified Harvard architecture. This can be seen in the block diagram by the separate program bus and data bus, along with the link between the two buses. This type of architecture greatly enhances the performance of the device.

In the upper left area of the block diagram, you will find the memory section, which consists of the boot ROM, sectored flash, and RAM. Also, you will notice that the six-channel DMA has its own set of buses.

In the lower left area of the block diagram, you will find the execution section, which consists of a 32-bit by 32-bit hardware multiplier, a read-modify-write atomic ALU, a floating-point unit, and a Viterbi complex math CRC unit. The control law accelerator coprocessor is an independent and separate unit that has its own set of buses.

The peripherals are grouped on the right side of the block diagram. The upper set is the control peripherals, which consists of the ePWM, eCAP, eQEP, and ADC. The lower set is the communication peripherals and consists of the multichannel buffered serial port, I2C, SCI, SPI, CAN, and USB.

The PIE block, or Peripheral Interrupt Expansion block, manages the interrupts from the peripherals. In the bottom right corner is the general-purpose I/O. Also, the CPU has a watchdog module and three 32-bit general-purpose timers available.
The F28069 utilizes a contiguous memory map, also known as a von-Neumann architecture. This type of memory map lends itself well to higher-level languages. This can be seen by the labels located at the top of the memory map where the memory blocks extend between both the data space and program space.

At the top of the map, we have two blocks of RAM called M0 and M1. Then we see PF0 through PF3, which are the peripheral frames. This is the area where you will find the peripheral registers. Also in this space, you will find the PIE block. Memory blocks L0 through L3 are accessible by the CPU and CLA. L5 through L8 are accessible by the DMA.

The user OTP is a one-time, programmable, memory block. TI reserves a small space in the map for the ADC and oscillator calibration data. The flash block contains a section for passwords, which are used by the code security module. The boot ROM and boot ROM vectors are located at the bottom of the memory map.
The C2000 devices feature a very fast interrupt response manager using the PIE block. This allows up to 96 possible interrupt vectors to be processed by the CPU. We will cover more details about this in the reset, interrupts, and system initialization module.

The DMA allows data to be transferred between peripherals and/or memory without intervention from the CPU.
from the CPU. The DMA can read data from the ADC result registers, transfer to or from memory blocks L5 through L8, transfer to or from the McBSP, and also modify registers in the ePWM. Triggers are used to initiate the transfers, and when completed the DMA can generate an interrupt.

The control law accelerator is an independent, 32-bit, floating-point, math accelerator. It executes algorithms independently and in parallel with the CPU. It has direct access to the ePWM, high-resolution PWM, eCAP, eQEP, ADC result and comparator registers. It responds to peripheral interrupts independently of the CPU and frees up the CPU for other tasks, such as communications and diagnostics.
Architecture Overview

Viterbi, Complex Math, CRC Unit (VCU)

Extends C2000 instruction set to support:

- Viterbi operations
- Decode for communications
- Complex math
  - 16-bit fixed-point complex FFT (5 cycle butterfly)
    - used in spread spectrum communications, and many signal processing algorithms
  - Complex filters
    - used to improve data reliability, transmission distance, and power efficiency
  - Power Line Communications (PLC) and radar applications
- Cyclic Redundancy Check (CRC)
  - Communications and memory robustness checks

The Viterbi complex math CRC unit extends the C2000 instruction set to support Viterbi operations used in communications; complex math, which includes complex FFTs and complex filters, and is used in power line communications and radar applications; and cyclical redundancy check, which is used in communications and memory robustness checks. Below lists a summary of the major architectural features covered in this module.

Architecture Summary

- High performance 32-bit CPU
- 32x32 bit or dual 16x16 bit MAC
- IEEE single-precision floating point unit (FPU)
- Hardware Control Law Accelerator (CLA)
- Viterbi, complex math, CRC unit (VCU)
- Atomic read-modify-write instructions
- Fast interrupt response manager
- 128Kw on-chip flash memory
- Code security module (CSM)
- Control peripherals
- 12-bit ADC module
- Comparators
- Direct memory access (DMA)
- Up to 54 shared GPIO pins
- Communications peripherals
Code Composer Studio is an integrated development environment that integrates the edit, code generation, and debug process. It is simple to use, with single-click access to various functions, and provides powerful debugging tools. Scripts automate tasks and BIOS is built-in. Code Composer Studio is based on the Eclipse open source software framework.
Code Composer Studio has “Edit” and “Debug” perspectives. Each perspective provides a set of functionality aimed at accomplishing a specific task. In the edit perspective, views used during code development are displayed. In the debug perspective, views used during debug are displayed.
A project contains files, such as C and assembly source files, libraries, BIOS configuration files, and linker command files. It also contains project settings, such as build options, which include the compiler, assembler, linker, and BIOS, as well as build configurations.

Creating a New CCSv5 Project

1. Project Name, Location, and Device

2. Advanced Settings

3. Project Templates and Examples

To create a project in Code Composer Studio, you would click on File → New → CCS Project.
First, you would fill in the project name, the location, and the device. Then you would complete the advanced settings and, finally, the project template. You will have a chance to try this in our first lab exercise.

CCSv5 Build Options – Compiler / Linker

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

- Linker
  - 8 categories for linking
  - Specify various link options
  - `$\{PROJECT_ROOT\}` specifies the current project directory

After a project is created, the build options are configured. In our lab exercise, we will look at the options for the compiler and linker.
Linking Sections in Memory

All code consists of different parts called sections. All default section names begin with a dot and are typically lower case. The compiler has default section names for initialized and uninitialized sections. For example, x and y are global variables, and they are placed in the section .ebss. Whereas 2 and 7 are initialized values, and they are placed in the section called .cinit. The local variables are in a section .stack, and the code is placed in a section called .text.
This is a small list of compiler default section names. The top group is initialized sections, and they are linked to flash. In our previous code example, we saw .txt was used for code, and .cinit for initialized values. The bottom group is uninitialized sections, and they are linked to RAM. Once again, in our previous example, we saw .ebss used for global variables and .stack for local variables.
Next, we need to place the sections that were created by the compiler into the appropriate memory spaces. The uninitialized sections, .ebss and .stack, need to be placed into RAM; while the initialized sections, .cinit, and .txt, need to be placed into flash.

The linker command file describes the physical hardware memory and specifies where the
sections are placed in the memory. The file created during the link process is a .out file. This is the file that will be loaded into the microcontroller. As an option, we can generate a map file. This map file will provide a summary of the link process, such as the absolute address and size of each section.

```
Linker Command File

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

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

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

A linker command file consists of two sections, a memory section and a sections section. In the memory section, page 0 defines the program memory space, and page 1 defines the data memory space. Each memory block is given a unique name, along with its origin and length. In the sections section, the section is directed to the appropriate memory block. In the lab exercise, you will have an opportunity to work with a linker command file.

Next, in the lab exercise, you will be working with a linker command file and will learn the basic skills for using Code Composer Studio. You will set up a target configuration, create a project, build a project, and step through the code.
Lab 1: Linker Command File

Objective

Use a linker command file to link the C program file (Lab1.c) into the system described below.

Lab 1: Linker Command File

System Description:
• TMS320F28069
• All internal RAM blocks allocated

Placement of Sections:
• .text into RAM Block L4SARAM on PAGE 0 (program memory)
• .cinit into RAM Block L4SARAM on PAGE 0 (program memory)
• .ebss into RAM Block M0SARAM on PAGE 1 (data memory)
• .stack into RAM Block M1SARAM on PAGE 1 (data memory)

Initial Hardware Set Up

Plug the F28069 controlSTICK into the computer USB port. This will power the controlSTICK using the power supplied by the computer USB port. Additionally, this USB port will provide the JTAG communication link between the device and Code Composer Studio.

Initial Software Set Up

Code Composer Studio must be installed in addition to the workshop files. A local copy of the required controlSUITE files is included with the lab files. This provides portability, making the workshop files self-contained and independent of other support files or resources. The lab directions for this workshop are based on all software installed in their default locations.

Procedure

Start Code Composer Studio and Open a Workspace

1. Start Code Composer Studio (CCS) by double clicking the icon on the desktop or selecting it from the Windows Start menu. When CCS loads, a dialog box will prompt you for the location of a workspace folder. Use the default location for the workspace and click OK.
This folder contains all CCS custom settings, which includes project settings and views when CCS is closed so that the same projects and settings will be available when CCS is opened again. The workspace is saved automatically when CCS is closed.

2. The first time CCS opens a “Welcome to Code Composer Studio v5” page appears. Close the page by clicking the X on the “TI Resource Explorer” tab. You should now have an empty workbench. The term workbench refers to the desktop development environment. Maximize CCS to fill your screen.

The workbench will open in the “CCS Edit Perspective” view. Notice the CCS Edit icon in the upper right-hand corner. A perspective defines the initial layout views of the workbench windows, toolbars, and menus which are appropriate for a specific type of task (i.e. code development or debugging). This minimizes clutter to the user interface. The “CCS Edit Perspective” is used to create or build projects. A “CCS Debug Perspective” view will automatically be enabled when the debug session is started. This perspective is used for debugging projects.

Setup Target Configuration

3. Open the emulator target configuration dialog box. On the menu bar click:

File → New → Target Configuration File

In the file name field type F28069_ctrlSTK.ccxml. This is just a descriptive name since multiple target configuration files can be created. Leave the “Use shared location” box checked and select Finish.

4. In the next window that appears, select the emulator using the “Connection” pull-down list and choose “Texas Instruments XDS100v1 USB Emulator”. In the “Board or Device” box type F28069 to filter the options. In the box below, check the box to select “controlSTICK – Piccolo F28069”. Click Save to save the configuration, then close the “F28069_ctrlSTK.ccxml” setup window by clicking the X on the tab.

5. To view the target configurations, click:

View → Target Configurations

and click the plus sign (+) to the left of User Defined. Notice that the F28069_ctrlSTK.ccxml file is listed and set as the default. If it is not set as the default, right-click on the .ccxml file and select “Set as Default”. Close the Target Configurations window by clicking the X on the tab.

Create a New Project

6. A project contains all the files you will need to develop an executable output file (.out) which can be run on the MCU hardware. To create a new project click:

File → New → CCS Project

In the Project name field type Lab1. Uncheck the “Use default location” box. Click the Browse… button and navigate to:

C:\C28x\Labs\Lab1\Project
Click OK.

7. The next section selects the device. Select the “Family” using the pull-down list and choose “C2000”. Set the “Variant” filter using the pull-down list to “2806x Piccolo” and choose the “controlSTICK – Piccolo F28069”. Leave the “Connection” box blank. We have already set up the target configuration.

8. Next, open the “Advanced settings” section and set the “Linker command file” to “<none>”. We will be using our own linker command file, rather than the one supplied by CCS. Leave the “Runtime Support Library” set to “<automatic>”. This will automatically select the “rts2800_fpu32.lib” runtime support library for floating-point devices.

9. Now open the “Project templates and examples” section and select the “Empty Project” template. Click Finish.

10. A new project has now been created. Notice the Project Explorer window contains Lab1. The project is set Active and the output files will be located in the Debug folder. At this point, the project does not include any source files. The next step is to add the source files to the project.

11. To add the source files to the project, right-click on Lab1 in the Project Explorer window and select:
    Add Files… or click: Project → Add Files…

    and make sure you’re looking in C:\C28x\Labs\Lab1\Files. With the file type set to view all files (*.*) select Lab1.c and Lab1.cmd then click Open. A “File Operation” window will open, choose “Copy files” and click OK. This will add the files to the project.

12. In the Project Explorer window, click the plus sign (+) to the left of Lab1 and notice that the files are listed.

Project Build Options

13. There are numerous build options in the project. Most default option settings are sufficient for getting started. We will inspect a couple of the default options at this time. Right-click on Lab1 in the Project Explorer window and select Properties or click:
    Project → Properties

14. A “Properties” window will open and in the section on the left under “Build” be sure that the “C2000 Compiler” and “C2000 Linker” options are visible. Next, under “C2000 Linker” select the “Basic Options”. Notice that .out and .map files are being specified. The .out file is the executable code that will be loaded into the MCU. The .map file will contain a linker report showing memory usage and section addresses in memory. Also notice the stack size is set to 0x300.

15. Under “C2000 Compiler” select the “Processor Options”. Notice the “Use large memory model” and “Unified memory” boxes are checked. Next, notice the “Specify CLA support” is set to cla0, the “Specify floating point support” is set to fpu32, and the “Specify VCU support” is set to vcu0. Select OK to close the Properties window.
**Linker Command File – Lab1.cmd**

16. Open and inspect `Lab1.cmd` by double clicking on the filename in the Project Explorer window. Notice that the `Memory{}` declaration describes the system memory shown on the “Lab1: Linker Command File” slide in the objective section of this lab exercise. Memory blocks L3DPSARAM and L4SARAM have been placed in program memory on page 0, and the other memory blocks have been placed in data memory on page 1.

17. In the `Sections{}` area notice that the sections defined on the slide have been “linked” into the appropriate memories. Also, notice that a section called `.reset` has been allocated. The `.reset` section is part of the rts2800_fpu32.lib and is not needed. By putting the `TYPE = DSECT` modifier after its allocation the linker will ignore this section and not allocate it. Close the inspected file.

**Build and Load the Project**

18. Two 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>Build</td>
<td>Full build and link of all source files</td>
</tr>
<tr>
<td>2</td>
<td>Debug</td>
<td>Automatically build, link, load and launch debug-session</td>
</tr>
</tbody>
</table>

19. Click the “Build” button and watch the tools run in the Console window. Check for errors in the Problems window (we have deliberately put an error in `Lab1.c`). When you get an error, you will see the error message in the Problems window. Expand the error by clicking on the plus sign (+) to the left of the “Errors”. Then simply double-click the error message. The editor will automatically open to the source file containing the error, with the code line highlighted with a question mark (?).

20. 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.

21. Build the project again. There should be no errors this time.

22. CCS can automatically save modified source files, build the project, open the debug perspective view, connect and download it to the target, and then run the program to the beginning of the main function.

   Click on the “Debug” button (green bug) or click Run → Debug.

   Notice the CCS Debug icon in the upper right-hand corner indicating that we are now in the “CCS Debug Perspective” view. The program ran through the C-environment initialization routine in the rts2800_fpu32.lib and stopped at `main()` in `Lab1.c`.

**Debug Environment Windows**

It is standard debug practice to watch local and global variables while debugging code. There are various methods for doing this in Code Composer Studio. We will examine two of them here: memory browser, and expressions.
23. Open a “Memory Browser” to view the global variable “z”.

   Click: View → Memory Browser on the menu bar.

   Type &z into the address field, select “Data” memory page, and then select Go. Note that you must use the ampersand (meaning “address of”) when using a symbol in a memory browser address box. Also note that CCS is case sensitive.

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

24. Notice the “Variables” window automatically opened and the local variables x and y are present. The variables 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 browser by setting the address to “SP” after the code function has been entered).

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

   Click the “Expressions” tab at the top of the window. In the empty box in the “Expression” column (Add new expression), type z and then enter. An ampersand is not used here. The expressions window knows you are specifying a symbol. (Note that the expressions window can be manually opened by clicking: View → Expressions on the menu bar).

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

**Single-stepping the Code**

26. Click the “Variables” tab at the top of the window to watch the local variables. Single-step through main() by using the <F5> key (or you can use the Step Into button on the horizontal 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?

**Terminate Debug Session and Close Project**

27. The Terminate button will terminate the active debug session, close the debugger and return CCS to the “CCS Edit Perspective” view.

   Click: Run → Terminate or use the Terminate icon: 

28. Next, close the project by right-clicking on Lab1 in the Project Explorer window and select Close Project.

   **End of Exercise**
Peripheral Register Header Files

Traditional Approach to C Coding

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

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

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

In the traditional approach to C coding, we used a `#define` to assign the address of the register and referenced it with a pointer. The first line of code on this slide we are writing to the entire register with a 16-bit value. The second line, we are ORing a bit field.

Advantages? Simple, fast, and easy to type. The variable names can exactly match the register names, so it's easy to remember. Disadvantages? Requires individual masks to be generated to manipulate individual bits, it cannot easily display bit fields in the debugger window, and it will generate less efficient code in many cases.
### Structure Approach to C Coding

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

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

**Disadvantages**
- Can be difficult to remember the structure names (Editor Auto Complete feature to the rescue!)
- More to type (again, Editor Auto Complete feature to the rescue)

The structure approach to C coding uses the peripheral register header files. First, a peripheral is specified, followed by a control register. Then you can modify the complete register or selected bits. This is almost self-commented code.

The first line of code on this slide we are writing to the entire register. The second line of code we are modifying a bit field. Advantages? Easy to manipulate individual bits, it works great with our tools, and will generate the most efficient code. Disadvantages? Can be difficult to remember the structure names and more to type; however, the edit auto complete feature of Code Composer Studio will eliminate these disadvantages.
With the traditional approach to coding using #define, we can only view the complete register values. As an example, notice the control register ADCCTL1 has a value of 0x40E4. We would need to refer to the reference guide to know the settings of the individual bit fields.

With the structure approach, we can add the peripheral to an expressions window, allowing us to
view, as well as modify individual bit fields in a register. No need for a reference guide to identify the bit fields.

**Structure Naming Conventions**

- The F2806x 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>PeripheralName.RegisterName.all</th>
<th>// Access full 16 or 32-bit register</th>
</tr>
</thead>
<tbody>
<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 F2806x 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,...).

The header files define all of the peripheral structures, all of the register names, all of the bit field names, and all of the register addresses. The most common naming conventions used are PeripheralName.RegisterName.all, which will access the full 16 or 32-bit register; and PeripheralName.RegisterName.bit.FieldName, which will access the specified bit fields of a register.
This demonstrates how the editor auto complete feature works. First, you type AdcRegs. Then, when you type a “.” a window opens up, allowing you to select a control register. In this example ADCCTL1 is selected. Then, when you type the “.” a window opens up, allowing you to select “all” or “bit”. In this example “bit” is selected. Then, when you type the “.” a window opens up, allowing you to select a bit field. In this example RESET is selected. And now, the structure is completed.
The F2806x header file package contains everything needed to use the structure approach. It defines all the peripheral register bits and register addresses. The header file package includes the header files, linker command files, code examples, and documentation. The header file package is available from controlSUITE.
Next, we will discuss the steps needed to use the header files with your project. The .h files contain the bit field structure definitions for each peripheral register.
three steps needed to use the header files. The first step is to include this file directly or indirectly in each source files.

Global Variable Definitions File

* Declares a global instantiation of the structure for each peripheral
* Each structure is placed in its own section using a DATA_SECTION pragma to allow linking to the correct memory (see next slide)

```c
#include "F2806x_Device.h"
...
#pragma DATA_SECTION(AdcRegs,"AdcRegsFile");
volatile struct ADC_REGS AdcRegs;
...
```

* Add this file to your CCS project:

* F2806x_GlobalVariableDefs.c

The global variable definition file declares a global instantiation of the structure for each peripheral. Each structure is placed in its own section using a DATA_SECTION pragma to allow linking to the correct memory. The second step for using the header files is to add F2806x_GlobalVariableDefs.c file to your project.
The header file package has two linker command file versions; one for non-BIOS projects and one for BIOS projects. This linker command file is used to link each structure to the address of the peripheral using the structures named section. The third and final step for using the header files is to add the appropriate linker command file to your project.

Peripheral Specific Examples
◆ Example projects for each peripheral
◆ Helpful to get you started
The peripheral register header file package includes example projects for each peripheral. This can be very helpful to getting you started.

### Peripheral Register Header Files

**Summary**

- Easier code development
- Easy to use
- Generates most efficient code
- Increases effectiveness of CCS watch window
- TI has already done all the work!

- Use the correct header file package for your device:
  - F2806x
  - F2803x
  - F2802x
  - F2833x and F2823x

Go to [http://www.ti.com](http://www.ti.com) and enter “controlSUITE” in the keyword search box

In summary, the peripheral register header files allow for easier code development, they are easy to use, generates the most efficient code, works great with Code Composer Studio, and TI has already done the work for you. Just make sure to use the correct header file package for your device.
Reset, Interrupts and System Initialization

Reset

There are various reset sources available for this device: an external reset pin, watchdog timer reset, power-on reset which generates a device reset during power-up conditions, brownout reset which generates a device reset if the power supply drops below specifications for the device, as well as a missing clock detect reset. Additionally, the device incorporates an on-chip voltage regulator to generate the core voltage.
After reset, the PIE block is disabled and the global interrupt line is disabled. The reset vector is fetched from the boot ROM and the bootloader process begins.

Then the bootloader determines if the emulator is connected by checking the JTAG test reset line. If the emulator is connected, we are in emulation boot mode. The boot is then determined by two RAM locations named EMU_KEY and EMU_BMODE, which are located in the PIE block. If the emulator is not connected, we are in stand-alone boot mode. The boot is then determined by two GPIO pins and two OTP locations named OTP_KEY and OTP_BMODE, which are located in the OTP.
In emulation boot mode, first the EMU_KEY register is checked to see if it has a value of 0x55AA. If either EMU_KEY or EMU_BMODE are invalid, the wait boot mode is used. These values can then be modified using the debugger and a reset issued to restart the boot process. This can be considered the default on power-up. At this point, you would like the device to wait until given a boot mode.

If EMU_KEY register has a value of 0x55AA, then the hex value in the EMU_BMODE register determines the boot mode. The boot modes are parallel I/O, SCI, SPI, I2C, OTP, CAN, M0 SARAM, FLASH, and Wait. In addition, there is a GetMode, which emulates the stand-alone boot mode.
In stand-alone boot mode, GPIO pins 37 and 34 determine if the boot mode is parallel I/O, SCI, or wait. The default unconnected pins would set the boot mode to GetMode. In GetMode, first the OTP_KEY register is checked to see if it has a value of 0x005A. An unprogrammed OTP is set to the FLASH boot mode, as expected.

If the OTP_KEY register has a value of 0x005A, then the hex value in the OTP_BMODE register determines the boot mode. The boot modes are SCI, SPI, I2C, OTP, CAN, and FLASH.
In summary, the reset code flow is as follows: The reset vector is fetched from the boot ROM. Then, the execution entry is determined by emulation boot mode or stand-alone boot mode. The boot mode options are M0SARAM, OTP, FLASH, and boot loading routines.

After reset how do we get to main()?

- **At the code entry point, branch to _c_int00()**
  - Part of compiler runtime support library
  - Sets up compiler environment
  - Calls main()

```asm
.sect "codestart"
LB _c_int00
```

**Note:** the above example is for boot mode set to M0 SARAM; to run out of Flash, the "codestart" section would be linked to the entry point of the Flash memory block.
compiler runtime support library is located at the code entry point. This branch to _c_int00 is executed, then the compiler environment is set up, and finally main is called.

Interrupts

The internal interrupt sources include the general purpose timers 0, 1, and 2, and all of the peripherals on the device. External interrupt sources include the three external interrupt lines, the trip zones, and the external reset pin. The core has 14 interrupt lines. As you can see, the number of interrupt sources exceeds the number of interrupt lines on the core. The PIE, or Peripheral Interrupt Expansion block, is connected to the core interrupt lines 1 through 12. This block manages and expands the 12 core interrupt lines, allowing up to 96 possible interrupt sources.
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.

It is easier to explain the interrupt processing flow from the core back out to the interrupt sources. The INTM is the master interrupt switch. This switch must be closed for any interrupts to propagate into the core. The next layer out is the interrupt enable register. The appropriate interrupt line switch must be closed to allow an interrupt through. The interrupt flag register gets set when an interrupt occurs. Once the core starts processing an interrupt, the INTM switch opens to avoid nested interrupts and the flag is cleared.
The core interrupt registers consist of the interrupt flag register, interrupt enable register, and interrupt global mask bit. Notice that the interrupt global mask bit is zero when enabled and one when disabled. The interrupt enable register is managed by ORing and ANDing mask values. The interrupt global mask bit is managed using inline assembly.
Peripheral Interrupt Expansion (PIE)

We have already discussed the interrupt process in the core. Now we need to look at the peripheral interrupt expansion block. This block is connected to the core interrupt lines 1 through 12. The PIE block consists of 12 groups. Within each group, there are eight interrupt sources. Each group has a PIE interrupt enable register and a PIE interrupt flag register.

As you can see, the interrupts are numbered from 1.1 through 12.8, giving us a maximum of 96 interrupt sources. Interrupt lines 13, 14, and NMI bypass the PIE block.
### F2806x 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>ADCINT9</td>
<td>XINT2</td>
<td>XINT1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT2</td>
<td>EPWM8</td>
<td>EPWM7</td>
<td>EPWM6</td>
<td>EPWM5</td>
<td>EPWM4</td>
<td>EPWM3</td>
<td>EPWM2</td>
</tr>
<tr>
<td></td>
<td>_TZINT</td>
<td>_TZINT</td>
<td>_TZINT</td>
<td>_TZINT</td>
<td>_TZINT</td>
<td>_TZINT</td>
<td>_TZINT</td>
</tr>
<tr>
<td>INT3</td>
<td>EPWM8</td>
<td>EPWM7</td>
<td>EPWM6</td>
<td>EPWM5</td>
<td>EPWM4</td>
<td>EPWM3</td>
<td>EPWM2</td>
</tr>
<tr>
<td></td>
<td>_INT</td>
<td>_INT</td>
<td>_INT</td>
<td>_INT</td>
<td>_INT</td>
<td>_INT</td>
<td>_INT</td>
</tr>
<tr>
<td>INT4</td>
<td>HRCAP2</td>
<td>HRCAP1</td>
<td></td>
<td></td>
<td>ECAP3</td>
<td>ECAP2</td>
<td>ECAP1</td>
</tr>
<tr>
<td></td>
<td>_INT</td>
<td>_INT</td>
<td></td>
<td></td>
<td>_INT</td>
<td>_INT</td>
<td>_INT</td>
</tr>
<tr>
<td>INT5</td>
<td></td>
<td></td>
<td>HRCAP4</td>
<td></td>
<td>HRCAP3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>_INT</td>
<td></td>
<td>_INT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT6</td>
<td>MXINTA</td>
<td>MRINTA</td>
<td>SPITX</td>
<td>INTB</td>
<td>SPIRX</td>
<td>INTB</td>
<td>SPIRX</td>
</tr>
<tr>
<td>INT7</td>
<td>DINTCH6</td>
<td>DINTCH5</td>
<td>DINTCH4</td>
<td>DINTCH3</td>
<td>DINTCH2</td>
<td>DINTCH1</td>
<td></td>
</tr>
<tr>
<td>INT8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>I2CINTA</td>
<td>I2CINT1</td>
</tr>
<tr>
<td>INT9</td>
<td></td>
<td></td>
<td>ECAN1</td>
<td>_INTA</td>
<td>ECAN0</td>
<td>_INTA</td>
<td>SCITX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>_INTA</td>
<td></td>
<td>_INTA</td>
<td></td>
<td>_INTA</td>
</tr>
<tr>
<td>INT10</td>
<td>ADCINT8</td>
<td>ADCINT7</td>
<td>ADCINT6</td>
<td>ADCINT5</td>
<td>ADCINT4</td>
<td>ADCINT3</td>
<td>ADCINT2</td>
</tr>
<tr>
<td>INT11</td>
<td>CLA1</td>
<td>CLA1</td>
<td>CLA1</td>
<td>CLA1</td>
<td>CLA1</td>
<td>CLA1</td>
<td>CLA1</td>
</tr>
<tr>
<td></td>
<td>_INT8</td>
<td>_INT7</td>
<td>_INT6</td>
<td>_INT5</td>
<td>_INT4</td>
<td>_INT3</td>
<td>_INT2</td>
</tr>
<tr>
<td>INT12</td>
<td>LUF</td>
<td>LVF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The interrupt assignment table tells us the location for each interrupt source within the PIE block. Notice the table is numbered from 1.1 through 12.8, perfectly matching the PIE block.

### PIE Registers

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

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

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

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

#### PIE Interrupt Acknowledge Register (PIEACK)

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

#### PIECTRL register (x = 1 to 12)

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

#include "F2806x_Device.h"

PieCtrlRegs.PIEFR1.bit.INTx4 = 1;  //manually set IFR for XINT1 in PIE group 1
PieCtrlRegs.PIEIER3.bit.INTx2 = 1;  //enable EPWM2_INT in PIE group 3
PieCtrlRegs.PIEACK.all = 0x0004;  //acknowledge the PIE group 3
PieCtrlRegs.PIECTRL.bit.ENPIE = 1;  //enable the PIE

The PIE registers consist of 12 PIE interrupt flag registers, 12 PIE interrupt enable registers, a PIE interrupt acknowledge register, and a PIE control register. The enable PIE bit in the PIE
control register must be set during initialization for the PIE block to be enabled.

The interrupt vector table, as mapped in the PIE interrupt assignment table, is located in the PieVect.c file. During initialization in main, we have a function call to PieCtrl.c. In this file, a memory copy function copies the interrupt vector table to the PIE RAM and then sets ENPIE to 1, enabling the PIE block. This process is done to set up the vectors for interrupts.
In summary, the PIE initialization code flow is as follows. After the device is reset and executes the boot code, the selected boot option determines the code entry point. This figure shows two different entry points. The one on the left is for memory block M0, and the one on the right is for flash.

In either case, CodeStartBranch.asm has a “Long Branch” to the entry point of the runtime support library. After the runtime support library completes execution, it calls main. In main, we have a function call to initialize the interrupt process and enable the PIE block. When an interrupt occurs, the PIE block contains a vector to the interrupt service routine located in DefaultIsr.c.
In summary, the following steps occur during an interrupt process. First, a peripheral interrupt is generated and the PIE interrupt flag register is set. If the PIE interrupt enable register is enabled, then the core interrupt flag register will be set. Next, if the core interrupt enable register and global interrupt mask is enabled, the PIE vector table will redirect the code to the interrupt service routine.
Oscillator / PLL Clock Module

The oscillator/PLL clock module has two internal, 10 MHz oscillators, and the availability of an external oscillator or crystal. This provides redundancy in case an oscillator fails, as well as the ability to use multiple oscillators. The asterisks in the multiplexers show the default settings. This module has the capability to clock the watchdog, core, and CPU timer 2 from independent clock sources, if needed. Next, we will look at the details for clocking the core.
F2806x PLL and LOSPCP

(lab file: SysCtrl.c)

<table>
<thead>
<tr>
<th>DIV</th>
<th>CLKin</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>OSCCLK / n * (PLL bypass)</td>
</tr>
<tr>
<td>0001</td>
<td>OSCCLK x 1 / n</td>
</tr>
<tr>
<td>0010</td>
<td>OSCCLK x 2 / n</td>
</tr>
<tr>
<td>0011</td>
<td>OSCCLK x 3 / n</td>
</tr>
<tr>
<td>0100</td>
<td>OSCCLK x 4 / n</td>
</tr>
<tr>
<td>0101</td>
<td>OSCCLK x 5 / n</td>
</tr>
<tr>
<td>0110</td>
<td>OSCCLK x 6 / n</td>
</tr>
<tr>
<td>0111</td>
<td>OSCCLK x 7 / n</td>
</tr>
<tr>
<td>1000</td>
<td>OSCCLK x 8 / n</td>
</tr>
<tr>
<td>1001</td>
<td>OSCCLK x 9 / n</td>
</tr>
<tr>
<td>1010</td>
<td>OSCCLK x 10 / n</td>
</tr>
<tr>
<td>1011</td>
<td>OSCCLK x 11 / n</td>
</tr>
<tr>
<td>1100</td>
<td>OSCCLK x 12 / n</td>
</tr>
<tr>
<td>1101</td>
<td>OSCCLK x 13 / n</td>
</tr>
<tr>
<td>1110</td>
<td>OSCCLK x 14 / n</td>
</tr>
<tr>
<td>1111</td>
<td>OSCCLK x 15 / n</td>
</tr>
<tr>
<td>1000</td>
<td>OSCCLK x 16 / n</td>
</tr>
<tr>
<td>1001</td>
<td>OSCCLK x 17 / n</td>
</tr>
<tr>
<td>1010</td>
<td>OSCCLK x 18 / n</td>
</tr>
<tr>
<td>1111</td>
<td>OSCCLK x 19 / n</td>
</tr>
</tbody>
</table>

A clock source can be fed directly into the core or multiplied using the PLL. The PLL gives us the capability to use the internal 10 MHz oscillator multiplied by 18/2, and run the device at the full 90 MHz clock frequency. If the input clock is removed after the PLL is locked, the input clock failed detect circuitry will issue a limp mode clock of 1 to 4 MHz. Additionally, an internal device reset will be issued. The low-speed peripheral clock prescaler is used to clock some of the communication peripherals.
Watchdog Timer Module

Watchdog Timer

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

The watchdog timer is a safety feature, which resets the device if the program runs away or gets trapped in an unintended infinite loop.

The watchdog counter runs independent of the CPU. If the counter overflows, a reset or interrupt is triggered. The CPU must write the correct data key sequence to reset the counter before it overflows. The watchdog must be serviced or disabled within 131,072 watchdog clock cycles after reset. This translates to 13.11 milliseconds with a 10 MHz watchdog clock.
The watchdog clock is divided by 512 and prescaled, if desired. The watchdog disable switch allows the watchdog to be enabled and disabled. The watchdog override switch is a safety mechanism, and once closed, it can only be open by resetting the device.

During initialization, “101” is written into the watchdog check bit fields. Any other values will cause a reset or interrupt. During run time, the correct keys must be written into the watchdog key register before the watchdog counter overflows and issues a reset or interrupt. Issuing a reset or interrupt is user-selectable.
### GPIO

#### F2806x GPIO Grouping Overview

(lab file: Gpio.c)

| GPIO Port A Mux1 Register (GPAMUX1) [GPIO 0 to 15] | GPIO Port A Direction Register (GPADIR) [GPIO 0 to 31] | Input Qual |
| GPIO Port A Mux2 Register (GPAMUX2) [GPIO 16 to 31] | | |
| GPIO Port B Mux1 Register (GPBMUX1) [GPIO 32 to 47] | GPIO Port B Direction Register (GPBDIR) [GPIO 32 to 63] | Input Qual |
| GPIO Port B Mux2 Register (GPBMUX2) [GPIO 48 to 63] | | |
| ANALOG I/O Mux1 Register (AIOMUX1) [AIO 0 to 15] | ANALOG Port Direction Register (AIODIR) [AIO 0 to 15] | |

Each general-purpose I/O pin has a maximum of four options, either general-purpose I/O or up to three possible peripheral pin assignments. This is selected using the GPIO port multiplexer. If the pin is set to GPIO, the direction register sets it as an input or an output. The input qualification will be explained shortly.
This figure shows a single GPIO pin. If the pin is set as a GPIO by the GPIO multiplexer, the direction will be set by the GPIO direction register. The GPIO data register will have the value of the pin if set as an input or write the value of the data register to the pin if set as an output.

The data register can be quickly and easily modified using set, clear, or toggle registers. As you can see, the GPIO multiplexer can be set to select up to three other possible peripheral pin assignments. Also, the pin has an option for an internal pull-up.
The GPIO input qualification feature allows filtering out noise on a pin. The user would select the number of samples and qualification period. Qualification is available on ports A and B only and is individually selectable per pin.
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

This lab exercise consists of two parts. In the first part we will test the behavior of the watchdog when disabled and enabled. In the second part we will initialize the peripheral interrupt expansion vectors and use the watchdog to generate an interrupt.
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:

- Setup the clock module – PLL, LOSPCP = /4, low-power modes to default values, enable all module clocks
- Disable the watchdog – clear WD flag, disable watchdog, WD prescale = 1
- Setup the watchdog and system control registers – DO NOT clear WD OVERRIDE bit, configure WD to generate a CPU reset
- Setup the 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 be tested by using the watchdog to generate an interrupt. This lab will make use of the F2806x 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

Open the Project

1. A project named Lab2 has been created for this lab. Open the project by clicking on Project \ Import Existing CCS Eclipse Project. The “Import” window will open then click Browse... next to the “Select search-directory” box. Navigate to: C:\C28x\Labs\Lab2\Project and click OK. Then click Finish to import the project.

2. In the Project Explorer window, click the plus sign (+) to the left of Lab2 to view the project files. All Build Options have been configured for this lab. The files used in this lab are:

   CodeStartBranch.asm       Lab.h
   DefaultIsr_2.c            Lab_2_3.cmd
   DelayUs.asm               Main_2.c
   F2806x_DefaultIsr.h       PieCtrl.c
   F2806x_GlobalVariableDefs.c PieVect.c
   F2806x_Headers_nonBIOS.cmd SysCtrl.c
   Gpio.c                    Watchdog.c
Modified Memory Configuration

3. Open and inspect the linker command file `Lab_2_3.cmd`. Notice that the user defined section “codestart” is being linked to a memory block named `BEGIN_M0`. The codestart section contains code that branches to the code entry point of the project. The bootloader must branch to the codestart section at the end of the boot process. Recall that the emulation boot mode "M0 SARAM" branches to address 0x000000 upon bootloader completion.

The linker command file (`Lab_2_3.cmd`) has a new memory block named `BEGIN_M0`: origin = 0x000000, length = 0x0002, in program memory. Additionally, the existing memory block `M0SARAM` in data memory has been modified to avoid overlaps with this new memory block.

4. In the linker command file, notice that RESET in the MEMORY section has been defined using the “(R)” qualifier. This qualifier indicates read-only memory, and is optional. It will cause the linker to flag a warning if any uninitialized sections are linked to this memory. The (R) qualifier can be used with all non-volatile memories (e.g., flash, ROM, OTP), as you will see in a later lab exercise.

System Initialization

5. Open and inspect `SysCtrl.c`. Notice that the PLL and module clocks have been enabled.

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

7. 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. Close the inspected files.

Build and Load

8. Click the “Build” button and watch the tools run in the Console window. Check for errors in the Problems window.

9. Click the “Debug” button (green bug). The “CCS Debug Perspective” view should open, the program will load automatically, and you should now be at the start of `main()`.

10. After CCS loaded the program in the previous step, it set the program counter (PC) to point to `_c_int00`. It then ran through the C-environment initialization routine in the `rts2800_fpu32.lib` and stopped at the start of `main()`. CCS did not do a device reset, and as a result the bootloader was bypassed.

In the remaining parts of this lab exercise, the device will be undergoing a reset due to the watchdog timer. Therefore, we must configure the device by loading values into `EMU_KEY` and `EMU_BMODE` so the bootloader will jump to “M0 SARAM” at address 0x000000. Set the bootloader mode using the menu bar by clicking:

`Scripts → EMU Boot Mode Select → EMU_BOOT_SARAM`
If the device is power cycled between lab exercises, or within a lab exercise, be sure to re-configure the boot mode to EMU_BOOT_SARAM.

**Run the Code – Watchdog Reset**

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

12. Place the cursor on the first line of code in `main()` and set a breakpoint by double clicking in the line number field to the left of the code line. Notice that line is highlighted with a blue dot indicating that the breakpoint has been set. (Alternately, you can set a breakpoint on the line by right-clicking the mouse and selecting Breakpoint (Code Composer Studio) → Breakpoint). The breakpoint is set to prove that the watchdog is disabled. If the watchdog causes a reset, code execution will stop at this breakpoint.

13. Run your code for a few seconds by using the “Resume” button on the toolbar, or by using Run → Resume on the menu bar (or F8 key). After a few seconds halt your code by using the “Suspend” button on the toolbar, or by using Run → Suspend on the menu bar (or Alt-F8 key). Where did your code stop? Are the results as expected? If things went as expected, your code should be in the “main loop”.

14. Switch to the “CCS Edit Perspective” view by clicking the CCS Edit icon in the upper right-hand corner. Modify the InitWatchdog() function to enable the watchdog (WDCR). In Watchdog.c change the WDCR register value to 0x00A8. This will enable the watchdog to function and cause a reset. Save the file.

15. Click the “Build” button. Select Yes to “Reload the program automatically”. Switch back to the “CCS Debug Perspective” view by clicking the CCS Debug icon in the upper right-hand corner.

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

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

**Setup PIE Vector for Watchdog Interrupt**

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

18. Switch to the “CCS Edit Perspective” view by clicking the CCS Edit icon in the upper right-hand corner. Notice that the following files are included in the project:
   DefaultIsr_2.c
   PieCtrl.c
   PieVect.c

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

   InitPieCtrl();

20. Using the “PIE Interrupt Assignment Table” shown in the slides find the location for the watchdog interrupt, “WAKEINT”. This is used in the next step.

   PIE group #: __________  # within group: _________

21. In main() notice the code used to enable global interrupts (INTM bit), and in InitWatchdog() the code used to enable the “WAKEINT” interrupt in the PIE (using the PieCtrlRegs structure) and to enable core INT1 (IER register).

22. Modify the system control and status register (SCSR) to cause the watchdog to generate a WAKEINT rather than a reset. In Watchdog.c change the SCSR register value to 0x0002. Save the modified files.

23. Open and inspect DefaultIsr_2.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.

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

**Build and Load**

25. Click the “Build” button and select Yes to “Reload the program automatically”. Switch to the “CCS Debug Perspective” view by clicking the CCS Debug icon in the upper right-hand corner.

**Run the Code – Watchdog Interrupt**

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

27. 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.
Terminate Debug Session and Close Project

28. Terminate the active debug session using the Terminate button. This will close the debugger and return CCS to the “CCS Edit Perspective” view.

29. Next, close the project by right-clicking on Lab2 in the Project Explorer window and select Close Project.

End of Exercise

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

ADC Module

The ADC module is based around a 12-bit converter. There are 16 input channels and 16 result registers. The SOC configuration registers select the trigger source, channel to convert, and the acquisition prescale window size. The triggers include software by selecting a bit, CPU timers 0, 1 and 2, EPWMA and EPWMB 1 through 8, and an external pin. Additionally, ADCINT 1 and 2 can be fed back for continuous conversions.

The ADC module can operate in sequential sampling mode or simultaneous sampling mode. In simultaneous sampling mode, the channel selected on the A multiplexer will be the same channel on the B multiplexer. The ADC interrupt logic can generate up to nine interrupts. The results for SOC 0 through 15 will appear in result registers 0 through 15.
The top example on this slide shows channels A2, B3, and A7 being converted with a trigger from EPWM1SOCB. After A7 is converted, ADCINT1 is generated.

The bottom examples extends this with channels A0, B0, and A5 being converted initially with a software trigger. After A5 is converted, ADCINT2 is generated, which is fed back as a trigger to start the process again.
Sample all channels continuously and provide Ping-Pong interrupts to CPU/system:

Example – ADC Triggering (simultaneous sampling)

The example on this slide shows channels A/B 0 through 7 being converted in simultaneous sampling mode, triggered initially by software. After channel A/B three is converted, ADCINT1 is generated. After channel A/B seven is converted, ADCINT2 is generated and fed back to start the process again. ADCINT1 and ADCINT2 are being used as ping-pong interrupts.
This device has three analog comparators that share the input pins with the analog-to-digital converter module. If neither the ADC or comparator input pins are needed, the input pins can be used as analog I/O pins. As you can see, one of the inputs to the comparator comes directly from the input pin, and the other input can be taken from the input pin or the 10-bit digital-to-analog converter. The output of the comparator is fed into the ePWM digital compare sub-module.

**ADC Control Registers** (file: `Adc.c`)

- **ADCTRL1** (ADC Control Register 1)
  - module reset, ADC enable
  - busy/busy channel
  - reference select
  - Interrupt generation control
- **ADCSC0SOCxCTL** (SOC0 to SOC15 Control Registers)
  - trigger source
  - channel
  - acquisition sampling window
- **ADCINTSOCxSEL0, 1** (Interrupt SOC Selection 1 and 2 Registers)
  - selects ADCINT1 / ADCINT2 trigger for SOCx
- **ADCSAMPLEMODE** (Sampling Mode Register)
  - sequential sampling / simultaneous sampling
- **INTSELxNy** (Interrupt x and y Selection Registers)
  - EOC0 – EOC15 source select for ADCINT1-9
- **ADCRESULTx** (ADC Result 0 to 15 Registers)

*Note: refer to the reference guide for a complete listing of registers*
The previous slide is a brief overview covering a few of the ADC control registers. It can be used as a reference during the lab exercise.

**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

Pulse width modulation, or PWM, is a technique used to represent a signal as a sequence of pulses. A PWM waveform has a fixed carrier frequency, fixed pulse amplitude, and the pulse width is proportional to the instantaneous signal amplitude.
Power-switching devices are difficult to control in the proportional region but are easy to control in the saturation and cutoff region. Since PWM is a digital signal and easy for microcontrollers to generate, it is ideal for use with power-switching devices.
An ePWM module can be synchronized with adjacent ePWM modules. The generated PWM waveforms are available as outputs on the GPIO pins. Additionally, the EPWM module can generate ADC starter conversion signals and generate interrupts to the PIE block. External trip zone signals can trip the output and generate interrupts, too. The outputs of the comparators are used as inputs to the digital compare sub-module. Next, we will look at the internal details of the ePWM module.
The ePWM, or enhanced PWM block diagram, consists of a series of sub-modules. In this section, we will learn about the operation and details of each sub-module.

In the time-base sub-module, the clock prescaler divides down the device core system clock and clocks the 16-bit time-base counter. The time-base counter is used to generate asymmetrical and
symmetrical waveforms using three different count modes: count-up mode, countdown mode, and count up and down mode. A period register is used to control the maximum count value. Additionally, the time-base counter has the capability to be synchronized and phase-shifted with other ePWM units.

The upper two figures show the time-base counter in the count-up mode and countdown mode. These modes are used to generate asymmetrical waveforms. The lower figure shows the time-base counter in the count up and down mode. This mode is used to generate symmetrical waveforms.
If needed, an ePWM module can be synchronized with adjacent ePWM modules. Synchronization is based on a synch-in signal, time-base counter equals zero, or time-base counter equals compare B register. Additionally, the waveform can be phase-shifted.

The compare sub-module uses two compare registers to detect time-base count matches. These
compare match events are fed into the action qualifier sub-module. Notice that the output of this block feeds two signals into the action qualifier.

The figures above show the compare matches that are fed into the action qualifier. Notice that with the count up and countdown mode, there are matches on the up-count and down-count.

The action qualifier sub-module uses the inputs from the compare logic and time-base counter to
generate various actions on the output pins. These first few modules are the main components used to generate a basic PWM waveform.

### ePWM Action Qualifier Actions
for EPWMA and EPWMB

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

This table shows the various action qualifier compare-match options for when the time-base counter equals zero, compare A match, compare B match, and period match. Based on the selected match option, the output pins can be configured to do nothing, clear low, set high, or toggle. Also, the output pins can be forced to any action using software.
The next few figures show how the action qualifier uses the compare matches to modulate the output pins. Notice that the output pins for EPWMA and EPWMB are completely independent. Here, on the EPWMA output, the waveform will be set high on zero match and clear low on compare A match. On the EPWMB output, the waveform will be set high on zero match and clear low on compare B match.
This figure has the EPWMA output set high on compare A match and clear low on compare B match, while the EPWMB output is configured to toggle on zero match.

Here you can see that we can have different output actions on the up-count and down-count using a single compare register. So, for the EPWMA and EPWMB outputs, we are setting high on the
compare A and B up-count matches and clearing low on the compare A and B down-down matches.

And finally, again using different output actions on the up-count and down-count, we have the EPWMA output set high on the compare A up-count match and clear low on the compare B down-count match. The EPWMB output will clear low on zero match and set high on period match.
The dead-band sub-module provides a means to delay the switching of a gate signal, thereby allowing time for gates to turn off and preventing a short circuit.

To explain further, power-switching devices turn on faster than they shut off. This issue would momentarily provide a path from supply rail to ground, giving us a short circuit. The dead-band...
sub-module alleviates this issue.

The PWM chopper sub-module uses a high-frequency carrier signal to modulate the PWM waveform. This is used with pulsed transformer-based gate drives to control power-switching elements.

As you can see in this figure, a high-frequency carrier signal is ANDed with the ePWM outputs.
Also, this circuit provides an option to include a larger, one-shot pulse width before the sustaining pulses.

The trip zone and digital compare sub-modules provide a protection mechanism to protect the output pins from abnormalities, such as over-voltage, over-current, and excessive temperature rise.
The inputs to the digital compare sub-module are the trip zone pins and the analog comparator outputs. This module generates compare events that can generate a PWM sync, generate an ADC start of conversion, trip a PWM output, and generate a trip interrupt. Optional blinking can be used to temporarily disable the compare action in alignment with PWM switching to eliminate noise effects.
Trip-Zone Features

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

The PWM trip zone has a fast, clock-independent logic path to the PWM output pins where the outputs can be forced to high impedance. Two actions are supported: One-shot trip for major short circuits or over-current conditions, and cycle-by-cycle trip for current limiting operation.

ePWM Event-Trigger Sub-Module

The event-trigger sub-module is used to provide a triggering signal for interrupts and the start of
conversion for the ADC.

Event-trigger interrupts and start of conversions can be generated on counter equals zero, counter equal period, counter equal zero or period, counter up equal compare A, counter down equal compare A, counter up equal compare B, counter down equal compare B. Notice counter up and down are independent and separate.
The high-resolution PWM feature significantly increases the resolution of conventionally-derived PWM. High-resolution PWM divides a clock cycle into smaller steps called micro steps. The step size is approximately 150 picoseconds. This is typically used when PWM resolution falls below approximately 9 or 10 bits, which occurs at frequencies greater than approximately 180 kHz with a system clock of 90 MHz.
The previous slide is a brief overview covering a few of the ePWM control registers. It can be used as a reference during the lab exercise.

**eCAP**

The eCAP module timestamps transitions on a capture input pin.

- Can be used to measure the time width of a pulse.

- Auxiliary PWM generation

The capture module allows time-based logging of external signal transitions on the capture input pins.
The capture module features a 32-bit time-stamp counter to minimize rollover. Each module has four capture registers. Polarity can be set to trigger on rising or falling edge, and trigger events can be pre-scaled. The capture module can operate in absolute time-stamp mode or difference mode where the counter resets on each capture.
If the capture module is not used, it can be configured as an asynchronous PWM module.

**eQEP**

**What is an Incremental Quadrature Encoder?**

*A digital (angular) position sensor*

![Incremental Optical Encoder](image1)

The QEP circuit decodes and counts the quadrature encoded input pulses. This circuit can be used to interface with an optical encoder to get position and speed information from a rotating machine.
How is Position Determined from Quadrature Signals?

Position resolution is $\theta/4$ degrees

Using a quadrature decoder state machine, we can determine if the counter is incrementing or decrementing, and therefore know if the disc is moving clockwise or counterclockwise.

The QEP module features a direct interface to encoders. In addition to channels A and B being used for rotational directional information, the index can be used to determine rotational speed,
and the strobe can be used for position from a homing sensor.

Next, in this lab exercise, a 2 kHz, 25 percent duty cycle PWM waveform will be generated by ePWM1. The waveform will be fed into ADC channel A0 using the jumper wire. The ADC is being triggered by ePWM2 at a rate of 50 kHz. After each conversion, the CPU will copy the results into a circular buffer. Using Code Composer Studio, we will view the waveform in the time domain and frequency domain. Additionally, we will experiment with the real-time emulation features.
Lab 3: Control Peripherals

Objective

The objective of this lab is to demonstrate 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 MCU 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 SOCA)

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 LD2 on the TMS320F28069 controlSTICK 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 SOCA 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 LED LD2 as a visual indication that it is running

Procedure

Open the Project

1. A project named Lab3 has been created for this lab. Open the project by clicking on Project → Import Existing CCS Eclipse Project. The “Import” window will open then click Browse… next to the “Select search-directory” box. Navigate to: C:\C28x\Labs\Lab3\Project and click OK. Then click Finish to import the project.

2. In the Project Explorer window, click the plus sign (+) to the left of Lab3 to view the project files. All Build Options have been configured for this lab. The files used in this lab are:

   - Adc.c
   - CodeStartBranch.asm
   - DefaultIsr_3_4.c
   - DelayUs.asm
   - EPwm.c
   - F2806x_DefaultIsr.h
   - F2806x_GlobalVariableDefs.c
   - F2806x_Headers_nonBIOS.cmd
   - Gpio.c
   - Lab.h
   - Lab_2_3.cmd
   - Main_3.c
   - PieCtrl.c
   - PieVect.c
   - SysCtrl.c
   - Watchdog.c

Setup GPIO and ePWM1

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

3. 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 the Lab.h file. Notice that ePWM2 has been initialized earlier in the code for the ADC. Close the inspected files.
Build and Load

4. Click the “Build” button and watch the tools run in the Console window. Check for errors in the Problems window.

5. Click the “Debug” button (green bug). The “CCS Debug Perspective” view should open, the program load automatically, and you should now be at the start of Main(). If the device has been power cycled since the last lab exercise, be sure to configure the boot mode to EMU_BOOT_SARAM using the Scripts menu.

Run the Code – PWM Waveform

6. Open a memory browser window to view some of the contents of the ADC results buffer. To open a memory browser window click: View → Memory Browser on the menu bar. The address label for the ADC results buffer is &AdcBuf in the “Data” memory page. Select Go to view the contents of the ADC result buffer.

7. Using a connector wire provided, connect the PWM1A (pin # 17) to ADCINA0 (pin # 3) on the controlSTICK.

8. Run your code for a few seconds by using the Resume button on the toolbar, or using Run → Resume on the menu bar. After a few seconds halt your code by using the Suspend button on the toolbar, or by using Run → Suspend on the menu bar. 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: Tools → Graph → Single Time and set the following values:

<table>
<thead>
<tr>
<th>Acquisition Buffer Size</th>
<th>50</th>
</tr>
</thead>
<tbody>
<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>Start Address</td>
<td>&amp;AdcBuf</td>
</tr>
<tr>
<td>Display Data Size</td>
<td>50</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 “measurement marker mode” graph feature. In the graph window toolbar, left-click on the ruler icon with the red arrow. Note when you hover your mouse over the icon, it will show “Toggle Measurement
Marker Mode”. Move the mouse to the first measurement position and left-click. Again, left-click on the Toggle Measurement Marker Mode icon. Move the mouse to the second measurement position and left-click. The graph will automatically calculate the difference between the two values taken over a complete waveform period. When done, clear the measurement points by right-clicking on the graph and select Remove All Measurement Marks (or Ctrl+Alt+M).

**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 data. Let's make a frequency domain plot of the contents in the ADC results buffer (i.e. the PWM waveform).

Click: Tools ➔ Graph ➔ FFT Magnitude and set the following values:

<table>
<thead>
<tr>
<th>Acquisiton Buffer Size</th>
<th>50</th>
</tr>
</thead>
<tbody>
<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>Start Address</td>
<td>AdcBuf</td>
</tr>
<tr>
<td>Data Plot Style</td>
<td>Bar</td>
</tr>
<tr>
<td>FFT Order</td>
<td>10</td>
</tr>
</tbody>
</table>

Select OK to save the graph options.

12. On the plot window, hold the mouse left-click key and move the marker line to 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 MCU is running. This not only allows graphs and watch windows to update, but also allows the user to change values in watch or memory windows, and have those changes affect the MCU behavior. This is very useful when tuning control law parameters on-the-fly, for example.

13. The memory and single time graph windows displaying AdcBuf should still be open. The connector wire between PWM1A (pin # 17) and ADCINA0 (pin # 3) should still be connected. In real-time mode, we will have our window continuously refresh at the default rate. To view the refresh rate click:

Window ➔ Preferences...

and in the section on the left select the “Code Composer Studio” category. Click the plus sign (+) to the left of “Code Composer Studio” and select “Debug”. In the section on the right notice the default setting:
• “Continuous refresh interval (milliseconds)” = 500

Click OK.

Note: Decreasing the “Continuous refresh interval” causes all enabled continuous refresh windows to refresh at a faster rate. This can be problematic when a large number of windows are enabled, as bandwidth over the emulation link is limited. Updating too many windows can cause the refresh frequency to bog down. In this case you can just selectively enable continuous refresh for the individual windows of interest.

14. Next we need to enable the graph window for continuous refresh. Select the “Single Time” graph. In the graph window toolbar, left-click on the yellow icon with the arrows rotating in a circle over a pause sign. Note when you hover your mouse over the icon, it will show “Enable Continuous Refresh”. This will allow the graph to continuously refresh in real-time while the program is running.

15. Enable the memory window for continuous refresh using the same procedure as the previous step.

16. Run the code and watch the windows update in real-time mode. Click:

   Scripts → Realtime Emulation Control → Run_Realtime_with_Reset

17. Carefully remove and replace the connector wire from ADCINA0 (pin # 3). Are the values updating as expected?

18. Fully halt the CPU in real-time mode. Click:

   Scripts → Realtime Emulation Control → Full_Halt

Terminete Debug Session and Close Project

19. Terminate the active debug session using the Terminate button. This will close the debugger and return CCS to the “CCS Edit Perspective” view.

20. Next, close the project by right-clicking on Lab3 in the Project Explorer window and select Close Project.

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, you may want to try this after the class).

End of Exercise
Lab Reference: F28069 controlSTICK Header Pin Diagram

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ADC-A6 COMP3 (+VE)</td>
<td>2</td>
<td>ADC-A2 COMP1 (+VE)</td>
</tr>
<tr>
<td>3</td>
<td>ADC-A0</td>
<td>4</td>
<td>3V3</td>
</tr>
<tr>
<td>5</td>
<td>ADC-A4 COMP2 (+VE)</td>
<td>6</td>
<td>ADC-B1</td>
</tr>
<tr>
<td>7</td>
<td>EPWM-4B GPIO-07</td>
<td>8</td>
<td>TZ1 GPIO-12</td>
</tr>
<tr>
<td>9</td>
<td>SCL-A GPIO-33</td>
<td>10</td>
<td>ADC-B6 COMP3 (-VE)</td>
</tr>
<tr>
<td>11</td>
<td>EPWM-4A GPIO-06</td>
<td>12</td>
<td>ADC-A1</td>
</tr>
<tr>
<td>13</td>
<td>SDA-A GPIO-32</td>
<td>14</td>
<td>ADC-B0</td>
</tr>
<tr>
<td>15</td>
<td>EPWM-3B GPIO-05</td>
<td>16</td>
<td>5V0 (Disabled by Default)</td>
</tr>
<tr>
<td>17</td>
<td>EPWM-1A GPIO-00</td>
<td>18</td>
<td>ADC-B4 COMP2 (-VE)</td>
</tr>
<tr>
<td>19</td>
<td>EPWM-3A GPIO-04</td>
<td>20</td>
<td>SPISOMI-A GPIO-17</td>
</tr>
<tr>
<td>21</td>
<td>EPWM-1B GPIO-01</td>
<td>22</td>
<td>ADC-A5</td>
</tr>
<tr>
<td>23</td>
<td>EPWM-2B GPIO-03</td>
<td>24</td>
<td>SPISIMO-A GPIO-16</td>
</tr>
<tr>
<td>25</td>
<td>SPISTE-A GPIO-19</td>
<td>26</td>
<td>ADC-B2 COMP1 (-VE)</td>
</tr>
<tr>
<td>27</td>
<td>EPWM-2A GPIO-02</td>
<td>28</td>
<td>GND</td>
</tr>
<tr>
<td>29</td>
<td>SPICLK-A GPIO-18</td>
<td>30</td>
<td>GPIO-34 (LED)</td>
</tr>
<tr>
<td>31</td>
<td>PWM1A-DAC (Filtered)</td>
<td>32</td>
<td>GND</td>
</tr>
</tbody>
</table>
The CPU itself performs the flash programming. The flash utility code is loaded into RAM and executed. It then reads the flash data and writes it into the flash memory. The flash utility code and the flash data can be loaded into the RAM directly using JTAG or through the ROM bootloader using SCI, SPI, I2C, CAN, or GPIO.
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 (8Kw or 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)

There are three steps for flash programming. First, erase, which sets all bits to zero and then one. Second, program, which programs selected bits with zero. And third, verify, which verifies the flash contents.

The minimum erase size is a sector. The minimum program size is a bit. Be sure not to lose power during erase step. Doing so may permanently lock the flash.
Programming Utilities and CCS Flash Programmer

Flash Programming Utilities

- **JTAG Emulator Based**
  - Code Composer Studio on-chip Flash programmer
  - BlackHawk Flash utilities (requires Blackhawk emulator)
  - Elprotronic FlashPro2000
  - Spectrum Digital SDFlash JTAG (requires SD emulator)
  - Signum System Flash utilities (requires Signum emulator)

- **SCI Serial Port Bootloader Based**
  - Code-Skin (http://www.code-skin.com)
  - Elprotronic FlashPro2000

- **Production Test/Programming Equipment Based**
  - BP Micro programmer
  - Data I/O programmer

- **Build your own custom utility**
  - Can use any of the ROM bootloader methods
  - Can embed flash programming into your application
  - Flash API algorithms provided by TI

* TI web has links to all utilities (http://www.ti.com/c2000)

There is a wide variety of flash programming utilities available. They include JTAG emulators, SCI serial port bootloaders, production test/programming equipment, and building your own custom utilities with flash API algorithms provided by TI. The TI website has links to all utilities available.

CCS On-Chip Flash Programmer

- On-Chip Flash programmer is integrated into the CCS debugger
The on-chip flash programmer is integrated into the CCS debugger. The programmer lets you configure the device clock configuration and specify flash program settings, such as erase-program-verify, select sectors to erase, set the code security password, as well as frequency test, depletion recovery, and calculate check sum. These options are available in the Debug perspective by clicking tools ➔ On-Chip Flash.

**Code Security Module and Password**

<table>
<thead>
<tr>
<th>Code Security Module (CSM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access to the following on-chip memory is restricted:</td>
</tr>
<tr>
<td>0x000A80</td>
</tr>
<tr>
<td>0x008000</td>
</tr>
<tr>
<td>0x008800</td>
</tr>
<tr>
<td>0x008C00</td>
</tr>
<tr>
<td>0x009000</td>
</tr>
<tr>
<td>0x00A000</td>
</tr>
<tr>
<td>0x00C000</td>
</tr>
<tr>
<td>0x3D7800</td>
</tr>
<tr>
<td>0x3D7C00</td>
</tr>
<tr>
<td>0x3D7C80</td>
</tr>
<tr>
<td>0x3D7CC0</td>
</tr>
<tr>
<td>0x3F7FF8</td>
</tr>
<tr>
<td>0x3F8000</td>
</tr>
<tr>
<td>0x3F8000</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

The code security module prevents reverse engineering and protects valuable intellectual property. Once a user-define password is stored in flash, data reads and writes from restricted memory are only allowed from code running from restricted memory. All other data read-write accesses are blocked. This includes JTAG emulated debugger, ROM bootloader, code running in external memory, or unrestricted internal memory.
The code security module 128-bit user-defined password is stored in flash starting at address 0x3F7FF8. The 128-bit key registers are used to lock and unlock the device. The key registers are mapped in memory space 0x000AE0 through 0x000AE7, and they are EALLOW protected.

The password match flow is as follows: Flash is secured after reset or run-time. A dummy read...
of the passwords is performed. If the passwords are all 0’s, then the device is permanently locked. If the passwords are all F’s, then the device is unlocked. If neither case, then the passwords are written into the key registers until the correct password is written to unlock the device. All new devices are shipped with passwords set to F’s, and the device is unlocked.

Next, in the lab exercise, we will program and execute code from the on-chip flash memory. This device has been designed for stand-alone operation in an embedded system. Using the on-chip flash eliminates the need for external, nonvolatile memory or a host processor from which to bootload. The step required to properly configure the software for execution from internal flash memory will be covered.
Lab 4: Programming the Flash

Objective

The objective of this lab is to program and execute code from the on-chip flash memory. The TMS320F28069 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.

Objective:
- Program system into Flash Memory
- Learn use of CCS Flash Programmer
- **DO NOT PROGRAM PASSWORDS**

Procedure

Open the Project

1. A project named Lab4 has been created for this lab. Open the project by clicking on Project → Import Existing CCS Eclipse Project. The “Import” window will open then click Browse... next to the “Select search-directory” box. Navigate to C:\C28x\Labs\Lab4\Project and click OK. Then click Finish to import the project.

2. In the Project Explorer window, click the plus sign (+) to the left of Lab4 to view the project files. All Build Options have been configured for this lab. The files used in this lab are:
Link Initialized Sections to Flash

Initialized sections, such as code and constants, must contain valid values at device power-up. Stand-alone operation of an F28069 embedded system means that no emulator is available to initialize the device RAM. Therefore, all initialized sections must be linked to the on-chip flash memory.

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

3. Open and inspect the linker command file Lab_4.cmd. Notice that a memory block named FLASH_ABCDEFGH has been been created at origin = 0x3D8000, length = 0x01FF80 on Page 0. This flash memory block length has been selected to avoid conflicts with other required flash memory spaces. See the reference slide at the end of this lab exercise for further details showing the address origins and lengths of the various memory blocks used.

4. In Lab_4.cmd the following compiler sections have been linked to on-chip flash memory block FLASH_ABCDEFGH:

   Compiler Sections:
   
   | .text | .cinit | .const | .econst | .pinit | .switch |

Copying Interrupt Vectors from Flash to RAM

The interrupt vectors must be located in on-chip flash memory and at power-up needs to be copied to the PIE RAM as part of the device initialization procedure. The code that performs this copy is located in InitPieCtrl(). The C-compiler runtime support library contains a memory copy function called memcpy() which will be used to perform the copy.

5. Open and inspect InitPieCtrl() in PieCtrl.c. Notice the memcpy() function used to initialize (copy) the PIE vectors. At the end of the file a structure is used to enable the PIE.
Initializing the Flash Control Registers

The initialization code for the flash control registers cannot execute from the flash memory (since it is changing the flash configuration!). Therefore, the initialization function for the flash control registers must be copied from flash (load address) to RAM (run address) at runtime. The memory copy function memcpy() will again be used to perform the copy. The initialization code for the flash control registers InitFlash() is located in the Flash.c file.

6. Open and inspect Flash.c. The C compiler CODE_SECTION pragma is used to place the InitFlash() function into a linkable section named “secureRamFuncs”.

7. The “secureRamFuncs” section will be linked using the user linker command file Lab_4.cmd. Open and inspect Lab_4.cmd. The “secureRamFuncs” will load to flash (load address) but will run from L4SARAM (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 MCU hardware or development tools perspective (since the C28x MCU has a unified memory architecture), historical convention is to link code to program memory space and data to data memory space. Therefore, notice that for the L4SARAM memory we are linking “secureRamFuncs” to, we are specifying “PAGE = 0” (which is program memory).

8. Open and inspect Main_4.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.

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

   InitFlash();

   at the desired spot in main().

Code Security Module and Passwords

The CSM module provides protection against unwanted copying (i.e. pirating!) of your code from flash, OTP memory, and the L0, L1, L2, L3 and L4 RAM blocks. The CSM uses a 128-bit password made up of 8 individual 16-bit words. They are located in flash at addresses 0x3F7FF8 to 0x3F7FFF. During this lab, dummy passwords of 0xFFFF will be used – therefore only dummy reads of the password locations are needed to unsecure the CSM. **DO NOT PROGRAM ANY REAL PASSWORDS INTO THE DEVICE.** After development, real passwords are typically placed in the password locations to protect your code. We will not be using real passwords in the workshop.

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

10. 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).

11. Open Lab_4.cmd and notice that the initialized sections for “passwords” and “csm_rsvd” are linked to memories named PASSWORDS and CSM_RSVD, respectively.

**Executing from Flash after Reset**

The F28069 device contains a ROM bootloader that will transfer code execution to the flash after reset. When the boot mode selection is set for “Jump to Flash” mode, the bootloader will branch to the instruction located at address 0x3F7FF6 in the flash. An instruction that branches to the beginning of your program needs to be placed at this address. Note that the CSM passwords begin at address 0x3F7FF8. There are exactly two words available to hold this branch instruction, and not coincidentally, a long branch instruction “LB” in assembly code occupies exactly two words. Generally, the branch instruction will branch to the start of the C-environment initialization routine located in the C-compiler runtime support library. The entry symbol for this routine is _c_int00. Recall that C code cannot be executed until this setup routine is run. Therefore, assembly code must be used for the branch. We are using the assembly code file named CodeStartBranch.asm.

12. 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 linked to a block of memory named BEGIN_FLASH.

13. In the earlier lab exercises, the section “codestart” was directed to the memory named BEGIN_M0. Open and inspect Lab_4.cmd and notice that the section “codestart” will now be directed to BEGIN_FLASH. Close the inspected files.

On power up the reset vector will be fetched and the ROM bootloader will begin execution. If the emulator is connected, the device will be in emulator boot mode and will use the EMU_KEY and EMU_BMODE values in the PIE RAM to determine the bootmode. This mode was utilized in an earlier lab. In this lab, we will be disconnecting the emulator and running in stand-alone boot mode (but do not disconnect the emulator yet!). The bootloader will read the OTP_KEY and OTP_BMODE values from their locations in the OTP. The behavior when these values have not been programmed (i.e., both 0xFFFF) or have been set to invalid values is boot to flash bootmode.

**Build – Lab.out**

14. Click the “Build” button to generate the Lab.out file to be used with the CCS Flash Programmer. Check for errors in the Problems window.

**Programming the On-Chip Flash Memory**

In CCS the on-chip flash programmer is integrated into the debugger. When the program is loaded CCS will automatically determine which sections reside in flash memory based on the linker command file. CCS will then program these sections into the on-chip flash memory. Additionally, in order to effectively debug with CCS, the symbolic debug information (e.g., symbol and label addresses, source file links, etc.) will automatically load so that CCS knows
Lab 4: Programming the Flash

where everything is in your code. Clicking the “Debug” button in the “CCS Edit Perspective” will automatically launch the debugger, connect to the target, and program the flash memory in a single step.

15. Program the flash memory by clicking the “Debug” button (green bug). *(If needed, when the “Progress Information” box opens select “Details...” in order to watch the programming operation and status).* After successfully programming the flash memory the “Progress Information” box will close.

**Running the Code – Using CCS**

16. Reset the CPU using the “Reset CPU” button or click:

   Run ➔ Reset ➔ Reset CPU

The program counter should now be at address 0x3FF75C in the “Disassembly” window, which is the start of the bootloader in the Boot ROM. If needed, click on the “View Disassembly...” button in the window that opens, or click View ➔ Disassembly.

17. Under Scripts on the menu bar click:

   EMU Boot Mode Select ➔ EMU_BOOT_FLASH.

   This has the debugger load values into EMU_KEY and EMU_BMODE so that the bootloader will jump to "FLASH" at address 0x3F7FF6.

18. Single-Step by using the <F5> key (or you can use the Step Into button on the horizontal toolbar) through the bootloader code until you arrive at the beginning of the codestart section in the CodeStartBranch.asm file. *(Be patient, it will take about 125 single-steps).* Notice that we have placed some code in CodeStartBranch.asm to give an option to first disable the watchdog, if selected.

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

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

21. You can now run the CPU, and you should observe the LED on the controlSTICK blinking. Try resetting the CPU, select the EMU_BOOT_FLASH boot mode, and then hitting run (without doing all the stepping and the Go Main procedure). The LED should be blinking again.

22. Halt the CPU.

**Terminate Debug Session and Close Project**

23. Terminate the active debug session using the Terminate button. This will close the debugger and return CCS to the “CCS Edit Perspective” view.

24. Next, close the project by right-clicking on Lab4 in the Project Explorer window and select Close Project.
Running the Code – Stand-alone Operation (No Emulator)


26. Disconnect the controlSTICK from the computer USB port.

27. Re-connect the controlSTICK to the computer USB port.

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

End of Exercise
Lab 4 Reference: Programming the Flash

Flash Memory Section Blocks

<table>
<thead>
<tr>
<th>Section</th>
<th>Origin</th>
<th>Length</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLASH</td>
<td>0x3D 8000</td>
<td>0xFF80</td>
<td>0</td>
</tr>
<tr>
<td>CSM_RSVD</td>
<td>0x3F 7FF8</td>
<td>0x76</td>
<td>0</td>
</tr>
<tr>
<td>BEGIN_FLASH</td>
<td>0x3F 7FF6</td>
<td>0x2</td>
<td>0</td>
</tr>
<tr>
<td>PASSWORDS</td>
<td>0x3F 7FF8</td>
<td>0x8</td>
<td>0</td>
</tr>
</tbody>
</table>

Startup Sequence from Flash Memory

1. RESET
2. Boot ROM (32Kw)
3. Boot Code (0x3FF75C)
4. {SCAN GPIO}
5. "rts2800_ml.lib"
6. "user" code sections
   main ()
   {
   ......  
   ......  
   }
7. "user" code sections
   main ()
   {
   ......  
   ......  
   }

Lab_4.cmd

SECTIONS
{
  codestart  :> BEGIN_FLASH, PAGE = 0
  passwords  :> PASSWORDS, PAGE = 0
  csm_rsvd   :> CSM_RSVD, PAGE = 0
}
The Next Step…

Training

C2000 MCU Multi-day Training Course

The C2000 Microcontroller Multi-day Workshop is a hands-on technical course taught by qualified Texas Instruments instructors. It provides further in-depth training covered in much greater detail of the topics shown in the outline on this slide. Please check the TI website for the workshop schedule and enrollment information.
ControlSUITE is a single portal for all C2000 software and has been designed to minimize software development time. Included in controlSUITE are device-specific drivers and support software, as well as complete system design examples used in sophisticated applications.

ControlSUITE is a one-stop, single centralized location to find all of your C2000 software needs. Download controlSUITE from the TI website.
Development Tools

C2000 Experimenter’s Kits
F28069, F28035, F28027, F28335, F2808, C28343, C28346, F28M35, F28377D

- Experimenter Kits include
  - controlCARD
  - USB docking station
  - C2000 Applications Software CD with example code and full hardware details
  - Code Composer Studio

- Docking station features
  - Access to controlCARD signals
  - Breadboard areas
  - Onboard USB JTAG Emulation
    - JTAG emulator not required

- Available through TI authorized distributors and the TI eStore

Part Number:
- TMDSDOCK28069
- TMDSDOCK28035
- TMDSDOCK28027
- TMDSDOCK28335
- TMDSDOCK2808
- TMDSDOCKH52C1
- TMDSDOCK28377D

JTAG emulator required for:
- TMDSDOCK28343
- TMDSDOCK28346-168

The C2000 development kits are designed to be modular and robust. These kits are complete, open source, evaluation and development tools where the user can modify both the hardware and software to best fit their needs.

The various Experimenter’s Kits shown on this slide include a specific controlCARD and Docking Station. Most have onboard USB JTAG emulation and no external emulator or power supply is required. However, where noted, the kits based on a DIMM-168 controlCARD include a 5-volt power supply and require an external JTAG emulator.
The Peripheral Explorer Kit provides a simple way to learn and interact with all F28335 peripherals. It includes onboard USB JTAG emulation.

The controlSTICK is an entry-level evaluation kit. It is a simple, stand-alone tool that allows users to learn the device and software quickly and easily.
The LaunchPad is a low-cost evaluation kit. Like the controlSTICK, it is a simple, stand-alone tool that allows users to learn the device and software quickly and easily. Additionally, various Booster Packs are available.

The controlCARD based Application Kits demonstrate the full capabilities of the C2000 device in
an application. All kits are completely open source with full documentation.

**C2000 Workshop Download Wiki**

![C2000 Workshop Download Wiki](http://www.ti.com/hands-on-training)

Please be sure to visit the C2000 Workshop Download Wiki. Here, you will find all of the materials for the C2000 One-day Workshop and the C2000 Multi-day Workshop, as well as the C2000 archived workshops, which include support for the F2407, F2812, F2808, and F28335 device families.
Development Support

For More Information . . .

- USA – Product Information Center (PIC)
  - Phone: 800-477-8924 or 512-434-1560
  - E-mail: support@ti.com
- TI E2E Community (videos, forums, blogs)
  - http://e2e.ti.com
- Embedded Processor Wiki
  - http://processors.wiki.ti.com
- TI Training
  - http://www.ti.com/training
- TI eStore
  - http://estore.ti.com
- TI website
  - http://www.ti.com

For more information and support, you can contact the product information center, visit the TI E2E community, embedded processor Wiki, TI training web page, TI eStore, and the TI website.

Thank you for attending this C2000 Microcontroller One-Day Workshop. We hope it was beneficial to you. For more technical information on the C2000 devices, please consider taking a multi-day hands-on workshop.
Notes: