F2803x Firmware Development Package

USER'S GUIDE

Copyright

Texas Instruments
12203 Southwest Freeway
Houston, TX 77477
http://www.ti.com/c2000

Revision Information

This is version V125 of this document, last updated on July 18, 2011.

Table of Contents

1 Introduction

The Texas Instruments® F2803x Firmware Development Package is a collection of device header files, common source files, helper libraries and example applications for the 2803X line of devices in the Piccolo portfolio.

The package comes with a complete set of example projects that demonstrate the basics of getting started with a Piccolo device and working with its different peripheral modules.

Chapter 2 talks about how the software package is structured, how the header files are organized and used in the example applications. The peripheral bit-field structure approach is presented in detail along with step-by-step instructions on how to use it in your code. A complete revision history of the header files is provided at the end of the chapter.

Chapter 3 provides step-by-step instructions on how to create a project from scratch and then go about debugging it. Its a good place to start if this is your first interaction with a piccolo device.

Chapter 4 covers all the examples provided in the development package; what each example does, its setup and observation procedures and, in a few cases, the mathematics involved in setting up control values for peripherals.
The examples for Piccolo(2803x) can be found in the *DSP2803x_examples_ccsv4* directory. As users move past evaluation, and get started developing their own application, TI recommends they maintain a similar project directory structure to that used in the example projects.

The Appendix covers the following topics

1. **Appendix A** - describes the default hardware prioritizing of Interrupt Software Routines and how it can be over-ridden in software.
2. **Appendix B** - Each factory programmed device from TI has compensation routines in OTP memory for oscillator drift due to temperature fluctuations. These routines are described here.
3. **Appendix C**- is the errata to version 6 of the Scale Factor Optimization Library. It describes updates to any of the SFO v6 library files

2 Header File Quickstart

2.1 Device Support

This software package supports 2803x devices. This includes the following: TMS320F28035, TMS320F28034, TMS320F28033, TMS320F28032, TMS320F28031, and TMS320F28030. Throughout this document, TMS320F28035, TMS320F28034, TMS320F28033, TMS320F28032, TMS320F28031, and TMS320F28030 are abbreviated as F28035, F28034, F28033, F28032, F28031, and F28030 respectively.

2.2 Introduction

The 2803x C/C++ peripheral header files and example projects facilitate writing in C/C++ Code for the Texas Instruments TMS320x2803x devices. The code can be used as a learning tool or as the basis for a development platform depending on the current needs of the user.

1. **Learning Tool**
   This download includes several example Code Composer Studio^{TM} v 4.0+ [1] projects for a 2803x development platform.
   These examples demonstrate the steps required to initialize the device and utilize the on-chip peripherals. The provided examples can be copied and modified giving the user a platform to quickly experiment with different peripheral configurations.
   These projects can also be migrated to other devices by simply changing the memory allocation in the linker command file.
2. **Development Platform**
   The peripheral header files can easily be incorporated into a new or existing project to provide a platform for accessing the on-chip peripherals using C or C++ code. In addition, the user can pick and choose functions from the provided code samples as needed and discard the rest.

To get started this document provides the following information:

1. Overview of the bit-field structure approach used in the 2803x C/C++ peripheral header files.
2. Overview of the included peripheral example projects.

[1] Code Composer Studio is a trademark of Texas Instruments (www.ti.com).

3. Steps for integrating the peripheral header files into a new or existing project.
4. Troubleshooting tips and frequently asked questions.
5. Migration tips for users moving from the 280x header files to the 2803x header files.

Finally, this document does not provide a tutorial on writing C code, using Code Composer Studio, or the C28x Compiler and Assembler. It is assumed that the reader already has a 2803x hardware platform setup and connected to a host with Code Composer Studio installed. The user should have a basic understanding of how to use Code Composer Studio to download code through JTAG and perform basic debug operations.

2.2.1 Revision History(Summary)

1. **Version 1.25 controlSUITE update**
   - Fixed bugs with examples and header files. (Details in section 2.9)
   - Updated all example comments to the doxygen format
2. **Version 1.24, controlSUITE update**
   - Improvements and additions to the CCS 4 projects in controlSUITE. GEL files for 2803x have been updated. A detailed revision history can be found in 2.9.
3. **Version 1.23, controlSUITE update**
   - Improvements and additions to the CCS 4 projects in controlSUITE. A detailed revision history can be found in 2.9.
4. **Version 1.22, controlSUITE update**
   - Improvements to the CCS 4 projects in controlSUITE. A detailed revision history can be found in 2.9.
5. **Version 1.21**
   - This version includes minor fixes to a couple of the files. A detailed revision history can be found in 2.9.
6. **Version 1.20**
   - This version includes minor corrections and comment fixes to the header files and examples. A detailed revision history can be found in 2.9.
7. **Version 1.10**
   - This version includes minor corrections and comment fixes to the header files and examples, and also adds a separate example folder, DSP2803x_examples_ccsv4, with examples supported by the Eclipse-based Code Composer Studio v4. A detailed revision history can be found in 2.9.
8. **Version 1.01**
   - This version includes minor corrections to comments in the common files, and adds additional LIN and ADC temperature sensor examples. A detailed revision history can be found in 2.9.
9. **Version 1.00**
   - This version is the first release of the 2803x header files and examples. It is an internal release used for customer trainings and tools releases.

2.2.2 Directory Structure

As installed, the 2803x C/C++ Header Files and Peripheral Examples is partitioned into a well-defined directory structure(see figure 2.1).

Table 2.1 describes the contents of the main directories used by 2803x header files and peripheral examples:

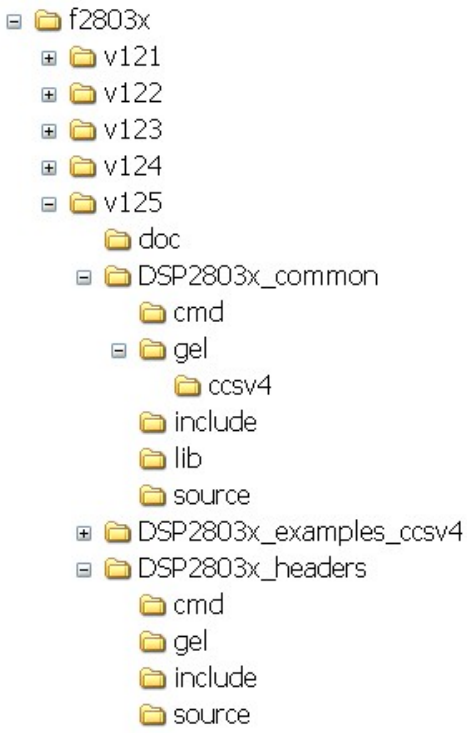

Figure 2.1: DSP2803x Main Directory Structure

Under the DSP2803x_headers and DSP2803x_common directories the source files are further broken down into sub-directories each indicating the type of file. Table 2.2 lists the sub-directories and describes the types of files found within each:

2.3 Understanding The Peripheral Bit-Field Structure Approach

The following application note includes useful information regarding the bit-field peripheral structure approach used by the header files and examples. This method is compared to traditional #define macros and topics of code efficiency and special case registers are also addressed. The information in this application note is important to understand the impact using bit fields can have on your application code.

Programming TMS320x28xx and 28xxx Peripherals in C/C++ (SPRAA85)

Directory	Description
<base>	Base install directory
<base>	Documentation including the revision history from the previous release.
<base>2803x_headers	Files required to incorporate the peripheral header files into a project. The header files use the bit-field structure approach described in Section 2.3. Integrating the header files into a new or existing project is described in Section 2.5.
<base>2803x_examples_ccsv4	Example Code Composer Studio v4 projects. These example projects illustrate how to configure many of the on-chip peripherals. An overview of the examples is given in Section 2.4.
<base>DSP2803x_common	Common source files shared across example projects to illustrate how to perform tasks using header file approach. Use of these files is optional, but may be useful in new projects. A list of these files is in Section 2.8.

Table 2.1: DSP2803x Main Directory Structure

Sub-Directory	Description
DSP2803x_headers\cmd	Linker command files that allocate the bit-field structures described in Section 2.3.
DSP2803x_headers\source	Source files required to incorporate the header files into a new or existing project.
DSP2803x_headers\include	Header files for each of the on-chip peripherals.
DSP2803x_common\cmd	Example memory command files that allocate memory on the devices.
DSP2803x_common\include	Common .h files that are used by the peripheral examples.
DSP2803x_common\source	Common .c files that are used by the peripheral examples.
DSP2803x_common\lib	Common library (.lib) files that are used by the peripheral examples.
DSP2803x_common\gel\ccsv4	Code Composer Studio v4.x GEL files for each device. These are optional.

Table 2.2: DSP2803x Sub-Directory Structure

2.4 Peripheral Example Projects

This section describes how to get started with and configure the peripheral examples included in the 2803x Header Files and Peripheral Examples software package.

2.4.1 Getting Started in Code Composer Studio v4.0+

To get started, follow these steps to load the 32-bit CPU-Timer example. Other examples are set-up in a similar manner.

1. **Have a hardware platform connected to a host with Code Composer Studio installed**
   NOTE: As supplied, the 2803x example projects are built for the 28035 device. If you are using another 2803x device, the memory definition in the linker command file (.cmd)

will need to be changed and the project rebuilt.

2. **Open the example project** Each example has its own project directory which is "imported"/opened in Code Composer Studio v4. To open the 2803x CPU-Timer example project directory, follow the following steps:
   - In Code Composer Studio v 4.x: Project->Import Existing CCS/CCE Eclipse Project.
   - Next to "Select Root Directory", browse to the CPU Timer example directory: DSP2803x_examples_ccsv4\cpu_timer. Select the Finish button. This will import/open the project in the CCStudio v4 C/C++ Perspective project window.
3. **Edit DSP28_Device.h** Edit the DSP2803x_Device.h file and make sure the appropriate device is selected. By default the 28035 is selected.

```
//---------------------------------------------------------------------------
// DSP2803x_headers\include\DSP2803x_Device.h
//---------------------------------------------------------------------------

#define   TARGET   1

//---------------------------------------------------------------------------
// User To Select Target Device:

#define   DSP28_28030PAG   0
#define   DSP28_28030PN    0

#define   DSP28_28031PAG   0
#define   DSP28_28031PN    0

#define   DSP28_28032PAG   0
#define   DSP28_28032PN    0

#define   DSP28_28033PAG   0
#define   DSP28_28033PN    0

#define   DSP28_28034PAG   0
#define   DSP28_28034PN    0

#define   DSP28_28035PAG   0
#define   DSP28_28035PN    TARGET
```

4. **Edit DSP2803x_Examples.h** Edit DSP2803x_Examples.h and specify the clock rate, the PLL control register value (PLLCR and DIVSEL). These values will be used by the examples to initialize the PLLCR register and DIVSEL bits.

   The default values will result in a 60MHz SYSCLKOUT frequency.

```
//---------------------------------------------------------------------------
// DSP2803x_common\include\DSP2803x_Examples.h
//---------------------------------------------------------------------------
//-----------------------------------------------------------------------
Specify the PLL control register (PLLCR) and divide
select (DIVSEL) value.
-----------------------------------------------------------------------*/
//#define DSP28_DIVSEL   0 // Enable /4 for SYSCLKOUT(default at reset)
//#define DSP28_DIVSEL   1 // Disable /4 for SYSCKOUT
```

```
#define DSP28_DIVSEL     2 // Enable /2 for SYSCLKOUT
//#define DSP28_DIVSEL   3 // Enable /1 for SYSCLKOUT

  #define DSP28_PLLCR   12
//#define DSP28_PLLCR   11
//#define DSP28_PLLCR   10
//#define DSP28_PLLCR    9
//#define DSP28_PLLCR    8
//#define DSP28_PLLCR    7
//#define DSP28_PLLCR    6
//#define DSP28_PLLCR    5
//#define DSP28_PLLCR    4
//#define DSP28_PLLCR    3
//#define DSP28_PLLCR    2
//#define DSP28_PLLCR    1
//#define DSP28_PLLCR    0  //(Default at reset) PLL is bypassed in
                             //this mode
//--------------------------------------------------------------------
```

In DSP2803x_Examples.h, also specify the SYSCLKOUT rate. This value is used to scale a delay loop used by the examples. The default value is for a 60 MHz SYSCLKOUT.

```
/*************************************************************
 DSP2803x_common\include\DSP2803x_Examples.h
*************************************************************/
...
#define CPU_RATE     16.667L   // for a 60MHz CPU clock speed (SYSCLKOUT)
//#define CPU_RATE   20.000L   // for a 50MHz CPU clock speed  (SYSCLKOUT)
//#define CPU_RATE   25.000L   // for a 40MHz CPU clock speed  (SYSCLKOUT)
//#define CPU_RATE   33.333L   // for a 30MHz CPU clock speed  (SYSCLKOUT)
...
```

5. **Review the comments at the top of the main source file: Example_2803xCpuTimer.c** A brief description of the example and any assumptions that are made and any external hardware requirements are listed in the comments at the top of the main source file of each example. In some cases you may be required to make external connections for the example to work properly.
6. **Perform any hardware setup required by the example** Perform any hardware setup indicated by the comments in the main source. The CPU-Timer example only requires that the hardware be setup for “Boot to SARAM” mode. Other examples may require additional hardware configuration such as connecting pins together or pulling a pin high or low. Table 2.3 shows a listing of the boot mode pin settings for your reference. Table 2.4 and Table 2.5 list the EMU boot modes (when emulator is connected) and the Get Mode boot mode options (mode is programmed into OTP) respectively. Refer to the documentation for your hardware platform for information on configuring the boot mode pins. For more information on the 2803x boot modes refer to the device specific *Boot ROM Reference Guide*.

   When the emulator is connected for debugging: TRSTn = 1, and therefore the device is in EMU boot mode. In this situation, the user must write the key value of 0x55AA to EMU_KEY at address 0x0D00 and desired EMU boot mode value to EMU_BMODE at 0x0D01 via the debugger window according to Table 2.4. The 2803x gel files in the DSP2803x_common/gel/ directory have a GEL function - EMU Boot Mode Select -> EMU_BOOT_SARAM() which performs the debugger write to boot to “SARAM” mode when called.

GPIO37 TDO	GPIO34 CMP2OUT	TRSTn	Mode
X	X	1	EMU Mode
0	0	0	Parallel I/O
0	1	0	SCI
1	0	0	Wait
1	1	0	“Get Mode”

Table 2.3: 2803x Boot Mode Settings

EMU_KEY 0x0D00	EMU_BMODE 0x0D01	Boot Mode Selected
!= 0x55AA	x	Wait
0x55AA	0x0000	Parallel I/O
	0x0001	SCI
	0x0002	Wait
	0x0003	Get Mode
	0x0004	SPI
	0x0005	I2C
	0x0006	OTP
	0x0007	eCAN
	0x0008	Wait
	0x000A	Boot to RAM
	0x000B	Boot to FLASH
	Other	Wait

Table 2.4: 2803x EMU Boot Modes (Emulator Connected)

When the emulator is not connected for debugging: SCI or Parallel I/O boot mode can be selected directly via the GPIO pins, or OTP_KEY at address 0x3D7BFE and OTP_BMODE at address 0x3D7BFF can be programmed for the desired boot mode per Table 2.5.

7. **Build and Load the code**

   Once any hardware configuration has been completed, in Code Composer Studio v4, go to *Target->Debug Active Project*.

   This will open the “Debug Perspective” in CCSv4, build the project, load the .out file into the 28x device, reset the part, and execute code to the start of the main function. By default, in Code Composer Studio v4, every time Debug Active Project is selected, the code is automatically built and the .out file loaded into the 28x device.

8. **Run the example, add variables to the watch window or examine the memory contents**

   At the top of the code in the comments section, there should be a list of “Watch variables”. To add these to the watch window, highlight them and right-click. Then select *Add Watch expression*. Now variables of interest are added to the watch window.

9. **Experiment, modify, re-build the example** If you wish to modify the examples it is suggested that you make a copy of the entire header file packet to modify or at least create a backup of the original files first. New examples provided by TI will assume that the base files are as supplied.

   Sections 2.4.2 and 2.4.2.3 describe the structure and flow of the examples in more detail.

10. **When done, delete the project from the Code Composer Studio v4 workspace**

    Go to *View->C/C++ Projects* to open up your project view. To remove/delete the project from the workspace, right click on the project’s name and select delete. Make sure the *Do not*

OTP_KEY 0x3D7BFE	OTP_BMODE 0x3D7BFF	Boot Mode Selected
!= 0x55AA	x	Get Mode - Flash
0x55AA	0x0001	Get Mode - SCI
	0x0003	Get Mode - Flash
	0x0004	Get Mode - SPI
	0x0005	Get Mode - I2C
	0x0006	Get Mode - OTP
	0x0007	Get Mode - eCAN
	Other	Get Mode - Flash

Table 2.5: 2803x GET Boot Modes (Emulator Disconnected)

delete contents button is selected, then select *Yes*. This does not delete the project itself. It merely removes the project from the workspace until you wish to open/import it again.

The examples use the header files in the DSP2803x_headers directory and shared source in the DSP2803x_common directory. Only example files specific to a particular example are located within in the example directory.

Note: Most of the example code included uses the .bit field structures to access registers. This is done to help the user learn how to use the peripheral and device. Using the bit fields has the advantage of yielding code that is easier to read and modify. This method will result in a slight code overhead when compared to using the .all method. In addition, the example projects have the compiler optimizer turned off. The user can change the compiler settings to turn on the optimizer if desired.

2.4.2 Example Program Structure

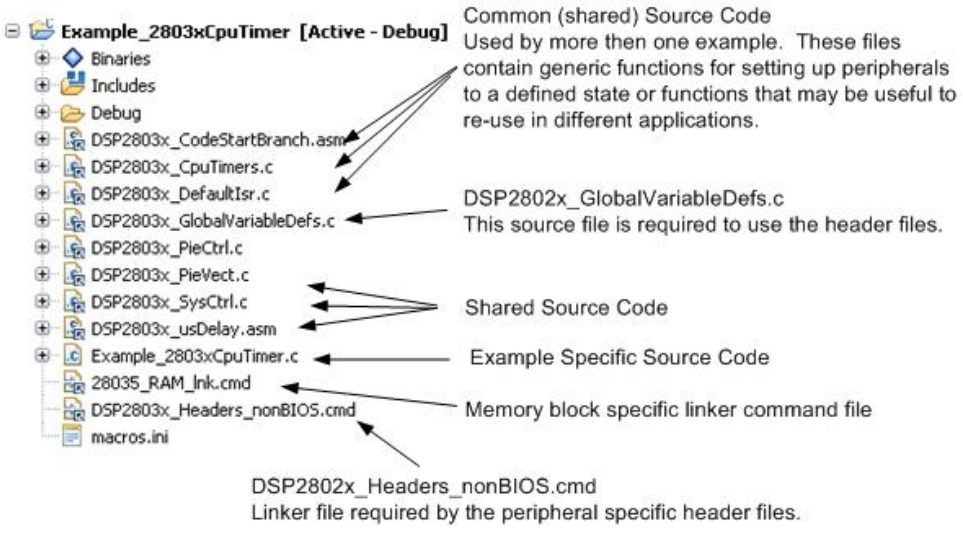

Figure 2.2: Example Program Structure

Each of the example programs has a very similar structure. This structure includes unique source code, shared source code, header files and linker command files.

```
//###########################################################################
// DSP2803x_examples\cpu_timer\Example_2803xCpuTimer.c
//###########################################################################

#include "DSP28x_Project.h" // Device Headerfile and Examples Include File
```

- **DSP28x_Project.h**
  This header file includes DSP2803x_Device.h and DSP2803x_Examples.h. Because the name is device-generic, example/custom projects can be easily ported between different device header files. This file is found in the <base>\DSP2803x_common\include directory.
- **DSP2803x_Device.h**
  This header file is required to use the header files. This file includes all of the required peripheral specific header files and includes device specific macros and typedef statements. This file is found in the <base>\DSP2803x_headers\include directory.
- **DSP2803x_Examples.h**
  This header file defines parameters that are used by the example code. This file is not required to use just the DSP2803x peripheral header files but is required by some of the common source files. This file is found in the <base>\DSP2803x_common\include directory.

2.4.2.1 Source Code

Each of the example projects consists of source code that is unique to the example as well as source code that is common or shared across examples.

- **DSP2803x_GlobalVariableDefs.c**
  Any project that uses the DSP2803x peripheral header files must include this source file. In this file are the declarations for the peripheral register structure variables and data section assignments. This file is found in the <base>\DSP2803x_headers\source directory.
- **Example specific source code**
  Files that are specific to a particular example have the prefix Example_2803x in their filename. For example Example_2803xCpuTimer.c is specific to the CPU Timer example and not used for any other example. Example specific files are located in the <base>\DSP2803x_examples_ccsv4\<example> directory.
- **Common source code**
  The remaining source files are shared across the examples. These files contain common functions for peripherals or useful utility functions that may be re-used. Shared source files are located in the DSP2803x_common\source directory. Users may choose to incorporate none, some, or the entire shared source into their own new or existing projects.

2.4.2.2 Linker Command Files

Each example uses two linker command files. These files specify the memory where the linker will place code and data sections. One linker file is used for assigning compiler generated sections to the memory blocks on the device while the other is used to assign the data sections of the peripheral register structures used by the DSP2803x peripheral header files.

- **Memory block linker allocation**
  The linker files shown in Table 2.6 are used to assign sections to memory blocks on the device. These linker files are located in the <base>\DSP2803x_common\cmd directory. Each example will use one of the following files depending on the memory used by the example.

Memory Linker Command File Examples	Location	Description
28035_RAM_lnk.cmd	DSP2803x_common	28035 memory linker command file. Includes all of the internal SARAM blocks on 28035 device. “RAM” linker files do not include flash or OTP blocks.
28034_RAM_lnk.cmd	DSP2803x_common	28034 SARAM memory linker command file.
28033_RAM_lnk.cmd	DSP2803x_common	28033 SARAM memory linker command file.
28032_RAM_lnk.cmd	DSP2803x_common	28032 SARAM memory linker command file.
28031_RAM_lnk.cmd	DSP2803x_common	28031 SARAM memory linker command file.
28030_RAM_lnk.cmd	DSP2803x_common	28030 SARAM memory linker command file
28035_RAM_CLA_lnk.cmd	DSP2803x_common	28035 SARAM CLA memory linker command file. Includes CLA message RAM.
28033_RAM_CLA_lnk.cmd	DSP2803x_common	28033 SARAM CLA memory linker command file.
F28035.cmd	DSP2803x_common	F28035 memory linker command file. Includes all Flash, OTP and CSM password protected memory locations.
F28035.cmd	DSP2803x_common	F28035 memory linker command file.
F28034.cmd	DSP2803x_common	F28034 memory linker command file.
F28033.cmd	DSP2803x_common	F28033 memory linker command file.
F28032.cmd	DSP2803x_common	F28032 memory linker command file.
F28031.cmd	DSP2803x_common	F28031 memory linker command file.
F28030.cmd	DSP2803x_common	F28030 memory linker command file.

Table 2.6: Included Memory Linker Command Files

- **Header file structure data section allocation**
  Any project that uses the header file peripheral structures must include a linker command file that assigns the peripheral register structure data sections to the proper memory location. These files are described in Table 2.7.

2.4.2.3 Documentation

This document is linked into each project so it can easily be opened through the project view. To do this, right click on the document within CCS, select “open with” and “system editor”.

2.4.3 Example Program Flow

All of the example programs follow a similar recommended flow for setting up a 2803x device.

Header File Linker Command File	Location	Description
DSP2803x_Headers_BIOS.cmd	DSP2803x_headers	Linker .cmd file to assign the header file variables in a BIOS project. This file must be included in any BIOS project that uses the header files. Refer to section 2.5.2.
DSP2803x_Headers_nonBIOS.cmd	DSP2803x_headers	Linker .cmd file to assign the header file variables in a non-BIOS project. This file must be included in any non-BIOS project that uses the header files. Refer to section 2.5.2.

Table 2.7: DSP2803x Peripheral Header Linker Command File

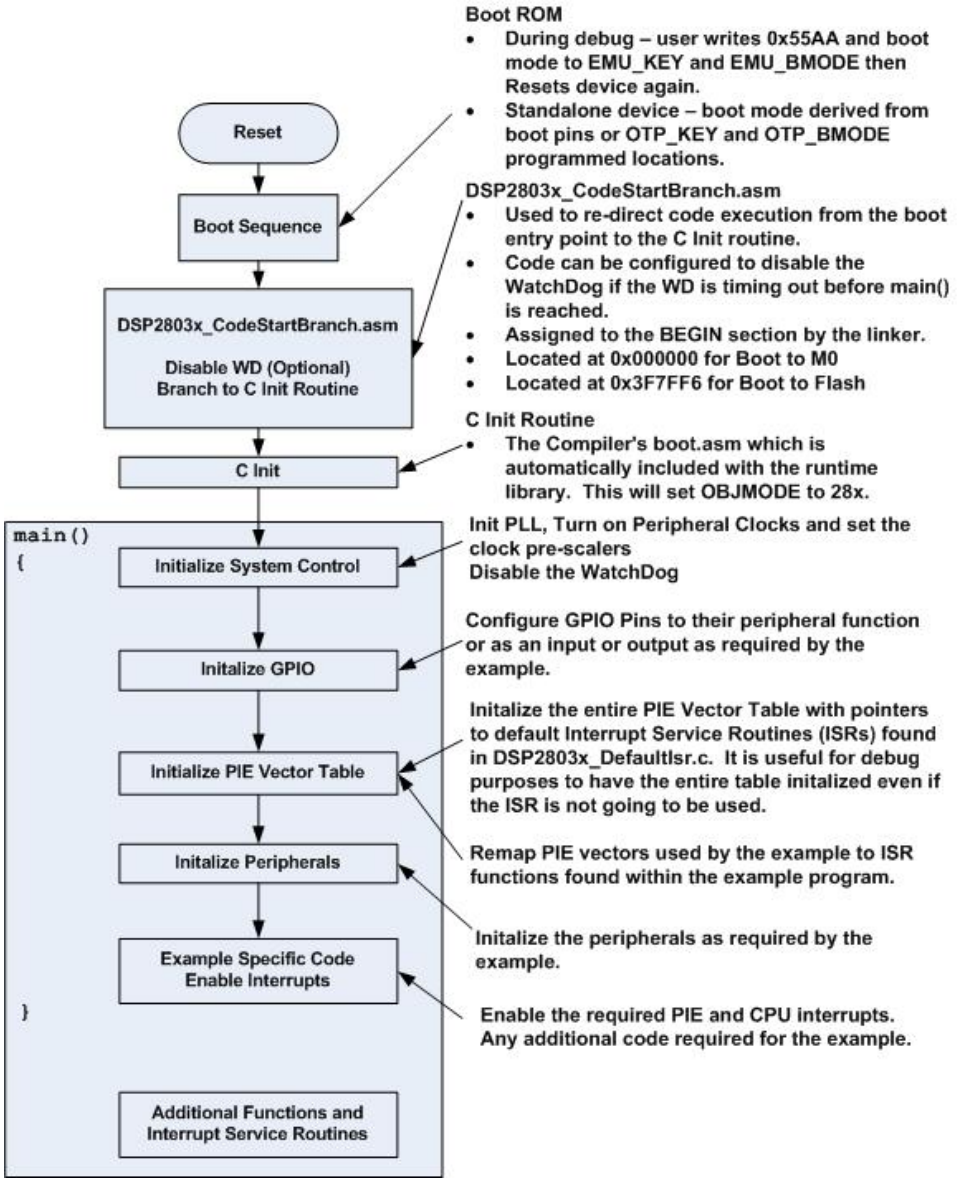

Figure 2.3: Flow for Example Programs

2.4.4 Included Examples

See Chapter 4 for a complete listing and description of available examples

2.4.5 Executing the Examples From Flash

Most of the DSP2803x examples execute from SARAM in “boot to SARAM” mode. One example, DSP2803x_examples\flash_f28035, executes from flash memory in “boot to flash” mode. This example is the PWM timer interrupt example with the following changes made to execute out of flash:

1. **Change the linker command file to link the code to flash**
   Remove 28035_RAM_lnk.cmd from the project and link one of the flash based linker files (ex: F28035.cmd, F28034.cmd, F28033.cmd, F28032.cmd, F28031.cmd, or F28030.cmd). These files are located in the <base>DSP2803x_common\cmd directory.
2. **Link the DSP2803x_common\source\DSP2803x_CSMPasswords.asm to the project**
   This file contains the passwords that will be programmed into the Code Security Module (CSM) password locations. Leaving the passwords set to 0xFFFF during development is recommended as the device can easily be unlocked. For more information on the CSM refer to the appropriate *System Control and Interrupts Reference Guide*.
3. **Modify the source code to copy all functions that must be executed out of SARAM from their load address in flash to their run address in SARAM**
   In particular, the flash wait state initialization routine must be executed out of SARAM. In the DSP2803x, functions that are to be executed from SARAM have been assigned to the ramfuncs section by compiler CODE_SECTION #pragma statements as shown in the example below.

```
/*********************************************************************
 DSP2803x_common\source\DSP2803x_SysCtrl.c
*********************************************************************/

#pragma CODE_SECTION(InitFlash, "ramfuncs");
```

   The ramfuncs section is then assigned to a load address in flash and a run address in SARAM by the memory linker command file as shown below:

```
/*********************************************************************
 DSP2803x_common\include\F28035.cmd
*********************************************************************/
SECTIONS
{
   ramfuncs    : LOAD = FLASHA,
                 RUN = RAML0,
                 LOAD_START(_RamfuncsLoadStart),
                 LOAD_END(_RamfuncsLoadEnd),
                 RUN_START(_RamfuncsRunStart),
                 PAGE = 0
}
```

   The linker will assign symbols as specified above to specific addresses as follows:
   These symbols can then be used to copy the functions from the Flash to SARAM using the included example MemCopy routine or the C library standard memcopy() function.
   To perform this copy from flash to SARAM using the included example MemCopy function:

   (a) Add the file DSP2803x_common\source\DSP2803x_MemCopy.c to the project.

Address	Symbol
Load start address	RamfuncsLoadStart
Load end address	RamfuncsLoadEnd
Run start address	RamfuncsRunStart

Table 2.8: Linker Symbol assignment

(b) Add the following function prototype to the example source code. This is done for you in the DSP2803x_Examples.h file.

```
/*************************************************************
 DSP2803x_common\include\DSP2803x_Examples.h
*************************************************************/

MemCopy(&RamfuncsLoadStart, &RamfuncsLoadEnd, &RamfuncsRunStart);
```

Note: IF RUNNING FROM FLASH, PLEASE COPY OVER THE SECTION "ramfuncs" FROM FLASH TO RAM PRIOR TO CALLING InitSysCtrl() or InitAdc(). THIS PREVENTS THE MCU FROM THROWING AN EXCEPTION WHEN A CALL TO DELAY_US() IS MADE.

(c) Add the following variable declaration to your source code to tell the compiler that these variables exist. The linker command file will assign the address of each of these variables as specified in the linker command file as shown in step 3. For the DSP2803x example code this has is already done in DSP2803x_Examples.h.

```
/*************************************************************
 DSP2803x_common\include\DSP2803x_GlobalPrototypes.h
*************************************************************/

extern Uint16 RamfuncsLoadStart;
extern Uint16 RamfuncsLoadEnd;
extern Uint16 RamfuncsRunStart;
```

(d) Modify the code to call the example MemCopy function for each section that needs to be copied from flash to SARAM.

```
/*************************************************************
 DSP2803x_examples\Flash source file
*************************************************************/

MemCopy(&RamfuncsLoadStart, &RamfuncsLoadEnd, &RamfuncsRunStart);
```

4. **Modify the code to call the flash initialization routine**
   This function will initialize the wait states for the flash and enable the Flash Pipeline mode.

```
/*************************************************************
 DSP2803x peripheral example .c file
*************************************************************/

InitFlash();
```

5. **Set the required jumpers for "boot to Flash" mode** The required jumper settings for each boot mode are shown in Table 2.9, Table 2.10, and Table 2.11.

GPIO37 TDO	GPIO34 CMP2OUT	TRSTn	Mode
X	X	1	EMU Mode
0	0	0	Parallel I/O
0	1	0	SCI
1	0	0	Wait
1	1	0	“Get Mode”

Table 2.9: 2803x Boot Mode Settings

EMU_KEY 0x0D00	EMU_BMODE 0x0D01	Boot Mode Selected
!= 0x55AA	x	Wait
0x55AA	0x0000	Parallel I/O
	0x0001	SCI
	0x0002	Wait
	0x0003	Get Mode
	0x0004	SPI
	0x0005	I2C
	0x0006	OTP
	0x0007	eCAN
	0x0008	Wait
	0x000A	Boot to RAM
	0x000B	Boot to FLASH
	Other	Wait

Table 2.10: 2803x EMU Boot Modes (Emulator Connected)

When the emulator is connected for debugging
TRSTn = 1, and therefore the device is in EMU boot mode. In this situation, the user must write the key value of 0x55AA to EMU_KEY at address 0x0D00 and the desired EMU boot mode value to EMU_BMODE at 0x0D01 via the debugger window according to Table 2.10.
When the emulator is not connected for debugging
SCI or Parallel I/O boot mode can be selected directly via the GPIO pins, or OTP_KEY at address 0x3D7BFE and OTP_BMODE at address 0x3D7BFF can be programmed for the desired boot mode per the tables above
Refer to the documentation for your hardware platform for information on configuring the boot mode selection pins. For more information on the 2803x boot modes refer to the appropriate *Boot ROM Reference Guide*.

6. **Program the device with the built code**
   In Code Composer Studio v4, when code is loaded into the device during debug, it automatically programs to flash memory.
   This can also be done using SDFlash available from Spectrum Digital’s website (Spectrum Digital). In addition the C2000 On-chip Flash programmer plug-in for Code Composer Studio v3.x can be used.
   These tools will be updated to support new devices as they become available. Please check for updates.
7. **In Code Composer Studio v3, to debug, load the project in CCS, select File->Load Symbols->Load Symbols Only**
   It is useful to load only symbol information when working in a debugging environment

OTP_KEY 0x3D7BFE	OTP_BMODE 0x3D7BFF	Boot Mode Selected
!= 0x55AA	**x**	**Get Mode - Flash**
0x55AA	0x0001	Get Mode - SCI
	0x0003	**Get Mode - Flash**
	0x0004	Get Mode - SPI
	0x0005	Get Mode - I2C
	0x0006	Get Mode - OTP
	0x0007	Get Mode - eCAN
	Other	**Get Mode - Flash**

Table 2.11: 2803x GET Boot Modes (Emulator Disconnected)

where the debugger cannot or need not load the object code, such as when the code is in ROM or flash. This operation loads the symbol information from the specified file.

2.5 Steps for Incorporating the Header Files and Sample Code

Follow these steps to incorporate the peripheral header files and sample code into your own projects. If you already have a project that uses the DSP280x or DSP281x header files then also refer to Section 2.7 for migration tips.

2.5.1 Before you begin

Before you include the header files and any sample code into your own project, it is recommended that you perform the following:

1. **Load and step through an example project**
   Load and step through an example project to get familiar with the header files and sample code. This is described in Section 2.4.
2. **Create a copy of the source files you want to use**
   DSP2803x_headers: code required to incorporate the header files into your project
   DSP2803x_common: shared source code much of which is used in the example projects.
   DSP2803x_examples_ccsv4: 2803x floating-point compiled example projects that use the header files and shared code.

2.5.2 Including the DSP2803x Peripheral Header Files

Including the DSP2803x header files in your project will allow you to use the bit-field structure approach in your code to access the peripherals on the DSP. To incorporate the header files in a new or existing project, perform the following steps:

1. **#include “DSP2803x_Device.h” (or #include “DSP28x_Project.h”) in your source files**
   The DSP2803x_Device.h include file will in-turn include all of the peripheral specific header files and required definitions to use the bit-field structure approach to access the peripherals.

```
/*************************************************************************
 User's source file
*************************************************************************/

#include "DSP2803x_Device.h"
```

Another option is to #include “DSP28x_Project.h” in your source files, which in-turn includes “DSP2803x_Device.h” and “DSP2803x_Examples.h” (if it is not necessary to include common source files in the user project, the #include “DSP2803x_Examples.h” line can be deleted). Due to the device-generic nature of the file name, user code is easily ported between different device header files.

```
/*************************************************************************
 User's source file
*************************************************************************/

#include "DSP28x_Project.h"
```

2. **Edit DSP2803x_Device.h and select the target you are building for**
   In the below example, the file is configured to build for the 28035 device.

```
/*************************************************************************
 DSP2803x_headers\include\DSP2803x_Device.h
*************************************************************************/
#define   TARGET        1
#define   DSP28_28035   TARGET    // Selects '28035
#define   DSP28_28034   0         // Selects '28034
#define   DSP28_28033   0         // Selects '28033Ĕ etc
```

   By default, the 28035 device is selected.
3. **Add the source file DSP2803x_GlobalVariableDefs.c to the project**
   This file is found in the DSP2803x_headers\source directory and includes:
   - Declarations for the variables that are used to access the peripheral registers.
   - Data section #pragma assignments that are used by the linker to place the variables in the proper locations in memory.
4. **Add the appropriate DSP2803x header linker command file to the project.** As described in Section 2.4, when using the DSP2803x header file approach, the data sections of the peripheral register structures are assigned to the memory locations of the peripheral registers by the linker.
   To perform this memory allocation in your project, one of the following linker command files located in DSP2803x_headers\cmd must be included in your project:
   - For non-DSP/BIOS[2] projects: *DSP2803x_Headers_nonBIOS.cmd*
   - For DSP/BIOS projects: *DSP2803x_Headers_BIOS.cmd*

   The method for adding the header linker file to the project depends on preference
   Method #1:
   - Right-click on the project in the project window of the C/C++ Projects perspective.
   - Select Link Files to Project...
   - Navigate to the DSP2803x_headers\cmd directory on your system and select the desired .cmd file.

[2] DSP/BIOS is a trademark of Texas Instruments

Note: The limitation with Method #1 is that the path to <install directory>\DSP2803x_headers\cmd\<cmd file>.cmd is fixed on your PC. If you move the installation directory to another location on your PC, the project will "break" because it still expects the .cmd file to be in the original location. Use Method #2 if you are using "linked variables" in your project to ensure your project/installation directory is portable across computers and different locations on the same PC. (For more information, see: Portable_Projects_in_CCSv4_for_C2000

Method #2:

- Right-click on the project in the project window of the C/C++ Projects perspective.
- Select New->File.
- Click on the Advanced» button to expand the window.
- Check the Link to file in the file system check-box.
- Select the Variables... button. From the list, pick the linked variable (macro defined in your macros.ini file) associated with your installation directory. (For the 2803x header files, this is INSTALLROOT_2803X_V<version #>). For more information on linked variables and the macros.ini file, see: Portable_Projects_in_CCSv4_for_C2000:_for_Linking_Files_to_Project
- Click on the Extend... button. Navigate to the desired .cmd file and select OK.

5. **Add the directory path to the DSP2803x header files to your project**
   Code Composer Studio 4.x:
   To specify the directory where the header files are located:

   - Open the menu: Project->Properties.
   - In the menu on the left, select "C/C++ Build".
   - In the "Tool Settings" tab, Select "C2000 Compiler -> Include Options:"
   - In the "Add dir to #include search path (–include_path, -I)" window, select the "Add" icon in the top right corner.
   - Select the "File system..." button and navigate to the directory path of DSP2803x_headers\include on your system.

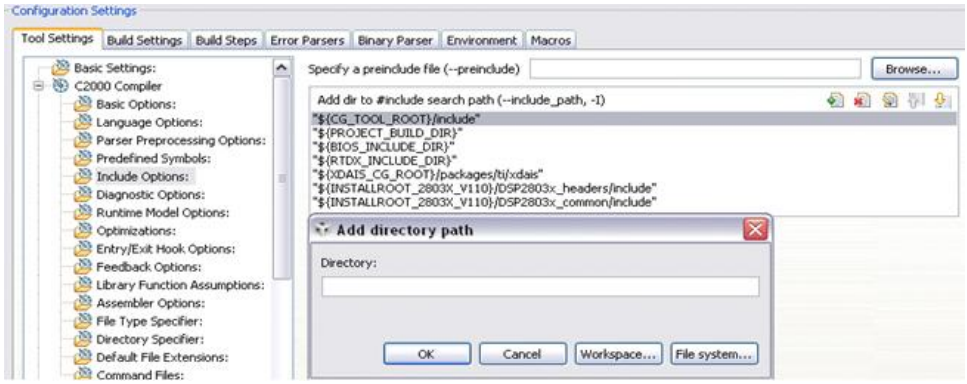

Figure 2.4: Adding device header file directories to the include search path

6. **Additional suggested build options** The following are additional compiler and linker options. The options can all be set via the Project-> Properties->Tool Settings sub-menus.
   - **C2000 Compiler**

  - **-ml Select Runtime Model Options and check -ml** Build for large memory model. This setting allows data sections to reside anywhere within the 4M-memory reach of the 28x devices.
  - **-pdr Select Diagnostic Options and check -pdr** Issue non-serious warnings. The compiler uses a warning to indicate code that is valid but questionable. In many cases, these warnings issued by enabling -pdr can alert you to code that may cause problems later on.

- **C2000 Linker**
  - **-w Select Diagnostics and check -w** Warn about output sections. This option will alert you if any unassigned memory sections exist in your code. By default the linker will attempt to place any unassigned code or data section to an available memory location without alerting the user. This can cause problems, however, when the section is placed in an unexpected location.
  - **-e Select Symbol Management and enter Program Entry Point -e** Defines a global symbol that specifies the primary entry point for the output module. For the DSP2802x examples, this is the symbol "code_start". This symbol is defined in the DSP2802x_common\source\DSP2802x_CodeStartBranch.asm file. When you load the code in Code Composer Studio, the debugger will set the PC to the address of this symbol. If you do not define a entry point using the -e option, then the linker will use _c_int00 by default.

2.5.3 Including Common Example Code

Including the common source code in your project will allow you to leverage code that is already written for the device. To incorporate the shared source code into a new or existing project, perform the following steps:

1. **#include "DSP2803x_Examples.h" (or "DSP28x_Project.h") in your source files.** The "DSP2803x_Examples.h" include file will include common definitions and declarations used by the example code.

```
/*****************************************************************
 User's source file
*****************************************************************/

#include "DSP2803x_Examples.h"
```

   Another option is to #include "DSP28x_Project.h" in your source files, which in-turn includes "DSP2803x_Device.h" and "DSP2803x_Examples.h". Due to the device-generic nature of the file name, user code is easily ported between different device header files.

```
/*****************************************************************
 User's source file
*****************************************************************/

#include "DSP28x_Project.h"
```

2. **Add the directory path to the example include files to your project.** To specify the directory where the header files are located:
   - Open the menu: Project->Properties.
   - In the menu on the left, select "C/C++ Build".

- In the “Tool Settings” tab, Select “C2000 Compiler -> Include Options:”
- In the “Add dir to #include search path (–include_path, -I)” window, select the “Add” icon in the top right corner.
- Select the “File system...” button and navigate to the directory path of DSP2803x_headers\include on your system.

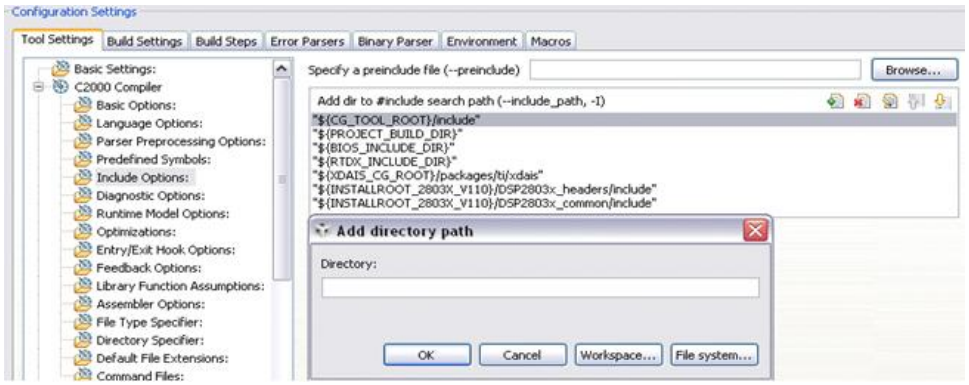

Figure 2.5: Adding Example header directories to the include search path

3. **Link a linker command file to your project.**
   The following memory linker .cmd files are provided as examples in the DSP2803x_common\cmd directory. For getting started the basic 28035_RAM_lnk.cmd file is suggested and used by most of the examples.

Memory Linker Command File Examples	Location	Description
28035_RAM_lnk.cmd	DSP2803x_common\cmd	28035 memory linker command file. Includes all of the internal SARAM blocks on a 28035 device. “RAM” linker files do not include flash or OTP blocks.
28034_RAM_lnk.cmd	DSP2803x_common\cmd	28034 SARAM memory linker command file.
28033_RAM_lnk.cmd	DSP2803x_common\cmd	28033 SARAM memory linker command file.
28032_RAM_lnk.cmd	DSP2803x_common\cmd	28032 SARAM memory linker command file.
28031_RAM_lnk.cmd	DSP2803x_common\cmd	28031 SARAM memory linker command file.
28030_RAM_lnk.cmd	DSP2803x_common\cmd	28030 SARAM memory linker command file.
28035_RAM_CLA_lnk.cmd	DSP2803x_common\cmd	28035 CLA memory linker command file. Includes CLA message RAM
28033_RAM_CLA_lnk.cmd	DSP2803x_common\cmd	28033 SARAM CLA memory linker command file.
F28035.cmd	DSP2803x_common\cmd	F28035 memory linker command file.
F28034.cmd	DSP2803x_common\cmd	F28034 memory linker command file.
F28033.cmd	DSP2803x_common\cmd	F28033 memory linker command file.
F28032.cmd	DSP2803x_common\cmd	F28032 memory linker command file.

Continued on next page

Table 2.12 – continued from previous page

Memory Linker Command File Examples	Location	Description
F28031.cmd	DSP2803x_common\cmd	F28031 memory linker command file.
F28030.cmd	DSP2803x_common\cmd	F28030 memory linker command file.

Table 2.12: Included Main Linker Command Files

4. **Set the CPU Frequency** In the DSP2803x_common\include\DSP2803x_Examples.h file specify the proper CPU frequency. Some examples are included in the file.

```
/*-----------------------------------------------------------------
 DSP2803x_common\include\DSP2803x_Examples.h
-----------------------------------------------------------------*/
...
#define CPU_RATE   16.667L   // for a 60MHz CPU clock speed (SYSCLKOUT)
//#define CPU_RATE   20.000L   // for a 50MHz CPU clock speed  (SYSCLKOUT)
//#define CPU_RATE   25.000L   // for a 40MHz CPU clock speed  (SYSCLKOUT)
...
```

5. **Link desired common source files to the project** The common source files are found in the DSP2803x_common\source directory.
6. **Include .c files for the PIE** Since all catalog 2803x applications make use of the PIE interrupt block, you will want to include the PIE support .c files to help with initializing the PIE. The shell ISR functions can be used directly or you can re-map your own function into the PIE vector table provided. A list of these files can be found in section 2.8.2.1

2.6 Troubleshooting Tips and Frequently Asked Questions

- **In the examples, what do “EALLOW;” and “EDIS;” do?**
  EALLOW; is a macro defined in DSP2803x_Device.h for the assembly instruction EALLOW and likewise EDIS is a macro for the EDIS instruction. That is EALLOW; is the same as embedding the assembly instruction asm(“ EALLOW”);
  Several control registers on the 28x devices are protected from spurious CPU writes by the EALLOW protection mechanism. The EALLOW bit in status register 1 indicates if the protection is enabled or disabled. While protected, all CPU writes to the register are ignored and only CPU reads, JTAG reads and JTAG writes are allowed. If this bit has been set by execution of the EALLOW instruction, then the CPU is allowed to freely write to the protected registers. After modifying the registers, they can once again be protected by executing the EDIS assembly instruction to clear the EALLOW bit.
  For a complete list of protected registers, refer to *TMS320x2803x System Control and Interrupts Reference Guide*.
- **Peripheral registers read back 0x0000 and/or cannot be written to**
  There are a few things to check:
  - Peripheral registers cannot be modified or unless the clock to the specific peripheral is enabled. The function InitPeripheralClocks() in the DSP2803x_common\source directory shows an example of enabling the peripheral clocks.
  - Some peripherals are not present on all 2803x family derivatives. Refer to the device datasheet for information on which peripherals are available.

  - The EALLOW bit protects some registers from spurious writes by the CPU. If your program seems unable to write to a register, then check to see if it is EALLOW protected. If it is, then enable access using the EALLOW assembly instruction. See *TMS320x2803x System Control and Interrupts Reference Guide* for a complete list of EALLOW protected registers.

- **Memory block L0, L1 read back all 0x0000**
  In this case most likely the code security module is locked and thus the protected memory locations are reading back all 0x0000. Refer to the *TMS320x2803x System Control and Interrupts Reference Guide* for information on the code security module.
- **Code cannot write to L0 or L1 memory blocks**
  In this case most likely the code security module is locked and thus the protected memory locations are reading back all 0x0000. Code that is executing from outside of the protected cannot read or write to protected memory while the CSM is locked. Refer to the *TMS320x2803x Control and Interrupts Reference Guide* for information on the code security module
- **A peripheral register reads back ok, but cannot be written to**
  The EALLOW bit protects some registers from spurious writes by the CPU. If your program seems unable to write to a register, then check to see if it is EALLOW protected. If it is, then enable access using the EALLOW assembly instruction. See *TMS320x2803x System Control and Interrupts Reference Guide* for a complete list of EALLOW protected registers.
- **I re-built one of the projects to run from Flash and now it doesn't work. What could be wrong?**
  Make sure all initialized sections have been moved to flash such as .econst and .switch. If you are using SDFlash, make sure that all initialized sections, including .econst, are allocated to page 0 in the linker command file (.cmd). SDFlash will only program sections in the .out file that are allocated to page 0.
- **Why do the examples populate the PIE vector table and then re-assign some of the function pointers to other ISRs?**
  The examples share a common default ISR file. This file is used to populate the PIE vector table with pointers to default interrupt service routines. Any ISR used within the example is then remapped to a function within the same source file. This is done for the following reasons:
  - The entire PIE vector table is enabled, even if the ISR is not used within the example. This can be very useful for debug purposes.
  - The default ISR file is left unmodified for use with other examples or your own project as you see fit.
  - It illustrates how the PIE table can be updated at a later time.
- **When I build the examples, the linker outputs the following: warning: entry point other than _c_int00 specified. What does this mean?**
  This warning is given when a symbol other then _c_int00 is defined as the code entry point of the project. For these examples, the symbol code_start is the first code that is executed after exiting the boot ROM code and thus is defined as the entry point via the -e linker option. This symbol is defined in the DSP2803x_CodeStartBranch.asm file. The entry point symbol is used by the debugger and by the hex utility. When you load the code, CCS will set the PC to the entry point symbol. By default, this is the _c_int00 symbol which marks the start of the C initialization routine. For the DSP2803x examples, the code_start symbol is used instead. Refer to the source code for more information.
- **When I build many of the examples, the compiler outputs the following: remark: controlling expression is constant. What does this mean?**
  Some of the examples run forever until the user stops execution by using a while(1) loop. The remark refers to the while loop using a constant and thus the loop will never be exited.

- **When I build some of the examples, the compiler outputs the following: warning: statement is unreachable. What does this mean?**
  Some of the examples run forever until the user stops execution by using a while(1) loop. If there is code after this while(1) loop then it will never be reached.
- **I changed the build configuration of one of the projects from "Debug" to "Release" and now the project will not build. What could be wrong?**
  When you switch to a new build configuration (Project->Active Build Configuration) the compiler and linker options changed for the project. The user must enter other options such as include search path and the library search path. Open the build options menu (Project-> Options) and enter the following information:
  * C2000 Compiler, Include Options: Include search path
  * C2000 Linker, File Search Path: Library search path
  * C2000 Linker, File Search Path: Include libraries(i.e. rts2800_ml.lib)

  Refer to section 5 for more details.
- **In the flash example I loaded the symbols and ran to main. I then set a breakpoint but the breakpoint is never hit. What could be wrong?**
  In the Flash example, the InitFlash function and several of the ISR functions are copied out of flash into SARAM. When you set a breakpoint in one of these functions, Code Composer will insert an ESTOP0 instruction into the SARAM location. When the ESTOP0 instruction is hit, program execution is halted. CCS will then remove the ESTOP0 and replace it with the original opcode. In the case of the flash program, when one of these functions is copied from Flash into SARAM, the ESTOP0 instruction is overwritten code. This is why the breakpoint is never hit. To avoid this, set the breakpoint after the SARAM functions have been copied to SARAM.
- **The eCAN control registers require 32-bit write accesses**
  The compiler will instead make a 16-bit write accesses if it can in order to improve code size and/or performance. This can result in unpredictable results.
  One method to avoid this is to create a duplicate copy of the eCAN control registers in RAM. Use this copy as a shadow register. First copy the contents of the eCAN register you want to modify into the shadow register. Make the changes to the shadow register and then write the data back as a 32-bit value. This method is shown in the DSP2803x_examples_ccsv4\ecan_back2back example project.

2.6.1 Effects of read-modify-write instructions

When writing any code, whether it be C or assembly, keep in mind the effects of read-modify-write instructions.
The 28x DSP will write to registers or memory locations 16 or 32-bits at a time. Any instruction that seems to write to a single bit is actually reading the register, modifying the single bit, and then writing back the results. This is referred to as a read-modify-write instruction. For most registers this operation does not pose a problem. A notable exception is:

1. **Registers with multiple flag bits in which writing a 1 clears that flag**
   For example, consider the PIEACK register. Bits within this register are cleared when writing a 1 to that bit. If more then one bit is set, performing a read-modify-write on the register may clear more bits then intended.
   The below solution is incorrect. It will write a 1 to any bit set and thus clear all of them:

```
/*************************************************************
 User's source file
*************************************************************/
```

```
PieCtrl.PIEACK.bit.Ack1 = 1;     // INCORRECT! May clear more bits.
```

The correct solution is to write a mask value to the register in which only the intended bit will have a 1 written to it:

```
/*************************************************************************
 User's source file
*************************************************************************/

#define PIEACK_GROUP1  0x0001
        ...
 PieCtrl.PIEACK.all = PIEACK_GROUP1;    // CORRECT!
```

2. **Registers with Volatile Bits**
   Some registers have volatile bits that can be set by external hardware.
   Consider the PIEIFRx registers. An atomic read-modify-write instruction will read the 16-bit register, modify the value and then write it back. During the modify portion of the operation a bit in the PIEIFRx register could change due to an external hardware event and thus the value may get corrupted during the write.
   The rule for registers of this nature is to never modify them during runtime. Let the CPU take the interrupt and clear the IFR flag.

2.7 Migration Tips for moving from the TMS320x280x header files to the TMS320x2803x header files

This section includes suggestions for moving a project from the 280x header files to the 2803x header files.

1. **Create a copy of your project to work with or back-up your current project**

2. **Open the project file(s) in a text editor**
   In Code Composer Studio v4.x:
   Open the .project, .cdtbuild, and macros.ini files in your example folder. Replace all instances of 280x with 2803x so that the appropriate source files and build options are used. Check the path names to make sure they point to the appropriate header file and source code directories. Also replace the header file version number for the paths and macro names as well where appropriate. For instance, if a macro name was INSTALLROOT_280X_V170 for your 280x project using 280x header files V1.70, change this to INSTALLROOT_2803X_V120 to migrate to the 2803x header files V1.20(or the latest version). If not using the default macro name for your header file version, be sure to change your macros according to your chosen macro name in the .project, .cdtbuild, and macros.ini files.
3. **Load the project into Code Composer Studio**
   Use the Edit-> find in files dialog to find instances of DSP280x_Device.h and DSP280x_Example.h for 280x header files. Replace these with DSP2803x_Device.h and DSP2803x_Example.h respectively (or instead with one DSP2803x_Project.h file).
4. **Make sure you are using the correct linker command files (.cmd) appropriate for your device and for the DSP2803x header files**
   You will have one file for the memory definitions and one file for the header file structure definitions. Using a 280x memory file can cause issues since the H0 memory block has been split, renamed, and/or moved on the 2803x.

5. **Build the project**
   The compiler will highlight areas that have changed. If migrating from the TMS320x280x header files, code should be mostly compatible after all instances of DSP280x are replaced with DSP2803x in all relevant files, and the above steps are taken. Additionally, several bits have been removed and/or replaced. See Table 2.13.

Peripheral	Register	Bit Name		Comment
		Old	New	
SysCtrlRegs	XCLK	Reserved(bit 6)	XCLKINSEL(bit 6)	On 2803x devices, XCLKIN can be fed via a GPIO pin. This bit selects either GPIO38 (default) or GPIO19 as XCLKIN input source.
	PLLSTS	CLKINDIV(bit 1)	DIVSEL (bits 8,7)	DIVSEL allows more values by which CLKIN can be divided.

Table 2.13: Summary of Register and Bit-Name Changes from DSP280x V1.60 DSP2803x V1.00

Additionally, unlike the DSP280x devices, the DSP2803x devices run off an internal oscillator (INTOSC1) by default. To switch between the 2 available internal clock sources and the traditional external oscillator clock source, a new register in the System Control register space - CLKCTL - is available.

2.8 Packet Contents

This section lists all of the files included in the release.

2.8.1 Header File Support - DSP2803x_headers

The DSP2803x header files are located in the <base>\DSP2803x_headers directory.

2.8.1.1 DSP2803x Header Files - Main Files

The files listed in Table 2.14 must be added to any project that uses the DSP2803x header files. Refer to section 2.5 for information on incorporating the header files into a new or existing project.

File	Location	Description
DSP2803x_Device.h	DSP2803x_headers\include	Main include file. Include this one file in any of your .c source files. This file in-turn includes all of the peripheral specific .h files listed below. In addition the file includes typedef statements and commonly used mask values. Refer to section 2.5.
DSP2803x_GlobalVariableDefs.c	DSP2803x_headers\source	Defines the variables that are used to access the peripheral structures and data section #pragma assignment statements. This file must be included in any project that uses the header files. Refer to section 2.5.
DSP2803x_Headers_nonBIOS.cmd	DSP2803x_headers\cmd	Linker .cmd file to assign the header file variables in a non-BIOS project. This file must be included in any non-BIOS project that uses the header files. Refer to section 2.5.

Table 2.14: DSP2803x Header Files - Main Files

2.8.1.2 DSP2803x Header Files - Peripheral Bit-Field and Register Structure Definition Files

The files listed in Table 2.15 define the bit-fields and register structures for each of the peripherals on the 2803x devices. These files are automatically included in the project by including DSP2803x_Device.h. Refer to section 2.4.2 for more information on incorporating the header files into a new or existing project.

File	Location	Description
DSP2803x_Adc.h	DSP2803x_headers\include	ADC register structure and bit-field definitions.
DSP2803x_BootVars.h	DSP2803x_headers\include	External boot variable definitions.
DSP2803x_Cla.h	DSP2803x_headers\include	CLA register structure and bit-field definitions
DSP2803x_Comp.h	DSP2803x_headers\include	Comparator register structure and bit-field definitions.
DSP2803x_CpuTimers.h	DSP2803x_headers\include	CPU-Timer register structure and bit-field definitions.
DSP2803x_DevEmu.h	DSP2803x_headers\include	Emulation register definitions
DSP2833x_ECan.h	DSP2803x_headers\include	eCAN register structures and bit-field definitions.
DSP2803x_ECap.h	DSP2803x_headers\include	eCAP register structures and bit-field definitions.
DSP2803x_EPwm.h	DSP2803x_headers\include	ePWM register structures and bit-field definitions.
DSP2833x_EQep.h	DSP2803x_headers\include	eQEP register structures and bit-field definitions.
DSP2803x_Gpio.h	DSP2803x_headers\include	General Purpose I/O (GPIO) register structures and bit-field definitions.
DSP2803x_I2c.h	DSP2803x_headers\include	I2C register structure and bit-field definitions.
DSP2833x_Lin.h	DSP2803x_headers\include	LIN register structures and bit-field definitions.
DSP2803x_NmiIntrupt.h	DSP2803x_headers\include	NMI interrupt register structure and bit-field definitions
DSP2803x_PieCtrl.h	DSP2803x_headers\include	PIE control register structure and bit-field definitions.
DSP2803x_PieVect.h	DSP2803x_headers\include	Structure definition for the entire PIE vector table.
DSP2803x_Sci.h	DSP2803x_headers\include	SCI register structure and bit-field definitions.
DSP2803x_Spi.h	DSP2803x_headers\include	SPI register structure and bit-field definitions.
DSP2803x_SysCtrl.h	DSP2803x_headers\include	System register definitions. Includes Watchdog, PLL, CSM, Flash/OTP, Clock registers.
DSP2803x_XIntrupt.h	DSP2803x_headers\include	External interrupt register structure and bit-field definitions.

Table 2.15: DSP2803x Header File Bit-Field Register Structure Definition Files

2.8.1.3 Variable Names and Data Sections

This section is a summary of the variable names and data sections allocated by the DSP2803x_headers\source\DSP2803x_GlobalVariableDefs.c file as shown in Table 2.16. Note that all peripherals may not be available on a particular 2803x device. Refer to the device datasheet for the peripheral mix available on each 2803x family derivative.

Peripheral	Starting Address	Structure Variable Name
ADC	0x007100	AdcRegs
ADC Mirrored Result Registers	0x000B00	AdcMirror
CLA1	0x001400	Cla1Regs
Code Security Module	0x000AE0	CsmRegs
Code Security Module Password Locations	0x3F7FF8-0x3F7FFF	CsmPwl
COMP1	0x006400	Comp1Regs
COMP2	0x006420	Comp2Regs
COMP3	0x006440	Comp3Regs
CPU Timer 0	0x000C00	CpuTimer0Regs
CPU Timer 1	0x000C08	CpuTimer1Regs
CPU Timer 2	0x000C10	CpuTimer2Regs
Device and Emulation Registers	0x000880	DevEmuRegs
System Power Control Registers	0x00985	SysPwrCtrlRegs
eCAN-A	0x006000	ECanaRegs
eCAN-A Mail Boxes	0x006100	ECanaMboxes
eCAN-A Local Acceptance Masks	0x006040	ECanaLAMRegs
eCAN-A Message Object Time Stamps	0x006080	ECanaMOTSRegs
eCAN-A Message Object Time-Out	0x0060C0	ECanaMOTORegs
ePWM1	0x006800	EPwm1Regs
ePWM2	0x006840	EPwm2Regs
ePWM3	0x006880	EPwm3Regs
ePWM4	0x0068C0	EPwm4Regs
ePWM5	0x006900	EPwm5Regs
ePWM6	0x006940	EPwm6Regs
ePWM7	0x006980	EPwm7Regs
eCAP1	0x006A00	ECap1Regs
eQEP1	0x006B00	EQep1Regs
External Interrupt Registers	0x007070	XIntruptRegs
Flash OTP Configuration Registers	0x000A80	FlashRegs
General Purpose I/O Data Registers	0x006fC0	GpioDataRegs
General Purpose Control Registers	0x006F80	GpioCtrlRegs
General Purpose Interrupt Registers	0x006fE0	GpioIntRegs
I2C	0x007900	I2caRegs
LIN-A	0x006C00	LinaRegs
NMI Interrupt	0x7060	NmiIntruptRegs
PIE Control	0x000CE0	PieCtrlRegs
SCI-A	0x007050	SciaRegs
SPI-A	0x007040	SpiaRegs
SPI-B	0x007740	SpibRegs

Table 2.16: DSP2803x Variable Names and Data Sections

2.8.2 Common Example Code - DSP2803x_common

2.8.2.1 Peripheral Interrupt Expansion (PIE) Block Support

In addition to the register definitions defined in DSP2803x_PieCtrl.h, this packet provides the basic ISR structure for the PIE block. These files are shown in Table 2.17.

File	Location	Description
DSP2803x_DefaultIsr.c	DSP2803x_common\source	Shell interrupt service routines (ISRs) for the entire PIE vector table. You can choose to populate one of functions or re-map your own ISR to the PIE vector table. Note: This file is not used for DSP/BIOS projects.
DSP2803x_DefaultIsr.h	DSP2803x_common\include	Function prototype statements for the ISRs in DSP2803x_DefaultIsr.c. Note: This file is not used for DSP/BIOS projects.
DSP2803x_PieVect.c	DSP2803x_common\source	Creates an instance of the PIE vector table structure initialized with pointers to the ISR functions in DSP2803x_DefaultIsr.c. This instance can be copied to the PIE vector table in order to initialize it with the default ISR locations.

Table 2.17: Basic PIE Block Specific Support Files

In addition, the files in Table 2.18 are included for software prioritizing of interrupts. These files are used in place of those above when additional software prioritizing of the interrupts is required. Refer to the example and documentation in *DSP2803x_examples_ccsv4\sw_prioritized_interrupts* for more information.

File	Location	Description
DSP2803x_SWPrioritizedDefaultIsr.c	DSP2803x_common\source	Default shell interrupt service routines (ISRs). These are shell ISRs for all of the PIE interrupts. You can choose to populate one of functions or re-map your own interrupt service routine to the PIE vector table. Note: This file is not used for DSP/BIOS projects.
DSP2803x_SWPrioritizedIsrLevels.h	DSP2803x_common\include	Function prototype statements for the ISRs in DSP2803x_DefaultIsr.c. Note: This file is not used for DSP/BIOS projects.
DSP2803x_SWPrioritizedPieVect.c	DSP2803x_common\source	Creates an instance of the PIE vector table structure initialized with pointers to the default ISR functions that are included in DSP2803x_DefaultIsr.c. This instance can be copied to the PIE vector table in order to initialize it with the default ISR locations.

Table 2.18: Software Prioritized Interrupt PIE Block Specific Support Files

2.8.2.2 Peripheral Specific Files

Several peripheral specific initialization routines and support functions are included in the peripheral .c source files in the DSP2803x_common\src directory. These files are shown in Table 2.19.

File	Description
DSP2803x_GlobalPrototypes.h	Function prototypes for the peripheral specific functions included in these files.
DSP2803x_Adc.c	ADC specific functions and macros.
DSP2803x_Comp.c	Comparator specific functions and macros
DSP2803x_CpuTimers.c	CPU-Timer specific functions and macros.
DSP2803x_ECan.c	eCAN module specific functions and macros
DSP2803x_ECap.c	eCAP module specific functions and macros.
DSP2803x_EPwm.c	ePWM module specific functions and macros.
DSP2803x_EPwm_defines.h	define macros that are used for the ePWM examples
DSP2803x_EQep.c	eQEP module specific functions and macros.
DSP2803x_Gpio.c	General-purpose IO (GPIO) specific functions and macros.
DSP2803x_I2C.c	I2C specific functions and macros.
DSP2803x_I2c_defines.h	define macros that are used for the I2C examples
DSP2803x_Lin.c	LIN specific functions and macros
DSP2803x_PieCtrl.c	PIE control specific functions and macros.
DSP2803x_Sci.c	SCI specific functions and macros.
DSP2803x_Spi.c	SPI specific functions and macros.
DSP2803x_SysCtrl.c	System control (watchdog, clock, PLL etc) specific functions and macros.

Table 2.19: Included Peripheral Specific Files

Note: The specific routines are under development and may not all be available as of this release. They will be added and distributed as more examples are developed.

2.8.2.3 Utility Function Source Files

File	Description
DSP2803x_CodeStartBranch.asm	Branch to the start of code execution. This is used to re-direct code execution when booting to Flash, OTP or M0 SARAM memory. An option to disable the watchdog before the C init routine is included.
DSP2803x_DBGIER.asm	Assembly function to manipulate the DEBIER register from C.
DSP2803x_DisInt.asm	Disable interrupt and restore interrupt functions. These functions allow you to disable INTM and DBGM and then later restore their state.
DSP2803x_usDelay.asm	Assembly function to insert a delay time in microseconds. This function is cycle dependant and must be executed from zero wait-stated RAM to be accurate. Refer to DSP2803x_examples_ccsv4/adc for an example of its use.
DSP2803x_CSMPasswords.asm	Include in a project to program the code security module passwords and reserved locations.

Table 2.20: Included Utility Function Source Files

2.8.2.4 Example Linker .cmd files

Example memory linker command files are located in the DSP2803x_common\cmd directory. For getting started the basic 28035_RAM_lnk.cmd file is suggested and used by many of the included examples.

The L0 SARAM block is mirrored on these devices. For simplicity these memory maps only include one instance of these memory blocks(Table 2.21).

Memory Linker Command File Examples	Location	Description
28035_RAM_lnk.cmd	DSP2803x_common\cmd	28035 memory linker command file. Includes all of the internal SARAM blocks on a 28035 device. “RAM” linker files do not include flash or OTP blocks.
28034_RAM_lnk.cmd	DSP2803x_common\cmd	28034 SARAM memory linker command file.
28033_RAM_lnk.cmd	DSP2803x_common\cmd	28033 SARAM memory linker command file.
28032_RAM_lnk.cmd	DSP2803x_common\cmd	28032 SARAM memory linker command file.
28031_RAM_lnk.cmd	DSP2803x_common\cmd	28031 SARAM memory linker command file.
28030_RAM_lnk.cmd	DSP2803x_common\cmd	28030 SARAM memory linker command file.
28035_RAM_CLA_lnk.cmd	DSP2803x_common\cmd	28035 CLA memory linker command file. Includes CLA message RAM
28033_RAM_CLA_lnk.cmd	DSP2803x_common\cmd	28033 SARAM CLA memory linker command file.
F28035.cmd	DSP2803x_common\cmd	F28035 memory linker command file.
F28034.cmd	DSP2803x_common\cmd	F28034 memory linker command file.
F28033.cmd	DSP2803x_common\cmd	F28033 memory linker command file.
F28032.cmd	DSP2803x_common\cmd	F28032 memory linker command file.
F28031.cmd	DSP2803x_common\cmd	F28031 memory linker command file.
F28030.cmd	DSP2803x_common\cmd	F28030 memory linker command file.

Table 2.21: Included Main Linker Command Files

2.8.2.5 Example Library .lib Files

Example library files are located in the DSP2803x_common\lib directory. For this release the IQMath library is included for use in the example projects. Please refer to the *C28x IQMath Library - A Virtual Floating Point Engine (SPRC087)* for more information on IQMath and the most recent IQMath library. The SFO libraries are also included for use in the example projects. Please refer to *TMS320x2802x, 2803x HRPWM Reference Guide (SPRUGE8)* for more information on SFO library usage and the HRPWM module(see Table 2.22).

Main Liner Command File Examples	Description
SFO_TI_Build_V6.lib	Please refer to the *TMS320x2802x, 2803x HRPWM Reference Guide (SPRUGE8)* for more information on the SFO V6 library. Requires user code to update HRMSTEP register with MEP_ScaleFactor value.
SFO_TI_Build_V6b.lib	Same as v6 lib file, but now automatically updates HRMSTEP register with MEP_ScaleFactor value.
SFO_V6.h	SFO V6 header file

Table 2.22: Included Library Files

2.9 Detailed Revision History

Changes from V1.24 to V1.25

- Changes to Header Files
  1. **DSP2803x_Sci.h -** SCIRXST_BITS structure was fixed by changing bits 8 to 15 as reserver and SCPRI_BITS with 3 reserved bits in the beginning (0 to 2) and 11 trailing reserved bits (5 to 15).
  2. **DSP2803x_SysCtrl.h -** Added JTAGDEBUG[JTAGDIS] register and bit to SysCtrlRegs and replaced GPIOINENCLK bit from PCLKCR3 register with reserved
  3. **DSP2803x_Comp.h -** Added DAC Control Bits and register structure
  4. **DSP2803x_EPwm.h -** Changed reserved bit in HRPCTL_BITS to PWMSYNCSEL
  5. **New Header File -** common/include/DSP2803x_HRCap.h
- Changes to Common Files
  1. **DSP2803x_SysCtrl.c, DSP2803x_usDelay.asm, DSP2803x_Adc.c -** Added warning on copying section “ramfuncs” from flash to RAM prior to calling InitSysCtrl() or InitAdc()
  2. **DSP2803x_SWPrioritizedPieVect.c -** Replaced undefined ISR routines , SCIRXINTB_ISR and SCITXINTB_ISR, with LIN0INTA_ISR and LIN1INTA_ISR respectively.
  3. **Added New Module Files -**
     - HCCal_Type0_V1.lib
     - headers/source/DSP2803x_HRCap.c
- Changes to Example Files
  1. **Updated comments in the following files -**
     - Example_2803xAdcTempSensor.c
     - Example_2803xClaAdcFir.c
     - Example_2803xClaAdcFirFlash.c
     - Example_2803xECanBack2Back.c
     - Example_2803xECap_apwm.c
     - Example_2803xEPwmDCEventTrip.c
     - Example_2803xEPwmDCEventTripComp.c
     - Example_2803xEPwmDeadBand.c
     - Example_2803xEPwmTripZone.c
     - Example_2803xGpioSetup.c
     - Example_2803xI2C_eeprom.c

- Example_2803xSci_Echoback.c
- Example_2803xScia_FFDLB.c
- Example_2803xSci_FFDLB_int.c
- Example_2803xSpi_FFDLB.c
- Example_2803xSpi_FFDLB_int.c
- Example_2803xSWPrioritizedInterrupts.c

2. **Example_2803xSWPrioritizedInterrupts.c -** Fixed the following issues
   - CASE 12 previously failed to compile.
   - Wrong interrupt group cleared in cases 7 and 8
3. **Example_2803xGpioSetup.c -** Fixed:
   - Corrected GPIO setup (from GPIO32 to GPIO34)
   - CANTXA and CANRXA pins flipped
   - Incorrectly setup GPIO7 for ECAP2 (No ECAP2 on 28035)
4. **Example_2803xECap_apwm.c -** Corrected example to vary the PWM period
5. **Example_2803xClaAdcFirFlash.c -** Corrected code to use ADC channel 2 for SOC 1 instead of channel 7
6. **Example_2806xSpi_FFDLB.c -** Removed delay_loop()
7. **HRPWM -** Replaced prototype void HRPWMx_Config(int); with void HRPWMx_Config(Uint16) in the following files:
   - Example_2803xHRPWM.c, Example_2803xHRPWM_Duty_SFO_V6.c
   - Example_2803xHRPWM_MultiCh_PrdUpDown_SFO_V6.c
   - Example_2803xHRPWM_PrdUp_SFO_V6.c
   - Example_2803xHRPWM_PrdUpDown_SFO_V6.c
   - Example_2803xHRPWM_slider.c
8. **Added New Examples -**
   - Example_2803xHRCap_Capture_HRPwm.c
   - Example_2803xHRCap_Capture_Pwm.c

Changes from V1.23 to V1.24

- Changes to Header Files
  1. **DSP2803x_EPwm.h -** Added 6 reserved words at end of data structure to facilitate use of pointers to EPWM_REGS structures in user software. Additionally, the TZDCSEL_BITS structure was fixed by added 4 reserved bits in positions 15:12.
  2. **DSP2803x_Headers_BIOS.cmd -** fixed comments at top of file.
  3. **DSP2803x_Adc.h -** Updated comments for COMPHYSTCTL register.
- Changes to Common Files
  1. **All 2803x device gel files -** Modified Device_Cal function to set address 0x714B with ADC trim value for Rev. A silicon and beyond (still sets address 0x714E for Rev. 0 silicon). Additionally, CCSv3.3 versions of these gel files have been removed from controlSUITE. In OnReset(), delete check for emu_boot_set. EMU_BOOT_SARAM and EMU_BOOT_FLASH are still in OnReset(), but commented out. Updated comments. Calibration regions have been updated on MemoryMap as readable regions.
  2. **DSP2803x_I2C.c -** Updated comments.
  3. **DSP2803x_usDelay.asm -** Change reference to “DSP2803x_Device.h” to “DSP2803x_Examples.h”
  4. **DSP2803x_OscComp.c- Added code to prevent over-/under -**saturation of FINETRIM.
- Changes to Example Files

1. **Example_2803xExternalInterrupt.c -** Updated #define for DELAY to be dependant on CPU_RATE.
2. **Example_2803xEpwmTripZoneComp.c -** Fixed comments referencing COMP2, and added delay after enabling ADC BG REF to allow BG REF to settle. Additionally set ePWM to 50% duty cycle instead of 100% duty cycle.
3. **Example_2803xEcanBack2Back.c -** Fixed example to read from receive mailboxes instead of transmit mailboxes.

Changes from V1.22 to V1.23

- Changes to Header Files
  1. **DSP2803x_I2c.h -** Added I2CEMDR register to structure
  2. **DSP2803x_Device.h -** Added common timer period definitions in milliseconds.
  3. **DSP2803x_Adc.h -** Added COMPHYSTCTL register and modified ADCREFTRIM register to support Rev. A silicon
- Changes to Common Files
  1. **All 2803x device gel files -** Modified Device_Cal function to set address 0x714B with ADC trim value for Rev. A silicon and beyond (still sets address 0x714E for Rev. 0 silicon). Additionally, CCSv3.3 versions of these gel files have been removed from controlSUITE.
  2. **DSP2803x_EPwm_defines.h -** Fixed dead-band control definitions (DBA_ENABLE and DBB_ENABLE have been switched).
- Changes to Example Files
  1. **Example_2803xOscComp.c -** Oscillator compensation on COMP2 has been uncommented.
  2. **Example_2803xAdcTempSensor.c -** Updated to consider ADC first sample errata for Rev. 0 silicon.
  3. **Example_2803xAdcSoc.c -** Updated to consider ADC first sample errata for Rev. 0 silicon.
  4. **Example_2803xEPwmRealTimeInt.c -** Added new example to demonstrate real time interrupt capability with the ePWM module.
  5. **Example_2803xHRPWM_MultiCh_PrdUpDown_SFO_V6.c -** Added new example to demonstrate high resolution period/frequency movement on multiple synchronized ePWm modules.
  6. **Example_2803xHRPWM_PrdUpDown_SFO_V6.c -** Modified the original high-resolution period/frequency movement example to demonstrate high-resolution period/frequency using the SFO_V6 library on a single PWM channel.
  7. **Example_2803xClaAdc.c, CLA.asm -** Fixed comments throughout example and added MDEBUGSTOP to allow single step debugging in CLA.asm file.

Changes from V1.21 to V1.22

- Changes to Example Files
  1. This update makes numerous improvements to the Code Composer Studio 4 projects.
     * Added the search path to the controlSUITE install of the IQmath library to each project.
     * Removed the local copy of the IQmath library and header file.
     * Removed the “release” configuration of each project as it has not been properly setup and tested.
     * Reviewed the compiler and linker switches and updated them as needed.
     * Linked this document to each project so it can be easily opened. Right click on the document and select “open with” and “system editor”.

Changes from V1.20 to V1.21

- Changes to Header Files
  1. **DSP2803x_Adc.h -** Fixed error in structure, under ADCCTL2, changed rsvd1 to 2 words wide instead of 3 words wide
- Changes to Common Files
  1. **DSP2803x_Cla_defines.h -** Increase repeated NOP's from 2 to 3.
  2. **All gel files -** update OnFileLoaded() function to call Device_Cal only if symbols are not being loaded.

Changes from V1.10 to V1.20

- Changes to Header Files
  1. **DSP2803x_SysCtrl.h -** Added SysPwrCtrlRegs structure with BORCFG register.
  2. **DSP2803x_DevEmu.h -** Removed BORCFG register (does not belong in this space).
  3. **DSP2803x_Headers_BIOS.cmd/DSP2803x_Headers_nonBIOS.cmd -** Added SysPwrCtrlRegs to 0x985-0x987, and reduced DevEmuRegs to 0x880 - 0x984.
  4. **DSP2803x_GlobalVariableDefs.c -** Added SysPwrCtrlRegs declaration.
  5. **DSP2803x_Adc.c -** Added changes to be implemented in Rev. A silicon (documented in ADC reference guide. Not implemented in Rev. 0 silicon). Added ONESHOT bit to SOCPRICTL register and added ADCCTL2 register. Also added API functions for re-calibrating ADC offset.
  6. **DSP2803x_Device.h -** Removed RSH package references.
- Changes to Common Files
  1. **DSP2803x_Cla_defines.h -** Increase repeated NOP's from 2 to 3.
  2. **F28031.cmd, F28030.cmd, 28031_RAM_lnk.cmd, 28030_RAM_lnk.cmd -** Added these files for the new 28031 and 28030 devices.
  3. **28031.gel and 28030.gel -** Added gel files for 28031 and 28030 devices.
  4. **All 2803x gel files -** Change comment references to "2802x" to "2803x". Correct MemoryFill function so that it properly fills L3 memory with ESTOP. Fixed Bypass() function so that PLLCR=0 prior to DIVSEL = divide by 1. In CCSv3.3 version of gel files, in Watch Emulation Registers function, changed PARTID address to 0x3d7e80. Added BORCFG register (address 0x985). In CCSv4 version of gel files - removed note about Watch FPU registers in OnPreFileLoaded() function, removed "Peripherals.gel" reference, and fixed Gel_Toolbar5() function for CCSv4 use.
  5. **DSP2803x_Examples.h -** Removed RSH package references.
  6. **DSP2803x_OscComp.c -** Adds functions required for internal oscillator frequency compensation over temperature. The file is added to the /DSP2803x_common/source/ directory. Also added document in /doc directory explaining various functions defined in this file.
  7. **DSP2803x_TempSensorConv.c -** Adds functions required for ADC temp sensor to convert digital ADC samples to Kelvin and Celsius temperature value. The file is added to the /DSP2803x_common/source/ directory.
  8. **DSP2803x_GlobalPrototypes.h -** Added function prototypes from DSP2803x_Adc.c, DSP2803x_OscComp.c and DSP2803x_TempSensorConv.c files.
  9. **DSP2803x_SysCtrl.c -** Added "EALLOW" for XtalOscSel and ExtOscSel functions so the bit settings take effect when calling these functions.
  10. **DSP2803x_Comp.c -** Fixed typo in comments referring to SDAA and SCLA operation - changed this to CMP1OUT and CMP2OUT.
  11. **DSP2803x_Epwm.c -** Fixed typo in comments such that GPIO3 refers to EPWM2B and not EPWM3B.

12. **SFO_TI_Build_V6.lib -** Original SFO_TI_Build_V6.lib did not automatically write the MEP_ScaleFactor into the HRMSTEP register. SFO_TI_Build_V6b.lib now updates the HRMSTEP register with the MEP_ScaleFactor value automatically. Additionally, added an errata document to the /doc directory explaining the difference.
13. **DSP2803x_ECan.c -** Removed ambiguous statement in comment concerning shadow register structure.
14. **28032_RAM_CLA_lnk.cmd and 28034_RAM_CLA_lnk.cmd -** Removed these files - there is no CLA module on these devices.

- Changes to Example Files
  1. **Example_2803xHRPWM_Duty_SFO_V6.c -** Changed temp to 32 -bit unsigned integer. Corrected equations so rounding is correct when AUTOCONV=0. Changed *ePWM array of structs to include only 4 ePWM's.
  2. **Example_2803xHRPWM_PrdUp_SFO_V6.c -** Changed *ePWM array of structs to include only 4 ePWM's.
  3. **Example_2803xHRPWM_PrdUpDown -** Changed *ePWM array of structs to include only 4 ePWM's.
  4. **All CCSv4 example .cdtbuild Files -** Replaced any hard coded references to "C:/tidcs/c28/DSP2803x/<version>" for OBJ and ASM directories and replaced with "$INSTALLROOT_2803X_<version>" macro. This did not affect CCSv4 project build in previous version, but improves portability.
  5. **Example_28030_Flash/Example_28031_Flash -** Added PJT and GEL files in the CCSv3.3 DSP2803x_examples/flash/ directory.

Changes from V1.01 to V1.10

- Changes to Header Files
  1. **DSP2803x_Gpio.h -**Removed GPBPUD register structure from GPBDAT register definition.
  2. **DSP2803x_DevEmu.h -** Added BORCFG register.
- Changes to Common Files
  1. **SFO_TI_Build_V6.lib and SFO_V6.h -** SFO library updated to generate error code of "2" when MEP_ScaleFactor>255 (previously returned "2" for MEP_ScaleFactor>254). Additionally, the library now only updates the HRMSTEP register used for auto-conversion with the calibrated MEP_ScaleFactor if MEP_ScaleFactor<=255. Otherwise it will use the last "good" value written to HRMSTEP for auto-conversions (previously, if the MEP_ScaleFactor>255, auto-conversion could not be used).
  2. **F28035.gel and F28033.gel -** Adjusted certain CLA registers which are 32-bits instead of 16-bits.
  3. **All device gel files -** Added 0xD00-0xE00 to Page 0 memory map (specifically to allow CCStudio to access these memories when using DSP/BIOS)
  4. **CCSv4 gel files -** Added ccsv4 directory in /gel directory for CCSv4-specific device gel files (GEL_WatchAdd() functions removed).
  5. **DSP2802x_CpuTimers.c -** Corrected note that DSP/BIOS reserved CpuTimer2 only and user must comment out CpuTimer2 code when using DSP/BIOS. CpuTimer0 and 1 have no such restriction.
- Changes to Example Files
  1. **All PJT Files -** Removed the line: Tool="DspBiosBuilder" from all example PJT files for easy migration path to CCSv4 Microcontroller-only (code-size limited) version users.
  2. **Added Example_2803xClaAdcFir.c and flash version of same example -** Added new CLA ADC examples.

3. **Updated CLA.asm file for cla_adc example-** Updated to better match CLA_FIR.asm in new CLA examples (i.e. Cla1T1End instead of ClaT1End).
4. **Example_2803xLPMHaltWake.c -** Updated description comments for wakeup.
5. **Example_2803xLEDBlink -** Removed FPU build options.
6. **Example_2803xSpi_FFDLB_int.c -** Changed FIFO level to 2 (while receiving/interrupting on 2 words, there are 2 remaining FIFO spaces for continuous receiving).
7. **Example_2803xSci_FFDLB_int.c -** Changed FIFO level to 2 (while receiving/interrupting on 2 words, there are 2 remaining FIFO spaces for continuous receiving).
8. **Added DSP2803x_examples_ccsv4 directories -**Added directories for CCSv4.x projects. The example projects in these directories are identical to those found in the normal CCSv3.x DSP2803x_examples directory with the exception that the examples now support the Code Composer Studio v4.x project folder format instead of the Code Composer Studio v3.x PJT files. The example gel files have also been removed for the CCSv4 example projects because the gel file functions used in the example gels are no longer supported.

Changes from V1.00 to V1.01

- Changes to Header Files
  1. **DSP2803x_Lin.h -** Corrected comments, removed bits that do not exist in design, and renamed other bits per function.
- Changes to Common Files
  1. **DSP2803x_PieVect.c -** Corrected comments pertaining to ADCINT1 and ADCINT2
  2. **DSP2803x_Lina.c -** Added initialization function for LIN-A.
  3. **DSP2803x_GlobalPrototypes.h -** Added prototypes for LIN from DSP2803x_Lina.c
- Changes to Example Files
  1. **Example_2803xAdcTempSensor -** Added one ADC temperature sensor example in adc_temp_sensor directory.
  2. **Example_2803xLina_EXALB -** Added LIN-A external analog loopback example in lina_external_loopback directory.
  3. **Example_2803xLin_Sci_Echoback -** Added LIN-A SCI echoback example in lina_sci_echoback directory.
  4. **Example_2803xLinSci_DLB_int -** Changed SCIGCR1.bit.CLOCK to SCIGCR1.bit.CLK_MASTER to match new bit name defined in DSP2803x_Lin.h

V1.00

- This version is the first release (packaged with development tools and customer trainings) of the DSP2803x header files and examples.

3 Getting Started with Project Creation and Debugging

3.1 Introduction

This chapter aims to give you, the user, a step by step guide on how to create and debug projects from scratch. This guide will focus on the user of a Piccolo controlCARD, but these same ideas should apply to other boards with minimal translation.

3.2 Project Creation

A typical Piccolo application consists of a single CCS project with multiple source files: C and ASM files for the C28 and ASM files for the CLA.

Project Creation

1. From the main CCS window select File -> New -> CCS Project. Name your project and choose a location for it to reside. Click Next.

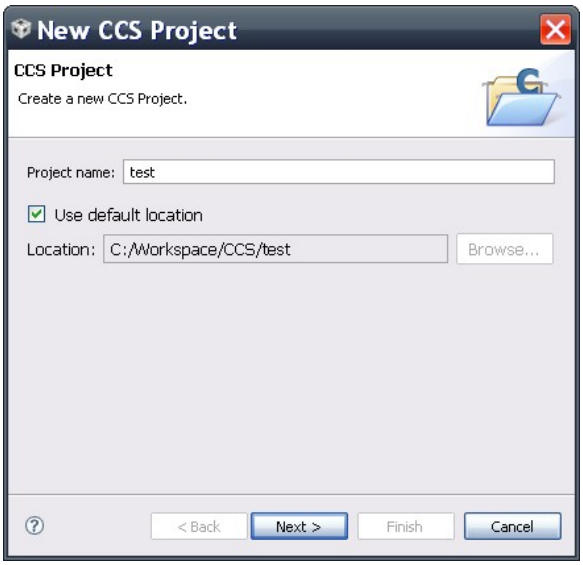

Figure 3.1: Creating a new project

2. Select C2000 as the project type on this next dialog box and click Next.

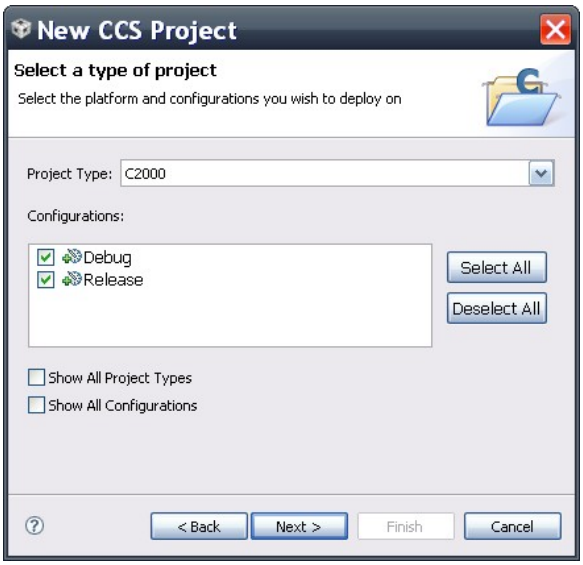

Figure 3.2: Project type dialog box

3. Define any inter-project dependencies. If this is your first Piccolo project you likely will have none, so just click Next.

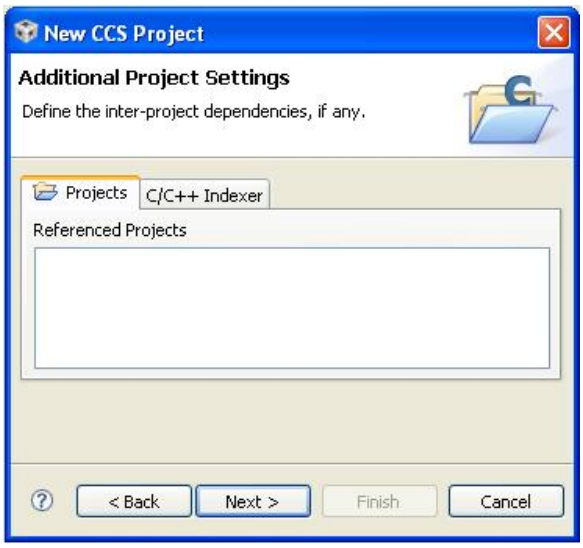

Figure 3.3: Inter-project dependencies dialog box

4. This next window is by far the most important as far as project setup goes. Ensure that you window matches the settings below except for perhaps the device variant. After you are satisfied with these settings select Finish and your project will be created.

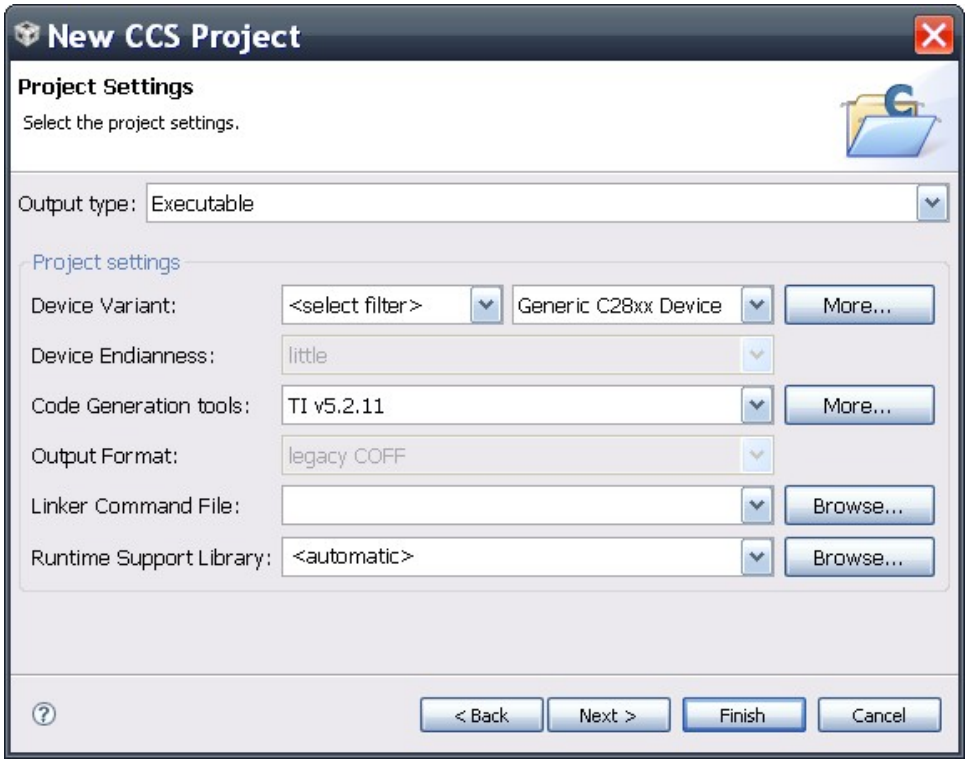

Figure 3.4: Project configuration dialog box

5. Before we can successfully build a project we need to setup some build specific settings. Right click on your project and select Build Properties. In the Tool Settings tab look for and select the Include Options. Click on the add directory icon to add a directory to the search path. Click the File System button to browse to the `DSP2803x_common\include` folder of your controlSUITE installation (typically `C:\TI\controlSUITE\device_support\f2803x\VERSION\DSP2803x_common\include`). Click ok to add this path, and repeat this same process to add the `DSP2803x_headers\include` directory.
   While you have this window open select the Symbol Management options under C2000 Linker. Specify the program entry point to be `code_start`. Select ok to close out of the Build Properties.

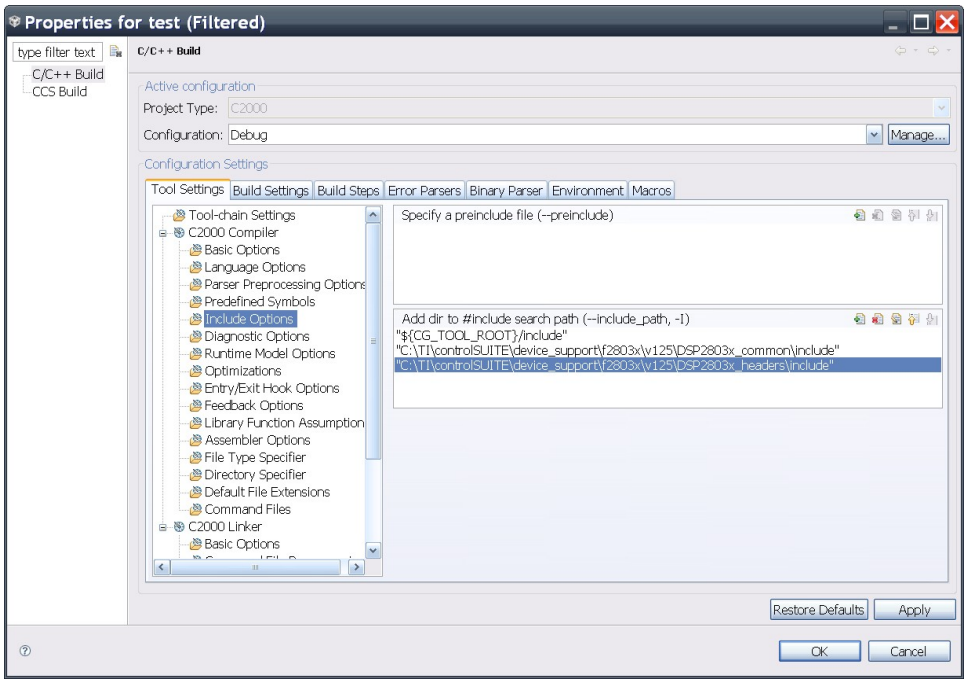

Figure 3.5: Include path setup

6. Next we need to link in a few files which are used by the header files. To do this right click on your project in the workspace and select Link Files... Navigate to the `DSP2803x_headers\source` directory, and select DSP2803x_GlobalVariableDefs.c. Link in the following files as well:
   - `DSP2803x_headers\cmd\DSP2803x_Header_nonBIOS.cmd`
   - `DSP2803x_common\source\DSP2803x_CodeStartBranch.asm`
   - `DSP2803x_common\source\DSP2803x_usDelay.asm`
   - `DSP2803x_common\cmd\28035_RAM_CLA_lnk.cmd` or another appropriate linker command file

   At this point your project workspace should look like the following:

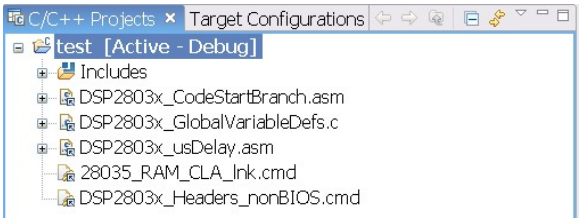

Figure 3.6: Linking files to project

   In this step we linked a file to the project which only created a symbolic link in the project to the actual file in the hard drive. This means that if you modify a linked file in CCS you are modifying the original file in controlSUITE. We won't be modifying the linker command file or header files, so this is ok.
7. Create a new file by right clicking on the project and selecting New -> File. Name this file main.c and copy the following code into it:

```
#include "DSP28x_Project.h"

void main(void)
{
    //
    // Disable Protection
    //
    EALLOW;

    //
    // Make Port B GPIOs outputs
    //
    GpioCtrlRegs.GPADIR.all = 0x0000FF00;

    while(1)
    {
        //
        // Toggle GPIOs 8-15
        //
        GpioDataRegs.GPADAT.all = 0x0000FF00;
        DELAY_US(100);
        GpioDataRegs.GPADAT.all = 0x00000000;
        DELAY_US(100);
    }

}
```

8. Save main.c and then build the project by right clicking on it and selecting Build Project. You have just built your first Piccolo project from scratch.

3.3 Debugging Applications

1. Ensure CCS version 4 is installed and up to date. You should have C2000 Code Generation Tools verion 5.2.6 or later.
2. Connect a USB cable from the computer to the USB port on the base board. Windows will enumerate and try to install drivers. As long as CCS is installed, Windows should automatically find and install drivers for the emulator.
3. Apply power either via USB or the 5V DC jack on the docking station. If you wish to use the onboard XDS100v1 emulator you will need to connect the USB cable. Alternatively you could connect an external JTAG emulator using the available header pins on the base board.
4. Create a new target configuration. Click File -> New -> Target Configuration File and name the file appropriately (i.e. XDS100v1_Piccolo_ControlCARD.ccxml). Select the emulator you intend to use (XDS100v1) from the drop down list, and then select the device variant present on your board (Piccolo controlCARDs have an Experimenters Kit - Piccolo F28035). Save the target configuration and close the window.

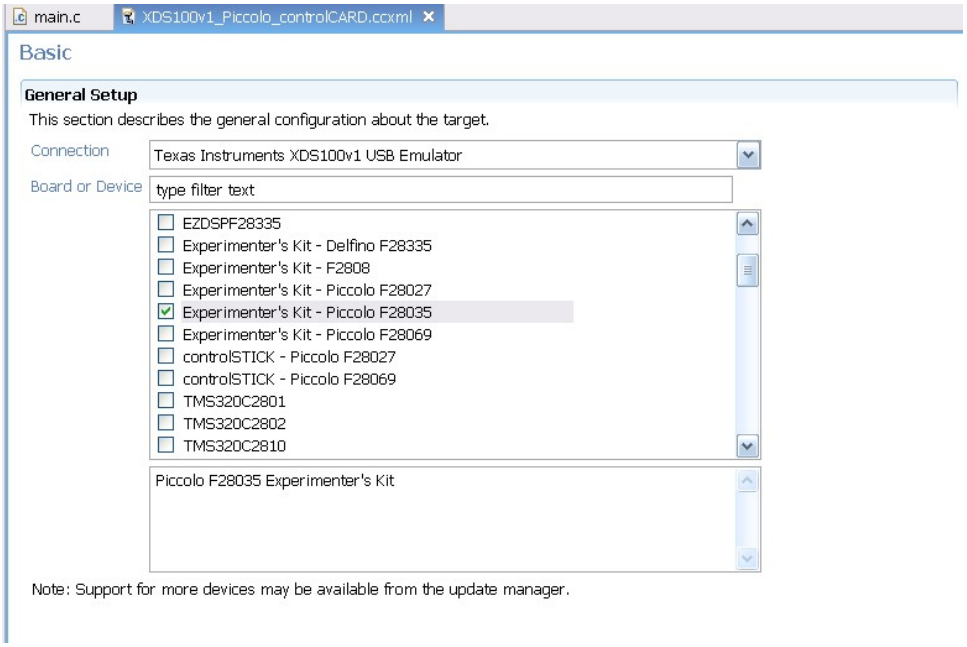

Figure 3.7: Piccolo Card Target Configuration Setup

5. Import the desired example projects (or skip this step if you are using projects you created in the Project Creation section). Click File -> Import, and in the CCS folder select Existing CCS/CCE Eclipse Projects before clicking Next. With the "Select search-directory" radio button checked, browse to the root of your controlSUITE installation. Device specific software as well as examples are stored in the `device_support/device_variant` folders. Navigate to the `f2803x` directory, and then to the `DSP2803x_examples_ccsv4` directory. Click OK and CCS will parse all of the projects in this directory. Import any projects you wish to run into the workspace. **Do not select "Copy projects into workspace".** These projects use relative paths to link to external resouces, so taking them out of controlSUITE will break the project.

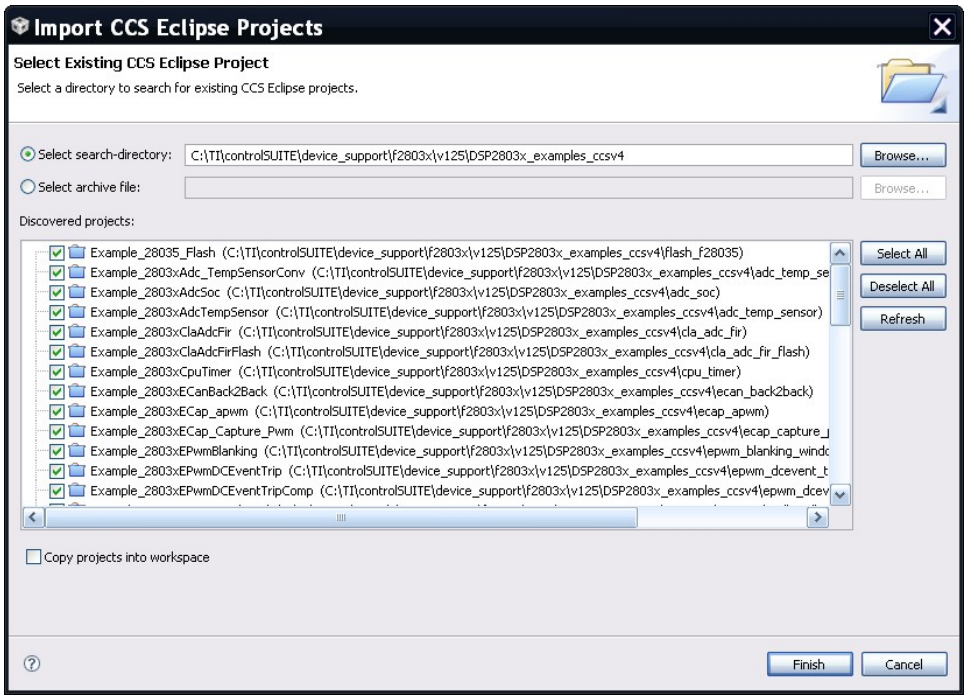

Figure 3.8: Importing Piccolo Projects

6. Build each of the example projects. Right click on each project title and select build project.

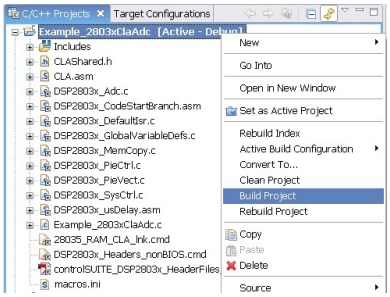

Figure 3.9: Building Piccolo Projects

7. Launch the previously created target configuration. Click View -> Target Configurations. In the window that opens, find the desired target configuration, right click on it and select "Launch Target Configuration".

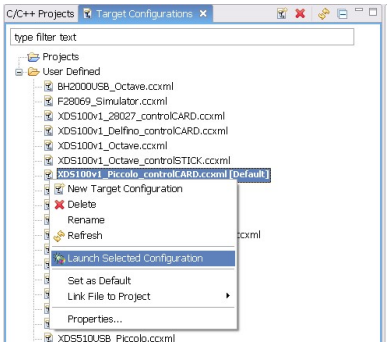

Figure 3.10: Launching a CCS Target Configuration

8. Connect to the device. Right click on each core in the debug window and select "Connect Target. This will connect CCS to the device and will allow you to load code and debug applications.

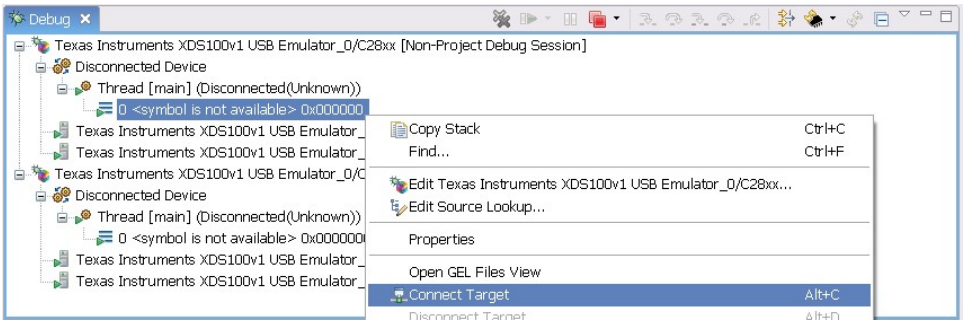

Figure 3.11: Connecting to a Target

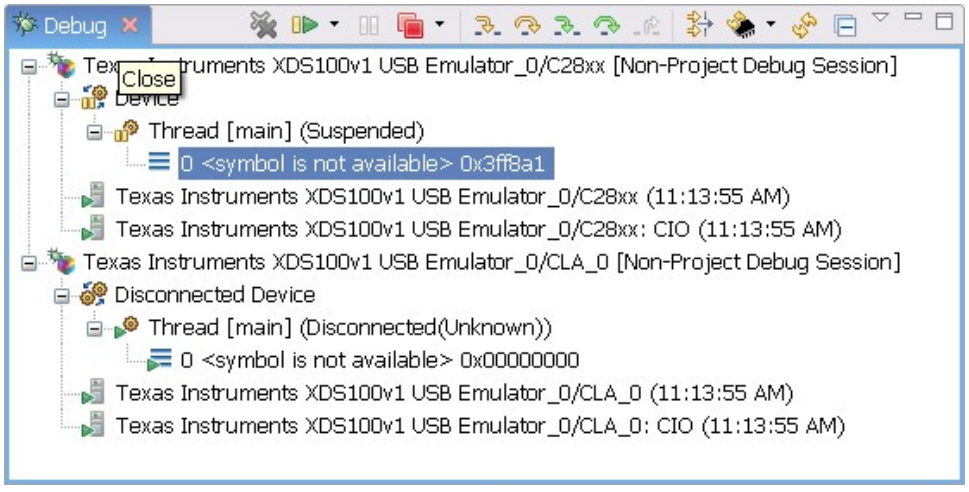

Figure 3.12: After connection to both cores

9. Load code onto the device. Select the C28x session in the debug window and then click Target -> Load Program. A dialog box is displayed which will allow you to select a program

to load.

10. At this point the C28 should have code loaded and be halted at main. From this point, users should be able to debug code. Please keep in mind that any action you take in CCS only has an effect on the session you currently have selected in the debug window. For instance if the C28 is selected, the register view will display the registers of the C28 system. The opposite would be true if the CLA were selected.

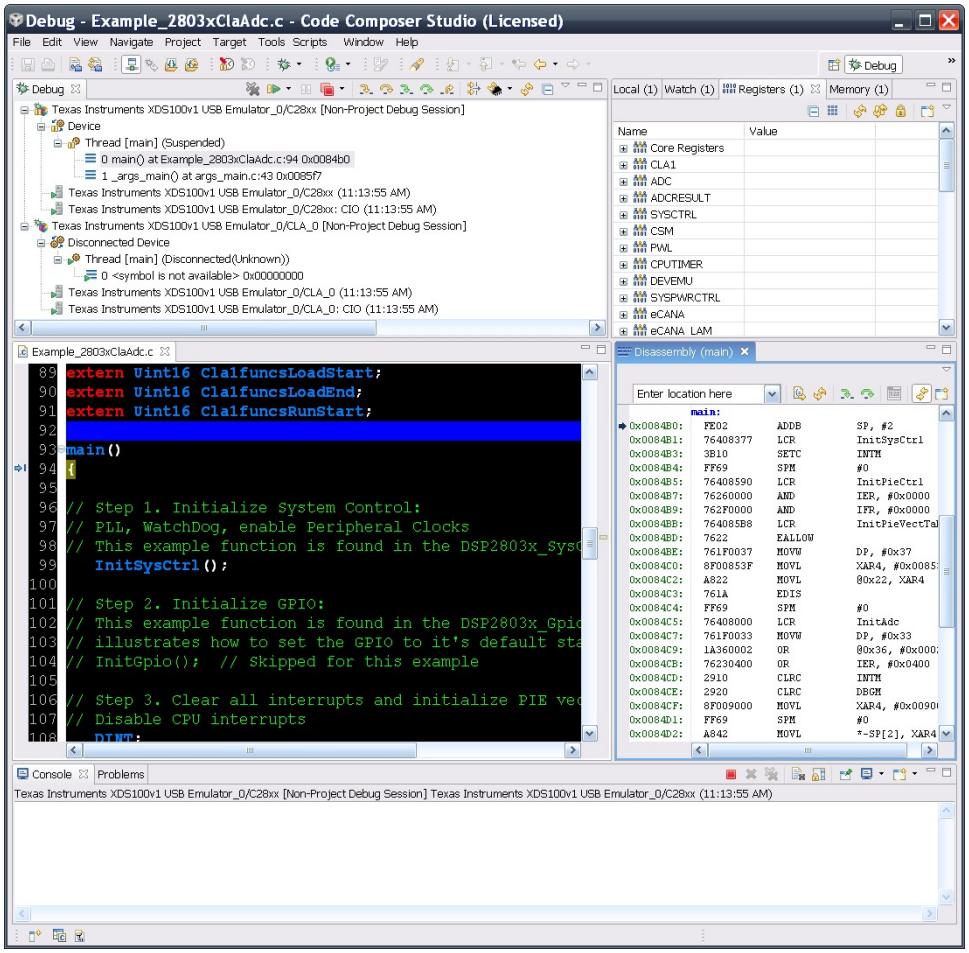

Figure 3.13: Projects loaded on the C28

3.4 Troubleshooting

There are a number of things that can cause the user trouble while bringing up a debug session the first time. This section will try to provide solutions to the most common problems encountered with the Concerto devices.

"I get a managed make error when I import the example projects"

This occurs when one imports a project for which he or she doesn't have the code generation tools for. Please ensure that you have at least version 5.2.6 of the C2000 Code Generation Tools.

"I cannot build the example projects"

This is caused by linked resources not being where the project expects them to be. For instance, if you imported the projects and selected "Copy projects to workspace", the projects would no longer build because the files they refernce aren't a part of your workspace. Always build and run the examples directly in the controlSUITE tree.

"I cannot connect to the target"

This is most often times caused by either a bad target configuration, or simply the emulator being physically disconnected. If you are unable to connect to a target check the following things:

1. Ensure the target configuration is correct for the device you have.
2. Ensure the emulator is plugged in to both the computer and the device to be debugged.
3. Ensure that the target device is powered.

"I cannot load code"

This is typically caused by an error in the GEL script or improperly linked code. GEL files shipped in controlSUITE are tested and should work without modification with Piccolo devices, but advanced users may potentially alter GEL files depending on their overall system configuration. If you are having trouble loading code, check the linker command files and maps to ensure that they match the deviceŠs memory map.

4 Piccolo F2803x Example Applications

These example applications show the user how to make use of various peripherals present on the Piccolo device. They are intended for demonstration purposes only and a good starting point for building new applications.

Notes

- All examples require the DSP2803x header files
- All examples set up the PLL in x12/2 mode which gives a system clock of 60MHz. This is the default setting assuming the input clock is derived from the 10MHz internal clock.
- Some examples like those related to HRPWM require the use of an external scope to see the results, while other examples may require external connections between headers on the baseboard (e.g. adc_soc). Each example will describe the setup procedure that is required to properly execute it.
- As supplied, all projects are configured for "boot to SARAM" operation unless specified otherwise in the example description. The 2803x Boot Mode table is shown below.
  - While an emulator is connected to your device, the TRSTn pin = 1, which sets the device into EMU_BOOT boot mode. In this mode, the peripheral boot modes are shown in the table below.
  - Write EMU_KEY to 0xD00 and EMU_BMODE to 0xD01 via the debugger with the values from the table
  - Build/Load project, reset the device, and run the example

Boot Mode	EMU_KEY	EMU_BMODE
	(0xD00)	(0xD01)
Wait	!=0x55AA	X
I/O	0x55AA	0x0000
SCI	0x55AA	0x0001
Wait	0x55AA	0x0002
Get_Mode	0x55AA	0x0003
SPI	0x55AA	0x0004
I2C	0x55AA	0x0005
OTP	0x55AA	0x0006
eCANA	0x55AA	0x0007
SARAM	0x55AA	0x000A (*Boot to SARAM*)
Flash	0x55AA	0x000B
Wait	0x55AA	Other

Table 4.1: Boot Modes for Piccolo 2803x

We have provided scripts to automate setting up watch variables and associated graphs called 'SetupDebugEnv.js' in several example folders. Once you have established a connection to the target device in debug mode go to View->Scripting Console. Within the console click the Open Command file icon in the far right corner of the console window and select the javascript file.

All of these examples reside in the `device_support/f2803x/<version>/DSP2803x_examples_ccsv` subdirectory of the ControlSUITE package.

4.1 ADC Start of Conversion (adc_soc)

This ADC example uses ePWM1 to generate a periodic ADC SOC - ADCINT1. Two channels are converted, ADCINA4 and ADCINA2.

Watch Variables

- Voltage1[10] - Last 10 ADCRESULT0 values
- Voltage2[10] - Last 10 ADCRESULT1 values
- ConversionCount - Current result number 0-9
- LoopCount - Idle loop counter

4.2 ADC Temperature Sensor (adc_temp_sensor)

In this example the ePWM1 is set up to generate a periodic ADC SOC interrupt - ADCINT1. One channel is converted - ADCINA5, which is internally connected to the temperature sensor.

Watch Variables

- TempSensorVoltage[10] - Last 10 ADCRESULT0 values
- ConversionCount - Current result number 0-9
- LoopCount - Idle loop counter

4.3 ADC Temperature Sensor Conversion (adc_temp_sensor_conv)

This example shows how to convert a raw ADC temperature sensor reading into deg. C and deg. K. Internal temperature is sampled continuously through ADCINA5. The coefficients required to compensate for temperature offset are read from TI OTP.

Note:

THIS EXAMPLE USES VARIABLES STORED IN OTP DURING FACTORY TEST. THESE OTP LOCATIONS ,0x3D7E90 to 0x3D7EA4, MAY NOT BE POPULATED. ENSURE THAT THESE MEMORY LOCATIONS IN TI OTP ARE POPULATED WITH VALUES DIFFERENT FROM 0XFFFF

Watch Variables

- temp
- degC
- degK

4.4 CLA ADC (cla_adc)

In this example ePWM1 is setup to generate a periodic ADC SOC. Channel ADCINA2 is converted. When the ADC begins conversion, it will assert ADCINT2 which will start CLA task 2.

Cla Task2 logs 20 ADCRESULT1 values in a circular buffer. When Task2 completes an interrupt to the CPU clears the ADCINT2 flag.

Watch Variables

- VoltageCLA - Last 20 ADCRESULT1 values
- ConversionCount - Current result number
- LoopCount - Idle loop counter

4.5 CLA ADC FIR (cla_adc_fir)

In this example ePWM1 is setup to generate a periodic ADC SOC. One channel is converted: ADCINA2 and the results are placed in the ADC RESULT1 register. When the ADC sample window ends and begins conversion, it will assert ADCINT7. The CLA responds to ADCINT7 and executes CLA Task 7. CLA Task7 is an FIR filter. The output from the filter is placed in VoltFilt. When Task 7 completes, it fires the CLA1_INT7 interrupt to the main CPU. The main CPU will clear the ADCINT flag, copy the CLA output to a buffer and record the raw ADCRESULT1 value for comparison After ADC_BUF_LEN samples are collected, the code will halt on an embedded software breakpoint. ePWM3 generates a square wave, which can be connected to the ADC for testing.

External Connections

- connect a jumper between to ADCINA2 and EPWM3A (GPIO4)

Watch Variables

- Uint16 AdcBuf[ADC_BUF_LEN] - Buffer of raw ADC RESULT1 values
- Uint16 AdcFiltBuf[ADC_BUF_LEN] - Buffer of CLA FIR filter outputs
- Uint16 SampleCount - Current sample number

4.6 CLA ADC FIR FLASH (cla_adc_fir_flash)

This example is the same as the cla_adc_fir example, except code is loaded into flash. Time critical code and CLA code are copied to RAM for execution. In this example ePWM1 is setup to generate a periodic ADC SOC. One channel is converted: ADCINA2 and the results are placed in the ADC RESULT1 register. When the ADC sample window ends and begins conversion, it will assert ADCINT7. The CLA responds to ADCINT7 and executes CLA Task 7. CLA Task7 is an FIR filter. The output from the filter is placed in VoltFilt. When Task 7 completes, it fires the CLA1_INT7 interrupt to the main CPU. The main CPU will clear the ADCINT flag, copy the CLA output to a buffer and record the raw ADCRESULT1 value for comparison After ADC_BUF_LEN samples are collected, the code will halt on an embedded software breakpoint. ePWM3 generates a square wave, which can be connected to the ADC for testing.

External Connections

- connect a jumper between to ADCINA2 and EPWM3A (GPIO4)

Watch Variables

- Uint16 AdcBuf[ADC_BUF_LEN] - Buffer of raw ADC RESULT1 values
- Uint16 AdcFiltBuf[ADC_BUF_LEN] - Buffer of CLA FIR filter outputs
- Uint16 SampleCount - Current sample number

4.7 Cpu Timer (cpu_timer)

This example configures CPU Timer0, 1, and 2 and increments a counter each time the timer asserts an interrupt.

Watch Variables

- CpuTimer0.InterruptCount
- CpuTimer1.InterruptCount
- CpuTimer2.InterruptCount

4.8 eCAN back to back (ecan_back2back)

This example tests eCAN by transmitting data back-to-back at high speed without stopping. The received data is verified. Any error is flagged. MBX0 transmits to MBX16, MBX1 transmits to MBX17 and so on.... This program illustrates the use of self-test mode

Watch Variables

- PassCount
- ErrorCount
- MessageReceivedCount

4.9 eCAP APWM (ecap_epwm)

This program sets up the eCAP pins in the APWM mode. eCAP1 will come out on the GPIO19 pin. This pin is configured to vary between 3 Hz and 6 Hz using the shadow registers to load the next period/compare values

4.10 eCAP capture PWM (ecap_capture_pwm)

This example configures ePWM3A for:

- Up count
- Period starts at 2 and goes up to 1000
- Toggle output on PRD

eCAP1 is configured to capture the time between rising and falling edge of the ePWM3A output.

External Connections

- eCAP1 is on GPIO19
- ePWM3A is on GPIO4
- Connect GPIO4 to GPIO19.

Watch Variables

- **ECap1PassCount** , Successful captures
- **ECap1IntCount** , Interrupt counts

4.11 ePWM Blanking Window (epwm_blanking_window)

This example configures ePWM1 and ePWM2

- ePWM1: DCAEVT1 forces EPWM1A high, a blanking window is used EPWM1B toggles on zero as a reference.
- ePWM2: DCAEVT1 forces EPWM2A high, no blanking window is used EPWM2B toggles on zero as a reference. ePWM1A is set to normally stay low. DCAEVT1 is true when TZ1 is low and TZ2 is high. When an event is true (DCAEVT1) EPWM1A is configured to be forced high. A blanking window is applied to keep the event from taking effect around the zero point. In other words, when the event is taken, EPWM1A will be forced high if there is no event, EPWM1A will remain low. Notice the blanking window keeps the event from forcing EPWM1A high around the zero point. ePWM2 is configured the same way as ePWM1 except no blanking window is applied.

Initially tie TZ1 (GPIO12) and TZ2 (GPIO13) high. During the test, monitor ePWM1 or ePWM2 outputs on a scope. Create DCAEVT1 by pulling TZ1 low and TZ2 high to see the effect.

External Connections

- ePWM1A is on GPIO0
- ePWM1B is on GPIO1
- ePWM2A is on GPIO2
- ePWM2B is on GPIO3
- TZ1 is on GPIO12
- TZ2 is on GPIO13

4.12 ePWM DC Event Trip (epwm_dcevent_trip)

In this example ePWM1, ePWM2, and ePWM3 are configured for PWM Digital Compare Event Trip using Trip zone pin inputs. DCAEVT1, DCAEVT2, DCBEVT1 and DCBEVT2 events are all defined as true when TZ1 is low and TZ2 is high. 3 Examples are included:

- ePWM1 has DCAEVT1 as a one shot trip source The trip event will pull ePWM1A high The trip event will pull ePWM1B low
- ePWM2 has DCAEVT2 as a cycle by cycle trip source The trip event will pull ePWM2A high The trip event will pull ePWM2B low
- ePWM3 reacts to DCAEVT2 and DCBEVT1 events The DCAEVT2 event will pull ePWM3A high The DCBEVT1 event will pull ePWM3B low

Initially tie TZ1 (GPIO12) and TZ2 (GPIO13) high. During the test, monitor ePWM1 or ePWM2 outputs on a scope pull TZ1 low and leave TZ2 high to create a DCAEVT1, DCAEVT2, DCBEVT1 and DCBEVT2. View the EPWM1A/B, EPWM2A/B, EPWM3A/B waveforms on an oscilloscope to see the effect of the events.

External Connections

- EPWM1A is on GPIO0
- EPWM1B is on GPIO1
- EPWM2A is on GPIO2
- EPWM2B is on GPIO3
- EPWM3A is on GPIO4
- EPWM3B is on GPIO5
- TZ1 is on GPIO12

- TZ2 is on GPIO13
- pull TZ1 low and leave TZ2 high to create a DCAEVT1, DCAEVT2, DCBEVT1 and DCBEVT2.

4.13 ePWM DC Event Trip Comparator (epwm_dcevent_trip_comp)

In this example ePWM1 is configured for PWM Digital Compare Event Trip using Comparator1A and comparator1B pin inputs. DCAEVT1, DCBEVT1 events are triggered by increasing the voltage on COMP1B pin to be higher than that of COMP1A pin. In this example:

- ePWM1 has DCAEVT1 and DCBEVT1 as one shot trip sources DCAEVT1 will pull EPWM1A high DCBEVT1 will pull EPWM1B low

Initially make the voltage level at COMP1A to be higher than that of COMP1B. Increase voltage on inverting side of comparator(COMP1B pin) to trigger a DCAEVT1, and DCBEVT1. ePWM1 will react to DCAEVT1 and DCBEVT1 as a 1 shot trip. View the EPWM1A/B waveforms on an oscilloscope to see the effect of the events.

External Connections

- EPWM1A is on GPIO0
- EPWM1B is on GPIO1
- COMP1A is on ADCA2
- COMP1B is on ADCB2
- pull COMP1B to a higher voltage level than COMP1A.

4.14 ePWM Deadband Generation (epwm_deadband)

This example configures ePWM1, ePWM2 and ePWM3 for:

- Count up/down
- Deadband 3 Examples are included:
- ePWM1: Active low PWMs
- ePWM2: Active low complementary PWMs
- ePWM3: Active high complementary PWMs

Each ePWM is configured to interrupt on the 3rd zero event when this happens the deadband is modified such that 0 <= DB <= DB_MAX. That is, the deadband will move up and down between 0 and the maximum value.

External Connections

- EPWM1A is on GPIO0
- EPWM1B is on GPIO1
- EPWM2A is on GPIO2
- EPWM2B is on GPIO3
- EPWM3A is on GPIO4
- EPWM3B is on GPIO5

4.15 ePWM Real-Time Interrupt (epwm_real-time_interrupts)

This example configures the ePWM1 Timer and increments a counter each time an interrupt is taken. ePWM interrupt can be configured as time critical to demonstrate real-time mode functionality and real-time interrupt capability. ControlCard LED2 (GPIO31) is toggled in main loop ControlCard LED3 (GPIO34) is toggled in ePWM1 Timer Interrupt. FREE_SOFT bits and DBBIER.INT3 bit must be set to enable ePWM1 interrupt to be time critical and operational in real time mode after halt command. In this example:

- ePWM1 is initialized
- ePWM1 is cleared at period match and set at Compare-A match
- Compare A match occurs at half period
- GPIOs for LED2 and LED3 are initialized
- Free_Soft bits and DBGIER are cleared
- An interrupt is taken on a zero event for the ePWM1 timer

Watch Variables

- EPwm1TimerIntCount
- EPwm1Regs.TBCTL.bit.FREE_SOFT
- EPwm1Regs.TBCTR
- DBGIER.INT3

4.16 ePWM Timer Interrupt (epwm_timer_interrupts)

This example configures the ePWM Timers and increments a counter each time an interrupt is taken. In this example:

- All ePWM's are initialized.
- All timers have the same period.
- The timers are started sync'ed.
- An interrupt is taken on a zero event for each ePWM timer.
- ePWM1: takes an interrupt every event.
- ePWM2: takes an interrupt every 2nd event.
- ePWM3: takes an interrupt every 3rd event.
- ePWM4: takes an interrupt every event. Thus the Interrupt count for ePWM1 and ePWM4 should be equal.The interrupt count for ePWM2 should be about half that of ePWM1 and the interrupt count for ePWM3 should be about 1/3 that of ePWM1.

Watch Variables

- EPwm1TimerIntCount
- EPwm2TimerIntCount
- EPwm3TimerIntCount
- EPwm4TimerIntCount

4.17 ePWM Trip Zone (epwm_trip_zone)

This example configures ePWM1 and ePWM2 as follows

- ePWM1 has TZ1 and TZ2 as one shot trip sources
- ePWM2 has TZ1 and TZ2 as cycle by cycle trip sources

Initially tie TZ1 and TZ2 high. During the test, monitor ePWM1 or ePWM2 outputs on a scope. Pull TZ1 or TZ2 low to see the effect.

External Connections

- EPWM1A is on GPIO0
- EPWM1B is on GPIO1
- EPWM2A is on GPIO2
- EPWM2B is on GPIO3
- TZ1 is on GPIO12
- TZ2 is on GPIO13

4.18 ePWM Action Qualifier Module using Upcount mode (epwm_up_aq)

This example configures ePWM1, ePWM2, ePWM3 to produce a waveform with independent modulation on EPWMxA and EPWMxB. The compare values CMPA and CMPB are modified within the ePWM's ISR. The TB counter is in upmode.

Monitor the ePWM1 - ePWM3 pins on an oscilloscope.

External Connections

- EPWM1A is on GPIO0
- EPWM1B is on GPIO1
- EPWM2A is on GPIO2
- EPWM2B is on GPIO3
- EPWM3A is on GPIO4
- EPWM3B is on GPIO5

4.19 ePWM Action Qualifier Module using up/down count (epwm_updown_aq)

This example configures ePWM1, ePWM2, ePWM3 to produce an waveform with independent modulation on EPWMxA and EPWMxB. The compare values CMPA and CMPB are modified within the ePWM's ISR. The TB counter is in up/down count mode for this example.

Monitor ePWM1-ePWM3 pins on an oscilloscope as described

External Connections

- EPWM1A is on GPIO0
- EPWM1B is on GPIO1
- EPWM2A is on GPIO2
- EPWM2B is on GPIO3
- EPWM3A is on GPIO4
- EPWM3B is on GPIO5

4.20 eQEP, Frequency measurement(eqep_freqcal)

This test will calculate the frequency and period of an input signal using eQEP module.

EPWM1A is configured to generate a frequency of 5 kHz.

See also:

section on Frequency Calculation for more details on the frequency calculation performed in this example.

In addition to the main example file, the following files must be included in this project:

- **Example_freqcal.c** , includes all eQEP functions
- **Example_EPwmSetup.c** , sets up EPWM1A for use with this example
- **Example_freqcal.h** , includes initialization values for frequency structure.

The configuration for this example is as follows

- Maximum frequency is configured to 10KHz (BaseFreq)
- Minimum frequency is assumed at 50Hz for capture pre-scalar selection

SPEED_FR: High Frequency Measurement is obtained by counting the external input pulses for 10ms (unit timer set to 100Hz).

$$SPEED\_FR = \frac{Count\ Delta}{10ms}$$

SPEED_PR: Low Frequency Measurement is obtained by measuring time period of input edges. Time measurement is averaged over 64edges for better results and capture unit performs the time measurement using pre-scaled SYSCLK

Note that pre-scaler for capture unit clock is selected such that capture timer does not overflow at the required minimum frequency This example runs forever until the user stops it.

Note:

CODE MODIFICATIONS ARE REQUIRED FOR 60 MHZ DEVICES. In DSP2803x_Examples.h in the common/include/ directory, set

- #define CPU_FRQ_60MHZ to 1, and
- #define CPU_FRQ_100MHZ to 0

External Connections

Connect GPIO20/EQEP1A to GPIO0/EPWM1A

Watch Variables

- **freq.freqhz_fr** , Frequency measurement using position counter/unit time out
- **freq.freqhz_pr** , Frequency measurement using capture unit

4.20.1 EPWM Setup(Example_EPwmSetup.c)

This file contains source for the ePWM initialization for the freq calculation module. EPWM1 is set to operate in up-down count mode at a frequency of 5KHz

4.20.2 Frequency Calculation (Example_freqcal.c)

This file includes the EQEP initialization and frequency calculation functions called by **Example_2803xEqep_freqcal.c**. The frequency calculation steps performed by FREQCAL_Calc()at SYSCLKOUT = 60 MHz are described below:

1. This program calculates: $**$**freqhz_fr**$**

$$freqhz\_fr\ or\ v = \frac{x_2 - x_1}{T} .....1$$

If

$$\frac{max}{base}\ freq = 10kHz => 10kHz = \frac{x_2 - x_1}{(2/100Hz)} .....2$$

$$max(x_2 - x_1) = 200counts = freqScaler\_fr$$

Note:
$T = \frac{2}{100Hz}$. 2 is from $\frac{x_2 - x_1}{2}$ because QPOSCNT counts 2 edges per cycle (rising and falling)

If both sides of Equation 2 are divided by 10 kHz, then:

$$1 = \frac{x_2 - x_1}{10kHz * (2/100Hz)}$$

where,

$$[10kHz * \frac{2}{100Hz}] = 200$$

Because

$$x_2 - x_1 < 200(max)$$

$$\frac{x_2 - x_1}{200} < 1$$

for all frequencies less than max

$$freq\_fr = \frac{x_2 - x_1}{200}\ or\ \frac{x_2 - x_1}{10kHz * (2/100Hz)} .....3$$

To get back to original velocity equation, Equation 1, multiply Equation 3 by 10 kHz

$$freqhz\_fr(or\ velocity) = 10kHz * \frac{x_2 - x_1}{10kHz * (2/100Hz)}$$

$$= \frac{x_2 - x_1}{(2/100Hz)} .....final\ equation$$

1. $**$**min freq**$** = $\frac{1\ count}{(2/100Hz)} = 50Hz$
2. $**$**freqhz_pr**$**

$$freqhz\_pr\ or\ v = \frac{X}{t_2 - t_1} .....4$$

If

$$\frac{max}{base}\ freq = 10kHz => 10kHz = \frac{(8/2)}{T} = \frac{8}{2T}$$

where,

- 8 = QCAPCTL [UPPS] (Unit timeout - once every 8 edges)
- 2 = divide by 2 because QPOSCNT counts 2 edges per cycle (rising and falling)
- T = time in seconds = $\frac{t_2 - t_1}{(100MHz/128)}, t_2 - t_1$ = # of QCAPCLK cycles, and $1\ QCAPCLK\ cycle = \frac{1}{(100MHz/128)} = QCPRDLAT$

So:

$$10kHz = 8 * \frac{(60MHz/128)}{2 * (t_2 - t_1)}$$

$$t_2 - t_1 = 8 * \frac{(60MHz/128)}{10kHz * 2} = \frac{(60MHz/128)}{((2 * 10KHz)/8)} .....5$$

$$= 188\ QCAPCLK\ cycles = maximum(t_2 - t_1) = freqScaler\_pr$$

Divide both sides by $(t_2 - t_1)$, and:

$$1 = \frac{188}{t_2 - t_1} = \frac{(60MHz/128)/((2 * 10KHz)/8)}{t_2 - t_1}$$

Because $(t_2 - t_1) < 188(max)$, $\frac{188}{t_2 - t_1} < 1$ for all frequencies less than max

$$freq\_pr = \frac{188}{t_2 - t_1}\ or\ \frac{(60MHz/128)/((2 * 10KHz)/8)}{t_2 - t_1} .....6$$

Now within velocity limits, to get back to original velocity equation, Equation 1, multiply Equation 6 by 10 kHz:

$$freqhz\_fr(or\ velocity) = 10kHz * \frac{(60MHz/128)/((2 * 10KHz)/8)}{t_2 - t_1}$$

$$= \frac{(60MHz/128) * 8}{2 * (t_2 - t_1)}$$

or

$$\frac{8}{2 * (t_2 - t_1) * (QCPRDLAT)} .....final\ equation$$

More detailed calculation results can be found in the Example_freqcal.xls spreadsheet included in the example folder.

4.21 eQEP Speed and Position measurement (eqep_pos_speed)

This example provides position measurement,speed measurement using the capture unit, and speed measurement using unit time out. This example uses the IQMath library. It is used merely to simplify high-precision calculations. The example requires the following hardware connections from EPWM1 and GPIO pins (simulating QEP sensor) to QEP peripheral.

- eQEP1A <- ePWM1A (simulates eQEP Phase A signal)
- eQEP1B <- ePWM1B (simulates eQEP Phase B signal)
- eQEP1I <- GPIO4 (simulates eQEP Index Signal) See DESCRIPTION in Example_posspeed.c for more details on the calculations performed in this example. In addition to this file, the following files must be included in this project:
- Example_posspeed.c - includes all eQEP functions

- Example_EPwmSetup.c - sets up ePWM1A and ePWM1B as simulated QA and QB encoder signals
- Example_posspeed.h - includes initialization values for pos and speed structure

Note:

- Maximum speed is configured to 6000rpm(BaseRpm)
- Minimum speed is assumed at 10rpm for capture pre-scalar selection
- Pole pair is configured to 2 (pole_pairs)
- QEP Encoder resolution is configured to 4000counts/revolution (mech_scaler)
- which means: 4000/4 = 1000 line/revolution quadrature encoder (simulated by EPWM1)
- EPWM1 (simulating QEP encoder signals) is configured for 5kHz frequency or 300 rpm (=4∗5000 cnts/sec ∗ 60 sec/min)/4000 cnts/rev)
- SPEEDRPM_FR: High Speed Measurement is obtained by counting the QEP input pulses for 10ms (unit timer set to 100Hz).
- SPEEDRPM_FR = (Position Delta/10ms) ∗ 60 rpm
- SPEEDRPM_PR: Low Speed Measurement is obtained by measuring time period of QEP edges. Time measurement is averaged over 64edges for better results and capture unit performs the time measurement using pre-scaled SYSCLK
- pre-scaler for capture unit clock is selected such that capture timer does not overflow at the required minimum RPM speed.

External Connections

- Connect eQEP1A(GPIO20) to ePWM1A(GPIO0)(simulates eQEP Phase A signal)
- Connect eQEP1B(GPIO21) to ePWM1B(GPIO1)(simulates eQEP Phase B signal)
- Connect eQEP1I(GPIO23) to GPIO4 (simulates eQEP Index Signal)

Watch Variables

- qep_posspeed.SpeedRpm_fr - Speed meas. in rpm using QEP position counter
- qep_posspeed.SpeedRpm_pr - Speed meas. in rpm using capture unit
- qep_posspeed.theta_mech - Motor mechanical angle (Q15)
- qep_posspeed.theta_elec - Motor electrical angle (Q15)

4.22 External Interrupt (external_interrupt)

This program sets up GPIO0 as XINT1 and GPIO1 as XINT2. Two other GPIO signals are used to trigger the interrupt (GPIO30 triggers XINT1 and GPIO31 triggers XINT2). XINT1 input is synched to SYSCLKOUT XINT2 has a long qualification - 6 samples at 510∗SYSCLKOUT each. GPIO34 will go high outside of the interrupts and low within the interrupts. This signal can be monitored on a scope. Each interrupt is fired in sequence - XINT1 first and then XINT2.

Monitor GPIO34 with an oscilloscope. GPIO34 will be high outside of the ISRs and low within each ISR.

External Connections

- Connect GPIO30 to GPIO0. GPIO0 is assigned to XINT1
- Connect GPIO31 to GPIO1. GPIO1 is assigned to XINT2

Watch Variables

- Xint1Count - XINT1 interrupt count
- Xint2Count - XINT2 interrupt count
- LoopCount - idle loop count

4.23 ePWM Timer Interrupt From Flash (flash_f28035)

This example runs the ePWM interrupt example from flash. ePwm1 Interrupt will run from RAM and puts the flash into sleep mode. ePwm2 Interrupt will run from RAM and puts the flash into standby mode. ePWM3 Interrupt will run from FLASH. All timers have the same period. The timers are started sync'ed. An interrupt is taken on a zero event for each ePWM timer.GPIO34 is toggled while in the background loop. Note:

- ePWM1: takes an interrupt every event
- ePWM2: takes an interrupt every 2nd event
- ePWM3: takes an interrupt every 3rd event Thus the Interrupt count for ePWM1, ePWM4-ePWM6 should be equal The interrupt count for ePWM2 should be about half that of ePWM1 and the interrupt count for ePWM3 should be about 1/3 that of ePWM1

Follow these steps to run the program.

- Build the project
- Flash the .out file into the device.
- Set the hardware jumpers to boot to Flash (put position 1 and 2 of SW2 on control Card to ON position).
- Use the included GEL file to load the project, symbols defined within the project and the variables into the watch window.

Steps that were taken to convert the ePWM example from RAM to Flash execution:

- Change the linker cmd file to reflect the flash memory map.
- Make sure any initialized sections are mapped to Flash. In SDFlash utility this can be checked by the View->Coff/Hex status utility. Any section marked as "load" should be allocated to Flash.
- Make sure there is a branch instruction from the entry to Flash at 0x3F7FF6 to the beginning of code execution. This example uses the DSP0x_CodeStartBranch.asm file to accomplish this.
- Set boot mode Jumpers to "boot to Flash"
- For best performance from the flash, modify the waitstates and enable the flash pipeline as shown in this example. Note: any code that manipulates the flash waitstate and pipeline control must be run from RAM. Thus these functions are located in their own memory section called ramfuncs.

Watch Variables

- EPwm1TimerIntCount
- EPwm2TimerIntCount
- EPwm3TimerIntCount

4.24 GPIO Setup (gpio_setup)

This example Configures the 2803x GPIO into two different configurations This code is verbose to illustrate how the GPIO could be setup.In a real application, lines of code can be combined for improved code size and efficiency.

This example only sets-up the GPIO.. nothing is actually done with the pins after setup.

In general:

- All pullup resistors are enabled. For ePWMs this may not be desired.
- Input qual for communication ports (eCAN, SPI, SCI, I2C) is asynchronous

- Input qual for Trip pins (TZ) is asynchronous
- Input qual for eCAP and eQEP signals is synch to SYSCLKOUT
- Input qual for some I/O's and interrupts may have a sampling window

4.25 GPIO Toggle Test (gpio_toggle)

Note:
ALL OF THE I/O'S TOGGLE IN THIS PROGRAM. MAKE SURE THIS WILL NOT DAMAGE YOUR HARDWARE BEFORE RUNNING THIS EXAMPLE.

Three different examples are included. Select the example (data, set/clear or toggle) to execute before compiling using the macros found at the top of the code.

Each example toggles all the GPIOs in a different way, the first through writing values to the GPIO DATA registers, the second through the SET/CLEAR registers and finally the last through the TOGGLE register

The pins can be observed using Oscilloscope.

4.26 HRCAP Capture HRPWM Pulses (hrcap_capture_hrpwm)

This program generates a high-resolution PWM output on ePWM1A (with falling edge moving by 8 HRPWM MEP steps per period), and the HRCAP is configured to generate an interrupt on either rising edges OR falling edges to continuously capture up to 5 PWM periods (5 high pulses and 5 low pulses) and calculate the high resolution pulse widths in integer + fractional HCCAPCLK cycles in Q16 format.

Monitor pulsewidthlow and pulsewidthhigh in the Watch Window (pulsewidthlow gradually decreases and pulsewidthhigh gradually increases as CMPAHR moves the falling edge MEP steps.) Another option is to monitor periodwidth in the Watch Window, which should not change much because the period stays the same.

- User must set
  - #define FALLTEST 1 and RISETEST 0 for falling edge interrupts
  - #define RISETEST 1 and FALLTEST 0 for rising edge interrupts
- To measure PERIOD, user must set: #define PERIODTEST 1
- To measure high pulse width or low pulse width, user must set: #define PERIODTEST 0

To determine the actual pulse $\frac{width}{period}$ time in $\frac{pulsewidthlow}{pulsewidthhigh}$ period:

$pulsewidthinseconds = pulsewidth[n] * (1HCCAPCLKcycle)$

(i.e. 1 HCCAPCLK cycle is 8.33 ns for 120 MHz HCCAPCLK)

Note:
- THE FOLLOWING DEFINITION AND STRUCTURE MUST BE DEFINED IN CODE IN ORDER TO USE THE HCCAL LIBRARY
  - #define NUM_HRCAP 3 // # of HRCAP modules + 1 (2 HRCAP's on 2803x + 1 = 3)
  - volatile struct HRCAP_REGS *HRCAP[NUM_HRCAP] = {0, &HRCap1Regs, &HRCap2Regs};

- Because there is no EMU FREE/STOP support in the HRCAP peripheral, the HRCAP results cannot easily be seen in "Real-time continuous refresh" debug mode. The only way to see accurate results of the capture is to set a breakpoint in the HRCAP ISR immediately after data has been captured. Otherwise the results read at any other time in the program via the debugger could be mid-capture between one period and the next, and therefore bogus.

External Connections

- HRCAP1 is on GPIO26
- ePWM1A is on GPIO0
- Connect output of ePWM1A to input of HRCAP1 (GPIO0->GPIO26)

Watch Variables

- pulsewidthlow
  - Pulse Width of Low Pulses (# of HCCAPCLK cycles - int + frac) in Q16 format (i.e. upper 16-bits integer, lower 16-bits fraction)
- pulsewidthhigh
  - Pulse Width of High Pulses (# of HCCAPCLK cycles - int + frac) in Q16 format (i.e. upper 16-bits integer, lower 16-bits fraction)
- periodwidth
  - Period Width (# of HCCAPCLK cycles - int + frac) in Q16 format (i.e. upper 16-bits integer, lower 16-bits fraction)

4.27 HRCAP Non-High Resolution Capture PWM Pulses (hrcap_capture_pwm)

HRCAP1 is configured for non high-resolution capture mode to capture the time between rising and falling edges of the PWM1A output.

This example configures ePWM1A for:

- Up count
- Period starts at 80 and goes up to 4000 then back down again.
- Toggle output on PRD

Note:

- On Piccolo-B, the HCCAPCLK frequency is 2 $*$ SYSCLK.EPWM period in up-count mode is TBPRD + 1 SYSCLK counts. Therefore,

  $ePWM\ period(up\ count\ mode) = 2 * (TBPRD + 1) HCCAPCLK\ counts$
- Because there is no EMU FREE/STOP support in the HRCAP peripheral, the HRCAP results cannot easily be seen in "Real-time continuous refresh" debug mode. The only way to see accurate results of the capture is to set a breakpoint in the HRCAP ISR immediately after data has been captured. Otherwise the results read at any other time in the program via the debugger could be mid-capture between one period and the next, and therefore bogus.

External Connections

- HRCAP1 is on GPIO26
- ePWM1A is on GPIO0
- Connect output of ePWM1A to input of HRCAP1 (GPIO0->GPIO26)

Watch Variables

- PULSELOW
  - Pulse Width of Low Pulses (# of HCCAPCLK cycles - integer)
- PULSEHIGH
  - Pulse Width of High Pulses (# of HCCAPCLK cycles - integer)

4.28 High Resolution PWM (hrpwm)

This example modifies the MEP control registers to show edge displacement due to the HRPWM control extension of the respective EPwm module All EPwm1A,2A,3A,4A channels (GPIO0, GPIO2, GPIO4, GPIO6) will have fine edge movement due to HRPWM logic

1. $PWM\ Freq = \frac{SYSCLK}{period=10}$,
   - ePWM1A toggle low/high with MEP control on rising edge
   - ePWM1B toggle low/high with NO HRPWM control
1. $PWM\ Freq = \frac{SYSCLK}{period=20}$,
   - ePWM2A toggle low/high with MEP control on rising edge
   - ePWM2B toggle low/high with NO HRPWM control
1. $PWM\ Freq = \frac{SYSCLK}{period=10}$,
   - ePWM3A toggle as high/low with MEP control on falling edge
   - ePWM3B toggle low/high with NO HRPWM control
1. $PWM\ Freq = \frac{SYSCLK}{period=20}$,
   - ePWM4A toggle as high/low with MEP control on falling edge
   - ePWM4B toggle low/high with NO HRPWM control

External Connections

Monitor ePWM1-ePWM4 pins on an oscilloscope as described below:

- ePWM1A is on GPIO0
- ePWM1B is on GPIO1
- ePWM2A is on GPIO2
- ePWM2B is on GPIO3
- ePWM3A is on GPIO4
- ePWM3B is on GPIO5
- ePWM4A is on GPIO6
- ePWM4B is on GPIO7

4.29 High Resolution PWM SFO V6 Duty Cycle (hrpwm_duty_sfo_v6)

This example modifies the MEP control registers to show edge displacement due to the HRPWM control extension of the respective ePWM module. This example calls the following TI's MEP Scale Factor Optimizer (SFO) software library V6 functions:

int SFO();

- updates MEP_ScaleFactor dynamically when HRPWM is in use
- updates HRMSTEP register (exists only in EPwm1Regs register space but valid for all channels) with MEP_ScaleFactor value
- **Returns**
  - 2 if error: MEP_ScaleFactor is greater than maximum value of 255 (Auto-conversion may not function properly under this condition)
  - 1 when complete for the specified channel
  - 0 if not complete for the specified channel

This example is intended to explain the HRPWM capabilities. The code can be optimized for code efficiency. Refer to TI's Digital power application examples and TI Digital Power Supply software libraries for details. All ePWM1A-7A channels will have fine edge movement due to the HRPWM logic

Note:

- This program requires the DSP2803x header files, which include the following files required for this example: SFO_V6.h and SFO_TI_Build_V6.lib
- For more information on using the SFO software library, see the 2803x High-Resolution Pulse Width Modulator (HRPWM) Reference Guide

External Connections

Monitor ePWM1A-ePWM4A (GPIO0-GPIO7) pins on an oscilloscope.

Running the Application

1. ∗∗**!!IMPORTANT!!**∗∗ : in SFO_V6.h, set PWM_CH to the max number of HRPWM channels plus one. For example, for the F2803x, the maximum number of HRPWM channels is 7. 7+1=8, so set #define PWM_CH 8 in SFO_V6.h. (Default is 5)
2. In this file, set #define AUTOCONVERT to 1 to enable MEP step auto-conversion logic. Otherwise, to manually perform MEP calculations in software, clear to 0.
3. Run this example at maximum SYSCLKOUT (60MHz)
4. Add "UpdateFine" variable to the watch window either manually or using the supplied javascript
5. Activate Real time mode
6. Run the code
7. Watch ePWM A channel waveforms on a Oscilloscope

Watch Variables

- UpdateFine
  - Set the variable UpdateFine = 1 to observe the ePWMxA output with HRPWM capabilities (default) Observe the duty cycle of the waveform change in fine MEP steps
  - Change the variable UpdateFine to 0, to observe the ePWMxA output without HRPWM capabilities Observe the duty cycle of the waveform change in coarse SYSCLKOUT cycle steps.

4.30 High Resolution PWM SFO V6 High-Resolution Period (Up-Down Count) Multi-channel (hrpwm_mult_ch_prdupdown_sfo_v6)

This example modifies the MEP control registers to show edge displacement for high-resolution period/frequency on multiple synchronized ePWM channels in Up-Down count mode due to the HRPWM control extension of the respective ePWM module.

All period and compare register updates occur in an ISR which interrupts at ePWM1 TBCTR = 0. This ensures that period and compare register updates across all ePWM modules occur within the same period. This example calls the following TI's MEP Scale Factor Optimizer (SFO) software library V6 functions:

int SFO();

- updates MEP_ScaleFactor dynamically when HRPWM is in use
- updates HRMSTEP register (exists only in EPwm1Regs register space but valid for all channels) with MEP_ScaleFactor value
- **Returns**
  - 2 if error: MEP_ScaleFactor is greater than maximum value of 255 (Auto-conversion may not function properly under this condition)
  - 1 when complete for the specified channel
  - 0 if not complete for the specified channel

All ePWM1A-4A channels will be synchronized to each other (with ePWM1 sync'd to the SWFSYNC) and have fine edge period movement due to the HRPWM logic. This example can be used as a primitive building block for applications which require high resolution frequency control with synchronized ePWM modules (i.e. resonant converter applications)

Note:

- This program requires the DSP2803x header files, which include the following files required for this example: SFO_V6.h and SFO_TI_Build_V6.lib
- For more information on using the SFO software library, see the 2803x High-Resolution Pulse Width Modulator (HRPWM) Reference Guide

External Connections

Monitor ePWM1A-ePWM4A (GPIO0-GPIO7) pins on an oscilloscope.

Running the Application

1. ∗∗**!!IMPORTANT!!**∗∗ : in SFO_V6.h, set PWM_CH to the max number of HRPWM channels used plus one. For this example, the maximum number of HRPWM channels used is 4. 4+1=5, so set #define PWM_CH 5 in SFO_V6.h. (Default is 5)
2. Run this example at maximum SYSCLKOUT (60 MHz)
3. Add "UpdateFine" and "UpdateCoarse" variables to the watch window either manually or using the supplied javascript.
4. Activate Real time mode
5. Run the code
6. Watch ePWM A channel waveforms on a Oscilloscope

Watch Variables

- UpdateFine
  - Set the variable UpdateFine = 1 to observe the ePWMxA output with HRPWM capabilities (default) Observe the period/frequency of the waveform changes in fine MEP steps
  - Change the variable UpdateFine to 0, to observe the ePWMxA output without HRPWM capabilities
- UpdateCoarse
  - To change the period/frequency in coarse steps, uncomment the relevant code, rebuild and re-run with UpdateCoarse = 1. Observe the period/frequency of the waveform changes in coarse SYSCLKOUT cycle steps.

4.31 High Resolution PWM SFO V6 High-Resolution Period (Up Count)(hrpwm_prdup_sfo_v6)

This example modifies the MEP control registers to show edge displacement for high-resolution period with ePWM in Up count mode due to the HRPWM control extension of the respective ePWM module. This example calls the following TI's MEP Scale Factor Optimizer (SFO) software library V6 functions:

int SFO();

- updates MEP_ScaleFactor dynamically when HRPWM is in use
- updates HRMSTEP register (exists only in EPwm1Regs register space but valid for all channels) with MEP_ScaleFactor value
- **Returns**
  - 2 if error: MEP_ScaleFactor is greater than maximum value of 255 (Auto-conversion may not function properly under this condition)
  - 1 when complete for the specified channel
  - 0 if not complete for the specified channel

This example is intended to explain the HRPWM capabilities. The code can be optimized for code efficiency. Refer to TI's Digital power application examples and TI Digital Power Supply software libraries for details. All ePWM1A-7A channels will have fine edge movement due to the HRPWM logic

Note:

- This program requires the DSP2803x header files, which include the following files required for this example: SFO_V6.h and SFO_TI_Build_V6.lib
- For more information on using the SFO software library, see the 2803x High-Resolution Pulse Width Modulator (HRPWM) Reference Guide

External Connections

Monitor ePWM1A-ePWM4A (GPIO0-GPIO7) pins on an oscilloscope.

Running the Application

1. $**$**!!IMPORTANT!!**$**$: in SFO_V6.h, set PWM_CH to the max number of HRPWM channels plus one. For example, for the F2803x, the maximum number of HRPWM channels is 7. 7+1=8, so set #define PWM_CH 8 in SFO_V6.h. (Default is 5)
2. Run this example at maximum SYSCLKOUT (60 MHz)
3. Add "UpdateFine" variable to the watch window either manually or using the supplied javascript.
4. Activate Real time mode
5. Run the code
6. Watch ePWM A channel waveforms on a Oscilloscope

Watch Variables

- UpdateFine
  - Set the variable UpdateFine = 1 to observe the ePWMxA output with HRPWM capabilities (default) Observe the period/frequency of the waveform changes in fine MEP steps
  - Change the variable UpdateFine to 0, to observe the ePWMxA output without HRPWM capabilities Observe the period/frequency of the waveform changes in coarse SYSCLKOUT cycle steps.

4.32 High Resolution PWM SFO V6 High-Resolution Period (Up-Down Count)(hrpwm_prdupdown_sfo_v6)

This example modifies the MEP control registers to show edge displacement for high-resolution period with ePWM in Up-Down count mode due to the HRPWM control extension of the respective ePWM module. This example calls the following TI's MEP Scale Factor Optimizer (SFO) software library V6 functions:

int SFO();

- updates MEP_ScaleFactor dynamically when HRPWM is in use
- updates HRMSTEP register (exists only in EPwm1Regs register space but valid for all channels) with MEP_ScaleFactor value
- **Returns**
  - 2 if error: MEP_ScaleFactor is greater than maximum value of 255 (Auto-conversion may not function properly under this condition)
  - 1 when complete for the specified channel
  - 0 if not complete for the specified channel

This example is intended to explain the HRPWM configuration for high resolution period/frequency. The code can be optimized for code efficiency. Refer to TI's Digital power application examples and TI Digital Power Supply software libraries for details. ePWM1A (GPIO0) will have fine edge movement due to the HRPWM logic

Note:

- This program requires the DSP2803x header files, which include the following files required for this example: SFO_V6.h and SFO_TI_Build_V6.lib
- For more information on using the SFO software library, see the 2803x High-Resolution Pulse Width Modulator (HRPWM) Reference Guide

External Connections

Monitor ePWM1A (GPIO0) pin on an oscilloscope.

Running the Application

1. ∗∗!!IMPORTANT!!∗∗ : in SFO_V6.h, set PWM_CH to the max used HRPWM channel plus one. For example, for the F2803x, the maximum number of HRPWM channels is 7. 7+1=8, so set #define PWM_CH 8 in SFO_V6.h. (Default is 5)
2. For this specific example, you could set #define PWM_CH 2 (because it only uses ePWM1), but to cover all examples, PWM_CH is currently set to a default value of 5.
3. Load the code and add the watch variables to the watch window. See below for a list of watch variables
4. Run this example at maximum SYSCLKOUT (60 or 40 MHz)
5. Activate Real time mode
6. Run the code
7. Watch ePWM1A waveform on a Oscilloscope

Watch Variables

- UpdateFine
  - Set the variable UpdateFine = 1 to observe the ePWMxA output with HRPWM capabilities (default) Observe the period/frequency of the waveform changes in fine MEP steps

  - Change the variable UpdateFine to 0, to observe the ePWMxA output without HRPWM capabilities Observe the period/frequency of the waveform changes in coarse SYSCLKOUT cycle steps.
- PeriodFine
- EPwm1Regs.TBPRD
- EPwm1Regs.TBPRDHR

4.33 High Resolution PWM with slider(hrpwm_slider)

This example modifies the MEP control registers to show edge displacement due to HRPWM control blocks of the respective EPwm module, EPwm1A, 2A, 3A, and 4A channels (GPIO0, GPIO2, GPIO4, and GPIO6) will have fine edge movement due to HRPWM logic.

External Connections

Monitor EPwm1-EPwm4 pins on an oscilloscope as described below.

- ePWM1A is on GPIO0
- ePWM1B is on GPIO1
- ePWM2A is on GPIO2
- ePWM2B is on GPIO3
- ePWM3A is on GPIO4
- ePWM3B is on GPIO5

Running the Application

1. Launch the target configuration and connect to the target first
2. Load the program and set it up to run in real time mode. Do not run yet!
3. Load the Example_2803xHRPWM_slider.gel file (provided in the folder)
4. Select the DSP2803x HRPWM FineDutySlider from the Scripts menu (debug perspective). A FineDuty slider graphics will show up in CCS. (See Below)
5. Add "DutyFine" variable to the watch window either manually or using the supplied javascript. This variable is controlled by the slider
6. Run the example
7. Use the Slider to and observe the EPwm edge displacement for each slider step change. This explains the MEP control on the EPwmxA channels,
   (a) $PWMFreq = \frac{SYSCLK}{period=10}$
   ePWM1A toggle low/high with MEP control on rising edge
   $PWMFreq = \frac{SYSCLK}{period=10}$
   ePWM1B toggle low/high with NO HRPWM control
   (b) $PWMFreq = \frac{SYSCLK}{period=20}$
   ePWM2A toggle low/high with MEP control on rising edge
   $PWMFreq = \frac{SYSCLK}{period=20}$
   ePWM2B toggle low/high with NO HRPWM control
   (c) $PWMFreq = \frac{SYSCLK}{period=10}$
   ePWM3A toggle as high/low with MEP control on falling edge
   $PWMFreq = \frac{SYSCLK}{period=10}$
   ePWM3B toggle low/high with NO HRPWM control
   (d) $PWMFreq = \frac{SYSCLK}{period=20}$
   ePWM4A toggle as high/low with MEP control on falling edge
   $PWMFreq = \frac{SYSCLK}{period=20}$
   ePWM4B toggle low/high with NO HRPWM control

Watch Variables

- DutyFine

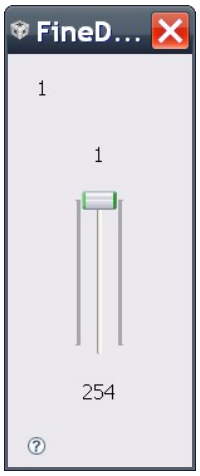

Figure 4.1: Fine Duty Slider

4.34 I2C EEPROM(i2c_eeprom)

This program requires an external I2C EEPROM connected to the I2C bus at address 0x50. This program will write 1-14 words to EEPROM and read them back. The data written and the EEPROM address written to are contained in the message structure, **I2cMsgOut1**. The data read back will be contained in the message structure **I2cMsgIn1**.

Note:

This program will only work on kits that have an on-board I2C EEPROM. T (e.g. F2803x eZdsp)

Watch Variables

- I2cMsgIn1
- I2cMsgOut1

4.35 LIN-A External Analog Loop Back(lina_external_loopback)

This program is intended to validate the analog loopback functionality of the LIN module.

The code configures the LIN module and then transmits data onto the LIN bus. The LIN module receives this data off the LIN bus. Correct data transmission is verified.

Note:

The LIN pins should be connected to a LIN transceiver.

Watch Variables

- Rx_Data_LIN_MASTER
- Tx_Data_LIN_MASTER
- intCount
- pstatus

- error
- Int_Flag
- int0Count
- int1Count

External Connections

Connect the LIN module TXD and RXD pins to a LIN transceiver board in order to achieve analog loopback.

- GPIO9 is LIN-TX
- GPIO11 is LIN-RX

4.36 LIN-SCI Digital Loop Back(lina_sci_echoback)

This test receives and echos back data through the LIN-A port which has been configured for SCI operation.

The PC application 'hypterterminal' can be used to view the data from the LIN and to send information to the LIN. Characters received by the LIN port are sent back to the host.

Running the Application

1. Configure hyperterminal: Use the included hyperterminal configuration file SCI_96.ht. To load this configuration in hyperterminal
   (a) Open hyperterminal
   (b) Go to file->open
   (c) Browse to the location of the project and select the SCI_96.ht file.
2. Check the COM port. The configuration file is currently setup for COM1. If this is not correct, disconnect (Call->Disconnect) Open the File-Properties dialog and select the correct COM port.
3. Connect hyperterminal Call->Call and then start the 2803x LIN-SCI echoback program execution.
4. The program will print out a greeting and then ask you to enter a character which it will echo back to hyperterminal.

Note:

If you are unable to open the .ht file, you can create a new one with the following settings

- Find correct COM port
- Bits per second = 19200
- Date Bits = 8
- Parity = None
- Stop Bits = 1
- Hardware Control = None

Watch Variables

- **LoopCount**, for the number of characters sent
- **ReceivedChar**, for character received from Hyperterminal

External Connections

Connect the LIN-A port to a PC via a transceiver and cable.

- GPIO9 is LIN-TX
- GPIO11 is LIN-RX

4.37 LIN-SCI Digital Loop Back Interrupts(lina_sci_loopback_interrupts)

This program uses the internal loop back test mode of the peripheral. Other then boot mode pin configuration, no other hardware configuration is required. Interrupts are enabled

A stream of data is sent and then compared to the received stream. The SCI-A sent data looks like this:

00 01 02 03

04 05 06 07

08 09 0A 0B

etc..

The pattern is repeated forever.

Watch Variables

- **sdataA**, Data being sent
- **rdataA**, Data received
- **LinL0IntCount**, Number of transmissions received

4.38 Low Power Modes: Halt Mode and Wakeup (lpm_haltwake)

This example puts the device into HALT mode. If the lowest possible current consumption in HALT mode is desired, the JTAG connector must be removed from the device board while the device is in HALT mode.

The example then wakes up the device from HALT using GPIO0. GPIO0 wakes the device from HALT mode when a high-to-low signal is detected on the pin. This pin must be pulsed by an external agent for wakeup.

The wakeup process begins as soon as GPIO0 is held low for the time indicated in the device datasheet. After the device wakes up, GPIO1 can be observed to go high.

GPIO0 is configured as the LPM wakeup pin to trigger a WAKEINT interrupt upon detection of a low pulse. Initially, pull GPIO0 high externally. To wake device from halt mode, pull GPIO0 low for at least the crystal startup time + 2 OSCLKS, then pull it high again.

To observe when device wakes from HALT mode, monitor GPIO1 with an oscilloscope (set to 1 in WAKEINT ISR)

4.39 Low Power Modes: Device Idle Mode and Wakeup(lpm_idlewake)

This example puts the device into IDLE mode then wakes up the device from IDLE using XINT1 which triggers on a falling edge from GPIO0.

This pin must be pulled from high to low by an external agent for wakeup. GPIO0 is configured as an XINT1 pin to trigger an XINT1 interrupt upon detection of a falling edge.

Initially, pull GPIO0 high externally. To wake device from idle mode by triggering an XINT1 interrupt, pull GPIO0 low (falling edge)

External Connections

To observe the device wakeup from IDLE mode, monitor GPIO1 with an oscilloscope, which goes high in the XINT_1_ISR.

4.40 Low Power Modes: Device Standby Mode and Wakeup(lpm_standbywake)

This example puts the device into STANDBY mode. If the lowest possible current consumption in STANDBY mode is desired, the JTAG connector must be removed from the device board while the device is in STANDBY mode.

GPIO0 is configured as the LPM wakeup pin to trigger a WAKEINT interrupt upon detection of a low pulse. Initially, pull GPIO0 high externally. To wake device from standby mode, pull GPIO0 low for at least (2+QUALSTDBY) OSCLKS, then pull it high again.

The example then wakes up the device from STANDBY using GPIO0. GPIO0 wakes the device from STANDBY mode when a low pulse (signal goes high->low->high)is detected on the pin. This pin must be pulsed by an external agent for wakeup.

As soon as GPIO0 goes high again after the pulse, the device should wake up, and GPIO1 can be observed to toggle.

External Connections

To observe when device wakes from STANDBY mode, monitor GPIO1 with an oscilloscope (set to 1 in WAKEINT ISR)

4.41 Internal Oscillator Compensation(osc_comp)

This program shows how to use the internal oscillator compensation functions in DSP2803x_OscComp.c. The temperature sensor is sampled and the raw temp sensor value is passed to the oscillator compensation function, which uses this parameter to compensate for frequency drift of the internal oscillator over temperature

Note:

- This program makes use of variables stored in OTP during factory test on 2803x TMS devices.
- These OTP locations on pre-TMS devices may not be populated. Ensure that the following memory locations in TI OTP are populated (not 0xFFFF) before use:
  - 0x3D7E90 to 0x3D7EA4

Watch Variables

- temp
- SysCtrlRegs.INTOSC1TRIM
- SysCtrlRegs.INTOSC2TRIM

4.42 SCI Echo Back(sci_echoback)

This test receives and echo-backs data through the SCI-A port.

The PC application 'hypterterminal' can be used to view the data from the SCI and to send information to the SCI. Characters received by the SCI port are sent back to the host.

Running the Application

1. Configure hyperterminal: Use the included hyperterminal configuration file SCI_96.ht. To load this configuration in hyperterminal
   (a) Open hyperterminal
   (b) Go to file->open
   (c) Browse to the location of the project and select the SCI_96.ht file.
2. Check the COM port. The configuration file is currently setup for COM1. If this is not correct, disconnect (Call->Disconnect) Open the File-Properties dialog and select the correct COM port.
3. Connect hyperterminal Call->Call and then start the 2803x SCI echoback program execution.
4. The program will print out a greeting and then ask you to enter a character which it will echo back to hyperterminal.

Note:

If you are unable to open the .ht file, you can create a new one with the following settings

- Find correct COM port
- Bits per second = 9600
- Date Bits = 8
- Parity = None
- Stop Bits = 1
- Hardware Control = None

Watch Variables

- **LoopCount**, for the number of characters sent
- ErrorCount

External Connections

Connect the SCI-A port to a PC via a transceiver and cable.

- GPIO28 is SCI_A-RXD (Connect to Pin3, PC-TX, of serial DB9 cable)
- GPIO29 is SCI_A-TXD (Connect to Pin2, PC-RX, of serial DB9 cable)

4.43 SCI Digital Loop Back(scia_loopback)

This program uses the internal loop back test mode of the peripheral. Other then boot mode pin configuration, no other hardware configuration is required.

This test uses the loopback test mode of the SCI module to send characters starting with 0x00 through 0xFF. The test will send a character and then check the receive buffer for a correct match.

Watch Variables

- **LoopCount** , Number of characters sent
- **ErrorCount** , Number of errors detected
- **SendChar** , Character sent
- **ReceivedChar** , Character received

4.44 SCI Digital Loop Back with Interrupts(scia_loopback_interrupts)

This program uses the internal loop back test mode of the peripheral. Other then boot mode pin configuration, no other hardware configuration is required. Both interrupts and the SCI FIFOs are used.

A stream of data is sent and then compared to the received stream. The SCI-A sent data looks like this:

00 01

01 02

02 03

....

FE FF

FF 00

etc..

The pattern is repeated forever.

Watch Variables

- **sdataA** , Data being sent
- **rdataA** , Data received
- **rdata_pointA** ,Keep track of where we are in the datastream. This is used to check the incoming data

4.45 SPI Digital Loop Back(spi_loopback)

This program uses the internal loop back test mode of the peripheral. Other then boot mode pin configuration, no other hardware configuration is required. Interrupts are not used.

A stream of data is sent and then compared to the received stream. The sent data looks like this:

0000 0001 0002 0003 0004 0005 0006 0007 .... FFFE FFFF

This pattern is repeated forever.

Watch Variables

- **sdata** , sent data
- **rdata** , received data

4.46 SPI Digital Loop Back with Interrupts(spi_loopback_interrupts)

This program uses the internal loop back test mode of the peripheral. Other then boot mode pin configuration, no other hardware configuration is required. Both interrupts and the SPI FIFOs are used.

A stream of data is sent and then compared to the received stream. The sent data looks like this:

0000 0001

0001 0002
0002 0003
....
FFFE FFFF
FFFF 0000
etc..
This pattern is repeated forever.

Watch Variables

- **sdata** , Data to send
- **rdata** , Received data
- **rdata_point** , Used to keep track of the last position in the receive stream for error checking

4.47 Software Prioritized Interrupts(sw_prioritized_interrupts)

For most applications, the hardware prioritizing of the the PIE module is sufficient. For applications that need custom prioritizing, this example illustrates how this can be done through software.

For more information on F2803x interrupt priorities, refer to the user guide in the DSP2803x/doc directory.

This program simulates interrupt conflicts by writing to the PIEIFR registers. This will cause multiple interrupt requests to come into the PIE block at the same time.

The interrupt service routines are software prioritized as per the table found in the DSP2803x_SWPrioritizedIsrLevels.h file.

Running the Application

1. Before compiling you must set the Global and Group interrupt priorities in the DSP2803x_SWPrioritizedIsrLevels.h file.
2. Select which test case you'd like to run with the #define CASE directive (1-9, default 1).
3. Compile the code, load, and run
4. At the end of each test there is a hard coded breakpoint (ESTOP0). When code stops at the breakpoint, examine the ISRTrace buffer to see the order in which the ISR's completed. All PIE interrupts will be added to the ISRTrace. The ISRTrace will consist of a list of hex values as shown:
   0x00wx <- PIE Group w interrupt x finished first
   0x00yz <- PIE Group y interrupt z finished next
5. If desired, set a new set of Global and Group interrupt priorities and repeat the test to see the change.

Watch Variables

- **ISRTrace** , Trace of ISR's in the order they complete. After each test, examine this buffer to determine if the ISR's completed in the order desired.

4.48 Timer based blinking LED(timed_led_blink)

This example configures CPU Timer0 for a 500 msec period, and toggles the GPIO34 LED once per interrupt. For testing purposes, this example also increments a counter each time the timer asserts an interrupt.

Watch Variables

- CpuTimer0.InterruptCount

External Connections

Monitor the GPIO34 LED blink on (for 500 msec) and off (for 500 msec) on the 2803x control card.

4.49 Watchdog interrupt Test(watchdog)

This program exercises the watchdog.

First the watchdog is connected to the WAKEINT interrupt of the PIE block. The code is then put into an infinite loop.

The user can select to feed the watchdog key register or not by commenting the following line of code in the infinite loop: **ServiceDog()**;

If the watchdog key register is fed by the ServiceDog function then the WAKEINT interrupt is not taken. If the key register is not fed by the ServiceDog function then WAKEINT will be taken.

Watch Variables

- **LoopCount** , for the number of times through the infinite loop
- **WakeCount** , for the number of times through WAKEINT

A Interrupt Service Routine Priorities

A.1 Interrupt Hardware Priority Overview

With the PIE block enabled, the interrupts are prioritized in hardware by default as follows:
Global Priority (CPU Interrupt level):

CPU Interrupt	Hardware Priority
Reset	1(Highest)
INT1	5
INT2	6
INT3	7
INT4	8
INT5	9
INT6	10
INT7	11
...	...
INT12	16
INT13	17
INT14	18
DLOGINT	19(Lowest)
RTOSINT	20
reserved	2
NMI	3
ILLEGAL	-
USER1	-(Software Interrupts)
USER2	-
...	...

CPU Interrupts INT1 - INT14, DLOGINT and RTOSINT are maskable interrupts. These interrupts can be enabled or disabled by the CPU Interrupt enable register (IER).

Group Priority (PIE Level):
If the Peripheral Interrupt Expansion (PIE) block is enabled, then CPU interrupts INT1 to INT12 are connected to the PIE. This peripheral expands each of these 12 CPU interrupt into 8 interrupts. Thus the total possible number of available interrupts in the PIE is 96. Note, not all of the 96 are used on a 2803x device.

Each of the PIE groups has its own interrupt enable register (PIEIERx) to control which of the 8 interrupts (INTx.1 - INTx.8) are enabled and permitted to issue an interrupt.

A.2 2803x Interrupt Priorities

The PIE block is organized such that the interrupts are in a logical order. Interrupts that typically require higher priority, are organized higher up in the table and will thus be serviced with a higher priority by default.

CPU Interrupt	PIE Group	PIE Interrupts							
		Highest——————Hardware Priority Within the Group——————-Lowest							
INT1	1	INT1.1	INT1.2	INT1.3	INT1.4	INT1.5	INT1.6	INT1.7	INT1.8
INT2	2	INT2.1	INT2.2	INT2.3	INT2.4	INT2.5	INT2.6	INT2.7	INT2.8
INT3	3	INT3.1	INT3.2	INT3.3	INT3.4	INT3.5	INT3.6	INT3.7	INT3.8
... etc ...									
... etc ...									
INT12	12	INT12.1	INT12.2	INT12.3	INT12.4	INT12.5	INT12.6	INT12.7	INT4.8

Table A.1: PIE Group Hardware Priority

The interrupts in a 2803x system can be categorized as follows (ordered highest to lowest priority):

1. **Non-Periodic, Fast Response**
   These are interrupts that can happen at any time and when they occur, they must be serviced as quickly as possible. Typically these interrupts monitor an external event.

   On the 2803x, such interrupts are allocated to the first few interrupts within PIE Group 1 and PIE Group 2. This position gives them the highest priority within the PIE group. In addition, Group 1 is multiplexed into the CPU interrupt INT1. CPU INT1 has the highest hardware priority. PIE Group 2 is multiplexed into the CPU INT2 which is the 2nd highest hardware priority.

2. **Periodic, Fast Response**
   These interrupts occur at a known period, and when they do occur, they must be serviced as quickly as possible to minimize latency. The A/D converter is one good example of this. The A/D sample must be processed with minimum latency.

   On the 2803x, such interrupts are allocated to the group 1 in the PIE table. Group 1 is multiplexed into the CPU INT1. CPU INT1 has the highest hardware priority

3. **Periodic**
   These interrupts occur at a known period and must be serviced before the next interrupt. Some of the PWM interrupts are an example of this. Many of the registers are shadowed, so the user has the full period to update the register values.

   In the 2803x PIE module, such interrupts are mapped to group 2 - group 5. These groups are multiplexed into CPU INT3 to INT5 (the ePWM and eCAP), which are the next lowest hardware priority.

4. **Periodic, Buffered**
   These interrupts occur at periodic events, but are buffered and hence the processor need only service such interrupts when the buffers are ready to filled/emptied. All of the serial ports (SCI / SPI / I2C / CAN) either have FIFOs or multiple mailboxes such that the CPU has plenty of time to respond to the events without fear of losing data.

   In the 2803x, such interrupts are mapped to INT6, INT8, and INT9, which are the next lowest hardware priority.

A.3 Software Prioritization of Interrupts - The DSP28 Example

The user will probably find that the PIE interrupts are organized where they should be for most applications. However, some software prioritization may still be required for some applications.

Recall that the basic software priority scheme on the C28x works as follows:

- **Global Priority**
  This priority can be managed by manipulating the CPU IER register. This register controls the 16 maskable CPU interrupts (INT1 - INT16).

- **Group Priority**
  This can be managed by manipulating the PIE block interrupt enable registers (PIEIERx). There is one PIEIERx per group and each control the 8-interrupts multiplexed within that group.

The DSP28 software prioritization of interrupt example demonstrates how to configure the Global priority (via IER) and group priority (via PIEIERx) within an ISR in order to change the interrupt service priority based on user assigned levels. The steps required to do this are:

1. **Set the global priority**
   Modify the IER register to allow CPU interrupts with a higher user priority to be serviced.

2. **Set the Group priority**
   Modify the appropriate PIEIERx register to allow group interrupts with a higher user set priority to be serviced.

3. **Enable interrupts**

The DSP28 software prioritized interrupts example provides a method using mask values that are configured during compile time to allow you to manage this easily.

To setup software prioritization for the DSP28 example, the user must first assign the desired global priority levels and group priority levels.

This is done in the DSP2803x_SWPrioritizedIsrLevels.h file as follows:

1. *User assigns global priority levels*
   INT1PL - INT16PL
   These values are used to assign a priority level to each of the 16 interrupts controlled by the CPU IER register. A value of 1 is the highest priority while a value of 16 is the lowest. More then one interrupt can be assigned the same priority level. In this case the default hardware priority would determine which would be serviced first. A priority of 0 is used to indicate that the interrupt is not used.

2. *User assigns PIE group priority levels*
   GxyPL (where x = PIE group number 1 - 12 and y = interrupt number 1 - 8)

   These values are used to assign a priority level to each of the 8 interrupts within a PIE group. A value of 1 is the highest priority while a value of 8 is the lowest. More then one interrupt can be assigned the same priority level. In this case the default hardware priority would determine which would be serviced first. A priority of 0 is used to indicate that the interrupt is not used.

Once the user has defined the global and group priority levels, the compiler will generate mask values that can be used to change the IER and PIEIERx registers within each ISR. In

this manner the interrupt software prioritization will be changed. The masks that are generated at compile time are:

- **IER mask values**
  MINT1 - MINT16

  The user assigned INT1PL - INT16PL values are used at compile time to calculate an IER mask for each CPU interrupt. This mask value will be used within an ISR to allow CPU interrupts with a higher priority to interrupt the current ISR and thus be serviced at a higher priority level.
- **PIEIERxy mask values**
  MGxy (where x = PIE group number 1 - 12 and y = interrupt number 1 - 8)

  The assigned group priority levels (GxyPL) are used at compile time to calculate PIEIERx masks for each PIE group. This mask value will be used within an ISR to allow interrupts within the same group that have a higher assigned priority to interrupt the current ISR and thus be serviced at a higher priority level.

A.3.1 Using the IER/PIEIER Mask Values

Within an interrupt service routine, the global and group priority can be changed by software to allow other interrupts to be serviced. The procedure for setting an interrupt priority using the mask values created in the DSP28_SWPrioritizedIsrLevels.h is the following:

1. **Set the global priority**
   - Modify IER to allow CPU interrupts from the same PIE group as the current ISR.
   - Modify IER to allow CPU interrupts with a higher user defined priority to be serviced.
2. **Set the group priority**
   - Save the current PIEIERx value to a temporary register.
   - The PIEIER register is then set to allow interrupts with a higher priority within a PIE group to be serviced.
3. **Enable interrupts**
   - Enable all PIE interrupt groups by writing all 1's to the PIEACK register
   - Enable global interrupts by clearing INTM
4. **Execute ISR.** Interrupts that were enabled in steps 1-3 (those with a higher software priority) will be allowed to interrupt the current ISR and thus be serviced first.
5. **Restore the PIEIERx register**
6. **Exit**

A.3.2 Example Code

The sample C code below shows an EV-A Comparator 1 Interrupt service routine software prioritization written in C. This interrupt is connected to PIE group 2 interrupt 1.

```
// Connected to PIEIER2_1 (use MINT2 and MG21 masks):
#if (G21PL != 0)
interrupt void EPWM1_TZINT_ISR(void)   // EPWM1 Trip Zone
{
    // Set interrupt priority:
    volatile Uint16 TempPIEIER = PieCtrlRegs.PIEIER2.all;
```

```
    IER |= M_INT2;
    IER &= MINT2;                    // Set "global" priority
    PieCtrlRegs.PIEIER2.all &= MG21;   // Set "group"  priority
    PieCtrlRegs.PIEACK.all = 0xFFFF;   // Enable PIE interrupts
    EINT;

    // Insert ISR Code here.......
    // for now just insert a delay
    for(i = 1; i <= 10; i++) {}

    // Restore registers saved:
    DINT;
    PieCtrlRegs.PIEIER2.all = TempPIEIER;

    //  Add ISR to Trace
    ISRTrace[ISRTraceIndex] = 0x0021;
    ISRTraceIndex++;
}
#endif

CMP1INT_ISR:
            ASP
            ADDB    SP,#1
            CLRC    OVM,PAGE0
            MOVW    DP,#0x0033
            MOV     AL,@36
            MOV     *-SP[1],AL
            OR      IER,#0x0002
            AND     IER,#0x0002
            AND     @36,#0x000E
            MOV     @33,#0xFFFF
            CLRC    INTM

            User code goes here...

            SETC    INTM
            MOV     AL,*-SP[1]
            MOV     @36,AL
            SUBB    SP,#1
            NASP
            IRET
```

The interrupt latency is approx 22 cycles.
/*!

B Internal Oscillator Compensation Functions

B.1 Introduction

To compensate the internal oscillator, the Texas Instruments factory takes measurements of the internal oscillator and temperature sensor. It then calculates a reference point for the temperature sensor and oscillator trim and calculates an oscillator trim slope. The trim slope can be used to adjust the oscillator fine trim as the temperature sensor reading moves away from that of the reference point.

The reference point for the internal oscillator consists of two pieces of data. The first is the temperature sensor reading at that point. The second is the oscillator trim values to get 10.0MHz at that temperature. This trim itself is composed of two parts: the fine trim and the coarse trim. Only the fine trim will be adjusted by the compensation procedure. The coarse trim remains the same no matter what temperature the device is at.

The oscillator compensation slope contains the information needed to adjust the oscillator fine trim from the reference fine trim as the temperature moves away from the reference temperature. This slope has the units of oscillator fine trim steps / ADC codes (temperature sensor output).

If X is considered to be the temperature sensor reading and Y is considered to be the oscillator fine trim, then the basic oscillator compensation equation is

$$Y_1 = m * (X_1 - X_0) + Y_0 \tag{B.1}$$

where,
Y_1 is the oscillator fine trim at the current temperature
Y_0 is the oscillator fine trim at the reference temperature
X_1 is the temperature sensor reading at the current temperature
X_0 is the temperature sensor reading at the reference temperature
m is the oscillator compensation slope, which is $\frac{change\ in\ oscillator\ fine\ trim}{change\ in\ temperature\ sensor\ reading}$

This is equivalent to a line with equation $Y = mX + b$:

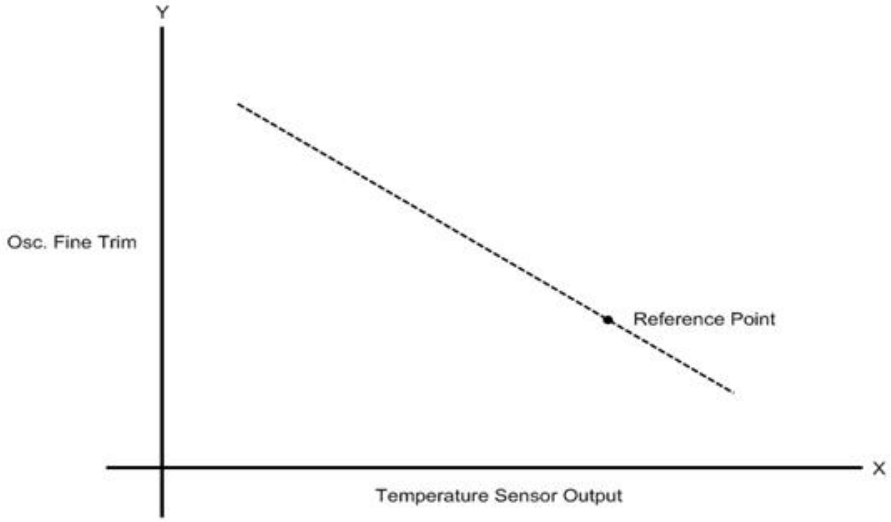

Figure B.1: Oscillator Reference

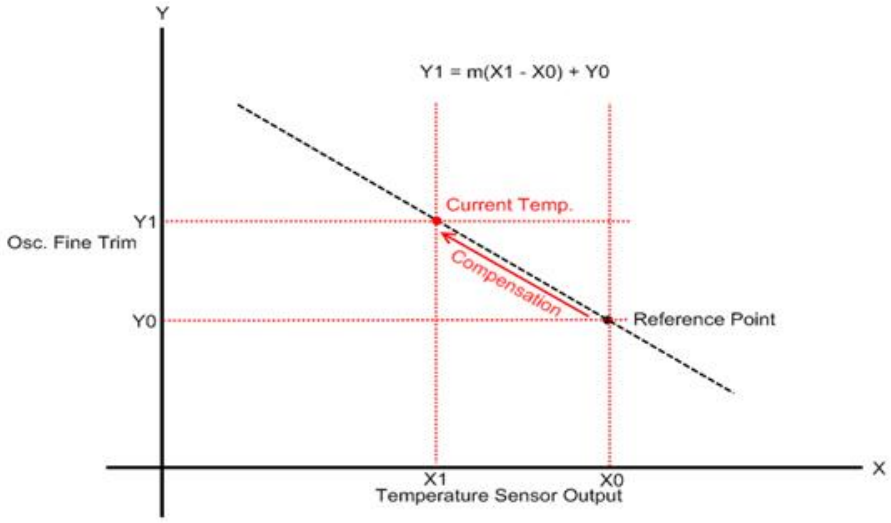

Figure B.2: Oscillator Fine Trim Compensation for change in Temperature

B.2 Oscillator Compensation Functions Available in the Header Files and Peripheral Examples Package

B.2.1 OTP Functions

The following functions in *DSP<Device>_OscComp.c* are programmed in OTP and return variables stored in OTP used for oscillator compensation.

Function Call: getRefTempOffset()
OTP address: 0x3D7EA2
Returns: Reference Temperature Offset
This is the temperature sensor reading of the reference point for oscillator compensation.

Function Call: getOsc1FineTrimOffset()
OTP address: 0x3D7E93
Returns: Oscillator 1 Fine Trim Offset
This is the fine trim of the reference point for oscillator 1. This is the fine trim required to get 10.0MHz when the temperature sensor reads the value of “High Temperature Offset”.

Function Call: getRefTempOffset()
OTP address: 0x3D7EA2
Returns: Reference Temperature Offset

Function Call: getOsc2FineTrimOffset ()
OTP address: 0x3D7E9C
Returns: Oscillator 2 Fine Trim Offset
This is the fine trim of the reference point for oscillator 2. This is the fine trim required to get 10.0MHz when the temperature sensor reads the value of “High Temperature Offset”.

Function Call: getOsc1FineTrimSlope()
OTP address: 0x3D7E90
Returns: Oscillator 1 Fine Trim Slope
This is the slope of the oscillator temperature characteristic determined by the factory for internal oscillator 1. Units are oscillator fine trim steps / ADC codes (temperature sensor output). This variable is stored as a Q0.15 fixed point number - e.g. if the slope = -0.04, then this value is stored as -0.04*(215) = -1311. Note that this will require us to use fixed point math to compensate the oscillator.

Function Call: getOsc2FineTrimSlope()
OTP address: 0x3D7E99
Returns: Oscillator 2 Fine Trim Slope
This is the slope of the oscillator temperature characteristic determined by the factory for internal oscillator 2. Units are oscillator fine trim steps / ADC codes (temperature sensor output). This variable is stored as a Q0.15 fixed point number - e.g. if the slope = -0.04, then this value is stored as -0.04*(215) = -1311. Note that this will require us to use fixed point math to compensate the oscillator.

Function Call: getOsc1CoarseTrim()
OTP address: 0x3D7E96
Returns: Oscillator 1 Coarse Trim

This is the coarse trim to always use for oscillator 1 when doing oscillator compensation.

Function Call: getOsc2CoarseTrim()
OTP address: 0x3D7E9F
Returns: Oscillator 2 Coarse Trim
This is the coarse trim to always use for oscillator 2 when doing oscillator compensation.

B.2.2 Oscillator Compensation User Functions

The following functions use the ADC temperature sensor sample as a parameter and update the internal oscillator coarse and fine trim value while compensating for temperature. These functions can be called directly via user application code.

Function Call: Osc1Comp(int16 sensorSample)
This function uses the temperature sensor sample reading to perform internal oscillator 1 compensation with reference values stored in OTP.

Function Call: Osc2Comp(int16 sensorSample)
This function uses the temperature sensor sample reading to perform internal oscillator 2 compensation with reference values stored in OTP.

C Scale Factor Optimization (SFO) V6 Library Errata

C.1 Introduction

This document describes the updates to the Scale Factor Optimization (SFO) V6 library files packaged with the C/C++ Header Files and Peripheral Examples software package.

The updates are applicable to the following files:

- SFO_TI_Build_V6x.lib (where “x” represents the alphabetical revision letter of the library).
- SFO_V6.h

C.2 Library Change Overview

Table C.1 lists the change(s) made to each library revision.

REVISION	CHANGES MADE
0	Original library release
b	**Changes:** Library re-released as SFO_TI_Build_V6b.lib **Errors Fixed:** The SFO_V6() function now properly updates the HRMSTEP register with MEP_ScaleFactor after calibration is complete.

Table C.1: SFO Library V6 Change Overview

C.3 Known Advisories in Library Versions

Table C.2 lists the known advisories in early library revisions and workarounds required in user code to account for these errors.

Advisory	HRMSTEP Register not Updated with calculated MEP_ScaleFactor
Revision(s) Affected	0
Details	The library function, SFO_V6(), does not properly update the HRMSTEP register with the newly calculated MEP_ScaleFactor value after SFO_COMPLETE status has been reached.

Workaround	After SFO_COMPLETE status has been reached on a call to SFO_V6(), user must manually update the HRMSTEP register with the newly calculated MEP_ScaleFactor Value.
	Example:
	`if (SFO_V6(n)==SFO_COMPLETE) {`
	`  EALLOW;`
	`  EPwm1Regs.HRMSTEP = MEP_ScaleFactor;`
	`  EDIS;`
	`}`

Table C.2: SFO V6 Library Advisories in Early Software Revisions

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

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

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Broadband	www.ti.com/broadband
DSP	dsp.ti.com	Digital Control	www.ti.com/digitalcontrol
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Military	www.ti.com/military
Logic	logic.ti.com	Optical Networking	www.ti.com/opticalnetwork
Power Mgmt	power.ti.com	Security	www.ti.com/security
Microcontrollers	microcontroller.ti.com	Telephony	www.ti.com/telephony
RFID	www.ti-rfid.com	Video & Imaging	www.ti.com/video
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265