F2805x Firmware Development Package

USER'S GUIDE

Copyright

Texas Instruments
13905 University Boulevard
Sugar Land, TX 77479
http://www.ti.com/c2000

Revision Information

This is version 2.05.00.00 of this document, last updated on Fri Nov 17 18:41:44 IST 2023.

Table of Contents

1 Introduction

The Texas Instruments® F2805x Firmware Development Package is a collection of device header files, common source files, helper libraries and example applications for the 2805x 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. It's 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 (2805x) can be found in the *f2805x\examples\c28* directory. As users move past evaluation, and get started developing their own application, TI recommends users maintain a similar project directory structure to that used in the example projects.
Chapter 5 provides details on the prototype CLA compiler including implementation of the C language as well as restrictions.
Chapter 6 covers the examples using the CLA compiler. The examples can be found in the *f2805x\examples\cla* directory.

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.

2 Header File Quickstart

2.1 Device Support

This software package supports 2805x devices. This includes the following: TMS320F28055, TMS320F28054, TMS320F28054m, TMS320F28054f, TMS320F28053, TMS320F28052, TMS320F28052m, TMS320F28052f, TMS320F28051, and TMS320F28050. Throughout this document, TMS320F28055, TMS320F28054, TMS320F28054m, TMS320F28054f, TMS320F28053, TMS320F28052, TMS320F28052m, TMS320F28052f, TMS320F28051, and TMS320F28050 are abbreviated as F28055, F28054, F28054m, F28054f, F28053, F28052, F28052m, F28052f, F28051, and F28050 respectively.

2.2 Introduction

The 2805x C/C++ peripheral header files and example projects facilitate writing in C/C++ Code for the Texas Instruments TMS320x2805x 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 5.1+ [1] projects for a 2805x 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] Code Composer Studio is a trademark of Texas Instruments (www.ti.com).

1. Overview of the bit-field structure approach used in the 2805x C/C++ peripheral header files.
2. Overview of the included peripheral example projects.
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 2805x 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 2805x 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 2.01.00.00**
   - Fixed issues in examples and source code. (Details in section 2.9)
2. **Version 2.00.00.00**
   - F2805x package updated and enhanced for C2000Ware (Details in section 2.9)
3. **Version 1.04**
   - Added PGA compensation example and cleaned up all example projects.
4. **Version 1.03**
   - Fixed bugs in example, source, and header code. Cleaned up device examples.
5. **Version 1.02**
   - Added F28055 Flash Kernel Example
6. **Version 1.01**
   - Bug fixes from previous version. Fixed bugs in example codes and header files.
7. **Version 1.00**
   - This version is the first release of the 2805x header files and examples.

2.2.2 Directory Structure

As installed, the 2805x C/C++ Header Files and Peripheral Examples are partitioned into a well-defined directory structure (see Figure 2.1).

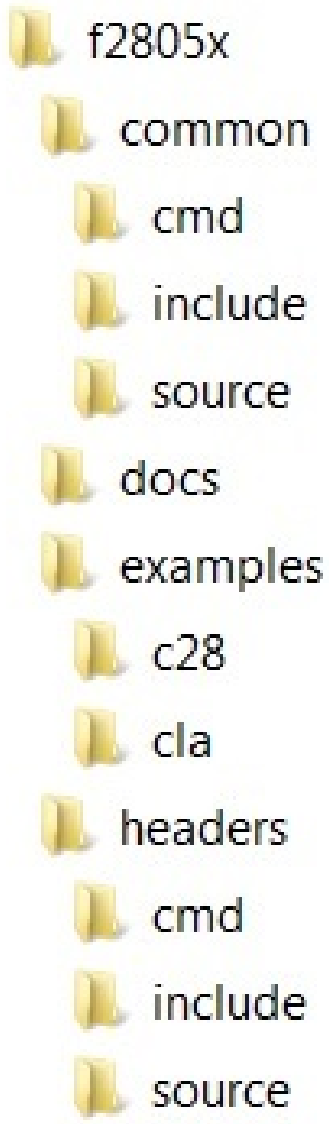

Figure 2.1: F2805x Main Directory Structure

Table 2.1 describes the contents of the main directories used by 2805x header files and peripheral examples.

Under the f2805x\headers and f2805x\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:

Directory	Description
<base>	Base install directory
<base>\docs	Documentation including the revision history from the previous release.
<base>\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>\examples\c28	Example Code Composer Studio v5 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>\examples\cla	CLA examples(Built with CLA C compiler). These example projects illustrate the programming model of the CLA in C. An overview of the examples is given in Chapter 6.
<base>\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: F2805x Main Directory Structure

Sub-Directory	Description
f2805x\headers\cmd	Linker command files that allocate the bit-field structures described in Section 2.3.
f2805x\headers\source	Source files required to incorporate the header files into a new or existing project.
f2805x\headers\include	Header files for each of the on-chip peripherals.
f2805x\common\cmd	Example memory command files that allocate memory on the devices.
f2805x\common\include	Common .h files that are used by the peripheral examples.
f2805x\common\source	Common .c files that are used by the peripheral examples.

Table 2.2: F2805x Sub-Directory Structure

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)

2.4 Peripheral Example Projects

This section describes how to get started with and configure the peripheral examples included in the 2805x Header Files and Peripheral Examples software package.

2.4.1 Getting Started in Code Composer Studio v5.1+

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 2805x example projects are built for the 28055 device. If you are using another 2805x 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 v5. To open the 2805x CPU-Timer example project directory, follow the following steps:
   - In Code Composer Studio v 5.x: Project->Import Existing CCS/CCE Eclipse Project.
   - Next to "Select Root Directory", browse to the CPU Timer example directory: f2805x\examples\c28\cpu_timer. Select the Finish button. This will import/open the project in the CCStudio v5 C/C++ Perspective project window.
3. **Edit F28_Device.h** Edit the F2805x_Device.h file and make sure the appropriate device is selected. By default the 28055 is selected.

```
/*-----------------------------------------------------------------------
 f2805x\headers\include\F2805x_Device.h
-----------------------------------------------------------------------*/

#define   TARGET   1
//---------------------------------------------------------------------------
// User To Select Target Device:

#define   DSP28_28050PN    0

#define   DSP28_28051PN    0

#define   DSP28_28052PN    0
#define   DSP28_28052FPN   0
#define   DSP28_28052MPN   0

#define   DSP28_28053PN    0

#define   DSP28_28054PN    0
#define   DSP28_28054FPN   0
#define   DSP28_28054MPN   0

#define   DSP28_28055PN    TARGET
```

4. **Edit F2805x_Examples.h** Edit F2805x_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.

```
/*************************************************************
 f2805x\common\include\F2805x_Examples.h
*************************************************************/
/*-------------------------------------------------------------------
 Specify the PLL control register (PLLCR) and divide select (DIVSEL) value.
-------------------------------------------------------------------*/

//#define DSP28_DIVSEL   0 // Enable /4 for SYSCLKOUT
//#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        // Uncomment for 60 MHz devices
    // [60 MHz = (10MHz * 12)/2]
//#define DSP28_PLLCR   11
//#define DSP28_PLLCR   10
//#define DSP28_PLLCR    9
//#define DSP28_PLLCR    8      // Uncomment for 40 MHz devices
    // [40 MHz = (10MHz * 8)/2]
//#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  // PLL is bypassed in this mode
//-------------------------------------------------------------------
```

   In F2805x_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.

```
/*************************************************************
 f2805x\common\include\F2805x_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_2805xCpuTimer.c** A brief description of the example, 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 2805x boot modes refer to the device specific *Boot ROM Reference Guide*.

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: 2805x 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	Wait
	0x0007	eCAN
	0x0008	Wait
	0x000A	Boot to RAM
	0x000B	Boot to FLASH
	Other	Wait

Table 2.4: 2805x EMU Boot Modes (Emulator Connected)

OTP_KEY	**OTP_BMODE**	**Boot Mode Selected**
!= 0x55AA	**x**	**Get Mode - Flash**
	Z2 OTP_BKEY = 0x55AA	**Z2 BOOTMODE OTP_BMODE**
0x55AA	0x0001	Get Mode - SCI
	0x0003	**Get Mode - Flash**
	0x0004	Get Mode - SPI
	0x0005	Get Mode - I2C
	0x0006	Get Mode - Flash
	0x0007	Get Mode - eCAN
	Other	**Get Mode - Flash**

Table 2.5: 2805x GET Boot Modes (Emulator Disconnected)

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 2805x gel files in the F2805x_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.

When the emulator is not connected for debugging: SCI or Parallel I/O boot mode can be selected directly via the GPIO pins, or Z1 or Z2 DCSM BOOTMODE[OTP_KEY] and Z1 or Z2 DCSM BOOTMODE[OTP_BMODE] values can be loaded from Z1 or Z2 OTP 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 v5, go to *Run->Debug Project*.

   This will open the “Debug Perspective” in CCSv5, 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 v5, every time Debug Project is selected, the code is automatically built and the .out file of the project highlighted in the Project Explorer window is loaded the 28x device.

8. **Run the example, then add variables to the expressions 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 expressions window, highlight them and right-click. Then select *Add Watch expression*. Now variables of interest are added to the expressions window. Another way to add the variable to the expressions window is the select the entire variable using your mouse and drag and drop it onto an existing expressions 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 v5 workspace**

    Go to *Window->Open Perspective->CCS Edit* to open up your project view. There are also icons on the toolbar in the upper right-hand corner labeled "CCS Debug" and "CCS Edit" to allow you to switch between perspectives faster. To remove/delete the project from the workspace, right click on the project’s name and select delete. Make sure the *Delete contents on disk (cannot be undone)* checkbox is not selected, then click *OK*. 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 f2805x\headers directory and shared source in the f2805x\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

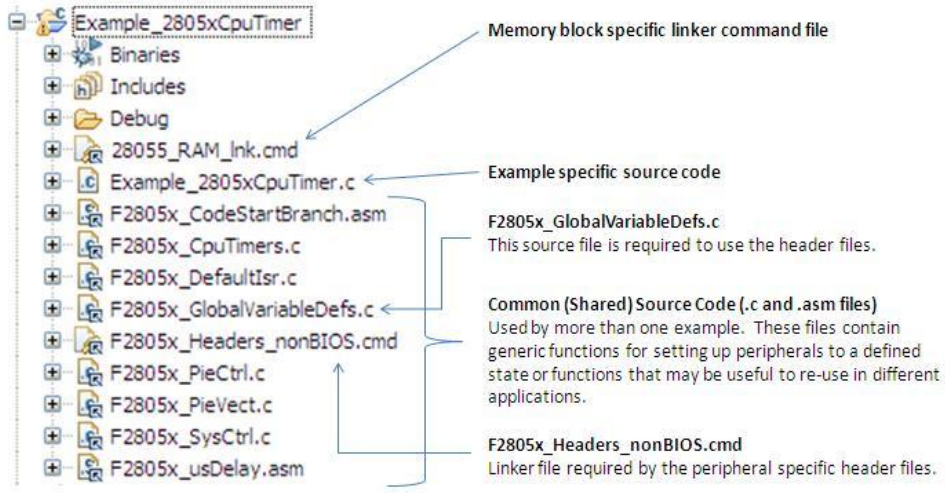

Figure 2.2: Example Program Structure

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.

```
//###########################################################################
// f2805x\examples\c28\cpu_timer\Example_2805xCpuTimer.c
//###########################################################################

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

- **DSP28x_Project.h**
  This header file includes F2805x_Device.h and F2805x_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>\common\include directory.
- **F2805x_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>\headers\include directory.
- **F2805x_Examples.h**
  This header file defines parameters that are used by the example code. This file is not required to use just the F2805x peripheral header files but is required by some of the common source files. This file is found in the <base>\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.

- **F2805x_GlobalVariableDefs.c**

  Any project that uses the F2805x 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>\headers\source directory.

- **Example specific source code**

  Files that are specific to a particular example have the prefix Example_2805x in their filename. For example Example_2805xCpuTimer.c is specific to the CPU Timer example and not used for any other example. Example specific files are located in the <base>\examples\c28\<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 f2805x\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 F2805x 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>\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
28055_RAM_lnk.cmd	f2805x\common\cmd	28055 memory linker command file. Includes all of the internal SARAM blocks on 28055 device. “RAM” linker files do not include flash or OTP blocks.
28054_RAM_lnk.cmd	f2805x\common\cmd	28054 SARAM memory linker command file.
28054m_RAM_lnk.cmd	f2805x\common\cmd	28054m SARAM memory linker command file.
28054f_RAM_lnk.cmd	f2805x\common\cmd	28054f SARAM memory linker command file.
28053_RAM_lnk.cmd	f2805x\common\cmd	28053 SARAM memory linker command file.
28052_RAM_lnk.cmd	f2805x\common\cmd	28052 SARAM memory linker command file.
28052m_RAM_lnk.cmd	f2805x\common\cmd	28052m SARAM memory linker command file.
28052f_RAM_lnk.cmd	f2805x\common\cmd	28052f SARAM memory linker command file.
28051_RAM_lnk.cmd	f2805x\common\cmd	28051 SARAM memory linker command file.
28050_RAM_lnk.cmd	f2805x\common\cmd	28050 SARAM memory linker command file
28055_RAM_CLA_lnk.cmd	f2805x\common\cmd	28055 SARAM CLA memory linker command file. Includes CLA message RAM.
28053_RAM_CLA_lnk.cmd	f2805x\common\cmd	28053 SARAM CLA memory linker command file.
F28055.cmd	f2805x\common\cmd	F28055 memory linker command file. Includes all Flash, OTP and DCSM password protected memory locations.
F28054.cmd	f2805x\common\cmd	F28054 memory linker command file.
F28054m.cmd	f2805x\common\cmd	F28054m memory linker command file.
F28054f.cmd	f2805x\common\cmd	F28054f memory linker command file.
F28053.cmd	f2805x\common\cmd	F28053 memory linker command file.
F28052.cmd	f2805x\common\cmd	F28052 memory linker command file.
F28052m.cmd	f2805x\common\cmd	F28052m memory linker command file.
F28052f.cmd	f2805x\common\cmd	F28052f memory linker command file.
F28051.cmd	f2805x\common\cmd	F28051 memory linker command file.
F28050.cmd	f2805x\common\cmd	F28050 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.

Header File Linker Command File	Location	Description
F2805x_Headers_BIOS.cmd	f2805x\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.
F2805x_Headers_nonBIOS.cmd	f2805x\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: F2805x Peripheral Header Linker Command File

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 2805x device.

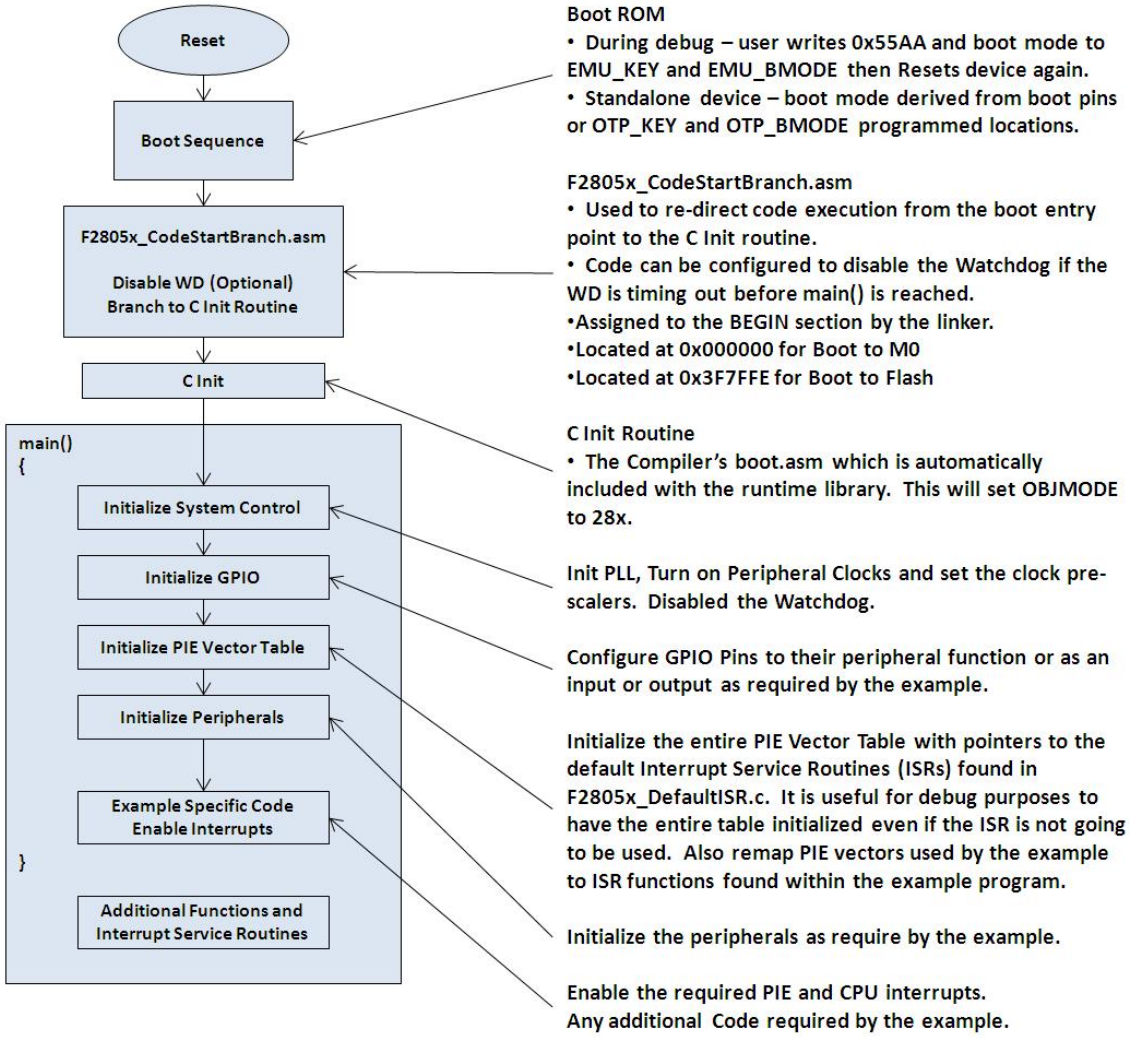

Figure 2.3: Flow for Example Programs

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 F2805x examples execute from SARAM in "boot to SARAM" mode. One example, f2805x\examples\c28\flash_f28055, 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 28055_RAM_lnk.cmd from the project and link the flash based linker file for your device (ex: F28055.cmd, F28054.cmd, F28054m.cmd, F28054f.cmd, F28053.cmd, F28052.cmd,

F28052m.cmd, F28052f.cmd, F28051.cmd, or F28050.cmd). These files are located in the <base>\common\cmd directory.

2. **Link the f2805x\common\source\F2805x_DCSM_Z1_ZoneSelectBlock.asm and the f2805x\common\source\F2805x_DCSM_Z2_ZoneSelectBlock.asm to the project**
   These files contain the Zone 1 and Zone 2 security zone select values respectively (including link pointer, zone sector definition, and passwords) that will be programed into the Zone 1 or Zone 2 DCSM OTP locations. Leaving the passwords set to 0xFFFF during development is recommended as the device can easily be unlocked. For more information on the DCSM 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 F2805x, 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.

```
//###########################################################################
// f2805x\common\source\F2805x_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:

```
//###########################################################################
// f2805x\common\cmd\F28055.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:

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

Table 2.8: Linker Symbol assignment

These symbols can then be used to copy the functions from the Flash to SARAM using the C library standard memcpy() function.

To perform this copy from flash to SARAM using the C library standard memcopy() function:

```
//###########################################################################
// f2805x\common\include\F2805x_Examples.h
```

```
*************************************************************************/

memcpy(&RamfuncsRunStart,&RamfuncsLoadStart, (Uint32)&RamfuncsLoadSize);
```

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.

4. 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 F2805x example code this has is already done in F2805x_Examples.h.

```
/*************************************************************************
 f2805x\common\include\F2805x_GlobalPrototypes.h
*************************************************************************/

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

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

```
/*************************************************************************
 F2805x_examples\Flash source file
*************************************************************************/

memcpy(&RamfuncsRunStart,&RamfuncsLoadStart, (Uint32)&RamfuncsLoadSize);
```

6. **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.

```
/*************************************************************************
 F2805x peripheral example .c file
*************************************************************************/

InitFlash();
```

- **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: 2805x 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: 2805x EMU Boot Modes (Emulator Connected)

OTP_KEY	OTP_BMODE	Boot Mode Selected
!= 0x55AA	**x**	**Get Mode - Flash**
	Z2 OTP_BKEY = 0x55AA	**Z2 BOOTMODE OTP_BMODE**
0x55AA	0x0001	Get Mode - SCI
	0x0003	**Get Mode - Flash**
	0x0004	Get Mode - SPI
	0x0005	Get Mode - I2C
	0x0006	Get Mode - Flash
	0x0007	Get Mode - eCAN
	Other	**Get Mode - Flash**

Table 2.11: 2805x GET Boot Modes (Emulator Disconnected)

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 and OTP_BMODEcan 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 2805x boot modes refer to the appropriate *Boot ROM Reference Guide*.

- **Program the device with the built code**

  In Code Composer Studio v5, 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 v5.x can be used.

  These tools will be updated to support new devices as they become available. Please check for updates.

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 F280x or F281x 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**
   f2805x\headers: code required to incorporate the header files into your project *f2805x\common*: shared source code much of which is used in the example projects. *f2805x\examples\c28*: F2805x example projects that use the header files and shared code.

2.5.2 Including the F2805x Peripheral Header Files

Including the F2805x header files in your project will allow you to use the bit-field structure approach in your code to access the peripherals on the F. To incorporate the header files in a new or existing project, perform the following steps:

1. **#include “F2805x_Device.h” (or #include “DSP28x_Project.h”) in your source files**
   The F2805x_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 "F2805x_Device.h"
```

   Another option is to #include “DSP28x_Project.h” in your source files, which in-turn includes “F2805x_Device.h” and “F2805x_Examples.h” (if it is not necessary to include common source files in the user project, the #include “F2805x_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 F2805x_Device.h and select the target you are building for**
   In the below example, the file is configured to build for the 28055 device.

```
//---------------------------------------------------------------------------
// f2805x\headers\include\F2805x_Device.h
//---------------------------------------------------------------------------
#define   TARGET   1
//---------------------------------------------------------------------------
// User To Select Target Device:

#define   DSP28_28050PN    0

#define   DSP28_28051PN    0

#define   DSP28_28052PN    0
#define   DSP28_28052FPN   0
#define   DSP28_28052MPN   0

#define   DSP28_28053PN    0

#define   DSP28_28054PN    0
#define   DSP28_28054FPN   0
#define   DSP28_28054MPN   0

#define   DSP28_28055PN    TARGET
```

   By default, the 28055 device is selected.

3. **Add the source file F2805x_GlobalVariableDefs.c to the project**
   This file is found in the f2805x\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 F2805x header linker command file to the project.** As described in Section 2.4, when using the F2805x 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 f2805x\headers\cmd must be included in your project:
   - For non-SYS/BIOS[2] projects: *F2805x__nonBIOS.cmd*
   - For SYS/BIOS projects: *F2805x_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 Add Files.
   - Navigate to the f2805x\headers\cmd directory on your system and select the desired .cmd file and click Open.
   - In the following window, select the Copy Files option.

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

Note: The limitation with Method #1 is that the path to <install directory>\f2805x\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_CCSv5_for_C2000

- Right-click on the project in the project window of the C/C++ Projects perspective.
- Select Add Files.
- Navigate to the f2805x\headers\cmd directory on your system and select the desired .cmd file and click Open.
- In the following window, select the Link to Files option.

5. **Add the directory path to the F2805x header files to your project**
   Code Composer Studio 5.x:
   To specify the directory where the header files are located:

   - Open the menu: Project->Properties.
   - In the menu on the left, expand "CCS Build".
   - Under "CCS Build", expand "C2000 Compiler".
   - 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 f2805x\headers\include on your system.

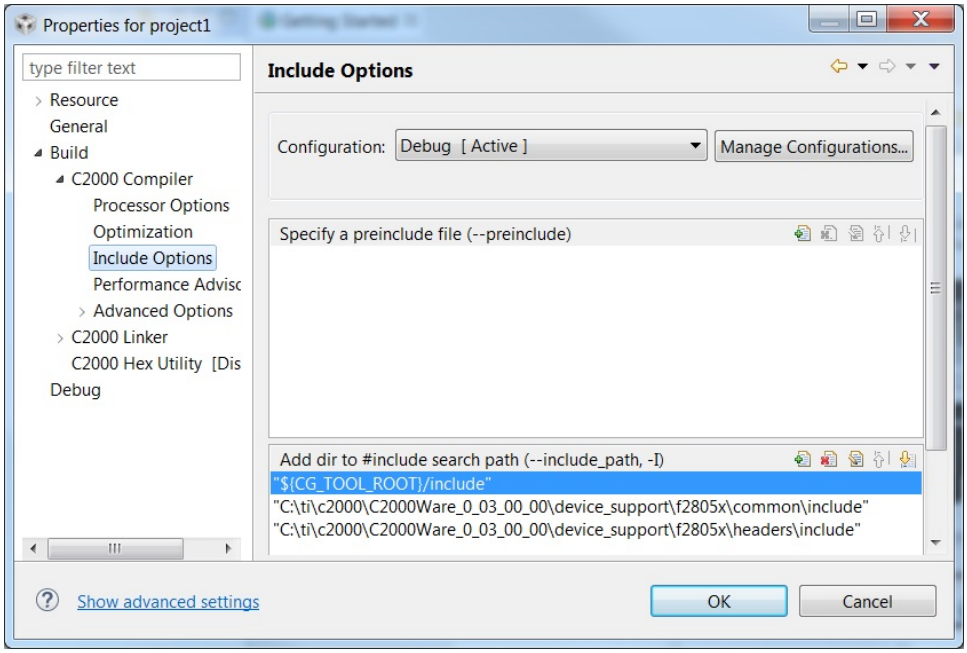

Figure 2.4: Adding device header file directories to the include search path

6. **Additional suggested build options** The following are additional compiler and linker options. The options can all be set via the Project->Properties->CCS Build 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 F2805x examples, this is the symbol “code_start”. This symbol is defined in the f2805x\common\source\F2805x_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 “F2805x_Examples.h” (or “DSP28x_Project.h”) in your source files.**
   The “F2805x_Examples.h” include file will include common definitions and declarations used by the example code.

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

#include "F2805x_Examples.h"
```

   Another option is to #include “DSP28x_Project.h” in your source files, which in-turn includes “F2805x_Device.h” and “F2805x_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, expand “CCS Build”.
   - Under “CCS Build”, expand “C2000 Compiler”.
   - 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 f2805x\headers\include on your system.

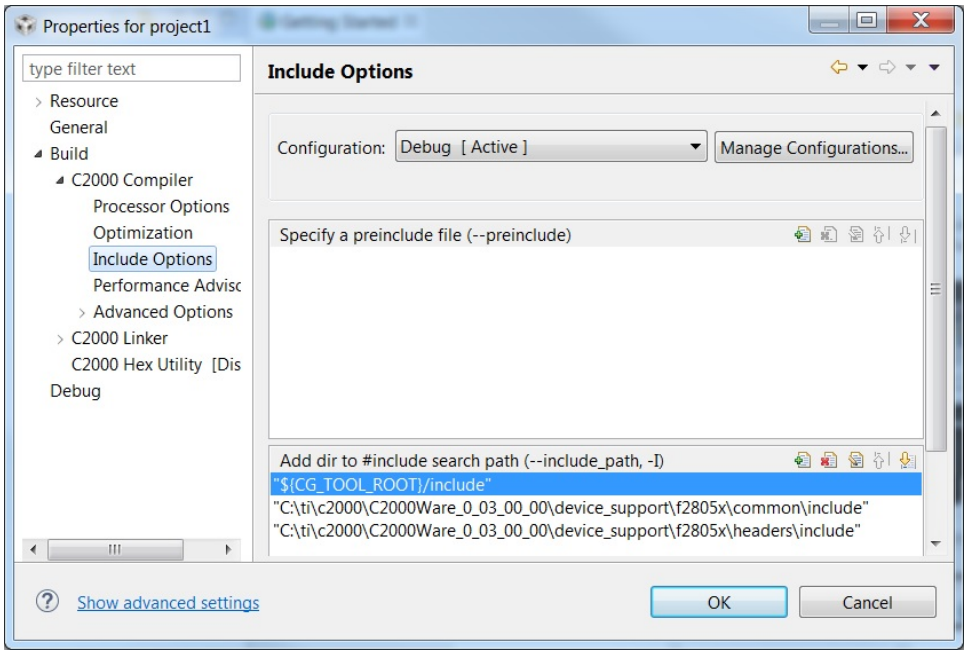

Figure 2.5: Adding Example header directories to the include search path

3. **Link a linker command file to your project.**
   The following memory linker .cmd files are provided as examples in the f2805x\common\cmd directory. For getting started the basic 28055_RAM_lnk.cmd file is suggested and used by most of the examples.

Memory Linker Command File Examples	Location	Description
28055_RAM_lnk.cmd	f2805x\common\cmd	28055 memory linker command file. Includes all of the internal SARAM blocks on a 28055 device. “RAM” linker files do not include flash or OTP blocks.
28054_RAM_lnk.cmd	f2805x\common\cmd	28054 SARAM memory linker command file.
28054m_RAM_lnk.cmd	f2805x\common\cmd	28054m SARAM memory linker command file.

Continued on next page

Table 2.12 – continued from previous page

Memory Linker Command File Examples	Location	Description
28054f_RAM_lnk.cmd	f2805x\common\cmd	28054f SARAM memory linker command file.
28053_RAM_lnk.cmd	f2805x\common\cmd	28053 SARAM memory linker command file.
28052_RAM_lnk.cmd	f2805x\common\cmd	28052 SARAM memory linker command file.
28052m_RAM_lnk.cmd	f2805x\common\cmd	28052m SARAM memory linker command file.
28052f_RAM_lnk.cmd	f2805x\common\cmd	28052f SARAM memory linker command file.
28051_RAM_lnk.cmd	f2805x\common\cmd	28051 SARAM memory linker command file.
28050_RAM_lnk.cmd	f2805x\common\cmd	28050 SARAM memory linker command file.
28055_RAM_CLA_lnk.cmd	f2805x\common\cmd	28055 CLA memory linker command file. Includes CLA message RAM
28053_RAM_CLA_lnk.cmd	f2805x\common\cmd	28053 SARAM CLA memory linker command file.
F28055.cmd	f2805x\common\cmd	F28055 memory linker command file.
F28054.cmd	f2805x\common\cmd	F28054 memory linker command file.
F28054m.cmd	f2805x\common\cmd	F28054m memory linker command file.
F28054f.cmd	f2805x\common\cmd	F28054f memory linker command file.
F28053.cmd	f2805x\common\cmd	F28053 memory linker command file.
F28052.cmd	f2805x\common\cmd	F28052 memory linker command file.
F28052m.cmd	f2805x\common\cmd	F28052m memory linker command file.
F28052f.cmd	f2805x\common\cmd	F28052f memory linker command file.
F28051.cmd	f2805x\common\cmd	F28051 memory linker command file.
F28050.cmd	f2805x\common\cmd	F28050 memory linker command file.

Table 2.12: Included Main Linker Command Files

4. **Set the CPU Frequency** In the f2805x\common\include\F2805x_Examples.h file specify the proper CPU frequency. Some examples are included in the file.

```
/*************************************************************
 f2805x\common\include\F2805x_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 f2805x\common\source directory.
6. **Include .c files for the PIE** Since all catalog 2805x 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 F2805x_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 *TMS320x2805x 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 f2805x\common\source directory shows an example of enabling the peripheral clocks.
  - Some peripherals are not present on all 2805x 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 *TMS320x2805x 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 dual code security module is locked and thus the protected memory locations are reading back all 0x0000. Refer to the *TMS320x2805x System Control and Interrupts Reference Guide* for information on the dual code security module.
- **Code cannot write to L0 or L1 memory blocks**
  In this case most likely the dual 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 DCSM is locked. Refer to the *TMS320x2805x Control and Interrupts Reference Guide* for information on the dual 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 *TMS320x2805x 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 F2805x_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 F2805x 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-> Build 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 F2805x_examples_ccsv5\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 TMS320x2805x header files

This section includes suggestions for moving a project from the 280x header files to the 2805x 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 v5.x:
   Open the .cproject and .cdtbuild in your example folder. Replace all instances of 280x with 2805x 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_2805X_V100 to migrate to the 2805x header files V1.00(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 .cproject and .cdtbuild files.
3. **Load the project into Code Composer Studio**
   Use the Edit->Find dialog to find instances of F280x_Device.h and F280x_Example.h for 280x header files. Replace these with F2805x_Device.h and F2805x_Example.h respectively (or instead with one F2805x_Project.h file).
4. **Make sure you are using the correct linker command files (.cmd) appropriate for your device and for the F2805x 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 2805x.
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 F280x are replaced with F2805x 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 2805x 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 F280x V1.60 to F2805x V1.00

Additionally, unlike the F280x devices, the F2805x 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 - f2805x\headers

The F2805x header files are located in the <base>\headers directory.

2.8.1.1 F2805x Header Files - Main Files

The files listed in Table 2.14 must be added to any project that uses the F2805x header files. Refer to section 2.5 for information on incorporating the header files into a new or existing project.

File	Location	Description
F2805x_Device.h	f2805x\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.
F2805x_GlobalVariableDefs.c	f2805x\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.
F2805x_Headers_nonBIOS.cmd	f2805x\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: F2805x Header Files - Main Files

2.8.1.2 F2805x 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 2805x devices. These files are automatically included in the project by including F2805x_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
F2805x_Adc.h	f2805x\headers\include	ADC register structure and bit-field definitions.
F2805x_AnalogSubsys.h	f2805x\headers\include	PGA and Comparator register structure and bit-field definitions.
F2805x_BootVars.h	f2805x\headers\include	External boot variable definitions.
F2805x_Cla.h	f2805x\headers\include	CLA register structure and bit-field definitions
F2805x_CpuTimers.h	f2805x\headers\include	CPU-Timer register structure and bit-field definitions.
F2805x_Dcsm.h	f2805x\headers\include	DCSM register structure and bit-field definitions.
F2805x_DevEmu.h	f2805x\headers\include	Emulation register definitions
F2833x_ECan.h	f2805x\headers\include	eCAN register structures and bit-field definitions.
F2805x_ECap.h	f2805x\headers\include	eCAP register structures and bit-field definitions.
F2805x_EPwm.h	f2805x\headers\include	ePWM register structures and bit-field definitions.
F2833x_EQep.h	f2805x\headers\include	eQEP register structures and bit-field definitions.
F2805x_Gpio.h	f2805x\headers\include	General Purpose I/O (GPIO) register structures and bit-field definitions.
F2805x_I2c.h	f2805x\headers\include	I2C register structure and bit-field definitions.
F2805x_NmiIntrupt.h	f2805x\headers\include	NMI interrupt register structure and bit-field definitions
F2805x_PieCtrl.h	f2805x\headers\include	PIE control register structure and bit-field definitions.
F2805x_PieVect.h	f2805x\headers\include	Structure definition for the entire PIE vector table.
F2805x_Sci.h	f2805x\headers\include	SCI register structure and bit-field definitions.
F2805x_Spi.h	f2805x\headers\include	SPI register structure and bit-field definitions.
F2805x_SysCtrl.h	f2805x\headers\include	System register definitions. Includes Watchdog, PLL, Flash/OTP, Clock registers.
F2805x_XIntrupt.h	f2805x\headers\include	External interrupt register structure and bit-field definitions.

Table 2.15: F2805x 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 f2805x\headers\source\F2805x_GlobalVariableDefs.c file as shown in Table 2.16. Note that all peripherals may not be available on a particular 2805x device. Refer to the device datasheet for the peripheral mix available on each 2805x family derivative.

Peripheral	Starting Address	Structure Variable Name
ADC	0x007100	AdcRegs
ADC Mirrored Result Registers	0x000B00	AdcMirror
Analog Subsystem Registers	0x006400	AnalogSubsysRegs
CLA1	0x001400	Cla1Regs
Dual Code Security Module Zone 1	0x000B80	DcsmRegsZ1
Dual Code Security Module Zone 2	0x000BC0	DcsmRegsZ2
Dual Code Security Module Z1 Password Locations	0x3D7A00-0x3D7BFF	DcsmOtpZ1
Dual Code Security Module Z2 Password Locations	0x3D7800-0x3D79FF	DcsmOtpZ2
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
NMI Interrupt	0x7060	NmiIntruptRegs
PIE Control	0x000CE0	PieCtrlRegs
SCI-A	0x007050	SciaRegs
SCI-B	0x007750	SciaRegs
SCI-C	0x007770	SciaRegs
SPI-A	0x007040	SpiaRegs

Table 2.16: F2805x Variable Names and Data Sections

2.8.2 Common Example Code

2.8.2.1 Peripheral Interrupt Expansion (PIE) Block Support

In addition to the register definitions defined in F2805x_PieCtrl.h, this packet provides the basic ISR structure for the PIE block. These files are shown in Table 2.17.

File	Location	Description
F2805x_DefaultIsr.c	f2805x\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 SYS/BIOS projects.
F2805x_DefaultIsr.h	f2805x\common\include	Function prototype statements for the ISRs in F2805x_DefaultIsr.c. Note: This file is not used for SYS/BIOS projects.
F2805x_PieVect.c	f2805x\common\source	Creates an instance of the PIE vector table structure initialized with pointers to the ISR functions in F2805x_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 *f2805x\examples\c28\sw_prioritized_interrupts* for more information.

File	Location	Description
F2805x_SWPrioritizedDefaultIsr.c	f2805x\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 SYS/BIOS projects.
F2805x_SWPrioritizedIsrLevels.h	f2805x\common\include	Function prototype statements for the ISRs in F2805x_DefaultIsr.c. Note: This file is not used for SYS/BIOS projects.
F2805x_SWPrioritizedPieVect.c	f2805x\common\source	Creates an instance of the PIE vector table structure initialized with pointers to the default ISR functions that are included in F2805x_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 f2805x\common\src directory. These files are shown in Table 2.19.

File	Description
F2805x_GlobalPrototypes.h	Function prototypes for the peripheral specific functions included in these files.
F2805x_Adc.c	ADC specific functions and macros.
F2805x_AnalogSubsys.c	Comparator and PGA specific functions and macros
F2805x_CpuTimers.c	CPU-Timer specific functions and macros.
F2805x_ECan.c	eCAN module specific functions and macros
F2805x_ECap.c	eCAP module specific functions and macros.
F2805x_EPwm.c	ePWM module specific functions and macros.
F2805x_EPwm_defines.h	define macros that are used for the ePWM examples
F2805x_EQep.c	eQEP module specific functions and macros.
F2805x_Gpio.c	General-purpose IO (GPIO) specific functions and macros.
F2805x_I2C.c	I2C specific functions and macros.
F2805x_I2c_defines.h	define macros that are used for the I2C examples
F2805x_PieCtrl.c	PIE control specific functions and macros.
F2805x_Sci.c	SCI specific functions and macros.
F2805x_Spi.c	SPI specific functions and macros.
F2805x_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
F2805x_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.
F2805x_DBGIER.asm	Assembly function to manipulate the DEBIER register from C.
F2805x_DisInt.asm	Disable interrupt and restore interrupt functions. These functions allow you to disable INTM and DBGM and then later restore their state.
F2805x_usDelay.asm	Assembly function to insert a delay time in microseconds. This function is cycle dependent and must be executed from zero wait-stated RAM to be accurate. Refer to F2805x_examples_ccsv5/adc for an example of its use.
F2805x_DCSM_Z1_ZoneSelectBlock.asm	Include in a project to program Zone 1 of the dual code security module passwords and reserved locations.
F2805x_DCSM_Z2_ZoneSelectBlock.asm	Include in a project to program Zone 2 of the dual 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 f2805x\common\cmd directory. For getting started the basic 28055_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
28055_RAM_lnk.cmd	f2805x\common\cmd	28055 memory linker command file. Includes all of the internal SARAM blocks on a 28055 device. “RAM” linker files do not include flash or OTP blocks.
28054_RAM_lnk.cmd	f2805x\common\cmd	28054 SARAM memory linker command file.
28054m_RAM_lnk.cmd	f2805x\common\cmd	28054m SARAM memory linker command file.
28054f_RAM_lnk.cmd	f2805x\common\cmd	28054f SARAM memory linker command file.
28053_RAM_lnk.cmd	f2805x\common\cmd	28053 SARAM memory linker command file.
28052_RAM_lnk.cmd	f2805x\common\cmd	28052 SARAM memory linker command file.
28052m_RAM_lnk.cmd	f2805x\common\cmd	28052m SARAM memory linker command file.
28052f_RAM_lnk.cmd	f2805x\common\cmd	28052f SARAM memory linker command file.
28051_RAM_lnk.cmd	f2805x\common\cmd	28051 SARAM memory linker command file.
28050_RAM_lnk.cmd	f2805x\common\cmd	28050 SARAM memory linker command file.
28055_RAM_CLA_lnk.cmd	f2805x\common\cmd	28055 CLA memory linker command file. Includes CLA message RAM
28053_RAM_CLA_lnk.cmd	f2805x\common\cmd	28053 SARAM CLA memory linker command file.
F28055.cmd	f2805x\common\cmd	F28055 memory linker command file.
F28054.cmd	f2805x\common\cmd	F28054 memory linker command file.
F28054m.cmd	f2805x\common\cmd	F28054m memory linker command file.
F28054f.cmd	f2805x\common\cmd	F28054f memory linker command file.
F28053.cmd	f2805x\common\cmd	F28053 memory linker command file.
F28052.cmd	f2805x\common\cmd	F28052 memory linker command file.
F28052m.cmd	f2805x\common\cmd	F28052m memory linker command file.
F28052f.cmd	f2805x\common\cmd	F28052f memory linker command file.
F28051.cmd	f2805x\common\cmd	F28051 memory linker command file.
F28050.cmd	f2805x\common\cmd	F28050 memory linker command file.

Table 2.21: Included Main Linker Command Files

2.9 Detailed Revision History

v2.01.00.00

- Updated all examples and removed deprecated compiler options
- F2805x_ECan.c - Corrected CAN init function to set SAM bit to zero

v2.00.00.00

- Updated directories and naming for C2000Ware package
- Cleaned up and reformatted source, header, and example code to new comment structure
- Libraries moved into C2000Ware libraries directory
- 28055_CLA_C_lnk.cmd - Fixed memory addresses for BEGIN and CSM_PWL_P0

V1.04

- Changes to Example Files
  1. **F2805x_examples_ccsv5 -** Cleaned up source and header formatting
     - Added PGA compensation example
     - Rebuilt with C2000 compiler v6.4.2
     - Fixed issue when importing into CCSv5
  2. **F2805x_examples_cla_ccsv5 -** Cleaned up source and header formatting
     - Rebuilt with C2000 compiler v6.4.2
     - Fixed issue when importing into CCSv5

V1.03

- Changes to Example Files
  1. **F2805x_examples_ccsv5 -** Cleaned up CCS warnings
     - Updated ADC examples with AdcOffsetSelfCal()
  2. **F2805x_examples_cla_ccsv5 -** Cleaned up CCS warnings
- Changes to Common Files
  1. **CLA Linker Command Files -** Updated for C2000 compiler v6.4.X support
  2. **F2805x_SysCtrl.c -** Added 1ms delay to XtalOscSel function
  3. **F2805x_Adc.c -** ACQPS value corrected
     - Added SOC conversion wait loops
- Changes to Header Files
  1. **F2805x_PieVect.h -** Updated PieVectTable to be volatile

V1.02

- Changes to Example Files
  1. **F2805x_examples_ccsv5 -** Added F28055 Flash Kernel Example

V1.01

- Changes to F2805x_Common Files
  1. **2805x_RAM_CLA_C_lnk.cmd -** Added .bss_cla and .const_cla

- Changes to Header Files
  1. **F2805x_Cla_typedefs.h -** Fixed the following issues
     - Macro wrapper corrected to reflect the header file name.
     - Added comment to clarify 64 bit usage.
     - Removed redefinition of Uint16 data type.
     - Replaced "#if defined()" with "#ifdef" for consistency.
  2. **IQmathLib.h -** Removed this file from F2805x_common/include/
- Changes to Example Files
  1. **flash_f28055 -** Added EALLOW and EDIS to enable access to protected registers
  2. **CLA Examples -**
     - Changed the CLA vector address calculation to Uint32
     - Modified CLA MMEMCFG access from bit fields to single line using macros

V1.00

- This version is the first release (packaged with development tools and customer trainings) of the F2805x header files and examples.

3 Getting Started with Project Creation and Debugging

3.1 Introduction

This guide 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 enter it in the “Project name” field. Choose a location for it to reside.
2. Uncheck the box to use the default location. This project will be using relative paths, so you will want to create it in C2000Ware to ease portability in the future. After unchecking the box, browse to the f2805x\examples directory. It is recommended to create a new folder for your custom projects. Here we have named it f2805x_custom_projects. Create a folder inside the f2805x_custom_projects folder for your project, in this case, we named it “test”.
3. Select C2000 as the “Family” under the Device section.
4. In the Variant line under Device, Select “2805x Piccolo” in the left box and the specific device in use in the right box.
5. Select the emulator you are using in the “Connection” field and click Finish.
6. Ensure that your 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.

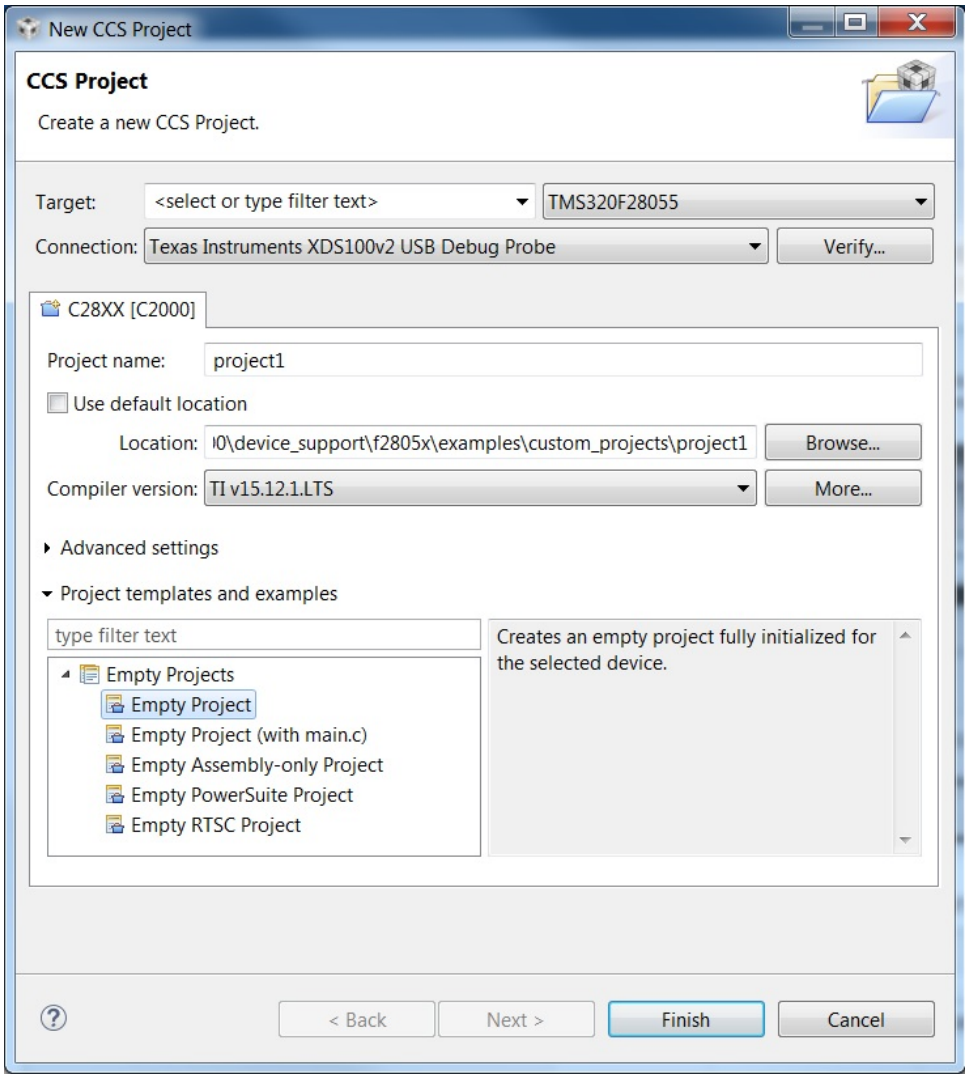

Figure 3.1: Creating a new project

7. 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 `f2805x\common\include` folder of your C2000Ware installation (typically `C:\ti\c2000\C2000Ware_<VERSION>\device_support\f2805x\common\include`). Click ok to add this path, and repeat this same process to add the `f2805x\headers\include` directory.

   While you have this window open, expand C2000 Linker under CCS Build and select Symbol Management. Specify the program entry point to be `code_start`. Select OK to close out of the Project Properties.

8. 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 Add Files... Navigate to the `f2805x\headers\source` directory, and select F2805x_GlobalVariableDefs.c. After opening the file, choose the Link to Files option on the prompt and click OK. This prevents creating

a duplicate copy of the file when it isn't necessary. Link in the following files as well:

- `f2805x\headers\cmd\F2805x_Header_nonBIOS.cmd`
- `f2805x\common\source\F2805x_CodeStartBranch.asm`
- `f2805x\common\source\F2805x_usDelay.asm`
- `f2805x\common\cmd\28055_RAM_CLA_lnk.cmd` or another appropriate linker command file

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

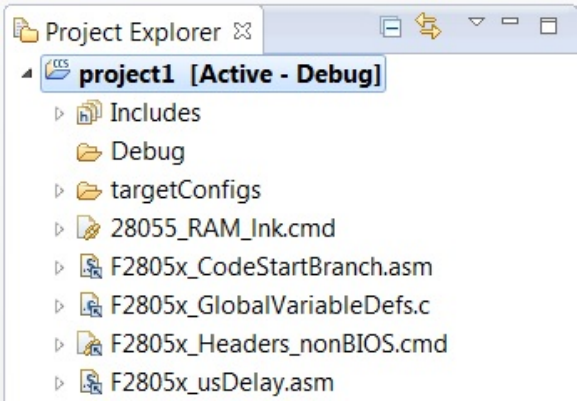

Figure 3.2: Linking files to project

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 C2000Ware. We won't be modifying the linker command file or header files, so this is ok.

9. 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);
        }

    }
```

10. 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 5 is installed and up to date. You should have C2000 Code Generation Tools verion 6.0.2 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 XDS100v2 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. XDS100v2_Piccolo_ControlCARD.ccxml). Select the emulator you intend to use (XDS100v2) from the drop down list, and then select the device variant present on your board (Piccolo controlCARDs have an Experimenters Kit - Piccolo F28055). Save the target configuration and close the window.

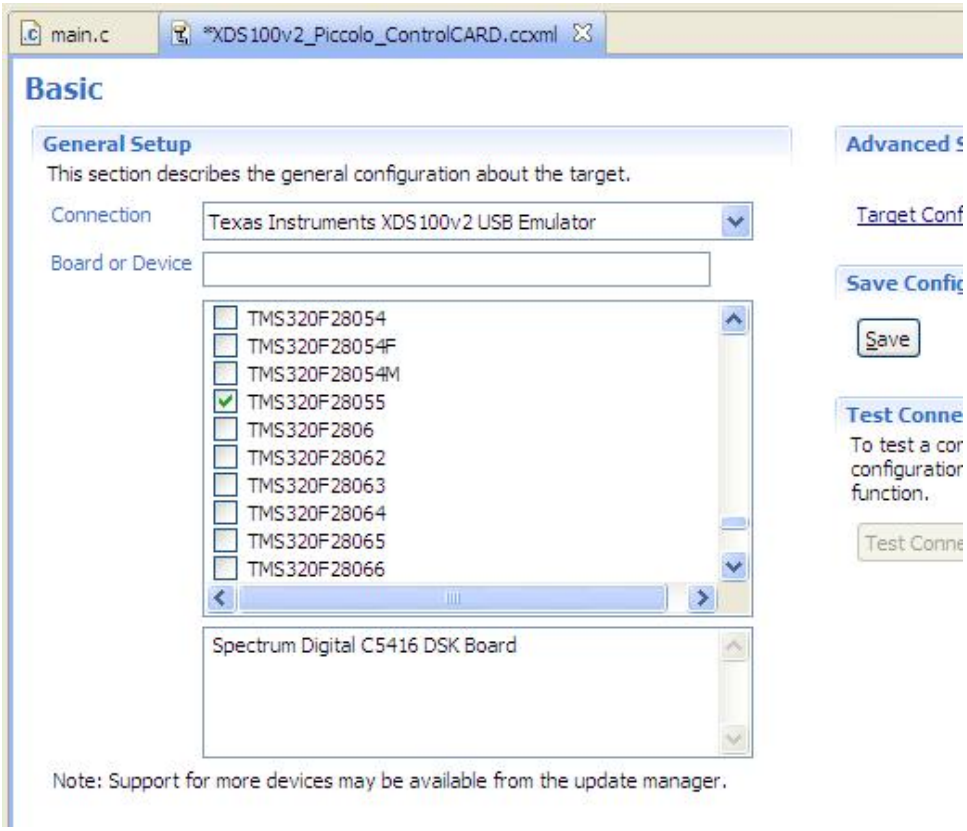

Figure 3.3: Piccolo Card Target Configuration Setup

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 Eclipse Projects before clicking Next. With the "Select search-directory" radio button checked, browse to the root of your C2000Ware installation. Device specific software as well as examples are stored in the `device_support/device_variant` folders. Navigate to the `f2805x` directory, and then to the `f2805x\examples\c28` 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 C2000Ware will break the project.

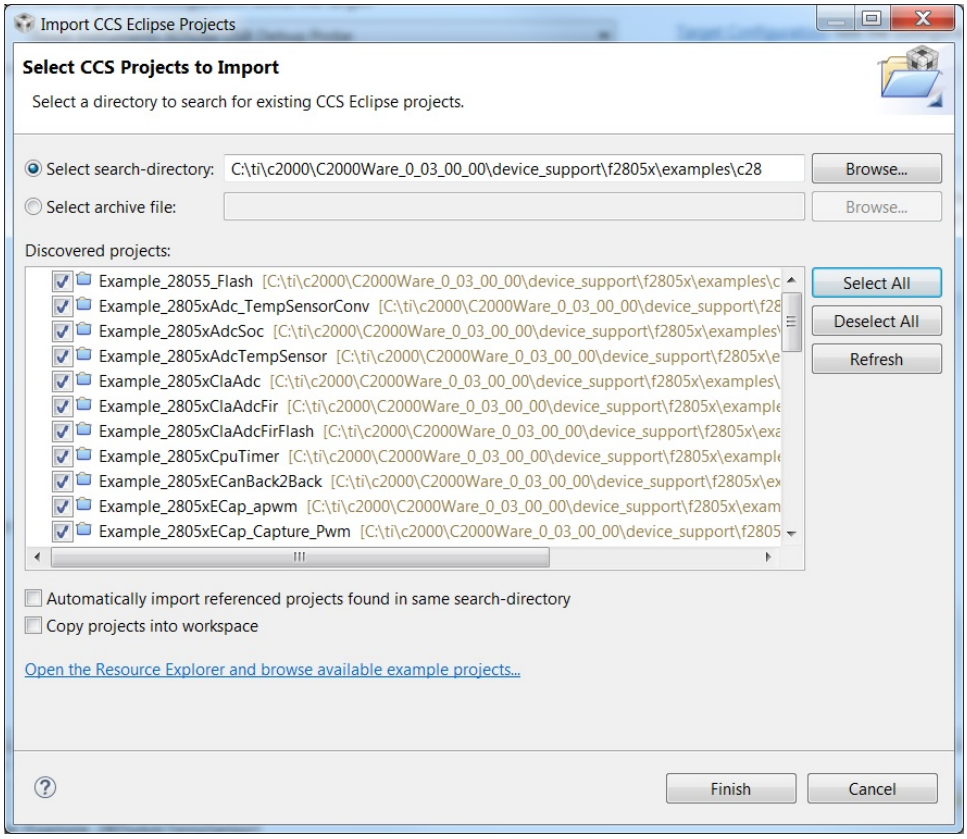

Figure 3.4: Importing Piccolo Projects

6. Build each of the example projects. Right click on each project title and select build project. To build all of the projects you may also use the Project -> Build All option.

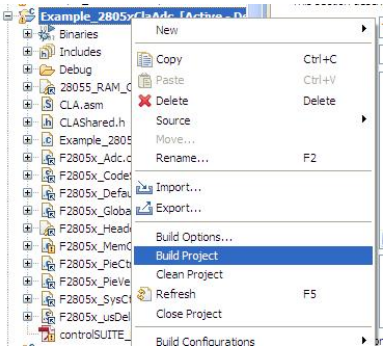

Figure 3.5: Building Piccolo Projects

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".

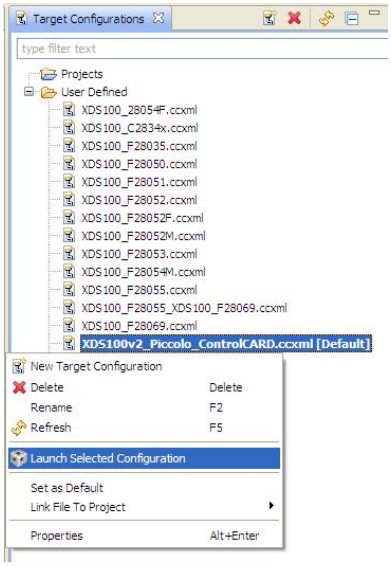

Figure 3.6: Launching a CCS Target Configuration

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.

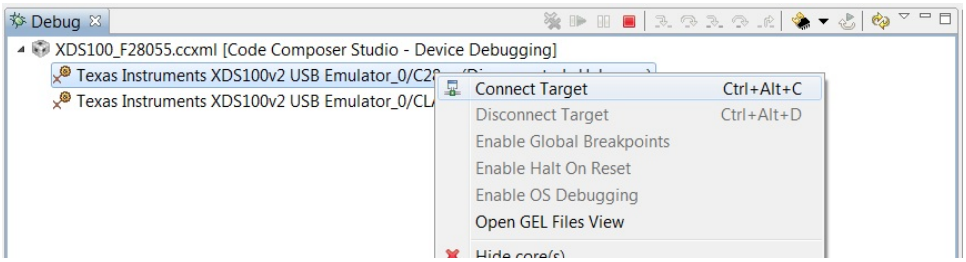

Figure 3.7: Connecting to a Target

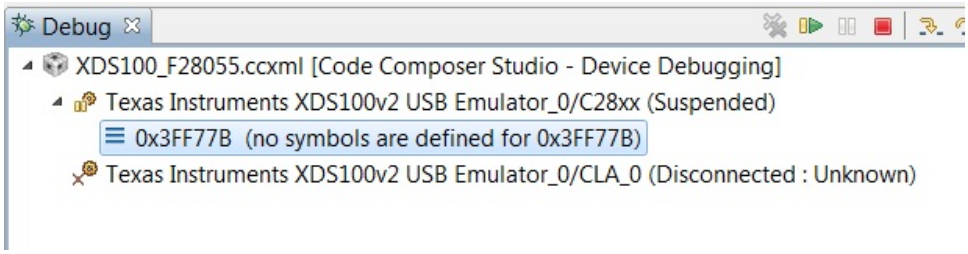

Figure 3.8: After Connection to the C28x Core

9. Load code onto the device. Select the C28x session in the debug window and then click Run -> Load -> 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.

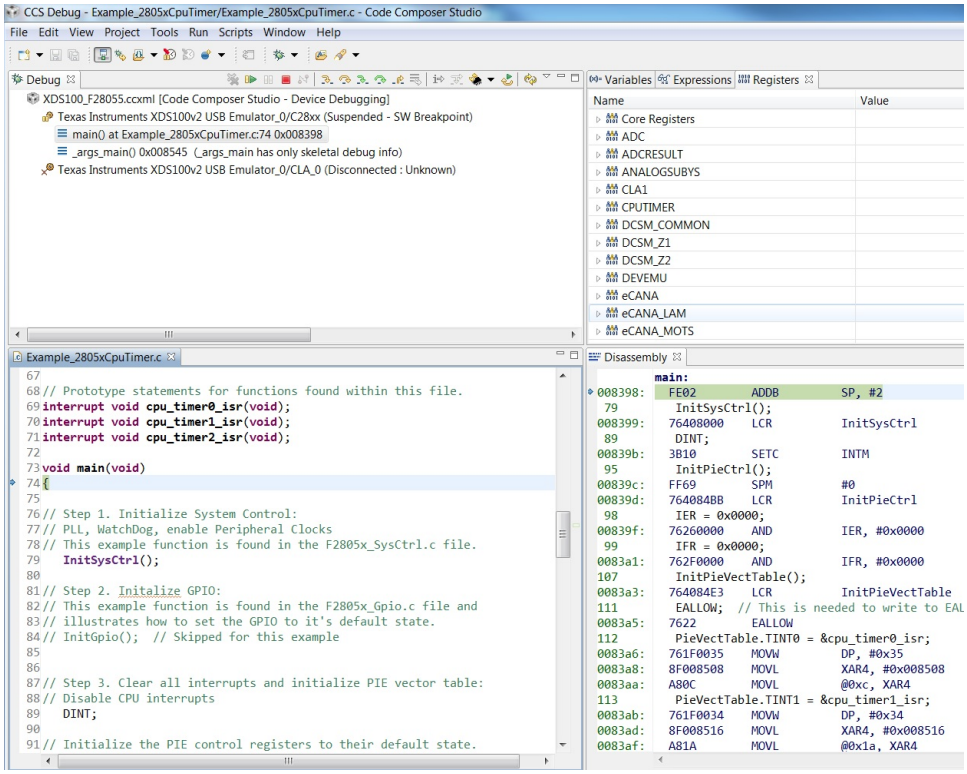

Figure 3.9: Project Loaded on the C28x Core

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 Piccolo devices.

"I get an error when I try to import the example projects"

This occurs when one imports a project for which he or she doesn't have the code generation tools for or the latest CCS device support update supporting your device. Please ensure that you have at least version 15.9.0 of the C2000 Code Generation Tools and have updated your CCS device support through the CCS "Install New Software" menu under "Help".

"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 C2000Ware 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 C2000Ware 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 device memory map.

4 Piccolo F2805x 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 F2805x 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 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 2805x 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
Wait	0x55AA	0x0006
eCANA	0x55AA	0x0007
SARAM	0x55AA	0x000A (*Boot to SARAM*)
Flash	0x55AA	0x000B
Wait	0x55AA	Other

Table 4.1: Boot Modes for Piccolo 2805x

All of these examples reside in the `device_support/f2805x/examples` subdirectory of the C2000Ware package.

4.1 ADC Start of Conversion

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

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

This example shows how to convert a raw ADC temperature sensor reading into degrees C and degrees 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

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

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

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

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

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

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

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 example uses the self-test mode of the CAN module. i.e. the transmission/reception happens within the module itself (even the required ACKnowldege is generated internally in the module). Therefore, there is no need for a CAN transceiver to run this particular test case and no activity will be seen in the CAN pins/bus. Because everything is internal, there is no need for a 120-ohm termination resistor. Note that a real-world CAN application needs a CAN transceiver and termination resistors on both ends of the bus.

Watch Variables

- PassCount
- ErrorCount
- MessageReceivedCount

4.9 eCAP APWM

This program sets up the eCAP pins in the APWM mode. eCAP1 will come out on the GPIO24 pin This pin is configured to vary between frequencies using the shadow registers to load the next period/compare values

4.10 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 GPIO24
- ePWM3A is on GPIO4
- Connect GPIO4 to GPIO19.

Watch Variables

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

4.11 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

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 GPIO16
- pull TZ1 low and leave TZ2 high to create a DCAEVT1, DCAEVT2, DCBEVT1 and DCBEVT2.

4.13 ePWM Deadband Generation

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

4.14 ePWM Real-Time Interrupt

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

4.15 ePWM Timer Interrupt

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

4.16 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

4.17 ePWM Safety Trip Zone

This example shows how to use ePWM trip zones 4 (EQEP1ERR), 5 (CLOCKFAIL), and 6 (EMUSTOP) to trip the ePWM outputs under certain safety-critical conditions. An emulator stop causes TZ6 to trip ePWM3A low and ePWM3B high. An EQEP phase error (PHE) causes TZ4 to trip ePWM1A and ePWM1B low. A CLOCKFAIL will cause TZ5 to trip ePWM2A and ePWM2B high.

The EMUSTOP will be the first trip to occur when the coded breakpoint is reached. You will need to press RUN to continue the simulation. TZ4 will occur next followed by TZ5. The delay between

trips may be increased or decreased to make viewing the ePWM outputs easier by modifying the delay function.

During the example, monitor ePWM1 ePWM2, and ePWM3 outputs on a scope

External Connections

- EPWM1A is on GPIO0 (One shot)
- EPWM1B is on GPIO1 (One shot)
- EPWM2A is on GPIO2 (One shot)
- EPWM2B is on GPIO3 (One shot)
- EPWM3A is on GPIO4 (Cycle by cycle)
- EPWM3B is on GPIO5 (Cycle by cycle)

4.18 ePWM Action Qualifier Module using Upcount mode

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

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

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.

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 Frequency Calculation (Example_freqcal.c)

This file includes the EQEP initialization and frequency calculation functions called by **Example_2805xEqep_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:

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

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

$$= 250\ QCAPCLK\ cycles = maximum(t_2 - t_1) = freqScaler\_pr$$

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

$$1 = \frac{250}{t_2 - t_1} = \frac{(60MHz/128)/((2 * 10KHz)/8)}{t_2 - t_1}$$

Because $(t_2 - t_1) < 250(max)$, $\frac{250}{t_2 - t_1} < 1$ for all frequencies less than max

$$freq\_pr = \frac{250}{t_2 - t_1}\ or\ \frac{(60MHz/128)/((2 * 10KHz)/8)}{t_2 - t_1} .....6$$

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

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.

- GPIO20/eQEP1A <- GPIO0/ePWM1A (simulates eQEP Phase A signal)
- GPIO21/eQEP1B <- GPIO1/ePWM1B (simulates eQEP Phase B signal)
- GPIO23/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

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. On the f28069 controlSTICK, GPIO-34 also toggles a red LED. 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 F28055 Flash Kernel

This example is for use with the SerialLoader2000 utility. This application is intended to be loaded into the device's RAM via the SCI boot mode. After successfully loaded this program implements a modified version of the SCI boot protocol that allows a user application to be programmed into flash

4.24 ePWM Timer Interrupt From Flash

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

4.25 Flash Programming

This example shows how to program flash using flash API. It shows.

- The required software setup before calling the API (setting the PLL, checking for limp mode etc.),
- How to copy the API from flash into SARAM for execution.
- How to call the API functions.

Note:
This example runs from Flash. First program the example into flash. The code will then copy the API's to RAM and modify the flash.

4.26 GPIO Setup

This example Configures the 2805x 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

4.27 GPIO Toggle Test

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.

4.28 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.(e.g. F2805x eZdsp)

Watch Variables

- I2cMsgIn1
- I2cMsgOut1

4.29 Low Power Modes: Halt Mode and Wakeup

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)

4.30 Low Power Modes: Device Idle Mode and Wakeup

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.

4.31 Low Power Modes: Device Standby Mode and Wakeup

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)

4.32 Internal Oscillator Compensation

This program shows how to use the internal oscillator compensation functions in F2805x_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 2805x 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

4.33 PGA Error Compensation

This ADC/PGA example uses ePWM1 to generate a periodic ADC SOC - ADCINT1. The PGAs are configured for a GAIN of 3 and are converted on channels ADCINA3 and ADCINA1. The ADC conversions are post-processed for error compensation.

Watch Variables

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

4.34 SCI Echo Back

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 2805x 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
- Data 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.

4.35 SCI Digital Loop Back

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

4.36 SCI Digital Loop Back with 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

4.37 SPI Digital Loop Back

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

4.38 SPI Digital Loop Back with 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

4.39 Software Prioritized Interrupts

For most applications, the hardware prioritizing of 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 F2805 interrupt priorities, refer to the "Example ISR Priorities" Appendix in the Firmware Development Users guide.

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 F2805x_SWPrioritizedIsrLevels.h file.

Running the Application

1. Before compiling you must set the Global and Group interrupt priorities in the F2805x_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.

4.40 Timer based blinking LED

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 2805x control card.

4.41 Watchdog Interrupt Test

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

5 CLA C Compiler

5.1 Introduction

The goal of the CLA compiler is to implement enough of the C programming environment to make it easier to access the capabilities of the CLA architecture and to make it easier to integrate CLA task code and data into a C28x application.

The purpose of testing the prototype is to:

1. Provide feedback on the compiler implementation and to discover issues
2. Generate sample code that can be integrated into TI's regression testing

The compiler is available on the Tools download page, CLA Beta Tools , along with each revision's README and defect history. Template projects that were part of the regression tests are now incorporated in the *f2805x\examples\cla* folder

All bugs, performance issues should be reported to Compiler Support, either at the forum Compiler Forum or ClearQuest Report

5.2 Overview

The README.txt file included in the compiler download package contains the latest details on the CLA compiler's C language implementation and it is highly recommended that you go over this document before you begin coding.

5.2.1 How to Invoke the CLA Compiler

The CLA compiler is invoked using the same command used for compiling C28x code (cl2000[.exe]).

Files that have a .cla extension will be recognized by the compiler as CLA C files. The shell will invoke separate CLA versions of the compiler passes to generate CLA-specific code. The object files generated by the compiler can then be linked with C28x objects files to create a C28x/CLA program.

Usage:

cl2000 -v28 –cla_support=cla0 [other options] file.cla

NOTE: THE COMPILER DOES NOT SUPPORT COMPILING BOTH CLA AND C28X C FILES IN ONE INVOCATION.

5.2.2 C Language Implementation

5.2.2.1 Characteristics

Language

Supports C only. No C++ or GCC extension support.

Data Types

(NOTE THE DIFFERENCES FROM C28X DATA TYPES!!)

- char,short - 16 bits
- int,long - 32 bits ('long long' data type is not supported)
- float, double, long double - 32 bits
- pointers - 16 bits

IMPORTANT NOTES:
The CLA and C28x CPU have different type sizes. • When declaring data that will be shared by both C28x and CLA use type declarations that will result in objects of the same size • To avoid ambiguity use typedefs for basic types that include size information (eg. int32, uint16, etc)
The CLA architecture is oriented for 32-bit data types. • 16-bit data types incur sign extension overhead and should primarily be used for load/store operations such as reading/writing 16-bit peripherals.
Pointers are **INTERPRETED** differently • Pointers on the C28 are 22-bits wide and require at minimum 2 contiguous 16-bit locations for storage. As such they are treated as 32-bit data types(since we cannot allocate 22 bit memory locations) • The CLA treats pointers as 16-bit data types. Any pointer shared between the C28 and CLA will be interpreted as a 16-bit location by the CLA compiler and this could cause undesired or bad data accesses by the CLA.
NOTE!! THE CLA COMPILER DOES NOT HAVE 64-BIT TYPE SUPPORT.

Pragmas

The compiler accepts C28x pragmas except for the FAST_FUNC_CALL

C Standard Library

In general, the C standard library is not supported. abs() and fabs() are supported as intrinsics. An inline fast floating-point divide is supported.

Keywords

The keywords '__cregister','far', and 'ioport' are not recognized

Intrinsics

The following intrinsics are supported:

- float __meisqrtf32(float)
- float __meinvf32(float)
- float __mminf32(float, float)
- float __mmaxf32(float, float)
- void __mswapf(float, float)
- short __mf32toi16r(float)
- unsigned short __mf32toui16r(float)
- float __mfracf32(float)
- __mdebugstop()
- __meallow()
- __medis()
- __msetflg(unsigned short, unsigned short)
- __mnop()

5.2.3 Language Restrictions

Global Initialization

Defining and initializing global data is not supported.

Since the CLA code is executed in an interrupt driven environment there is no C system boot sequence. As a result, definitions of the form 'int global_var = 5;' are not allowed for variables that are defined globally (outside the scope of a function). Initialization of global data must either be done by the C28x driver code or within a function.

Variables defined as 'const' can be initialized globally. The compiler will create initialized data sections named **.const_cla** to hold these variables. The same restriction applies to variables declared as 'static'. Even if the variable is defined within a function.

Stack

Local variables and compiler temps are placed into a scratchpad memory area and accessed directly using the symbols **'__cla_scratchpad_start'** and **'__cla_scratchpad_end'**. It is expected that the user will manage this area and define these symbols using a linker command file.

IMPORTANT NOTES:
Local variables and compiler temps are expected to be placed into a scratchpad memory area and accessed directly using the symbols **'__cla_scratchpad_start'** and **'__cla_scratchpad_end'**. • It is expected that the user will manage this area and define these symbols using a linker command file. • This scratchpad serves as a CLA stack.
To allow debug of local variables, the linker .cmd file has been updated from that originally distributed • Please ensure the changes to the .cmd file shown below are made before proceeding. • The linker file should look like the code shown below. • This also required a compiler released after July 21, 2011.

The following is an example of what needs to be added to a linker command file to define the CLA compiler scratchpad memory:

- Define the scratchpad size - **CLA_SCRATCHPAD_SIZE** is a linker defined symbol that can added to the application's linker command file to designate the size of the scratchpad memory.
- A SECTION's directive can reference this symbol to allocate the scratchpad area. This directive reserves a 0x100 word memory hole to be used as the compiler scratchpad area.
- The scratchpad area is named **CLAscratch** and is allotted to CLA Data RAM 1 (CLARAM1)
- The value of CLA_SCRATCHPAD_SIZE can be changed based on the application.

```
// Define a size for the CLA scratchpad area that will be used
```

```
// by the CLA compiler for local symbols and temps
// Also force references to the special symbols that mark the
// scratchpad area.

// If using --define CLA_SCRATCHPAD_SIZE=0x100, remove next line
CLA_SCRATCHPAD_SIZE = 0x100;
--undef_sym=__cla_scratchpad_end
--undef_sym=__cla_scratchpad_start

.....
MEMORY
{
.....
}
SECTIONS
{
   //
   // Must be allocated to memory the CLA has write access to
   //
   CLAscratch :
{ *.obj(CLAscratch)
. += CLA_SCRATCHPAD_SIZE;
*.obj(CLAscratch_end) } > CLARAM1,  PAGE = 1
}
```

The scratchpad size can alternatively be defined and altered in the linker options of a project as shown below

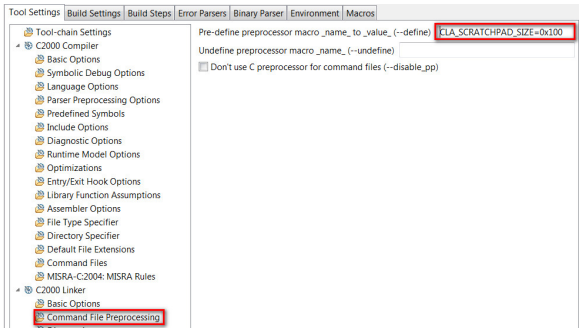

Figure 5.1: Adjusting scratchpad size through the linker options

Function Nesting

Only 2 levels of call stack depth is supported. See Section 5.2.5 for details on the calling conventions.

Recursion

Recursive function calls are not supported.

Function Pointers

Function pointers are not supported.

Other Operations

The following operations are currently not supported due to lack of instruction set support making them expensive to implement. It is not clear that these operations are critical for typical CLA algorithms.

- Integer divide, modulus
- Integer unsigned compares

5.2.4 Memory Model - Sections

CLA Program

The CLA compiler will place CLA code into section **"Cla1Prog"** as per the current convention used for CLA assembly.

Global Data

Uninitialized global data will be placed in the section **".bss_cla"**

Constants

Initialized constant data will be placed in section **".const_cla"**

Heap

There is no support for operations such as malloc(). Therefore there is no C system heap for CLA.

5.2.5 Function Structure and Calling Conventions

Function Nesting

The compiler supports 2 level of function calls. Functions declared as interrupts may call leaf functions only. Leaf function may not call other functions. Functions not declared as interrupt will be considered leaf functions. NOTE: THE CLA TASKS ARE PREFIXED WITH THE KEYWORD '__interrupt' TO SET THEM APART FROM LEAF FUNCTIONS. THEY ARE NOT TO BE CONFUSED WITH C28X INTERRUPT SERVICE ROUTINES

Register Calling Convention

The CLA compiler supports calling functions with up to 2 arguments.

- Pointer arguments are passed in MAR0/MAR1.
- Integer/float arguments are passed in MR0,MR1.
- Integer and float return values from functions are passed in MR0.
- Pointer or return by reference value from functions are passed in MAR0.

Register Save/Restore

All registers except for MR3 are saved on call. MR3 is saved on entry. NOTE: IF YOU ARE WRITING AN ASM ROUTINE TO BE CALLED IN THE C CONTEXT IT IS YOUR RESPONSIBILITY TO SAVE/RESTORE MR3 UPON ENTRY AND EXIT RESPECTIVELY

Local Variables

A static scratchpad area is used as a stack for locals and compiler temporary variables. NOTE:THE USER IS RESPONSIBLE FOR ENSURING THE SCRATCHPAD AREA IS ALLOCATED INTO THE MEMORY MAP AND IS LARGE ENOUGH. THIS IS DONE USING THE EITHER THE LINKER COMMAND FILE OR THROUGH THE PROJECT'S LINKER OPTIONS (SEE ABOVE).

Mixing CLA C and Assembly

When interfacing with CLA assembly language modules use the calling conventions defined above to interface with compiled CLA code.

5.3 Framework

The CLA examples are in the folder *"f2805x\examples\cla"*. Each example within this folder share a similar structure as shown in the figure below (Fig. 5.2)

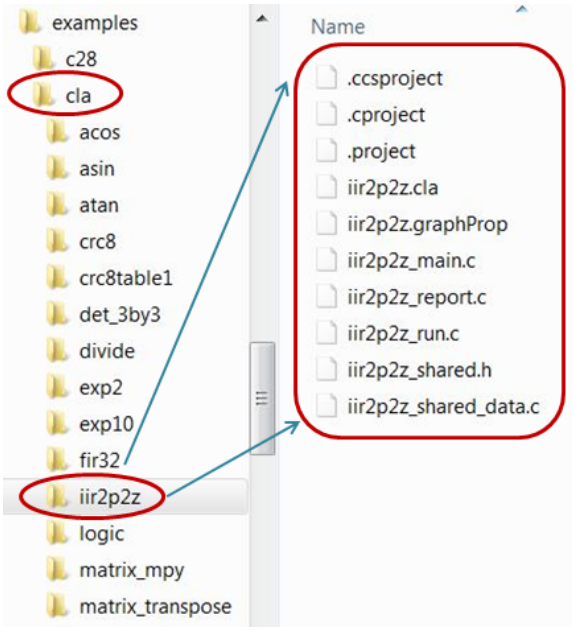

Figure 5.2: Structure of a CLA example

For any given *example* there are 6 specific files associated with it as described in Table. 5.1.

Source File	**Description**
<*example*>_main.c	implements the main() routine. System and peripheral intialization is done here along with setting up the CLA configuration registers.
<*example*>.cla	The C implementation of all the CLA tasks. File level data global to the CLA only(not shared with the C28x) should also be defined in this file.
<*example*>_run.c	Contains routine test_run() called by main(). test_run() determines which CLA tasks are triggered and if/how any data is passed to the task through the message RAMs
<*example*>_shared.h	External declarations for the global data defined in the C28x code and referenced by the CLA task code.
<*example*>_shared_data.c	Variables declared in <*example*>_shared.h are defined here and allocated to memory (using #pragma DATA_SECTION). CLA VARIABLES MUST BE ALLOCATED TO A MEMORY SPACE THAT THE CLA HAS ACCESS TO, NAMELY THE CLA<->CPU MESSAGE RAMS OR THE CLA DATA RAMS.
<*example*>_report.c	Implements the routine test_report() which is called by main() after all CLA tasks run to completion. The convention used for reporting test results is to have the CLA task write results to a shared variable(s) and have the C28x driver code call test_report() to compare the data to expected values and to output the test run status to the CCS console. The convention used to check the status of a test run is to expect the string “PASS” on success and “FAIL” on failure. Other information can also be printed (eg. actual results vs expected result) but the testing harness will expect, at the minimum, either PASS or FAIL to be output.

Table 5.1: Example specific files

5.4 Getting Started with the CLA Compiler

The C code for the CLA is saved to a file with the *.cla* extension. If running an older version of CCSv5 (v5.2 or older) that does not recognize the extension, you can follow these steps:
NOTE: FOR EACH NEW WORKSPACE THE USER MUST CONFIGURE CCS IN THE MANNER DESCRIBED BELOW

1. Go to Windows->Preferences->C/C++->File Types.
2. Select “New”
3. Type in *.cla in the top text box
4. In the drop down menu select C source file(see Fig. 5.3).
5. Select “ok”

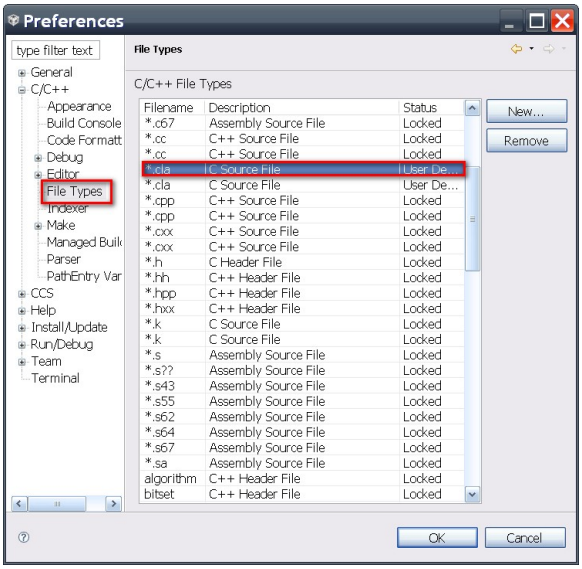

Figure 5.3: Configuring CCS5 to recognize the .cla extension

The IDE will now recognize the .cla extension as code to be compiled.

IF YOU CREATE A NEW WORKSPACE YOU WILL NEED TO REPEAT THIS PROCESS. THERE WILL BE A FUTURE UPDATE TO CCS TO ADD THE ASSOCIATION AUTOMATICALLY.

5.4.1 Creating Your Own Project

The simplest way to start writing code is to copy over an existing project (from the examples folder) and to edit it. Lets take an example: I would like to create a new project, **exp2**, from an existing project, **atan**.

1. Copy a Project:
   - Make a copy of the **atan** folder in the example directory and rename it to **exp2**

2. Rename Files:
   - Rename all files atan*.* to exp2*.*. (Notice the naming convention. All files have the test folder name as a prefix, see Fig. 5.4 below)

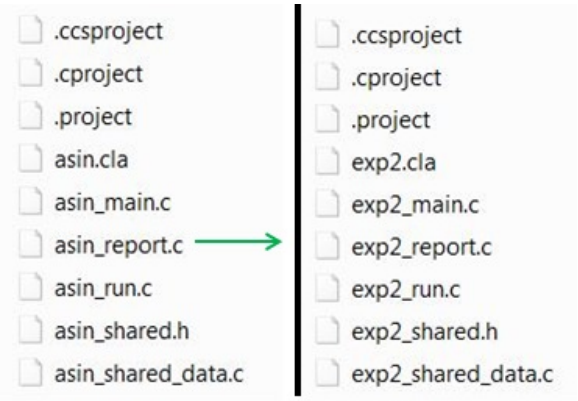

Figure 5.4: Creating a new project from existing examples

3. Edit the Project Files:
   - Open the .cproject and .project files in any text editor and replace all instances of the word atan with exp2.
   - This will ensure all the object files come out with the correct name and any directory dependencies are taken care of.
   - Each project has a predefined symbol, TEST_NAME=<test_name>. For e.g. the atan project will have a predefined symbol, TEST_NAME=atan. By altering the .cproject files in the manner described you won't have to change the build settings for each new project .
4. Import the Project:
   - Import the exp2 project into your workspace (see Fig. 5.5).
   - The files highlighted in the red box are common to all the CLA examples and are linked in by the .project file. The rest of the source files are specific to each test case

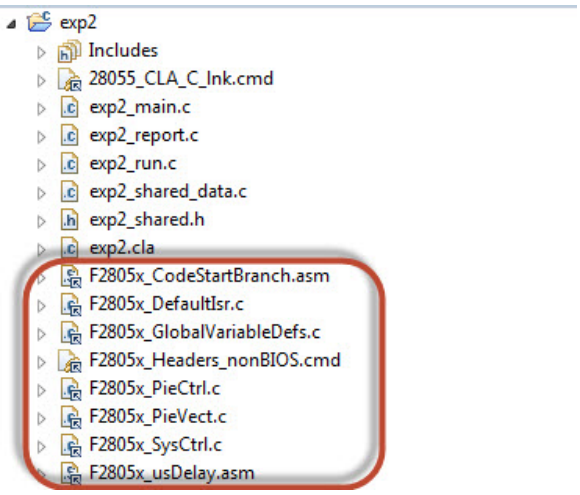

Figure 5.5: Common source files for each CLA example

5. Modify the Source:
   - Edit the test specific source files.

5.4.2 Suggested Build Options

The following table lists build options that are useful for CLA C code. You can setup build properties that apply only to *.cla file by right clicking the file and selecting Properties. In the Properties window, expand 'Build' and its submenus.

Option	**Notes**
Basic Options->Debugging Model->Full Symoblic debug (-g)	If you would like to access watch variables etc while debugging(default setting).
Basic Options->Debugging Model->Suppress symbolic debug information	View compiler generated assembly code without all the debug information.
Optimization->Optimization Level = none - O2	DUE TO THE SMALL NUMBER OF REGISTERS AVAILABLE LESS AGGRESSIVE OPTIMIZATION MAY YIELD BETTER RESULTS (EG. -O1 VS -O2).
Assembler Options -> Keep generated assembly files (-k)	Useful if you want to compare compiler generated code with hand coded assembly.

Table 5.2: Suggested Build Options

5.5 Debugging

The user can follow these steps to start debugging their code on the CLA (The project *exp2* is used as an example here)

1. Add __mdebugstop()
   - Place an __mdebugstop() at the beginning of the CLA task you wish to debug. For example, task 1 of exp2.cla.
2. Set build options:
   - You can setup individual build properties for the *.cla file seperately from the rest of the application.
   - Right click the .cla file and select **Properties->C/C++ Build**.
3. Connect to the CLA:
   - Once you have built your project and launched the debug session CCS, by default, will connect to only the C28 core.
   - To be able to debug CLA code you will need to connect to the CLA core. The action of connecting to the CLA core enables all software breakpoints and single-stepping abilities.
   - IF YOU WISH TO STEP THROUGH C CODE BUILD THE PROJECT WITH -G (FULL SYMBOLIC DEBUG) TO GENERATE THE SYMBOLS THAT WILL BE LOADED TO THE DEBUGGER.
     (a) Click on the CLA debug session (highlighted in Fig. 5.6)
     (b) Select *Target->Connect to Target* or hit Alt-C.
     (c) Once the CLA core is connected proceed to load the project symbols by clicking on *Target->Load Symbols-><example>.out* (e.g. exp2.out).

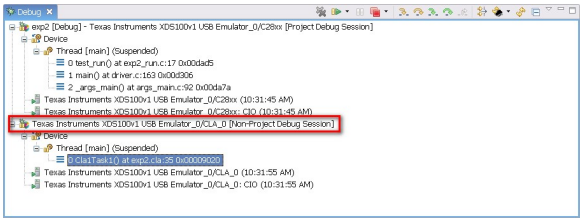

Figure 5.6: CLA Debug Session

4. Run the C28x:
   - In the exp2 example we have enabled task 1 of the CLA and we trigger it in software on the C28 side. When we run the code on the C28 debug session it seems to stall at the Cla1ForceTask1andWait() routine. It is waiting for the CLA task 1 to run to completion. When we switch over to the CLA session we see that execution has stopped at the __mdebustop() intrinsic
5. Debug the Code:
   - At this point we can proceed to single step through the code or continue till completion.
   - There are some restrictions to debugging the CLA and they are discussed next.

5.6 Known Debugging Issues

1. The CLA pipeline is not flushed on a single step and so results may not be visible until a few instructions later. Please refer to the CLA user guide or the device *Technical Reference Manual* for more details about the pipeline.
   UNLIKE THE C28, SINGLE-STEPPING ON THE CLA DOES NOT FLUSH THE PIPELINE AND EXECUTE AN INSTRUCTION , IT MERELY MOVES THE PIPELINE FORWARD BY ONE STAGE)
2. If you plan to debug (single step) code on the CLA it is necessary that MNOPs are placed prior to any MSTOP to ensure the instructions prior to the MSTOP proceed through the pipeline before the MSTOP executes. The compiler will insert these MNOPs if compiling with debug (-g). The MNOPs are unnecessary if you are not debugging the CLA code.
3. YOU WILL NOT BE ABLE TO EXECUTE THE "RUN TO LINE" OR "STEP OVER" COMMANDS ON THE CLA. BE SURE TO PLACE __MDEBUGSTOP() INTRINSICS AROUND FUNCTIONS YOU WISH TO STEP OVER AND HAVE THE CORE RUN TO THESE BREAKPOINTS DIRECTLY

5.7 Tips and Tricks

5.7.1 Dealing with Pointers

Pointers are interpreted differently on the C28x and the CLA. The C28 treats them as 32-bit data types(address size is 22-bits) while the CLA can only use an address size of 16 bits. Assume the following structure is declared in a shared header file(i.e. common to the C28 and CLA) and defined and allocated to a memory section in a .c file

```
/*************************************************************************
 Shared Header File
*************************************************************************/
typedef struct{
  float a;
  float *b;
  float *c;
}foo;

/*************************************************************************
 main.c
*************************************************************************/
#pragma(X,"CpuToCla1MsgRam") //Assign X to section CpuToCla1MsgRam
foo X;

/*************************************************************************
 test.cla
*************************************************************************/
__interrupt void Cla1Task1 ( void )
{
  float f1,f2;
  f1 = *(X.b);
  f2 = *(X.c); //Pointer incorrectly dereferenced
               //Tries to access location 0x1503 instead
```

```
                //of 0x1504
}
```

Assume that the C28 compiler will allocate space for X at the top of the section **CpuToCla1MsgRam** as follows:

Element	Address
X.a	0x1500
X.b	0x1502
X.c	0x1504

The CLA compiler will interpret this structure differently

Element	Address
X.a	0x1500
X.b	0x1502
X.c	**0x1503**

The CLA compiler treats pointers **b** and **c** as 16-bits wide and therefore incorrectly dereferences pointer **c**.

The solution to this is to declare a new pointer as follows:

```
/*************************************************************************
 Shared Header File
*************************************************************************/
typedef union{
  float *ptr; //Aligned to lower 16-bits
  Uint32 pad; //32-bits
}CLA_FPTR;

typedef struct{
  float a;
  CLA_FPTR b;
  CLA_FPTR c;
}foo;

/*************************************************************************
 main.c
*************************************************************************/
#pragma(X,"CpuToCla1MsgRam") //Assign X to section CpuToCla1MsgRam
foo X;

/*************************************************************************
 test.cla
*************************************************************************/
__interrupt void Cla1Task1 ( void )
{
  float f1,f2;
  f1 = *(X.b.ptr);
  f2 = *(X.c.ptr); //Correct Access
}
```

The new pointer **CLA_FPTR** is a union of a 32-bit integer and a pointer to a float. The CLA compiler recognizes the size of the larger of the two elements(the 32 bit integer) and therefore aligns the pointer to the lower 16-bits. Now both the pointers **b** and **c** will occupy 32-bit memory spaces and any instruction that tries to dereference pointer **c** will access the correct address 0x1504.

5.7.2 Benchmarking

The CLA does not support the clock function and therefore it is not possible to get a direct cycle count of a particular task. The user can configure the time base module on an ePWM to keep track of the execution time of a task

Setup the time base of ePWM1(or any ePWM) to run at SYSCLKOUT in the up-count mode as shown below:

```
void InitEPwm(void)
{
   // Setup TBCLK
   EPwm1Regs.TBCTL.bit.CTRMODE = TB_COUNT_UP; // Count up
   EPwm1Regs.TBPRD = 0xFFFF;                  // Set timer period
   EPwm1Regs.TBCTL.bit.PHSEN = TB_DISABLE;    // Disable phase loading
   EPwm1Regs.TBPHS.half.TBPHS = 0x0000;       // Phase is 0
   EPwm1Regs.TBCTR = 0x0000;                  // Clear counter
   EPwm1Regs.TBCTL.bit.HSPCLKDIV = TB_DIV1;   // Clock ratio to SYSCLKOUT
   EPwm1Regs.TBCTL.bit.CLKDIV = TB_DIV1;
}
```

Proceed to define two macros **READ_CLOCK** and **RESTART_CLOCK**, the former to freeze the ePWM timer and copy the elapsed time to a variable, and the latter to restart the ePWM timer.

```
#define READ_CLOCK(X) __meallow();\
                      EPwm1Regs.TBCTL.bit.CTRMODE = TB_FREEZE;\
                      X = EPwm1Regs.TBCTR;\
                      __medis();
#define RESTART_CLOCK __meallow();\
                      EPwm1Regs.TBCTL.bit.CTRMODE = TB_FREEZE;\
                      EPwm1Regs.TBCTR = 0;\
                      EPwm1Regs.TBCTL.bit.CTRMODE = TB_COUNT_UP;\
                      __medis();
```

Define a variable e.g. **ulCycleCount** to hold the cycle count

```
#pragma DATA_SECTION(ulCycleCount,"Cla1ToCpuMsgRAM");
unsigned long ulCycleCount;
```

Place the macro **RESTART_CLOCK** at the beginning of a task to restart the ePWM timer and place **READ_CLOCK** at the end of the task to read the value of the timer. The elapsed time will be give you the cycle count plus a minimal overhead from the two macros

```
__interrupt void Cla1Task1 ( void )
{
```

```
  //Local Variables
  float a;

  __mdebugstop();
  RESTART_CLOCK;
  a = 10;
  ...
  ...
  ...
  READ_CLOCK(ulCycleCount);
}
```

5.7.3 Mixing C and Assembly Tasks

It is possible to implement some tasks in assembly while some in C. In a .asm you can declare the task with a symbol e.g. **(**_Cla1Task1) and must assign it to the memory section **Cla1Prog**. Define a symbol at the head of the section e.g. **_Cla1Prog_ASM_start**

```
    .sect   "Cla1Prog"
_Cla1Prog_ASM_start:
    .align  2
_Cla1Task1:
       ...
       ...
       ...
   MSTOP
_Cla1Task1End:
```

The C tasks are defined in the usual way. The only difference between the two is the manner in which the MVECT's are initialized

For assembly tasks the MVECT is as follows:

```
Cla1Regs.MVECT1 = (Uint16) (&Cla1Task1 - &Cla1Prog_ASM_start)*sizeof(Uint32);
```

while for C tasks, it is:

```
Cla1Regs.MVECT1 = (Uint16)((Uint32)&Cla1Task1 - (Uint32)&Cla1Prog_Start);
```

where **Cla1Prog_Start** is a linker defined variable which must be declared in a .c file with the storage classifier *extern*

6 CLA 'C' Example Applications

These examples demonstrate the use of the C compiler, programming model and coding practices to generate efficient code.

Notes

- All examples require the F2805x header files
- All examples run at max SYSCLOCK of 60MHz. This is the default setting assuming the input clock is derived from the 10MHz internal clock.
- C code for the CLA is in the files with the .cla extension

6.1 ACOS Table-Lookup Algorithm

This example implements a table lookup method of determining the arccosine of a value.

Watch Variables

- y - Accumulated results (angles in radians)

Memory Allocation

- CLA1 Math Tables (RAML2)
  - CLAacosinTable - Lookup table
- CLA1 to CPU Message RAM
  - fResult - Result of the lookup algorithm
- CPU to CLA1 Message RAM
  - fVal - Sample input to the lookup algorithm

6.2 ASIN Table-Lookup Algorithm

This example implements a table lookup method of determining the arcsine of a value.

Watch Variables

- y - Accumulated results (angles in radians)

Memory Allocation

- CLA1 Math Tables (RAML2)
  - CLAasinTable - Lookup table
- CLA1 to CPU Message RAM
  - fResult - Result of the lookup algorithm
- CPU to CLA1 Message RAM
  - fVal - Sample input to the lookup algorithm

6.3 ATAN Table-Lookup Algorithm

This example implements a table lookup method of determining the arctangent of a value.

Watch Variables

- y - Accumulated results (angles in radians)

Memory Allocation

- CLA1 Math Tables (RAML2)
  - CLAatan2Table - Lookup table
- CLA1 to CPU Message RAM
  - fResult - Result of the lookup algorithm
- CPU to CLA1 Message RAM
  - fNum - Numerator of sample input
  - fDen - Denominator of sample input

6.4 CRC8 Table-Lookup Algorithm

This example implements a table lookup method of determining the 8-bit CRC of a message sequence. The polynomial used is 0x07

Watch Variables

- crc8_msg1 - CRC of message 1
- crc8_msg2 - CRC of message 2
- crc8_msg3 - CRC of message 3
- crc8_msg4 - CRC of message 4
- fail_flag

Memory Allocation

- CLA1 Data RAM 1(RAML2)
  - table - CRC Lookup table
- CLA1 to CPU Message RAM
  - crc8_msg1 - CRC of message 1
  - crc8_msg2 - CRC of message 2
  - crc8_msg3 - CRC of message 3
  - crc8_msg4 - CRC of message 4
- CPU to CLA1 Message RAM
  - msg1 - Test message 1
  - msg2 - Test message 2
  - msg3 - Test message 3
  - msg4 - Test message 4

6.5 CRC8 Table-generation Algorithm

This example will generate the lookup table for an 8bit CRC checker with the polynomial 0x07.

Watch Variables

- table - Lookup table

Memory Allocation

- CLA1 Data RAM 1(RAML2)
  - table - CRC Lookup table

6.6 Determinant of a 3X3 Matrix

In this example, Task 1 of the CLA will calculate the determinant of a 3x3 matrix.

Watch Variables

- fDet - Determinant of the 3x3 matrix

Memory Allocation

- CLA1 to CPU Message RAM
  - fDet - Determinant of the 3x3 matrix
- CPU to CLA1 Message RAM
  - x - 3x3 input matrix

6.7 Division: Newton Raphson Approximation

In this example, Task 1 of the CLA will divide two input numbers using multiple approximations in the Newton Raphson method

Watch Variables

- Num - Numerator of input
- Den - Denominator of input
- Res - Result of the division operation

Memory Allocation

- CLA1 to CPU Message RAM
  - Res - Result of the division operation
- CPU to CLA1 Message RAM
  - Num - Numerator of input
  - Den - Denominator of input

6.8 10^X using a lookup table

In this example, Task 1 of the CLA will calculate the Xth power of 10 using a table lookup method.

Watch Variables

- Val - input
- ExpRes - Result of 10^{Val}

Memory Allocation

- CLA1 Math Tables (RAML2)
  - CLAexpTable - Lookup table
- CLA1 to CPU Message RAM
  - ExpRes - Result of the exponentiation operation
- CPU to CLA1 Message RAM
  - Val - Test input

6.9 $e^{\frac{A}{B}}$ using a lookup table

In this example, Task 1 of the CLA will divide two input numbers using multiple approximations in the Newton Raphson method and then calculate the exponent of the result using a lookup table

Watch Variables

- Num - Numerator of input
- Den - Denominator of input
- ExpRes - Result of $e^{\frac{Num}{Den}}$

Memory Allocation

- CLA1 Math Tables (RAML2)
  - CLAexpTable - Lookup table
- CLA1 to CPU Message RAM
  - ExpRes - Result of the exponentiation operation
- CPU to CLA1 Message RAM
  - Num - Numerator of input
  - Den - Denominator of input

6.10 Finite Impulse Response Filter

A 5 tap FIR filter is implemented in Task 1 of the CLA.

Watch Variables

- xResult - Result of the FIR operation
- xAdcInput - Simulated ADC input

Memory Allocation

- CLA1 Data RAM 1 (RAML2)
  - fCoeffs - Filter Coefficients
  - fDelayLine - Delay line memory elements
- CLA1 to CPU Message RAM
  - xResult - Result of the FIR operation
- CPU to CLA1 Message RAM
  - xAdcInput - Simulated ADC input

6.11 2 Pole 2 Zero Infinite Impulse Response Filter

This example implements a Transposed Direct Form II IIR filter, commonly known as a Biquad. The input vector is a software simulated noisy signal that is fed to the biquad one sample at a time, filtered and then stored in an output buffer for storage.

Watch Variables

- fBiquadoutput

Memory Allocation

- CLA1 Data RAM 1 (RAML2)
  - S1_A - Feedback coefficients
  - S1_B - Feedforward coefficients
- CLA1 to CPU Message RAM
  - yn - Output of the Biquad
- CPU to CLA1 Message RAM
  - xn - Sample input to the filter

The coefficients in this example were generated in MATLAB using "**fdatool**" as shown in fig.6.1. The Frequency specification shown in the image is for demonstration purposes only and was not used to generate the coefficients in the example code.

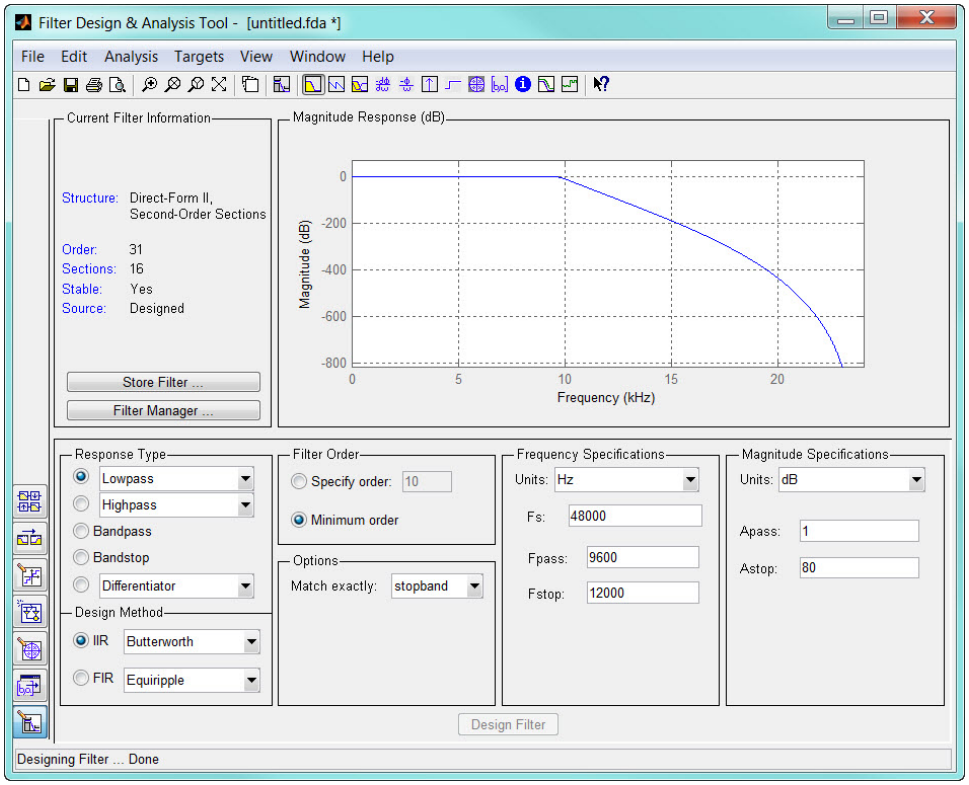

Figure 6.1: MATLAB: fdatool

Once the parameters are set and the filter designed, the user can view the coefficients from the analysis menu shown below

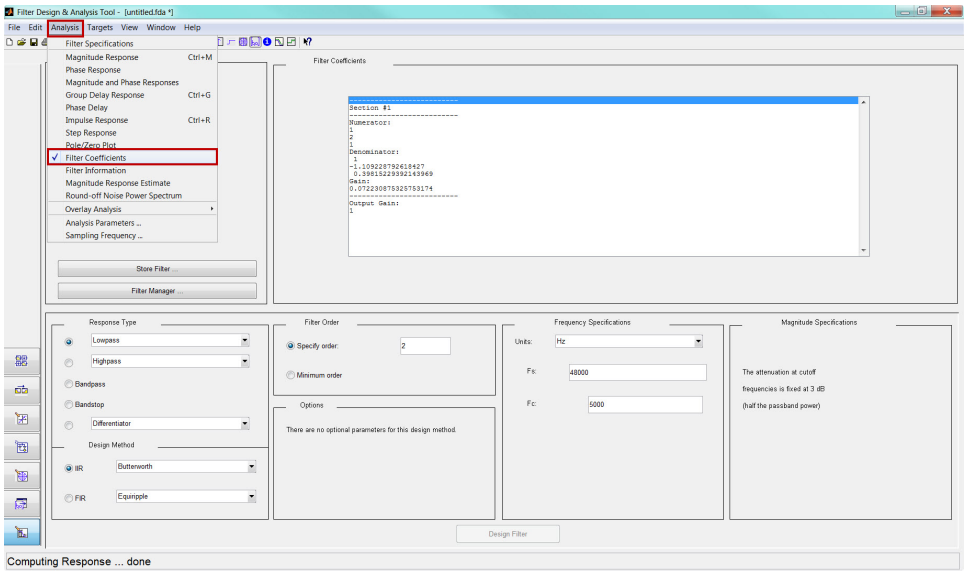

Figure 6.2: Viewing the coefficients

MATLAB implements IIR filters in cascading "**Second Order Section**" of the form

$$\frac{Y(z)}{X(z)} = \frac{b(0) + b(1)z^{-1} + b(2)z^{-2}}{a(0) + a(1)z^{-1} + a(2)z^{-2}} \tag{6.1}$$

In the code, however, the equation is implemented as follows

$$y(n) = \frac{1}{a(0)} \times (b(0)x(n) + b(1)x(n-1) + b(2)x(n-2) + (-a(1))y(n-1) + (-a(2))y(n-2)) \tag{6.2}$$

The "a" coefficients, with the exception of a(0), must be negated. In the example, the original coefficients were

Numerator = [0.02008, 0.04017, 0.02008]
Denominator = [1.0, -1.56102, 0.64135]

The "a" coefficients(except a(0)) are negated to fit the second-order structure implemented in code.

```
#pragma DATA_SECTION(S1_B,"Cla1DataRam1")
float S1_B[]={0.02008, 0.04017, 0.02008};
#pragma DATA_SECTION(S1_A,"Cla1DataRam1")
float S1_A[]={1.0, 1.56102, -0.64135};
```

Note that a(0) is hardcoded to a constant(1) and, as such, is not required in the coefficient array. It is present only to facilitate readability of the code.

6.12 Logic Test

In this example, Task 1 of the CLA implements a set of logic tests. More information about these logic statements can be found at:

http://graphics.stanford.edu/~seander/bithacks.html#OperationCounting

Watch Variables

- cla_pass_count - Logic test pass count
- cla_fail_count - Logic test fail count

Memory Allocation

- CLA1 to CPU Message RAM
  - cla_pass_count - Logic test pass count
  - cla_fail_count - Logic test fail count

6.13 Matrix Multiplication

Task 1 multiplies two 3x3 matrices

Watch Variables

- x - 3X3 Input Matrix
- y - 3X3 Input Matrix
- z - Result of the matrix multiplication

Memory Allocation

- CLA1 to CPU Message RAM
  - z - Result of the matrix multiplication
- CPU to CLA1 Message RAM
  - x - 3X3 Input Matrix
  - y - 3X3 Input Matrix

6.14 Matrix Transpose

Task 1 calculates the transpose of a 3x3 matrices.

Watch Variables

- x - 3X3 Input Matrix
- z - Transposed Matrix

Memory Allocation

- CLA1 to CPU Message RAM
  - z - Transposed Matrix
- CPU to CLA1 Message RAM
  - x - 3X3 Input Matrix

6.15 Primes

Task 1 calculates the set of prime numbers up to a length defined by the user

Watch Variables

- in - Input test vector
- out - Set of primes

Memory Allocation

- CLA1 Data RAM 1 (RAML2)
  - out - Set of primes

6.16 Shell Sort

Task 1 will perform the shell sort iteratively. Task 2 will do the same with mswapf intrinsic and Task 3 will also implement an in-place sort on an integer vector

Watch Variables

- vector3 - Input/Output to task 3(in-place sorting)
- vector1_sorted - Sorted output Task 1
- vector2_sorted - Sorted output Task 2
- vector1 - Input vector to task 1
- vector2 - Input vector to task 2

Memory Allocation

- CLA1 Data RAM 1 (RAML2)
  - vector3 - Input/Output to task 3(in-place sorting)
- CLA1 to CPU Message RAM
  - vector1_sorted - Sorted output Task 1
  - vector2_sorted - Sorted output Task 2
- CPU to CLA1 Message RAM
  - vector1 - Input vector to task 1
  - vector2 - Input vector to task 2

6.17 Square Root

Task 1 calculates the square root of a number using multiple iterations of the Newton-Raphson approximation

Watch Variables

- fVal - Input value
- fResult - $\sqrt{fVal}$

Memory Allocation

- CLA1 to CPU Message RAM
  - fResult - $\sqrt{fVal}$
- CPU to CLA1 Message RAM
  - fVal - Input value

6.18 Vector Inverse

Task 1 calculates the element-wise inverse of a vector while Task 2 calculates the element-wise inverse of a vector and saves the result in the same vector

Watch Variables

- vector1 - Input vector to task 1
- vector1_inverse - Inverse of input vector1
- vector2 - Input/Output vector for task 2

Memory Allocation

- CLA1 Data RAM 1 (RAML2)
  - vector2 - Input/Output vector for task 2
- CLA1 to CPU Message RAM
  - vector1_inverse - Inverse of input vector1
- CPU to CLA1 Message RAM
  - vector1 - Input vector to task 1

6.19 Vector Maximum

Task 1 calculates the vector max moving backward through the array.

Task 2 calculates the vector max moving forward through the array.

Task 3 calculates the vector max using the ternary operator.

Task 2 calculates the vector max using min/max intrinsics.

Watch Variables

- vector1 - Input vector to task 1
- vector2 - Input vector to task 2
- vector3 - Input vector to task 3
- vector4 - Input vector to task 4
- max1 - Maximum value in vector 1
- index1 - Index of the maximum value in vector 1
- max2 - Maximum value in vector 2
- index2 - Index of the maximum value in vector 2
- max3 - Maximum value in vector 3
- index3 - Index of the maximum value in vector 3
- max4 - Maximum value in vector 4
- min4 - Minimum value in vector 4

Memory Allocation

- CLA1 to CPU Message RAM
  - max1 - Maximum value in vector 1
  - index1 - Index of the maximum value in vector 1
  - max2 - Maximum value in vector 2
  - index2 - Index of the maximum value in vector 2
  - max3 - Maximum value in vector 3
  - index3 - Index of the maximum value in vector 3
  - max4 - Maximum value in vector 4
  - min4 - Minimum value in vector 4
- CPU to CLA1 Message RAM
  - vector1 - Input vector to task 1
  - vector2 - Input vector to task 2
  - vector3 - Input vector to task 3
  - vector4 - Input vector to task 4
  - length1 - Length of vector 1
  - length2 - Length of vector 2

6.20 Vector Minimum

Task 1 calculates the vector min moving backward through the array.

Task 2 calculates the vector min moving forward through the array.

Task 3 calculates the vector min using the ternary operator.

Watch Variables

- vector1 - Input vector to task 1
- vector2 - Input vector to task 2
- vector3 - Input vector to task 3
- min - Minimum value in vector 1
- index1 - Index of the minimum value in vector 1
- min2 - Minimum value in vector 2
- index2 - Index of the minimum value in vector 2
- min3 - Minimum value in vector 3
- index3 - Index of the minimum value in vector 3

Memory Allocation

- CLA1 to CPU Message RAM
  - min1 - Minimum value in vector 1
  - index1 - Index of the minimum value in vector 1
  - min2 - Minimum value in vector 2
  - index2 - Index of the minimum value in vector 2
  - min3 - Minimum value in vector 3

- index3 - Index of the minimum value in vector 3

- CPU to CLA1 Message RAM
  - vector1 - Input vector to task 1
  - vector2 - Input vector to task 2
  - vector3 - Input vector to task 3
  - length1 - Length of vector 1
  - length2 - Length of vector 2
  - length3 - Length of vector 3

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 2805x 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.

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

A.2 2805x 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.

The interrupts in a 2805x 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 2805x, 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 2805x, 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 2805x 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 2805x, such interrupts are mapped to INT6, INT8, and INT9, which are the next lowest hardware priority.

A.3 Software Prioritization of Interrupts - The F2805x 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 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 example, the user must first assign the desired global priority levels and group priority levels.

This is done in the F2805x_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 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 $Y = mX + b$:

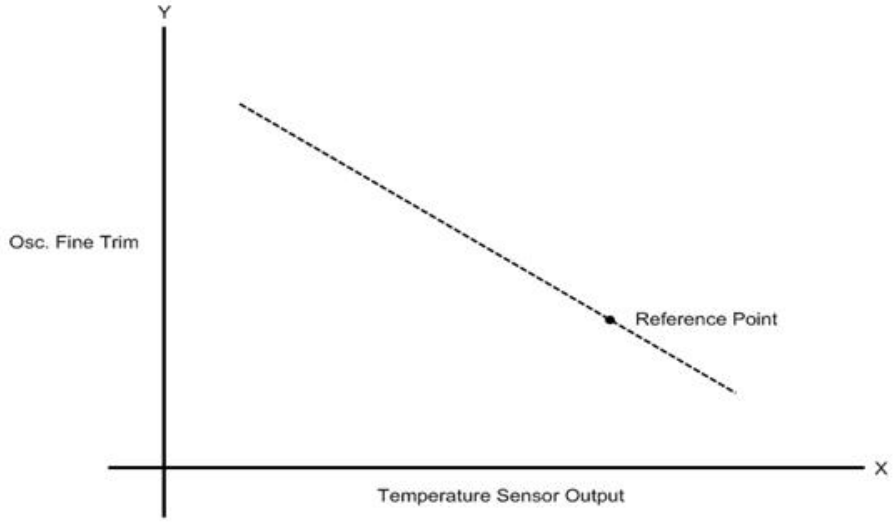

Figure B.1: Oscillator Reference

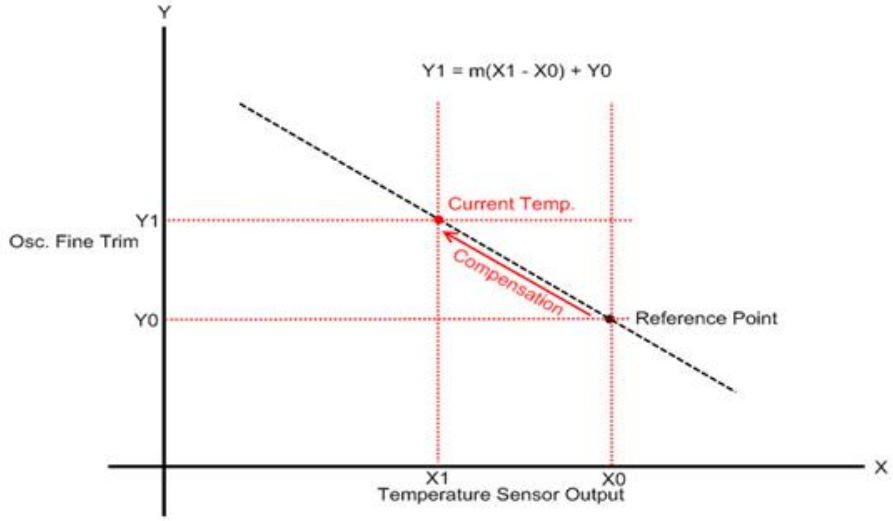

Figure B.2: Oscillator Fine Trim Compensation for change in Temperature

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

B.2.1 OTP Functions

The following functions in *F<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.

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