C280x C/C++ Header Files and Peripheral Examples

1 Introduction: 2
1.1 Where Files are Located (Directory Structure) 3
2 Understanding The Peripheral Bit-Field Structure Approach 6
2.1 Traditional #define approach: 6
2.2 Bit-field and Structure Approach: 7
2.2.1 Peripheral Register Structures 7
2.3 Adding Bit-Fields 9
2.3.1 Read-Modify-Write Considerations When Using Bit-Fields: 10
2.3.2 Code-Size Considerations when using Bit-Fields: 11
3 Peripheral Example Projects 12
3.1 Getting Started 12
3.2 Example Program Structure 15
3.2.1 Include Files 15
3.2.2 Source Code 16
3.2.3 Linker Command Files 16
3.3 Example Program Flow 18
3.4 Included Examples: 19
3.5 Executing the Examples From Flash 21
4 Steps for Incorporating the Header Files and Sample Code 24
4.1 Before you begin 24
4.2 Including the DSP280x Peripheral Header Files 24
4.3 Including Common Example Code 28
5 Troubleshooting Tips & Frequently Asked Questions 30
5.1 Effects of read-modify-write instructions. 32
5.1.1 Registers with multiple flag bits in which writing a 1 clears that flag. 32
5.1.2 Registers with Volatile Bits. 33
6 Migration Tips from DSP281x to DSP280x 34
7 Packet Contents: 36
7.1 Header File Support – DSP280x_headers 36
7.1.1 DSP280x Header Files – Main Files 36
7.1.2 DSP280x Header Files – Peripheral Bit-Field and Register Structure Definition Files 37
7.1.3 Code Composer .gel Files 37
7.1.4 Variable Names and Data Sections 38
7.2 Common Example Code – DSP280x_common 39
7.2.1 Peripheral Interrupt Expansion (PIE) Block Support 39
7.2.2 Peripheral Specific Files 40
7.2.3 Utility Function Source Files 41
7.2.4 Example Linker .cmd files 41

1 Introduction:

The DSP280x C/C++ peripheral header files and example projects facilitate writing in C/C++ Code for the Texas Instruments '280x DSPs. 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.

- Learning Tool:

  This download includes several example Code Composer Studio™[†] projects for a '280x development platform. One such platform is the eZdsp™[††] F2808 USB from Spectrum Digital Inc. (www.spectrumdigital.com).

  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 the '2801 and '2806 devices by simply changing the memory allocation in the linker command file.

- 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:

- Overview of the bit-field structure approach used in the DSP280x C/C++ peripheral header files.
- Overview of the included peripheral example projects.
- Steps for integrating the peripheral header files into a new or existing project.
- Troubleshooting tips and frequently asked questions.
- Migration tips for users moving from the DSP281x header files to the DSP280x 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 2808 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.

[†] Code Composer Studio is a trademark of Texas Instruments (www.ti.com).
[††] eZdsp is a trademark of Spectrum Digital Inc (www.spectrumdigital.com).
Trademarks are the property of their respective owners.

1.1 Revision History

V1.10

Changes to Header Files:

a) **DSP280x_EPwm.h:**
Added the following Hi-Resolution ePWM (HiRes) registers:

Register Name	Address offset	Description
TBPHSHR	0x0002	HiRes extension of phase TBPHS register
CMPAHR	0x0008	HiRes extension of compare A CMPA register
HRCNFG	0x0020	HiRes Configuration register

The header file definition of the CMPA and TBPHS registers have been changed to a union with the HiRes extension registers to provide for .half (16-bit) and .all (32-bit) accesses. This was done to allow 16-bit access to CMPA and TBPHS as well as 32-bit access to the extended registers CMPA:CMPAHR and TBPHS:TBPHSHR.

Accessing the registers is done as follows:

```
EPwm1Regs.CMPA.half.CMPA = 0x1234;      // Access 16-bit CMPA register
EPwm1Regs.CMPA.half.CMPAHR = 0x5600;    // Access only HiRes extension
EPwm1Regs.CMPA.all = 0x12345600;        // 32-bit write CMPA:CMPAHR

EPwm1Regs.TBPHS.half.TBPHS = 0x1234;    // Access 16-bit TBPHS register
EPwm1Regs.TBPHS.half.TBPHSHR = 0x5600;  // Access only HiRes extension
EPwm1Regs.TBPHS.all = 0x12345600;       // 32-bit write TBPHS:TBPHSHR
```

Note, accesses to COMPB remain as is and do not require .all:

```
EPwm1Regs.CMPB = 0x5000;
```

This change requires users migrating from the DSP280x 1.00 header files to make modifications to their ePWM code. The changes required are as follows:

	DSP280x V1.00	DSP280x V1.10
Access CMPA	EPwm1Regs.CMPA=VALUE;	EPwm1Regs.CMPA**.half.CMPA**=VALUE;
Access TBPHS	EPwm1Regs.TBPHS=VALUE;	Epwm1Regs.TBPHS**.half.TBPHS**=VALUE;

Note:

The HiRes extension is not available on all ePWM modules. The register file definition used, however, is identical for all ePWM modules. Thus, HiRes register definitions will appear even if the ePWM module does not include the HiRes extension.

b) **DSP280x_EPwm.h**
Made the following changes to the DBCTL register (Dead Band Control)

- ❑ Changed the MODE bit-field name to OUT_MODE
- ❑ Changed reserved bits 5:4 to the IN_MODE bit field

This corresponds to a silicon change made on Flash devices as of Rev A silicon.

c) **DSP280x_ECap.h**
The STOPVALUE bit-field in the ECCTL2 register was changed to STOP_WRAP. This corresponds to a silicon change made on Flash devices as of Rev A silicon. This register was previously used as a stop value for one-shot capture mode. It is now also used to specify a wrap value when using continuous capture mode.

d) **DSP280x_EQep.h**
Added UPEVNT (bit 7) to the QEPSTS register. This reflects changes made as of F280x Rev A devices.

e) **DSP280x_Spi.h**
Added definitions for SPI-B, SPI-C, SPI-D.

f) **DSP280x_Headers_nonBIOS.cmd and DSP280x_Headers_BIOS.cmd**
Updated the memory space allocated for ePWM1 – ePWM6 registers to include the HiRes configuration register (HRCNFG).

g) **DSP280x_Peripheral.gel:**
The hotmenu item for `EPwm2Regs` was repeated twice. Removed the duplicate instance.

h) **DSP280x_EPwm_defines.h:**
Added useful #defines for the HiRes.

Changes to examples:

a) The root of the default path in all example project files was changed from C:\tidcs\c28\DSP280x\v100\ to C:\tidcs\c28\DSP280x\v110\ to reflect the version change.

b) ecap_capture_pwm example:
Updated the function that initializes the eCAP peripheral.

c) Updated the CMPA and TBPHS register accesses in all ePWM and eQEP examples to use the .half of the union introduced for HiRes register extension.

d) Added two ePWM with HiRes extension example.

e) Updates to examples as required for changes described above to the header files.

V1.00

- ❑ This version was the first customer release of the DSP280x header files and examples.

1.2 Where Files are Located (Directory Structure)

As installed, the *C280x C/C++ Header Files and Peripheral Examples* is partitioned into a well-defined directory structure. By default, the source code is installed into the c:\tidcs\c28\DSP280x\<version> directory.

Table 1 describes the contents of the main directories used by DSP280x:

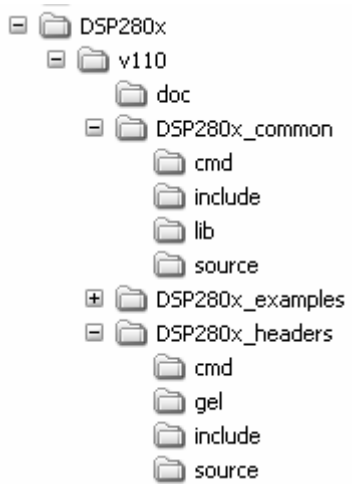


Table 1. DSP280x Main Directory Structure

Directory	Description
<base>	Base install directory. By default this is c:\tidcs\c28\DSP280x\v110. For the rest of this document <base> will be omitted from the directory names.
<base>\doc	Documentation including the revision history from the previous release.
<base>\DSP280x_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. Integrating the header files into a new or existing project is described in Section 4.
<base>\DSP280x_examples	Example Code Composer Studio projects based on the DSP280x header files. These example projects illustrate how to configure many of the '280x on-chip peripherals. An overview of the examples is given in Section 3.
<base>DSP280x_common	Common source files shared across a number of the DSP280x example projects to illustrate how to perform tasks using the DSP280x header file approach. Use of these files is optional, but may be useful in new projects. A list of these files is in Section 6.

Under the *DSP280x_headers* and *DSP280x_common* directories the source files are further broken down into sub-directories each indicating the type of file. Table 2 lists the sub-directories and describes the types of files found within each:

Table 2. DSP280x Sub-Directory Structure

Sub-Directory	Description
DSP280x_headers\cmd	Linker command files that allocate the bit-field structures described in Section 2.
DSP280x_headers\source	Source files required to incorporate the header files into a new or existing project.
DSP280x_headers\include	Header files for each of the 280x on-chip peripherals.
DSP280x_common\cmd	Example memory command files that allocate memory on the '280x devices.
DSP280x_common\include	Common .h files that are used by the peripheral examples.
DSP280x_common\source	Common .c files that are used by the peripheral examples.
DSP280x_common\lib	Common library (.lib) files that are used by the peripheral examples.

2 Understanding The Peripheral Bit-Field Structure Approach

The *DSP280x C/C++ Header Files and Peripheral Examples in C* use a bit-field structure approach for mapping and accessing peripheral registers on the TI ‘280x based DSPs. This section will describe this approach and compare it to the more traditional #define approach.

2.1 Traditional #define approach:

The traditional approach for accessing registers in C-code has been to use #define macros to create an address label for each register. For example:

```
/*************************************************************
* Traditional header file
*************************************************************/

                                                    // Memory Map
                                                    // Addr   Register
#define CPUTIMER0_TIM (volatile unsigned long *)0x0C00  // 0xC00  Timer0 Count Low
                                                    // 0xC01  Timer0 Count High
#define CPUTIMER0_TIM (volatile unsigned long *)0x0C02  // 0xC02  Timer0 Period Low
                                                    // 0xC03  Timer0 Period High
#define CPUTIMER0_TIM (volatile unsigned int *)0x0C04   // 0xC04  Timer0 Control
                                                    // 0xC05  reserved
#define CPUTIMER0_TIM (volatile unsigned int *)0x0C06   // 0xC06  Timer0 Pre-scale Low
#define CPUTIMER0_TIM (volatile unsigned int *)0x0C07   // 0xC07  Timer0 Pre-scale High
```

This same #define approach would then be repeated for every peripheral register on every peripheral. Even if the peripheral were a duplicate, such as in the case of SCI-A and SCI-B, each register would have to be specified separately with its given address. The disadvantages to the traditional #define approach include:

- Bit-fields within the registers are not easily accessible.
- Cannot easily display bit-fields within the Code Composer Studio watch window.
- Cannot take advantage of Code Maestro, which is the auto-completion feature of Code Composer Studio.
- The header file developer cannot take advantage of re-use for duplicate peripherals.

2.2 Bit-field and Structure Approach:

The bit-field and structure approach uses C-code structures to group together all of the registers belonging to a particular peripheral. Each C-code structure is then memory mapped over the peripheral registers by the linker. This mapping allows the compiler to access the peripheral registers directly using the CPU's data page pointer (DP). In addition, bit-fields are defined for many registers allowing the compiler to read or manipulate single bit fields within a register.

2.2.1 Peripheral Register Structures

In Section 2.1 we defined the CPU Timer 0 registers using the traditional #define approach. In this section, we will define the same CPU Timer 0 registers, but instead will use C-code structures to group the CPU Timer registers together. The linker will then be used to map the structure over the CPU-Timer 0 registers in memory.

The following code example shows the C-Code structure that corresponds to a '280x CPU-Timer peripheral:

```
/*-----------------------------------------------------------------
* CPU-Timer header file using structures
-----------------------------------------------------------------*/

struct CPUTIMER_REGS
{
    Uint32 TIM;   // Timer counter register
    Uint32 PRD;   // Period register
    Uint16 TCR;   // Timer control register
    Uint16 rsvd1; // reserved
    Uint16 TPR;   // Timer pre-scale low
    Uint16 TPRH;  // Timer pre-scale high
};
```

Notice the following points:

- The register names appear in the same order as they are arranged in memory.
- Locations that are reserved in memory are held within the structure by a reserved variable (rsvd1, rsvd2 etc). The reserved structure members are not used except to hold the space in memory.
- Uint16 and Uint32 are typedefs for unsigned 16-bit and 32-bit values, respectively. In the case of the '28x, these are unsigned int and unsigned long. This is done for portability. The corresponding typedef statements can be found in the file: <base>\DSP280x_headers\include\DSP280x_Device.h.

The register file structure definition is then used to declare a variable that will be used to access the registers. This is done for each of the peripherals on the device. Multiple instances of the same peripheral use the same structure definition. For example, if there are three CPU-Timers on a device, then three variables of type volatile struct CPUTIMER_REGS can be created as:

```
/*-----------------------------------------------------------------------
* CPU-Timer header file using structures
-----------------------------------------------------------------------*/

volatile struct CPUTIMER_REGS CpuTimer0Regs;
volatile struct CPUTIMER_REGS CpuTimer1Regs;
volatile struct CPUTIMER_REGS CpuTimer2Regs;
```

The volatile keyword is important in the variable declaration. Volatile indicates to the compiler that the contents of the variable can be changed by hardware and thus the compiler will not optimize out code that uses a volatile variable.

Each variable corresponding to a peripheral register structure is then assigned to a data section using the compiler's DATA_SECTION #pragma. In the example shown below, the variable CpuTimer0Regs is assigned to the data section CpuTimer0RegsFile.

```
/*-----------------------------------------------------------------------
* DSP280x_headers\source\DSP280x_GlobalVariableDefs.c
-----------------------------------------------------------------------*/
/* Assign the variable CpuTimer0Regs to the CpuTimer0RegsFile output section
   using the #pragma compiler statement
   C and C++ use different forms of the #pragma statement
   When compiling a C++ program, the compiler will define __cplusplus automatically
*/

#ifdef __cplusplus                                      // used by C++
#pragma DATA_SECTION("CpuTimer0RegsFile")
#else                                                   // used by C
#pragma DATA_SECTION(CpuTimer0Regs,"CpuTimer0RegsFile");
#endif
volatile struct CPUTIMER_REGS CpuTimer0Regs;            // variable CpuTimer0Regs
                                                        // of type CPUTIMER_REGS
```

This data section assignment is repeated for each peripheral register structure variable for the device. With each structure assigned to its own data section, the linker is then used to map each data section directly to the memory mapped registers for that peripheral as shown below.

```
/*-----------------------------------------------------------------------
* DSP280x_headers\include\DSP280x_Headers_nonBIOS.cmd
-----------------------------------------------------------------------*/
MEMORY
{
   PAGE 1:
   CPU_TIMER0  : origin = 0x000C00, length = 0x000008     /* CPU Timer0 registers
}
SECTIONS
{
   CpuTimer0RegsFile : > CPU_TIMER0,  PAGE = 1
}
```

By mapping the variable directly to the same memory address of the peripheral registers, the user can now access the registers in C-code by simply accessing the required member of the variable. For example, to write to the CPU-Timer 0 TCR register, the user just has to access the TCR member of the CpuTimer0Regs variable:

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

CpuTimer0Regs.TCR.all = TSS_MASK; // Example of accessing the TCR register
```

2.3 Adding Bit-Fields

It is often desirable to access bit fields within the registers directly. With the bit-field structure approach *C280x C/C++ Header Files and Peripheral Examples in C* provides bit-field definitions for many of the on-chip peripheral registers. For example, a bit-field definition can be established for each of the CPU-Timer registers. The bit-field definitions for the CPU-Timer control register is shown below:

```
/************************************************************************
* DSP280x_headers\include\DSP280x_CpuTimers.h CPU-Timer header file
************************************************************************/

struct  TCR_BITS {          // bits  description
   Uint16    rsvd1:4;       // 3:0   reserved
   Uint16    TSS:1;         // 4     Timer Start/Stop
   Uint16    TRB:1;         // 5     Timer reload
   Uint16    rsvd2:4;       // 9:6   reserved
   Uint16    SOFT:1;        // 10    Emulation modes
   Uint16    FREE:1;        // 11
   Uint16    rsvd3:2;       // 12:13 reserved
   Uint16    TIE:1;         // 14    Output enable
   Uint16    TIF:1;         // 15    Interrupt flag
};
```

A union declaration is then used to allow the register to be accessed in terms of the defined bit field structure or as a whole 16-bit or 32-bit quantity. For example, the timer control register union definition is shown below:

```
/************************************************************************
* DSP280x_headers\include\DSP280x_CpuTimers.h CPU-Timer header file
************************************************************************/

union TCR_REG {
   Uint16           all;
   struct TCR_BITS  bit;
};
```

Once bit-field and union definitions are established for each of the registers, the CPU-Timer register structure can be re-written in terms of the union definitions.

```
/*-----------------------------------------------------------------
* DSP280x_headers\include\DSP280x_CpuTimers.h CPU-Timer header file
-----------------------------------------------------------------*/

struct CPUTIMER_REGS
{
    union TIM_GROUP TIM;   // Timer counter register
    union PRD_GROUP PRD;   // Period register
    union TCR_REG   TCR;   // Timer control register
    Uint16          rsvd1; // reserved
    union TPR_REG   TPR;   // Timer pre-scale low
    union TPRH_REG  TPRH;  // Timer pre-scale high
};
```

In C-code the CpuTimer register can now be accessed either by bit-fields or as a single quantity:

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

CpuTimer0Regs.TCR.bit.TSS = 1;    // Example of accessing a single bit
CpuTimer0Regs.TCR.all = TSS_MASK; // Example of accessing the whole register
```

The bit-field structure approach has the following advantages:

- Bit-fields can be manipulated without the user needing to determine mask values
- Register files and bit-fields can be viewed in the Code Composer Studio watch window
- When using Code Composer Studio, the editor will prompt you with a list of possible structure/bit field elements as you type. This auto completion feature makes it easier to code without having to refer to documentation for the register and bit field names.

2.3.1 Read-Modify-Write Considerations When Using Bit-Fields:

When writing to a single bit-field within a register, a read-modify-write operation is performed in hardware. That is, the register contents are read, the single bit field is modified and the whole register is written back. This can happen as quickly as a single cycle on the '28x.

When the write-back occurs, other bits within the register will be written to with the same value as what was read. If this value was a 1, and the bit is "write one to clear" then this read-modify write operation will have the effect of clearing that bit which may not be desired.

Some registers do not have unions defined because it is not recommended to access them in this manner. Exceptions are made when it may be beneficial to poll (read) single bits within the registers. This includes:

- Registers with write-1-to-clear bits.
- Registers with bits which must be written to in a particular manner whenever accessing the register such as the watchdog control register.

Registers that **do not have** bit-field and union definitions are accessed without the .bit or .all designations. For example:

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

SysCtrlRegs.WDCR = 0x0068;
```

2.3.2 Code-Size Considerations when using Bit-Fields:

Using the bit-field definitions to access registers results in code that is easy to read, easy to modify, and easy to maintain. This approach is also efficient when accessing a single bit within a register or when polling a bit. Keep in mind, however, that if a number of accesses to one register are made, then using the defined .bit fields for each access may result in more code then using .all to write to the register all at once. For example:

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

CpuTimer0Regs.TCR.bit.TSS = 1;      // 1 = Stop timer
CpuTimer0Regs.TCR.bit.TRB = 1;      // 1 = reload timer
CpuTimer0Regs.TCR.bit.SOFT = 1;     // Timer Free Run
CpuTimer2Regs.TCR.bit.FREE = 1;     // Timer Free Run
CpuTimer2Regs.TCR.bit.TIE = 1;      // 1 = Enable Timer Interrupt
```

This results in very readable code that is easy to modify. The penalty is slight code overhead. If code size is of greater concern then use the .all structure to write to the register all at once.

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

   CpuTimer0Regs.TCR.all = TCR_MASK;
```

3 Peripheral Example Projects

In the *DSP280x_examples*\ directory of *C280x C/C++ Header Files and Peripheral Examples in C* there are several example projects that use the DSP280x header files to configure the on-chip peripherals. A listing of the examples is included in Section 3.4.

3.1 Getting Started

To get started, follow these steps to load the DSP280x CPU-Timer example. Other examples are set-up in a similar manner.

1. **Have a 280x hardware platform, such as the eZdsp F2808 USB, connected to a host with Code Composer Studio installed.**

   NOTE: As supplied, the example projects are built for the '2808 device. If you are using another device within the '280x family (ie '2806 or 2801), the memory definition in the linker command file (.cmd) will need to be modified and the project rebuilt.

2. **Load the example's GEL file (.gel) or Project file (.pjt).**

   Each example includes a Code Composer Studio GEL file to help automate loading of the project, compiling of the code and populating of the watch window. Alternatively, the project file itself (.pjt) can be loaded instead of using the included GEL file.

   To load the CPU-Timer example's GEL file follow these steps:

   a. In Code Composer Studio: *File->Load GEL*

   b. Browse to the CPU Timer example directory: *DSP280x_examples\cpu_timer*

   c. Select *Example_280xCpuTimer.gel* and click on *open*.

   d. From the Code Composer GEL pull-down menu select

      DSP280x CpuTimerExample-> Load_and_Build_Project

      This will load the project and build compile the project.

3. **Review the comments at the top of the main source file: Example_280xCpuTimer.c.**

   A brief description of the example and any assumptions that are made and any external hardware requirements are listed in the comments at the top of the main source file of each example. In some cases you may be required to make external connections for the example to work properly.

4. Perform any hardware setup required by the example.

Perform any hardware setup indicated by the comments in the main source. The DSP280x 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 3 shows a listing of the boot mode pin settings for your reference. Refer to the documentation for your hardware platform for information on configuring the boot mode pins. For more information on the '280x boot modes refer to the *TMS320x280x Boot ROM Reference Guide (SPRU722)*.

Table 3. 280x Boot Mode Settings

GPIO18	GPIO29	GPIO34	Mode
1	1	1	Boot to flash 0x3F7FF6
1	1	0	Call SCI-A boot loader
1	0	1	Call SPI-A boot loader
1	0	0	Call I2C boot loader
0	1	1	Call eCAN-A boot loader
0	1	0	Boot to M0 SARAM 0x000000
0	0	1	Boot to OTP 0x3D7800
0	0	0	Call parallel boot loader

5. Load the code

Once any hardware configuration has been completed, from the Code Composer GEL pull-down menu select

DSP280x CpuTimerExample-> Load_Code

This will load the .out file into the 28x device, populate the watch window with variables of interest, reset the part and execute code to the start of the main function. The GEL file is setup to reload the code every time the device is reset so if this behavior is not desired, the GEL file can be removed at this time. To remove the GEL file, right click on its name and select *remove*.

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

7. Experiment, modify, re-build the example.

If you wish to modify the examples it is suggested that you make a copy of the entire DSP280x 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 3.2 and 3.3 describe the structure and flow of the examples in more detail.

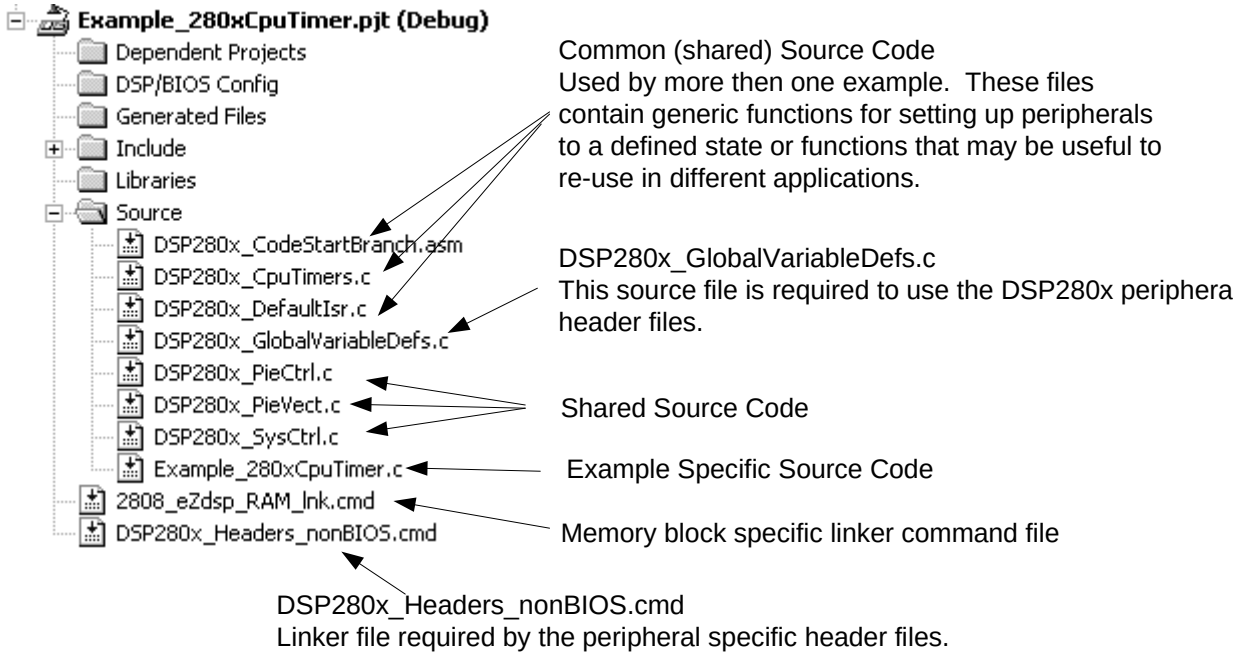
8. **When done, remove the example's GEL file and project from Code Composer Studio.**

   To remove the GEL file, right click on its name and select *remove*.

The examples use the header files in the *DSP280x_headers* directory and shared source in the *DSP280x_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.

3.2 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.

3.2.1 Include Files

All of the example source code #include two header files as shown below:

```
/*************************************************************
* DSP280x_examples\cpu_timer\Example_280xCpuTimer.c
*************************************************************/

#include "DSP280x_Device.h"     // DSP280x Headerfile Include File
#include "DSP280x_Examples.h"   // DSP280x Examples Include File
```

- **DSP280x_Device.h**

  This header file is required to use the DSP280x peripheral 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>\DSP280x_headers\include* directory.

- **DSP280x_Examples.h**

  This header file defines parameters that are used by the example code. This file is not required to use just the DSP280x peripheral header files but is required by some of the common source files. This file is found in the <*base*>*DSP280x_common**include* directory.

3.2.2 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.

- **DSP280x_GlobalVariableDefs.c**

  Any project that uses the DSP280x 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*>*DSP280x_headers**source* directory.

- **Example specific source code:**

  Files that are specific to a particular example have the prefix Example_280x on their filename. For example Example_280xCpuTimer.c is specific to the CPU Timer example and not used for any other example. Example specific files are located in the <*base*>*DSP280x_examples*\<*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 DSP280x_shared\source directory. Users may choose to incorporate none, some, or the entire shared source into their own new or existing projects.

3.2.3 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 DSP280x peripheral header files.

- **Memory block linker allocation:**

The linker files shown in Table 4 are used to assign sections to memory blocks on the device. These linker files are located in the <*base*>*DSP280x_common**cmd* directory. Each example will use one of the following files depending on the memory used by the example.

Table 4. Included Memory Linker Command Files

Memory Linker Command File Examples	Location	Description
2808_eZdsp_RAM_lnk.cmd	DSP280x_common\cmd	eZdsp F2808 USB memory map that only allocates SARAM locations. No Flash, OTP, or CSM password protected locations are used. This linker command file is used for most of the examples.
F2808.cmd	DSP280x_common\cmd	F2808 memory linker command file. Includes all Flash, OTP and CSM password protected memory locations.
F2806.cmd	DSP280x_common\cmd	F2806 memory linker command file.
F2801.cmd	DSP280x_common\cmd	F2801 memory linker command file.

- **DSP280x header file structure data section allocation:**

  Any project that uses the DSP280x 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 5.

Table 5. DSP280x Peripheral Header Linker Command File

DSP280x Peripheral Header File Linker Command File	Location	Description
DSP280x_Headers_BIOS.cmd	DSP280x_headers\cmd	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 4.2.
DSP280x_Headers_nonBIOS.cmd	DSP280x_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 4.2.

3.3 Example Program Flow

All of the example programs follow a similar recommended flow for setting up the 280x devices. Figure 1 outlines this basic flow:

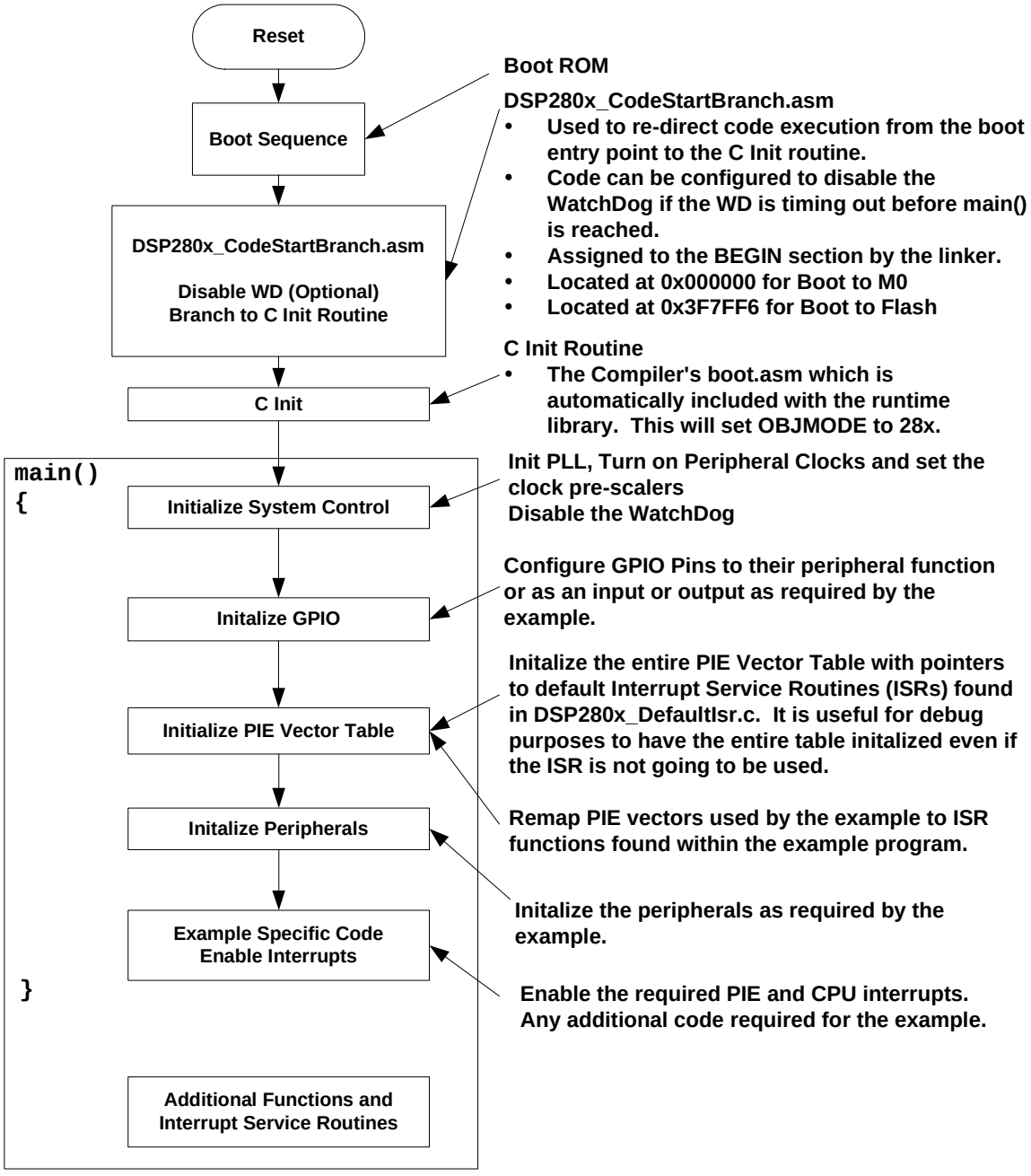


Figure 1. Flow for Example Programs

3.4 Included Examples:

Table 6. Included Examples

Example	Description
adc_seq_ovd_tests	ADC test using the sequencer override feature.
adc_seqmode_test	ADC Seq Mode Test. Channel A0 is converted forever and logged in a buffer
adc_soc	ADC example to convert two channels: ADCINA3 and ADCINA2. Interrupts are enabled and PWM1 is configured to generate a periodic ADC SOC on SEQ1.
cpu_timer	Configures CPU Timer0 and increments a count each time the ISR is serviced.
ecan_a_to_b_xmit	Transmit from eCANa to eCANb
ecan_back2back	eCAN self-test mode example. Transmits eCAN data back-to-back at high speed without stopping.
ecap_apwm	This example sets up the alternate eCAP pins in the APWM mode
ecap_capture_pwm	Captures the edges of a ePWM signal.
epwm_deadband	Example deadband generation via ePWM3
epwm_timer_interrupts	Starts ePWM1-ePWM6 timers. Every period an interrupt is taken for each ePWM.
epwm_trip_zone	Uses the trip zone signals to set the ePWM signals to a particular state.
epwm_up_aq	Generate a PWM waveform using an up count time base ePWM1-ePWM3 are used.
epwm_updown_aq	Generate a PWM waveform using an up/down time base. ePWM1 – ePWM3 are used.
eqep_freqcal	Frequency cal using eQEP1
eqep_pos_speed	Pos/speed calculation using eQEP1
external_interrupt	Configures GPIO0 as XINT1 and GPIO1 as XINT2. The interrupts are fired by toggling GPIO30 and GPIO31 which are connected to XINT1 (GPIO0) and XINT2 (GPIO1) externally by the user.
flash	ePWM timer interrupt project moved from SARAM to Flash. Includes steps that were used to convert the project from SARAM to Flash. Some interrupt service routines are copied from FLASH to SARAM for faster execution.
gpio_setup	Three examples of different pinout configurations.
gpio_toggle	Toggles all of the I/O pins using different methods – DATA, SET/CLEAR and TOGGLE registers. The pins can be observed using an oscilloscope.
hires_epwm	Sets up ePWM1-ePWM4 and controls the edge of output A using the HiRes extension. Both rising edge and falling edge are controlled.
hires_epwm_slider	This is the same as the hires_epwm example except the control of CMPAHR is now controlled by the user via a slider bar. The included .gel file sets up the slider.
i2c_eeprom	Communicate with the EEPROM on the eZdsp F2808 USB platform via I2C
sci_autobaud	Externally connect SCI-A to SCI-B and send data between the two peripherals. Baud lock is performed using the autobaud feature of the SCI. This test is repeated for different baud rates.
sci_echoback	SCI-A example that can be used to echoback to a terminal program such as hyperterminal. A transceiver and a connection to a PC is required.
scia_loopback	SCI-A example code that uses the loop-back test mode of the SCI module to send characters This example uses bit polling and does not use interrupts.
scia_loopback_interrupts	SCI-A example code that uses the internal loop-back test mode to transfer data through SCI-A. Interrupts and FIFOs are both used in this example.
spi_loopback	SPI-A example that uses the peripherals loop-back test mode to send data.
spi_loopback_interrupts	SPI-A example that uses the peripherals loop-back test mode to send data. Both interrupts and FIFOs are used in this example.

sw_prioritized_interrupts	The standard hardware prioritization of interrupts can be used for most applications. This example shows a method for software to re-prioritize interrupts if required.
watchdog	Illustrates feeding the dog and re-directing the watchdog to an interrupt.

3.5 Executing the Examples From Flash

Most of the DSP280x examples execute from SARAM in “boot to SARAM” mode. One example, *DSP280x_examples\Flash*, 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 2808_eZdsp_RAM_lnk.cmdfrom the project and add F2808.cmd, F2806.cmd or F2801.cmd. F2808.cmd, F2806.cmd and F2801.cmd are located in the *<base>DSP280x_common\cmd\ directory*.

2. **Add the *DSP280x_common\source\DSP280x_CSMPasswords.asm* to the project.**

   This file contains the passwords that will be programmed into the Code Security Module (CSM) password locations. Leaving the passwords set to 0xFFFF during development is recommended as the device can easily be unlocked. For more information on the CSM refer to the *TMS320x280x System Control and Interrupts Reference Guide* (spru712).

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 DSP280x examples, 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.

```
/*********************************************************************
* DSP280x_common\source\DSP280x_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:

```
/*********************************************************************
* DSP280x_common\include\F2808.cmd
*********************************************************************/
SECTIONS
{
   ramfuncs    : LOAD = FLASHD,
                 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

These symbols can then be used to copy the functions from the Flash to SARAM using the included example MemCopy routine or the C library standard memcopy() function.

To perform this copy from flash to SARAM using the included example MemCopy function:

a. Add the file *DSP280x_common\source\DSP280x_MemCopy.c* to the project.

b. Add the following function prototype to the example source code. This is done for you in the *DSP280x_Examples.h* file.

```
/*************************************************************
* DSP280x_common\include\DSP280x_Examples.h
*************************************************************/

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

c. Add the following variable declaration to your source code to tell the compiler that these variables exist. The linker command file will assign the address of each of these variables as specified in the linker command file as shown in step 3. For the DSP280x example code this has is already done in *DSP280x_Examples.h*.

```
/*************************************************************
* DSP280x_common\include\DSP280x_GlobalPrototypes.h
*************************************************************/

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

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

```
/*************************************************************
* DSP280x_examples\Flash source file
*************************************************************/

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

4. **Modify the code to call the flash initialization routine:**

   This function will initialize the wait states for the flash and enable the Flash Pipeline mode.

```
/*************************************************************
* DSP280x peripheral example .c file
*************************************************************/

InitFlash();
```

5. **Set the required jumpers for "boot to Flash" mode.**

   The required jumper settings for each boot mode are shown in

Table 7. 280x Boot Mode Settings

GPIO18	GPIO29	GPIO34	Mode
1	1	1	Boot to flash 0x3F7FF6
1	1	0	Call SCI-A boot loader
1	0	1	Call SPI-A boot loader
1	0	0	Call I2C boot loader
0	1	1	Call eCAN-A boot loader
0	1	0	Boot to M0 SARAM 0x000000
0	0	1	Boot to OTP 0x3D7800
0	0	0	Call parallel boot loader

   Refer to the documentation for your hardware platform for information on configuring the boot mode selection pins.

   For more information on the '280x boot modes refer to the *TMS320x280x Boot ROM Reference Guide* (SPRU722).

6. **Program the device with the built code.**

   This can be done using SDFlash available from Spectrum Digital's website (www.spectrumdigital.com). In addition the C2000 on-chip Flash programmer plug-in for Code Composer Studio will be available for the F280x family in 2Q05.

7. **To debug, load the project in CCS, select *File->Load Symbols->Load Symbols Only*.**

   It is useful to load only symbol information when working in a debugging environment where the debugger cannot or need not load the object code, such as when the code is in ROM or flash. This operation loads the symbol information from the specified file.

4 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 DSP281x header files then also refer to Section 6 for migration tips.

4.1 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 3.

2. Create a copy of the source files you want to use.

- *DSP280x_headers:* code required to incorporate the header files into your project
- *DSP280x_common*: shared source code much of which is used in the example projects.
- *DSP280x_examples*: example projects that use the header files and shared code.

4.2 Including the DSP280x Peripheral Header Files

Including the DSP280x header files in your project will allow you to use the bit-field structure approach in your code to access the peripherals on the DSP. To incorporate the header files in a new or existing project, perform the following steps:

3. #include "DSP280x_Device.h" in your source files.

This 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 "DSP280x_Device.h"
```

4. Edit DSP280x_Device.h and select the target you are building for:

In the below example, the file is configured to build for the '2808 device.

```
/*********************************************************************
* DSP280x_headers\include\DSP280x_Device.h
*********************************************************************/

#define    TARGET                1
#define    DSP28_2808            TARGET
#define    DSP28_2806            0
#define    DSP28_2801            0
```

By default, the '2808 device is selected. This is a superset of the other devices.

5. Add the source file *DSP280x_GlobalVariableDefs.c* to the project.

This file is found in the *DSP280x_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.

6. Add the appropriate DSP280x header linker command file to the project.

As described in Section 2.2, when using the DSP280x 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 *DSP280x_headers\cmd* must be included in your project:

– For non-DSP/BIOS† projects: *DSP280x_Headers_nonBIOS.cmd*

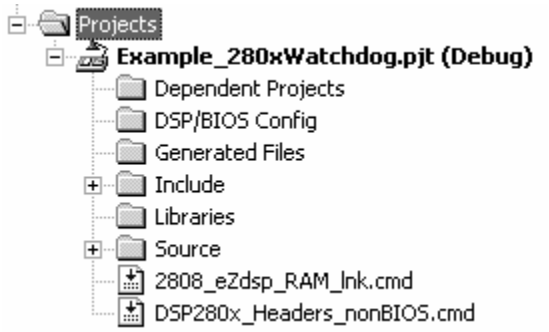
– For DSP/BIOS projects: *DSP280x_Headers_BIOS.cmd*

The method for adding the header linker file to the project depends on the version of Code Composer Studio being used.

Code Composer Studio V2.2 and later:

As of CCS 2.2, more then one linker command file can be included in a project.

Add the appropriate header linker command file (BIOS or nonBIOS) directly to the project.



Code Composer Studio prior to V2.2

Prior to CCS 2.2, each project contained only one main linker command file. This file can, however, call additional .cmd files as needed. To include the required memory allocations for the DSP280x header files, perform the following two steps:

1) Update the project's main linker command (.cmd) file to call one of the supplied DSP280x peripheral structure linker command files using the -l option.

```
/*********************************************************************
* User's linker .cmd file
*********************************************************************/

/* Use this include file only for non-BIOS applications */
-l DSP280x_Headers_nonBIOS.cmd
/* Use this include file only for BIOS applications */
/* -l DSP280x_Headers_BIOS.cmd */
```

† DSP/BIOS is a trademark of Texas Instruments

2) Add the directory path to the DSP280x peripheral linker .cmd file to your project.

a. Open the menu: *Project->Build Options*

b. Select the *Linker tab* and then Select *Basic.*

c. In the *Library Search Path*, add the directory path to the location of the *DSP280x_headers\cmd* directory on your system.

7. Add the directory path to the DSP280x header files to your project.

To specify the directory where the header files are located:

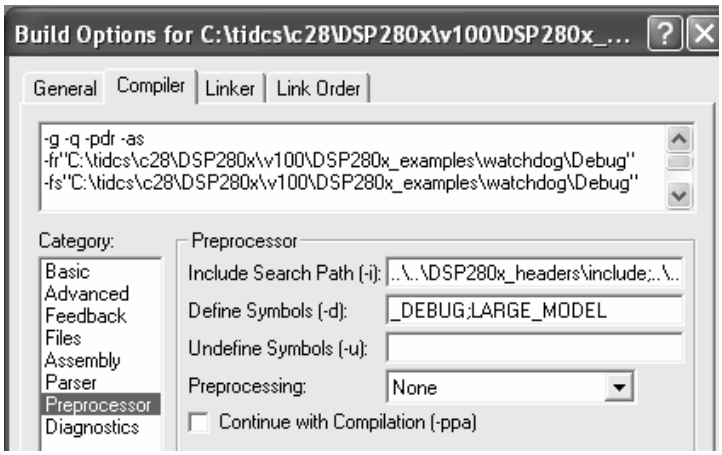
a. Open the menu:

Project->Build Options

b. Select the *Compiler tab*

c. Select *pre-processor*.

d. In the *Include Search Path*, add the directory path to the location of *DSP280x_headers\include* on your system.



8. Additional suggested build options:

The following are additional compiler and linker options. The options can all be set via the *Project->Build Options* menu.

– ***Compiler Tab:***

❑ **-ml Select *Advanced* 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 *Diagnostics* 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.

– ***Linker Tab:***

❑ **-w Select *Advanced* 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 *Basic* and enter Code Entry Point –e**

Defines a global symbol that specifies the primary entry point for the output module. For the DSP280x examples, this is the symbol “code_start”. This symbol is defined in the DSP280x_common\source\DSP280x_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.

4.3 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 "DSP280x_Examples.h" in your source files.

This include file will include common definitions and declarations used by the example code.

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

#include "DSP280x_Examples.h"
```

2. Add the directory path to the example include files to your project.

To specify the directory where the header files are located:

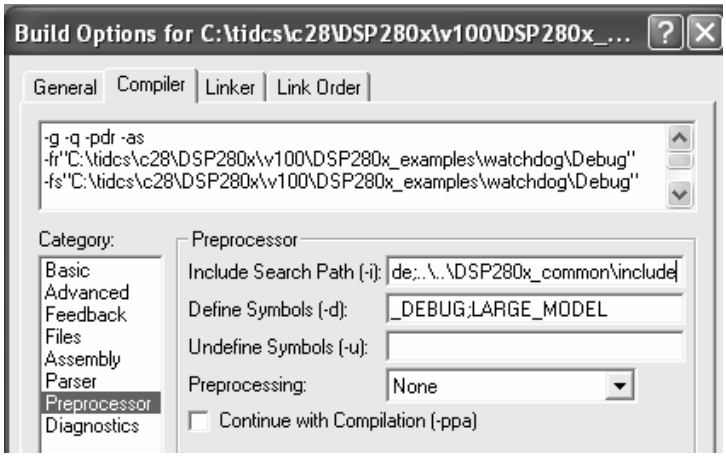
a. Open the menu:

Project->Build Options

b. Select the *Compiler tab*

c. Select *pre-processor*.

d. In the *Include Search Path*, add the directory path to the location of DSP280x_common/include on your system.
Use a semicolon between directories.



For example the directory path for the included projects is:
..\..\DSP280x_headers\include;..\..\DSP280x_common\include

3. Add a linker command file to your project.

The following memory linker .cmd files are provided as examples in the *DSP280x_common\cmd* directory. For getting started the basic *2808_eZdsp_RAM_lnk.cmd* file is suggested and used by most of the examples.

Table 8. Included Main Linker Command Files

Main Liner Command File Examples	Description
2808_eZdsp_RAM_lnk.cmd	Main eZdsp F2808 USB example linker file. Only uses only SARAM locations that are not protected by the code security module. This memory map is used for all of the examples to run out of the box on an eZdsp F2808 USB platform. No Flash or OTP locations are used.
F2808.cmd	Main F2808 linker command file. Includes all Flash and OTP memory locations.
F2806.cmd	Main F2806 linker command file. Includes all Flash, OTP memory locations.
F2801.cmd	Main F2801 linker command file. Includes all Flash, OTP memory locations.

4. Set the CPU Frequency

In the *DSP280x_common\include\DSP280x_Examples.h* file specify the proper CPU frequency. Some examples are included in the file.

```
/*-----------------------------------------------------------------
* DSP280x_common\include\DSP280x_Examples.h
-----------------------------------------------------------------*/

#define CPU_RATE   10.000L   // for a 100MHz CPU clock speed (SYSCLKOUT)
//#define CPU_RATE   13.330L   // for a 75MHz CPU clock speed (SYSCLKOUT)
//#define CPU_RATE   20.000L   // for a 50MHz CPU clock speed  (SYSCLKOUT)
```

5. Add desired common source files to the project.

The common source files are found in the *DSP280x_common\source* directory.

6. Include .c files for the PIE.

Since all catalog ‘280x 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 7.2.1.

5 Troubleshooting Tips & Frequently Asked Questions

- **In the examples, what do “EALLOW;” and “EDIS;” do?**

  EALLOW; is a macro defined in DSP280x_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 *TMS320x280x Control and Interrupts Reference Guide* (SPRU712).

- **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 DSP280x_common\source directory shows an example of enabling the peripheral clocks.
  - Some peripherals are not present on all 280x 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. *TMS320x280x Control and Interrupts Reference Guide* (SPRU712) for a complete list of EALLOW protected registers.

- **Memory block L0, L1 read back all 0x0000.**

  In this case most likely the code security module is locked and thus the protected memory locations are reading back all 0x0000. Refer to the for information on the code security module.

- **Code cannot write to L0 or L1 memory blocks.**

  In this case most likely the code security module is locked and thus the protected memory locations are reading back all 0x0000. Code that is executing from outside of the protected cannot read or write to protected memory while the CSM is locked. Refer to the *TMS320x280x Control and Interrupts Reference Guide* (SPRU712) for information on the code security module

- **A peripheral register reads back ok, but cannot be written to.**

  The EALLOW bit protects some registers from spurious writes by the CPU. If your program seems unable to write to a register, then check to see if it is EALLOW protected. If it is, then enable access using the EALLOW assembly instruction. *TMS320x280x Control and Interrupts Reference Guide* (SPRU712) 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 un-modified 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 DSP280x_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 DSP280x 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->Configurations*) 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:

  – Compiler Tab, Preprocessor: Include search path

  – Linker Tab, Basic: Library search path

  – Linker Tab, Basic: Include libraries (ie rts2800_ml.lib)

  Refer to section 3.5 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 codesize 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 DSP280x_examples\ ecan_back2back example project.

5.1 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:

5.1.1 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!
```

5.1.2 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.

6 Migration Tips for moving from the 281x header files (DSP281x V1.00) to the 280x header files (DSP280x V1.10)

This section includes suggestions for moving a project from the 281x header files to the 280x header files.

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

2. **Open the project (.pjt) file in a text editor**

   Replace DSP281x with DSP280x so that the appropriate source files are used. Check the path names to make sure they point to the appropriate header file and source code directories.

3. **Load the project into Code Composer Studio**

   Use the edit-> find in files dialog to find instances of DSP281x_Device.h and DSP281x_Example.h. Replace these with DSP280x_Device.h and DSP280x_Example.h respectively.

4. **Make sure you are using the correct linker command files (.cmd) appropriate for your device and for the DSP280x header files.**

   You will have one file for the memory definitions and one file for the header file structure definitions. Using a 281x memory file can cause issues since the H0 memory block has moved to a new location on the 280x devices.

5. **Build the project.**

   The compiler will highlight areas that have changed. Most of these changes will fall into one of the following categories:

   - Bit-name or register name corrections to align with the peripheral user guides. See Table 9 for a listing of these changes.
   - Code that was written for the 281x event manager (EV) will need to be re-written for the 280x ePWM, eCAP and eQEP peripherals.
   - Code for the 281x McBSP and XINTF will need to be removed as these peripherals are not available on the 280x devices.

Table 9. Summary of Register and Bit-Name Changes from DSP281x V1.00 to DSP280x V1.00 or V1.10

Peripheral	Register	Bit Name		Comment
		Old	New	
AdcRegs				
	ADCTRL2	EVB_SOC_ SEQ2	EPWM_SOCB_ SEQ2	SOC is now performed by ePWM
		EVA_SOC_ SEQ1	EPWM_SOCA_ SEQ1	SOC is now performed by ePWM
		EVB_SOC_ SEQ	EPWM_SOCB_ SEQ	SOC is now performed by ePWM
DevEmuRegs				
	DEVICEID		PARTID REVID	Split into two registers, PARTID and REVID
EcanaRegs				
	CANMDL	BYTE1	BYTE3	Order of bytes was incorrect
		BYTE3	BYTE1	
		BYTE4	BYTE0	
	CANMDH	BYTE5	BYTE7	Order of bytes was incorrect
		BYTE7	BYTE5	
		BYTE8	BYTE4	
GpioMuxRegs				
				The GPIO peripheral has been redesigned from the 281x. All of the registers have moved from 16-bit to 32-bits. The GpioMuxRegs are now the GpioCtrlRegs and the bit definitions have all changed. Please refer to *TMS320x280x Control and Interrupts Reference Guide* (SPRU712) for more information on the GPIO peripheral.
PieCtrlRegs				
	PIECTRL	PIECRTL	PIECTRL	Typo
SciaRegs, ScibRegs				
	SCIFFTX	TXFFILIL	TXFFIL	Typo
		TXINTCLR	TXFFINTCLR	Alignment with user's guide.
	SCIFFRX	RXFIFST	RXFFST	Typo – Also corrected in user's guide

7 Packet Contents:

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

7.1 Header File Support – DSP280x_headers

The DSP280x header files are located in the <*base*>*DSP280x_headers*\ directory.

7.1.1 DSP280x Header Files – Main Files

The following files must be added to any project that uses the DSP280x header files. Refer to section 4.2 for information on incorporating the header files into a new or existing project.

Table 10. DSP280x Header Files – Main Files

File	Location	Description
DSP280x_Device.h	DSP280x_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 4.2.
DSP280x_GlobalVariableDefs.c	DSP280x_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 4.2.
DSP280x_Headers_BIOS.cmd	DSP280x_headers\cmd	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 4.2.
DSP280x_Headers_nonBIOS.cmd	DSP280x_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 4.2.

7.1.2 DSP280x Header Files – Peripheral Bit-Field and Register Structure Definition Files

The following files define the bit-fields and register structures for each of the peripherals on the 280x devices. These files are automatically included in the project by including *DSP280x_Device.h.* Refer to section 4.2 for more information on incorporating the header files into a new or existing project.

Table 11. DSP280x Header File Bit-Field & Register Structure Definition Files

File	Location	Description
DSP280x_Adc.h	DSP280x_headers\include	ADC register structure and bit-field definitions.
DSP280x_CpuTimers.h	DSP280x_headers\include	CPU-Timer register structure and bit-field definitions.
DSP280x_DevEmu.h	DSP280x_headers\include	Emulation register definitions
DSP280x_ECan.h	DSP280x_headers\include	eCAN register structures and bit-field definitions.
DSP280x_ECap.h	DSP280x_headers\include	eCAP register structures and bit-field definitions.
DSP280x_EPwm.h	DSP280x_headers\include	ePWM register structures and bit-field definitions.
DSP280x_EQep.h	DSP280x_headers\include	eQEP register structures and bit-field definitions.
DSP280x_Gpio.h	DSP280x_headers\include	General Purpose I/O (GPIO) register structures and bit-field definitions.
DSP280x_I2c.h	DSP280x_headers\include	I2C register structure and bit-field definitions.
DSP280x_PieCtrl.h	DSP280x_headers\include	PIE control register structure and bit-field definitions.
DSP280x_PieVect.h	DSP280x_headers\include	Structure definition for the entire PIE vector table.
DSP280x_Sci.h	DSP280x_headers\include	SCI register structure and bit-field definitions.
DSP280x_Spi.h	DSP280x_headers\include	SPI register structure and bit-field definitions.
DSP280x_SysCtrl.h	DSP280x_headers\include	System register definitions. Includes Watchdog, PLL, CSM, Flash/OTP, Clock registers.
DSP280x_XIntrupt.h	DSP280x_headers\include	External interrupt register structure and bit-field definitions.

7.1.3 Code Composer .gel Files

The following Code Composer Studio .gel files are included for use with the DSP280x Header File peripheral register structures.

Table 12. DSP280x Included GEL Files

File	Location	Description
DSP280x_Peripheral.gel	DSP280x_headers\gel	Provides GEL pull-down menus to load the DSP280x data structures into the watch window. You may want to have CCS load this file automatically by adding a GEL_LoadGel("<base>DSP280x_headers\/gel\DSP280x_peripheral.gel") function to the standard F2808.gel that was included with CCS.

7.1.4 Variable Names and Data Sections

This section is a summary of the variable names and data sections allocated by the DSP280x_headers\source\DSP280x_GlobalVariableDefs.c file. Note that all peripherals may not be available on a particular 280x device. Refer to the device datasheet for the peripheral mix available on each 280x family derivative.

Table 13. DSP280x Variable Names and Data Sections

Peripheral	Starting Address	Structure Variable Name
ADC	0x007100	AdcRegs
ADC Mirrored Result Registers	0x000B00	AdcMirror
Code Security Module	0x000AE0	CsmRegs
Code Security Module Password Locations	0x3F7FF6- 0x3F7FFF	CsmPwl
CPU Timer 0	0x000C00	CpuTimer0Regs
Device and Emulation Registers	0x000880	DevEmuRegs
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
eCAN-B	0x006200	ECanbRegs
eCAN-B Mail Boxes	0x006300	ECanbMboxes
eCAN-B Local Acceptance Masks	0x006240	ECanbLAMRegs
eCAN-B Message Object Time Stamps	0x006280	ECanbMOTSRegs
eCAN-B Message Object Time-Out	0x0062C0	ECanbMOTORegs
ePWM1	0x006800	EPwm1Regs
ePWM2	0x006840	EPwm2Regs
ePWM3	0x006880	EPwm3Regs
ePWM4	0x0068C0	EPwm4Regs
ePWM5	0x006900	EPwm5Regs
ePWM6	0x006940	EPwm6Regs
eCAP1	0x006A00	ECap1Regs
eCAP2	0x006A20	ECap2Regs
eCAP3	0x006A40	ECap3Regs
eCAP4	0x006A60	ECap4Regs
eQEP1	0x006B00	EQep1Regs
eQEP2	0x006B40	EQep2Regs
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

Peripheral	Starting Address	Structure Variable Name
PIE Control	0x000CE0	PieCtrlRegs
SCI-A	0x007050	SciaRegs
SCI-B	0x007750	ScibRegs
SPI-A	0x007040	SpiaRegs
SPI-B	0x007740	SpibRegs
SPI-C	0x007760	SpicRegs
SPI-D	0x007780	SpidRegs

7.2 Common Example Code – DSP280x_common

7.2.1 Peripheral Interrupt Expansion (PIE) Block Support

In addition to the register definitions defined in DSP280x_PieCtrl.h, this packet provides the basic ISR structure for the PIE block. These files are:

Table 14. Basic PIE Block Specific Support Files

File	Location	Description
DSP280x_DefaultIsr.c	DSP280x_common\source	Shell interrupt service routines (ISRs) for the entire PIE vector table. You can choose to populate one of functions or re-map your own ISR to the PIE vector table. **Note: This file is not used for DSP/BIOS projects.**
DSP280x_DefaultIsr.h	DSP280x_common\include	Function prototype statements for the ISRs in DSP280x_DefaultIsr.c. **Note: This file is not used for DSP/BIOS projects.**
DSP280x_PieVect.c	DSP280x_common\source	Creates an instance of the PIE vector table structure initialized with pointers to the ISR functions in DSP280x_DefaultIsr.c. This instance can be copied to the PIE vector table in order to initialize it with the default ISR locations.

In addition, the following files are included for software prioritization of interrupts. These files are used in place of those above when additional software prioritization of the interrupts is required. Refer to the example and documentation in *DSP280x_examples\sw_prioritized_interrupts* for more information.

Table 15. Software Prioritized Interrupt PIE Block Specific Support Files

File	Location	Description
DSP280x_SWPrioritizedDefaultIsr.c	DSP280x_common\source	Default shell interrupt service routines (ISRs). These are shell ISRs for all of the PIE interrupts. You can choose to populate one of functions or re-map your own interrupt service routine to the PIE vector table. **Note: This file is not used for DSP/BIOS projects.**
DSP280x_SWPrioritizedIsrLevels.h	DSP280x_common\include	Function prototype statements for the ISRs in DSP280x_DefaultIsr.c. **Note: This file is not used for DSP/BIOS projects.**
DSP280x_SWPrioritizedPieVect.c	DSP280x_common\source	Creates an instance of the PIE vector table structure initialized with pointers to the default ISR functions that are included in DSP280x_DefaultIsr.c. This instance can be copied to the PIE vector table in order to initialize it with the default ISR locations.

7.2.2 Peripheral Specific Files

Several peripheral specific initialization routines and support functions are included in the peripheral .c source files in the *DSP280x_common\src* directory. These files include:

Table 16. Included Peripheral Specific Files

File	Description
DSP280x_GlobalPrototypes.h	Function prototypes for the peripheral specific functions included in these files.
DSP280x_Adc.c	ADC specific functions and macros.
DSP280x_CpuTimers.c	CPU-Timer specific functions and macros.
DSP280x_ECan.c	Enhanced CAN specific functions and macros.
DSP280x_ECap.c	eCAP module specific functions and macros.
DSP280x_EPwm.c	ePWM module specific functions and macros.
DSP280x_EPwm_defines.h	#define macros that are used for the ePWM examples
DSP280x_EQep.c	eQEP module specific functions and macros.
DSP280x_Gpio.c	General-purpose IO (GPIO) specific functions and macros.
DSP280x_I2C.c	I2C specific functions and macros.
DSP280x_I2c_defines.h	#define macros that are used for the I2C examples
DSP280x_PieCtrl.c	PIE control specific functions and macros.
DSP280x_Sci.c	SCI specific functions and macros.
DSP280x_Spi.c	SPI specific functions and macros.
DSP280x_SysCtrl.c	System control (watchdog, clock, PLL etc) specific functions and macros.

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.

7.2.3 Utility Function Source Files

Table 17. Included Utility Function Source Files

File	Description
DSP280x_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.
DSP280x_DBGIER.asm	Assembly function to manipulate the DEBIER register from C.
DSP280x_DisInt.asm	Disable interrupt and restore interrupt functions. These functions allow you to disable INTM and DBGM and then later restore their state.
DSP280x_usDelay.asm	Assembly function to insert a delay time in microseconds. This function is cycle dependant and must be executed from zero wait-stated RAM to be accurate. Refer to *DSP280x_examples\adc* for an example of its use.
DSP280x_CSMPasswords.asm	Include in a project to program the code security module passwords and reserved locations.

7.2.4 Example Linker .cmd files

Example memory linker command files are located in the *DSP280x_common\cmd* directory. For getting started using the 280x devices, the basic 2808_eZdsp_RAM_lnk.cmd file is suggested and used by many of the included examples.

On 280x devices, the SARAM blocks L1, L2 and H0 are mirrored. For simplicity these memory maps only include one instance of these memory blocks.

Table 18. Included Main Linker Command Files

Main Liner Command File Examples	Description
2808_eZdsp_RAM_lnk.cmd	eZdsp F2808 USB memory linker example. Only allocates SARAM locations. This memory map is used for all of the examples that run out of the box on an eZdsp F2808 USB. No Flash, OTP, or CSM password protected locations are used.
F2808.cmd	F2808 memory linker command file. Includes all Flash, OTP and CSM password protected memory locations.
F2806.cmd	F2806 memory linker command file. . Includes all Flash, OTP and CSM password protected memory locations.
F2801.cmd	F2801 memory linker command file. . Includes all Flash, OTP and CSM password protected memory locations.

7.2.5 Example Library .lib Files

Example library files are located in the *DSP280x_common\lib* directory. For this release only the IQMath library is included for use in the example projects. Please refer to the *C28x IQMath Library - A Virtual Floating Point Engine (*SPRC087) for more information on IQMath and the most recent IQMath library.

Table 19. Included Library Files

Main Liner Command File Examples	Description
IQmath.lib	Please refer to the *C28x IQMath Library - A Virtual Floating Point Engine (*SPRC087) for more information on IQMath.
IQmathLib.h	IQMath header file.