# F2802x Firmware Development Package

# USER'S GUIDE


F2802x-FRM-EX-UG-200

# Copyright



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

# Revision Information

This is version 200 of this document, last updated on Tue Jul 24 10:01:48 CDT 2012.

# Table of Contents

# 1 Introduction

The Texas Instruments® F2802x0 Firmware Development Package is a collection of device header files, common source files, helper libraries and example applications for the 2802X0 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** 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(2802x0) can be found in the *F2802x0_examples_ccsv4* directory. As users move past evaluation, and get started developing their own application, TI recommends they maintain a similar project directory structure to that used in the example projects.

The Appendix covers the following topics

1. **Appendix A** - describes the default hardware prioritizing of Interrupt Software Routines and how it can be over-ridden in software.
2. **Appendix B** - Each factory programmed device from TI has compensation routines in OTP memory for oscillator drift due to temperature fluctuations. These routines are described here.

# 2 Driver Library and Header File Quickstart



## 2.1 Device Support

This software package supports 2802x0 devices. This includes the following: TMS320F280220, TMS320F280230, TMS320F280260, and TMS320F280270. Throughout this document, TMS320F280220, TMS320F280230, TMS320F280260, and TMS320F280270 are abbreviated as F280220, F280230, F280260, and F280270 respectively.

## 2.2 Introduction

The 2802x0 C/C++ peripheral header files, driver library, and example projects facilitate writing in C/C++ Code for the Texas Instruments TMS320x2802x0 devices. The code can be used as a learning tool or as the basis for a development platform depending on the current needs of the user.

1. **Learning Tool**
   This download includes several example Code Composer Studio$^{TM}$ v 4.0+ [1] projects for a 2802x0 development platform.
   These examples demonstrate the steps required to initialize the device and utilize the on-chip peripherals. The provided examples can be copied and modified giving the user a platform to quickly experiment with different peripheral configurations.
   These projects can also be migrated to other devices by simply changing the memory allocation in the linker command file.
2. **Development Platform**
   The peripheral header files can easily be incorporated into a new or existing project to provide a platform for accessing the on-chip peripherals using C or C++ code. In addition, the user can pick and choose functions from the provided code samples as needed and discard the rest.

To get started this document provides the following information:

1. Overview of the bit-field structure approach used in the 2802x0 C/C++ peripheral header files.
2. Overview of the included peripheral example projects.

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

3. Steps for integrating the peripheral header files into a new or existing project.
4. Troubleshooting tips and frequently asked questions.
5. Migration tips for users moving from the 2802x header files to the 2802x0 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 2802x0 hardware platform setup and connected to a host with Code Composer Studio installed. The user should have a basic understanding of how to use Code Composer Studio to download code through JTAG and perform basic debug operations.

### 2.2.1 Revision History(Summary)

1. **Version 1.00**
   - This version is the first release of the 2802x0 header files and examples.

### 2.2.2 Directory Structure

As installed, the 2802x0 C/C++ Header Files and Peripheral Examples is partitioned into a well-defined directory structure(see figure 2.1).

Table 2.1 describes the contents of the main directories used by 2802x0 header files and peripheral examples:

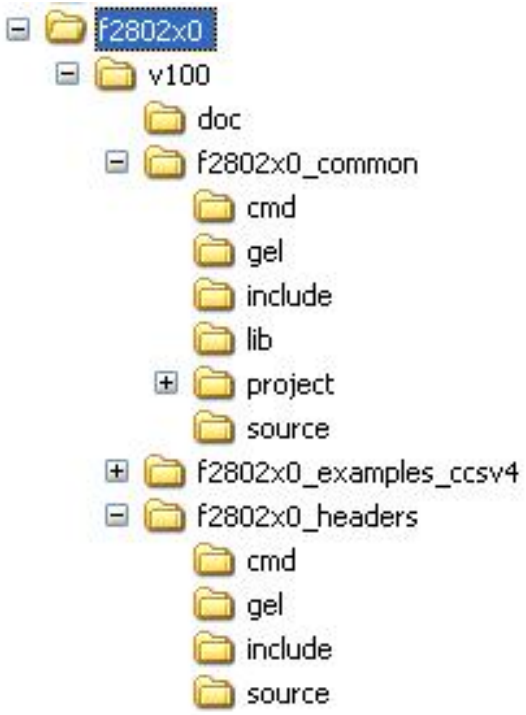


Figure 2.1: F2802x0 Main Directory Structure

| Directory | Description |
|---|---|
| <base> | Base install directory |
| <base> | Documentation including the revision history from the previous release. |
| <base>2802x0_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>2802x0_examples_ccsv4 | Example Code Composer Studio v4 projects. These example projects illustrate how to configure many of the on-chip peripherals. An overview of the examples is given in Section 2.4. |
| <base>2802x0_common | Drivers and 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 2.9. |

Table 2.1: DSP2802x0 Main Directory Structure

Under the F2802x0_headers and F2802x0_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:

| Sub-Directory | Description |
|---|---|
| F2802x0_headers\cmd | Linker command files that allocate the bit-field structures described in Section 2.3. |
| F2802x0_headers\source | Source files required to incorporate the header files into a new or existing project. |
| F2802x0_headers\include | Header files for each of the on-chip peripherals. |
| F2802x0_common\cmd | Example memory command files that allocate memory on the devices. |
| F2802x0_common\include | Driver and common .h files that are used by the peripheral examples. |
| F2802x0_common\source | Driver and common .c files that are used by the peripheral examples. |
| F2802x0_common\lib | Driver and common library (.lib) files that are used by the peripheral examples. |
| F2802x0_common\project | Project files for the driver library. |
| F2802x0_common\gel\ccsv4 | Code Composer Studio v4.x GEL files for each device. These are optional. |

Table 2.2: F2802x0 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 2802x0 Driver Library, Header Files, and Peripheral Examples software package.

## 2.4.1 Getting Started in Code Composer Studio v4.0+

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

1. **Have a hardware platform connected to a host with Code Composer Studio installed**

   **NOTE: As supplied, the 2802x0 example projects are built for the 280270 device. If you are using another 2802x0 device, the memory definition in the linker command file (.cmd) will need to be changed and the project rebuilt.**

2. **Open the example project** Each example has its own project directory which is “imported”/opened in Code Composer Studio v4. To open the 2802x0 CPU-Timer example project directory, follow the following steps:

   - In Code Composer Studio v 4.x: Project->Import Existing CCS/CCE Eclipse Project.
   - Next to “Select Root Directory”, browse to the CPU Timer example directory: F2802x0_examples_ccsv4\cpu_timer. Select the Finish button. This will import/open the project in the CCStudio v4 C/C++ Perspective project window.

3. **Edit F2802x0_Device.h** Edit the F2802x0_Device.h file and make sure the appropriate device is selected. By default the 28069 is selected.

```
/*-----------------------------------------------------------------------
 F2802x0_Device.h
-----------------------------------------------------------------------*/

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

#define   DSP28_280220PT   0
#define   DSP28_280220DA   0

#define   DSP28_280230PT   0
#define   DSP28_280230DA   0

#define   DSP28_280260PT   0
#define   DSP28_280260DA   0

#define   DSP28_280270PT   TARGET
#define   DSP28_280270DA   0
```

4. **Edit F2802x0_Examples.h** Edit F2802x0_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 50MHz SYSCLKOUT frequency.

```
/*-----------------------------------------------------------------
 F2802x0_common\include\F2802x0_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   10      // Uncomment for 50 Mhz devices [50 Mhz = (10MHz * 10)/2]
//#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 F2802x0_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 50 MHz SYSCLKOUT.

```
/*-----------------------------------------------------------------
 F2802x0_common\include\F2802x0_Examples.h
-----------------------------------------------------------------*/
...
#define CPU_RATE   12.500L   // for a 80MHz CPU clock speed (SYSCLKOUT)
//#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. **Review the comments at the top of the main source file: Example_2802x0CpuTimer.c** A brief description of the example and any assumptions that are made and any external hardware requirements are listed in the comments at the top of the main source file of each example. In some cases you may be required to make external connections for the example to work properly.

6. **Perform any hardware setup required by the example** Perform any hardware setup indicated by the comments in the main source. The CPU-Timer example only requires that the hardware be setup for “Boot to SARAM” mode. Other examples may require additional hardware configuration such as connecting pins together or pulling a pin high or low. Table 2.3 shows a listing of the boot mode pin settings for your

reference. Table 2.4 and Table 2.5 list the EMU boot modes (when emulator is connected) and the Get Mode boot mode options (mode is programmed into OTP) respectively. Refer to the documentation for your hardware platform for information on configuring the boot mode pins. For more information on the 2802x0 boot modes refer to the device specific *Boot ROM* chapter in the *Technical Reference Manual*.

| 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: 2802x0 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 |
| | 0x000A | Boot to RAM |
| | 0x000B | Boot to FLASH |
| | Other | Wait |

Table 2.4: 2802x0 EMU Boot Modes (Emulator Connected)

| OTP_KEY 0x3D7BFB | OTP_BMODE 0x3D7BFE | Boot Mode Selected |
|---|---|---|
| != 0x55AA | x | Get Mode - Flash |
| 0x55AA | 0x0001 | Get Mode - SCI |
| | 0x000B | Get Mode - Flash |
| | 0x0004 | Get Mode - SPI |
| | 0x0005 | Get Mode - I2C |
| | 0x0006 | Get Mode - OTP |
| | Other | Get Mode - Flash |

Table 2.5: 2802x0 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 2802x0 gel files in the F2802x0_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 OTP_KEY at address 0x3D7BFB and OTP_BMODE at address 0x3D7BFE can be programmed for the desired boot mode per Table 2.5.

7. **Build and Load the code**

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

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

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

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

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

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

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

    Go to *View->C/C++ Projects* to open up your project view. To remove/delete the project from the workspace, right click on the project's name and select delete. Make sure the *Do not delete* contents button is selected, then select *Yes*. This does not delete the project itself. It merely removes the project from the workspace until you wish to open/import it again.

    The examples use the header files in the F2802x0_headers directory and shared source in the F2802x0_common directory. Only example files specific to a particular example are located within the example directory.

    **NOTE: Most of the example code included uses the driver library, but shows the equivalent bit field approach in the comments. This is done to help the user learn how to use the peripheral and device. In addition, the example projects have the compiler optimizer turned off to aid in debugging. 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 driver library, and linker command files.

```
//###########################################################################
// F2802x0_examples\cpu_timer\Example_2802x0CpuTimer.c
//###########################################################################

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

- **DSP28x_Project.h**

  This header file includes F2802x0_Device.h and F2802x0_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>\F2802x0_common\include directory.

- **F2802x0_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 can also be used to include all of the driver library header files by simply defining INCLUDE_ALL before this files is included (or in the case that DSP28x_Project.h is used, before that file is included). This file is found in the <base>\F2802x0_headers\include directory.

- **F2802x0_Examples.h**

  This header file defines parameters that are used by the example code. This file is not required to use just the F2802x0 peripheral header files but is required by some of the common source files. This file is found in the <base>\F2802x0_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.

- **F2802x0_GlobalVariableDefs.c**

  Any project that uses the F2802x0 peripheral header files must include this source file IF THEY ARE NOT USING THE DRIVER LIBRARY. In this file are the declarations for the peripheral register structure variables and data section assignments. This file is found in the <base>\F2802x0_headers\source directory.

- **Example specific source code**

  Files that are specific to a particular example have the prefix Example_2802x0 in their filename. For example Example_2802x0CpuTimer.c is specific to the CPU Timer example and not used for any other example. Example specific files are located in the <base>\F2802x0_examples\<example> directory.

- **driverlib.lib**

  This file contains the objects for all of the driver library functions. This file must be linked into the project if the project uses the driver library.

#### 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 F2802x0 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>\F2802x0_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 |
|---|---|---|
| 280220_RAM_lnk.cmd | F2802x0_common | 28062 SARAM memory linker command file. |
| 280230_RAM_lnk.cmd | F2802x0_common | 28063 SARAM memory linker command file. |
| 280260_RAM_lnk.cmd | F2802x0_common | 28066 SARAM memory linker command file. |
| 280270_RAM_lnk.cmd | F2802x0_common | 28067 SARAM memory linker command file. |
| F2802x0_generic_ram.cmd | F2802x0_common | Generic SARAM memory linker command file. |
| F280220.cmd | F2802x0_common | F28062 memory linker command file. Includes all Flash, OTP and CSM password protected memory locations. |
| F280230.cmd | F2802x0_common | F28063 memory linker command file. |
| F280260.cmd | F2802x0_common | F28066 memory linker command file. |
| F280270.cmd | F2802x0_common | F28067 memory linker command file. |
| F2802x0_generic_flash.cmd | F2802x0_common | Generic flash 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 |
|---|---|---|
| F2802x0_Headers_BIOS.cmd | F2802x0_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. |
| F2802x0_Headers_nonBIOS.cmd | F2802x0_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: F2802x0 Peripheral Header Linker Command File

### 2.4.3 Example Program Flow

All of the example programs follow a similar recommended flow for setting up a 2802x0 device.

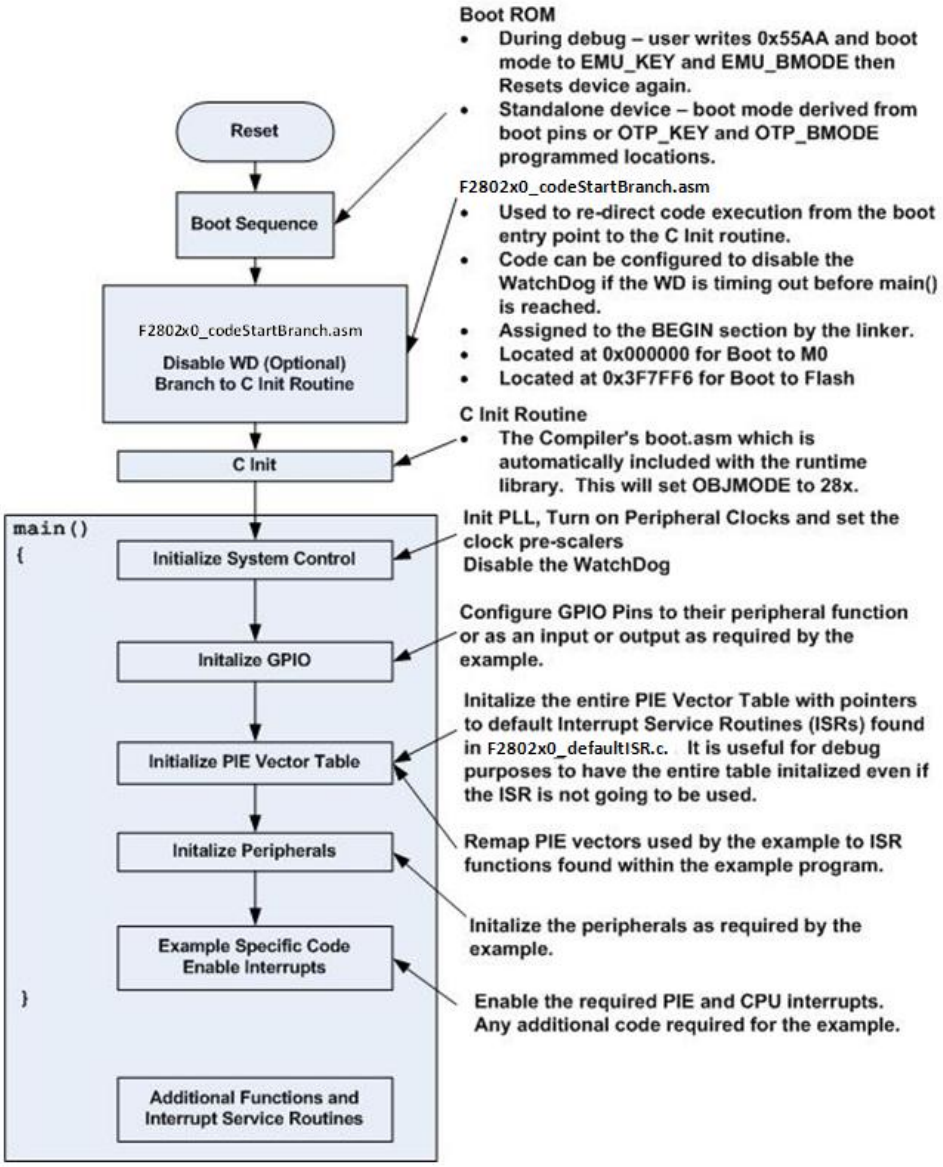


Figure 2.3: Flow for Example Programs

### 2.4.4 Included Examples

See Chapter 3 for a complete listing and description of available examples

### 2.4.5 Executing the Examples From Flash

Most of the F2802x0 examples execute from SARAM in “boot to SARAM” mode. One example, F2802x0_examples\flash_f28069, 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 280270_RAM_lnk.cmd from the project and link one of the flash based linker files (ex: F280270.cmd). These files are located in the <base>F2802x0_common\cmd directory.
2. **Link the F2802x0_common\source\F2802x0_CSMPasswords.asm to the project**
   This file contains the passwords that will be programmed into the Code Security Module (CSM) password locations. Leaving the passwords set to 0xFFFF during development is recommended as the device can easily be unlocked. For more information on the CSM refer to the appropriate *System Control and Interrupts* chapter of the *Technical Reference Manual*.
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 F2802x0, 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.

```
//*************************************************************************
 F2802x0_common\source\F2802x0_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:

```
/*************************************************************************
 F2802x0_common\include\F280270.cmd
*************************************************************************/
SECTIONS
{
   ramfuncs     : LOAD = FLASHA,
                  RUN = RAML0,
                  LOAD_START(_RamfuncsLoadStart),
                  LOAD_END(_RamfuncsLoadEnd),
                  RUN_START(_RamfuncsRunStart),
                  LOAD_SIZE(_RamfuncsLoadSize),
                  PAGE = 0
}
```

   The linker will create symbols for the block “ramfuncs”. These are described in the Table 2.8.
   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 included example memcpy function:

   (a) Include **string.h** at the top of the file.
   **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.**

| Address | Symbol |
|---|---|
| Load start address | RamfuncsLoadStart |
| Load end address | RamfuncsLoadEnd |
| Run start address | RamfuncsRunStart |
| Load Size | RamfuncsLoadSize |

Table 2.8: Linker Symbol assignment

(b) 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 F2802x0 example code this has is already done in F2802x0_GlobalPrototypes.h.

```
/*************************************************************
 F2802x0_common\include\F2802x0_GlobalPrototypes.h
*************************************************************/

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

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

```
/*************************************************************
 F2802x0_examples\Flash source file
*************************************************************/

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

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.

```
/*************************************************************
 F2802x0 peripheral example .c file
*************************************************************/

InitFlash();
```

5. **Set the required jumpers for "boot to Flash" mode** The required jumper settings for each boot mode are shown in Table 2.3, Table 2.4, and Table 2.5.

6. **Program the device with the built code**
   In Code Composer Studio v4, when code is loaded into the device during debug, it automatically programs to flash memory.
   This can also be done using SDFlash available from Spectrum Digital's website (Spectrum Digital).
   These tools will be updated to support new devices as they become available. Please check for updates.
7. **In Code Composer Studio v3, to debug, load the project in CCS, select File->Load Symbols->Load Symbols Only**

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

# 2.5 Steps for Incorporating the Driver Library and/or Header Files

Follow these steps to incorporate the driver library into your own projects. If you already have a project that uses the DSP2802x header files then also refer to Section 2.7 for migration tips.

## 2.5.1 Before you begin

Before you include the driver library or header files 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 driver library. This is described in Section 2.4.
2. **Create a copy of the example files you want to use**
   *F2802x0_examples*: 2802x0 fixed-point compiled example projects that use the driver files. Create a copy of any of the example directories in this folder and work on the copied folder to ensure the original is still there for reference purposes.

## 2.5.2 Including the F2802x Driver Library and Header Files

Including the F2802x driver library in your project will allow you to use the drivers for every peripheral on the device, substantially decreasing the learning curve of this device. This guide also shows how to include the header files in a project, but their use is discouraged in end application software.

To incorporate the drivers and or headers in a new or existing project perform the following steps:

1. **Include the necessary driver header files**
   The F2802x0_Device.h include file by default only includes the neccessary files to make use of the peripheral bitfield structures. Include statements for all the driver header files may be conditionally compiled in by defining the variable INCLUDE_ALL.

```
//###########################################################################
//* User's source file
//###########################################################################
#ifndef INCLUDE_ALL
#define INCLUDE_ALL
#endif

#include "F2802x0_Device.h"
```

Another option is to #include “DSP28x_Project.h” in your source files, which in-turn includes “F2802x0_Device.h” and “F2802x0_Examples.h”. Keep in mind that INCLUDE_ALL must still be defined in order for F2802x0_Device.h to bring in the driver headers. Due to the device-generic nature of the file name, user code is easily ported between different device header files.

```
//*************************************************************************
//* User's source file
//*************************************************************************
#ifndef INCLUDE_ALL
#define INCLUDE_ALL
#endif

#include "DSP28x_Project.h"
```

Finally if the user doesn't wish to include all of the driver headers in a particular source file they may include each needed file individually.

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

#include "f2802x0_common/include/adc.h"
#include "f2802x0_common/include/cpu.h"
#include "f2802x0_common/include/clk.h"
```

2. **Edit F2802x0_Device.h and select the target you are building for**
   In the below example, the file is configured to build for the 280270 device.

```
//*************************************************************************
//* F2802x0_headers\include\F2802x0_Device.h
//*************************************************************************
#define   TARGET   1
//-------------------------------------------------------------
// User To Select Target Device:

#define   DSP28_280220PT   0
#define   DSP28_280220DA   0

#define   DSP28_280230PT   0
#define   DSP28_280230DA   0

#define   DSP28_280260PT   0
#define   DSP28_280260DA   0

#define   DSP28_280270PT   TARGET
#define   DSP28_280270DA   0
```

   By default, the 280270 device is selected.

3. **If you are not using the driver library, add the source file F2802x0_GlobalVariableDefs.c to the project. If you are using the driver library, the driver library includes the objects from this source file.**
   This file is found in the F2802x0_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 F2802x0 header linker command file to the project.** As described in Section 2.4, when using the F2802x0 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 F2802x0_headers\cmd must be included in your project:

   - For non-DSP/BIOS[2] projects: *F2802x0_Headers_nonBIOS.cmd*
   - For DSP/BIOS projects: *F2802x0_Headers_BIOS.cmd*

   The method for adding the header linker file to the project depends on preference

   **Method #1:**

   - Right-click on the project in the project window of the C/C++ Projects perspective.
   - Select Link Files to Project...
   - Navigate to the F2802x0_headers\cmd directory on your system and select the desired .cmd file.

   **Note: The limitation with Method #1 is that the path to <install directory>\F2802x0_headers\cmd\<cmd file>.cmd is fixed on your PC. If you move the installation directory to another location on your PC, the project will "break" because it still expects the .cmd file to be in the original location. Use Method #2 if you are using "linked variables" in your project to ensure your project/installation directory is portable across computers and different locations on the same PC. For more information, see:** Portable_Projects_in_CCSv4_for_C2000

   **Method #2:**

   - Right-click on the project in the project window of the C/C++ Projects perspective.
   - Select New->File.
   - Click on the Advanced» button to expand the window.
   - Check the Link to file in the file system check-box.
   - Select the Variables... button. From the list, pick the linked variable (macro defined in your macros.ini file) associated with your installation directory. (e.g. INSTALLROOT_2802X0_V<version#>). For more information on linked variables and the macros.ini file, see: Portable_Projects_in_CCSv4_for_C2000
   - Click on the Extend... button. Navigate to the desired .cmd file and select OK.

5. **If you intend to use the driver library, add the library file to your project.** If you would like to use the driver library you will need to add it to the project in much the same way as you added a linker file in the previous step. The library file can be found in f2802x0_common/lib/ and is called driverlib.lib. **Note: Debug and Release configurations of the driver library are available in its associated CCS project (found in f2802x0_common/project), but the two build configurations output their library file to the same location. By default controlSUITE includes the debug configuration, but if the library is rebuilt all of the example projects will use the driver library build configuration that was last built.**

6. **Add the directory path to the F2802x0 device support files to your project**
   **Code Composer Studio 4.x:**

   To specify the directory where the header files are located:

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

- Open the menu: Project->Properties.
- In the menu on the left, select “C/C++ Build”.
- In the “Tool Settings” tab, Select “C2000 Compiler -> Include Options:”
- In the “Add dir to #include search path (–include_path, -I)” window, select the “Add” icon in the top right corner.
- Select the “File system...” button and navigate to the directory path of controlSUITE\device_support\f2802x0\version\ on your system.

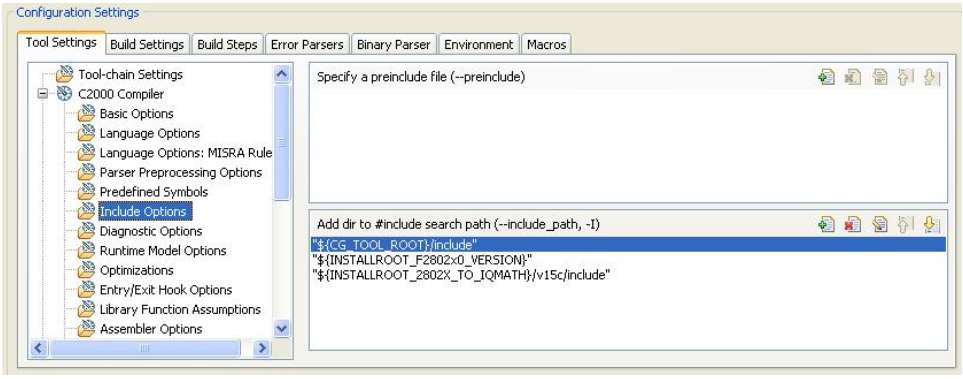


Figure 2.4: Adding device header file directories to the include search path

7. **Additional suggested build options** The following are additional compiler and linker options. The options can all be set via the Project-> Properties->Tool Settings sub-menus.
   - **C2000 Compiler**
     - **-ml Select Runtime Model Options and check -ml** Build for large memory model. This setting allows data sections to reside anywhere within the 4M-memory reach of the 28x devices.
     - **-pdr Select Diagnostic Options and check -pdr** Issue non-serious warnings. The compiler uses a warning to indicate code that is valid but questionable. In many cases, these warnings issued by enabling -pdr can alert you to code that may cause problems later on.
   - **C2000 Linker**
     - **-w Select Diagnostics and check -w** Warn about output sections. This option will alert you if any unassigned memory sections exist in your code. By default the linker will attempt to place any unassigned code or data section to an available memory location without alerting the user. This can cause problems, however, when the section is placed in an unexpected location.
     - **-e Select Symbol Management and enter Program Entry Point -e** Defines a global symbol that specifies the primary entry point for the output module. For the F2802x0 examples, this is the symbol “code_start”. This symbol is defined in the F2802x0_common\source\F2802x0_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 an entry point using the -e option, then the linker will use _c_int00 by default.

### 2.5.3 Including Common Example Code

**Note: This section describes files which are no longer being maintained. We suggest you use the true driver library instead of these older sample source files.**

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 "f2802x0_common\include\F2802x0_Examples.h" (or "DSP28x_Project.h") in your source files.**
   The "F2802x0_Examples.h" include file will include common definitions and declarations used by the example code.

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

#include "f2802x0_common\include\F2802x0_Examples.h"
```

   Another option is to #include "DSP28x_Project.h" in your source files, which in-turn includes "F2802x0_Device.h" and "F2802x0_Examples.h". Due to the device-agnostic nature of the file name, user code is easily ported between different device header files.

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

#include "DSP28x_Project.h"
```

2. **Link a linker command file to your project.**
   The following memory linker .cmd files are provided as examples in the F2802x0_common\cmd directory. For getting started the basic 280270_RAM_lnk.cmd file is suggested and used by most of the examples.

| Memory Linker Command File Examples | Location | Description |
|---|---|---|
| 280220_RAM_lnk.cmd | F2802x0_common/cmd | 280220 SARAM memory linker command file. |
| 280230_RAM_lnk.cmd | F2802x0_common/cmd | 280230 SARAM memory linker command file. |
| 280260_RAM_lnk.cmd | F2802x0_common/cmd | 280260 SARAM memory linker command file. |
| 280270_RAM_lnk.cmd | F2802x0_common/cmd | 280270 SARAM memory linker command file. |
| F280220.cmd | F2802x0_common/cmd | F280220 memory linker command file. Includes all Flash, OTP and CSM password protected memory locations. |
| F280230.cmd | F2802x0_common/cmd | F280230 memory linker command file. |
| F280260.cmd | F2802x0_common/cmd | F280260 memory linker command file. |
| F280270.cmd | F2802x0_common/cmd | F280270 memory linker command file. |

Table 2.9: Included Main Linker Command Files

3. **Set the CPU Frequency** In the F2802x0_common\include\F2802x0_Examples.h file specify the proper CPU frequency. Some examples are included in the file.

```
//*************************************************************
//* F2802x0_common\include\F2802x0_Examples.h
//*************************************************************
...
#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)
//#define CPU_RATE   41.667L   // for a 24MHz CPU clock speed  (SYSCLKOUT)
//#define CPU_RATE   50.000L   // for a 20MHz CPU clock speed  (SYSCLKOUT)
//#define CPU_RATE   66.667L   // for a 15MHz CPU clock speed  (SYSCLKOUT)
//#define CPU_RATE  100.000L   // for a 10MHz CPU clock speed  (SYSCLKOUT)
...
```

4. **Link desired common source files to the project** The common source files are found in the F2802x0_common\source directory.
5. **Include .c files for the PIE** Since all catalog 2802x0 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 2.9.2.1

## 2.6 Troubleshooting Tips and Frequently Asked Questions

- **In the examples, what do “EALLOW;” and “EDIS;” do?**
  EALLOW; is a macro defined in F2802x0_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.
  The driver library defines new names for these two statements in order to make the code more readable. The new names are: ”ENABLE_PROTECTED_REGISTER_WRITE_MODE;” and ”DISABLE_PROTECTED_REGISTER_WRITE_MODE;”. The definitions for these new macros can be found in cpu.h.
  For a complete list of protected registers, refer to *System Control and Interrupts* chapter of the *Technical Reference Manual*
- **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 F2802x0_common\source directory shows an example of enabling the peripheral clocks.
  * The driver library also includes functions to enable the clocks for a given peripheral. Take a look at clk.c in F2802x0_common\source.

  - Some peripherals are not present on all 2802x0 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 *System Control and Interrupts* chapter in the *Technical Reference Manual* 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 *System Control and Interrupts* chapter in the *Technical Reference Manual* 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 *System Control and Interrupts* chapter in the *Technical Reference Manual* for information on the code security module
- **A peripheral register reads back ok, but cannot be written to**
  The EALLOW bit protects some registers from spurious writes by the CPU. If your program seems unable to write to a register, then check to see if it is EALLOW protected. If it is, then enable access using the EALLOW assembly instruction. See *System Control and Interrupts* chapter in the *Technical Reference Manual* 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 F2802x0_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 F2802x0 examples, the code_start symbol is used instead. Refer to the source code for more information.
- **When I build many of the examples, the compiler outputs the following: remark: controlling expression is constant. What does this mean?**
  Some of the examples run forever until the user stops execution by using a while(1) loop. The remark refers to the while loop using a constant and thus the loop will never be exited.
- **When I build some of the examples, the compiler outputs the following: warning: statement is unreachable. What does this mean?**

Some of the examples run forever until the user stops execution by using a while(1) loop. If there is code after this while(1) loop then it will never be reached.

- **I changed the build configuration of one of the projects from "Debug" to "Release" and now the project will not build. What could be wrong?**
  When you switch to a new build configuration (Project->Active Build Configuration) the compiler and linker options changed for the project. The user must enter other options such as include search path and the library search path. Open the build options menu (Project-> Options) and enter the following information:
  * C2000 Compiler, Include Options: Include search path
  * C2000 Linker, File Search Path: Library search path
  * C2000 Linker, File Search Path: Include libraries(i.e. rts2800_ml.lib)

  Refer to section 2.5.3 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 by code. This is why the breakpoint is never hit. To avoid this, set the breakpoint after the SARAM functions have been copied to SARAM.

### 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 TMS320x2802x header files to the TMS320x2802x0 header files

This section includes suggestions for moving a project from the 2802x header files to the 2802x0 drivers.

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**
   Open the .project, .cdtbuild, and macros.ini files in your example folder. Replace all instances of 2802x with 2802x0 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_2802X_V170 for your 2802x project using 2802x header files V1.70, change this to INSTALLROOT_2802X0_V100 to migrate to the 2802x0 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 .project, .cdtbuild, and macros.ini files.
3. **Load the project into Code Composer Studio**
   Open the project in CCS, and open the build properties so that we can modify the include search paths. The structure of the include files has been modified to reduce the number of include paths that need to be added to each project. If there are paths to f2802x_common/include or f2802x_headers/include remove these and replace them with "${INSTALLROOT_F2802x0_VERSION}". Make sure your project includes DSP28x_Project.h and the appropriate header files will automatically be included.
4. **Link the driver library to your project**
   Traditional C28x projects required that many additional source files be included with a project for things such as header file global variable declaration and system control initialization. Most of these files have been built into the driver library such that the user need not compile the individual source files, just link in the library. To link in the library right click on the project and select link files. Navigate to f2802x0_common/lib and select driverlib.lib. If you do not wish to use the driver library this step can be omitted, but be sure your project has F2802x0_GlobalVariableDefs.c in it.
5. **Make sure you are using the correct linker command files (.cmd) appropriate for your device and for the F2802x0 header files**
   You will have one file for the memory definitions and one file for the header file structure definitions. Using a different memory file can cause issues because the memory map changes slightly between devices.

6. **Build the project**
   The compiler will highlight areas that have changed. If migrating from the ]MS320x2802x header files, code should be compatible with the exception of peripherals that aren't present on these devices.

## 2.8 F2802x driver API to F2802x0 driver library API code suggestions

1. **Initialize System Control**
   Headers:

```
InitSysCtrl();
```

   Drivers:

```
// Perform basic system initialization
WDOG_disable(myWDog);
CLK_enableAdcClock(myClk);
(*Device_cal)();

//Select the internal oscillator 1 as the clock source
CLK_setOscSrc(myClk, CLK_OscSrc_Internal);

// Setup the PLL for x10 /2 which will yield 50Mhz = 10Mhz * 10 / 2
PLL_setup(myPll, PLL_Multiplier_10, PLL_DivideSelect_ClkIn_by_2);

// Disable the PIE and all interrupts
PIE_disable(myPie);
PIE_disableAllInts(myPie);
CPU_disableGlobalInts(myCpu);
CPU_clearIntFlags(myCpu);
```

2. **Initialize PIE**
   Headers:

```
DINT;
InitPieCtrl();
IER = 0x0000;
IFR = 0x0000;
InitPieVectTable();
```

   Drivers:

```
PIE_setDebugIntVectorTable(myPie);
PIE_enable(myPie);
```

3. **Register an Interrupt Handler**
   Headers:

```
EALLOW;
PieVectTable.ADCINT1 = &adc_isr;
EDIS;
```

   Drivers:

```
PIE_registerPieIntHandler(myPie, PIE_GroupNumber_10, PIE_SubGroupNumber_1, (intV
```

4. **Initialize the ADC**
   Headers:

```
InitAdc();
```

   Drivers:

```
ADC_enableBandGap(myAdc);
ADC_enableRefBuffers(myAdc);
ADC_powerUp(myAdc);
ADC_enable(myAdc);
ADC_setVoltRefSrc(myAdc, ADC_VoltageRefSrc_Int);
```

5. **Initialize GPIO**
   Headers:

```
InitEPwm1Gpio();
```

   Drivers:

```
GPIO_setPullUp(myGpio, GPIO_Number_0, GPIO_PullUp_Disable);
GPIO_setPullUp(myGpio, GPIO_Number_1, GPIO_PullUp_Disable);
GPIO_setMode(myGpio, GPIO_Number_0, GPIO_0_Mode_EPWM1A);
GPIO_setMode(myGpio, GPIO_Number_1, GPIO_1_Mode_EPWM1B);
```

6. **Acknowledge a PIE interrupt**
   Headers:

```
PieCtrlRegs.PIEACK.all = PIEACK_GROUP3;
```

   Drivers:

```
PIE_clearInt(myPie, PIE_GroupNumber_3);
```

# 2.9 Packet Contents

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

## 2.9.1 Header File Support - F2802x0_headers

The F2802x0 header files are located in the <base>\F2802x0_headers directory.

### 2.9.1.1 F2802x0 Header Files - Main Files

The files listed in Table 2.10 must be added to any project that uses the F2802x0 header files. Refer to section 2.5 for information on incorporating the header files into a new or existing project.

| File | Location | Description |
|---|---|---|
| F2802x0_Device.h | . | 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. |
| F2802x0_GlobalVariableDefs.c | F2802x0_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. If a project includes the driverlib.lib file F2802x0_GlobalVariableDefs.c need not be included because it is already a part of the driver library. Refer to section 2.5. |
| F2802x0_Headers_nonBIOS.cmd | F2802x0_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 2.10: F2802x0 Header Files - Main Files

#### 2.9.1.2 F2802x0 Header Files - Peripheral Bit-Field and Register Structure Definition Files

The files listed in Table 2.11 define the bit-fields and register structures for each of the peripherals on the 2802x0 devices. These files are automatically included in the project by including F2802x0_Device.h. Refer to section 2.4.2 for more information on incorporating the header files into a new or existing project.

#### 2.9.1.3 Variable Names and Data Sections

This section is a summary of the variable names and data sections allocated by the F2802x0_headers\source\F2802x0_GlobalVariableDefs.c file as shown in Table 2.12. Note that all peripherals may not be available on a particular 2802x0 device. Refer to the device datasheet for the peripheral mix available on each 2802x0 family derivative.

| File | Description |
|---|---|
| F2802x0_Adc.h | ADC register structure and bit-field definitions. |
| F2802x0_BootVars.h | External boot variable definitions. |
| F2802x0_Comp.h | Comparator register structure and bit-field definitions. |
| F2802x0_CpuTimers.h | CPU-Timer register structure and bit-field definitions. |
| F2802x0_DevEmu.h | Emulation register definitions |
| F2802x0_ECap.h | eCAP register structures and bit-field definitions. |
| F2802x0_EPwm.h | ePWM register structures and bit-field definitions. |
| F2802x0_EQep.h | eQEP register structures and bit-field definitions. |
| F2802x0_Gpio.h | General Purpose I/O (GPIO) register structures and bit-field definitions. |
| F2802x0_I2c.h | I2C register structure and bit-field definitions. |
| F2802x0_NmiIntrupt.h | NMI interrupt register structure and bit-field definitions |
| F2802x0_PieCtrl.h | PIE control register structure and bit-field definitions. |
| F2802x0_PieVect.h | Structure definition for the entire PIE vector table. |
| F2802x0_Sci.h | SCI register structure and bit-field definitions. |
| F2802x0_Spi.h | SPI register structure and bit-field definitions. |
| F2802x0_SysCtrl.h | System register definitions. Includes Watchdog, PLL, CSM, Flash/OTP, Clock registers. |
| F2802x0_XIntrupt.h | External interrupt register structure and bit-field definitions. |

Table 2.11: F2802x0 Header File Bit-Field Register Structure Definition Files(F2802x0_headers\include)

| 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 | 0x3F7FF8-0x3F7FFF | CsmPwl |
| COMP1 | 0x006400 | Comp1Regs |
| COMP2 | 0x006420 | Comp2Regs |
| 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 |
| ePWM1 | 0x006800 | EPwm1Regs |
| ePWM2 | 0x006840 | EPwm2Regs |
| ePWM3 | 0x006880 | EPwm3Regs |
| eCAP1 | 0x006A00 | ECap1Regs |
| 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 |
| SPI-A | 0x007040 | SpiaRegs |

Table 2.12: F2802x0 Variable Names and Data Sections

## 2.9.2 Common Example Code - F2802x0_common

### 2.9.2.1 Peripheral Interrupt Expansion (PIE) Block Support

In addition to the register definitions defined in F2802x0_PieCtrl.h, this packet provides the basic ISR structure for the PIE block. These files are shown in Table 2.13.

| File | Location | Description |
|---|---|---|
| F2802x0_DefaultIsr.c | F2802x0_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. |
| F2802x0_DefaultIsr.h | F2802x0_common\include | Function prototype statements for the ISRs in F2802x0_DefaultIsr.c. Note: This file is not used for DSP/BIOS projects. |
| F2802x0_PieVect.c | F2802x0_common\source | Creates an instance of the PIE vector table structure initialized with pointers to the ISR functions in F2802x0_DefaultIsr.c. This instance can be copied to the PIE vector table in order to initialize it with the default ISR locations. |

Table 2.13: Basic PIE Block Specific Support Files

In addition, the files in Table 2.14 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 *F2802x0_examples\sw_prioritized_interrupts* for more information.

| File | Location | Description |
|---|---|---|
| F2802x0_SWPrioritizedDefaultIsr.c | F2802x0_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. |
| F2802x0_SWPrioritizedIsrLevels.h | F2802x0_common\include | Function prototype statements for the ISRs in F2802x0_DefaultIsr.c. Note: This file is not used for DSP/BIOS projects. |
| F2802x0_SWPrioritizedPieVect.c | F2802x0_common\source | Creates an instance of the PIE vector table structure initialized with pointers to the default ISR functions that are included in F2802x0_DefaultIsr.c. This instance can be copied to the PIE vector table in order to initialize it with the default ISR locations. |

Table 2.14: Software Prioritized Interrupt PIE Block Specific Support Files

#### 2.9.2.2 Peripheral Specific Files

Several peripheral specific initialization routines and support functions are included in the peripheral .c source files in the F2802x0_common\source directory. These files are shown in Table 2.15.

| File | Description |
|---|---|
| F2802x0_GlobalPrototypes.h | Function prototypes for the peripheral specific functions included in these files. |
| F2802x0_Adc.c | ADC specific functions and macros. |
| F2802x0_Comp.c | Comparator specific functions and macros |
| F2802x0_CpuTimers.c | CPU-Timer specific functions and macros. |
| F2802x0_ECap.c | eCAP module specific functions and macros. |
| F2802x0_EPwm.c | ePWM module specific functions and macros. |
| F2802x0_EPwm_defines.h | define macros that are used for the ePWM examples |
| F2802x0_Gpio.c | General-purpose IO (GPIO) specific functions and macros. |
| F2802x0_I2C.c | I2C specific functions and macros. |
| F2802x0_I2c_defines.h | define macros that are used for the I2C examples |
| F2802x0_PieCtrl.c | PIE control specific functions and macros. |
| F2802x0_Sci.c | SCI specific functions and macros. |
| F2802x0_Spi.c | SPI specific functions and macros. |
| F2802x0_SysCtrl.c | System control (watchdog, clock, PLL etc) specific functions and macros. |

Table 2.15: 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.**

#### 2.9.2.3 Utility Function Source Files

| File | Description |
|---|---|
| F2802x0_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. |
| F2802x0_DBGIER.asm | Assembly function to manipulate the DEBIER register from C. |
| F2802x0_DisInt.asm | Disable interrupt and restore interrupt functions. These functions allow you to disable INTM and DBGM and then later restore their state. |
| F2802x0_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 F2802x0_example\adc_soc for an example of its use. |
| F2802x0_CSMPasswords.asm | Include in a project to program the code security module passwords and reserved locations. |

Table 2.16: Included Utility Function Source Files

#### 2.9.2.4 Example Linker .cmd files

Example memory linker command files are located in the F2802x0_common\cmd directory. For getting started the basic 280270_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 2.9).

#### 2.9.2.5 Example Library .lib Files

Example library files are located in the F2802x0_common\lib directory. For this release the IQMath library is included for use in the example projects. Please refer to the *C28x IQMath Library - A Virtual Floating Point Engine (SPRC087)* for more information on IQMath and the most recent IQMath library.

## 2.10 Detailed Revision History

**V2.00**

- A driver library was added and major changes made to many of the examples and support files. This is a major release.

**V1.29**

- Additional cleanup of the CCS 4 projects in controlSUITE, and GEL files have been updated.

**V1.28**

- Improvements and additions to the CCS 4 projects in controlSUITE.

**V1.27**

- Improvements to the CCS 4 projects in controlSUITE.

**V1.26**

- This version includes minor corrections to the header files and examples.

**V1.25**

- This version includes minor corrections to the header files and examples. The most notable change is that gel files and cmd linker files for the 280200 devices now include 1K additional L0 RAM.

**V1.21b**

- This version update only updates the V1.21 Quick Start Readme to adjust wording for the controlSUITE software package. No changes to the header file and peripheral example code were made.

**V1.21**

- This version includes minor corrections and comment fixes to the header files and examples.

**V1.20**

- This version includes corrections and comment fixes to the header files and examples. It adds examples pertaining to the ADC temperature sensor and compensation of the oscillator frequency over temperature, and it also fixes an error in the SFO$_T I_B uild_V 6.liblibraryinthenewSFO_T I_B uild_V 6b.liblibrary.$

**V1.10**

- This version is the second release of the F2802x0 header files and examples. Minor changes were made to the drivers and a header file to driver migration guide is included in the documentation.

**V1.00**

- This version is the first release (packaged with development tools and customer trainings) of the F2802x0 header files and examples.

# 3 Piccolo F2802x0 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 F2802x0 header files
- All examples set up the PLL in x10/2 mode which gives a system clock of 50MHz. This is the default setting assuming the input clock is derived from the 10MHz internal clock.
- Some examples like those related to HRPWM require the use of an external scope to see the results, while other examples may require external connections between headers on the base-board (e.g. adc_soc). Each example will describe the setup procedure that is required to properly execute it.
- As supplied, almost all projects are supplied with both RAM and Flash build configurations. The 2802x0 Boot Mode table is shown below.
  - While an emulator is connected to your device, the TRSTn pin = 1, which sets the device into EMU_BOOT boot mode. In this mode, the peripheral boot modes are shown in the table below.
  - Write EMU_KEY to 0xD00 and EMU_BMODE to 0xD01 via the debugger with the values from the table
  - Build/Load project, reset the device, and run the example

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

Table 3.1: Boot Modes for Piccolo 2802x0

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

All of these examples reside in the `F2802x0_examples_ccsv4` subdirectory of the ControlSUITE package.

# 3.1 Examples

## ADC Start-Of-Conversion (SOC)

Interrupts are enabled and the ePWM1 is setup 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

## ADC Temperatrue Sensor

Interrupts are enabled and 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

## ADC Temperature Sensor Conversion

This program shows how to convert a raw ADC temperature sensor reading into deg. C or deg. K.
Watch Variables

- temp
- degC
- degK

## CPU Timer

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

- timer0IntCount
- timer1IntCount
- timer2IntCount

## ECAP Asymetric PWM

This program sets up the eCAP pins in the APWM mode. This program runs at 50 MHz or 40 MHz SYSCLKOUT assuming a 10 MHz OSCCLK depending on the max frequency allowed by a particular device.

eCAP1 will come out on the GPIO5 pin. This pin is configured to vary between 3 Hz and 6 Hz (at 50 MHz SYSCLKOUT) or 2 Hz and 4 Hz (at 40 MHz SYSCLKOUT) using the shadow registers to load the next period/compare values.

Monitor eCAP1 pin on GPIO5 for PWM frequency

## ECAP Capture EPwm3

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 PWM3A output. Connect eCAP1 (GPIO5) to ePWM3A (GPIO4).

## PWM Blanking Window

This example configures ePWM1 and ePWM2

2 Examples are included:

- 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.

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

EPWM1A is on GPIO0

EPWM1B is on GPIO1

EPWM2A is on GPIO2

EPWM2B is on GPIO3

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, but if there is no event, EPWM1A will remain low. EPWM1B is togged at zero for a reference. Notice the blanking window keeps the event from forcing EPWM1A high around the zero point.

ePWM2 is configured the same as ePWM1 except no blanking window is applied.

View the EPWM1A/B, EPWM2A/B waveforms via an oscilloscope to see the effect of DCAEVT1

## PWM Digital Compare Event Trip Zone

This example configures ePWM1 and ePWM2

2 Examples are included:

- ePWM1 has DCAEVT1 as a one shot trip source
- ePWM2 has DCAEVT2 as a cycle by cycle trip source
- ePWM3 reacts to DCAEVT2 and DCBEVT1 events

During the test, monitor ePWM1, ePWM2, or ePWM3 outputs on a scope pull TZ1 low and leave TZ2 high to create a DCAEVT1, DCAEVT2, DCBEVT1 and DCBEVT2.

EPWM1A is on GPIO0

EPWM1B is on GPIO1

EPWM2A is on GPIO2

EPWM2B is on GPIO3

EPWM3A is on GPIO4

EPWM3B is on GPIO5

DCAEVT1, DCAEVT2, DCBEVT1 and DCBEVT2 are all defined as true when TZ1 is low and TZ2 is high

ePWM1 will react to DCAEVT1 as a 1-shot trip. The trip event will pull EPWM1A high. The trip event will pull EPWM1B low.

ePWM2 will react to DCAEVT2 as a cycle-by-cycle trip. The trip event will pull EPWM2A high. The trip event will pull EPWM2B low.

ePWM3 will react to DCAEVT2 and DCBEVT1 events. The DCAEVT2 event will pull EPWM3A high. The DCBEVT1 event will pull EPWM3B low.

## PWM Trip Zone Test with Comparator Inputs

This example configures ePWM1 and its associated trip zone.

Initially make voltage on pin COMP1A greater than COMP1B if using dual pin compare; else make internal DAC output lower than V on COMP1A.

During the test, monitor ePWM1 outputs on a scope. Increase the voltage on inverting side of comparator(either through COMP1B pin or internal DAC setting) to trigger a DCAEVT1, and DCBEVT1.

EPWM1A is on GPIO0

EPWM1B is on GPIO1

DCAEVT1, DCBEVT1 a are all defined as true when COMP1OUT is low.

ePWM1 will react to DCAEVT1 and DCBEVT1 as a 1 shot trip. DCAEVT1 will pull EPWM1A high. DCBEVT1 will pull EPWM1B low .

## PWM 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.

View the EPWM1A/B, EPWM2A/B and EPWM3A/B waveforms via an oscilloscope: EPWM1A is on GPIO0

EPWM1B is on GPIO1

EPWM2A is on GPIO2

EPWM2B is on GPIO3
EPWM3A is on GPIO4
EPWM3B is on GPIO5

## PWM 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) toggled in main loop.

ControlCard LED3 (GPIO34) 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.

As supplied:

ePWM1 is initalized.

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

## PWM Timer Interrupt

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

As supplied:

All ePWM's are initalized.

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

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

## PWM Trip Zone

This example configures ePWM1 and ePWM2

2 Examples are included:

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

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.

View the EPWM1A/B, EPWM2A/B waveforms via an oscilloscope to see the effect of TZ1 and TZ2

Initially tie TZ1 (GPIO12) and TZ2 (GPIO13) high.

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

EPWM1A is on GPIO0

EPWM1B is on GPIO1

EPWM2A is on GPIO2

EPWM2B is on GPIO3

ePWM1 will react as a 1 shot trip.

ePWM2 will react as a cycle by cycle trip and will be cleared if TZ1 and TZ2 are both pulled back high.

## Action Qualifier Module Upcount mode

This example configures ePWM1, ePWM2, ePWM3 to produce a waveform with independant modulation on EPWMxA and EPWMxB.

The compare values CMPA and CMPB are modified within the ePWM's ISR.

The TB counter is in upmode for this example.

View the EPWM1A/B, EPWM2A/B and EPWM3A/B waveforms via an oscilloscope:

EPWM1A is on GPIO0

EPWM1B is on GPIO1

EPWM2A is on GPIO2

EPWM2B is on GPIO3

EPWM3A is on GPIO4

EPWM3B is on GPIO5

## Action Qualifier Module - Using up/down count

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 up/down count mode for this example.

View the EPWM1A/B, EPWM2A/B and EPWM3A/B waveforms via an oscilloscope:

EPWM1A is on GPIO0

EPWM1B is on GPIO1

EPWM2A is on GPIO2

EPWM2B is on GPIO3

EPWM3A is on GPIO4

EPWM3B is on GPIO5

## External Interrupts

This program sets up GPIO0 as XINT1 and GPIO1 as XINT2. Two other GPIO signals are used to trigger the interrupt (GPIO28 triggers XINT1 and GPIO29 triggers XINT2). The user is required to externally connect these signals for the program to work properly.

XINT1 input is synched to SYSCLKOUT. XINT2 has a long qualification - 6 samples at 510∗SYSCLKOUT each.

GPIO34 will go high outside of the interrupts and low within the interrupts. This signal can be monitored on a scope.

Each interrupt is fired in sequence - XINT1 first and then XINT2

Watch Variables:

- Xint1Count for the number of times through XINT1 interrupt
- Xint2Count for the number of times through XINT2 interrupt
- LoopCount for the number of times through the idle loop

This example runs the EPwm interrupt example from flash.

(a) Build the project
(b) Flash the .out file into the device.
(c) Set the hardware jumpers to boot to Flash
(d) 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:

(a) Change the linker cmd file to reflect the flash memory map.
(b) 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.
(c) 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.
(d) Set boot mode Jumpers to "boot to Flash"
(e) 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.

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.

As supplied:

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.

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

Watch Variables:

- EPwm1TimerIntCount
- EPwm2TimerIntCount
- EPwm3TimerIntCount

Toggle GPIO34 while in the background loop.

## GPIO Setup

Configures the 2802x 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 is synch to SYSCLKOUT
- Input qual for some I/O's and interrupts may have a sampling window

## GPIO Toggle

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

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

The pins can be observed using Oscilloscope.

## High Resolution PWM

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

(a) PWM Freq = SYSCLK/(period=10), ePWM1A toggle low/high with MEP control on rising edge
PWM Freq = SYSCLK/(period=10), ePWM1B toggle low/high with NO HRPWM control

(a) PWM Freq = SYSCLK/(period=20), ePWM2A toggle low/high with MEP control on rising edge
PWM Freq = SYSCLK/(period=20), ePWM2B toggle low/high with NO HRPWM control

(a) PWM Freq = SYSCLK/(period=10), ePWM3A toggle as high/low with MEP control on falling edge
PWM Freq = SYSCLK/(period=10), ePWM3B toggle low/high with NO HRPWM control

(a) PWM Freq = SYSCLK/(period=20), ePWM4A toggle as high/low with MEP control on falling edge
PWM Freq = SYSCLK/(period=20), ePWM4B toggle low/high with NO HRPWM control

Monitor ePWM1-ePWM4 pins on an oscilloscope.

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

## High Resolution PWM SFO Duty Cycle Control

This example modifies the MEP control registers to show edge displacement due to the HRPWM control extension of the respective ePWM module.

This example calls the following TI's MEP Scale Factor Optimizer (SFO) software library V6 functions:

int SFO(); updates MEP_ScaleFactor dynamically when HRPWM is in use updates HRMSTEP register (exists only in EPwm1Regs register space but valid for all channels) with MEP_ScaleFactor value

- returns 2 if error: MEP_ScaleFactor is greater than maximum value of 255 (Auto-conversion may not function properly under this condition)
- returns 1 when complete for the specified channel
- returns 0 if not complete for the specified channel

This example is intended to explain the HRPWM capabilities. The code can be optimized for code efficiency. Refer to TI's Digital power application examples and TI Digital Power Supply software libraries for details.

All ePWM1A-4A channels (GPIO0 through GPI07) will have fine edge movement due to the HRPWM logic

==========
NOTE: For more information on using the SFO software library, see the 2802x High-Resolution Pulse Width Modulator (HRPWM) Reference Guide
==========

To load and run this example:

(a) ∗∗!!IMPORTANT!!∗∗ - in SFO_V6.h, set PWM_CH to the max number of HRPWM channels plus one. For example, for the F2802x, the maximum number of HRPWM channels is 4. 4+1=5, so set define PWM_CH 5 in SFO_V6.h. (Default is 5)

(b) In this file, set define AUTOCONVERT to 1 to enable MEP step auto-conversion logic. Otherwise, to manually perform MEP calculations in software, clear to 0.

(c) Run this example at maximum SYSCLKOUT (60 or 40 MHz)

(d) Load the Example_2802xHRPWM_Duty_SFO_V6.gel and observe variables in the watch window

(e) Activate Real time mode

(f) Run the code

(g) Watch ePWM1-4 waveforms on a Oscillosope

(h) In the watch window: Set the variable UpdateFine = 1 to observe the ePWMxA output with HRPWM capabilites (default) Observe the duty cycle of the waveform change in fine MEP steps

(i) In the watch window: Change the variable UpdateFine to 0, to observe the ePWMxA output without HRPWM capabilites Observe the duty cycle of the waveform change in coarse SYSCLKOUT cycle steps.

## High Resolution PWM SFO Period Control

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

Notice that all period and compare register updates occur in an ISR which interrupts at ePWM1 TBCTR = 0. This ensures that period and compare register updates across all ePWM modules occur within the same period.

This example calls the following TI's MEP Scale Factor Optimizer (SFO) software library V6 functions:

int SFO(); updates MEP_ScaleFactor dynamically when HRPWM is in use updates HRMSTEP register (exists only in EPwm1Regs register space) with MEP_ScaleFactor value

- returns 2 if error: MEP_ScaleFactor is greater than maximum value of 255 (Auto-conversion may not function properly under this condition)
- returns 1 when complete for the specified channel
- returns 0 if not complete for the specified channel

All ePWM1A-4A channels will be synchronized to each other (with ePWM1 sync'd to the SWFSYNC) and have fine edge period movement due to the HRPWM logic.

This example can be used as a primitive building block for applications which require high resolution frequency control with synchronized ePWM modules (i.e. resonant converter applications)

==========
NOTE: For more information on using the SFO software library, see the 2802x High-Resolution Pulse Width Modulator (HRPWM) Reference Guide
==========

To load and run this example:

(a) $**$!!IMPORTANT!!$**$ - in SFO_V6.h, set PWM_CH to the max number of HRPWM channels plus one. For example, for the F2802x, the maximum number of HRPWM channels is 4. 4+1=5, so set define PWM_CH 5 in SFO_V6.h. (Default is 5)
(b) Run this example at maximum SYSCLKOUT (60 MHz or 40 MHz)
(c) Add "UpdateFine" and "UpdateCourse" variables to the watch window.
(d) Activate Real time mode
(e) Run the code
(f) Watch ePWM A channel waveforms on a Oscillosope
(g) In the watch window: Set the variable UpdateFine = 1 to observe the ePWMxA output with HRPWM capabilites (default) Observe the period/frequency of the waveform changes in fine MEP steps
(h) In the watch window: Change the variable UpdateFine to 0, to observe the ePWMxA output without HRPWM capabilites To change the period/frequency in coarse steps, uncomment the relevant code, re-build and re-run with UpdateCoarse = 1. Observe the period/frequency of the waveform changes in coarse SYSCLKOUT cycle steps.

## High Resolution PWM SFO Period (Up Count) Control

This example modifies the MEP control registers to show edge displacement for high-resolution period with ePWM in Up count mode due to the HRPWM control extension of the respective ePWM module.

This example calls the following TI's MEP Scale Factor Optimizer (SFO) software library V6 functions:

int SFO(); updates MEP_ScaleFactor dynamically when HRPWM is in use updates HRMSTEP register (exists only in EPwm1Regs register space) with MEP_ScaleFactor value

- returns 2 if error: MEP_ScaleFactor is greater than maximum value of 255 (Auto-conversion may not function properly under this condition)
- returns 1 when complete for the specified channel
- returns 0 if not complete for the specified channel

This example is intended to explain the HRPWM capabilities. The code can be optimized for code efficiency. Refer to TI's Digital power application examples and TI Digital Power Supply software libraries for details.

All ePWM1A-4A channels (GPIO0 through GPI07) will have fine edge movement due to the HRPWM logic

==========
NOTE: For more information on using the SFO software library, see the 2802x High-Resolution Pulse Width Modulator (HRPWM) Reference Guide
==========

To load and run this example:

(a) **!!IMPORTANT!!** - in SFO_V6.h, set PWM_CH to the max number of HRPWM channels plus one. For example, for the F2802x, the maximum number of HRPWM channels is 4. 4+1=5, so set define PWM_CH 5 in SFO_V6.h. (Default is 5)

(b) Run this example at maximum SYSCLKOUT (60 or 40 MHz)

(c) Load the Example_2802xHRPWM_PrdUpDown_SFO_V6.gel and observe variables in the watch window

(d) Activate Real time mode

(e) Run the code

(f) Watch ePWM1-4 waveforms on a Oscillosope

(g) In the watch window: Set the variable UpdateFine = 1 to observe the ePWMxA output with HRPWM capabilites (default) Observe the period/frequency of the waveform changes in fine MEP steps

(h) In the watch window: Change the variable UpdateFine to 0, to observe the ePWMxA output without HRPWM capabilites Observe the period/frequency of the waveform changes in coarse SYSCLKOUT cycle steps.

## High Resolution PWM SFO Period (Up-Down Count) Control

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

This example calls the following TI's MEP Scale Factor Optimizer (SFO) software library V6 functions:

int SFO(); updates MEP_ScaleFactor dynamically when HRPWM is in use updates HRMSTEP register (exists only in EPwm1Regs register space) with MEP_ScaleFactor value

- returns 2 if error: MEP_ScaleFactor is greater than maximum value of 255 (Auto-conversion may not function properly under this condition)
- returns 1 when complete for the specified channel
- returns 0 if not complete for the specified channel

This example is intended to explain the HRPWM configuration for high resolution period/frequency. The code can be optimized for code efficiency. Refer to TI's Digital power application examples and TI Digital Power Supply software libraries for details.

ePWM1A (GPIO0) will have fine edge movement due to the HRPWM logic

==========
NOTE: For more information on using the SFO software library, see the 2802x High-Resolution Pulse Width Modulator (HRPWM) Reference Guide
==========

To load and run this example:

(a) **!!IMPORTANT!!** - in SFO_V6.h, set PWM_CH to the max used HRPWM channel # plus one. For example, for the F2802x, the maximum number of HRPWM channels is 4. 4+1=5, so set define PWM_CH 5 in SFO_V6.h. (Default is 5)

- Note for this specific example, you could set define PWM_CH 2 (because it only uses ePWM1), but to cover all examples, PWM_CH is currently set to the maximum possible for the device.

(a) Load the code and add the following watch variables to the watch window:
- UpdateFine
- PeriodFine
- EPwm1Regs.TBPRD
- EPwm1Regs.TBPRDHR

(b) Run this example at maximum SYSCLKOUT (60 or 40 MHz)

(c) Activate Real time mode

(d) Run the code

(e) Watch ePWM1A waveform on a Oscillosope

(f) In the watch window: Set the variable UpdateFine = 1 to observe the ePWMxA output with HRPWM capabilites (default) Observe the period/frequency of the waveform changes in fine MEP steps

(g) In the watch window: Change the variable UpdateFine to 0, to observe the ePWMxA output without HRPWM capabilites

## High Resolution PWM Slider

This example modifies the MEP control registers to show edge displacement due to HRPWM control blocks of the respective EPwm module, EPwm1A, 2A, 3A, and 4A channels (GPIO0, GPIO2, GPIO4, and GPIO6) will have fine edge movement due to HRPWM logic. Load the Example_2802xHRPWM_slider.gel file. Select the HRPWM_eval from the GEL menu. A FineDuty slider graphics will show up in CCS. Load the program and run. Use the Slider to and observe the EPwm edge displacement for each slider step change. This explains the MEP control on the EPwmxA channels

(a) PWM Freq = SYSCLK/(period=10), ePWM1A toggle low/high with MEP control on rising edge
PWM Freq = SYSCLK/(period=10), ePWM1B toggle low/high with NO HRPWM control

(a) PWM Freq = SYSCLK/(period=20), ePWM2A toggle low/high with MEP control on rising edge
PWM Freq = SYSCLK/(period=20), ePWM2B toggle low/high with NO HRPWM control

(a) PWM Freq = SYSCLK/(period=10), ePWM3A toggle as high/low with MEP control on falling edge
PWM Freq = SYSCLK/(period=10), ePWM3B toggle low/high with NO HRPWM control

(a) PWM Freq = SYSCLK/(period=20), ePWM4A toggle as high/low with MEP control on falling edge
PWM Freq = SYSCLK/(period=20), ePWM4B toggle low/high with NO HRPWM control

## I2C EEPROM

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.

This program will work with the on-board I2C EEPROM supplied on the F2802x eZdsp or another EEPROM connected to the devices I2C bus with a slave address of 0x50

## LED BoosterPack Capacitive Touch Demo

This example allows the user to control the LED Boosterpack's color by spinning their finger on the capacitive touch boosterpack when plugged into the LED boosterpack.

For this example to work S4 on the C2000 LaunchPad should be in the down position, while S1 on the LED Boosterpack should be in the up position.

This example uses the same control techniques used in the other LED BoosterPack examples, but uses the MSP430 present on the LED BoosterPack as a sensor hub. The MSP430 interfaces with the Capacitive Touch BoosterPack directly and reports actions to the Piccolo device on the C2000 LaunchPad via an asynchronous serial interface.

After loading and running this example, press the center section of the capacitive touch BoosterPack twice to turn on the LEDs. Moving ones finger around the outer sections of the touchpad will change the LED color. Touching the center section again will turn off the LEDs.

## LED BoosterPack PC GUI Demo

This example allows the user to control the LED Boosterpack's color and intensity via a GUI application on a host PC.

For this example to work S4 on the C2000 LaunchPad should be in the up position, while S1 on the LED Boosterpack should be in the down position.

This example uses the same control techniques used in the other LED BoosterPack examples and uses a serial communications link to communicate with a PC Gui application to allow for control of the BoosterPack's LEDs.

After loading and running this example, open up the PC GUI application and connect to the XDS100's USB/Serial Port. Move the slides to turn on and off the LEDs and adjust the color.

## Device 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 (toggled in WAKEINT ISR)

## Device Idle Mode and Wakeup

This example puts the device into IDLE mode.

The example 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.

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

## 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.

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.

## Internal Oscillator Compensation

This program shows how to use the internal oscillator compensation functions in osc.c or F2802x_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.

Watch Variables:

- temp
- SysCtrlRegs.INTOSC1TRIM
- SysCtrlRegs.INTOSC2TRIM

## SCI Echoback

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

(a) Configure hyperterminal: Use the included hyperterminal configuration file SCI_96.ht. To load this configuration in hyperterminal: file->open and then select the SCI_96.ht file.

(b) 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.

(c) Connect hyperterminal Call->Call and then start the 2802x SCI echoback program execution.

(d) The program will print out a greeting and then ask you to enter a character which it will echo back to hyperterminal.

Watch Variables:

- LoopCount for the number of characters sent
- ErrorCount

## SCI FIFO Digital Loop Back

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
- RecievedChar - Character recieved

## SCI Digital Loop Back with Interrupts

This program is a SCI example that uses the internal loopback of the peripheral. Both interrupts and the SCI FIFOs are used.

A stream of data is sent and then compared to the recieved 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

## SPI Digital Loop Back

This program is a SPI example that uses the internal loopback of the peripheral. Interrupts are not used.
A stream of data is sent and then compared to the recieved 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

## SPI Digital Loop Back with Interrupts

This program is a SPI-A example that uses the internal loopback of the peripheral. 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[2] - Data to send
- rdata[2] - Received data
- rdata_point - Used to keep track of the last position in the receive stream for error checking.

## Software Prioritized Interrupts

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

For more information on F2802x interrupt priorities, refer to the Software ISR Priorities section of the firmware examples guide document included with the F2802x/doc directory.

This program simulates interrupt conflicts by writing to the PIEIFR registers. This will simulate multiple interrupts coming into the PIE block at the same time.

The interrupt service routine routines are software prioritized by the table found in the F2802x_SWPrioritizedIsrLevels.h file.

(a) Before compiling you must set the Global and Group interrupt priorities in the F2802x_SWPrioritizedIsrLevels.h file.
(b) Set the define CASE directive at the top of the code in this file to determine which test case to run
(c) Compile the code, load, and run
(d) 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 add to the ISRTrace.
(e) If desired, set a new set of Global and Group interrupt priorities and repeat the test to see the change.

Watch Variables:

- ISRTrace[50] - 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. The ISRTrace will consist of a list of hex values as shown:
  0x00wx <- PIE Group w interrup x finished first
  0x00yz <- PIE Group y interrupt z finished next

## LED Blink

This example configures CPU Timer0 for a 500 msec period, and toggles the GPIO0-4 LEDs once per interrupt. For testing purposes, this example also increments a counter each time the timer asserts an interrupt.

Watch Variables:

- interruptCount

Monitor the GPIO0-4 LEDs blink on (for 500 msec) and off (for 500 msec) on the 2802x0 control card.

## Watchdog Interrupt

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 one line of code in the infinite loop.

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

Watch Variables:

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

# A Interrupt Service Routine Priorities



## A.1 Interrupt Hardware Priority Overview

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

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

CPU Interrupts INT1 - INT14, DLOGINT and RTOSINT are maskable interrupts. These interrupts can be enabled or disabled by the CPU Interrupt enable register (IER).
**Group Priority (PIE Level):**
If the Peripheral Interrupt Expansion (PIE) block is enabled, then CPU interrupts INT1 to INT12 are connected to the PIE. This peripheral expands each of these 12 CPU interrupt into 8 interrupts. Thus the total possible number of available interrupts in the PIE is 96. Note, not all of the 96 are used on a 2802x0 device.
Each of the PIE groups has its own interrupt enable register (PIEIERx) to control which of the 8 interrupts (INTx.1 - INTx.8) are enabled and permitted to issue an interrupt.

## A.2 F2802x0 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 2802x0 system can be categorized as follows (ordered highest to lowest priority):

| 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) **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 2802x0, 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.

(b) **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 2802x0, 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

(c) **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 2802x0 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.

(d) **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/ McBSP) 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 2802x0, such interrupts are mapped to INT6, INT8, and INT9, which are the next lowest hardware priority.

# A.3 Software Prioritization of Interrupts - The DSP28 Example

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

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

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

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

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

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

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

(c) **Enable interrupts**

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

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

This is done in the F2802x0_SWPrioritizedIsrLevels.h file as follows:

(a) *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.

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

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

Once the user has defined the global and group priority levels, the compiler will generate mask values that can be used to change the IER and PIEIERx registers within each ISR. In this manner the interrupt software prioritization will be changed. The masks that are generated at compile time are:

- **IER mask values**
  MINT1 - MINT16

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

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

## A.3.1 Using the IER/PIEIER Mask Values

Within an interrupt service routine, the global and group priority can be changed by software to allow other interrupts to be serviced. The procedure for setting an interrupt priority using the mask values created in the F2802x0_SWPrioritizedIsrLevels.h is the following:

(a) **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.

(b) **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.

(c) **Enable interrupts**
- Enable all PIE interrupt groups by writing all 1's to the PIEACK register
- Enable global interrupts by clearing INTM

(d) **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.

(e) **Restore the PIEIERx register**

(f) **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 $Y = mX + b$:

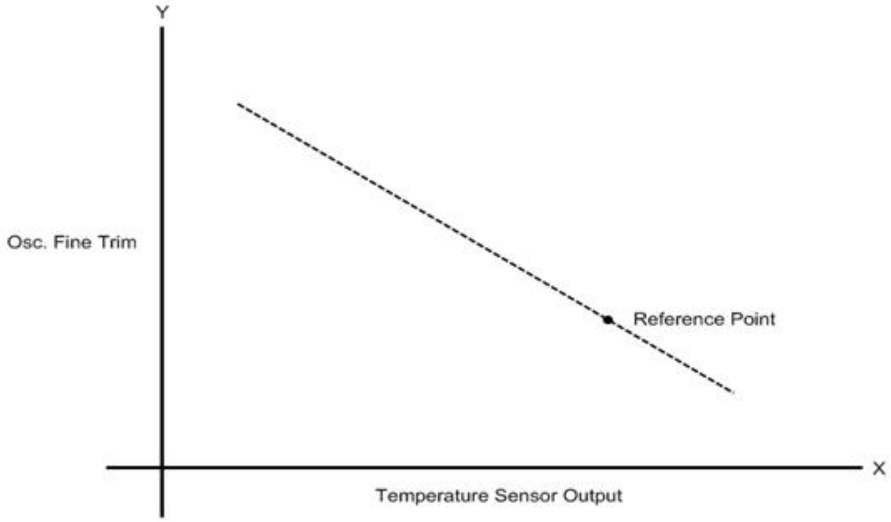


Figure B.1: Oscillator Reference

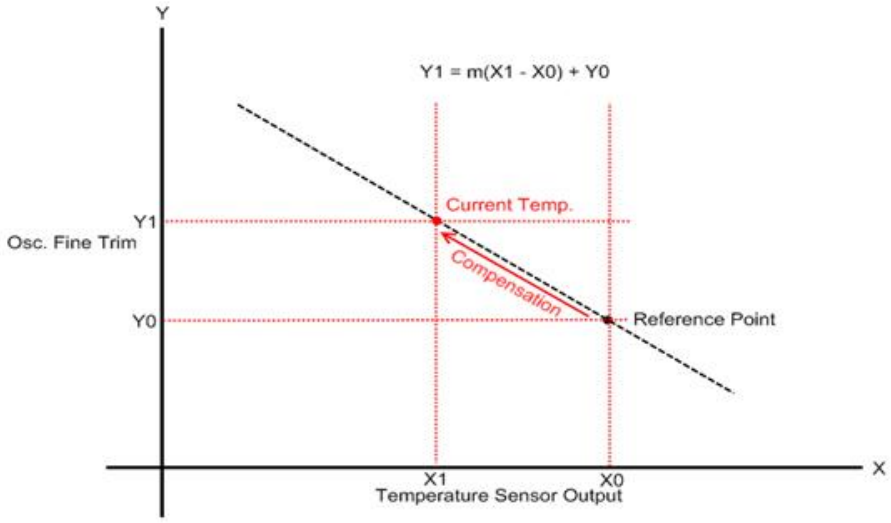


Figure B.2: Oscillator Fine Trim Compensation for change in Temperature

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

## B.2.1 OTP Functions

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

Similar functions following the new conventions of the driver library model can be found in *f2802x0_common/include/osc.h*. These functions are functionally identical but are named with an OSC_ prefix. For instance instead of getOsc1FineTrimOffset the function is called OSC_getFineTrimOffset1.

**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