THE DATASHEET OF

TMDSC2KWRKSHPKIT

Running an Application from Internal Flash Memory on the TMS320F28xxx DSP

David M. Alter *Embedded Processors and Microcontrollers - Semiconductor Group*

ABSTRACT

Several special requirements exist for running an application from on-chip flash memory on the TMS320F28xxx. These requirements generally do not manifest themselves during code development in RAM since the Code Composer Studio™ (CCS) debugger can mask problems associated with initialized sections and their linkage in memory. This application report covers the application software modifications needed for execution from on-chip flash memory. Requirements for both DSP/BIOS™ and non-DSP/BIOS projects are presented. Some performance considerations and techniques are also discussed.

Example CCS v5 projects are provided for the F2812, F2808, F28335, F28027, F28035, F28055, and F28069 (i.e., usually the superset device in each F28xxx sub-family). These can be downloaded from http://www-s.ti.com/sc/techlit/spra958.zip, and can provide a starting point for code development independent of this application report.

Note that the issues discussed in this application report apply to these current members of the TMS320F28xxx device family, specifically:

F281x: F2810, F2811, F2812
F280x/2801x/28044: F2801, F2802, F2806, F2808, F2809, F28015, F28016, F28044
F2823x/2833x: F28232, F28234, F28235, F28332, F28334, F28335
F2802x: F28020, F28021, F28022, F28023, F28026, F28027, F280200
F2803x: F28030, F28031, F28032, F28033, F28034, F28035
F2805x: F28050, F28051, F28052, F28053, F28054, F28055
F2806x: F28062, F28063, F28064, F28065, F28066, F28067, F28068, F28069

Applicability to other F28xxx devices, although likely, is not guaranteed. Further, the code and methods presented apply to the development tool versions utilized, specifically:

CCS v5.3.0, Code Generation Tools v6.1.1, DSP/BIOS v5.42.0.07

Be aware that future tool versions may have differences although in all likelihood backwards compatibility will be maintained so that the techniques discussed here should still work. Also note that the operation and setup for SYS/BIOS (BIOS v6) differs from that of DSP/BIOS (BIOS v5). This application report only applies to DSP/BIOS.

Finally, this application report does not provide a tutorial on writing and building code for the F28xxx. It is assumed that the reader already has at least the main framework of their application running from RAM. This report only identifies the special items that must be considered when moving the application into on-chip flash memory.

Code Composer Studio and DSP/BIOS are trademarks of Texas Instruments.
Trademarks are the property of their respective owners.

Contents

1 Introduction 3
2 Creating a User Linker Command File 3
2.1 Non-DSP/BIOS Projects 3
2.2 DSP/BIOS Projects 4
3 Where to Link the Sections 5
3.1 Non-DSP/BIOS Projects 6
3.2 DSP/BIOS Projects 7
4 Copying Sections from Flash to RAM 9
4.1 Copying the Interrupt Vectors (non-DSP/BIOS projects only) 9
4.2 Copying the .hwi_vec Section (DSP/BIOS projects only) 10
4.3 Copying the .trcdata Section (DSP/BIOS projects only) 11
4.4 Initializing the Flash Control Registers (DSP/BIOS and non-DSP/BIOS projects) 12
4.5 Maximizing Performance by Executing Time-critical Functions from RAM 15
4.6 Maximizing Performance by Linking Critical Global Constants to RAM 15
4.6.1 Method 1: Running All Constant Arrays from RAM 16
4.6.2 Method 2: Running a Specific Constant Array from RAM 19
5 Programming the Code Security Module Passwords 20
5.1 Single-Zone Security Devices (DSP/BIOS and non-DSP/BIOS projects) 21
5.2 Dual-Zone Security Devices (DSP/BIOS and non-DSP/BIOS projects) 24
6 Executing Your Code from Flash after a DSP Reset 32
7 Disabling the Watchdog Timer During C-Environment Boot 34
8 C-Code Examples 36
8.1 General Overview 36
8.2 Directory Structure 38
8.3 Additional Information 39
References 43
Revision History 44

Figures

Figure 1. Specifying the User Init Function in the DSP/BIOS Configuration tool 11
Figure 2. Specifying the Link Order In Code Composer Studio v5 18
Figure 3. DSP/BIOS MEM Properties for CSM Password Locations 23
Figure 4. DSP/BIOS MEM Properties for CSM Reserved Locations 24
Figure 5. DCSM Zone Select Block 25
Figure 6. DCSM Security Zone Configuration Table OTP Memory 26
Figure 7. DSP/BIOS MEM Properties for DCSM_OTP_Z2_P0 Memory 29
Figure 8. DSP/BIOS MEM Properties for DCSM_ZSEL_Z2 Memory 30
Figure 9. DSP/BIOS MEM Properties for DCSM_OTP_Z1_P0 Memory 30
Figure 10. DSP/BIOS MEM Properties for DCSM_ZSEL_Z1_P0 Memory 31
Figure 11. DSP/BIOS MEM Properties for Jump to Flash Entry Point 33

Tables

Table 1. Section Linking for Non-DSP/BIOS Projects (Large memory model) 6
Table 2. Section Linking for DSP/BIOS Projects (Large Memory Model) 7
Table 3. CCS Example Code Directory Descriptions 39

1 Introduction

The TMS320F28xxx DSP family has been designed for standalone operation in embedded controller applications. The on-chip flash usually eliminates the need for external non-volatile memory and a host processor from which to bootload. Configuring an application to run from flash memory is a relatively easy matter provided that one follows a few simple steps. This report covers the major concerns and steps needed to properly configure application software for execution from internal flash memory. Requirements for both DSP/BIOS and non-DSP/BIOS projects are presented. Some performance considerations and techniques are also discussed.

Example CCS v5 projects are provided for the F2812, F2808, F28335, F28027, F28035, F28055, and F28069 (i.e., usually the superset device in each F28xxx sub-family). These can be downloaded from http://www-s.ti.com/sc/techlit/spra958.zip, and can provide a starting point for code development independent of this application report.

Note that the issues discussed in this application report apply to these current members of the TMS320F28xxx device family, specifically:

F281x: F2810, F2811, F2812
F280x/2801x/28044: F2801, F2802, F2806, F2808, F2809, F28015, F28016, F28044
F2823x/2833x: F28232, F28234, F28235, F28332, F28334, F28335
F2802x: F28020, F28021, F28022, F28023, F28026, F28027, F280200
F2803x: F28030, F28031, F28032, F28033, F28034, F28035
F2805x: F28050, F28051, F28052, F28053, F28054, F28055
F2806x: F28062, F28063, F28064, F28065, F28066, F28067, F28068, F28069

Applicability to other F28xxx devices, although likely, is not guaranteed. Further, the code and methods presented apply to the development tool versions utilized, specifically:

CCS v5.3.0, Code Generation Tools v6.1.1, DSP/BIOS v5.42.0.07

Be aware that future tool versions may have differences although in all likelihood backwards compatibility will be maintained so that the techniques discussed here should still work. Also note that the operation and setup for SYS/BIOS (BIOS v6) differs from that of DSP/BIOS (BIOS v5). This application report only applies to DSP/BIOS.

Finally, this application report does not provide a tutorial on writing and building code for the F28xxx. It is assumed that the reader already has at least the main framework of their application running from RAM. This report only identifies the special items that must be considered when moving the application into on-chip flash memory.

2 Creating a User Linker Command File

2.1 Non-DSP/BIOS Projects

In non-DSP/BIOS applications, the user linker command file will be where most memory is defined, and where the linking of most sections is specified. The format of this file is no different than the linker command file you are currently using to run your application from RAM. The difference will be in where you link the sections (to be discussed in Section 3). More information on linker command files can be found in reference [11]. The non-DSP/BIOS code projects that accompany this application report contain linker command files that can be used for reference.

The peripheral header files (see references [25 - 32]) contain linker command files named

DSP281x_Headers_nonBIOS.cmd	F2802x_Headers_nonBIOS.cmd
DSP280x_Headers_nonBIOS.cmd	DSP2803x_Headers_nonBIOS.cmd
DSP2804x_Headers_nonBIOS.cmd	F2805x_Headers_nonBIOS.cmd
DSP2833x_Headers_nonBIOS.cmd	F2806x_Headers_nonBIOS.cmd

These files contain linker MEMORY and SECTIONS declarations for linking the peripheral register structures. Since CCS supports having more than one linker command file in a project, all one needs to do is add the appropriate one of these linker command files to your code project in addition to your user linker command file.

In general, the order of the linker command files is unimportant since during a project build, CCS evaluates the MEMORY section of every linker command file before evaluating the SECTIONS section of any linker command file. This ensures that all memories are defined before linking any sections to those memories. However, advanced users may need manual control over the order of linker command file evaluation in some rare situations. This can be specified within CCS v5 on the Project → Properties menu, then select the CCS Build category, Link Order tab.

2.2 DSP/BIOS Projects

The DSP/BIOS configuration tool generates a linker command file that specifies how to link all DSP/BIOS generated sections, and by default all C-compiler generated sections. When running your application from RAM, this linker command file may be the only one in use. However, when executing from flash memory, there will likely be a need to generate and link one or more user defined sections. In particular, any code that configures the on-chip flash control registers (e.g. flash wait-states) cannot execute from flash. In addition, one may want to run certain time critical functions from RAM (instead of flash) to maximize performance. A user linker command file must be created to handle these user defined sections.

Beyond the user and DSP/BIOS generated linker command files, the peripheral header files (see references [25 - 32]) contain linker command files named

DSP281x_Headers_BIOS.cmd	F2802x_Headers_BIOS.cmd
DSP280x_Headers_BIOS.cmd	DSP2803x_Headers_BIOS.cmd
DSP2804x_Headers_BIOS.cmd	F2805x_Headers_BIOS.cmd
DSP2833x_Headers_BIOS.cmd	F2806x_Headers_BIOS.cmd

These file contains linker MEMORY and SECTIONS declarations for linking the peripheral register structures. Since CCS supports having more than one linker command file in a project, all one needs to do is add all three linker command files to their project.

In general, the order of the linker command files is unimportant since during a project build, CCS evaluates the MEMORY section of every linker command file before evaluating the SECTIONS section of any linker command file. This ensures that all memories are defined before linking any sections to those memories. However, advanced users may need manual control over the order of linker command file evaluation in some rare situations (for example, to preempt and override DSP/BIOS linkage of a section). This can be specified within CCS v5 on the Project → Properties menu, then select the CCS Build category, Link Order tab.

3 Where to Link the Sections

Two basic section types exist: initialized, and uninitialized. Initialized sections must contain valid values at device power-up. For example, code and constants are found in initialized sections. When designing a stand-alone embedded system with the F28xxx DSP (e.g., no emulator or debugger in use, no host processor present to perform bootloading), all initialized sections must be linked to non-volatile memory (e.g., on-chip flash). An uninitialized section does not contain valid values at device power-up. For example, variables are found in uninitialized sections. Code will write values to the variable locations during code execution. Therefore, uninitialized sections must be linked to volatile memory (e.g., RAM).

It is suggested that the -w linker option be invoked (it is selected by default for all newly created CCS projects). The -w option will produce a warning if the linker encounters any sections in your project that have not been explicitly specified for linking in a linker command file. When the linker encounters an unspecified section, it uses a default allocation algorithm to link the section into memory (it will link the section to the first defined memory with enough available free space). This is almost always risky, and can lead to unreliable and unpredictable code behavior. The -w option will identify any unspecified sections (e.g., those accidentally forgotten by the user) so that the user can make the necessary addition to the appropriate linker command file. In CCS v5, the -w option checkbox is found on the Project → Properties menu, then select the Build → C2000 Linker → Diagnostics category.

CAUTION:

It is important that the large memory model be used with the C-compiler (as opposed to the small memory model). Small memory model requires certain initialized sections to be linked to non-volatile memory in the lower 64Kw of addressable space. However, no flash memory is present in this region on any F28xxx devices, and this will likely be true for future F28xxx devices as well. Therefore, large memory model should be used. For CCS v5 projects, the large memory model checkbox is found on the Project → Properties menu, then select Build → C2000 Compiler → Basic Options category. It is selected by default for all newly created CCS projects.

For non-DSP/BIOS projects, one should include the large memory model C-compiler runtime support library into their code project. For the fixed-point devices, this is library *rts2800_ml.lib* (as opposed to *rts2800.lib*, which is for the small memory model). For the floating-point devices, this is file *rts2800_fpu32.lib* for plain C code, or rts2800_fpu32_eh.lib for C++ code (there are no small memory model libraries for the floating-point devices). In CCS v5, there is an "Automatic" setting for the library that can be used if desired to have CCS select the correct library for you based on your project settings (e.g., floating point support and memory model selections).

For DSP/BIOS projects, DSP/BIOS will take care of including the required library. The user should not include any runtime support library in a DSP/BIOS project.

3.1 Non-DSP/BIOS Projects

The compiler uses a number of specific sections. These sections are the same whether you are running from RAM or flash. However, when running a program from flash, all initialized sections must be linked to non-volatile memory, whereas all uninitialized sections must be linked to volatile memory. Table 1 shows where to link each compiler generated section on the F28xxx DSP. Information on the function of each section can be found in reference [12]. Any user created initialized section should be linked to flash (e.g., those sections created using the CODE_SECTION compiler pragma), whereas any user created uninitialized sections should be linked to RAM (e.g., those sections created using the DATA_SECTION compiler pragma).

Table 1. Section Linking for Non-DSP/BIOS Projects (Large memory model)

Section Name	Where to Link
.cinit	Flash
.cio	RAM
.const	Flash
.econst	Flash
.pinit	Flash
.switch	Flash
.text	Flash
.bss	RAM
.ebss	RAM
.stack	Lower 64Kw RAM
.sysmem	RAM
.esysmem	RAM
.reset	RAM[1]

Table 1 Notes:

[1] The *.reset* section contains nothing more than a 32-bit interrupt vector that points to the C-compiler boot function in the runtime support library (the _c_int00 routine). It is almost never used. Instead, the user typically creates their own branch instruction to point to the starting point of the code (see Sections 6 and 0). When not in use, the *.reset* section should be omitted from the code build by using a DSECT modifier in the linker command file. For example:

```
/*************************************************************************
* User linker command file
*************************************************************************/

SECTIONS
{
      .reset       : > FLASH,     PAGE = 0, TYPE = DSECT
}
```

3.2 DSP/BIOS Projects

The memory section manager in the DSP/BIOS configuration tool allows one to specify where to link the various DSP/BIOS and C-compiler generated sections. Table 2 indicates where the sections shown on each tab of the memory section manager should be linked (i.e., RAM or FLASH). Note that this information has been tabulated specifically for the DSP/BIOS version used in this report (see Section 1). Later versions of DSP/BIOS, although quite likely to be the same, may have some differences. The reader should check the version they are using and simply be aware of potential differences while proceeding. In CCS v5, the DSP/BIOS version is tied to each individual project. Go to the Project → Properties menu, then select the General category.

Table 2. Section Linking for DSP/BIOS Projects (Large Memory Model)

Memory Section Manager TAB	Section Name	Where to Link
General	Segment for DSP/BIOS Objects	RAM
	Segment for malloc()/free()	RAM
BIOS Data	Argument Buffer Section (.args)	RAM
	Stack Section (.stack)	Lower 64Kw RAM
	DSP/BIOS Init Tables (.gblinit)	Flash
	TRC Initial Values (.trcdata)	RAM[1]
	DSP/BIOS Kernel State (.sysdata)	RAM
	DSP/BIOS Conf Sections (*.obj)	RAM
BIOS Code	BIOS Code Section (.bios)	Flash
	Startup Code Section (.sysinit)	Flash
	Function Stub Memory (.hwi)	Flash
	Interrupt Service Table Memory (.hwi_vec)	PIEVECT RAM[2]
	RTDX Text Segment (.rtdx_text)	Flash
Compiler Sections	Text Section (.text)	Flash
	Switch Jump Tables (.switch)	Flash
	C Variables Section (.bss)	RAM
	C Variables Section (.ebss)	RAM
	Data Initialization Section (.cinit)	Flash
	C Function Initialization Table (.pinit)	Flash
	Constant Section (.econst)	Flash

Compiler Sections (continued)	Constant Section (.const)	Flash
	Data Section (.data)	Flash
	Data Section (.cio)	RAM
Load Address	Load Address - BIOS Code Section (.bios)	Flash[3]
	Load Address - Startup Code Section (.sysinit)	Flash[3]
	Load Address - DSP/BIOS Init Tables (.gblinit)	Flash[3]
	Load Address - TRC Initial Value (.trcdata)	Flash[1]
	Load Address - Text Section (.text)	Flash[3]
	Load Address - Switch Jump Tables (.switch)	Flash[3]
	Load Address - Data Initialization Section (.cinit)	Flash[3]
	Load Address - C Function Initialization Table (.pinit)	Flash[3]
	Load Address - Constant Section (.econst)	Flash[3]
	Load Address - Constant Section (.const)	Flash[3]
	Load Address - Data Section (.data)	Flash[3]
	Load Address - Function Stub Memory (.hwi)	Flash[3]
	Load Address - Interrupt Service Table Memory (.hwi_vec)	Flash[2]
	Load Address - RTDX Text Segment (.rtdx_text)	Flash[3]

Table 2 Notes:

[1] The *.trcdata* section must be copied by the user from its load address (specified on the Load_Address tab) to its run address (specified on the BIOS_Data tab) at runtime. See Section 4.3 for details on performing this copy.

[2] The PIEVECT RAM is a specific block of RAM associated with the Peripheral Interrupt Expansion (PIE) peripheral. On F28xxx devices covered in this report, the PIE RAM is a 256x16 block starting at address 0x000D00 in data space. For other devices, confirm the address in the device datasheet. The memory section manager in the DSP/BIOS configuration tool should already have a pre-defined memory named PIEVECT. The *.hwi_vec* section must be copied by the user from its load address (specified on the memory section manager Load_Address Tab) to its run address (specified on the memory section manager BIOS_Code Tab) at runtime. See Section 4.2 for details on performing this copy.

[3] The specific flash memory selected as the load address for this section should be the same flash memory selected previously as the run address for the section (e.g., on the BIOS Data, BIOS Code, or Compiler Sections tab).

4 Copying Sections from Flash to RAM

4.1 Copying the Interrupt Vectors (non-DSP/BIOS projects only)

The Peripheral Interrupt Expansion (PIE) module manages interrupt requests on F28xxx devices. At power-up, all interrupt vectors must be located in non-volatile memory (i.e., flash), but copied to the PIEVECT RAM as part of the device initialization procedure in your code. The PIEVECT RAM is a specific block of RAM, which on F28xxx devices covered in this report is a 256x16 block starting at address 0x000D00 in data space.

Several approaches exist for linking the interrupt vectors to flash and then copying them to the PIEVECT RAM at runtime. One approach is to create a constant C-structure of function pointers that contains all 128 32-bit vectors. If using the peripheral header file structures (see references [25-32]), such a structure, called *PieVectTableInit*, has already been created in the corresponding file *DSP28xxx_PieVect.c*. Since this structure is declared using the const type qualifier, it will be placed in the *.econst* section by the compiler. One simply needs to copy this structure to the PIEVECT RAM at runtime. The C-compiler runtime support library contains a memory copy function called *memcpy()* that can be used to perform the copy task as follows:

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

/*********************************************************************
* NOTE: This function assumes use of the Peripheral Header File
* structures (see References [25 - 32]).
*********************************************************************/

#include <string.h>

void main(void)
{
/*** Initialize the PIE_RAM ***/
      PieCtrlRegs.PIECTRL.bit.ENPIE = 0;  // Disable the PIE
      asm(" EALLOW");                     // Enable EALLOW protected register access
      memcpy((void *)0x000D00, &PieVectTableInit, 256);
      asm(" EDIS");                       // Disable EALLOW protected register access
}
```

Note that the copy length is 256 because *memcpy()* copies 16-bit words.

The above example uses a hard coded address for the start of the PIE RAM, specifically 0x000D00. If this is objectionable (as hard coded addresses are not good programming practice), one can use a DATA_SECTION pragma to create an uninitialized dummy variable, and link this variable to the PIE RAM. The name of the dummy variable can then be used in place of the hard coded address. For example, when using any of the ‘28xxx device peripheral structures, an uninitialized structure called *PieVectTable* is created and linked over the PIEVECT RAM. The *memcpy()* instruction in the previous example can be replaced by:

```
memcpy(&PieVectTable, &PieVectTableInit, 256);
```

Lastly, on some devices, specifically the 'Piccolo' devices (F2802x, F2803x, F2805x, F2806x), the first three 32-bit PIE vector locations are used for bootmode selection when the debugger is in use. Therefore, the code should be modified to avoid overwriting these locations as follows:

```
memcpy((void *)0x000D06, (Uint16 *)&PieVectTableInit+6, 256-6);
```

or

```
memcpy((Uint16 *)&PieVectTable+6, (Uint16 *)&PieVectTableInit+6, 256-6);
```

4.2 Copying the .hwi_vec Section (DSP/BIOS projects only)

The DSP/BIOS *.hwi_vec* section contains the interrupt vectors, and must be loaded to flash but run from RAM. The user is responsible for copying this section from its load address to its run address. This is typically done in *main()*. The DSP/BIOS configuration tool generates global symbols that can be accessed by code in order to determine the load address, run address, and length of the *.hwi_vec* section. These symbol names are:

hwi_vec_loadstart	hwi_vec_loadend
hwi_vec_loadsize	hwi_vec_runstart

Each symbol should be self-explanatory from its name. Note that the symbols are not pointers, but rather symbolically reference the 16-bit data value found at the corresponding location (i.e., start or end) of the section. The C-compiler runtime support library contains a memory copy function called *memcpy()* that can be used to perform the copy task. A C-code example of how to use this function to perform the section copy follows.

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

#include <string.h>

extern unsigned int hwi_vec_loadstart;
extern unsigned int hwi_vec_loadsize;
extern unsigned int hwi_vec_runstart;

void main(void)
{
/*** Initialize the .hwi_vec section ***/
      asm(" EALLOW");                  /* Enable EALLOW protected register access */
      memcpy(&hwi_vec_runstart, &hwi_vec_loadstart, (Uint32)&hwi_vec_loadsize);
      asm(" EDIS");                    /* Disable EALLOW protected register access */
}
```

Lastly, on some devices, specifically the 'Piccolo' devices (F2802x, F2803x, F2805x, F2806x), the first three 32-bit PIE vector locations are used for bootmode selection when the debugger is in use. Therefore, the code should be modified to avoid overwriting these locations as follows:

```
memcpy(&hwi_vec_runstart+6, &hwi_vec_loadstart+6,
                            (Uint32)(&hwi_vec_loadsize-(Uint16*)6));
```

4.3 Copying the .trcdata Section (DSP/BIOS projects only)

The DSP/BIOS *.trcdata* sections must be loaded to flash, but run from RAM. The user is responsible for copying this section from its load address to its run address. However, unlike the *.hwi_vec* section, the copying of *.trcdata* must be performed prior to *main()*. This is because DSP/BIOS modifies the contents of *.trcdata* during DSP/BIOS initialization (which also occurs prior to *main()*).

The DSP/BIOS configuration tool provides for a user initialization function which can be utilized to perform the *.trcdata* section copy prior to both main() and DSP/BIOS initialization. This can be found in the CCS v5 project configuration file under System → Global Settings Properties, as shown in Figure 1.

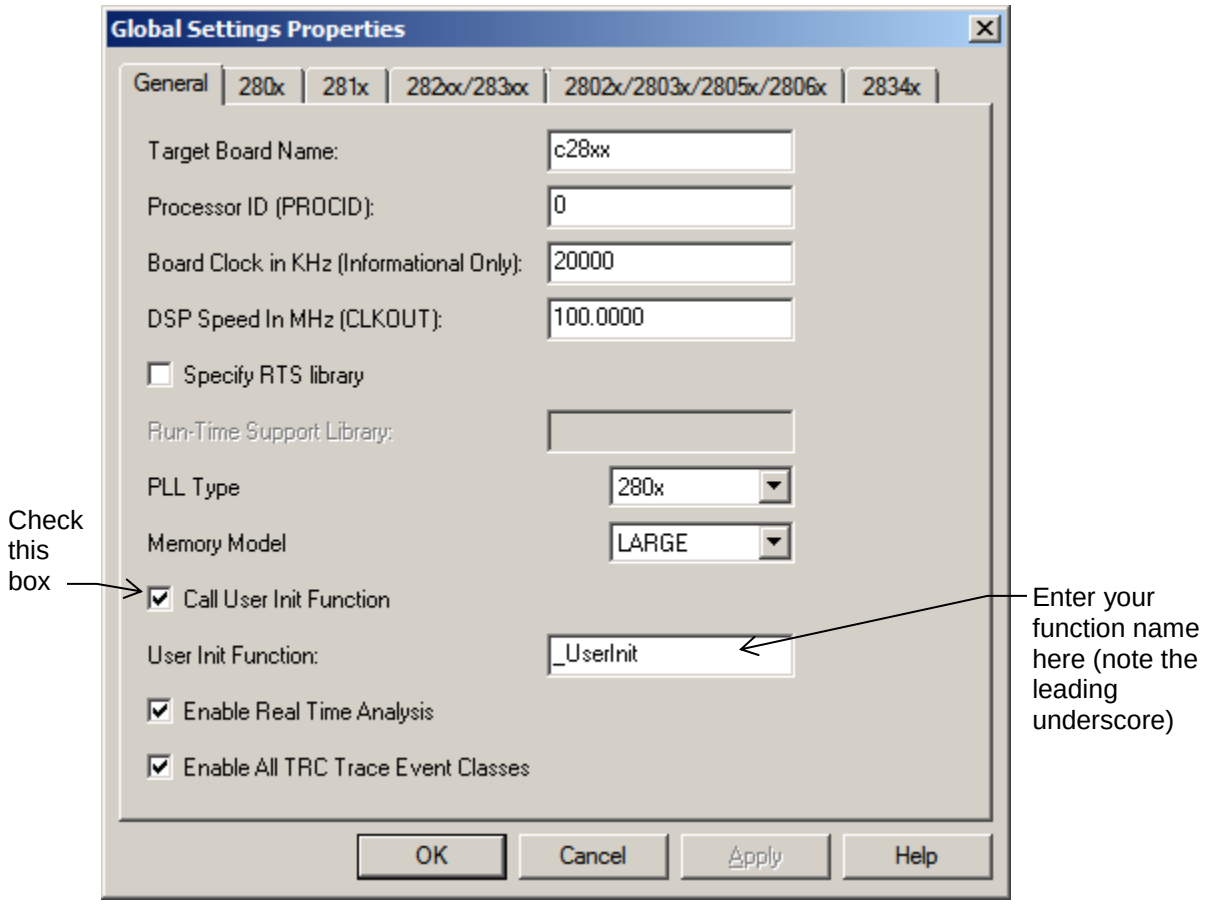

Figure 1. Specifying the User Init Function in the DSP/BIOS Configuration tool

What remains is to create the user initialization function. The DSP/BIOS configuration tool generates global symbols that can be accessed by code in order to determine the load address, run address, and length of each section. These symbol names are:

trcdata_loadstart
trcdata_loadsize
trcdata_loadend
trcdata_runstart

Each symbol should be self-explanatory from its name. Note that the symbols are not pointers, but rather symbolically reference the 16-bit data value found at the corresponding location (i.e., start or end) of the section. The C-compiler runtime support library contains a memory copy function called *memcpy()* that can be used to perform the copy task. A C-code example of a user init function that performs the *.trcdata* section copy follows.

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

#include <string.h>

extern unsigned int trcdata_loadstart;
extern unsigned int trcdata_loadsize;
extern unsigned int trcdata_runstart;

void UserInit(void)
{
/*** Initialize the .trcdata section before main() ***/
      memcpy(&trcdata_runstart, &trcdata_loadstart, (Uint32)&trcdata_loadsize);
}
```

4.4 Initializing the Flash Control Registers (DSP/BIOS and non-DSP/BIOS projects)

The initialization code for the flash control registers cannot be executed from the flash memory or unpredictable results may occur. Therefore, the initialization function for the flash control registers must be copied from flash (its load address) to RAM (its run address) at runtime.

CAUTION:

The flash control registers are protected by either the Code Security Module (CSM) or the Dual Code Security Module (DCSM), depending on the device.

On devices with the CSM, if the CSM is secured you must run the flash register initialization code from CSM secure RAM or the initialization code will be unable to access the flash registers (see the memory map section of the device data sheet to identify the secure memories). Note that the CSM is always locked at device reset, although the ROM bootloader will unlock it if you are using dummy passwords of 0xFFFF.

On devices with the DCSM, the protection of the flash control registers is controlled by the SEM field of the FLSEM register. At reset, the SEM default is such that the flash control registers can be written to by code running from any memory. Normally, software will configure the flash control registers prior to changing the SEM field, and therefore the user need only copy the configuration code to any RAM and execute it from there. However, if software first changes the SEM field, the flash configuration code will need to be executed from a RAM block that has security access to the flash configuration registers. This access is determined by the SEM field value. See reference [18] for information on the SEM field of the FLSEM register.

The CODE_SECTION pragma of the C compiler can be used to create a separately linkable section for the flash initialization function. For example, suppose the flash register configuration is to be performed in the C function *InitFlash()*, and it is desired to place this function into a linkable section called *secureRamFuncs*. The following C-code example shows proper use of the CODE_SECTION pragma along with an example configuration of the flash registers:

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

/*********************************************************************
* NOTE: The InitFlash() function shown here is just an example of an
* initialization for the flash control registers.  Consult the device
* datasheet for production wait state values and any other relevant
* information.  Wait-states shown here are specific to current F280x
* devices operating at 100 MHz.
* NOTE: This function assumes use of the Peripheral Header File
* structures (see References [25 - 32]).
*********************************************************************/

#pragma CODE_SECTION(InitFlash, "secureRamFuncs")
void InitFlash(void)
{
      asm(" EALLOW");                         // Enable EALLOW protected register access
      FlashRegs.FPWR.bit.PWR = 3;                     // Flash set to active mode
      FlashRegs.FSTATUS.bit.V3STAT = 1;               // Clear the 3VSTAT bit
      FlashRegs.FSTDBYWAIT.bit.STDBYWAIT = 0x01FF;    // Sleep to standby cycles
      FlashRegs.FACTIVEWAIT.bit.ACTIVEWAIT = 0x01FF; // Standby to active cycles
      FlashRegs.FBANKWAIT.bit.RANDWAIT = 3;           // F280x Random access wait states
      FlashRegs.FBANKWAIT.bit.PAGEWAIT = 3;           // F280x Paged access wait states
      FlashRegs.FOTPWAIT.bit.OTPWAIT = 5;             // F280x OTP wait states
      FlashRegs.FOPT.bit.ENPIPE = 1;                  // Enable the flash pipeline
      asm(" EDIS");                          // Disable EALLOW protected register access

/*** Force a complete pipeline flush to ensure that the write to the last register
     configured occurs before returning.  Safest thing is to wait 8 full cycles.  ***/

      asm(" RPT #6 || NOP");               // Takes 8 cycles to execute

} //end of InitFlash()
```

The section secureRamFuncs can then be linked using the user linker command file. This section will require separate load and run addresses. Further, it is desired to have the linker generate some global symbols that can be used to determine the load address, run address, and length of the section. This information is needed to perform the copy from the sections load address to its run address. The user linker command file would appear as follows:

```
/*********************************************************************
* User linker command file
*********************************************************************/

SECTIONS
{
/*** User Defined Sections ***/
secureRamFuncs:    LOAD = FLASH,       PAGE = 0
                   RUN = SECURE_RAM,   PAGE = 0
                   LOAD_START(_secureRamFuncs_loadstart),
                   LOAD_SIZE(_secureRamFuncs_loadsize),
                   RUN_START(_secureRamFuncs_runstart)
}
```

In this example, the memories FLASH and SECURE_RAM are assumed to have been defined either in the MEMORY section of the user linker command file (for non-DSP/BIOS projects) or in the memory section manager of the DSP/BIOS configuration tool (for DSP/BIOS projects). The PAGE designation for these memories should match that of the memory definition. The above example assumes both memories have been declared on PAGE 0 (program memory space). The LOAD_START, LOAD_SIZE, and RUN_START directives will generate global symbols with the specified names for the corresponding addresses. Note the use of the leading underscore on the global symbol definitions (e.g., *_secureRamFuncs_runstart*)

Finally, the section must be copied from flash to RAM at runtime. As in Sections 4.1 - 4.3, the function *memcpy()* from the compiler runtime support library can be used.

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

#include <string.h>

extern unsigned int secureRamFuncs_loadstart;
extern unsigned int secureRamFuncs_loadsize;
extern unsigned int secureRamFuncs_runstart;

void main(void)
{
/* Copy the secureRamFuncs section */
      memcpy(&secureRamFuncs_runstart,
             &secureRamFuncs_loadstart,
             (Uint32)&secureRamFuncs_loadsize);

/* Initialize the on-chip flash registers */
      InitFlash();
}
```

4.5 Maximizing Performance by Executing Time-critical Functions from RAM

(DSP/BIOS and non-DSP/BIOS projects)

The on-chip RAM memory of the F28xxx devices covered in this report provides code execution in MIPS (millions of instructions per second) that is equal to the operating clock frequency of the device in MHz (e.g., 150 MIPS at 150 MHz, 100 MIPS at 100 MHz, etc.). However, the on-chip flash memory gives effective code execution performance that is somewhat less, depending on the device frequency and the application. Rough flash execution performance estimates for the devices covered in this report are[1]:

90 - 95 MIPS at 150 MHz
80 - 85 MIPS at 100 MHz
65 - 70 MIPS at 80 MHz
50 - 55 MIPS at 60 MHz
37 - 39 MIPS at 40 MHz

It may therefore be desirable to run certain time-critical or computationally demanding routines from on-chip RAM, especially for devices running at 150 MHz. In a standalone embedded system however, all code must initially reside in non-volatile memory. Separate load and run addresses must be setup for those functions running from RAM, and a copy must be performed to move them from on-chip flash to the RAM at runtime. To do this, apply the same procedure previously described in Section 4.4.

Using the CODE_SECTION pragma, one can add multiple functions to the same linkable section. The entire section can then be assigned to run from a particular RAM block, and the user can copy the entire section to RAM all at once, as discussed in Section 4.4. If finer linking granularity is required, separate section names can be created for each function.

4.6 Maximizing Performance by Linking Critical Global Constants to RAM

(DSP/BIOS and non-DSP/BIOS projects)

Constants are those data structures declared using the C language *const* type modifier. The compiler places all constants in the *.econst* section (large memory model assumed). While special pipelining on the F28xxx devices covered by this report accelerates effective flash performance for code execution, no such pipelining exists for accessing data constants located in the on-chip FLASH. Each flash data access can take multiple cycles. Typical flash wait-states will be 5 cycles at 150 MHz, 3 cycles at 100 or 90 MHz, 2 cycles at 80 or 60 MHz, and 1 cycle at or below 50 MHz device[1].

CAUTION:
Flash timings vary among the different F28xxx devices. It is important to check the specific datasheet for the device in use.

[1] These estimates apply only to the devices covered by this report (see Section 1), which have all been fabricated in the same 180 nm CMOS process. The newest F28xxx devices (e.g. F28M35x, F28M36x) are fabricated in a 65 nm process and also have a wider flash pre-fetch buffer. They have significantly higher effective flash execution performance than the 180 nm devices.

It may therefore be desirable to keep heavily accessed constants and constant tables in on-chip RAM. However, a standalone embedded system requires that all initialized data (e.g., constants) initially reside in non-volatile memory. Therefore, separate load and run addresses must be setup for those constants you wish to access in RAM, and a copy must be performed to move them from the on-chip flash to the RAM at runtime. Two different approaches for accomplishing this will be presented.

4.6.1 Method 1: Running All Constant Arrays from RAM

This approach involves specifying separate load and run addresses for the entire *.econst* section. The advantage of this approach is ease of use, while the disadvantage is excessive RAM usage (there may be only a few constants that require high-speed access, but with this method all constants are relocated into RAM).

4.6.1.1 Non-DSP/BIOS Projects

The same approach discussed in Section 4.4 can be used. Simply specify separate load and run address for the *.econst* section in the user linker command file, and then add code to your project to copy the entire *.econst* section to RAM at runtime. For example:

```
/*************************************************************************
* User linker command file
*************************************************************************/

SECTIONS
{
.econst:           LOAD = FLASH,  PAGE = 0
                   RUN = RAM,     PAGE = 1
                   LOAD_START(_econst_loadstart),
                   LOAD_SIZE(_econst_loadsize),
                   RUN_START(_econst_runstart)
}
```

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

#include <string.h>

extern unsigned int econst_loadstart;
extern unsigned int econst_loadsize;
extern unsigned int econst_runstart;

void main(void)
{
/* Copy the .econst section */
      memcpy(&econst_runstart, &econst_loadstart, (Uint32)&econst_loadsize);
}
```

4.6.1.2 DSP/BIOS Projects

Although the DSP/BIOS configuration tool allows the specification of different load and run addresses for the *.econst* section, it will not generate the code accessible labels that are needed to perform the memory copy. Therefore, the user must preemptively link the *.econst* section in the user linker command file before the DSP/BIOS generated linker command file is evaluated. The user linker command file would appear as follows:

```
/*************************************************************
* User linker command file (DSP/BIOS Projects)
*************************************************************/

SECTIONS
{
/*** Preemptively link the .econst section ***/
/* Must come before DSP/BIOS linker command file is evaluated */

.econst:            LOAD = FLASH,  PAGE = 0
                    RUN = RAM,     PAGE = 1
                    LOAD_START(_econst_loadstart),
                    LOAD_SIZE(_econst_loadsize),
                    RUN_START(_econst_runstart)
}
```

To guarantee that the user linker command file is evaluated before the DSP/BIOS generated linker command file during the project build, one must specify the link order in CCS. To do this in CCS v5, go to Project → Properties, then select the Build category, Link Order tab. You can then specify the appropriate order for the linker command files in question by clicking on the 'Add…' button. Note that the DSP/BIOS generated linker command file will not be explicitly listed in the file selection list. Rather, you should select "$(GEN_CMDS_QUOTED)" which means the DSP/BIOS generated .cmd file. Figure 2 shows an example of this, where *F2808_BIOS_flash.cmd* is the user linker command file and "$(GEN_CMDS_QUOTED)" refers to the *F2808_example_BIOS_flashcfg.cmd* file generated by DSP/BIOS.

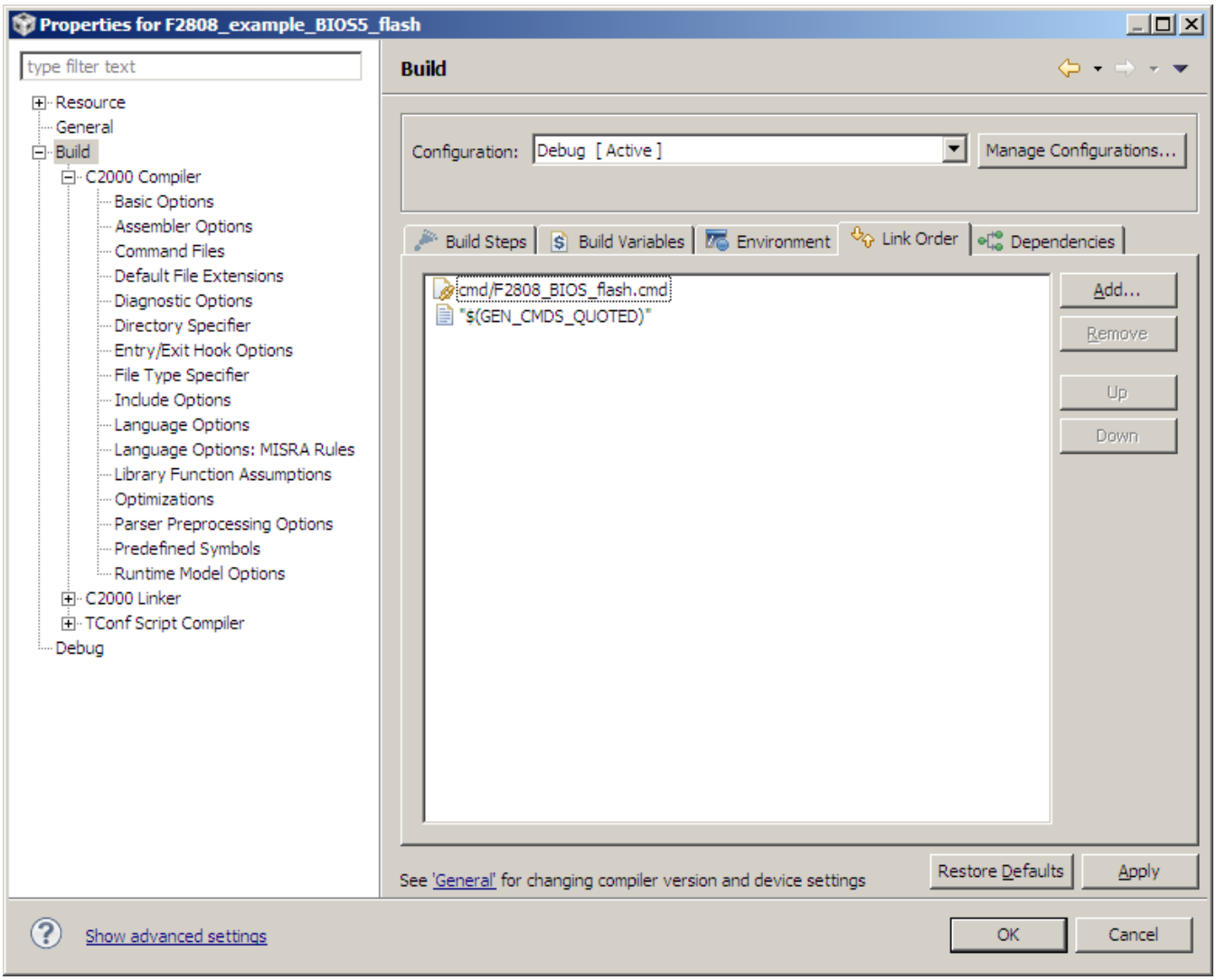

Figure 2. Specifying the Link Order In Code Composer Studio v5

The *.econst* section can then be copied from its load address to its run address as follows:

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

#include <string.h>

extern unsigned int econst_loadstart;
extern unsigned int econst_loadsize;
extern unsigned int econst_runstart;

void main(void)
{
/* Copy the .econst section */
     memcpy(&econst_runstart, &econst_loadstart, (Uint32)&econst_loadsize);
}
```

4.6.2 Method 2: Running a Specific Constant Array from RAM

(DSP/BIOS and non-DSP/BIOS projects)

This method involves selectively copying constants from flash to RAM at runtime. The procedure to accomplish this is similar to that of Method 1, except that only selected constants are placed in a named section and copied to RAM (rather than copying all constants to RAM).

Suppose for example that one wants to create a 5 word constant array called *table[ ]* to be run from RAM. A DATA_SECTION pragma is used to place *table[ ]* in a user defined section called *ramconsts*. The C-source file would appear as follows:

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

#pragma DATA_SECTION(table, "ramconsts")
const int table[5] = {1,2,3,4,5};

void main(void)
{

}
```

The section *ramconsts* is linked to load to flash but run from RAM using the user linker command file, and global symbols are generated to facilitate the memory copy. The user linker command file would appear as follows:

```
/*********************************************************************
* User linker command file
*********************************************************************/

SECTIONS
{
/*** User Defined Sections ***/
ramconsts:          LOAD = FLASH, PAGE = 0
                    RUN = RAM,    PAGE = 1
                    LOAD_START(_ramconsts_loadstart),
                    LOAD_SIZE(_ramconsts_loadsize),
                    RUN_START(_ramconsts_runstart)
}
```

Finally, *table[ ]* must be copied from its load address to its run address at runtime:

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

#include <string.h>

extern unsigned int ramconsts_loadstart;
extern unsigned int ramconsts_loadsize;
extern unsigned int ramconsts_runstart;

void main(void)
{
/* Initialize the ramconsts section */
      memcpy(&ramconsts_runstart, &ramconsts_loadstart, (Uint32)&ramconsts_loadsize);
}
```

5 Programming the Code Security Module Passwords

The code-security module on F28xxx devices provides protection against unwanted copying of and tampering with your software. Devices covered by this report have either a single-zone code security module (CSM), or a dual-zone code security module (DCSM). The CSM uses a single 128-bit password that restricts access to all secure memories. The DCSM provides for two 128-bit passwords, each restricting access to one of the two security zones. Some secure memory resources are fixed in their assignment to one of the two zones, while others can be assigned by software to either zone.

Single-zone Code Security Module Devices:

F281x: F2810, F2811, F2812 **F280x/2801x/28044:** F2801, F2802, F2806, F2808, F2809, F28015, F28016, F28044 **F2823x/2833x:** F28232, F28234, F28235, F28332, F28334, F28335 **F2802x:** F28020, F28021, F28022, F28023, F28026, F28027, F280200 **F2803x:** F28030, F28031, F28032, F28033, F28034, F28035 **F2806x:** F28062, F28063, F28064, F28065, F28066, F28067, F28068, F28069

Dual-zone Code Security Module Devices:

F2805x: F28050, F28051, F28052, F28053, F28054, F28055

The sub-sections that follow explain how to incorporate the code security passwords into your code project for the single-zone and dual-zone security devices. It is well beyond the scope of this report to explain in detail the operation of the security modules. The reader is referred to references [13 - 19] for this information.

5.1 Single-Zone Security Devices (DSP/BIOS and non-DSP/BIOS projects)

On single-zone security devices, the CSM secures the entire flash, the OTP memory, and some of the 'L' SARAM blocks (see the device datasheet for device specific information). The flash configuration registers are secured as well. When locked, only code executing from secure memory can access data (read or write) in other secured memory. Code executing from unsecure memory can branch to (or call) code in secure memory, but cannot access any data in secure memory.

The CSM uses a 128-bit password comprised of 8 individual 16-bit words. For CSM devices covered by this report, these passwords are stored in the upper most 8 words of the flash (e.g., addresses 0x3F7FF8 through 0x3F7FFF on F281x, F280x, F2801x, F28044, F2802x, F2803x, and F2806x devices, and addresses 0x33FFF8 through 0x33FFFF on F2823x and F2833x devices). During development, it is recommended that dummy passwords of 0xFFFF be used. When dummy passwords are used, only dummy reads of the password locations are needed to unsecure the CSM. Placing dummy passwords into the password locations is easy to do since 0xFFFF will be the state of these locations after the flash is erased during flash programming. Users need only avoid linking any sections to the password addresses in their code project, and the passwords will retain the 0xFFFF.

After development, one may want to use real passwords. In addition, to properly lock the CSM module for devices covered in this report, values of 0x0000 must be programmed into the 118 flash addresses beginning 120 words prior to the start of the CSM passwords, e.g., addresses 0x3F7F80 through 0x3F7FF5 on F281x, F280x, F2801x, F28044, F2802x, F2803x, and F2806x devices, and addresses 0x33FF80 through 0x33FFF5 on F2823x and F2833x devices (see references [1 - 6, and 8]). An easy way to accomplish both of these tasks is with a little simple assembly language programming. The following example assembly code file specifies the desired password values and places them in a named initialized section called *passwords*. It also creates a named initialized section called *csm_rsvd* that contains all 0x0000 values and is of proper length to fit in the aforementioned 118 word address ranges. See reference [11] for more information on the assembly language directives used.

```
;***************************************************************************
* File: Passwords.asm
;***************************************************************************

;***************************************************************************
* Dummy passwords of 0xFFFF are shown.  The user can change these to
* desired values.
*
* CAUTION: Do not use 0x0000 for all 8 passwords or the CSM module can
* be permanently locked.  See References [13 - 17, and 19] for more
* information.
;***************************************************************************
     .sect "passwords"
     .int   0xFFFF                      ;PWL0 (LSW of 128-bit password)
     .int   0xFFFF                      ;PWL1
     .int   0xFFFF                      ;PWL2
     .int   0xFFFF                      ;PWL3
     .int   0xFFFF                      ;PWL4
     .int   0xFFFF                      ;PWL5
     .int   0xFFFF                      ;PWL6
     .int   0xFFFF                      ;PWL7 (MSW of 128-bit password)
;---------------------------------------------------------------------------
     .sect "csm_rsvd"
     .loop (3F7FF5h - 3F7F80h + 1)
     .int   0x0000
     .endloop
;---------------------------------------------------------------------------

     .end                               ;end of file passwords.asm
```

Note that this example is showing dummy password values of 0xFFFF. Replace these values with your desired passwords.

CAUTION:

Do not use 0x0000 for all 8 passwords. Doing so can permanently lock the CSM module! See references [13 - 17, and 19] for more information.

The *passwords* and *csm_rsvd* sections should be placed in memory with the user linker command file.

For non-DSP/BIOS projects, the user should define memories named (for example) *PASSWORDS* and *CSM_RSVD* on PAGE 0 in the MEMORY portion of the user linker command file. The sections *passwords* and *csm_rsvd* can then be linked to these memories. The following example applies to F281x, F280x, F2801x, F28044, F2802x, F2803x, and F2806x devices. For other devices, consult the device datasheet to confirm the addresses of the password and CSM reserved locations.

```
/*************************************************************************
* User linker command file (non-DSP/BIOS Projects)
*************************************************************************/

MEMORY
{
PAGE 0:    /* Program Memory */
   CSM_RSVD    : origin = 0x3F7F80, length = 0x000076
   PASSWORDS   : origin = 0x3F7FF8, length = 0x000008
}


SECTIONS
{
/*** Code Security Password Locations ***/
   passwords:     > PASSWORDS,    PAGE = 0
   csm_rsvd:      > CSM_RSVD,     PAGE = 0
}
```

For DSP/BIOS projects, the user should define the memories named (for example) *PASSWORDS* and *CSM_RSVD* using the memory section manager of the DSP/BIOS configuration tool. The two figures that follow show the DSP/BIOS memory section manager properties for these memories on F281x, F280x, F2801x, F28044, F2802x, F2803x, and F2806x devices. As always, consult the device datasheet to confirm the correct addresses and lengths, or for memory map information of other F28xx devices.

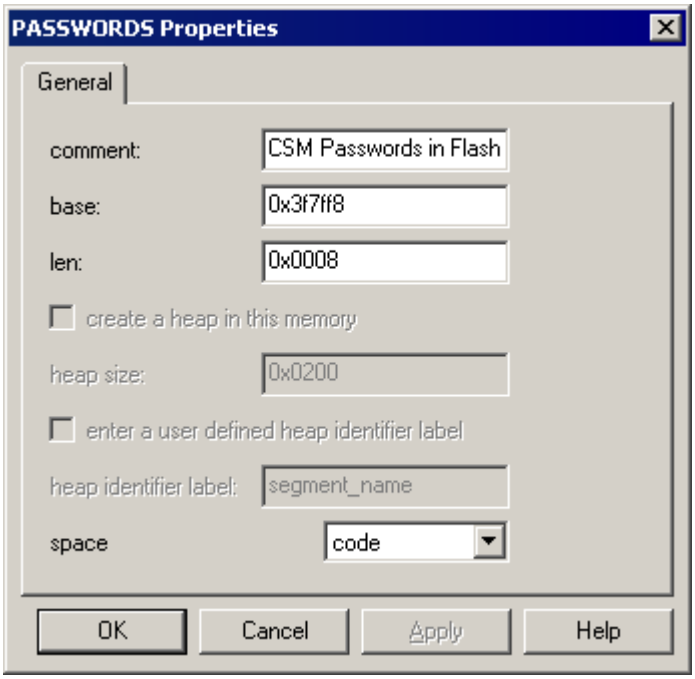

Figure 3. DSP/BIOS MEM Properties for CSM Password Locations

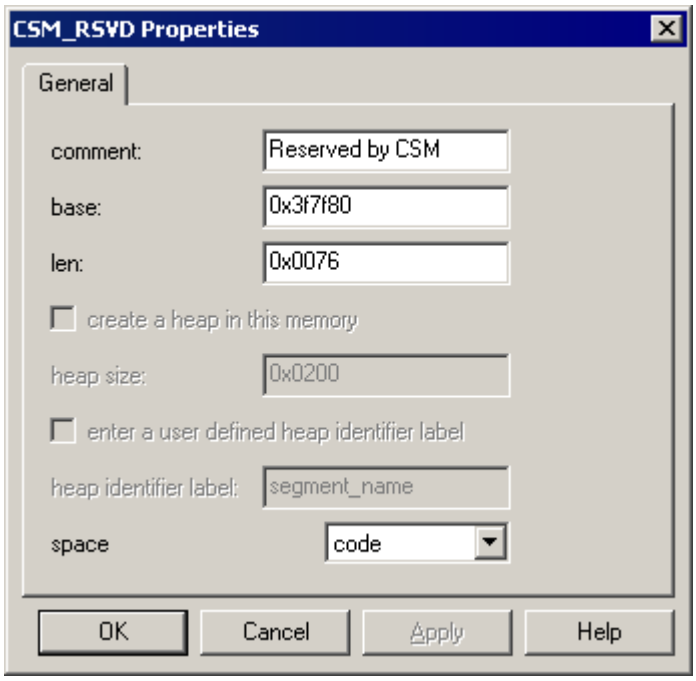

Figure 4. DSP/BIOS MEM Properties for CSM Reserved Locations

The sections *passwords* and *csm_rsvd* can then be linked to these memories in the user linker command file. For DSP/BIOS projects, the user linker command file would appear as:

```
/*************************************************************************
* User linker command file (DSP/BIOS Projects)
*************************************************************************/

SECTIONS
{
/*** Code Security Password Locations ***/
   passwords:     > PASSWORDS,     PAGE = 0
   csm_rsvd:      > CSM_RSVD,      PAGE = 0
}
```

5.2 Dual-Zone Security Devices (DSP/BIOS and non-DSP/BIOS projects)

Dual-zone security operates similar to the single-zone security described in the previous section except for three major differences:

1. Code executing from a secured zone can only access data located either in that zone, or in unsecured memories. Code in one secured zone cannot access data located in the other secured zone, although it can branch to or call code that resides in any memory. Optionally, a memory block can be configured as execute only such that no code is able to access data in that memory (including code executing from that same memory).

2. Each zone uses a 16x16 bit ‘Zone Select Block’ located in OTP memory to control security access to that zone. The zone select block contains the 128-bit security passwords for that zone as well as zone assignment and execute-only designations for securable memories. Figure 5 shows the entries and arrangement of a zone select block.

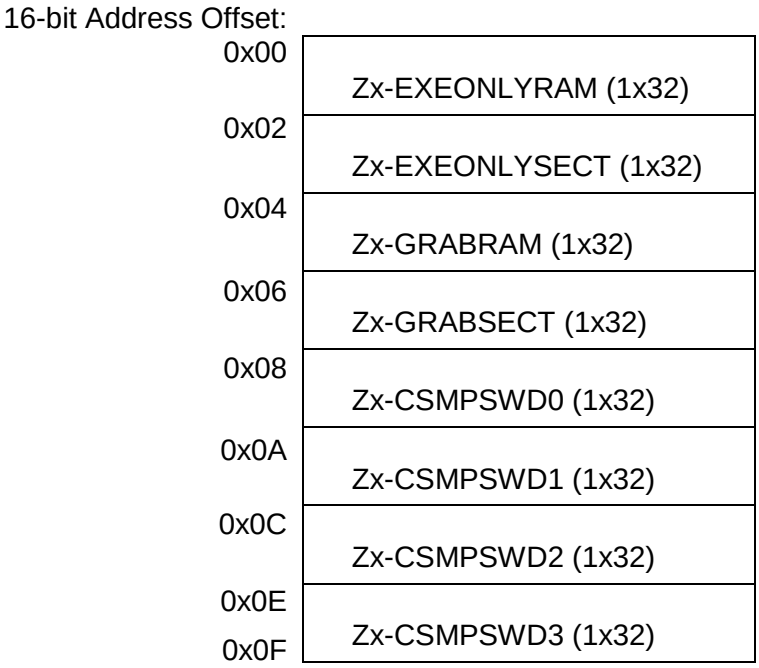

Figure 5. DCSM Zone Select Block

3. Since the security configuration tables are located in OTP memory, each entry in a zone select block can be written only once (to be exact, a 1 bit can be flipped to a 0 bit at any time, but a 0 bit cannot be changed). To provide the flexibility to change security configurations a few times over the life of a product and to aid with development, multiple zone select blocks exist as part of an overall 512x16 bit zone configuration table in OTP memory. The first entry in the configuration table is a link value that specifies which zone select block in the table is active. The link value is updatable by sequentially changing 1 bits to 0 bits, starting with bit position 0. In this way, the zone select block can be ‘Updated’ up to thirty times. Figure 6 shows the entries and arrangement in memory of a zone configuration table. To be clear there are two configuration tables in OTP memory, one for each of the two security zones.

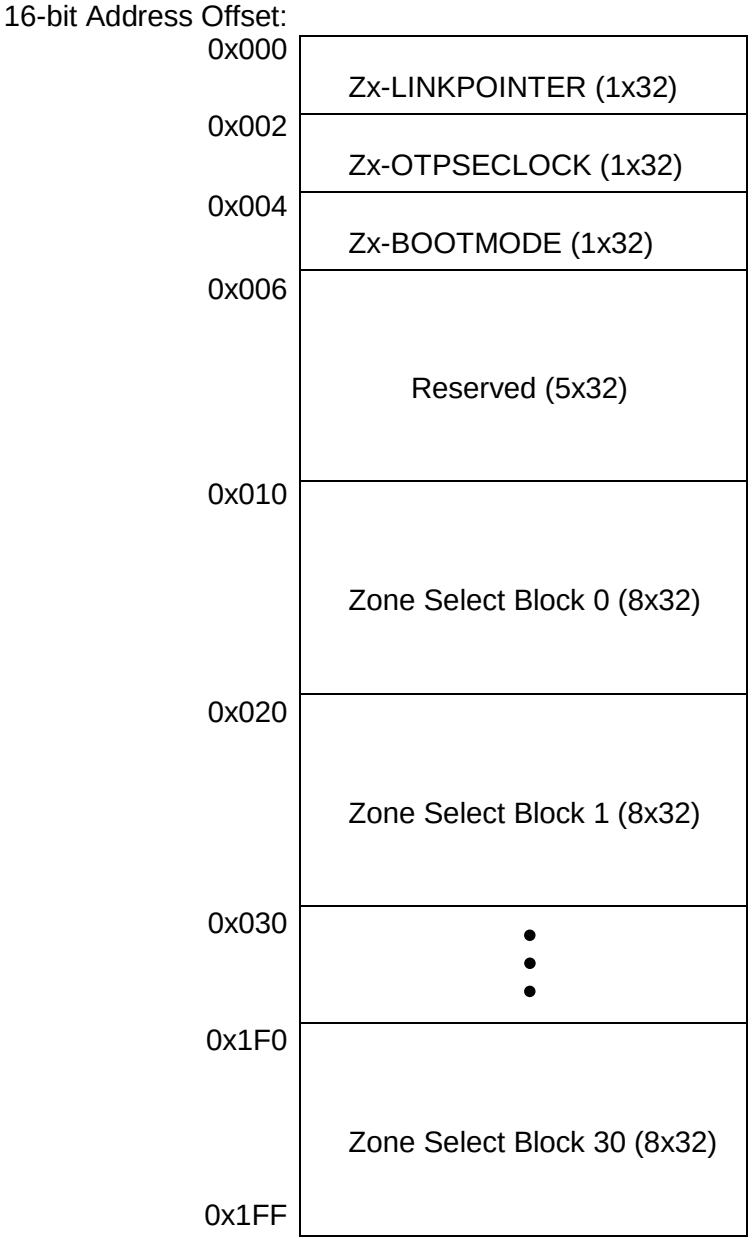

Figure 6. DCSM Security Zone Configuration Table OTP Memory

For a given device, the datasheet should be consulted to determine the base address for each of the two DCSM security configuration tables. For example, on F2805x devices the zone 1 security configuration table starts at address 0x3D7A00, while that of zone 2 starts at address 0x3D7800.

As previously stated, the read should refer to the documentation for their particular device for a fully detailed explanation of DCSM operation and the security configuration tables. The primary intent here is to show how to place the security configuration table in memory.

A simple way to create the zone configuration tables is with a small amount of assembly language coding. For each of the two configuration tables, two data sections are needed (meaning a total of four data sections). The first section is the table base section, and contains the first three entries in the table: Zx-LINKPOINTER, Zx-OTPSECLOCK, and Zx-BOOTMODE. It is always linked to the base address of the configuration table. The second section contains the zone select block that was depicted in Figure 5. It should be linked to memory at an offset from the table base address. This offset is specified by the value of the Zx-LINKPOINTER. See the device documentation for Zx-LINKPOINTER values. It is not a simple 'value equals offset' relationship. In general, users will utilize the Zone Select Blocks in sequence, starting with Zone Select Block 0 at offset 0x010. To change to a new (the next) Zone Select Block, the least-significant 1 bit in the Zx-LINKPOINTER is programmed to a 0, and the new Zone Select Block would be programmed to the next offset address in the configuration table.

The following two example assembly code files create the above described needed data elements for each of the two security zones. The table base sections are named *dcsm_otp_z1* and *dcsm_otp_z2*, while the zone select block sections are named *dcsm_zsel_z1* and *dcsm_zsel_z2*. See reference [11] for more information on the assembly language directives used.

```
;***************************************************************************
; File: Passwords_zone1.asm
;***************************************************************************

;***************************************************************************
; Dummy values of 0xFFFFFFFF are shown for all entries.  The user can
; change these to desired values.
;
; CAUTION: Do not use 0x00000000 for all 4 passwords or the DCSM zone
; can be permanently locked.  See Reference [18] for more information.
;***************************************************************************

     .sect "dcsm_otp_z1"
     .long  0xFFFFFFFF                  ;Z1-LINKPOINTER
     .long  0xFFFFFFFF                  ;Z1-OTPSECLOCK
     .long  0xFFFFFFFF                  ;Z1-BOOTMODE

     .sect "dcsm_zsel_z1"
     .long  0xFFFFFFFF                  ;Z1-EXEONLYRAM
     .long  0xFFFFFFFF                  ;Z1-EXEONLYSECT
     .long  0xFFFFFFFF                  ;Z1-GRABRAM
     .long  0xFFFFFFFF                  ;Z1-GRABSECT
     .long  0xFFFFFFFF                  ;Z1-CSMPSWD0 (LSW of 128-bit password)
     .long  0xFFFFFFFF                  ;Z1-CSMPSWD1
     .long  0xFFFFFFFF                  ;Z1-CSMPSWD2
     .long  0xFFFFFFFF                  ;Z1-CSMPSWD3 (MSW of 128-bit password)

     .end                               ;end of file Passwords_zone1.asm
```

```
;***************************************************************************
* File: Passwords_zone2.asm
;***************************************************************************

;***************************************************************************
* Dummy values of 0xFFFFFFFF are shown for all entries.  The user can
* change these to desired values.
*
* CAUTION: Do not use 0x00000000 for all 4 passwords or the DCSM zone
* can be permanently locked.  See Reference [18] for more information.
;***************************************************************************

     .sect "dcsm_otp_z2"
     .long  0xFFFFFFFF                  ;Z2-LINKPOINTER
     .long  0xFFFFFFFF                  ;Z2-OTPSECLOCK
     .long  0xFFFFFFFF                  ;Z2-BOOTMODE

     .sect "dcsm_zsel_z2"
     .long  0xFFFFFFFF                  ;Z2-EXEONLYRAM
     .long  0xFFFFFFFF                  ;Z2-EXEONLYSECT
     .long  0xFFFFFFFF                  ;Z2-GRABRAM
     .long  0xFFFFFFFF                  ;Z2-GRABSECT
     .long  0xFFFFFFFF                  ;Z2-CSMPSWD0 (LSW of 128-bit password)
     .long  0xFFFFFFFF                  ;Z2-CSMPSWD1
     .long  0xFFFFFFFF                  ;Z2-CSMPSWD2
     .long  0xFFFFFFFF                  ;Z2-CSMPSWD3 (MSW of 128-bit password)

     .end                               ;end of file Passwords_zone2.asm
```

Note that this example is showing dummy values of 0xFFFFFFFF for all configuration values. These will be replaced with your desired values when you are ready to secure the device.

CAUTION:

Do not use 0x00000000 for all 4 passwords in a zone select block. Doing so can permanently lock that zone in the DCSM module! See reference [18] for more information.

The *dcsm_otp_z1*, *dcsm_otp_z2*, *dcsm_zsel_z1*, and *dcsm_zsel_z2* sections now need to be linked to memory.

For non-DSP/BIOS projects, the user should define four memories to hold these sections in the user linker command file. For example, define *DCSM_OTP_Z1_P0* and *DCSM_OTP_Z2_P0* to hold the two table base entry sections, and *DCSM_ZSEL_Z1_P0* and *DCSM_ZSEL_Z2_P0* to hold the two zone select block sections. The sections d*csm_otp_z1*, *dcsm_otp_z2*, *dcsm_zsel_z1*, and *dcsm_zsel_z2* sections can then be linked to these memories as shown in the following example.

```
/*************************************************************************
* User linker command file (non-DSP/BIOS Projects)
*************************************************************************/

MEMORY
{
PAGE 0:    /* Program Memory */
   DCSM_OTP_Z2_P0   : origin = 0x3D7800, length = 0x000006     /* Z2 DCSM Table Base */
   DCSM_ZSEL_Z2_P0  : origin = 0x3D7810, length = 0x000010     /* Z2 Select Block */
   DCSM_OTP_Z1_P0   : origin = 0x3D7A00, length = 0x000006     /* Z1 DCSM Table Base */
   DCSM_ZSEL_Z1_P0  : origin = 0x3D7A10, length = 0x000010     /* Z1 Select Block */

PAGE 1:    /* Data Memory */
}


SECTIONS
{
   dcsm_otp_z1:     > DCSM_OTP_Z1_P0,     PAGE = 0
   dcsm_zsel_z1:    > DCSM_ZSEL_Z1_P0,    PAGE = 0
   dcsm_otp_z2:     > DCSM_OTP_Z2_P0,     PAGE = 0
   dcsm_zsel_z2:    > DCSM_ZSEL_Z2_P0,    PAGE = 0
}
```

For DSP/BIOS projects, the four memories are defined using the memory section manager of the DSP/BIOS configuration tool. The four figures that follow show the DSP/BIOS memory section manager properties for these memories on F2805x devices. As always, consult the device datasheet to confirm the correct addresses and lengths, or for memory map information of other F28xx devices with the DCSM module.

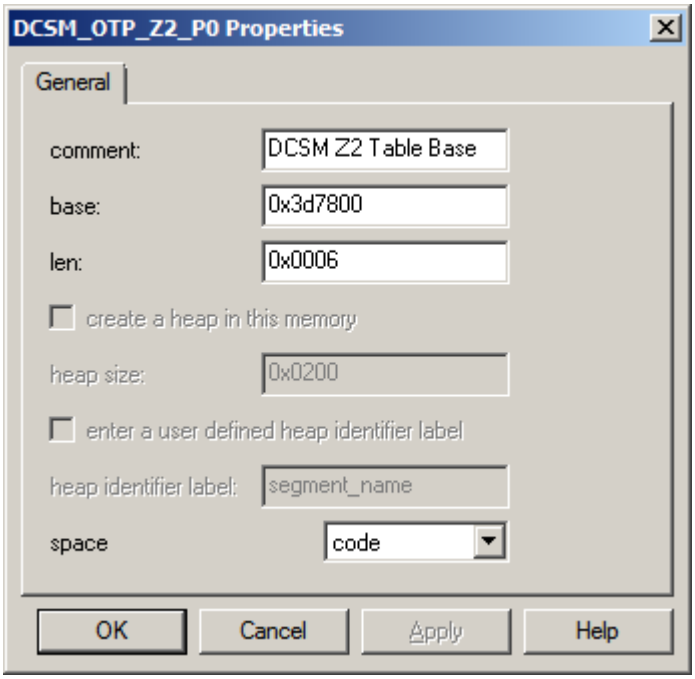

Figure 7. DSP/BIOS MEM Properties for DCSM_OTP_Z2_P0 Memory

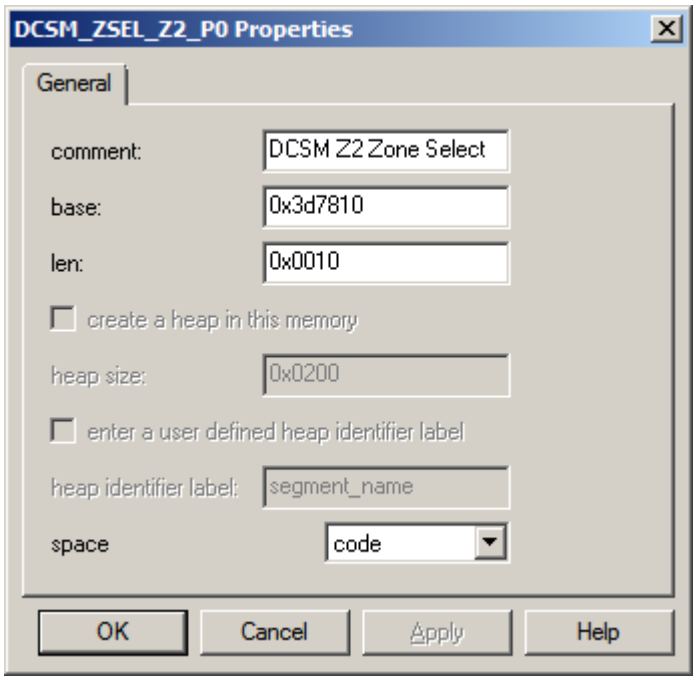

Figure 8. DSP/BIOS MEM Properties for DCSM_ZSEL_Z2 Memory

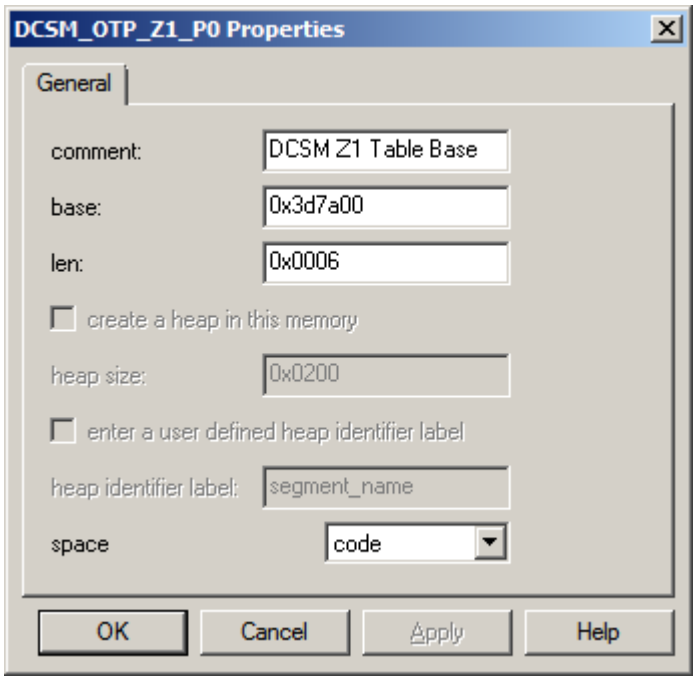

Figure 9. DSP/BIOS MEM Properties for DCSM_OTP_Z1_P0 Memory

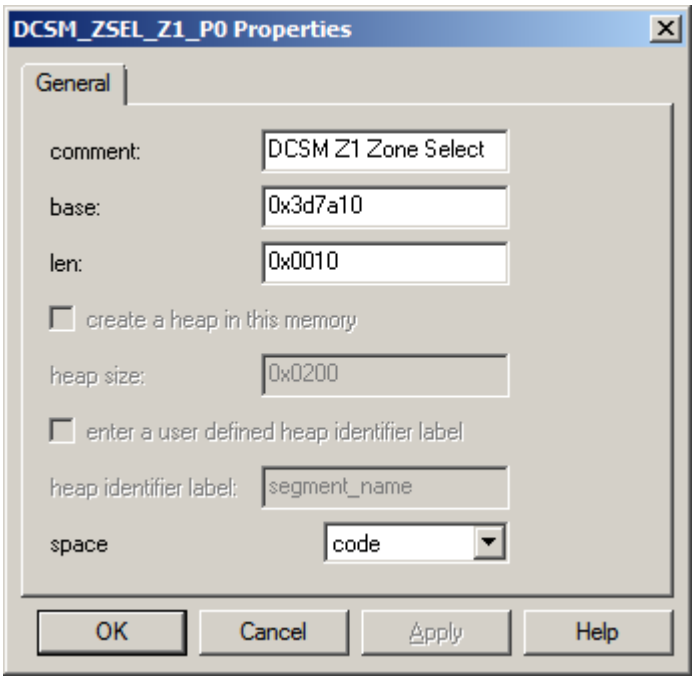

Figure 10. DSP/BIOS MEM Properties for DCSM_ZSEL_Z1_P0 Memory

The four created security sections can then be linked to these memories with the user linker command file. This is the same as the SECTIONS portion of the .cmd file for non-DSP/BIOS projects, as shown below:

```
/*************************************************************************
* User linker command file (DSP/BIOS Projects)
*************************************************************************/

SECTIONS
{
   dcsm_otp_z1:      > DCSM_OTP_Z1_P0,     PAGE = 0
   dcsm_zsel_z1:     > DCSM_ZSEL_Z1_P0,    PAGE = 0
   dcsm_otp_z2:      > DCSM_OTP_Z2_P0,     PAGE = 0
   dcsm_zsel_z2:     > DCSM_ZSEL_Z2_P0,    PAGE = 0
}
```

6 Executing Your Code from Flash after a DSP Reset

(DSP/BIOS and non-DSP/BIOS projects)

F28xxx devices contain a ROM bootloader that can transfer code execution to the flash after a device reset. The ROM bootloader is detailed in references [18 - 24]. When the boot mode selection pins are configured for 'Jump-to-Flash' mode, the ROM bootloader will branch to the instruction located at the jump-to-Flash target address in the flash. This address is 0x3F7FF6 for F281x, F280x, F2801x, F28044, F2802x, F2803x, and F2806x devices, 0x3F7FFE for F280x5 devices, and 0x33FFF6 for F2833x and F2823x devices. Always confirm the address for your specific device in either the device datasheet [1 - 8], or in the ROM bootloader documentation [18 - 24]. The user should place an instruction that branches to the beginning of their code at this address. A long branch (LB in assembly code) occupies two 16-bit words, and the device memory map is designed to accommodate this.

In general, the branch instruction will branch to the start of the C-environment initialization routine located in the C-compiler runtime support library. The entry symbol for this routine is *_c_int00*. No C code can be executed until this setup routine is run. Alternately, there is sometimes a need to execute a small amount of assembly code prior to starting your C application (for example, to disable the watchdog timer peripheral). In this case, the branch instruction should branch to the start of your assembly code. Regardless, there is a need to properly locate this branch instruction in the flash. The easiest way to do this is with assembly code. The following example creates a named initialized section called *codestart* that contains a long branch to the C-environment setup routine. The codestart section should be placed in memory with the user linker command file.

```
;*************************************************************************
; CodeStartBranch.asm
;*************************************************************************

     .ref _c_int00

     .sect "codestart"
     LB _c_int00                       ;branch to start of code

     .end                              ;end of file CodeStartBranch.asm
```

For non-DSP/BIOS projects, the user should define a memory named (for example) *BEGIN_FLASH* on PAGE 0 in the MEMORY portion of the user linker command file. The section *codestart* can then be linked to this memory. The 0x3F7FF6 address used in the following example applies to F281x, F280x, F2801x, F28044, F2802x, F2803x, and F2806x devices. Modify the address for other devices.

```
/*********************************************************************
* User's linker command file (non-DSP/BIOS Projects)
*********************************************************************/

MEMORY
{
PAGE 0:    /* Program Memory */
   BEGIN_FLASH : origin = 0x3F7FF6, length = 0x000002
PAGE 1:    /* Data Memory */
}

SECTIONS
{
/*** Jump to Flash boot mode entry point ***/
codestart:      > BEGIN_FLASH,   PAGE = 0
}
```

For DSP/BIOS projects, the user should define the memory named *BEGIN_FLASH* (for example) using the memory section manager of the DSP/BIOS configuration tool. Figure 11 shows the memory section manager properties for this memory with the 'base' address entry set for a F281x, F280x, F2801x, F2802x, F2803x, F28044, or F2806x device.

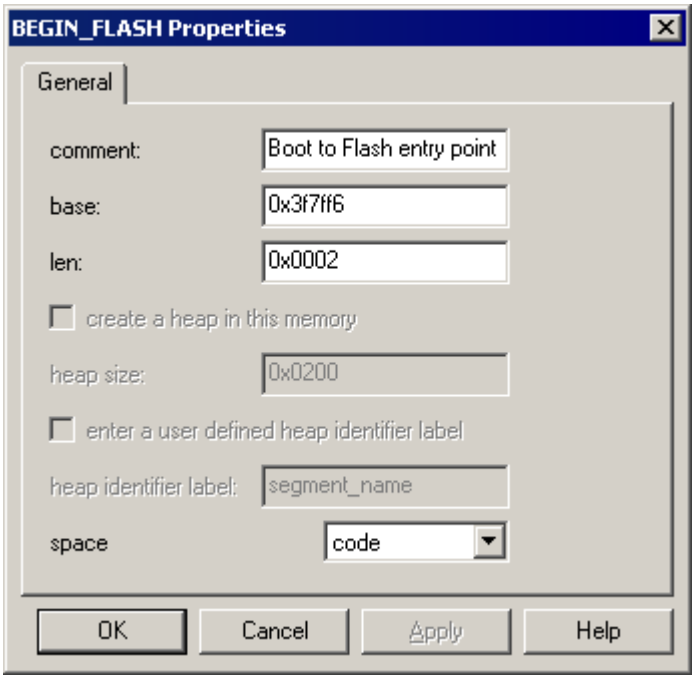

Figure 11. DSP/BIOS MEM Properties for Jump to Flash Entry Point

The section *codestart* can then be linked to this memory in the user linker command file. For DSP/BIOS projects, the linker command file would appear as:

```
/*************************************************************
* User's linker command file (DSP/BIOS projects)
*************************************************************/

SECTIONS
{
/*** Jump to Flash boot mode entry point ***/
codestart:      > BEGIN_FLASH,  PAGE = 0
}
```

7 Disabling the Watchdog Timer During C-Environment Boot

(DSP/BIOS and non-DSP/BIOS projects)

The C-environment initialization function in the C compiler runtime support library, *_c_int00*, performs the initialization of global and static variables. This involves a data copy from the *.cinit* section (located in on-chip flash memory) to the *.ebss* section (located in RAM) for each initialized global variable. For example, when a global variable is declared in source code as:

```
int x=5;
```

the "5" is placed into the initialized section *.cinit*, whereas space is reserved in the .ebss section for the symbol "x." The *_c_int00* routine then copies the "5" to location "x" at runtime. When a large number of initialized global and static variables are present in the software, the watchdog timer can timeout before the C-environment boot routine can finish and call *main()* (where the watchdog can be either configured and serviced, or disabled). This problem may not manifest itself during code development in RAM since the data copy from a *.cinit* section linked to RAM will occur at a fast pace. However, when the *.cinit* section is linked to internal flash, copying each data word will take multiple cycles since the internal flash memory defaults to the maximum number of wait-states (wait-states are not configured until the user code reaches *main()*). In addition, the code performing the data copying is executing from flash, which further increases the time needed to complete the data copy (the code fetches and data reads must share access to the flash). Combined with the fact that the watchdog timeout period defaults to its minimum possible value, a watchdog timeout becomes a real possibility.

You can detect the presence of this problem in your system by using the CCS debugger. Set a breakpoint at the start of main(), and also set a breakpoint at the start of _c_int00. Reset the processor, and then run. You should hit the breakpoint at _c_int00. If you do not, you have a bootmode configuration problem. If you get to _c_int00, run again. This time you should get to the breakpoint in main(). If you do not, the watchdog is timing out before you get there.

The easiest method for correcting the watchdog timeout problem is to disable the watchdog before starting the C-environment boot routine. The watchdog can later be re-enabled after reaching *main()* and starting your normal code execution flow. The watchdog is disabled by setting the WDDIS bit to a 1 in the WDCR register. To disable the watchdog before the boot routine, assembly code must be used (since the C environment is not yet setup). In Section 6, the *codestart* assembly code section implemented a branch instruction that jumped to the C-environment initialization routine, *_c_int00*. To disable the watchdog, this branch should instead jump to watchdog disabling code, which can then branch to the *_c_int00* routine. The following code example performs these tasks:

```
;=========================================================================
; File: CodeStartBranch.asm
; Devices: TMS320F28xxx
; Author: David M. Alter, Texas Instruments Inc.
; History: 02/11/05 - original (D. Alter)
;=========================================================================

WD_DISABLE    .set   1      ;set to 1 to disable WD, else set to 0

    .ref _c_int00

;=========================================================================
; Function: codestart section
; Description: Branch to code starting point
;=========================================================================
    .sect "codestart"
    .if WD_DISABLE == 1
        LB wd_disable       ;Branch to watchdog disable code
    .else
        LB _c_int00         ;Branch to start of boot.asm in RTS library
    .endif
;end codestart section

;=========================================================================
; Function: wd_disable
; Description: Disables the watchdog timer
;=========================================================================
    .if WD_DISABLE == 1

    .text
wd_disable:
    EALLOW                  ;Enable EALLOW protected register access
    MOVZ DP, #7029h>>6      ;Set data page for WDCR register
    MOV @7029h, #0068h      ;Set WDDIS bit in WDCR to disable WD
    EDIS                    ;Disable EALLOW protected register access
    LB _c_int00             ;Branch to start of boot.asm in RTS library

    .endif

;end wd_disable
;=========================================================================

    .end                    ; end of file CodeStartBranch.asm
```

8 C-Code Examples

8.1 General Overview

A code download containing CCS v5 code projects for the superset device in each of the F281x, F280x, F2802x, F2803x, F2805x, F2806x, and F2833x sub-families accompanies this report. With the exception of the F2802x, F2803x, and F2805x each device type has four distinct projects:

- F28xxx_example_nonBIOS_ram.pjt - non-DSP/BIOS project that runs from on-chip RAM
- F28xxx_example_nonBIOS_flash.pjt - non-DSP/BIOS project that runs from on-chip Flash
- F28xxx_example_BIOS_ram.pjt - DSP/BIOS project that runs from on-chip RAM
- F28xxx_example_BIOS_flash.pjt - DSP/BIOS project that runs from on-chip Flash

The F2802x, F2803x, and F2805x examples do not include the F28xxx_example_BIOS_ram.pjt since from a practical perspective there is insufficient RAM on these devices to support DSP/BIOS without utilizing the flash memory to hold the bulk of the code. Also, although the focus of this report is running code from flash, the RAM examples are provided for completeness and can be useful during the early stages of development work.

These are just examples, and have only been tested briefly. No guarantee is made about their suitability for application usage. The examples were built and tested using the following development tool versions:

CCS v5.3.0, Code Generation Tools v6.1.1, DSP/BIOS v5.42.0.07

Note that it is unlikely that the CCS projects will work correctly with earlier versions of CCS, even minor revisions earlier. CCS versions are generally backwards compatible, not forwards compatible. The projects may appear to import successfully, but they may be missing some project options and/or produce build errors. If you are using an earlier CCSv5 version, update to at least v5.3.0. Alternately, you can create a new CCS project with your old version and put the example source code of interest into the new project. The code itself will work fine. It is just the CCS project files themselves (i.e. .project, .cproject, and .ccsproject) that could have difficulty with earlier CCS versions.

The source code uses the following versions of the C2000 peripheral header files for accessing the device peripheral registers:

DSP281x Peripheral Header File structures v1.20
DSP280x Peripheral Header File structures v1.70
DSP2833x Peripheral Header File structures v1.33
F2802x Peripheral Header File structures v2.10
DSP2803x Peripheral Header File structures v1.26
F2805x Peripheral Header File structrure v1.00
F2806x Peripheral Header File structures v1.35

All needed files from the header file packages are included here (see references [25 - 32]). However, the user is encouraged to obtain the complete header file package for additional information. For the newer devices, the header files are available via the ControlSuite tool, available at http://www.ti.com/tool/controlsuite. For older devices, they are downloadable from the TI website, http://www.ti.com.

The projects were developed on the eZdspF2812, eZdspF2808, and eZdspF28335 development boards, the F2808, F28335, F28027, F28035, F28055, and F28069 Experimenter's Kits, and the F28027 and F28069 'Piccolo' Control Sticks. However, they will also run on other F28xxx board as follows:

F281x examples: These will run on any F281x board as they run entirely from internal memory and use only the flash memory common to all three devices. If running on a different board, be aware that the code configures the GPIOA0/PWM1 and GPIOF14/XF_XPLLDIS* pins as outputs. Also note that the code does configure the external memory interface on the F2812 as part of the DSP initialization process. Since most of the external memory interface does not exist on F2810 and F2811 devices (exception is the XCLKOUT pin), this initialization is not needed on these two devices (although it is harmless).

F280x examples: These will run on any F2808 board. They can also be adapted to run on other F280x, F2801x, and F28044 boards by adjusting the memory definitions (RAM and Flash) in the .cmd file for non-DSP/BIOS projects, or the .tcf file for DSP/BIOS projects. The PLL setting may also need to be adjusted depending on the crystal or oscillator used on the board and the operating frequency of the device. If running on a different board, the user should be aware that the code configures the GPIO0/ePWM1A and GPIO34 pins as outputs.

F2833x examples: These will run on any F28335 board. They can also be adapted to run on other F2833x or F2823x boards by adjusting the memory definitions (RAM and Flash) in the .cmd file for non-DSP/BIOS projects, or the .tcf file for DSP/BIOS projects. The PLL setting may also need to be adjusted depending on the crystal or oscillator used on the board and the operating frequency of the device. Also, for F2823x devices, you should change the project build options in CCS v5 to disable floating point support (go to Project → Properties, select the Build → C2000 Compiler → Runtime Model Options category, and change the 'Specify floating point support' box to blank). If running on a different board, the user should be aware that the code configures the GPIO0/ePWM1A, GPIO32, and GPIO34 pins as outputs.

F2802x examples: These will run on any F28027 board. They can also be adapted to run on other F2802x boards by adjusting the memory definitions (RAM and Flash) in the .cmd file for non-DSP.BIOS projects, or the .tcf file for DSP/BIOS projects. The PLL setting may also need to be adjusted depending on the operating frequency of the device. If running on a different board, the user should be aware that the code configures the GPIO0/ePWM1A and GPIO34 pins as outputs.

F2803x examples: These will run on any F28035 board. They can also be adapted to run on other F2803x boards by adjusting the memory definitions (RAM and Flash) in the .cmd file for non-DSP.BIOS projects, or the .tcf file for DSP/BIOS projects. If running on a different board, the user should be aware that the code configures the GPIO0/ePWM1A and GPIO34 pins as outputs.

F2805x examples: These will run on any F28055 board. They can also be adapted to run on other F2805x boards by adjusting the memory definitions (RAM and Flash) in the .cmd file for non-DSP.BIOS projects, or the .tcf file for DSP/BIOS projects. If running on a different board, the user should be aware that the code configures the GPIO0/ePWM1A and GPIO34 pins as outputs.

F2806x examples: These will run on any F28069 board. They can also be adapted to run on other F2806x boards by adjusting the memory definitions (RAM and Flash) in the .cmd file for non-DSP.BIOS projects, or the .tcf file for DSP/BIOS projects. If running on a different board, the user should be aware that the code configures the GPIO0/ePWM1A and GPIO34 pins as outputs.

Each code project performs the same functions:

- Illustrates F28xxx device initialization. The PLL is configured for the maximum clock speed allowed for each device.
- Enables the real-time emulation mode of Code Composer Studio.
- Toggles the GPIOF14 pin on the F2812, the GPIO34 pin on F2808, F28027, F28035, F28055, and F28069, and the GPIO32 and GPIO34 pins on the F28335. This blinks the LED on the development board (for F28335, only one GPIO is connected to an LED, depending on which board is in use). In non-DSP/BIOS projects, this is done in the ADCINT ISR. In DSP/BIOS projects, a periodic function is used.
- Configures the ADC to sample the ADCINA0 channel. On F2812, F2808, F28335, and F28069 devices, this is done at a 50 kHz rate. A 25 kHz rate is used for the slower 60 MHz devices: F28027, F28035, and F28055. The 25 kHz rate is needed due to CPU loading constraints in the DSP/BIOS examples, which illustrates why you should not put a high-speed interrupt under DSP/BIOS control as was done in these examples (i.e., the ADC SWI)! Instead, execute your high frequency interrupt routines directly in the HWI. DSP/BIOS can still manage the overall system and lower frequency ISRs.
- Services the ADC interrupt. The ADC result is placed in a circular buffer of length 50 words.
- Sends out 2 kHz symmetric PWM on either the PWM1 pin (for F281x), or the ePWM1A signal mapped to the GPIO0 pin (for F2808, F28335, F28027, F28035, F28055, and F28069).
- Configures the capture unit #1. On F2808, F28335, F28027, F28035, F28055, and F28069 devices, the eCAP1 signal is mapped to the GPIO5 pin.
- Services the capture #1 interrupt. Reads the capture result and computes the pulse width.

8.2 Directory Structure

Each code project is completely self-contained in terms of all needed files (with the exception of the C-compiler runtime support (RTS) library which is taken from the folder of the code generation tools being used by the CCS project). Table 3 provides a description of the directory structure for the individual CCS projects.

Table 3. CCS Example Code Directory Descriptions

File Directory	Contents
<project_root>	Contains the CCS created project files: .ccsproject, .cdtbuild, .cdtproject, .project, and DSP/BIOS .tcf.
<project_root>\.settings	Contains CCS created project setting files
<project_root>\cmd	Contains the user linker command file (.cmd file)
<project_root>\DSP28xxx_headers\cmd or <project_root>\F28xxx_headers\cmd	Contains the needed linker command file from the Peripheral Header File structures for the targeted device.
<project_root>\DSP28xxx_headers\include or <project_root>\F28xxx_headers\include	Contains the needed include files from the Peripheral Header File structures for the targeted device.
<project_root>\include	Contains include files (.h files)
<project_root>\src	Contains source code files (.c and .asm files)

8.3 Additional Information

1) After building a project, the .out file will be located in the <project_root>\Debug directory.

2a) If using the RAM examples, your board should be configured for "Jump to H0 SARAM" (F2812) or “Jump to M0 SARAM” (F2808, F28027, F28035, F28055, F28069, F28335) boot mode. Check the reference manual for your board to confirm any needed jumper settings, and also see the Boot ROM user’s guide for your device (references [18 - 24]). A summary for some development boards is given below. Check the board jumpers/dip-switch to be:

eZdspF2812: JP1 2-3 (MP/MC*)
JP9 1-2 (PLL)
JP7 2-3 (boot mode selection)
JP8 2-3 (boot mode selection)
JP11 1-2 (boot mode selection)
JP12 2-3 (boot mode selection)

eZdspF2808: DIP SW1: 1 = ON
2 = OFF
3 = ON

eZdspF28335: DIP SW1: 1 = ON
2 = ON
3 = OFF
4 = ON

F2808 Experimenter's Kit: Current versions of this board are hardwired for jump-to-flash bootmode. The debugger can be used to set the PC to begin execution from the code entry point (e.g. *_c_int00*). In CCS v5, Run → Restart will set the PC to this entry point.

F28335 Experimenter's Kit: Current versions of this board are hardwired for jump-to-flash bootmode. The debugger can be used to set the PC to begin execution from the code entry point (e.g. *_c_int00*). In CCS v5, Run → Restart will set the PC to this entry point.

F28027 Experimenter's Kit or ControlStick: When the emulator is connected to the F28027, the debugger is used to select the boot mode. Check the Scripts menu in CCS v5 for boot mode selection options.

F28035 Experimenter's Kit: When the emulator is connected to the F28035, the debugger is used to select the boot mode. Check the Scripts menu in CCS v5 for boot mode selection options.

F28055 Experimenter's Kit: When the emulator is connected to the F28055, the debugger is used to select the boot mode. Check the Scripts menu in CCS v5 for boot mode selection options. Note that on the F28055 ISO Control Card, the JTAG emulation is electrically isolated from the processor side of the board. A by-product of this design is that DIP SW4.1 controls TRSTn signal connection between the processor and the on-board emulation chip. SW4.1 must be in the ON position (up) in order for the JTAG to function (the ON position makes the TRSTn connection). See the F28055 ISO Control Card documentation for more information.

F28069 Experimenter's Kit or ControlStick: When the emulator is connected to the F28069, the debugger is used to select the boot mode. Check the Scripts menu in CCS v5 for boot mode selection options.

If the above methods do not seem to be working, check the reference manual for your board to confirm any needed jumper settings, and also see the Boot ROM user's guide for your device (references [18 - 24]).

2b) If using the FLASH examples, your board should be configured for "Jump to Flash" boot mode. Check the reference manual for your board to confirm any needed jumper settings, and also see the Boot ROM user's guide for your device (references [18 - 24]). A summary for some development boards is given below. Check the board jumpers/dip-switch to be:

eZdspF2812: JP1 2-3 (MP/MC*)
JP9 1-2 (PLL)
JP7 1-2 (boot mode selection)
JP8 don't care (boot mode selection)
JP11 don't care (boot mode selection)
JP12 don't care (boot mode selection)

eZdspF2808: DIP SW1: 1 = OFF
2 = OFF
3 = OFF

eZdspF28335: DIP SW1: 1 = OFF
2 = OFF
3 = OFF
4 = OFF

F2808 Experimenter's Kit: This board is hardwired jump-to-flash boot mode.

F28335 Experimenter's Kit: This board defaults to jump-to-flash boot mode.

F28027 Experimenter's Kit or ControlStick: When the emulator is connected, the debugger is used to select the boot mode. Check the Scripts menu in CCS v5 for selection options. When the emulator is disconnected, the board will boot in jump-to-flash mode provided the OTP_KEY and OTP_BMODE locations in the OTP have not been otherwise programmed, and on the Experimenter's Kit only:

DIP SW1: 1 = ON
2 = ON

F28035 Experimenter's Kit: When the emulator is connected, the debugger is used to select the boot mode. Check the Scripts menu in CCS v5 for boot mode selection options. When the emulator is disconnected, the board will boot in jump-to-flash mode provided the OTP_KEY and OTP_BMODE locations in the OTP have not been otherwise programmed, and DIP SW2 is set as:

DIP SW2: 1 = ON
2 = ON

F28055 Experimenter's Kit: When the emulator is connected, the debugger is used to select the boot mode. Check the Scripts menu in CCS v5 for boot mode selection options. When the emulator is disconnected, the board will boot in jump-to-flash mode provided the OTP_KEY and OTP_BMODE locations in the OTP have not been otherwise programmed, and the following DIP switches are set as:

DIP SW1: 1 = ON
2 = ON
DIP SW4: 1 = OFF
2 = don't care

The SW4 setting is required because the JTAG emulation on the F28055 ISO Control Card is electrically isolated from the processor side of the board. A by-product of this design is that SW4.1 must be in the OFF position (down) for the processor to operate correctly in standalone mode (no emulator connected). SW4.1 controls the connection of the TRSTn signal between the processor and the on-board emulator chip, and the OFF position breaks the connection. See the F28055 ISO Control Card documentation for more information.

F28069 Experimenter's Kit or ControlStick: When the emulator is connected, the debugger is used to select the boot mode. Check the Scripts menu in CCS v5 for boot mode selection options. When the emulator is disconnected, the board will boot in jump-to-flash mode provided the OTP_KEY and OTP_BMODE locations in the OTP have not been otherwise programmed, and on the Experimenter's Kit only:

DIP SW2: 1 = ON

2 = ON

If the above methods do not seem to be working, check the reference manual for your board to confirm any needed jumper settings, and also see the Boot ROM user's guide for your device (references [18 - 24]).

3) There has not been too much attention given to where everything is linked. The goal in writing these example projects was to simply get them working. If these projects are used as a starting point for code development, the linking may need to be tuned to get better performance (e.g., to avoid memory block access contention, or to better manage memory block utilization).

4) For non-DSP/BIOS projects, a complete set of interrupt service routines are defined in the file *DefaultIsr_nonBIOS.c*. Each interrupt is executed directly in its hardware ISR. However, with the exception of the ADCINT and ECAP1INT (or CAPINT1 on F2812), each ISR actually executes an ESTOP0 instruction (emulation stop) to trap spurious interrupts during debug, followed by an endless loop. In production code, you would want to vector unused interrupts to some sort of error handling routine of your own design (as opposed to just trapping the code). Note that each ISR is using the "interrupt" keyword which tells the compiler to perform a context save/restore upon function entry/exit.

5) For DSP/BIOS projects, a complete set of (hardware) interrupt service routines are defined in the file *DefaultIsr_BIOS.c*. Each ISR can be hooked to the desired interrupt using the HWI manager in the DSP/BIOS configuration tool. As is, the code examples only assign the two ISRs in use, specifically ADCINT1 and ECAP1INT. Also, the DSP/BIOS Interrupt Dispatcher is being used to handle the context save/restore, which is why the ISRs are not using the "interrupt" keyword (as in the non-DSP/BIOS case). In these examples, the ECAP1INT ISR (or CAPINT1 ISR for F2812) is performed directly in the *DefaultIsr_BIOS.c* file (as an example of reducing latency), whereas the ADC interrupt function in *DefaultIsr_BIOS.c* posts a SWI to perform the ADC routine. These are just examples. Note that the ECAP1INT (and CAPINT1) ISRs are using the DSP/BIOS dispatcher to perform context save/restore (as selected in the HWI manager of the configuration tool). If absolute minimum latency is required (for some time critical ISR), one could disable the interrupt dispatcher for that interrupt, and add the "interrupt" keyword to the ISR function declaration. Note that doing so will preclude the user for utilizing any DSP/BIOS functionality in that ISR. Also, one should consider the potential impact to task stack size requirements since the time critical ISR can now run within the context of any of the existing stacks (system stack, or any task stack) depending on what thread is active when the critical interrupt occurs. Finally, the HWI manager in the DSP/BIOS Configuration tool defaults all unused interrupts to a routine called HWI_unused(). This routine is essentially and code trap using an endless loop. For production code, you should vector all unused interrupts to some sort of error handling routine of your own design.

References

1. TMS320F2810, TMS320F2811, TMS320F2812, TMS320C2810, TMS320C2811, TMS320C2812 Digital Signal Processors Data Manual (SPRS174)
2. TMS320F2809, TMS320F2808, TMS320F2806, TMS320F2802, TMS320F2801, TMS320C2802, TMS320C2801, TMS320F2801x DSPs Data Manual (SPRS230)
3. TMS320F28044 Digital Signal Processor Data Manual (SPRS357)
4. TMS320F28335, TMS320F28334, TMS320F28332, TMS320F28235, TMS320F28234, TMS320F28232 Digital Signal Controllers Data Manual (SPRS439)
5. TMS320F28020, TMS320F28021, TMS320F28022, TMS320F28023, TMS320F28026, TMS320F28027 Piccolo Microcontrollers Data Manual (SPRS523)
6. TMS320F28030, TMS320F28031, TMS320F28032, TMS320F28033, TMS320F28034, TMS320F28035 Piccolo Microcontrollers Data Manual (SPRS584)
7. TMS320F28050, TMS320F28051, TMS320F28052, TMS320F28053, TMS320F28054, TMS320F28055 Piccolo Microcontrollers Data Manual (SPRS797)
8. TMS320F28062, TMS320F28063, TMS320F28064, TMS320F28065, TMS320F28066, TMS320F28067, TMS320F28068, TMS320F28069 Piccolo Microcontrollers Data Manual (SPRS698)
9. TMS320C28x CPU and Instruction Set Reference Guide (SPRU430)
10. TMS320C28x Floating Point Unit and Instruction Set Reference Guide (SPRUE02)
11. TMS320C28x Assembly Language Tools User's Guide (SPRU513)
12. TMS320C28x Optimizing C/C++ Compiler User's Guide (SPRU514)
13. TMS320x281x DSP System Control and Interrupts Reference Guide (SPRU078)
14. TMS320x280x, 2801x, 2804x DSP System Control and Interrupts Reference Guide (SPRU712)
15. TMS320x2833x System Control and Interrupts Reference Guide (SPRUFB0)
16. TMS320x2802x System Control and Interrupts Reference Guide (SPRUFN3)
17. TMS320x2803x System Control and Interrupts Reference Guide (SPRUGL8)
18. TMS320x2805x Piccolo Technical Reference Manual (SPRUHE5)
19. TMS320x2806x Piccolo Technical Reference Manual (SPRUH18)
20. TMS320x281x DSP Boot ROM Reference Guide (SPRU095)
21. TMS320x280x, 2801x, 2804x Boot ROM Reference Guide (SPRU722)
22. TMS320x2833x, 2832x Boot ROM Reference Guide (SPRU963)
23. TMS320x2802x Piccolo Boot ROM Reference Guide (SPRUFN6)
24. TMS320x2803x Piccolo Boot ROM Reference Guide (SPRUGO0)
25. F281x C/C++ Header Files and Peripheral Examples (SPRC097)
26. F280x C/C++ Header Files and Peripheral Examples (SPRC191)
27. F2804x C/C++ Header Files and Peripheral Examples (SPRC324)
28. F2833x/C2823x C/C++ Header Files and Peripheral Examples (available in ControlSuite)
29. F2802x C/C++ Header Files and Peripheral Examples (available in ControlSuite)
30. F2803x C/C++ Header Files and Peripheral Examples (available in ControlSuite)
31. F2805x C/C++ Header Files and Peripheral Examples (available in ControlSuite)
32. F2806x C/C++ Header Files and Peripheral Examples (available in ControlSuite)

Revision History

Revision	Date	Who	Description of Major Changes from Previous Version
SPRA958L	Jan 23, 2013	D. Alter	- Minor text touch-up. - Added F2805x device support including code examples. - Expanded Section 5 to cover the dual-zone code security module on F2805x. - Removed CCS v4 code examples. - Changed F28069 example to 80 MHz to 90 MHz operation. - All code tested with CCS v5.3.0, C-compiler v6.1.1, and DSP/BIOS v5.42.0.07.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as “components”) are sold subject to TI’s terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI’s terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

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

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license 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 significant portions 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. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

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

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI’s goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or “enhanced plastic” are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have ***not*** been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

Products		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Applications Processors	www.ti.com/omap	**TI E2E Community**	e2e.ti.com
Wireless Connectivity	www.ti.com/wirelessconnectivity		

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2013, Texas Instruments Incorporated

Looking for pricing, stock, or lifecycle information?

Click below to explore more details on WIN SOURCE:

- View TMDSC2KWRKSHPKIT on WIN SOURCE
- Texas Instruments Information

Optimize Your Supply Chain with WIN SOURCE Solutions

- Global Sourcing Solution
- Obsolete Management
- Cost Control Management
- Shortage Management
- Alternative Solution
- Excess Inventory Management