TEXAS INSTRUMENTS

Application Report

SPRA958D – September 2004

Running an Application from Internal Flash Memory on the TMS320F281x DSP

David M. Alter *DSP Applications - Semiconductor Group*

ABSTRACT

Several special requirements exist for running an application from on-chip flash memory on the TMS320F281x DSP. These requirements generally do not manifest themselves during development in RAM since the Code Composer Studio™ debugger can mask problems associated with initialized sections and how they are linked to memory. This application report covers the requirements needed to properly configure application software 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 code projects are included that run from on-chip flash on the eZdsp™ F2812 development board (or alternately any F2810, F2811, or F2812 DSP board). Code examples that run from internal RAM are also provided for completeness. These code examples provide a starting point for code development, if desired.

Note that the issues discussed in this application report apply directly to F281x members of the F28x DSP family, specifically the F2810, F2811, and F2812 DSP devices. Applicability to future devices in the F28x family, although quite likely, is not guaranteed. In addition, the requirements and techniques presented in this application report for DSP/BIOS projects apply specifically to Code Composer Studio v2.21.04, v2.21.00, and v2.20.18. Earlier versions of DSP/BIOS were incomplete in their support of the F281x DSP, and it is suggested that the reader upgrade to the latest version. Future versions of DSP/BIOS may have differences that make some of the items discussed in this report unnecessary (although in all likelihood backwards compatibility will be maintained, so that the techniques discussed here should still work). The reader should keep this in mind if using a newer version.

Finally, this application report does not provide a tutorial on writing and building code for the F281x DSP. It is assumed that the reader already has at least the main framework of their application code running from RAM, probably using the Code Composer Studio debugger to perform the code download. This report only identifies the special items that must be considered when moving the application into on-chip flash memory.

Revision History

Revision	Date	Who	Description of Changes from Previous Version
SPRA958D	Sept. 2004	D. Alter	Code Corrections only. The appnote text itself did not change. - Corrected counter bug in DelayUs() in DelayUs.asm. - Corrected typo in DelayUs() function header in DelayUs.asm. - Correct comment field for T2CON.TPSx bit in Ev.c.
SPRA958C	May 2004	D. Alter	Code Corrections only. The appnote text itself did not change. - Corrected GPIO pin names in ACTRA register comments in Ev.c. - Added PLL lock delay loop in Sysctrl.c. - Removed PLL configuration from .cdb files for example_BIOS_ram.pjt and example_BIOS_flash.pjt. - Corrected comment field for ADCTRL1.5 bit in ADCTRL1 initialization in Adc.c. - Corrected comment field for GPTCONA.8_7 bit in GPTCONA initialization in Ev.c. - Removed redundant PIE acknowlege from CAP1INT_ISR in DefaultISR_nonBIOS.c. - Fixed the length of the memcpy() for the PieVect section in PieCtrl_BIOS.c (removed the “+ 1”). - In main_BIOS.c and main_nonBIOS.c, changed the following comment from: /*** Copy all FLASH sections that need to run from RAM (use far_memcpy() from RTS library) ***/ to: /*** Copy all FLASH sections that need to run from RAM (use memcpy() from RTS library) ***/

SPRA958B	Jan. 2004	D. Alter	- Deleted “+ 1” length for examples of memcpy() function calls. For example, previously:
			`memcpy(&econst_runstart,`
			`        &econst_loadstart,`
			`        &econst_loadend - &econst_loadstart + 1);`
			Is now:
			`memcpy(&econst_runstart,`
			`        &econst_loadstart,`
			`        &econst_loadend - &econst_loadstart);`
			- Fixed italics on references 4 and 5.
			- Fixed code comment in main_BIOS.c and main_nonBIOS.c to say memcpy() instead of far_memcpy().

SPRA958A	Nov. 2003	D. Alter	- Corrected RAMH0 length in files f2812_nonBIOS_ram.cmd and f2812_nonBIOS_flash.cmd. - Added .cio section to f2812_nonBIOS_ram.cmd and f2812_nonBIOS_flash.cmd. - Added .cio section to Table 1. - Removed c:\ti2\c2000\rtdx\include and c:\ti2\c2000\bios\include search paths from f2812_nonBIOS_ram.pjt and f2812_nonBIOS_flash.pjt, Project -> Build_Options, Compiler Tab, Preprocessor Category, Include search path. They are not needed. - Fixed .trcdata section copy instructions in Section 4 to require copy to be done before main(). Modified example_BIOS_ram and example_BIOS_flash example code to incorporate this. - Tested code with C2000 Code Composer Studio v2.21.04.
SPRA958	Sept. 2003	D. Alter	- Original

Contents

Figures

Tables

1 Introduction

The TMS320F281x 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 for a host processor from which to bootload. Configuring an application to run from flash memory is a relatively easy matter provided that one follow 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.

Note that the issues discussed in this report apply directly to F281x members of the F28x DSP family, specifically F2810, F2811, and F2812 devices. Applicability to future F28x devices, although quite likely, is not guaranteed. In addition, the requirements and techniques presented here for DSP/BIOS projects apply specifically to Code Composer Studio v2.21.04, v2.21.00, and v2.20.18. Earlier DSP/BIOS versions were incomplete in their support for the F28x, and it is suggested that the reader upgrade to the latest version. Future versions may have differences that make some of the items discussed here unnecessary (although in all likelihood backwards compatibility will be maintained, so that the techniques discussed here should still work). The reader should keep this in mind if using a later version.

Finally, this application report does not provide a tutorial on writing and building code for the F281x DSP. It is assumed that the reader already has at least the main framework of their application code running from RAM, probably using the Code Composer Studio debugger to perform the code download. 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 [4]. The non-DSP/BIOS code projects that accompany this application report contain linker command files that can be used for reference.

When using the DSP281x Header Files v1.00 (or later) to define C structures for the on-chip peripherals (see reference [7]), a linker command file named *DSP281x_Headers_nonBIOS.cmd* is provided with them. This file contains linker MEMORY and SECTIONS declarations for linking the structures. Starting with Code Composer Studio v2.20, more than one linker command file can be added to a project. Hence, all one needs to do is add both the user linker command file as well as the DSP281x Peripheral Structures linker command file to their project. In general, the order of the linker command files is unimportant since during a project build, Code Composer Studio 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 Code Composer Studio on the Project → Build_Options, 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.

Starting with Code Composer Studio v2.20, more than one linker command file can be added to a project. Hence, all one needs to do is add both the user linker command file, as well as the DSP/BIOS generated linker command file, to their project. In general, the order of the linker command files is unimportant since during a project build, Code Composer Studio 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 Code Composer Studio on the Project → Build_Options, Link Order tab.

When using the DSP281x Header Files v1.00 (or later) to define C structures for the on-chip peripherals (see reference [7]), a linker command file named *DSP281x_Headers_BIOS.cmd* is provided with them. This file contains linker MEMORY and SECTIONS declarations for linking the structures. Simply add this linker command file to the code project as well.

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 F281x 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. 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. The -w option can be selected in Code Composer Studio on the Project → Build_Options menu, Linker tab, select the Advanced category, and then check the -w option box.

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 F2812, F2811, or F2810 devices (probably true on future F28x devices as well). Therefore, large memory model should be used. In Code Composer Studio, the large memory model is on the Project → Build_Options menu. Select the Compiler tab, choose the Advanced category, and check the -ml option box. For non-DSP/BIOS projects, one should include the large memory model C-compiler runtime support library, *rts2800_ml.lib*, into their code project (as opposed to *rts2800.lib*, which is for the small memory model). For DSP/BIOS projects, DSP/BIOS will take care of including the required library. The user should not include the *rts2800_ml.lib* (or *rts2800.lib*) library into 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 F281x DSP. Information on the function of each section can be found in reference [2]. 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 in Non-DSP/BIOS Projects

Section Name	Where to Link	Restrictions
.cinit	Flash	None
.cio	RAM	None
.const	Flash[1]	Lower 64Kw of memory
.econst	Flash	None
.pinit	Flash	None
.switch	Flash	None
.text	Flash	None
.bss	RAM[2]	Lower 64Kw of memory
.ebss	RAM	None
.stack	RAM	Lower 64Kw of memory
.sysmem	RAM[2]	Lower 64Kw of memory
.esysmem	RAM	None
.reset	RAM[3]	None

Table 1 Notes:

[1] The *.const* section is only used with the small memory model. When using large memory model, the *.econst* section is used instead. However, it is good practice to specify all possible sections in the linker command file. The *.const* section is restricted to the lower 64Kw of memory, but in reality there is no flash memory available in this address range on F281x devices. Since the *.const* section is not used with large memory model, just link the section to any memory. Then, check the .map file generated by the linker to confirm that the *.const* section is of zero length (meaning that it is not in use).

[2] The *.bss* and *.sysmem* sections are only used with the small memory model. When using large memory model, the *.ebss* and *.esysmem* sections are used instead. However, it is good practice to specify all possible sections in the linker command file. It is also wise to check the .map file generated by the linker to confirm that the .bss and .sysmem sections have a length of zero (meaning they are not in use).

[3] 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 generally is not used. Instead, the user typically creates their own branch instruction to point to the start of his code (see Sections 6 and 7). When not in use, the *.reset* section should be omitted from your code build by using a DSECT modifier in the linker command file. For example:

```
/**********************************************************************
* User's 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 Code Composer Studio v2.21.04, v2.21.00, and v2.20.18. Later versions of Code Composer Studio, although quite likely to be the same, may have some differences. The reader should check the version they are using (go to the Help → About menu in Code Composer Studio) and simply be aware of potential differences while proceeding.

Table 2. Section Linking In DSP/BIOS Projects

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)	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[3]
	C Variables Section (.ebss)	RAM
	Data Initialization Section (.cinit)	Flash
	C Function Initialization Table (.pinit)	Flash
	Constant Section (.econst)	Flash
	Constant Section (.const)	Flash[3]
	Data Section (.data)	Flash
	Data Section (.cio)	RAM
Load Address	Load Address - BIOS Code Section (.bios)	Flash[4]
	Load Address - Startup Code Section (.sysinit)	Flash[4]
	Load Address - DSP/BIOS Init Tables (.gblinit)	Flash[4]
	Load Address - TRC Initial Value (.trcdata)	Flash[1]

	Load Address - Text Section (.text)	Flash[4]
	Load Address - Switch Jump Tables (.switch)	Flash[4]
	Load Address - Data Initialization Section (.cinit)	Flash[4]
	Load Address - C Function Initialization Table (.pinit)	Flash[4]
	Load Address - Constant Section (.econst)	Flash[4]
	Load Address - Constant Section (.const)	Flash[3,4]
	Load Address - Data Section (.data)	Flash[4]
	Load Address - Function Stub Memory (.hwi)	Flash[4]
	Load Address - Interrupt Service Table Memory (.hwi_vec)	Flash[2]
	Load Address - RTDX Text Segment (.rtdx_text)	Flash[4]

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 F2810, F2811, and F2812 devices, 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 *.const* and *.bss* sections are only used with the small memory model, and are restricted to the lower 64Kw of memory space. When using large memory model, the *.econst* and *.ebss* sections are used instead (these have no lower 64Kw restriction). Large memory model should always be used, and hence it does not matter where *.const* and *.bss* sections are linked. It is good practice to check the .map file generated by the linker to confirm that the *.const* and *.bss* sections have a length of zero (meaning they are not in use).

[4] 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 F281x devices. At power-up, all interrupt vectors must be located in non-volatile memory (i.e., flash), but copied to the PIE RAM as part of the device initialization procedure in your code. The PIE RAM is a specific block of RAM, which on F2810, F2811, and F2812 devices is a 256x16 block starting at address 0x000D00 in data space. For other devices, check the address in the device datasheet.

Several approaches exist for linking your interrupt vectors to flash and then copying them to the PIE RAM at runtime. One such approach is to create a constant C-structure of function pointers that contains all 128 vectors. If using the DSP281x peripheral structures (see reference [7]), such a structure, called *PieVectTableInit*, has already been created in the file *DSP281x_PieVect.c*. Since this structure is declared using the const type qualifier, it will be placed in the *.econst* section by the compiler (large memory model assumed). One simply needs to copy this structure to the PIE 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. This function is used as follows:

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

/*********************************************************************
* NOTE: This function assumes use of the DSP281x Header File
* structures (TI Literature #SPRC097).
*********************************************************************/

#include <string.h>

void main(void)
{
/*** Initialize the PIE_RAM ***/
      PieCtrlRegs.PIECRTL.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
}
```

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 particularly 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 the DSP281x peripheral structures, an uninitialized structure called *PieVectTable* is created and linked over the PIE RAM. The *memcpy()* instruction in the previous example can be replaced by:

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

Note that the length is 256. The memcpy function copies 16-bit words (as opposed to copying 128 32-bit words).

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_runstart

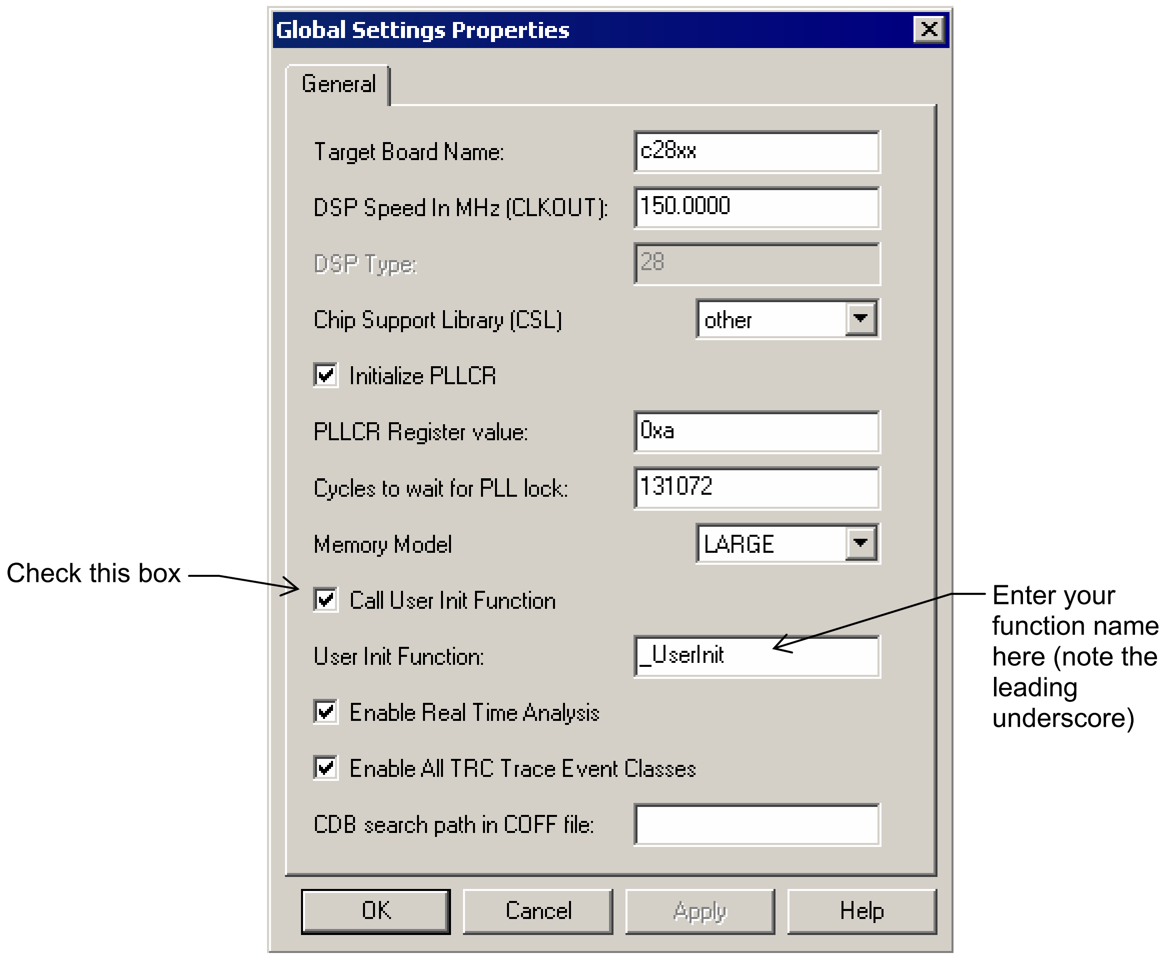
Each symbol is self-explanatory from its name. Note that the symbols are not pointers, but rather symbolically reference the 16-bit data value found at the corresponding location (i.e., start or end) of the section. The C-compiler runtime support library contains a memory copy function called *memcpy()* 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. Note that the PIE RAM is EALLOW protected. Therefore, inline EALLOW and EDIS assembly instructions must bracket the memory copy of the *.hwi_vec* section, as shown.

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

#include <string.h>

extern unsigned int hwi_vec_loadstart;
extern unsigned int hwi_vec_loadend;
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,
             &hwi_vec_loadend - &hwi_vec_loadstart);

      asm(" EDIS");                     /* Disable EALLOW protected register access */
}
```

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 project configuration file under System - Global Settings Properties, as shown in Figure 1.


Global Settings Properties
General
OK
Cancel
Apply
Help
Check this box
Enter your function name here (note the leading underscore)

Target Board Name: c28xx
DSP Speed In MHz (CLKOUT): 150.0000
DSP Type: 28
Chip Support Library (CSL) other
[✓] Initialize PLLCR
PLLCR Register value: 0xa
Cycles to wait for PLL lock: 131072
Memory Model LARGE
[✓] Call User Init Function
User Init Function: _UserInit
[✓] Enable Real Time Analysis
[✓] Enable All TRC Trace Event Classes
CDB search path in COFF file:

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_loadend

trcdata_runstart

Each symbol is 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_loadend;
extern unsigned int trcdata_runstart;

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

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

The initialization code for the flash control registers, FOPT, FPWR, FSTDBYWAIT, FACTIVEWAIT, FBANKWAIT, and FOTPWAIT, 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 the Code Security Module (CSM). If the CSM is secured, you must run the flash register initialization code from secured RAM (e.g., L0 or L1 SARAM) or the initialization code will be unable to access the flash registers. Note that the CSM is always secured at device reset, even if you are using dummy passwords of 0xFFFF.

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.  Also, this function assumes use of the DSP281x
* Header File structures (TI Literature #SPRC097).
**********************************************************************/

#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 = 5;           // Random access waitstates
      FlashRegs.FBANKWAIT.bit.PAGEWAIT = 5;           // Paged access waitstates
      FlashRegs.FOTPWAIT.bit.OTPWAIT = 5;             // Random access waitstates
      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");

} //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, we will want 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's linker command file
**********************************************************************/

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

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 RUN_START, LOAD_START, and LOAD_END 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 compiler runtime support library memory copy function *memcpy()* can be used:

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

#include <string.h>

extern unsigned int secureRamFuncs_loadstart;
extern unsigned int secureRamFuncs_loadend;
extern unsigned int secureRamFuncs_runstart;

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

/* 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 flash memory on F2810, F2811, and F2812 devices provides effective code execution performance of roughly 110 to 120 Mips (millions of instructions per second) with a DSP clock of 150 MHz. This compares to the full 150 Mip performance of on-chip RAM. It may therefore be desirable to run certain time-critical or computationally demanding routines from on-chip RAM. However, in a standalone embedded system, all code must initially reside in non-volatile memory. Therefore, separate load and run addresses must be setup for those functions running from RAM, and a copy must be performed to move them from the 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 (when using the large memory model). While special pipelining on F281x devices accelerates effective flash performance for code execution, accessing data constants located in the on-chip FLASH can take multiple cycles per access. Typical flash wait-states on a 150 MHz F281x DSP will be 5 cycles (see the device datasheet for final wait-state specifications). It may therefore be desirable to run 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 we 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's linker command file
********************************************************************/

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

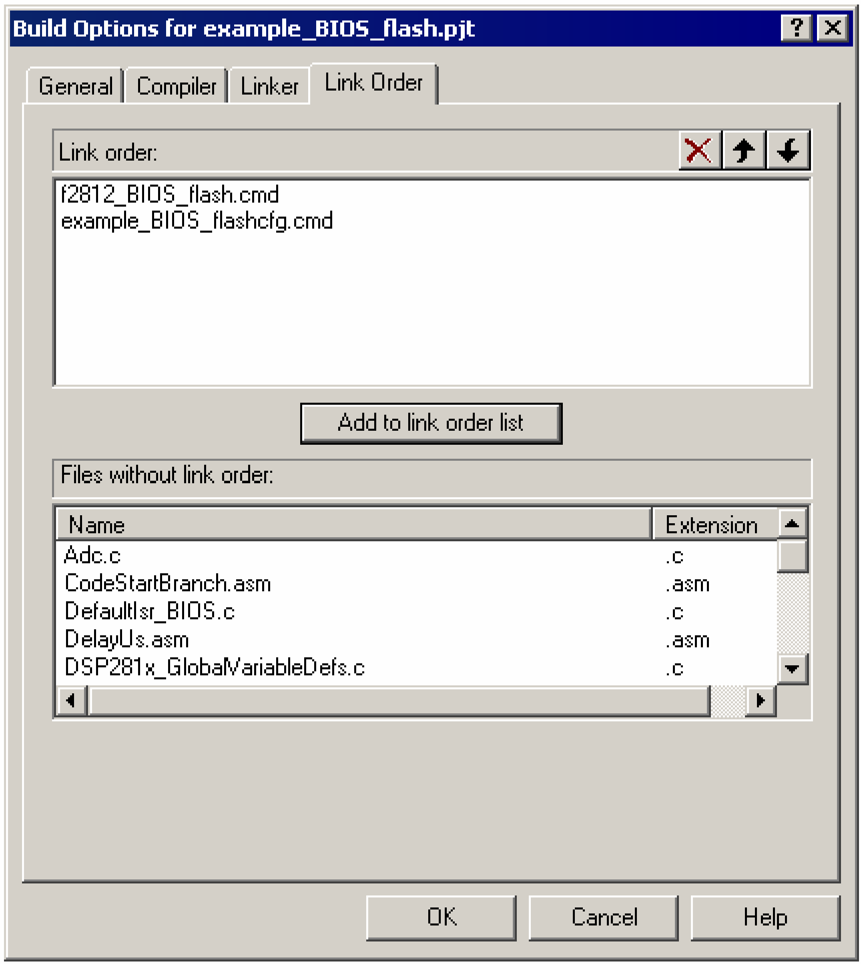
```
/**********************************************************************
* User's C-source file
**********************************************************************/

#include <string.h>

extern unsigned int econst_loadstart;
extern unsigned int econst_loadend;
extern unsigned int econst_runstart;

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

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's 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 = 0
                     RUN_START(_econst_runstart),
                     LOAD_START(_econst_loadstart),
                     LOAD_END(_econst_loadend)
}
```

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 Code Composer Studio. This is done by clicking on Project → Build_Options, selecting the Link Order tab, and then specifying the appropriate order for the linker command files in question. Figure 2 shows an example of this, where *f2812_BIOS_flash.cmd* is the name of the user linker command file, and *example_BIOS_flashcfg.cmd* is the name of the DSP/BIOS generated linker command file.



Link order:

f2812_BIOS_flash.cmd
example_BIOS_flashcfg.cmd

Files without link order:

Name	Extension
Adc.c	.c
CodeStartBranch.asm	.asm
DefaultIsr_BIOS.c	.c
DelayUs.asm	.asm
DSP281x_GlobalVariableDefs.c	.c

Figure 2. Specifying the Link Order In Code Composer Studio

Note that since the DSP/BIOS generated linker command file will also attempt to link the *.econst* section, the linker will give a warning stating "Multiple definitions of SECTION named '.econst'." This warning may be safely ignored.

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_loadend;
extern unsigned int econst_runstart;

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

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 slightly more involved than with Method 1. An initialization constant must be created using the *const* type modifier. This constant will be placed in the *.econst* section by the C compiler, and linked to flash. In addition, the actual constant is created in a user specified section using the DATA_SECTION pragma. This user specified section is linked to RAM. Finally, the user must copy the initialization array to the actual array at runtime.

Suppose for example that one wants to create a 5 word constant array called *table[ ]* to be run from RAM. We will create an initialization array called *table_init[ ]* to hold the constant values, and use a DATA_SECTION pragma to place *table[ ]* in a user defined section called *ramconsts*. The C-source file would appear as follows:

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

const int table_init[5]={1,2,3,4,5};      //initialization array

#pragma DATA_SECTION(table, "ramconsts")
int table[5];                             //actual array

void main(void)
{

}
```

The section *ramconsts* can then be linked to RAM using the user linker command file, and global symbols generated to facilitate the memory copy. The user linker command file would appear as follows:

```
/********************************************************************
* User's linker command file
********************************************************************/

SECTIONS
{
/*** User Defined Sections ***/
ramconsts:          RUN = RAM,    PAGE = 1
                    RUN_START(_ramconsts_runstart),
                    RUN_END(_ramconsts_runend),
}
```

Note that the *ramconsts* section does not require separate load and run addresses. It is an uninitialized section (you must initialize it with the data contained in the initialization array).

Finally, *table[ ]* must be initialized from *table_init[ ]* at runtime:

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

#include <string.h>

extern unsigned int ramconsts_runstart;
extern unsigned int ramconsts_runend;

void main(void)
{
/* Initialize the ramconsts section */
      memcpy(&ramconsts_runstart,
             &table_init,
             &ramconsts_runend - &ramfuncs_runstart);
}
```

5 Programming the Code Security Module Passwords

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

The CSM module on F281x devices provides protection against unwanted copying of your software. On F281x devices, the entire flash, the OTP memory, and the L0 and L1 RAM blocks are secured by the CSM (the flash configuration registers are secured as well). When secured, only code executing from secured memory can access data (read or write) in other secured memory. Code executing from unsecured memory cannot access data in secured memory. Detailed information on the CSM module can be found in reference [3].

The CSM uses a 128-bit password comprised of 8 individual 16-bit words. On F2810, F2811, and F2812 devices, these passwords are stored in the upper most 8 words of the flash (i.e., addresses 0x3F7FF8 to 0x3F7FFF). On other F28x devices, refer to the device datasheet for this information. During development, it is recommended that dummy passwords of 0xFFFF be left in the password locations. 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 place real passwords in the password locations. In addition, the CSM module on F2810, F2811, and F2812 devices requires programming values of 0x0000 into flash addresses 0x3F7F80 through 0x3F7FF5, inclusive, in order to properly secure the CSM (see reference [1]). The easiest way to accomplish both of these tasks is with some simple assembly language programming. The following is an example of an assembly code file that 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 addresses 0x3F7F80 to 0x3F7FF5, inclusive. See reference [4] for more information on the assembly language directives being used.

```
**********************************************************************
* 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 will
* be permanently locked.  See the "TMS320F28x System Control
* and Interrupts Peripheral Reference Guide" (SPRU078) 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:

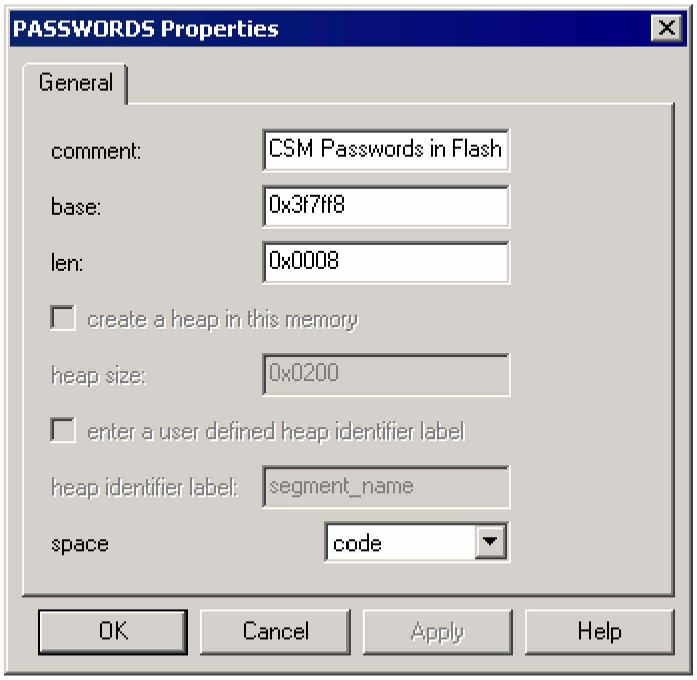
Do not use 0x0000 for all 8 passwords. Doing so will permanently lock the CSM module! See reference [3] for more information.

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 F2810, F2811, and F2812 devices (for other devices, consult the device datasheet to confirm the addresses of the password and CSM reserved locations).

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

MEMORY
{
PAGE 0:    /* Program Memory */
   CSM_RSVD    : origin = 0x3F7F80, length = 0x000076
   PASSWORDS   : origin = 0x3F7FF8, length = 0x000008
PAGE 1:    /* Data Memory */
}


SECTIONS
{
/*** Code Security Password Locations ***/
passwords:         LOAD = PASSWORDS,    PAGE = 0
csm_rsvd:          LOAD = 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 F2810, F2811, and F2812 devices. For other devices, consult the device datasheet to confirm the correct addresses and lengths.


PASSWORDS Properties

General

comment: CSM Passwords in Flash

base: 0x3f7ff8

len: 0x0008

☐ create a heap in this memory

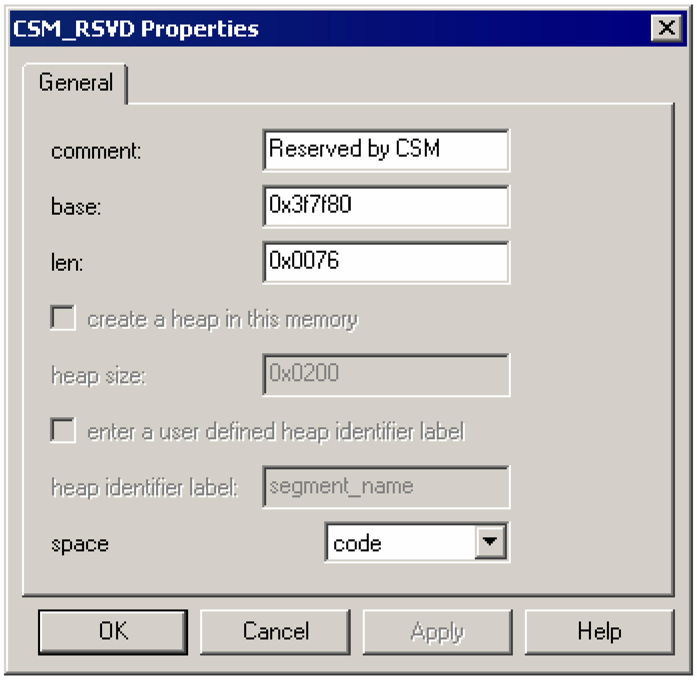
heap size: 0x0200

☐ enter a user defined heap identifier label

heap identifier label: segment_name

space: code

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


CSM_RSVD Properties

General

comment: Reserved by CSM

base: 0x3f7f80

len: 0x0076

☐ create a heap in this memory

heap size: 0x0200

☐ enter a user defined heap identifier label

heap identifier label: segment_name

space: code

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's linker command file (DSP/BIOS Projects)
********************************************************************/

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

6 Executing Your Code from Flash after a DSP Reset

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

F281x devices contain a ROM bootloader that can transfer code execution to the flash after a device reset. The ROM bootloader is detailed in reference [5]. When the boot mode selection pins are configured for "Jump to Flash" mode, the ROM bootloader will branch to the instruction located at address 0x3F7FF6 in the flash. The user should place an instruction that branches to the beginning of their code at this address. Recall that the CSM passwords begin at address 0x3F7FF8, such that exactly 2 words are available to hold this branch instruction. Not coincidentally, a long branch instruction (LB in assembly code) occupies exactly 2 words.

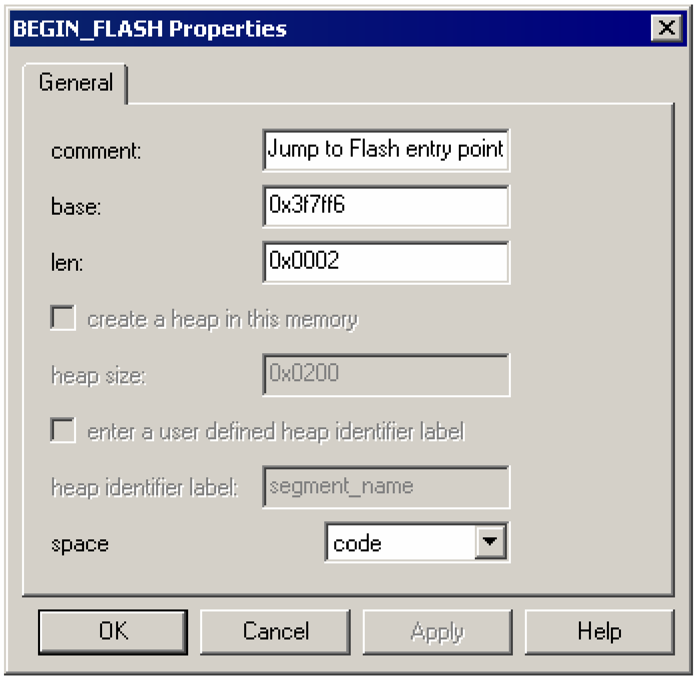
In general, the branch instruction will branch to the start of the C-environment initialization routine located in the C-compiler runtime support library. The entry symbol for this routine is *_c_int00*. No C code can be executed until this setup routine is run. Alternately, there is sometimes a need 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 following example applies to F2810, F2811, and F2812 devices. For other F28x devices, consult the device datasheet to confirm the boot to flash target address.

```
/**********************************************************************
* 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:          LOAD = BEGIN_FLASH,   PAGE = 0
}
```

For DSP/BIOS projects, the user should define the memory named (for example) *BEGIN_FLASH* using the memory section manager of the DSP/BIOS configuration tool. Figure 5 shows the memory section manager properties for this memory on F2810, F2811, and F2812 devices. For other devices, consult the datasheet to confirm the correct addresses and lengths.



comment: Jump to Flash entry point
base: 0x3f7ff6
len: 0x0002
☐ create a heap in this memory
heap size: 0x0200
☐ enter a user defined heap identifier label
heap identifier label: segment_name
space: code

Figure 5. 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:          LOAD = 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. With the far memory model, 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.

There is an easy method to detect this problem using Code Composer Studio. To test for a watchdog timeout:

1. Load the symbols for the code you have programmed into the flash (click File → Load_Symbols → Load_Symbols_Only).
2. Reset the DSP (click Debug → Reset_CPU).
3. Restart the DSP (click Debug → Restart) (this step is not necessary if the bootloader is configured for "Jump to Flash" mode).

4. Run to *main()* (click Debug → Go_Main). If you do not get to *main()*, it is pretty likely that the watchdog is expiring before the C-environment boot routine is able to complete.

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: TMS320F2812, TMS320F2811, TMS320F2810
* Author: David M. Alter, Texas Instruments Inc.
* History: 09/08/03 - 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 four different code projects accompanies this application report:

• example_nonBIOS_ram.pjt	-	non-DSP/BIOS project that runs from on-chip RAM
• example_nonBIOS_flash.pjt	-	non-DSP/BIOS project that runs from on-chip Flash
• example_BIOS_ram.pjt	-	DSP/BIOS project that runs from on-chip RAM
• example_BIOS_flash.pjt	-	DSP/BIOS project that runs from on-chip Flash

These are just examples, and have only been tested briefly. No guarantee is made about their suitability or performance for application usage. These examples were built and tested using C2000 Code Composer Studio version v2.21.04. Although the focus of this application report is running code from flash, the RAM examples are provided for completeness.

The projects were developed on the eZdsp F2812 development board. However, they will also run on any other F2810, F2811, or F2812 board as they run entirely from internal memory. If running on a different board, the user should be aware that the code configures the GPIOA0/PWM1 and GPIOF14/XF_XPLLDIS* pins as outputs. Also note that although all code and data are linked to internal memories, 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, this initialization is not needed on these two devices (although it is harmless). The only exception is the configuration of the XCLKOUT pin, which is present on F2810 and F2811 devices.

The source code utilizes the DSP281x Header File structures v1.00 for accessing peripheral registers on the F2812 device. All needed files from the DSP281x package are included here. However, the user is encouraged to download the complete DSP281x package for additional information. This is available on the TI website, http://www.ti.com (see reference [7]).

Each of the code projects perform the same functions:

- Illustrates F2812 (or F2810, or F2811) DSP initialization. The PLL is configured for x5 operation (i.e., 30 MHz XCLKIN which results in 150 MHz CPU operation).
- Enables the real-time emulation mode of Code Composer Studio.
- Toggles the GPIOF14 pin to blink the LED on the eZdsp F2812 board. 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 on ADCINA0 channel at a 50 kHz rate.
- Services the ADC interrupt. The ADC result is placed in a circular buffer of length 50 words.
- Sends out 2 kHz symmetric PWM on the PWM1 pin.
- Configures the capture unit #1.
- Services the capture #1 interrupt. Reads the capture result and computes the pulse width.

8.2 Directory Structure and File Utilizations

The four code projects mostly share the same source code files. This illustrates how the same source code can be used in RAM and flash applications, and DSP/BIOS and non-DSP/BIOS applications. Table 3 shows the directory structure of the example code, while Table 4 provides a complete inventory of all files and their utilization by each project.

Table 3. Example Code File Directories

File Directory	Contents
\DSP281x_headers\include	Contains the needed files from the DSP281x Header File structures v1.00. Note that this directory has exactly the same contents as the *DSP281x_headers\include* directory from v1.00 of reference [7].
\projects	Contains the example projects (.cdb, .cmd, .h, and .pjt files)
\src	Contains common and non-common source code files (.c and .asm files)

Table 4. Example Code File Inventory and Utilization

Filename	Project Utilization			
	example_nonBIOS_ram	example_nonBIOS_flash	example_BIOS_ram	example_BIOS_flash
\DSP281x_headers\include\DSP281x_Adc.h[1]	✓	✓	✓	✓
\DSP281x_headers\include\DSP281x_CpuTimers.h[1]	✓	✓	✓	✓
\DSP281x_headers\include\DSP281x_DefaultIsr.h[1]	✓	✓		
\DSP281x_headers\include\DSP281x_DevEmu.h[1]	✓	✓	✓	✓
\DSP281x_headers\include\DSP281x_Device.h[1]	✓	✓	✓	✓
\DSP281x_headers\include\DSP281x_Ecan.h[1]	✓	✓	✓	✓
\DSP281x_headers\include\DSP281x_Ev.h[1]	✓	✓	✓	✓
\DSP281x_headers\include\DSP281x_Gpio.h[1]	✓	✓	✓	✓
\DSP281x_headers\cmd\DSP281x_Headers_BIOS.cmd[1]			✓	✓
\DSP281x_headers\cmd\DSP281x_Headers_nonBIOS.cmd[1]	✓	✓		
\DSP281x_headers\include\DSP281x_Mcbsp.h[1]	✓	✓	✓	✓
\DSP281x_headers\include\DSP281x_PieCtrl.h[1]	✓	✓	✓	✓
\DSP281x_headers\include\DSP281x_PieVect.h[1]	✓	✓	✓[2]	✓[2]
\DSP281x_headers\include\DSP281x_Sci.h[1]	✓	✓	✓	✓

\DSP281x_headers\include\DSP281x_Spi.h[1]	✓	✓	✓	✓
\DSP281x_headers\include\DSP281x_SysCtrl.h[1]	✓	✓	✓	✓
\DSP281x_headers\include\DSP281x_Xintf.h[1]	✓	✓	✓	✓
\DSP281x_headers\include\DSP281x_XIntrupt.h[1]	✓	✓	✓	✓
\projects\example_BIOS.h			✓	✓
\projects\example_BIOS_flash.cdb				✓
\projects\example_BIOS_flash.pjt				✓
\projects\example_BIOS_flashcfg.cmd[3]				✓
\projects\example_BIOS_ram.cdb			✓	
\projects\example_BIOS_ram.pjt			✓	
\projects\example_BIOS_ramcfg.cmd[3]			✓	
\projects\f2812_BIOS_flash.cmd				✓
\projects\f2812_BIOS_ram.cmd			✓	
\projects\example_nonBIOS.h	✓	✓		
\projects\example_nonBIOS_flash.pjt		✓		
\projects\example_nonBIOS_ram.pjt	✓			
\projects\f2812_nonBIOS_flash.cmd		✓		
\projects\f2812_nonBIOS_ram.cmd	✓			
\src\Adc.c	✓	✓	✓	✓
\src\CodeStartBranch.asm	✓	✓	✓	✓
\src\DefaultIsr_BIOS.c			✓	✓
\src\DefaultIsr_nonBIOS.c	✓	✓		
\src\DelayUs.asm	✓	✓	✓	✓
\src\DSP281x_GlobalVariableDefs.c[1]	✓	✓	✓	✓
\src\Ev.c	✓	✓	✓	✓
\src\Gpio.c	✓	✓	✓	✓
\src\main_BIOS.c			✓	✓
\src\main_nonBIOS.c	✓	✓		
\src\passwords.asm		✓		✓
\src\PieCtrl_BIOS.c			✓	✓
\src\PieCtrl_nonBIOS.c	✓	✓		
\src\PieVect_nonBIOS.c	✓	✓		
\src\SetDBGIER.asm	✓	✓	✓	✓
\src\SysCtrl.c	✓	✓	✓	✓
\src\Xintf.c	✓	✓	✓	✓
\disclaimer.txt	Documentation file			
\readme.txt	Documentation file			

Table 4 Notes:

[1] This file is identical to the file of the same name found in the \v100\DSP281x_Headers subdirectory of the DSP281x Header File structures download, v1.00 (see reference [7]).

[2] Although *DSP281x_PieVect.h* is included into the flash projects, the structure *PieVectTable* that it defines (and which is linked over the PIE RAM) is not actually used by the code in DSP/BIOS projects. It is included more for completeness, and to assist with debug (e.g., for viewing the pie vectors in a watch window).

[3] The files *example_BIOS_flashcfg.cmd* and *example_BIOS_ramcfg.cmd* are created by the DSP/BIOS configuration tool, and should not be modified by the user. They are provided here only to avoid a project open error message from Code Composer Studio (since these .cmd files have been included in their respective code projects). The files *example_BIOS_flash.cdb* and *example_BIOS_ram.cdb* contain everything Code Composer Studio needs to create these two .cmd files at project build time.

8.3 Additional Information

1) The .pjt project files can be found in the \project_BIOS and \project_nonBIOS directories. After compiling a project, the .out file will be located in the \projects\Debug directory.

2) Project options have been configured assuming that C2000 Code Composer Studio has been installed in either the c:\ti or c:\ti2 directory. If your tools are installed in a different directory, you will need to modify the project options (specifically, the search path for compiler include files, and the search path for linker library files) in order to get the projects to build without error.

3a) If using the RAM examples, the F2812 on the eZdsp board should be configured for "Jump to H0" bootmode, and also have the PLL jumpered for enable. Check the board jumpers to be:

JP1 2-3 (MP/MC*)

JP9 1-2 (PLL)

JP7 2-3 (boot mode)

JP8 2-3 (boot mode)

JP11 1-2 (boot mode)

JP12 2-3 (boot mode)

If this does not seem to be working, check the reference manual for your eZdsp board to confirm the jumper settings. Jumper settings may have changed if the eZdsp board was revised.

3b) If using the FLASH examples, the F2812 on the eZdsp board should be configured for "Jump to Flash" bootmode, and also have the PLL jumpered for enable. Check the board jumpers to be:

JP1 2-3 (MP/MC*)

JP9 1-2 (PLL)

JP7 1-2 (boot mode)

JP8 don't care (boot mode)

JP11 don't care (boot mode)

JP12 don't care (boot mode)

If this does not seem to be working, check the reference manual for your eZdsp board to confirm the jumper settings. Jumper settings may have changed if the eZdsp board was revised.

4) The ram examples are linking sections in various places that may look unnecessary (e.g., the section *ramfuncs* is loaded to one ram area, and copied to and run from another ram area. On the surface, this look rather pointless. These things were done in preparation to build the flash project. In reality, a real embedded system cannot run on ram alone. It must have non-volatile memory someplace. Hence, in the flash system, you will see the same sections being loaded to flash, but copied to and run from ram.

5) 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 correctly. 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).

6) 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 CAPINT1, each ISR actually executes an ESTOP0 instruction (emulation stop) to trap spurious interrupts during debug. Note that each ISR is using the "interrupt" keyword which tells the compiler to perform a context save/restore upon function entry/exit.

7) 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. Also, the DSP/BIOS Interrupt Dispatcher can be 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 CAPINT1 ISR 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 CAPINT1 is still 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.

References

1. *TMS320F2810, TMS320F2811, TMS320F2812, TMS320C2810, TMS320C2811, TMS320C2812 Digital Signal Processors Data Manual* (SPRS174)
2. *TMS320C28x Optimizing C/C++ Compiler User's Guide* (SPRU514)
3. *TMS320F28x System Control and Interrupts Peripheral Reference Guide* (SPRU078)
4. *TMS320C28x Assembly Language Tools User's Guide* (SPRU513)
5. *TMS320F28x Boot ROM Peripheral Reference Guide* (SPRU095)
6. *TMS320C28x DSP CPU and Instruction Set Reference Guide* (SPRU430)
7. *C281x C/C++ Header Files and Peripheral Example Code* (SPRC097)