

Running an Application from Internal Flash Memory on the TMS320F28xx DSP

David M. Alter

DSP Applications - Semiconductor Group

ABSTRACT

Several special requirements exist for running an application from on-chip flash memory on the TMS320F28xx 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 and eZdsp F2808 development boards (or alternately, any F2801, F2806, F2808, 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 current F281x and F280x members of the TMS320F28xx DSP family, specifically the F2801, F2806, F2808, F2810, F2811, and F2812 DSP devices. Applicability to future devices in the F28xx family, although quite likely, is not guaranteed. In addition, the code and techniques presented in this application report for DSP/BIOS projects was developed on Code Composer Studio v3.1 using C-compiler v4.1.1 and DSP/BIOS v5.20. Earlier versions of DSP/BIOS used a different configuration file format. 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 F28xx 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.

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.....	5
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).....	10
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	14
4.6	Maximizing Performance by Linking Critical Global Constants to RAM	15
4.6.1	Method 1: Running All Constant Arrays from RAM	15
4.6.2	Method 2: Running a Specific Constant Array from RAM	18
5	Programming the Code Security Module Passwords.....	19
6	Executing Your Code from Flash after a DSP Reset.....	23
7	Disabling the Watchdog Timer During C-Environment Boot.....	25
8	C-Code Examples.....	27
8.1	General Overview.....	27
8.2	Directory Structure and File Utilizations	28
8.3	Additional Information.....	33
	References.....	35
	Revision History.....	36

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	17
Figure 3.	DSP/BIOS MEM Properties for CSM Password Locations	22
Figure 4.	DSP/BIOS MEM Properties for CSM Reserved Locations	22
Figure 5.	DSP/BIOS MEM Properties for Jump to Flash Entry Point.....	24

Tables

Table 1.	Section Linking in Non-DSP/BIOS Projects (Large memory model)	6
Table 2.	Section Linking In DSP/BIOS Projects (Large Memory Model).....	7
Table 3.	Example Code File Directories.....	28
Table 4.	F281x Example Code File Inventory and Utilization	29
Table 5.	F280x Example Code File Inventory and Utilization	31

1 Introduction

The TMS320F28xx 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 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 application report apply directly to current F281x and F280x members of the TMS320F28xx DSP family. Throughout the remainder of this report, the term *current F28xx devices* will refer specifically to the F2801, F2806, F2808, F2810, F2811, and F2812 DSP devices. Applicability to future devices in the F28xx family, although quite likely, is not guaranteed. In addition, the code and techniques presented in this application report for DSP/BIOS projects was developed on Code Composer Studio v3.1 using C-compiler v4.1.1 and DSP/BIOS v5.20. Earlier versions of DSP/BIOS used a different configuration file format. 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 F28xx 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 [6]. The non-DSP/BIOS code projects that accompany this application report contain linker command files that can be used for reference.

The DSP281x and DSP280x peripheral header files contain linker command files named *DSP281x-Headers_nonBIOS.cmd* and *DSP280x-Headers_nonBIOS.cmd*, respectively (see references [10] and [11]). These files contain linker MEMORY and SECTIONS declarations for linking the peripheral register structures. Since Code Composer Studio supports having more than one linker command file in a project, all one needs to do is add the header file linker command file to their project in addition to their user linker command file. 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.

Code Composer Studio supports having more than one linker command file in 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.

The DSP281x and DSP280x peripheral header files contain linker command files named *DSP281x-Headers_nonBIOS.cmd* and *DSP280x-Headers_nonBIOS.cmd*, respectively (see references [10] and [11]). These files contain linker MEMORY and SECTIONS declarations for linking the peripheral register structures. Simply add the appropriate one of these linker command files to your 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 F28xx 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. It is checked by default for new projects.

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 current F28xx devices, and this will likely be true for future F28xx 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 F28xx DSP. Information on the function of each section can be found in reference [3]. 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 (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 generally is not used. Instead, the user typically creates their own branch instruction to point to the starting point of the code (see Sections 6 and 7). 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'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 DSP/BIOS v5.20. 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. To check your DSP/BIOS version from within Code Composer Studio, go to the Help → About menu, click the Component_Manager button, and view the TMS320C28XX DSP/BIOS version under the Target_Content_(DSP/BIOS) tree.

Table 2. Section Linking In 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
	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 current F28xx 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 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 F28xx 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 current F28xx devices 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 DSP281x or DSP280x peripheral structures (see references [10] and [11]), such a structure, called *PieVectTableInit*, has already been created in the file *DSP281x_PieVect.c* or *DSP280x_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. This function is used as follows:

```

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

/*****
* NOTE: This function assumes use of the DSP281x or DSP280x Header
* File structures (TI Literature #SPRC097 and #SPRC191).
*****/

#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
}

```

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 the DSP281x or DSP280x 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);
```

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 PIEVECT 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.

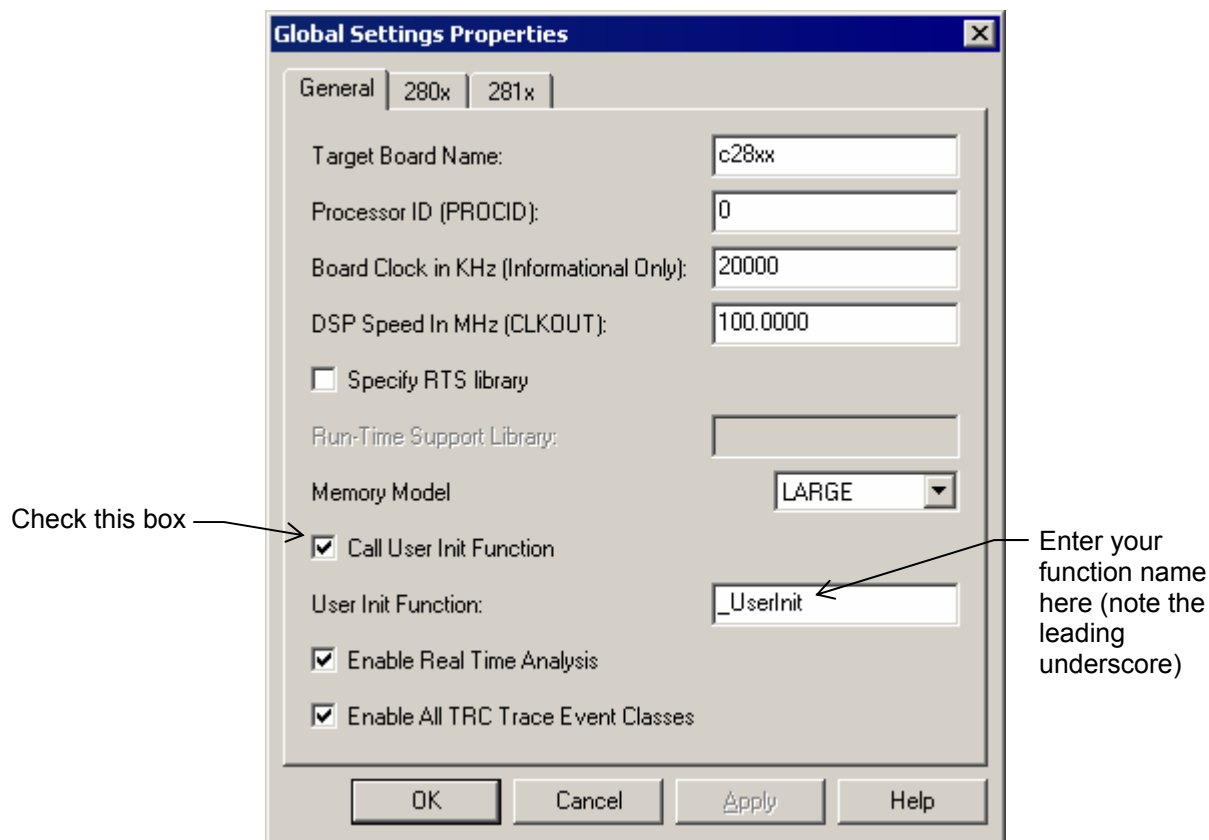


Figure 1. Specifying the User Init Function in the DSP/BIOS Configuration tool

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, although the ROM bootloader will unlock it 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. Wait-states shown here are specific to current F280x
* devices operating at 100 MHz.
* Also, this function assumes use of the DSP281x or DSP280x Header
* File structures (TI Literature #SPRC097 and #SPRC191).
*****/

#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");

} //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 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_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 RAM memory on current F28xx devices provides code execution performance of 150 MIPS (millions of instructions per second) at 150 MHz on F281x devices, and 100 MIPS at 100 MHz on F280x devices. However, the on-chip flash memory on these devices provides effective code execution performance that is slightly less: roughly 90 – 100 MIPS on F281x devices, and roughly 85 – 90 MIPS on F280x devices. 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 (large memory model assumed). While special pipelining on current F28xx 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 will be 5 cycles on a 150 MHz F281x DSP, and 3 cycles on a 100 MHz F280x DSP (see the device datasheet for flash wait-state specifications). 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'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 = 1
                  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 `f2808_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.

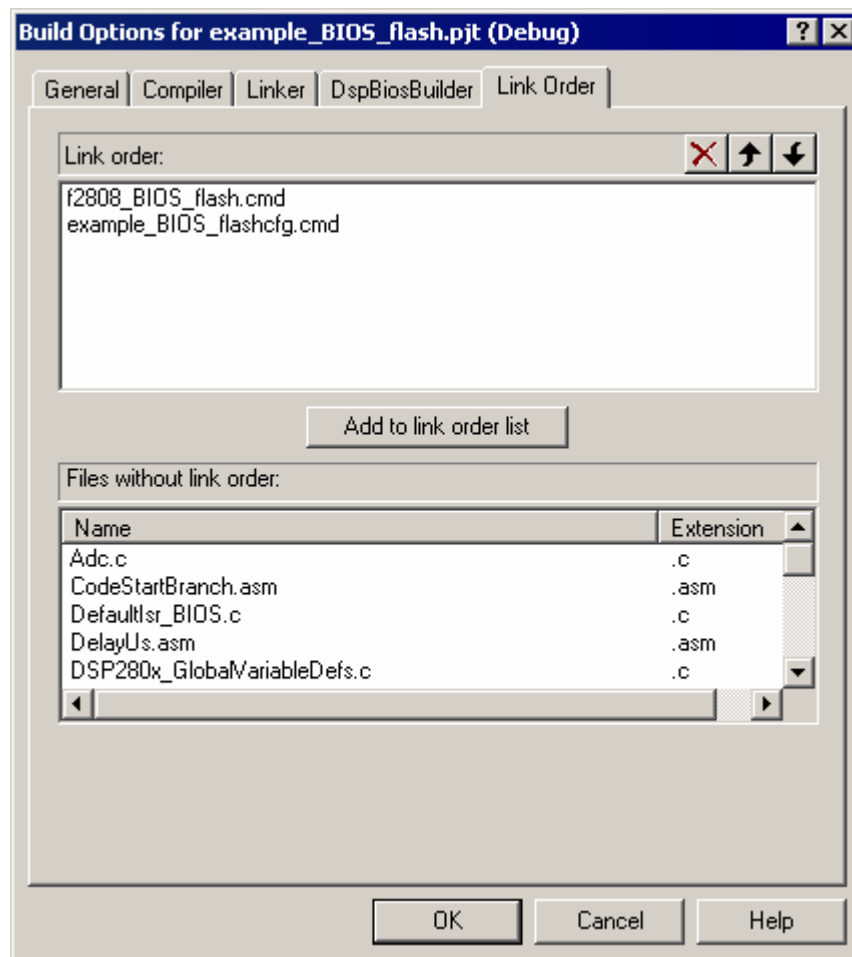


Figure 2. Specifying the Link Order In Code Composer Studio

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 can 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 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` pragmas 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's linker command file
*****/

SECTIONS
{
/** User Defined Sections */
ramconsts:          LOAD = FLASH, PAGE = 0
                    RUN  = RAM,   PAGE = 1
                    LOAD_START(_ramconsts_loadstart),
                    LOAD_END(_ramconsts_loadend),
                    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_loadend;
extern unsigned int ramconsts_runstart;

void main(void)
{
/* Initialize the ramconsts section */
    memcpy(&ramconsts_runstart,
           &ramconsts_loadstart,
           &ramconsts_loadend - &ramconsts_loadstart);
}

```

5 Programming the Code Security Module Passwords

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

The CSM module on F28xx devices provides protection against unwanted copying of your software. On current F28xx 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 references [4] and [5].

The CSM uses a 128-bit password comprised of 8 individual 16-bit words. On current F28xx devices, these passwords are stored in the upper most 8 words of the flash (i.e., addresses 0x3F7FF8 to 0x3F7FFF). 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 current F28xx devices requires programming values of 0x0000 into flash addresses 0x3F7F80 through 0x3F7FF5, inclusive, in order to properly secure the CSM (see references [1] and [2]). The easiest way to accomplish both of these tasks is with a little 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 [6] 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 will
* be permanently locked. See the "TMS320x281x System Control and
* Interrupts Peripheral Reference Guide" (SPRU078) or the "TMS320x280x
* System Control and Interrupts Peripheral Reference Guide" (SPRU712)
* 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 references [4] and [5] 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 current F28xx 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:      > 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 current F28xx devices. For other devices, consult the device datasheet to confirm the correct addresses and lengths.

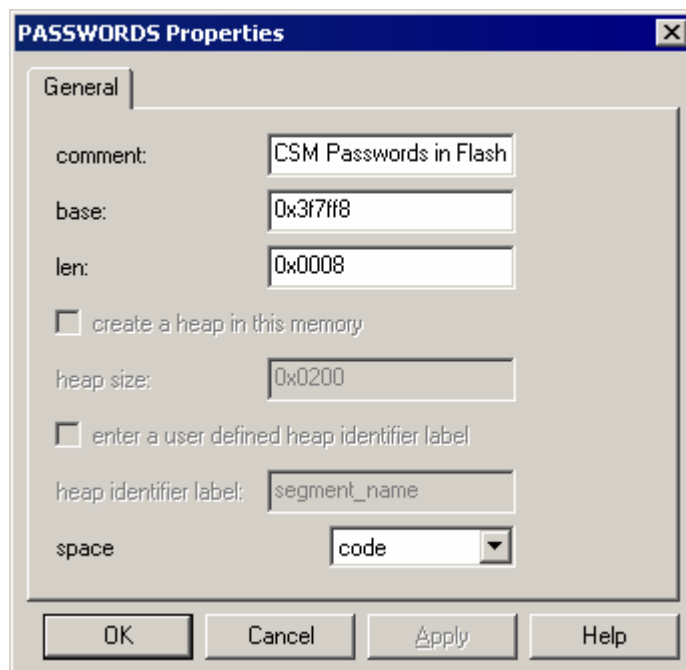


Figure 3. DSP/BIOS MEM Properties for CSM Password Locations

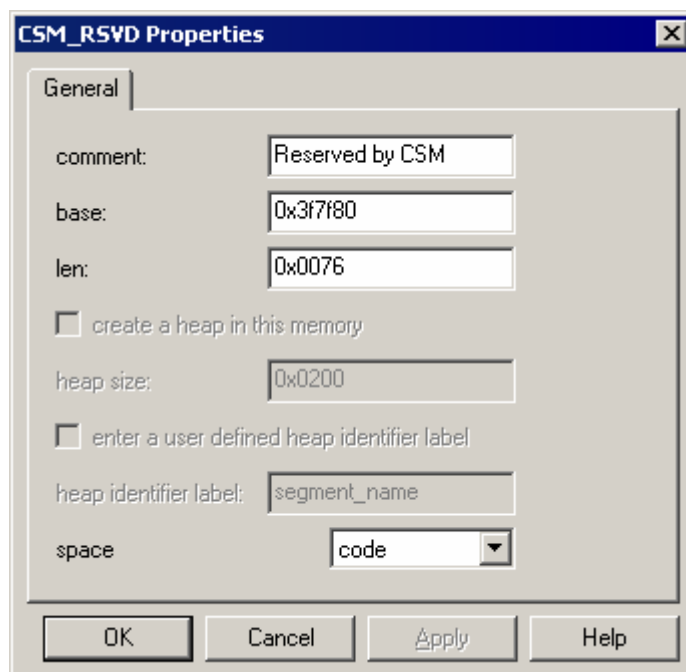


Figure 4. DSP/BIOS MEM Properties for CSM Reserved Locations

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

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

```

6 Executing Your Code from Flash after a DSP Reset

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

F28xx devices contain a ROM bootloader that can transfer code execution to the flash after a device reset. The ROM bootloader is detailed in references [7] and [8]. 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 current F28xx devices. For other F28xx 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:    > 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 current F28xx devices. For other devices, consult the datasheet to confirm the boot to flash target address.

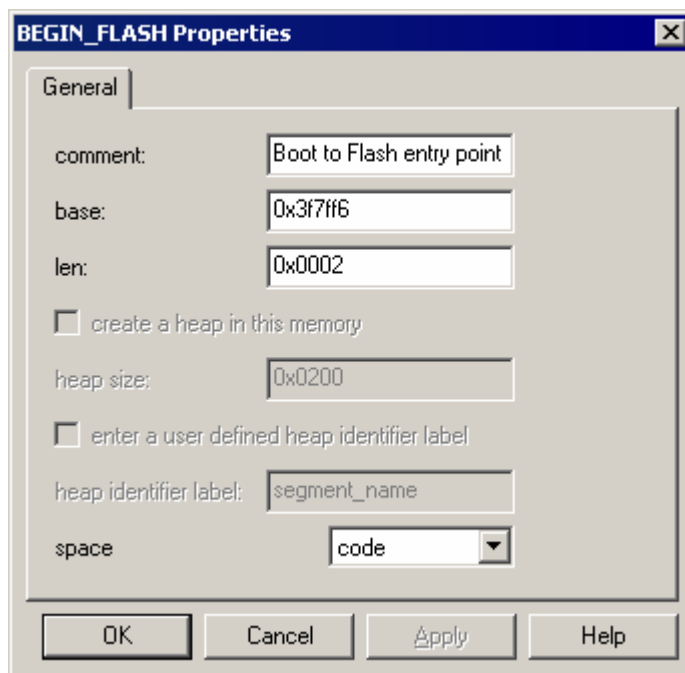


Figure 5. DSP/BIOS MEM Properties for Jump to Flash Entry Point

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:      > 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.

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: TMS320F2801, F2806, F2808, F2810, F2811, F2812
* 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 for different code projects for each of F281x and F280x devices 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 v3.1, C-compiler v4.1.1, and DSP/BIOS v5.20.**

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 F2808 and eZdsp F2812 development boards. However, they will also run on other F28xx board as follows:

F281x examples: These will run on any F2810, F2811, or F2812 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, 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.

F280x examples: These will run on any F2808 board. They can also be adapted to run on a F2806 or F2801 board 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 GPIOF34 pins as outputs.

The source code utilizes the DSP281x Header File structures v1.00 and the DSP280x Header File structures v1.20 for accessing peripheral registers on the F28xx devices. All needed files from the header file packages are included here. However, the user is encouraged to download the complete header files packages for additional information. These are available on the TI website, <http://www.ti.com> (see references [10] and [11]).

Each of the code projects perform the same functions:

- Illustrates F28xx DSP initialization. The PLL is configured for x5 operation.
- Enables the real-time emulation mode of Code Composer Studio.
- Toggles the GPIOF14 pin on the eZdsp F2812 board, or the GPIOF34 pin on eZdspF2808, to blink the LED on the 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 either the PWM1 pin (for F281x), or on the ePWM1A signal mapped to the GPIO0 pin (for F280x).
- Configures the (enhanced) capture unit #1. On F280x 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 and File Utilizations

The four code projects for each device 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
\cmd	Contains linker command files (.cmd files)
\DSP281x_headers\cmd or \DSP280x_headers\cmd	Contains the needed linker command files from the DSP281x Header File structures v1.00, or the DSP280x Header File structures v1.20.
\DSP281x_headers\include or \DSP280x_headers\include	Contains the needed include files from the DSP281x Header File structures v1.00, or the DSP280x Header File structures v1.20.
\include	Contains include files (.h files)
\projects	Contains the example projects (.cmd, .h, .pjt, and .tcf files)
\src	Contains common and non-common source code files (.c and .asm files)

Table 4. F281x Example Code File Inventory and Utilization

Filename	Project Utilization			
	example_nonBIOS_ram	example_nonBIOS_flash	example_BIOS_ram	example_BIOS_flash
eZdspF2812\cmd\2812_BIOS_flash.cmd				✓
eZdspF2812\cmd\2812_BIOS_ram.cmd			✓	
eZdspF2812\cmd\2812_nonBIOS_flash.cmd		✓		
eZdspF2812\cmd\2812_nonBIOS_ram.cmd	✓			
eZdspF2812\DSP281x_headers\cmd\DSP281x_Headers_BIOS.cmd ¹			✓	✓
eZdspF2812\DSP281x_headers\cmd\DSP281x_Headers_nonBIOS.cmd ¹	✓	✓		
eZdspF2812\DSP281x_headers\include\DSP281x_Adc.h ¹	✓	✓	✓	✓
eZdspF2812\DSP281x_headers\include\DSP281x_CpuTimers.h ¹	✓	✓	✓	✓
eZdspF2812\DSP281x_headers\include\DSP281x_DevEmu.h ¹	✓	✓	✓	✓
eZdspF2812\DSP281x_headers\include\DSP281x_Device.h ¹	✓	✓	✓	✓
eZdspF2812\DSP281x_headers\include\DSP281x_ECan.h ¹	✓	✓	✓	✓
eZdspF2812\DSP281x_headers\include\DSP281x_Ev.h ¹	✓	✓	✓	✓
eZdspF2812\DSP281x_headers\include\DSP281x_Gpio.h ¹	✓	✓	✓	✓
eZdspF2812\DSP281x_headers\include\DSP281x_Mcbsp.h ¹	✓	✓	✓	✓
eZdspF2812\DSP281x_headers\include\DSP281x_PieCtrl.h ¹	✓	✓	✓	✓
eZdspF2812\DSP281x_headers\include\DSP281x_PieVect.h ¹	✓	✓	✓ ²	✓ ²
eZdspF2812\DSP281x_headers\include\DSP281x_Sci.h ¹	✓	✓	✓	✓
eZdspF2812\DSP281x_headers\include\DSP281x_Spi.h ¹	✓	✓	✓	✓
eZdspF2812\DSP281x_headers\include\DSP281x_SysCtrl.h ¹	✓	✓	✓	✓
eZdspF2812\DSP281x_headers\include\DSP281x_Xintf.h ¹	✓	✓	✓	✓
eZdspF2812\DSP281x_headers\include\DSP281x_XIntrupt.h ¹	✓	✓	✓	✓
eZdspF2812\include\DSP281x_DefaultIsr.h ¹	✓	✓		
eZdspF2812\include\example_BIOS.h			✓	✓
eZdspF2812\include\example_nonBIOS.h	✓	✓		
eZdspF2812\projects\example_BIOS_flash.pjt				✓
eZdspF2812\projects\example_BIOS_flash.tcf				✓

eZdspF2812\projects\example_BIOS_flashcfg.cmd ³				✓
eZdspF2812\projects\example_BIOS_ram.pjt			✓	
eZdspF2812\projects\example_BIOS_ram.tcf			✓	
eZdspF2812\projects\example_BIOS_ramcfg.cmd ³			✓	
eZdspF2812\projects\example_nonBIOS_flash.pjt		✓		
eZdspF2812\projects\example_nonBIOS_ram.pjt	✓			
eZdspF2812\src\Adc.c	✓	✓	✓	✓
eZdspF2812\src\CodeStartBranch.asm	✓	✓	✓	✓
eZdspF2812\src\DefaultIsr_BIOS.c			✓	✓
eZdspF2812\src\DefaultIsr_nonBIOS.c	✓	✓		
eZdspF2812\src\DelayUs.asm	✓	✓	✓	✓
eZdspF2812\src\DSP281x_GlobalVariableDefs.c ¹	✓	✓	✓	✓
eZdspF2812\src\Ev.c	✓	✓	✓	✓
eZdspF2812\src\Gpio.c	✓	✓	✓	✓
eZdspF2812\src\main_BIOS.c			✓	✓
eZdspF2812\src\main_nonBIOS.c	✓	✓		
eZdspF2812\src\passwords.asm		✓		✓
eZdspF2812\src\PieCtrl_BIOS.c			✓	✓
eZdspF2812\src\PieCtrl_nonBIOS.c	✓	✓		
eZdspF2812\src\PieVect_nonBIOS.c	✓	✓		
eZdspF2812\src\SetDBGIER.asm	✓	✓	✓	✓
eZdspF2812\src\SysCtrl.c	✓	✓	✓	✓
eZdspF2812\src\Xintf.c	✓	✓	✓	✓
eZdspF2812\disclaimer.txt	Documentation file			
eZdspF2812\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, v1.00 (see reference [10]).

² Although *DSP281x_PieVect.h* is included into the flash projects, the structure *PieVectTable* that it defines (and which is linked over the PIEVECT 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. The files *example_BIOS_flash.tcf* and *example_BIOS_ram.tcf* contain everything Code Composer Studio needs to create these two .cmd files at project build time.

Table 5. F280x Example Code File Inventory and Utilization

Filename	Project Utilization			
	example_nonBIOS_ram	example_nonBIOS_flash	example_BIOS_ram	example_BIOS_flash
eZdspF2808\cmd\2808_BIOS_flash.cmd				✓
eZdspF2808\cmd\2808_BIOS_ram.cmd			✓	
eZdspF2808\cmd\2808_nonBIOS_flash.cmd		✓		
eZdspF2808\cmd\2808_nonBIOS_ram.cmd	✓			
eZdspF2808\DSP280x_headers\cmd\DSP280x_Headers_BIOS.cmd ¹			✓	✓
eZdspF2808\DSP280x_headers\cmd\DSP280x_Headers_nonBIOS.cmd ¹	✓	✓		
eZdspF2808\DSP280x_headers\include\DSP280x_Adc.h ¹	✓	✓	✓	✓
eZdspF2808\DSP280x_headers\include\DSP280x_CpuTimers.h ¹	✓	✓	✓	✓
eZdspF2808\DSP280x_headers\include\DSP280x_DevEmu.h ¹	✓	✓	✓	✓
eZdspF2808\DSP280x_headers\include\DSP280x_Device.h ¹	✓	✓	✓	✓
eZdspF2808\DSP280x_headers\include\DSP280x_ECan.h ¹	✓	✓	✓	✓
eZdspF2808\DSP280x_headers\include\DSP280x_ECap.h ¹	✓	✓	✓	✓
eZdspF2808\DSP280x_headers\include\DSP280x_EPwm.h ¹	✓	✓	✓	✓
eZdspF2808\DSP280x_headers\include\DSP280x_EQep.h ¹	✓	✓	✓	✓
eZdspF2808\DSP280x_headers\include\DSP280x_Gpio.h ¹	✓	✓	✓	✓
eZdspF2808\DSP280x_headers\include\DSP280x_I2c.h ¹	✓	✓	✓	✓
eZdspF2808\DSP280x_headers\include\DSP280x_PieCtrl.h ¹	✓	✓	✓	✓
eZdspF2808\DSP280x_headers\include\DSP280x_PieVect.h ¹	✓	✓	✓ ²	✓ ²
eZdspF2808\DSP280x_headers\include\DSP280x_Sci.h ¹	✓	✓	✓	✓
eZdspF2808\DSP280x_headers\include\DSP280x_Spi.h ¹	✓	✓	✓	✓
eZdspF2808\DSP280x_headers\include\DSP280x_SysCtrl.h ¹	✓	✓	✓	✓
eZdspF2808\DSP280x_headers\include\DSP280x_XIntrupt.h ¹	✓	✓	✓	✓
eZdspF2808\include\DSP280x_DefaultIsr.h ¹	✓	✓		
eZdspF2808\include\example_BIOS.h			✓	✓
eZdspF2808\include\example_nonBIOS.h	✓	✓		
eZdspF2808\projects\example_BIOS_flash.pjt				✓
eZdspF2808\projects\example_BIOS_flash.tcf				✓

eZdspF2808\projects\example_BIOS_flashcfg.cmd ³				✓
eZdspF2808\projects\example_BIOS_ram.pjt			✓	
eZdspF2808\projects\example_BIOS_ram.tcf			✓	
eZdspF2808\projects\example_BIOS_ramcfg.cmd ³			✓	
eZdspF2808\projects\example_nonBIOS_flash.pjt		✓		
eZdspF2808\projects\example_nonBIOS_ram.pjt	✓			
eZdspF2808\src\Adc.c	✓	✓	✓	✓
eZdspF2808\src\CodeStartBranch.asm	✓	✓	✓	✓
eZdspF2808\src\DefaultIsr_BIOS.c			✓	✓
eZdspF2808\src\DefaultIsr_nonBIOS.c	✓	✓		
eZdspF2808\src\DelayUs.asm	✓	✓	✓	✓
eZdspF2808\src\DSP280x_GlobalVariableDefs.c ¹	✓	✓	✓	✓
eZdspF2808\src\ECap.c	✓	✓	✓	✓
eZdspF2808\src\EPwm.c	✓	✓	✓	✓
eZdspF2808\src\Gpio.c	✓	✓	✓	✓
eZdspF2808\src\main_BIOS.c			✓	✓
eZdspF2808\src\main_nonBIOS.c	✓	✓		
eZdspF2808\src\passwords.asm		✓		✓
eZdspF2808\src\PieCtrl_BIOS.c			✓	✓
eZdspF2808\src\PieCtrl_nonBIOS.c	✓	✓		
eZdspF2808\src\PieVect_nonBIOS.c	✓	✓		
eZdspF2808\src\SetDBGIER.asm	✓	✓	✓	✓
eZdspF2808\src\SysCtrl.c	✓	✓	✓	✓
eZdspF2808\disclaimer.txt	Documentation file			
eZdspF2808\readme.txt	Documentation file			

Table 5 Notes:

¹ This file is identical to the file of the same name found in the \v120\DSP280x-Headers subdirectory of the DSP280x Header File structures, v1.20 (see reference [11]).

² Although *DSP280x_PieVect.h* is included into the flash projects, the structure *PieVectTable* that it defines (and which is linked over the PIEVECT 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. The files *example_BIOS_flash.tcf* and *example_BIOS_ram.tcf* 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.

IMPORTANT:

For the flash projects, the .out file CANNOT simply be loaded in the DSP using File->Load_Project in Code Composer Studio. The flash memory must be PROGRAMMED using a flash programming utility. One such utility is the C2000 Flash Programming Plugin tool for Code Composer Studio (available from the TI website, www.ti.com).

2a) If using the RAM examples, the eZdsp board should be configured for "Jump to H0" (F2812) or "Jump to M0" (F2808) bootmode. Check the board jumpers/dip-switch to be:

eZdsp F2812: 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)

eZdsp F2808: JP9 1-2 (PLL)

DIP SW1: 1 = ON

2 = OFF

3 = ON

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.

2b) If using the FLASH examples, the eZdsp board should be configured for "Jump to Flash" bootmode. Check the board jumpers/dip-switches to be:

eZdsp F2812: 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)

eZdsp F2808: JP9 1-2 (PLL)

DIP SW1: 1 = OFF

2 = OFF

3 = OFF

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.

3) 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.

4) 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. 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).

5) 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 for F2812, and ECAP1INT for F2808, 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.

6) For DSP/BIOS projects, a complete set of (hardware) interrupt service routines are defined in the file *DefaultIsr_BIOS.c*. Each ISR is hooked to the desired interrupt using the HWI manager in the DSP/BIOS configuration tool. 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 CAPINT1 ISR for F2812 and the ECAP1INT ISR for F2808 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 and ECAP1INT 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.

References

1. TMS320F2810, TMS320F2811, TMS320F2812, TMS320C2810, TMS320C2811, TMS320C2812 Digital Signal Processors Data Manual (SPRS174)
2. TMS320F2808, TMS320F2806, TMS320F2801, UCD9510 Digital Signal Processors Data Manual (SPRS230)
3. TMS320C28x Optimizing C/C++ Compiler User's Guide (SPRU514)
4. TMS320x281x System Control and Interrupts Peripheral Reference Guide (SPRU078)
5. TMS320x280x System Control and Interrupts Peripheral Reference Guide (SPRU712)
6. TMS320C28x Assembly Language Tools User's Guide (SPRU513)
7. TMS320x281x DSP Boot ROM Reference Guide (SPRU095)
8. TMS320x280x DSP Boot ROM Reference Guide (SPRU722)
9. TMS320C28x DSP CPU and Instruction Set Reference Guide (SPRU430)
10. C281x C/C++ Header Files and Peripheral Example Code (SPRC097)
11. C280x C/C++ Header Files and Peripheral Example Code (SPRC191)

Revision History

Revision	Date	Who	Description of Changes from Previous Version
SPRA958F	Feb. 2006	D. Alter	<ul style="list-style-type: none"> - Changed the following in the code examples: <ul style="list-style-type: none"> * Fixed DelayUs.asm to properly preserve the EALLOW bit setting. * Edited project build option search and file paths to use \$(Proj_dir) path symbol. This makes the projects more portable. * Modified the directory structure to use \cmd and \include sub-directories. * In file ECap.c, changed ECCTL2 register bits 2-1 from 00b to 11b in order to support F2808/6/1 rev. A and later silicon. Also changed REARM field value from 1 to 0. The function works with REARM set either way, but a 0 value is more appropriate here. (F280x only) * Fixed header in DelayUs.asm for F281x that referenced F280x devices. (F281x only) - Modified Tables 4 and 5 in the appnote to reflect the modified file directory structure.
SPRA958E	Sept. 2005	D. Alter	<ul style="list-style-type: none"> - Added support for F280x devices. - Removed lower 64Kw restriction on .const, .bss, and .system from Tables 1 and 2. Added lower 64Kw restriction on .stack to Table 2. - Simplifier Section 4.6.2 methodology (Running a Specific Constant Array from RAM). - Updated all code examples to use CCS v3.1 and DSP/BIOS v5.20. - Changed the following in the F281x code examples: <ul style="list-style-type: none"> * Corrected OTP length in F2812_nonBIOS_RAM.cmd and F2812_nonBIOS_Flash.cmd (from 0x800 to 0x400). * Removed unused section pie_vect from F2812_nonBIOS_Flash.cmd. * Changed ADC power-up delays to 10 ms and 1 ms (from 7 ms and 20 us) to match worst case values in F281x datasheet. * Changed SCSR init in SysCtrl.c so that WD generates a reset instead of an interrupt. * Changed OTPWAIT value to 8 (from 5) in SysCtrl.c. * Made linkage and memory names consistent with the BIOS examples in example_nonBIOS_RAM.cmd and example_nonBIOS_Flash.cmd. * Added WD servicing to PLL lock wait loop in SysCtrl.c and inner loop of DelayUs() in DelayUs.asm.
SPRA958D	Sept. 2004	D. Alter	<p>Code Corrections only. The appnote text itself did not change.</p> <ul style="list-style-type: none"> - 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	<p>Code Corrections only. The appnote text itself did not change.</p> <ul style="list-style-type: none"> - 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 acknowledge 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: <pre>/** Copy all FLASH sections that need to run from RAM (use far_memcpy() from RTS library) */</pre> to: <pre>/** Copy all FLASH sections that need to run from RAM (use memcpy() from RTS library) */</pre>
SPRA958B	Jan. 2004	D. Alter	<ul style="list-style-type: none"> - Deleted "+ 1" length for examples of memcpy() function calls. For example, previously: <pre>memcpy(&econst_runstart, &econst_loadstart, &econst_loadend - &econst_loadstart + 1);</pre> Is now: <pre>memcpy(&econst_runstart, &econst_loadstart, &econst_loadend - &econst_loadstart);</pre> - 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	<ul style="list-style-type: none"> - 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	<ul style="list-style-type: none"> - Original

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

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

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

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

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		Applications	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments
Post Office Box 655303 Dallas, Texas 75265