Creating a Second-Level Bootloader for FLASH Bootloading on TMS320C6000 Platform With Code Composer Studio

Kimberly Daniel
Shivashankar Gangadhar
George Mock
Alan Campbell

Digital Signal Processing Solutions

ABSTRACT

In many DSP applications there is a need to copy code and/or data from one location to another at boot. C6000 DSPs offer three types of boot configurations: no boot process, ROM boot process, and host boot process. The most commonly selected boot configuration is the ROM boot process. When ROM boot is selected as the boot configuration, 1K byte (on C621x/C671x/C64x™) of code will automatically be copied from CE1 to address 0 by the EDMA following the release of /RESET while the CPU is stalled.

DSP applications are not limited to 1K byte of code. In the event the application size exceeds 1K byte, a custom boot routine will need to be developed to copy the additional sections of code not copied by the ROM boot. The custom boot routine is referred to as the second level bootload, or the secondary bootloader. This application note describes how to create a secondary bootloader by converting a RAM-based application to a flash-based application. This was done by migrating a C6000 DSP-based DSP/BIOS application developed on the Code Composer Studio development environment to an actual embedded product. This application note will use DSP/BIOS Reference Framework Level 3 (RF3) example to illustrate flash booting on a dsk6713 board. The appendix of this application note also provides an example of a secondary bootloader for a non-BIOS application. Code for both the DSP/BIOS and non-BIOS examples are available for download with this application note.

This application report contains project code that can be downloaded from this link.
http://www.−s.ti.com/sc/psheets/spra999a1/spraa999a1.zip

Contents

List of Figures

List of Tables

1 Introduction

In many DSP applications there is a need to copy code and/or data from one location to another at boot. C6000 DSPs offer three types of boot configurations: no boot process, ROM boot process, and host boot process. The boot process that is selected is determined by the configuration of the BOOTMODE pins. Refer to the device specific data sheet to learn about the boot modes supported by a particular device and configuring the device for a particular boot mode.

The most commonly selected boot configuration is the ROM boot process (also referred to as the on-chip bootloader in this document). When selected, the ROM boot process copies a fixed amount of memory located at the beginning of the external ROM to address 0 using the DMA/EDMA controller. The transfer is automatically completed as a single frame block transfer from ROM to address 0. This transfer occurs when the device is released from external reset while the CPU is internally stalled. Upon completion of the block transfer the CPU is released from the stalled state and starts executing from address 0.

The ROM boot process differs between specific C6000 devices.

- 620x/670x DMA copies 64K bytes from CE1 to address 0
- 621x/671x/64x EDMA copies 1K bytes from beginning of CE1 to address 0.

Application size determines whether the on-chip bootload facility is sufficient or whether there is a need for a secondary bootloader. If the application size is less than the size copied by the ROM boot, then a secondary bootloader (custom boot code) is not required. Typically, 621x/671x/64x applications need a secondary bootloader because the application size is greater than the 1K bytes of memory copied by the on-chip bootloader.

In the applications that require a secondary bootloader, this custom boot code usually resides at the beginning of ROM memory so that it can be automatically transferred by the on-chip bootloader to internal memory, address 0. Once the transfer is complete, the CPU begins executing from address 0 and therefore runs the custom boot code. The secondary bootloader then copies the rest of the application into RAM. Figure 1 shows the sequence of events that occur when the application is bootloaded using a secondary bootloader.

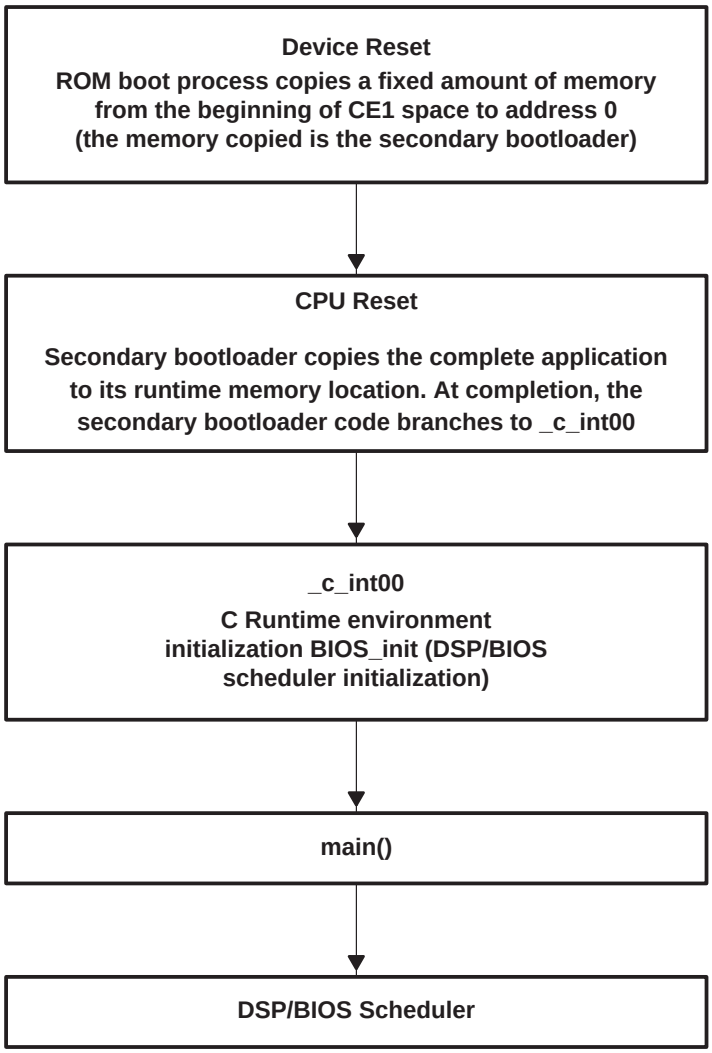

Figure 1. Start-Up Sequence of Application which Uses Secondary Bootloader

1.1 Second-Level Bootload Considerations

As discussed above, ROM booting is the preferred boot method for many of the C6000 applications. If the ROM boot mode is selected and a secondary bootloader is required, then several things must be considered:

- COFF section placement
- Building the application
- Writing the custom boot code (secondary bootloader)
- Burning the application into flash

1.2 COFF Section Placement

A COFF section is a block of code or data that occupies contiguous space in the memory map. COFF sections are of three types, namely code, initialized data, and uninitialized data. Each COFF section has a load and a run attribute. The load attribute of a section tells the loader or Flash Burn Utility where to place the section following the project build. The run attribute of a section specifies where the section will execute from when the device comes out of RESET. Therefore, if different load and run addresses are specified for a section, the section must be copied from the load address to the run address by the secondary bootloader.

Consider the example below:

```
.text:  LOAD = FLASH, RUN = IRAM
```

In the above case, .text is placed in flash and then copied to IRAM by the secondary bootloader during boot. All references to the .text section in the program application refer to its run address of IRAM. Each section with a load address in flash ROM forms part of the load image. For ROM booting, the load image consists of sections of code and initialized data.

Determination of the run address of a section is based on how frequently the CPU accesses that particular section. If a section is accessed only once by the CPU then typically it will not have a run address in RAM. This will save RAM space for other purposes. For example, the .cinit section which is accessed only once during boot, typically will have a load as well as a run address in flash. Sections that need faster access by the CPU will have a run address in RAM. The secondary bootloader is responsible for copying all the sections from their load space to their run space if a particular section has different load and run addresses. All the un-initialized data sections are placed in RAM and these sections will have the same load and run address.

Refer to the *TMS320C6000 Assembly Language Tools User's Guide* (SPRU186) for more details on COFF section description and load run specification. Table 2 gives a list of DSP/BIOS sections and suggested memory placement.

1.3 Building and Linking the Application

After the project build, the linker will generate a map file that contains the section link information that can be used to determine the location in memory where each section was placed. The map file contains information such as the size of the section, the load address, and the run address. An excerpt from a map file will look similar to the following:

```
.bios

90005600 00001f00 RUN ADDR = 800063e0
90005600 00000700 biosi.a62 : swi.o62 (.bios)
90005d00 00000540 lnkrtdx.a62 : rtdx.o62 (.bios)
90006240 00000400 biosi.a62 : hwi_disp_asm.o (.bios)
90006640 00000300 : prd.o62 (.bios)
90006940 00000280 : rta.o62 (.bios)
```

This says that the .bios section is 0x00001f00 bytes long, has a load address of 0x90005600, and a run address of 0x800063e0. This information is needed to create the custom boot code. The map file also contains detailed information about memory sections, sub-sections, and symbols.

1.4 Writing the Custom Boot Code

Once the project has been built and linked, the custom boot code should be written. The custom boot routine is typically written in assembly language because the C environment is not initialized at boot-time. The following is a list of tasks usually performed by the custom boot code:

1. Configure the PLL. This step is recommended for C6x devices that have a software programmable PLL in order to improve boot performance.
2. Configure the EMIF to access external memory.
3. Copy the initialized sections from the ROM to the memory location specified by the section's run address.
4. Call _c_int00().

Once the custom boot code is complete, it should be included in the project and the project should be re-built. Section 2.3.1 contains sample secondary boot code.

1.5 Burning the Application into Flash

Applications built in Code Composer Studio will be of COFF format (.out). Flash burn utilities (ROM programmers) typically accept the file in ASCII hexadecimal format, therefore the application COFF format output executables need to be converted into .hex format using a hex conversion utility before burning the flash. This can be accomplished by using the hex conversion utility that is provided with Code Composer Studio.

2 Bootloading a DSP/BIOS Application

The bootloading process involved for a DSP/BIOS application can be broadly separated into the following steps:

1. DSP/BIOS memory configuration for ROM booting
2. Building the application
3. Writing the custom boot code
4. Burning the application into flash

This application note provides a sample project that includes the necessary changes to RF3 to ROM boot on the DSK6713. The appendix of this application note also provides an example of a secondary bootloader for a non-BIOS application. The code for both the DSP/BIOS and non−BIOS examples are available for download with this application note.

2.1 DSP/BIOS Memory Configuration for ROM Booting

2.1.1 Defining Memory Segments

Additional memory segments, such as FLASH_BOOT and FLASH_REST below, should be defined to specify two locations in ROM. These memory segments are required to distinguish between the memory sections that will automatically be copied by the on-chip bootloader into RAM following reset and those sections that must be copied by the secondary bootloader.

To create additional memory segments, open the DSP/BIOS configuration (.cdb) file, right-click on the MEM − Memory Section Manager object, and choose Insert MEM. For example, on the 6713 DSK, create these additional segments:

```
FLASH_BOOT: origin = 0x90000000, length = 0x400
FLASH_REST: origin = 0x90000400, length = 0x1FC00
BOOT_RAM:   origin = 0x0, length = 0x400
```

The FLASH_BOOT segment holds the secondary bootloader code. On device reset, contents of the FLASH_BOOT segment are copied into the BOOT_RAM segment by the on-chip boot facility. When creating the BOOT_RAM section for the 2nd level bootloader code, ensure that the box labeled “create a heap in this memory,” is unchecked and space attribute is code/data. FLASH_REST is used to store all other memory sections apart from the secondary bootloader code, which have a load address in flash. Complete memory sections defined using GCONF will look like Table 1.

Table 1. Memory Section Definitions

	Segment Name	Base	Length
Flash memory split into two segments	FLASH_BOOT	0x90000000	0x400
	FLASH_REST	0x90000400	0x1fc00
Internal SRAM (L2) split into two segments	BOOT_RAM	0x0	0x400
	IRAM	0x400	0xfc00
	SDRAM	0x80000000	0x1000000

2.1.2 Memory (COFF) Section Placement

As explained in the section 1.2, all the code and initialized data sections should have load addresses in Flash and depending on the application requirement these sections could have run address in RAM. For uninitialized data sections the only significant address in the run address in RAM.

Table 2 provides a list of the DSP/BIOS sections, as well as the compiler sections, and suggested memory placement.

Table 2. DSP/BIOS Sections and Suggested Memory Placement

Section Name	Section	Description	Suggested Placement
.args	Uninitialized data	Argument buffer	LOAD, RUN = RAM
.stack	Uninitialized data	Stack space	LOAD, RUN = RAM
.bios	Code	DSP/BIOS code	LOAD = ROM, RUN = RAM
.sysinit	Code	Init code, run only during startup	LOAD, RUN = ROM
.gblinit	Initialized data	Init table, used only during startup	LOAD, RUN= ROM
.trcdata	Initialized data	Trace mask selection, must have a run address in RAM	LOAD =ROM, RUN = RAM
.sysdata	Uninitialized data	Kernel data	LOAD, RUN = RAM
.hwi_vec	Code	Interrupt vector table	LOAD = ROM, RUN= RAM
.rtdx_text	Code	Code	LOAD=ROM, RUN=RAM
All other BIOS sections	Uninitialized data	Object memory, etc.	LOAD, RUN = RAM
Compiler Sections			
.const	Initialized data	Constant data	LOAD, RUN=ROM
.text	Code	Program code	LOAD = ROM, RUN = RAM
.data .cio.	Uninitialized data	Miscellaneous data sections	LOAD, RUN = RAM
.cinit, .pinit, .switch	Initialized data	C variable initialization tables	LOAD, RUN=ROM
.bss, .far	Uninitialized data	C variables	LOAD, RUN=RAM

Based on the above table the user can decide on load/run specifications for each section and configure them through the DSP/BIOS memory manager. Also, under the Memory Section Manager, turn off Reuse startup code space. This checkbox will be highlighted if .sysinit section is placed in data segment. Since the .sysinit section will be placed in flash memory, this memory cannot be reused at run time for data storage.

After deciding upon the sections that require different load/run specifications, open the DSP/BIOS configuration cdb and choose the Memory Section Manager. Under the Memory Section Manager Load Address Tab, check the box that specifies separate load addresses and choose the FLASH_REST memory section as the load address for all of the memory sections with a load address in ROM. Figure 2 shows the way this is done using GCONF interface in Code Composer Studio.

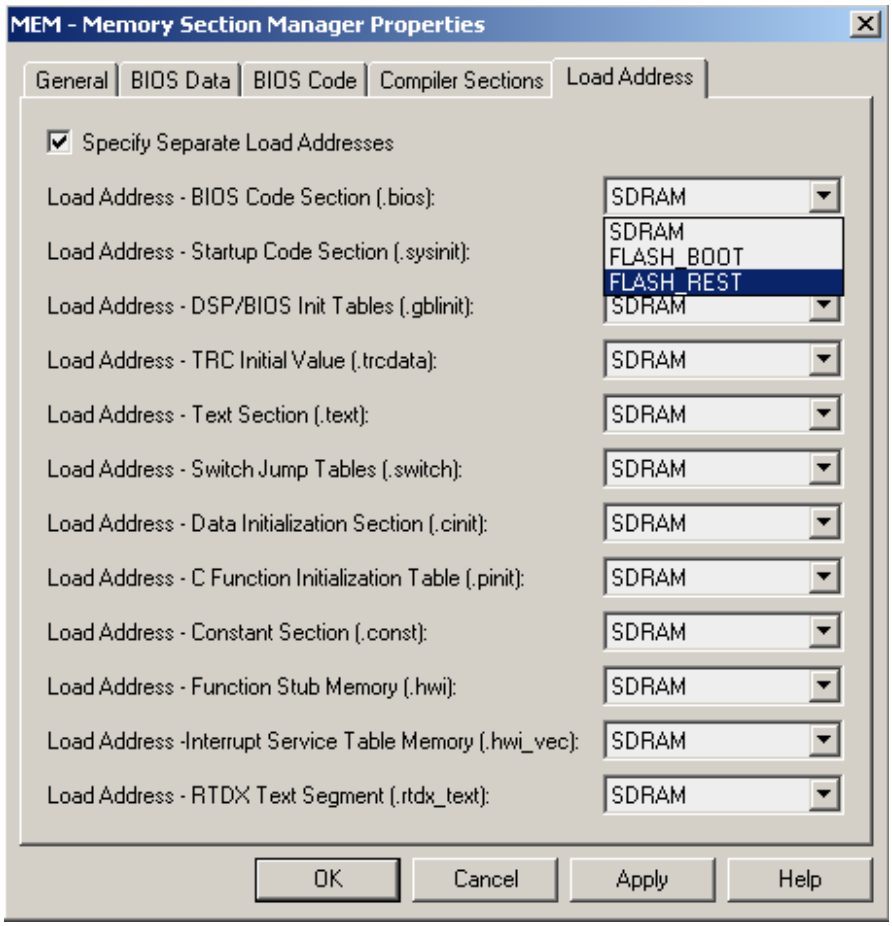

Figure 2. Load/Run Address Specification Using DSP/BIOS GCONF Interface

In addition to the memory sections described above, a new memory section .boot_load is created which holds the secondary bootloader code. This section is needed to ensure the custom boot code is placed properly in the FLASH_BOOT segment. To create the .boot_load section, a new linker command file must be created. This procedure is described in section 2.2.

```
.boot_load: {} load = FLASH_BOOT, run = BOOT_RAM
```

This code will give the section .boot_load a load address in FLASH_BOOT and a run address in BOOT_RAM. On device reset the custom boot code is copied from FLASH_BOOT to BOOT_RAM automatically by the on-chip boot facility. If multiple sections are placed in FLASH_BOOT, make sure the line linking the section for the secondary boot code appears before the lines for the other sections. This ensures the secondary boot code will be linked to the first address in ROM.

2.2 Building the Application

2.2.1 Build Options

Applications should be built with the run time auto initialization model. From the Code Composer Studio menu bar, select Project → Build Options and click the linker tab. Choose auto init model as Run-time Autoinitialization (−c), this will ensure that global variables are initialized at run time on startup using cinit records.

The secondary bootloader code obtains the information regarding the sections to be copied from load address to run address through a set of table entries. These table entries can be created by looking at the map file. To configure the linker to generate a map file, select Project → Build Options → Linker tab from the Code Composer Studio menu bar.

2.2.2 Linker Command File

In order to link the new memory section .boot_load, which holds the secondary bootloader code, a new linker command file must be created. An excerpt from the new linker command file is shown below. The new linker command file includes the BIOS generated linker command file.

```
-l app.cmd /*DSP/BIOS generated cmd file from cdb*/
SECTIONS {
       .boot_load : LOAD = FLASH_BOOT, RUN = BOOT_RAM
}
```

2.3 Writing the Secondary Bootloader

The secondary bootloader (custom boot code) becomes necessary when the amount of memory copied by the built in bootload mechanism is not sufficient for big applications. On the C6713 DSK, the external memory interface (EMIF) needs to be correctly programmed to enable access to external memory. Once this is done, the custom boot code should copy initialized data sections from their load addresses to their run addresses, and then branch to _c_int00, the usual program entry point.

Figure 3 provides *sample* code that uses the CPU instructions to copy certain sections into RAM. The source address, destination address, and size of all sections to copy are stored in the copy table. Also refer to boot_C671x.s62 in app.pjt, which has this sample code. The boot_C671x.s62 code is generic for any C671x device. A device-specific .asm file, c6713_emif.s62, is included in the project to define the addresses and values of the EMIF registers specifically for the C6713 DSK.

```
;  ======== boot_c671x.s62 ========

;

            .title  "Flash bootup utility"
            .option D,T
            .length 102
            .width  140
; global EMIF symbols defined for the c671x family
            .include        boot_c671x.h62
            .sect ".boot_load"
            .global _boot
_boot:
;***************************************************************************
;* DEBUG LOOP -  COMMENT OUT B FOR NORMAL OPERATION
;***************************************************************************
zero B1
_myloop:  ; [!B1] B _myloop
            nop  5
_myloopend: nop
;***************************************************************************
;* CONFIGURE EMIF
;***************************************************************************
        ;*************************************************************
        ; *EMIF_GCTL = EMIF_GCTL_V;
        ;*************************************************************
            mvkl  EMIF_GCTL,A4
      ||    mvkl  EMIF_GCTL_V,B4
            mvkh  EMIF_GCTL,A4
      ||    mvkh  EMIF_GCTL_V,B4
            stw   B4,*A4
        ;*************************************************************
        ; *EMIF_CE0 = EMIF_CE0_V
        ;*************************************************************
            mvkl  EMIF_CE0,A4
      ||    mvkl  EMIF_CE0_V,B4
            mvkh  EMIF_CE0,A4
      ||    mvkh  EMIF_CE0_V,B4
            stw   B4,*A4
```

Figure 3. Boot Code

```
  ;*************************************************************
  ; *EMIF_CE1 = EMIF_CE1_V (setup for 8-bit async)
  ;*************************************************************
      mvkl  EMIF_CE1,A4
||    mvkl  EMIF_CE1_V,B4
      mvkh  EMIF_CE1,A4
||    mvkh  EMIF_CE1_V,B4
      stw   B4,*A4
  ;*************************************************************
  ; *EMIF_CE2 = EMIF_CE2_V (setup for 32-bit async)
  ;*************************************************************
      mvkl  EMIF_CE2,A4
||    mvkl  EMIF_CE2_V,B4
      mvkh  EMIF_CE2,A4
||    mvkh  EMIF_CE2_V,B4
      stw   B4,*A4
  ;*************************************************************
  ; *EMIF_CE3 = EMIF_CE3_V (setup for 32-bit async)
  ;*************************************************************
||    mvkl  EMIF_CE3,A4
||    mvkl  EMIF_CE3_V,B4     ;
      mvkh  EMIF_CE3,A4
||    mvkh  EMIF_CE3_V,B4
      stw   B4,*A4
  ;*************************************************************
  ; *EMIF_SDRAMCTL = EMIF_SDRAMCTL_V
  ;*************************************************************
||    mvkl  EMIF_SDRAMCTL,A4
||    mvkl  EMIF_SDRAMCTL_V,B4    ;
      mvkh  EMIF_SDRAMCTL,A4
||    mvkh  EMIF_SDRAMCTL_V,B4
      stw   B4,*A4
  ;*************************************************************
  ; *EMIF_SDRAMTIM = EMIF_SDRAMTIM_V
  ;*************************************************************
||    mvkl  EMIF_SDRAMTIM,A4
||    mvkl  EMIF_SDRAMTIM_V,B4    ;
      mvkh  EMIF_SDRAMTIM,A4
||    mvkh  EMIF_SDRAMTIM_V,B4
      stw   B4,*A4
```

Figure 3. Boot Code (Continued)

```
        ;*************************************************************
        ; *EMIF_SDRAMEXT = EMIF_SDRAMEXT_V
        ;*************************************************************
      ||    mvkl  EMIF_SDRAMEXT,A4
      ||    mvkl  EMIF_SDRAMEXT_V,B4    ;
            mvkh  EMIF_SDRAMEXT,A4
      ||    mvkh  EMIF_SDRAMEXT_V,B4
            stw   B4,*A4
;*****************************************************************************
; copy sections
;*****************************************************************************
        mvkl  COPY_TABLE, a3 ; load table pointer
        mvkh  COPY_TABLE, a3
        ldw   *a3++, b1      ; Load entry point
copy_section_top:
        ldw   *a3++, b0      ; byte count
        ldw   *a3++, a4      ; ram start address
        nop   3
 [!b0]  b copy_done          ; have we copied all sections?
        nop   5
copy_loop:
        ldb   *a3++,b5
        sub   b0,1,b0        ; decrement counter
 [ b0]  b     copy_loop      ; setup branch if not done
 [!b0]  b     copy_section_top
        zero  a1
 [!b0]  and   3,a3,a1
        stb   b5,*a4++
 [!b0]  and   -4,a3,a5       ; round address up to next multiple of 4
 [ a1]  add   4,a5,a3        ; round address up to next multiple of 4
;*****************************************************************************
; jump to entry point
;*****************************************************************************
copy_done:
        b    .S2 b1
        nop   5
```

Figure 3. Boot Code (Continued)

2.3.1 The Section Copy Table

The secondary bootloader copies the contents of memory sections from its load address to its run address using a section copy table. Table 3 shows the typical copy table format. The copy table contains entries for all the memory sections that need to be copied from their load address to their run address. Each table entry contains information describing the size of the section of memory, the destination address or address from where the section will execute, and the source address or the address where the section was loaded.

There are a number of ways to create the section copy table:

- Inspecting the map file
- Using the –boot option in the hex conversion utility
- Using linker options (LOAD_START, RUN_START, SIZE)

The following sections discuss two ways (inspecting the map file and using the hex conversion utility) to create the section copy table for a DSP/BIOS application. The third method, using the linker options mentioned above, would require significant modifications to the BIOS-generated linker command file. Hence this option is not presently recommended with BIOS applications and is discussed in the appendix of this document for a non-BIOS application.

Table 3. Copy Table Format

Size of memory section1
Destination address for section1
Source address for section 1
Size of memory section2
Destination address for section2
–
0 ; End of Table
0 ; End of Table
0 ; End of Table

2.3.1.1 Creating the Section Copy Table By Inspecting the Map File

Each copy table entry can be determined by inspecting the map file. If this method is chosen to create the copy table, then the appropriate value of the size of the section, the destination address of the section, and the source address of the section must be manually filled in for that entry. This also means that each time the project is compiled, the map file must be inspected again and the copy table updated with any changes. A copy table entry corresponding to the map file shown in section 1.3 is shown below. The copy table is inserted at the end of the custom boot code.

```
COPY_TABLE:
               ;;size
               ;;destination (run address)
               ;;source  (load address)
               ;;bios
               .word 0x00001f00
               .word 0x800063e0
               .word 0x90005600
```

2.3.1.2 Creating the Section Copy Table Using the Hex Conversion Utility

As described above, the second-level bootloader uses a copy table to transfer sections of memory from its load address to its run address. Manually filling in the copy table entries by inspecting the map file is a tedious activity and prone to error. The hex conversion utility (hex6x v4.3.3 and later) provided with Code Composer Studio provides a more convenient method for creating the section copy table by automatically building the copy table when the appropriate options are specified in the hex conversion utility command file. The Code Composer Studio project described above must be updated to use the hex conversion utility to generate the copy table and the changes necessary are described in the following paragraphs.

The hex conversion utility will take care of placing the memory sections in the appropriate locations in ROM so it is no longer necessary to create additional memory segments such as FLASH_BOOT and FLASH_REST or to specify separate load addresses.

Therefore, the first step in converting the project to use the hex conversion utility to build the copy table is to update the BIOS generated configuration (*.cdb) file. The FLASH_BOOT section and FLASH_REST section should be removed by opening the DSP/BIOS configuration (.cdb) file, right-clicking on the MEM − Memory Section Manager object, and then choosing Delete. Also, the Memory Section Manager properties should be updated by selecting the Load Address tab and then removing the check from the Specify Separate Load Addresses box.

Furthermore, the second step in updating the project is changing the user defined linker command file. The user defined linker command file should be updated to contain the following section definition.

```
-l app.cmd  /*DSP/BIOS generated cmd file from cdb*/
SECTIONS {
.boot_load :> BOOT_RAM
}
```

The next step is to update the custom boot code by removing the copy table from the end of the file. This piece of code is no longer needed because the copy table will automatically be generated by the hex conversion utility. Also, the location of the copy table must be defined in the custom boot code. This is done with the following piece of code.

```
;Address of the generated copy table
COPY_TABLE   .equ     0x90000400
```

Finally, the –boot, −bootorg, and –bootsection options should be added to the hex conversion utility command file. This is described in section 2.4.1.

2.3.1.3 Creating the Section Copy Table With the Linker Table Directive

The linker bundled with Code Composer Studio 3.00 introduces a new feature for creating copy tables that is simpler and more flexible than the methods mentioned in the previous two sections. The new feature is called the table directive. The table directive is described in section 3. An example using the table directive is described in Appendix B.

2.4 Programming Flash

Code Composer Studio comes with several utilities that help with flashing applications into ROM. For C6x architectures a flash programmer and a hex conversion utility are provided. Flash programmers work only with the hex format hence the .out (COFF format) obtained from Code Composer Studio must be converted to .hex through the hex conversion utility.

The following procedure describes how to create an application to program into flash:

1. Build the project to generate the .out file
2. Use the hex conversion utility to create a .hex file from the .out file
3. Program the flash with the .hex file

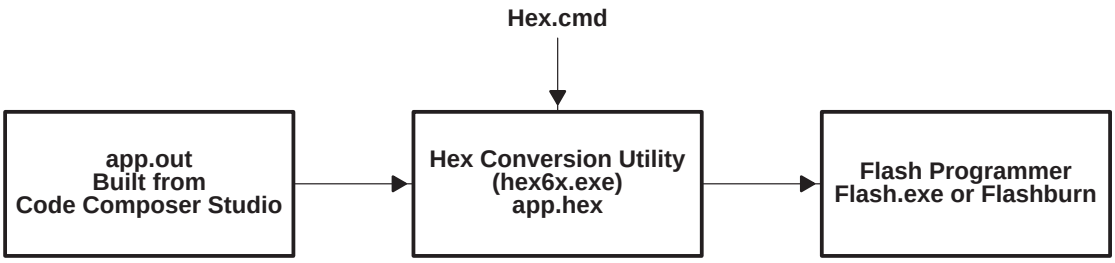

Figure 4. Flash Programming Sequence

2.4.1 Hex Conversion Utility

The hex conversion utility accepts a COFF object (.out) file as an input and converts this file into ASCII hexadecimal format. The Hex6x utility is part of the code generation tools shipped with Code Composer Studio. The Hex6x utility operates using command files. Command files are ASCII files that contain information defining options and filenames, ROM directives, and SECTIONS directive.

Figure 5 shows an example hex command file that can be used when the copy table is created by either inspecting the map file or using the linker options. In the command file, the user must specify the input .out file, the format for the output .hex file, the type and size of the ROM on the board, and which sections to be placed in ROM (includes all those that were given a load address in ROM).

```
/*
** ======== app_hex.cmd ========
** hex6x command file
*/
".Debug\app.out     /* input COFF file */
-map .\Hex\apphex.map   /* generate hex.map map file */
-a     /* ASCII HEX format */
-image        /* set image mode */
-zero     /* reset address origin to 0 */
-memwidth 8      /* 8-bit wide ROM */
ROMS
{
  FLASH: org = 0x90000000, len = 0x40000,romwidth = 8, files = {.\Hex\app.hex}
}
SECTIONS /* list of COFF sections to be ROMed */
{
.boot_code
.bios
.sysinit
.gblinit
.trcdata
.rtdx_text
.text
.cinit
.pinit
.const
.switch
.hwi_vec
}
```

Figure 5. Hex Command File

As described above, the hex conversion utility automatically builds the copy table when the appropriate options are specified in the hex command file. Table 4 lists the conversion utility options that are available to be added to the hex command file to automatically generate the copy table.

Table 4. Hex Utility Boot Options

Option	Description
−boot	Converts all initialized sections into bootable form
−bootorg	Specifies the address of the copy table
−bootsection	Specifies the section containing the custom boot code

Figure 6 displays an example hex command file implementing the hex utility boot options. Since the –boot option was selected, the SECTIONS directive was not included. When the –boot option is selected, all initialized sections will be assigned a load address in flash and therefore the SECTIONS directive is not required. The –bootorg option specifies that the copy table will reside at address 0x90000400 and the –bootsection option specifies the address in ROM (0x90000000) that will contain the custom boot code.

```
".\Debug\app.out"                    /* input COFF file */
-a                                   /* create ASCII image */
-image                               /* Create a memory image (no discontinuities) */
-zero                                /* reset address origin to 0 for outputfile(s)*/
-memwidth 8                          /* Width of ROM/Flash memory */
-map .\Hex\apphex.map                /* create a hex map file */
-boot                               /* create a boot table for all initialized sects*/
-bootorg 0x90000400                  /* address of the boot/copy-table */
-bootsection .boot_load 0x90000000  /* section containing our asm boot routine */
ROMS
{
    FLASH: org = 0x90000000, len = 0x0040000, romwidth = 8, files = {.\Hex\app.hex}
}
```

Figure 6. Hex Command File using the –boot Option

Refer to the *TMS320C6000 Assembly Language Tools User's Guide* (SPRU186) for more details on the hex conversion utility.

2.4.2 Flash Burn Utility

Once the .hex file is generated, use a flash programming utility to write the hex image to the board's ROM. A GUI based Flash Burn Utility is available and is present under the Tools menu in Code Composer Studio. If this tool is not available as part of the Code Composer Studio installation, visit the TI web site to download this utility through update advisor.

Make sure Code Composer Studio is not running before running the Flash Burn Utility, reset the board, and then execute the flash programming utility. It takes a few seconds to download the program into flash. The code accompanying this application note includes a *.cdd file that is used by the Flash Burn Utility. The *File To Burn* and *FBTC Program File* fields in the .cdd file need to be updated to point to the appropriate directories where the files are located.

After flashing the application, setup the device boot pins for the correct boot mode and switch on the board. At power on, figure 1 shows the sequence of events that take place. Note that after the custom boot code completes, control branches to _c_int00 and then, initialization for the C environment and BIOS starts.

All the steps that were followed to program the flash are summarized below.

Flash/Boot Procedure

1. DSP/BIOS memory configuration for flash booting
   - Create the necessary memory segments
   - Configure the memory sections with the proper load/run address
2. Write the secondary bootloader
   - Create the user linker command file
   - Create the copy table by either:
     Building the project and extracting the section's load/run address and size from the map file or
     Using the hex conversion utility
   - Build the project to create the final app.out
3. Convert COFF format (.out) to hex file for flash programmer
   - Create the hex.cmd file with the proper options
   - Run hex6x to create the .hex file
   - Program the flash using the Flash Burn Utility

3 Linker Copy Tables

This application note discusses the methods available for creating the copy table prior to the release of Code Composer Studio 3.00:

- Using addresses listed in the map file is described in section 2.3.1.1
- Using the hex conversion utility is described in section 2.3.1.2
- Using the linker directives LOAD_START, RUN_START and SIZE is described in Appendix A

The linker bundled with Code Composer Studio 3.00 introduces a new feature for creating and managing copy tables that is both easier to use and more flexible. The feature is the table directive. Consider this simple example:

```
.text   load = FLASH_REST, run = RAM, table(BINIT)
.data   load = FLASH_REST, run = RAM, table(BINIT)
.binit  load = FLASH_REST       /* allocate the copy table */
```

The table directive instructs the linker to create a copy table for that section. The BINIT argument to the table directive instructs the linker to observe special conventions for boot loading. These special conventions are as follows:

- The symbol associated with the start of the boot time copy table is ___binit__ (three underscores, binit, two underscores).
- The copy tables are placed in the input section named .binit.

Note the last statement in the above example collects the .binit input sections together into an output section also named .binit, and allocates that output section to the FLASH_REST memory range.

If the copy tables were written in assembly, it would be similar to the following:

```
    .sect                ".binit"             ; name the section
    .global              ___binit__           ; name of table is global

___binit__:                                   ; base address of the copy table
    .short               12                   ; size of one copy table record
    .short               2                    ; how many copy table records

    ; copy table record for .text section
    .word                text load address
    .word                text run address     ; linker fills in these values
    .word                text length

    ; copy table record for .data section
    .word                data load address
    .word                data run address     ; linker fills in these values
    .word                data length
```

The general technique is to apply table(BINIT) to each section that is copied from FLASH to on-chip memory at boot time. Each instance of the table directive adds another record to the BINIT copy table.

The code in the second level boot load routine must process the copy table according to the format described above.

The remaining steps in the process such as converting the COFF file to ASCII hex format, and burning the FLASH memory, are the same as that described in Appendix A.

Appendix B walks through using the table directive to boot the RF3 example application from FLASH on the DSK6713.

For those who use code generation tools such as the linker outside of Code Composer Studio, the linker table directive is introduced in C6000 Code Generation Tools version 5.00.

Note the table directive can also be used to manage overlays. For more information consult the *TMS320C6000 Assembly Language Tools User's Guide* (SPRU186, revision N or higher), or the application note *Advanced Linker Techniques for Convenient and Efficient Memory Usage* (SPRAA46).

4 Tips for Debugging

To debug a flashed application, insert an infinite loop in the boot code so that after boot, the PC will be at the beginning of the program. Load the application symbol information using Code Composer Studio. Symbol information is loaded using File→Load symbol menu. Once the symbol information is loaded, the user can use the Code Composer Studio debugger facilities to debug the application.

The following are some pointers to ensure proper functioning of flashed application:

- Ensure proper selection of boot modes with the correct selection of boot pins.

- Make sure the endianness of built application matches to endianness provided while creating the hex image.
- Program EMIF on boot up, to correctly access the external memory on the board.
- Whenever the project is rebuilt with changes make sure the copy table is updated by looking at the map file.
- Make sure cache coherency is maintained if the application environment enables cache.
- Use hardware breakpoints to put breakpoints in flash memory.
- If loading application code into external memory, verify proper configuration of the EMIF to that CE space.
- If using a gel file during application development, confirm that the tasks performed in the gel file are included in the tasks performed in the application for the stand alone system.

5 References

1. *TMS320C6000 Tools: Vector Table and Boot ROM Creation* (SPRA544)
2. *TMS320C6000 EMIF to External Flash Memory* (SPRA568)
3. *TMS320C6000 Peripherals Overview Reference Guide* (SPRU190)
4. *TMS320C6000 Assembly Language Tools User's Guide* (SPRU186)
5. *DSP/BIOS Sizing Guidelines for the TMS320C62x DSP* (SPRA667)
6. *A DSK Flash Memory Programmer* (SPRA804A)
7. *C6000 Boot Mode and Emulation Reset* (SPRA978)
8. *TMS320C6000 DSP Integration Workshop* (IW6000)
   http://focus.ti.com/docs/training/traininghomepage.jhtml

Appendix A

A.1 C620x/C670x Bootloader

The C620x/C670x on-chip bootloader copies 64K bytes of data from CE1 space to IPRAM. In this case if the application size is less than 64K byte, then the application can boot without a secondary bootloader.

In C620x/670x architecture, IPRAM holds only the program data therefore the bootloader should not copy data or initialized data sections into IPRAM. This constraint means that any of the initialized data like .cinit, .const should work from the flash (external) memory unless the user writes a secondary bootloader to copy the needed initialized sections to IDRAM.

A.2 Bootloading a Non-BIOS Application

The bootloading process for a non-BIOS application includes the same steps described in section 2 for a DSP/BIOS application. These steps are as follows:

1. Configuring memory for flash booting
2. Building the application
3. Writing custom boot code
4. Flashing the application

Although the bootloading process for a non-BIOS application includes the same steps as the bootloading process for a DSP/BIOS application, the method of implementing each step differs. The major difference is in the method used to configure the memory which includes defining memory segments and memory (COFF) section placement. Slight changes must also be made to the custom boot code described above. In addition to these changes, another method of creating the copy table is supported for non−BIOS applications. Section A.2.1 to Section A.2.3 detail how to complete these tasks for a non-BIOS application. The non−BIOS secondary boot loader code is included in the Blink D5K6713.pjt file and is available for download with this application note.

A.2.1 Defining Memory Segments

As in a DSP/BIOS application, for a non-BIOS application, additional memory segments should be defined to specify two locations in ROM if the hex conversion utility is not being used to create the copy table. These memory segments are required to distinguish between the memory sections that will automatically be copied by the on-chip bootloader into RAM following reset and those sections that must be copied by the secondary bootloader.

To create memory sections in a non-BIOS application, the linker command file must be updated. Figure A−1 displays an excerpt from a linker command file that creates the following memory segments.

```
FLASH_BOOT: origin = 0x90000000, length=0x0400
FLASH_REST: origin=0x90000400, length=0x0001fc00
BOOT_RAM:: origin=0x0, length=0x0400
IRAM: origin=0x0400, length=0xfc00
SDRAM: origin=0x80000000, length=0x10000000
```

```
      MEMORY
      {
      /*the FLASH_BOOT and FLASH_REST sections are not needed if the hex converter */
      /*is used to create the copy table*/
        FLASH_BOOT:   o = 0x90000000  l = 0x00000400 /* Flash - for custom boot code  */
        FLASH_REST:   o = 0x90000400  l = 0x0001FC00 /* Flash - for application code */
        BOOT_RAM      o = 0x00000000  l = 0x00000400   /* L2- for custom boot code*/
        IRAM:         o = 0x00000400  l = 0x0000fc00   /* L2- for non-custom boot code*/
        SDRAM:        o = 0x80000000  l = 0x10000000   /* EMIF - CE1 - SDRAM        */

      }
```

Figure A−1. Non-BIOS Memory Segment Definition

A.2.2 Memory (COFF) Section Placement

For a non-BIOS application, only the memory sections listed under compiler sections in Table 1 are relevant. The memory placement suggested in table 1 for the compiler sections still apply. Based on the information in Table 1 the user can decide on load/run specifications for each section and configure them in the linker command file. In addition to the compiler sections, the load/run addresses for the user defined memory section .boot_load should also be configured in the linker command file. Figure A−2 displays an excerpt from a linker command file that configures the load/run address of each section mentioned above. Note that when a section is not copied by the secondary bootloader, only its run address must be specified in the linker command file.

```
SECTIONS
{
      /*When using the hex converter to generate the copy table, the load
        address as well as the LOAD_START, RUN_START, AND SIZE linker options
        do not need to be specified. Only the run address must be specified */
     .boot_load : LOAD = FLASH_BOOT, RUN = BOOT_RAM

     .text : LOAD = FLASH_REST, RUN = IRAM
     LOAD_START(_text_ld_start),
     RUN_START(_text_rn_start),
      SIZE(_text_size)
      /*LOAD_START, RUN_START, AND SIZE are only required when using the linker
        options to generate the copy table */
     .const    >     FLASH_REST
     .cinit    >     FLASH_REST
     .pinit    >     FLASH_REST
     .switch   >     FLASH_REST
     .data     >     IRAM
     .cio      >     IRAM
     .bss      >     IRAM
     .far      >     IRAM
}
```

Figure A−2. Non-BIOS Memory Section Placement

A.2.3 Creating the Section Copy Table

Similar to a DSP/BIOS application, a secondary bootloader for a non-BIOS application also uses a section copy table to copy memory sections from their load address to their run address. For a non-BIOS application there are three ways to create the section copy table.

- Inspecting the map file

- Using the –boot option in the hex conversion utility
- Using linker options (LOAD_START, RUN_START, SIZE)

The first two methods, inspecting the map file and using the hex conversion utility, were discussed above and the steps described to implement these methods still apply for a non-BIOS application. The most convenient of the three methods is using the hex conversion utility. The following paragraphs describe the third method, using the linker options.

Code Composer Studio 2.2 and later offers linker options that increase the ease in which the copy table can be created, removing the necessity to inspect the map file. The linker options used to accomplish this task are LOAD_START, RUN_START, and SIZE.

To utilize these linker options we must make modifications to the project described above. First, the user-defined linker command file must be updated to include LOAD_START, RUN_START, and SIZE directives. A sample of a linker command file with these updates is shown in Figure A−2. Each memory section to be copied by the secondary bootloader from its load address to its run address is defined here with a LOAD_START, RUN_START, and SIZE directive. After evaluating each of these commands, the specified symbol will contain the section's load address, the section's run, and the size of the section respectively. For example, in the linker command file shown in Figure A−2, _text_ld_start will contain the .text section's load address, _text_rn_start will contain the .text section's run address, and so forth.

Once the user-defined linker command file has been updated, the custom boot code must be updated as well. Each of the symbols defined in the linker command file must also be defined as global variables in the custom boot code. An example of this is shown below for the .text section.

```
.global _test_size
.global _text_rn_start
.global _text_ld_start
```

Finally, these symbols should be added to the copy table. The copy table example shown in section 2.3.1.1 can now be updated with the following code.

```
COPY_TABLE:
               ;;size
               ;;destination  (run address)
               ;;source (load address)
               ;;     .text
               .word  _text_size
               .word  _text_rn_start
               .word  _text_ld_start
```

Figure 5 displays the hex command file that should be used with this method of creating the copy table. Please note when selecting this method for creating the copy table, the copy routine as well as the final branch statement must be updated. All of the changes described here are included in BlinkD5K6713.pjt.

Appendix B Example of Linker Table Directive Usage

B.1 Introduction

The example described here executes on the DSK6713. It requires use of Code Composer Studio version 3.00 or higher. The files for this example are in the rf3_dsk6713_boot_with_table directory of the .zip file supplied with this application note. The main application project app.pjt is in the directory rf3_dsk6713_boot_with_table\referenceframeworks\apps\rf3_table_boot\dsk6713. A simpler version of this example accompanies the application note which describes Reference Frameworks Level 3: *Reference Frameworks for eXpressDSP Software: RF3, A Flexible, Multi-Channel, Multi-Algorithm, Static System* (SPRA793).

B.2 Use the Table Directive

Bootloading DSP/BIOS applications using the table(BINIT) linker directive is a little tricky at present. Because the DSP/BIOS generated linker command file (*prog_name*cfg.cmd) does not yet support applying the table directive to load/run sections, special steps are required. Enhancement request SDSsq37348 “Generated .cmd file should include table() for load/run sections” has been logged.

The textual configuration (Tconf) script memory setting as follows:

```
// Set runtime critical sections to be copied from Flash -> RAM
tibios.MEM.LOADBIOSSEG = FLASHREST
tibios.MEM.BIOSSEG = tibios.IRAM;    /*DSP/BIOS Code Section (.bios)
```

is visualized in graphical configuration (gconf) in Figure B−1

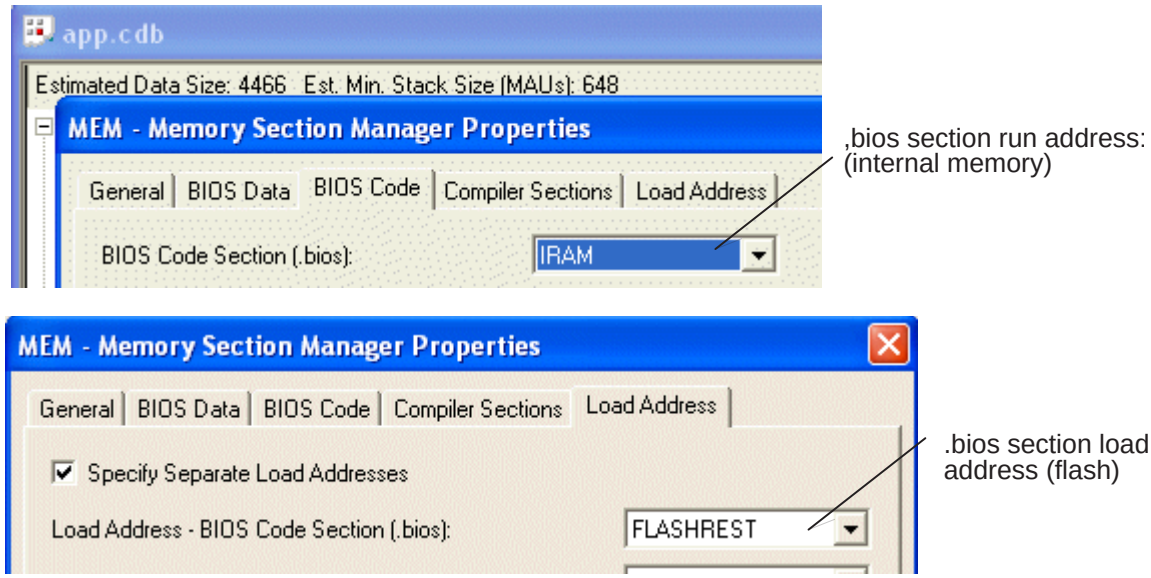

Figure B−1. Existing Load/Run Mechanism in DSP/BIOS Present in Code Composer Studio 2.21

This setting generates the following DSP/BIOS linker command file entry.

```
.bios:...{} load > FLASHREST, run > IRAM
```

The missing piece is the table(BINIT) directive for bootloading initialized sections with separate load and run addresses. For example, the command file entry in Figure B−1 should be:

```
.bios:    {} load > FLASHREST, run > IRAM, table(BINIT)
```

As a workaround, a Javascript function, applyTableBinitCmdFile.tci, does the following:

- Reads in the DSP/BIOS generated linker command file
- Searches for section specifications with load and run directives, as well as the memory assignment operator >
- For each section specification that meets the search criteria, adds the string ", table(BINIT)" to the end of the section specification
- Saves the modifications to the original prog_namecfg.cmd file

This sounds complex. Why not manually add the ", table(BINIT)" string to the *prog_name*cfg.cmd file?The problem is that any modification of the DSP/BIOS configuration, or even performing a Rebuild All, would generate a new *prog_name*cfg.cmd file and thus overwrite the table additions.

How does this Javascript function get called? Most DSP/BIOS users have switched to Tconf for their DSP/BIOS objects configuration since it is a more scalable, portable solution than relying on the graphical equivalent. In the configuration file processed by Tconf, the Javascript function is called after the generation of the *prog_name*cfg.cmd file, as follows:

```
// Name of file: appcfg.tcf – configuration file processed by Tconf
// generate the DSP/BIOS configuration
   prog.gen();
// after generating *cfg.cmd file, now modify it for table() operator
   ti_tcapps_utils.applyTableBinitCmdFile (prog.name + "cfg.cmd",
                                           prog.name + "cfg.cmd",
                                           ", table(BINIT)" );
```

The function takes an input file (e.g. appcfg.cmd), and produces an output file (e.g. appcfg.cmd) with a string ", table(BINIT)" appended to each load/run section.

B.3 Notes on Running the Example

The DSP/BIOS example attached to this application note includes the Javascript function for adding the table directive. Other notes on this example:

- The assembly boot code which does the section copying using the linker generated copy tables is easily ported to other C6000 systems. For example, this code only requires simple changes to the EMIF settings to run on the DSKC6711.
- Compiler sections are placed in the project's link.cmd instead of implicitly in the DSP/BIOS generated *prog_name*cfg.cmd file. This is done via setting `tibios.MEM.ENABLELOADADDR = true`; in the Tconf script. While this step is not strictly necessary, it shows usage of the table directive in the project's linker command file.
- At this writing, the only way to connect Code Composer Studio to the DSK6713 is with an emulator such as the XDS−560. Drivers that support direct connection over USB will be released at a later date.
- Files for programming the flash are found in the hex sub−directory of the main application project directory. A readme.txt file in that directory has more detail.
- Plays audio out of the box. After you have burned the code into Flash via Flashburn it will play audio on board power−up. Ensure you have an audio input source plugged into Line In and speakers attached to Line Out.

The default GEL file supplied with the emulator is not specifically configured to the DSK6713. Replace it with the DSK6713.gel file located in the rf3_dsk6713_boot_with_table directory of the .zip file.

1. Open Code Composer Studio Setup.
2. Configure the system to use an emulator connected to a C671x target system.
3. In the leftmost column, right click on the name of the emulator and select **Properties**.
4. Select the tab **Startup GEL File(s)**.
5. Click the **"..."** box on the far right, browse to rf3_dsk6713_boot_with_table, select the filename DSK6713.gel, then click **Open**, then click **Finish**.
6. Save the configuration. Select **File | Save**.
7. Quit Code Composer Studio Setup. Select **File | Exit**.

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