TEXAS INSTRUMENTS

*Application Report*
*SPRA743 - March 2001*

# *Developing a DSP/BIOS Application for ROM on the TMS320C6000 Platform with CCS 1.2*

*Alec Bellanca* *Software Applications, TI Santa Barbara*

**ABSTRACT**

For many DSP applications, it is desirable to boot the application from ROM. When using DSP/BIOS, several issues need to be considered while preparing the application for storage in ROM. These include DSP/BIOS configuration settings, placement in memory of code and data sections and use of ROM programming software utilities. This application report describes the process of implementing a DSP/BIOS application in ROM with Code Composer Studio 1.2, and notes the BIOS–specific issues encountered during this process. The reader will learn how to create boot code, link the program appropriately, and prepare the ROM image. Examples are given for using FLASH ROM on the TMS320C6211 DSK development board.

**Contents**

**List of Figures**

**List of Tables**

## 1 ROM Boot Overview

When a DSP/BIOS application is booted from ROM, the following procedure should take place:

- Upon reset, the TMS320C6000 will initialize a segment of memory (copy from ROM into RAM), including custom boot code.
- The custom boot code will then execute and perform additional section copying to initialize program and data memory as needed.
- DSP/BIOS startup procedures will initialize C variables in the .bss section with data contained in the .cinit section, as well as perform general DSP/BIOS setup.
- The application will begin execution at main().

Several things to be considered when setting up your application to follow this procedure. First, your custom boot code must be linked so that it is loaded and executed properly upon reset. This boot code must be written so that it will copy the correct COFF sections out of ROM. Next, these sections must be properly linked so that they can be loaded into ROM and run from RAM. Finally, the ROM image must be created and programmed into ROM.

## 2 Project Configuration

Before building your project, there are a few basic settings to configure so that Code Composer Studio will generate code suitable for booting from ROM.

***Linker Options***

In the Code Composer Studio menu bar, select Project–>Options and click on the Linker tab. Specify a map filename so that the linker will generate a map file. This file will contain linking information that is necessary to understand section placement and to write your boot code.

***BIOS Options***

Open your project's DSP/BIOS configuration (.cdb) file, and go to the property page for "Global Settings." Select ROM as the C Autoinitialization Model. This will ensure that the .cinit section is built for run-time data initialization. Also, under the Memory Section Manager, turn off "Reuse startup code space." If checked, this option will try to reuse the .sysinit section as heap space. However, the .sysinit section will end up being placed in ROM, which can not be used for a heap.

## 3 Automatic Boot Initialization

To program an application for ROM, it is necessary to understand the target board's automatic boot procedure. The TMS320C6000 platform can be configured to boot in several ways. When configured to boot from ROM, the boot process is as follows:

- Copy first 1K or 64K of ROM (CE1 address space) into memory mapped at address 0x00.
  - TMS320C6x0x: 64K is copied. Memory mapped to address 0x00 is either CE0 space or IPRAM (memory maps 0 and 1 respectively).
  - TMS320C6x1x: 1K is copied. Memory mapped to address 0x00 is IRAM (usually later used by the application as L2 cache).

- Begin execution at address 0x00.

This automatic copying is used to load your custom boot code and begin executing it. Note that on the ‘C6x0x architectures, enough memory is copied to initialize all internal program memory.

## 4 Defining Memory Sections

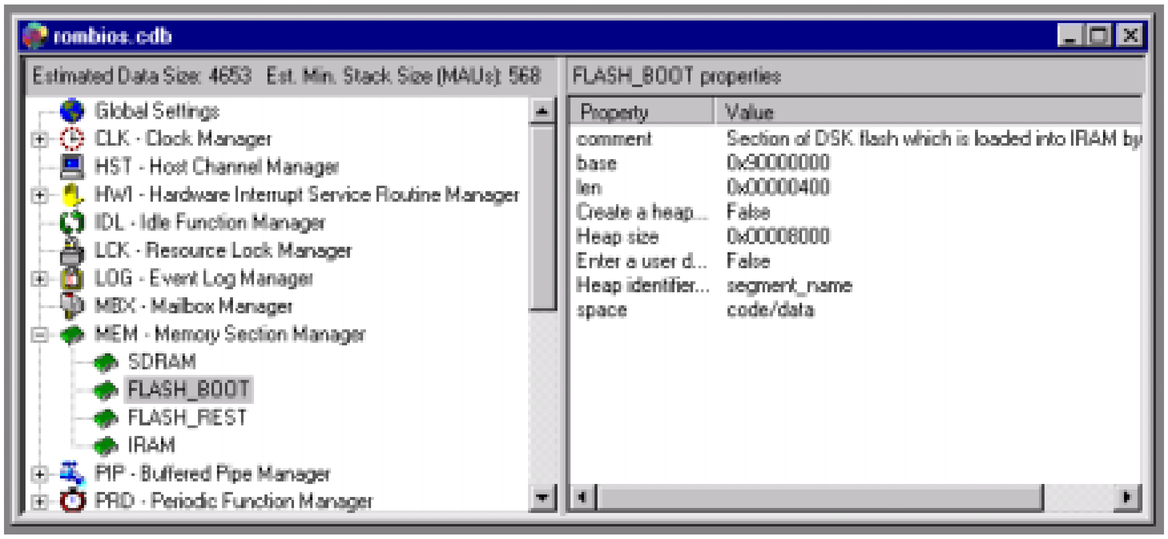
To properly link your project, you will first need to define extra memory sections so that you can specify locations in ROM. Open your DSP/BIOS configuration (.cdb) file, right-click on the “MEM – Memory Section Manager” object, and choose “Insert MEM”. At the least, you will need to create one section for the region in memory occupied by your ROM. It is often important to distinguish between the portion of ROM that is automatically copied into RAM during reset from the rest of ROM by creating two separate ROM sections. For example, on the 6211 DSK, create these two sections:

```
FLASH_BOOT: origin = 0x90000000, length = 0x400
FLASH_REST: origin = 0x90000400, length = 0xFC00
```

It may also be necessary to define a section for the location in RAM where the data in FLASH_BOOT will be copied to. On the 6211 DSK, this is:

```
IRAM: origin = 0x00000000, length = 0x400
```

After adding these sections to the default 6211 DSK configuration file, the configuration tool’s Memory Section Manager will look like this:


rombios.cdb
Estimated Data Size: 4653 Est. Min. Stack Size (MAUs): 568
Global Settings
CLK - Clock Manager
HST - Host Channel Manager
HWI - Hardware Interrupt Service Routine Manager
IDL - Idle Function Manager
LCK - Resource Lock Manager
LOG - Event Log Manager
MBX - Mailbox Manager
MEM - Memory Section Manager
SDRAM
FLASH_BOOT
FLASH_REST
IRAM
PIP - Buffered Pipe Manager
PRD - Periodic Function Manager
FLASH_BOOT properties

| Property | Value |
|---|---|
| comment | Section of DSK flash which is loaded into IRAM by |
| base | 0x90000000 |
| len | 0x00000400 |
| Create a heap... | False |
| Heap size | 0x00008000 |
| Enter a user d... | False |
| Heap identifier... | segment_name |
| space | code/data |

**Figure 1. DSP/BIOS Configuration File with Memory Sections for a 6211 DSK**

## 5 Linking

When linking your application, you need to be very careful where you place the application's COFF sections. In many cases, you will need to specify two different addresses for a section: a load address and a run address. The load address determines where a loader places the raw data for the section. Any references to the section (such as labels in it) refer to its run address. This means that when a section is accessed by your application, by reading or writing data or executing code, it will expect to find the section located at its run address. Therefore, if you specify different load and run addresses for a section, the section must be copied from its load address to its run address before it can be used. In some cases, this can be done automatically during chip reset. If this is not the case, you will need to write custom boot code to manually copy those sections. For example, on C6x0x architectures, 64K is copied out of ROM automatically – this can be used to initialize all internal program memory. If you need to initialize any data memory or external memory, however, this must be done by custom boot code.

When deciding where to link your sections, keep the following ideas in mind. All code and initialized data must have a load address in ROM. All non–constant data must have a run address in RAM. Uninitialized data does not need a separate load address. Any code that your board automatically copies from ROM into RAM (such as boot code) should be given appropriate load and run addresses in ROM and RAM. With other code sections, you have the option of keeping them in ROM by having a load and run address in ROM. Having a run address in ROM may be inefficient, as it is usually an order of magnitude slower than RAM, however it will save some space in RAM and you won't have to copy the section. In most cases, speed is more important, and you should give these code sections run addresses in RAM. A notable exception is the sections that contain initialization tables or startup code. These sections will be used only once, during startup, therefore it would be superfluous to copy them into RAM first.

### 5.1 COFF Section Descriptions

Since DSP/BIOS creates several extra sections with many different purposes, it is helpful to know something about each section, so you can make an educated decision on where to link it to.

**Table 1. DSP/BIOS Sections**

| Section Name | Type of section | Description / Notes | Suggested placement on a 6211 DSK |
|---|---|---|---|
| .args | Uninitialized Data | Argument buffer | Load = run = RAM |
| .stack | Uninitialized Data | The stack | 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 tables, used only during startup | Load = run = ROM |
| .trcinit | Initialized Data | Trace mask section, must have a run address in RAM | Load = ROM, run = RAM |
| .sysdata | Uninitialized Data | Kernel state | Load = run = RAM |
| .hwi_vec | Code | Interrupt table | Load = ROM, run = RAM |
| All other BIOS sections | Uninitialized Data | Object memory, etc. | Load = run = RAM |

**Table 2. Compiler Sections**

| Section Name | Type of section | Description / Notes | Suggested placement on a 6211 DSK |
|---|---|---|---|
| .text | Code | Program code, may contain some DSP/ BIOS code as well | Load = ROM, run = RAM |
| .switch | Initialized Data | Switch statement jump tables | Load = ROM, run = RAM |
| .bss, .far | Uninitialized Data | C variables. Automatically initialized at run time from the .cinit table | Load = run = RAM |
| .cinit, .pinit | Initialized Data | C variable and function init tables | Load = run = ROM |
| .const | Initialized Data | Constants. Note: LOG_printf() requires load and run addresses to be the same for correct execution. | Load = run = ROM |
| .data, .cio, .sysmem | Uninitialized Data | Misc. data sections | Load = run = RAM |

In addition to these, your custom boot code should have its own section. Link this code so that it will be copied from FLASH ROM to internal RAM and executed during startup. Do this by specifying a load address at the beginning of FLASH memory, and a run address in internal ram at address 0x00. On a 6211 DSK, this corresponds to the sections FLASH_BOOT and IRAM respectively.

## 5.2 Specifying Different Load and Run Addresses

Once you have decided where you are going to put your sections, certain measures need to be taken to properly specify their placement. Sections that do not have different load and run addresses can be configured with the DSP/BIOS graphical configuration utility. However, with Code Composer Studio 1.2, you can not directly specify different load and run addresses using the configuration tool. To do this, you must manually edit the linker command file that is generated. This file is called xxxcfg.cmd, where xxx is the name of your DSP/BIOS configuration (.cdb) file. Inside the file you will find lines that look like this:

```
.bios: {} > SDRAM
```

As it is, this will link the .bios section to have a load and run address in SDRAM. To specify your own separate load and run addresses, simply change it to something like this:

```
.bios: {} load = FLASH_REST, run = SDRAM
```

Also, if you have created a separate section for boot code, you will need to add a line to specify where it is to be linked. For example:

```
.my_boot_code: {} load = FLASH_BOOT, run = IRAM
```

This will give the section ".my_boot_code" a load address in FLASH_BOOT and a run address in IRAM, so that the custom boot code is loaded and executed at reset. If you place multiple sections in FLASH_BOOT, make sure that the line linking the section for boot code appears before the lines for the other sections. This way, the boot code will be linked to the first address in ROM.

**WARNING:**

Be careful when hand-editing this linker command file. If you make changes to your DSP/BIOS configuration file, then the linker command file will be regenerated and you will need to re-enter all of your changes.

### 5.3 Inspecting the Map File

After you build your project, you will have a map file generated by the linker that will contain important section link information. To determine the memory location where a section was resolved to, look in the file for segments that look like this:

```
.bios      0    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 useful when writing your boot code. The map file also contains detailed information about memory sections, sub–sections, and symbols.

## 6 Writing Custom Boot Code

In many cases it is necessary to include startup code to enable access to the ROM, or to copy data from ROM into RAM before starting the application. On the 6211 DSK, the External Memory Interface (EMIF) needs to be correctly programmed to enable access to external memory (such as ROM). Once this is done, the boot code should perform copying of initialized data sections from their load addresses to their run addresses, and then jump to _c_int00, the usual program entry point. Here is a segment of boot code that demonstrates section copying:

```
; QDMA registers and values
QDMA_OPT      .equ        0x02000000  ;QDMA options register
QDMA_OPT_VAL        .equ        0x21200001  ;QDMA options
QDMA_SRC      .equ        0x02000004  ;QDMA source address register
QDMA_CNT      .equ        0x02000008  ;QDMA count register
QDMA_DST      .equ        0x0200000c  ;QDMA destination address register
QDMA_S_IDX    .equ        0x02000030  ;QDMA index pseudo–register
      .sect ”.myBootCode”
      .global myBootCode
      .ref _c_int00
;
; ======== myBootCode ========
;
myBootCode:

; **************
; Configure EMIF
; **************

            … … …
; *************
; Copy Sections
; *************
                  mvkl  copyTable, a3     ; load table pointer
                  mvkh  copyTable, a3
```

```
copySectionLoop:

            ldw   *a3++, b0   ; byte count
                  ldw   *a3++, a4   ; load ram start address
                  ldw   *a3++, b4   ; load flash start address
                  nop   2
      [!b0]             b copyDone        ; have we copied all sections?
                  nop 5

      ; copy this section with QDMA
                  mvkl  QDMA_OPT,A5 ; set QDMA options
                  mvkl  QDMA_OPT_VAL,B5
                  mvkh  QDMA_OPT,A5
                  mvkh  QDMA_OPT_VAL,B5
                  stw   B5,*A5
                  mvkl  QDMA_SRC,A5 ; load source address
                  mvkh  QDMA_SRC,A5
                  stw   B4,*A5
                  shr   B0,2,B1           ; divide size by 4 (because we're in
32–bit mode)
                  mvkl  QDMA_CNT,A5       ; load word count
                  mvkh  QDMA_CNT,A5
                  stw   B1,*A5
                  mvkl  QDMA_DST,A5 ; load destination address
                  mvkh  QDMA_DST,A5
                  stw   A4,*A5
                  mvkl  QDMA_S_IDX,A5 ; set index. writing to this register
will
                  mvkh  QDMA_S_IDX,A5 ; also initiate the transfer.
                  zero  B5
                  stw   B5,*A5        ; go!
      ; next section
                  b       copySectionLoop
                  nop   5
copyDone:                     ; done with section copying.

; *************
; Start Program
; *************

                  mvkl .S2 _c_int00, B0
                  mvkh .S2 _c_int00, B0
                  B    .S2 B0             ; jump to _c_int00
                  nop   5
```

**Figure 2. Section Copy Code Example**

This code uses the 6211 Quick DMA (QDMA) in 32–bit mode to copy certain sections into RAM. The source address, destination address, and size of all sections to copy are stored in a table for easy manipulation:

```
    ; Table of sections to copy. Format is:
    ; word 0:     byte count
    ; word 1:     run address
    ; word 2:     load address

      .ref textSize,    textRun     ; these symbols created with
      .ref biosSize,    biosRun     ; linker command file
      .ref trcinitSize, trcinitRun
      .ref hwi_vecRun

copyTable:
    ; .text
    .word (textSize)
    .word (textRun)
    .word 0x90001a20

    ; .bios
    .word (biosSize)
    .word (biosRun)
    .word 0x900055e0

    ; .hwi_vec
    .word 0x200                     ; hwi_vec is 0x200 bytes
    .word (hwi_vecRun)
    .word 0x90001820

    ; .trcinit
    .word (trcinitSize)
    .word (trcinitRun)
    .word 0x90001448

    ; end of table
    .word 0
    .word 0
    .word 0
```

**Figure 3. Section Info Table Example**

Fill in this table with the load addresses of the sections, which can be found by inspecting the .map file produced by building the project. The sizes and run addresses of sections can be automatically calculated by the linker, and referenced symbolically in the table. This is done in the above example with the symbols textSize, textRun, etc. These symbols are generated with special code placed in the linker command file. For example, to generate symbols for the sections .hwi_vec, .bios. and .text, locate the lines in the linker command file where those sections are placed, and insert code within the curly braces like this:

```
.hwi_vec: {
     HWI_A_VECS = .; /*(placed by DSP/BIOS)*/
     hwi_vecRun = .;
     *(.hwi_vec)
           hwi_vecSize = . – hwi_vecRun
} load = FLASH, run = SDRAM align = 0x400
       …
.text:    {
     textRun = .;
     *(.text)
     textSize = . – textRun;
} load = FLASH, run = SDRAM
     …
.bios:    {
     biosRun = .;
     *(.bios)
     biosSize = . – biosRun;
} load = FLASH, run = SDRAM
```

**Figure 4. Example Linker Command File Segments**

This will create symbols to store the sizes and run addresses of these sections, so you won't need to look this information up in the map file. Note that you still need to reference the map file for load addresses.

## 7 Programming Your ROM

Code Composer Studio comes with several utilities that can help you write your application into ROM. For both the C54x and C6x architectures, a flash programmer and a hex conversion utility are provided. To write an application to a 6211 DSK, follow this procedure. First, build your project to generate a .out file. Then, use the C6211/C6711 hex conversion utility, hex6x.exe, to create a .hex file from the .out file. This utility operates using command files. In the command file, you can specify the input .out file, the format for the outputted .hex file, the type and size of ROM in your board, and what sections to be placed in ROM. These sections should be all those that were given a load address in ROM. Here is an example hex command file for a 6211 DSK:

```
/*
** ======== hex.cmd ========
** hex6x command file for ROM audio example
*/
rombios.out      /* input COFF file */
–map hex.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 = 0x10000,romwidth = 8, files = {rombios.hex}
}
SECTIONS         /* list of COFF sections to be ROMed */
{
  .myBootCode
  .bios
  .sysinit
  .gblinit
  .trcinit
  .rtdx_text
  .text
  .cinit
  .pinit
  .const
  .switch
  .hwi_vec
}
```

**Figure 5. Hex.cmd – Example hex6x.exe Command File**

Once you have generated a .hex file, you can use a flash programming utility to write the hex image to the board’s ROM through the parallel port. For the 6211 DSK, you can use the “`flash.exe`” program, which is located in the directory “`TI\c6000\examples\dsk\flash\Host\Debug`”. Make sure Code Composer Studio is not running, reset the board, and then execute the flash programming utility. In a few seconds, your application will be in ROM and ready to run.

## 8 Debugging

Debugging an application that is booted from ROM can be challenging. There are a few things to keep in mind that will make the process easier. On the 6211 DSK, you should begin debugging by resetting the chip manually and then starting up Code Composer Studio. This will ensure that the board has booted up and the emulation settings are initialized. Once CCS is loaded, it is necessary to load the symbolic information for your application to enable debugging features. When loading a program on to the target, this is done automatically after load, but when booting from ROM, you need to do it yourself by going to File–>Load Symbol and selecting your application (.out) file.

Here are some more tips for debugging:

- If the TMS320C6000 executes invalid code it may not be possible to gain control of the DSP using Code Composer Studio's debugger, since this code may have caused the emulation settings to be modified and potentially made invalid. It is therefore a good idea to place an infinite loop at location 0x00 (reset vector) while debugging, to be sure that the DSP is executing valid code while connecting to CCS. Once CCS is running, this loop can be terminated by modifying a conditional register or by manually advancing the program counter (PC). Further debug can then occur using breakpoints to verify program execution sequence.
- When using breakpoints, note that only hardware breakpoints may be set inside ROM, and you are limited to one hardware breakpoint to be active at a time. Also keep in mind that if you set a software breakpoint in RAM and then restart your application, the breakpoint risks being overwritten when section copying occurs.
- Make sure your application never tries to write into ROM. This can cause many unpredictable problems. This situation can easily occur if you accidentally place a data section in ROM that is not read–only, such as .trcinit.

## 9 Additional Resources

- For more information on TMS320C6000 ROM boot procedures, refer to application report SPRA544B – "TMS320C6000 Tools: Vector Table and Boot ROM Creation".
- For more information on programming the TSM320C6000 External Memory Interface (EMIF) for access to external FLASH ROM, see application report SPRA568 –"TMS320C6000 EMIF to External Flash Memory".
- For more information on using the TMS320C6000 DMA controller, reference the TMS320C6000 Peripherals Reference Guide (Literature Number: SPRU190C).
- For more information on using the linker and linker command files, browse the Code Composer Studio 1.2 online help, or the TMS320C6000 Assembly Language Tools User's Guide (Literature Number: SPRU186G).
- For more information DSP/BIOS COFF sections, read application report SPRA667 – "DSP/BIOS Sizing Guidelines for the TMS320C62x DSP".