TMS320C6000 Tools: Vector Table and Boot ROM Creation

David Bell *Digital Signal Processing Solutions*

ABSTRACT

Texas Instruments TMS320C6000™ digital signal processors (DSPs) provide a variety of boot configurations that determine which actions are performed after device reset, to prepare for initialization. The boot process is determined by latching the boot configuration settings at reset.

The boot process performed by the DSP is to either load code from an external read-only memory (ROM) space, or to have code loaded through the host interface by an external host processor.

This document describes:

- The ROM boot process
- How to create a vector table
- How to build a ROM boot code in C and assembly

Contents

TMS320C6000 is a trademark of Texas Instruments.

Trademarks are the property of their respective owners.

List of Figures

1 Introduction

Several device settings are configured at reset to determine how the device operates. These settings include boot configuration, input clock mode, endian mode, and other device-specific configurations. The boot process is determined by the boot configuration selected, as described for each C6000™ individually in the text below. Up to three types of boot processes are available:

- **No boot process:** The central processing unit (CPU) begins direct execution from the memory located at address 0. If synchronous dynamic random-access memory (SDRAM) is used in the system, the CPU is held until SDRAM initialization is complete. This feature is not available on the TMS320C621x/C671x.
- **ROM boot process:** The program, located in external ROM, is copied to address 0 by the direct memory access/extended direct memory access (DMA/EDMA) controller. Although the boot process begins when the device is released from external reset, this transfer occurs while the CPU is internally held in reset. On TMS320C62x™/TMS320C67x™, this boot process also lets you choose the width of the ROM. In this case, the extended-memory interface (EMIF) automatically assembles consecutive 8-bit bytes or 16-bit half-words to form the 32-bit instruction words to be moved. For the TMS320C620x/TMS320C670x, these values are expected to be stored in little-endian format in the external memory, typically a ROM device. For the C621x/C671x, the data should be stored in the endian format that the system is using (either big or little endian). The TMS320C64x™ only supports 8-bit ROM boot, where data should be stored in the endian format that the system is using. The transfer is automatically done by the DMA/EDMA as a single-frame/block transfer from the ROM to address 0. After completion of the block transfer, the CPU is removed from reset and allowed to run from address 0.

C6000, TMS320C62x, TMS320C67x, and TMS320C64x are trademarks of Texas Instruments.

The ROM boot process differs slightly between specific C6000 devices.

- **C620x/C670x:** The DMA copies 64K bytes from CE1 to address 0, using default ROM timings. After the transfer, the CPU begins executing from address 0.
- **C621x/C671x/C64x:** The EDMA copies 1K bytes from the beginning of CE1 to address 0 using default ROM timings. After the transfer, the CPU begins executing from address 0. Some C64x devices have more than one EMIF, refer to the device specific data manual to determine which EMIF is used for booting purposes.
- **Host boot process:** The CPU is held in reset while the remainder of the device is released. During this period, an external host can initialize the CPU's memory space as necessary through the host interface, including internal configuration registers, such as those that control the EMIF or other peripherals. Once the host is finished with all necessary initialization, it must set the DSP interrupt (DSPINT) to complete the boot process. This transition causes the boot configuration logic to remove the CPU from its reset state. The CPU then begins execution from address 0. The CPU does not latch the DSPINT condition, because it occurs while the CPU is still in reset. Also, DSPINT wakes the CPU from internal reset only if the host boot process is selected. All memory may be written to, and read by, the host. This allows for the host to verify what it sends to the processor, if required.

  Note that the host interface used during host boot varies between different devices, depending on the host interface peripheral that is available on the device. Refer to the device data sheet for the specific peripheral set. One of the following host interfaces is used for host boot:

- Host port interface (HPI): For devices with HPI, the HPI can be used for host boot. The HPI is always a slave interface, and needs no special configuration.
- Expansion bus (XBUS): For devices with XBUS, the XBUS can be used for the host boot. The type of host interface is determined by a set of latched signals during rest.
- PCI: For devices with peripheral component interconnect (PCI), the PCI can be used for the host boot.

For the C620x/C670x the dedicated BOOTMODE[4:0] pins determine the device boot configurations.

For the C621x/C671x, the pullup/pulldown resistors on the host port interface, along with the CLKMODE0 input pin, are used to determine the boot and device configurations at reset. The TMS320C6712 has dedicated configuration pins LENDIAN, BOOTMODE[1:0], and CLKMODE0. Pins HD[4:3] of the host port are used to determine the boot configuration at reset. Only two of the five BOOTMODE bits are required because the C621x/C671x only has one memory map, which places internal memory at address 0. The HD[4:3] pins (or BOOT-MODE[1:0] pins on C6712) map to the BOOTMODE[4:3] pins of the C620x/C670x.

For the C6202(B)/C6203/C6204, the pullup/pulldown resistors on the XBUS are used to determine the boot configuration (pins XD[4:0]) and other device configurations (pins XD[31:5]) at reset. The XD[4:0] lines directly map to BOOTMODE[4:0] during reset. Reserved configuration fields should be pulled down. The input clock mode is configured through the CLKMODE input pins at reset.

C64x is a trademark of Texas Instruments.

For the C6205, the pullup/pulldown resistors on the EMIF data bus are also used to determine the boot mode selection (pins ED[4:0]), and other device configurations (pins ED[31:5]), at reset. The ED[4:0] lines directly map to BOOTMODE[4:0] during reset. Also, the C6205 CLKMODE0 input pin is used in conjunction with the EMIF data bus pins to determine the input clock mode at reset. Reserved configuration fields should be pulled down.

For C64x devices, the boot configuration is set through pullup/down resistors located on specific pins of the EMIF address bus. Other device configurations are also set through pullup/pulldown resistors on other pins. For example, the C6414 device configurations are determined by the pullup/down resistor on the HPI data pin HD5, along with the pullup/down resistors on the EMIFB address bus (BEA[20:1]), latched at device reset. The C6415 device configurations are determined by the PCI_EN pin, the MCBSP2_EN pin, the pullup/down resistor on the HPI data pin HD5, and the pullup/down resistors on the EMIFB address bus (BEA[20:1]), latched at device reset. The device data manual for the device being used details which pins are used to set the boot option as well as other device configurations.

Please see the *TMS320C6000 Peripherals Reference Guide* (SPRU190) for a complete description of device boot configuration.

This document describes only the ROM boot process:

- The creation of a vector table for assembly and C frameworks
- How to build code to be downloaded from ROM

2 ROM Boot Process

During DSP reset, a fixed-size memory block from an external ROM memory space is transferred to memory located at address 0, as shown in Figure 1. The DMA/EDMA controller performs this transfer. At the end of the transfer, the CPU is removed from reset and allowed to start from memory location 0.

On the C620x/C670x, the DMA copies 64K bytes from CE1 to address 0, using default ROM timings. After the transfer, the CPU begins executing from address 0. On the C621x/C671x/C64x, the EDMA copies 1K bytes from the beginning of CE1 to address 0, using default ROM timings. After the transfer, the CPU begins executing from address 0.

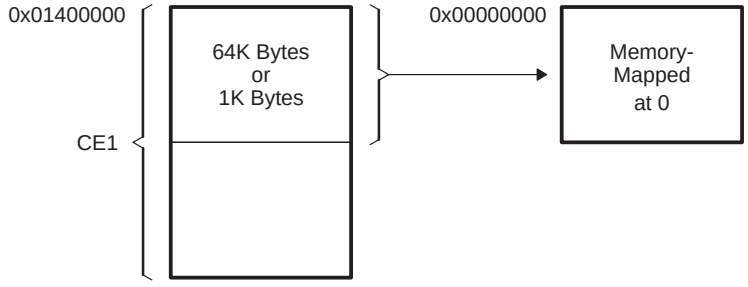

Figure 1. TMS320C6000 ROM Boot Process

For C6000 processors with multiple memory maps, the boot configuration determines the type of memory located at the reset address for processor operation at address 0. When the boot configuration selects MAP 1, this memory is internal. When the device mode is in MAP 0, the memory is external. When external memory is selected, the boot configuration also determines the type of memory at the reset address. These options effectively provide alternative reset values to the appropriate EMIF control registers.

The C621x/C671x/C64x always have internal random-access memory (RAM) at address 0, regardless of the boot configuration. The C621x/C671x have one memory map, and the ROM boot process copies 1k bytes from the beginning of CE1 to address 0, using default ROM timings.

For the C620x/C670x, the memory mapped at address 0 can either be the internal program memory (MAP1) or the external memory space CE0 (MAP0), which may contain SDRAM, synchronous burst static RAM (SBSRAM), or asynchronous memory, (DRAM/ASRAM).

3 EMIF-ROM Interface

The ROM interface to the C6000 DSP is dependant on the particular DSP within the platform. The implementation varies slightly, and the differences must be taken into account. There are two styles of interface when dealing with non-32-bit ROM (for 32-bit, all are identical). The C62x™/C67x™ devices let you choose the width of the ROM. In these devices, the EMIF automatically assembles consecutive 8-bit bytes or 16-bit half-words to form the 32-bit instruction words to be moved during the boot process. For the C620x/C670x, these values are expected to be stored in little endian format in the external memory, when a ROM device is used in the boot-up process. For C621x/C671x, the data should be stored in the endian format that the system is using (either big- or little-endian). The C64x only supports 8-bit ROM boot, where data should be stored in the endian format that the system is using.

3.1 C620x/C670x EMIF

The C620x/C670x EMIF is designed primarily for 32-bit interfaces. However, a feature was included for dealing with 8- and 16-bit ROM, to allow a more efficient interface to a typically slow interface.

When in ROM mode, the C620x/C670x EMIF packs the 8- or 16-bit data into a 32-bit words before passing it on internally to the device. The packing is always performed in little-endian fashion, regardless of the endianness of the device. Figure 2 shows the packing of data from a 16-bit ROM. Figure 3 shows the packing of data from an 8-bit ROM.

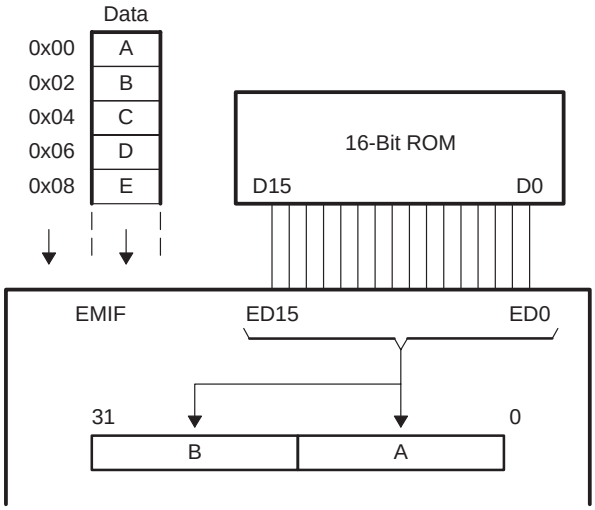

Figure 2. 16-Bit ROM Little-Endian Packing for 32-Bit EMIF

C62x and C67x are trademarks of Texas Instruments.

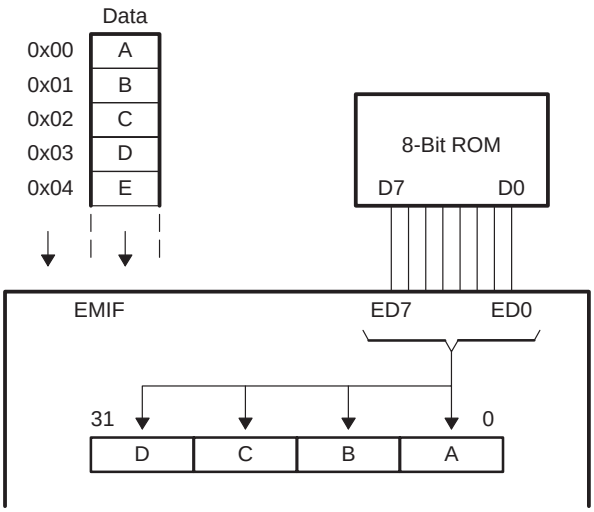

Figure 3. 8-Bit ROM Little-Endian Packing for 32-Bit EMIF

Because the packing methodology of the C620x/C670x EMIF is little-endian only, boot ROM created for a big-endian system must take this into account to function properly. Big-endian data is loaded in with the most significant byte or half-word first. The byte ordering within a ROM is selectable with an option to the hex utility, and is described later in this document.

3.2 C621x/C671x EMIF

The C621x/C671x EMIF is designed to function with 8-, 16-, and 32-bit memories of all supported types (SDRAM, SBSRAM, and asynchronous). Endianness is always taken into consideration when packing is performed on a data access. When operating in little-endian mode, the C621x/C671x EMIF packing is identical to that of the C620x/C670x EMIF. The ROM should be connected as shown in Figure 2 and Figure 3, for little-endian C621x/C671x systems.

When operating in big-endian mode, the ROM, (as well as any other 8- or 16-bit memory), should be aligned with the most significant bit (MSB) at ED31. Data loaded in from an 8- or 16-bit memory is always loaded with the least significant byte or half-word first, regardless of endian mode. It is therefore necessary for the ROM endian format to match the endianness of the system. The interconnection of a 16-bit ROM for a big-endian C621x/C671x system is shown in Figure 4. The interconnection of an 8-bit ROM for a big-endian C621x/C671x system is shown in Figure 5.

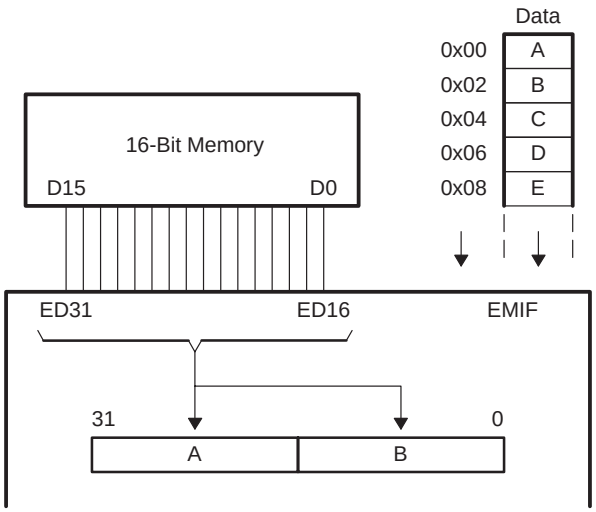

Figure 4. 16-Bit ROM Big-Endian Packing for 32-Bit EMIF

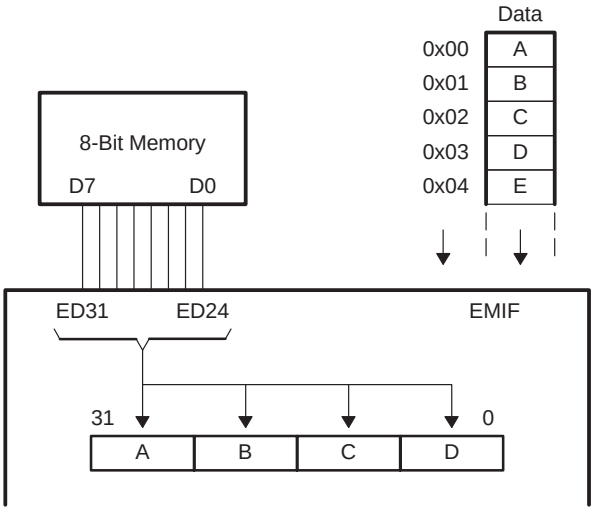

Figure 5. 8-Bit ROM Big-Endian Packing for 32-Bit EMIF

For additional details about the connection between a C6000 DSP and ROM device, please see the *TMS320C6000 Peripherals Reference Guide* (SPRU190) and *TMS320C6000 CPU and Instruction Set Reference Guide* (SPRU189).

3.3 C64x EMIF

The C64x EMIF provide a glueless interface to asynchronous memories (SRAM and EEPROM), and synchronous memories (SBSRAM, SDRAM, ZBT, SRAM, and FIFO). The size of the EMIF data bus and the number of EMIFs present on a device is different across C64x DSPs. To determine the number of EMIFs and size of their data bus for a particular device, refer to the device specific data manual.

The C64x only supports 8-bit ROM boot, where data should be stored in the endian format that the system is using. The CE1 space of the EMIF used for booting can only be used for the ROM boot process on these device series. The EMIF of the C64x packs data into 16-bit half-words before passing it on internally to the device. Figure 6 shows the packing of data from an 8-bit ROM in little-endian format. Figure 7 shows the packing of data from an 8-bit ROM in big-endian format.

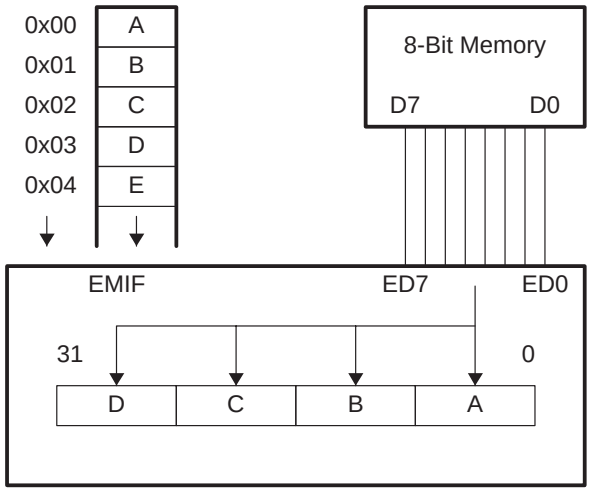

Figure 6. 8-Bit ROM Little-Endian Packing for C64x EMIF

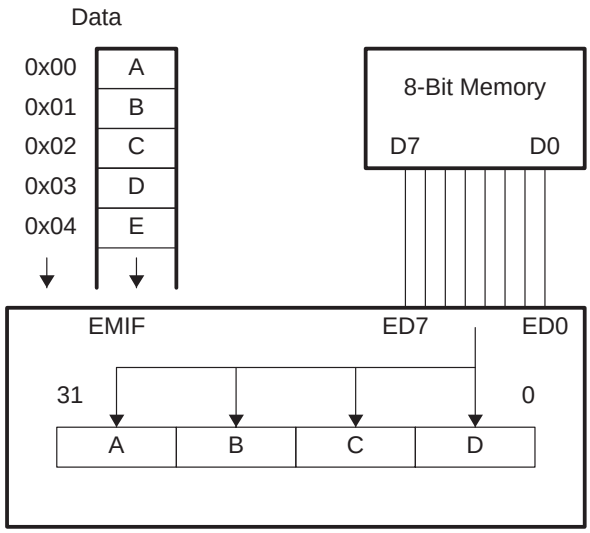

Figure 7. 8-Bit ROM Big-Endian Packing for C64x EMIF

The ROM must be connected to data pins BED[7:0], regardless of the endian mode. Data loaded in from an 8-bit memory is always loaded with the least significant byte first, regardless of endian mode. It is therefore necessary for the ROM endian format to match the endianness of the system.

4 Vector Table (Interrupt Service Table)

The processor is placed in reset when $\overline{\text{RESET}}$ is driven low. On the rising edge of $\overline{\text{RESET}}$, the boot configuration is latched and the boot process begins. Once the ROM boot process is complete (that is, the transfer from external ROM to memory mapped at 0 has finished), registers are initialized to their default value, the program counter is loaded with the reset vector (which is always 0), and the CPU begins running code at address 0. This location is referred to as the reset vector.

By default, the interrupt service table (IST) is also located at address 0. The IST is a set of interrupt vectors that get fetched when a particular CPU interrupt is serviced. Each vector is one fetch packet in length and is aligned on an 8-word boundary.

The IST is relocatable and may be moved to any 0x400-byte boundary. The IST boundary is 32 fetch packets in size. To move the IST, the interrupt service table pointer (ISTP) must be modified. The ISTP initially holds a value of 0 at reset, aligning the IST with the reset vector. At the end of all interrupt sequences, the CPU loads the vector address from the IST, plus a fetch packet offset equal to the interrupt number, to the program counter.

If the IST is relocated, the reset vector is no longer a part of it. The code placed at 0 can be either a second-level bootloader or the code itself. If the IST is kept at 0, the reset vector must call the initialization routine.

4.1 Creating a Vector Table

The IST is made up of 32[1] service vectors. Each vector corresponds to a CPU interrupt:

- Interrupt zero is reserved for the reset vector.
- Interrupt one is reserved for the non-maskable interrupt (NMI).
- Interrupts four through fifteen are programmable to signal a number of events.

The remaining interrupt numbers are reserved. Any unused vector space can be used to either extend a vector into multiple spaces, or to have other code sections.

Each interrupt vector is aligned on a fetch packet boundary. The formula to determine the location of an interrupt vector is:

Address = ISTP + 32 * <interrupt number>

For example, if the ISTP is set to 0x1000, the vector for CPU interrupt 4 would begin at address (0x1000 + 0x20 * 4), or 0x1080.

Each vector must contain eight instructions. If an interrupt service routine (ISR) cannot be completed in eight instructions (or more, if adjacent interrupt vectors are unused), the vector must contain a branch to the actual ISR. A portion of a sample IST located at 0 (including the reset vector) is shown in Figure 8. (See the *TMS320C6000 CPU and Instruction Set Reference Guide* (SPRU189) for additional details about the interrupt service table.)

[1] There are 16 CPU interrupts available. Interrupts 16 through 31 are reserved.

```
                .sect vectors

RESET:          MVK  .S2  Start, B0 ; Load Start address
                MVKH .S2  Start, B0 ; Load Start address
                B    .S2  B0        ; Branch to start
                NOP
                NOP
                NOP
                NOP
                NOP
NMI_ISR:        MVK  .S2  Nmi_isr, B0
                MVKH .S2  Nmi_isr, B0
                B    .S2  B0
                NOP
                NOP
                NOP
                NOP
                NOP
```

Figure 8. Example of Interrupt Service Table

4.2 C Language Convention

When creating an interrupt vector table that works with a C program, it is critical that the standard C convention is followed.

The C compiler run-time support library automatically creates the function _c_int00 when the –c or –cr linker options are invoked. This function corresponds to the entry point of the C program, and the reset vector must be set up to branch to _c_int00.

When a C function is interrupted by an interrupt, no CPU registers are stored to memory. If any registers are required to be used by an interrupt vector, the values currently in the registers must be pushed onto the stack before being modified. Once the interrupt servicing is complete, they must be popped back off into their registers before returning to the function.

Interrupt service routines written in C should use the interrupt keyword. For example:

```
interrupt void myISR(void)

{
      /* Code for myISR */
            …
}
```

This keyword ensures that all registers retain their current values on completion of the function. The function will also return to the interrupt return pointer (IRP), rather than the calling function.

The vector associated with the interrupt service routine, myISR, must also follow C conventions. If a branch to a register is used, the register must first be stored to the stack before branching to the interrupt service routine, and then restored. For example, the vector for the C interrupt service routine, myISR, would be:

```
        .ref _myISR

INTx:   STW   .D2  B0,*B15--[1]       ;push B0 to stack
   ||   MVKL  .S2  _myISR, B0         ;store address of
        MVKH  .S2  _myISR, B0         ;myISR to B0
        B     .S2  B0                 ;branch to B0
        LDW   .D2  *++B15[1], B0      ;restore B0
        NOP                           ;
        NOP                           ;
        NOP  2                        ;branch occurs
```

4.2.1 32-Bit EPROM Boot

Consider the following system requirements:

- The TMS320C6000 is booting from four 8-bit external EPROM devices mapped in CE1.
- Internal program memory is located at address 0.
- The boot code and application are written in C, and everything fits into internal program memory.

To avoid having the initialized data sections copied into internal memory during the boot process, those sections are linked at address 0x01410000 (outside the first 64K bytes of CE1).

In this example, the TMS320C6201 is connected to four 8-bit EPROM devices. As shown in Figure 9, BOOTMODE[4:0] is set to 11101b, that is:

- MAP 1, internal program memory mapped at 0.
- ROM boot process, selected from a 32-bit with the default timing.

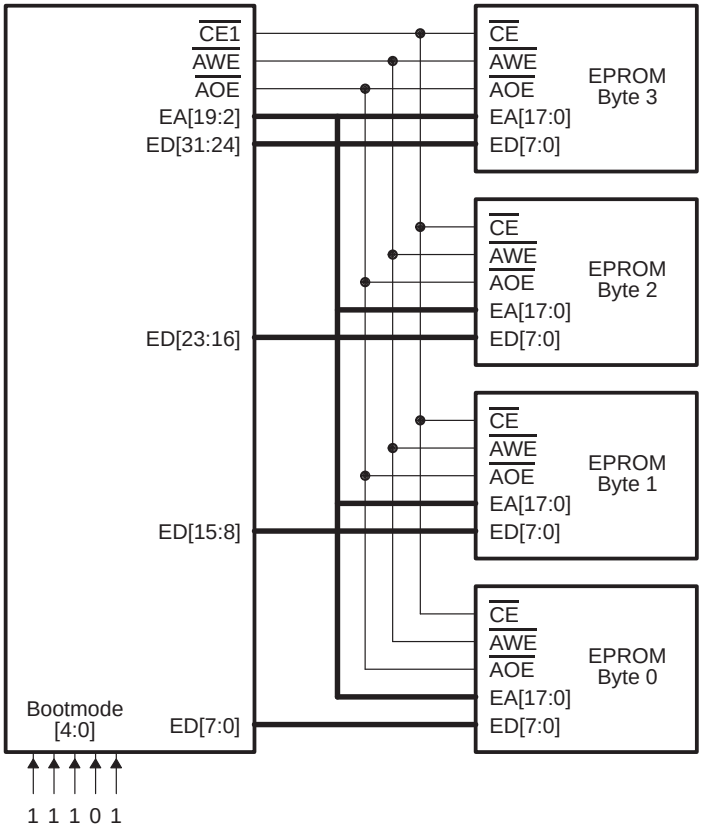

Figure 9. TMS320C6201 Connected to Four 8-Bit EPROM Memories

Figure 10 shows the linker command file that can be used for this example. Note that –c option forces _c_int00 to initialize the C environment. The object file, *vector.obj*, which consists of the interrupt vector table, is linked with *main.obj* to generate a .out file. All sections are linked with a load address in the ROM, but a run address within the on-chip memory. The *vectors* and *.text* section are automatically copied to internal program memory during the boot process; the *.data*, *.cinit*, and *.const* sections must be manually copied into data memory before use.

```
/*************************************************************/
/*  lnk.cmd                                                  */
/*  Copyright ©   1996-1997  Texas Instruments Inc.          */
/*************************************************************/
-c
vector.obj
main.obj
-o main.out
-heap  0x200
-stack 0x200
-l rts6201.lib

MEMORY
{
  VECS:    o = 00000000h  l = 00000200h
  PMEM:    o = 00000200h  l = 0000FC00h
  DMEM:    o = 80000000h  l = 00010000h
  CE0:     o = 00400000h  l = 01000000h
  CE1VECS: o = 01400000h  l = 00000200h
  CE1PMEM: o = 01400200h  l = 0000FC00h
  CE1init: o = 01410000h  l = 003F0000h
  CE2:     o = 02000000h  l = 01000000h
  CE3:     o = 03000000h  l = 01000000h

}

SECTIONS
{
  vectors :  load=CE1VECS, run=VECS
  .text   :  load=CE1PMEM, run=PMEM
  .cinit  :  load=CE1init, run=DMEM
  .const  :  load=CE1init, run=DMEM
  .data   :  load=CE1init, run=DMEM
  .cio    >  DMEM
  .far    >  DMEM
  .stack  >  DMEM
  .bss    >  DMEM
  .sysmem >  DMEM
}
```

Figure 10. Command File for the Linker

Texas Instruments provides a hex conversion utility that converts the output of the linker, a common object file format (COFF), into one of the several standards suitable for loading into an EEPROM programmer. Figure 11 shows the command file for the hex converter utility corresponding to the example. See the *TMS320C6000 Assembly Language Tools User's Guide* (SPRU186) for details.

```
main.out
-i
-byte
-image
-memwidth 32
-romwidth 8
-order L

ROMS
{
   EPROM:  org = 0x01400000, length = 0x20000,
           files = {rom.i0, rom.i1, rom.i2, rom.i3}
}
```

Figure 11. Hex Utility Command File for Little-Endian 32-Bit ROM

4.2.2 8-Bit EPROM Boot

Consider Example 1 with only one byte-wide EEPROM mapped into CE1. The hex converter command file changes, as shown in Figure 12. By specifying a memory width (memwidth) of 8, the hex utility will create a file to be loaded into a single 8-bit ROM. The data lines of the ROM are physically tied to the lower eight data bits of the C6201 EMIF.

```
main.out
-i
-byte
-image
-memwidth 8
-romwidth 8
-order L

ROMS
{
   EPROM:  org = 0x01400000, length = 0x20000,
           files = {rom.i0}
}
```

Figure 12. Hex Utility Command File for Little-Endian 8-Bit ROM

4.2.3 8-Bit Big-Endian EPROM Boot

Consider Example 2 again, this time for the C6211, with the program compiled in big-endian mode. The linker file is almost identical to that of the previous example. The memory listing is modified to reflect that of a C6211, and the big endian run-time support library is referenced (rts6201e.lib). See Figure 13.

```
/************************************************************/
/*  lnk.cmd                                                 */
/*  Copyright ©   1996-1997  Texas Instruments Inc.         */
/************************************************************/
-c
vector.obj
main.obj

-o main.out
-heap  0x200
-stack 0x200
-l rts6201e.lib

MEMORY
{
  VECS:    o = 00000000h  l = 00000200h
  PMEM:    o = 00000200h  l = 00007E00h
  DMEM:    o = 00008000h  l = 00008000h
  CE0:     o = 80000000h  l = 01000000h
  CE1VECS: o = 90000000h  l = 00000200h
  CE1PMEM: o = 90000200h  l = 00007E00h
  CE1init: o = 90008000h  l = 003F8000h
  CE2:     o = A0000000h  l = 01000000h
  CE3:     o = B0000000h  l = 01000000h

}

SECTIONS
{
  vectors :  load=CE1VECS, run=VECS
  .text   :  load=CE1PMEM, run=PMEM
  .cinit  :  load=CE1init, run=DMEM
  .const  :  load=CE1init, run=DMEM
  .data   :  load=CE1init, run=DMEM
  .cio    >  DMEM
  .far    >  DMEM
  .stack  >  DMEM
  .bss    >  DMEM
  .sysmem >  DMEM
}
```

Figure 13. Command File for the Linker

Because the C6211 expects the byte ordering of the ROM to match the endianness of the device, the –order option must be modified to provide a big-endian output. This change is shown in Figure 14. The hex utility creates a file to be loaded into a single 8-bit ROM. The data lines of the ROM are physically tied to the upper eight data bits of the C6211 EMIF.

```
main.out
-i
-byte
-image
-memwidth 8
-romwidth 8
-order M

ROMS
{
   EPROM:  org = 0x01400000, length = 0x20000,
           files = {rom.i0}
}
```

Figure 14. Hex Utility Command File for Big-Endian 8-Bit ROM

5 References

1. *TMS320C6000 Peripherals Reference Guide* (SPRU190).
2. *TMS320C6000 CPU and Instruction Set Reference Guide* (SPRU189).
3. *TMS320C6000 Assembly Language Tools User's Guide* (SPRU186).
4. *TMS320C6000 EMIF to External Asynchronous SRAM Interface* (SPRA542).
5. *TMS320C6000 Optimizing C Compiler User's Guide* (SPRU187).
6. AT29LV020 Data Sheet, Atmel Corporation, http://www.atmel.com

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

Copyright © 2004, Texas Instruments Incorporated