Abstract

The Texas Instruments (TI) 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 ROM memory space or 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 Tools: Vector Table and Boot ROM Creation .............................................................. 2
ROM Boot Process .............................................................................................................................. 3
EMIF-ROM Interface ......................................................................................................................... 3
Vector Table (or Interrupt Service Table) .......................................................................................... 6
Creating a Vector Table ................................................................................................................. 6
C Language Convention .................................................................................................................. 7
Example 1 ......................................................................................................................................... 8
Example 2 ......................................................................................................................................... 11
Example 3 ......................................................................................................................................... 12

Figures

Figure 1. TMS320C6000 ROM Boot Process .................................................................................... 3
Figure 2. 16-Bit ROM Little-Endian Packing ................................................................................... 4
Figure 3. 8-Bit ROM Little-Endian Packing ...................................................................................... 4
Figure 4. 16-Bit ROM Big-Endian Packing ....................................................................................... 5
TMS320C6000 Tools: Vector Table and Boot ROM Creation

The TMS320C6000 devices can be configured to one of several boot processes:

- **No boot:** The CPU starts direct execution at address 0.
- **ROM boot:** A memory block is automatically copied from external ROM in CE1 to memory located at address 0.
- **Host boot:** A host processor (connected to the DSP through either the host port interface (HPI) or the host interface of the expansion bus) initializes DSP memory before releasing the DSP from reset.

In addition, the user must select the memory map (when multiple maps are available) and the type of memory mapped at address 0. The boot configuration is determined by latching signals `BOOTMODE[4:0]` with the rising edge of /RESET. The TMS320C6201 and C6701 devices have dedicated `BOOTMODE` pins to determine their configuration.

The C6202 latches its boot configuration from five data lines on the expansion bus, `XD[4:0]`. These data pins map directly to `BOOTMODE[4:0]` during reset.

The C6211 and C6711 latch their boot configuration from the HPI data lines. Only two of the five configuration lines are required because the C6211 and C6711 have only one memory map, with internal memory at address 0. The `HD[4:3]` pins map to the `BOOTMODE[4:3]` pins.

Please see Chapter 10 of the *TMS320C6000 Peripherals Reference Guide* 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
ROM Boot Process

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

On the C6201, C6701, and C6202, the size of the memory block transferred is 64K bytes; 1K bytes are transferred on the C6211 and C6711.

Figure 1. TMS320C6000 ROM Boot Process

For the C6201, C6701 and C6202, 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, SBSRAM, or asynchronous memory.

The C6211 and C6711 have one memory map and the ROM boot process copies 1K bytes from the beginning of CE1 to address 0, using default ROM timings. Internal memory is always located at address 0.

EMIF-ROM Interface

The ROM interface to the C6000 DSP is dependant on the particular DSP within the platform. All C6000 DSPs support 8-, 16-, and 32-bit ROM. 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 C6201, C6701, and C6202 always expect the ROM to be in little-endian format and on the lowest end of the EMIF data bus. The C6211 and C6711 expect the ROM to be in the same format as the device and its placement on the data bus to coincide with endianness as well.
C6201/C6701/C6202 EMIF

The C6201/C6701/C6202 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 C6201/C6701/C6202 EMIF packs the 8- or 16-bit data into a 32-bit word 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**

```
+-------+-------+-------+-------+
|       |       |       |       |
|       |       |       |       |
|       |       |       |       |
|       |       |       |       |
|       |       |       |       |
|       |       |       |       |
|       |       |       |       |
|       |       |       |       |
|       |       |       |       |
|       |       |       |       |
+-------+-------+-------+-------+
```

**Figure 3. 8-Bit ROM Little-Endian Packing**

```
+-------+-------+-------+-------+
|       |       |       |       |
|       |       |       |       |
|       |       |       |       |
|       |       |       |       |
|       |       |       |       |
|       |       |       |       |
|       |       |       |       |
|       |       |       |       |
|       |       |       |       |
+-------+-------+-------+-------+
```

Because the packing methodology of the C6201/C6701/C6202 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 halfword first. The byte ordering within a ROM is selectable with an option to the hex utility and is described later in this document.
C6211/C6711 EMIF

The C6211/C6711 EMIF is designed to function with 8-, 16-, and 32-bit memories of all supported types (SDRAM, SBSRAM, and asynchronous). Endianness is therefore always taken into consideration when packing is performed on a data access. When operating in little-endian mode, the C6211/C6711 EMIF packing is identical to that of the C6201/C6701/C6202 EMIF. ROM should therefore be connected as shown in Figure 2 and Figure 3 in little-endian C6211/C6711 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 halfword 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 C6211/C6711 system is shown in Figure 4. The interconnection of an 8-bit ROM for a big-endian C6211/C6711 system is shown in Figure 5.

Figure 4. 16-Bit ROM Big-Endian Packing

Figure 5. 8-Bit ROM Big-Endian Packing

For additional details about the connection between a C6000 DSP and ROM device, please see the TMS320C6000 Peripherals Reference Guide and TMS320C62x/C67x CPU and Instruction Set Reference Guide.
Vector Table (or Interrupt Service Table)

The processor is placed in reset when /RESET is driven low. On the rising edge of /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.

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:

\[
\text{Address} = \text{ISTP} + 32 \times \text{<interrupt number>}
\]

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

Each vector must contain eight instructions. If an 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 (includes the reset vector) is shown in Figure 6. (See the TMS320C62x/C67x CPU and Instruction Set Reference Guide for additional details about the interrupt service table.)

---

\(^1\) There are 16 CPU interrupts available. Interrupts 16 through 31 are reserved.
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 on to 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:

```c
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:
Example 1

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. As shown in Figure 7, 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 7. TMS320C6201 Connected to Four 8-Bit EPROM Memories

Figure 8 shows the linker commands file that can be used for this example. Note –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.
Figure 8. Command File for the Linker

```plaintext
/*****************************/
/* 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
}

Texas Instruments provides a hex conversion utility that converts the output of the linker (a COFF object file) into one of the several standards suitable for loading into an EEPROM programmer. Figure 9 shows the command file for the hex converter utility corresponding to our example. (See Chapter 9 of the TMS320C6x Assembly Language Tools User’s Guide for details.)
Figure 9. Hex Utility Command File for Little-Endian 32-Bit ROM

```
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}
}
```

Example 2

Consider Example 1 with only one byte-wide EEPROM mapped into CE1. The hex converter command file changes, as shown in Figure 10. 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.

Figure 10. Hex Utility Command File for Little-Endian 8-Bit ROM

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

ROMS
{
  EPROM:  org = 0x01400000, length = 0x20000,
          files = {rom.i0}
}
```
Example 3

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 11.

Figure 11. Command File for the Linker

```c
/**
  * 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
}
```

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

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

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

References

*TMS320C6000 Peripherals Reference Guide*, Literature number SPRU190, March 1999, Texas Instruments

*TMS320C62x/C67x CPU and Instruction Set Reference Guide*, Literature number SPRU189, March 1999, Texas Instruments


IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

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

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.

Copyright © 2000, Texas Instruments Incorporated