Creating a vector table and boot ROM for the TMS320C6201

APPLICATION REPORT: PRELIMINARY

Author : Eric Biscondi
LBE : DSP
Date : Tuesday, March 24, 1998

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software 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.

Certain application using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

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

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI 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.

TRADEMARKS

TI is a trademark of Texas Instruments Incorporated.

Other brands and names are the property of their respective owners.

Contents

Figures

Creating a vector table and boot ROM code for the TMS320C6201

Abstract

Three types of boot processes are available on the TMS320C6201. The boot process is determined by the BOOTMODE[4:0] pins.

This document describes:

- the ROM Boot process,
- how to create a vector table,
- how to build a ROM boot code in C and Assembly language through an example.

Overview

The ‘C6201 uses various types of boot configuration. There are three types of boot process:

- The CPU starts direct execution at address 0.
- A 16K 32-bit words memory block is automatically copied from the beginning of the CE1 memory space to memory located at the address 0 through the DMA channel 0.
- A Host processor (connected to the ‘C6201 through the Host Port Interface) maintain the ‘C6201 core in reset while initializing the ‘C6201 memory space, including external memory spaces.

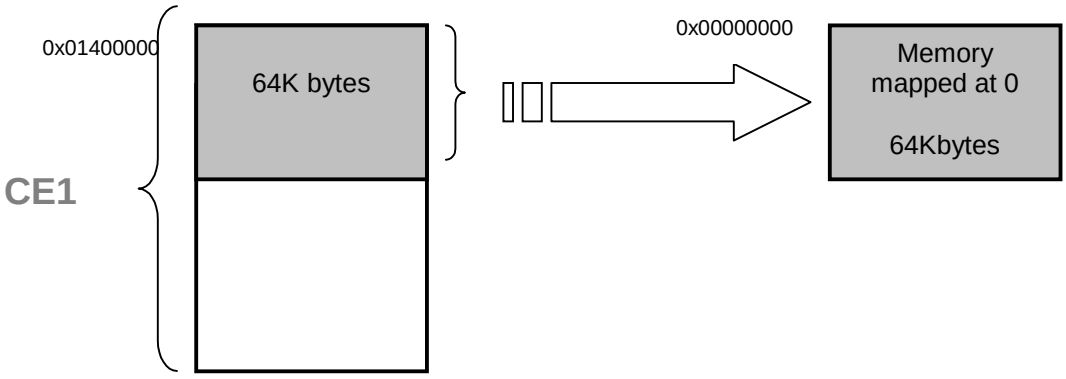
In addition to these three modes, the user need also to select the memory map and the type of memory which is mapped at the address 0. This setting is determined by the BOOTMODE[4:0] pins during the reset. Please refer to the section 7.3 of the TMS320C62xx Peripherals Reference Guide for a complete description of the BOOTMODE[4:0] pins.

The following chapters describe ROM boot process and how to create a vector table for assembly and C framework, how to build a code to be downloaded using the ROM boot process.

ROM Boot Process

During ‘C6201 reset, the 64Kbyte memory block mapped at the beginning of the memory space CE1 is transferred to the memory located at the address 0, as shown in Figure 1. DMA channel 0 performs that transfer. Once channel 0 has performed the transfer, the CPU is removed from reset and allowed to start from the memory location 0.

Figure 1. ROM Boot process

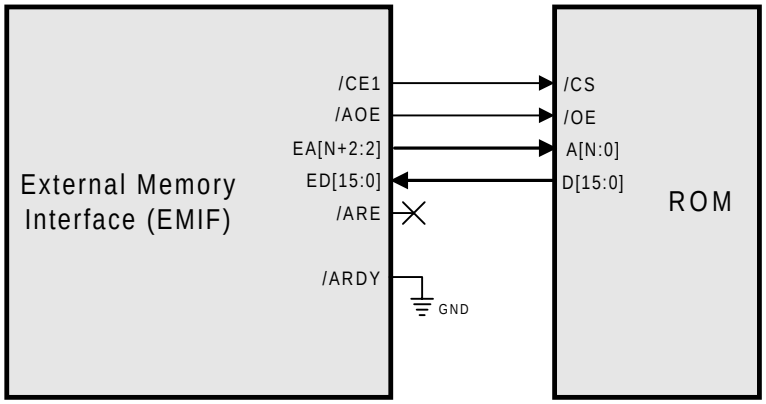


The memory mapped at address 0 can either be the internal program memory (MAP1) or the external memory space CE0 (MAP0), which may contain SDRAM, SBSRAM or Asynchronous memory.

Typically a ROM device is connected to the TMS3206201, as shown on the Figure 2, and mapped at the beginning of the memory space CE1.

Notice that the values stored in the memory space CE1 are expected to be in little endian format.

Figure 2. EMIF-ROM interface (16-bit)



For any further details about the connection between TMS320C6201 and ROM devices, please refer to [1] section 6.5 and [2].

Vector table (or Interrupt Service Table)

When the RESET pin on the processor is driven low, then high, the device is reset. Once the ROM boot process is done (i.e. DMA channel 0 has completed data transfer from CE1 to memory mapped at 0) , 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. Reset is the highest priority interrupt. At the end of all interrupt sequence, the CPU loads the vector address from the Interrupt Service Table (IST) to the program counter. The IST is a table containing code for servicing the interrupts. The reset vector is always located at the address 0.

Creating a vector table

As a fetch packet contains eight 32-bit instructions, each vector is aligned on a fetch packet boundary that means each packet must contain eight instructions. Each vector may either contain the branch to the interrupt service routine (with some padding NOPs or fill them with setup code for the interrupt service routine), or may contain the complete interrupt service routine if that one is less than eight instructions. On the Figure 3, the Interrupt Service Table is included in the code section vectors, which is typically linked at the address 0. Refer to [2] for any further details about the Interrupt service table.

Figure 3. Example of Interrupt Service Table

```
          .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
          .
                .
```

Creating a vector table respecting the C language conventions

Because each vector included into the interrupt vector table has to be aligned on a fetch packet boundary, the vector table is always written in assembly language. When C language is used for the application framework, C conventions have to be respected when writing the vector table.

The C compiler run-time support library is automatically creating a function, *_c_int00*, when the –c or –cr linker options are invoked. This function correspond to the entry point of the C program and the reset vector needs to be setup to branch to *_c_int00*. The Annex A give a complete example of table vector respecting the C convention.

The *interrupt* keyword allows the user to write interrupt service routine in C language. For example:

```
interrupt void myISR(void)
{
   /* Code for myISR */
   …
}
```

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

```
      .ref    _myISR

INTx: STW   .D2  B0,*B15--[1]  ;push B0 to stack
   || MVK   .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
```

Creating a Boot ROM code

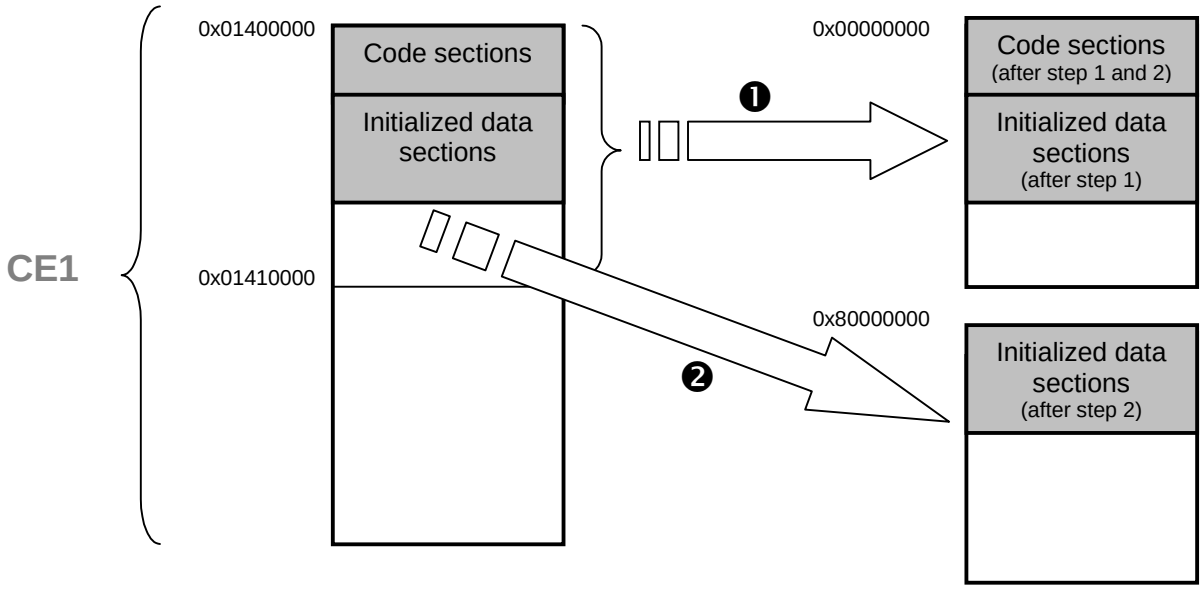
The boot code is the piece of code that is transferred during the ROM boot process to memory at address 0. When the complete application is small enough (less than 64 Kbytes), the boot code corresponds to the whole application. First, the boot code takes care of the system initialization. The boot code might be written either in assembly or C language. To initialize variables, the boot code needs to copy initial values to variables. The user needs to take care of this task when writing code in assembly. With C language, the routine *_c_int00*, which is included in the runtime-support library, performs the variable initialization automatically when you use the –c linker option, then it branches to the *main* function. (Cf. to [4] sections 8.8 System Initialization).

A particular attention needs to be taken when using the ROM boot process in MAP 1. TMS320C6201's CPU can not perform data accesses to/from on-chip program memory. Initialized data sections have to be linked in the CE1 memory space.

As shown on the Figure 4, step ① is performed automatically by the ROM boot process. Once the 64Kbyte memory block is transferred (step ①), CPU starts to execute the program at address 0, which is in charge of copy initialized value to variables, step ②.

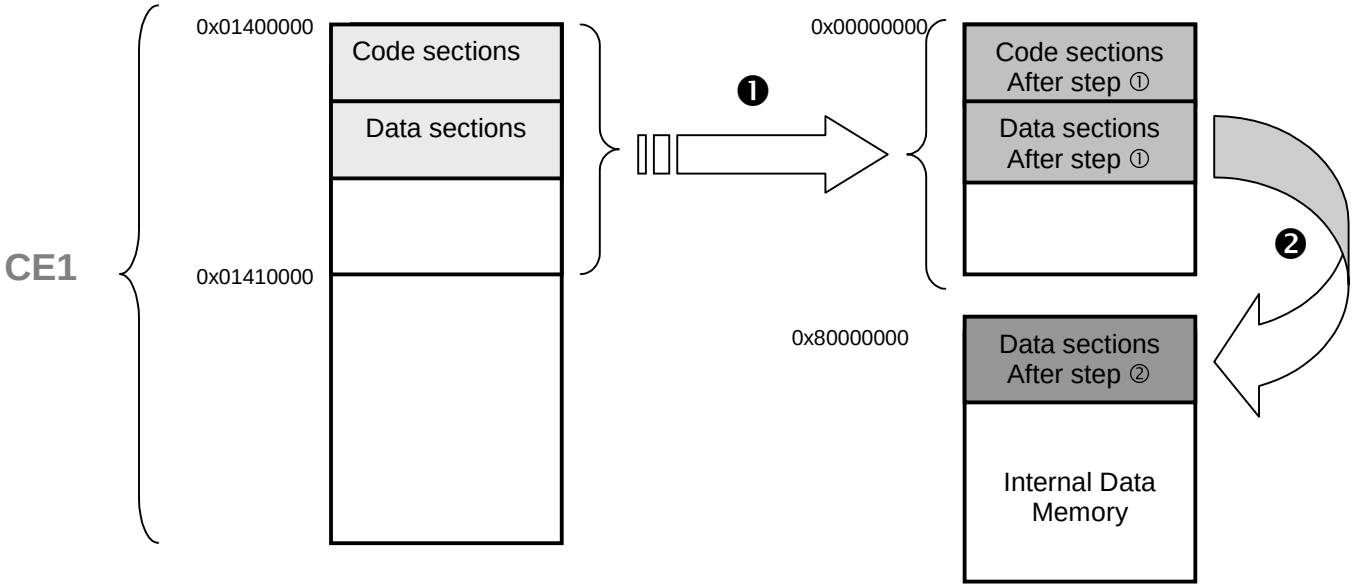
To avoid getting the initialized data sections into internal program memory after the step 1, the user may choose to link those sections outside the first 64kbytes of CE1. (i.e. at address 0x01410000 or after).

Figure 4. Sections organization for a ROM boot code with MAP1



When configuring the TMS320C6201 with memory map 0, there is no restriction regarding where code and initialized data sections are linked within CE1. Once the CPU is removed from reset and it starts from the memory location 0, which is memory space CE0, and may also access the initialized data sections stored into CE0.

Figure 5. Sections organization for a ROM boot code with MAP0



As shown on the Figure 5, because when using MAP 0 the CPU is able to perform data accesses to CE0, the initialized values may be copied with all the code sections by the ROM boot process. Then, once CPU is released from the reset state, it starts at address 0 and runs the boot code previously downloaded from CE0. Boot code performs the system initialization by reading and copying initialized values to the on-chip data memory. (Step ②)

Step ② is automatically performed by the function *_c_int00* when using the C compiler and the linker option –cr.

Example

Let’s consider the system requirements are the following:

- The TMS320C6201 is booting from four 8-bit external EPROM mapped in CE1 memory space.
- Memory map MAP1 is used, i.e. once the boot code has been transferred the CPU will start to run at address 0, which is the internal on-chip memory.
- The boot code and the application are written in C and every thing fits into internal program memory.

To avoid getting the initialized data sections copied into internal memory after the boot process, those sections are linked at address 0x01410000. (Outside the first 64Kbytes of CE1).

In this example, the TMS320C6201 is connected to four 8-bit EPROM. As shown on the Figure 6, BOOTMODE[4:0] is equal to 11101, i.e. :

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

Figure 6. TMS320C6201 connected to four 8-bit EPROM Memories

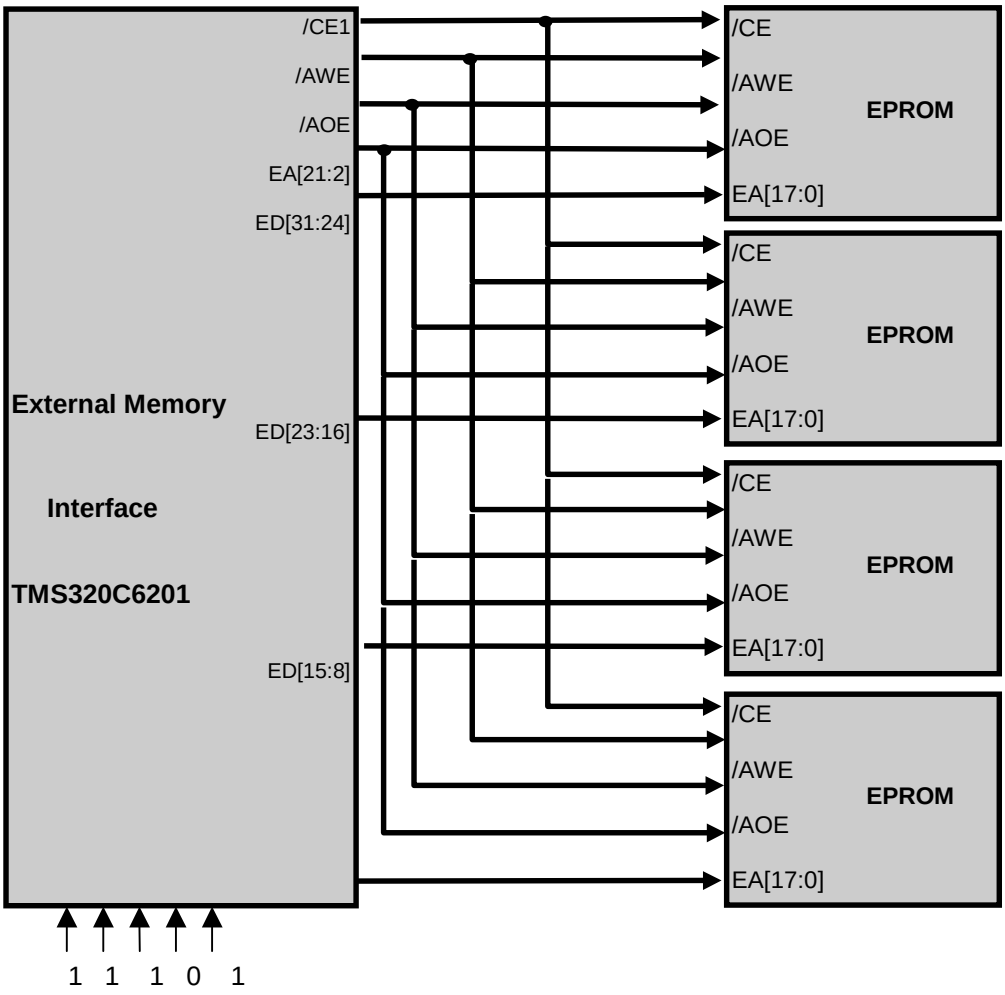


Figure 7 gives the linker commands file which may be used for this example. Note –c option forces *_c_int00* to initialize the C environment. The file *vector.asm*, which contains for example the table vector included in Annex A, is linked with *main.obj. cinit* and *const* are mapped in the memory space CE1.

Figure 7. Command file for the linker

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


-o main.out
-heap  0x0200
-stack 0x0200
-l rts6201.lib

MEMORY
{
   VECS:  o = 00000000h    l = 0000200h
   PMEM:  o = 00000200h    l = 000FC00h
   DMEM:  o = 80000000h    l = 0010000h
   CE0:   o = 00400000h    l = 1000000h
   CE1:   o = 01400000h    l = 0010000h
   CE1init:o = 01410000h   l = 0010000h
   CE2:   o = 02000000h   l = 1000000h
   CE3:   o = 03000000h   l = 1000000h
}

SECTIONS
{
    vectors >       VECS
    .text   >       PMEM
    .far    >       DMEM
    .stack  >       DMEM
    .bss    >       DMEM
    .sysmem >       DMEM
    .cinit  >       CE1init
    .cio    >       DMEM
    .const  >       CE1init
    .data   >       DMEM
}
```

Texas Instruments is providing a hex conversion utility, which converts the output of the linker (a COFF object file) into one of the several standards suitable for loading into an EEPROM programmer. Figure 8 shows the command file for the hex converter utility corresponding to our example. Refer to [7] Chapter 9 for any further details.

Figure 8. Command file for the hex converter utility

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

ROMS
{
   EPROM:  org = 0x0, length = 0x20000
   files = {u22.int, u24.int, u23.int, u25.int}
}
SECTIONS
{
   .text:    paddr=0x00000
   .cinit:   paddr=0x10000
   .const:   paddr=0x10000
}
```

References

[1]. TMS320C62xx Peripherals Reference Guide, Texas Instruments 1997.

[2]. TMS320C62xx CPU and Instruction Set Reference Guide, Texas Instruments 1997.

[3]. Application Report : Interfacing TMS320C62xx to external Asynchronous SRAM, Texas Instruments 1998.

[4]. TMS320C6x Optimizing C Compiler User's Guide, Texas Instruments 1997.

[5]. Data Sheet: AT29LV020, Atmel.

[6]. Application Report: Interfacing the TMS320C62xx to External Flash Memory.

[7]. TMS320C6x Assembly Language Tools – User's Guide, Texas Instruments 1997.

Annexes

Annex A

Example of vector table supporting C language.

```
/*********************************************
**/
/* vectors.asm:                               */
/* TMS320C6201 vector table                   */
/* supporting C conventions                   */
/*                                            */
/*                  © Texas Instruments       */
/*********************************************
*/

        .ref
        _c_int00,_c_nmi01,_c_int04,_c_int05,
        .ref
        _c_int06,_c_int07,_c_int08,_c_int09,
        .ref
        _c_int10,_c_int11,_c_int12,_c_int13
        .ref  _c_int14, _c_int15

        .sect vectors

RESET:      B  .S2       _c_int00
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP

NMI:        B  .S2       _c_nmi01
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP

RESV1:      B  .S2       RESV1
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP

RESV2:      B  .S2       RESV2
            NOP
```

```
          NOP
          NOP
          NOP
          NOP
          NOP
          NOP

INT4:     B  .S2      _c_int04
          NOP
          NOP
          NOP
          NOP
          NOP
          NOP
          NOP

INT5:     B  .S2      _c_int05
          NOP
          NOP
          NOP
          NOP
          NOP
          NOP
          NOP

INT6:     B  .S2      _c_int06
          NOP
          NOP
          NOP
          NOP
          NOP
          NOP
          NOP

INT7:     B  .S2      _c_int07
          NOP
          NOP
          NOP
          NOP
          NOP
          NOP

          NOP

INT8:     B  .S2      _c_int08
          NOP
          NOP
          NOP
          NOP
          NOP
          NOP
          NOP

INT9:     B  .S2      _c_int09
          NOP
          NOP
          NOP
          NOP
          NOP
          NOP
          NOP

INT10:    B  .S2      _c_int10
          NOP
          NOP
          NOP
          NOP
          NOP
          NOP
          NOP

INT11:    B  .S2      _c_int11
          NOP
          NOP
          NOP
          NOP
          NOP
          NOP
          NOP

INT12:    B  .S2      _c_int12
          NOP
          NOP
```

```
            NOP
            NOP
            NOP
            NOP
            NOP

INT13:      B  .S2       _c_int13
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP

INT14:      B  .S2       _c_int14
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP

INT15:      B  .S2       _c_int15
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP
```