Setting Up TMS320C6201 Interrupts in C

Carlos A.P.N. Solis

World Wide Broadband Access Group
LAN Products
DSP Networking

Abstract

How do I set and use interrupts on C for the Texas Instruments (TI™) TMS320C6x DSP?

Writing Interrupt Service Routines (ISRs) in C is straightforward as long as you follow the simple rules set out in this document. The problem consists of four parts:

- ❒ Selecting the interrupt source and writing the ISR
- ❒ Creating and initializing the interrupt vector table
- ❒ Setting the proper registers to enable and process the interrupt(s)
- ❒ Linking the parts together in the linker command file

Clearly there are variations on this theme. Some ISRs can be written in C and some in assembly as long as the declaration conventions and vector tables are followed and initialized.

Contents

Figures

Design Problem

How do I use interrupts from C?

Solution

There are four parts to this problem:

- Selecting the interrupt source and writing the ISR
- Creating and initializing the interrupt vector table
- Setting the proper registers to enable and process the interrupt(s)
- Linking the parts together in the linker command file

Interrupt Service Table (IST) and Interrupt Service Table Pointer Register (ISTP)

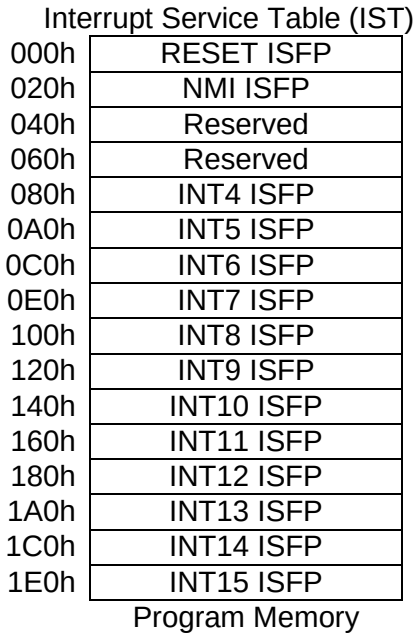
When the CPU begins processing an interrupt, it references the interrupt service table (IST). The IST is a table of fetch packets containing code to service the interrupts. The IST consists of 16 consecutive fetch packets. Each interrupt service fetch packet (ISFP) contains eight instructions. A simple interrupt service routine may fit in an individual fetch packet.

Figure 1 shows the addresses and contents of the IST. Because each fetch packet contains eight 32-bit instruction words (or 32 bytes), each address in the table is incremented by 32 bytes (20h) from the one adjacent to it.

The interrupt service table pointer (ISTP) register is used to locate the interrupt service routine. One of its fields, ISTB, identifies the base portion of the address of the IST; another field, HPEINT, identifies the specific interrupt and locates the specific fetch packet within the IST.

The reset fetch packet must be located at address 0, but the rest of the IST can be at any program memory location that is on a 256-word boundary. The interrupt service table base (ISTB) field of the ISTP determines the location of the IST.

Figure 1. Interrupt Service Table



Interrupt Service Fetch Packet (ISFP)

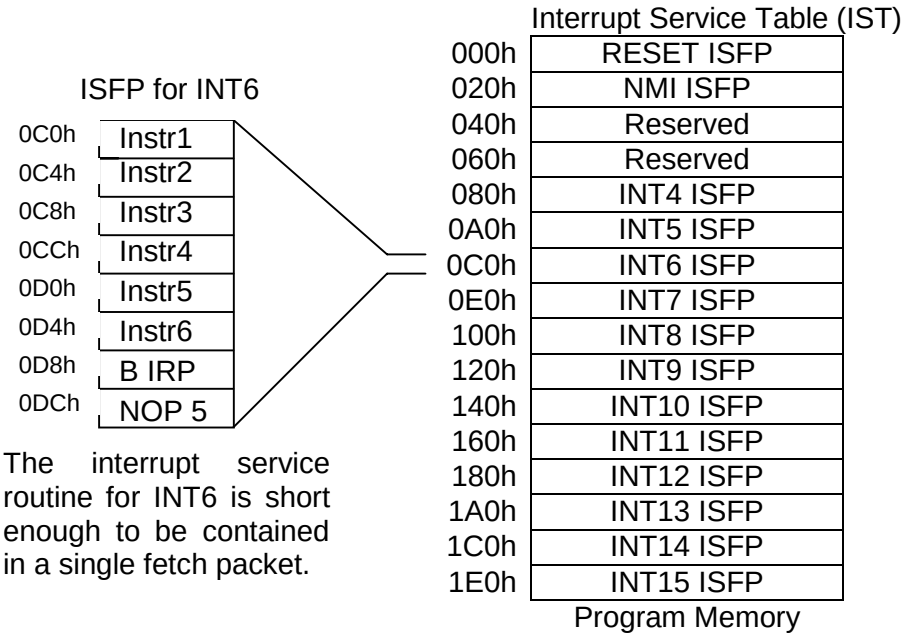
An ISFP is a fetch packet used to service an interrupt. Figure 2 shows an ISFP that contains an interrupt service routine small enough to fit in a single fetch packet (FP). To branch back to the main program, the FP contains a branch to the interrupt return pointer instruction (**B IRP**). This is followed by a **NOP** 5 instruction (5 No_operation instructions) to allow the branch target to reach the execution stage of the pipeline.

NOTE:
If the **NOP** 5 was not in the routine, the CPU would execute the next five execute packets that are associated with the next ISFP.

If the interrupt service routine for an interrupt is too large to fit in a single FP, a branch to the location of additional interrupt service routine code is required.

If you write your code in c, the C compiler takes care of all this (generation of the IST, information on the ISTP and allocation of the ISFPs for each ISR), but if you write it in assembly, then you have to take care of it by yourself.

Figure 2. Interrupt Service Fetch Packet



Creation and Installation of the Interrupt Vector Table

Because each vector included into the interrupt vector table must 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 must be respected when writing the 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 less than eight instructions. The next example shows the interrupt service table included in the code section vectors, which is typically linked at address 0. (Refer to the TI *TMS320C62x/C67x CPU and Instruction Set Reference Guide* for details about the interrupt service table.)

The C compiler run-time support library automatically creates a function, _c_int00, when the –c or –cr linker options are invoked. This function corresponds to the entry point of the C program. The reset vector must be set up to branch to _ c_int00.

The following example shows one way to initialize and install the interrupt vector table. To do this, you must reserve space for the interrupt vector table (".sect" command in an assembly language file) and tell the linker where in memory you want to install the interrupt vector table.

The following file (intr_.asm) is included with the 'C6x peripheral support library; therefore, you do not need to create it if working without an OS. You must only link intr_.asm with the rest of your code files (the "intr.c" and "regs.h" files, which are mentioned in the "intr_.asm" file, are also provided as part of the 'C6x peripheral support library, devlib6x.src). You must also compile and link them with the rest of your program files.

```
;   intr_.asm
;
;    This file provides run time installable ISR capability through the use of the intr_jump_table
;    which is defined in intr.c.  This file provides the ISFPs (Interrupt Service Fetch Packets)
;    for the IST (Interrupt Service Table).  If the address in the jump table index corresponding
;    to the interrupt service CPU interrupt is 0, no branch is executed and control is returned
;    to the previous thread.
;
                .ref        _c_int00              ; reset ISR
                .ref        _isr_jump_table    ; defined in intr.c : Interrupt Service Routine Jump Table
;near unsigned int isr_jump_table[16] ={ (unsigned int)c_int00, 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
                .global     _istb                 ; interrupt service table base, defined in regs.h

                .text                             ;compile in code section
                .sect       ".vec"                ;space allocation for the vector table
_istb:
                mvk         _c_int00,b0  ; c_int00 is resolved at compile time
                mvkh        _c_int00,b0  ; c_int00 must be the entry point when programming in C
                b           .s2 b0            ; c_int00 is the entry point for boot.obj
                nop         5
                nop
                nop
                nop
                nop

                .asg        1, vec
                .loop       15

                stw     .d2   b0, *--b15

                ldw     .d2   *+b14(_isr_jump_table + vec * 4), b0

                nop         4

             [b0]  b      .s2   b0

                .if ( vec == 1 )

             [!b0] b      .s2   nrp        ;NMI ISR
            ||      ldw     .d2   *b15++, b0

                .else

             [!b0] b      .s2   irp        ;Non-NMI ISR
            ||      ldw     .d2   *b15++, b0

                .endif

                nop         4
                nop         1

                .eval       vec + 1, vec
                .endloop
```

A C Language ISR

In a typical DSP system, hardware interrupts are triggered either by devices external to the DSP or by on-chip peripherals. In either case, the interrupt causes the processor to vector (jump) to the ISTB entry.

Hardware ISRs may be written in C, assembly language, or a combination of both. But since ISRs written in C require the entire C context to be saved (TI's C compiler takes care of that), highly time-critical ISRs are usually written in either linear assembly or assembly language.

The C compiler requires that each ISR be declared in either of the following ways:

- The 'C6x compiler extends the C language by adding the **interrupt** keyword, which specifies that a function is treated as an interrupt function.

  Functions that handle interrupts follow special register-saving rules and a special return sequence. When C code is interrupted, the interrupt routine must preserve the contents of all machine registers used by the routine or by any function called by the routine.

  When you use the interrupt keyword with the definition of the function, the compiler generates register saves based on the rules for interrupt functions and the special return sequence for interrupts. You can only use the interrupt keyword with a function that is defined to return void and that has no parameters. The body of the interrupt function can have local variables and is free to use the stack or global variables.

  For example:

  ```
  interrupt void int_handler()
  {
  unsigned int flags;
  ...
  }
  ```

  A good practice is to include the interrupts in a separate file called ints.c or something similar. This makes for a more modular style, simpler maintenance, and easier to understand software.

- Another way to define ISR is by using the **INTERRUPT pragma**. The **INTERRUPT pragma** enables you to handle interrupts directly with C code. The argument *func* is the name of a function. The pragma syntax is:

  #pragma INTERRUPT (*func*);

  For example:

  #pragma INTERRUPT(int_handler)

Setting the Interrupt Control Registers

There are eight interrupt control registers on the 'C62x devices. The control status register (CSR) and the interrupt enable register (IER) enable or disable interrupt processing. The interrupt flag register (IFR) identifies pending interrupts. The interrupt set register (ISR) and interrupt clear register (ICR) can be used in manual interrupt processing.

There are three pointer registers. ISTP points to the interrupt service table. NRP and IRP are the return pointers used when returning from a nonmaskable or a maskable interrupt, respectively.

To be able to process an interrupt, you must take the following steps (be aware that this is not the only way to do it, we are only providing an example of how you could do it).

1) Initialize the Interrupt Service Table Pointer (ISTP) based on the global vec_table that is resolved at link time (refer to linker command file for this value). The ISTP should point to the beginning of the interrupt service table (IST) at the memory address defined in the linker command file (under the "vec" section).

2) Map the interrupt source number (isn) to a cpu interrupt number (cpu_intr). You must load the proper isn value into the INTSEL field of the appropriate Interrupt multiplexer register. The 'C6x peripheral set has 16 interrupt sources; however, the CPU has 12 interrupt sources available for us. The interrupt selector allows you to choose and prioritize which 12 of the 16 your system must use.

3) Manually clear the selected interrupt by writing a 1 to the specified bit in the ICR. This ensures that there's no unwanted/unexpected data in any of the bit fields of this register (this step is not absolutely necessary but it's highly recommended)

4) Hook your ISR to the cpu interrupt signal.

5) Enable the non-maskable interrupt (NMI). If this interrupt is not enabled, the rest of the interrupts will not be seen/processed.

6) Enable the CPU interrupt number for the interrupt you mapped in step #2 and hooked in step # 4. In this step, you enable the interrupt bit in the interrupt enable register (IER).

7) Globally enable all maskable interrupts by setting the GIE bit in the control status register. If this bit is not enabled, the rest of the interrupts will not be seen/processed.

For a detailed description of all the register's structure, refer to the TI *TMS320C62x/C67x CPU and Instruction Set Reference Guide* (literature number SPRU189C) and the TI *TMS320C6201/C6701 Peripherals Reference Guide* (literature number SPRU190B).

The following lines include an example of how you can accomplish these steps using the 'C6x instructions (with functions and macros provided on the 'C6x peripheral support library). We provided the commands, a description of each command, the file name where each command is defined, and any file that needs to be called (included) when compiling the C code.

In this particular example, CPU interrupt 14 is tied to the interrupt signal generated by the CPU timer 0:

```
/* includes */
#include <intr.h>              /* Interrupts Support - 'C6x peripheral support library*/
#include <regs.h>              /* Device register support - 'C6x peripheral support library*/

/* variables definitions */

Int status;

/* To call within the main() function: */
```

```
/* Interrupts settings */

intr_init();                    // it initializes the ISTP with the address of the global label
                                // vec_table, which is defined in intr_.asm, and  resolved at link
                                // time. Defined in intr.c as a callable function, intr.c is a included
                                // with the 'C6x peripheral support library and should be compiled
                                // and linked with the rest of the program files

intr_map(CPU_INT14,ISN_TINT0);  // it places the indicated Interrupt Service Number
                                // (ISN) value in the appropriate field of the
                                // appropriate interrupt multiplexer register. Defined in
                                // intr.c as a callable function

INTR_CLR_FLAG(CPU_INT14);       // it manually clears the selected interrupt by writing
                                // a 1 to the specified bit in the ICR. This is just to
                                // be sure that there's no unwanted/unexpected
                                // data in any of the bit fields of this register.
                                // Defined in intr.h as a macro . Even though
                                // this is not absolutely necessary, it is highly
                                // recommended.

intr_hook(timerISR,CPU_INT14);  // it places the function pointer indicated by the first
                                // parameter (a pointer to an ISR declared in C) into
                                // isr_jump_table[], at the location specified by the
                                // second parameter (ISR to invoke when servicing
                                // this interrupt).

INTR_ENABLE(CPU_INT_NMI);       // it enables the non-maskable interrupt (NMI). If this
                                // interrupt is not enabled, the rest of the interrupts
                                // won't be seen/processed. Defined in intr.h as a
                                // macro.

INTR_ENABLE(CPU_INT14);         // it enables CPU interrupt 14 by enabling its bit in the
                                // the interrupt enable register (IER).We have previously
                                // mapped this interrupt number with the cpu clock 0
                                // interrupt signal. Defined in intr.h as a macro.


INTR_GLOBAL_ENABLE();           // it globally enables all maskable interrupts by setting the
                                // GIE bit in the control status register (CSR). If this bit is
                                // not enabled/set, the rest of the interrupts won't be seen
                                // nor processed. Defined in intr.h as a macro
```

That takes care of all of the registers that deal with interrupts. This program will execute the function "timerISR" whenever an CPU Timer 0 interrupt signal is triggered (you still must set the timer parameters, which is beyond the scope of this document. For more information, see the TI *TMS320C6201/C6701 Peripherals Reference Guide* (literature number SPRU190B) and the TI *TMS320X6x Peripheral Support Library Programmer's Guide* (literature number SPRU273)).

If you want **to disable** any **interrupt**(s) (e.g., when the program calls an ISR and you do not want the processor to be interrupted while it is servicing the current interrupt), you must clear the proper bit(s) on the IER register. You can do this by calling:

```
#include <intr.h>

INTR_DISABLE(bit);      // bit is the CPU interrupt (value 0 -15) that disables the specified
                        // interrupt;
```

If you want to disable multiple interrupts at once, you either must call this function with the proper bit value for each interrupt you want to disable or make the following function call:

```
#include <regs.h>

SET_REG(IER,val);       // val is the decimal value of the binary bitmask that represents the
                        // bits of the interrupts you want to disable.
```

If you want to disable all of the interrupts at once, you must make the following call:

```
#include <intr.h>

INTR_GLOBAL_DISABLE;    // It globally disables all maskable interrupts by clearing the
                        // GIE bit in the CSR.
```

Linking Everything Together

When linking all of your files, you must link with them the files provided with the peripheral support library (intr.h, intr_.asm., regs.h, etc.). To accomplish this, link the "dev6x.lib" library file by using the -l linker option when linking all of your files:

"-ldev6x.lib".

On your "linker command file" you must include the following sections:

/* file name == mylink.cmd */

The MEMORY section must identify the location of the int vectors.

;For Memory Map 1:

```
MEMORY
{
VECTORS: origin = 0h, length = 200h ; internal program memory (IPM)
...
}
```

;For Memory Map 0:

```
MEMORY
{
VECTORS: origin = 01400000h, length = 200h ; internal program memory (IPM)
...
}
```

The SECTIONS section needs to map the user-defined section called “vec” to the memory location.

```
SECTIONS
{
  vec : > VECTORS
...
}
```

If you plan to work with SPOX, you must use the functions and macros provided with SPOX (in the file **c62.h**); otherwise, you could have problems with function calls being incompatible. Refer to the *SPOX-KNL Programming Guide TMS320C62x*, version 2.2 or newer, for a detailed description of how to work with interrupts on SPOX.

Summary

Writing interrupt routines in C is straightforward as long as you follow the simple rules set out in this document. You must make sure to generate the interrupt vector table (either by writing your own assembly file, using the SPOX utilities, or by any other means). You must also provide the linker with all of the necessary information to link the ISRs, vector table, and section names into the correct locations. Clearly there are variations on this theme: some ISRs can be written in C and some in assembly as long as the declaration conventions and vector tables are followed and initialized.

TI Contact Numbers

INTERNET

TI Semiconductor Home Page
www.ti.com/sc

TI Distributors
www.ti.com/sc/docs/distmenu.htm

PRODUCT INFORMATION CENTERS

Americas

Phone	+1(972) 644-5580
Fax	+1(972) 480-7800
Email	sc-infomaster@ti.com

Europe, Middle East, and Africa

Phone	
Deutsch	+49-(0) 8161 80 3311
English	+44-(0) 1604 66 3399
Español	+34-(0) 90 23 54 0 28
Francais	+33-(0) 1-30 70 11 64
Italiano	+33-(0) 1-30 70 11 67
Fax	+44-(0) 1604 66 33 34
Email	epic@ti.com

Japan

Phone	
International	+81-3-3457-0972
Domestic	0120-81-0026
Fax	
International	+81-3-3457-1259
Domestic	0120-81-0036
Email	pic-japan@ti.com

Asia

Phone	
International	+886-2-23786800
Domestic	
Australia	1-800-881-011
TI Number	-800-800-1450
China	10810
TI Number	-800-800-1450
Hong Kong	800-96-1111
TI Number	-800-800-1450
India	000-117
TI Number	-800-800-1450
Indonesia	001-801-10
TI Number	-800-800-1450
Korea	080-551-2804
Malaysia	1-800-800-011
TI Number	-800-800-1450
New Zealand	000-911
TI Number	-800-800-1450
Philippines	105-11
TI Number	-800-800-1450
Singapore	800-0111-111
TI Number	-800-800-1450
Taiwan	080-006800
Thailand	0019-991-1111
TI Number	-800-800-1450
Fax	886-2-2378-6808
Email	tiasia@ti.com

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 acknowledgement, 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.

CERTAIN APPLICATIONS 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, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF TI PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE FULLY AT THE CUSTOMER'S RISK.

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.