Application Report
SPRA495

Setting up TMS320C2xx Interrupts in Assembler or C

Frank Zeller *Digital Signal Processoring Solutions*

Typically, interrupts are generated by devices that need to give or take data from the Texas Instruments (TI™) TMS320C2xx, for example, analog-to-digital (A/D) and digital-to-analog (D/A) converters or other processors. When the C2xx recognizes an interrupt signal, it suspends execution of the code specific to the particular interrupt event (Interrupt Subroutine).

The C2xx family supports three or four different sorts of interrupts:

- ❑ Maskable external interrupts (INT1-3)
- ❑ Non-maskable external interrupts (NMI, RS)
- ❑ Maskable internal interrupts (TINT, RINT, XINT, TXRXINT)
- ❑ Software interrupts (INT8-16, INT20-31)

This document shows as simply as possible the general way to use these interrupts. The TMS320C2xx handles interrupts in three phases:

- ❑ Reception of the interrupt request
- ❑ Acknowledgement of the interrupt
- ❑ Execution of the corresponding Interrupt Subroutine (ISR)

Contents

Figures

Tables

Setting Up TMS320C2xx Interrupts in Assembler

Introduction

Typically, interrupts are generated by devices that need to give or take data from the Texas Instruments (TI™) TMS320C2xx, for example, analog-to-digital (A/D) and digital-to-analog (D/A) converters or other processors. When the C2xx recognizes an interrupt signal, it suspends execution of the code specific to the particular interrupt event (Interrupt Sub Routine).

The C2xx family supports three or four different sorts of interrupts:

- ❑ Maskable external interrupts (INT1-3)
- ❑ Non-maskable external interrupts (NMI, RS)
- ❑ Maskable internal interrupts (TINT, RINT, XINT, TXRXINT)
- ❑ Software interrupts (INT8-16, INT20-31)

This document shows as simply as possible the general way to use these interrupts. The TMS320C2xx handles interrupts in three phases:

- ❑ Reception of the interrupt request
- ❑ Acknowledgement of the interrupt
- ❑ Execution of the corresponding Interrupt Sub Routine (ISR)

Interrupt Management

Flags

When a valid signal is generated on an interrupt pin, the corresponding flag bit is set in the **IFR** (Interrupt Flag Register). Because INT2/INT3 share the same bit in the IFR, two additional flags (FINT2 and FINT3) located in the ICR (Interrupt Control Registers) allow us to distinguish between both.

Masks

The user can select which interrupts the TMS320C2xx should respond to at a given time. A “1” written to any mask bit of the **IMR** (Interrupt Masked Register) enables the corresponding interrupt. This IMR includes two “combo-flags”:

- ❑ INT1/HOLD
- ❑ INT2/INT3

Hence, to separate the combined masks, two additional bits (MINT2 and MINT3 in the ICR) allow us to distinguish those interrupts.

The HOLD and INT1 signals share the same external pin. Although INT1/HOLD are combined in the IFR, they are mutually exclusive. The MODE bit (bit 4 of the ICR) separates both types of inquiries:

- MODE = "0" HOLD is selected
- MODE = "1" INT1 is chosen

Even through HOLD is shown in the IMR, the HOLD function cannot be masked.

Controls

The **ICR** (Interrupt Controls Register) provides the capability to:

- Individually mask INT2 and INT3 (MINT2 and MINT3 bits)
- Clearly identify which interrupt INT2 or INT3 has been requested (FINT2 and FINT3 bits)
- Differentiate HOLD and INT (MODE bit)

Global Enabling

The **INTM** (Interrupt Mode bit, bit 9 of the ST0) globally enables or disables all maskable interrupts.

- INTM = '0' enables masked interrupts
- INTM = '1' inhibits masked interrupts

Interrupt Initialization

Registers Definition

Some registers are mapped to data space (from 0000h to 0060h). The .mmregs directive defines global symbols for all memory-mapped registers (listed on page 5-35 of the *TMS320C2xx User's Guide*, literature number SPRU127B).

To facilitate the access to I/O mapped registers, you can define the address of on-chip registers mapped to I/O space. This table depends on the target DSP.

The on-chip registers mapped to I/O space for the TMS320C203 are:

```
CLK        .set   0FFE8h   ;CLK Register
IC         .set   0FFECh   ;Interrupt Control Register
SDTR       .set   0FFF0h   ;Synch. Serial Port Transmit/Receive Register
SSPCR      .set   0FFF1h   ;Synch. Serial Port Control Register
ADTR       .set   0FFF4h   ;Async. Serial Port Receive/Transmit Register
ASPCR      .set   0FFF5h   ;Async. Serial Port Control Register
IOSR       .set   0FFF6h   ;Input/Output Status Register.
BRD        .set   0FFF7h   ;Baud Rate Divisor Register
TCR        .set   0FFF8h   ;Timer Control Register
PRD        .set   0FFF9h   ;Timer Period Register
TIM        .set   0FFFAh   ;Timer Counter Register
WSGR       .set   0FFFCh   ;Wait State Generator Control Register
```

Steps of the Initialization

During the initialization of the processor, the user can define his working environment and enable/disable interrupts according to the application. We can distinguish three steps in the initialization process.

Global Disable Interrupts

To the prevent interruption of processor initialization, all interrupts are disabled by setting the INTM bit using the following command:

```
SETC            INTM                    ; disable interrupts
```

Mask Interrupts

- ❑ IMR setting. As this is a memory-mapped register and assuming that the IMR value is contained in IMR_Value:

```
SPLK       IMR,#IMR_Value        ; mask interrupts
```

- ❑ ICR setting. As this is a I/O-mapped register and assuming that the ICR value is contained in ICR_Value:

```
SPLK       #IC_Value,TEMP      ;load ICR_Value in a temporary variable
OUT        TEMP,IC             ;write ICR_Value in ICR
```

Flag Clear

To avoid servicing an interrupt, the flag bits have to be cleaned up. The IFR is cleared by writing a ‘1’ in each bit whereas the flag in the ICR can be cleared ; meanwhile the mask is set (see previous point).

```
IFR_CLR         .set       0FFFFh
                SPLK       #IFR_CLR,IFR
```

Global Reenable All Unmasked Interrupts.

Before starting the main function, the INTM bit is reset. Thus, all unmasked interrupts are enabled.

```
CLRC       INTM          ;enable all unmasked interrupts
```

Table 1. Register Setting

Interrupt	Condition on IMR	Condition on ICR	Observations
$\overline{INT1}$	bit 0 = ‘1’	bit 4 = ‘1’	mutually exclusive with $\overline{HOLD}$
$\overline{HOLD}$	bit 0 = ‘1’	bit 4 = ‘0’	mutually exclusive with INT1
$\overline{INT2}$	bit 1 = ‘1’	bit 0 = ‘1’	
$\overline{INT3}$	bit 1 = ‘1’	bit 1 = ‘1’	
TINT	bit 2 = ‘1’	not involved	

Vector Table Allocation

Once the interrupt is received, the C2xx branches to its corresponding subroutine called an ISR (Interrupt Service Routine). The C2xx follows the branch instruction you place at the predetermined address (the vector location) and executes the ISR you have written. The vector location table is shown on page 5-16 of the *TMS320C2xx User's Guide*, literature number SPRU127B). The user must map it at address 0000h in the program space via the command file (interr.cmd). Typically, a .sect directive is used.

For each interrupt, two words are reserved: one to code the branch instruction and the other for the address to be branched. If one of those interrupts is unused, replace the branch instruction by the directive, .space 2*16. It reserves 2*16 = 32 bits. The following is an example of an interrupt table :

```
.sect          “vectors”
B              INIT              ;reset
B              IT1HOLD           ;INT1/HOLD
B              INT2_3            ;INT2 and INT3
B              TINT              ;TINT
```

Interrupt Service Routine

Before returning from any interrupt, you generally need to reenable unmasked interrupts. Thus, the ISR ends with:

```
CLRC     INTM       ;reenable unmasked interrupts

RET                 ;return from interrupt
```

HOLD and INT1

Both signals are connected to the same pin. Thus, they share the same mask bit and the same flag. The MODE bit (in the ICR) distinguishes them. To know which subroutine has to be branched, we test the MODE. If MODE = ‘0’, the HOLD state is set up ;otherwise, the interrupt INT1 has to be served. The function of this pin is selected by the user and may depend on the part of the running program. For example,

```
*Registers values
FLAGIT1        .set      0010h              ;identify the HOLD mode
ICRHOLD        .set      0000h              ;Hold mode
ICRINT1        .set      0010h              ;INT1 mode

               SPLK      #ICRINT1,TEMP  ;set-up INT1 (use ICRHOLD to set up HOLD)
               OUT       TEMP,IC            ;INT1 is enabled
```

The code below is a possible test.

```
IT1HOLD:    IN      TEMP,IC         ;capture ICR
            LACL    TEMP
            AND     FLAGIT1         ;test MODE bit
            BCND    HOLD,EQ         ;if HOLD mode, branch to HOLD

INT1:       NOP                     ;interrupt1 service routine
            SETC    XF              ;XF=1, shows that INT1 is operating
            RPT     TEMP2           ;repeat 32768 times
            NOP
            CLRC    XF              ;INT1 ended
            CLRC    INTM            ;enable interrupts before return
            RET                     ;return from interrupt

HOLD:       LACL    IMR             ;save the current IMR
            SPLK    #1,IMR          ;mask all interrupts
                                    ;only a positive edge on the INT1/HOLD pin
                                    ;may issue the HOLD mode

            IDLE                    ;power down mode
                                    ;(HOLD mode-HOLDA is asserted)
            SPLK    #HOLD_CLR,IFR   ;clear HOLD to prevent
                                    ;a 2nd service of the IT
            SACL    IMR             ;restore the mask
            CLRC    INTM            ;enable unmasked interrupts before return
            RET                     ;return from HOLD mode
```

There are three methods for exiting the HOLD mode while de-asserting HOLDA:

- ❑ Rising edge on the INT1/HOLd pin
- ❑ Reset
- ❑ NMI

Even if any other unmasked interrupt can exit an idle state, the HOLD would not be properly left (the HOLDA will not be de-asserted). That is why the current IMR is saved at the beginning of the Hold subroutine and changed to 0001h (only the INT1/HOLD is enabled) before the idle. On a rising edge of the INT1/HOLd pin, the idle state is exited and the old IMR is restored.

INT2 and INT3

Bit 1 of the IFR is the flag for both INT2 and INT3. This bit is cleared automatically by the CPU when either interrupt is serviced. To determine which one was received, the ISR must read FINT2 and FINT3 in the ICR and then branch as required to the proper place in the ISR.

```
FLAGIT2      .set    0004h       ;mask of the ICR differentiate INT2 from INT3
IMR_VAL      .set    0002h       ;IMR mask - enable INT2 and INT3
ICR_VAL      .set    0003h       ;ICR value - unmask INT2 / mask INT3  : 0001h
                                 ;ICR value - mask INT2 / unmask INT3  : 0002h
                                 ;ICR value - unmask both INT2 and INT3: 0003h

INT2_3:      IN      TEMP,IC     ;capture ICR
             LACL    TEMP
             AND     #FLAGIT2    ;test FINT2
             BCND    INT3,EQ     ;branch to INT3 subroutine if FINT2=0
INT2:        NOP                 ;here is the ISR corresponding to INT2
             ...
INT3:        NOP                 ;here is the ISR corresponding to INT3
             ...
```

FINT2 and FINT3 are not cleared automatically by the CPU. Consequently, clearing is done in the interrupt subroutine.

```
             IN      TEMP,IC     ;capture ICR
             OUT     TEMP,IC     ;clear the flag
```

TINT

The timer consists of three I/O-mapped registers. The process used to fix the value of those registers according to the timing rate is described in the *TMS320C2xx User's Guide*, literature number SPRU127B.

- TCR (Timer Control Register)
- PRD (Period Register)
- TIM (Timer Value Register)

The Timer setup is generally done during the initialization using the following steps:

1) Stop the timer (TSS = '0', bit 4 of TCR) and initialize the TDDR value (1st byte of the TCR).

2) Initialize the PRD value.

3) Load TIM with PRD and PSC with TDDR and start the timer.

It leads to the following code:

```
*Following values have been computed to generate a TINT at a rate of
*16 kHz with a DSP running with an internal cycle time of 25 nsec.
*               PRD = 2499d = 9C3h
*               PSC = 0d = 0h
TCR_STOP        .set    0010h   ;stop Timer - load TDDR
TCR_RUN         .set    0020h   ;load PSC with TDDR (TRB = 0)
                                ;load TIM with PRD (TRB = 0)
                                ;start Timer (TSS = 1)
PRD_VAL         .set    09C3h   ;Configuration of the Timer Period Register

*The following lines configures the Timer
                SPLK    #IMR_VAL,IMR            ;Mask of interrupts: enables TINT

                SPLK    TCR_STOP,TEMP           ;stop Timer
                OUT     TEMP,TCR                ;initialization of TDDR

                SPLK    PRD_VAL,TEMP            ;initialization of PRD
                OUT     TEMP,PRD

                SPLK    TCR_RUN,TEMP            ;load TIM with PRD
                OUT     TEMP,TCR                ;load PSC with TDDR
                                                ;start Timer
                CLRC    INTM                    ;enable all interrupts
```

Setting Up TMS320C2xx Interrupts in C

Program Developed in C

C Compiler

The TMS320C2x/C2xx/C5x C Compiler is compatible with ANSI C standards and made up of the preprocessor, parser, optimizer and code generator. The code generator produces assembly code that can be assembled and linked.

Runtime Support

Some tasks that a C program must perform (e.g., memory allocation, string searches ...) are not part of the C language. The ANSI C standard defines a complete set of runtime support functions performing these tasks. The TMS320 fixed-point compiler includes a library that contains ANSI standard runtime support functions gathered in two libraries:

- ❑ rts2xx.lib (TMS320C2xx standard runtime support functions)
- ❑ rts.src (source of library functions).

Both files are described in the *TMS320C2x/C2xx/C5x Optimizing C Compiler User's Guide*, literature number SPRU024D.

Linker

The user must create a linker command file that specifies precise placement of sections. The structure linker command files used for C program remains the same as for assembly program. In fact, when linking C code through, the following two considerations must be observed :

- ❑ Set both stack and heap size using the -stack and -heap options.
- ❑ Allocate the seven sections produced by the C-compiler into memory. These include four initialized sections and three uninitialized sections.

Table 2. Memory Allocation

Directive	**Type**	**Description**	**Link to**
.text	initialized	executable code	program memory
.cinit	initialized	data tables to initialize global and static variables	program memory
.switch	initialized	tables for switch statements	program memory
.const	initialized	data constants declared by const	data memory
.bss	uninitialized	global and static variables	data memory
.stack	uninitialized	c system stack	data memory
.sysmem	uninitialized	heap (dynamic memory)	data memory

Initializing the C Environment

Before running a C program, the C environment has to be created. This can be done either by using the *boot.asm* module (in *rts2xx.lib*) or by writing your own boot routine. In both cases, this boot routine has to ensure four operations :

- ❑ *Initialization of the stack* (creation of the .stack section and setup) This step performs the creation of the .stack section and the initialization of both stack and frame pointers.
- ❑ *Initialization of the status registers*. This allows starting the processor in a known state.
- ❑ *Auto-initialization of global variables*
- ❑ *Calling the main function* (do not forget that a C program has necessarily a main function.)

Interrupt initialization

This section describes how to access the involved registers from C. There are three types of registers:

- ❑ Those mapped to I/O space
- ❑ Those mapped to data space
- ❑ Status registers (ST0 and ST1)

I/O-Mapped Registers

The first thing to do is to declare these registers:

For example:

```
Name            Address
#define IC      0xFFEC          /*Interrupt Control register*/
```

To access these registers we can use the writeport and readport functions. These functions are assembler-coded functions you can find in inout.asm. The address depends on the used DSP.

```
                 Register name     Value written to the register
                       |               |
writport ((int *)     IC,             IC_Value);

                 Register name     Temp variable
                       |               |
readport((int *)      IC,             &I);
```

Data-Mapped Registers

First, these registers have to be defined:

```
unsigned int *IMR = (unsigned int *) 0x0004;      /*c203 IMR definition/
unsigned int *IFR = (unsigned int *) 0x0006;      /*c203 IFR definition/
```

To write into these registers follow this statement:

```
 *IMR = IMR_Value;
```

This is equivalent to:

```
asm ("           SPLK          #IFR_CLR,IFR");
```

To read those registers use:

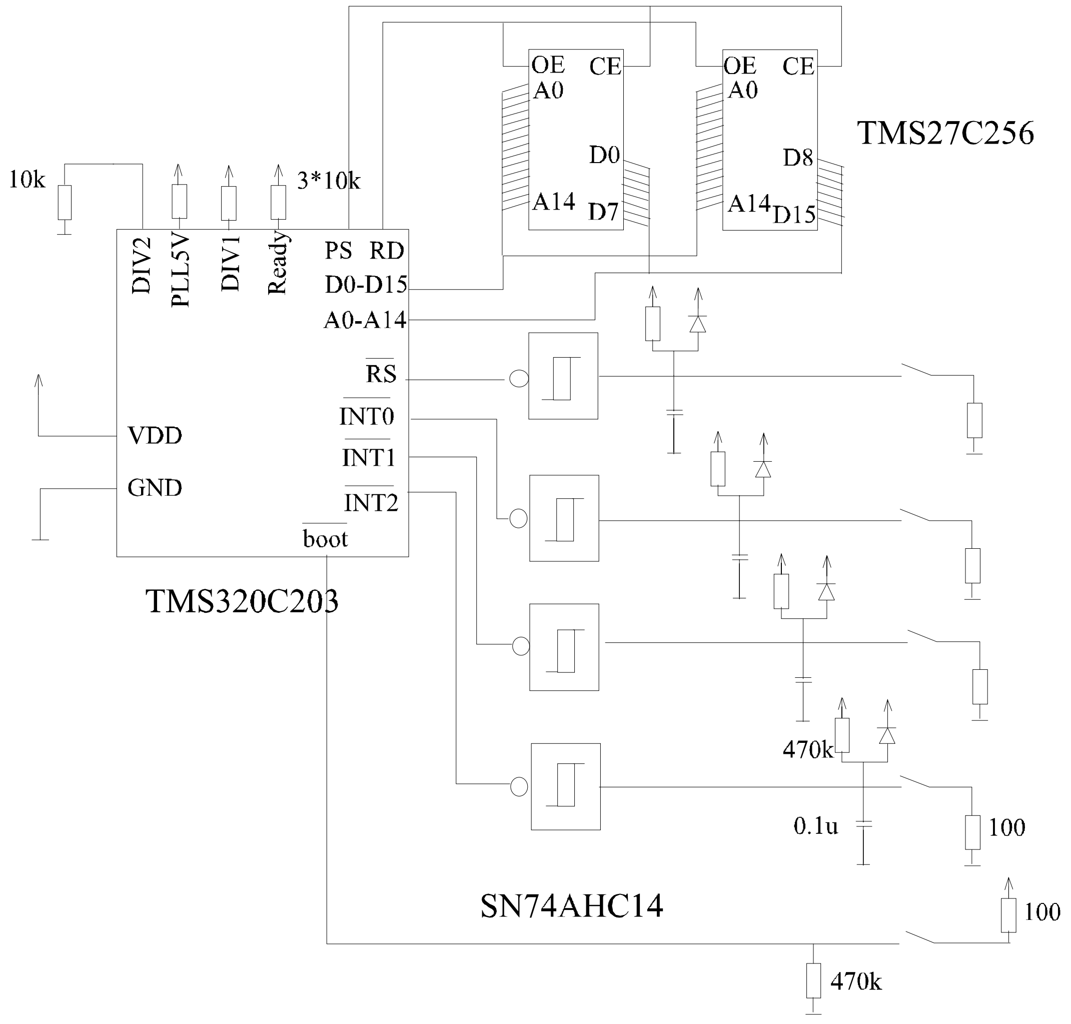
```
IMR_BUF = *IMR;
```

Status Registers

The status registers can only be accessed by using asm-statements, such as the one shown above.

Hardware

Figure 1. Circuit Diagram



Hardware

The circuit diagram shown in Figure 1 is not complete. It suggests one method but there are other options that may offer improved operation.

Comments

- ❑ DIV1 and DIV2 are chosen to get an input clock mode of clock mode*1.
- ❑ For detailed information on the Reset pin, please consult D/B and D/S.
- ❑ We are not booting from the EPROM but just using it as external memory.

INTERNET

www.ti.com

Register with TI&ME to build custom information pages and receive new product updates automatically via email.

TI Semiconductor Home Page
http://www.ti.com/sc

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

PRODUCT INFORMATION CENTERS

US TMS320

Hotline	(281) 274-2320
Fax	(281) 274-2324
BBS	(281) 274-2323
email	dsph@ti.com

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
Francais	+33-(0) 1-30 70 11 64
Italiano	+33-(0) 1-30 70 11 67
Fax	+33-(0) 1-30-70 10 32
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	+886-2-3786800
Domestic	
Australia	1-800-881-011

Asia (continued)

TI Number	-800-800-1450
China	10811
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