TMS320 DSP
DESIGNER'S NOTEBOOK
Number 86-E

Setting up TMS320C2xx interrupts

Contributed by Agnès Delauriere

Introduction

Typically interrupts are generated by devices which need to give or take data from the 'C2xx for example A/D- and 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 3 or 4 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 DNP wants to show as simply as possible the general way how 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).

Contents

1. Setting up TMS320C2xx interrupts in Assembler

1.1 Interrupt management

1.1.1 Flags

When a valid signal is generated on an interrupt pin, the corresponding flag bit is set in the **IFR** (Interrupt Flag Register). As 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.

1.1.2 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 and INT2/INT3.

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

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

1.1.3 Controls

The **ICR** (Interrupt Controls Register) permits :

- individually mask INT2 and INT3 (MINT2 and MINT3 bits)
- clearly identify which interrupts INT2 or INT3 has been requested (FINT2 and FINT3 bits)
- differentiate HOLD and INT (MODE bit)

1.1.4 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

1.2 Interrupt initialization

1.2.1 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 page 5-35 of SPRU127B).

In order 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.

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

```
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
```

1.2.2 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.

1.2.2.1 Global disable interrupts.

In order to the prevent interruption of processor initialization, all interrupts are disabled by setting the INTM bit.

```
SETC   INTM                ; disable interrupts
```

1.2.2.2 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
```

1.2.2.3 Flag clear

In order to avoid to serve 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
```

1.2.2.4 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
```

1.2.2.5 Registers 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	

1.2.3 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 predertermined address (the vector location) and executes the ISR you have written. The vector location table is shown page 5-16 of 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, 2 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.
```

Below is an example of interrupt table :

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

1.2.4 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
```

1.2.4.1 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. In order to know which subroutine has to be branched, we test the MODE. If MODE = '0', the HOLD state is set-up ; else, the interrupt INT1 has to be served. The function of this pin is selected by the user and may depends 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 innterrupts before return
          RET                         ;return from HOLD mode
```

There are three methods for exiting the HOLD mode desassertind HOLDA :

- a rising edge on the INT1/HOLd pin
- a reset
- a NMI

Even if any other unmasked interrupt can exit an idle state, the HOLD would not be properly left (the HOLDA will not be dissasserted). That is why, the current IMR is saved at the beginning of the Hold subroutine, 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.

1.2.4.2 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 figure out 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, it has to be done in the interrupt subroutine.

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

1.2.4.3 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 (SPRU127B).

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

The Timer setup is generaly done during the initialization. There are the three 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
```

2. Setting up TMS320C2xx interrupts in C

2.1 Program developped in C

2.1.1 C compiler

The *TMS320C2x/C2xx/C5x Optimizing 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.

2.1.2 Runtime support :

Some tasks that a C program must perform (eg. 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 which are gathered in two libraries : rts2xx.lib (TMS320C2xx standard runtime support functions) and rts.src (source of library functions). Both files are described in the *TMS320C2x/C2xx/C5x Optimizing C Compiler User's Guide* (SPRU024D*).*

2.1.3 Linker

The user must create a linker command file which 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.

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

2.1.4 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 case, this boot routine has to ensure four operations :

1. *initialization of the stack* (creation of the .stack section and set-up) This step performs the creation of the .stack section and the initialization of both stack and frame pointers.
2. *initialization of the status registers.* This allows to start the processor in a known state.
3. *auto-initialization of global variables*
4. *calling the main function* (do not forget that a C program has necessarily a main function).

2.2 Interrupt initialization

As we have discussed above how to initialize registers, we only present how to access to the involved registers from C. There are 3 types of registers :

1. those which are mapped to I/O space
2. those which are mapped to data space
3. the status registers (ST0 and ST1).

2.2.1 I/O mapped registers

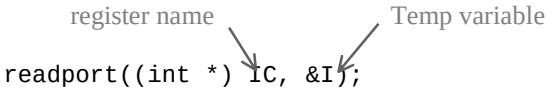
The first thing to do is to declare these registers:

For example:

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

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


```
writport ((int *) IC, IC_Value);
```




```
readport((int *) IC, &I);
```

2.2.2 Data mapped registers

First of all they 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 the below statement.

```
 *IMR = IMR_Value;
```

This is equivalent to :

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

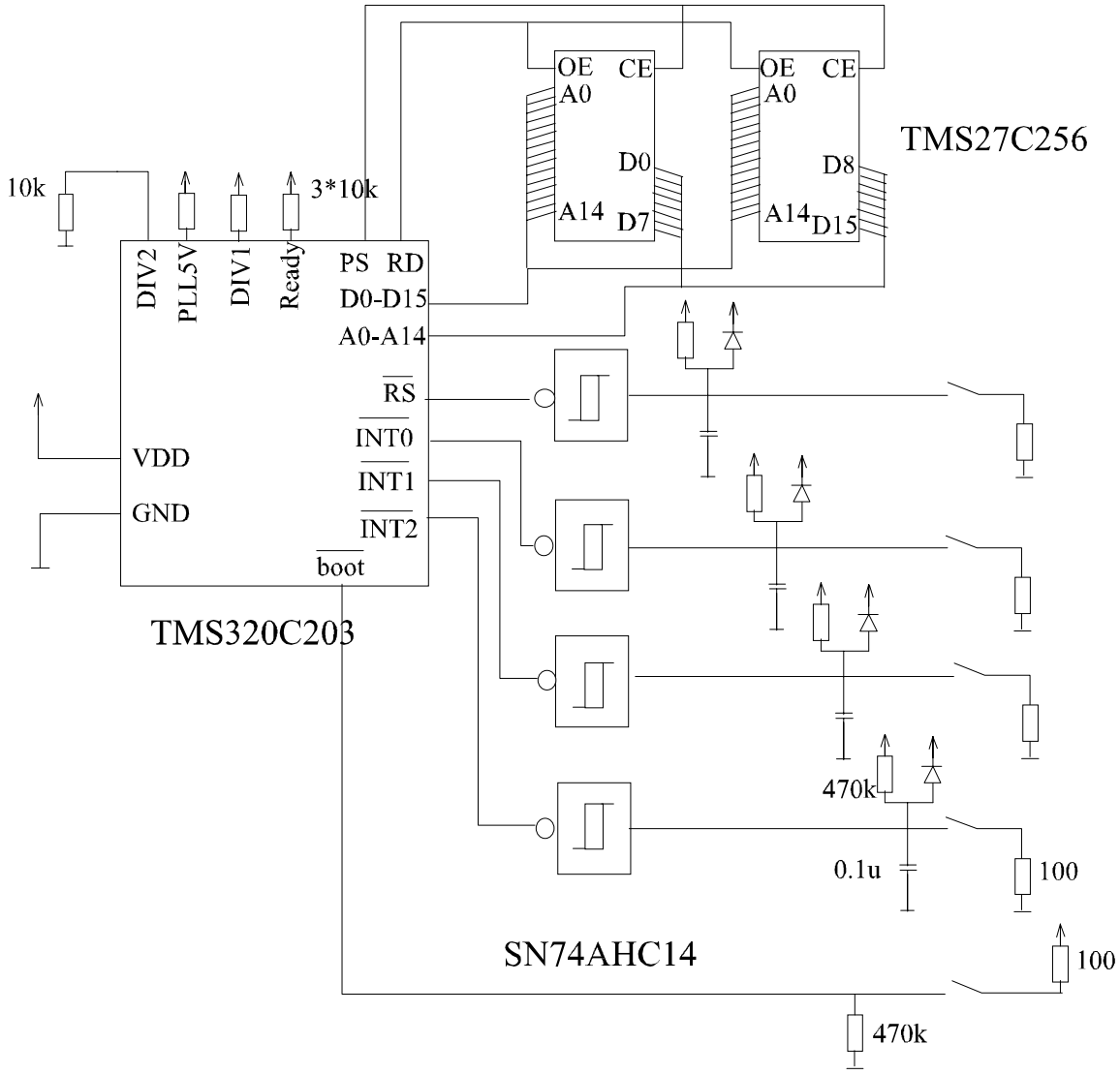
To read those registers use:

```
IMR_BUF = *IMR;
```

2.2.3 Status registers

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

3. Hardware



Hardware:

This circuit diagram is not complete and there are certainly better ways of doing it. It is shown to suggest one way of doing it.

Comments:

- DIV1 and DIV2 are chosen to get an input clock mode of clock mode*1
- To get 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.