# Mod5270 Interrupts

---

# Application Note

Revision 1.1
Feb 25, 2005
Document Status: Released

# Table of Contents

# 1. Introduction

The 5270 has a very flexible interrupt system. It is somewhat different than the traditional M68K or 5206 ColdFire interrupt system. On first inspection it is somewhat confusing, as the interrupt documentation is split into multiple chapters of the 5270 User's Manual, and it is not easy to identify what is going on. This application note will attempt to clear up some of this confusion.

# 2. General Procedure

To configure an interrupt on the 5270, you must really configure it twice. The first part of the configuration configures the hardware module that causes the interrupt. Timers, serial I/O, external pins, Ethernet, etc... The documentation for this part of the process is defined in the ColdFire 5270 User's Manual and is specific to the processor peripheral you will use in your application. The second part of the configuration involves setting up the interrupt controller for that specific interrupt source.

In general, the following procedure is used to enable interrupt handling in an application:

- Review your system architecture and determine which interrupt level (1-7) is appropriate. Level 1 is the lowest priority, level 7 is the highest. Use caution with level 7, since it is unique in that it is a non-maskable interrupt. If you use level 7, the ISR cannot call any uCOS functions, or use the INTERRUPT( ) macro.
- Write an Interrupt Service Routine (ISR). If you are using IRQ1 – 6, use the INTERRUPT( ) macro to make coding the ISR easier. If you are using the level 7 unmaskable IRQ, then you must write the program code to save and restore the CPU registers.
- Call the SetIntc( ) function to set up the interrupt vector, level and priority. The function should be called prior to any interrupts occurring.

## *2.1 Hardware Configuration*

The IRQ pins for external interrupts on the 5270 run through a hardware module called the Edge Port Module. The Edge Port Module allows you to configure the IRQ1...7 pins as inputs, outputs and interrupt generators. They can be configured to be edge or level sensitive. See chapter 15 of the 5270 User's Manual for a discussion of these functions. It is very important to perform the specific action required by the hardware module to clear the interrupt.

The IRQ button on the NetBurner Module Development Board is attached to the IRQ1 pin. The example interrupt program at the end of this document configures IRQ1 as an interrupt source, and enables you to generate an interrupt by pressing the button.

## 2.2 Interrupt Controllers

All interrupts in the 5270 go through the interrupt controller described in chapter 13. The interrupt controller supports 42 different interrupt sources. The Interrupt controller is organized around vectors and the first step in using the interrupt controller is to identify the vector associated with the peripheral generating the interrupt request. This is shown in chapter 13, table 13-13, in section 13.2.1.6.1 of the 5270 users manual. Once you have identified the vector, you need to set up the vector destination and priority. In NetBurner 5270 based devices we have provided a helper function to do this:

```
extern "C"
{
/* This function sets up the 5270 interrupt controller */
void SetIntc(long func, int vector, int level, int prio );
}
```

| Parameter | Type | Usage |
|---|---|---|
| func | long | The address of the ISR function |
| long | vector | The vector number to use. This depends on which processor peripheral you use, per table 13-13 of the 5270 User's Manual. |
| int | level | The interrupt level to assign this function 1..7, 7 being non-maskable |
| int | prio | The priority to assign this function. Priority is used to differentiate between multiple interrupts requesting at the same level. |

With this type of interrupt controller structure, your application can have multiple interrupt sources at the same level. For example, on the Mod5270, IRQ 3 is used for the UARTs and IRQ5 is used for the uCOS timer. You may also use external interrupts IRQ3 and IRQ5, and they will not conflict with the timer and serial ports, because each has their own ISR vectors. If you have two or more interrupt sources at the same level (eg. IRQ5), the IRQ that occurs first will be processed first. If two IRQ sources at the same level occur simultaneously, priority is determined as follows:

- The source with the highest **priority level** as specified in the interrupt control register. Note that this is a different setting than the **interrupt level**, which is 1-7.
- If priority levels are identical, then the source with the lowest vector number get processed first.

### *2.3 The INTERRUPT( ) MACRO*

When you create an Interrupt Service Routine (ISR), the ISR must save the state of processor registers upon entry to the ISR, and restore them before exiting. In addition, interrupts 1-6 interface with the uC/OS operating system. The INTERRUPT ( ) macro is provided to handle these all these issues.

**Note that a level 7 unmaskable interrupt is completely independent of the RTOS. It cannot call any uC/OS functions, and no uC/OS operations will be executed, such as the task scheduler. Therefore, the INTERRUPT( ) macro cannot be used.** See appendix A for an example of how to write a level 7 ISR without using the INTERRUPT( ) macro.

## 3. Example Program

The following example program will configure IRQ1 to operate with the IRQ pushbutton on the Mod5270 development board. Each time the button is pressed an interrupt will be counted.

```
#include "predef.h"
#include <stdio.h>
#include <ctype.h>
#include <startnet.h>
#include <autoupdate.h>
#include <dhcpclient.h>
#include <../mod5270/system/sim5270.h>
#include <cfinter.h>


/* An interrupt setup helper defined in bsp.c */
extern "C"
{
   /* This function sets up the 5270 interrupt controller */
   void SetIntc(long func, int vector, int level,
                int prio );
}

OS_SEM IrqPostSem; /* We are going to use a Semaphore to
communicate between the IRQ1 pin ISR and the main application */

/* Declare our interrupt procedure....
name: our_irq1_pin_isr
masking level (The value of the ColdFire SR during the interrupt:

use 0x2700 to mask all interrupts.
      0x2500 to mask levels 1-5 etc...
      0x2100 to mask level 1 */

INTERRUPT(out_irq1_pin_isr, 0x2100 )
{
  /* WARNING WARNING WARNING
  Only a very limited set of RTOS functions can be called from
  within an interrupt service routine.
  Basically, only OS POST functions and LED functions should be used
  No I/O (read, write or printf may be called), since they can block. */

  sim.eport.epfr=0x02; /* Clear the interrupt edge 0 0 0 0 0 0 1 0 */
  OSSemPost(&IrqPostSem);
}


extern "C" {
      void UserMain(void * pd);
      void putdisp(unsigned short w);
}

/* Helper function to display a decimal number */
void PutDispDecimal(WORD val, BOOL blank_zero )
{
      WORD w;

      W = (val / 1000) * 0x1000;
      if ((w == 0) && (blank_zero)) w = 0xF000;
      w += ((val / 100)% 10) * 0x0100;
      if (w == 0xF000) w = 0xFF00;
      w += ((val / 10)% 10) * 0x0010;
      if (w == 0xFF00) w = 0xFFF0;
      w += ((val )% 10) * 0x0001;
      putdisp(w);
}


void UserMain(void * pd)
{
  InitializeStack();
  if (EthernetIP==0)GetDHCPAddress();
  OSChangePrio(MAIN_PRIO);
  EnableAutoUpdate();

  DWORD isr_count=0; /* Count how many times the switch was hit */

  /* Initialize the semaphore we are using */
  OSSemInit(&IrqPostSem,0);

  /* First set up the Eport module to use IRQ1 as a falling-edge active  IRQ pin
  (See the 5270 UM chapter 15). Set the pin assignment register   irq1 pin falling
  edge sensitive*/
  sim.eport.eppar=0x0008; /* 00 00 00 00 00 00 10 00  see table
                            15-3 in UM */

  sim.eport.epddr=0x0;    /* All edge port pins as inputs */
  sim.eport.epier = 0x0002; /* Enable IRQ1 only 0 0 0 0 0 0 1 0 */

  /* Now enable the actual interrupt controller. See users manual chapter 10 for
  more information. We are going to use the BSP helper function declared above and
  implemented in BSP.c */
  SetIntc(
    (long)&out_irq1_pin_isr, /* Our interrupt function */
     1, /* The vector number from the users manual table 10-13 */
     1, /* Set this to priority 1 but any value from 1 to 6 would
           be valid.*/
     1 /* The priority within the gross levels; see chapter 10, any
```

```
           value from 0 to 7 is ok */
     );

  iprintf("Application started\n");
  iprintf("Press the IRQ button on the development board.\n");
  while (1)
  {
    OSSemPend(&IrqPostSem,0 /* Wait forever */);
    PutDispDecimal(++isr_count,true);
    iprintf("The interrupt Switch was hit %ld times\r\n", isr_count);
  }
}
```

# Appendix A – Writing a Level 7 ISR

The following example of a level 7 non-maskable ISR consists of two parts: a section written in assembly language to save and restore the CPU registers, and a section written in C/C++ that does the actual work.

**Note that a level 7 unmaskable interrupt cannot call any uCOS functions, and therefore cannot use the INTERRUPT( ) macro.**

```
/* The extern directive is only needed if you are creating this code in a .cpp
   file; it prevents name mangling. If you are crating a .c file, do not include
   the extern section.
*/
extern "C" {
                void NMI_C_Part();   // The part of the ISR written in C/C++
                void NMI_ASM_Part(); // The part of the ISR written in Assembly
           }

/* The “Function_Holder()” is just a place holder so we can create some inline
assembly language code. It will never be called. Note that in the __asm__
statements, you need a leading space between the (“ and the instruction. However,
a space is not used for labels. */

void Function_Holder()
{
   __asm__  (" .global  NMI_ASM_Part"); // export the label for ASM part
   __asm__  (" .extern  NMI_C_Part");   // label for the C part of the ISR
   __asm__  (“NMI_ASM_Part:”);          // label for the assembly part of the ISR
   __asm__  (" move.w #0x2700,%sr ");   // set the IRQ mask to mask all
   __asm__  (" lea -60(%a7),%a7 ");     // make space on the system stack
   __asm__  (" movem.l  %d0-%d7/%a0-%a6,(%a7) "); // save all registers
   __asm__  (" jsr NMI_C_Part  ");
   __asm__  (" movem.l  (%a7),%d0-%d7/%a0-%a6 ");
   __asm__  (" lea 60(%a7),%a7 ");
   __asm__  (" rte");
}

/* This is the C/C++ part of the ISR that is called from the assembly code */
void NMI_C_Part()
{
   // Your C/C++ application code goes here
}
```

In the SetIntc( ) call, use the name “NMI_ASM_Part” to set the function address for the “vector” parameter..