Mod5234 Interrupt Controllers

Application Note

Revision 1.0
June 22, 2007
Document Status: Second Release

Table of Contents

Introduction

The interrupt controllers of the MCF5234 supports up 110 interrupt sources. The 103 fully-programmable and 7 fixed-level interrupt sources for the two interrupt controllers handle the complete set of interrupt sources from all the modules on the device. Each of the sources has a unique interrupt control register to define the software-assigned levels and priorities within the level. This application note will help explain how to use the interrupt controller as well as provide an example program. Please refer to Chapter 13: Interrupt Controller Modules of the MCF5235 Reference Manual for a thorough and detailed description of the MCF5234 interrupt controllers.

General Procedure

There are two parts to setting up an interrupt on the MOD5234. The first part of the the setup involves configuring the hardware module that generates the interrupt (PIT, DMA Timers, Edge Ports, et cetera). The documentation for this part of the process is defined in the MCF5235 Reference Manual chapters that are specific to the processor peripherals being used in the application. The second part of the setup involves configuring the interrupt controller for that specific interrupt source.

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

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

Peripheral Module Configuration

Each peripheral module that has an interrupt source vector listed in Table 13-13 and 13-14 of the MCF5235 Reference Manual has its own set of registers for interrupt configuration. For example, if you plan on using the PIT (Programmable Interrupt Timer) Module to generate interrupts, then you will need to refer to Chapter 23 of the manual to set that up, as well as what registers to configure. The Edge Port peripheral module is used to configure IRQ1 for interrupt generation as an example in this application note.

The IRQ pins for external interrupts [IRQ1 (J2-43), IRQ3 (J2-45), IRQ5 (J2-47), and IRQ7 (J2-48)] on the MOD5234 run through a hardware module called the Edge Port Module. The module allows configuration of the IRQ*n* pins as inputs, outputs, and interrupt generators. They can be configured to be edge or level sensitive. See Chapter 15 of the reference 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 program example found at the end of this application note demonstrates the generation of interrupts by IRQ1 as a result of signal inputs triggering on falling-edge sensitivity.

Interrupt Controller

All interrupts on the MOD5234 go through one of the two interrupt controllers. It 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 Section 13.2.1.6.1, Table 13-13 and 13-14 of the MCF5235 Reference Manual. Once the vector is identified, the vector destination, level, and priority needs to be set up. A helper function that does this is provided for the MOD5234:

```
extern "C"
{
   // This function sets up the MCF5234 interrupt controller
   void SetIntc(int intc, long func, int vector, int level, int prio);
}
```

Parameter	Type	Usage
intc	**int**	The interrupt controller number assigned to for the interrupt source (0 or 1)
func	**long**	The address of the interrupt service routine function.
vector	**int**	The vector number to use. This depends on the peripheral module used. Please refer to Table 13-13/14 of the MCF5235 Reference Manual.
level	**int**	The interrupt level to assign this function. Levels available are 1-7. Level 7 is non-maskable.
prio	**int**	The priority to assign this function. Priority is used to differentiate between multiple interrupts requesting the same level. Values available are 0-7.

With this type of interrupt controller structure, an application can have multiple interrupt sources at the same level. For example, if the uCOS timer uses IRQ5 and a custom application uses external interrupt IRQ5 from the Edge Port Module, then they will not conflict with each other since both have their own ISR vectors. If two or more interrupt sources are at the same level, then the IRQ that occurs first will be processed first. If two IRQ sources at the same level occur simultaneously, priority will be determined as follows:

- The source with the highest priority is specified in the interrupt control register. Note that this is a different setting from the interrupt level.
- If priorities are identical, then the source with the lowest vector number gets processed first.

Please note that while interrupt sources 8 and above are fully programmable, interrupt sources 1 through 7 are fixed. The seven interrupt sources correspond to external interrupts IRQ1-IRQ7 on the Edge Port Module. Each external interrupt stays fixed at their designated level (level 1 for IRQ1, level 2 for IRQ2, etcetera), and all seven stays fixed at a midpoint priority between 3 and 4, where priority is any number between 0 (lowest) and 7 (highest). Any value assigned to the level and priority parameters of the `SetIntc()` function when using IRQ1-IRQ7 will be ignored. On the MOD5234, only IRQ1, IRQ3, IRQ5, and IRQ7 of the Edge Port Module are made available for use.

The `INTERRUPT()` *Macro*

When you create an interrupt service routine, it must save the state of the processor registers upon entry into the ISR, and restore them before exiting. In addition, interrupt levels 1-6 interface with the uCOS. The `INTERRUPT()` macro is provided to handle all these issues. Note that a level 7 unmaskable interrupt cannot call any uCOS functions, and therefore cannot use the macro.

Level 7 ISR Example

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

```
//
// The extern directive is only needed if you are creating this code in
// a *.cpp file; it prevents name mangling. If you are creating a *.c
// file, then 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
}

////////////////////////////////////////////////////////////////////////
// Function_Holder - This 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 (" and the
// instruction. However, a space is not used for labels.
//
void Function_Holder()
{
   // Export the label for ASM part
   __asm__ (" .global NMI_ASM_Part ");
   // Label for the C part of the ISR
   __asm__ (" .extern NMI_C_Part ");
   // Label for the assembly part of the ISR
   __asm__ ("NMI_ASM_Part:");
   // Set the IRQ mask to mask all
   __asm__ (" move.w  #0x2700,%sr ");
   // Make space on the system stack
   __asm__ (" lea     -60(%a7),%a7 ");
   // Save all registers
   __asm__ (" movem.l %d0-%d7/%a0-%a6,(%a7) ");
   __asm__ (" jsr     NMI_C_Part ");
   __asm__ (" movem.l (%a7),%d0-%d7/%a0-%a6 ");
   __asm__ (" lea     60(%a7),%a7 ");
   __asm__ (" rte ");
}

////////////////////////////////////////////////////////////////////////
// NMI_C_Part - 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()` function call, use the name “NMI_ASM_Part” to set the function address for the “vector” parameter.

Program Example

```
/////////////////////////////////////////////////////////////////////
// This program demonstrates the use of interrupts via IRQ1 of the   //
// edge port module.  For this program to properly work, the MOD5234 //
// should be mounted on a MOD-DEV-100 carrier board; this program    //
// utilizes the IRQ button (found next to the reset button).         //
//                                                                   //
// As the user presses the IRQ button, the transition to active low  //
// triggers an interrupt, invoking the interrupt service routine as  //
// a result, since the program configures IRQ1 for falling-edge      //
// sensitivity.  At the beginning of each loop, the program waits    //
// for the semaphore to be posted/available before continuing on.    //
// When the ISR is called, the interrupt is cleared followed by the  //
// posting of the semaphore, in turn allowing the rest of the        //
// program code in the WHILE loop to execute before waiting for the  //
// semaphore again in the next loop.                                 //
//                                                                   //
// Activity of the interrupts triggering and the ISR being called    //
// are shown to the user in MTTTY via the debug monitor serial port. //
/////////////////////////////////////////////////////////////////////

#include "predef.h"
#include <stdio.h>
#include <ctype.h>
#include <startnet.h>
#include <autoupdate.h>
#include <dhcpclient.h>
#include <sim5234.h>
#include <cfinter.h>

//
// An interrupt setup helper defined in bsp.c
//
extern "C"
{
   // This function sets up the MCF5234 interrupt controller
   void SetIntc( int intc, long func, int vector, int level,
                 int prio );

   void UserMain( void* pd );
   void putdisp( unsigned short w );
}


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

/////////////////////////////////////////////////////////////////////
// INTERRUPT() - Declare our interrupt procedure
//
// Name:
//    out_irq1_pin_isr
// Masking level (the value of the ColdFire SR during the interrupt):
//    Use 0x2700 to mask all interrupts
```

```
//        0x2600 to mask levels 1-6
//        0x2500 to mask levels 1-5
//        ...
//        0x2100 to mask level 1
//
INTERRUPT( out_irq1_pin_isr, 0x2100 )
{
   //
   // 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 (e.g.,
   // read(), write(), or printf()), since they can block.
   //
   sim.eport.epfr = 0x02;      // 0000 0010 - Clear the interrupt edge
   OSSemPost( &IrqPostSem );
}

////////////////////////////////////////////////////////////////////////
// PutDispDecimal() - 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 );
}

////////////////////////////////////////////////////////////////////////
// UserMain() - Main task
//
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

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

   //
   // First set up the edge port module to use IRQ1 as a falling-edge
   // sensitive IRQ pin (see the MCF5235 Reference Manual Section
   // 15.4.1.1).
   //
   sim.eport.eppar = 0x0008;      // 0000 0000 0000 1000 - See Table
                                  // 15-3 in the MCF5235 Ref Man
```

```
    sim.eport.epddr = 0x00;         // 0000 0000 - All edge port pins as
                                    // inputs

    sim.eport.epier = 0x02;         // 0000 0010 - Enable IRQ1 only

    //
    // Now enable the actual interrupt controller.  We are going to use
    // the BSP helper function declared above and defined in
    // \Nburn\MOD5234\system\bsp.c
    //
    SetIntc( 0,                             // The intc number
             ( long ) &out_irq1_pin_isr, // Our interrupt function
             1,                             // Interrupt source vector
             1,                             // Interrupt level
             1 );                           // Interrupt priority

    iprintf( "Application started\r\n" );
    iprintf( "Press the IRQ button on the development board\r\n" );

    while ( 1 )
    {
       OSSemPend( &IrqPostSem, 0 /* Wait forever */ );
       PutDispDecimal( ++isr_count, true );
       iprintf( "The interrupt switch was hit %ld times\r\n",
                isr_count );
    }
}
```