Mod5213 Interrupt Controller

Application Note

Revision 1.0
February 22, 2006
Document Status: Initial Release

Table of Contents

Introduction 3

General Procedure 3

- Peripheral Module Configuration 4
- Interrupt Controller 4
- The `INTERRUPT()` Macro 5
- Level 7 ISR Example 6

Program Example 7

Introduction

The interrupt controller of the 5213 supports up 57 interrupt sources. The 50 fully-programmable and 7 fixed-level interrupt sources for the interrupt controller 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 12: Interrupt Controller Module of the MCF5213 Reference Manual for a thorough and detailed description of the 5213 interrupt controller.

General Procedure

There are two parts to setting up an interrupt on the 5213. The first part of the the setup involves configuring the hardware module that generates the interrupt (PIT, DMA Timers, PWM, GPT, etcetera). The documentation for this part of the process is defined in the MCF5213 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 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 12-13 of the MCF5213 Reference Manual has its own set of registers for interrupt configuration. For example, if you plan on using the PWM (Pulse Width Modulation) Module to generate interrupts, then you will need to refer to Chapter 24 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 (Pin 6), IRQ4 (Pin 7), and IRQ7 (Pin 8)] on the 5213 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 13 of the MCF5213 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 edge sensitivity.

Interrupt Controller

All interrupts on the 5213 go through the interrupt controller. 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 12.3.6.1, Table 12-13 of the MCF5213 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 MOD5213:

```
extern "C"
{
   // This function sets up the 5213 interrupt controller
   void SetIntc( long func, int vector, int level, int prio );
}
```

Parameter	Type	Usage
`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 12-13 of the MCF5213 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 through 63 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 MOD5213, only IRQ1, IRQ4, 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, pins 6 and 7 //
// of the MOD5213 must be jumpered, and that the MOD-DEV-40 carrier  //
// board be used in order to utilize the LEDs.                       //
//                                                                   //
// With pin 7 (IRQ4) configured as GPIO, it outputs an alternating   //
// low and high signal every second. Pin 6 (IRQ1) is configured as   //
// IRQ1, its primary function. Set to be an input pin, it receives   //
// the alternating signals from pin 7. Since IRQ1 is configured to   //
// trigger an interrupt on rising and falling edge sensitivity, it   //
// generates an interrupt every second. If it was configured to      //
// trigger an interrupt only on a rising or falling edge, then it    //
// would interrupt every two seconds.                                //
//                                                                   //
// As a simplified explanation, when an interrupt occurs, the        //
// interrupt flag becomes set, and then the ISR function irq_led     //
// runs. Within the ISR function, the interrupt flag is reset and    //
// the LED count is incremented. The incrementing LEDs help signify  //
// when an interrupt is occurring.                                   //
//////////////////////////////////////////////////////////////////////

#include "predef.h"
#include <stdio.h>
#include <ctype.h>
#include <basictypes.h>
#include <serialirq.h>
#include <system.h>
#include <constants.h>
#include <ucos.h>
#include <serialupdate.h>
#include <..\MOD5213\system\sim5213.h>
#include <cfinter.h>
#include <utils.h>      // For use of the putleds() function
#include <pins.h>       // For use of the Pins class

//
// Instruct the C++ compiler not to mangle the function name
//
extern "C"
{
   void UserMain( void *pd );

   // This function sets up the 5213 interrupt controller
   void SetIntc( long func, int vector, int level, int prio );
}

//
// Name for development tools to identify this application
//
const char *AppName = "Mod5213IRQDemo";

int count = 0;          // Initialize the interrupt counter for LEDs
```

```
////////////////////////////////////////////////////////////////////////
// INTERRUPT - Declare the interrupt procedure. Note that the mask
// value of 0x2100 will disable IRQs of level 1 and higher. Use a
// different mask if a different interrupt is used. Other mask values
// are:
//
//    IRQ1 = 0x2100   IRQ4 = 0x2400   IRQ7 = 0x2700
//    IRQ2 = 0x2200   IRQ5 = 0x2500
//    IRQ3 = 0x2300   IRQ6 = 0x2600
//
// 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 may be called
// (i.e., read, write, or printf), since they can block.
//
INTERRUPT( irq_led, 0x2100 )
{
   //
   // When an interrupt occurs, this ISR clears the IRQ1 flag and
   // increments the binary value representation of the LEDs on the
   // MOD-DEV-40 carrier board.
   //
   sim.eport.epfr |= 0x02;       // Clear IRQ1 flag
   putleds( count++ );           // Increment value of LEDs
}

////////////////////////////////////////////////////////////////////////
// UserMain
//
void UserMain( void *pd )
{
   //
   // Default initialization functions
   //
   OSChangePrio( MAIN_PRIO );
   EnableSerialUpdate();
   SimpleUart( 0, SystemBaud );
   assign_stdio( 0 );

   Pins[6].function( PIN6_IRQ1 );      // Configure pin 6 as IRQ1
   Pins[7].function( PIN7_GPIO );      // Configure pin 7 for GPIO

   sim.eport.eppar |= 0x000C;          // Configure IRQ1 to trigger an
                                       // interrupt on rising and
                                       // falling edge sensitivity

   sim.eport.epddr &= ~0x02;           // Configure IRQ1 as input
   sim.eport.epier = 0x02;             // Enable IRQ1

   //
   // Now enable the actual interrupt controller. See Chapter 12 of the
   // MCF5213 Reference Manual for more information. We are going to
   // use the BSP helper function declared above and implemented in
   // bsp.c
   //
   // 1st input parameter: Interrupt service routine
   // 2nd input parameter: Vector number
```

```
    // 3rd input parameter: Interrupt level
    // 4th input parameter: Interrupt priority
    //
    SetIntc( ( long ) &irq_led, 1, 1, 1 );

    while ( 1 )
    {
       OSTimeDly( TICKS_PER_SECOND );    // 1-second delay
       Pins[7] = 0;                      // Set pin 7 as output low
       OSTimeDly( TICKS_PER_SECOND );    // 1-second delay
       Pins[7] = 1;                      // Set pin 7 as output high
    }
}
```