Mod5213 Programmable Interrupt Timer

Application Note

Revision 1.0
January 23, 2006
Document Status: Initial Release

Table of Contents

Introduction

The Mod5213 has two programmable interrupt timer modules, PIT0 and PIT1. Each PIT is a 16-bit timer that provides precise interrupts at regular intervals with minimal processor intervention. The timer can either count down from the value written in the modulus register, or it can be a free-running down-counter.

This application note will describe the use of the timers, as well as provide an example program that uses one of the timers to create interrupts at a regular interval. The example is useful both for learning how to use interrupts as well as the PIT functions. Additional information concerning the PITs are available in the MCF5213 Reference Manual, Chapter 17 (Revision 1.1).

PIT Registers

The PIT modules involve the use of three types of 16-bit registers: the PIT Control and Status Register (PCSR), PIT Modulus Register (PMR), and the PIT Count Register (PCNTR). Each PIT module has its own set of registers. The PCSR and PMR registers have read and write access while the PCNTR register only has read access. The following subsections describe these registers.

PIT Control and Status Register (PCSR)

The PCSR registers configure the corresponding timer's operation. You can enable/disable interrupts and the PITs, set the starting value for down-counting, determine when to reload a new starting value, and more with this register. Additional information about the use of this register can be found in section 17.2.1 of the reference manual.

An important note that requires attention is the configuration of the prescalar on bits 11-8 of the PSCR register. Configuring and generating the PIT clock requires knowing the system clock, PIT modulus value, and prescalar value for the desired timeout period. The following equation is used to calculate the timeout period for the PIT clock:

```
T = Timeout Period
P = Prescalar
M = PIT Modulus Value
S = System Clock

T = ( P x ( M + 1 ) x 2 ) / S
```

For example, lets say that a timeout period of about 0.001 second is needed. The system clock of the Mod5213 is 66.3552 Mhz, so we have a value of 66,355,200 Hz for S. Using a system clock divisor value of 2 is sufficient for 0.001 second. Knowing what prescalar value is needed depends on the desired timeout period. The PIT Modulus Register is

only 16 bits, so the highest starting value it will count down from is 0xFFFF, or 65,535. If a larger time interval is used, such as 1 second, then a system clock divisor of 2 is not enough. The divisor would need to be larger, such as 32,768 (a bit configuration value of "1111" for the PCSR register from table 17-3) in order to keep the value of the modulus register under the maximum value. All that's left now is determining the modulus value, which comes out to be about 16588. With the prescalar bit configuration set to "0001" (from the prescalar table for system clock divisor of 2 of table 17-3), PIT Modulus Register written with the value 16588, and PCSR[1:0] written with "11" (reloads the down-counter from PMR once it reaches zero and enables the PIT), you get a PIT clock cycle of 0.001 second.

PIT Modulus Register (PMR)

The 16-bit read/write PMR contains the timer modulus value that is loaded into the PIT counter when the count reaches 0x0000 and the PCSR[RLD] (reload) bit is set. Additional information on how to calculate the modulus value can be found in the previous section on the PIT Control and Status Register.

When the PSCR[OVW] (overwrite) bit is set, PMR is transparent, and the value written to PMR is immediately loaded into the PIT counter. The prescalar counter is reset (0xFFFF) anytime a new value is loaded into the PIT counter and also during reset. Reading the PMR returns the value written in the modulus latch. Reset initializes the PMR to 0xFFFF.

PIT Count Register (PCNTR)

The 16-bit read-only PCNTR contains the counter value. Reading the 16-bit counter with two 8-bit reads is not guaranteed to be coherent. Writing to PCNTR has no effect, and write cycles are terminated normally.

Program Example

The general procedure for setting up a PIT typically involves three steps: 1) Define an interrupt service routine, 2) Configure the PIT, and 3) Start the PIT. The following example program will configure IRQ2 to interrupt one PIT request event approximately every 1/1000th second via the PIT1 module.

```
/*********************************************************************/
/* This example program exercises the programmable interval timer in */
/* the MCF5213 CPU.                                                  */
/*********************************************************************/

#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 <cfinter.h>
#include <utils.h>
#include <..\MOD5213\system\sim5213.h>

//
// Function prototypes - instruct the C++ compiler not to mangle the
// function name(s)
//
extern "C"
{
   void UserMain( void *pd );

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

//
// Global variables
//
volatile DWORD pitr_count;

//
// Name for development tools to identify this application
//
const char *AppName = "Mod5213PITDemo";

////////////////////////////////////////////////////////////////////////
// INTERRUPT - PIT interrupt service routine.
//
INTERRUPT( my_pitr_func, 0x2600 )
{
   WORD tmp = sim.pit[1].pcsr;   // Use PIT 1
```

```
    //
    // Clear PIT 1 - refer to table 17-3 for more information on what
    // bits are being cleared and set
    //
    tmp &= 0xFF0F;           // Bits 4-7 cleared
    tmp |= 0x0F;             // Bits 0-3 set
    sim.pit[1].pcsr = tmp;

    //
    // You can add your ISR code here
    // - Do not call any RTOS function with pend or init in the function
    //   name
    // - Do not call any functions that perform a system I/O read,
    //   write, printf, iprint, etc.
    //
    pitr_count++;
}

////////////////////////////////////////////////////////////////////////
// SetUpPITR - PIT setup function. See chapter 17 of the 5213 reference
// manual for details.
//
void SetUpPITR( int pitr_ch, WORD clock_interval, BYTE pcsr_pre /* See
    table 17-3 in the users manual for bits 8-11 */ )
{
    WORD tmp;

    if ( pitr_ch != 1 )
    {
        return;
    }

    //
    // Populate the interrupt vector in the interrupt controller
    //
    SetIntc( ( long ) &my_pitr_func, 55 + pitr_ch, 2 /* IRQ 2 */, 3 );

    sim.pit[pitr_ch].pmr = clock_interval;   // Set the PIT modulus
                                             // value
    tmp = pcsr_pre;
    tmp = ( tmp << 8 ) | 0x0F;
    sim.pit[pitr_ch].pcsr = tmp;             // Set the system clock
                                             // divisor to 2
}

////////////////////////////////////////////////////////////////////////
// UserMain
//
void UserMain( void *pd )
{
    OSChangePrio( MAIN_PRIO );
    EnableSerialUpdate();

    SimpleUart( 0, SystemBaud );
    assign_stdio( 0 );
```

```
    //
    // Waiting 16588 counts with an internal bus clock divisor of 2
    // equals approximately one PITR event every 1/1000th second.
    //
    SetUpPITR( 1 /* Use PITR 1 */, 16588 /* Wait 16588 clocks */, 1 /*
       Divide by 2 from table 17-3 */ );

    iprintf( "Application started\r\n" );

    while ( 1 )
    {
        OSTimeDly( TICKS_PER_SECOND );
        iprintf( "Pitr count = %ld\r\n", pitr_count );
    }
}
```