Technical Notes on using Analog Devices' DSP components and development tools
Contact our technical support by phone: (800) ANALOG-D or e-mail: dsp.support@analog.com
Or visit our on-line resources http://www.analog.com/dsp and http://www.analog.com/dsp/EZAnswers

Using C To Implement Interrupt-Driven Systems On ADSP-219x DSPs

Contributed by Joe B *March 19, 2003*

Introduction

This Engineer-to-Engineer note will describe the process for implementing a timer routine for the ADSP-219x family of DSPs using the C programming language. Please refer to chapter 12 of the "ADSP-219x DSP Hardware Reference" for further details regarding the timers.

The referenced code in this application note was verified using VisualDSP++™ 3.0 for ADSP-21xx DSPs and revision 2.0 of the ADDS-2191-EZLITE development board.

ADSP-219x Family Timers

Contrary to the ADSP-218x family of Digital Signal Processors (DSPs), the ADSP-219x family has three timers, each of which can be configured in any of three modes. On the assembly level, configuration of the timers and use of interrupts is fairly straightforward. However, in C, using interrupts and accessing the timer registers requires knowledge of some of the system header files and an understanding of how the C run-time environment works with embedded DSP programs. The process explained herein describes how to implement an interrupt-driven timer routine in C but the methods used can be applied to any C-coded interrupt routines.

The ADSP-218x C libraries defined three C-callable functions for timer management (*timer_on*, *timer_off*, and *timer_set*), which were created to make life for the C developer a little easier. In an ADSP-219x-based project, these functions no longer apply because now there are THREE possible sets of timer registers to associate these functions to, the registers for programming the timers are different, and the register memory space is also changed significantly from the ADSP-218x family of DSPs. Additionally, the 32-bit period, width, and count timer registers are now broken into low and high words, giving the ADSP-219x timers 65536 times the duration that the ADSP-218x timer had.

Using Features In VisualDSP++

VisualDSP++ 3.0 comes equipped with several built-in, or *intrinsic*, functions designed to make programming a DSP in a C environment even more user-friendly.

Formerly, memory-mapped registers were accessed by a referencing-dereferencing scheme using casted pointers (*(int *)). Access to non-memory-mapped registers in C was a tedious task, requiring embedded assembly source code. Now, the intrinsic functions *sysreg_read* and *sysreg_write* are available for manipulating these registers. In addition to these system register functions, the ADSP-219x tools feature two other intrinsic functions, *io_space_read* and *io_space_write*, which provides access to all the memory-mapped I/O space registers as well.

These four functions are defined in the header file *sysreg.h*. In this header, there is also an enumerated type, listing the system registers that can be accessed using *sysreg_read* and *sysreg_write*. One will also find all the bit manipulation instructions here as well (*sysreg_bit_clr, sysreg_bit_set, sysreg_bit_tgl*).

The *io_space_read* and *io_space_write* intrinsic functions require the architecture definition header file, *def2191.h*, be included also. This header details the addresses for all of the I/O space registers. It should be noted that the addressing scheme utilized in this header assumes that the user has already set the IO Page (IOPG) register appropriately, which is one of the system registers detailed in the enumerated type in *sysreg.h* (sysreg_IOPG). The IO Pages are also given convenient names in *def2191.h.*

For example, if the user wanted to configure their ADSP-2191 EZ-KIT to have LEDs 8-11 as outputs, this routine would do the trick:

```
#include <sysreg.h>
#include <def2191.h>
main( )
{
   sysreg_write (sysreg_IOPG, General_Purpose_IO);
      io_space_write (DIRS,  0x000F);
}
```

LEDs 8-11 map to the Programmable Flag Pins 0-3. Therefore, we want to configure PF0-3 to be outputs. This information is contained in the DIRS register, where a 1 means that the PFx pin is an output. The first thing we need to do is to make sure that we are on the correct IO Page for accessing the DIRS register. Because IOPG is a system register, this is accomplished using the *sysreg_write* intrinsic with the correct arguments. The first argument is the register to be written to, *sysreg_IOPG* (as defined in the enumeration in *sysreg.h*). The second argument is the value to be written to the IOPG register, *General_Purpose_IO*, which is #defined in *def2191.h* to be 0x06 (the page offset required to access the GPIO register set).

Now that the IOPG register is set appropriately to access the GPIO registers, we have access to the DIRS register. The second line of code uses the *io_space_write* intrinsic because DIRS is an IO space register. Here, the arguments are the address to be written to, *DIRS*, which is #defined in def2191.h to be 0x001 (the physical I/O address on IO page 6 of the DIRS register) and the value to be written to that address, 0x000F, where bits 0-3 are set to 1 to enable the corresponding PF pins 0-3 to be outputs.

Accessing the Timer0 registers is done in the same fashion, as you will see in the attached code. First, you must set the IOPG register using the *sysreg_write* intrinsic to have the offset for the *Timer_Page*. Then you'll have access to the timer register addresses and must use the *io_space_write* intrinsic to configure the associated registers as appropriate (see lines 27-34 of the attached source).

Interrupt Handling In C

The interrupt handling in C is unchanged from the ADSP-218x family tools from a coding perspective. Users still utilize the header-defined *interrupt(signal, subroutine)* module to take care of everything. First and foremost, this function associates a specific ISR module to be run for any given signal that could be received during run-time. The list of possible signals is detailed in the *signal.h* header file. This function also sets the correct bit in IMASK to enable servicing for that interrupt. In addition to this interrupt-registering scheme, the user will have to globally enable interrupts by using the intrinsic function *enable_interrupts()*.

Prior to version 6.1.1 of the ADSP-219x Compiler, global enabling of interrupts was automatically performed by the *interrupt(signal, subroutine)* module. From version 6.1.1 on, explicit use of the

enable_interrupts() module is required.

One new feature of the ADSP-219x family of DSPs is that the interrupts are now optionally configurable. There is a set of *interrupt priority registers*, namely IPR0, IPR1, IPR2, and IPR3, that can be configured prior to using the *interrupt* module in C to tell the hardware which interrupts have higher priority (i.e., which IMASK bit needs to be set to enable interrupt servicing). In this example, we will not touch these registers and will go with the default priority settings. Please refer to page C-3 of the "ADSP-219x Hardware Reference" for more information regarding interrupt priority.

In table C-2, the reader will see the "Peripheral Interrupts and Priority at Reset", where it is depicted that the Timer0 interrupt has an ID of 9. However, it should be noted that this ID is actually an *offset* from the four non-configurable highest-priority interrupts (Reset, Power-Down, Loop and PC Stack, and Emulation Kernel), which use interrupt vectors 0, 1, 2, and 3, respectively. Therefore, the actual signal used for the default Timer0 interrupt is *SIG_INT13*, not *SIG_INT9*. Because of this, the line of code for configuring the interrupt in C (line 36) reads as follows:

```
interrupt (SIG_INT13, Timer0_ISR);
```

As was already explained, *SIG_INT13* is the default signal number for the Timer0 interrupt. If you chose to utilize the option to prioritize your interrupts, just know that you will need to use a different SIG_INT value in this line of code based upon the priority value you gave to the Timer0 interrupt. For example, if you gave it an ID of 0 (highest priority after the four non-configurable interrupts), you'd be using *SIG_INT4*. Conversely, if you gave it an ID of 11 (lowest priority), you'd use *SIG_INT15*. In this example, *Timer0_ISR()* is the function that is called to service the Timer0 interrupt once it has been latched in the *interrupt latch* (IRPTL) register.

The interrupt gets latched into IRPTL based on the contents of the peripheral's status register. Since the timers' status bits are "sticky", they require a write-1-to-clear operation to be performed a few cycles before the RTI occurs. This will allow the status write to take effect before the RTI is executed, which will ensure that the same interrupt is not latched immediately by IRPTL.

The Code Example Itself

The following example code has been referenced throughout this application note. This example is the C equivalent to the assembly example provided in the Timer Chapter of the "ADSP-219x/2191 DSP Hardware Reference" on pages 12-15 through 12-18. This module sets up the timer initialization and interrupt routines for a timer in Pulsewidth Modulation (PWMOUT) Mode. The main module re-maps the Interrupt Vector Table (IVT) to internal memory, configures the appropriate LEDs to be outputs, sets up the timer registers, associates the ISR to the timer signal, and starts the timer. The ISR checks the polarity of the output PF pins 0-3 to check the status of LEDs 8-11 and toggles them upon each instance of a timer expiration.

Physically, on the ADDS-2191-EZ-KIT-LITE, LEDs 8, 9, 10, and 11 will alternately light/extinguish for roughly 1 second, assuming a 160 MHz clock.

Main Code

```
/* C-interrupts Example for ADSP-2191 EZ-KIT
   Created 10/12/2001 - JB
   Modified 3/14/2003 - JB */

#include <signal.h>    /* Interrupts */
#include <def2191.h>   /* MMRs */
#include <sysreg.h>    /* Intrinsics */

void Timer0_ISR();     /* ISR Prototype */
```

```
main()
{
   int temp;

   sysreg_write(sysreg_IOPG,
             Clock_and_System_Control_Page);
   temp = io_space_read(SYSCR);
   temp |= 0x0010;     /* Map IVT To Page 0 */
   io_space_write (SYSCR, temp);

// Clear/Reset All Interrupts
   sysreg_write(sysreg_IRPTL, 0x0000);
   sysreg_write(sysreg_ICNTL, 0x0000);
   sysreg_write(sysreg_IMASK, 0x0000);

   sysreg_write(sysreg_IOPG,
                General_Purpose_IO);
   io_space_write(DIR, 0x000F); // set outputs

/* Go To Timer Page – Initialize Timer0 */
   sysreg_write(sysreg_IOPG, Timer_Page);

/* SET: PWM_OUT Mode, Positive Active Pulse,
 * Count To End Of Period, Int Request Enable,
 * Timer_Pin Select */
   io_space_write(T_CFGR0, 0x001D);

/* Timer0 Period Register (High Word) */
   io_space_write(T_PRDH0, 0x0410);

/* Timer0 Period Register (Low  Word) */
   io_space_write(T_PRDL0, 0x5A00);

/* Timer0 Width Register (High Word) */
   io_space_write(T_WHR0,  0x0410);

/* Timer0 Width Register (Low  Word) */
   io_space_write(T_WLR0,  0x2D00);

/* Enable Timer0 */
   io_space_write(T_GSR0,  0x0100);

/* INT13 Is Default Timer0 Interrupt */
   interrupt(SIG_INT13, Timer0_ISR);
/* Globally Enable Interrupts */
   enable_interrupts();

   while(1);   /* wait for interrupts */
} /* end of main */
```

ISR Code

```
void Timer0_ISR()
{
   int Timer__Flag_Polarity;  /* Check Flags */

/* Go To Timer I/O Page */
   sysreg_write(sysreg_IOPG, Timer_Page);

/* Clear TMR0 Interrupt Latch Bit */
   io_space_write(T_GSR0, 0x1);

/* Go To GPIO I/O Page */
   sysreg_write(sysreg_IOPG, General_Purpose_IO);

/* Get Values Of PF Flags */
   Timer__Flag_Polarity = io_space_read(FLAGS);
   if ((Timer__Flag_Polarity & 0x000F) == 0)
        /* If The LEDs Aren't On */
        io_space_write(FLAGS, 0x000F);
                      /* turn EZ-KIT LEDs ON */
        else          /* otherwise they are ON */
        io_space_write(FLAGC, 0x000F);
                      /* turn EZ-KIT LEDs OFF */
} // end Timer0_ISR
```

References

[1] ADSP-219x/2191 DSP Hardware Reference Manual, First Edition, July 2001

Document History

Version	Description
March 19, 2003 by Joe B	Initial Release