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 Create Interrupt-Driven Systems On Blackfin® Processors

Contributed by Joe B.

May 28, 2003

Introduction

This Engineer-to-Engineer note will describe the process for implementing an interrupt-driven general-purpose timer routine for the Blackfin® Processor family using the C programming language.

The referenced code in this application note was verified using VisualDSP++™ 3.1 for Blackfin and the project was run on the EZ-KIT Lite™ evaluation systems for both the ADSP-BF535 and the ADSP-BF533 Blackfin Processors (ADDS-BF535-EZLITE, Rev 1.0 and ADDS-BF533-EZLITE, Rev 1.0).

Timer Programming Model

When coding in a pure assembly environment, configuration of the timers and use of interrupts is fairly straightforward, as described in the Hardware Reference Manuals. However, when programming in C, using interrupts and accessing the timer registers requires knowledge of some of the system header files and an understanding of the C run-time environment as it relates to embedded system programming. The process explained herein describes how to implement an interrupt-driven timer routine in C, but the methods employed can be applied to any C-coded interrupt routines for the Blackfin processors.

Two test platforms are necessary for this paper because the demonstration utilizes the on-board LEDs on the Blackfin evaluation platforms. On the ADDS-BF535-EZKIT, the four LEDs are mapped directly to the general purpose flag pins, whereas, on the ADDS-BF533-EZKIT, the six LEDs are accessible only through the on-board flash port, which requires an alternate set of instructions in the Interrupt Service Routine (ISR) in order to update the LED display. The differences in the software will be discussed later.

Using Features In VisualDSP++

VisualDSP++ 3.1 comes with a set of header files that makes programming the Blackfin processors in a C environment simpler. These headers can be found under the Blackfin include directory and are named using the "cdef" prefix to denote *C definitions*.

Memory-mapped registers are accessed in C via a referencing-dereferencing scheme using casted pointers, like this:

```
volatile long *pIMASK     = (long *)IMASK;
```

IMASK is defined for the ADSP-BF535 in "defblackfin.h" and for the ADSP-BF533 in "def_LPblackfin.h" to be the address of the IMASK register in Blackfin address space (0xFFE02104). If the user wanted to set bit 0 of the IMASK register in C, a proper 32-bit read-modify-write would be performed indirectly through the pointer defined above:

```
*pIMASK |= 0x1;
```

Using this convention, the label "pIMASK" is a data element of type `long int *`, which consumes 4 bytes of memory and whose sole purpose is to hold the address of IMASK so that the user can indirectly write a 1 to the register by accessing it through its mapped address. This convention wastes 4 bytes of memory for every register accessed in this fashion!

An alternate method of manipulating the IMASK register without consuming memory for the pointer is to force a casted pointer in the code to modify the address directly:

```
        *(volatile long *)IMASK |= 0x1;
```

In the Blackfin C-defines header files, all of the memory-mapped registers have already been set up for access in C in a series of `#define` statements, like this:

```
   #define pIMASK ((volatile long *)IMASK)
```

This method allows the user to use the same C syntax in their code, `*pIMASK |= 0x1`, to modify the register's contents without dedicating memory for the pointer. Instead, the compiler's pre-processor substitutes the casted pointer syntax for what appears to be pointer-modifying code. So, if one knows the name of the register, one can easily access that register using standard C code for pointer modification:

```
            *pREGISTER_NAME = VALUE;
```

These "cdef" header files were created for each individual processor based upon the widths of the registers being represented (i.e., a 16-bit register uses the `short int` type, which the Blackfin compiler treats as 16-bit data, whereas a 32-bit register is represented as a `long int` or `int` type).

Blackfin Interrupts (Hardware)

The Blackfin family of processors has an intricate interrupt system that includes core interrupts, enabled in the IMASK register, and peripheral interrupts, which are configured in the System Interrupt Controller (SIC). These system-level peripheral interrupts are optionally user-configurable in a set of *System Interrupt Assignment Registers*: SIC_IAR0, SIC_IAR1, and SIC_IAR2. Refer to Chapter 4 (*Program Sequencer*) of the appropriate Hardware Reference Manual (either ADSP-BF535 or ADSP-BF533) for more information regarding interrupt priority assignments.

Several peripheral sources can share the *same* SIC interrupt assignment. Those sources with common interrupt assignments share the same General-purpose interrupt (IVGx). Table 4-8 of the Hardware Reference Manual depicts the "*IVG-Select Definitions*", which is where the values programmed into the specific fields of the SIC_IARx registers are translated into an IVG priority. Use the IVG priority detailed here to determine which IMASK bit to set in the Core Event Controller (CEC) to enable interrupt handling for this group of peripherals.

In addition to having a group of interrupt sources enabled in the CEC, handling for each source's individual interrupt is enabled separately on the SIC level in the SIC_IMASK register. For example, one could set each peripheral's IVG priority to the *same* level, thus making one group of 24 potential interrupt sources. If only *one* of those peripheral interrupts is enabled in SIC_IMASK, then the IVG effectively describes that single interrupt source because that single source is the only source enabled at the SIC level to be recognized by the core as a peripheral interrupt source.

An abridged overview of how to set peripheral interrupts up correctly goes like this:

1. Enable peripheral's individual interrupt in SIC_IMASK register.
2. Program the interrupt priority levels into the SIC_IARx registers. This step is *optional*.
3. Use the values in the SIC_IARx registers with Table 4-8 of the Hardware Reference Manual to determine which interrupt group (IVGx) the peripheral of interest is assigned to.
4. Set the appropriate IVGx bit in the IMASK register.

This example uses the interrupt of the General Purpose Timer 0 to generate LED activity on the evaluation boards. Following the process detailed above, the first step is to configure the SIC_IMASK register. The Timer0 interrupt is enabled in SIC_IMASK by setting either bit 14 (ADSP-BF535) or bit 16 (ADSP-BF533):

```
*pSIC_IMASK = 0x00004000; // ADSP-BF535
*pSIC_IMASK = 0x00010000; // ADSP-BF533
```

For 0.x revisions of the ADSP-BF535, the active low setting is used to *enable* interrupts in SIC_IMASK:

```
*pSIC_IMASK = ~0x00004000;
```

This anomaly was fixed in revision 1.0 of the ADSP-BF535 and doesn't apply to any other ADSP-BF53x processor.

Once the interrupt is enabled in the SIC, the next step is to optionally program the interrupt priority and assign the peripheral interrupt to an IVG. This example doesn't reconfigure the interrupt priorities but uses the default values in the SIC_IARx registers.

According to the reset values of the SIC_IARx registers, the default value in the Timer0 field of SIC_IAR1 is 0x4. Table 4-8 of the Hardware Reference Manual tells us that the value 0x4 maps to IVG11, therefore, IVG11 is the default core interrupt channel for the Timer0 interrupt. So, the last step in setting up peripheral interrupts in the hardware is to enable IVG11 in the CEC's IMASK register.

If this were a Core Timer application, the steps required to initialize the SIC would not be needed because the Core Timer interrupt is a Core Event only.

Once the interrupts are properly configured, the final step is to have the software in place for proper interrupt servicing.

Software And Code Flow

The Blackfin processors feature 16 Event Vector Table (EVT0-15) registers, which hold the vector addresses of the individual core interrupt channels. The program will jump to these addresses whenever an event is latched by ILAT and enabled in IMASK in the CEC.

In the assembly examples shipped with VisualDSP++ 3.1, the "startup.asm" driver sets up a default vector table using the following assembler commands:

```
p0.l = lo(EVT2);  p0.h = hi(EVT2);
r0.l = _NHANDLER; r0.h = _NHANDLER;
[p0++] = r0;      // NMI Handler (Int2)
```

This code sets up a pointer register (P0) to write to the EVT2 register and then writes the address of the `_NHANDLER` label to EVT2. The result is that any external Non-Maskable Interrupt (NMI) detected by the CEC will force an immediate vector to the address of `_NHANDLER`, which is the label associated with the NMI's ISR.

In C, however, the "startup.asm" is substituted with the C run-time library. The library function

```
register_handler(IVGx, ISR_Name)
```

assigns the name of the ISR function, `ISR_Name`, to the core event represented by `IVGx` by writing the ISR function's address to the corresponding EVTx register. The function prototype and enumeration of interrupt kinds can be found in the "sys\exception.h" header file.

Checking the "sys\exception.h" header file, it can be seen that the signal number for IVG11 is `ik_ivg11.` Therefore, use of this macro in this example is:

```
register_handler(ik_ivg11, Timer0_ISR);
```

In addition to setting the EVT11 register to contain the address of the `Timer0_ISR` label, this macro *also* sets the IVG11 bit in the IMASK register and globally enables interrupts.

In addition to registering the interrupt handler, the interrupt routine itself must *also* be set-up, which is accomplished using a second library function:

```
EX_INTERRUPT_HANDLER (ISR_Name)
```

The `EX_INTERRUPT_HANDLER` macro can be used in two ways. It *must* be used as the function signature for all ISRs. It generates the label `ISR_Name` to be used by the `register_handler` function later in the source code. It also tells the compiler that the module is an ISR, which means that processor context must be saved and restored upon entry and exit, respectively, and that the final instruction in the assembly produced by the compiler must be a return from interrupt (`RTI;`). Basically, the `EX_INTERRUPT_HANDLER` macro hides special compiler pragma directives from the user.

The `EX_INTERRUPT_HANDLER` macro can *also* be globally used as a function prototype if the `register_handler(IVGx, ISR_Name)` function is called *before* the `ISR_Name` label is defined.

A compiler error will be generated if the `register_handler` function attempts to use the `ISR_Name` label as an argument *before* the `EX_INTERRUPT_HANDLER` macro defines the `ISR_Name` label. If you get "undefined label" compiler errors, try using the `EX_INTERRUPT_HANDLER` macro to prototype the ISR globally.

Once the ISR is registered and all the hardware considerations have been accounted for, the last thing needed is an appropriate ISR. For peripheral interrupts, the ISR code *must* clear the IRQ in the peripheral's hardware. In most cases, this is done by performing a write-one-to-clear (W1C) operation to the associated IRQ bit. For the Timer0 peripheral, a 1 must be written to the TIMIL0 location of the Timer's Status register in order to clear the IRQ before the ISR completes and application code begins running again.

In this example, if the core did not explicitly clear the IRQ in the Timer0 ISR, the same interrupt would be immediately latched again upon completion of the ISR. Even though the ISR executes, the SIC continues to monitor the status of the timer's IRQ, which generated the event to begin with. If this IRQ remains active, the SIC will continue to hold the bit in the SIC Interrupt Status Register (SIC_ISR) active, which will result in the interrupt staying latched in the core (ILAT). Once the core executes the `RTI;` instruction in the ISR, the IMASK register is restored and the ISR would be triggered again because the CEC sees that an enabled interrupt is currently latched.

If several sources share the same IVG priority, it is up to the user to implement a software solution to poll each enabled peripheral's IRQ in the group to determine which source generated the interrupt request.

The Code Example

The code in Listing 1 at the end of this note was developed for the ADSP-BF535 EZ-KIT. The code was then modified to perform the same functions on the ADSP-BF533 EZ-KIT. The two critical differences between the two evaluation platforms are:

1. ADSP-BF533 has a TIMER_ENABLE register and uses 32-bit timer registers whereas the ADSP-21535 uses pairs of registers to represent the upper 16 bits and lower 16 bits separately
2. ADSP-BF533 EZ-KIT Lite LEDs are accessed via the on-board flash rather than the PFx flags, as previously mentioned

In Listing 1, there is a commented pre-processor directive, `#define BF533_PROJECT`, at the top of the code that must be uncommented to build a project to run on an ADSP-BF533 EZ-KIT Lite.

Throughout the source code, there are several pre-processor directives based upon whether or not `BF533_PROJECT` is defined. If `BF533_PROJECT` *is* defined, the correct ADSP-BF533 code is built into the project. Otherwise, the code is built assuming an ADSP-BF535 project.

This application code initializes Timer0 in Pulsewidth Modulation (PWMOUT) Mode. It 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 code increments a counter and displays the counter value in binary on the LEDs. Once all LEDs are lit, the counter resets to 0. The ISR also clears the Timer0 interrupt request.

Listing 2 is the C source for the ADSP-BF535 EZ-KIT and Listing 3 is the source code for the ADSP-BF533 EZ-KIT, without the pre-processor directives.

mixed.c

```
/*****************************************************************************
 * Uncomment the following pre-processor directive if building code for the   *
 * ADSP-BF533 EZ-KIT.  ADSP-BF535-EZKIT users do not need to modify this.    *
 * Including BF533_PROJECT changes the legacy ADSP-BF535 code to use the     *
 * BF533 header, redefine the MAX value on the LEDs, use the BF533 timer     *
 * registers, and configure the flash to use the LEDs to output patterns     *
 *****************************************************************************/

//#define        BF533_PROJECT     // Uncomment for BF533-EZKIT Code

#ifdef   BF533_PROJECT
    #include <cdefBF533.h>         // BF533 Register Pointer Definitions
#else
    #include <cdefBF535.h>         // BF535 Register Pointer Definitions
#endif

#include <sys/exception.h>         // Interrupt Handling Header

// Prototypes
#ifdef   BF533_PROJECT
    void Init_EBIU_Flash(void);    // Flash Init Code Needed For BF533-EZKIT
#else
    void Init_Flags(void);         // Flags Mapped To LEDs For BF535-EZKIT
#endif

void Init_Timer(void);
void Init_Interrupts(void);

// Selects All LEDs, Also Used For ISR Compare For All LEDs Lit
#ifdef   BF533_PROJECT
    #define MAX_LED_DISPLAY        0x3F   // 6 LEDs on BF533, all LIT
#else
    #define MAX_LED_DISPLAY        0xF    // 4 LEDs on BF535, all LIT
#endif

#ifdef   BF533_PROJECT       // Flash Register Pointers For Updating BF533 LEDs
    #define pFlashA_PortB_Dir      (volatile unsigned char *)0x20270007
    #define pFlashA_PortB_Data     (volatile unsigned char *)0x20270005
#endif
```

```
EX_INTERRUPT_HANDLER(Timer_ISR)
{
    static int count=-1;

    if (++count > MAX_LED_DISPLAY)              // check for all LEDs lit
         count=0;                               // clear all LEDs on wrap

    #ifdef BF533_PROJECT
         *pTIMER_STATUS             = 1;        // Clear Timer0 Interrupt
         *pFlashA_PortB_Data        = 0x0;      // clear LEDs
         *pFlashA_PortB_Data        = count;    // write new LED pattern to Port B
    #else
         *pTIMER0_STATUS            = 1;        // Clear Timer0 Interrupt
         *pFIO_FLAG_C               = 0xF;      // clear LEDs
         *pFIO_FLAG_S               = count;    // display new LED pattern
    #endif
} // end Timer_ISR

main()
{
    #ifdef BF533_PROJECT                  // If this is for the BF533-EZKIT…
         Init_EBIU_Flash();               // initialize EBIU for FLASH interface
    #else                                 // Otherwise, it is for a BF535-EZKIT…
         Init_Flags();                    // Set All 4 LED PFx Flags To Outputs
    #endif

    Init_Timer();                         // Initialize the Timer0 Registers
    Init_Interrupts();                    // Configure the Interrupts

    while(1);                             // wait forever for interrupts
} // end main

#ifdef BF533_PROJECT
    void Init_EBIU_Flash(void)
    {
         *pEBIU_AMBCTL0 = 0x7bb07bb0;
         *pEBIU_AMBCTL1 = 0x7bb07bb0;
         *pEBIU_AMGCTL        = 0x000f;

         *pFlashA_PortB_Dir = MAX_LED_DISPLAY;        // 6 LEDs
    } // end Init_EBIU_Flash
#else
    void Init_Flags(void)
    {
         *pFIO_DIR = 0xF;                 // Configure PF0-3 As Outputs
    } // end Init_Flags
#endif

void Init_Timer(void)
{
         *pTIMER0_CONFIG      = 0x0019;         // PWM Mode, Period Count,
Interrupt

    #ifdef BF533_PROJECT
         *pTIMER0_PERIOD      = 0x01000000;     // Configure Timer Period
         *pTIMER0_WIDTH       = 0x00800000;     // Configure Timer Width
         *pTIMER_ENABLE       = 0x0001;         // Enable Timer
    #else
         *pTIMER0_PERIOD_HI   = 0x0100;         // Configure Timer Period
```

```
        *pTIMER0_PERIOD_LO   = 0x0000;
        *pTIMER0_WIDTH_HI    = 0x0080;         // Configure Timer Width
        *pTIMER0_WIDTH_LO    = 0x0000;
        *pTIMER0_STATUS      = 0x0100;         // Enable Timer0
    #endif
} // end Init_Timer

void Init_Interrupts(void)
{
    // Enable the SIC interrupt
#ifdef BF533_PROJECT
    *pSIC_IMASK = 0x00010000;      // Timer0 Default IRQ Is Bit 16 on BF533
#else // if using Rev 1.0 or higher of BF535 silicon, delete the ~ below
    *pSIC_IMASK = ~0x00004000;     // Timer0 Default IRQ Is Bit 14 on BF535
#endif
    // install the handler (also sets IVG11 in IMASK)
    register_handler(ik_ivg11, Timer_ISR);
} // end Init_Interrupts
```

Listing 1: mixed.c

BF535_C_IRQ.c

```
#include <cdefBF535.h>       // BF535 Register Pointer Definitions
#include <sys/exception.h>   // Interrupt Handling Header

// Prototypes
void Init_Flags(void);       // Flags Mapped To LEDs For BF535-EZKIT
void Init_Timer(void);
void Init_Interrupts(void);

// Selects All LEDs, Also Used For ISR Compare For All LEDs Lit
#define MAX_LED_DISPLAY      0xF   // 4 LEDs on BF535, all LIT

EX_INTERRUPT_HANDLER(Timer_ISR)
{
    static int count=-1;

    if (++count > MAX_LED_DISPLAY)             // check for all LEDs lit
        count=0;                               // clear all LEDs on wrap

    *pTIMER0_STATUS          = 1;        // Clear Timer0 Interrupt
    *pFIO_FLAG_C             = 0xF;      // clear LEDs
    *pFIO_FLAG_S             = count;    // display new LED pattern
} // end Timer_ISR

main()
{
    Init_Flags();            // Set All 4 LED PFx Flags To Outputs
    Init_Timer();            // Initialize the Timer0 Registers
    Init_Interrupts();       // Configure the Interrupts

    while(1);                // wait forever for interrupts
} // end main

void Init_Flags(void)
{
    *pFIO_DIR = 0xF;                // Configure PF0-3 As Outputs
} // end Init_Flags

void Init_Timer(void)
{
    *pTIMER0_CONFIG     = 0x0019;         // PWM Mode, Period Count, Interrupt
    *pTIMER0_PERIOD_HI  = 0x0100;         // Configure Timer Period
    *pTIMER0_PERIOD_LO  = 0x0000;
    *pTIMER0_WIDTH_HI   = 0x0080;         // Configure Timer Width
    *pTIMER0_WIDTH_LO   = 0x0000;
    *pTIMER0_STATUS     = 0x0100;         // Enable Timer0
} // end Init_Timer

void Init_Interrupts(void)
{
    // Enable SIC interrupt, if using Rev 1.0 or higher of ADSP-BF535 silicon,
    // delete the ~ below
    *pSIC_IMASK = ~0x00004000;      // Timer0 Default IRQ Is Bit 14

    // install the handler (also sets IVG11 in IMASK)
    register_handler(ik_ivg11, Timer_ISR);
} // end Init_Interrupts
```

Listing 2: BF535_C_IRQ.c

BF533_C_IRQ.c

```
#include <cdefBF533.h>       // BF533 Register Pointer Definitions
#include <sys/exception.h>   // Interrupt Handling Header

// Prototypes
void Init_EBIU_Flash(void);  // Flash Init Code Needed For BF533-EZKIT
void Init_Timer(void);
void Init_Interrupts(void);

// Selects All LEDs, Also Used For ISR Compare For All LEDs Lit
#define MAX_LED_DISPLAY      0x3F  // 6 LEDs on BF533, all LIT

// Map Flash Port B Registers For LED Access
#define pFlashA_PortB_Dir    (volatile unsigned char *)0x20270007
#define pFlashA_PortB_Data   (volatile unsigned char *)0x20270005

EX_INTERRUPT_HANDLER(Timer_ISR)
{
    static int count=-1;

    if (++count > MAX_LED_DISPLAY)             // check for all LEDs lit
         count=0;                              // clear all LEDs on wrap

    *pTIMER_STATUS              = 1;        // Clear Timer0 Interrupt
    *pFlashA_PortB_Data         = 0x0;      // clear LEDs
    *pFlashA_PortB_Data         = count;    // write new LED pattern to Port B

} // end Timer_ISR

main()
{
    Init_EBIU_Flash();              // initialize EBIU for FLASH interface
    Init_Timer();                   // Initialize the Timer0 Registers
    Init_Interrupts();              // Configure the Interrupts

    while(1);                       // wait forever for interrupts
} // end main

void Init_EBIU_Flash(void)
{
    *pEBIU_AMBCTL0      = 0x7bb07bb0;
    *pEBIU_AMBCTL1      = 0x7bb07bb0;
    *pEBIU_AMGCTL       = 0x000f;

    *pFlashA_PortB_Dir = MAX_LED_DISPLAY;       // 6 LEDs
} // end Init_EBIU_Flash

void Init_Timer(void)
{
    *pTIMER0_CONFIG     = 0x0019;         // PWM Mode, Period Count, Interrupt
    *pTIMER0_PERIOD     = 0x01000000;     // Configure Timer Period
    *pTIMER0_WIDTH      = 0x00800000;     // Configure Timer Width
    *pTIMER_ENABLE      = 0x0001;         // Enable Timer
} // end Init_Timer

void Init_Interrupts(void)
{
    // Enable the SIC interrupt
    *pSIC_IMASK = 0x00010000;       // Timer0 Default IRQ Is Bit 16

    // install the handler (also sets IVG11 in IMASK)
    register_handler(ik_ivg11, Timer_ISR);
} // end Init_Interrupts
```

Listing 3: BF533_C_IRQ.c

References

[1] ADSP-BF535 Blackfin Hardware Reference. Revision 1.0, November 2002. Analog Devices, Inc.

[2] ADSP-BF533 Blackfin Hardware Reference. Preliminary Revision, March 2003. Analog Devices, Inc.

[3] VisualDSP++ 3.0 C/C++ Compiler and Library Manual for Blackfin Processors. Second Revision, April 2002. Analog Devices, Inc.

Document History

Version	Description
May 28, 2003 by Joe B.	Updated interrupt service routine example code.
April 30, 2003 by Joe B.	Initial Release