Interrupts C

C.1 INTRODUCTION

The C Runtime Library provides function for supporting interrupts service routines written C. These functions install your C function as the interrupt handler for the designated interrupt.

C.2 HARDWARE INTERRUPTS

There are two types of interrupts that are supported in the ADSP-21xxx processors. The type used by most applications are the hardware interrupts. These interrupts are supported in hardware on the processor.

The ADSP-21020 supports these hardware interrupts:

`SIG_SOVF`	Loop stack, status stack overflow interrupt
`SIG_TMZ0`	Timer (high priority) expired interrupt
`SIG_IRQ3`	Interrupt 3
`SIG_IRQ2`	Interrupt 2
`SIG_IRQ1`	Interrupt 1
`SIG_IRQ0`	Interrupt 0
`SIG_CB7`	DAG7 circular buffer overflow interrupt
`SIG_CB15`	DAG15 circular buffer overflow interrupt
`SIG_TMZ`	Timer (low priority) expired interrupt
`SIG_FIX`	Fixed-point overflow interrupt
`SIG_FLTO`	Floating-point overflow interrupt
`SIG_FLTU`	Floating-point underflow interrupt
`SIG_FTLI`	Floating-point invalid operation interrupt
`SIG_USR0`	User interrupt 0
`SIG_USR1`	User interrupt 1
`SIG_USR2`	User interrupt 2
`SIG_USR3`	User interrupt 3
`SIG_USR4`	User interrupt 4
`SIG_USR5`	User interrupt 5
`SIG_USR6`	User interrupt 6
`SIG_USR7`	User interrupt 7

C Interrupts

The ADSP-2106x SHARC supports these hardware interrupts:

SIG_SOVF	Status stack or Loop stack overflow or PC stack full
SIG_TMZ0	Timer = 0 (high priority option)
SIG_VIRPTI	Vector Interrupt
SIG_IRQ2	Interrupt 2
SIG_IRQ1	Interrupt 1
SIG_IRQ0	Interrupt 0
SIG_SPR0I	SPORT0 receive DMA channel
SIG_SPR1I	SPORT1 receive (or LBUF0) DMA channel
SIG_SPT0I	SPORT0 transmit DMA channel
SIG_SPT1I	SPORT1 transmit (or LBUF1) DMA channel
SIG_LP2I	Link buffer 2 (LBUF2) DMA channel
SIG_LP3I	Link buffer 3 (LBUF3) DMA channel
SIG_EP0I	External Port DMA channel 0 (or LBUF4)
SIG_EP1I	External Port DMA channel 1 (or LBUF5)
SIG_EP2I	External Port DMA channel 2
SIG_EP3I	External Port DMA channel 3
SIG_LSRQ	Link service request
SIG_CB7	Circular buffer 7 overflow
SIG_CB15	Circular buffer 15 overflow
SIG_TMZ	Timer = 0 (low priority option)
SIG_FIX	Fixed-point overflow
SIG_FLTO	Floating-point overflow exception
SIG_FLTU	Floating-point underflow exception
SIG_FLTI	Floating-point invalid exception
SIG_USR0	User interrupt 0
SIG_USR1	User software interrupt 1
SIG_USR2	User software interrupt 2
SIG_USR3	User software interrupt 3

The interrupts are listed in order of priority. In the C runtime environment, the ADSP-21xxx interrupt nesting mode is **on**. This means that if an interrupt service routine is in progress and a higher priority interrupt occurs, the higher priority interrupt is serviced immediately.

As an example, assume that your application was setup to respond to interrupts 0 and 3. When interrupt 0 occurs, your application begins to execute the code that you had designated as the interrupt handler for that interrupt. When interrupt 3 occurs, the program vectors to that handler and services interrupt 3 immediately.

After the interrupt 3 service routine completes, the execution returns to the interrupt 0 service routine. After the interrupt 0 service routine completes, execution continues at the point the first interrupt occurred.

This example appears like this in your program:

```
interrupt(SIG_IRQ3, irq3handler);
interrupt(SIG_IRQ0, irq0handler);
```

The fixed- and floating-point exceptions are triggered by the corresponding bits in the STKY register. It is possible that one of these interrupts occurred during normal program operation. Be careful when you setup a signal handler to deal with these interrupts.

C.3 ANSI C INTERRUPTS

Besides the hardware interrupts available with the ADSP-21xxx processors, there are 6 interrupts that are required by the ANSI C standard. They are:

`SIGABRT`	Abort signal
`SIGFPE`	Floating point exception
`SIGILL`	Illegal instruction
`SIGINT`	System interrupt
`SIGSEGV`	Segmentation violation
`SIGTERM`	Software termination signal

These signals were implemented to support the ANSI standard, they are not supported in hardware on the ADSP-21000 family of processors. The `raise()` function can invoke them, but they do not occur independently.

You can use these interrupts in an application. They follow the same rules as the hardware interrupts, but since they can only be invoked with a call to the `raise()` function, it is much more efficient to call the routines directly.

C.4 FEATURES OF INTERRUPT SERVICE ROUTINES

An interrupt service routine (ISR) is a special routine that is executed outside of the normal program flow. An ISR is invoked in response to a particular interrupt occurring at an undetermined time.

Since an interrupt occurs at an unknown time, it cannot return a value directly to a program. Thus, all ISRs are of return-type void. The void type indicates that no value is returned from the routine. According to the C standard, all ISRs are called with a number indicating the interrupt that invoked them as their parameter. This tells the routine what caused its invocation. The prototype of an interrupt service routine is always:

```
   void handler1(int sig);
```

This prototype indicates that the function `handler1` accepts an integer parameter (the signal name), and has no return value. When you create your ISR, be sure that it has this prototype. If the prototype is incorrect, the compiler warns you with a message similar to this one:

```
foo.c: In function 'main':
foo.c:10: warning: passing arg 2 of '_signal060'
from
                      incompatible pointer type
```

This warning indicates that the handler routine does not have the correct prototype. Although your program might work even with this warning, you should determine and correct the cause.

One way for an ISR to return information to the main program is through the use of global variables. These variables are accessible to both the handler and the main program. When an ISR needs to pass information to the main program, it sets a global variable to that value. When the main program has time to deal with the information, it reads the global variable.

Declare these global variables as `volatile.` The `volatile` qualifier tells the compiler that this variable can change in ways and at times unexpected by normal program flow.

The indication that a variable is `volatile` causes the compiler to treat it specially. Since a `volatile` variable can change outside of the scope of normal program flow, it is not optimized away, even if it does not appear to be set anywhere.

For example, if your program has a loop that tests the state of a variable that is never modified in the loop, the compiler thinks that it can remove the test from the loop (since the value cannot change within the loop).

```
void do_something(int param1);
int global_variable;

main()
{
  global_variable = 0;
  while(1)   /* Stay in this loop */
  {
     if( global_variable )
       do_something(global_variable);
  }
}
```

In the section of code shown, the compiler can see that `global_variable` is set to zero outside of the loop and not modified within the loop. It assumes that it can execute the test once outside of the loop, and not need to perform it at every iteration.

In general, this is a safe assumption, and can result in a significant performance improvement for your code. If, on the other hand, `global_variable` was modified by an interrupt service routine outside of the scope of normal program flow, this assumption is incorrect.

The `volatile` qualifier tells the compiler that it cannot make any assumptions about the variable, and that it must perform the test every iteration of the loop.

When you write an interrupt service routine, try to spend as little time in the ISR as possible. When an interrupt is serviced, all lower priority interrupts are not serviced until the higher priority interrupt exits.

To reiterate, consider interrupts as exceptional situations. The less time spent in an interrupt service routine, the better.

C.5 THE DIFFERENCE BETWEEN SIGNAL() & INTERRUPT()

There are two ways to setup a routine as a handler. The first way, which is ANSI compliant, is to use the `signal()` routine. The `signal()` routine (as defined by the ANSI standard) setups up an ISR to respond to **one** invocation of the specified interrupt. After the ISR has responded once, the interrupt will be ignored in the future.

Using the `signal()` function, the only way to respond to an interrupt more than once is to reinitialize the ISR with the `signal()` function every time the ISR is executed.

Since most systems designed with the ADSP-21000 family of processors need to respond to some interrupts continuously, Analog Devices created an extension to the standard that sets up an ISR to respond continuously to an interrupt. The extension is the `interrupt()` routines. These routines takes the same parameters as the `signal()` routine; the only difference is that an ISR that has been setup with `interrupt()` responds continuously to an interrupt, while one that is setup with `signal()` only responds once.

C.6 INTERRUPT DISPATCHERS

The ADSP-21000 Family Runtime C Library provides interrupt dispatcher routines.

The `signal.h` header file provides function prototypes for the standard ANSI `signal.h` routines and also for several ADSP-21xxx family extensions such as `interrupt()` and `clear_interrupt()`. It also includes ANSI-standard signal handling functions of the C library.

The signal handling functions process conditions (hardware signals) that can occur during program execution. They determine the way that your C program responds to these signals. The functions are designed to process such signals as external interrupts and timer interrupts.

There are three interrupt dispatchers, *normal*, *fast*, and *super*. The interrupt dispatchers let you disable interrupts and modify the processor’s MODE1 register from an interrupt handler. The *normal interrupt dispatcher* and *fast interrupt dispatcher* preserve MODE1 register writes in an interrupt handler after the interrupt is serviced, but the *super interrupt dispatcher* does not. The interrupt dispatchers are described below.

Normal Interrupt Dispatcher—Saves all scratch registers and the loop stack. Do loop and interrupt nesting is allowed because data is pushed onto the stack. Requires approximately 125 cycles for interrupt overhead. To access this dispatcher, use `interrupt()` or `signal()` .

Fast Interrupt Dispatcher—Does not save the loop stack, therefore DO loop handling is restricted to six levels (specified in hardware). If the interrupt service routine (ISR) uses one level of nesting, your code cannot exceed five levels. Interrupt nesting is not restricted (20 levels available). This dispatcher does not send the interrupt number type to the ISR as a parameter. Requires approximately 60 cycles for interrupt overhead. To access this dispatcher, use `interruptf()` or `signalf()`.

Super Interrupt Dispatcher—Does not save the loop stack, therefore do loop handling is restricted to six levels (specified in hardware). Interrupt nesting is disabled. This dispatcher does not send the interrupt number type to the ISR as a parameter. This dispatcher uses the secondary register set. This dispatcher only works with the ADSP-21020 (Rev. 1) and with all ADSP-2106x SHARC processors. Requires approximately 30 cycles for interrupt overhead. To access this dispatcher, use `interrupts()` or `signals()` .

C.7 THE IDLE() FUNCTION

If your main program waits for an interrupt to occur before doing something, you can use the `idle()` function supplied in the C Runtime Library. The `idle()` routine executes the IDLE function of the ADSP-21xxx processor. The IDLE instruction causes the ADSP-21xxx to wait at the IDLE instruction until an interrupt occurs. When an interrupt occurs, the ISR is executed, and normal program flow continues after the IDLE instruction.

C.8 THE CLEAR_INTERRUPT() FUNCTION

This function is used to clear a pending interrupt. When an interrupt occurs, the processor latches that interrupt in the IRPTL register. The interrupt remains latched until it is serviced or cleared. If your application is about to setup an ISR for an interrupt, a meaningless occurrence of that interrupt could be pending. If you do not wish to service it, clear the interrupt before calling `signal()` or `interrupt()`.

The `clear_interrupt()` function takes the signal name as its parameter.

C.9 THE RAISE() FUNCTION

This function is used to cause an interrupt to be invoked manually. It sets the appropriate bit in the IRPTL register. If the interrupt is not masked, the processor vectors to the ISR (through the dispatcher).

C.10 A SIMPLE EXAMPLE

This section provides a very simple example of how to set up an interrupt service routine and how to interface with your main program. It uses the techniques discussed in the preceding sections.

```
#include <signal.h>

/* These are function that do useful work. */
extern void do_timer_things(void);
extern void do_irq0_things(void);

/* Be sure to declare any variables reference by an ISR as volatile
*/
volatile int timer_expired;
volatile int irq0_occurred;

/* Be sure to have the correct prototype for an ISR */
void timer_handler(int signal)
{
    timer_expired = 1;
}

void irq0_handler(int signal)
{
    irq0_occurred = 1;
}

main()
{
/* Set timer expired to zero at the beginning */
    timer_expired = 0;
    irq0_occurred = 0;

/* Set up the routines to respond to interrupts */
    interrupt(SIG_IRQ0, irq0_handler);
    interrupt(SIG_TMZ, timer_handler);
/* Set up the timer */
    timer_set((unsigned int)10000, (unsigned int)10000);
    timer_on();

/* Loop continuously and respond to interrupts */
    while(1)
    {
        idle();   /* Return from this function after an interrupt */

/* If the timer has expired, clear flag and do something*/
        if( timer_expired )
        {
            timer_expired = 0;
            do_timer_things();
        }

/* If irq0 has occurred, clear flag and do something*/
        if( irq0_occurred )
        {
            irq0_occurred = 0;
            do_irq0_things();
        }

    }

}
```

You can use this example as a shell for your programs that use interrupts. There are two interrupts that we use this example. The first is `irq0`, which, in this example, is connected to a push button. When the button is depressed, an `irq0` interrupts occurs. The global variable `irq0_occurred` informs the main program that this has occurred.

The second interrupt we use is the timer. In the main routine we initialize the timer to trigger an interrupt every 10,000 cycles. This means that the `timer_handler` routine is executed every 10,000 cycles.

C.11 INTERRUPT OVERHEAD

This section discuss interrupt overhead and processing. It assumes that you have a knowledge of the internals of the ADSP-21xxx processor. If you encounter terms that you are unfamiliar with, refer to the *ADSP-21020 User's Manual* or *ADSP-2106x SHARC User's Manual*.

When an interrupt (that is not masked) occurs on the ADSP-21xxx, the processor vectors to a specific address based on the interrupt that occurred. The runtime header (`060_hdr.asm` or `020_hdr.asm`) contains the vectors for each interrupt.

The code in the vector location executes a JUMP to the interrupt dispatcher that is part of the runtime library. The cache update is disabled and the global interrupt enable bit is turned off.

The cache is frozen because otherwise the ISR fills the cache. This is inefficient, since most ISRs are short and do not contain loops. By disabling the cache, it retains its contents. So, it still contains useful information when it returns to the code that was interrupted.

The global interrupt enable bit is shut off temporarily. This is done so that the dispatcher is able to maintain the C runtime stack. There are approximately 9 cycles where interrupts are completely disabled. As soon as the dispatcher performs stack management, interrupts are reenabled.

The interrupt dispatcher stores all scratch registers on the C runtime stack before calling the ISR. The dispatcher needs to save the scratch registers because C routines expect that they can overwrite all scratch registers without saving them. Since the interrupt occurred at an unknown point in program execution, it is likely that these registers have meaningful values in them.

The regular dispatcher then sets up what will happen on the next interrupt, determines the current interrupt, and calls the ISR. This process requires about 100 cycles.

After the ISR returns, the dispatcher restores the saved registers and executes an RTI to return to the interrupted code. This requires about 50 cycles to complete.

In addition to the processing overhead, the regular dispatcher requires a maximum 30 locations on the C runtime stack to store registers.

The interrupt dispatcher may be modified in future releases. Refer to your release note to see if the timing is altered.

C.12 INTERRUPT RESTRICTIONS

There are a few extraordinary situations that might cause problems when using interrupts. When an interrupt occurs and the chip transfers control to the dispatcher, the return address is pushed on the PC stack. The PC stack is a hardware stack on the processor, which is limited to 20 locations.

The C runtime model does not use the PC stack very much, so in most cases it is possible to nest interrupts 20 deep. It is unusual for an application to require this many simultaneous interrupts, but it is possible that the PC stack could overflow in such a case. If your application only uses a few interrupts it is extremely unlikely that PC stack limitations will ever be encountered.

Note: Some fast emulation routines are called using the PC stack.

The interrupt dispatcher is designed to support the C runtime model. If your program has assembly routines that change the mode of the processor, it is possible that this could cause trouble with interrupts. If for example, you code enables the bit reverse mode of the chip, and an interrupt occurs while bit reverse is on, and the ISR uses DAG0, this could cause problems. (The dispatcher does not use DAG0, so you could disable bit reversal in the ISR and re-enable it before exiting).

C Interrupts