

Today, many embedded systems require interrupt service routines (ISRs) to process external hardware interrupts in a timely manner. The CPU runs the ISR after it is interrupted. The Nios® development kits include software subroutines that make it easy to implement ISRs in your Nios system. These routines allow you to spend your time writing the ISR instead of working on the overhead necessary to set up the ISR. For example, one line of C code installs the ISR and saves and restores the context when the interrupt occurs.

This document describes the basics of interrupt handling in Nios systems using the Altera-provided system software subroutines. The document uses an example ISR for an interrupt-driven UART to demonstrate these software subroutines. “Code Examples” on page 14 contains the source code. You can also download the code from the literature page on the Altera web site at www.altera.com/literature/lit-nio.html.

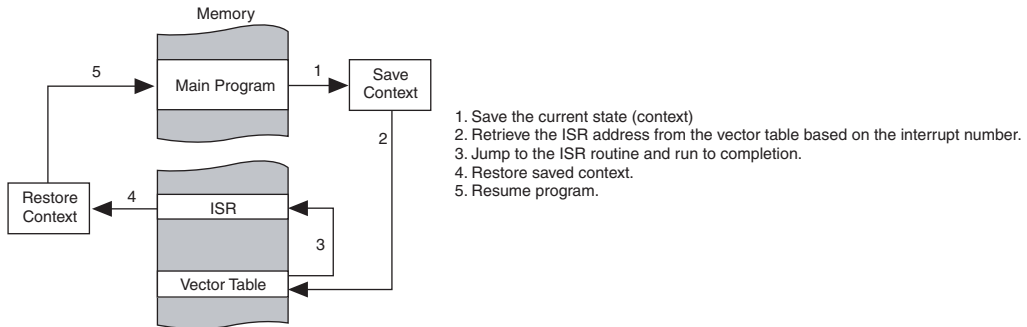


For more detailed information about the Nios instruction set or software development kit (SDK) utilities, refer to the following documents:

- *Nios Embedded Processor 32-Bit Programmer's Reference Manual*
- *Nios Embedded Processor 16-Bit Programmer's Reference Manual*
- *Nios Software Development Reference Manual*

Basics of Exception Handling

Exceptions interrupt the processor: they stop the current operation to take care of something more important. Different sources can cause exceptions, including external hardware, direct software interrupts, and internal exceptions during normal execution of code. Figure 1 illustrates a simple case of what happens in most embedded systems when an exception occurs.

Figure 1. Exception Handling Process

Before an interrupt occurs, the program executes normally from the main program memory. The vector table contains the first address for all of the ISRs. Typically, the vector table is a sequential list of the ISR addresses where the number of the interrupt is the index into the list (i.e., the vector). When an interrupt occurs:

1. The current state, or context, of the system is saved.
2. The address of the ISR corresponding to the interrupt is loaded from the vector table.
3. The CPU jumps to the ISR address and runs the ISR.
4. When the ISR completes, the context is restored and the CPU continues running the original code as if nothing happened.

Many systems also support nested interrupts, which allow higher priority interrupts to interrupt lower priority interrupts. The same process occurs when the current ISR is running and it is interrupted by the higher priority interrupt. Before it can finish, the first ISR must wait until the higher priority ISR finishes.

Nios Processor Interrupt Handling Basics

This section describes how the Nios processor handles interrupts, including:

- Exception Handling
- External Hardware Interrupt Sources
- Internal Exception Sources
- Generating Internal Exceptions
- Direct Software Exceptions (TRAP Instructions)
- Exception Vector Table

Exception Handling

The Nios processor allows up to 64 vectored exceptions. Exceptions can be enabled or disabled globally by the IE control-bit in the STATUS register, or selectively enabled on a priority basis by the IPRI field in the STATUS register. Exceptions can be generated from any of three sources: external hardware interrupts, internal exceptions, or explicit software TRAP instructions.

Exception-handling subroutines always execute in a newly opened register window, allowing very low interrupt latency. The exception handler does not need to preserve the interruptee's register contents manually.

External Hardware Interrupt Sources

An external source can request a hardware interrupt by driving an encoded 6-bit interrupt number on the Nios CPU `irq_number` inputs while simultaneously asserting (i.e., a logic 1) the Nios CPU `irq` input pin. The Nios CPU processes the indicated exception if the IE bit is true (i.e., 1) and the requested interrupt number is smaller (higher priority) than the current value in the IPRI field of the STATUS register. Control is transferred to the exception handler whose number is given by the `irq_number` inputs.



External sources that generate interrupts should assert their `irq` output signals until the interrupt is acknowledged by software (e.g., by writing a register inside the interrupting peripheral to 0). Interrupts that are asserted and then de-asserted before the Nios CPU core can begin processing the exception are ignored.



For more information about external exceptions and the Avalon™ bus, refer to the *Nios Embedded Processor 16-Bit Programmer's Reference Manual* or the *Nios Embedded Processor 32-Bit Programmer's Reference Manual*.

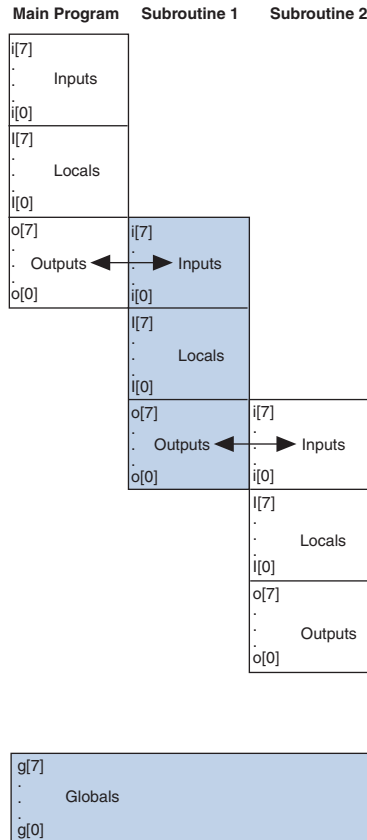
Internal Exception Sources

There are two sources of internal exceptions: register window overflows and register window underflows. To understand these exceptions, you must first understand the Nios windowed register architecture.

You can configure the Nios processor to contain 128, 256, or 512 general-purpose registers. Of these, 32 are visible to the software at any particular moment. The registers are named `%r0` - `%r31`. You can also refer to them as `%g0` - `%g7` (global), `%o0` - `%o7` (out), `%l0` - `%l7` (local), and `%i0` - `%i7` (in).

The current window pointer (CWP) bits of the Nios STATUS register (%ctl0, which is readable via the RDCTL instruction) determines which 32 registers are visible. Figure 2 shows the windowed register file.

Figure 2. Windowed Register File



Subroutines execute a SAVE instruction, which decrements the CWP by one, revealing 16 “new” registers (8 local and 8 out). Therefore, a 512 register file system has 32 register windows available. The caller’s %o registers are visible to the callee as %i registers. Subroutines execute a RESTORE instruction, which increments the CWP by one, when they are ready to return to the caller’s register window.



Refer to the *Nios 16-Bit Programmer’s Reference Manual* or *Nios 32-Bit Programmer’s Reference Manual* for details on the topics described in this section.

Generating Internal Exceptions

Every time the processor executes a SAVE or RESTORE instruction, it compares the value of the CWP to the WVALID register (%ctl2). If the new CWP value would be greater than HI_LIMIT or less than LO_LIMIT, the processor generates an internal exception. If $CWP > HI_LIMIT$, the processor generates exception 2, which is a register window overflow. If $CWP < LO_LIMIT$, the processor generates exception 1, which is a register window underflow.



Writing directly to CWP does not generate a register window overflow or underflow exception.

The Nios SDK includes, and automatically installs by default, register window overflow and underflow handlers that virtualize the register file using the stack as temporary storage. By default, every Nios system uses the CWP manager.



Altera recommends that you use the CWP manager unless you know that your system will not use more register windows than are available.

A new window is opened for every subroutine call, hardware interrupt, internal exception, and direct software exception (TRAP instruction). A 512-register file system has 32 register windows available, a 256-register file system has 16 register windows, and a 128-register file system has 8 register windows.

You do not have to use the CWP manager. If you do not want to use the CWP manager, turn off the CWP Manager option in the Nios CPU wizard in SOPC Builder. If this option is turned on (default), the SOPC Builder software installs the CWP manager's underflow and overflow handlers.

The MFLAT option makes the Nios register file appear flat, or limited to a fixed 32 registers. Using MFLAT, the average context-switching time may increase, because register contents are saved to memory during each interrupt. However, the worst-case time is predictable (i.e., the time required to save all registers to memory), and significantly less than the worst-case time for the windowed register file (i.e., overflow or underflow condition). You enable this option using the `-mflat` switch.



For more information on the MFLAT compiler option, see the **readme.txt** file in the `<project>/<cpu name>_sdk/src/mflat` directory.

Direct Software Exceptions (TRAP Instructions)

By issuing a TRAP instruction, software can directly request that control be transferred to an exception handler. The IMM6 field of the instruction gives the exception number. TRAP instructions are always processed, regardless of the IE or IPRI bit settings. TRAP instructions do not have a delay slot. Therefore, the instruction immediately following a TRAP is not executed before control is transferred to the indicated exception-handler. The TRAP instruction disables interrupts (IE = 0) before passing control to the exception handler. A reference to the instruction following TRAP is saved in %o7, therefore, a TRET instruction transfers control back to the instruction following TRAP at the conclusion of exception processing.

Exception Vector Table

The exception vector table is a set of 64 exception-handler addresses and each entry is 4 bytes for a 32-bit Nios processor and 2 bytes for 16-bit Nios processor. The base-address (VECBASE) of the exception vector table is configurable. The IRQ number of the interrupt determines its priority and its location in the exception vector table. The highest priority is 0 and the lowest priority is 63. For example, interrupt 25 is located at address $\text{VECBASE} + (25 \times 4)$. This interrupt has priority over interrupts 26 through 63. When the Nios CPU processes exception number n , it fetches the n th entry from the exception vector table, doubles the fetched value, and loads the result into the Program Counter (PC).

Exceptions 0 - 15 are reserved for the following internal exceptions:

- The `_start` subroutine uses exception 0.
- Exception 1 is the register window underflow exception.
- Exception 2 is the register window overflow exception.
- The GNUPro debugger uses exceptions 3 - 5.
- Exceptions 6 - 15 are reserved for future use.

The exception vector table can physically reside in RAM or ROM, depending on the hardware memory map of the target system. A ROM exception vector table does not require run-time initialization. For each Nios CPU in your system, you can specify the location of the exception vector table in SOPC Builder in the **More <CPU name> Settings** tab.



For more details, refer to the *SOPC Builder Data Sheet*.

Software Subroutines

The Nios SDK contains software subroutines that install user-created ISRs and manage the windowed register file. These subroutines are appropriate for interrupts that do not need to be executed quickly because they are easy to use and simplify the software code. The subroutines save and restore the context so all you have to do is write the ISR. Altera recommends using these subroutines for interrupts that do not have a low latency requirement. [“Implementing an ISR in a Nios system” on page 8](#) discusses the latency for interrupts when using these subroutines.

To install an ISR call the `nr_installuserisr2()` or `nr_installuserisr()` subroutine and pass it three variables: the interrupt number, the address of the ISR, and a context variable. The optional context variable parameter can be any value. Your ISR code must clear the source of the interrupt. Interrupts are enabled upon entry into the ISR.

If you do not want to use these subroutines and instead want to manipulate the vector table directly, you must completely understand the mechanisms of the Nios register window, control registers, etc. so that interrupt requests execute and return properly.



You can use either `nr_installuserisr2()` or `nr_installuserisr()`. `nr_installuserisr2()` is useful if you want to use the same ISR for multiple interrupt sources. The differences are described in the following sections.

nr_installuserisr

This subroutine installs a user ISR for a specific interrupt number. If you use `nr_installuserisr()` to set up the interrupt vector table, you can specify standard compiled C functions as interrupt service subroutines. You do not need to know the low-level details of the Nios interrupt vector table to use `nr_installuserisr()`. This function is declared in the include file **excalibur.h**.

When called, the user ISR receives only the context value as an argument. Interrupts are enabled before control is passed to the ISR. The ISR must clear any interrupt condition for a peripheral it services.

Syntax

```
void nr_installuserisr(int trapNumber,
                      void *nios_isrhandlerproc, int context);
```

nr_installuserisr2

This subroutine is similar to `nr_installuserisr()` except when the user ISR is called, the interrupt number, interrupted PC, and context are passed to the user interrupt handler. This function is declared in **excalibur.h**.

Syntax

```
void nr_installuserisr2(int trapNumber,  
    void *nios_isrhandlerproc2, int context);
```

Enabling & Disabling Interrupts

The built-in ISR handler automatically enables interrupts before control is passed to the ISR. If you want to disable interrupts, insert the following Assembly code into your ISR:

```
//Disable Interrupts  
asm("PFX 8");  
asm("WRCTL %g0");
```

If you disabled interrupts and want to enable them, insert the following Assembly code into your ISR:

```
//Enable Interrupts  
asm("PFX 9");  
asm("WRCTL %g0");
```

Implementing an ISR in a Nios system

This section explores how to generate a variety of serial port subroutines that should fit most Nios applications. The principles also apply to all types of ISRs. A UART example demonstrates how to implement an ISR in a Nios system using `nr_installuserisr2()` (or `nr_installuserisr()`).

Serial Communication Basics

This section describes the basic concepts of serial communications and covers the following topics:

- General UART receiver and transmitter
- Relating the general receiver and transmitter concepts to the Nios UART
- Implementing C code for the ISR and supporting functions
- Discussing the latency of ISRs when using `nr_installuserisr2()` or `nr_installuserisr()`



For more information on the functionality of the Nios UART, refer to the *Nios UART Data Sheet*.

Receiver

For low baud rates, polling the serial port may be adequate. As the baud rate increases however, it becomes harder and harder to insure that an incoming byte is not missed, particularly for UARTs that do not have a FIFO buffer. To overcome this problem, UARTs commonly generate interrupts, which cause the processor attached to the UART to stop operation and service the interrupt.

In response to the interrupt, the processor runs an ISR to retrieve the byte from the UART and stores it in a receiver buffer until the normal software subroutines can handle it. This buffer is typically a circular buffer. If the receiver buffer is full, RTS (used in hardware handshaking) can be lowered to inform the sender that no more characters can be processed. If hardware handshaking signals are not available, the buffer overflows and data is lost.

The size of this buffer should depend on the baud rate and how often and how efficiently the processor can remove bytes from the receiver buffer. This buffer can be thought of as an array of characters, for example,

```
unsigned char RxBuf[buffer_size]
```

A circular buffer consists of a block of memory and has two indices, which are called head and tail in this discussion. The head is the index into the buffer array where the next received byte is stored. The tail points to the next location to be read from the buffer by subroutines like `getc()`. When the two indices are equal, the buffer is empty and no more bytes are available. When a byte arrives the UART interrupts the processor, which then runs the ISR.

The ISR typically performs the following tasks:

- Moves the byte from the UART rxdata register and places it in the circular receiver buffer
- Clears the interrupt
- Increments the RxHead index
- Checks for overflow of the RxHead index
- Checks to see if the FIFO buffer is almost full and de-asserts the RTS signal (optional)
- Performs error handling (optional)

Transmitter

In many applications, even with higher baud rates, it is not necessary to have an ISR handle the outgoing bytes. For non-blocking code, or code that does not wait for all bytes to be transferred before continuing normal execution, transmit data is handled by `putc()` adding bytes into the circular transmitter buffer and incrementing the TxHead index. Then a small service routine, which is called frequently from the main loop, checks the status of the UART transmitter. If it is ready for a new byte and there is one in the transmitter buffer, it is sent. Then, the TxTail index is incremented.

This polling method works well unless you need the maximum output from the UART and it does not have a FIFO buffer. For blocking code, or code that does not continue normal execution until all the bytes have been transferred, a transmit buffer is not needed. This discussion assumes non-blocking code styles.

If you need the maximum output, the receiver ISR must be enhanced to handle the transmitter side. The ISR performs the following tasks:

- Determines if the interrupt was generated by the UART receiver or transmitter
- If the interrupt was generated by the receiver, it performs the tasks described in [“Receiver” on page 9](#)

If the interrupt was generated by the transmitter, it performs the following tasks:

- Checks the transmit buffer for bytes that need to be sent
- If hardware handshaking is supported, it checks the CTS pin to verify that data can be sent (optional)
- Moves the byte from the transmit buffer into the txdata register and increments the TxTail index
- Checks for TxTail index overflow
- Disables the interrupt if there is no data in the transmit buffer

You must also modify the `putc()` routine to enable the transmitter interrupt when a byte is placed into the transmit buffer.

Implementing the Nios UART

The UART peripheral provided with the Nios core does not have a built-in FIFO on the transmitter or receiver. Therefore, you may need to create an ISR for handling serial port communications. You can implement these routines using the `nr_installuserisr2()` or `nr_installuserisr()` subroutine provided in the Nios SDK. Because `nr_installuserisr2()` and `nr_installuserisr()` are generic ISR front ends, they save all of the registers before calling your ISR.

The Nios UART has six registers: `rxdata`, `txdata`, `status`, `control`, `divisor`, and `endofpacket`. The following sections give a simplified view of these registers and ignore all errors. Refer to the *Nios UART Data Sheet* for detailed information.

Receiver

A byte is in the `rxdata` register and ready to read when the `RRDY` bit is set (`rrdy = 1`) in the UART status register. The `RRDY` is cleared (`rrdy = 0`) when the `rxdata` register is read. An interrupt is generated if the `RRDY` bit in the status register is set and `iRRDY` in the control register is set.

Transmitter

A byte can be placed in the `txdata` register whenever the `TRDY` bit is set (`trdy = 1`) in the UART status register. As soon as a byte is placed in the `txdata` register, the `TRDY` bit is cleared (`trdy = 0`) until the data is moved to the output shift register. At that point, `TRDY` is re-asserted (`trdy = 1`). An interrupt is generated if the `TRDY` bit in the status register is set and the `iTRDY` bit in the control register is set.

CTS/RTS

You can configure the Nios UART to implement clear to send (CTS) and ready to send (RTS) pins and capture logic. These pins and logic allow your software code to write to the RTS output pin and read from the CTS input pin. However, it does not change the UART transmit or receive logic.

Interrupt Outputs

The Nios UART has one interrupt output pin. When appropriate, each bit in the status register has a corresponding bit in the control register. An interrupt is generated when a bit in the status register is set and its corresponding interrupt enable bit is set in the control register.

Implementing the Nios UART ISR

The following sections describe how to create ISRs for Nios systems and provide example code that you can run on the Nios development board. You can use the example code in a Nios design that has a UART named **uart1**, such as the Nios reference designs. You can connect to the Nios development board using a bash shell in terminal mode (type `nios-run -t` at the bash shell prompt).



If you are not familiar with using the bash shell to communicate with the Nios development board, refer to the *Nios Embedded Processor Hardware Tutorial* for information.

Simple Receiver ISR

Figure 3 on page 15 shows sample code for handling an ISR for the UART receiver. The main program prints the character to verify that it was received correctly. This example assumes that the only interrupt coming from the serial port is for handling received data.

The code in Figure 3 on page 15 is small but efficient. Because only the receiver ready interrupt (RRDY) is assumed to be active, there must be data in the `np_uartrxdata` register. Therefore, the UART moves the data in `np_uartrxdata` to the receiver buffer called `RxBuf[]`. The UART throws away errors by writing any value to the `np_uartstatus` register. The `RxHead` index is then incremented and checked to see if it is larger than the buffer (`RXBUFSIZE-1`). If so, it is reset to 0.

Figure 3 on page 15 implements one of the simplest receiver ISRs you can make. To read data from the receive buffer, use a subroutine such as `_getchar()` or `int getcharSerial()`. The function `getcharSerial()` is the non-blocking version of the `get` routine and `_getchar()` is blocking, so it continues to wait until a character arrives.

If your design has more than one UART, you should use unique names for each `_getchar` and `simpleReceiver` (ISR). This naming keeps the code fast but uses up a bit more code space. Otherwise, you would need to add extra code in the ISR to determine which UART to handle.

Simple Transmitter ISR

Interrupt-driven transmitter ISRs have some of the same features as receivers except the interrupt must be disabled by the ISR when there are no more bytes waiting to be sent. The transmitter is enabled by the `_putchar()` subroutine, which writes the byte to the transmit buffer. This action is necessary because when the `TxBuf` is empty, the `TRDY` interrupt must be disabled or the software calls the ISR forever because `TRDY` is high. The example code in [Figure 4 on page 16](#) prints a string, `Nios`, to verify that the transmitter ISR is working.

Combining Receiver & Transmitter ISRs

In many systems, the transmitter and receiver ISRs are combined into one ISR because there is only one interrupt line per serial port. The Nios UART has only one line per serial port, so you should combine the ISRs.



You must check to see if the transmitter interrupt is enabled, by reading `ITRDY` (interrupt enable for `TRDY`), before handling any transmitter operations. Simply reading `trdy` does not work because `trdy` is high whenever the transmitter is ready for data even if there is no data to be sent.

[Figure 5 on page 17](#) shows a combined ISR.

Hardware Handshaking

The Nios processor outputs `RTS` to indicate that it is ready for the other device to send characters. `CTS` is an input to the Nios processor indicating that the other device is ready for it to send characters. If `RTS` and `CTS` are asserted, the device can send; if they are deasserted, the device should not send.

The ISR receiver enables `RTS` whenever the buffer is below a predefined limit (`RXBUFSIZE - 5` for the `_getchar()` routine shown in [Figure 7](#)). When the buffer exceeds this predefined limit, the ISR must deassert `RTS` so that no more data is transmitted.

The ISR transmitter ensures that `CTS` is asserted before sending any data. The transmitter ISR runs once for every character no matter how full the transmit buffer is. Every time the ISR runs there is overhead sent preparing to run the ISR and then going back to the main program (i.e., to the address before the interrupt occurred).

To optimize the transmitter ISR code to empty the transmit buffer before exiting the ISR, you can monitor the CTS line. For example, instead of using IF statements, implement a loop that checks the status of CTS before processing each character until the transmit buffer is empty.

Figure 6 on page 18 shows the hardware-handshaking ISR source code. Figures 7 through 9 show the `_getchar()` routine and other related routines for hardware handshaking.

Execution Time

Altera ran the UART ISR with handshaking support on the standard 32-bit reference design provided with the Nios Development Kit version 2.11. The CPU clock ran at 33 MHz and the UART baud rate was 115,200 bps. Table 1 shows the latency for entering and exiting the ISR.

Table 1. UART ISR Latency for 33-MHz Clock		
Item	Time (μs)	CPU Cycles
ISR entry latency	2.79	93
Running the ISR	3.21	107
ISR exit latency	1.92	64
Total	7.92	264

The data provided in Table 1 was determined by simulating the design in the ModelSim software and referring to the objdump file. The ISR takes 264 CPU cycles to complete. A baud rate of 115,200 bps is about 10,472 characters per second, assuming 11 bits per character with 8 data bits, 1 start bit, 2 stop bits and no parity bit. Therefore, when the UART runs at the full baud rate of 115,200, the UART ISR requires roughly 2.76 million cycles per second, which is about 8.4% of the 33-MHz CPU.



For more information on simulating Nios designs, refer to *AN 189: Simulating Nios Embedded Processor Designs*.

Conclusion

Many embedded systems need interrupts to handle certain high-priority processes in a timely manner. It is easy to implement ISRs in Nios systems using the software subroutines provided with the Nios development kits. These routines are generic, so they will work for any system. In many situations, the latency for entering and exiting the ISR using the provided subroutines is acceptable. For situations in which latency is critical, more complexity is required, which is beyond the scope of this document.

Code Examples

Figures 3 through 9 provide example code that is referenced throughout the document.

Figure 3. Code Handling a Simple Receiver ISR

```

#include "buffer.h" // defines RXBUFSIZE and TXBUFSIZE
#include "excalibur.h"

//Global variables
unsigned char RxBuf[RXBUFSIZE]; // the receiver buffer.
int RxHead = 0; // the circular buffer index
int RxTail = 0;
unsigned char TxBuf[TXBUFSIZE]; // the transmit buffer.
int TxHead = 0; // the circular buffer index
int TxTail = 0;

void simpleReceiver1(int context);
unsigned char _getchar();

int main(void)
{
    int context = 0;
    unsigned char rxchar;

    nr_installuserisr(na_uart1_irq, simpleReceiver1, context); // install UART ISR (receiver)
    na_uart1->np_uartcontrol = np_uartcontrol_irrdy_mask; // enable rrdy interrupt

    while(1)
    {
        rxchar = _getchar();
        printf("\nfrom RxBuf: %c \n",rxchar);
    }

    return 0;
}

// pass an int to make it compatible with nr_installuserisr routine
// contents of third argument ("context") in nr_installuserisr is passed to ISR
void simpleReceiver1(int context)
{
    // put new char into buffer from UART
    RxBuf[RxHead] = na_uart1->np_uartrxdata;
    // clear the errors and interrupts
    na_uart1->np_uartstatus=0;

    if ((++RxHead) > (RXBUFSIZE-1))
    {
        RxHead = 0;
    }
}

```

Figure 4. Code Handling a Simple Transmitter ISR

```
#include "excalibur.h"
#include "buffer.h" // defines RXBUFSIZE and TXBUFSIZE

//Global variables
unsigned char RxBuf[RXBUFSIZE]; // the receiver buffer.
int RxHead = 0; // the circular buffer index
int RxTail = 0;
unsigned char TxBuf[TXBUFSIZE]; // the transmit buffer.
int TxHead = 0; // the circular buffer index
int TxTail = 0;

unsigned char _getchar();
void simple_transmitter1(int context);
int _putchar(int in_char);

int main(void)
{
    int context = 0;
    int i = 0;

    // install UART ISR (transmitter)
    nr_installuserisr(na_uart1_irq, simple_transmitter1, context);

    _putchar('N');
    _putchar('i');
    _putchar('o');
    _putchar('s');
    _putchar('\n');

    nr_delay(1);
    return 0;
}

/*****

void simple_transmitter1(int context)
{
    if (TxTail != TxHead)
    {
        na_uart1->np_uarttxdata = TxBuf[TxTail]; // send a byte of data
        if (++TxTail > (TXBUFSIZE - 1)) // check for wrap around.
            TxTail = 0;
    }
    else
    {
        /* tx buffer empty, disable iTRDY. Since this ISR only deals with iTRDY
        and iRRDY, we will accomplish this by enabling just iRRDY. */
        na_uart1->np_uartcontrol = np_uartcontrol_irrdy_mask;
    }
}
// end of transmitter
```


Figure 5. Code Handling a Combined Transmitter & Receiver ISR

```

#include "excalibur.h"
#include "buffer.h" // defines RXBUFSIZE and TXBUFSIZE

//Global variables
unsigned char RxBuf[RXBUFSIZE]; // the receiver buffer.
int RxHead = 0; // the circular buffer index
int RxTail = 0;
unsigned char TxBuf[TXBUFSIZE]; // the transmit buffer.
int TxHead = 0; // the circular buffer index
int TxTail = 0;

unsigned char _getchar();
void serial_simple_ISR(int context);
int _putchar(int in_char);

int main(void)
{
    int context = 0;

    nr_installuserisr(na_uart1_irq, serial_simple_ISR, context); // install UART ISR
    na_uart1->np_uartcontrol = np_uartcontrol_irrdy_mask; // enable rx interrupts

    while(1)
    {
        _putchar( (int)(_getchar()) );
    }

    return 0;
}

/*****
void serial_simple_ISR(int data)
{
    // need to retrieve the status register
    int sr = na_uart1 ->np_uartstatus;

    if(sr & np_uartstatus_rrdy_mask)
    { // something ready for the receiver.
        // put new char into buffer from UART
        RxBuf[RxHead] = na_uart1->np_uartrxdata;
        // clear the errors and interrupts
        na_uart1->np_uartstatus=0;
        if ((++RxHead) > (RXBUFSIZE-1))
        {
            RxHead = 0;
        }
    } // end of receiver
}
*****/

```

Code Continued on Next Page...

```
if(sr & np_uartstatus_trdy_mask)
{ // transmitter ready
    if(np_uart1->np_uartcontrol & np_uartcontrol_itrdy_mask)
    { // ready to handle transmitter interrupts
        if (TxTail != TxHead)
        {
            na_uart1->np_uarttxdata = TxBuf[TxTail]; // send a byte of data
            if (++TxTail > (TXBUFSIZE -1))// check for wrap around.
                TxTail = 0;
        }
        else
        { /* tx buffer empty, disable iTRDY. Since this ISR only deals with iTRDY
            and iRRDY, we will accomplish this by enabling just iRRDY. */
            na_uart1->np_uartcontrol = np_uartcontrol_irrdy_mask;
        }
    }
} // end of transmitter
} // end of ISR
```

Figure 6. Hardware-Handshaking ISR Source Code

```
// Includes ISR for UART with handshaking support using CTS/RTS

#include "excalibur.h"
#include "buffer.h" // defines RXBUFSIZE and TXBUFSIZE

// global definitions.
unsigned char RxBuf[RXBUFSIZE]; // receive buffer
int RxHead = 0; // circular buffer index
int RxTail = 0; // circular buffer index
unsigned char TxBuf[TXBUFSIZE]; // transmit buffer
int TxHead = 0; // circular buffer index
int TxTail = 0; // circular buffer index

unsigned char _getchar();
void uart_isr(int context);
int _putchar(int in_char);

int main(void)
{
    int context = 0;

    nr_installuserisr(na_uart1_irq, uart_isr, context); // install UART ISR
    na_uart1->np_uartcontrol = np_uartcontrol_irrdy_mask; // enable rx interrupts

    // Continually wait for chars and transmit them as they come in.
    while(1)
    {
        _putchar( (int)(_getchar()) );
    }

    return 0;
}
```

Code Continued on Next Page...

```

// UART ISR with handshaking support using CTS/RTS
void uart_isr(int context)
{
    int size;
    // retrieve the status register
    int sr = na_uart1->np_uartstatus;

    // start of receiver
    if(na_uart1->np_uartstatus & np_uartstatus_rrdy_mask)
    { // something ready for the receiver.
        // put new char into buffer from UART
        RxBuf[RxHead] = na_uart1->np_uartrxdata;
        // clear the errors and interrupts
        na_uart1->np_uartstatus=0;

        if ((++RxHead) > (RXBUFSIZE - 1))
        {
            RxHead = 0;
        }

        // check to see if need to turn RTS off
        if (RxHead >= RxTail)
        {
            size = RxHead - RxTail;
        }
        else
        {
            size = ((RXBUFSIZE) - RxTail) + RxHead;
        }
        if (size > RXBUFSIZE - 5) /* don't overflow RxBuf[] */
        {
            // turn off the RTS bit
            na_uart1->np_uartcontrol = na_uart1->np_uartcontrol & ~np_uartcontrol_rts_mask;
        }
    } // end of receiver

    // start of transmitter
    if(sr & np_uartstatus_trdy_mask)
    { // transmitter ready
        if(na_uart1->np_uartcontrol & np_uartcontrol_itrdy_mask)
        { // set to handle transmitter interrupts
            // check CTS too
            if ((TxTail != TxHead) && (na_uart1->np_uartstatus & np_uartstatus_cts_mask))
            {
                na_uart1->np_uarttxdata = TxBuf[TxTail]; // send a byte of data
                if (++TxTail > (TXBUFSIZE - 1)) // check for wrap around.
                    TxTail = 0;
            }
            else
            { // buffer empty, disable TRDY interrupts.
                // Remember, this ISR handles only RRDY and TRDY (no error handling).
                na_uart1->np_uartcontrol = np_uartcontrol_irrdy_mask ;
            }
        }
    } // end of transmitter
} // end of ISR

```

Figure 7. *_getchar()* Subroutine

```
/******  
//  
//      _getchar() - Read char from Input Stream (Receive  
//                  Buffer).  
//  
//      This is a blocking routine so it will wait forever  
//      until a char comes in  
//  
/******  
  
#include "buffer.h" // defines RXBUFSIZE  
  
//Global variables defined in main program  
extern unsigned char RxBuf[RXBUFSIZE]; // the receiver buffer.  
extern int RxHead; // the circular buffer index  
extern int RxTail;  
  
unsigned char _getchar()  
{  
    unsigned char temp;  
  
    while (RxTail == RxHead)  
    {  
        nr_delay(1);  
    }  
    temp = RxBuf[RxTail];  
  
    if (++RxTail > (RXBUFSIZE -1))  
    {  
        RxTail = 0;  
    }  
  
    return(temp);  
}
```

Figure 8. *_getcharSerial()* Subroutine

```
/******  
//  
//    getcharSerial() - Read char from RxBuf  
//  
//    If no char to be read then returns a -1  
//  
/******  
  
#include "buffer.h" // defines RXBUFSIZE  
  
//Globals variables defined in main program  
extern unsigned char RxBuf[RXBUFSIZE]; // the receiver buffer.  
extern int RxHead; // the circular buffer index  
extern int RxTail;  
  
int getcharSerial()  
{  
    int temp = -1;  
  
    if (RxTail != RxHead)  
    {  
        temp = RxBuf[RxTail];  
        if (++RxTail > (RXBUFSIZE -1))  
        {  
            RxTail = 0;  
        }  
    }  
  
    return(temp);  
}
```

Figure 9. `_putchar()` Subroutine

```
/******  
// _putchar(int in_char)  
//   Takes a char and puts it in the Transmit buffer.  
//   In the simple case it can actually stuff a  
//   char into the UART.  
//   After putting a char into the buffer it:  
//       Increments the TxHead index and  
//       Watches for wrap around. If the buffer  
//       gets too full the chars are thrown away.  
//  
/******  
  
#include "excalibur.h"  
#include "buffer.h" // defines TXBUFSIZE  
  
//Globals variables  
extern unsigned char TxBuf[TXBUFSIZE]; // the transmit buffer.  
extern int TxHead; // the circular buffer index  
extern int TxTail;  
  
int _putchar(int in_char)  
{  
    int size;  
    int sr = na_uart1->np_uartstatus; // get the status register  
  
    if ((TxHead == TxTail) && (sr & np_uartstatus_trdy_mask))  
    {  
        na_uart1->np_uarttxdata = in_char; // send directly  
    }  
    else  
    {  
        if (TxHead >= TxTail) // Find out how full the TX buffer is  
            size = TxHead - TxTail;  
        else  
            size = ((TXBUFSIZE-1) - TxTail) + TxHead;  
  
        if (size > (TXBUFSIZE - 3)) // don't overflow TxBuf[]  
            return (-1); // just throw the char away  
  
        TxBuf[TxHead] = in_char;  
        if (++TxHead > (TXBUFSIZE-1))  
            TxHead = 0;  
  
        // enable trdy interrupt  
        na_uart1->np_uartcontrol = na_uart1->np_uartcontrol | np_uartcontrol_itrdy_mask;  
    }  
  
    return(1);  
}
```



101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
www.altera.com
[Applications Hotline:](#)
(800) 800-EPLD
[Literature Services:](#)
lit_req@altera.com

Copyright © 2003 Altera Corporation. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services. All rights reserved.



I.S. EN ISO 9001