8. Exception Handling

NII52006-10.0.0

Introduction

This chapter discusses how to write programs to handle exceptions in the Nios® II processor architecture. Emphasis is placed on how to process hardware interrupt requests by registering a user-defined interrupt service routine (ISR) with the hardware abstraction layer (HAL). This information applies to software projects created with the Nios II Software Build Tools (SBT), either in Eclipse™ or on the command line.

This chapter contains the following sections:

For low-level details about handling exceptions and hardware interrupts on the Nios II architecture, refer to the *Programming Model* chapter of the *Nios II Processor Reference Handbook*.

Nios II Exception Handling Overview

The Nios II processor provides the following exception types:

- Hardware interrupts
- Software exceptions, which fall into the following categories:
  - Unimplemented instructions
  - Software traps
  - Miscellaneous exceptions

The Nios II processor offers two distinct approaches to handling hardware interrupts:

- The internal interrupt controller (IIC)
- The external interrupt controller (EIC) interface

The interrupt controllers are discussed in detail in "Interrupt Controllers" on page 8–3.

Exception Handling Terminology

The following list of HAL terms outlines basic exception handling concepts:

- Application context—The status of the Nios II processor and the HAL during normal program execution, outside of exception funnels and handlers.
- Context switch—The process of saving the Nios II processor's registers on a software exception or hardware interrupt, and restoring them on return from the exception handling routine or ISR.
- Exception—A transfer of control away from a program's normal flow of execution, caused by an event, either internal or external to the processor, which requires immediate attention. Exceptions include software exceptions and hardware interrupts.
- Exception context—The status of the Nios II processor and the HAL after a software exception or hardware interrupt, when funnel code, a software exception handler, or an ISR is executing.
- Exception handling system—The complete system of software routines that service all exceptions, including hardware interrupts, and pass control to software exception handlers and ISRs as necessary.
- Exception (or interrupt) latency—The time elapsed between the event that causes the exception (such as an unimplemented instruction or interrupt request) and the execution of the first instruction at the exception (or interrupt vector) address.
- Exception (or interrupt) response time—The time elapsed between the event that causes the exception and the execution of the handler.
- Exception overhead—Additional processing required to service a software exception or hardware interrupt, including HAL-specific processing and RTOS-specific processing if applicable.
- Funnel code—HAL-provided code that sets up the correct processor environment for an exception-specific handler, such as an ISR.
- Handler—Code specific to the exception type. The handler code is distinct from the funnel code, which takes care of general exception overhead tasks.
- Hardware interrupt—An exception caused by an explicit hardware request signal from an external device. A hardware interrupt diverts the processor's execution flow to a interrupt service routine, to ensure that a hardware condition is handled in a timely manner.
- Implementation-dependent instruction—A Nios II processor instruction that is not supported on all implementations of the Nios II core. For example, the `mul` and `div` instructions are implementation-dependent, because they are not supported on the Nios II/e core.
- Interrupt—Hardware interrupt.
- Interrupt controller—Hardware enabling the Nios II processor to respond to an interrupt by transferring control to an interrupt service routine.
- Interrupt request (IRQ)—Hardware interrupt.
- Interrupt service routine (ISR)—A software routine that handles an individual hardware interrupt.

- Invalid instruction—An instruction that is not defined for any implementation of the Nios II processor.
- Maskable exceptions—Exceptions that can be disabled with the `status.PIE` flag, including internal hardware interrupts, maskable external hardware interrupts, and software exceptions, but not including nonmaskable external interrupts.
- Maximum disabled time—The maximum amount of continuous time that the system spends with maskable exceptions disabled.
- Maximum masked time—The maximum amount of continuous time that the system spends with a single interrupt masked.
- Miscellaneous exception—A software exception which is neither an unimplemented instruction nor a `trap` instruction. For further information, refer to "Miscellaneous Exceptions" on page 8–33.
- Nested interrupts—See pre-emption.
- Pre-emption—The process of a high-priority interrupt taking control when a lower-priority ISR is already running. Also: nested interrupts.
- Software exception—An exception caused by a software condition; that is, any exception other than a hardware interrupt. This includes unimplemented instructions and `trap` instructions.
- Unimplemented instruction—An implementation-dependent instruction that is not supported on the particular Nios II core implementation that is in your system. For example, in the Nios II/e core, `mul` and `div` are unimplemented.
- Worst-case exception (or interrupt) latency—The value of the exception (or interrupt) latency, including the maximum disabled time or maximum masked time. Including the maximum disabled or masked time accounts for the case when the exception (or interrupt) occurs at the beginning of the masked or disabled time.

Interrupt Controllers

The configuration of Nios II exception processing depends on the type of hardware interrupt controller. You select the hardware interrupt controller when you instantiate the Nios II processor in SOPC Builder. This section describes the kinds of interrupt controllers available with the Nios II processor.

For details about selecting a hardware interrupt controller, refer to the *Instantiating the Nios II Processor in SOPC Builder* chapter of the *Nios II Processor Reference Handbook.*

Internal Interrupt Concepts

With the IIC, Nios II exception handling is implemented in classic RISC fashion. All exception types, including hardware interrupts, are dispatched through a single top-level exception funnel. This means that all exceptions (hardware and software) are handled by code residing at a single location, the exception address.

The IIC is a simple, nonvectored hardware interrupt controller. Upon receipt of an interrupt request, the IIC transfers control to the general exception address. The hardware indicates which IRQ is currently asserted, and allows software to mask individual interrupts.

With the IIC, the HAL interrupt funnel identifies the hardware interrupt cause in software, and dispatches the registered ISR.

The IIC is available in all revisions of the Nios II processor.

External Interrupt Concepts

The EIC interface enables the Nios II processor to work with a separate external interrupt controller component. An EIC can be a custom component that you provide. Altera provides an example of an EIC, the vectored interrupt controller (VIC).

For details about the VIC, refer to the *Vectored Interrupt Controller Core* chapter in the *Embedded Peripherals IP User Guide*.

With an EIC, hardware interrupts are handled separately from software exceptions. Hardware interrupts have separate vectors and funnels. Each interrupt can have its own handler, or handlers can be shared. Software exception handling is the same as with the IIC.

The EIC interface provides extensive capabilities for customizing your interrupt hardware. You can design, connect and configure an interrupt controller that is optimal for your application.

When an external hardware interrupt occurs, the Nios II processor transfers control to an individual vector address, which can be unique for each interrupt. The HAL provides the following services:

- Registering ISRs
- Setting up the vector table
- Transferring control from the vector table to your ISR

An EIC can be used with shadow register sets. A shadow register set is a complete alternate set of Nios II general-purpose registers, which can be used to maintain a separate runtime context for an ISR.

An EIC provides the following information about each hardware interrupt:

Requested Handler Address

The requested handler address (RHA) specifies the address of the funnel associated with the hardware interrupt. The availability of an RHA for each interrupt allows the Nios II processor to jump directly to the interrupt funnel specific to the interrupting device, reducing interrupt latency.

Requested Interrupt Level

The Nios II processor uses the requested interrupt level (RIL) to prioritize the hardware interrupt request versus any interrupt it is currently processing. While handling an interrupt, the Nios II processor normally only takes higher-level interrupts.

Requested Register Set

If shadow register sets are implemented on the Nios II core, an EIC specifies a requested register set (RRS) when it asserts an interrupt request. When the Nios II processor takes the hardware interrupt, the processor switches to the requested register set. When an interrupt has a dedicated register set, the ISR avoids the overhead of saving registers for a context switch.

Multiple hardware interrupts can be configured to share a register set. However, at run time, the Nios II processor does not allow pre-emption between interrupts assigned to the same register set unless this feature is specifically enabled. In this case, the ISRs must be written so as to avoid register corruption.

Refer to the *Vectored Interrupt Controller Core* chapter in the *Embedded Peripherals IP User Guide* for an example of a driver that manages pre-emption within a register set.

Requested NMI Mode

If the interrupt is configured as a nonmaskable interrupt (NMI), the EIC asserts requested NMI (RNMI). Any hardware interrupt can be nonmaskable, depending on the configuration of the EIC. An NMI typically signals a critical system event requiring immediate handling, to ensure either system stability or deterministic real-time performance.

Shadow Register Sets

Although shadow register sets can be implemented independently of the EIC interface, typically the two features are used together. Combining shadow register sets with an appropriate EIC, you can minimize or eliminate the context switch overhead for critical hardware interrupts.

Latency and Response Time

Exception (interrupt) latency, as defined in the previous section, is the time required for the hardware to respond to an exception. Response time, in contrast, is the time required to begin executing code specific to the exception cause, such as a particular ISR. Response time includes latency plus the time required for the HAL to carry out some or all of the following overhead tasks:

- Context save—Saving registers on the stack
- RTOS context switch—Calling context-switch function(s) if an RTOS is implemented
- Dispatch handler—Determining the cause of the exception, and transferring control to a specific handler or ISR

If you are concerned with system performance, response time is the more important than latency, because it reflects the time elapsed between the physical event and the system's specific response to that event.

This section discusses the available options for exception handling, and their impact on latency and response time.

Internal or External Interrupt Controller

The Nios II IIC is nonvectored, requiring the processor to dispatch ISRs with a software routine. An EIC, by contrast, can be vectored. With a vectored EIC, such as the Altera® VIC, ISR dispatch is managed by hardware, eliminating the processing time required for ISR dispatch, and substantially reducing hardware interrupt response time.

An EIC has no impact on software exception latency or response time.

Shadow Register Sets

In conjunction with an EIC, shadow register sets speed up hardware interrupt response by making it unnecessary to save registers on the stack. This feature has no impact on interrupt latency, but significantly reduces interrupt response time.

Shadow register sets have no impact on software exception response time.

How the Hardware Works

The Nios II processor can respond to exceptions including software exceptions and hardware interrupts. When the Nios II processor responds to an exception, it performs the following tasks:

1. Saves the `status` register in `estatus`. This means that if hardware interrupts are enabled, the `PIE` bit of `estatus` is set.
2. Disables hardware interrupts.
3. Saves the next execution address in `ea` (`r29`).
4. Transfers control to the appropriate exception address, as follows:
   - Software exception or internal hardware interrupt—Nios II processor general exception address
   - External hardware interrupt—Device-specific interrupt address

All Nios II exception types are precise. This means that after an exception is handled, the Nios II processor can re-execute the instruction that caused the exception.

The Nios II processor always re-executes the instruction after the software exception handler or ISR has completed, when the exception processing system returns to the application context.

Several exception types, such as the advanced exceptions, are optional in the Nios II processor core. The presence of these exception types depends on how the hardware designer configures the Nios II core at the time of hardware generation.

The processor's response to hardware interrupts depends on which interrupt controller is implemented. The following sections describe the hardware behavior with each interrupt controller.

For details about the Nios II processor exception controller and hardware interrupt controllers, including a list of optional exception types, refer to the *Processor Architecture* chapter of the *Nios II Processor Reference Handbook*.

How the Internal Interrupt Controller Works

With the IIC, 32 independent hardware interrupt signals are available. These interrupt signals allow software to prioritize interrupts, although the interrupt signals themselves have no inherent priority.

With the IIC, Nios II exceptions are not vectored. Therefore, the same exception address receives control for all types of exceptions. The general exception funnel at that address must determine the type of software exception or hardware interrupt.

How an External Interrupt Controller Works

With an EIC, the Nios II processor supports an arbitrary number of independent hardware interrupt signals. Interrupts are typically vectored, with interrupt priority levels associated in hardware. Vectoring allows the Nios II processor to transfer control directly to each ISR. Hardware interrupt levels allow the most critical interrupts to pre-empt lower-priority interrupts. Because both of these features are implemented in hardware, the system can handle an interrupt without executing general exception funnel code.

The details of hardware interrupt vectoring and prioritization are specific to the EIC implementation. To see an example of an EIC implementation, refer to the *Vectored Interrupt Controller Core* chapter in the *Embedded Peripherals IP User Guide*.

The HAL supports external interrupt controllers only if they are connected in one of the following ways:

- Directly to the Nios II EIC interface
- Through the daisy-chain port on another EIC

Interrupt Service Routines

Software often communicates with peripheral devices using hardware interrupts. When a peripheral asserts its IRQ, it diverts the processor's normal execution flow. When such an interrupt occurs, an appropriate ISR must handle this interrupt and return the processor to its pre-interrupt state on completion.

When you create a board support package (BSP) project, the build tools include all needed device drivers. You do not need to write HAL ISRs unless you are interfacing to a custom peripheral. For reference purposes, this section describes the framework provided by HAL BSPs for handling hardware interrupts.

Refer to existing handlers for Altera SOPC Builder components for examples of how to write HAL ISRs.

For more details about the Altera-provided HAL handlers, refer to the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

HAL APIs for Hardware Interrupts

The HAL provides an enhanced application program interface (API) for writing, registering and managing ISRs. This API is compatible with both internal and external hardware interrupt controllers.

Altera also supports a legacy hardware interrupt API. This API supports only the IIC. If you have a custom driver written prior to Nios II v9.1, it uses the legacy API.

Both interrupt APIs include the following types of routines:

- Routines to be called by a device driver to register an ISR
- Routines to be called by an ISR to manage its environment
- Routines to be called by BSP or application code to control ISR behavior

Both interrupt APIs support the following types of BSPs:

- HAL BSP without an RTOS
- HAL-based RTOS BSP, such as a MicroC/OS-II BSP

The legacy API is deprecated. Write new drivers using the enhanced API, even if they are only intended to support the IIC. Drivers for devices supporting an EIC must use the enhanced API. Existing legacy drivers continue to be supported until further notice. Make plans to port them to the enhanced API.

When an EIC is present, the controller's driver provides driver settings for the BSP, which can be used to configure the driver. The number and types of the settings depends on the EIC implementation and the number of EICs present.

For an example of EIC driver settings, refer to the *Vectored Interrupt Controller Core* chapter in the *Embedded Peripherals IP User Guide*.

Selecting an Interrupt API

When the SBT creates a BSP, it determines whether the BSP must implement the legacy interrupt API. Each driver that supports the enhanced API publishes this capability to the SBT through its <*driver name*>**_sw.tcl** file. The BSP implements the enhanced API if all drivers support it. It implements the legacy API only if required by the drivers.

In determining the interrupt API to use, the SBT ignores any devices whose interrupts are not connected to the Nios II processor associated with the BSP.

A driver can publish its interrupt API support by way of a software property. The driver's <*driver name*>**_sw.tcl** file uses the `set_sw_property` command to set `supported_interrupt_apis` to either `legacy_interrupt_api`, `enhanced_interrupt_api`, or both.

Drivers supporting the enhanced API always publish that support. If `supported_interrupt_apis` is undefined, the SBT assumes that the driver only supports the legacy API.

Starting in 9.1, all Altera device drivers support both APIs. These drivers can be used in a BSP along with legacy drivers. The SBT determines whether the legacy API is required, and implements it only if it is required. If there are no drivers requiring the legacy API, the BSP implements the enhanced API.

A driver can be written to support only the enhanced API. However, you cannot combine such a driver with legacy drivers.

For details about writing a driver to support both APIs, refer to "Supporting Multiple Interrupt APIs" on page 8–11.

The Enhanced HAL Interrupt API

The enhanced HAL interrupt API defines the functions listed in Table 8–1 to manage hardware interrupt processing.

Table 8–1. Enhanced HAL Interrupt API Functions

Function Name	Implemented By
`alt_ic_isr_register()`	Interrupt controller driver *(1)*
`alt_ic_irq_enable()`	Interrupt controller driver *(1)*
`alt_ic_irq_disable()`	Interrupt controller driver *(1)*
`alt_ic_irq_enabled()`	Interrupt controller driver *(1)*
`alt_irq_disable_all()`	HAL
`alt_irq_enable_all()`	HAL
`alt_irq_enabled()`	HAL

Note to Table 8–1:

(1) If the system is based on an EIC, these functions must be implemented by the EIC driver. If the system is based in the IIC, the functions are implemented by the HAL. For details about each function, refer to the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

The functions in Table 8–1 work for both internal and external interrupt controllers.

For details about the enhanced interrupt API functions, refer to the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

Using the enhanced HAL API to implement ISRs requires that you perform the following steps:

1. Write your ISR that handles hardware interrupts for a specific device.
2. Ensure that your program registers the ISR with the HAL by calling the `alt_ic_isr_register()` function. `alt_ic_isr_register()` enables hardware interrupts for you.

The SBT inserts the following symbol definitions in **system.h**, indicating the configuration of the processor's interrupt-related hardware options:

- `NIOS2_EIC_PRESENT`—If defined, indicates that one or more EICs are present
- `NIOS2_NUM_OF_SHADOW_REG_SETS`—Indicates how many shadow register sets are present. The maximum value is 63. If there are no shadow register sets, the value is 0.

The External Interrupt Controller Driver

To be compliant with the HAL enhanced interrupt API, the driver for an EIC must support the functions listed under "The Enhanced HAL Interrupt API". In addition, it can provide functions to support any special hardware features. For examples, refer to "Using the HAL Interrupt API with the VIC".

Using the HAL Interrupt API with the VIC

The Altera driver for the VIC component supports the HAL enhanced interrupt API.

The VIC driver provides support for multiple, daisy-chained VIC devices. It also includes support for shadow register sets. A BSP driver setting allows you to enable automatic pre-emption (fast nested interrupts). Automatic pre-emption means that the Nios II processor leaves maskable exceptions enabled when accepting a hardware interrupt.

For more information about fast nested interrupts, refer to "Exception Processing" in the *Programming Model* chapter of the *Nios II Processor Reference Handbook*.

The VIC device driver also provides the following device-specific functions:

- `int alt_vic_sw_interrupt_set(alt_u32 ic_id, alt_u32 irq);`
- `int alt_vic_sw_interrupt_clear(alt_u32 ic_id, alt_u32 irq);`
- `alt_u32 alt_vic_sw_interrupt_status(alt_u32 ic_id, alt_u32 irq);`
- `int alt_vic_irq_set_level(alt_u32 ic_id, alt_u32 irq, alt_u32 level);`

For a detailed discussion of the VIC device-specific driver routines, refer to the *Vectored Interrupt Controller Core* chapter in the *Embedded Peripherals IP User Guide*.

The EIC driver controls where hardware interrupt vector tables are located. For example, the Altera VIC driver locates the vector table in the `.text` section by default, but allows you to position the vector table in a different section with a driver setting.

The memory in which you place the vector table must be connected to both instruction and data master ports on the Nios II processor.

The Legacy HAL Interrupt API

The legacy HAL interrupt API defines the following functions to manage hardware interrupt processing:

- `alt_irq_register()`
- `alt_irq_disable()`
- `alt_irq_enable()`
- `alt_irq_disable_all()`
- `alt_irq_enable_all()`
- `alt_irq_interruptible()`
- `alt_irq_non_interruptible()`
- `alt_irq_enabled()`

For details about these functions, refer to the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

Legacy drivers do not define the `supported_interrupt_apis` property. The absence of this property indicates to the SBT that they require the legacy interrupt API.

Using the legacy HAL API to implement ISRs requires that you perform the following steps:

1. Write your ISR that handles hardware interrupts for a specific device.
2. Ensure that your program registers the ISR with the HAL by calling the `alt_irq_register()` function. `alt_irq_register()` enables hardware interrupts for you, by calling `alt_irq_enable_all()`.

Supporting Multiple Interrupt APIs

When you write or update a custom device driver, Altera recommends that you write it in one of two ways:

- Write it to support the enhanced HAL interrupt API—Write the driver this way if you intend to use it only in combination with other drivers supporting the enhanced API.
- Write it to support both the enhanced and the legacy API—Write the driver this way if you need to use it in combination with legacy drivers supporting only the legacy API.

Altera recommends using the enhanced API even if your Nios II processor implements the IIC. The enhanced API supports both types of interrupt controller, and the legacy API is deprecated.

When the SBT selects the interrupt API, it defines one of the following symbols in **system.h**, to identify which interrupt API is available:

- `ALT_ENHANCED_INTERRUPT_API_PRESENT`—Defined if the enhanced API is implemented
- `ALT_LEGACY_INTERRUPT_API_PRESENT`—Defined if the legacy API is implemented

In your driver code, use these symbols to determine which API calls to make.

To support both APIs, your driver must publish its interrupt API support by way of a software property. In your driver's *<driver name>*_**sw.tcl** file, use the `set_sw_property` command to set `supported_interrupt_apis` to both `legacy_interrupt_api` and `enhanced_interrupt_api`.

For details about the `set_sw_property` command, refer to the "Tcl Commands" section of the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

HAL ISR Restrictions

When your system has an EIC, the HAL interrupt support imposes the following restrictions:

- Nonmaskable hardware interrupts must use a shadow register set.
- Nonmaskable hardware interrupts cannot share a register set with a maskable hardware interrupt.

Writing an ISR

The ISR you write must match the prototype that `alt_ic_isr_register()` expects. The prototype for your ISR function must match the following prototype:

```
void (*alt_isr_func) (void* isr_context)
```

The parameter definition of `context` is the same as for the `alt_ic_isr_register()` function.

From the point of view of the HAL exception handling system, the most important function of an ISR is to clear the associated peripheral's interrupt condition. The procedure for clearing an hardware interrupt condition is specific to the peripheral.

For details, refer to the relevant chapter in the *Embedded Peripherals IP User Guide*.

When the ISR has finished servicing the hardware interrupt, it must return to the HAL interrupt funnel that called it.

If you write your ISR in assembly language, use `ret` to return. The HAL interrupt funnel issues an `eret` after restoring the application context.

Using Interrupt Funnels

The HAL creates a vector table for each EIC connected to the Nios II processor. In the vector table, the HAL inserts a branch to the correct funnel for each interrupt-driven device supported by the BSP, depending on the device driver characteristics and pre-emption settings. Funnels can be shared by multiple hardware interrupts, if the drivers have compatible characteristics.

The funnel code receives control from the general exception or interrupt vector, depending on which interrupt controller is implemented. The funnel performs tasks such as switching the stack pointer, saving registers and calling RTOS context-switch routines, and transfers control to the handler. When the handler returns, the funnel code performs tasks such as calling RTOS process-dispatch routines and restoring registers, and transfers control to the appropriate foreground task.

The HAL includes the following interrupt funnels:

- Shadow register set, pre-emption disabled—Hardware interrupt assigned to a shadow register set, with pre-emption within the register set disabled. This funnel does not preserve register context. Hardware guarantees that only one ISR runs with the shadow register set at any time.
- Shadow register set, pre-emption enabled—Hardware interrupt assigned to a shadow register set. An interrupt can pre-empt another interrupt using the same register set. This funnel preserves register context, so that handlers is assigned to the same register set do not corrupt one another's context.
- Nonmaskable interrupt—Nonmaskable hardware interrupt assigned to a shadow register set, with pre-emption within the register set disabled. This funnel does not preserve register context. Hardware guarantees that only one ISR runs in the shadow register set at any time.

The HAL funnel code is called from the vector table.

Running in a Restricted Environment

ISRs run in a restricted environment. A large number of the HAL API calls are not available from ISRs. For example, accesses to the HAL file system are not permitted. As a general rule, when writing your own ISR, never include function calls that can block for any reason (such as waiting for a hardware interrupt).

The *HAL API Reference* chapter of the *Nios II Software Developer's Handbook* identifies those API functions that are not available to ISRs.

Be careful when calling ANSI C standard library functions inside of an ISR. Avoid using the C standard library I/O API, because calling these functions can result in deadlock within the system, that is, the system can become permanently blocked in the ISR.

In particular, do not call `printf()` from within an ISR unless you are certain that `stdout` is mapped to a non-interrupt-based device driver. Otherwise, `printf()` can deadlock the system, waiting for a hardware interrupt that never occurs because interrupts are disabled.

Managing Pre-Emption

The HAL enhanced interrupt API supports interrupt pre-emption. When pre-emption is enabled, a higher-level interrupt can take control even if an ISR is already running. A device driver must be specifically written to function correctly under pre-emption. When a device driver supports pre-emption, it publishes this capability through the `isr_preemption_supported` driver setting. When constructing the BSP, the SBT checks each device driver to determine whether it supports pre-emption. If all drivers in the BSP support pre-emption, it is enabled.

Legacy device drivers do not publish the `isr_preemption_supported` property. Therefore the SBT assumes that they do not support pre-emption. If your legacy custom driver supports pre-emption, and you want to allow pre-emption in the BSP, you must update the driver to use the enhanced interrupt API.

To enable the enhanced interrupt API, ensure that all drivers in the system are updated to use the enhanced interrupt API.

For details about the `isr_preemption_supported` driver setting, refer to the `set_sw_property` command in the "Tcl Commands" section of the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Operating systems can also publish the `isr_preemption_supported` property.

The HAL enhanced interrupt API supports automatic pre-emption. Automatic pre-emption means that maskable exceptions remain enabled when the processor accepts the hardware interrupt. This means that your ISR can immediately be pre-empted by a higher-level ISR, without any need to execute the `eret` instruction.

Automatic pre-emption can only take place when the pre-empting hardware interrupt uses a different register set from the interrupt being pre-empted.

Automatic pre-emption is only available if you enable it in the BSP settings.

Registering an ISR with the Enhanced Interrupt API

Before the software can use an ISR, you must register it by calling `alt_ic_isr_register()`. The prototype for `alt_ic_isr_register()` is:

```
int alt_ic_isr_register(alt_u32 ic_id,
                        alt_u32 irq,
                        alt_isr_func isr,
                        void *isr_context,
                        void* flags)
```

The function has the following parameters:

- `ic_id` is the interrupt controller identifier (ID) as defined in **system.h**. With daisy-chained EICs, `ic_id` identifies the EIC in the daisy chain. With the IIC, `ic_id` is not significant.
- `irq` is the hardware interrupt number for the device, as defined in **system.h**.
  - For the IIC, `irq` is the IRQ number. Interrupt priority corresponds inversely to the IRQ number. Therefore, IRQ_0 represents the highest priority interrupt and IRQ_{31} is the lowest.
  - For an EIC, `irq` is the interrupt port ID.
- `isr_context` points to a data structure associated with the device driver instance. `isr_context` is passed as the input argument to the isr function. It is used to pass context-specific information to the ISR, and can point to any ISR-specific information. The context value is opaque to the HAL; it is provided entirely for the benefit of the user-defined ISR.
- `isr` is a pointer to the ISR function that is called in response to IRQ number `irq`. The ISR function prototype is:

  ```
  void (void* isr_context);
  ```

  The input argument provided to this function is the `isr_context`.

  Registering a null pointer for `isr` results in the interrupt being disabled.

- `flags` is reserved.

The HAL registers the ISR by one of the following methods:

- For the IIC, by the storing the function pointer, `isr`, in a lookup table.
- For an EIC, by configuring the vector table with the appropriate funnel code, as described in "Using Interrupt Funnels" on page 8–12.

The return code from `alt_ic_isr_register()` is zero if the function succeeded, and nonzero if it failed.

If the HAL registers your ISR successfully, the associated Nios II hardware interrupt (as defined by `irq`) is enabled on return from `alt_ic_isr_register()`.

Hardware-specific initialization might also be required.

When a specific interrupt occurs, the HAL code ensures that the registered ISR is correctly dispatched.

For details about hardware interrupt initialization specific to your peripheral, refer to the relevant chapter of the *Embedded Peripherals IP User Guide*. For details about `alt_ic_isr_register()`, refer to the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

The HAL legacy interrupt API provides a different function for registering hardware interrupts. For all new and updated drivers, Altera recommends using the enhanced API described in this section. The legacy API function, `alt_irq_register()`, is described in the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook.*

Enabling and Disabling Interrupts

The HAL enhanced interrupt API provides the functions `alt_ic_irq_disable()`, `alt_ic_irq_enable()`, `alt_ic_irq_enabled()`, `alt_irq_disable_all()`, `alt_irq_enable_all()`, and `alt_irq_enabled()` to allow a program to disable hardware interrupts for certain sections of code, and reenable them later. `alt_ic_irq_disable()` and `alt_ic_irq_enable()` allow you to disable and enable individual interrupts. `alt_irq_disable_all()` disables all interrupts, and returns a context value. To reenable hardware interrupts, you call `alt_irq_enable_all()` and pass in the context parameter. In this way, interrupts are returned to their state prior to the call to `alt_irq_disable_all()`. `alt_irq_enabled()` returns nonzero if maskable exceptions are enabled. `alt_ic_irq_enabled()` determines whether a specified interrupt is enabled.

Disable hardware interrupts for as short a time as possible. Maximum interrupt latency increases with the longest amount of time interrupts are disabled. For more information about disabled interrupts, refer to "Keep Interrupts Enabled" on page 8–20.

For details about these functions, refer to the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

The HAL legacy interrupt API provides different functions for enabling and disabling individual interrupts. For all new and updated drivers, Altera recommends using the enhanced API described in this section. The legacy API functions, `alt_irq_disable()` and `alt_irq_enable()`, are described in the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

Configuring an External Interrupt Controller

The driver for an EIC provides specialized driver settings that are created at the time you generate the BSP. These settings customize the driver to the EIC configuration found in the Nios II system. The number and type of settings depends on the EIC implementation, as well as on the number and configuration of EICs in the hardware system. The SBT creates the BSP with default values, selected to ensure useful system performance. You can optimize these settings at the time you create the BSP. For details of how to manipulate the EIC driver settings, refer to the documentation for your specific EIC.

The driver for an EIC can provide specialized functions to manage any implementation-specific features of the EIC. An example would be modifying interrupt priority levels at runtime.

For examples, refer to the *Vectored Interrupt Controller Core* chapter in the *Embedded Peripherals IP User Guide*.

C Example

Example 8–1 illustrates an ISR that services a hardware interrupt from a button parallel I/O (PIO) component. This example is based on a Nios II system with a 4-bit PIO peripheral connected to push buttons. An IRQ is generated any time a button is pushed. The ISR code reads the PIO peripheral's edge capture register and stores the value to a global variable. The address of the global variable is passed to the ISR in the context pointer.

Example 8–1. An ISR to Service a Button PIO Interrupt

```
#include "system.h"
#include "altera_avalon_pio_regs.h"
#include "alt_types.h"

#ifdef ALT_ENHANCED_INTERRUPT_API_PRESENT
static void handle_button_interrupts(void* context)
#else
static void handle_button_interrupts(void* context, alt_u32 id)
#endif
{
  /* Cast context to edge_capture's type. It is important that this
     be declared volatile to avoid unwanted compiler optimization. */
  volatile int* edge_capture_ptr = (volatile int*) context;

  /*
   * Read the edge capture register on the button PIO.
   * Store value.
   */
  *edge_capture_ptr =
    IORD_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE);

  /* Write to the edge capture register to reset it. */
  IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE, 0);

  /* Read the PIO to delay ISR exit. This is done to prevent a
     spurious interrupt in systems with high processor -> pio
     latency and fast interrupts. */
  IORD_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE);
}
```

Example 8–2 shows an example of the code for the main program that registers the ISR with the HAL.

Based on this code, the following execution flow is possible:

1. Button is pressed, generating an IRQ.
2. The ISR gains control.
   - With the IIC, the HAL general exception funnel gains control of the processor, and dispatches the `handle_button_interrupts()` ISR.
   - With an EIC, the processor branches to the address in the vector table, which transfers control to the `handle_button_interrupts()` ISR.
3. `handle_button_interrupts()` services the hardware interrupt and returns.
4. Normal program operation continues with an updated value of `edge_capture`.

Example 8–2. Registering the Button PIO ISR with the HAL

```
#include "sys/alt_irq.h"
#include "system.h"

...
/* Declare a global variable to hold the edge capture value. */
volatile int edge_capture;
...

/* Initialize the button_pio. */
static void init_button_pio()
{
    /* Recast the edge_capture pointer to match the
        alt_irq_register() function prototype. */
    void* edge_capture_ptr = (void*) &edge_capture;

    /* Enable all 4 button interrupts. */
    IOWR_ALTERA_AVALON_PIO_IRQ_MASK(BUTTON_PIO_BASE, 0xf);

    /* Reset the edge capture register. */
    IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE, 0x0);

    /* Register the ISR. */
#ifdef ALT_ENHANCED_INTERRUPT_API_PRESENT
    alt_ic_isr_register(BUTTON_PIO_IRQ_INTERRUPT_CONTROLLER_ID,
                        BUTTON_PIO_IRQ,
                        handle_button_interrupts,
                        edge_capture_ptr, 0x0);
#else
    alt_irq_register( BUTTON_PIO_IRQ,
                        edge_capture_ptr,
                        handle_button_interrupts );
#endif
}
```

Additional software examples that demonstrate implementing ISRs, such as the `count_binary` example project template, are installed with the Nios II Embedded Design Suite (EDS).

Upgrading to the Enhanced HAL Interrupt API

If you have custom device drivers, Altera recommends that you upgrade them to use the enhanced HAL interrupt API. The enhanced API maintains compatibility with the IIC, while supporting external interrupt controllers. The legacy HAL interrupt API is deprecated, and will be removed in a future release of the Nios II EDS.

If you plan to use an EIC, you must upgrade your custom driver to the enhanced HAL interrupt API.

Upgrading your device driver is very simple, requiring only minor changes to some function calls.

Table 8–2 shows the legacy API functions that need to be modified, with the corresponding enhanced API functions.

For details of the API functions, refer to the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

Table 8–2. HAL Interrupt API Functions to Upgrade

Legacy API Function	Enhanced API Function
`alt_irq_register()`	`alt_ic_isr_register()`
`alt_irq_disable()`	`alt_ic_irq_disable()`
`alt_irq_enable()`	`alt_ic_irq_enable()`

If your upgraded driver might need to function in a BSP with legacy drivers, code it to support both APIs, as described in "Supporting Multiple Interrupt APIs" on page 8–11.

Improving ISR Performance

If your software uses hardware interrupts extensively, the performance of ISRs is probably the most critical determinant of your overall software performance. This section discusses both hardware and software strategies to improve ISR performance.

Software Performance Improvements

In improving your ISR performance, you probably consider software changes first. However, in some cases it might require less effort to implement hardware design changes that increase system efficiency. For a discussion of hardware optimizations, refer to "Hardware Performance Improvements" on page 8–24.

The following sections describe changes you can make in the software design to improve ISR performance.

Execute Time-Intensive Algorithms in the Application Context

ISRs provide rapid, low latency response to changes in the state of hardware. They do the minimum necessary work to clear the hardware interrupt condition and then return. If your ISR performs lengthy, noncritical processing, it can interfere with more critical tasks in the system.

If your ISR requires lengthy processing, design your software to perform this processing outside of the exception context. The ISR can use a message-passing mechanism to notify the application code to perform the lengthy processing tasks.

Deferring a task is simple in systems based on an RTOS such as MicroC/OS-II. In this case, you can create a thread to handle the processor-intensive operation, and the ISR can communicate with this thread using any of the RTOS communication mechanisms, such as event flags or message queues.

You can emulate this approach in a single-threaded HAL-based system. The main program polls a global variable managed by the ISR to determine whether it needs to perform the processor-intensive operation.

Implement Time-Intensive Algorithms in Hardware

Processor-intensive tasks must often transfer large amounts of data to and from peripherals. A general-purpose processor such as the Nios II processor is not the most efficient way to do this. Use direct memory access (DMA) hardware if it is available.

For information about programming with DMA hardware, refer to "Using DMA Devices" in the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

Increase Buffer Size

If you are using DMA to transfer large data buffers, the buffer size can affect performance. Small buffers imply frequent interrupts, which lead to high overhead.

Increase the size of the transaction data buffer(s).

Use Double Buffering

Using DMA to transfer large data buffers might not provide a large performance increase if the Nios II processor must wait for DMA transactions to complete before it can perform the next task.

Double buffering allows the Nios II processor to process one data buffer while the hardware is transferring data to or from another.

Keep Interrupts Enabled

When interrupts are disabled, the Nios II processor cannot respond quickly to hardware interrupt events. Buffers and queues can fill or overflow. Even in the absence of overflow, maximum interrupt processing time can increase after interrupts are re-enabled, because the ISRs must process data backlogs.

Disable interrupts as infrequently as possible, and for the briefest time possible.

Instead of disabling all interrupts, call `alt_ic_irq_disable()` and `alt_ic_irq_enable()` to enable and disable individual interrupts.

To protect shared data structures, use RTOS structures such as semaphores.

Disable all interrupts only during critical system operations. In the code where interrupts are disabled, perform only the bare minimum of critical operations, and reenable interrupts immediately.

Use Fast Memory

ISR performance depends on memory speed.

For best performance, place the ISRs and the stack in the fastest available memory: preferably tightly-coupled memory (if available), or on-chip memory.

If it is not possible to place the main stack in fast memory, consider using a separate exception stack, mapped to a fast memory section, as described in the next section.

For more information about mapping memory, refer to "Memory Usage" in the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*. For more information about tightly-coupled memory, refer to the *Cache and Tightly-Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.

Use a Separate Exception Stack

The HAL implements two types of separate exception stack. Their availability depends on the interrupt controller, as described in this section. Table 8–3 outlines the availability of separate exception stacks, and how they can be used with each type of interrupt controller.

Using a separate exception stack entails a slight additional overhead. When processing a software exception or hardware interrupt, the processor must execute an additional instruction on entry and exit, to change the stack pointer. Take this additional processing time into account if your interrupt response requirements are extremely strict.

Separate General Exception Stack

The separate general exception stack is available with either the internal or the external interrupt controller.

Use the `hal.linker.enable_exception_stack` BSP setting to enable a separate general exception stack.

The HAL general exception funnel code takes care of correctly changing the stack pointer on entry to and exit from an exception handler.

Separate Hardware Interrupt Stack

The separate hardware interrupt stack is available with the EIC interface. The separate hardware interrupt stack is not applicable to the IIC. With the IIC, hardware interrupts and software exceptions use the same stack.

The following BSP settings enable you to control the separate hardware interrupt stack:

- `hal.linker.enable_interrupt_stack` enables a separate hardware interrupt stack.
- `hal.linker.interrupt_stack_size` controls the size of the hardware interrupt stack.
- `hal.linker.interrupt_stack_memory_region_name` enables you to control where the hardware interrupt stack is positioned in memory.

The HAL funnel code takes care of correctly changing the stack pointer on entry to and exit from an ISR.

Table 8–3. Separate Exception Stack Usage

Interrupt Controller	BSP Settings: Separate General Exception Stack Enabled	BSP Settings: Separate Hardware Interrupt Stack Enabled	Application Stack	General Exception Stack	Hardware Interrupt Stack
Internal	No	—	■ Application ■ Software exceptions ■ Hardware interrupts	—	—
	Yes	—	Application	■ Software exceptions ■ Hardware interrupts	—
External	No	No	■ Application ■ Software exceptions ■ Hardware interrupts	—	—
		Yes	■ Application ■ Software exceptions	—	Hardware interrupts
	Yes	No	■ Application ■ Hardware interrupts	Software exceptions	—
		Yes	Application	Software exceptions	Hardware interrupts

If your ISR is located in a vector table, the HAL does not provide funnel code. In this case, your code must manage the stack pointer, as well as all other funnel code functions.

For further details about implementing a separate hardware interrupt stack, refer to *AN595: Vectored Interrupt Controller Applications and Usage*.

Use Nested Hardware Interrupts

By default, the HAL disables interrupts when it dispatches an ISR. This means that only one ISR can execute at any time, and ISRs are executed on a first-come first-served basis. This reduces the system overhead associated with interrupt processing, and simplifies ISR development. The ISR does not need to be reentrant. ISRs can use and modify any global or static data structures or hardware registers that are not shared with application code.

However, first-come first-served execution means that the HAL hardware interrupt priorities only have an effect if two IRQs are active at the same time. A low-priority interrupt occurring before a higher-priority interrupt can prevent the higher-priority ISR from executing. This is a form of priority inversion, and it can have a significant impact on ISR performance in systems that generate frequent interrupts.

A software system can achieve full hardware interrupt prioritization by using nested ISRs. With nested ISRs, higher-priority interrupts are allowed to interrupt lower-priority ISRs.

This technique can improve the response time for higher-priority interrupts.

Nested ISRs increase the processing time for lower-priority hardware interrupts.

If your ISR is very short, it might not be worth the overhead to enable nested hardware interrupts. Enabling nested interrupts for a short ISR can actually increase the response time for higher-priority interrupts.

If you use a separate exception stack with the IIC, you cannot nest hardware interrupts. For more information about separate exception stacks, refer to "Use a Separate Exception Stack".

Nested Hardware Interrupts with the Internal Interrupt Controller

To implement nested hardware interrupts with the IIC, use the `alt_irq_interruptible()` and `alt_irq_non_interruptible()` functions to bracket code in a processor-intensive ISR. The call to `alt_irq_interruptible()` adjusts the interrupt mask so that higher-priority interrupts can take control from the running ISR. When your ISR calls `alt_irq_non_interruptible()`, the interrupt mask is returned to its previous state.

If your ISR calls `alt_irq_interruptible()`, it must call `alt_irq_non_interruptible()` before returning. Otherwise, the HAL exception handling system might lock up.

Nested Hardware Interrupts with an External Interrupt Controller

The HAL enhanced interrupt API supports nested hardware interrupts, also known as interrupt pre-emption. A device driver must be specifically written to function correctly under pre-emption.

Legacy device drivers do not publish the `isr_preemption_supported` property. Therefore the SBT assumes that they do not support pre-emption. If your legacy custom driver supports pre-emption, and you want to allow pre-emption in the BSP, you must update the driver to use the enhanced HAL interrupt API.

The HAL enhanced interrupt API also supports automatic pre-emption. Automatic pre-emption means that maskable exceptions remain enabled when the processor accepts the hardware interrupt.

For details about pre-emption with an EIC, refer to "Managing Pre-Emption" on page 8–13.

In the vector table, the HAL inserts a branch to the correct funnel for each hardware interrupt, depending on the pre-emption settings.

Locate ISR Body in Vector Table

If you are using a vectored EIC, and you have a critical ISR of small size, you might achieve a performance improvement by positioning the ISR code directly in the vector table. In this way, you eliminate the overhead of branching from the vector table through the HAL funnel to your ISR.

The EIC's driver provides a default vector table entry size. For example, with the Altera VIC, the default size is 16 bytes. To accommodate your ISR, adjust the entry size with a driver setting when you create the BSP.

Positioning an ISR in a vector table is an advanced and error-prone technique, not directly supported by the HAL. You must exercise great caution to ensure that the ISR code fits in the vector table entry. If your ISR overflows the vector table entry, it corrupts other entries in the vector table, and your entire interrupt handling system. When your ISR is located in the vector table, it does not need to be registered. Do not call `alt_ic_isr_register()`, because it overwrites the contents of the vector table. The HAL does not provide funnel code. Therefore, your code must manage all funnel code functions.

For further details about locating an ISR in a vector table, refer to *AN595: Vectored Interrupt Controller Applications and Usage*.

Use Compiler Optimization

For the best performance both in exception context and application context, use compiler optimization level `-O3`. Level `-O2` also produces good results. Removing optimization altogether significantly increases exception response time.

For further information about compiler optimizations, refer to "Reducing Code Footprint" in the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

Hardware Performance Improvements

Several simple hardware changes can provide a substantial improvement in ISR performance. These changes involve editing and regenerating the SOPC Builder module, and recompiling the Quartus® II design.

In some cases, these changes also require changes in the software architecture or implementation. For a discussion of these and other software optimizations, refer to "Software Performance Improvements" on page 8–19.

The following sections describe changes you can make in the hardware design to improve ISR performance.

Use Vectored Hardware Interrupts

By default, the Nios II processor has a nonvectored IIC. The HAL provides software to dispatch each hardware interrupt to its specific ISR. By contrast, vectoring allows the processor to transfer control directly to the ISR with minimal software intervention.

The options available for hardware interrupt vectoring depend on the interrupt controller configured in the Nios II hardware, as described in this section.

Using the Interrupt Vector Custom Instruction

The Nios II processor core offers an interrupt vector custom instruction that accelerates hardware interrupt vector dispatch in the HAL. You can include this custom instruction to improve your program's interrupt response time.

When the interrupt vector custom instruction is present in the Nios II processor, the HAL source detects it at compile time and generates code using the custom instruction.

When using an interrupt vector custom instruction, you cannot use a separate exception stack.

For further information about the interrupt vector custom instruction, refer to "Interrupt Vector Custom Instruction" in the *Instantiating the Nios II Processor in SOPC Builder* chapter of the *Nios II Processor Reference Handbook*.

Using an External Interrupt Controller

The Nios II EIC port allows you to connect a customizable external interrupt controller component. An EIC can be vectored. An example is the Altera VIC.

For details about the VIC, refer to the *Vectored Interrupt Controller Core* chapter in the *Embedded Peripherals IP User Guide*.

Add Fast Memory

Increase the amount of fast on-chip memory available for data buffers. Ideally, implement tightly-coupled memory that the software can use for buffers.

For further information about tightly-coupled memory, refer to the *Cache and Tightly-Coupled Memory* chapter of the *Nios II Software Developer's Handbook*, or to the *Using Nios II Tightly Coupled Memory Tutorial*.

Add a DMA Controller

A DMA controller performs bulk data transfers, reading data from a source address range and writing the data to a different address range. Add DMA controllers to move large data buffers. This allows the Nios II processor to carry out other tasks while data buffers are being transferred.

For information about DMA controllers, refer to the *DMA Controller Core* and *Scatter-Gather DMA Controller Core* chapters in the *Embedded Peripherals IP User Guide*.

Place the Handler in Fast Memory

For the fastest execution of exception handler code, place the handler in a fast memory device. For example, an on-chip RAM with zero wait states is preferable to a slow SDRAM. For best performance, store exception handling code and data in tightly-coupled memory. The Nios II EDS includes example designs that demonstrate the use of tightly-coupled memory for ISRs.

Use a Fast Nios II Core

For processing in both the exception context and the application context, the Nios II/f core is the fastest, and the Nios II/e core (designed for small size) is the slowest.

Select Hardware Interrupt Priorities

Hardware interrupt priority levels can have a significant impact on system performance. If two interrupts can be asserted at the same time, it is important to assign a higher priority level to the more critical interrupt, so that it runs in preference to the less critical interrupt.

Hardware Interrupt Priorities with the Internal Interrupt Controller

When selecting the IRQ for each peripheral, remember that the HAL hardware interrupt funnel treats IRQ_0 as the highest priority. Assign each peripheral's interrupt priority based on its need for fast servicing in the overall system. Avoid assigning multiple peripherals to the same IRQ.

Hardware Interrupt Priorities with an External Interrupt Controller

With an EIC, the hardware interrupt priority level can be more flexible than with the IIC. The method of assigning priority levels to IRQs depends on the specific EIC implementation.

For example, with the Altera VIC, you can adjust hardware interrupt priority levels at runtime, with the `alt_vic_irq_set_level()` function.

For details about the VIC, refer to the *Vectored Interrupt Controller Core* chapter in the *Embedded Peripherals IP User Guide*.

Debugging ISRs

You can debug an ISR by setting breakpoints in the ISR. The debugger completely halts the processor on reaching a breakpoint. In the meantime, however, the other hardware in your system continues to operate. Therefore, it is inevitable that other interrupts are ignored while the processor is halted. You can use the debugger to step through the ISR code, but the status of other interrupt-driven device drivers is generally invalid by the time you return the processor to normal execution. You must reset the processor to return the system to a valid state.

With the IIC, the `ipending` register (`ctl4`) is masked to all zeros during single-stepping. This masking prevents the processor from servicing interrupts that are asserted while you single-step through code. As a result, if you try to single-step through a part of the exception handling system that reads the `ipending` register, such as `alt_irq_entry()` or `alt_irq_handler()`, the code does not detect any pending interrupts. This issue does not affect debugging software exceptions. You can set breakpoints in your ISR code (and single-step through it), because the interrupt funnel has already used `ipending` to determine which device caused the hardware interrupt.

HAL Exception Handling System Implementation

This section describes the HAL exception handling system implementation. This is one of many possible implementations of an exception handling system for the Nios II processor. Some features of the HAL exception handling system are constrained by the Nios II hardware, while others provide generally useful services.

You can take advantage of the HAL exception handling system without a complete understanding of the HAL implementation. For details about how to install ISRs using the HAL API, refer to "Interrupt Service Routines" on page 8–7.

Exception Handling System Structure

The exception handling system consists of the following components:

- The general exception funnel
- The software exception funnel
- The hardware interrupt funnel(s)
- An ISR for each peripheral that generates hardware interrupts

With the IIC, there is a single hardware interrupt funnel. This funnel manages processor context switch and RTOS overhead (if any). It determines the source of the IRQ, and dispatches the correct ISR.

With an EIC, hardware interrupt funnels are configured by the EIC driver. With a vectored EIC, such as the Altera VIC, there are multiple hardware interrupt funnels. Each funnel manages processor context switch if necessary, and RTOS overhead if any. ISR dispatch is managed by hardware.

With the IIC, when the Nios II processor generates an exception, the general exception funnel receives control. The general exception funnel passes control to either the hardware interrupt funnel or the software exception funnel. The hardware interrupt funnel passes control to one or more ISRs.

Each time an exception occurs, the exception handling system services either a software exception or hardware interrupts, with hardware interrupts having a higher priority. The HAL IIC support does not include nested exceptions, but can handle multiple hardware interrupts per context switch. For details, refer to "Hardware Interrupt Funnel" on page 8–28.

With an EIC, the general exception funnel handles only software exceptions. An IRQ causes the processor to transfer control to one of the interrupt funnels, which branches directly to the ISR.

General Exception Funnel

The general exception funnel provided with the HAL is located at the Nios II processor's exception address. When a software exception or internal hardware interrupt occurs, and control transfers to the general exception funnel, it does the following:

1. Switches to the separate exception stack (if enabled)
2. Stores register values onto the stack
3. Determines the type of exception, and passes control to the software exception funnel or the hardware interrupt funnel

Hardware Interrupt Dispatch with the Internal Interrupt Controller

With the IIC, the general exception funnel dispatches hardware interrupts as well as software exceptions. Figure 8–1 shows the algorithm that the HAL general exception funnel uses to distinguish between hardware interrupts and software exceptions.

The general exception funnel looks at the `estatus` register to determine the interrupt enable status. If the `PIE` bit is set, hardware interrupts were enabled at the time the exception happened. If so, the general exception funnel transfers control to the hardware interrupt funnel. The hardware interrupt funnel looks at the IRQ bits in `ipending`. If any IRQs are asserted, the interrupt funnel calls the appropriate hardware interrupt handler.

If hardware interrupts are not enabled at the time of the exception, it is not necessary to look at `ipending`.

If no IRQs are active, there is no hardware interrupt, and the exception is a software exception. In this case, the general exception funnel calls the software exception funnel.

All hardware interrupts are higher priority than software exceptions.

With an EIC, IRQs are dispatched by hardware. The HAL general exception funnel only handles software exceptions.

For details about the Nios II processor `estatus` and `ipending` registers, refer to the *Programming Model* chapter of the *Nios II Processor Reference Handbook*.

Returning from Exceptions

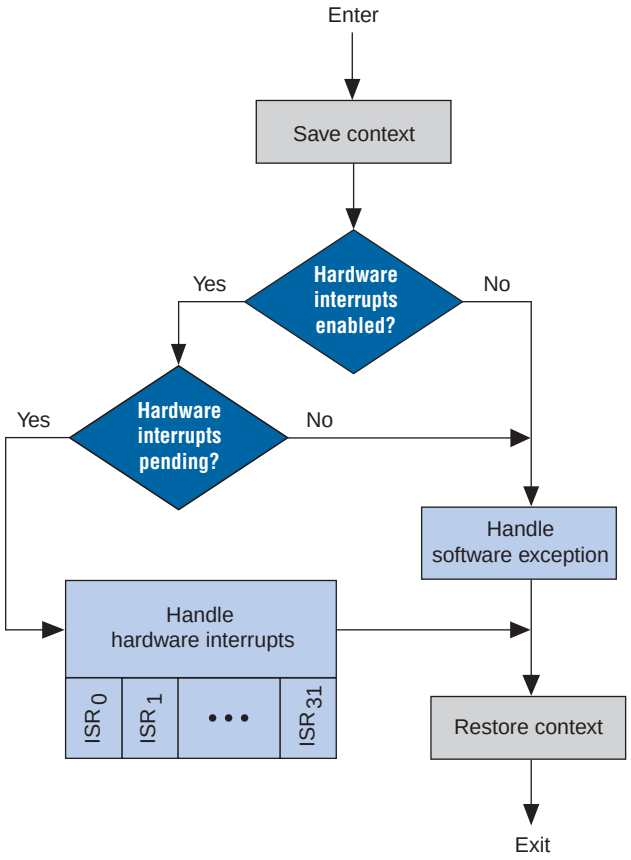
After returning from the ISR or software exception handler, the general exception funnel performs the following tasks:

1. Restores the stack pointer, if a separate exception stack is used
2. Restores the registers from the stack
3. Exits by issuing an `eret` (exception return) instruction

Hardware Interrupt Funnel

The configuration of the HAL hardware interrupt funnel depends on the interrupt controller implemented in the Nios II processor core.

Figure 8–1. HAL Exception Handling System with the Internal Interrupt Controller



Interrupt Funnel for the Internal Interrupt Controller

With the IIC, the Nios II processor supports 32 hardware interrupts. In the HAL funnel, hardware interrupt 0 has the highest priority, and 31 the lowest. This prioritization is a feature of the HAL funnel, and is not inherent in the Nios II interrupt controller.

The hardware interrupt funnel calls the user-registered ISRs. It goes through the IRQs in `ipending` starting at 0, and finds the first (highest priority) active IRQ. Then it calls the corresponding registered ISR. After this ISR executes, the funnel begins scanning the IRQs again, starting at IRQ_0. In this way, higher-priority interrupts are always processed before lower-priority interrupts. When all IRQs are clear, the hardware interrupt funnel returns to the top level. Figure 8–2 shows a flow diagram of the HAL hardware interrupt funnel.

When the interrupt vector custom instruction is present in the Nios II processor, the HAL source detects it at compile time and generates code using the custom instruction. For further information, refer to "Using the Interrupt Vector Custom Instruction" on page 8–25.

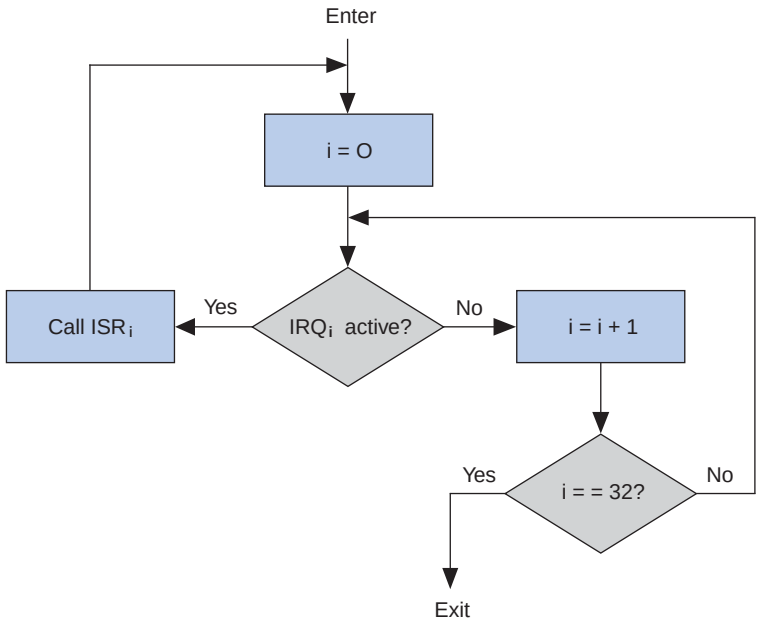
Interrupt Funnels for External Interrupt Controllers

With the EIC interface, the Nios II processor supports a potentially unlimited number of hardware interrupts on daisy-chained EICs. The interrupt priority level can be software-configurable. Details of setting interrupt priorities depend on the particular EIC implementation. The hardware ensures that the highest-priority interrupt is always serviced first.

You register ISRs at system initialization time. Interrupt dispatch is handled by hardware.

For details, refer to "Exception Handling System Structure" on page 8–27.

Figure 8–2. HAL Hardware Interrupt Funnel for the Internal Interrupt Controller



The HAL provides the following interrupt funnels:

- Shadow register set, pre-emption disabled
- Shadow register set, pre-emption enabled
- Nonmaskable interrupt

For details, refer to "Using Interrupt Funnels" on page 8–12.

Software Exception Funnel

Software exceptions can include unimplemented instructions, traps, and miscellaneous exceptions.

Software exception handling depends on options selected in the BSP. If you have enabled unimplemented instruction emulation, the software exception funnel first checks whether an unimplemented instruction caused the exception. If so, it emulates the instruction. Otherwise, it handles traps and miscellaneous exceptions.

Unimplemented Instructions

You can include a handler to emulate unimplemented instructions. The Nios II processor architecture defines the following implementation-dependent instructions:

- `mul`
- `muli`
- `mulxss`
- `mulxsu`
- `mulxuu`
- `div`
- `divu`

For details about unimplemented instructions, refer to "Unimplemented Instructions" in the *Processor Architecture* chapter of the *Nios II Processor Reference Handbook*.

Unimplemented instructions are different from invalid instructions, which are described in "Invalid Instructions" on page 8–33.

When to Use the Unimplemented Instruction Handler

You do not normally need the unimplemented instruction handler, because the HAL includes software emulation for unimplemented instructions from its run-time libraries if you are compiling for a Nios II processor that does not support the instructions.

You might need the unimplemented instruction handler under the following circumstances:

- You are running a Nios II program on an implementation of the Nios II processor other than the one you compiled for. The best solution is to build your program for the correct Nios II processor implementation. Resort to the unimplemented instruction handler only if it is not possible to determine the processor implementation at compile time.
- You have assembly language code that uses an implementation-dependent instruction.

Figure 8–3 shows a flowchart of the HAL software exception funnel, including the optional instruction emulation logic. If instruction emulation is not enabled, this logic is omitted.

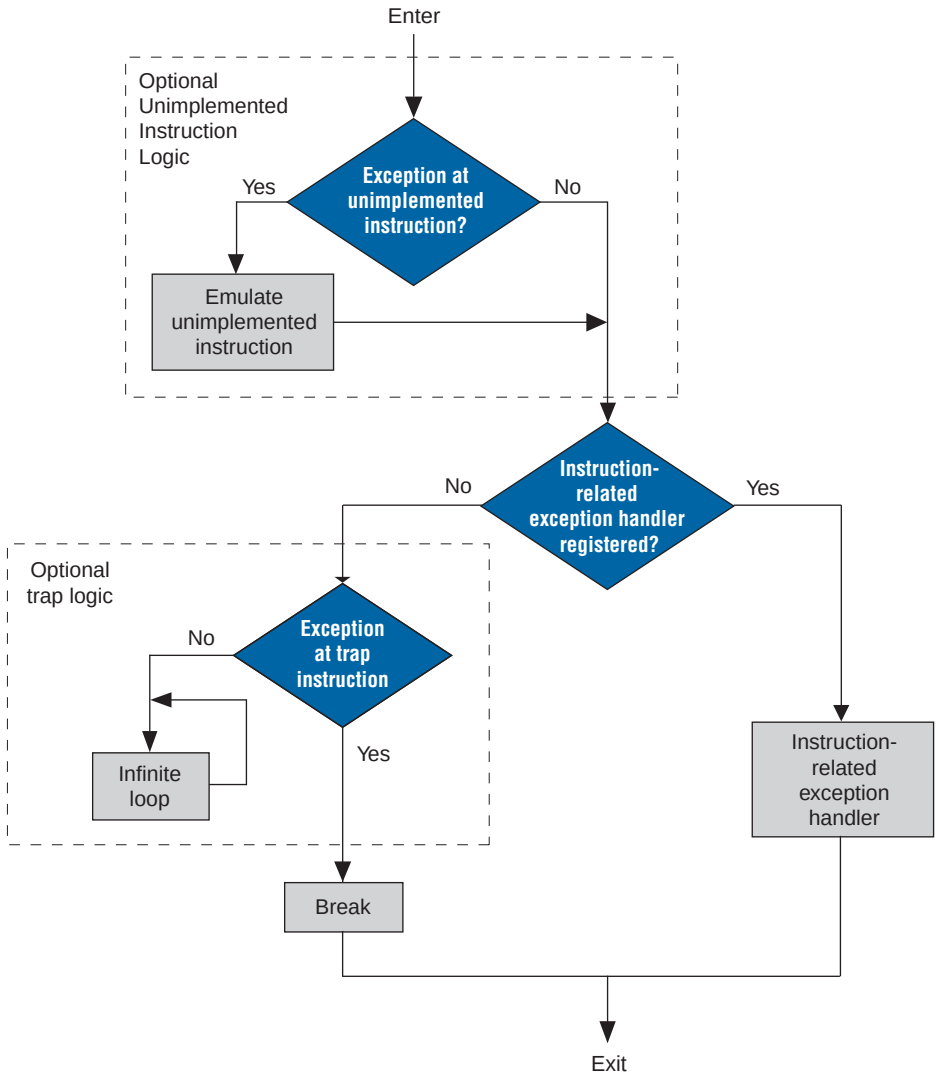
If unimplemented instruction emulation is disabled, but the processor encounters an unimplemented instruction, the software exception funnel treats the exception as a miscellaneous exception. Miscellaneous exceptions are described in "Miscellaneous Exceptions" on page 8–33.

Using the Unimplemented Instruction Handler

To include the unimplemented instruction handler, turn on the `hal.enable_mul_div_emulation` BSP property. The emulation routines occupy less than ¾ kilobyte(KB) of memory.

An exception handler must never execute an unimplemented instruction. The HAL exception handling system does not support nested software exceptions.

Figure 8–3. HAL Software Exception Funnel



Instruction-Related Exceptions

If the cause of the software exception is not an unimplemented instruction, the HAL software exception funnel checks for a registered instruction-related exception handler. If no instruction-related exception handler is registered, the exception is handled as described in "Software Trap Handling". If a handler is registered, the HAL software exception funnel calls it, then restores context and returns. Refer to "The Instruction-Related Exception Handler" for a description of the instruction-related exception handler and how to register it.

Software Trap Handling

If no instruction-related exception handler is registered, the HAL software exception funnel checks for a `trap` instruction. If the exception is caused by a `trap` instruction, the trap exception handler executes a `break` instruction. The `break` instruction transfers control to a hardware debug core, if one is available. If the exception is not caused by a `trap` instruction, it is treated as a miscellaneous exception.

Miscellaneous Exceptions

If the software exception is not caused by an unimplemented instruction or a trap, it is a miscellaneous exception.

If a debug core is present in the Nios II processor, traps and miscellaneous exceptions are handled identically, by executing a `break` instruction. Figure 8–3 shows a flowchart of the HAL software exception funnel, including the optional trap logic. If a debug core is present in the Nios II processor, the trap logic is omitted.

In a debugging environment, the processor executes a `break`, allowing the debugger to take control. In a nondebugging environment, the processor enters an infinite loop.

For details about the Nios II processor `break` instruction, refer to the *Programming Model* and *Instruction Set Reference* chapters of the *Nios II Processor Reference Handbook*.

Miscellaneous exceptions can occur for these reasons:

- Advanced exceptions, the memory protection unit (MPU), or the memory management unit (MMU) are implemented in the Nios II processor core. To handle advanced and MPU exceptions, refer to "The Instruction-Related Exception Handler". To handle MMU exceptions, you need to implement a full-featured operating system, as mentioned in the *Programming Model* chapter of the *Nios II Processor Reference Handbook*.
- You need to include the unimplemented instruction handler, discussed in "Unimplemented Instructions" on page 8–31.
- A peripheral is generating spurious hardware interrupts. This is a symptom of a serious hardware problem. A peripheral might generate spurious hardware interrupts if it deasserts its interrupt output before an ISR has explicitly serviced it.

Invalid Instructions

An invalid instruction word contains invalid codes in the OP or OPX field. For normal Nios II core implementations, the result of executing an invalid instruction is undefined; processor behavior is dependent on the Nios II core.

Therefore, the software exception funnel cannot detect or respond to an invalid instruction.

Invalid instructions are different from unimplemented instructions, which are described in "Unimplemented Instructions" on page 8–31.

For more information, refer to the *Nios II Core Implementation Details* chapter of the *Nios II Processor Reference Handbook*.

The Instruction-Related Exception Handler

The software exception funnel lets you handle instruction-related exceptions, such as the advanced exceptions. The instruction-related exception handler is a custom handler. Your software registers the instruction-related exception handler with the HAL at startup time.

The `hal.enable_instruction_related_exceptions_api` setting must be enabled in the BSP in order for you to register an instruction-related exception handler.

For further information about the Nios II instruction-related exceptions, refer to the *Programming Model* chapter of the *Nios II Processor Reference Handbook*. For details about enabling instruction-related exception handlers, refer to "Settings" in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

When you register an instruction-related exception handler, it takes the place of the break/optional trap logic.

When you remove the instruction-related exception handler, the HAL restores the default break/optional trap logic.

Writing an Instruction-Related Exception Handler

The prototype for an instruction-related exception handler is as follows:

```
alt_exception_result handler (
  alt_exception_cause  cause,
  alt_u32              addr,
  alt_u32               bad_addr );
```

The instruction-related exception handler's return value is a flag requesting that the HAL either re-execute the instruction, or skip it.

The HAL exception funnel calls the instruction-related exception handler with the following arguments:

- `cause`—A value representing the exception type, as shown in Table 8–4
- `addr`—Instruction address at which exception occurred
- `bad_addr`—Bad address register (if implemented)

Include the following header file in your instruction-related exception handler code:

```
#include "sys/alt_exceptions.h"
```

alt_exceptions.h provides type macro definitions required to interface your instruction-related exception handler to the HAL, including the cause codes shown in Table 8–4.

The API function `alt_exception_cause_generated_bad_addr()` is provided by the HAL, for the use of the instruction-related exception handler. This function parses the `cause` argument and determines if `bad_addr` contains the exception-causing address.

For further information about Nios II processor exception causes, refer to "Exception Processing" in the *Programming Model* chapter of the *Nios II Processor Reference Handbook*.

Table 8–4. Nios II Exception Cause Codes

Exception	Cause Code	Cause Symbol *(1)*
Reset	0	NIOS2_EXCEPTION_RESET
Processor-only Reset Request	1	NIOS2_EXCEPTION_CPU_ONLY_RESET_REQUEST
Hardware Interrupt	2	NIOS2_EXCEPTION_INTERRUPT
Trap Instruction	3	NIOS2_EXCEPTION_TRAP_INST
Unimplemented Instruction	4	NIOS2_EXCEPTION_UNIMPLEMENTED_INST
Illegal Instruction	5	NIOS2_EXCEPTION_ILLEGAL_INST
Misaligned Data Address	6	NIOS2_EXCEPTION_MISALIGNED_DATA_ADDR
Misaligned Destination Address	7	NIOS2_EXCEPTION_MISALIGNED_TARGET_PC
Division Error	8	NIOS2_EXCEPTION_DIVISION_ERROR
Supervisor-only Instruction Address	9	NIOS2_EXCEPTION_SUPERVISOR_ONLY_INST_ADDR
Supervisor-only Instruction	10	NIOS2_EXCEPTION_SUPERVISOR_ONLY_INST
Supervisor-only Data Address	11	NIOS2_EXCEPTION_SUPERVISOR_ONLY_DATA_ADDR
Translation lookaside buffer (TLB) Miss	12	NIOS2_EXCEPTION_TLB_MISS
TLB Permission Violation (execute)	13	NIOS2_EXCEPTION_TLB_EXECUTE_PERM_VIOLATION
TLB Permission Violation (read)	14	NIOS2_EXCEPTION_TLB_READ_PERM_VIOLATION
TLB Permission Violation (write)	15	NIOS2_EXCEPTION_TLB_WRITE_PERM_VIOLATION
MPU Region Violation (instruction)	16	NIOS2_EXCEPTION_MPU_INST_REGION_VIOLATION
MPU Region Violation (data)	17	NIOS2_EXCEPTION_MPU_DATA_REGION_VIOLATION
Cause unknown *(2)*	-1	NIOS2_EXCEPTION_CAUSE_NOT_PRESENT

Notes to Table 8–4:

(1) Cause symbols are defined in **sys/alt_exceptions.h**.

(2) This value is passed to the instruction-related exception handler if the `cause` argument if the cause is not known; for example, if the `cause` register not implemented in the Nios II processor core.

If there is an instruction-related exception handler, it is called at the end of the software exception funnel (if the funnel has not recognized a hardware interrupt, unimplemented instruction or trap). It takes the place of the break or infinite loop. Therefore, to support debugging, execute a break on a trap instruction.

It is possible for an instruction-related exception to occur during execution of an ISR.

Registering an Instruction-Related Exception Handler

The HAL API function `alt_instruction_exception_register()` registers a single instruction-related exception handler.

The function prototype is as follows:

```
alt_instruction_exception_register (
    alt_exception_result (*handler)
        ( alt_exception_cause, alt_u32, alt_u32 ));
```

The `handler` argument is a pointer to the instruction-related exception handler.

To use `alt_instruction_exception_register()`, include the following header file:

```
#include "sys/alt_exceptions.h"
```

The `hal.enable_instruction_related_exceptions_api` setting must be enabled in the BSP in order for you to register an instruction-related exception handler.

For details, refer to "Settings" in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Register the instruction-related exception handler as early as possible in function `main()`. This allows you to handle abnormal condition during startup. You register an exception handler from the `alt_main()` function.

For more information about `alt_main()`, refer to "Boot Sequence and Entry Point" in the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

Removing an Instruction-Related Exception Handler

To remove a registered instruction-related exception handler, your C code must call the `alt_instruction_exception_register()` function, as follows:

```
alt_instruction_exception_register ( null, null );
```

When the HAL removes the instruction-related exception handler, it restores the default break/optional trap logic.

Referenced Documents

This chapter references the following documents:

- *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*
- *Cache and Tightly-Coupled Memory* chapter of the *Nios II Software Developer's Handbook*
- *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*
- *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*
- *Processor Architecture* chapter of the *Nios II Processor Reference Handbook*
- *Programming Model* chapter of the *Nios II Processor Reference Handbook*
- *Instantiating the Nios II Processor in SOPC Builder* chapter of the *Nios II Processor Reference Handbook*
- *Nios II Core Implementation Details* chapter of the *Nios II Processor Reference Handbook*
- *Instruction Set Reference* chapter of the *Nios II Processor Reference Handbook*

- *Embedded Peripherals IP User Guide*
- *Using Nios II Tightly Coupled Memory Tutorial*
- *AN595: Vectored Interrupt Controller Applications and Usage*

Document Revision History

Table 8–5 shows the revision history for this document.

Table 8–5. Document Revision History (Part 1 of 2)

Date & Document Version	Changes Made	Summary of Changes
July 2010 v10.0.0	Maintenance release	
November 2009 v9.1.0	■ Described HAL support for external interrupt controller interface ■ Described HAL support for shadow register sets with external interrupt controller interface ■ Described enhanced HAL interrupt API ■ Removed information specific to the Nios II Integrated Development Environment (IDE)	■ External interrupt controller interface ■ Enhanced HAL interrupt API ■ Legacy HAL interrupt API deprecated
March 2009 v9.0.0	■ Reorganized and updated information and terminology to clarify role of Nios II Software Build Tools. ■ Corrected minor typographical errors.	
May 2008 v8.0.0	Maintenance release	
October 2007 v7.2.0	Maintenance release	
May 2007 v7.1.0	■ Added table of contents to "Introduction" section. ■ Added Referenced Documents section.	
March 2007 v7.0.0	Maintenance release	
November 2006 v6.1.0	■ Describes support for the interrupt vector custom instruction.	Interrupt vector custom instruction added.
May 2006 v6.0.0	■ Corrected error in `alt_irq_enable_all()` usage ■ Added illustrations ■ Revised text on optimizing ISRs ■ Expanded and revised text discussing HAL exception handler code structure.	
October 2005 v5.1.0	■ Updated references to HAL exception-handler assembly source files in section "HAL Exception Handler Files". ■ Added description of `alt_irq_disable()` and `alt_irq_enable()` in section "Interrupt Service Routines".	
May 2005 v5.0.0	Added tightly-coupled memory information.	

Table 8–5. Document Revision History (Part 2 of 2)

Date & Document Version	Changes Made	Summary of Changes
December 2004 v1.2	Corrected the “Registering the Button PIO ISR with the HAL” example.	
September 2004 v1.1	■ Changed examples. ■ Added ISR performance data.	
May 2004 v1.0	Initial release	