

This section provides information about several advanced programming topics. It includes the following chapters:

- Chapter 8, Exception Handling
- Chapter 9, Cache and Tightly-Coupled Memory
- Chapter 10, MicroC/OS-II Real-Time Operating System
- Chapter 11, Ethernet and the NicheStack TCP/IP Stack - Nios II Edition
- Chapter 12, Read-Only Zip File System
- Chapter 13, Publishing Component Information to Embedded Software



## 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, either in Eclipse™ or on the command line.

This chapter contains the following sections:

- “Introduction” on page 8–1
- “Nios II Exception Handling Overview” on page 8–1
- “Interrupt Service Routines” on page 8–7
- “Improving ISR Performance” on page 8–18
- “Debugging ISRs” on page 8–25
- “HAL Exception Handling System Implementation” on page 8–26
- “The Instruction-Related Exception Handler” on page 8–32
- “Referenced Documents” on page 8–35



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 8-32.
- 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*. For details about the VIC, refer to the *Vectored Interrupt Controller Core* chapter in *Volume 5: Embedded Peripherals* of the *Quartus® II 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).

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 *Volume 5: Embedded Peripherals of the Quartus II Handbook* 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 *Volume 5: Embedded Peripherals of the Quartus II Handbook*.



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 *Volume 5: Embedded Peripherals* of the *Quartus II Handbook*.

### Selecting an Interrupt API

When the Software Build Tools create a BSP, they determine whether the BSP must implement the legacy interrupt API. Each driver that supports the enhanced API publishes this capability to the Software Build Tools 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 Software Build Tools ignore 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 Software Build Tools assume 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 Software Build Tools determine whether the legacy API is required, and implement 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
<code>alt_ic_isr_register()</code>	Interrupt controller driver (1)
<code>alt_ic_irq_enable()</code>	Interrupt controller driver (1)
<code>alt_ic_irq_disable()</code>	Interrupt controller driver (1)
<code>alt_ic_irq_enabled()</code>	Interrupt controller driver (1)
<code>alt_irq_disable_all()</code>	HAL
<code>alt_irq_enable_all()</code>	HAL
<code>alt_irq_interruptible()</code>	HAL
<code>alt_irq_enabled()</code>	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 Software Build Tools insert 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 *Volume 5: Embedded Peripherals* of the *Quartus II Handbook*.

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 Software Build Tools 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 Software Build Tools select the interrupt API, they define 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 *Volume 5: Embedded Peripherals* of the *Quartus II Handbook*.

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 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 Software Build Tools check 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 Software Build Tools assume 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, `IRQ0` represents the highest priority interrupt and `IRQ31` 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 Funnel” 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 *Volume 5: Embedded Peripherals* of the *Quartus II Handbook*. 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 8-19.

 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 Software Build Tools create 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 *Volume 5: Embedded Peripherals* of the *Quartus II Handbook*.

## 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
<code>alt_irq_register()</code>	<code>alt_ic_isr_register()</code>
<code>alt_irq_disable()</code>	<code>alt_ic_irq_disable()</code>
<code>alt_irq_enable()</code>	<code>alt_ic_irq_enable()</code>



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 8-23](#).

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		Application Stack	General Exception Stack	Hardware Interrupt Stack
	Separate General Exception Stack Enabled	Separate Hardware Interrupt Stack Enabled			
Internal	No	—	<ul style="list-style-type: none"> <li>■ Application</li> <li>■ Software exceptions</li> <li>■ Hardware interrupts</li> </ul>	—	—
	Yes	—	Application	<ul style="list-style-type: none"> <li>■ Software exceptions</li> <li>■ Hardware interrupts</li> </ul>	—
External	No	No	<ul style="list-style-type: none"> <li>■ Application</li> <li>■ Software exceptions</li> <li>■ Hardware interrupts</li> </ul>	—	—
		Yes	<ul style="list-style-type: none"> <li>■ Application</li> <li>■ Software exceptions</li> </ul>	—	Hardware interrupts
	Yes	No	<ul style="list-style-type: none"> <li>■ Application</li> <li>■ Hardware interrupts</li> </ul>	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 Software Build Tools assume 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 8-18](#).

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 *Volume 5: Embedded Peripherals* of the *Quartus II Handbook*.

### 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 *Volume 5: Embedded Peripherals* of the *Quartus II Handbook*.

### 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<sub>0</sub> 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 *Volume 5: Embedded Peripherals of the Quartus II Handbook*.

## 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 (`ctl14`) 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 8-27](#).

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*.

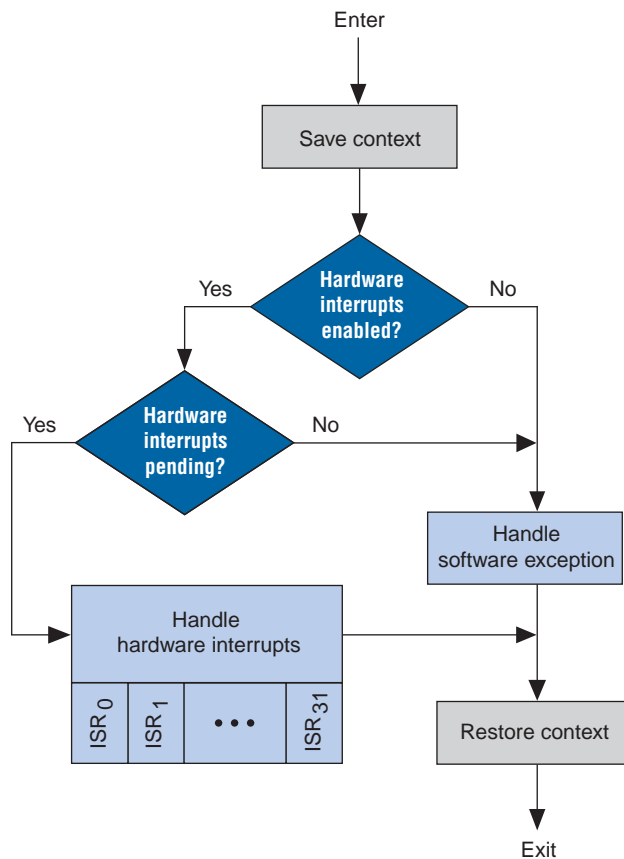
### Returning from Exceptions

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<sub>0</sub>. 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 8-24.

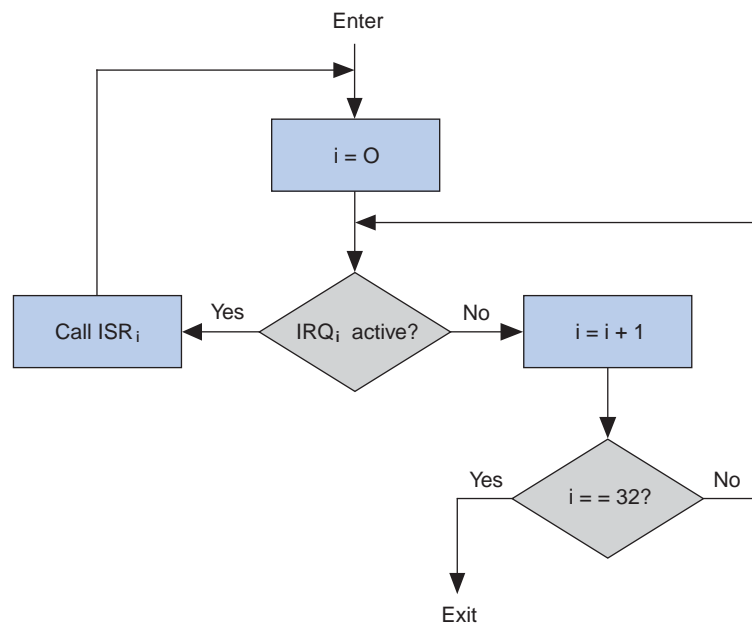
## Interrupt Funnels for External Interrupt Controllers

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 8-26.

**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 8-32.

### 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. Only if this is not possible should you resort to the unimplemented instruction handler.
- 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 8-32.

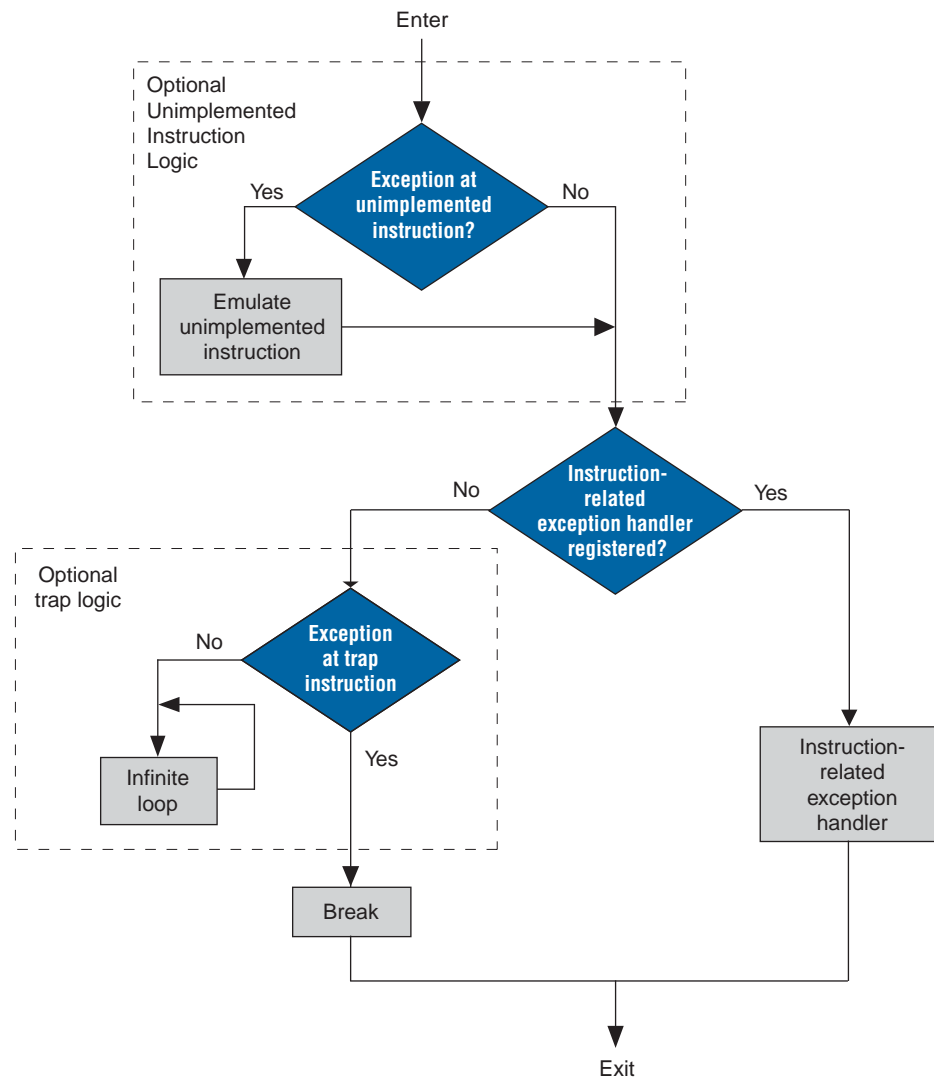
### Using the Unimplemented Instruction Handler

To include the unimplemented instruction handler, turn on the `hal.enable_mul_div_emulation` BSP property. The emulation routines occupy less than  $\frac{3}{4}$  KBytes 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 8-30.
- 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 8-30.

 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*
- *Volume 5: Embedded Peripherals* of the *Quartus II Handbook*

- [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
November 2009 9.1.0	<ul style="list-style-type: none"> <li>■ Described HAL support for external interrupt controller interface</li> <li>■ Described HAL support for shadow register sets with external interrupt controller interface</li> <li>■ Described enhanced HAL interrupt API</li> <li>■ Removed information specific to the Nios II IDE</li> </ul>	<ul style="list-style-type: none"> <li>■ External interrupt controller interface</li> <li>■ Enhanced HAL interrupt API</li> <li>■ Legacy HAL interrupt API deprecated</li> </ul>
March 2009 v9.0.0	<ul style="list-style-type: none"> <li>■ Reorganized and updated information and terminology to clarify role of Nios II Software Build Tools.</li> <li>■ Corrected minor typographical errors.</li> </ul>	
May 2008 v8.0.0	No change from previous release.	
October 2007 v7.2.0	No change from previous release.	
May 2007 v7.1.0	<ul style="list-style-type: none"> <li>■ Added table of contents to “Introduction” section.</li> <li>■ Added Referenced Documents section.</li> </ul>	
March 2007 v7.0.0	No change from previous release.	
November 2006 v6.1.0	<ul style="list-style-type: none"> <li>■ Describes support for the interrupt vector custom instruction.</li> </ul>	Interrupt vector custom instruction added.
May 2006 v6.0.0	<ul style="list-style-type: none"> <li>■ Corrected error in <code>alt_irq_enable_all()</code> usage</li> <li>■ Added illustrations</li> <li>■ Revised text on optimizing ISRs</li> <li>■ Expanded and revised text discussing HAL exception handler code structure.</li> </ul>	
October 2005 v5.1.0	<ul style="list-style-type: none"> <li>■ Updated references to HAL exception-handler assembly source files in section “HAL Exception Handler Files”.</li> <li>■ Added description of <code>alt_irq_disable()</code> and <code>alt_irq_enable()</code> in section “Interrupt Service Routines”.</li> </ul>	
May 2005 v5.0.0	Added tightly-coupled memory information.	
December 2004 v1.2	Corrected the “Registering the Button PIO ISR with the HAL” example.	

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

<b>Date &amp; Document Version</b>	<b>Changes Made</b>	<b>Summary of Changes</b>
September 2004 v1.1	<ul style="list-style-type: none"><li>■ Changed examples.</li><li>■ Added ISR performance data.</li></ul>	
May 2004 v1.0	Initial Release.	



## Introduction

Nios® II processor cores may contain instruction and data caches. This chapter discusses cache-related issues that you need to consider to guarantee that your program executes correctly on the Nios II processor. Fortunately, most software based on the Nios II hardware abstraction layer (HAL) works correctly without any special accommodations for caches. However, some software must manage the cache directly. For code that needs direct control over the cache, the Nios II architecture provides facilities to perform the following actions:

- Initialize lines in the instruction and data caches
- Flush lines in the instruction and data caches
- Bypass the data cache during load and store instructions

This chapter discusses the following common cases in which you must manage the cache:

- Initializing cache after reset
- Writing device drivers
- Writing program loaders or self-modifying code
- Managing cache in multi-master or multi-processor systems

This chapter contains the following sections:

- [“Initializing Cache after Reset” on page 9–2](#)
- [“Writing Device Drivers” on page 9–4](#)
- [“Writing Program Loaders or Self-Modifying Code” on page 9–5](#)
- [“Managing Cache in Multi-Master/ Multi-Processor Systems” on page 9–5](#)
- [“Tightly-Coupled Memory” on page 9–7](#)

## Nios II Cache Implementation

Depending on the Nios II core implementation, a Nios II processor system may or may not have data or instruction caches. You can write programs generically so that they function correctly on any Nios II processor, regardless of whether it has cache memory. For a Nios II core without one or both caches, cache management operations are benign and have no effect.

The current Nios II cores have no hardware cache coherency mechanism. Therefore, if multiple masters can access shared memory, software must explicitly maintain coherency across all masters.



For complete details about the features of each Nios II core implementation, refer to the [Nios II Core Implementation Details](#) chapter of the *Nios II Processor Reference Handbook*.

The details for a particular Nios II processor system are defined in the **system.h** file. **Example 9-1** shows an excerpt from the **system.h** file, defining the cache properties, such as cache size and the size of a single cache line.

**Example 9-1.** An Excerpt from system.h that Defines the Cache Structure

```
#define NIOS2_ICACHE_SIZE 4096
#define NIOS2_DCACHE_SIZE 0
#define NIOS2_ICACHE_LINE_SIZE 32
#define NIOS2_DCACHE_LINE_SIZE 0
```

This system has a 4 Kbyte instruction cache with 32 byte lines, and no data cache.

## HAL API Functions for Managing Cache

The HAL application program interface (API) provides the following functions for managing cache memory:

- alt\_dcachel\_flush()
- alt\_dcachel\_flush\_all()
- alt\_icachel\_flush()
- alt\_icachel\_flush\_all()
- alt\_uncached\_malloc()
- alt\_uncached\_free()
- alt\_remap\_uncached()
- alt\_remap\_cached()



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

## Further Information

This chapter covers only cache management issues that affect Nios II programmers. It does not discuss the fundamental operation of caches. Refer to *The Cache Memory Book* by Jim Handy for a discussion of general cache management issues.

## Initializing Cache after Reset

After reset, the contents of the instruction cache and data cache are unknown. They must be initialized at the start of the software reset handler for correct operation.

The Nios II caches cannot be disabled by software; they are always enabled. To allow proper operation, a processor reset causes the instruction cache to invalidate the one instruction cache line that corresponds to the reset handler address. This forces the instruction cache to fetch instructions corresponding to this cache line from memory. The reset handler address must be aligned to the size of the instruction cache line.

It is the responsibility of the first eight instructions of the reset handler to initialize the remainder of the instruction cache. The Nios II `init_i` instruction initializes a single instruction cache line. Do not use the `flush_i` instruction because it may cause undesired effects when used to initialize the instruction cache in future Nios II implementations.

Place the `init_i` instruction in a loop that executes `init_i` for each instruction cache line address. [Example 9-2](#) shows an example of assembly code to initialize the instruction cache.

---

**Example 9-2.** Assembly Code to Initialize the Instruction Cache

```

mov     r4, r0
movhi   r5, %hi(NIOS2_ICACHE_SIZE)
ori     r5, r5, %lo(NIOS2_ICACHE_SIZE)
icache_init_loop:
init_i  r4
addi    r4, r4, NIOS2_ICACHE_LINE_SIZE
bltu   r4, r5, icache_init_loop

```

---

After the instruction cache is initialized, the data cache must also be initialized. The Nios II `init_d` instruction initializes a single data cache line. Do not use the `flush_d` instruction for this purpose, because it writes dirty lines back to memory. The data cache is undefined after reset, including the cache line tags. Using `flush_d` can cause unexpected writes of random data to random addresses. The `init_d` instruction does not write back dirty data.

Place the `init_d` instruction in a loop that executes `init_d` for each data cache line address. [Example 9-3](#) shows an example of assembly code to initialize the data cache:

---

**Example 9-3.** Assembly Code to Initialize the Data Cache

```

mov     r4, r0
movhi   r5, %hi(NIOS2_DCACHE_SIZE)
ori     r5, r5, %lo(NIOS2_DCACHE_SIZE)
dcache_init_loop:
init_d  0(r4)
addi    r4, r4, NIOS2_DCACHE_LINE_SIZE
bltu   r4, r5, dcache_init_loop

```

---

It is legal to execute instruction and data cache initialization code on Nios II cores that do not implement one or both of the caches. The `init_i` and `init_d` instructions are simply treated as `nop` instructions if there is no cache of the corresponding type present.

## For HAL Users

Programs based on the HAL need not manage the initialization of cache memory. The HAL C run-time code (`crt0.S`) provides a default reset handler that performs cache initialization before `alt_main()` or `main()` is called.

## Writing Device Drivers

Device drivers typically access control registers associated with their device. These registers are mapped into the Nios II address space. When accessing device registers, the data cache must be bypassed to ensure that accesses are not lost or deferred due to the data cache.

When writing a device driver, bypass the data cache with the `ldio/stio` family of instructions. On Nios II cores without a data cache, these instructions behave just like their corresponding `ld/st` instructions, and therefore are benign.



Declaring a C pointer `volatile` does not make pointer accesses bypass the data cache. The `volatile` keyword merely prevents the compiler from optimizing out accesses using the pointer. This `volatile` behavior is different from the methodology for the first-generation Nios processor.

### For HAL Users

The HAL provides the C-language macros `IORD` and `IOWR` that expand to the appropriate assembly instructions to bypass the data cache. The `IORD` macro expands to the `ldwio` instruction, and the `IOWR` macro expands to the `stwio` instruction. These macros should be used by HAL device drivers to access device registers.

Table 9-1 shows the available macros. All of these macros bypass the data cache when they perform their operation. In general, your program passes values defined in `system.h` as the `BASE` and `REGNUM` parameters. These macros are defined in the file `<Nios II EDS install path>/components/altera_nios2/HAL/inc/io.h`.

**Table 9-1.** HAL I/O Macros to Bypass the Data Cache

Macro	Use
<code>IORD(BASE, REGNUM)</code>	Read the value of the register at offset <code>REGNUM</code> in a device with base address <code>BASE</code> . Registers are assumed to be offset by the address width of the bus.
<code>IOWR(BASE, REGNUM, DATA)</code>	Write the value <code>DATA</code> to the register at offset <code>REGNUM</code> in a device with base address <code>BASE</code> . Registers are assumed to be offset by the address width of the bus.
<code>IORD_32DIRECT(BASE, OFFSET)</code>	Make a 32-bit read access at the location with address <code>BASE+OFFSET</code> .
<code>IORD_16DIRECT(BASE, OFFSET)</code>	Make a 16-bit read access at the location with address <code>BASE+OFFSET</code> .
<code>IORD_8DIRECT(BASE, OFFSET)</code>	Make an 8-bit read access at the location with address <code>BASE+OFFSET</code> .
<code>IOWR_32DIRECT(BASE, OFFSET, DATA)</code>	Make a 32-bit write access to write the value <code>DATA</code> at the location with address <code>BASE+OFFSET</code> .
<code>IOWR_16DIRECT(BASE, OFFSET, DATA)</code>	Make a 16-bit write access to write the value <code>DATA</code> at the location with address <code>BASE+OFFSET</code> .
<code>IOWR_8DIRECT(BASE, OFFSET, DATA)</code>	Make an 8-bit write access to write the value <code>DATA</code> at the location with address <code>BASE+OFFSET</code> .

## Writing Program Loaders or Self-Modifying Code

Software that writes instructions to memory, such as program loaders or self-modifying code, needs to ensure that old instructions are flushed from the instruction cache and processor pipeline. This flushing is accomplished with the `flushi` and `flushp` instructions, respectively. Additionally, if new instruction(s) are written to memory using store instructions that do not bypass the data cache, you must use the `flushd` instruction to flush the new instruction(s) from the data cache to memory.

[Example 9-4](#) shows assembly code that writes a new instruction to memory.

### Example 9-4. Assembly Code That Writes a New Instruction to Memory

```
/*
 * Assume new instruction in r4 and
 * instruction address already in r5.
 */
stw    r4, 0(r5)
flushd 0(r5)
flushi  r5
flushp
```

The `stw` instruction writes the new instruction in `r4` to the instruction address specified by `r5`. If a data cache is present, the instruction is written just to the data cache and the associated line is marked dirty. The `flushd` instruction writes the data cache line associated with the address in `r5` to memory and invalidates the corresponding data cache line. The `flushi` instruction invalidates the instruction cache line associated with the address in `r5`. Finally, the `flushp` instruction ensures that the processor pipeline has not prefetched the old instruction at the address specified by `r5`.

Notice that [Example 9-4](#) uses the `stw/flushd` pair instead of the `stwio` instruction. The `stwio` instruction does not flush the data cache, and therefore might leave stale data in the data cache.

This code sequence is correct for all Nios II implementations. If a Nios II core does not have a particular kind of cache, the corresponding flush instruction (`flushd` or `flushi`) is executed as a `nop`.

## For Users of the HAL

The HAL API does not provide functions for this cache management case.

## Managing Cache in Multi-Master/ Multi-Processor Systems

The Nios II architecture does not provide hardware cache coherency. Instead, software cache coherency must be provided when communicating through shared memory. The data cache contents of all processors accessing the shared memory must be managed by software to ensure that all masters read the most recent values and do not overwrite new data with stale data. This management is done by using the data cache flushing and bypassing facilities to move data between the shared memory and the data cache(s) as needed.

The `flushd` instruction ensures that the data cache and memory contain the same value for one line. If the line contains dirty data, it is written to memory. The line is then invalidated in the data cache.

Consistently bypassing the data cache is very important. The processor does not check if an address is in the data cache when bypassing the data cache. If software cannot guarantee that a particular address is in the data cache, it must flush the address from the data cache before bypassing it for a load or store. This action guarantees that the processor does not bypass new (dirty) data in the cache, and mistakenly access old data in memory.

## Bit-31 Cache Bypass

The `ldio/stio` family of instructions explicitly bypass the data cache. Bit-31 provides an alternate method to bypass the data cache. Using the bit-31 cache bypass, the normal `ld/st` family of instructions may be used to bypass the data cache if the most significant bit of the address (bit 31) is set to one. The value of bit 31 is only used internally to the processor; bit 31 is forced to zero in the actual address accessed. This limits the maximum byte address space to 31 bits.

Using bit 31 to bypass the data cache is a convenient mechanism for software because the cacheability of the associated address is contained in the address. This usage allows the address to be passed to code that uses the normal `ld/st` family of instructions, while still guaranteeing that all accesses to that address consistently bypass the data cache.

Bit-31 cache bypass is only explicitly provided in the Nios II/f core, and should not be used for other Nios II cores. The other Nios II cores that do not support bit-31 cache bypass limit their maximum byte address space to 31 bits to ease migration of code from one implementation to another. They effectively ignore the value of bit 31, which allows code written for a Nios II/f core using bit 31 cache bypass to run correctly on other current Nios II implementations. In general, this feature depends on the Nios II core implementation.



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

## For HAL Users

The HAL provides the C-language `IORD_*DIRECT` macros that expand to the `ldio` family of instructions and the `IOWR_*DIRECT` macros that expand to the `stio` family of instructions. Refer to [Table 9-1 on page 9-4](#). These macros are provided to access noncacheable memory regions.


The HAL provides the `alt_uncached_malloc()`, `alt_uncached_free()`, `alt_remap_uncached()`, and `alt_remap_cached()` routines to allocate and manipulate regions of uncached memory. These routines are available on Nios II cores with or without a data cache—code written for a Nios II core with a data cache is completely compatible with a Nios II core without a data cache.

The `alt_uncached_malloc()` and `alt_remap_uncached()` routines guarantee that the allocated memory region is not in the data cache and that all subsequent accesses to the allocated memory regions bypass the data cache.

## Tightly-Coupled Memory

If you want the performance of cache all the time, place your code or data in a tightly-coupled memory. Tightly-coupled memory is fast on-chip memory that bypasses the cache and has guaranteed low latency. Tightly-coupled memory gives the best memory access performance. You assign code and data to tightly-coupled memory partitions in the same way as other memory sections.

Cache instructions do not affect tightly-coupled memory. However, cache-management instructions become NOPs, which might result in unnecessary overhead.

 For more information, refer to “Memory Usage” in the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer’s Handbook*.

## Referenced Documents

This chapter references the following documents:

- *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer’s Handbook*
- *HAL API Reference* chapter of the *Nios II Software Developer’s Handbook*
- *Nios II Core Implementation Details* chapter of the *Nios II Processor Reference Handbook*

## Document Revision History

Table 9–2 shows the revision history for this document.

**Table 9–2.** Document Revision History (Part 1 of 2)

Date & Document Version	Changes Made	Summary of Changes
November 2009 v9.1.0	No change from previous release.	
March 2009 v9.0.0	<ul style="list-style-type: none"> <li>■ Reorganized and updated information and terminology to clarify role of Nios II Software Build Tools.</li> <li>■ Corrected minor typographical errors.</li> </ul>	
May 2008 v8.0.0	No change from previous release.	
October 2007 v7.2.0	No change from previous release.	
May 2007 v7.1.0	<ul style="list-style-type: none"> <li>■ Added table of contents to “Introduction” section.</li> <li>■ Added Referenced Documents section.</li> </ul>	
March 2007 v7.0.0	No change from previous release.	

**Table 9-2.** Document Revision History (Part 2 of 2)

<b>Date &amp; Document Version</b>	<b>Changes Made</b>	<b>Summary of Changes</b>
November 2006 v6.1.0	No change from previous release.	
May 2006 v6.0.0	No change from previous release.	
October 2005 v5.1.0	Added detail to section “Tightly-Coupled Memory”.	
May 2005 v5.0.0	Added tightly-coupled memory section.	
May 2004 v1.0	Initial Release.	

## Introduction

This chapter describes the MicroC/OS-II real-time kernel for the Nios® II processor. This chapter contains the following sections:

- “Overview” on page 10-1
- “Other RTOS Providers” on page 10-2
- “The Nios II Implementation of MicroC/OS-II ” on page 10-2
- “Implementing MicroC/OS-II Projects for the Nios II Processor” on page 10-6

## Overview

MicroC/OS-II is a popular real-time kernel produced by Micrium Inc. MicroC/OS-II is a portable, ROMable, scalable, pre-emptive, real-time, multitasking kernel. First released in 1992, MicroC/OS-II is used in hundreds of commercial applications. It is implemented on more than 40 different processor architectures in addition to the Nios II processor.

MicroC/OS-II provides the following services:

- Tasks (threads)
- Event flags
- Message passing
- Memory management
- Semaphores
- Time management

The MicroC/OS-II kernel operates on top of the hardware abstraction layer (HAL) board support package (BSP) for the Nios II processor. Because of this architecture, MicroC/OS-II development for the Nios II processor has the following advantages:

- Programs are portable to other Nios II hardware systems.
- Programs are resistant to changes in the underlying hardware.
- Programs can access all HAL services, calling the UNIX-like HAL application program interface (API).
- ISRs are easy to implement.

## Further Information



This chapter discusses the details of how to use MicroC/OS-II for the Nios II processor only. For complete reference of MicroC/OS-II features and usage, refer to *MicroC/OS-II - The Real-Time Kernel* by Jean J. Labrosse (CMP Books). You can obtain further information from Micrium ([www.micrium.com](http://www.micrium.com)).

## Licensing

Altera distributes MicroC/OS-II in the Nios II Embedded Design Suite (EDS) for evaluation purposes only. If you plan to use MicroC/OS-II in a commercial product, you must obtain a license from Micrium ([www.micrium.com](http://www.micrium.com)).



Micrium offers free licensing for universities and students. Contact Micrium for details.

## Other RTOS Providers

Altera distributes MicroC/OS-II to provide you with immediate access to an easy-to-use RTOS. In addition to MicroC/OS-II, many other RTOSs are available from third-party vendors.



For a complete list of RTOSs that support the Nios II processor, visit the [Embedded Software](#) page of the Altera website.

## The Nios II Implementation of MicroC/OS-II

Altera has ported MicroC/OS-II to the Nios II processor. Altera distributes MicroC/OS-II in the Nios II EDS, and supports the Nios II implementation of the MicroC/OS-II kernel. Ready-made, working examples of MicroC/OS-II programs are installed with the Nios II EDS. In addition, Nios development boards are preprogrammed with a web server reference design based on MicroC/OS-II and the NicheStack® TCP/IP Stack - Nios II Edition.

The Altera implementation of MicroC/OS-II is designed to be easy to use. Using the Nios II project settings, you can control the configuration for all the RTOS's modules.

You need not modify source files directly to enable or disable kernel features. Nonetheless, Altera provides the Nios II processor-specific source code in case you wish to examine it. The MicroC/OS-II source code is located in the following directories:

- Processor-specific code: *<Nios II EDS install path>/components/altera\_nios2/UCOSII*
- Processor-independent code: *<Nios II EDS install path>/components/micrium\_uc\_osii*

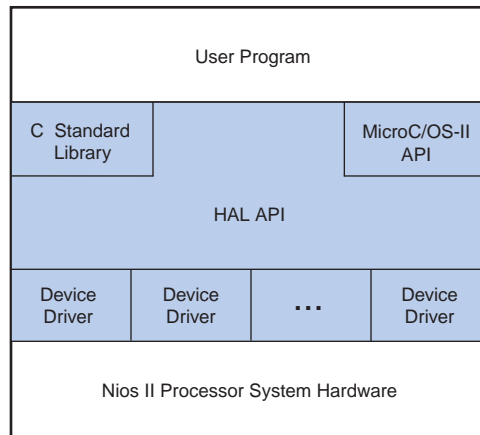
The MicroC/OS-II software package behaves like the drivers for SOPC Builder hardware components: When MicroC/OS-II is included in a Nios II project, the header and source files from **components/micrium\_uc\_osii** are included in the project path, causing the MicroC/OS-II kernel to compile and link as part of the project.

## MicroC/OS-II Architecture


The Altera implementation of MicroC/OS-II for the Nios II processor extends the single-threaded HAL environment to include the MicroC/OS-II scheduler and the associated MicroC/OS-II API. The complete HAL API is available to all MicroC/OS-II projects.

Figure 10-1 shows the architecture of a program based on MicroC/OS-II and its relationship to the HAL API.

**Figure 10-1.** Architecture of MicroC/OS-II Programs




The multi-threaded environment affects certain HAL functions.

 For details about the consequences of calling a particular HAL function in a multi-threaded environment, refer to the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.


## MicroC/OS-II Thread-Aware Debugging

When you debug a MicroC/OS-II application, the debugger can display the current state of all threads in the application, including backtraces and register values. You cannot use the debugger to change the current thread, so it is not possible to use the debugger to change threads or to single-step a different thread.

 Thread-aware debugging does not change the behavior of the target application in any way.

## MicroC/OS-II Device Drivers

Each peripheral (that is, each SOPC Builder component) can provide include files and source files in the **inc** and **src** subdirectories of the component's **HAL** directory.

 For more information, refer to the *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

In addition to the **HAL** directory, a component can optionally provide a **UCOSII** directory that contains code specific to the MicroC/OS-II environment. Similar to the **HAL** directory, the **UCOSII** directory contains **inc** and **src** subdirectories.

When you create a MicroC/OS-II project, these directories are added to the search paths for source and include files.

The Nios II Software Build Tools copy the files to your BSP's **obj** subdirectory.

- For more information about specifying file paths with the Nios II Software Build Tools, refer to “Nios II Software Projects” in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer’s Handbook*.

You can use the **UCOSII** directory to provide code that is used only in a multi-threaded environment. Other than these additional search directories, the mechanism for providing MicroC/OS-II device drivers is identical to the process for any other device driver.

- For details about developing device drivers, refer to the *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer’s Handbook*.

The HAL system initialization process calls the MicroC/OS-II function `OSInit()` before `alt_sys_init()`, which instantiates and initializes each device in the system. Therefore, the complete MicroC/OS-II API is available to device drivers, although the system is still running in single-threaded mode until the program calls `OSStart()` from within `main()`.

## Thread-Safe HAL Drivers

To enable a driver to be ported between the HAL and MicroC/OS-II environments, Altera defines a set of operating system-independent macros that provide access to operating system facilities. These macros implement functionality that is only relevant to a multi-threaded environment. When compiled for a MicroC/OS-II project, the macros expand to MicroC/OS-II API calls. When compiled for a single-threaded HAL project, the macros expand to benign empty implementations. These macros are used in Altera-provided device driver code, and you can use them if you need to write a device driver with similar portability.

Table 10-1 lists the available macros and their functions.

- For more information about the functionality in the MicroC/OS-II environment, refer to *MicroC/OS-II: The Real-Time Kernel*.

The path listed for the header file is relative to the `<Nios II EDS install path>/components/micrium_uc_osii/UCOSII/inc` directory.

**Table 10-1.** OS-Independent Macros for Thread-Safe HAL Drivers (Part 1 of 2)

Macro	Defined in Header	MicroC/OS-II Implementation	Single-Threaded HAL Implementation
<code>ALT_FLAG_GRP(group)</code>	<code>os/alt_flag.h</code>	Create a pointer to a flag group with the name <code>group</code> .	Empty statement.
<code>ALT_EXTERN_FLAG_GRP(group)</code>	<code>os/alt_flag.h</code>	Create an external reference to a pointer to a flag group with name <code>group</code> .	Empty statement.
<code>ALT_STATIC_FLAG_GRP(group)</code>	<code>os/alt_flag.h</code>	Create a static pointer to a flag group with the name <code>group</code> .	Empty statement.
<code>ALT_FLAG_CREATE(group, flags)</code>	<code>os/alt_flag.h</code>	Call <code>OSFlagCreate()</code> to initialize the flag group pointer, <code>group</code> , with the flags value <code>flags</code> . The error code is the return value of the macro.	Return 0 (success).

**Table 10-1.** OS-Independent Macros for Thread-Safe HAL Drivers (Part 2 of 2)

Macro	Defined in Header	MicroC/OS-II Implementation	Single-Threaded HAL Implementation
ALT_FLAG_PEND( <i>group</i> , <i>flags</i> , <i>wait_type</i> , <i>timeout</i> )	os/alt_flag.h	Call OSFlagPend() with the first four input arguments set to <i>group</i> , <i>flags</i> , <i>wait_type</i> , and <i>timeout</i> respectively. The error code is the return value of the macro.	Return 0 (success).
ALT_FLAG_POST( <i>group</i> , <i>flags</i> , <i>opt</i> )	os/alt_flag.h	Call OSFlagPost() with the first three input arguments set to <i>group</i> , <i>flags</i> , and <i>opt</i> respectively. The error code is the return value of the macro.	Return 0 (success).
ALT_SEM( <i>sem</i> )	os/alt_sem.h	Create an OS_EVENT pointer with the name <i>sem</i> .	Empty statement.
ALT_EXTERN_SEM( <i>sem</i> )	os/alt_sem.h	Create an external reference to an OS_EVENT pointer with the name <i>sem</i> .	Empty statement.
ALT_STATIC_SEM( <i>sem</i> )	os/alt_sem.h	Create a static OS_EVENT pointer with the name <i>sem</i> .	Empty statement.
ALT_SEM_CREATE( <i>sem</i> , <i>value</i> )	os/alt_sem.h	Call OSSemCreate() with the argument <i>value</i> to initialize the OS_EVENT pointer <i>sem</i> . The return value is zero on success, or negative otherwise.	Return 0 (success).
ALT_SEM_PEND( <i>sem</i> , <i>timeout</i> )	os/alt_sem.h	Call OSSemPend() with the first two argument set to <i>sem</i> and <i>timeout</i> respectively. The error code is the return value of the macro.	Return 0 (success).
ALT_SEM_POST( <i>sem</i> )	os/alt_sem.h	Call OSSemPost() with the input argument <i>sem</i> .	Return 0 (success).

## The Newlib ANSI C Standard Library

Programs based on MicroC/OS-II can also call the ANSI C standard library functions. Some consideration is necessary in a multi-threaded environment to ensure that the C standard library functions are thread-safe. The newlib C library stores all global variables in a single structure referenced through the pointer `_impure_ptr`. However, the Altera MicroC/OS-II implementation creates a new instance of the structure for each task. During a context switch, the value of `_impure_ptr` is updated to point to the current task's version of this structure. In this way, the contents of the structure pointed to by `_impure_ptr` are treated as thread local. For example, through this mechanism each task has its own version of `errno`.

This thread-local data is allocated at the top of the task's stack. You must make allowance for thread-local data storage when allocating memory for stacks. In general, the `_reent` structure consumes approximately 900 bytes of data for the normal C library, or 90 bytes for the reduced-footprint C library.



For further details about the contents of the `_reent` structure, refer to the newlib documentation. On the Windows Start menu, click **Programs > Altera > Nios II > Nios II Documentation**.


In addition, the MicroC/OS-II implementation provides appropriate task locking to ensure that heap accesses (calls to `malloc()` and `free()`) are also thread-safe.

## Interrupt Service Routines for MicroC/OS-II

Implementing ISRs for MicroC/OS-II normally involves some housekeeping details, as described in *MicroC/OS-II: The Real-Time Kernel*. However, because the Nios II implementation of MicroC/OS-II is based on the HAL, several of these details are taken care of for you. The HAL performs the following housekeeping tasks for your interrupt service routine (ISR):


- Saves and restores processor registers
- Calls `OSIntEnter()` and `OSIntExit()`


The HAL also allows you to write your ISR in C, rather than assembly language.

 For more detail about writing ISRs with the HAL, refer to the *Exception Handling* chapter of the *Nios II Software Developer's Handbook*.

## Implementing MicroC/OS-II Projects for the Nios II Processor

To create a program based on MicroC/OS-II, start by setting the BSP properties so that it is a MicroC/OS-II project. You can control the configuration of the MicroC/OS-II kernel using BSP settings with the Nios II Software Build Tools for Eclipse™, or on the Nios II command line.

 You do not need to edit header files (such as `OS_CFG.h`) or source code to configure the MicroC/OS-II features. The project settings are reflected in the BSP's `system.h` file; `OS_CFG.h` simply includes `system.h`.

 For a list of available MicroC/OS-II BSP settings, refer to “Settings” in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*. MicroC/OS-II settings are identified by the prefix `ucosii`. For information about how to configure MicroC/OS-II with BSP settings, refer to the *Getting Started with the Graphical User Interface, Nios II Software Build Tools*, and *Nios II Software Build Tools Reference* chapters of the *Nios II Software Developer's Handbook*. The meaning of each setting is defined fully in *MicroC/OS-II: The Real-Time Kernel*.

## Referenced Documents

This chapter references the following documents:

- *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook*
- *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*
- *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*
- *Exception Handling* 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*
- *Using the MicroC/OS-II RTOS with the Nios II Processor Tutorial*
- *MicroC/OS-II: The Real-Time Kernel*, Jean J. Labrosse, CMP Books
- Newlib ANSI C standard library documentation installed with the Nios II EDS
- The [Embedded Software](#) page of the Altera website

## Document Revision History

Table 10-2 shows the revision history for this document.

**Table 10-2.** Document Revision History

Date & Document Version	Changes Made	Summary of Changes
November 2009 v9.1.0	<ul style="list-style-type: none"> <li>■ Remove tables of Nios II IDE-specific setting names. Refer solely to BSP setting names.</li> </ul>	<ul style="list-style-type: none"> <li>■ Introduced the Nios II Software Build Tools for Eclipse</li> </ul>
March 2009 v9.0.0	<ul style="list-style-type: none"> <li>■ Reorganized and updated information and terminology to clarify role of Nios II Software Build Tools.</li> <li>■ Corrected minor typographical errors.</li> </ul>	
May 2008 v8.0.0	No change from previous release.	
October 2007 v7.2.0	<ul style="list-style-type: none"> <li>■ Added documentation for MicroC/OS-II development with the Nios II Software Build Tools.</li> <li>■ Added description of HAL ISR support</li> </ul>	
May 2007 v7.1.0	<ul style="list-style-type: none"> <li>■ Added table of contents to "Introduction" section.</li> <li>■ Added Referenced Documents section.</li> </ul>	
March 2007 v7.0.0	No change from previous release.	
November 2006 v6.1.0	No change from previous release.	
May 2006 v6.0.0	No change from previous release.	
October 2005 v5.1.0	No change from previous release.	
May 2005 v5.0.0	No change from previous release.	
December 2004 v1.1	Added thread-aware debugging paragraph.	
May 2004 v1.0	Initial Release.	



## Overview

The NicheStack® TCP/IP Stack - Nios® II Edition is a small-footprint implementation of the TCP/IP suite. The focus of the NicheStack TCP/IP Stack implementation is to reduce resource usage while providing a full-featured TCP/IP stack. The NicheStack TCP/IP Stack is designed for use in embedded systems with small memory footprints, making it suitable for Nios II processor systems.


Altera provides the NicheStack TCP/IP Stack as a software package that you can add to your board support package (BSP), available through the Nios II Software Build Tools. The NicheStack TCP/IP Stack includes these features:

- Internet Protocol (IP) including packet forwarding over multiple network interfaces
- Internet control message protocol (ICMP) for network maintenance and debugging
- User datagram protocol (UDP)
- Transmission Control Protocol (TCP) with congestion control, round trip time (RTT) estimation, and fast recovery and retransmit
- Dynamic host configuration protocol (DHCP)
- Address resolution protocol (ARP) for Ethernet
- Standard sockets application program interface (API)

This chapter discusses the details of how to use the NicheStack TCP/IP Stack for the Nios II processor only. This chapter contains the following sections:

- “Prerequisites” on page 11-2
- “Introduction” on page 11-2
- “Other TCP/IP Stack Providers” on page 11-3
- “Using the NicheStack TCP/IP Stack” on page 11-3
- “Configuring the NicheStack TCP/IP Stack in a Nios II Program” on page 11-9
- “Further Information” on page 11-10
- “Known Limitations” on page 11-10

## Prerequisites

 To make the best use of information in this chapter, you should be familiar with these topics:

- Sockets. Several books are available on the topic of programming with sockets. Two good texts are *Unix Network Programming* by Richard Stevens and *Internetworking with TCP/IP Volume 3* by Douglas Comer.
- The Nios II Embedded Design Suite (EDS). Refer to the *Overview* chapter of the *Nios II Software Developer's Handbook* for more information about the Nios II EDS.
- The MicroC/OS-II RTOS. To learn about MicroC/OS-II, refer to the *MicroC/OS-II Real-Time Operating System* chapter of the *Nios II Software Developer's Handbook*, or to the *Using MicroC/OS-II RTOS with the Nios II Processor Tutorial*.

## Introduction

Altera provides the Nios II implementation of the NicheStack TCP/IP Stack, including source code, in the Nios II EDS. The NicheStack TCP/IP Stack provides you with immediate access to a stack for Ethernet connectivity for the Nios II processor. The Altera implementation of the NicheStack TCP/IP Stack includes an API wrapper, providing the standard, well documented socket API.

The NicheStack TCP/IP Stack uses the MicroC/OS-II RTOS multithreaded environment. Therefore, to use the NicheStack TCP/IP Stack with the Nios II EDS, you must base your C/C++ project on the MicroC/OS-II RTOS. The Nios II processor system must also contain an Ethernet interface, or media access control (MAC). The Altera-provided NicheStack TCP/IP Stack includes driver support for the following two MACs:

- The SMSC lan91c111 device
- The Altera® Triple Speed Ethernet MegaCore® function

The Nios II Embedded Design Suite includes hardware for both MACs. The NicheStack TCP/IP Stack driver is interrupt-based, so you must ensure that interrupts for the Ethernet component are connected.


Altera's implementation of the NicheStack TCP/IP Stack is based on the hardware abstraction layer (HAL) generic Ethernet device model. In the generic device model, you can write a new driver to support any target Ethernet MAC, and maintain the consistent HAL and sockets API to access the hardware.

 For details about writing an Ethernet device driver, refer to the *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

## The NicheStack TCP/IP Stack Files and Directories

You need not edit the NicheStack TCP/IP Stack source code to use the stack in a Nios II C/C++ program. Nonetheless, Altera provides the source code for your reference. By default the files are installed with the Nios II EDS in the `<Nios II EDS install path>/components/altera_iniche/UCOSII` directory. For the sake of brevity, this chapter refers to this directory as `<iniche path>`.


Under `<iniche path>`, the original code is maintained—as much as possible—under the `<iniche path>/src/downloads` directory. This organization facilitates upgrading to more recent versions of the NicheStack TCP/IP Stack. The `<iniche path>/src/downloads/packages` directory contains the original NicheStack TCP/IP Stack source code and documentation; the `<iniche path>/src/downloads/30src` directory contains code specific to the Nios II implementation of the NicheStack TCP/IP Stack, including source code supporting MicroC/OS-II.

 The reference manual for the NicheStack TCP/IP Stack is available on the [Literature: Nios II Processor](#) page of the Altera website, under **Other Related Documentation**.

Altera's implementation of the NicheStack TCP/IP Stack is based on version 3.1 of the protocol stack, with wrappers around the code to integrate it with the HAL.

## Licensing

The NicheStack TCP/IP Stack is a TCP/IP protocol stack created by InterNiche Technologies, Inc. You can license the NicheStack TCP/IP Stack from Altera by going to [the Altera website](#).

 You can license other protocol stacks directly from InterNiche. You can obtain details from InterNiche Technologies, Inc. ([www.interniche.com](http://www.interniche.com))

## Other TCP/IP Stack Providers

Other third party vendors also provide Ethernet support for the Nios II processor. Notably, third party RTOS vendors often offer Ethernet modules for their particular RTOS frameworks.

 For up-to-date information about products available from third party providers, visit the [Embedded Software](#) page of the Altera website.

## Using the NicheStack TCP/IP Stack

This section discusses how to include the NicheStack TCP/IP Stack in a Nios II program.

The primary interface to the NicheStack TCP/IP Stack is the standard sockets interface. In addition, you call the following functions to initialize the stack and drivers:

- `alt_iniche_init()`
- `netmain()`

You also use the global variable `iniche_net_ready` in the initialization process.

You must provide the following simple functions, which the HAL system code calls to obtain the MAC address and IP address:

- `get_mac_addr()`
- `get_ip_addr()`

## Nios II System Requirements

To use the NicheStack TCP/IP Stack, your Nios II system must meet the following requirements:

- The system hardware generated in SOPC Builder must include an Ethernet interface with interrupts enabled.
- The BSP must be based on MicroC/OS-II.
- The MicroC/OS-II RTOS must be configured to have the following settings:
  - TimeManagement / OSTimeTickHook must be enabled.
  - Maximum Number of Tasks must be 4 or higher.
- The system clock timer must be set to point to an appropriate timer device.

## The NicheStack TCP/IP Stack Tasks

The NicheStack TCP/IP Stack, in its standard Nios II configuration, consists of two fundamental tasks. Each of these tasks consumes a MicroC/OS-II thread resource, along with some memory for the thread's stack. In addition to the tasks your program creates, the following tasks run continuously:

- **The NicheStack main task**, `tk_netmain()`—After initialization, this task sleeps until a new packet is available for processing. Packets are received by an interrupt service routine (ISR). When the ISR receives a packet, it places it in the receive queue, and wakes up the main task.
- **The NicheStack tick task**, `tk_nettick()`—This task wakes up periodically to monitor for time-out conditions.

These tasks are started when the initialization process succeeds in the `netmain()` function, as described in “[netmain\(\)](#)”.




You can modify the task priority and stack sizes using `#define` statements in the configuration file `ippport.h`. You can create additional system tasks by enabling other options in the NicheStack TCP/IP Stack by editing `ippport.h`.

## Initializing the Stack

Before you initialize the stack, start the MicroC/OS-II scheduler by calling `OSStart()` from `main()`. Perform stack initialization in a high priority task, to ensure that your code does not attempt further initialization until the RTOS is running and I/O drivers are available.

To initialize the stack, call the functions `alt_iniche_init()` and `netmain()`. Global variable `iniche_net_ready` is set `true` when stack initialization is complete.

 Ensure that your code does not use the sockets interface before `iniche_net_ready` is set to `true`. For example, call `alt_iniche_init()` and `netmain()` from the highest priority task, and wait for `iniche_net_ready` before allowing other tasks to run, as shown in [Example 11-1](#).

### **alt\_iniche\_init()**

`alt_iniche_init()` initializes the stack for use with the MicroC/OS-II operating system. The prototype for `alt_iniche_init()` is:

```
void alt_iniche_init(void)
```

`alt_iniche_init()` returns nothing and has no parameters.

### **netmain()**


`netmain()` is responsible for initializing and launching the NicheStack tasks. The prototype for `netmain()` is:

```
void netmain(void)
```

`netmain()` returns nothing and has no parameters.

### **iniche\_net\_ready**

When the NicheStack stack has completed initialization, it sets the global variable `iniche_net_ready` to a non-zero value.

 Do not call any NicheStack API functions (other than for initialization) until `iniche_net_ready` is `true`.

[Example 11-1](#) illustrates the use of `iniche_net_ready` to wait until the network stack has completed initialization.

#### **Example 11-1.** Instantiating the NicheStack TCP/IP Stack

---

```
void SSSInitialTask(void *task_data)
{
    INT8U error_code;

    alt_iniche_init();
    netmain();

    while (!iniche_net_ready)
        TK_SLEEP(1);

    /* Now that the stack is running, perform the application
       initialization steps */

    .
    .
    .
}
```

---

Macro `TK_SLEEP()` is part of the NicheStack TCP/IP Stack operating system (OS) porting layer.

**get\_mac\_addr() and get\_ip\_addr()**

The NicheStack TCP/IP Stack system code calls `get_mac_addr()` and `get_ip_addr()` during the device initialization process. These functions are necessary for the system code to set the MAC and IP addresses for the network interface, which you select with the `altera_iniche.iniche_default_if` BSP setting. Because you write these functions yourself, your system has the flexibility to store the MAC address and IP address in an arbitrary location, rather than a fixed location hard-coded in the device driver. For example, some systems might store the MAC address in flash memory, while others might have the MAC address in on-chip embedded memory.

Both functions take as parameters device structures used internally by the NicheStack TCP/IP Stack. However, you do not need to know the details of the structures. You only need to know enough to fill in the MAC and IP addresses.

The prototype for `get_mac_addr()` is:

```
int get_mac_addr(NET net, unsigned char mac_addr[6]);
```

You must implement the `get_mac_addr()` function to assign the MAC address to the `mac_addr` argument. Leave the `net` argument untouched.

The prototype for `get_mac_addr()` is in the header file `<iniche path>/inc/alt_iniche_dev.h`. The `NET` structure is defined in the `<iniche path>/src/downloads/30src/h/net.h` file.

**Example 11-2** shows an implementation of `get_mac_addr()`. For demonstration purposes only, the MAC address is stored at address `CUSTOM_MAC_ADDR` in this example. There is no error checking in this example. In a real application, if there is an error, `get_mac_addr()` must return -1.

**Example 11-2. An Implementation of get\_mac\_addr()**


---

```
#include <alt_iniche_dev.h>
#include "includes.h"
#include "ipport.h"
#include "tcpport.h"
#include <io.h>
int get_mac_addr(NET net, unsigned char mac_addr[6])
{
    int ret_code = -1;

    /* Read the 6-byte MAC address from wherever it is stored */
    mac_addr[0] = IORD_8DIRECT(CUSTOM_MAC_ADDR, 4);
    mac_addr[1] = IORD_8DIRECT(CUSTOM_MAC_ADDR, 5);
    mac_addr[2] = IORD_8DIRECT(CUSTOM_MAC_ADDR, 6);
    mac_addr[3] = IORD_8DIRECT(CUSTOM_MAC_ADDR, 7);
    mac_addr[4] = IORD_8DIRECT(CUSTOM_MAC_ADDR, 8);
    mac_addr[5] = IORD_8DIRECT(CUSTOM_MAC_ADDR, 9);
    ret_code = ERR_OK;

    return ret_code;
}
```

---

You must write the function `get_ip_addr()` to assign the IP address of the protocol stack. Your program can either assign a static address, or request the DHCP to find an IP address. The function prototype for `get_ip_addr()` is:

```
int get_ip_addr(alt_iniche_dev* p_dev,
               ip_addr*      ipaddr,
               ip_addr*      netmask,
               ip_addr*      gw,
               int*          use_dhcp);
```

`get_ip_addr()` sets the return parameters as follows:

```
IP4_ADDR(&ipaddr, IPADDR0, IPADDR1, IPADDR2, IPADDR3);
IP4_ADDR(&gw, GWADDR0, GWADDR1, GWADDR2, GWADDR3);
IP4_ADDR(&netmask, MSKADDR0, MSKADDR1, MSKADDR2, MSKADDR3);
```

For the dummy variables `IP_ADDR0-3`, substitute expressions for bytes 0-3 of the IP address. For `GWADDR0-3`, substitute the bytes of the gateway address. For `MSKADDR0-3`, substitute the bytes of the network mask. For example, the following statement sets `ip_addr` to IP address 137.57.136.2:

```
IP4_ADDR ( ip_addr, 137, 57, 136, 2 );
```

To enable DHCP, include the line:

```
*use_dhcp = 1;
```

The NicheStack TCP/IP stack attempts to get an IP address from the server. If the server does not provide an IP address within 30 seconds, the stack times out and uses the default settings specified in the `IP4_ADDR()` function calls.

To assign a static IP address, include the lines:

```
*use_dhcp = 0;
```

The prototype for `get_ip_addr()` is in the header file `<iniche path>/incl/alt_iniche_dev.h`.

**Example 11-3** shows an implementation of `get_ip_addr()` and shows a list of the necessary include files.



There is no error checking in **Example 11-3**. In a real application, you might need to return -1 on error.

`INICHE_DEFAULT_IF`, defined in **system.h**, identifies the network interface that you defined in SOPC Builder. You can control `INICHE_DEFAULT_IF` through the `iniche_default_if` BSP setting.

`DHCP_CLIENT`, also defined in **system.h**, specifies whether to use the DHCP client application to obtain an IP address. You can set or clear this property with the `altera_iniche.dhcp_client` setting.

## Calling the Sockets Interface

After you initialize your Ethernet device, use the sockets API in the remainder of your program to access the IP stack.

To create a new task that talks to the IP stack using the sockets API, you must use the function `TK_NEWTASK()`. The `TK_NEWTASK()` function is part of the NicheStack TCP/IP Stack operating system (OS) porting layer. `TK_NEWTASK()` calls the MicroC/OS-II `OSTaskCreate()` function to create a thread, and performs some other actions specific to the NicheStack TCP/IP Stack.

The prototype for `TK_NEWTASK()` is:

```
int TK_NEWTASK(struct inet_task_info* nettask);
```

---

**Example 11-3.** An Implementation of `get_ip_addr()`

---

```
#include <alt_iniche_dev.h>
#include "includes.h"
#include "ipport.h"
#include "tcpport.h"
int get_ip_addr(alt_iniche_dev* p_dev,
               ip_addr* ipaddr,
               ip_addr* netmask,
               ip_addr* gw,
               int*      use_dhcp)
{
    int ret_code = -1;
    /*
     * The name here is the device name defined in system.h
     */
    if (!strcmp(p_dev->name, "/dev/" INICHE_DEFAULT_IF))
    {
        /* The following is the default IP address if DHCP
         fails, or the static IP address if DHCP_CLIENT is
         undefined. */
        IP4_ADDR(&ipaddr, 10, 1, 1, 3);
        /* Assign the Default Gateway Address */
        IP4_ADDR(&gw, 10, 1, 1, 254);
        /* Assign the Netmask */
        IP4_ADDR(&netmask, 255, 255, 255, 0);

#ifdef DHCP_CLIENT
        *use_dhcp = 1;
#else
        *use_dhcp = 0;
#endif /* DHCP_CLIENT */

        ret_code = ERR_OK;
    }
    return ret_code;
}
```

---

The prototype is defined in `<iniche path>/src/downloads/30src/nios2/osport.h`. You can include this header file as follows:

```
#include "osport.h"
```


You can find other details of the OS porting layer in the `osport.c` file in the NicheStack TCP/IP Stack component directory, `<iniche path>/src/downloads/30src/nios2/`.



For more information about how to use `TK_NEWTASK()` in an application, refer to the [Using the NicheStack TCP/IP Stack - Nios II Edition Tutorial](#).

## Configuring the NicheStack TCP/IP Stack in a Nios II Program

The NicheStack TCP/IP Stack has many options that you can configure using `#define` directives in the file `ippport.h`. The Nios II EDS allows you to configure certain options (that is, modify the `#defines` in `system.h`) without editing source code. The most commonly accessed options are available through a set of BSP options, identifiable by the prefix `altera_iniche`.

 For further information about BSP settings for the NicheStack, refer to the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Some less-frequently-used options are not accessible through the BSP settings. If you need to modify these options, you must edit the `ippport.h` file manually.

You can find `ippport.h` in the `debug/system_description` directory for your BSP project.

The following sections describe the features that you can configure using the Nios II Software Build Tools. Both development flows provide a default value for each feature. In general, these values provide a good starting point, and you can later fine-tune the values to meet the needs of your system.

### NicheStack TCP/IP Stack General Settings

The ARP, UDP, and IP protocols are always enabled. [Table 11-1](#) shows the protocol options.

**Table 11-1.** Protocol Options

Option	Description
TCP	Enables and disables the TCP.

[Table 11-2](#) shows the global options, which affect the overall behavior of the TCP/IP stack.

**Table 11-2.** Global Options

Option	Description
Use DHCP to automatically assign IP address	If this option is turned on, the component uses DHCP to acquire an IP address. If this option is turned off, you must assign a static IP address.
Enable statistics	If this option is turned on, the stack keeps counters of packets received, errors, etc. The counters are defined in <code>mib</code> structures defined in various header files in directory <code>&lt;iniche path&gt;/src/downloads/30src/h</code> . For details about <code>mib</code> structures, refer to the NicheStack documentation.
MAC interface	If the IP stack has more than one network interface, this parameter indicates which interface to use. Refer to <a href="#">“Known Limitations” on page 11-10</a> .

## IP Options

Table 11-3 shows the IP options.

**Table 11-3.** IP Options

Option	Description
Forward IP packets	If there is more than one network interface, this option is turned on, and the IP stack for one interface receives packets that are not addressed to it, the stack forwards the packet out of the other interface. Refer to “Known Limitations” on page 11-10.
Reassemble IP packet fragments	If this option is turned on, the NicheStack TCP/IP Stack reassembles IP packet fragments as full IP packets. Otherwise, it discards IP packet fragments. This topic is explained in <i>Unix Network Programming</i> by Richard Stevens.

## TCP Options

Table 11-4 shows the TCP zero copy option, which is only available if the TCP option is turned on.

**Table 11-4.** TCP Options

Option	Description
Use TCP zero copy	This option enables the NicheStack zero copy TCP API. This option allows you to eliminate buffer-to-buffer copies when using the NicheStack TCP/IP Stack. For details, refer to the NicheStack reference manual. You must modify your application code to take advantage of the zero copy API.

## Further Information

For further information about the Altera NicheStack implementation, refer to the [Using the NicheStack TCP/IP Stack - Nios II Edition Tutorial](#). The tutorial provides in-depth information about the NicheStack TCP/IP Stack, and illustrates how to use it in a networking application.



For details about NicheStack, refer to the NicheStack TCP/IP Stack reference manual, available on the [Literature: Nios II Processor](#) page of the Altera website, under **Other Related Documentation**.

## Known Limitations

Although the NicheStack code contains features intended to support multiple network interfaces, these features are not tested in the Nios II edition. Refer to the NicheStack TCP/IP Stack reference manual and source code for information about multiple network interface support.

## Referenced Documents

This chapter references the following documents:

- *Overview* chapter of the *Nios II Software Developer's Handbook*
- *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*
- *MicroC/OS-II Real-Time Operating System* chapter of the *Nios II Software Developer's Handbook*
- *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*
- NicheStack TCP/IP Stack reference manual, available on the [Literature: Nios II Processor](#) page of the Altera website, under **Other Related Documentation**
- *Using the NicheStack TCP/IP Stack - Nios II Edition Tutorial*
- *Using MicroC/OS-II RTOS with the Nios II Processor Tutorial*

## Document Revision History

Table 11-5 shows the revision history for this document.

**Table 11-5.** Document Revision History

Date and Document Version	Changes Made	Summary of Changes
November 2009 v9.1.0	<ul style="list-style-type: none"> <li>■ Nios II IDE information removed to <i>Appendix A. Using the Nios II Integrated Development Environment</i>.</li> </ul>	Introduced the Nios II Software Build Tools for Eclipse™
March 2009 v9.0.0	<ul style="list-style-type: none"> <li>■ Reorganized and updated information and terminology to clarify role of Nios II Software Build Tools.</li> <li>■ Corrected minor typographical errors.</li> </ul>	
May 2008 v8.0.0	No change from previous release.	
October 2007 v7.2.0	No change from previous release.	
May 2007 v7.1.0	<ul style="list-style-type: none"> <li>■ Minor clarifications added to content.</li> <li>■ Added table of contents to Overview section.</li> <li>■ Added Referenced Documents section.</li> </ul>	
March 2007 v7.0.0	No change from previous release.	
November 2006 v6.1.0	Initial Release.	



## Introduction

Altera provides a read-only zip file system for use with the hardware abstraction layer (HAL). The read-only zip file system provides access to a simple file system stored in flash memory. The drivers take advantage of the HAL generic device driver framework for file subsystems. Therefore, you can access the zip file subsystem using the ANSI C standard library I/O functions, such as `fopen()` and `fread()`.

The Altera® read-only zip file system is provided as a software package. All source and header files for the HAL drivers are located in the directory `<Nios II EDS install path>/components/altera_ro_zipfs/HAL/`.

## Using the Zip File System in a Project

The read-only zip file system is supported by both Nios® II software development flows. You need not edit any source code to include and configure the file system. To use the zip file system, you use the Nios II development tools to include it as a software package for the board support package (BSP) project.

You must specify the following four parameters to configure the file system:

- The name of the flash device where you wish to program the file system.
- The offset in the address space of this flash device.
- The name of the mount point for this file subsystem in the HAL file system. For example, if you name the mount point `/mnt/zipfs`, the following code opens a file in the zip file:

```
fopen("/mnt/zipfs/hello", "r");
```

This code, called from within a HAL-based program, opens the file `hello` for reading.

- The name of the zip file you wish to use.

The next time you build your project after you make these settings, the Nios II development tools include and link the file subsystem in the project. After you rebuild the project, the `system.h` file reflects the presence of this software package in the system.

## Preparing the Zip File


The zip file must be uncompressed. The Altera read-only zip file system uses the zip format only for bundling files together; it does not provide the file decompression features for which zip utilities are known.

Creating a zip file with no compression is straightforward using the WinZip GUI. Alternately, use the `-e0` option to disable compression when using either `winzip` or `pkzip` from a command line.

## Programming the Zip File to Flash

For your program to access files in the zip file subsystem, you must first program the zip data to flash. As part of the project build process, the Nios II development tools create a Motorola S-record file (**.flash**) that includes the data for the zip file system.

You then use the Nios II Flash Programmer to program the zip file system data to flash memory on the board.

 For details about programming flash, refer to the *Nios II Flash Programmer User Guide*.

## Referenced Documents

This chapter references the following document:

*Nios II Flash Programmer User Guide*

## Document Revision History

Table 12-1 shows the revision history for this document.

**Table 12-1.** Document Revision History (Part 1 of 2)

Date & Document Version	Changes Made	Summary of Changes
November 2009 v9.1.0	No change from previous release.	
March 2009 v9.0.0	<ul style="list-style-type: none"> <li>■ Reorganized and updated information and terminology to clarify role of Nios II Software Build Tools.</li> <li>■ Corrected minor typographical errors.</li> </ul>	
May 2008 v8.0.0	No change from previous release.	
October 2007 v7.2.0	No change from previous release.	
May 2007 v7.1.0	<ul style="list-style-type: none"> <li>■ Added table of contents to “Introduction” section.</li> <li>■ Added Referenced Documents section.</li> </ul>	
March 2007 v7.0.0	No change from previous release.	
November 2006 v6.1.0	No change from previous release.	
May 2006 v6.0.0	No change from previous release.	
October 2005 v5.1.0	No change from previous release.	

**Table 12-1.** Document Revision History (Part 2 of 2)

<b>Date &amp; Document Version</b>	<b>Changes Made</b>	<b>Summary of Changes</b>
May 2005 v5.0.0	No change from previous release.	
May 2004 v1.0	Initial Release.	

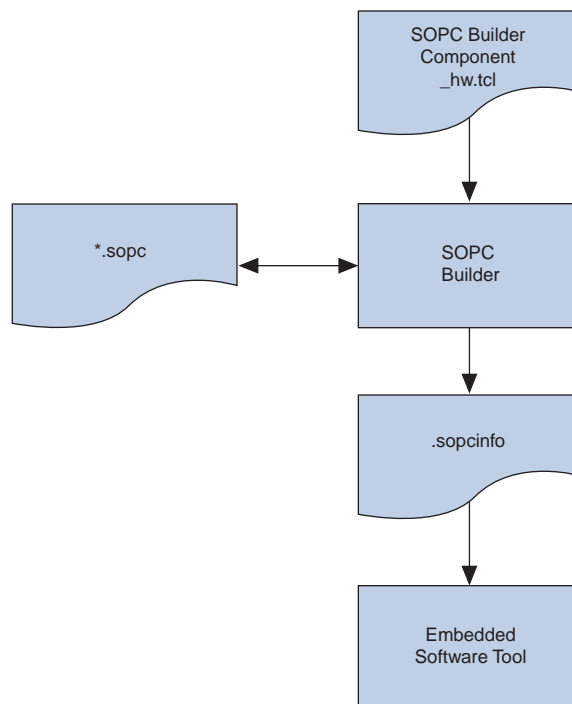


This document describes how to publish SOPC Builder component information for embedded software tools. You can publish component information for use by software, such as a C compiler and a board support package (BSP) generator. Information used by a C compiler might be a set of `#define` statements that describe some aspect of a component. Information used by a BSP generator might be the identification of memory components, so that the BSP generator can create a linker script.

## Component Information Flow

Figure 13–1 shows the flow of information from SOPC Builder components to embedded software tools.

**Figure 13–1.** Component Information Flow



A component publishes information by including Tcl assignment statements in its component description file, `<component_name>_hw.tcl`. Each assignment is a name-value pair that can be associated with the entire component, or with a single interface. When the assignment statement applies to the entire component, it is set using the `set_module_assignment` command. Assignment statements that apply to an interface are set using the `set_interface_assignment` command.

Example 13–1 shows the syntax for these assignment statements.

**Example 13-1.** Syntax of Assignment Statements

---


```
# These assignments apply to the entire component
# This is the syntax for the set_module_assignment command:
#   set_module_assignment <assignment_name> <value>

# Here are 3 examples
set_module_assignment embeddedsw.CMacro.colorSpace "CMYK"
set_module_assignment embeddedsw.configuration.cpuArchitecture "My processor"
set_module_assignment embeddedsw.memoryInfo.IS_FLASH 1

# This is the syntax of the set_interface_assignment command:
#   set_interface_assignment <interface_name> <assignment_name> <value>

# Here is an example
set_interface_assignment lcd0 embeddedsw.configuration.isPrintableDevice 1
```

---

 For more information about the `_hw.tcl` file and using Tcl to define SOPC Builder components, refer to the *Component Interface Tcl Reference* chapter in *Volume 4: SOPC Builder of the Quartus® II Handbook*.

When you generate an SOPC Builder system, SOPC Builder creates an `<sopc_builder_system>.sopcinfo` file that includes all of the assignments for your component. The embedded software tools use these assignments for further processing. SOPC Builder does not require any of the information included in these assignments to build the hardware representation of the component. SOPC Builder simply passes the assignments from the `_hw.tcl` file to the SOPC Information File (`.sopcinfo`).

## Embedded Software Assignments

Embedded software assignments are organized in a period-separated namespace. All of the assignments for embedded software tools have the prefix `embeddedsw`. The `embeddedsw` namespace is further divided into the following three sub-namespaces:

- C Macro—Assignment name prefix `embeddedsw.CMacro`
- Configuration—Assignment name prefix `embeddedsw.configuration`
- Memory Initialization—Assignment name prefix `embeddedsw.memoryInfo`

### C Macro Namespace

You can use the C macro namespace to publish information about your component that is converted to `#define`'s in a C or C++ `system.h` file. C macro assignments are associated with the entire SOPC Builder component, not with individual interfaces.

The name of an assignment in the C macro namespace is `embeddedsw.CMacro.<assignmentName>`. You must format the value as a legal C or C++ expression.

Example [Example 13-2](#) illustrates a Tcl assignment statement for the `BAUD_RATE` of `uart_0` in an SOPC Builder system.

**Example 13-2.** C Macro Example

```
# Tcl assignment statement included in the _hw.tcl file
add_parameter BAUD_RATE_PARAM integer 9600 "This is the default baud rate."


# Dynamically reassign the baud rate based on the parameter value
set_module_assignment embeddedsw.CMacro.BAUD_RATE \
    [get_parameter_value BAUD_RATE_PARAM]
```

Example 13-3 illustrates the corresponding C or C++ #define. The string BAUD\_RATE is appended to the name of the component. This #define is included in the system.h file.

**Example 13-3.** Generated Macro in system.h

```
/* Generated macro in the system.h file after dynamic reassignment */
#define UART_0_BAUD_RATE 15200
```

Table 13-1 provides examples of how to format constants for 32-bit processors using the GNU Compiler Collection (GCC) C/C++ compiler.

 For complete details on formatting constants, refer to the [GNU web page](#).

**Table 13-1.** GCC C/C++ 32-bit Processor Constants

C Data Type	Examples
boolean (char, short, int)	1, 0
32-bit signed integer (int, long)	123, -50
32-bit unsigned integer (unsigned int, unsigned long)	123u, 0xef8472a0
64-bit signed integer (long long int)	4294967296LL, -4294967296LL
64-bit unsigned integer (unsigned long long int)	4294967296ULL, 0xac458701fd64ULL
32-bit floating-point (float)	3.14f
64-bit floating-point (double)	2.78, 314e-2
character (char)	'x'
string (const char*)	"Hello World!"

**Configuration Namespace**

You can use the configuration namespace to pass configuration information to embedded software tools. You can associate configuration namespace assignments with the entire component or with single interface.

The assignment name for the configuration namespace is embeddedsw.configuration.<name>. Altera embedded software tools already have definitions for the data types of the configuration names listed in this section.

Table 13-2 shows how to format configuration assignment values based on defined data types.

**Table 13-2.** Configuration Data Types

Configuration Data Type	Format
boolean	1, 0
32-bit integer	123, -50
64-bit integer	4294967296, -4294967296
32-bit floating-point	3.14
64-bit floating-point	2.78, 314e-2
string	ABC

Table 13-3 includes the embedded software configuration names that apply to the entire component.

**Table 13-3.** Component Configuration Information - Assign with `set_module_assignment`

Configuration Name	Type	Meaning	Example
<code>cpuArchitecture</code>	string	Processor instruction set architecture. Provide this assignment if you want your component to be considered a processor.	My 8051

Table 13-4 includes the embedded software configuration names that apply to an Avalon Memory-Mapped® (Avalon-MM) slave interface. All of these assignments are optional.

**Table 13-4.** Memory-Mapped Slave Information - Assign with `set_interface_assignment` (Part 1 of 2)

Configuration Name	Type	Default	Meaning	Examples
<code>isMemoryDevice</code>	boolean	0	The slave port provides access to a memory device.	Altera® On-Chip Memory Component, DDR Controller, erasable programmable configurable serial (EPCS) Controller
<code>isPrintableDevice</code>	boolean	0	The slave port provides access to a character-based device.	Altera UART, Altera JTAG UART, Altera LCD
<code>isTimerDevice</code>	boolean	0	The slave port provides access to a timer device.	Altera Timer
<code>isEthernetMacDevice</code>	boolean	0	The slave port provides access to an Ethernet media access control (MAC).	Altera Triple-Speed Ethernet

**Table 13–4.** Memory-Mapped Slave Information - Assign with `set_interface_assignment` (Part 2 of 2)

Configuration Name	Type	Default	Meaning	Examples
<code>isNonVolatileStorage</code> (1)	boolean	0	The memory device is a non-volatile memory device. The contents of a non-volatile memory device are fixed and always present. In normal operation, you can only read from this memory. If this property is true, you must also set <code>isMemoryDevice</code> to true.	Common flash interface (CFI) Flash, EPCS Flash, on-chip FPGA memory configured as a ROM
<code>isFlash</code>	boolean	0	The memory device is a flash memory device. If <code>isFlash</code> is true, you must also set <code>isMemoryDevice</code> and <code>isNonVolatileStorage</code> to true.	CFI Flash, EPCS Flash
<code>hideDevice</code>	boolean	0	Do not make this slave port visible to the embedded software tools.	Nios® II debug slave port
<code>affectsTransactionsOnMasters</code>	string	empty string	A list of master names delimited by spaces, for example <code>m1 m2</code> . Used when the slave port provides access to Avalon-MM control registers in the component. The control registers control transfers on the specified master ports.  The slave port can configure the control registers for master ports on the listed components. The address space for this slave port is composed of the address spaces of the named master ports.  Nios II embedded software tools use this information to generate <code>#define</code> directives describing the address space of these master ports.	Altera direct memory access (DMA), Altera Scatter/Gather DMA

**Note to Table 13–4:**

- (1) Some FPGA RAMs support initialization at power-up from the SRAM Object File (`.sof`) or programmer object file (`.pof`), but are not considered non-volatile because this ability might not be used.

Table 13–5 includes the embedded software configuration names that apply to an Avalon Streaming® (Avalon-ST) slave interface. All of these assignments are optional.

**Table 13-5.** Streaming Source Information - Assign with `set_interface_assignment`

Configuration Name	Type	Default	Meaning	Examples
<code>isInterruptControllerSender</code> (1)	boolean	0	The interface sends interrupts to an interrupt controller receiver interface.	Altera Vectored Interrupt Controller
<code>transportsInterruptsFromReceivers</code> (2)	string	empty string	A list of interrupt receiver interface names delimited by spaces. Used when the interrupt controller sender interface can transport daisy-chained interrupts from one or more interrupt controller receiver ports on the same module.	Altera Vectored Interrupt Controller daisy-chain input

**Note to Table 13-5:**

- (1) An interrupt sender interface is an Avalon-ST source providing interrupt information according to the external interrupt controller (EIC) protocol.
- (2) An interrupt receiver interface is an Avalon-ST sink receiving interrupt information from an EIC.

Table 13-6 includes the embedded software configuration names that apply to an Avalon-ST sink interface. All of these assignments are optional.

**Table 13-6.** Streaming Sink Information - Assign with `set_interface_assignment`

Configuration Name	Type	Default	Meaning	Examples
<code>isInterruptControllerReceiver</code> (1)	boolean	0	The interface receives interrupts (optionally daisy-chained) from an interrupt controller sender interface.	Altera Vectored Interrupt Controller, Altera Nios II

**Note to Table 13-6:**

- (1) An interrupt receiver interface is an Avalon-ST sink receiving interrupt information from an EIC.

## Memory Initialization Namespace

You use the memory initialization namespace to pass memory initialization information to embedded software tools. Use this namespace to create memory initialization files, including `.flash`, `.hex`, `.dat`, and `.sym` files. You use memory initialization files for the following tasks:

- Flash programming
- RTL simulation
- Creating initialized FPGA RAMs for Quartus II compilation

You only need to provide these assignments if your component is a memory device that you want to initialize.

The assignment name for the memory initialization namespace is `embeddedsw.memoryInfo.<name>`. Altera embedded software tools already have definitions for the data types of the possible values. Table 13-7 shows how to format memory initialization assignment values for all defined data types.

**Table 13-7.** Memory Initialization Data Types

Memory Initialization Data Type	Format
boolean	1, 0
32-bit integer	123, -50
string (1)	ABC

**Note to Table 13-7:**

(1) Quotation marks are not required.

Memory initialization assignments are associated with an entire component.

Table 13-8 shows the embedded software memory initialization names.

**Table 13-8.** Memory Initialization Information - Assign with set\_module\_assignment Command

Memory Initialization Name	Type	Default	Meaning
HAS_BYTE_LANE	boolean	0	Create a memory initialization file for each byte.
IS_FLASH	boolean	0	Component is a flash device.
IS_EPCS	boolean	0	If IS_FLASH and IS_EPCS are both 1, component is an EPCS flash device. If IS_FLASH is 1 and IS_EPCS is 0, the component is a CFI flash device. If IS_EPCS is 1, IS_FLASH must also be 1.
GENERATE_HEX	boolean	0	Create an Intel hexadecimal file (.hex).
GENERATE_DAT_SYM	boolean	0	Create a .dat and a .sym file.
GENERATE_FLASH	boolean	0	Create a Motorola S-record File (.flash).
INCLUDE_WARNING_MSG	string	empty string	Display a warning message when creating memory initialization files.
MEM_INIT_FILENAME	string	Module instance name	Name of the memory initialization file, without any file type suffix.
MEM_INIT_DATA_WIDTH	32-bit integer	none (mandatory)	Width of memory initialization file in bits. May be different than the slave port data width.

## Referenced Documents

This chapter references the following document:

*Component Interface Tcl Reference* chapter in *Volume 4: SOPC Builder of the Quartus II Handbook*

## Document Revision History

Table 13-9 shows the revision history for this document.

**Table 13-9.** Document Revision History

<b>Date &amp; Document Version</b>	<b>Changes Made</b>	<b>Summary of Changes</b>
November 2009 v9.1.0	Add the following Avalon-ST interface settings, to support external interrupt controllers: <ul style="list-style-type: none"><li>■ <code>embeddedsw.configuration.isInterruptControllerReceiver</code></li><li>■ <code>embeddedsw.configuration.isInterruptControllerSender</code></li><li>■ <code>embeddedsw.configuration.transportsInterruptsFromReceivers</code></li></ul>	External interrupt controller support
March 2009 v9.0.0	Initial Release.	