Nios® II

Nios II Software Developer's Handbook

ALTERA®

101 Innovation Drive
San Jose, CA 95134
www.altera.com

NII5V2-7.2

Contents

Section I. Nios II Software Development

Chapter 1. Overview

Chapter 2. Nios II Integrated Development Environment

Chapter 3. Introduction to the Nios II Software Build Tools

Chapter 4. Using the Nios II Software Build Tools

Section II. The Hardware Abstraction Layer

Chapter 5. Overview of the Hardware Abstraction Layer

Chapter 6. Developing Programs Using the Hardware Abstraction Layer

Chapter 7. Developing Device Drivers for the Hardware Abstraction Layer

Section III. Advanced Programming Topics

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

Section IV. Appendices

Chapter 12. HAL API Reference

Chapter 13. Altera-Provided Development Tools

Chapter 14. Nios II Software Build Tools Reference

Chapter 15. Read-Only Zip File System

Chapter 16. Ethernet and Lightweight IP

Chapter Revision Dates

The chapters in this book, *Nios II Software Developer's Handbook*, were revised on the following dates. Where chapters or groups of chapters are available separately, part numbers are listed.

Chapter 1. Overview
Revised: *October 2007*
Part number: *NII52001-7.2.0*

Chapter 2. Nios II Integrated Development Environment
Revised: *October 2007*
Part number: *NII52002-7.2.0*

Chapter 3. Introduction to the Nios II Software Build Tools
Revised: *October 2007*
Part number: *NII52014-7.2.0*

Chapter 4. Using the Nios II Software Build Tools
Revised: *October 2007*
Part number: *NII52015-7.2.0*

Chapter 5. Overview of the Hardware Abstraction Layer
Revised: *October 2007*
Part number: *NII52003-7.2.0*

Chapter 6. Developing Programs Using the Hardware Abstraction Layer
Revised: *October 2007*
Part number: *NII52004-7.2.0*

Chapter 7. Developing Device Drivers for the Hardware Abstraction Layer
Revised: *October 2007*
Part number: *NII52005-7.2.0*

Chapter 8. Exception Handling
Revised: *October 2007*
Part number: *NII52006-7.2.0*

Chapter 9. Cache and Tightly-Coupled Memory
Revised: *October 2007*
Part number: *NII52007-7.2.0*

Chapter 10. MicroC/OS-II Real-Time Operating System
Revised: *October 2007*
Part number: *NII52008-7.2.0*

Chapter 11. Ethernet and the NicheStack TCP/IP Stack - Nios II Edition
Revised: *October 2007*
Part number: *NII52013-7.2.0*

Chapter 12. HAL API Reference
Revised: *October 2007*
Part number: *NII52010-7.2.0*

Chapter 13. Altera-Provided Development Tools
Revised: *October 2007*
Part number: *NII520011-7.2.0*

Chapter 14. Nios II Software Build Tools Reference
Revised: *October 2007*
Part number: *NII52016-7.2.0*

Chapter 15. Read-Only Zip File System
Revised: *October 2007*
Part number: *NII520012-7.2.0*

Chapter 16. Ethernet and Lightweight IP
Revised: *October 2007*
Part number: *NII52009-7.2.0*

About this Handbook

This handbook provides comprehensive information about developing software for the Altera® Nios® II processor. This handbook does not document how to use the Nios II integrated development environment (IDE). For a complete reference on the Nios II IDE, start the IDE and open the Nios II IDE help system.

How to Contact Altera

For the most up-to-date information about Altera products, refer to the following table.

Contact *(1)*	Contact Method	Address
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	custrain@altera.com
Product literature	Website	www.altera.com/literature
Altera literature services	Email	literature@altera.com
Non-technical support (General)	Email	nacomp@altera.com
(Software Licensing)	Email	authorization@altera.com

Note to table:

(1) You can also contact your local Altera sales office or sales representative.

Typographic Conventions

This document uses the typographic conventions shown below.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Command names, dialog box titles, checkbox options, and dialog box options are shown in bold, initial capital letters. Example: **Save As** dialog box.
bold type	External timing parameters, directory names, project names, disk drive names, filenames, filename extensions, and software utility names are shown in bold type. Examples: $\mathbf{f_{MAX}}$, **\qdesigns** directory, **d:** drive, **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Document titles are shown in italic type with initial capital letters. Example: *AN 75: High-Speed Board Design.*
Italic type	Internal timing parameters and variables are shown in italic type. Examples: t_{PIA}, $n + 1$. Variable names are enclosed in angle brackets (< >) and shown in italic type. Example: <*file name*>, <*project name*>**.pof** file.
Initial Capital Letters	Keyboard keys and menu names are shown with initial capital letters. Examples: Delete key, the Options menu.
"Subheading Title"	References to sections within a document and titles of on-line help topics are shown in quotation marks. Example: "Typographic Conventions."
`Courier type`	Signal and port names are shown in lowercase Courier type. Examples: `data1`, `tdi`, `input`. Active-low signals are denoted by suffix `n`, e.g., `resetn`. Anything that must be typed exactly as it appears is shown in Courier type. For example: `c:\qdesigns\tutorial\chiptrip.gdf`. Also, sections of an actual file, such as a Report File, references to parts of files (e.g., the AHDL keyword `SUBDESIGN`), as well as logic function names (e.g., `TRI`) are shown in Courier.
1., 2., 3., and a., b., c., etc.	Numbered steps are used in a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ● •	Bullets are used in a list of items when the sequence of the items is not important.
✓	The checkmark indicates a procedure that consists of one step only.
☞	The hand points to information that requires special attention.
⚠ CAUTION	A caution calls attention to a condition or possible situation that can damage or destroy the product or the user's work.
⚠ WARNING	A warning calls attention to a condition or possible situation that can cause injury to the user.
↵	The angled arrow indicates you should press the Enter key.
👣	The feet direct you to more information on a particular topic.

Section I. Nios II Software Development

This section introduces information for Nios® II software development.

This section includes the following chapters:

1. Overview

NII52001-7.2.0

Introduction

This chapter provides the software developer with a high-level overview of the software development environment for the Nios® II processor. This chapter introduces the Nios II software development environment, the Nios II embedded design suite (EDS) tools available to you, and the process for developing software. This chapter contains the following sections:

Getting Started

Writing software for the Nios II processor is similar to the software development process for any other microcontroller family. The easiest way to start designing effectively is to purchase a development kit from Altera® that includes documentation, a ready-made evaluation board, and all the development tools necessary to write Nios II programs.

The *Nios II Software Developer's Handbook* assumes you have a basic familiarity with embedded processor concepts. You do not need to be familiar with any specific Altera technology or with Altera development tools. Familiarity with Altera hardware development tools can give you a deeper understanding of the reasoning behind the Nios II software development environment. However, software developers can develop and debug applications without further knowledge of Altera technology.

Modifying existing code is perhaps the most common and comfortable way that software designers learn to write programs in a new environment. The Nios II EDS provides many example software designs that you can examine, modify, and use in your own programs. The provided examples range from a simple "Hello world" program, to a working real-time operating system (RTOS) example, to a full transmission control protocol/Internet protocol (TCP/IP) stack running a web server. Each example is documented and ready to compile.

Nios II Software Development Environment

The Nios II EDS provides a consistent software development environment that works for all Nios II processor systems. With a PC, an Altera FPGA, and a Joint Test Action Group (JTAG) download cable (such as an Altera USB-Blaster™ download cable), you can write programs for, and communicate with, any Nios II processor system. The Nios II processor's JTAG debug module provides a single, consistent method to communicate with the processor using a JTAG download cable. Accessing the processor is the same, regardless of whether a device implements only a Nios II processor system, or whether the Nios II processor is embedded deeply in a complex multiprocessor system. Therefore, you do not need to spend time manually creating interface mechanisms for the embedded processor.

The Nios II EDS provides two distinct design flows and includes many proprietary and open-source tools (such as the GNU C/C++ tool chain) for creating Nios II programs. The Nios II EDS automates board support package (BSP) creation for Nios II-based systems, eliminating the need to spend time manually creating BSPs. Altera BSPs contain the Altera hardware abstraction layer (HAL), an optional RTOS, and device drivers. The BSP provides a C/C++ runtime environment, insulating you from the hardware in your embedded system.

Nios II Programs

Each Nios II program you develop in either Nios II EDS design flow consists of an application project, optional library projects, and a BSP project. You build your Nios II program into an Executable And Linked Format File (**.elf)** which runs on a Nios II processor. While terminology sometimes differs between the two design flows, the Nios II programs you develop are conceptually equal.

The following sections describe the project types that comprise a Nios II program:

Application Project

A Nios II C/C++ application project consists of a collection of source code combined into one executable (**.elf)** file. A typical characteristic of an application is that one of the source files contains function `main()`. An application includes code that calls functions in libraries and BSPs.

Library Project

A library project is a collection of source code contained within a single library archive (**.a**) file. Libraries often contain reusable, general purpose functions that multiple application projects can share. A collection of common arithmetical functions is one example. A library does not contain a function `main()`.

BSP Project

A Nios II BSP project is a specialized library containing system-specific support code. A BSP provides a software runtime environment customized for one processor in an SOPC Builder system. The Nios II EDS provides tools to modify settings that control the behavior of the BSP.

☞ The Nios II integrated development environment (IDE) and the Nios II IDE design flow documentation use the term "system library" when referring to a BSP.

A BSP contains the following elements:

- Hardware Abstraction Layer (HAL)
- Newlib C Standard Library
- Device Drivers
- Optional Software Packages
- Optional Real-Time Operating System (RTOS)

Hardware Abstraction Layer (HAL)

The HAL provides a non-threaded, UNIX-like, C/C++ runtime environment. The HAL provides generic I/O devices, allowing you to write programs that access hardware using the newlib C standard library routines, such as `printf()`. The HAL minimizes (or eliminates) the need to access hardware registers directly to control and communicate with peripherals.

For complete details about the HAL, refer to the *The Hardware Abstraction Layer* section and the *HAL API Reference* chapter of the *Nios II Software Developer’s Handbook.*

Newlib C Standard Library

Newlib is an open source implementation of the C standard library intended for use on embedded systems. It is a collection of common routines such as `printf()`, `malloc()`, and `open()`.

Device Drivers

Each device driver manages a hardware component. By default, the HAL instantiates a device driver for each component in your SOPC Builder system that needs a device driver. In the Nios II software development environment, a device driver has the following properties:

- A device driver is associated with a specific SOPC Builder component.
- A device driver might have settings that impact its compilation. These settings become part of the BSP settings.

Optional Software Packages

A software package is source code that you can optionally add to a BSP project to provide additional functionality. The NicheStack® TCP/IP - Nios II Edition is an example of a software package.

☞ The Nios II IDE and the Nios II IDE design flow documentation use the term "software component" when referring to a software package.

In the Nios II software development environment, a software package typically has the following properties:

- A software package is not associated with specific hardware.
- A software package might have settings that impact its compilation. These settings become part of the BSP settings.

☞ In the Nios II software development environment, a software package is distinct from a library project. A software package is part of the BSP project, not a separate library project.

Optional Real-Time Operating System (RTOS)

The Nios II EDS includes an implementation of the third-party MicroC/OS-II RTOS that you can optionally include in your BSP. MicroC/OS-II is built on the HAL, and implements a simple, well-documented RTOS scheduler. You can modify settings that become part of the BSP settings. Other operating systems are available from third-party vendors.

Design Flows for Creating Nios II Programs

The Nios II EDS provides two distinct design flows for creating Nios II programs. You can work entirely within the Nios II integrated development environment (IDE), or you can use the Nios II software build tools in command line and scripted environments and then import your work into the IDE for debugging.

The two design flows are not interchangeable. Source code for your applications, libraries, and drivers works in either flow, but the makefiles in the two flows are different and not compatible. Once you have committed to using one design flow, you cannot switch to using the other design flow for that project without starting over.

The Nios II IDE Design Flow

In the Nios II IDE design flow, you create, modify, build, run, and debug Nios II programs with the Nios II IDE graphical user interface (GUI). The IDE creates and manages your project makefiles for you. This design flow is best if you only need limited control over the build process and the project settings, and do not require customized scripting.

The Nios II IDE is based on the popular Eclipse IDE framework and the Eclipse C/C++ development toolkit (CDT) plug-ins. The Nios II IDE runs other tools behind the scenes, shields you from the details of low-level tools, and presents a unified development environment.

With wizards to assist in creating and configuring projects, the Nios II IDE is easy to use, and is especially helpful for Nios II beginners. The Nios II IDE is available on both Windows and Linux operating systems.

For details about the Nios II IDE, refer to the *Nios II Integrated Development Environment* chapter of the *Nios II Software Developer's Handbook.*

The Nios II Software Build Tools Design Flow

In the Nios II software build tools design flow, you create, modify, build, and run Nios II programs with commands typed at a command line or embedded in a script. This design flow is best if you need fine control over the build process and the project settings, or if you require customized scripting.

The Nios II software build tools are utilities and scripts that provide similar functionality to the **New Project** wizard and the **System Library** properties page in the Nios II IDE. The Nios II software build tools design flow allows you to integrate your Nios II software development with other parts of your development flow. Using scripting, your software development flow is fully repeatable and archivable.

At debug time, you import your Nios II software build tools projects into the IDE as Nios II IDE projects for debugging. You can further edit, rebuild, run, and debug your imported application project in the IDE. You can also import library and BSP projects to be available for source code viewing in the debugger, however, imported library and BSP projects are not directly buildable in the IDE.

Like the Nios II IDE, the Nios II software build tools are available on both Windows and Linux operating systems. The Nios II software build tools are the basis for Altera's future Nios II development.

For further information about the Nios II software build tools, refer to the *Introduction to the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*

Design Flow Tools

This section introduces the tools you use to create Nios II programs for each design flow. The tables are organized to help you determine the level of control you need. You can use the tables as a quick-reference guide to remind you of the differences between the design flows.

Table 1–1 shows the tools that offer a highly-automated level of control for tasks in each Nios II design flow. At this level of control, you create an entire example Nios II program (consisting of an application project and BSP project) or just an example BSP project from Altera-provided software examples using default settings. Use the highly-automated tools as a starting point for your development or when you do not need customization.

Table 1–1. Highly-Automated Design Flow Tools

Task	Nios II IDE Design Flow	Nios II Software Build Tools Design Flow
Building an example Nios II program	**File > New > Nios II C/C++ Application**	**create-this-app** script
Building an example BSP	**File > New > Nios II System Library**	**create-this-bsp** script
Debugging	**Run > Debug As > Nios II Hardware**	● **File > Import > Altera Nios II > Existing Nios II software build tools project or folder into workspace** ● **Run > Debug As > Nios II Hardware**

Table 1–2 shows the tools that offer an intermediate level of control for tasks in each Nios II design flow. At this level of control, you create a Nios II program from your custom code or from Altera-provided

software examples. Use the intermediate tools as a starting point for your development when you need more control than the default settings provide.

Table 1–2. Intermediate Design Flow Tools

Task	Nios II IDE Design Flow	Nios II Software Build Tools Design Flow
Building an application	**File** > **New** > **Nios II C/C++ Application**	**nios2-app-generate-makefile** utility
Building a library	**File** > **New** > **Nios II C/C++ Library**	**nios2-lib-generate-makefile** utility
Building a BSP	● **File** > **New** > **Nios II System Library** ● **Project** > **Properties** > **System Library**	**nios2-bsp** script
Debugging	**Run** > **Debug As** > **Nios II Hardware**	● **File** > **Import** > **Altera Nios II** > **Existing Nios II software build tools project or folder into workspace** ● **Run** > **Debug As** > **Nios II Hardware**

Table 1–3 shows the tools that offer an advanced level of control for tasks related to BSPs in each Nios II design flow. At this level of control, you create a Nios II BSP with sophisticated, scriptable control. Use the advanced tools when you need total control over the BSP build process and the BSP project settings.

Table 1–3. Advanced BSP Design Flow Tools

Task	Nios II IDE Design Flow	Nios II Software Build Tools Design Flow
Building a BSP	Scriptable control of a system library project is not supported.	● **nios2-bsp-create-settings** utility ● **nios2-bsp-generate-files** utility ● Tcl scriptable
Updating a BSP	Scriptable control of a system library project is not supported.	● **nios2-bsp-update-settings** utility ● **nios2-bsp-generate-files** utility ● Tcl scriptable
Querying a BSP	Not supported.	● **nios2-bsp-query-settings** utility ● Tcl scriptable
Customizing newlib	Not supported.	**nios2-bsp** script using **CUSTOM_NEWLIB_FLAGS** setting

Additional EDS Support

In addition to the Nios II IDE and Nios II software build tools, the Nios II EDS includes the following items:

- GNU Tool Chain
- Instruction Set Simulator
- Example Designs

GNU Tool Chain

The Nios II compiler tool chain is based on the standard GNU gcc compiler, assembler, linker, and make facilities.

For more information on GNU, see **www.gnu.org**.

Instruction Set Simulator

The Nios II instruction set simulator (ISS) allows you to begin developing programs before the target hardware platform is ready. The Nios II IDE allows you to run programs on the ISS as easily as running on a real hardware target.

Example Designs

The Nios_II EDS includes software examples and hardware designs to demonstrate all prominent features of the Nios II processor and the development environment.

Third-Party Support

Several third-party vendors support the Nios II processor, providing products such as design services, operating systems, stacks, other software libraries, and development tools.

For the most up-to-date information on third-party support for the Nios II processor, visit the Nios II processor home page at **www.altera.com/nios2**.

Migrating from the First-Generation Nios Processor

If you are a user of the first-generation Nios processor, Altera recommends that you migrate to the Nios II processor for future designs. The straightforward migration process is discussed in *AN 350: Upgrading Nios Processor Systems to the Nios II Processor*.

Further Nios II Information

This handbook is one part of the complete Nios II processor documentation suite. Consult the following references for further Nios II information:

- The *Nios II Processor Reference Handbook* defines the processor hardware architecture and features, including the instruction set architecture.

- The *Quartus® II Handbook, Volume 5: Embedded Peripherals* provides a reference for the peripherals distributed with the Nios II processor. This handbook describes the hardware structure and Nios II software drivers for each peripheral.
- The Nios II IDE provides tutorials and complete reference for using the features of the GUI. The help system is available within the Nios II IDE.
- The Altera Knowledge Database is an Internet resource that offers solutions to frequently asked questions via an easy-to-use search engine. Go to **answers.altera.com/altera/index.jsp.**
- Altera application notes and tutorials offer step-by-step instructions on using the Nios II processor for a specific application or purpose. These documents are available on the Literature: Nios II Processor page at **www.altera.com/literature/lit-nio2.jsp.**

Referenced Documents

This chapter references the following documents:

- *The Hardware Abstraction Layer* section of the *Nios II Software Developer's Handbook*
- *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*
- *Nios II Integrated Development Environment* chapter of the *Nios II Software Developer's Handbook*
- *Introduction to the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*
- *AN 350: Upgrading Nios Processor Systems to the Nios II Processor*
- *Nios II Processor Reference Handbook*
- *Quartus II Handbook, Volume 5: Embedded Peripherals*

Document Revision History

Table 1–4 shows the revision history for this document.

Table 1–4. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	No change from previous release.	
May 2007 v7.1.0	● Revised entire chapter to introduce Nios II EDS design flows, Nios II programs, Nios II software build tools, and Nios II BSPs. ● Added table of contents to Introduction section. ● Added Referenced Documents section.	Nios II software build tools
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.	
May 2004 v1.0	Initial Release.	

2. Nios II Integrated Development Environment

NII52002-7.2.0

Introduction

This chapter familiarizes you with the main features of the Nios® II integrated development environment (IDE). This chapter is only a brief introduction to the look and feel of the Nios II IDE—it is not a user guide. The easiest way to get started using the Nios II IDE is to launch the tool and perform the Nios II software development tutorial, available in the help system.

☞ The figures in this chapter may differ slightly from the actual GUI due to improvements in the software.

For more information on all IDE-related topics, refer to the Nios II IDE help system.

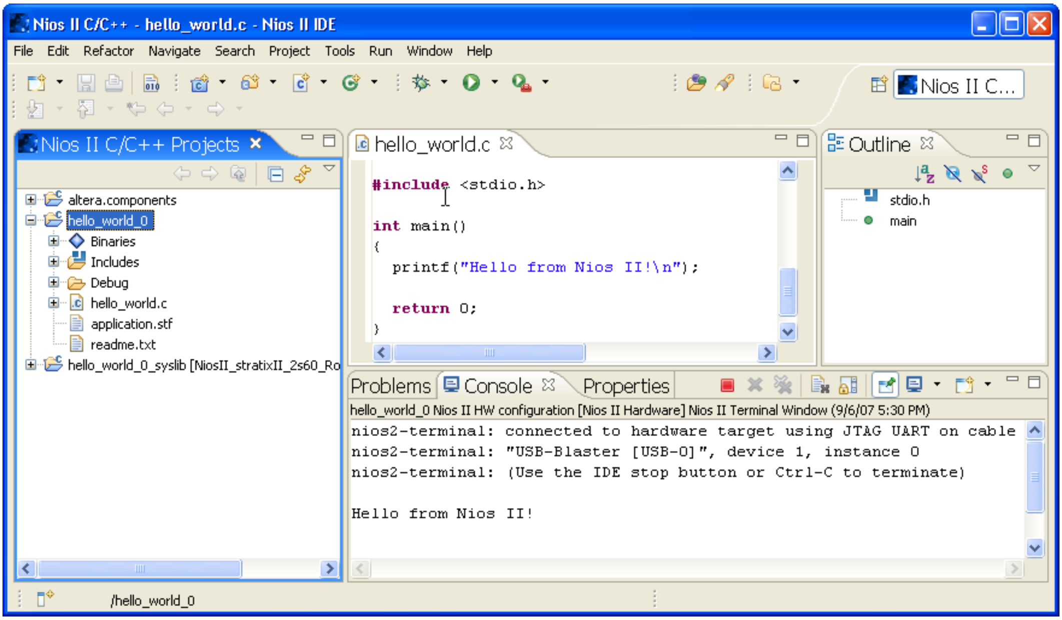
This chapter contains the following sections:

- "The Nios II IDE Workbench" on page 2–1
- "EDS Design Flows and the IDE" on page 2–3
- "Creating a New IDE-Managed Project" on page 2–4
- "Building and Managing Projects" on page 2–6
- "Running and Debugging Programs" on page 2–8
- "Importing User-Managed Projects" on page 2–11
- "Programming Flash" on page 2–20
- "Help System" on page 2–22

The Nios II IDE Workbench

The term "workbench" refers to the desktop development environment for the Nios II IDE. The workbench is where you edit, compile and debug your programs in the IDE. Figure 2–1 on page 2–2 shows an example of the workbench.

Figure 2–1. The Nios II IDE Workbench



```
#include <stdio.h>

int main()
{
  printf("Hello from Nios II!\n");

  return 0;
}
```

hello_world_0 Nios II HW configuration [Nios II Hardware] Nios II Terminal Window (9/6/07 5:30 PM)

```
nios2-terminal: connected to hardware target using JTAG UART on cable
nios2-terminal: "USB-Blaster [USB-0]", device 1, instance 0
nios2-terminal: (Use the IDE stop button or Ctrl-C to terminate)

Hello from Nios II!
```

Perspectives, Editors, and Views

Each workbench window contains one or more perspectives. Each perspective provides a set of capabilities for accomplishing a specific type of task. For example, Figure 2–1 shows the Nios II C/C++ development perspective.

Most perspectives in the workbench comprise an editor area and one or more views. An editor allows you to open and edit a project resource (i.e., a file, folder, or project). Views support editors, provide alternative presentations, and ways to navigate the information in your workbench. Figure 2–1 shows a C program open in the editor, and the Nios II C/C++ Projects view in the left-hand pane of the workbench. The Nios II C/C++ Projects view displays information about the contents of open Nios II projects.

Any number of editors can be open at once, but only one can be active at a time. The main menu bar and toolbar for the workbench window contain operations that are applicable to the active editor. Tabs in the editor area indicate the names of resources that are currently open for editing. An asterisk (*) indicates that an editor has unsaved changes. Views can also provide their own menus and toolbars, which, if present, appear along the top edge of the view. To open the menu for a view, click the drop-down arrow icon at the right of the view's toolbar or right-click in the view. A view might appear on its own, or stacked with other views in a tabbed notebook.

EDS Design Flows and the IDE

The Nios II IDE is an integral part of both Nios II embedded design suite (EDS) design flows. The main distinction between the two design flows is in the management of the project.

IDE-Managed Projects and Makefiles

In the Nios II IDE design flow, the IDE manages Nios II C/C++ application and system library projects and makefiles that you create with the **New Project** wizard in Nios II IDE. In IDE-managed projects, the IDE manages the makefiles for you. The best way to modify and build an IDE-managed project is through the IDE. You manage the system library project settings with the **System Library** page of the **Properties** dialog box.

You can manually convert an IDE-managed project to a user-managed project. For details, see the *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

User-Managed Projects and Makefiles

In the Nios II software build tools design flow, you manage Nios II application, library, and BSP projects and makefiles, giving you total control. Typically, you create user-managed projects outside of the IDE and then import them into the IDE for debugging.

The best way to create a user-managed project, and modify the settings, is with the Nios II software build tools through a command shell or scripting tool. You can create the makefile by hand, or you can use the Nios II software build tools to create it.

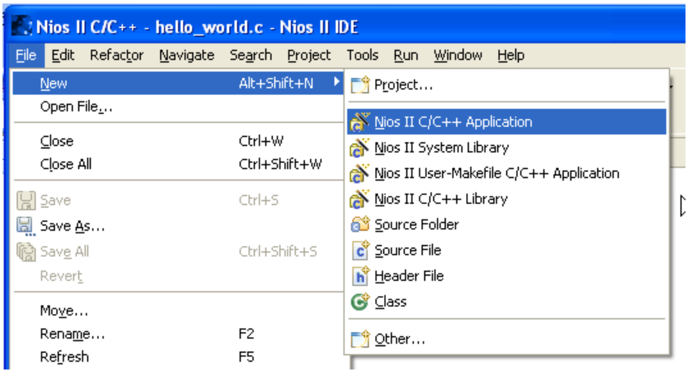
When you import a user-managed C/C++ application project into the IDE for debugging, the Nios II IDE does not manage the project or makefile. The IDE does not support management of or changes to a BSP project after import. You must manage the BSP with the software build tools, outside of the IDE.

☞ User-managed projects and IDE-managed projects are not interchangeable.

Creating a New IDE-Managed Project

The Nios II IDE provides a **New Project** wizard that guides you through the steps to create new IDE-managed projects. To start the **New Project** wizard for Nios II C/C++ application projects, on the File menu in the Nios II C/C++ perspective, point to **New**, and then click **Nios II C/C++ Application**, as shown in Figure 2–2.

Figure 2–2. Starting the Nios II C/C++ Application New Project Wizard



The Nios II C/C++ application **New Project** wizard prompts you to specify:

1. A name for your new Nios II project.

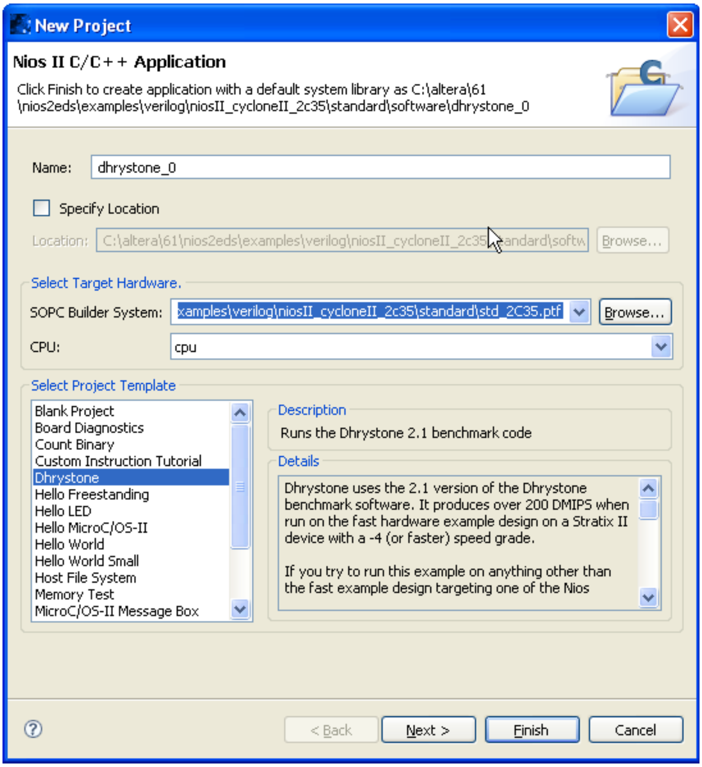
2. The target hardware.

3. A template for the new project.

Project templates are ready-made, working designs that serve as examples to show you how to structure your own Nios II projects. It is often easier to start with a working “Hello World” project, than to start a blank project from scratch.

Figure 2–3 shows the Nios II C/C++ application **New Project** wizard, with the template for a Dhrystone benchmark design selected.

Figure 2–3. The Nios II C/C++ Application New Project Wizard



Nios II C/C++ Application

Click Finish to create application with a default system library as C:\altera\61\nios2eds\examples\verilog\niosII_cycloneII_2c35\standard\software\dhrystone_0

Name: dhrystone_0

□ Specify Location

Location: C:\altera\61\nios2eds\examples\verilog\niosII_cycloneII_2c35\standard\softw

Select Target Hardware.

SOPC Builder System: xamples\verilog\niosII_cycloneII_2c35\standard\std_2C35.ptf

CPU: cpu

Select Project Template

- Blank Project
- Board Diagnostics
- Count Binary
- Custom Instruction Tutorial
- Dhrystone
- Hello Freestanding
- Hello LED
- Hello MicroC/OS-II
- Hello World
- Hello World Small
- Host File System
- Memory Test
- MicroC/OS-II Message Box

Description

Runs the Dhrystone 2.1 benchmark code

Details

Dhrystone uses the 2.1 version of the Dhrystone benchmark software. It produces over 200 DMIPS when run on the fast hardware example design on a Stratix II device with a -4 (or faster) speed grade.

If you try to run this example on anything other than the fast example design targeting one of the Nios

After you click **Finish**, the Nios II IDE creates the new project. The IDE also creates a system library project, ***_syslib** (for example, **dhrystone_0_syslib**for Figure 2–3 on page 2–5). These projects show up in the Nios II C/C++ Projects view of the workbench.

☞ The first time you create or build a Nios II project, the Nios II IDE automatically creates a project in your workspace called **altera.components**. This project contains links to the source code files for all Altera®-provided device drivers and software packages, enabling you to step through system code in the debugger, set breakpoints, and use other debugger features. The **altera.components** project appears in the Nios II C/C++ Projects view.

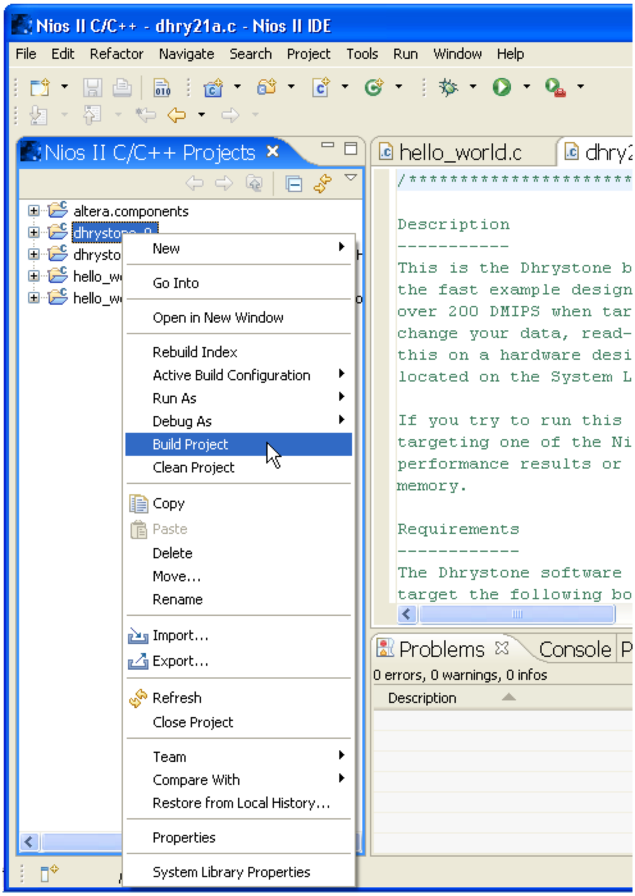
The Nios II C/C++ view protects the source files in **altera.components** from accidental deletion, because they are shared among all software projects. Do not attempt to circumvent this protection.

Building and Managing Projects

Right-clicking on any resource (a file, folder, or project) opens a context-sensitive menu containing commands that you can perform on the resource. Right-clicking is usually the quickest way to find the command you need, though commands are also available in menus and toolbars.

To compile a Nios II project, right-click the project in the Nios II C/C++ Projects view, and click **Build Project**. Figure 2–4 on page 2–7 shows the context-sensitive menu for the project `dhrystone_0`, with the **Build Project** option chosen. When building, the Nios II IDE first builds the system library project (and any other project dependencies), and then compiles the main project. Any warnings or errors are displayed in the Tasks view.

Figure 2–4. Building a Project Using the Context-Sensitive (Right-Click) Menu



```
/**********************

Description
-----------
This is the Dhrystone b
the fast example design
over 200 DMIPS when tar
change your data, read-
this on a hardware desi
located on the System L

If you try to run this
targeting one of the Ni
performance results or
memory.

Requirements
------------
The Dhrystone software
target the following bo
```

0 errors, 0 warnings, 0 infos

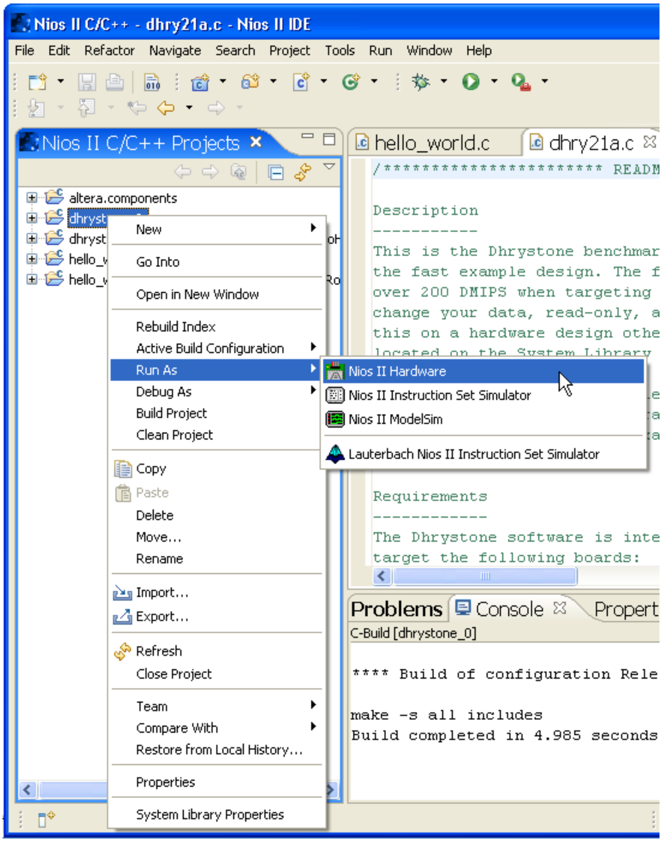
Right-clicking a project in the Nios II C/C++ Projects view also allows you to access the following important options for managing the project:

- **Properties**—Manage the dependencies on target hardware and other projects
- **System Library Properties**—Manage hardware-specific settings, such as communication devices and memory partitioning
- **Build Project**—i.e., make
- **Run As**—Run the program on hardware or under simulation
- **Debug As**—Debug the program on hardware or under simulation

Running and Debugging Programs

Run and debug operations are available by right-clicking the Nios II project. The Nios II IDE allows you to run or debug the project either on a target board, under the Nios II instruction set simulator (ISS), or under ModelSim®. For example, to run the program on a target board, right-click the project in the Nios II C/C++ Projects view, point to Run As, and then click **Nios II Hardware**. See Figure 2–5. Character I/O to `stdout` and `stderr` are displayed in the Console view.

Figure 2–5. Running a Program on Target Hardware

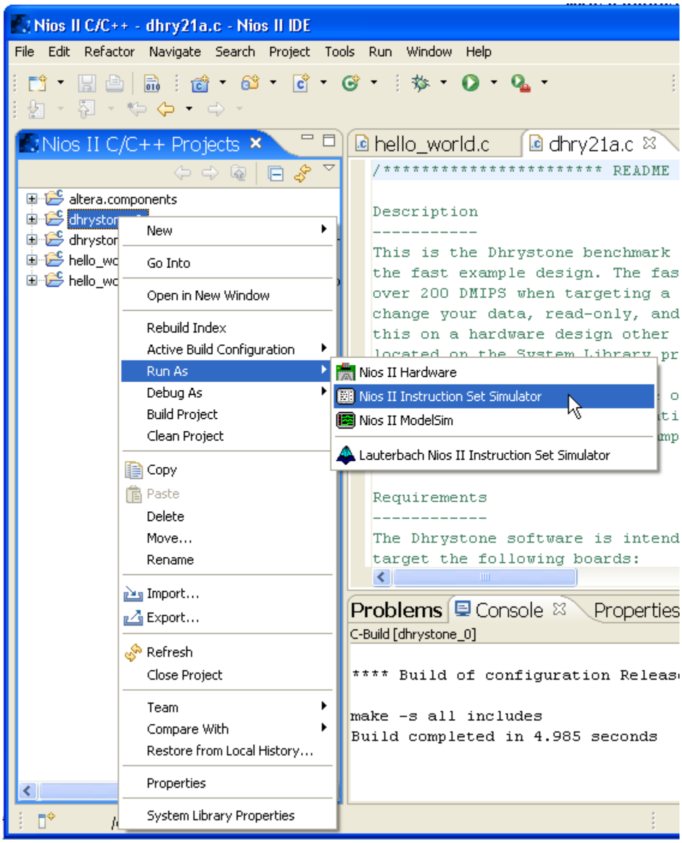


```
/*********************** READM

Description
-----------
This is the Dhrystone benchmar
the fast example design. The f
over 200 DMIPS when targeting
change your data, read-only, a
this on a hardware design othe
located on the System Library

Requirements
------------
The Dhrystone software is inte
target the following boards:
```

```
C-Build [dhrystone_0]

**** Build of configuration Rele

make -s all includes
Build completed in 4.985 seconds
```

Starting a debug session is similar to starting a run session. For example, to debug the program on the ISS, right-click the project in the Nios II C/C++ Projects view, point to **Debug As**, and then click **Nios II Instruction Set Simulator**. See Figure 2–6 on page 2–9.

Figure 2–6. Launching the Instruction Set Simulator



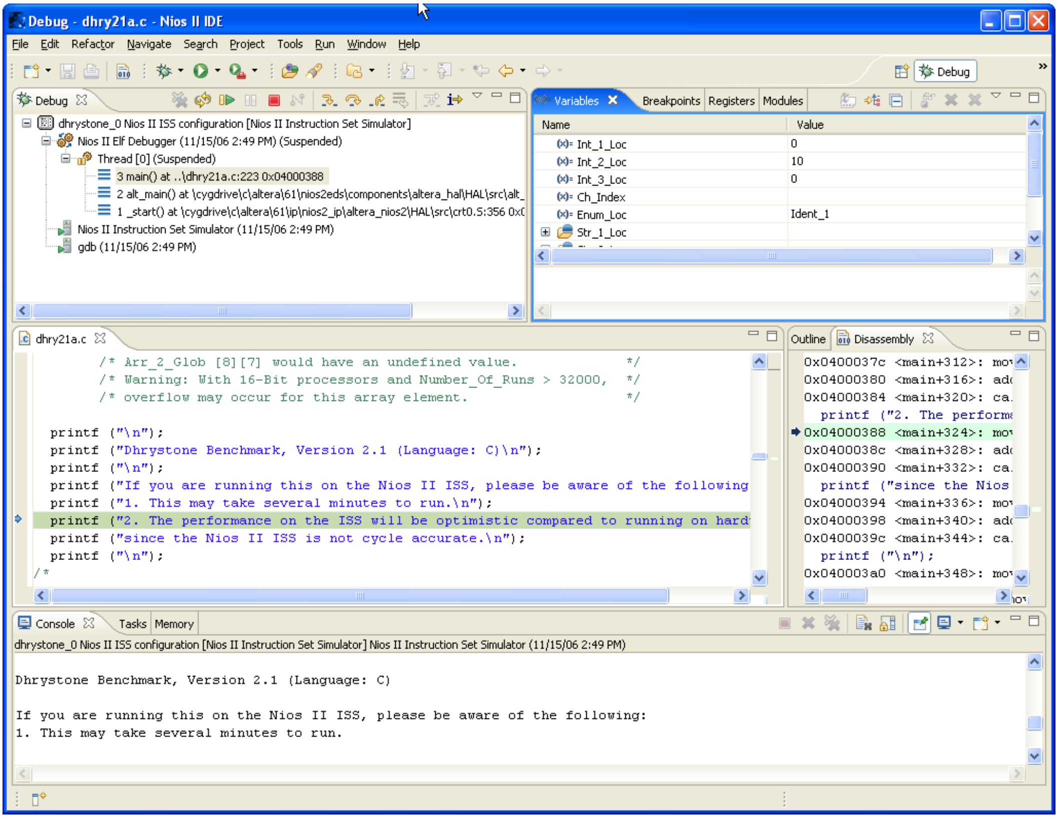
```
/*********************** README

Description
-----------
This is the Dhrystone benchmark
the fast example design. The fas
over 200 DMIPS when targeting a
change your data, read-only, and
this on a hardware design other
located on the System Library pr
```

```
Requirements
------------
The Dhrystone software is intend
target the following boards:
```

```
**** Build of configuration Releas

make -s all includes
Build completed in 4.985 seconds
```

Figure 2–7 on page 2–10 shows a debug session in progress for the `dhrystone_0` project.

Figure 2–7. Debugging dhrystone_0 on the ISS


Debug - dhry21a.c - Nios II IDE
File Edit Refactor Navigate Search Project Tools Run Window Help
Debug
Debug
Variables
Breakpoints
Registers
Modules
dhry21a.c
Outline
Disassembly
Console
Tasks
Memory

```
dhrystone_0 Nios II ISS configuration [Nios II Instruction Set Simulator]
  Nios II Elf Debugger (11/15/06 2:49 PM) (Suspended)
    Thread [0] (Suspended)
      3 main() at ..\dhry21a.c:223 0x04000388
      2 alt_main() at \cygdrive\c\altera\61\nios2eds\components\altera_hal\HAL\src\alt_
      1 _start() at \cygdrive\c\altera\61\ip\nios2_ip\altera_nios2\HAL\src\crt0.S:356 0x0
  Nios II Instruction Set Simulator (11/15/06 2:49 PM)
  gdb (11/15/06 2:49 PM)
```

Name	Value
Int_1_Loc	0
Int_2_Loc	10
Int_3_Loc	0
Ch_Index	
Enum_Loc	Ident_1
Str_1_Loc	

```
        /* Arr_2_Glob [8][7] would have an undefined value.          */
        /* Warning: With 16-Bit processors and Number_Of_Runs > 32000,  */
        /* overflow may occur for this array element.                */

    printf ("\n");
    printf ("Dhrystone Benchmark, Version 2.1 (Language: C)\n");
    printf ("\n");
    printf ("If you are running this on the Nios II ISS, please be aware of the following
    printf ("1. This may take several minutes to run.\n");
    printf ("2. The performance on the ISS will be optimistic compared to running on hard
    printf ("since the Nios II ISS is not cycle accurate.\n");
    printf ("\n");
/*
```

```
0x0400037c <main+312>: mov
0x04000380 <main+316>: add
0x04000384 <main+320>: ca.
  printf ("2. The performe
0x04000388 <main+324>: mov
0x0400038c <main+328>: add
0x04000390 <main+332>: ca.
  printf ("since the Nios
0x04000394 <main+336>: mov
0x04000398 <main+340>: add
0x0400039c <main+344>: ca.
  printf ("\n");
0x040003a0 <main+348>: mov
```

dhrystone_0 Nios II ISS configuration [Nios II Instruction Set Simulator] Nios II Instruction Set Simulator (11/15/06 2:49 PM)

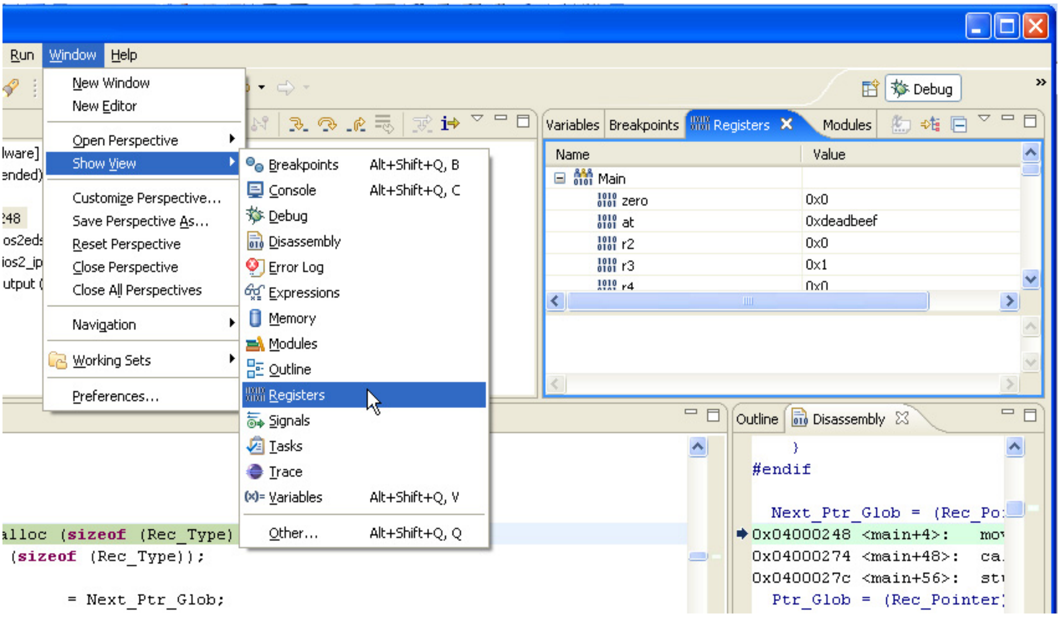
```
Dhrystone Benchmark, Version 2.1 (Language: C)

If you are running this on the Nios II ISS, please be aware of the following:
1. This may take several minutes to run.
```

Launching the debugger changes the workbench perspective to the debug perspective. You can easily switch between the debug perspective and the Nios II C/C++ development perspective by clicking on the **Open Perspective** icon at the upper right corner of the workbench window.

After you start a debug session, the debugger loads the program, sets a breakpoint at `main()`, and begins executing the program. You use the usual controls to step through the code: Step Into, Step Over, Resume, Terminate, etc. To set a breakpoint, double click in the left-hand margin of the code view, or right-click in the margin and then click **Add Breakpoint**.

The Nios II IDE offers many views that allow you to examine the status of the processor while debugging: Variables, Expressions, Registers, Memory, etc. Figure 2–8 on page 2–11 shows the Registers view.

Figure 2–8. The Registers View While Debugging



Name	Value
Main	
zero	0x0
at	0xdeadbeef
r2	0x0
r3	0x1
r4	0x0

```
alloc (sizeof (Rec_Type)
 (sizeof (Rec_Type));

      = Next_Ptr_Glob;
```

```
    }
#endif

  Next_Ptr_Glob = (Rec_Po
0x04000248 <main+4>:   mov
0x04000274 <main+48>:  ca.
0x0400027c <main+56>:  stv
  Ptr_Glob = (Rec_Pointer)
```

Importing User-Managed Projects

In the Nios II software build tools design flow, you import user-managed projects, created with the Nios II software build tools, into the IDE for debugging. This section discusses that process.

When you create a C/C++ application (and its associated BSP) with the Nios II software build tools, the application is ready to import into the IDE as a user-managed project. No additional preparation is necessary.

The IDE imports four kinds of Nios II software build tools projects:

- User-managed C/C++ application project
- User-managed board support package (BSP) project
- User-managed library project
- C/C++ source project (a directory tree containing supporting source code)

The IDE treats each type of imported project as listed in Table 2–1.

Table 2–1. IDE Capabilities for Imported Projects

Type of project	Source editable?	Buildable?	Debuggable?	Settings Manageable?
User-managed C/C++ application project	Yes	Yes	Yes	No
User-managed BSP project	Yes	No*(1)*	*(2)*	No
User-managed library project	Yes	No*(1)*	*(2)*	No
C/C++ source project	Yes	No	*(2)*	No

Notes to Table 2–1:

(1) When the IDE builds a C/C++ application project, it also builds the associated BSP, and any associated libraries. It is not necessary to import BSPs and libraries to build them as part of a C/C++ application in the IDE.

(2) When the IDE debugs a C/C++ application, it can step into any associated BSP, library, or supporting C/C++ source code that you have imported.

The IDE imports each type of project through the **Import** wizard. The **Import** wizard determines the kind of project you are importing, and configures it appropriately.

You can continue to develop project code in your user-managed project after importing the project into the IDE. You can edit source files and rebuild the project, using either the IDE tool chain or the software build tools. However, you must manage BSP settings with the software build tools.

For further information about creating projects with the software build tools, refer to the *Introduction to the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*

Road Map

Importing and debugging a project typically involves several of the following tasks. You do not need to perform these tasks in this order, and you can repeat or omit some tasks, depending on your needs.

- Import a user-managed C/C++ application
- Import a supporting project
- Debug a user-managed C/C++ application
- Edit user-managed C/C++ application code

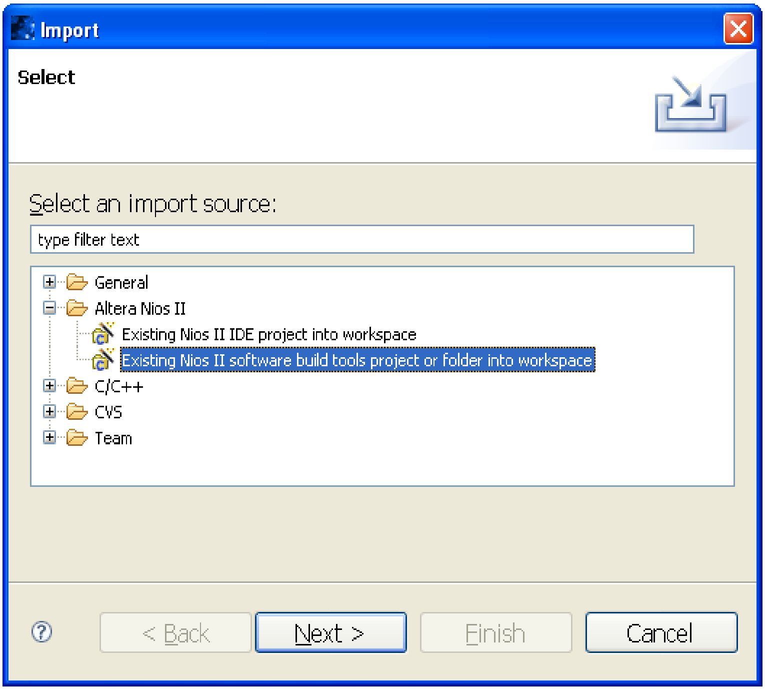
The following sections describe these tasks.

Import a User-Managed C/C++ Application

To import a user-managed C/C++ application, perform the following steps:

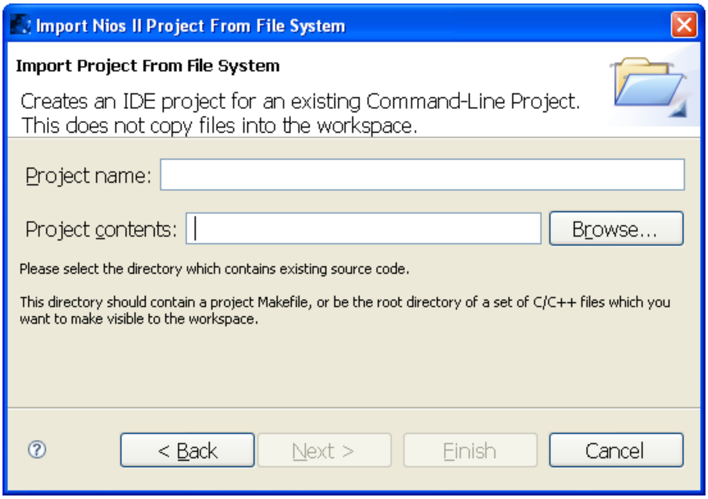
1. Launch the IDE.
2. On the File menu, click **Import**. The **Import** dialog box appears.

Figure 2–9. Launching the Import Wizard


Import
Select
Select an import source:
type filter text
General
Altera Nios II
Existing Nios II IDE project into workspace
Existing Nios II software build tools project or folder into workspace
C/C++
CVS
Team
< Back
Next >
Finish
Cancel

3. Expand the **Altera Nios II** folder, and select **Existing Nios II software build tools project or folder into workspace**, as shown in Figure 2–9.
4. Click **Next**. The **Import** wizard appears, as shown in Figure 2–10 on page 2–14.

Figure 2–10. User-Managed Project Import Wizard



Import Nios II Project From File System

Import Project From File System

Creates an IDE project for an existing Command-Line Project.
This does not copy files into the workspace.

Project name:

Project contents:

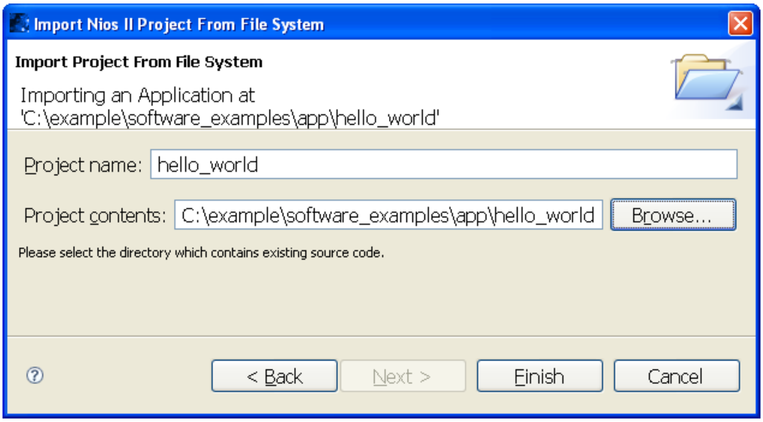
Please select the directory which contains existing source code.

This directory should contain a project Makefile, or be the root directory of a set of C/C++ files which you want to make visible to the workspace.

5. Click **Browse** and locate the directory containing the C/C++ application project to import.

6. Click **OK**. The wizard fills in the project name and path, as shown in Figure 2–11. The project name is the directory name. You can override the project name by typing a new name in the **Project name** box.

Figure 2–11. Importing a User-Managed C/C++ Application



Import Nios II Project From File System

Import Project From File System

Importing an Application at
'C:\example\software_examples\app\hello_world'

Project name: hello_world

Project contents: C:\example\software_examples\app\hello_world

Please select the directory which contains existing source code.

☞ You might see a warning saying "There is already a **.project** file at: <*path*>". This warning indicates that the directory already contains an IDE project. Either it is an IDE-managed project, or it is a user-managed project that is already imported into the IDE.

If the project is not already in your workspace, you can import it, but be aware that the **Import** wizard does not convert it from one type to another.

If the project is already in your workspace, do not re-import it.

7. Click **Finish**. The wizard imports the project, creating a new C/C++ application project in the workspace.

At this point, the IDE can build, debug, and run the complete program, including the BSP and any libraries, by using the user-managed makefiles in your imported C/C++ application project. The IDE can display and step through application source code, exactly as if the project were IDE-managed. However, the IDE does not have direct information about where BSP or library code resides. If you need to view, debug or step through BSP or library source code, you need to import the BSP or library. The process of importing supporting projects, such as BSPs and libraries, is described in the next section.

Import a Supporting Project

While debugging a C/C++ application, you might need to view, debug or step through source code in a supporting project, such as a BSP or library. To make supporting project source code visible in the IDE debug perspective, you need to import the supporting project.

If you do not need BSP or library source code visible in the debugger, you can skip this task, and go directly to "Debug a User-Managed C/C++ Application" on page 2–18.

☞ To debug or step through the code in the GNU newlib library, import the **nios2-gnutools** folder as a C/C++ source project, as described in "Importing a C/C++ Source Project" on page 2–17.

If you have several C/C++ applications based on one BSP or library, import the BSP or library once, and then import each application that is based on the BSP or library. Each application's makefile contains the information needed to find and build any associated BSP or libraries.

Importing a User-Managed BSP

To import a user-managed BSP project, perform the following steps:

1. On the File menu, click **Import**. The **Import** dialog box appears.
2. Expand the **Altera Nios II** folder, and select **Existing Nios II software build tools project or folder into workspace**.
3. Click **Next**. The **Import** wizard appears.
4. Click **Browse** and locate the directory containing the BSP project to import.
5. Click **OK**. The wizard fills in the project name and path. The project name is the directory name. You can override the project name by typing a new name in the **Project name** box.

   You might see a warning saying "There is already a **.project** file at: <*path*>". This warning indicates that the directory already contains an IDE project. Either it is an IDE-managed project, or it is a user-managed project that is already imported into the IDE.

   If the project is not already in your workspace, you can import it, but be aware that the **Import** wizard does not convert it from one type to another.

   If the project is already in your workspace, do not re-import it.

6. Click **Finish**. The wizard imports the project, creating a new BSP project in the workspace.

After import, a user-managed BSP looks the same as a user-managed C/C++ application. However, you cannot directly build or run a user-managed BSP in the IDE.

Importing a User-Managed Library

To import a user-managed library, perform the following steps:

1. On the File menu, click **Import**. The **Import** dialog box appears.
2. Expand the **Altera Nios II** folder, and select **Existing Nios II software build tools project or folder into workspace**.
3. Click **Next**. The **Import** wizard appears.

4. Click **Browse** and locate the directory containing the library project to import.

5. Click **OK**. The wizard fills in the project name and path. The project name is the directory name. You can override the project name by typing a new name in the **Project name** box.

   ☞ You might see a warning saying "There is already a **.project** file at: <*path*>". This warning indicates that the directory already contains an IDE project. Either it is an IDE-managed project, or it is a user-managed project that is already imported into the IDE.

   If the project is not already in your workspace, you can import it, but be aware that the **Import** wizard does not convert it from one type to another.

   If the project is already in your workspace, do not re-import it.

6. Click **Finish**. The wizard imports the project, creating a new library project in the workspace.

☞ After import, a user-managed library looks the same as a user-managed C/C++ application. However, you cannot directly build or run a user-managed library in the IDE.

Importing a C/C++ Source Project

To import a C/C++ Source Project, such as newlib, perform the following steps:

1. On the File menu, click **Import**. The **Import** dialog box appears.

2. Expand the **Altera Nios II** folder, and select **Existing Nios II software build tools project or folder into workspace**.

3. Click **Next**. The **Import** wizard appears.

4. Click **Browse** and locate the directory containing the C/C++ source project to import.

5. Click **OK**. The wizard fills in the project name and path. The project name is the directory name. You can override the project name by typing a new name in the **Project name** box.

☞ You might see a warning saying "There is already a **.project** file at: <*path*>". This warning indicates that the directory already contains an IDE project. Either it is an IDE-managed project, or it is a user-managed project that is already imported into the IDE.

If the project is not already in your workspace, you can import it, but be aware that the **Import** wizard does not convert it from one type to another.

If the project is already in your workspace, do not re-import it.

6. Click **Finish**. The wizard imports the project, creating a new C/C++ source project in the workspace.

☞ After import, user-managed C/C++ source code looks the same as a user-managed C/C++ application. However, you cannot directly build or run user-managed C/C++ source code in the IDE.

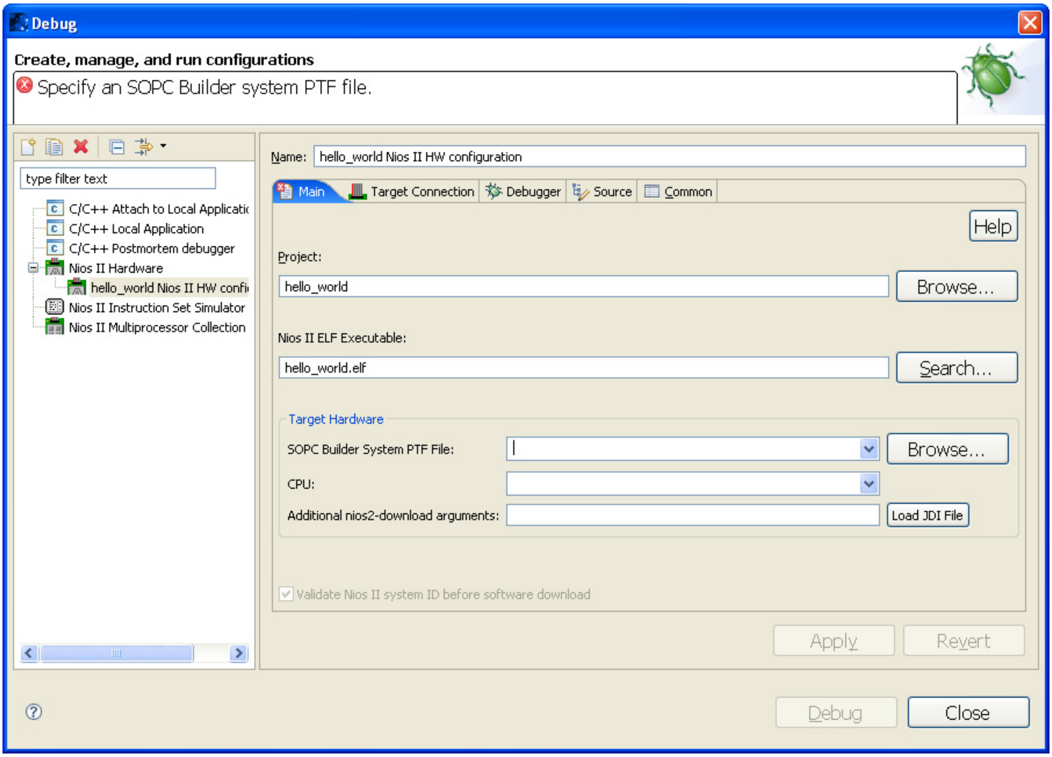
Debug a User-Managed C/C++ Application

To debug an imported user-managed C/C++ application project, perform the following steps:

1. In the Nios II C/C++ Projects view, right click the project name, point to **Debug As**, and click **Nios II Hardware**.

   The debug configuration shows the message: "Specify an SOPC Builder system PTF file", as in Figure 2–12 on page 2–19. The debugger needs information about the target system in order to establish communications.

Figure 2–12. Debug Configuration Manager Initial View


Debug
Create, manage, and run configurations
type filter text
C/C++ Attach to Local Applicatic
C/C++ Local Application
C/C++ Postmortem debugger
Nios II Hardware
hello_world Nios II HW confi
Nios II Instruction Set Simulator
Nios II Multiprocessor Collection
Main
Target Connection
Debugger
Source
Common
Help
Browse...
Search...
Target Hardware
Browse...
Load JDI File
Apply
Revert
Debug
Close

Specify an SOPC Builder system PTF file.

Name: hello_world Nios II HW configuration

Project: hello_world

Nios II ELF Executable: hello_world.elf

SOPC Builder System PTF File:

CPU:

Additional nios2-download arguments:

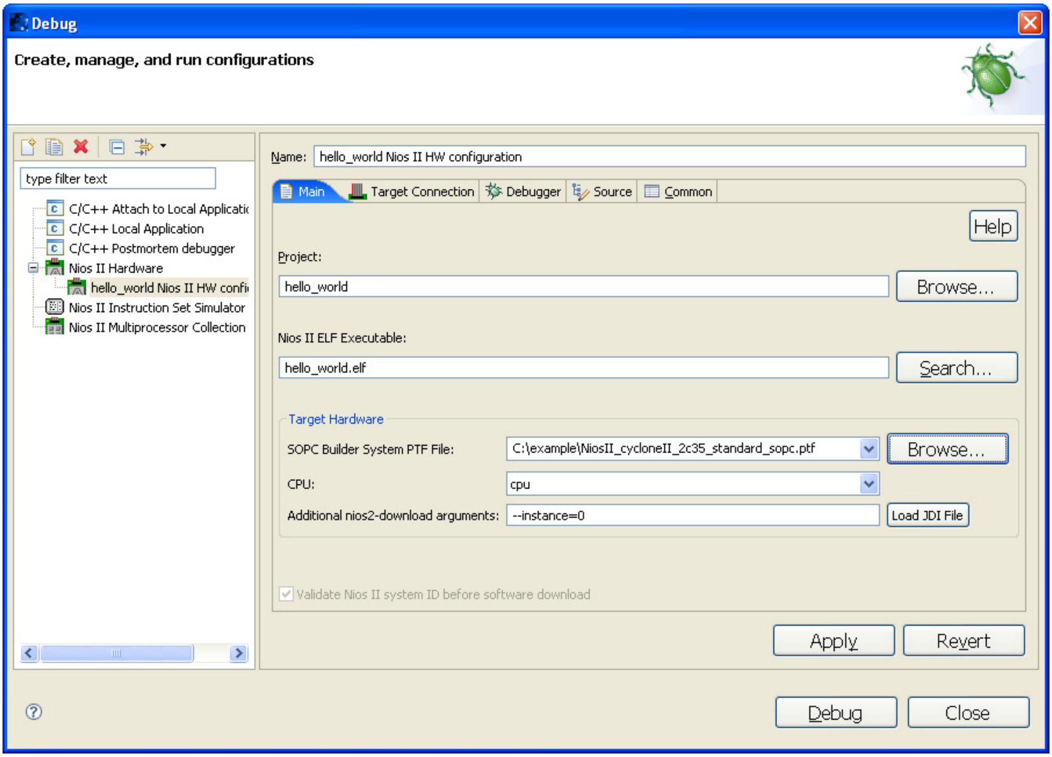
☑ Validate Nios II system ID before software download

2. Click **Browse** at the right of the **SOPC Builder System PTF File** box.

3. Locate the SOPC Builder System File (**.ptf**) on which the application's BSP is based. For example, if you are using a Nios II software example, the SOPC Builder System File is three levels up in the directory tree from the software project.

After you select the file, the message disappears, as shown in Figure 2–13 on page 2–20.

Figure 2–13. Debug Configuration Manager Final View



Debug
Create, manage, and run configurations
type filter text
C/C++ Attach to Local Applicatio
C/C++ Local Application
C/C++ Postmortem debugger
Nios II Hardware
hello_world Nios II HW confi
Nios II Instruction Set Simulator
Nios II Multiprocessor Collection
Main
Target Connection
Debugger
Source
Common
Help
Browse...
Search...
Target Hardware
Browse...
Load JDI File
Apply
Revert
Debug
Close

Name: hello_world Nios II HW configuration

Project: hello_world

Nios II ELF Executable: hello_world.elf

SOPC Builder System PTF File: C:\example\NiosII_cycloneII_2c35_standard_sopc.ptf

CPU: cpu

Additional nios2-download arguments: --instance=0

☑ Validate Nios II system ID before software download

4. Click **Apply**.

Your software application is ready to run or debug exactly as you would run or debug an IDE-managed project. For details about running and debugging applications in the Nios II IDE, see "Running and Debugging Programs" on page 2–8.

Edit User-Managed C/C++ Application Code

You can edit the code in an imported user-managed project with the editor exactly the same way you edit the code in an IDE-managed project.

Programming Flash

Many Nios II processor systems use external flash memory to store one or more of the following items:

- Program code

- Program data
- FPGA configuration data
- File systems

The Nios II IDE provides a Flash Programmer utility to help you manage and program the contents of flash memory. Figure 2–14 on page 2–22 shows the Flash Programmer.

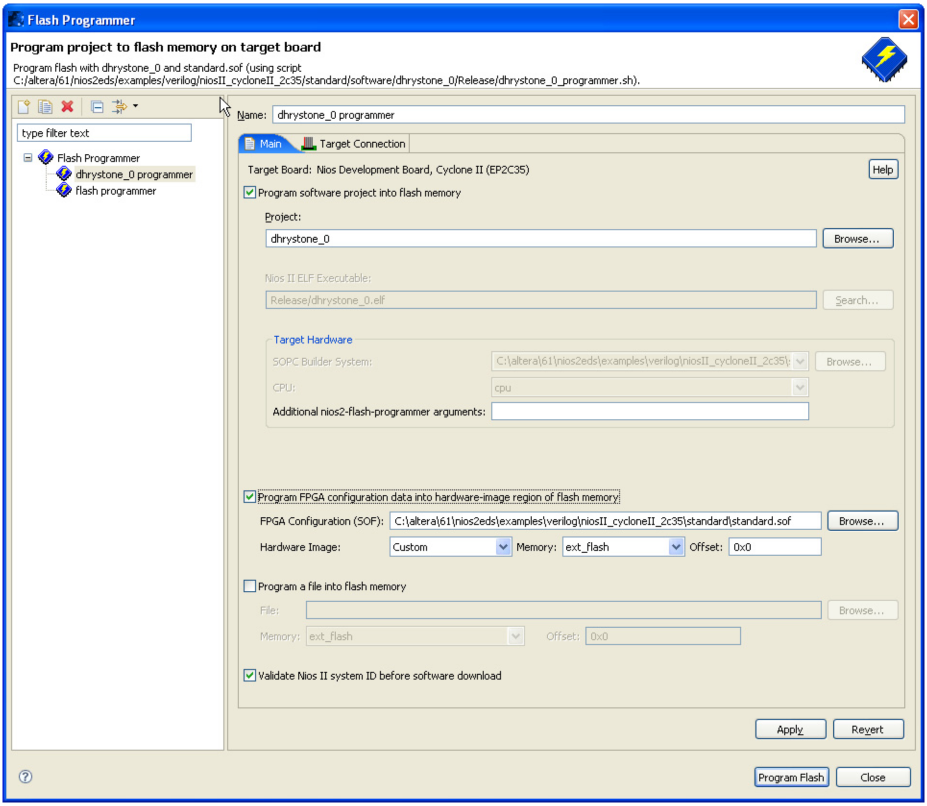
☞ To program a user-managed C/C++ application to flash memory, you must first specify an SOPC Builder System File, as follows:

1. Click **Browse** at the right of the **SOPC Builder System PTF File** box.

2. Locate the SOPC Builder System File on which the application's BSP is based. For example, if you are using a Nios II software example, the SOPC Builder System File is three levels up in the directory tree from the software project.

   This procedure is identical to specifying the SOPC Builder System File before debugging a user-managed C/C++ application, as described in "Debug a User-Managed C/C++ Application" on page 2–18.

Figure 2–14. The Nios II IDE Flash Programmer



Program project to flash memory on target board

Program flash with dhrystone_0 and standard.sof (using script C:/altera/61/nios2eds/examples/verilog/niosII_cycloneII_2c35/standard/software/dhrystone_0/Release/dhrystone_0_programmer.sh).

Name: dhrystone_0 programmer

Target Board: Nios Development Board, Cyclone II (EP2C35)

☑ Program software project into flash memory

Project: dhrystone_0

Nios II ELF Executable: Release/dhrystone_0.elf

Target Hardware

SOPC Builder System: C:\altera\61\nios2eds\examples\verilog\niosII_cycloneII_2c35\

CPU: cpu

Additional nios2-flash-programmer arguments:

☑ Program FPGA configuration data into hardware-image region of flash memory

FPGA Configuration (SOF): C:\altera\61\nios2eds\examples\verilog\niosII_cycloneII_2c35\standard\standard.sof

Hardware Image: Custom Memory: ext_flash Offset: 0x0

☐ Program a file into flash memory

File:

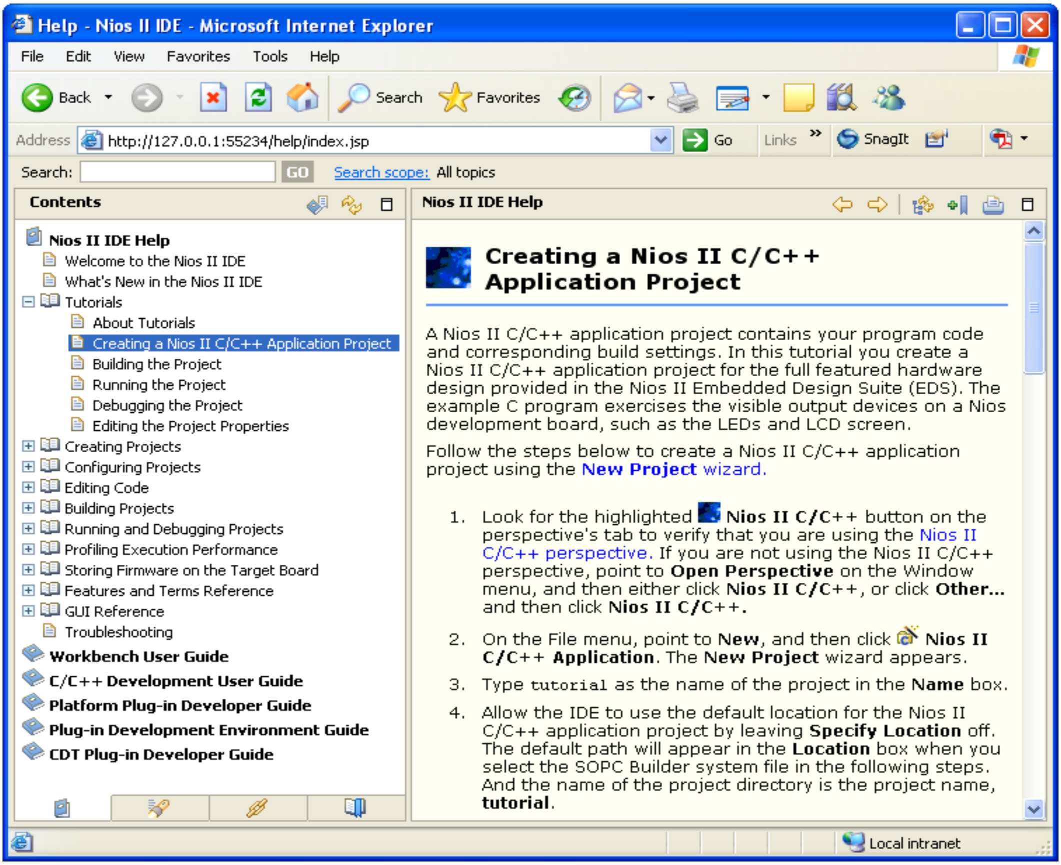
Memory: ext_flash Offset: 0x0

☑ Validate Nios II system ID before software download

Help System

The Nios II IDE help system provides documentation on all IDE topics. To launch the help system, click **Help Contents** on the Help menu. You can also press F1 on Windows (Shift-F1 on Linux) at any time for context-sensitive help. The Nios II IDE help system contains hands-on tutorials that guide you step-by-step through the process of creating, building, and debugging Nios II projects. Figure 2–15 on page 2–23 shows the Nios II IDE help system displaying a tutorial.

Figure 2–15. Tutorials in the Nios II IDE Help System


Help - Nios II IDE - Microsoft Internet Explorer
File
Edit
View
Favorites
Tools
Help
Back
Search
Favorites
Address
http://127.0.0.1:55234/help/index.jsp
Go
Links
SnagIt
Search:
GO
Search scope:
All topics
Contents
Nios II IDE Help
Nios II IDE Help
Welcome to the Nios II IDE
What's New in the Nios II IDE
Tutorials
About Tutorials
Creating a Nios II C/C++ Application Project
Building the Project
Running the Project
Debugging the Project
Editing the Project Properties
Creating Projects
Configuring Projects
Editing Code
Building Projects
Running and Debugging Projects
Profiling Execution Performance
Storing Firmware on the Target Board
Features and Terms Reference
GUI Reference
Troubleshooting
Workbench User Guide
C/C++ Development User Guide
Platform Plug-in Developer Guide
Plug-in Development Environment Guide
CDT Plug-in Developer Guide
Local intranet

Creating a Nios II C/C++ Application Project

A Nios II C/C++ application project contains your program code and corresponding build settings. In this tutorial you create a Nios II C/C++ application project for the full featured hardware design provided in the Nios II Embedded Design Suite (EDS). The example C program exercises the visible output devices on a Nios development board, such as the LEDs and LCD screen.

Follow the steps below to create a Nios II C/C++ application project using the **New Project** wizard.

1. Look for the highlighted **Nios II C/C++** button on the perspective's tab to verify that you are using the Nios II C/C++ perspective. If you are not using the Nios II C/C++ perspective, point to **Open Perspective** on the Window menu, and then either click **Nios II C/C++**, or click **Other...** and then click **Nios II C/C++**.
2. On the File menu, point to **New**, and then click **Nios II C/C++ Application**. The **New Project** wizard appears.
3. Type `tutorial` as the name of the project in the **Name** box.
4. Allow the IDE to use the default location for the Nios II C/C++ application project by leaving **Specify Location** off. The default path will appear in the **Location** box when you select the SOPC Builder system file in the following steps. And the name of the project directory is the project name, **tutorial**.

Referenced Documents

This chapter references the following documents:

- *Introduction to the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*
- *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*

Document Revision History

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

Table 2–2. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	**altera.components** project added	**altera.components** project added
May 2007 v7.1.0	● Added instructions for importing user-managed projects ● Changed chapter title. ● Added table of contents to Introduction section. ● Added Referenced Documents section.	Nios II software build tools
March 2007 v7.0.0	No change from previous release.	
November 2006 v6.1.0	Describes updated look and feel, including Nios II C/C++ perspective and Nios II C/C++ Projects views, renamed project types.	Updated look and feel based on Eclipse 3.2.
May 2006 v6.0.0	No change from previous release.	
October 2005 v5.1.0	Updated for the Nios II IDE version 5.1.	
May 2005 v5.0.0	No change from previous release.	
September 2004 v1.1	Updated screen shots.	
May 2004 v1.0	Initial Release.	

3. Introduction to the Nios II Software Build Tools

NII52014-7.2.0

Introduction

This chapter provides an introduction to the Nios® II software build tools.

The Nios II software build tools provide more detailed control over the software build process than the Nios II integrated development environment (IDE), allowing you to incorporate the software build process into a scripted design flow, or to archive software projects in a version control system. The Nios II software build tools make these tasks easier.

Advantages of the Nios II Software Build Tools

The Nios II software build tools allow you to construct a wide variety of complex software systems using simple command utilities.You can use scripts (or other tools) to combine command utilities in many useful ways.

The Nios II software build tools design flow provides the following advantages over the Nios II IDE design flow:

- Fully repeatable control over all build options using command line options, Tcl scripts, or both
- Simplified project file management and naming
- Simplified makefiles
- Versioned device drivers
- Independence from Eclipse code and Eclipse projects
- Self-contained board support packages (BSPs), making hand-off and version control easier than is possible with Nios II IDE system library projects

Like the Nios II IDE, the Nios II software build tools are available on both Windows and Linux operating systems.

Outline of the Nios II Software Build Tools

Before you can begin to learn how to use the software build tools, you need a basic understanding of what they are.

The Parts of the Software Build Tools

The Nios II software build tools consist of:

- Command line utilities
- Command line scripts
- Tcl commands
- Tcl scripts

These elements work in concert in a **bash** shell environment to create software projects, as described in the next section.

What the Build Tools Create

The purpose of the build tools is to create and build user-managed Nios II software projects. In a user-managed project you (the user) are responsible for the content of the project makefile.

The software build tools can create the following types of user-managed projects:

- Nios II application — a program implementing some desired functionality, such as control or signal processing.
- Nios II BSP — a library providing access to hardware in the Nios II system, such as universal asynchronous receiver/transmitters (UARTs) and other I/O devices. A BSP also includes the operating system, and other basic system software components such as communications protocol stacks. A BSP provides a software runtime environment customized for one processor in an SOPC Builder system.
- User library — a library implementing a collection of reusable functions, such as graphics algorithms.

For a discussion of user-managed software projects, and how they differ from IDE-managed software projects, refer to the *Overview* chapter of the *Nios II Software Developer’s Handbook.* Refer to “Makefiles and the Software Build Tools” for more information about project makefiles.

Makefiles and the Software Build Tools

The central component of a user-managed Nios II software project is its makefile. The makefile describes all the components of a software project and how they are compiled and linked. With a makefile and a complete set of C/C++ source files, your Nios II software project is fully defined. No special project file is needed.

The Nios II build tools include utilities and scripts to create project makefiles. When you are starting out, it is easiest to use these utilities and scripts to create makefiles for you.

The *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook* provides detailed information about creating user-managed makefiles.

As you become experienced with Nios II makefiles, you can modify the application makefile that the build tools generate for you. With further experience, you might choose to create your application makefile from scratch. Studying the autogenerated application makefiles, and experimenting with the makefile generation tools, can help you understand how to create and modify your own makefiles.

☞ Altera® does not recommend creating or modifying BSP makefiles by hand.

Getting Started

The best way to learn about the Nios II software build tools is to use them. The following tutorial guides you through the process of creating, building, running and debugging a “Hello World” program with a minimal number of steps. Later chapters will provide more of the underlying details, allowing you to take more control of the process. But for this chapter the goal is to show you how simple and straightforward it is to get started.

The Nios II software build tools include a number of scripts that demonstrate how to combine command utilities to obtain the results you need. This tutorial shows you one such script: **create-this-app**.

What You Need

To carry out this tutorial, you need the following items:

- Altera® Quartus® II development software, version 7.1 or later. The software must be installed on a Windows or Linux computer that meets the Quartus II minimum requirements.
- A Nios development board
- A download cable such as the Altera USB Blaster™ cable.

Creating hello_world for a Nios Development Board

In this section you create a simple “Hello World” project. To create and build the `hello_world` example for a Nios development board, carry out the following steps:

1. Launch a command shell.

   To open a Nios II command shell under Windows, in the Start menu, point to **Programs**, **Altera**, **Nios II EDS**, and click **Nios II Command Shell**.

   Under Linux, use the shell of your preference.

2. Create a working directory for your hardware and software projects. The following steps refer to this directory as <*projects*>.

3. Change to the <*projects*> directory, as follows:

```
cd <projects>↵
```

4. Locate a Nios II hardware example corresponding to your Nios development board. For example, if you have a Cyclone® II development board, you might select <*Nios II EDS install path*>**/examples/verilog/niosII_cycloneII_2c35/standard**.

   This example uses the Verilog hardware description language (HDL) standard hardware example design. You can select the language you prefer (Verilog HDL or VHDL), and any type of example design except `small`.

5. Copy the hardware example into your <*projects*> working directory, using a command such as the following:

```
cp -R $SOPC_KIT_NIOS2/examples/verilog/niosII_cycloneII_2c35/standard .↵
```

☞ `SOPC_KIT_NIOS2` is a predefined environment variable representing <*Nios II EDS install path*>.

6. Ensure that the working directory and all subdirectories are writable, as follows:

```
chmod -R +w .↵
```

7. The <*projects*> directory contains a subdirectory named **software_examples/app/hello_world**. The following steps refer to this directory as <*application*>.

   Change to the <*application*> directory, as follows:

```
cd <application>↵
```

8. Create and build the application with the **create-this-app** script as follows:

```
./create-this-app↵
```

The **create-this-app** script copies the application source code into the <*application*> directory, runs **nios2-app-generate-makefile** to create a makefile (named **Makefile**), and then runs **make** to create your executable (**.elf**) file. The **create-this-app** script finds a compatible BSP by looking in <*projects*>**/software_examples/bsp**. In the case of `hello_world`, it selects the `hal_default` BSP.

To create the example BSP, **create-this-app** calls the **create-this-bsp** script in the BSP directory.

Running hello_world on a Nios Development Board

To run the `hello_world` example on a Nios development board, carry out the following steps:

1. Launch a Nios II command shell, as described in "Creating hello_world for a Nios Development Board" on page 3–3.

2. When targeting Nios II hardware, you must configure the FPGA on the development board with your project's associated SOPC Builder system. Download the SRAM object file (**.sof**) for the Quartus® II project to the Nios development board. The SRAM object file resides in <*projects*>, along with your Quartus II project file (**.qpf)**. You download it by changing to the <*projects*> directory, then running **nios2-configure-sof**, as follows:

```
cd <projects>↵
nios2-configure-sof↵
```

The board is configured, and ready to run the project's executable code.

nios2-configure-sof runs the Quartus II Programmer to download the SRAM object file. You can also run **quartus_pgm** directly.

For more information about programming the hardware, refer to the *Nios II Hardware Development Tutorial*.

3. Launch another command shell. If practical, make both command shells visible on your desktop.

4. In the second command shell, run the Nios II terminal application to connect to the Nios development board via the Joint Test Action Group (JTAG) UART port, as follows:

```
nios2-terminal↵
```

5. Return to the original command shell, and make sure <*projects*>**/software_examples/app/hello_world** is the current working directory.

6. Download and run the `hello_world` executable as follows:

```
nios2-download -g hello_world.elf↵
```

At this point, you see the following output in the second command shell:

```
Hello from Nios II!
```

Debugging hello_world

An IDE is the most powerful environment for debugging a software project. You debug a user-managed makefile project by importing it into the Nios II IDE. After import, the IDE uses your makefiles to build the project. This two-step process lets you maintain the advantages of user-managed makefiles, while gaining the convenience of a graphical user interface (GUI) debugger.

This section discusses the process of importing and debugging the **hello_world** application.

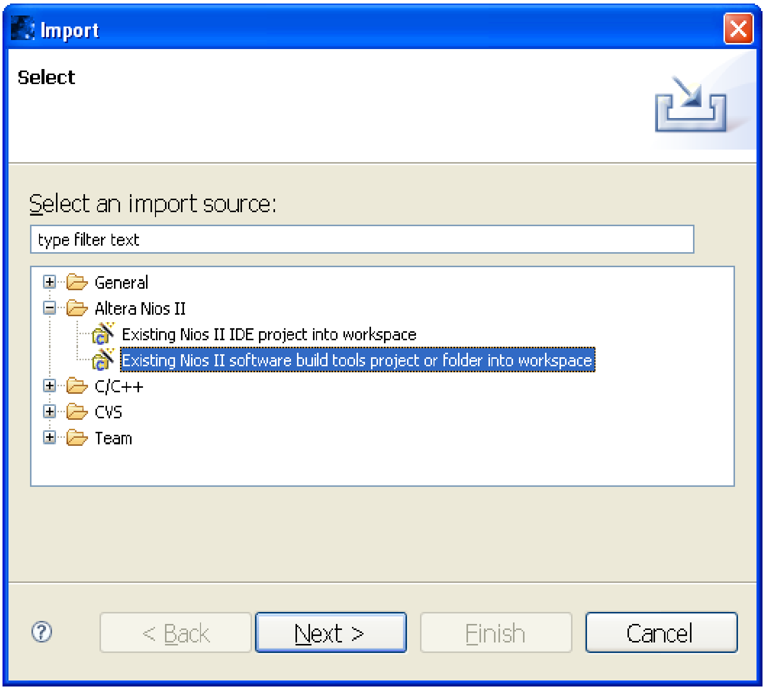
Import the hello_world Application

To import the **hello_world** application, perform the following steps:

1. Launch the IDE.

2. On the File menu, click **Import**. The **Import** dialog box appears.

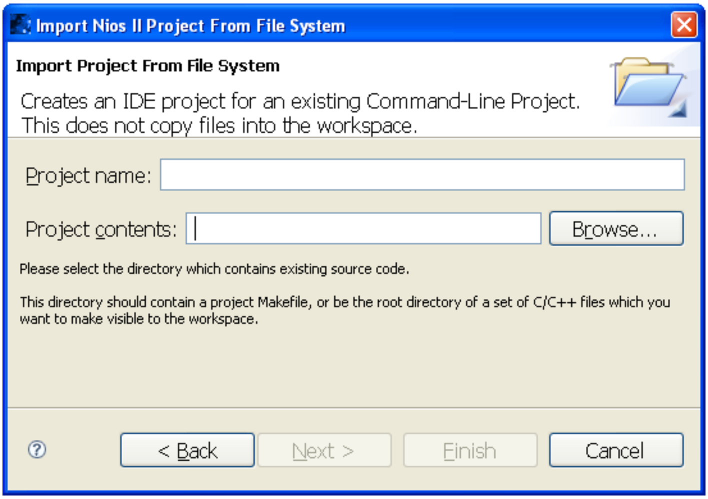
Figure 3–1. Launching the Import Wizard


Import
Select
Select an import source:
type filter text
General
Altera Nios II
Existing Nios II IDE project into workspace
Existing Nios II software build tools project or folder into workspace
C/C++
CVS
Team
< Back
Next >
Finish
Cancel

3. Expand the **Altera Nios II** folder, and select **Existing Nios II software build tools project or folder into workspace**, as shown in Figure 3–1.

4. Click **Next**. The **Import** wizard appears, as shown in Figure 3–2 on page 3–8.

Figure 3–2. User-Managed Project Import Wizard



Import Nios II Project From File System

Import Project From File System

Creates an IDE project for an existing Command-Line Project.
This does not copy files into the workspace.

Project name:

Project contents:

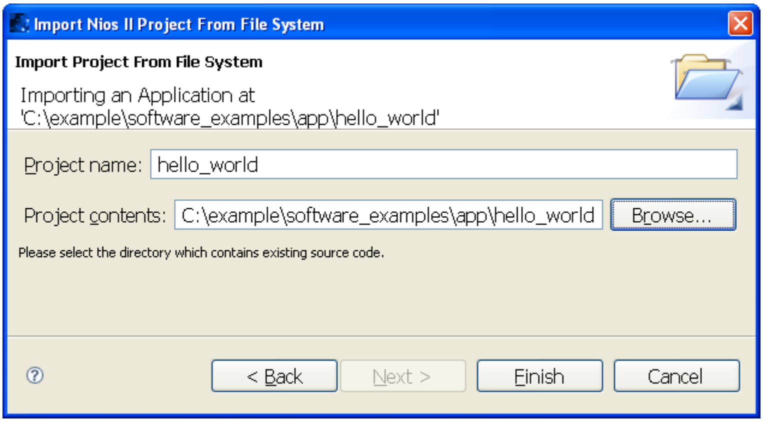
Please select the directory which contains existing source code.

This directory should contain a project Makefile, or be the root directory of a set of C/C++ files which you want to make visible to the workspace.

5. Click **Browse** and navigate to the <*application*> directory, containing the **hello_world** application project.

6. Click **OK**. The wizard fills in the project name and path, as shown in Figure 3–3. The project name defaults to the directory name. You can override the project name by typing a new name in the **Project name** box.

Figure 3–3. Importing a User-Managed C/C++ Application



Import Nios II Project From File System

Import Project From File System

Importing an Application at
'C:\example\software_examples\app\hello_world'

Project name: hello_world

Project contents: C:\example\software_examples\app\hello_world

Please select the directory which contains existing source code.

7. Click **Finish**. The wizard imports the project, creating a new C/C++ application project in the workspace.

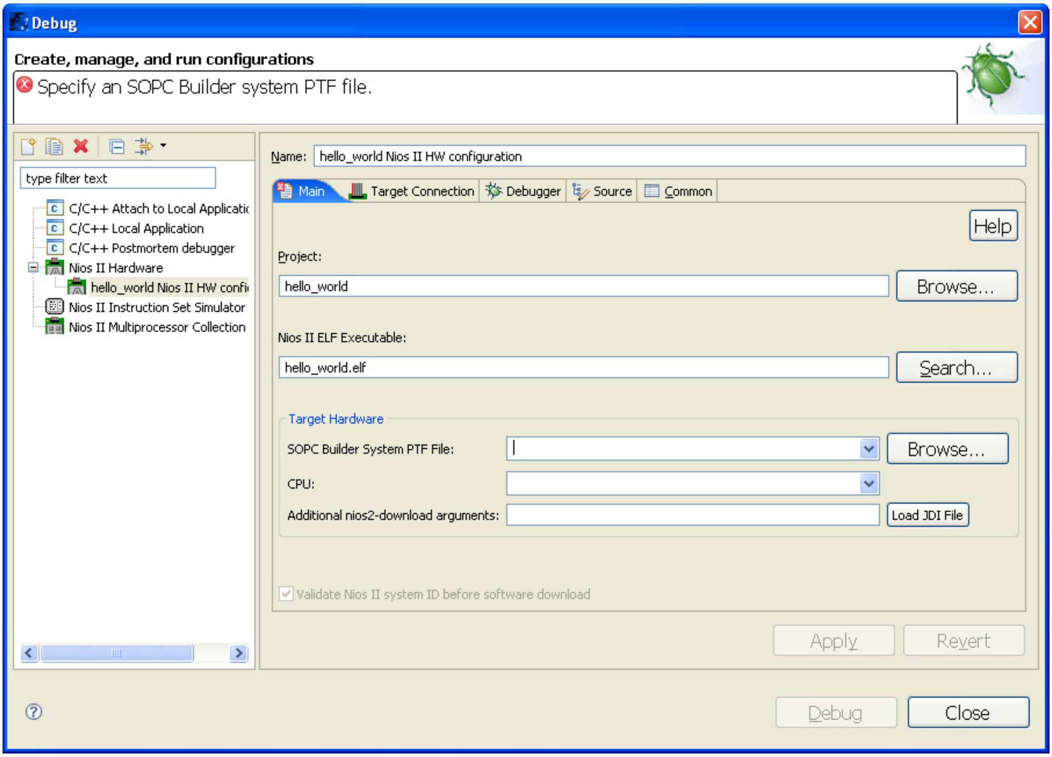
Set Up a Debug Configuration

Before you can debug a project in the Nios II IDE, you must create a debug configuration, which specifies how to run the software. To set up a debug configuration for the **hello_world** project, perform the following steps:

1. In the Nios II C/C++ Projects view, right click the project name **hello_world**, point to **Debug As**, and click **Nios II Hardware**.

   The debug configuration manager displays the message: "Specify an SOPC Builder system PTF file", as shown in Figure 3–4. The debugger needs information about the target system in order to establish communications.

Figure 3–4. Debug Configuration Manager Initial View



Debug

Create, manage, and run configurations

Specify an SOPC Builder system PTF file.

Name: hello_world Nios II HW configuration

Project: hello_world

Nios II ELF Executable: hello_world.elf

Target Hardware

SOPC Builder System PTF File:

CPU:

Additional nios2-download arguments:

☑ Validate Nios II system ID before software download

2. Click **Browse** at the right of the **SOPC Builder System PTF File** box.

3. Locate the SOPC Builder System File (**.ptf**) on which the application's BSP is based. Because you are using a Nios II software example, the SOPC Builder System File is three levels up in the directory tree from the software project in your <*project*> directory.

4. Click **Open**.

   After you select the file, the "Specify an SOPC Builder system PTF file" message disappears.

5. Click **Apply**.

Download Executable Code and Start the Debugger

1. Click **Debug**.

2. If the **Confirm Perspective Switch** dialog box appears, click **Yes**.

   After a moment, you see the `main()` function in the editor. There is a blue arrow next to the first line of code, indicating that execution is stopped on this line.

   When targeting Nios II hardware, the **Debug As** command does the following tasks:

   - Creates a default debug configuration for the target board.
   - Establishes communication with the target board, and verifies that the expected SOPC Builder system is configured in the FPGA.
   - Downloads the executable file (**.elf**) to memory on the target board.
   - Sets a breakpoint at `main()`.
   - Instructs the Nios II processor to begin executing the code.

3. In the Run menu, click **Resume** to resume execution. You can also resume execution by pressing **F8**.

When debugging a project in the Nios II IDE, you can also pause, stop, and single-step the program, set breakpoints, examine variables, and perform many other common debugging tasks.

For more information about debugging software projects in the Nios II IDE, refer to the *Nios II Integrated Development Environment* chapter of the *Nios II Software Developer's Handbook*. For detailed information about IDE debugging features, refer to the Nios II IDE help system.

Next Steps

Now that you have created, built, run and debugged a sample program, you probably want to start working with a real project. The next section, "Creating a Script", shows you how to get started on your own script.

For detailed information about the using Nios II software build tools, refer to the *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*. For a description of the differences between the Nios II IDE and the software build tools, refer to the *Overview* chapter of the *Nios II Software Developer's Handbook*.

Creating a Script

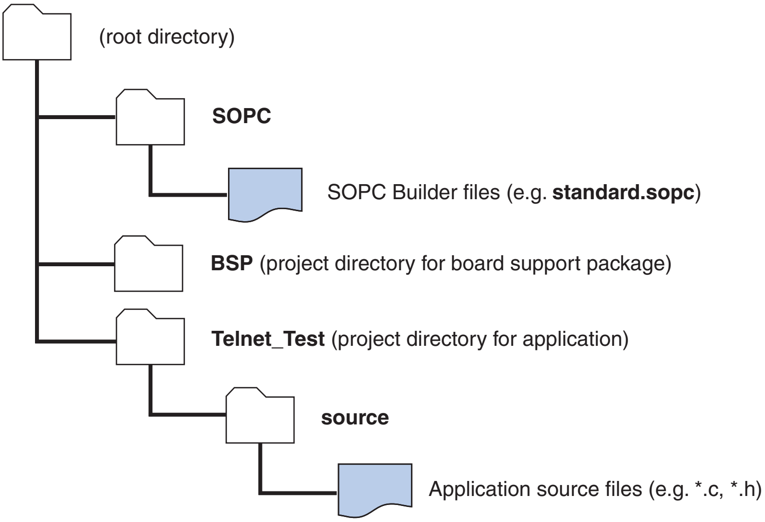
In simple cases, you can do everything you need by running Nios II software build tools utilities from the command line. More commonly, developers create some simple scripts, either from scratch, or based on example scripts.

Scripting Basics

This section gives an example of how you can create a software application using a script.

Suppose you want to build a software application for a Nios II system that features the **lan91c111** component and supports the NicheStack™ TCP/IP stack. Furthermore, suppose that you have organized the hardware design files and the software source files as shown in "Simple Software Project Directory Structure".

Figure 3–5. Simple Software Project Directory Structure


(root directory)
SOPC
SOPC Builder files (e.g. **standard.sopc**)
BSP (project directory for board support package)
Telnet_Test (project directory for application)
source
Application source files (e.g. *.c, *.h)

One easy method for creating a BSP is to use the **nios2-bsp** script. The following script creates a BSP and then builds it.

```
nios2-bsp ucosii . ../SOPC/ --cmd enable_sw_package altera_iniche \
    --set altera_iniche.iniche_default_if lan91c111
make
```

The arguments to **nios2-bsp** have the following meanings:

- `ucosii` sets the BSP type to MicroC/OS-II.
- `.` specifies the directory in which the BSP is to be created.
- `../SOPC/` points to the location of the SOPC Builder system.
- `--cmd enable_sw_package altera_iniche` adds the NicheStack TCP/IP stack software package to the BSP.
- `--set altera_iniche.iniche_default_if lan91c111` specifies the default hardware interface for the NicheStack TCP/IP stack.

You create application projects with **nios2-app-generate-makefile**. The following script creates an application project and builds it.

```
nios2-app-generate-makefile --bsp-dir ../BSP --elf-name telnet-test.elf --src-dir source/
make
```

The arguments to **nios2-app-generate-makefile** have the following meanings:

- `--bsp-dir ../BSP` specifies the location of the BSP on which this application is based
- `--elf-name telnet-test.elf` specifies the name of the executable file.
- `--src-dir source/` tells **nios2-app-generate-makefile** where to find the C source files.

These simple scripts are all you need to create and build your application.

Nios II Scripting Examples

The Nios II Embedded Design Suite (EDS) includes many hardware and software examples based on the Nios II processor. These include hardware designs that you can download to Nios development boards, and software examples that run on these designs. The examples can be very helpful as you start the development of your custom design. They provide a stable starting point for exploring design options. Also, they demonstrate many commonly used features of the Nios II EDS.

The Nios II software examples come with scripts to create and build the software projects using the Nios II software build tools. These scripts do everything necessary to create a BSP and an application project for each software example. You can copy and modify these scripts to create your custom software design.

The hardware examples for each Nios II development board reside in:

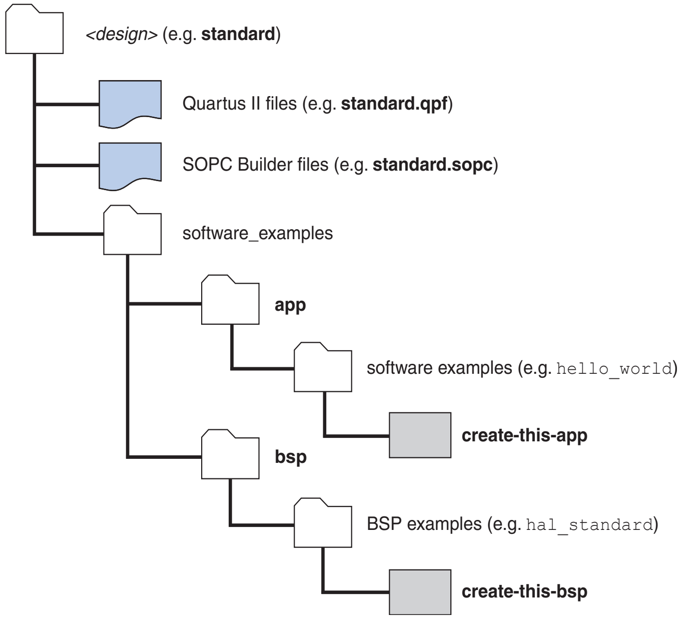
*<Nios II EDS install path>***/examples/*****<language>*****/*****<board>***

<language> is either **vhdl** or **verilog** and *<board>* is the name of the development board. For example, the standard Verilog HDL example design for the Nios II 1S40 development board resides at:

*<Nios II EDS install path>***/examples/verilog/niosII_stratix_1s40/standard**

Figure 3–6 shows the directory structure under each hardware example design. There are multiple software examples and BSP examples, each with its own directory. Each software example directory contains a **create-this-app** script and each BSP example directory contains a **create-this-bsp** script. These scripts create software projects, as demonstrated in "Creating hello_world for a Nios Development Board" on page 3–3.

Figure 3–6. Software Example Design Directory Structure


<design> (e.g. standard)
Quartus II files (e.g. standard.qpf)
SOPC Builder files (e.g. standard.sopc)
software_examples
app
software examples (e.g. hello_world)
create-this-app
bsp
BSP examples (e.g. hal_standard)
create-this-bsp

For more detail about the software example scripts, refer to the *Example Design Scripts* section in the *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

Referenced Documents

This chapter references the following documents:

- *Overview* chapter of the *Nios II Software Developer's Handbook*
- *Nios II Integrated Development Environment* chapter of the *Nios II Software Developer's Handbook*
- *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*
- Nios II Hardware Development Tutorial

Document Revision History

Table 3–1 shows the revision history for this document.

Table 3–1. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Repurpose this chapter as a “getting started” guide. Move descriptive and reference material to separate chapters.	Additional “getting started” material. Descriptive and reference material in separate chapters.
May 2007 v7.1.0	Initial Release.	—

4. Using the Nios II Software Build Tools

NII52015-7.2.0

Introduction

This chapter describes how to use the Nios® II software build tools to create and build software projects. The software build tools are a set of command utilities and scripts that create and build user-managed C/C++ application projects, library projects, and board support packages (BSPs). They are helpful if you need a repeatable, scriptable and archivable process for creating your software product. The Nios II software build tools are the basis for Altera®'s future Nios II development.

The purpose of this chapter is to tell you how to use the Nios II software build tools to create and build your software project. This chapter provides what you need to know to develop the most common kinds of software projects.

The chapter contains the following sections:

Read the *Introduction to the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook* before starting this chapter. This chapter also assumes familiarity with the following topics:

- The GNU **make** utility. Altera recommends you use version 3.79 or later, provided with the Nios II Embedded Design Suite (EDS).
- Board support packages.

Depending on how you use the tools, you might also need to be familiar with the following topics:

- Micrium MicroC/OS-II. For information, refer to *MicroC/OS-II - The Real Time Kernel* by Jean J. Labrosse (CMP Books).
- Tcl scripting language.

For an overview of Nios II EDS design flows, including the Nios II integrated development environment (IDE) and Nios II software build tools, refer to the *Overview* chapter of the *Nios II Software Developer's Handbook.* For an overview of all command-line tools provided with the Nios II EDS, refer to the *Altera-Provided Development Tools* chapter of the *Nios II Software Developer's Handbook*. General information on GNU make is available at **www.gnu.org**.

Advantages of the Software Build Tools Design Flow

The Altera Nios II software build tools design flow emphasizes the following qualities:

- Modularity
- Simplicity
- Flexibility
- Extensibility

A major difference between the Nios II IDE software development flow and the Nios II software build tools flow is the difference in makefile implementation. The Nios II software build tools include the makefile generator, which generates user-managed makefiles that you can further edit. You can also create your makefiles by hand with the Nios II software build tools.

The key differences between user-managed makefiles and IDE-managed makefiles are as follows:

- You have control over the contents of a user-managed makefile.
- The syntax of user-managed makefiles is clearer than the IDE-managed makefiles.
- User-managed makefiles are less fragmented than IDE-managed makefiles.

For further information about user-managed makefiles, see "User-Managed Makefiles" on page 4–9.

Road Map to the Nios II Software Build Tools

This section provides a road map to the software build tools.

Before you start using the Nios II software build tools seriously, it is important to learn what their scope is. You need to understand their purpose, what they include, and what they do. This helps you determine how they fit into your development process, what parts of the tools you need, and what features you can disregard for now.

Software Build Process

When you create a software project with the Nios II software build tools, you go through several broad steps:

1. Obtain the hardware design that the software is to run on. When you are learning about the build tools, this might be a Nios II example design. When you are developing a product, it is probably a design developed by someone in your organization. Either way, you need to have the SOPC Builder system file (**.sopc**).

2. Decide what features the BSP requires. For example, does it need to support a real time operating system (RTOS)? Does it need other specialized software support, such as a TCP/IP stack? Does it need to fit in a small memory footprint? The answers to these questions tell you what BSP features and settings to use.

   For more information about available BSP settings, refer to "Settings for BSPs, Software Packages and Device Drivers" in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

3. Define a BSP. Use some of the Nios II software build tools to specify the components in the BSP, and the values of any relevant settings. The result of this step is a BSP settings file, called **settings.bsp**. For more information about creating BSPs, see "Board Support Packages" on page 4–12.

4. Create a BSP makefile. The Nios II build tools can do this for you, which is the easiest approach. You can also create a makefile by hand, or you can autogenerate a makefile and then customize it by hand. For more information about creating makefiles, see "User-Managed Makefiles" on page 4–9.

5. Optionally create a user library. If you need to include a custom software library, you collect the library source files into a single directory, and create a library makefile. The Nios II build tools can create a makefile for you, which is the easiest approach. You can also create a makefile by hand, or you can autogenerate a makefile and then customize it by hand. For more information about creating user library projects, see "Applications and Libraries" on page 4–11.

6. Collect your application source code. When you are learning, this might be a Nios II software example. When you are developing a product, it is probably a collection of C/C++ source files developed by someone in your organization. For more information about creating application projects, see "Applications and Libraries" on page 4–11.

7. Create an application makefile. The Nios II build tools can do this for you, which is the easiest approach. You can also create a makefile by hand, or you can autogenerate a makefile and then customize it by hand. For more information about creating makefiles, see "User-Managed Makefiles" on page 4–9.

Generators, Utilities, and Scripts

The Nios II software build tools consist of generators, utilities, and scripts. This section discusses each of these portions of the build tools.

Generators

Generators are sets of tools that create specific parts of your software project. The command utilities and scripts included in the Nios II software build tools combine to form the following generators:

- Nios II BSP generator — a set of tools to create and manage settings for a BSP
- Nios II makefile generator — a set of tools to create makefiles for BSPs, C/C++ applications and libraries

For more information about the generators, see "Applications and Libraries" on page 4–11 and "Board Support Packages" on page 4–12.

Utilities

Table 4–1 summarizes the command line utilities provided by the Nios II software build tools, and their relationships to the generators. You can invoke these utilities on the command line or from a scripting language of your choice (such as **perl** or **bash**). On Windows, these utilities have a **.exe** extension. The Nios II software build tools reside in the **sdk2/bin** directory under <*Nios II EDS install path*>.

☞ In the Nios II command shell, <*Nios II EDS install path*> is specified by the `SOPC_KIT_NIOS2` environment variable.

Table 4–1. Nios II Software Build Tools Command Utilities

Command	Summary	BSP Generator	Makefile Generator
nios2-app-generate-makefile	Creates an application makefile		✓
nios2-lib-generate-makefile	Creates a library makefile		✓
nios2-bsp-create-settings	Creates a BSP settings file	✓	
nios2-bsp-update-settings	Updates the contents of a BSP settings file	✓	
nios2-bsp-query-settings	Queries the contents of a BSP settings file	✓	
nios2-bsp-generate-files	Generates all files for a given BSP settings file	✓	✓

Scripts

Nios II software build tools scripts implement complex behavior that extends the capabilities provided by the utilities.

Table 4–2 summarizes the scripts provided with the Nios II software build tools, and their relationships to the generators.

Table 4–2. Nios II Software Build Tools Scripts

Command	Summary	BSP Generator	Makefile Generator
nios2-bsp	Creates or updates a BSP	✓	
nios2-c2h-generate-makefile	Creates application makefile fragment for the Nios II C2H compiler		✓
create-this-app *(1)*	Creates a software example and builds it.	✓	✓
create-this-bsp *(1)*	Creates a BSP for a specific hardware example design and builds it.	✓	✓

Note to Table 4–2:

(1) There are **create-this-app** scripts for each software example and several **create-this-bsp** scripts for each hardware example design. For more details, see "Using Nios II Example Design Scripts" on page 4–6.

Using Nios II Example Design Scripts

The Nios II software build tools include scripts that allow you to create sample BSPs and applications. This section describes each script and its location in the example design directory structure. Each hardware example design in the Nios II EDS includes a **software_examples** directory with **app** and **bsp** subdirectories.

The **bsp** subdirectory contains a variety of example BSPs. Table 4–3 lists all potential BSP examples provided under the **bsp** directory. The **bsp** directory for each hardware example only includes BSP examples supported by the associated hardware example.

Table 4–3. BSP Examples

Example BSP *(1)*	Summary
`hal_dhrystone`	HAL BSP configured for the Dhrystone benchmark
`hal_hostfs`	HAL BSP configured with the Altera host file system
`hal_reduced_footprint`	HAL BSP configured to minimize memory footprint
`hal_default`	HAL BSP configured with all defaults
`hal_zipfs`	HAL BSP configured with the Altera read-only Zip file system
`ucosii_net`	MicroC/OS-II BSP configured with networking
`ucosii_net_zipfs`	MicroC/OS-II BSP configured with networking and the Altera read-only Zip file system
`ucosii_net_tse`	MicroC/OS-II BSP configured with networking support for the Altera triple-speed Ethernet media access control (MAC)
`ucosii_net_tse_zipfs`	MicroC/OS-II BSP configured with networking support for the Altera triple-speed Ethernet MAC and the Altera read-only Zip file system
`ucosii_default`	MicroC/OS-II BSP configured with all defaults

Note to Table 4–3:

(1) Some BSP examples might not be available on some hardware examples.

In the **app** subdirectory, there is a further subdirectory for each software example supported by the hardware example, as listed in Table 4–4.

Table 4–4. Application Examples (1)

Application Name	Summary
Hello World	Prints 'Hello from Nios II'
Board Diagnostics	Tests peripherals on the development boards
Count Binary	Displays a running count of 0x00 to 0xff
Dhrystone	Runs the Dhrystone 2.1 benchmark code
Hello Freestanding	Prints 'Hello from Nios II' from a freestanding application
Hello LED	Displays a bouncing pattern on light-emitting diodes (LEDs)
Hello MicroC/OS-II	Prints 'Hello from Nios II' using the MicroC/OS-II RTOS
Hello World Small	Prints 'Hello from Nios II' from a small footprint program
Host File System	Reads and writes to files on the host using the GNU Debugger (GDB) Host File System
Memory Test	Runs diagnostic tests on both volatile and flash memory
Simple Socket Server	Runs a TCP/IP socket server
Tightly Coupled Memory	Shows performance gain using tightly coupled memory
MicroC/OS-II Message Box	Demonstrates the use of MicroC/OS-II message boxes
Web Server	Runs a web server from a file system in flash memory
Zip File System	Reads from a file system in flash memory

Note to Table 4–4:

(1) Some application examples might not be available on some hardware examples, depending on BSP support.

create-this-bsp

Each BSP subdirectory contains a **create-this-bsp** script. **create-this-bsp** calls the **nios2-bsp** script to create a BSP in the current directory. The **create-this-bsp** script has a relative path to the directory containing the SOPC Builder system file. The SOPC Builder system file resides two directory levels above the directory containing the **create-this-bsp** script.

The **create-this-bsp** script takes no command line arguments. Your current directory must be the same directory as the **create-this-bsp** script. The exit value is zero on success and one on error.

create-this-app

Each application subdirectory contains a **create-this-app** script. **create-this-app** copies the C/C++ application source code into the current directory, runs **nios2-app-generate-makefile** to create a makefile (named **Makefile**), and then runs **make** to build your application executable (**.elf**) file. Each **create-this-app** script uses a particular example BSP. For further information, look at the script to see which example BSP it uses. If the BSP does not exist when **create-this-app** runs, it invokes the associated **create-this-bsp** script to create the BSP.

The **create-this-app** script takes no command line arguments. Your current directory must be the same directory as the **create-this-app** script. The exit value is zero on success and one on error.

Finding create-this-app and create-this-bsp

The **create-this-app** and **create-this-bsp** scripts are installed with your Nios II example designs. You can easily find them once you know the following information:

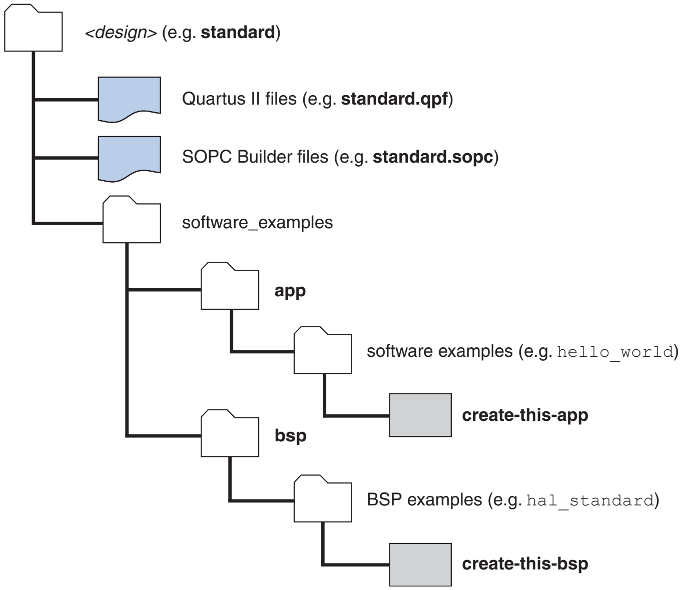
- Where the Nios II EDS is installed
- Which Nios development board you are using
- Which hardware definition language (HDL) you are using
- Which Nios II hardware example design you are using
- The name of the Nios II software example

The **create-this-app** script for each software example design is in ***<Nios II EDS install path>**examples**<HDL>*\niosII_*<board type>**<design name>*\software_examples\app\*<example name>***. For example, the **create-this-app** script for the **Hello World** software example running on the Verilog HDL full-featured example design for the Nios II Development Kit, Cyclone® II Edition, might be located in **C:\altera\71\nios2eds\examples\verilog\niosII_cycloneII_2c35\full_featured\software_examples\app\hello_world**.

Similarly, the **create-this-bsp** script for each software example design is in ***<Nios II EDS install path>**examples**<HDL>*\niosII_*<board type>**<design name>*\software_examples\bsp\*<BSP_type>***. For example, the **create-this-bsp** script for the basic HAL BSP to run on the Verilog HDL full-featured example design for the Nios II Development Kit, Cyclone II Edition, might be located in **C:\altera\71\nios2eds\examples\verilog\niosII_cycloneII_2c35\full_featured\software_examples\bsp\hal_default**.

Figure 4–1 shows the directory structure under each hardware example design.

Figure 4–1. Software Example Design Directory Structure


<design> (e.g. **standard**)
Quartus II files (e.g. **standard.qpf**)
SOPC Builder files (e.g. **standard.sopc**)
software_examples
app
software examples (e.g. hello_world)
create-this-app
bsp
BSP examples (e.g. hal_standard)
create-this-bsp

User-Managed Makefiles

Makefiles are a key element of user-managed projects. The Nios II software build tools include powerful tools to create makefiles. An understanding of how these tools work can help you make the most optimal use of them.

If you choose to create your makefiles by hand, you might still find it helpful to understand how makefile generation works. Letting the software build tools generate makefiles is an excellent way to see examples of powerful makefile usage.

The makefile generators (incorporated in Nios II software build tools) create two kinds of user-managed makefiles:

- Application or library makefile — a simple makefile that you can edit by hand with a text editor.
- BSP makefile — a more complex makefile, generated to conform to user-specified settings and the requirements of the target SOPC Builder system.

It is not necessary to use to the generated application and library makefiles if you prefer to write your own. However, Altera recommends strongly that you use the software build tools to manage and modify BSP makefiles.

For an overview of the user-managed and IDE-managed concepts, refer to the *Nios II Integrated Development Environment* chapter of the *Nios II Software Developer's Handbook.*

Generated makefiles are platform-independent, invoking only commands provided with the Nios II EDS (such as **nios2-elf-gcc**).

The generated makefiles have a straightforward structure and in-depth comments explaining how they work. Altera recommends that you study them for hints about how to use the makefile generator. Generated BSP makefiles consist of a single main file and a small number of makefile fragments, all of which reside in the BSP directory. Each application and library has one makefile, located in the application or library directory.

Makefile Targets

Table 4–5 shows the application makefile targets. Altera recommends that you study the generated makefiles for further details on these targets.

Table 4–5. Application Makefile Targets

Target	Operation
`help`	Displays all available application makefile targets.
`all` (default)	Builds the associated BSP and libraries, and then builds the application executable file.
`app`	Builds only the application executable file.
`bsp`	Builds only the BSP.
`libs`	Builds only the libraries and the BSP.
`clean`	Cleans the application, i.e, deletes all application-related generated files (leaves associated BSP and libraries alone).
`clean_all`	Cleans the application, and associated BSP and libraries (if any).
`clean_bsp`	Cleans the BSP.
`clean_libs`	Cleans the libraries and the BSP.
`download-elf`	Builds the application executable file and then downloads and runs it.
`program-flash`	Runs the Nios II flash programmer to program your flash memory.

You can specify multiple targets on a **make** command line. For example, the following command removes existing object files in the current project directory, builds the project, downloads the project to a board and runs it:

```
make clean download-elf↵
```

Nios II C2H Makefiles

The Nios II software build tools support the Nios II C2H compiler via the **nios2-c2h-generate-makefile** command. This command creates the C2H makefile fragment, **c2h.mk**, which specifies all accelerators and accelerator options for an application.

nios2-c2h-generate-makefile creates a new **c2h.mk** each time it is executed, overwriting the existing **c2h.mk**.

☞ You must use the `--c2h` flag when calling **nios2-app-generate-makefile** in order to build your application with the C2H compiler. This flag causes your application makefile to include the static C2H make rules. These rules in turn include the **c2h.mk** fragment generated by **nios2-c2h-generate-makefile**.

For more detail about using the C2H compiler with the software build tools, see "Using the Nios II C2H Compiler" on page 4–36.

For more detail about **nios2-c2h-generate-makefile**, refer to the *"Build Tools Utilities"* section in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Applications and Libraries

The Nios II software build tools have nearly identical support for C/C++ applications and libraries. The support for applications and libraries is very simple. The **nios2-app-generate-makefile** and **nios2-lib-generate-makefile** commands each generate a private makefile (named **Makefile**). The private makefile is used to build the application or library.

nios2-lib-generate-makefile also generates a public makefile, called **public.mk**. The public makefile is included in the private makefile for any application (or other library) that uses the library.

If you need to change an application or library makefile after generation, you can edit it using a text editor, or utilities such as **perl** and **sed**.

The private makefile builds one of two types of files:

- An executable and linked format (**.elf**) file — for an application
- An archive file (**.a**) — for a library

The command is passed a list of source files and a reference to a BSP directory. The BSP directory is mandatory for applications and optional for libraries.

The **nios2-app-generate-makefile** and **nios2-lib-generate-makefile** commands examine the extension of each source file to determine the programming language. Table 4–6 shows the supported programming languages with the corresponding file extensions.

Table 4–6. Supported Source File Types

Programming Language	File Extensions
C	.c
C++	.cpp, .cxx, .cc
Nios II assembly language	.s, .S

Board Support Packages

A Nios II board support package (BSP) project is a specialized library containing system-specific support code. A BSP provides a software runtime environment customized for one processor in an SOPC Builder system. The BSP isolates your application from system-specific details such as the memory map, available devices, and processor configuration.

A BSP consists of a library archive file, header files (for example, **system.h**), and a linker script (**linker.x**). You use these BSP files when creating an application.

The Nios II software build tools support two types of BSPs: Altera HAL and Micrium MicroC/OS-II. MicroC/OS-II is a layer on top of the Altera HAL and shares a common structure.

Overview of BSP Creation

You create a BSP with the Nios II BSP generator.This tool provides a great deal of power and flexibility, enabling you to control details of your BSP implementation while maintaining compatibility with an SOPC Builder system which might change.

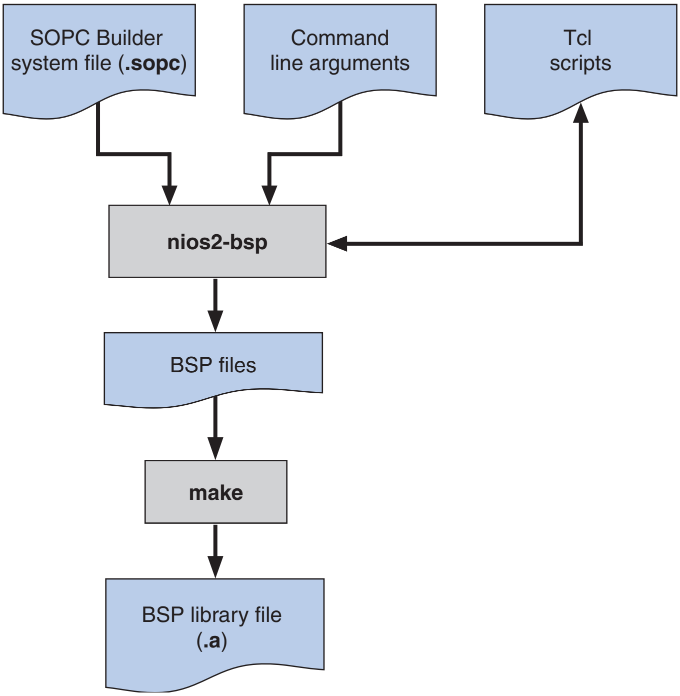
By default, the tools generate a basic BSP for a Nios II system. If this is what you need, you can skip the remainder of this chapter.

If you need more detailed control over the characteristics of your BSP, Nios II software build tools provide that control. The more features you use, the more complex your commands and scripts become.

The remainder of this section describes how to get the most out of the Nios II software build tools.

Figure 4–2 shows the flow to create a BSP using the **nios2-bsp** command. **nios2-bsp** uses the SOPC Builder system file to create the BSP files. You can override default settings chosen by **nios2-bsp** by supplying command line arguments, Tcl scripts, or both.

Figure 4–2. nios2-bsp Command Flow



nios2-bsp puts all BSP files in the BSP directory. After running **nios2-bsp**, you run **make**, which compiles the source code. The result of compilation is the BSP library file, also in the BSP directory. The BSP is ready to be used by your application.

Generated and Copied Files

To understand how to build and modify user-managed projects, it is important to understand the difference between copied and generated files.

A copied file is installed with the Nios II EDS, and copied into your BSP directory when you create your BSP. A copied file is only written if the file does not already exist in your BSP directory. Thus you can freely modify copied files, without losing your changes when you update your BSP.

A generated file is dynamically created by the **nios2-bsp-generate-files** command. A generated file is written every time **nios2-bsp-generate-files** is run. Generated files reside in the top-level BSP directory.

Coordinating with Hardware Changes

If you change your SOPC Builder system, you almost always need to update your BSP. How you update the BSP depends on the nature of the system change. The BSP settings file does not duplicate information available in the SOPC Builder system file, but it does contain system-dependent settings that make references to system information. Because of these system-dependent settings, a BSP settings file can become inconsistent with its system if the system changes. For example, if the `stdio` device is set up to use a module named `uart0` and you rename it to `uart1`, the BSP settings file must be changed.

If you are not sure whether the change to your SOPC Builder system file has introduced inconsistencies with your BSP settings, you can simply rerun **nios2-bsp** to recreate your settings file.

Altera HAL BSP

The Altera HAL is a basic single-threaded run-time environment.

For more information on the Altera HAL, see the *Overview of the Hardware Abstraction Layer* and *Developing Programs using the HAL* chapters of the *Nios II Software Developer's Handbook*.

HAL BSP Files and Folders

Figure 4–3 on page 4–15 shows the HAL BSP directory after the **nios2-bsp-create-settings** command has created a settings file named **settings.bsp**. Figure 4–3 assumes that the **my_hal_bsp** directory is initially empty.

Figure 4–3. HAL BSP After Creating Settings

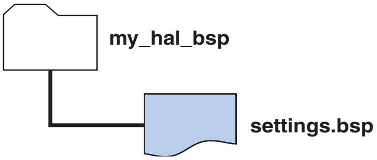


Figure 4–4 shows the **my_hal_bsp** directory after the **nios2-bsp-generate-files** command has generated BSP files. Figure 4–4 also represents the BSP directory after running the **nios2-bsp** command to create or update a BSP. **nios2-bsp-generate-files** programmatically generates all top-level files, except **settings.bsp**. It also copies files into the HAL and drivers directories from their installed locations.

settings.bsp

The **settings.bsp** file is a file that contains all BSP settings. It is coded in XML. This file is created by the **nios2-bsp-create-settings** command, and optionally updated by the **nios2-bsp-update-settings** command. It also can be copied from another BSP directory. The **nios2-bsp-query-settings** command is available to parse information from the settings file for your scripts. The **settings.bsp** file is an input to **nios2-bsp-generate-files**.

summary.html

The **summary.html** file is a generated file that provides summary documentation of the BSP. You can view **summary.html** with a hypertext viewer or browser, such as **Internet Explorer** or **FireFox**. If you change the **settings.bsp** file (manually or by running **nios2-bsp-update-settings**), the next time you run **nios2-bsp-generate-files**, it updates the **summary.html** file.

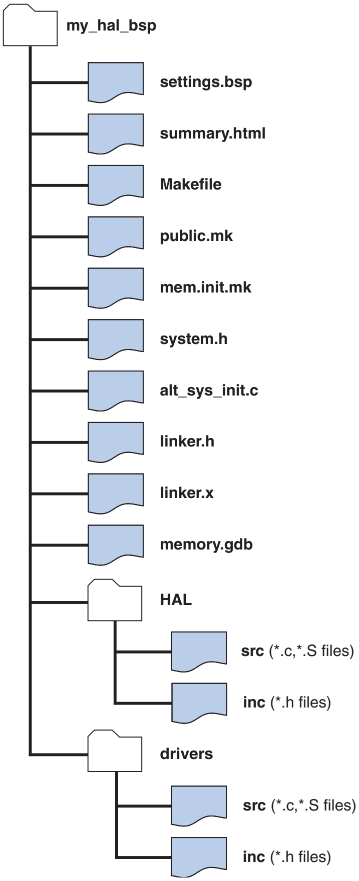
Makefile

The **Makefile** file is a generated file used to build the BSP. The targets you use most often are `all` and `clean`. The `all` target (the default) builds the **libhal_bsp.a** library file. The `clean` target removes all files created by a **make** of the `all` target.

public.mk

The **public.mk** file is a generated makefile fragment that provides public information about the BSP. The file is designed to be included in other makefiles that use the BSP, such as application makefiles. The BSP **Makefile** also includes **public.mk**.

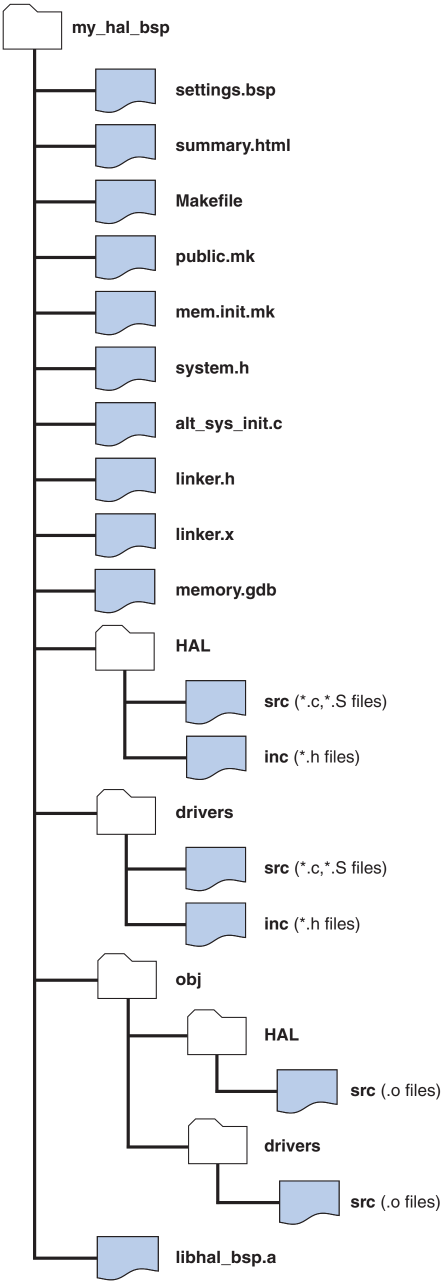
Figure 4–4. HAL BSP After Generating Files



mem_init.mk

The **mem_init.mk** file is a generated makefile fragment that defines targets and rules to convert an application executable file into memory initialization files (**.dat**, **.hex**, and **.flash**) for HDL simulation, flash programming, and initializable FPGA memories. The **mem_init.mk** file is designed to be included by an application makefile. For usage, see the example application makefile generated when you run **nios2-app-generate-makefile**.

Figure 4–5. HAL BSP After Build



alt_sys_init.c

The **alt_sys_init.c** file is a generated file used to initialize device driver instances and software packages.

For further details about this file, see the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*.

system.h

The **system.h** file is a generated file that contains the memory map and other system information.

For further details about this file, see the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*.

linker.h

The **linker.h** file is a generated file that contains information about the linker memory layout. **system.h** includes the **linker.h** file for backwards compatibility with code developed in the Nios II IDE.

linker.x

The **linker.x** file is a generated file that contains a linker script for the GNU linker. The **linker.x** file is the same as the **generated.x** file created by the Nios II IDE.

memory.gdb

The **memory.gdb** file is a generated file that contains memory region declarations for the GNU debugger. The **memory.gdb** file is the same as the **generated.gdb** file created by the Nios II IDE.

HAL Directory

The **HAL** directory contains HAL source code files. These are all copied files. The **src** directory contains the C-language and assembly-language source files. The **inc** directory contains the header files.

The **crt0.S** source file, containing HAL C run-time startup code, resides in the **HAL/src** directory.

drivers Directory

The **drivers** directory contains all driver source code. These are all copied files. The **drivers** directory has **src** and **inc** subdirectories like the **HAL** directory.

obj Directory

The **obj** directory contains the object code files for all source files in the BSP. The hierarchy of the BSP source files is preserved under the **obj** directory.

libhal_bsp.a Library

The **libhal_bsp.a** file contains the HAL BSP library. All object files are combined into the library file.

User-managed library files are always named **libhal_bsp.a**.

Figure 4–5 on page 4–17 shows the **my_hal_bsp** directory after executing **make.**

Micrium MicroC/OS-II BSP

The Micrium MicroC/OS-II is a multi-threaded run-time environment. It is built on the Altera HAL.

The MicroC/OS-II directory structure is a superset of the HAL BSP directory structure. All HAL BSP generated files also exist in the MicroC/OS-II BSP.

The MicroC/OS-II source code resides under the **UCOSII** directory. The **UCOSII** directory is under the BSP directory, like the **HAL** directory, and has the same structure (that is, **src** and **inc** directories). The **UCOSII** directory contains only copied files.

The MicroC/OS-II BSP library archive is named **libucosii_bsp.a**. You use this file the same way you use **libhal_bsp.a** in a HAL BSP.

Common BSP Tasks

nios2-bsp creates a BSP for you with useful default settings. However, for many tasks you need to manipulate the BSP explicitly. This section describes some common BSP tasks, and how you carry them out. The following tasks are covered:

- "Using Version Control" on page 4–20
- "Copying, Moving, or Renaming a BSP" on page 4–21
- "Handing Off a BSP" on page 4–22
- "Running a Nios II System with ModelSim" on page 4–22
- "Creating Memory Initialization Files" on page 4–23
- "Modifying Linker Memory Regions" on page 4–24
- "Creating a Custom Linker Section" on page 4–26
- "Changing the Default Linker Memory Region" on page 4–31
- "Changing a Linker Section Mapping" on page 4–31
- "Creating a BSP for a Nios Development Board" on page 4–32

- “Querying Settings” on page 4–33
- “Managing Device Drivers” on page 4–34
- “Creating a Custom Version of newlib” on page 4–34
- “Controlling the stdio Device” on page 4–35

Adding the Nios II Software Build Tools to Your Tool Flow

A common reason for using the software build tools is to enable you to integrate your software build process with other tools that you use for system development, including non-Altera tools. This section describes several scenarios where you can incorporate the build tools into an existing toolchain.

Using Version Control

One common issue is version control. By placing an entire software project, including both source and makefiles, under version control, you can ensure reproducible results from software builds.

When you are using version control, it is important to know exactly what files you need to add to your version control database. With the Nios II software build tools, the version control requirements are a function of what you are trying to do and of how you create the BSP.

If you create a BSP by running your own script that calls **nios2-bsp**, you can put your script under version control. If your script provides any Tcl scripts to **nios2-bsp** (using the `--script` option), you must also put these Tcl scripts under version control. If you install a new release of Nios II EDS and run your script to create a new BSP or to update an existing BSP, the internal implementation of your BSP might change slightly due to improvements in Nios II EDS.

If you create a BSP by running **nios2-bsp** manually on the command line or by running your own script that calls **nios2-bsp-generate-files**, you can put your BSP settings file (typically named **settings.bsp**) under version control. As in the scripted **nios2-bsp** case, if you install a new release of Nios II EDS and re-create your BSP, the internal implementation might change slightly.

If you want the exact same BSP after installing a new release of Nios II EDS, create your BSP and then put the entire BSP directory under version control before running **make**. If you have already run **make**, run `make clean` to remove all built files before adding the directory contents to your version control database. The BSP generator puts all the files required to build a BSP in the BSP directory. If you install a new release of Nios II EDS and run **make** on your BSP, the implementation is the same, but the binary output might not be identical.

If you create a script that uses the command line tools **nios2-bsp-create-settings** and **nios2-bsp-generate-files** explicitly, or you use these tools directly on the command line, it is possible to create the BSP settings file in a directory different from the directory where the generated BSP files reside. However, in most cases, when you want to store a BSP's generated files directory under source control, you also want to store the BSP settings file. Therefore, it is best to keep the settings file with the other BSP files. You can rebuild the project without the BSP settings file, but the settings file allows you to update and query the BSP.

Because the BSP depends on an SOPC Builder system file, you probably need to store the SOPC Builder system file in source control as well along with the BSP. The BSP settings file stores the SOPC Builder system file path as a relative or absolute path, depending on how it is entered with the **nios2-bsp** or **nios2-bsp-create-settings** commands. You need to take this into account when retrieving the BSP and the SOPC Builder system file from source control.

Copying, Moving, or Renaming a BSP

User-managed BSP makefiles have only relative path references to project source files, so you are free to copy, move or rename the entire BSP. If you specify a relative path to the SOPC system file when you create the BSP, you have to make sure the SOPC Builder system file can still be reached from the new location of the BSP. This SOPC Builder system file path is stored in the BSP settings file.

Do a `make clean` when you copy, move or rename a BSP. The **make** dependency (**.d**) files have absolute path references. `make clean` removes the **make** dependency files, as well as linker object files (**.o**) and archive files. You need to rebuild the BSP, of course, before linking an application with it. You can use the `make clean_bsp` command to combine these two operations.

For information about **make** dependency files, refer to the GNU **make** documentation, available at **www.gnu.org**.

Another way to copy a BSP is to run the **nios2-bsp-generate-files** command to populate a BSP directory and pass it the path to the BSP settings file of the BSP that you wish to copy.

If you rename or move a BSP, it is your responsibility to update any application or library makefile references to the old BSP name or location.

Handing Off a BSP

In some engineering organizations, one group (such as systems engineering) creates a BSP and hands it off to another group (such as applications software) to use while developing an application. In this situation, Altera recommends that you as the BSP developer generate the files for a BSP without building it (that is, do not run **make**) and then bundle the entire BSP directory, including the settings file, with a utility such as **tar** or **zip**. The software engineer who receives the BSP can then modify the BSP files as needed, or simply run **make** to build the BSP.

Running a Nios II System with ModelSim

To run a Nios II system with ModelSim®, you must create the simulation directory when you generate the system in SOPC Builder. First, make sure that the following environment variables are defined:

- `QUARTUS_PROJECT_DIR` — the path where your Quartus® II project resides.
- `SOPC_NAME` — the name of your Quartus II project.

Then generate the SOPC Builder system as follows:

```
sopc_builder --generate --simulation=1↵
```

This command creates a directory named `$(QUARTUS_PROJECT_DIR)/$(SOPC_NAME)`**_sim**.

Next, type:

```
make all mem_init_install↵
```

This command creates a **mem_init** directory under the application directory.

Copy the contents of this directory to the Quartus II Project directory, and copy the contents of **mem_init/hdlsim** to the `$(SOPC_NAME)`**_sim** directory.

Set the `$(SOPC_NAME)`**_sim** directory as the current working directory.

```
cd $(QUARTUS_PROJECT_DIR)/$(SOPC_NAME)_sim↵
```

Run ModelSim.

```
vsim↵
```

For more information about the `mem_init_install` make target, see "Creating Memory Initialization Files" on page 4–23.

Linking and Locating

When autogenerating a HAL BSP, the software build tools make some reasonable assumptions about how you want to use memory, as described in "Specifying the Default Memory Map" on page 4–48. However, in some cases these assumptions might not work for you. For example, you might implement a custom boot configuration that requires a bootloader in a specific location; or you might want to control what memory holds your interrupt service routines (ISRs).

This section describes several common scenarios where the software build tools allow you to control details of memory usage.

Creating Memory Initialization Files

The **mem_init.mk** file includes targets designed to help you create memory initialization files (**.dat**, **.hex**, **.sym**, and **.flash**). The **mem_init.mk** file is designed to be included in your application makefile. Memory initialization files are used for HDL simulation, for Quartus II compilation of initializable FPGA on-chip memories, and for flash programming. Initializable memories include M512 and M4K, but not MRAM.

Table 4–7 shows the **mem_init.mk** targets. Although the application

Table 4–7. mem_init.mk Targets

Target	Operation
`mem_init_install`	Generates memory initialization files in the application **mem_init** directory. If the `QUARTUS_PROJECT_DIR` variable is defined, **mem_init.mk** copies memory initialization files into your Quartus II project directory named `$(QUARTUS_PROJECT_DIR)`. If the `SOPC_NAME` variable is defined, **mem_init.mk** copies memory initialization files into your HDL simulation directory named `$(QUARTUS_PROJECT_DIR)/$(SOPC_NAME)_sim`.
`mem_init_generate`	Generates all memory initialization files in the application **mem_init** directory.
`mem_init_clean`	Removes the memory initialization files from the application **mem_init** directory.
`hex`	Generates all hex files.
`dat`	Generates all dat files.
`sym`	Generates all sym files.
`flash`	Generates all flash files.
<memory-name>	Generates all memory initialization files for *<memory-name>* component.

makefile provides all these targets, it does not invoke any of them by default. The makefile generator creates the memory initialization files in the application directory (under a directory named **mem_init**). It optionally copies them into your Quartus II project directory and HDL simulation directory, as described in Table 4–7.

☞ The BSP generator does not generate a definition of `QUARTUS_PROJECT_DIR` in your application makefile. If you have an on-chip random-access memory (RAM), and need to have a compiled software image inserted in your SRAM Object File at Quartus II compilation, you need to manually specify `QUARTUS_PROJECT_DIR` in your application makefile.

You must define `QUARTUS_PROJECT_DIR` before **mem_init.mk** file is included in the application makefile, as in the following example:

```
QUARTUS_PROJECT_DIR = ../my_hw_design
MEM_INIT_FILE := $(BSP_ROOT_DIR)/mem_init.mk
include $(MEM_INIT_FILE)
```

Modifying Linker Memory Regions

If the linker memory regions that are created by default do not meet your needs, there are BSP Tcl commands that let you modify the memory regions as desired.

Suppose you have a memory region named `onchip_ram`. Example 4–1 shows a Tcl script named **reserve_1024_onchip_ram.tcl** that separates out the top 1024 bytes of `onchip_ram` into a new region named `onchip_special`.

For an explanation of each Tcl command used in this example, see the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Example 4–1. Reserved Memory Region

```
# Get region information for onchip_ram memory region.
# Returned as a list.
set region_info [get_memory_region onchip_ram]
# Extract fields from region information list.
set region_name [lindex $region_info 0]
set slave_desc [lindex $region_info 1]
set offset [lindex $region_info 2]
set span [lindex $region_info 3]
# Remove the existing memory region.
delete_memory_region $region_name
```

```
# Compute memory ranges for replacement regions.
set split_span 1024
set new_span [expr $span-$split_span]
set split_offset [expr $offset+$new_span]
# Create two memory regions out of the original region.
add_memory_region onchip_ram $slave_desc $offset $new_span
add_memory_region onchip_special $slave_desc $split_offset $split_span
```

If you pass this Tcl script to **nios2-bsp**, it runs after the default Tcl script runs and sets up a linker region named `onchip_ram0`. You pass the Tcl script to **nios2-bsp** as follows:

```
nios2-bsp hal my_bsp --script reserve_1024_onchip_ram.tcl↵
```

If you run **nios2-bsp** again to update your BSP without providing the `--script` option, your BSP reverts to the default linker memory regions and your `onchip_special` memory region disappears. To preserve it, you can either provide the `--script` option to your Tcl script or pass the `DONT_CHANGE` keyword to the default Tcl script as follows:

```
nios2-bsp hal my_bsp --default_memory_regions DONT_CHANGE↵
```

Altera recommends using the `--script` approach when updating your BSP because it allows the default Tcl script to update memory regions if memories are added, removed, renamed, or re-sized. Using the `DONT_CHANGE` keyword approach does not handle any of these cases because the default Tcl script does not update the memory regions at all.

Creating a Custom Linker Section

The Nios II software build tools provide a Tcl command to create a linker section. The Nios II software build tools support the same default section names as the Nios II IDE. Table 4–8 lists the default section names.

Table 4–8. Nios II Default Section Names
`.entry`
`.exceptions`
`.text`
`.rodata`
`.rwdata`
`.bss`
`.heap`
`.stack`

The default Tcl script creates these default sections for you using the `add_section_mapping` Tcl command.

To create your own section named `special_section` that is mapped to the linker region named `onchip_special`, here is the Tcl command you use with **nios2-bsp**:

```
nios2-bsp hal my_bsp --cmd add_section_mapping special_section onchip_special↵
```

When the **nios2-bsp-generate-files** command (invoked by **nios2-bsp**) generates the linker script **linker.x**, the linker script has a new section mapping. The order of section mappings in the linker script is determined by the order in which the `add_section_mapping` command creates the sections. If you use **nios2-bsp**, the default Tcl script runs before the `--cmd` option that creates the `special_section` section.

If you run **nios2-bsp** again to update your BSP, you do not need to provide the `add_section_mapping` command again because the default Tcl script only modifies section mappings for the default sections listed in Table 4–8.

Dividing a Linker Region to Create a New Region and Section

Example 4–2 creates a section named `.isrs` out of the `tightly_coupled_instruction_memory` on-chip memory.

Example 4–2. Create hal_isrs_section.tcl script

```
# Get region information for tightly_coupled_instruction_memory memory
region.
# Returned as a list.
set region_info [get_memory_region tightly_coupled_instruction_memory]
# Extract fields from region information list.
set region_name [lindex $region_info 0]
set slave [lindex $region_info 1]
set offset [lindex $region_info 2]
set span [lindex $region_info 3]
# Remove the existing memory region.
delete_memory_region $region_name
# Compute memory ranges for replacement regions.
set split_span 1024
set new_span [expr $span-$split_span]
set split_offset [expr $offset+$new_span]
# Create two memory regions out of the original region.
add_memory_region tightly_coupled_instruction_memory $slave $offset
$new_span
add_memory_region isrs_region $slave $split_offset $split_span
add_section_mapping .isrs isrs_region
```

The following steps describe the use of this script:

1. Create a working directory for your hardware and software projects. The following steps refer to this directory as <*projects*>.

2. Make <*projects*> the current working directory.

3. Find the full-featured Nios II hardware example corresponding to your Nios development board. For example, if you have a Cyclone II development board, select <*Nios II EDS install path*>**/examples/verilog/niosII_cycloneII_2c35/full_featured**.

   This example uses the Verilog HDL standard hardware example design. You can select the language you prefer (Verilog HDL or VHDL)

4. Copy the hardware example into your working directory, using a command such as the following:

```
cp -R $SOPC_KIT_NIOS2/examples/verilog/niosII_cycloneII_2c35/full_featured .↵
```

5. Ensure that the working directory and all subdirectories are writable, as follows:

```
chmod -R +w .↵
```

6. The <*projects*> directory contains a subdirectory named **software_examples**/**bsp**. Make this directory the current working directory.

```
cd full_featured/software_examples/bsp↵
```

7. Under **bsp** there is a directory named **hal_default**, containing the **create-this-bsp** script for a default HAL-based BSP. Make a copy of this directory, named **hal_isrs_section**, and make it the current working directory.

```
cp -R hal_default hal_isrs_section↵
cd hal_isrs_section↵
```

8. Create **isrs_section_script.tcl**, as shown in Example 4–2 on page 4–27. This script splits off 1 KByte of RAM from the region named `tightly_coupled_instruction_memory`, gives it the name `isrs_region`, then calls **add_section_mapping** to add the `.isrs` section to `isrs_region`.

9. The <*projects*> directory contains a subdirectory named **software_examples/app/tcm**. Make this directory the current working directory.

```
cd ../../app/tcm↵
```

10. Edit the **create-this-app** script. Change occurrences of `hal_default` to `hal_isrs_section`.

11. Create and build the application with the `create-this-app` script as follows:

```
./create-this-app↵
```

12. Edit **timer_interrupt_latency.h**. In the `timer_interrupt_latency_irq()` function, change the `.section` directive from `.exceptions` to `.isrs`.

13. Rebuild the application by running **make.**

```
make↵
```

14. After **make** completes successfully, examine the object dump file, **tcm.objdump**, as shown in Example 4–3. You see the new `.isrs` section located in the tightly coupled instruction memory.

15. Examine the linker script file, **linker.x**, as shown in Example 4–4 on page 4–31. You see the new region `isrs_region` located in tightly-coupled instruction memory, adjacent to the `tightly_coupled_instruction_memory` region.

You can run the example by carrying out the following steps:

1. Open another shell and run **nios2-terminal**.

2. If your hardware is not already configured with the correct SRAM object file, enter the following command:

```
nios2-configure-sof ../../../*.sof↵
```

3. In your original shell, enter the following command:

```
nios2-download -g tcm.elf↵
```

Example 4–3. Excerpts from tcm.objdump

```
Sections:
Idx Name              Size      VMA       LMA       File off  Algn

    .
    .
    .

   6 .isrs             000000c0  04000c00  04000c00  000000b4  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE

    .
    .
    .

   9 .tightly_coupled_instruction_memory 00000000  04000000  04000000  00
013778  2**0
                  CONTENTS
    .
    .
    .

SYMBOL TABLE:
00000000 l    d  .entry  00000000
30000020 l    d  .exceptions  00000000
30000150 l    d  .text  00000000
30010e14 l    d  .rodata  00000000
30011788 l    d  .rwdata  00000000
30013624 l    d  .bss  00000000
04000c00 l    d  .isrs  00000000
00000020 l    d  .ext_flash  00000000
03200000 l    d  .epcs_controller  00000000
04000000 l    d  .tightly_coupled_instruction_memory  00000000
04004000 l    d  .tightly_coupled_data_memory  00000000

    .
    .
    .
```

Example 4–4. Excerpt From linker.x

```
MEMORY
{
 reset : ORIGIN = 0x0, LENGTH = 32
 tightly_coupled_instruction_memory : ORIGIN = 0x4000000, LENGTH = 3072
 isrs_region : ORIGIN = 0x4000c00, LENGTH = 1024

    .
    .
    .

}
```

Changing the Default Linker Memory Region

The default Tcl script chooses the largest memory region connected to your Nios II as the default region. All default memory sections specified in Table 4–8 on page 4–26 are mapped to this default region. You can pass in a command line option to the default Tcl script to override this default region. To map all default sections to `onchip_ram`, type the following command:

```
nios2-bsp hal my_bsp --default_sections_mapping onchip_ram↵
```

If you run **nios2-bsp** again to update your BSP, the default Tcl script overrides your default sections mapping. To prevent your default sections mapping from being changed, provide **nios2-bsp** with the original `--default_sections_mapping` command line option or pass it the `DONT_CHANGE` value for the memory name instead of `onchip_ram`.

Changing a Linker Section Mapping

If some of the default section mappings created by the default Tcl script do not meet your needs, you can use a Tcl command to override the section mappings selectively. To map the `.stack` and `.heap` sections into a memory region named `ram0`, use the following command:

```
nios2-bsp hal my_bsp --cmd add_section_mapping .stack ram0 \
    --cmd add_section_mapping .heap ram0↵
```

The other section mappings (for example, `.text`) are still mapped to the default linker memory region.

If you run **nios2-bsp** again to update your BSP, the default Tcl script overrides your section mappings for `.stack` and `.heap` because they are default sections. To prevent your section mappings from being changed, provide **nios2-bsp** with the original `add_section_mapping` command line options or pass the `--default_sections_mapping DONT_CHANGE` command line to **nios2-bsp**.

Altera recommends using the `--cmd` approach when updating your BSP because it allows the default Tcl script to update the default sections mapping if memories are added, removed, renamed, or re-sized.

Other BSP Tasks

This section covers some other common situations where the software build tools are useful.

Creating a BSP for a Nios Development Board

In some situations, you need to create a BSP separate from any application. Creating a BSP is similar to creating an application. To create a BSP, carry out the following steps:

1. Launch a command shell. Under Windows, use a Nios II Command Shell. Under Linux, use the shell of your preference.

2. Create a working directory for your hardware and software projects. The following steps refer to this directory as <*projects*>.

3. Make <*projects*> the current working directory.

4. Find a Nios II hardware example corresponding to your Nios development board. For example, if you have a 2C35 development board, you might select <*Nios II EDS install path*>**/examples/verilog/niosII_cycloneII_2c35/standard**.

   This example uses the Verilog HDL standard hardware example design. You can select the language you prefer (Verilog HDL or VHDL), and any type of example design except `small`.

5. Copy the hardware example into your working directory, using a command such as the following:

```
cp -R $SOPC_KIT_NIOS2/examples/verilog\
    /niosII_cycloneII_2c35/standard .↵
```

6. Ensure that the working directory and all subdirectories are writable, as follows:

```
chmod -R +w .↵
```

7. The *<projects>* directory contains a subdirectory named **software_examples/bsp**. Under **bsp** are several BSP example directories, such as **hal_default**. For a description of the example BSPs, see Table 4–3 on page 4–6. Select the directory containing an appropriate BSP, and make it the current working directory.

8. Create and build the BSP with the `create-this-bsp` script, as follows:

```
./create-this-bsp↵
```

At this point, you have a BSP, with which you can create and build an application.

☞ Altera recommends that you examine the contents of the **create-this-bsp** script. It might be a helpful example if you are creating your own script to build a BSP. **create-this-bsp** calls **nios2-bsp** with a few command line options to create a customized BSP, and then calls **make** to build the BSP.

Querying Settings

If you need to write a script that gets some information from the BSP settings file, use the **nios2-bsp-query-settings** command. To maintain upwards compatibility with future releases of the Nios II EDS, avoid developing your own code to parse the BSP settings file.

If you want to know the value of one or more settings, run **nios2-bsp-query-settings** with the appropriate command line options. It sends the values of the settings you requested to `stdout`. Just capture the output of `stdout` into some variable in your script when you call **nios2-bsp-query-settings**. By default, the output of **nios2-bsp-query-settings** is an ordered list of all option values. Use the `-show-names` option to display the name of the setting with its value.

For details of the **nios2-bsp-query-settings** command line options, see the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Managing Device Drivers

Like the Nios II IDE, by default the Nios II software build tools create an **alt_sys_init.c** file that assumes that every device connected to the Nios II that has a driver available uses that driver. However, you might want to use a different version of the driver, or you might not want a driver at all (for example, because your application accesses the device directly).

The BSP generator includes BSP Tcl commands to manage device drivers. With these commands you can control which driver is used for each device. When the **alt_sys_init.c** file is generated, it is set up to initialize drivers as you have requested.

If you are using **nios2-bsp**, you disable the driver for the `uart0` device as follows:

```
nios2-bsp hal my_bsp --cmd set_driver none uart0↵
```

Use the `--cmd` option to call a Tcl command on the command line. If you call **nios2-bsp-create-settings** instead of **nios2-bsp**, you also use the same `--cmd` option. You can also put the set_driver command in a Tcl script and pass the path to that script to **nios2-bsp** or **nios2-bsp-create-settings** with the `--script` option.

You replace the default driver for `uart0` with a specific version of a driver as follows:

```
nios2-bsp hal my_bsp --cmd set_driver altera_avalon_uart:6.1 uart0↵
```

Creating a Custom Version of newlib

The Nios II EDS comes with a number of pre-compiled libraries. These libraries include the newlib libraries (**libc.a** and **libm.a**). The Nios II software build tools allow you to create your own custom compiled version of the newlib libraries.

To create a custom compiled version of newlib, set a BSP setting to the desired compiler flags. If you are using **nios2-bsp**, you use the following command:

```
nios2-bsp hal my_bsp --set CUSTOM_NEWLIB_FLAGS "-O0 -pg"↵
```

Because newlib uses the open source **configure** utility, its build flow differs from other files in the BSP. When **Makefile** builds the BSP, it invokes the **configure** utility. The **configure** utility creates a makefile in the build directory, which compiles the newlib source. The newlib library files end up in the BSP directory named newlib. The newlib source files are not copied from Nios II EDS into the BSP.

Controlling the stdio Device

To prevent a default `stdio` device from being chosen, use the following command:

```
nios2-bsp hal my_bsp --default_stdio none↵
```

To override the default `stdio` device and replace it with `uart1`, use the following command:

```
nios2-bsp hal my_bsp --default_stdio uart1↵
```

To override the `stderr` device and replace it with `uart2`, while allowing the default Tcl script to choose the default `stdout` and `stdin` devices, use the following command:

```
nios2-bsp hal my_bsp --set hal.stderr uart2↵
```

In all these cases, if you run **nios2-bsp** again to update your BSP, you need to provide the original command line options again or else the default Tcl script chooses its own default `stdio` devices. Alternatively, you can call `--default_stdio` with the `DONT_CHANGE` keyword to prevent the default Tcl script from changing the `stdio` device settings.

Porting Nios II IDE Projects

If you have a Nios II IDE-managed system library, application, or library project, you do not have to rewrite the code to use the Nios II software build tools. However, the BSP generator uses a different directory structure and settings file format than the IDE. Therefore, you need to port IDE-managed projects to the Nios II software build tools manually. This section describes the required steps.

Applications

Open the application project in the Nios II IDE to determine any settings that you changed from the default. Use the **nios2-app-generate-makefile** command to create a makefile in your corresponding application project directory. Use command line options or edit the generated makefile to match the settings of your project. Make sure to provide the path to all of your application source files and to provide the path to your BSP.

The Nios II software build tools flow does not include separate Debug and Release builds as implemented in the Nios II IDE design flow. Make sure to port compiler flags, like the optimization level, debug, and custom instruction options, to the application **Makefile**.

System Libraries

Open the system library project in the Nios II IDE to determine any settings that you changed from the default. Create a new BSP directory and use the **nios2-bsp** script, **nios2-bsp-create-settings** or **nios2-bsp-generate-files** to populate it. Use command line options, Tcl scripts, or both to set the BSP settings to match system library settings.

If you have a HAL system library without any operating system extension, create a HAL BSP. If you have a HAL system library with the MicroC/OS-II operating system extension, create a MicroC/OS-II BSP. For details, see "Setting the BSP Type" on page 4–56.

If your system only uses Altera-provided hardware components or software packages, the Nios II software build tools copy and link in support files like device driver header files. If you have custom components that require specialized device driver support, or if you use third party components and software packages, further work is required to have Nios II software build tools manage the custom device driver files and settings.

User Libraries

Porting a user library to the software build tools is almost exactly the same as porting an application. Open the library project in the Nios II IDE to determine any settings that you changed from the default. Use the **nios2-lib-generate-makefile** command to create a makefile in your library project directory. Use command line options or edit the generated makefile to match the settings of your project. Make sure to provide the path to all of your library source files. If your library is dependent on your BSP, provide the path to the BSP.

The Nios II software build tools flow does not have the concept of Debug and Release build as implemented in the Nios II IDE design flow. Make sure to port compiler flags, like the optimization level, debug, and custom instruction options, to the application **Makefile**.

Using the Nios II C2H Compiler

The Nios II software build tools support the Nios II C2H compiler via the **nios2-c2h-generate-makefile** command. The following walk-through outlines how to use this command to create and build a software project with a C2H accelerator.

1. Create a working directory for your hardware and software projects. The following steps refer to this directory as <*projects*>.

2. Locate a Nios II hardware example corresponding to your Nios development board, and copy the hardware example into your *<projects>* working directory.

3. Select an application in a subdirectory of **software_examples/app** in the *<projects>* directory. The following steps refer to the application directory as *<application>*.

4. Select a BSP appropriate to your application. The following steps refer to the BSP directory as *<BSP>*. Create and build the BSP with the **create-this-bsp** script.

5. Create the application project, as follows:

```
nios2-app-generate-makefile --c2h --bsp-dir <BSP> --src-dir <application>↵
```

The `--c2h` command line option causes **Makefile** to include the C2H makefile fragment, **c2h.mk**.

6. Create the C2H makefile fragment, as follows:

```
nios2-c2h-generate-makefile \
    --sopc=../c2h_tutorial_hw/NiosII_<board name>_standard_sopc.sopc \
    --accelerator=do_dma,dma_c2h_tutorial.c --enable_quartus=1↵
```

When **nios2-c2h-generate-makefile** completes, you can find the makefile fragment, **c2h.mk**, in the ***<application>*** directory.

7. Build the application project, by typing `make`. To build the project, the makefiles carry out the following tasks:

   a. Launch the C2H Compiler to analyze the accelerated function, generate the hardware accelerator, and generate the C wrapper function.

   b. Invoke SOPC Builder to connect the accelerator into the SOPC Builder system. The build process modifies the SOPC Builder system file (**.sopc**) to include the new accelerator as a component in the system.

   c. Invoke the Quartus II software to compile the hardware project and regenerate the SRAM object file.

   d. Rebuild the C/C++ application project and link the accelerator wrapper function into the application.

For more detail about **nios2-c2h-generate-makefile**, see the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*. For more detail about the C2H acceleration example given here, refer to the *Getting Started Tutorial* chapter of the *Nios II C2H Compiler User Guide*.

Details of BSP Creation

Figure 4–6 on page 4–42 shows more details about how **nios2-bsp** creates a BSP. The **nios2-bsp** command determines whether a BSP already exists, and uses the **nios2-bsp-create-settings** command to create a new BSP settings file or the **nios2-bsp-update-settings** command to update an existing BSP settings file. For detailed information about BSP settings files, see "BSP Settings File Creation" on page 4–43. **nios2-bsp** assumes that the BSP settings file is named **settings.bsp** and resides in the BSP directory, which you specify on the **nios2-bsp** command line.

nios2-bsp uses the **nios2-bsp-generate-files** command to create the BSP files. The **nios2-bsp-generate-files** command places all source files in your BSP directory. It copies some files from the Nios II EDS installation directory. Others, such as **system.h** and **Makefile,** it generates dynamically.

nios2-bsp manages copied files differently from generated files. If copied files, such as source files, already exist, it does not overwrite them. Subsequent **runs** of **nios2-bsp-generate-files** do not overwrite these files. Preserving existing copied files allows you to directly modify C source files in any BSP, for example to customize a device driver.

By contrast, **nios2-bsp** always overwrites generated files, such as the BSP **Makefile**, **system.h**, and **linker.x**. A comment at the top of each generated file warns you not to edit it.

CAUTION

Nothing prevents you from modifying a generated file. However, once you do so, you can no longer update your BSP to match changes in your SOPC Builder system. If you update your BSP (by running **nios2-bsp** or **nios2-bsp-update-settings**), your changes to the generated file are destroyed.

Tcl Scripts for Board Support Package Settings

You control the characteristics of your BSP by manipulating BSP settings, using the Tcl commands described in the *"Settings for BSPs, Software Packages and Device Drivers"* section in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*. The most powerful way of using Tcl commands is by combining them into Tcl scripts.

The **nios2-bsp-create-settings**, **nios2-bsp-query-settings**, and **nios2-bsp-update-settings** commands all support Tcl scripts, via the `--script` command line argument.

The Tcl script in Example 4–5 is a very simple example that sets `stdio` to a device with the name `my_uart`.

Example 4–5. Simple Tcl script

```
set default_stdio my_uart
set_setting hal.stdin $default_stdio
set_setting hal.stdout $default_stdio
set_setting hal.stderr $default_stdio
```

The advantage of Tcl scripts over command line arguments is that Tcl scripts can obtain information from **nios2-bsp** and then use it later in the script. See Example 4–6 for an illustration of how you might use this facility. This Tcl script is similar to **bsp-stdio-utils.tcl**, which examines the hardware system and determines what device to use for `stdio`.

Example 4–6. Tcl Script to Examine Hardware and Choose Settings

```
# Select a device connected to the CPU as the default STDIO device.

# It returns the slave descriptor of the selected device.
# It gives first preference to devices with stdio in the name.
# It gives second preference to JTAG UARTs.
# If no JTAG UARTs are found, it uses the last character device.
# If no character devices are found, it returns “none”.

# Procedure that does all the work of determining the stdio device
proc choose_default_stdio {} {
    set last_stdio “none”
    set first_jtag_uart “none”

    # Get all slaves attached to the CPU.
    set slave_descs [get_slave_descs]

    foreach slave_desc $slave_descs {
        # Lookup module class name for slave descriptor.
        set module_name [get_module_name $slave_desc]
        set module_class_name [get_module_class_name $module_name]

        # If the module_name contains “stdio”, we’ll choose it
        # and return immediately.
        if { [regexp .*stdio.* $module_name] } {
            return $slave_desc
        }

        # Assume it is a JTAG UART if the module class name contains
        # the string “jtag_uart”.  In that case, return the first one
        # found.
        if { [regexp .*jtag_uart.* $module_class_name] } {
            if {$first_jtag_uart == “none”} {
                set first_jtag_uart $slave_desc
            }
        }

        # Track last character device in case no JTAG UARTs found.
        if { [is_char_device $slave_desc] } {
            set last_stdio $slave_desc
        }
    }

    if {$first_jtag_uart != “none”} {
        return $first_jtag_uart
    }

    return $last_stdio
}

# Call routine to determine stdio
set default_stdio [choose_default_stdio]

# Set stdio settings to use results of above call.
set_setting hal.stdin $default_stdio
set_setting hal.stdout $default_stdio
set_setting hal.stderr $default_stdio
```

nios2-bsp uses a Tcl script (named **bsp-set-defaults.tcl**) to specify default values for system-dependent settings. System-dependent settings are BSP settings that make references to system information in the SOPC Builder system file.

For details about the default Tcl script, see “Specifying BSP Defaults” on page 4–45. The path to the default Tcl script is passed to **nios2-bsp-create-settings** or **nios2-bsp-update-settings** before any user input. As a result, user input overrides settings made by the default Tcl script. You can also pass in command line options to the default Tcl script to override the choices it makes or to prevent it from making changes to settings.

The default Tcl script makes the following choices for you based on your SOPC Builder system:

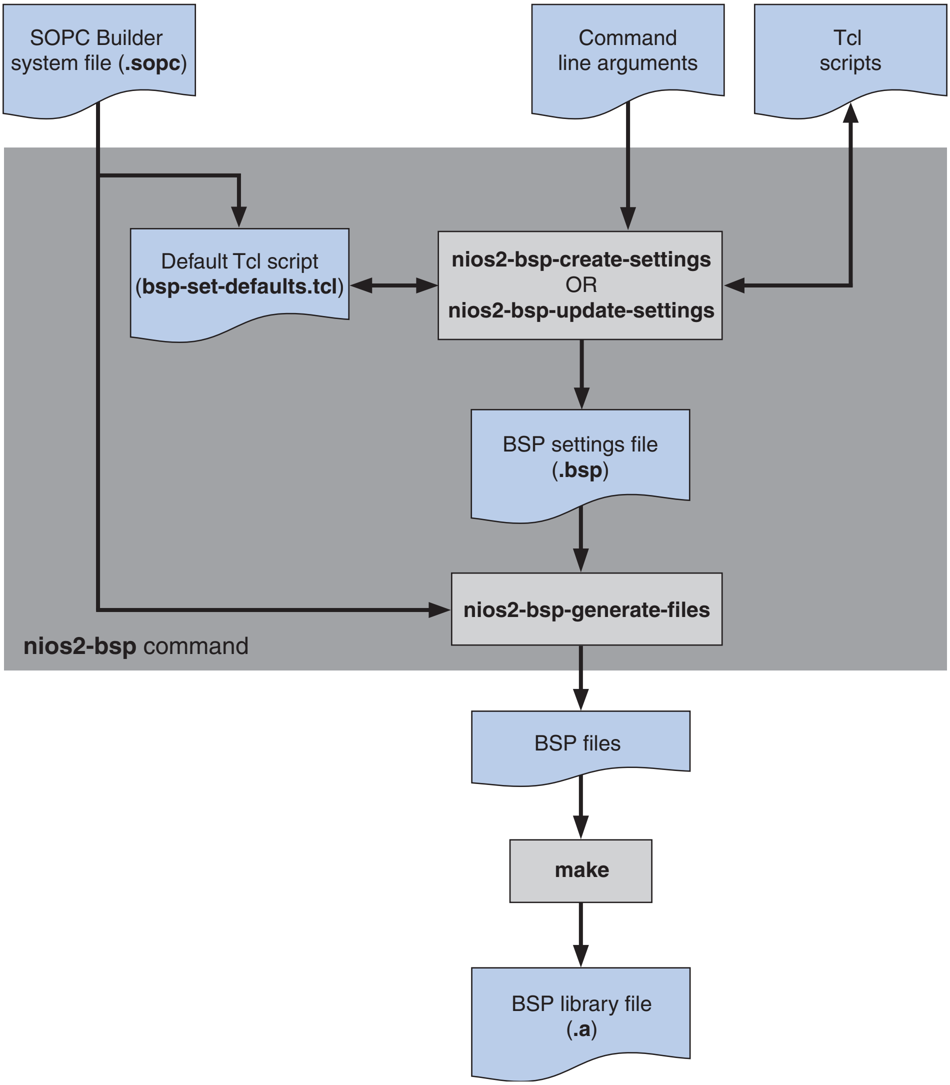
- stdio character device
- System timer device
- Default linker memory regions
- Default linker sections mapping
- Default boot loader settings

The default Tcl scripts use slave descriptors to assign devices. For further information about slave descriptors, see "Device Drivers and Software Packages" on page 4–50.

If a component has only one slave port connected to the Nios II, the slave descriptor is the same as the name of the component (for example, onchip_mem_0). If a component has multiple slave ports connecting the Nios II to multiple resources in the component, the slave descriptor is the name of the component followed by an underscore and the slave port name (for example, onchip_mem_0_s1).

Figure 4–6 on page 4–42 shows that the default Tcl script and **nios2-bsp-generate-files** both use the SOPC Builder system file. The BSP settings file does not need to duplicate system information (such as base addresses of devices), because the **nios2-bsp-generate-files** command has access to the SOPC Builder system file.

Figure 4–6. nios2-bsp Command Expanded Flow

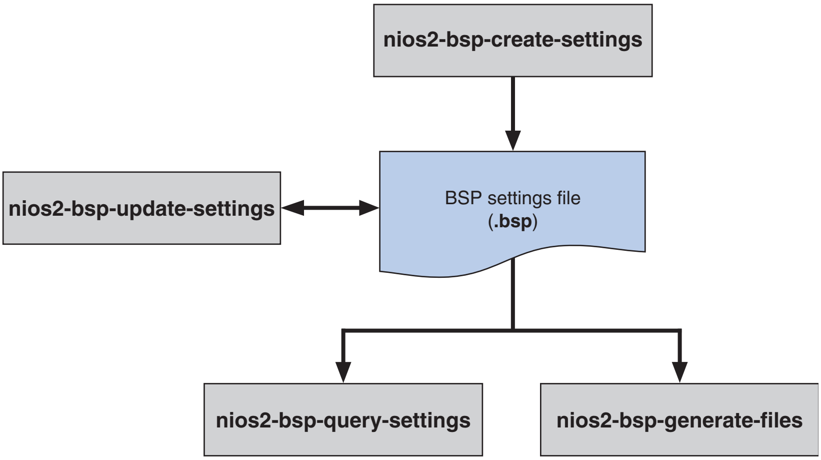


BSP Settings File Creation

Each BSP has an associated settings file that saves the values of all BSP settings. The BSP settings file is in extensible markup language (XML) format and has a **.bsp** extension by convention. When you create or update your BSP, the BSP generator writes the value of all settings into the settings file.

Figure 4–7 shows how the BSP generator interacts with the BSP settings file. The **nios2-bsp-create-settings** command creates a new BSP settings file. The **nios2-bsp-update-settings** command updates an existing BSP settings file. The **nios2-bsp-query-settings** command reports the setting values in an existing BSP settings file. The **nios2-bsp-generate-files** command generates a BSP from the BSP settings file.

Figure 4–7. BSP Settings File and BSP Commands



Modifying the BSP

You may need to update an existing BSP if your requirements change (for example, you change the compiler optimization level), or because of changes to the SOPC Builder system to which it refers. There are three approaches to updating a BSP. In order of preference, these approaches are:

- Recreate the BSP using a Tcl script
- Run **nios2-bsp**
- Run **nios2-bsp-update-settings** and **nios2-bsp-generate-files**

The following sections discuss each approach.

Recreate the BSP Using a Tcl Script

This approach gives you the maximum control. When you initially create the BSP, create a Tcl script specifying all BSP settings. Use the same Tcl script to recreate the BSP.

The Tcl script explicitly specifies the contents of the settings file. Because you are recreating the settings file as well as all generated files, you can guarantee that system-dependent settings are adjusted correctly based on any changes in the SOPC Builder system.

Run nios2-bsp

This approach keeps your settings up to date with your SOPC Builder system in most circumstances.

nios2-bsp runs **nios2-bsp-update-settings** to update the settings file as needed and then runs **nios2-bsp-generate-files** to update your BSP files. You can then run **make** to build a new BSP library file.

When you run **nios2-bsp** on an existing BSP, you generally do not need to supply the command line arguments and Tcl scripts. Most of the original BSP settings persist in the BSP settings file.

The exception is default settings specified by the default Tcl script. **nios2-bsp** executes the default Tcl script every time it runs, overwriting previous default settings. If you want to preserve all settings, including the default settings, use the `DONT_CHANGE` keyword, described in "Top Level Script for BSP Defaults" on page 4–46. Alternatively, you can provide **nios2-bsp** with command line options or Tcl scripts to override the default settings.

Run nios2-bsp-update-settings and nios2-bsp-generate-files

You can use this approach if you are certain that your settings file needs updating.

Coordinating with SOPC Builder System Changes

Every BSP is based on a Nios II processor in an SOPC Builder system. If the SOPC Builder system changes after you generate your BSP, you usually need to update the BSP.

If all BSP system-dependent settings are still consistent with the new SOPC Builder system file, you can just run **nios2-bsp-generate-files** with the existing BSP settings file to create an updated BSP. **nios2-bsp-generate-files** reads the SOPC Builder system file for basic system parameters such as module base addresses and clock frequencies.

The following list shows examples of system changes that do not affect BSP system-dependent settings. Although these changes do not require you to regenerate your settings file, you still need to run the **nios2-bsp-generate-files** command to regenerate files such as the **Makefile**, **system.h**, and the linker script.

- Changing base addresses
- Changing interrupt numbers
- Changing clock frequencies
- Changing most processor options (for example, cache size or core type)
- Changing most component options (except for the size of memories)
- Adding bridges
- Adding new components
- Removing or renaming non-memory components other than the `stdio` device or system timer device
- Adding or removing interrupts

The following are examples of system changes that **do** affect BSP system-dependent settings:

- Renaming the processor
- Renaming or removing memories, the `stdio` device, or the system timer device
- Changing memory sizes
- Changing the processor reset and exception slave ports or offsets

If changes to your SOPC Builder system file make it inconsistent with your BSP, you must update or recreate the BSP. Use one of the methods described in "Modifying the BSP" on page 4–43.

Specifying BSP Defaults

Table 4–9 lists the components of the BSP default Tcl scripts, included in the BSP generator. These scripts specify default BSP settings. The scripts all located in:

<Nios II EDS install path>`/sdk2/bin`

Table 4–9. Default Tcl Script Components

Script	Level	Summary
bsp-set-defaults.tcl	Top-level	Sets system-dependent settings to default values.
bsp-call-proc.tcl	Top-level	Calls a specified procedure in one of the helper scripts.
bsp-stdio-utils.tcl	Helper	Specifies `stdio` device settings.
bsp-timer-utils.tcl	Helper	Specifies system timer device setting.
bsp-linker-utils.tcl	Helper	Specifies memory regions and section mappings for linker script.
bsp-bootloader-utils.tcl	Helper	Specifies boot loader-related settings.

Top Level Script for BSP Defaults

The top level Tcl script for settings BSP defaults is **bsp-set-defaults.tcl**. This script specifies BSP system-dependent settings, which are a function of the SOPC Builder system. **nios2-bsp-create-settings** and **nios2-bsp-update-settings** do not call the default Tcl script when creating or updating a BSP settings file. The `--script` option must be used to specify **bsp-set-defaults.tcl** explicitly. The **nios2-bsp** command invokes the default Tcl script by calling either **nios2-bsp-create-settings** or **nios2-bsp-update-settings** with the `--script bsp-set-defaults.tcl` option.

The default Tcl script consists of a top-level Tcl script named **bsp-set-defaults.tcl** plus the helper Tcl scripts listed in Table 4–9. The helper Tcl scripts do the real work of examining the SOPC Builder system file and choosing appropriate defaults.

The **bsp-set-defaults.tcl** script sets the following defaults:

- `stdio` character device (**bsp-stdio-utils.tcl**)
- System timer device (**bsp-timer-utils.tcl**)
- Default linker memory regions (**bsp-linker-utils.tcl**)
- Default linker sections mapping (**bsp-linker-utils.tcl**)
- Default boot loader settings (**bsp-bootloader-utils.tcl**)

You run the default Tcl script in the context of a `--script` argument to the **nios2-bsp-create-settings**, **nios2-bsp-query-settings**, or **nios2-bsp-update-settings** command. It has the following usage:

`bsp-set-defaults.tcl` [*<argument name> <argument value>*]*

Table 4–10 on page 4–47 lists default Tcl script arguments in detail. All arguments are optional. If present, each argument must be in the form of a name and argument value, separated by white space. All argument values are strings. For any argument not specified, the corresponding helper script chooses a suitable default value. In every case, if the argument value is DONT_CHANGE, the default Tcl scripts leave the setting unchanged. The DONT_CHANGE value allows fine-grained control of what settings the default Tcl script changes and is useful when updating an existing BSP.

Table 4–10. Default Tcl Script Command Line Options

Argument Name	Argument Value
default_stdio	Slave descriptor of default stdio device (stdin, stdout, stderr). Set to none if no stdio device desired.
default_sys_timer	Slave descriptor of default system timer device. Set to none if no system timer device desired.
default_memory_regions	Controls generation of memory regions By default, **bsp-linker-utils.tcl** removes and regenerates all current memory regions. Use the DONT_CHANGE keyword to suppress this behavior.
default_sections_mapping	Slave descriptor of the memory device to which the default sections are mapped. This argument has no effect if default_memory_regions == DONT_CHANGE.
enable_bootloader	Boolean: 1 if a boot loader is present; 0 otherwise.

Specifying the Default stdio Device

The **bsp-stdio-utils.tcl** script provides procedures to choose a default stdio slave descriptor and to set the hal.stdin, hal.stdout, and hal.stderr BSP settings to that value.

For more information about these settings, see the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

The script searches the SOPC Builder system file for a slave descriptor with the string stdio in its name. If **bsp-stdio-utils.tcl** finds any such slave descriptors, it chooses the first as the default stdio device. If the script finds no such slave descriptor, it looks for a slave descriptor with the string jtag_uart in its component class name. If it finds any such slave descriptors, it chooses the first as the default stdio device. If the script finds no slave descriptors fitting either description, it chooses the last character device slave descriptor connected to the Nios II. If **bsp-stdio-utils.tcl** does not find any character devices, there is no `stdio` device.

Specifying the Default System Timer

The **bsp-timer-utils.tcl** script provides procedures to choose a default system timer slave descriptor and to set the `hal.sys_clk_timer` BSP setting to that value.

For more information about this setting, see the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

The script searches the SOPC Builder system file for a slave descriptor with the string "timer" in its component class name. If found, the script chooses the first such slave descriptor connected to the Nios II, giving preference to any slave descriptor with the string "sys_clk" in its name. If not found, there is no system timer device.

Specifying the Default Memory Map

The **bsp-linker-utils.tcl** script provides procedures to add the default linker script memory regions and map the default linker script sections to a default region. The **bsp-linker-utils.tcl** script uses the `add_memory_region` and `add_section_mapping` BSP Tcl commands.

For more information about these commands, see the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

The script chooses the largest volatile memory region as the default memory region. If there is no volatile memory region, **bsp-linker-utils.tcl** chooses the largest non-volatile memory region. The script maps the `.text`, `.rodata`, `.rwdata`, `.bss`, `.heap`, and `.stack` section mappings to this default memory region. The script also sets the `hal.linker.exception_stack_memory_region` BSP setting to the default memory region. The setting is available in case the separate exception stack option is enabled (this setting is disabled by default).

For more information about this setting, see the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Specifying Default Bootloader Parameters

The **bsp-bootloader-utils.tcl** script provides procedures to specify the following BSP boolean settings:

- `hal.linker.allow_code_at_reset`
- `hal.linker.enable_alt_load_copy_rodata`
- `hal.linker.enable_alt_load_copy_rwdata`
- `hal.linker.enable_alt_load_copy_exceptions`

For more information about these settings, see the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

The script examines the `.text` section mapping and the Nios II reset slave port. If the `.text` section is mapped to the same memory as the Nios II reset slave port and the reset slave port is a flash memory device, the script assumes that a boot loader is being used. You can override this behavior by passing the `enable_bootloader` option to the default Tcl script.

Table 4–11 shows how the **bsp-bootloader-utils.tcl** script specifies the value of boot loader-dependent settings. If a boot loader is enabled, the assumption is that the boot loader is located at the reset address and handles the copying of sections on reset. If there is no boot loader, the BSP might need to provide code to handle these functions. You can use the `alt_load()` function to implement a boot loader.

Table 4–11. Boot Loader-Dependent Settings

Setting name *(1)*	Value When Boot Loader Enabled	Value When Boot Loader Disabled
`hal.linker.allow_code_at_reset`	0	1
`hal.linker.enable_alt_load_copy_rodata`	0	1 if `.rodata` memory different than `.text` memory and `.rodata` memory is volatile; 0 otherwise
`hal.linker.enable_alt_load_copy_rwdata`	0	1 if `.rwdata` memory different than `.text` memory; 0 otherwise
`hal.linker.enable_alt_load_copy_exceptions`	0	1 if `.exceptions` memory different than `.text` memory and `.exceptions` memory is volatile; 0 otherwise

Notes to Table 4–11:

(1) For further information about these settings, see the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Invoking Procedures in the Default Tcl Script

The procedure call Tcl script consists of the top-level **bsp-call-proc.tcl** script plus the helper scripts listed in Table 4–9 on page 4–46. The procedure call Tcl script allows you to invoke a specific procedure in the helper scripts, if you want to invoke some of the default Tcl functionality without running the entire default Tcl script.

The procedure call Tcl script has the following usage:

`bsp-call-proc.tcl` <*proc-name*> [<*args*>]`*`

bsp-call-proc.tcl calls the specified procedure with the specified (optional) arguments. Refer to the default Tcl scripts to see what functions are available and their arguments. The **bsp-call-proc.tcl** script includes the same files as the **bsp-set-defaults.tcl** script, so any function in those included files is available.

Device Drivers and Software Packages

The BSP generator can incorporate device drivers and software packages supplied by Altera, supplied by other third-party developers, or created by you. The process required to develop a device driver is nearly identical to that required to develop a software package. This section describes how to create device drivers and software packages, and prepare them so the BSP generator recognizes and adds them to a generated BSP.

Assumptions and Requirements

This section makes the following assumptions about the device driver or software package that you are developing.

- You develop a device driver or software package for eventual incorporation into a BSP. This section assumes that the driver or package is to be incorporated into the BSP by an end user who has limited knowledge of the driver or package internal implementation. To add your driver or package to a BSP, the end user can rely on the driver or package settings that you create with the tools described in this section.
- After BSP generation, the device driver resides in a directory, called the root directory, which might have subdirectories.
- Your device driver is to be compatible with both the Nios II IDE design flow and the Nios II software build tools design flow. The Nios II software build tools provide a less rigid set of requirements for your drivers and software packages. However, Altera recommends that you use the Nios II IDE conventions to maintain build-flow compatibility.

- You are familiar with Altera's guidelines for developing device driver and software package source code for the Nios II IDE design flow.

For a device driver or software package to work in the Nios II software build tools design flow, it must meet the following criteria:

- It must have a defining Tcl script. The Tcl script for each driver or software package provides the BSP generator with a complete description of the driver or software. This description includes the following information:
  - Name — A unique name identifying the driver or software package
  - Source files — The location, name, and type of each C/C++ or assembly language source or header file
  - Associated hardware class (device drivers only) — The name of the hardware peripheral class the driver supports
  - Versioning and compatibility information
  - BSP type(s) — Supported operating system(s)
  - Settings — Visible parameters controlling software build and runtime configuration
- The Tcl script resides in the driver or software package root directory
- The Tcl script's file name ends with **_sw.tcl.** Example: **custom_ip_block_sw.tcl.**
- The root directory of the driver or software package is in one of the following places:
  - In any directory appended to the `SOPC_BUILDER_PATH` environment variable, or in any directory located one level beneath it. This approach is recommended if your driver or software packages are installed in a distribution you create.
  - In any directory one level beneath the Quartus II project directory containing the design your BSP targets. This approach is recommended if your driver or software package is used only once, in a specific hardware project.
- File names and directory structures conform to the conventions described in the *Developing Device Drivers for the HAL* chapter of the *Nios II Software Developer's Handbook.*
- Each device driver has a **component.mk** file, as described in the *Developing Device Drivers for the HAL* chapter of the *Nios II Software Developer's Handbook.* This does not apply to software packages.
- If your driver or software package uses the HAL autoinitialization mechanism, the `INSTANCE` and `INIT` macros must be defined in a header file named <*hardware component class*>**.h**.

  ☞ This convention matches the Nios II IDE design flow.

For information on writing a device driver or software package suitable for use with the Nios II IDE design flow, please refer to *The Hardware Abstraction Layer* section of the *Nios II Software Developer's Handbook*. The *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook* describes the commands you can use in the Tcl Script.

The Nios II software build tools use slave descriptors to refer to components connected to the Nios II. A slave descriptor is the unique name of an SOPC Builder component's slave port.

The Nios II BSP Generator Flow

When you invoke any BSP generator utility, a library of available drivers and software packages is populated.

The BSP generator locates software packages and drivers by inspecting a list of known locations determined by the Altera Nios II EDS, Quartus II software, and MegaCore IP Library installers, as well as searching locations specified in certain system environment variables. The Nios II BSP tools identify drivers and software packages by locating and sourcing Tcl scripts with file names ending in **_sw.tcl** in these locations.

After locating each driver and software package, the Nios II software build tools search for a suitable driver for each hardware module in the SOPC system (mastered by the Nios II CPU that the BSP is generated for), as well as software packages that the BSP creator requested.

In the case of device drivers, the highest version of driver that is compatible with the associated hardware peripheral is added to the BSP, unless specified otherwise by the device driver management commands.

For further information, see the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

The BSP generator adds software packages to the BSP if they are specifically requested during BSP generation.

For further details, see `enable_sw_package` in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*

If no specific device driver is requested, and no compatible device driver is located for a particular hardware module, the BSP generator issues an informative message visible in either the `debug` or `verbose` generation output. This behavior is normal for many types of hardware in the SOPC

Builder system, such as memory devices, that do not have device drivers. If a software package or specific driver is requested and cannot be located, an error is generated and BSP generation or settings update halts.

When the Nios II software build tools add each driver or software package to the system, they use the data in the Tcl script defining the driver or software package to control each file copied in to the BSP. This rule also affects generated BSP files such as the BSP **Makefile**, **public.mk**, **system.h**, and the BSP settings and summary HTML files.

File Names and Locations

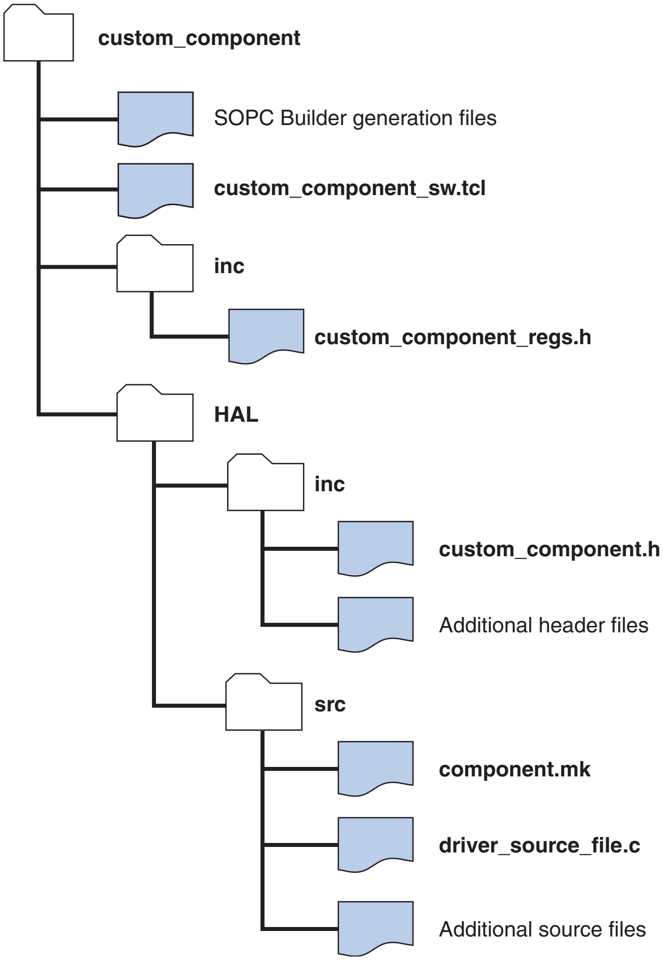
The Nios II BSP tools find device drivers and software packages by searching for Tcl scripts and sourcing them. The tools can find Tcl scripts for drivers and software packages installed with the Nios II EDS, Quartus II software, and MegaCore IP libraries.

☞ The Nios II IDE finds any user-defined components added to the Nios II EDS, Quartus II software, or MegaCore IP library installations alongside other Altera components. For run-time efficiency, the BSP generator only looks at driver files that conform to the criteria described in "The Nios II BSP Generator Flow" on page 4–52.

Example

Figure 4–8 illustrates a file hierarchy suitable for the Nios II software build tools design flow. This example assumes a device driver supporting a hardware component named `custom_component`.

Figure 4–8. Example Device Driver File Hierarchy and Naming



The file hierarchy shown in Figure 4–8 on page 4–54 is also compatible with the Nios II IDE.

Driver and Software Package Tcl Script Creation

This section discusses writing a Tcl script to describe your software package or driver. The exact contents of the Tcl script depends on the structure and complexity of your driver or software. For many simple device drivers, you only need to include a few commands. For more complex software, the Nios II software build tools provide powerful features to give the end-user of the BSP control of your software or driver's operation.

The Tcl command and argument descriptions in this section are not exhaustive. Please refer to the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook* for a detailed explanation of each command and all possible arguments.

For a reference in creating your own driver or software Tcl files, you can also view the driver and software package Tcl scripts included with Nios II EDS and the MegaCore IP library. These scripts are in the <*Nios II EDS install path*>**/components** and <*MegaCore IP library install path*>**/sopc_builder_ip** folders, respectively.

Tcl Command Walkthrough for a Typical Driver or Software Package

The following Tcl excerpts describe a typical device driver or software package.

The example in this section creates a device driver for a hardware peripheral whose SOPC Builder component class name is `my_custom_component`. The driver supports both HAL and MicroC/OS-II BSP types. It has a single C source file (**.c**) and two C header files (**.h**), organized like the example in Figure 4–8 on page 4–54.

Creating and Naming the Driver or Package

The first command in any driver or software package Tcl script must be the **create_driver** or **create_sw_package** command (the remaining commands can be in any order). Use the appropriate **create** command only once per Tcl file. Choose a unique driver or package name. For drivers, Altera recommends appending `_driver` to the associated hardware class name. This is illustrated in the following example:

```
create_driver my_custom_component_driver
```

Identifying the Hardware Component Class

Each driver must identify the hardware component class the driver is associated with in the **set_sw_property** command's `hw_class_name` argument. The following example associates the driver with a hardware class called `my_custom_component`:

```
set_sw_property hw_class_name my_custom_component
```

☞ The **set_sw_property** command accepts several argument types. Each call to **set_sw_property** sets or overwrites a property to the value specified in the second argument. For further information about **set_sw_property**, see the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

The hw_class_name argument does not apply to software packages.

If you are creating your own driver to use in place of an existing one (for example, a custom UART driver for the altera_avalon_uart component), specify a driver name different from the standard driver. The Nios II software build tools use your driver only if you specify it explicitly.

For further details, see the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Choose a name for your driver or software package that does not conflict with other Altera-supplied software or IP, or any third-party software or IP installed on your host system. The BSP generator uses the name you specify to look up the software package or driver during BSP creation. If the Nios II software build tools find multiple compatible drivers or software packages with the same name, they might pick any of them.

If you intend to distribute your driver or software package, Altera recommends prefixing all names with your organization's name.

Setting the BSP Type

You must specify each operating system (or BSP type) that your driver or software package supports. Use the **add_sw_property** command's supported_bsp_type argument to specify each compatible operating system. In most cases, a driver or software package supports both Altera HAL (hal) and Micrium MicroC/OS-II (ucosii) BSP types, as in the following example:

```
add_sw_property supported_bsp_type hal
add_sw_property supported_bsp_type ucosii
```

☞ The **add_sw_property** command accepts several argument types. Each call to **add_sw_property** adds the final argument to the property specified in the second argument.

☞ Support for additional operating system and BSP types is not present in this release of the Nios II software build tools.

Specifying an Operating System

Many drivers and software packages do not require any particular operating system. However, you can structure your software to provide different source files depending on the operating system used.

If your driver or software has different source files, paths, or settings that depend on the operating system used, write a Tcl script for each variant of the driver or software package. Each script must specify the same software package or driver name in the **create_driver** or **create_sw_package** command, and same `hw_class_name` in the case of device drivers. Each script must specify only the files, paths, and other settings that pertain to that operating system. During BSP generation, only drivers or software packages that specify compatibility with the selected OS type are eligible to add to the BSP.

Specifying Source Files

Using the Tcl command interface, you must specify each source file in your driver or software package that you want in the generated BSP. The commands discussed in this section add driver source files and specify their location in the file-system and generated BSP.

The **add_sw_property** command's `c_source` and `asm_source` arguments add a single C or Nios II assembly (**.s** or **.S**) source file to your driver or software package. You must express path information to the source relative to the driver root (the location of the Tcl file). **add_sw_property** copies source files into BSPs that incorporate the driver, using the path information specified, and adds them to source file list in the generated BSP **Makefile**. When you compile the BSP using **make**, the driver source files are compiled as follows:

```
add_sw_property c_source HAL/src/my_driver.c
```

The **add_sw_property** command's `include_source` argument adds a single header file in the path specified to the driver. The paths are relative to the driver root. **add_sw_property** copies header files into the BSP during generation, using the path information specified at generation time. It does not include header files in the makefile.

```
add_sw_property include_source inc/my_custom_component_regs.h
add_sw_property include_source HAL/inc/my_custom_component.h
```

Specifying a Subdirectory

You can optionally specify a subdirectory in the generated BSP for your driver or software package files using the `bsp_subdirectory` argument to **set_sw_property**. All driver source and header files are copied into this directory, along with any path or hierarchy information specified with each source or header file. If no bsp_subdirectory is specified, your driver or software package is placed under the **drivers** folder of the generated BSP. Set the subdirectory as follows:

```
set_sw_property bsp_subdirectory my_driver
```

☞ If the path begins with the BSP type (e.g HAL or UCOSII), the BSP type is removed and replaced with the value of the bsp_subdirectory property. The path is modified this way to provide compatibility with the Nios II IDE design flow, as well as to provide clarity in the generated BSP directory structure.

Enabling Software Initialization

If your driver or software package uses the HAL autoinitialization mechanism, your source code includes INSTANCE and INIT macros, to create storage for each driver instance, and to call any initialization routines. The generated **alt_sys_init.c** file invokes these macros, which must be defined in a header file named <*hardware component class*>**.h**.

For further detail, refer to the *Developing Device Drivers for the HAL* chapter of the *Nios II Software Developer's Handbook*.

To support this functionality in Nios II BSPs, you must set the **set_sw_property** command's auto_initialize argument to true, as follows:

```
set_sw_property auto_initialize true
```

If you do not turn on this attribute, **alt_sys_init.c** does not invoke the INIT and INSTANCE macros.

Adding Include Paths

By default, the generated BSP **Makefile** and **public.mk** add include paths to find header files in **/inc** or <*BSP type*>**/inc** folders.

You might need to set up a header file directory hierarchy to logically organize your code. You can add additional include paths to your driver or software package using the **add_sw_property** command's include_directory argument as follows:

```
add_sw_property include_directory UCOSII/inc/protocol/h
```

Similar to adding source files, if the directory begins with the BSP type, it is stripped and replaced with the bsp_subdirectory provided. This matches the behavior of where the files are copied to during BSP generation.

Additional include paths are added to the pre-processor flags in the BSP **public.mk** file. These pre-processor flags allow BSP source files, as well as application and library source files that reference the BSP, to find the include path while each source file is compiled.

☞ Adding additional include paths is not required if your source code includes header files with explicit path names. You can also specify the location of the header files with a `#include` directive similar to the following:

```
#include "protocol/h/<filename>"
```

Version Compatibility

Your device driver or software package can optionally specify versioning information through the Tcl command interface. The driver and software package Tcl commands specifying versioning information allow the following functionality:

- You can request a specific version of your driver or software package via BSP settings
- You can make updates to your device driver and specify that the driver is still compatible with a minimum hardware class version, or specific hardware class versions. This facility is especially useful in situations where a hardware design is stable and you foresee making software updates over time.

The *<version>* argument in each of the following versioning-related commands can be a string containing numbers and characters. Examples of version strings: `1.0`, `5.1.1`, `6.1`, `6.1sp1`. The `"."` character is treated as a separator. The BSP generator compares versions against each other to determine if one is more recent than the other, or if two are equal, by successively comparing the strings between each separator. Thus, `2.1` is greater than `2.0`, and `2.1sp1` is greater than `2.1`. Two versions are equal if their version assignment strings are identical.

Use the `version` argument of **set_sw_property** to assign a version to your driver or software package. If you do not assign a version to your software or device driver, the version of the Nios II EDS installation (containing the Nios II BSP commands being executed) is set for your driver or software package:

```
set_sw_property version 7.1
```

Device drivers (but not software packages) can use the `min_compatible_hw_version` and `specific_compatible_hw_version` arguments to establish compatibility with their associated hardware class:

```
set_sw_property min_compatible_hw_version 5.0.1
add_sw_property specific_compatible_hw_version 6.1sp1
```

You can add multiple specific compatible versions. This functionality allows you to roll out a new version of a device driver that tracks changes supporting a hardware peripheral change.

For device drivers, if no compatible version information is specified, the version of the device driver must be equal to the associated hardware class. Thus, if you do not wish to use this feature, Altera recommends setting the `min_compatible_hw_version` of your driver to the lowest version of the associated hardware class your driver is compatible with.

Creating Settings for Device Drivers and Software Packages

The BSP generator allows you to publish settings for individual device drivers and software packages. These settings are visible and can be modified by the BSP user, if the BSP includes your driver or software package. Use the Tcl command interface to create settings.

The Tcl command that publishes settings is especially useful if your driver or software package has build or runtime options that are normally specified with `#define` statements or makefile definitions at software build-time. Settings can also add custom variable declarations to the BSP **Makefile**.

Settings affect the generated BSP in several ways:

- As additions to either the BSP **system.h** or **public.mk**, or variable additions to the BSP **Makefile**
- Settings are stored in the BSP settings file, named with hierarchy information to prevent namespace collision
- A default value of your choice is assigned to the setting so that the end user of the driver or package does not need to explicitly specify the setting when creating or updating a BSP
- Settings are displayed in the BSP **summary.html** document, along with description text of your choice

Use the **add_sw_setting** Tcl command to add a setting. **add_sw_setting** requires each of the following arguments, in order, to specify the details:

1. `type` — The data type, which controls formatting of the setting's value assignment in the appropriate generated file

2. `destination` — The destination file in the BSP

3. displayName — The name that is used to identify the setting when changing BSP settings or viewing the BSP **summary.html** document

4. identifier — Conceptually, this argument is the macro defined in C language definition (the text immediately following #define), or the name of a variable in a makefile.

5. value — A default value assigned to the setting if the BSP user does not manually change it

6. description — Descriptive text, shown in the BSP **summary.html** document.

Data Types

Several setting data types are available, controlled by the type argument to **add_sw_setting**. They correspond to the data types you can express as #define statements or values concatenated to makefile variables. The specific setting type depends on your software's structure or BSP build needs. The available data types, and their typical uses, are shown in Table 4–12.

Table 4–12. Data Type Settings

Data Type	Setting Value	Notes
Boolean definition	boolean_define_only	A definition that is generated when true, and absent when false. Use a boolean definition in your C source files with the `#ifdef <setting> ... #endif` construct.
Boolean assignment	boolean	A definition assigned to 1 when true, 0 when false. Use a boolean assignment in your C source files with the `#if <setting> ... #else ...` construct.
Character	character	A definition with one character surrounded by single quotation marks (')
Decimal number	decimal_number	A definition with an unquoted, unformatted decimal number, such as 123. Useful for defining values in software that, for example, might have a configurable buffer size, such as `int buffer[SIZE];`
Double precision number	double	A definition with a double-precision floating point number such as 123.4
Floating point number	float	A definition with a single-precision floating point number such as 234.5
Hexadecimal number	hex_number	A definition with a number prefixed with `0x`, such as `0x1000`. Useful for specifying memory addresses or bit masks
Quoted string	quoted_string	A definition with a string in quotes, such as `"Buffer"`
Unquoted string	unquoted_string	A definition with a string not in quotes, such as `BUFFER`

Setting Destination Files

The `destination` argument of **add_sw_setting** specifies settings and their assigned values. This argument controls which file the setting is saved to in the BSP. The BSP generator formats the setting's assigned value based on the definition file and type of setting. Table 4–13 shows possible values of the `destination` argument.

Table 4–13. Destination File Settings

Destination File	Setting Value	Notes
system.h	system_h_define	This destination file is recommended in most cases. Your source code must use a `#include <system.h>` statement to make the setting definitions available. Settings appear as `#define` statements in **system.h**.
public.mk	public_mk_define	Definitions appear as `-D` statements in **public.mk**, in the C preprocessor flags assembly. This setting type is passed directly to the compiler during BSP and is visible during compilation of application and libraries referencing the BSP.
BSP makefile	makefile_variable	Settings appear as makefile variable assignments in the BSP makefile.

☞ Certain setting types are not compatible with the **public.mk** or **Makefile** destination file types. For detailed information, see the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Setting Display Name

The setting `displayName` controls what the end user of the driver or package (the BSP developer) types to control the setting in their BSP. BSPs append the `displayName` text after a "." (dot) separator to your driver or software package's name (as defined in the **create_driver** or **create_sw_package** command). For example, if your driver is named `my_peripheral_driver` and your setting's `displayName` is `small_driver`, BSPs with your driver have a setting `my_peripheral_driver.small_driver`. Thus each driver and software package has its own settings name-space.

Setting Generation Name

The setting `generationName` of **add_sw_setting** controls the physical name of the setting in the generated BSP files. The physical name corresponds to the definition being created in **public.mk** and **system.h**, or the make variable created in the BSP **Makefile**. The `generationName` is commonly the text that your software uses in conditionally-compiled code. For example, if your software creates a buffer:

```
unsigned int driver_buffer[MY_DRIVER_BUFFER_SIZE];
```

Enter the exact text, `MY_DRIVER_BUFFER_SIZE`, in the `generationName` argument.

Setting Default Value

The `value` argument of **add_sw_setting** holds the default value of your setting. This value propagates to the generated BSP unless the end user of the driver or package (the BSP developer) changes the setting's assignment before BSP generation.

☞ The value assigned to any setting, whether it is the default value in the driver or software package Tcl script, or entered by the user configuring the BSP, must be compatible with the selected setting. For details, see the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Setting Description

The `description` argument of **add_sw_setting** contains a brief description of the setting. The `description` argument is required. Place quotation marks (`""`) around the text of the description. The description text appears in the generated BSP **summary.html** document.

Setting Creation Example

Example 4–7 implements a setting for a driver that has two variants of a function, one implementing a “small” (code footprint) and the other a “fast” (efficient execution) as follows:

Example 4–7. Supporting Driver Settings

```
#include "system.h"
#ifdef MY_CUSTOM_DRIVER_SMALL
int send_data( <args> )
{
 // Small implementation
}
#else
int send_data( <args> )
{
 // fast implementation
}
#endif
```

In Example 4–7, a simple Boolean definition setting is added to your driver Tcl file. This feature allows BSP users to control your driver through the BSP settings interface. When users set the setting to `true` or `1`, the BSP defines `MY_CUSTOM_DRIVER_SMALL` in either **system.h** or the BSP **public.mk** file. When the user compiles the BSP, your driver is compiled with the appropriate routine is built into the resultant object. When a user disables the setting, `MY_CUSTOM_DRIVER_SMALL` is not defined.

The above setting is added to your driver or software package as follows using the **add_sw_setting** Tcl command:

```
add_sw_setting boolean_define_only system_h_define small_driver
    MY_CUSTOM_DRIVER_SMALL false
    "Enable the small implementation of the driver for my_peripheral"
```

☞ Each Tcl command must reside on a single line of the Tcl file. This example is wrapped due to space constraints.

There are several variants of each argument. For detailed usage and restrictions, see the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*

Porting Advanced Nios II IDE Projects

Some Nios II IDE projects consist of an application project plus a system library based on standard drivers. You can convert such a project to a user-managed makefile project by following the procedures described in "Porting Nios II IDE Projects" on page 4–35. However, if your project includes more advanced components, such as precompiled libraries, you need to take additional steps to port those components to the user-managed makefile flow. This section discusses porting advanced project components.

Custom Component Device Drivers

In the Nios II IDE design flow, a makefile fragment named **component.mk** specifies device drivers. By contrast, in the Nios II software build tools design flow, a Tcl script defines the device driver structure. For specific information on upgrading IDE-managed device drivers to work with the Nios II software build tools design flow, see "Device Drivers and Software Packages" on page 4–50.

Creating a Tcl script allows you to put extra definitions into the **system.h** file, enable automatic driver initialization through the **alt_sys_init.c** structure, and enable the Nios II software build tools to control any extra parameters that might exist.

With the Tcl software definition files are in place, the BSP generator reads in the Tcl file and populates the makefiles and other support files accordingly.

As an alternative to creating a driver, you can compile the device-specific code as a library, using the **nios2-lib-generate-makefile** tool, and link it with the application. This approach is workable if the device-specific code is independent of the BSP, and does not require any of the extra services offered by the BSP, such as the ability to add definitions to the **system.h** file.

Precompiled Libraries

You can add precompiled libraries to the BSP **public.mk** or application **Makefile**. The following variables must be updated in the makefile:

- `ALT_LIBRARY_DIRS` – Add path to directory in which **lib**<*name*>**.a** files reside.
- `ALT_LIBRARY_NAMES` – Names of the libraries being added. If the library is named **libuart_driver.a**, then **uart_driver** is typically the name.
- `ALT_LDDEPS` – Add full path to each of the archive files. This variable is used for linker dependency in the case where the precompiled library files are updated.
- `ALT_INCLUDE_DIRS` – Add path to directory in which C header files (**.h**) reside, if required

In the same manner, you can add libraries generated by running **nios2-lib-generate-makefile**.

Non-HAL Device Drivers

You can add precompiled non-HAL device drivers to the BSP **public.mk** or application **Makefile** the same way you add pre-compiled libraries. The following variables must be updated in the makefile:

- `ALT_LIBRARY_DIRS` – Add path to directory in which **lib**<*name*>**.a** files reside.
- `ALT_LIBRARY_NAMES` – Names of the libraries being added. If the library is named **libuart_driver.a**, then **uart_driver** is typically the name.
- `ALT_LDDEPS` – Add full path to each of the archive files. This variable is used for linker dependency in the case where the precompiled library files are updated.
- `ALT_INCLUDE_DIRS` – Add path to directory in which C header files (**.h**) reside, if required

Non-HAL device drivers work in a HAL BSP. However, they do not use built-in HAL mechanisms such as **alt_sys_init.c**.

Boot Configurations

The HAL and MicroC/OS-II BSPs support several boot configurations. The default Tcl script configures an appropriate boot configuration based on your SOPC Builder system and other settings.

For detailed information about the HAL boot loader process, see the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*.

Table 4–14 shows the memory types that the default Tcl script is aware of in making decisions about your boot configuration. The default Tcl script uses the `IsFlash` and `IsNonVolatileStorage` properties to determine what kind of memory is in the system.

The `IsFlash` property of the memory module (defined in the SOPC Builder system file) indicates whether the SOPC Builder system file identifies the memory as a flash memory device. The `IsNonVolatileStorage` property indicates whether the SOPC Builder system file identifies the memory as a non-volatile storage device. The contents of a non-volatile memory device are fixed and always present.

☞ Some FPGA memories can be initialized when the FPGA is configured. They are not considered non-volatile because the default Tcl script has no way to determine whether they are actually initialized in a particular system.

Table 4–14. Memory Types

Memory Type	Examples	IsFlash	IsNonVolatileStorage
Flash	Common flash interface (CFI), erasable programmable configurable serial (EPCS) device	true	true
ROM	On-chip memory configured as read-only memory (ROM), Hardcopy ROM	false	true
RAM	On-chip memory configured as random-access memory (RAM), Hardcopy RAM, synchronous dynamic random access memory (SDRAM), synchronous static random access memory (SSRAM)	false	false

The following sections describe each supported build configuration in detail. The `alt_load()` facility is HAL code that optionally copies sections from the boot memory into RAM. You can set an option to enable the boot copy. This option only adds the code to your BSP if it needs to copy boot segments. The `hal.enable_alt_load` setting enables `alt_load()` and there are settings for each of the three sections it can copy (such as `hal.enable_alt_load_copy_rodata`). Enabling `alt_load()` also has the effect of modifying the memory layout specified in your linker script.

Boot from Flash Configuration

The reset address points to a boot loader in a flash memory. The boot loader initializes the instruction cache, copies each memory section to its virtual memory address (VMA), and then jumps to `_start`.

This boot configuration has the following characteristics:

- `alt_load()` not called
- No code at reset in executable file

The default Tcl script chooses this configuration when the memory associated with the CPU reset address is a flash memory and the `.text` section is mapped to a different memory (for example, SDRAM).

Altera provides example boot loaders for CFI and EPCS in the Nios II EDS, precompiled into Motorola S-record (**.srec**) files. You can use one of these example boot loaders, or provide your own.

Boot from Monitor Configuration

The reset address points to a monitor in a nonvolatile ROM or initialized RAM. The monitor initializes the instruction cache, downloads the application memory image (for example, using a UART or Ethernet connection), and then jumps to the entry point provided in the memory image.

This boot configuration has the following characteristics:

- `alt_load()` not called
- No code at reset in executable file

The default Tcl script assumes no boot loader is in use, so it never chooses this configuration unless you enable it. To enable this configuration, pass the following argument to the default Tcl script:

```
enable_bootloader 1
```

If you are using the **nios2-bsp** command, invoke it as follows:

```
nios2-bsp hal my_bsp --use_bootloader 1↵
```

Run from Initialized Memory Configuration

The reset address points to the beginning of the application in memory (no boot loader). The reset memory must have its contents initialized before the CPU comes out of reset. The initialization might occur by means such as using a non-volatile reset memory (for example, flash, ROM, initialized FPGA RAM) or by an external master (for example, another CPU) that writes the reset memory. The HAL C run-time startup code (`crt0`) initializes the instruction cache, uses `alt_load()` to copy select sections to their VMAs, and then jumps to `_start`. For each associated section (`.rwdata`, `.rodata`, `.exceptions`), boolean settings control this behavior. The default Tcl scripts set these to default values as described in Table 4–11 on page 4–49.

`alt_load()` must copy the `.rwdata` section (either to another RAM or to a reserved area in the same RAM as the `.text` RAM) if `.rwdata` needs to be correct after multiple resets.

This boot configuration has the following characteristics:

- `alt_load()` called
- Code at reset in executable file

The default Tcl script chooses this configuration when the reset and `.text` memory are the same.

Run-time Configurable Reset Configuration

The reset address points to a memory that contains code that executes before the normal reset code. When the CPU comes out of reset, it executes code in the reset memory that computes the desired reset address and then jumps to it. This boot configuration allows a CPU with a hard-wired reset address to appear to reset to a programmable address.

This boot configuration has the following characteristics:

- `alt_load()` might be called (depends on boot configuration)
- No code at reset in executable file

Because the CPU reset address points to an additional memory, the algorithms used by the default Tcl script to select the appropriate boot configuration might make the wrong choice. The individual BSP settings specified by the default Tcl script need to be explicitly controlled.

Restrictions

The Nios II software build tools have the following restrictions:

- The Nios II software build tools are only supported by SOPC Builder release 7.1 or later. The Nios II software build tools require an SOPC Builder system file (**.sopc)** for the system description. If you have a legacy hardware design based on a **.ptf** file, SOPC Builder can convert your **.ptf** into an SOPC Builder system file.
- The Nios II software build tools do not directly support multiple build configurations (**Debug** and **Release**) as the Nios II IDE does. However, it is easy to copy an existing BSP and modify it to create the equivalent of a different build configuration. For details, see "Copying, Moving, or Renaming a BSP" on page 4–21.
- At release 7.1, the Nios II software build tools support Altera® hardware abstraction layer (HAL) and Micrium MicroC/OS-II only.

Referenced Documents

This chapter references the following documents:

- *Overview* chapter of the *Nios II Software Developer's Handbook*
- *Nios II Integrated Development Environment* chapter of the *Nios II Software Developer's Handbook*

- *Introduction to the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*
- *The Hardware Abstraction Layer* section of the *Nios II Software Developer's Handbook*
- *Overview of the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*
- *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*
- *Developing Device Drivers for the HAL* chapter of the *Nios II Software Developer's Handbook*
- *Altera-Provided Development Tools* chapter of the *Nios II Software Developer's Handbook*
- *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*

Document Revision History

Table 4–15 shows the revision history for this document.

Table 4–15. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Initial release. Material moved here from former *Nios II Software Build Tools* chapter.	—

Section II. The Hardware Abstraction Layer

This section provides information on the hardware abstraction layer (HAL).

This section includes the following chapters:

5. Overview of the Hardware Abstraction Layer

NII52003-7.2.0

Introduction

This chapter introduces the hardware abstraction layer (HAL) for the Nios® II processor. This chapter contains the following sections:

- "Getting Started" on page 5–1
- "HAL Architecture" on page 5–2
- "Supported Peripherals" on page 5–5

The HAL is a lightweight runtime environment that provides a simple device driver interface for programs to communicate with the underlying hardware. The HAL application program interface (API) is integrated with the ANSI C standard library. The HAL API allows you to access devices and files using familiar C library functions, such as `printf()`, `fopen()`, `fwrite()`, etc.

The HAL serves as a device driver package for Nios II processor systems, providing a consistent interface to the peripherals in your system. Tight integration between SOPC Builder and the Nios II software development tools automates the construction of a HAL instance for your hardware. After SOPC Builder generates a hardware system, the Nios II IDE or the Nios II software build tools can generate a custom HAL system library or board support package (BSP) to match the hardware configuration. Changes in the hardware configuration automatically propagate to the HAL device driver configuration, preventing changes in the underlying hardware from creating bugs.

HAL device driver abstraction provides a clear distinction between application and device driver software. This driver abstraction promotes reusable application code that is resistant to changes in the underlying hardware. In addition, the HAL standard makes it straightforward to write drivers for new hardware peripherals that are consistent with existing peripheral drivers.

Getting Started

The easiest way to get started using the HAL is to perform the tutorials provided with the Nios II IDE. In the process of creating a new project in the Nios II IDE, you also create a HAL system library. You do not have to create or copy HAL files, and you do not have to edit any of the HAL source code. The Nios II IDE generates and manages the HAL system library for you.

In the Nios II software build tools design flow, you can create an example BSP based on the HAL, using one of the **create-this-bsp** scripts supplied with the Nios II embedded design suite.

You must base the HAL on a specific SOPC Builder system. An SOPC Builder system is a Nios II processor core integrated with peripherals and memory (which is generated by SOPC Builder). If you do not have a custom SOPC Builder system, you can base your project on an Altera®-provided example hardware system. In fact, you can first start developing projects targeting an Altera Nios development board, and later re-target the project to a custom board. It is easy to change the target SOPC Builder system later.

For information about creating a new project with the Nios II IDE, refer to the *Nios II Integrated Development Environment* chapter of the *Nios II Software Developer's Handbook*, or to the help system in the Nios II IDE. For information about creating a new project with the Nios II software build tools, refer to the *Introduction to the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook.*

HAL Architecture

This section describes the fundamental elements of the HAL architecture.

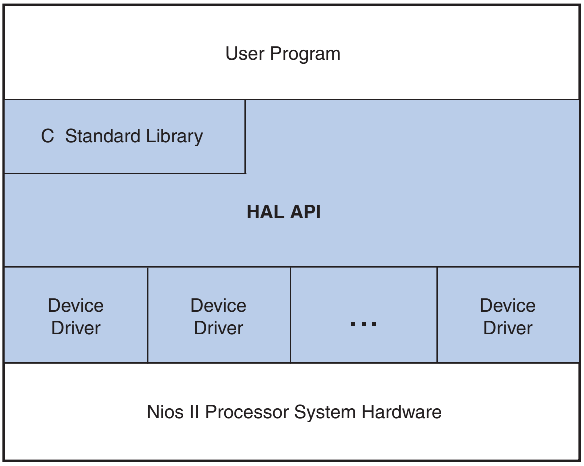
Services

The HAL provides the following services:

- Integration with the newlib ANSI C standard library—provides the familiar C standard library functions
- Device drivers—provides access to each device in the system
- The HAL API—provides a consistent, standard interface to HAL services, such as device access, interrupt handling, and alarm facilities
- System initialization—performs initialization tasks for the processor and the runtime environment before `main()`
- Device initialization—instantiates and initializes each device in the system before `main()`

Figure 5–1 shows the layers of a HAL-based system, from the hardware level up to a user program.

Figure 5–1. The Layers of a HAL-Based System



Applications vs. Drivers

Programmers fall into two distinct groups: application developers and device driver developers. Application developers are the majority of users, and are responsible for writing the system's `main()` routine, among other routines. Applications interact with system resources either through the C standard library, or through the HAL API. Device driver developers are responsible for making device resources available to application developers. Device drivers communicate directly with hardware through low-level hardware-access macros.

For further details on the HAL, refer to the following chapters:

- The *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook* describes how to take advantage of the HAL to write programs without considering the underlying hardware.
- The *Developing Device Drivers for the HAL* chapter of the *Nios II Software Developer's Handbook* describes how to communicate directly with hardware and how to make hardware resources available via the abstracted HAL API.

Generic Device Models

The HAL provides generic device models for classes of peripherals found in embedded systems, such as timers, Ethernet MAC/PHY chips, and I/O peripherals that transmit character data. The generic device models are at the core of the HAL's power. The generic device models allow you to write programs using a consistent API, regardless of the underlying hardware.

Device Model Classes

The HAL provides a model for the following classes of devices:

- Character-mode devices—hardware peripherals that send and/or receive characters serially, such as a UART.
- Timer devices—hardware peripherals that count clock ticks and can generate periodic interrupt requests.
- File subsystems—provide a mechanism for accessing files stored within physical device(s). Depending on the internal implementation, the file subsystem driver might access the underlying device(s) directly or use a separate device driver. For example, you can write a flash file subsystem driver that accesses flash using the HAL API for flash memory devices.
- Ethernet devices—provide access to an Ethernet connection for a networking stack such as the Altera-provided NicheStack® TCP/IP Stack - Nios II Edition. You need a networking stack to use an ethernet device.
- DMA devices—peripherals that perform bulk data transactions from a data source to a destination. Sources and destinations can be memory or another device, such as an Ethernet connection.
- Flash memory devices—nonvolatile memory devices that use a special programming protocol to store data.

Benefits to Application Developers

The HAL defines a set of functions that you use to initialize and access each class of device. The API is consistent, regardless of the underlying implementation of the device hardware. For example, to access character-mode devices and file subsystems, you can use the C standard library functions, such as `printf()` and `fopen()`. For application developers, you do not have to write low-level routines just to establish basic communication with the hardware for these classes of peripherals.

Benefits to Device Driver Developers

Each device model defines a set of driver functions necessary to manipulate the particular class of device. If you are writing drivers for a new peripheral, you only need to provide this set of driver functions. As a result, your driver development task is pre-defined and well documented. In addition, you can use existing HAL functions and applications to access the device, which saves software development effort. The HAL calls driver functions to access hardware. Application programmers call the ANSI C or HAL API to access hardware, rather than calling your driver routines directly. Therefore, the usage of your driver is already documented as part of the HAL API.

C Standard Library—newlib

The HAL integrates the ANSI C standard library into its runtime environment. The HAL uses newlib, an open-source implementation of the C standard library. newlib is a C library for use on embedded systems, making it a perfect match for the HAL and the Nios II processor. newlib licensing does not require you to release your source code or pay royalties for projects based on newlib.

The ANSI C standard library is well documented. Perhaps the most well-known reference is *The C Programming Language* by B. Kernighan and D. Ritchie, published by Prentice Hall and available in over 20 languages. Redhat also provides online documentation for newlib at **http://sources.redhat.com/newlib**.

Supported Peripherals

Altera provides many peripherals for use in Nios II processor systems. Most Altera peripherals provide HAL device drivers that allow you to access the hardware via the HAL API. The following Altera peripherals provide full HAL support:

- Character mode devices:
  - UART core
  - JTAG UART core
  - LCD 16207 display controller
- Flash memory devices
  - Common flash interface compliant flash chips
  - Altera’s EPCS serial configuration device controller
- File subsystems
  - Altera host based file system
  - Altera zip read-only file system
- Timer devices
  - Timer core
- DMA devices
  - DMA controller core
  - Scatter-gather DMA controller core
- Ethernet devices
  - Triple Speed Ethernet MegaCore
  - LAN91C111 Ethernet MAC/PHY Controller

The LAN91C111 and Triple Speed Ethernet components require the MicroC/OS-II runtime environment. For more information, refer to the *Ethernet and the NicheStack® TCP/IP Stack - Nios II Edition* chapter of the *Nios II Software Developer's Handbook.*

Third-party vendors offer additional peripherals not listed here. For a list of other peripherals available for the Nios II processor, visit Altera's Embedded Software Partners page at **http://www.altera.com/products/ip/processors/nios2/tools/embed-partners/ni2-embed-partners.html**.

All peripherals (both from Altera and third party vendors) must provide a header file that defines the peripheral's low-level interface to hardware. By this token, all peripherals support the HAL to some extent. However, some peripherals might not provide device drivers. If drivers are not available, use only the definitions provided in the header files to access the hardware. Do not access a peripheral using hard-coded addresses or other such "magic numbers".

Inevitably certain peripherals have hardware-specific features with usage requirements that do not map well to a general-purpose API. The HAL handles hardware-specific requirements by providing the UNIX-style `ioctl()` function. Because the hardware features depend on the peripheral, the `ioctl()` options are documented in the description for each peripheral.

Some peripherals provide dedicated accessor functions that are not based on the HAL generic device models. For example, Altera provides a general-purpose parallel I/O (PIO) core for use in Nios II processor system. The PIO peripheral does not fit into any class of generic device models provided by the HAL, and so it provides a header file and a few dedicated accessor functions only.

For complete details regarding software support for a peripheral, refer to the peripheral's description. For further details on Altera-provided peripherals, see the *Quartus® II Handbook, Volume 5: Embedded Peripherals*.

Referenced Documents

This chapter references the following documents:

- *Nios II Integrated Development Environment* chapter of the *Nios II Software Developer's Handbook*
- *Introduction to the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*
- *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*
- *Developing Device Drivers for the HAL* chapter of the *Nios II Software Developer's Handbook*

- *NicheStack TCP/IP Stack - Nios II Edition* chapter of the *Nios II Software Developer's Handbook*
- *Quartus II Handbook, Volume 5: Embedded Peripherals*

Document Revision History

Table 5–1 shows the revision history for this document.

Table 5–1. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	No change from previous release.	
May 2007 v7.1.0	● Scatter-gather DMA core ● Triple-speed Ethernet MAC ● Refer to HAL generation with Nios II software build tools. ● Chapter 4 was formerly chapter 3. ● Added table of contents to Introduction section. ● Added Referenced Documents section.	● Scatter-gather DMA core ● Triple-speed Ethernet MAC ● Nios II software build tools
March 2007 v7.0.0	No change from previous release.	
November 2006 v6.1.0	NicheStack TCP/IP Stack - Nios II Edition	
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.	
May 2004 v1.0	Initial Release.	

6. Developing Programs Using the Hardware Abstraction Layer

NII52004-7.2.0

Introduction

This chapter discusses how to develop programs for the Nios® II processor based on the Altera® hardware abstraction layer (HAL). This chapter contains the following sections:

The API for HAL-based systems is readily accessible to software developers who are new to the Nios II processor. Programs based on the HAL use the ANSI C standard library functions and runtime environment, and access hardware resources via the HAL API's generic device models. The HAL API largely conforms to the familiar ANSI C standard library functions, though the ANSI C standard library is separate from the HAL. The close integration of the ANSI C standard library and the HAL makes it possible to develop useful programs that never call the HAL functions directly. For example, you can manipulate character mode devices and files using the ANSI C standard library I/O functions, such as `printf()` and `scanf()`.

☞ This document does not cover the ANSI C standard library. An excellent reference is *The C Programming Language, Second Edition*, by Brian Kernighan and Dennis M. Ritchie (Prentice-Hall).

Nios II Design Flows

As described in the *Overview* chapter of the *Nios II Software Developer's Handbook*, the Nios II EDS offers the following two distinct design flows:

- The Nios II IDE design flow
- The Nios II software build tools design flow

Most of the information in this chapter applies to both design flows. Design flow differences are noted explicitly.

☞ Both design flows create board support packages (BSPs). However, the Nios II IDE design flow refers to a BSP as a system library.

For more detailed information about developing programs in the Nios II software build tools design flow, refer to the *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

HAL BSP Settings

Every Nios II BSP possesses settings, which determine the BSP's characteristics. For example, HAL BSPs have settings that determine the hardware components associated with standard devices such as `stdout`. Defining and manipulating BSP settings is an important part of Nios II project creation.

How you manipulate BSP settings depends on which design flow you are using. In the Nios II IDE, you manipulate BSP (system library) settings through the **System Library Properties** page. With the Nios II software build tools, you manipulate BSP settings with command line options or Tcl scripts.

For details of how to control BSP settings, refer to:

- The Nios II IDE help system — for IDE-managed projects
- The *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook* — for user-managed projects.

Many HAL settings are reflected in the **system.h** file, which can provide a helpful reference if you need to know details about your BSP. For information about **system.h**, refer to "The system.h System Description File" on page 6–4.

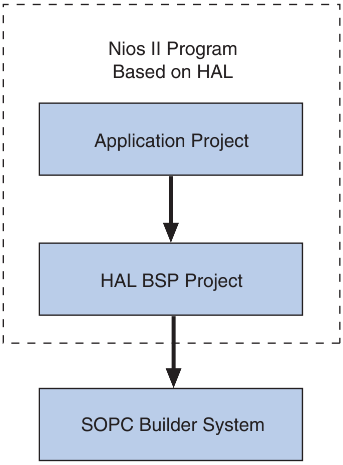
☞ Do not edit **system.h**. Both design flows provide tools to manipulate system settings.

The Nios II Project Structure

The creation and management of software projects based on the HAL is integrated tightly with both Nios II design flows. This section discusses the Nios II projects as a basis for understanding the HAL.

Figure 6–1 shows the blocks of a Nios II program with emphasis on how the HAL BSP fits in. The label for each block describes what or who generated that block, and an arrow points to each block's dependency.

Figure 6–1. The Nios II HAL Project Structure



Also known as: Your program, or user project
Defined by: .c, .h, .s files
Created by: You

Also known as: HAL system library project
Defined by: Nios II BSP settings
Created by: Nios II IDE or Nios II software build tools

Also known as: Nios II processor system, or the hardware
Defined by: .ptf or .sopc file
Created by: SOPC Builder

Every HAL-based Nios II program consists of two Nios II projects, as shown in Figure 6–1. Your application-specific code is contained in one project (the user application project), and it depends on a separate BSP project (the HAL BSP).

The application project contains all the code you develop. The executable image for your program ultimately results from building both projects.

In the Nios II IDE flow, the Nios II IDE creates the HAL BSP (system library) project when you create your application project. In the Nios II software build tools, you create the BSP using **nios2-create-bsp** or a related tool.

The HAL BSP project contains all information needed to interface your program to the hardware. The HAL drivers relevant to your SOPC Builder system are built into the BSP project.

The BSP project depends on the SOPC Builder system, defined by an SOPC Builder system file (**.sopc** or **.ptf**). Both build flows can automatically keep your BSP up to date with the SOPC Builder system. This project dependency structure isolates your program from changes to the underlying hardware, and you can develop and debug code without concern about whether your program matches the target hardware.

In an IDE-managed project, the Nios II IDE manages the HAL BSP (system library) and updates the driver configurations to accurately reflect the system hardware. If the SOPC Builder system changes — i.e., the SOPC Builder system file (**.ptf**) is updated — the IDE rebuilds the HAL system library the next time you build or run your application program.

When you rebuild a user-managed project, the Nios II software build tools can automatically update your BSP to match the hardware. You control whether and when you allow these updates to take place.

For details about how the software build tools keep your BSP up to date with your hardware system, refer to the *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

In summary, when your program is based on a HAL BSP, you can always keep it synchronized with the target hardware by simply rebuilding your software.

The system.h System Description File

The **system.h** file provides a complete software description of the Nios II system hardware. Not all information in **system.h** is useful to you as a programmer, and it is rarely necessary to include it explicitly in your C source files. Nonetheless, **system.h** holds the answer to the fundamental question, "What hardware is present in this system?"

The **system.h** file describes each peripheral in the system and provides the following details:

- The hardware configuration of the peripheral
- The base address
- The IRQ priority (if any)
- A symbolic name for the peripheral

Both Nios II design flows generate the **system.h** file for HAL BSP projects. The contents of **system.h** depend on both the hardware configuration and the HAL BSP properties.

☞ Do not edit **system.h**. Both design flows provide tools to manipulate system settings.

For details of how to control BSP settings, see "HAL BSP Settings" on page 6–2.

The code in Example 6–1 from a **system.h** file shows some of the hardware configuration options it defines.

Example 6–1. Excerpts from a system.h File

```
/*
 * sys_clk_timer configuration
 *
 */

#define SYS_CLK_TIMER_NAME "/dev/sys_clk_timer"
#define SYS_CLK_TIMER_TYPE "altera_avalon_timer"
#define SYS_CLK_TIMER_BASE 0x00920800
#define SYS_CLK_TIMER_IRQ 0
#define SYS_CLK_TIMER_ALWAYS_RUN 0
#define SYS_CLK_TIMER_FIXED_PERIOD 0

/*
 * jtag_uart configuration
 *
 */

#define JTAG_UART_NAME "/dev/jtag_uart"
#define JTAG_UART_TYPE "altera_avalon_jtag_uart"
#define JTAG_UART_BASE 0x00920820
#define JTAG_UART_IRQ 1
```

Data Widths and the HAL Type Definitions

For embedded processors such as the Nios II processor, it is often important to know the exact width and precision of data. Because the ANSI C data types do not explicitly define data width, the HAL uses a set of standard type definitions instead. The ANSI C types are supported, but their data widths are dependent on the compiler's convention.

The header file **alt_types.h** defines the HAL type definitions; Table 6–1 shows the HAL type definitions.

Table 6–1. The HAL Type Definitions

Type	Meaning
`alt_8`	Signed 8-bit integer.
`alt_u8`	Unsigned 8-bit integer.

Table 6–1. The HAL Type Definitions

Type	Meaning
`alt_16`	Signed 16-bit integer.
`alt_u16`	Unsigned 16-bit integer.
`alt_32`	Signed 32-bit integer.
`alt_u32`	Unsigned 32-bit integer.
`alt_64`	Signed 64-bit integer.
`alt_u64`	Unsigned 64-bit integer.

Table 6–2 shows the data widths that the Altera-provided GNU toolchain uses.

Table 6–2. GNU Toolchain Data Widths

Type	Meaning
`char`	8 bits.
`short`	16 bits.
`long`	32 bits.
`int`	32 bits.

UNIX-Style Interface

The HAL API provides a number of UNIX-style functions. The UNIX-style functions provide a familiar development environment for new Nios II programmers, and can ease the task of porting existing code to run under the HAL environment. The HAL primarily uses these functions to provide the system interface for the ANSI C standard library. For example, the functions perform device access required by the C library functions defined in **stdio.h**.

The following list is the complete list of the available UNIX-style functions:

- `_exit()`
- `close()`
- `fstat()`
- `getpid()`
- `gettimeofday()`
- `ioctl()`
- `isatty()`
- `kill()`
- `lseek()`

- `open()`
- `read()`
- `sbrk()`
- `settimeofday()`
- `stat()`
- `usleep()`
- `wait()`
- `write()`

The most commonly used functions are those that relate to file I/O. See "File System" on page 6–7.

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

File System

The HAL provides infrastructure for UNIX-style file access. You can use this infrastructure to build a file system on any storage devices available in your hardware.

For an example, see the *Read-Only Zip File System* chapter of the *Nios II Software Developer's Handbook*.

You can access files within a HAL-based file system by using either the C standard library file I/O functions in the newlib C library (for example `fopen()`, `fclose()`, and `fread()`), or using the UNIX-style file I/O provided by the HAL.

The HAL provides the following UNIX style functions for file manipulation:

- `close()`
- `fstat()`
- `ioctl()`
- `isatty()`
- `lseek()`
- `open()`
- `read()`
- `stat()`
- `write()`

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

The HAL registers a file subsystem as a mount point within the global HAL file system. Attempts to access files below that mount point are directed to the file subsystem. For example, if a read-only zip file subsystem (**zipfs**) is mounted as **/mount/zipfs0**, the **zipfs** file subsystem handles calls to `fopen()` for **/mount/zipfs0/myfile**.

There is no concept of a current directory. Software must access all files using absolute paths.

The HAL file infrastructure also allows you to manipulate character mode devices via UNIX-style path names. The HAL registers character mode devices as nodes within the HAL file system. By convention, **system.h** defines the name of a device node as the prefix **/dev/** plus the name assigned to the hardware component in SOPC builder. For example, a UART peripheral **uart1** in SOPC builder is **/dev/uart1** in **system.h**.

The code in Example 6–2 shows reading characters from a read-only zip file subsystem **rozipfs** that is registered as a node in the HAL file system. The standard header files `stdio.h`, `stddef.h`, and `stdlib.h` are installed with the HAL.

Example 6–2. Reading Characters from a File Subsystem

```
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>

#define BUF_SIZE (10)

int main(void)
{
  FILE* fp;
  char buffer[BUF_SIZE];

  fp = fopen ("/mount/rozipfs/test", "r");
  if (fp == NULL)
  {
    printf ("Cannot open file.\n");
    exit (1);
  }
  fread (buffer, BUF_SIZE, 1, fp);

  fclose (fp);

  return 0;
}
```

For more information on the use of these functions, refer to the newlib C library documentation installed with the Nios II Embedded Design Suite (EDS). On the Windows **Start** menu, click **Programs**, **Altera**, **Nios II** *<version>*, **Nios II Documentation**.

Using Character-Mode Devices

A character-mode device is a hardware peripheral that sends and/or receives characters serially. A common example is the universal asynchronous receiver/transmitter (UART). Character mode devices are registered as nodes within the HAL file system. In general, a program associates a file descriptor to a device's name, and then writes and reads characters to or from the file using the ANSI C file operations defined in **file.h**. The HAL also supports the concept of standard input, standard output, and standard error, allowing programs to call the **stdio.h** I/O functions.

Standard Input, Standard Output and Standard Error

Using standard input (`stdin`), standard output (`stdout`), and standard error (`stderr`) is the easiest way to implement simple console I/O. The HAL manages `stdin`, `stdout`, and `stderr` behind the scenes, which allows you to send and receive characters through these channels without explicitly managing file descriptors. For example, the HAL directs the output of `printf()` to standard out, and `perror()` to standard error. You associate each channel to a specific hardware device by manipulating BSP settings.

The code in Example 6–3 on page 6–10 shows the classic Hello World program. This program sends characters to whatever device is associated with `stdout` when compiled in Nios II IDE.

Example 6–3. Hello World

```
#include <stdio.h>
int main ()
{
  printf ("Hello world!");
  return 0;
}
```

When using the UNIX-style API, you can use the file descriptors `stdin`, `stdout`, and `stderr`, defined in **unistd.h**, to access, respectively, the standard in, standard out, and standard error character I/O streams. **unistd.h** is installed with the Nios II EDS as part of the newlib C library package.

General Access to Character Mode Devices

Accessing a character-mode device (besides `stdin`, `stdout`, or `stderr`) is as easy as opening and writing to a file. The code in Example 6–4 demonstrates writing a message to a UART called `uart1`.

Example 6–4. Writing Characters to a UART

```
#include <stdio.h>
#include <string.h>

int main (void)
{
  char* msg = "hello world";
  FILE* fp;

  fp = fopen ("/dev/uart1", "w");
  if (fp!=NULL)
  {
    fprintf(fp, "%s",msg);
    fclose (fp);
  }
  return 0;
}
```

C++ Streams

HAL-based systems can use the C++ streams API for manipulating files from C++.

/dev/null

All systems include the device **/dev/null**. Writing to **/dev/null** has no effect, and all data is discarded. **/dev/null** is used for safe I/O redirection during system startup. This device could also be useful for applications that wish to sink unwanted data.

This device is purely a software construct. It does not relate to any physical hardware device within the system.

Lightweight Character-Mode I/O

The HAL offers several methods of reducing the code footprint of character-mode device drivers. For details, see "Reducing Code Footprint" on page 6–29.

Using File Subsystems

The HAL generic device model for file subsystems allows access to data stored in an associated storage device using the C standard library file I/O functions. For example the Altera zip read-only file system provides read-only access to a file system stored in flash memory.

A file subsystem is responsible for managing all file I/O access beneath a given mount point. For example, if a file subsystem is registered with the mount point **/mnt/rozipfs**, all file access beneath this directory, such as `fopen("/mnt/rozipfs/myfile", "r")`, is directed to that file subsystem.

As with character mode devices, you can manipulate files within a file subsystem using the C file I/O functions defined in **file.h**, such as `fopen()` and `fread()`.

For more information on the use of these functions, refer to the newlib C library documentation installed with the Nios II EDS. On the Windows Start menu, click Programs, Altera, Nios II <*version*>, **Nios II Documentation**.

Using Timer Devices

Timer devices are hardware peripherals that count clock ticks and can generate periodic interrupt requests. You can use a timer device to provide a number of time-related facilities, such as the HAL system clock, alarms, the time-of-day, and time measurement. To use the timer facilities, the Nios II processor system must include a timer peripheral in hardware.

The HAL API provides two types of timer device drivers:

- System clock driver. This type of driver supports alarms, such as you would use in a scheduler.

- Timestamp driver. This driver supports high-resolution time measurement.

An individual timer peripheral can behave as either a system clock or a timestamp, but not both.

The HAL-specific API functions for accessing timer devices are defined in **sys/alt_alarm.h** and **sys/alt_timestamp.h**.

System Clock Driver

The HAL system clock driver provides a periodic "heartbeat", causing the system clock to increment on each beat. Software can use the system clock facilities to execute functions at specified times, and to obtain timing information. You select a specific hardware timer peripheral as the system clock device by manipulating BSP settings.

For details of how to control BSP settings, see "HAL BSP Settings" on page 6–2.

The HAL provides implementations of the following standard UNIX functions: `gettimeofday()`, `settimeofday()`, and `times()`. The times returned by these functions are based on the HAL system clock.

The system clock measures time in units of "ticks". For embedded engineers who deal with both hardware and software, do not confuse the HAL system clock with the clock signal driving the Nios II processor hardware. The period of a HAL system clock tick is generally much longer than the hardware system clock. **system.h** defines the clock tick frequency.

At runtime, you can obtain the current value of the system clock by calling the `alt_nticks()` function. This function returns the elapsed time in system clock ticks since reset. You can get the system clock rate, in ticks per second, by calling the function `alt_ticks_per_second()`. The HAL timer driver initializes the tick frequency when it creates the instance of the system clock.

The standard UNIX function `gettimeofday()` is available to obtain the current time. You must first calibrate the time of day by calling `settimeofday()`. In addition, you can use the `times()` function to obtain information on the number of elapsed ticks. The prototypes for these functions appear in **times.h**.

For more information on the use of these functions, refer to the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

Alarms

You can register functions to be executed at a specified time using the HAL alarm facility. A software program registers an alarm by calling the function alt_alarm_start():

```
int alt_alarm_start (alt_alarm* alarm,
                     alt_u32    nticks,
                     alt_u32    (*callback) (void* context),
                     void*      context);
```

The function callback() is called after nticks have elapsed. The input argument context is passed as the input argument to callback() when the call occurs. The HAL does not use the context parameter. It is only used as a parameter to the callback() function.

Your code must allocate the alt_alarm structure, pointed to by the input argument alarm. This data structure must have a lifetime that is at least as long as that of the alarm. The best way to allocate this structure is to declare it as a static or global.alt_alarm_start() initializes *alarm.

The callback function can reset the alarm. The return value of the registered callback function is the number of ticks until the next call to callback. A return value of zero indicates that the alarm should be stopped. You can manually cancel an alarm by calling alt_alarm_stop().

One alarm is created for each call to alt_alarm_start(). Multiple alarms can be running simultaneously

Alarm callback functions execute in an interrupt context. This imposes functional restrictions which you must observe when writing an alarm callback.

For more information on the use of these functions, refer to the *Exception Handling* chapter of the *Nios II Software Developer's Handbook*.

The code fragment in Example 6–5 demonstrates registering an alarm for a periodic callback every second.

Example 6–5. Using a Periodic Alarm Callback Function

```
#include <stddef.h>
#include <stdio.h>
#include "sys/alt_alarm.h"
#include "alt_types.h"
```

```
/*
 * The callback function.
 */

alt_u32 my_alarm_callback (void* context)
{
  /* This function will be called once/second */
  return alt_ticks_per_second();
}

...

/* The alt_alarm must persist for the duration of the alarm. */
static alt_alarm alarm;

...

  if (alt_alarm_start (&alarm,
                       alt_ticks_per_second(),
                       my_alarm_callback,
                       NULL) < 0)
  {
    printf ("No system clock available\n");
  }
```

Timestamp Driver

Sometimes you want to measure time intervals with a degree of accuracy greater than that provided by HAL system clock ticks. The HAL provides high resolution timing functions using a timestamp driver. A timestamp driver provides a monotonically increasing counter that you can sample to obtain timing information. The HAL only supports one timestamp driver in the system.

You specify a hardware timer peripheral as the timestamp device by manipulating BSP settings. The Altera-provided timestamp driver uses the timer that you specify.

If a timestamp driver is present, the following functions are available:

- `alt_timestamp_start()`
- `alt_timestamp()`

Calling `alt_timestamp_start()` starts the counter running. Subsequent calls to `alt_timestamp()` return the current value of the timestamp counter. Calling `alt_timestamp_start()` again resets the counter to zero. The behavior of the timestamp driver is undefined when the counter reaches ($2^{32} – 1$).

You can obtain the rate at which the timestamp counter increments by calling the function `alt_timestamp_freq()`. This rate is typically the hardware frequency that the Nios II processor system runs at—usually millions of cycles per second. The timestamp drivers are defined in the **alt_timestamp.h** header file.

For more information on the use of these functions, refer to the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

The code fragment in Example 6–6 shows how you can use the timestamp facility to measure code execution time.

Example 6–6. Using the Timestamp to Measure Code Execution Time

```
#include <stdio.h>
#include "sys/alt_timestamp.h"
#include "alt_types.h"

int main (void)
{
  alt_u32 time1;
  alt_u32 time2;
  alt_u32 time3;

  if (alt_timestamp_start() < 0)
  {
    printf ("No timestamp device available\n");
  }
  else
  {
    time1 = alt_timestamp();
    func1(); /* first function to monitor */
    time2 = alt_timestamp();
    func2(); /* second function to monitor */
    time3 = alt_timestamp();

    printf ("time in func1 = %u ticks\n",
            (unsigned int) (time2 – time1));
    printf ("time in func2 = %u ticks\n",
            (unsigned int) (time3 – time2));
    printf ("Number of ticks per second = %u\n",
            (unsigned int)alt_timestamp_freq());
  }
  return 0;
}
```

Using Flash Devices

The HAL provides a generic device model for nonvolatile flash memory devices. Flash memories use special programming protocols to store data. The HAL API provides functions to write data to flash. For example, you can use these functions to implement a flash-based file subsystem.

The HAL API also provides functions to read flash, although it is generally not necessary. For most flash devices, programs can treat the flash memory space as simple memory when reading, and do not need to call special HAL API functions. If the flash device has a special protocol for reading data, such as the Altera EPCS serial configuration device, you must use the HAL API to both read and write data.

This section describes the HAL API for the flash device model. The following two APIs provide a different level of access to the flash:

- Simple flash access—functions that write buffers into flash and read them back at the block level. In writing, if the buffer is less than a full block, these functions erase pre-existing flash data above and below the newly written data.
- Fine-grained flash access—functions that write buffers into flash and read them back at the buffer level. In writing, if the buffer is less than a full block, these functions preserve pre-existing flash data above and below the newly written data. This functionality is generally required for managing a file subsystem.

The API functions for accessing flash devices are defined in **sys/alt_flash.h**.

For more information on the use of these functions, refer to the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*. For details of the Common Flash Interface, including the organization of CFI erase regions and blocks, see the JEDEC web site at **www.jedec.org**. You can find the CFI standard by searching for document JESD68.

Simple Flash Access

This interface consists of the functions `alt_flash_open_dev()`, `alt_write_flash()`, `alt_read_flash()`, and `alt_flash_close_dev()`. The code "Using the Simple Flash API Functions" on page 6–17 shows the usage of all of these functions in one code example. You open a flash device by calling alt_flash_open_dev(), which returns a file handle to a flash device. This function takes a single argument that is the name of the flash device, as defined in **system.h**.

Once you have obtained a handle, you can use the alt_write_flash() function to write data to the flash device. The prototype is:

```
int alt_write_flash(alt_flash_fd* fd,
                 int          offset,
                 const void*   src_addr,
                 int           length )
```

A call to this function writes to the flash device identified by the handle fd. The driver writes the data starting at offset bytes from the base of the flash device. The data written comes from the address pointed to by src_addr, the amount of data written is length.

There is also an alt_read_flash() function to read data from the flash device. The prototype is:

```
int alt_read_flash( alt_flash_fd* fd,
                    int           offset,
                    void*         dest_addr,
                    int           length )
```

A call to alt_read_flash() reads from the flash device with the handle fd, offset bytes from the beginning of the flash device. The function writes the data to location pointed to by dest_addr, and the amount of data read is length. For most flash devices, you can access the contents as standard memory, making it unnecessary to use alt_read_flash().

The function alt_flash_close_dev() takes a file handle and closes the device. The prototype for this function is:

```
void alt_flash_close_dev(alt_flash_fd* fd )
```

The code in Example 6–7 shows the usage of simple flash API functions to access a flash device named **/dev/ext_flash**, as defined in **system.h**.

Example 6–7. Using the Simple Flash API Functions

```
#include <stdio.h>
#include <string.h>
#include "sys/alt_flash.h"
#define BUF_SIZE 1024

int main ()
{
  alt_flash_fd* fd;
  int           ret_code;
  char          source[BUF_SIZE];
  char          dest[BUF_SIZE];

  /* Initialize the source buffer to all 0xAA */
  memset(source, 0xAA, BUF_SIZE);

  fd = alt_flash_open_dev("/dev/ext_flash");
  if (fd!=NULL)
  {
    ret_code = alt_write_flash(fd, 0, source, BUF_SIZE);
    if (ret_code==0)
    {
      ret_code = alt_read_flash(fd, 0, dest, BUF_SIZE);
      if (ret_code==0)
      {
        /*
         * Success.
         * At this point, the flash is all 0xAA and we
         * should have read that all back into dest
         */
      }
    }
    alt_flash_close_dev(fd);
  }
  else
  {
    printf("Can’t open flash device\n");
  }
  return 0;
}
```

Block Erasure or Corruption

Generally, flash memory is divided into blocks. `alt_write_flash()` might need to erase the contents of a block before it can write data to it. In this case, it makes no attempt to preserve the existing contents of a block. This action can lead to unexpected data corruption (erasure), if you are performing writes that do not fall on block boundaries. If you wish to preserve existing flash memory contents, use the fine-grained flash functions. See "Fine-Grained Flash Access" on page 6–20.

Table 6–3. Example of Writing Flash and Causing Unexpected Data Corruption

Address	Block	Time t(0)	Time t(1)	Time t(2)	Time t(3)	Time t(4)
		Before First Write	First Write		Second Write	
			After Erasing Block(s)	After Writing Data 1	After Erasing Block(s)	After Writing Data 2
`0x0000`	1	??	FF	AA	AA	AA
`0x0400`	1	??	FF	AA	AA	AA
`0x0800`	1	??	FF	AA	AA	AA
`0x0C00`	1	??	FF	AA	AA	AA
`0x1000`	2	??	FF	AA	FF	FF *(1)*
`0x1400`	2	??	FF	FF	FF	BB
`0x1800`	2	??	FF	FF	FF	BB
`0x1C00`	2	??	FF	FF	FF	FF

Notes to Table 6–3:

(1) Unintentionally cleared to FF during erasure for second write.

Table 6–3 shows how you can cause unexpected data corruption by writing using the simple flash-access functions. Table 6–3 shows the example of an 8 Kbyte flash memory comprising two 4 Kbyte blocks. First write 5 Kbytes of all `0xAA` into flash memory at address `0x0000`, and then write 2 Kbytes of all `0xBB` to address `0x1400`. After the first write succeeds (at time t(2)), the flash memory contains 5 Kbyte of `0xAA`, and the rest is empty (i.e., `0xFF`). Then the second write begins, but before writing into the second block, the block is erased. At this point, t(3), the flash contains 4 Kbyte of `0xAA` and 4 Kbyte of `0xFF`. After the second write finishes, at time t(4), the 2 Kbyte of `0xFF` at address `0x1000` is corrupted.

Fine-Grained Flash Access

There are three additional functions that provide complete control over writing flash contents at the highest granularity: alt_get_flash_info(),alt_erase_flash_block(),and alt_write_flash_block().

By the nature of flash memory, you cannot erase a single address within a block. You must erase (i.e., set to all ones) an entire block at a time. Writing to flash memory can only change bits from 1 to 0; to change any bit from 0 to 1, you must erase the entire block along with it.

Therefore, to alter a specific location within a block while leaving the surrounding contents unchanged, you must read out the entire contents of the block to a buffer, alter the value(s) in the buffer, erase the flash block, and finally write the whole block-sized buffer back to flash memory. The fine-grained flash access functions automate this process at the flash block level.

alt_get_flash_info() gets the number of erase regions, the number of erase blocks within each region, and the size of each erase block. The prototype is:

```
int alt_get_flash_info( alt_flash_fd*  fd,
                        flash_region** info,
                        int*           number_of_regions)
```

If the call is successful, upon return the address pointed to by number_of_regions contains the number of erase regions in the flash memory, and *info points to an array of flash_region structures. This array is part of the file descriptor.

The flash_region structure is defined in **sys/alt_flash_types.h**, and the typedef is:

```
typedef struct flash_region
{
  int offset;   /* Offset of this region from start of the flash */
  int region_size;       /* Size of this erase region */
  int number_of_blocks;  /* Number of blocks in this region */
  int block_size;    /* Size of each block in this erase region */
}flash_region;
```

With the information obtained by calling alt_get_flash_info(),you are in a position to erase or program individual blocks of the flash.

alt_erase_flash() erases a single block in the flash memory. The prototype is:

```
int alt_erase_flash_block( alt_flash_fd* fd,
                           int           offset,
                           int          length)
```

The flash memory is identified by the handle `fd`. The block is identified as being `offset` bytes from the beginning of the flash memory, and the block size is passed in `length`.

`alt_write_flash_block()` writes to a single block in the flash memory. The prototype is:

```
int alt_write_flash_block( alt_flash_fd* fd,
                           int           block_offset,
                           int           data_offset,
                           const void    *data,
                           int           length)
```

This function writes to the flash memory identified by the handle `fd`. It writes to the block located `block_offset` bytes from the start of the flash. The function writes `length` bytes of data from the location pointed to by `data` to the location `data_offset` bytes from the start of the flash device.

☞ These program and erase functions do not perform address checking, and do not verify whether a write operation spans into the next block. You must pass in valid information about the blocks to program or erase.

The code in Example 6–8 on page 6–21 demonstrates the usage of the fine-grained flash access functions.

Example 6–8. Using the Fine-Grained Flash Access API Functions

```
#include <string.h>
#include "sys/alt_flash.h"
#include "stdtypes.h"
#include "system.h"
#define BUF_SIZE 100

int main (void)
{
  flash_region* regions;
  alt_flash_fd* fd;
  int           number_of_regions;
  int           ret_code;
  char          write_data[BUF_SIZE];

  /* Set write_data to all 0xa */
  memset(write_data, 0xA, BUF_SIZE);
  fd = alt_flash_open_dev(EXT_FLASH_NAME);

  if (fd)
  {
    ret_code = alt_get_flash_info(fd, &regions, &number_of_regions);

    if (number_of_regions && (regions->offset == 0))
    {
      /* Erase the first block */
      ret_code = alt_erase_flash_block(fd,
                                       regions->offset,
                                       regions->block_size);
      if (ret_code == 0)
      {
         /*
          * Write BUF_SIZE bytes from write_data 100 bytes into
          * the first block of the flash
          */
         ret_code = alt_write_flash_block (
                 fd,
                 regions->offset,
                 regions->offset+0x100,
                 write_data,
                 BUF_SIZE );
      }
    }
  }
  return 0;
}
```

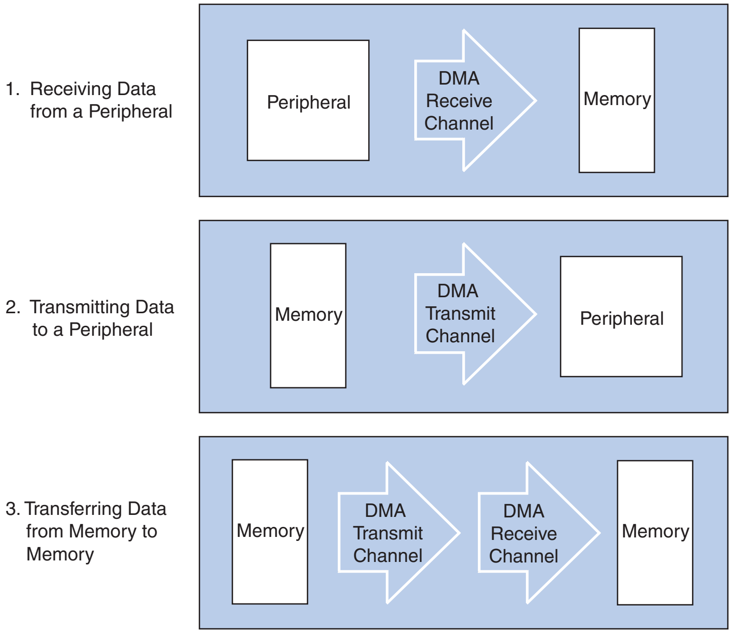
Using DMA Devices

The HAL provides a device abstraction model for direct memory access (DMA) devices. These are peripherals that perform bulk data transactions from a data source to a destination. Sources and destinations can be memory or another device, such as an Ethernet connection.

In the HAL DMA device model, DMA transactions fall into one of two categories: transmit or receive. As a result, the HAL provides two device drivers to implement transmit channels and receive channels. A transmit channel takes data in a source buffer and transmits it to a destination device. A receive channel receives data from a device and deposits it into a destination buffer. Depending on the implementation of the underlying hardware, software might have access to only one of these two endpoints.

Figure 6–2 shows the three basic types of DMA transactions. Copying data from memory to memory involves both receive and transmit DMA channels simultaneously.

Figure 6–2. Three Basic Types of DMA Transactions



The API for access to DMA devices is defined in **sys/alt_dma.h**.

For more information on the use of these functions, refer to the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

DMA devices operate on the contents of physical memory, therefore when reading and writing data you must consider cache interactions.

For more information on cache memory, refer to the *Cache and Tightly-Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.

DMA Transmit Channels

DMA transmit requests are queued up using a DMA transmit device handle. To obtained a handle, use the function `alt_dma_txchan_open()`. This function takes a single argument, the name of a device to use, as defined in **system.h**.

The code in Example 6–9 on page 6–24 shows how to obtain a handle for a DMA transmit device `dma_0`.

Example 6–9. Obtaining a File Handle for a DMA Device

```
#include <stddef.h>
#include "sys/alt_dma.h"

int main (void)
{
  alt_dma_txchan tx;

  tx = alt_dma_txchan_open ("/dev/dma_0");
  if (tx == NULL)
  {
    /* Error */
  }
  else
  {
    /* Success */
  }
  return 0;
}
```

You can use this handle to post a transmit request using `alt_dma_txchan_send()`. The prototype is:

```
typedef void (alt_txchan_done)(void* handle);

int alt_dma_txchan_send (alt_dma_txchan   dma,
                         const void*      from,
                         alt_u32          length,
                         alt_txchan_done* done,
                         void*            handle);
```

Calling `alt_dma_txchan_send()` posts a transmit request to channel `dma`. Argument `length` specifies the number of bytes of data to transmit, and argument `from` specifies the source address. The function returns before the full DMA transaction completes. The return value indicates whether the request is successfully queued. A negative return value indicates that the request failed. When the transaction completes, the user-supplied function `done` is called with argument `handle` to provide notification.

Two additional functions are provided for manipulating DMA transmit channels: `alt_dma_txchan_space()`, and `alt_dma_txchan_ioctl()`. The `alt_dma_txchan_space()` function returns the number of additional transmit requests that can be queued to the device. The `alt_dma_txchan_ioctl()` function performs device-specific manipulation of the transmit device.

☞ If you are using the Altera Avalon-MM DMA device to transmit to hardware (not memory-to-memory transfer), call the `alt_dma_txchan_ioctl()` function with the request argument set to `ALT_DMA_TX_ONLY_ON`.

For further information, refer to *"alt_dma_txchan_ioctl()"* in the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

DMA Receive Channels

DMA receive channels operate in a similar manner to DMA transmit channels. Software can obtain a handle for a DMA receive channel using the `alt_dma_rxchan_open()` function. You can then use the `alt_dma_rxchan_prepare()` function to post receive requests. The prototype for `alt_dma_rxchan_prepare()` is:

```
typedef void (alt_rxchan_done)(void* handle, void* data);

int alt_dma_rxchan_prepare (alt_dma_rxchan   dma,
                            void*            data,
                            alt_u32          length,
                            alt_rxchan_done* done,
                            void*            handle);
```

A call to this function posts a receive request to channel `dma`, for up to `length` bytes of data to be placed at address `data`. This function returns before the DMA transaction completes. The return value indicates whether the request is successfully queued. A negative return value indicates that the request failed. When the transaction completes, the user-supplied function `done()` is called with argument `handle` to provide notification and a pointer to the receive data.

Certain errors can prevent the DMA transfer from completing. Typically this is caused by a catastrophic hardware failure; for example, if a component involved in the transfer fails to respond to a read or write request. If the DMA transfer does not complete (i.e., less than `length` bytes are transferred), function `done()` is never called.

Two additional functions are provided for manipulating DMA receive channels: `alt_dma_rxchan_depth()` and `alt_dma_rxchan_ioctl()`.

☞ If you are using the Altera Avalon-MM DMA device to receive from hardware, (not memory-to-memory transfer), call the `alt_dma_rxchan_ioctl()` function with the request argument set to `ALT_DMA_RX_ONLY_ON`.

alt_dma_rxchan_depth() returns the maximum number of receive requests that can be queued to the device. alt_dma_rxchan_ioctl() performs device-specific manipulation of the receive device.

For further details, see the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

The code in Example 6–10 shows a complete example application that posts a DMA receive request, and blocks in main() until the transaction completes.

Example 6–10. A DMA Transaction on a Receive Channel

```
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include "sys/alt_dma.h"
#include "alt_types.h"

/* flag used to indicate the transaction is complete */
volatile int dma_complete = 0;

/* function that is called when the transaction completes */
void dma_done (void* handle, void* data)
{
  dma_complete = 1;
}

int main (void)
{
  alt_u8 buffer[1024];
  alt_dma_rxchan rx;

  /* Obtain a handle for the device */
  if ((rx = alt_dma_rxchan_open ("/dev/dma_0")) == NULL)
  {
    printf ("Error: failed to open device\n");
    exit (1);
  }
  else
  {
    /* Post the receive request */
    if (alt_dma_rxchan_prepare (rx, buffer, 1024, dma_done, NULL) < 0)
    {
      printf ("Error: failed to post receive request\n");
      exit (1);
    }

    /* Wait for the transaction to complete */
    while (!dma_complete);
    printf ("Transaction complete\n");
    alt_dma_rxchan_close (rx);
  }
  return 0;
}
```

Memory-to-Memory DMA Transactions

Copying data from one memory buffer to another buffer involves both receive and transmit DMA drivers. The code in Example 6–11 shows the process of queuing up a receive request followed by a transmit request to achieve a memory-to-memory DMA transaction.

Example 6–11. Copying Data from Memory to Memory

```
#include <stdio.h>
#include <stdlib.h>

#include "sys/alt_dma.h"
#include "system.h"

static volatile int rx_done = 0;

/*
 * Callback function that obtains notification that the data has
 * been received.
 */

static void done (void* handle, void* data)
{
  rx_done++;
}

/*
 *
 */

int main (int argc, char* argv[], char* envp[])
{
  int rc;

  alt_dma_txchan txchan;
  alt_dma_rxchan rxchan;

  void* tx_data = (void*) 0x901000;  /* pointer to data to send */
  void* rx_buffer = (void*) 0x902000;  /* pointer to rx buffer */

  /* Create the transmit channel */
  if ((txchan = alt_dma_txchan_open("/dev/dma_0")) == NULL)
  {
   printf ("Failed to open transmit channel\n");
   exit (1);
  }

  /* Create the receive channel */

  if ((rxchan = alt_dma_rxchan_open("/dev/dma_0")) == NULL)
  {
    printf ("Failed to open receive channel\n");
    exit (1);
  }

  /* Post the transmit request */

  if ((rc = alt_dma_txchan_send (txchan,
                                 tx_data,
                                 128,
                                 NULL,
                                 NULL)) < 0)
  {
   printf ("Failed to post transmit request, reason = %i\n", rc);
   exit (1);
  }

  /* Post the receive request */

  if ((rc = alt_dma_rxchan_prepare (rxchan,
                                    rx_buffer,
                                    128,
                                    done,
                                    NULL)) < 0)
  {
   printf ("Failed to post read request, reason = %i\n", rc);
   exit (1);
  }

  /* wait for transfer to complete */

  while (!rx_done);

  printf ("Transfer successful!\n");

  return 0;
}
```

Reducing Code Footprint

Code size is always of concern for system developers, because there is a cost associated with the memory device that stores code. The ability to control and reduce code size is important in controlling this cost.

The HAL environment is designed to include only those features that you request, minimizing the total code footprint. If your Nios II hardware system contains exactly the peripherals used by your program, the HAL contains only the drivers necessary to control the hardware, and nothing more.

The following sections describe options to consider when you need to further reduce code size. The **hello_world_small** example project demonstrates the use of some of these options to reduce code size to the absolute minimum.

Enable Compiler Optimizations

To enable compiler optimizations, use the `-O3` compiler optimization level for the `nios2-elf-gcc` compiler. You can specify this command-line option in the project properties; for details, refer to the Nios II IDE help system. Alternatively, you can specify the `-O3` option on the command line. With this option turned on, the Nios II IDE compiles code with the maximum optimization available, for both size and speed. You must set this option for both the BSP (system library) and the application project.

For details of how to control BSP settings, see "HAL BSP Settings" on page 6–2.

Use Reduced Device Drivers

Some devices provide two driver variants, a "fast" variant and a "small" variant. Which features are provided by these two variants is device specific. The "fast" variant is full-featured, while the "small" variant provides a reduced code footprint.

By default the HAL always uses the fast driver variants. You can choose the small footprint drivers by turning on the **Reduced device drivers** option for your HAL BSP (system library) in the Nios II IDE. Alternatively, on the command line, you can use the preprocessor option `-DALT_USE_SMALL_DRIVERS` when building the HAL BSP (system library).

In a user-managed software project, you can select the reduced device driver for an individual component.

For details of how to control BSP settings, see "HAL BSP Settings" on page 6–2.

Table 6–4 lists the Altera Nios II peripherals that currently provide small footprint drivers. The small footprint option might also affect other peripherals. Refer to each peripheral's data sheet for complete details of its driver's small footprint behavior.

Table 6–4. Altera Peripherals Offering Small Footprint Drivers

Peripheral	Small Footprint Behavior
UART	Polled operation, rather than IRQ-driven.
JTAG UART	Polled operation, rather than IRQ-driven.
Common flash interface controller	Driver is excluded in small footprint mode.
LCD module controller	Driver is excluded in small footprint mode
EPCS serial configuration device	Driver is excluded in small footprint mode

Reduce the File Descriptor Pool

The file descriptors that access character mode devices and files are allocated from a file descriptor pool. Software can control the size of this pool with the **Max file descriptors** system library property in the Nios II IDE. Alternatively, on the GNU command line, use the compile time constant `ALT_MAX_FD`. The default is 32.

For details of how to control BSP settings, see "HAL BSP Settings" on page 6–2.

Use /dev/null

At boot time, standard input, standard output and standard error are all directed towards the null device, i.e., **/dev/null**. This direction ensures that calls to `printf()` during driver initialization do nothing and therefore are harmless. Once all drivers have been installed, these streams are then redirected towards the channels configured in the HAL. The footprint of the code that performs this redirection is small, but you can eliminate it entirely by selecting `null` for `stdin`, `stdout`, and `stderr`. This selection assumes that you want to discard all data transmitted on standard out or standard error, and your program never receives input via `stdin`. You can control the assignment of `stdin`, `stdout`, and `stderr` channels by manipulating BSP settings.

For details of how to control BSP settings, see "HAL BSP Settings" on page 6–2.

Use a Smaller File I/O Library

Use the Small newlib C Library

The full newlib ANSI C standard library is often unnecessary for embedded systems. The GNU Compiler Collection (GCC) provides a reduced implementation of the newlib ANSI C standard library, omitting features of newlib that are often superfluous for embedded systems. The small newlib implementation requires a smaller code footprint. You can control the newlib implementation as a system library property in the Nios II IDE. When you use `nios2-elf-gcc` in command line mode, the `-msmallc` command-line option enables the small C library.

For details of how to control BSP settings, see "HAL BSP Settings" on page 6–2.

Table 6–5 summarizes the limitations of the Nios II small newlib C library implementation.

Table 6–5. Limitations of the Nios II Small newlib C Library (Part 1 of 2)

Limitation	Functions Affected

No floating-point support for `printf()` family of routines. The functions listed are implemented, but `%f` and `%g` options are not supported. *(1)*	`asprintf()`
	`fiprintf()`
	`fprintf()`
	`iprintf()`
	`printf()`
	`siprintf()`
	`snprintf()`
	`sprintf()`

No floating-point support for `vprintf()` family of routines. The functions listed are implemented, but `%f` and `%g` options are not supported.	`vasprintf()`
	`vfiprintf()`
	`vfprintf()`
	`vprintf()`
	`vsnprintf()`
	`vsprintf()`

No support for `scanf()` family of routines. The functions listed are not supported.	`fscanf()`
	`scanf()`
	`sscanf()`
	`vfscanf()`
	`vscanf()`
	`vsscanf()`

No support for seeking. The functions listed are not supported.	`fseek()`
	`ftell()`

Table 6–5. Limitations of the Nios II Small newlib C Library (Part 2 of 2)

Limitation	Functions Affected

No support for opening/closing `FILE *`. Only pre-opened `stdout`, `stderr`, and `stdin` are available. The functions listed are not supported.	`fopen()`
	`fclose()`
	`fdopen()`
	`fcloseall()`
	`fileno()`

No buffering of **stdio.h** output routines.	functions supported with no buffering:
	`fiprintf()`
	`fputc()`
	`fputs()`
	`perror()`
	`putc()`
	`putchar()`
	`puts()`
	`printf()`
	functions not supported:
	`setbuf()`
	`setvbuf()`

No **stdio.h** input routines. The functions listed are not supported.	`fgetc()`
	`gets()`
	`fscanf()`
	`getc()`
	`getchar()`
	`gets()`
	`getw()`
	`scanf()`

No support for locale.	`setlocale()`
	`localeconv()`

No support for C++, because the above functions are not supported.	

Notes to Table 6–5:

(1) These functions are a Nios II extension. GCC does not implement them in the small newlib C library.

The small newlib C library does not support MicroC/OS II.

For details of the GCC small newlib C library, refer to the newlib documentation installed with the Nios II EDS. On the Windows **Start** menu, click **Programs**, **Altera**, **Nios II** <*version*>, **Nios II Documentation**.

The Nios II implementation of the small newlib C library differs somewhat from GCC. Table 6–5 provides details of the differences.

Use UNIX-Style File I/O

If you need to reduce the code footprint further, you can omit the newlib C library, and use the UNIX-style API. See "UNIX-Style Interface" on page 6–6.

The Nios II EDS provides ANSI C file I/O, in the newlib C library, because there is a per-access performance overhead associated with accessing devices and files using the UNIX-style file I/O functions. The ANSI C file I/O provides buffered access, thereby reducing the total number of hardware I/O accesses performed. Also the ANSI C API is more flexible and therefore easier to use. However, these benefits are gained at the expense of code footprint.

Emulate ANSI C Functions

If you choose to omit the full implementation of newlib, but you need a limited number of ANSI-style functions, you can implement them easily using UNIX-style functions. The code in Example 6–12 shows a simple, unbuffered implementation of `getchar()`.

Example 6–12. Unbuffered getchar()

```
/* getchar: unbuffered single character input */
int getchar ( void )
{
  char c;
  return ( read ( 0, &c, 1 ) == 1 ) ? ( unsigned char ) c : EOF;
}
```

This example is from *The C Programming Language, Second Edition*, by Brian W. Kernighan and Dennis M. Ritchie. This standard textbook contains many other useful functions.

Use the Lightweight Device Driver API

The lightweight device driver API allows you to minimize the overhead of accessing device drivers. It has no direct effect on the size of the drivers themselves, but lets you eliminate driver API features which you might not need, reducing the overall size of the HAL code.

The lightweight device driver API is available for character-mode devices. The following device drivers support the lightweight device driver API:

- JTAG UART
- UART
- Optrex 16207 LCD

For these devices, the lightweight device driver API conserves code space by eliminating the dynamic file descriptor table and replacing it with three static file descriptors, corresponding to `stdin`, `stdout` and `stderr`. Library functions related to opening, closing and manipulating file descriptors are unavailable, but all other library functionality is available. You can refer to `stdin`, `stdout` and `stderr` as you would to any other file descriptor. You can also refer to the following predefined file numbers:

```
#define STDIN 0
#define STDOUT 1
#define STDERR 2
```

This option is appropriate if your program has a limited need for file I/O. The Altera Host Based File System and the Altera Zip Read-only File System are not available with the reduced device driver API.

You can turn on the **Lightweight device driver API** system library property in the Nios II IDE.

For details of how to control BSP settings, see "HAL BSP Settings" on page 6–2.

Alternatively, on the command line, you can use the preprocessor option `-DALT_USE_DIRECT_DRIVERS` when building the HAL BSP. By default, the lightweight device driver API is disabled.

For further details about the lightweight device driver API, see the *Developing Device Drivers for the HAL* chapter of the *Nios II Software Developer's Handbook*.

Use the Minimal Character-Mode API

If you can limit your use of character-mode I/O to very simple features, you can reduce code footprint by using the minimal character-mode API. This API includes the following functions:

- `alt_printf()`
- `alt_putchar()`
- `alt_putstr()`
- `alt_getchar()`

These functions are appropriate if your program only needs to accept command strings and send simple text messages. Some of them are helpful only in conjunction with the lightweight device driver API, discussed in "Use the Lightweight Device Driver API" on page 6–33.

To use the minimal character-mode API, include the header file **sys/alt_stdio.h**.

The following sections outline the effects of the functions on code footprint.

alt_printf()

This function is similar to `printf()`, but supports only the `%c`, `%s`, `%x` and `%%` substitution strings. `alt_printf()` takes up substantially less code space than `printf()`, regardless whether you select the lightweight device driver API. `alt_printf()` occupies less than 1Kbyte with compiler optimization level `-O2`.

alt_putchar()

Equivalent to `putchar()`. In conjunction with the lightweight device driver API, this function further reduces code footprint. In the absence of the lightweight API, it calls `putchar()`.

alt_putstr()

Similar to `puts()`, except that it does not append a newline character to the string. In conjunction with the lightweight device driver API, this function further reduces code footprint. In the absence of the lightweight API, it calls `puts()`.

alt_getchar()

Equivalent to `getchar()`. In conjunction with the lightweight device driver API, this function further reduces code footprint. In the absence of the lightweight API, it calls `getchar()`.

For further details on the minimal character-mode functions, refer to the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

Eliminate Unused Device Drivers

If a hardware device is present in the system, by default the Nios II design flows assume the device needs drivers, and configure the HAL BSP accordingly. If the HAL can find an appropriate driver, it creates an instance of this driver. If your program never actually accesses the device, resources are being used unnecessarily to initialize the device driver.

If the hardware includes a device that your program never uses, consider removing the device from the hardware. This reduces both code footprint and FPGA resource usage.

However, there are cases when a device must be present, but runtime software does not require a driver. The most common example is flash memory. The user program might boot from flash, but not use it at runtime; thus, it does not need a flash driver.

In the Nios II IDE, you can prevent the HAL from including the flash driver by defining the `ALT_EXCLUDE_CFI_FLASH` preprocessor option in the properties for the BSP (system library) project. Alternatively, you can specify the `-DALT_EXCLUDE_CFI_FLASH` option to the preprocessor on the command line.

In a user-managed project, you can selectively omit any individual driver, select a specific driver version, or substitute your own driver.

For further information on controlling driver configurations, refer to the *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook.*

Another way to control the device driver initialization process is by using the free-standing environment. See "Boot Sequence and Entry Point" on page 6–37.

Eliminate Unneeded Exit Code

The HAL calls the `exit()` function at system shutdown to provide a clean exit from the program. `exit()` flushes all of the C library internal I/O buffers and calls any C++ functions registered with `atexit()`. In particular, `exit()` is called upon return from `main()`. Two HAL options allow you to minimize or eliminate this exit code.

Eliminate Clean Exit

To avoid the overhead associated with providing a clean exit, your program can use the function `_exit()` in place of `exit()`. This function does not require you to change source code. You can control exit behavior through the **Clean exit (flush buffers)** system library property in the Nios II IDE. Alternatively, on the command line, you can specify the preprocessor option `-Dexit=_exit`.

Eliminate All Exit Code

Many embedded systems never exit at all. In such cases, exit code is unnecessary.

You can configure the HAL to omit all exit code (`exit()` and `_exit()`) from the BSP by turning on **Program never exits** in the system library properties in the Nios II IDE. Alternatively, on the command line, you can use the preprocessor option `-DALT_NO_EXIT` when building the HAL BSP (system library).

☞ If you enable this option, make sure your `main()` function (or `alt_main()` function) does not return.

Turn off C++ Support

By default, the HAL provides support for C++ programs, including default constructors and destructors. You can omit this support code by turning off the **Support C++** system library property in the Nios II IDE. Alternatively, on the command line, you can use the preprocessor option `-DALT_NO_C_PLUS_PLUS` when building the HAL BSP (system library).

Boot Sequence and Entry Point

Normally, your program's entry point is the function `main()`. There is an alternate entry point, `alt_main()`, that you can use to gain greater control of the boot sequence. The difference between entering at `main()` and entering at `alt_main()` is the difference between hosted and free-standing applications.

Hosted vs. Free-Standing Applications

The ANSI C standard defines a hosted application as one that calls `main()` to begin execution. At the start of `main()`, a hosted application presumes the runtime environment and all system services are initialized and ready to use. This is true in the HAL environment. If you are new to Nios II programming, the HAL's hosted environment helps you come up to speed more easily, because you don't have to consider what devices exist in the system or how to initialize each one. The HAL initializes the whole system.

The ANSI C standard also provides for an alternate entry point that avoids automatic initialization, and assumes that the Nios II programmer manually initializes any needed hardware. The `alt_main()` function provides a free-standing environment, giving you complete control over the initialization of the system. The free-standing environment places upon the programmer the burden of manually initializing any system feature used in the program. For example, calls to `printf()` do not function correctly in the free-standing environment, unless `alt_main()` first instantiates a character-mode device driver, and redirects `stdout` to the device.

☞ Using the freestanding environment increases the complexity of writing Nios II programs, because you assume responsibility for initializing the system. If your main interest is to reduce code footprint, you should use the suggestions described in "Reducing Code Footprint" on page 6–29. It is easier to reduce the HAL BSP footprint by using BSP settings, than to use the freestanding mode.

The Nios II EDS provides examples of both free-standing and hosted programs.

For more information, refer to the Nios II IDE help system.

Boot Sequence for HAL-Based Programs

The HAL provides system initialization code in the C runtime library (crt0.S). This code performs the following boot sequence:

- Flushes the instruction and data cache
- Configures the stack pointer
- Configures the global pointer register
- Zero initializes the BSS region using the linker supplied symbols `__bss_start` and `__bss_end`. These are pointers to the beginning and the end of the BSS region
- If there is no boot loader present in the system, copies to RAM any linker section whose run address is in RAM, such as `.rwdata`, `.rodata`, and `.exceptions`. See "Global Pointer Register" on page 6–44.
- Calls `alt_main()`

The HAL provides a default implementation of the `alt_main()` function, which performs the following steps:

- Calls `ALT_OS_INIT()` to perform any necessary operating system specific initialization. For a system that does not include an OS scheduler, this macro has no effect.

- If you are using the HAL with an operating system, initializes the alt_fd_list_lock semaphore, which controls access to the HAL file systems.
- Initializes the interrupt controller, and enable interrupts.
- Calls the alt_sys_init() function, which initializes all device drivers and software components in the system. The Nios II design flow creates the file alt_sys_init.c for each HAL BSP.
- Redirects the C standard I/O channels (stdin, stdout, and stderr) to use the appropriate devices.
- Calls the C++ constructors, using the _do_ctors() function.
- Registers the C++ destructors to be called at system shutdown.
- Calls main().
- Calls exit(), passing the return code of main() as the input argument for exit().

alt_main.c, installed with the Nios II EDS, provides this default implementation. In an IDE-managed project, you can find it in *<Nios II EDS install path>*/**components/altera_hal/HAL/src**. For user-managed projects, the software build tools copy **alt_main.c** into your BSP directory.

Customizing the Boot Sequence

You can provide your own implementation of the start-up sequence by simply defining alt_main() in your Nios II project. This gives you complete control of the boot sequence, and gives you the power to selectively enable HAL services. If your application requires an alt_main() entry point, you can copy the default implementation as a starting point and customize it to your needs.

Function alt_main() calls function main(). After main() returns, the default alt_main() enters an infinite loop. Alternatively, your custom alt_main() might terminate by calling exit(). Do not use a return statement.

The prototype for alt_main() is:

```
void alt_main (void)
```

The HAL build environment includes mechanisms to override default HAL BSP code. This lets you override boot loaders, as well as default device drivers and other system code, with your own implementation.

In the IDE-managed build flow, all source and header files are located using a search path. The build system always searches the BSP (system library) project’s paths first. You can override any HAL source file, including **alt_sys_init.c**, by placing your own implementation in your system project directory. Your custom file is used in place of the auto-generated version.

In the user-managed build flow, **alt_sys_init.c** is a generated file, which you should not modify. However, the Nios II software build tools enable you to control the generated contents of **alt_sys_init.c**. To specify the initialization sequence in **alt_sys_init.c**, you manipulate the `auto_initialize` and `alt_sys_init_priority` properties of each driver, using the **set_sw_property** Tcl command.

For more information about generated files in user-managed projects, and how to control the contents of **alt_sys_init.c**, refer to the *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*. For general information about **alt_sys_init.c**, refer to the *Developing Device Drivers for the HAL* chapter of the *Nios II Software Developer's Handbook*. For details about the **set_sw_property** Tcl command, refer to the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Memory Usage

This section describes the way that the HAL uses memory and how the HAL arranges code, data, stack, and other logical memory sections, in physical memory.

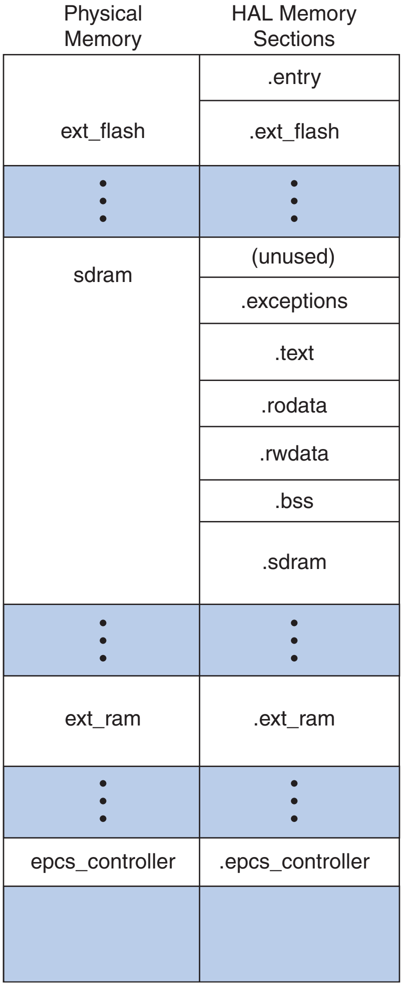
Memory Sections

By default, HAL-based systems are linked using an automatically-generated linker script that is created and managed by the Nios II IDE. This linker script controls the mapping of code and data within the available memory sections. The auto-generated linker script creates standard code and data sections (.text, .rodata, .rwdata, and .bss), plus a section for each physical memory device in the system. For example, if there is a memory component named `sdram` defined in the **system.h** file, there is a memory section named `.sdram`. Figure 6–3 on page 6–41 shows the organization of a typical HAL link map.

The memory devices that contain the Nios II processor's reset and exception addresses are a special case. The Nios II tools construct the 32-byte `.entry` section starting at the reset address. This section is reserved exclusively for the use of the reset handler. Similarly, the tools construct a `.exceptions` section, starting at the exception address.

In a memory device containing the reset or exception address, the linker creates a normal (non-reserved) memory section above the `.entry` or `.exceptions` section. If there is a region of memory below the `.entry` or `.exceptions` section, it is unavailable to the Nios II software. Figure 6–3 on page 6–41 illustrates an unavailable memory region below the `.exceptions` section.

Figure 6–3. Sample HAL Link Map



Assigning Code and Data to Memory Partitions

This section describes how to control the placement of program code and data in specific memory sections. In general, the Nios II design flow automatically specifies a sensible default partitioning. However, you might wish to change the partitioning in special situations.

For example, to enhance performance, it is a common technique to place performance-critical code and data in RAM with fast access time. It is also common during the debug phase to reset (i.e., boot) the processor from a location in RAM, but then boot from flash memory in the released version of the software. In these cases, you have to specify manually which code belongs in which section.

Simple Placement Options

The reset handler code is always placed at the base of the `.reset` partition. The exception handler code is always the first code within the section that contains the exception address. By default, the remaining code and data are divided into the following output sections:

- `.text`—all remaining code
- `.rodata`—the read-only data
- `.rwdata`—read-write data,
- `.bss`—zero-initialized data

You can control the placement of `.text`, `.rodata`, `.rwdata`, and all other memory partitions by manipulating BSP settings.

For details of how to control BSP settings, see "HAL BSP Settings" on page 6–2.

For more information, in the Nios II IDE help system, search for the **"System Library Properties"** topic.

Advanced Placement Options

Within your program source code, you can specify a target memory section for each piece of code. In C or C++, you can use the `section` attribute. This attribute must be placed in a function prototype; you cannot place it in the function declaration itself. The code in Example 6–13 shows placing a variable `foo` within the memory named `ext_ram`, and the function `bar()` in the memory named `sdram`.

Example 6–13. Manually Assigning C Code to a Specific Memory Section

```
/* data should be initialized when using the section attribute */
int foo __attribute__ ((section (".ext_ram.rwdata"))) = 0;

void bar (void) __attribute__ ((section (".sdram.txt")));

void bar (void)
{
  foo++;
}
```

In assembly you do this using the `.section` directive. For example, all code after the following line is placed in the memory device named `ext_ram`:

```
.section .ext_ram.txt
```

☞ The section names `ext_ram` and `sdram` are examples. You need to use section names corresponding to your hardware. When creating section names, use the following extensions:

- `.txt` for code: for example, `.sdram.txt`
- `.rodata` for read-only data: for example, `.cfi_flash.rodata`
- `.rwdata` for read-write data: for example, `.ext_ram.rwdata`

For details of the usage of these features, refer to the GNU compiler and assembler documentation. This documentation is installed with the Nios II EDS. To find it, open the **Nios II Literature** page, scroll down to **Software Development,** and click **Using the GNU Compiler Collection (GCC).**

Placement of the Heap and Stack

By default, the heap and stack are placed in the same memory partition as the `.rwdata` section. The stack grows downwards (toward lower addresses) from the end of the section. The heap grows upwards from the last used memory within the `.rwdata` section. You can control the placement of the heap and stack by manipulating BSP settings.

By default, the HAL performs no stack or heap checking. This makes function calls and memory allocation faster, but it means that `malloc()` (in C) and `new` (in C++) are unable to detect heap exhaustion. You can enable run-time stack checking by manipulating BSP settings. With stack checking on, `malloc()` and `new()` can detect heap exhaustion.

To specify the heap size limit, set the preprocessor symbol `ALT_MAX_HEAP_BYTES` to the maximum heap size in decimal. For example, the preprocessor argument `-DALT_MAX_HEAP_SIZE=1048576` sets the heap size limit to 0x100000. You can specify this command-line option in the system library properties; for details, refer to the Nios II IDE help system. Alternatively, you can specify the option on the command line.

Stack checking has performance costs. If you choose to leave stack checking turned off, you must code your program so as to ensure that it operates within the limits of available heap and stack memory.

See the Nios II IDE help system for details of selecting stack and heap placement, and setting up stack checking.

For details of how to control BSP settings, see "HAL BSP Settings" on page 6–2.

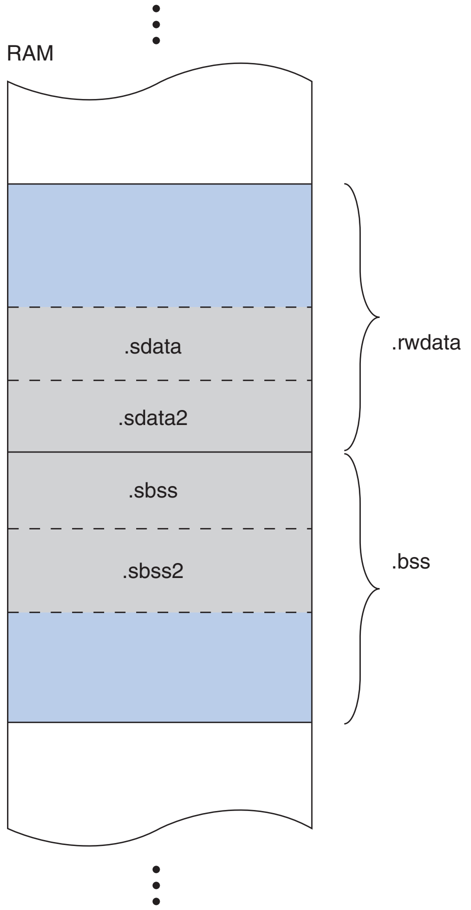
Global Pointer Register

The global pointer register enables fast access to global data structures in Nios II programs. The Nios II compiler implements the global pointer, and determines which data structures to access with it. You do not need to do anything unless you want to change the default compiler behavior.

The global pointer register can access a single contiguous region of 64K bytes. To avoid overflowing this region, the compiler only uses the global pointer with small global data structures. A data structure is considered "small" if its size is less than a specified threshold. By default, this threshold is eight bytes.

The "small" data structures are allocated to the small global data sections, `.sdata`, `.sdata2`, `.sbss`, and `.sbss2`. The small global data sections are subsections of the `.rwdata` and `.bss` sections. They are located together, as shown in Figure 6–4 on page 6–45, to enable the global pointer to access them.

Figure 6–4. Small Global Data sections



If the total size of the small global data structures happens to be more than 64K bytes, they overflow the global pointer region. The linker produces an error message saying `"Unable to reach` *<variable name>* `... from the global pointer ... because the offset ... is out of the allowed range, -32678 to 32767."`

You can fix this with the `-G` compiler option. This option sets the threshold size. For example, `-G 4` restricts global pointer usage to data structures four bytes long or smaller. Reducing the global pointer threshold reduces the size of the small global data sections.

The -G option's numeric argument is in decimal. You can specify this compiler option in the project properties; for details, refer to the Nios II IDE help system. Alternatively, you can specify the option on the command line. You must set this option to the same value for both the BSP and the application project.

Boot Modes

The processor's boot memory is the memory that contains the reset vector. This device might be an external flash or an Altera EPCS serial configuration device, or it might be an on-chip RAM. Regardless of the nature of the boot memory, HAL-based systems are constructed so that all program and data sections are initially stored within it. The HAL provides a small boot loader program which copies these sections to their run time locations at boot time. You can specify run time locations for program and data memory by manipulating BSP settings.

If the runtime location of the .text section is outside of the boot memory, the Altera flash programmer places a boot loader at the reset address, which is responsible for loading all program and data sections before the call to _start. When booting from an EPCS device, this loader function is provided by the hardware.

However, if the runtime location of the .text section is in the boot memory, the system does not need a separate loader. Instead the _reset entry point within the HAL executable is called directly. The function _reset initializes the instruction cache and then calls _start. This initialization sequence lets you develop applications that boot and execute directly from flash memory.

When running in this mode, the HAL executable must take responsibility for loading any sections that require loading to RAM. The .rwdata, .rodata, and .exceptions sections are loaded before the call to alt_main(), as required. This loading is performed by the function alt_load(). To load any additional sections, use the alt_load_section() function.

For more information, refer to *"alt_load_section()"* in the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

Paths to HAL Files

You might wish to view files in the HAL, especially header files, for reference. This section describes how to find HAL source files.

IDE-Managed Projects

In the IDE-managed build flow, HAL source files (and other BSP files) are referred to by path names. Do not edit HAL files in IDE-managed projects.

Finding HAL Files

HAL source files are in several directories because of the custom nature of Nios II systems. Each Nios II system can include different peripherals, and therefore the HAL BSP for each system is different. You can find HAL-related files in the following locations:

- The <*Nios II EDS install path*>/**components** directory contains most HAL source files.
- <*Nios II EDS install path*>/**components/altera_hal/HAL/inc/sys** contains header files defining the HAL generic device models. In a `#include` directive, reference these files relative to <*Nios II EDS install path*>/**components/altera_hal/HAL/inc/**. For example, to include the DMA drivers, use `#include sys/alt_dma.h`
- Each Nios II IDE system project directory contains the **system.h** file generated for that BSP (system library)**.**
- <*Nios II EDS install path*>/**bin** contains the newlib ANSI C library header files.
- The Altera design suite includes HAL drivers for SOPC Builder components distributed with the Quartus® II Complete Design Suite. For example, if the Altera design suite is installed in **c:\altera\72**, you can find the drivers under **c:\altera\72\ip\sopc_builder_ip**.

Overriding HAL Functions

To provide your own implementation of a HAL function, include the file in your Nios II IDE system project. When building the executable, Nios II IDE finds your function, and uses it in place of the HAL version.

User-Managed Projects

In the user-managed build flow, HAL source files (and other BSP files) are copied into the BSP directory. You are free to modify copied HAL source files.

Finding HAL Files

You determine the location of HAL source files when you create the BSP.

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

Overriding HAL Functions

HAL source files are copied into your BSP directory when you create your BSP. You can freely modify copied files, without losing your changes when you update your BSP.

For more information, refer to *"Generated and Copied Files"* in the *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook.*

Referenced Documents

This chapter references the following documents:

- *Overview* chapter of the *Nios II Software Developer's Handbook*
- *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*
- *Developing Device Drivers for the HAL* chapter of the *Nios II Software Developer's Handbook*
- *Exception Handling* 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*
- *Read-Only Zip File System* chapter of the *Nios II Software Developer's Handbook*
- *The C Programming Language, Second Edition*, by Brian Kernighan and Dennis M. Ritchie (Prentice-Hall)
- *GNU documentation* on the Nios II Literature page installed with the Nios II EDS.

Document Revision History

Table 6–6 shows the revision history for this document.

Table 6–6. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	● Added documentation for HAL program development with the Nios II software build tools. ● Additional documentation of alarms functions ● Correct `alt_erase_flash_block()` example	—
May 2007 v7.1.0	● Added table of contents to Introduction section. ● Added Referenced Documents section.	—
March 2007 v7.0.0	No change from previous release.	
November 2006 v6.1.0	● **Program never exits** system library option ● **Support C++** system library option ● **Lightweight device driver API** system library option ● Minimal character-mode API	
May 2006 v6.0.0	● Revised text on instruction emulation. ● Added section on global pointers.	
October 2005 v5.1.0	● Added `alt_64` and `alt_u64 types` to Table 6–1. ● Made changes to section "Placement of the Heap and Stack".	
May 2005 v5.0.0	Added `alt_load_section()` function information.	
December 2004 v1.2	● Added boot modes information. ● Amended compiler optimizations. ● Updated *Reducing Code Footprint* section.	
September 2004 v1.1	Corrected DMA receive channels example code.	
May 2004 v1.0	Initial Release.	

7. Developing Device Drivers for the Hardware Abstraction Layer

NII52005-7.2.0

Introduction

Embedded systems typically have application-specific hardware features that require custom device drivers. This chapter describes how to develop device drivers and integrate them with the hardware abstraction layer (HAL). This chapter contains the following sections:

Direct interaction with the hardware should be confined to device driver code. In general, most of your program code should be free of low-level access to the hardware. Wherever possible, use the high-level HAL application programming interface (API) functions to access hardware. This makes your code more consistent and more portable to other Nios® II systems that might have different hardware configurations.

When you create a new driver, you can integrate the driver into the HAL framework at one of the following two levels:

- Integration into the HAL API
- Peripheral-specific API

Integration into the HAL API

Integration into the HAL API is the preferred option for a peripheral that belongs to one of the HAL generic device model classes, such as character-mode or direct memory access (DMA) devices.

For descriptions of the HAL generic device model classes, refer to the *Overview of the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

For integration into the HAL API, you write device access functions as specified in this chapter, and the device becomes accessible to software via the standard HAL API. For example, if you have a new LCD screen device that displays ASCII characters, you write a character-mode device driver. With this driver in place, programs can call the familiar `printf()` function to stream characters to the LCD screen.

Peripheral-Specific API

If the peripheral does not belong to one of the HAL generic device model classes, you need to provide a device driver with an interface that is specific to the hardware implementation, and the API to the device is separate from the HAL API. Programs access the hardware by calling the functions you provide, not the HAL API.

The up-front effort to implement integration into the HAL API is higher, but you gain the benefit of the HAL and C standard library API to manipulate devices.

For details on integration into the HAL API, see "Integrating a Device Driver into the HAL" on page 7–18.

All the other sections in this chapter apply to integrating drivers into the HAL API and creating drivers with a peripheral-specific API.

☞ Although C++ is supported for programs based on the HAL, HAL drivers should not be written in C++. Restrict your driver code to either C or assembler, and preferably C for portability.

Before You Begin

This chapter assumes that you are familiar with C programming for the HAL. You should be familiar with the information in the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*, before reading this chapter.

☞ This document uses the variable <*Altera installation*> to represent the location where the Altera® design suite is installed. On a Windows system, by default, that location is **c:\altera\**<*nn*>, where <*nn*> represents the current version number.

Development Flow for Creating Device Drivers

The steps to develop a new driver for the HAL are very much dependent on your device details. However, the following generic steps apply to all device classes.

1. Create the device header file that describes the registers. This header file might be the only interface required.

2. Implement the driver functionality.

3. Test from `main()`.

4. Proceed to the final integration of the driver into the HAL environment.

5. Integrate the device driver into the HAL framework.

SOPC Builder Concepts

This section discusses concepts about the Altera® SOPC Builder hardware design tool that enhance your understanding of the driver development process. You need not use SOPC Builder to develop Nios II device drivers.

The Relationship between system.h and SOPC Builder

The **system.h** header file provides a complete software description of the Nios II system hardware, and is a fundamental part of developing drivers. Because drivers interact with hardware at the lowest level, it is worth mentioning the relationship between **system.h** and SOPC Builder that generates the Nios II processor system hardware. Hardware designers use SOPC Builder to specify the architecture of the Nios II processor system and integrate the necessary peripherals and memory. Therefore, the definitions in **system.h**, such as the name and configuration of each peripheral, are a direct reflection of design choices made in SOPC Builder.

For more information on the **system.h** header file, see the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*.

Using SOPC Builder for Optimal Hardware Configuration

If you find less-than-optimal definitions in **system.h**, remember that the contents of **system.h** can be modified by changing the underlying hardware with SOPC Builder. Before you write a device driver to accommodate imperfect hardware, it is worth considering whether the hardware can be improved easily with SOPC Builder.

Components, Devices and Peripherals

SOPC Builder uses the term "component" to describe hardware modules included in the system. In the context of Nios II software development, SOPC Builder components are devices, such as peripherals or memories. In the following sections, "component" is used interchangeably with "device" and "peripheral" when the context is closely related to SOPC Builder.

Accessing Hardware

Software accesses the hardware via macros that abstract the memory-mapped interface to the device. This section describes the macros that define the hardware interface for each device.

All SOPC Builder components provide a directory that defines the device hardware and software. For example, each component provided in the Quartus® II software has its own directory in the <*Altera installation*>**/ip/sopc_builder_ip** directory. Many components provide a header file that defines their hardware interface. The header file is <*component name*>**_regs.h** and is included in the **inc** subdirectory for the specific component. For example, the Altera-provided JTAG UART component defines its hardware interface in the file <*Altera installation*>**/ip/sopc_builder_ip/altera_avalon_jtag_uart/inc/altera_avalon_jtag_uart_regs.h**.

The **_regs.h** header file defines the following access:

- Register access macros that provide a read and/or write macro for each register within the component that supports the operation. The macros are:

  - **IORD**_<*component name*>_<*register name*> (*component base address*)
  - **IOWR**_<*component name*>_<*register name*> (*component base address*, *data*).

  For example, **altera_avalon_jtag_uart_regs.h** defines the following macros:

  - `IORD_ALTERA_AVALON_JTAG_UART_DATA()`
  - `IOWR_ALTERA_AVALON_JTAG_UART_DATA()`
  - `IORD_ALTERA_AVALON_JTAG_UART_CONTROL()`
  - `IOWR_ALTERA_AVALON_JTAG_UART_CONTROL()`

- Register address macros that return the physical address for each register within a component. The address register returned is the component's base address + the specified register offset value. These macros are named **IOADDR**_<*component name*>_<*register name*> (*component base address*).
  For example, **altera_avalon_jtag_uart_regs.h** defines the following macros:

  - `IOADDR_ALTERA_AVALON_JTAG_UART_DATA()`
  - `IOADDR_ALTERA_AVALON_JTAG_UART_CONTROL()`

Use these macros only as parameters to a function that requires the specific address of a data source or destination. For example, a routine that reads a stream of data from a particular source register in a component might require the physical address of the register as a parameter.

- Bit-field masks and offsets that provide access to individual bit-fields within a register. These macros have the following names:

  - *<component name>*_*<register name>*_*<name of field>*`_MSK` — a bit-mask of the field
  - *<component name>*_*<register name>*_*<name of field>*`_OFST` — the bit offset of the start of the field

  For example, `ALTERA_AVALON_UART_STATUS_PE_MSK` and `ALTERA_AVALON_UART_STATUS_PE_OFST` access the `pe` field of the status register.

Only use the macros defined in the **_regs.h** file to access a device's registers. You must use the register access functions to ensure that the processor bypasses the data cache when reading and or writing the device. Do not use hard-coded constants, because they make your software susceptible to changes in the underlying hardware.

If you are writing the driver for a completely new hardware device, you have to prepare the **_regs.h** header file.

For more information on the effects of cache management and device access, see the *Cache and Tightly-Coupled Memory* chapter of the *Nios II Software Developer's Handbook*. For a complete example of the **_regs.h** file, see the component directory for any of the Altera-supplied SOPC Builder components, such as *<Altera installation>***\ip\sopc_builder_ip\altera_avalon_jtag_uart\inc**.

Creating Drivers for HAL Device Classes

The HAL supports a number of generic device model classes, as defined in the *Overview of the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*. By writing a device driver as described in this section, you describe to the HAL an instance of a specific device that falls into one of its known device classes. This section defines a consistent interface for driver functions so that the HAL can access the driver functions uniformly.

The following sections define the API for the following classes of devices:

- Character-mode devices
- File subsystems
- DMA devices
- Timer devices used as system clock
- Timer devices used as timestamp clock
- Flash memory devices
- Ethernet devices

The following sections describe how to implement device drivers for each class of device, and how to register them for use within HAL-based systems.

Character-Mode Device Drivers

This section describes how to create a device instance and register a character device.

Create a Device Instance

For a device to be made available as a character mode device, it must provide an instance of the `alt_dev` structure. The following code defines the `alt_dev` structure:

```
typedef struct {
  alt_llist    llist;     /* for internal use */
  const char*  name;
  int (*open)  (alt_fd* fd, const char* name, int flags, int mode);
  int (*close) (alt_fd* fd);
  int (*read)  (alt_fd* fd, char* ptr, int len);
  int (*write) (alt_fd* fd, const char* ptr, int len);
  int (*lseek) (alt_fd* fd, int ptr, int dir);
  int (*fstat) (alt_fd* fd, struct stat* buf);
  int (*ioctl) (alt_fd* fd, int req, void* arg);
} alt_dev;
```

The `alt_dev` structure, defined in *<Nios II EDS install path>***/components/altera_hal/HAL/inc/sys/alt_dev.h**, is essentially a collection of function pointers. These functions are called in response to application accesses to the HAL file system. For example, if you call the function `open()` with a file name that corresponds to this device, the result is a call to the `open()` function provided in this structure.

For more information on `open()`, `close()`, `read()`, `write()`, `lseek()`, `fstat()`, and `ioctl()`, see the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

None of these functions directly modify the global error status, `errno`. Instead, the return value is the negation of the appropriate error code provided in **errno.h**.

For example, the `ioctl()` function returns `-ENOTTY` if it cannot handle a request rather than set `errno` to `ENOTTY` directly. The HAL system routines that call these functions ensure that `errno` is set accordingly.

The function prototypes for these functions differ from their application level counterparts in that they each take an input file descriptor argument of type `alt_fd*` rather than `int`.

A new `alt_fd` structure is created upon a call to `open()`. This structure instance is then passed as an input argument to all function calls made for the associated file descriptor.

The following code defines the `alt_fd` structure.

```
typedef struct
{
  alt_dev* dev;
  void* priv;
  int fd_flags;
} alt_fd;
```

where:

- `dev` is a pointer to the device structure for the device being used.
- `fd_flags` is the value of `flags` passed to `open()`.
- `priv` is a reserved, implementation-dependent argument, defined by the driver. If the driver requires any special, non-HAL-defined values to be maintained for each file or stream, you can store them in a data structure, and use `priv` maintains a pointer to the structure. The HAL ignores `priv`.

  Allocate storage for the data structure in your `open()` function (pointed to by the `alt_dev` structure). Free the storage in your `close()` function.

  ☞ To avoid memory leaks, make sure the `close()` function is called when the file or stream is no longer needed.

A driver is not required to provide all of the functions within the `alt_dev` structure. If a given function pointer is set to `NULL`, a default action is used instead. Table 7–1 shows the default actions for each of the available functions.

Table 7–1. Default Behavior for Functions Defined in alt_dev

Function	Default Behavior
`open`	Calls to `open()` for this device succeed, unless the device was previously locked by a call to `ioctl()` with `req = TIOCEXCL`.
`close`	Calls to `close()` for a valid file descriptor for this device always succeed.
`read`	Calls to `read()` for this device always fail.
`write`	Calls to `write()` for this device always fail.
`lseek`	Calls to `lseek()` for this device always fail.
`fstat`	The device identifies itself as a character mode device.
`ioctl`	`ioctl()` requests that cannot be handled without reference to the device fail.

In addition to the function pointers, the `alt_dev` structure contains two other fields: `llist` and `name`. `llist` is for internal use, and should always be set to the value `ALT_LLIST_ENTRY`. `name` is the location of the device within the HAL file system and is the name of the device as defined in **system.h**.

Register a Character Device

Having created an instance of the `alt_dev` structure, the device must be made available to the system by registering it with the HAL and by calling the following function:

```
int alt_dev_reg (alt_dev* dev)
```

This function takes a single input argument, which is the device structure to register. A return value of zero indicates success. A negative return value indicates that the device can not be registered.

Once a device is registered with the HAL file system, you can access it via the HAL API and the ANSI C standard library. The node name for the device is the name specified in the `alt_dev` structure.

For more information, refer to the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*.

File Subsystem Drivers

A file subsystem device driver is responsible for handling file accesses beneath a specified mount point within the global HAL file system.

Create a Device Instance

Creating and registering a file system is very similar to creating and registering a character-mode device. To make a file system available, create an instance of the `alt_dev` structure (see "Character-Mode Device Drivers" on page 7–6). The only distinction is that the `name` field of the device represents the mount point for the file subsystem. Of course, you must also provide any necessary functions to access the file subsystem, such as `read()` and `write()`, similar to the case of the character-mode device.

If you do not provide an implementation of `fstat()`, the default behavior returns the value for a character-mode device, which is incorrect behavior for a file subsystem.

Register a File Subsystem Device

You can register a file subsystem using the following function:

```
int alt_fs_reg  (alt_dev* dev)
```

This function takes a single input argument, which is the device structure to register. A negative return value indicates that the file system can not be registered.

Once a file subsystem is registered with the HAL file system, you can access it via the HAL API and the ANSI C standard library. The mount point for the file subsystem is the `name` specified in the `alt_dev` structure.

For more information, refer to the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*.

Timer Device Drivers

This section describes the system clock and timestamp drivers.

System Clock Driver

A system clock device model requires a driver to generate the periodic "tick". There can be only one system clock driver in a system. You implement a system clock driver as an interrupt service routine (ISR) for a timer peripheral that generates a periodic interrupt. The driver must provide periodic calls to the following function:

```
void alt_tick (void)
```

The expectation is that `alt_tick()` is called in interrupt context.

To register the presence of a system clock driver, call the following function:

```
int alt_sysclk_init (alt_u32 nticks)
```

The input argument `nticks` is the number of system clock ticks per second, which is determined by your system clock driver. The return value of this function is zero upon success, and non-zero otherwise.

For more information on writing interrupt service routines, see the *Exception Handling* chapter of the *Nios II Software Developer's Handbook*.

Timestamp Driver

A timestamp driver provides implementations for the three timestamp functions: `alt_timestamp_start()`, `alt_timestamp()`, and `alt_timestamp_freq()`. The system can only have one timestamp driver.

For more information on using these functions, see the *Developing Programs using the HAL* and *HAL API Reference* chapters of the *Nios II Software Developer's Handbook*.

Flash Device Drivers

This section describes how to create a flash driver and register a flash device.

Create a Flash Driver

Flash device drivers must provide an instance of the `alt_flash_dev` structure, defined in **sys/alt_flash_dev.h**. The following code shows the structure:

```
struct alt_flash_dev
{
  alt_llist                 llist; // internal use only
  const char*               name;
  alt_flash_open            open;
  alt_flash_close           close;
  alt_flash_write           write;
  alt_flash_read            read;
  alt_flash_get_flash_info  get_info;
  alt_flash_erase_block     erase_block;
  alt_flash_write_block     write_block;
  void*                     base_addr;
  int                       length;
  int                       number_of_regions;
  flash_region    region_info[ALT_MAX_NUMBER_OF_FLASH_REGIONS];
};
```

The first parameter `llist` is for internal use, and should always be set to the value ALT_LLIST_ENTRY. `name` is the location of the device within the HAL file system and is the name of the device as defined in **system.h**.

The seven fields `open` to `write_block` are function pointers that implement the functionality behind the application API calls to:

- `alt_flash_open_dev()`
- `alt_flash_close_dev()`
- `alt_write_flash()`
- `alt_read_flash()`
- `alt_get_flash_info()`
- `alt_erase_flash_block()`
- `alt_write_flash_block()`

where:

- the `base_addr` parameter is the base address of the flash memory
- `length` is the size of the flash in bytes
- `number_of_regions` is the number of erase regions in the flash
- `region_info` contains information about the location and size of the blocks in the flash device

For more information on the format of the `flash_region` structure, refer to the "Using Flash Devices" section of the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*.

Some flash devices such as common flash interface (CFI) compliant devices allow you to read out the number of regions and their configuration at run time. Otherwise, these two fields must be defined at compile time.

Register a Flash Device

After creating an instance of the alt_flash_dev structure, you must make the device available to the HAL system by calling the following function:

```
int alt_flash_device_register( alt_flash_fd* fd)
```

This function takes a single input argument, which is the device structure to register. A return value of zero indicates success. A negative return value indicates that the device could not be registered.

DMA Device Drivers

The HAL models a DMA transaction as being controlled by two endpoint devices: a receive channel and a transmit channel. This section describes the drivers for each type of DMA channel separately.

For a complete description of the HAL DMA device model, refer to the “Using DMA Devices” section of the *Developing Programs using the HAL* chapter of the *Nios II Software Developer’s Handbook*.

The DMA device driver interface is defined in **sys/alt_dma_dev.h**.

DMA Transmit Channel

A DMA transmit channel is constructed by creating an instance of the alt_dma_txchan structure:

```
typedef struct alt_dma_txchan_dev_s alt_dma_txchan_dev;
struct alt_dma_txchan_dev_s
{
  alt_llist   llist;
  const char* name;
  int         (*space) (alt_dma_txchan  dma);
  int         (*send) (alt_dma_txchan   dma,
                       const void*      from,
                       alt_u32          len,
                       alt_txchan_done* done,
                       void*            handle);
  int         (*ioctl) (alt_dma_txchan dma, int req, void* arg);
};
```

Table 7–2 shows the available fields and their functions.

Table 7–2. Fields in the alt_dma_txchan Structure

Field	Function
llist	This field is for internal use, and must always be set to the value ALT_LLIST_ENTRY.
name	The name that refers to this channel in calls to alt_dma_txchan_open(). name is the name of the device as defined in **system.h**.
space	A pointer to a function that returns the number of additional transmit requests that can be queued to the device. The input argument is a pointer to the alt_dma_txchan_dev structure.
send	A pointer to a function that is called as a result of a call to the application API function alt_dma_txchan_send().This function posts a transmit request to the DMA device. The parameters passed to alt_txchan_send() are passed directly to send(). For a description of parameters and return values, see alt_dma_txchan_send() in the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.
ioctl	This function provides device specific I/O control. See **sys/alt_dma_dev.h** for a list of the generic options that a device might wish to support.

Both the space and send functions need to be defined. If the ioctl field is set to null, calls to alt_dma_txchan_ioctl() return -ENOTTY for this device.

After creating an instance of the alt_dma_txchan structure, you must register the device with the HAL system to make it available by calling the following function:

```
int alt_dma_txchan_reg (alt_dma_txchan_dev* dev)
```

The input argument dev is the device to register. The return value is zero upon success, or negative if the device cannot be registered.

DMA Receive Channel

A DMA receive channel is constructed by creating an instance of the alt_dma_rxchan structure:

```
typedef alt_dma_rxchan_dev_s alt_dma_rxchan;
struct alt_dma_rxchan_dev_s
{
  alt_llist   list;
  const char* name;
  alt_u32     depth;
  int         (*prepare) (alt_dma_rxchan   dma,
                          void*            data,
                          alt_u32          len,
                          alt_rxchan_done* done,
                          void*            handle);
  int          (*ioctl) (alt_dma_rxchan dma, int req, void* arg);
};
```

Table 7–3 shows the available fields and their functions.

Table 7–3. Fields in the alt_dma_rxchan Structure

Field	Function
llist	This function is for internal use and should always be set to the value ALT_LLIST_ENTRY.
name	The name that refers to this channel in calls to alt_dma_rxchan_open(). name is the name of the device as defined in **system.h**.
depth	The total number of receive requests that can be outstanding at any given time.
prepare	A pointer to a function that is called as a result of a call to the application API function alt_dma_rxchan_prepare(). This function posts a receive request to the DMA device. The parameters passed to alt_dma_rxchan_prepare() are passed directly to prepare(). For a description of parameters and return values, see alt_dma_rxchan_prepare() in the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.
ioctl	This is a function that provides device specific I/O control. See **sys/alt_dma_dev.h** for a list of the generic options that a device might wish to support.

The prepare() function is required to be defined. If the ioctl field is set to null, calls to alt_dma_rxchan_ioctl() return -ENOTTY for this device.

After creating an instance of the alt_dma_rxchan structure, you must register the device driver with the HAL system to make it available by calling the following function:

```
int alt_dma_rxchan_reg (alt_dma_rxchan_dev* dev)
```

The input argument dev is the device to register. The return value is zero upon success, or negative if the device cannot be registered.

Ethernet Device Drivers

The HAL generic device model for Ethernet devices provides access to the NicheStack® TCP/IP Stack - Nios II Edition running on the MicroC/OS-II operating system. You can provide support for a new Ethernet device by supplying the driver functions that this section defines.

Before you consider writing a device driver for a new Ethernet device, you need a basic understanding of the Altera implementation of the NicheStack TCP/IP Stack and its usages.

For more information, refer to the *Ethernet and the NicheStack TCP/IP Stack - Nios II Edition* chapter of the *Nios II Software Developer's Handbook*.

The easiest way to write a new Ethernet device driver is to start with Altera's implementation for the SMSC lan91c111 device, and modify it to suit your Ethernet media access controller (MAC). This section assumes you take this approach. Starting from a known-working example makes it easier for you to learn the most important details of the NicheStack TCP/IP Stack implementation.

The source code for the lan91c111 driver is provided with the Quartus II software in <*Altera installation*>**/ip/sopc_builder_ip/altera_avalon_lan91c111/UCOSII**. For the sake of brevity, this section refers to this directory as <*SMSC path*>. The source files are in the <*SMSC path*>**/src/iniche** and <*SMSC path*>**/inc/iniche** directories.

A number of useful NicheStack TCP/IP Stack files are installed with the Nios II EDS, under 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*>.

For more information on the NicheStack TCP/IP Stack implementation, see the *NicheStack Technical Reference Manual*, available at **www.altera.com/literature/lit-nio2.jsp**.

You need not edit the NicheStack TCP/IP Stack source code to implement a NicheStack-compatible driver. Nevertheless, Altera provides the source code for your reference. The files are installed with the Nios II EDS in the <*iniche path*> directory. The Ethernet device driver interface is defined in <*iniche path*>**/inc/alt_iniche_dev.h**.

The following sections describe how to provide a driver for a new Ethernet device.

Provide the NicheStack Hardware Interface Routines

The NicheStack TCP/IP Stack architecture requires several network hardware interface routines:

- Initialize hardware
- Send packet
- Receive packet
- Close
- Dump statistics

These routines are fully documented in the *Porting Engineer Provided Functions* chapter of the *NicheStack Technical Reference*. The corresponding function in the SMSC lan91c111 device driver are:

Table 7–4. SMSC lan91c111 Hardware Interface Routines

Prototype function	lan91c111 function	File	Notes
`n_init()`	`s91_init()`	**smsc91x.c**	The initialization routine can install an ISR if applicable
`pkt_send()`	`s91_pkt_send()`	**smsc91x.c**	
Packet receive mechanism	`s91_isr()`	**smsc91x.c**	Packet receive includes three key actions: ● `pk_alloc()` — allocate a `netbuf` structure ● `putq()` — place `netbuf` structure on `rcvdq` ● `SignalPktDemux()` — notify the IP layer so that it can demux the packet
	`s91_rcv()`	**smsc91x.c**	
	`s91_dma_rx_done()`	**smsc_mem.c**	
`n_close()`	`s91_close()`	**smsc91x.c**	
`n_stats()`	`s91_stats()`	**smsc91x.c**	

The NicheStack TCP/IP Stack system code uses the `net` structure internally to define its interface to device drivers. The `net` structure is defined in **net.h**, in <*iniche path*>**/src/downloads/30src/h**. Among other things, the `net` structure contains the following things:

- A field for the IP address of the interface
- A function pointer to a low-level function to initialize the MAC device
- Function pointers to low-level functions to send packets

Typical NicheStack code refers to type `NET`, which is defined as `*net`.

Provide *INSTANCE and *INIT Macros

So that the HAL can use your driver, you must provide two HAL macros. The names of these macros are based on the name of your network interface component, according to the following templates:

- <*component name*>`_INSTANCE`
- <*component name*>`_INIT`

For examples, see ALTERA_AVALON_LAN91C111_INSTANCE and ALTERA_AVALON_LAN91C111_INIT in <*SMSC path*>**/inc/iniche/altera_avalon_lan91c111_iniche.h**, which is included in <*iniche path*>**/inc/altera_avalon_lan91c111.h**.

You can copy **altera_avalon_lan91c111_iniche.h** and modify it for your own driver. The HAL expects to find the *INIT and *INSTANCE macros in <*component name*>**.h**, as discussed in "Device Driver Files for the HAL" on page 7–19. You can accomplish this with a #include directive as in **altera_avalon_lan91c111.h**, or you can define the macros directly in <*component name*>**.h.**

Your *INSTANCE macro declares data structures required by an instance of the MAC. These data structures must include an alt_iniche_dev structure. The *INSTANCE macro must initialize the first three fields of the alt_iniche_dev structure, as follows:

- The first field, llist, is for internal use, and must always be set to the value ALT_LLIST_ENTRY.
- The second field, name, must be set to the device name as defined in system.h. For example, **altera_avalon_lan91c111_iniche.h** uses the C preprocessor's ## (concatenation) operator to reference the LAN91C111_NAME symbol defined in **system.h**.
- The third field, init_func, must point to your software initialization function, as described in "Provide a Software Initialization Function". For example, **altera_avalon_lan91c111_iniche.h** inserts a pointer to alt_avalon_lan91c111_init().

Your *INIT macro initializes the driver software. Initialization must include a call to the alt_iniche_dev_reg() macro, defined in **alt_iniche_dev.h**. This macro registers the device with the HAL by adding the driver instance to alt_iniche_dev_list.

When your driver is included in a Nios II BSP project, the HAL automatically initializes your driver by invoking the *INSTANCE and *INIT macros from its alt_sys_init() function. See "Device Driver Files for the HAL" on page 7–19 for further detail about the *INSTANCE and *INIT macros.

Provide a Software Initialization Function

The *INSTANCE() macro inserts a pointer to your initialization function into the alt_iniche_dev structure, as described in "Provide *INSTANCE and *INIT Macros" on page 7–16. Your software initialization function must do at least the three following things:

- Initialize the hardware and verify its readiness
- Finish initializing the `alt_iniche_dev` structure
- Call `get_mac_addr()`

The initialization function must perform any other initialization your driver needs, such as creation and initialization of custom data structures and ISRs.

For details about the `get_mac_addr()` function, see the *Ethernet and the NicheStack TCP/IP Stack - Nios II Edition* chapter of the *Nios II Software Developer's Handbook*.

For an example of a software initialization function, see `alt_avalon_lan91c111_init()` in <*SMSC path*>**/src/iniche/smsc91x.c**.

Integrating a Device Driver into the HAL

This section discusses how to take advantage of the HAL's ability to instantiate and register device drivers during system initialization. You can take advantage of this service, whether you created a device driver for one of the HAL generic device models, or you created a peripheral-specific device driver. Taking advantage of the automation provided by the HAL is mainly a process of placing files in the appropriate place in the HAL directory structure.

Design Flows

As described in the *Overview* chapter of the *Nios II Software Developer's Handbook*, the Nios II EDS offers the following two distinct design flows:

- The Nios II IDE design flow
- The Nios II software build tools design flow

Although HAL device drivers work the same in both flows, there are slight differences in how you create a device driver.

Most of the discussion in the remainder of this section applies to both design flows. Design flow differences are noted explicitly.

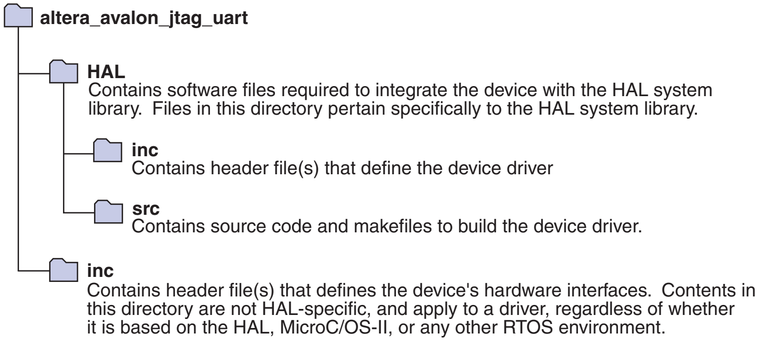
☞ Both design flows include board support packages (BSPs). However, the Nios II IDE design flow refers to a BSP as a system library.

Directory Structure for HAL Devices

Each peripheral in a Nios II system is associated with a specific SOPC Builder component directory. This directory contains a file defining the software interface to the peripheral. See "Accessing Hardware " on page 7–4.

This section uses the example of Altera's JTAG UART component to demonstrate the location of files. Figure 7–1 shows the directory structure of the JTAG UART component directory, which is located in the <*Altera installation*>**/ip/sopc_builder_ip** directory.

Figure 7–1. Directory Structure for HAL UART Driver


altera_avalon_jtag_uart
HAL
Contains software files required to integrate the device with the HAL system library. Files in this directory pertain specifically to the HAL system library.
inc
Contains header file(s) that define the device driver
src
Contains source code and makefiles to build the device driver.
inc
Contains header file(s) that defines the device's hardware interfaces. Contents in this directory are not HAL-specific, and apply to a driver, regardless of whether it is based on the HAL, MicroC/OS-II, or any other RTOS environment.

Device Driver Files for the HAL

This section describes how to provide appropriate files to integrate your device driver into the HAL.

A Device's HAL Header File and alt_sys_init.c

At the heart of the HAL is the auto-generated source file, **alt_sys_init.c**. **alt_sys_init.c** contains the source code that the HAL uses to initialize the device drivers for all supported devices in the system. In particular, this file defines the `alt_sys_init()` function, which is called before `main()` to initialize all devices and make them available to the program.

Example 7–1 on page 7–20 shows excerpts from an **alt_sys_init.c** file.

Example 7–1. Excerpt from an alt_sys_init.c File Performing Driver Initialization

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

/*
 * device headers
 */
#include "altera_avalon_timer.h"
#include "altera_avalon_uart.h"

/*
 * Allocate the device storage
 */
ALTERA_AVALON_UART_INSTANCE( UART1, uart1 );
ALTERA_AVALON_TIMER_INSTANCE( SYSCLK, sysclk );

/*
 * Initialise the devices
 */
void alt_sys_init( void )
{
    ALTERA_AVALON_UART_INIT( UART1, uart1 );
    ALTERA_AVALON_TIMER_INIT( SYSCLK, sysclk );
}
```

When you create a new software project, the Nios II design flow tools generate the contents of **alt_sys_init.c** to match the specific hardware contents of the SOPC Builder system.

In the IDE-managed design flow, for each device visible to the processor, the generator utility searches for an associated header file in the device's **HAL/inc** directory. The name of the header file depends on the SOPC Builder component name. For example, for Altera's JTAG UART component, the generator finds the file **altera_avalon_jtag_uart/HAL/inc/altera_avalon_jtag_uart.h**. If the generator utility finds such a header file, it inserts code into **alt_sys_init.c** to perform the following actions:

- Include the device's header file.
- Call the macro <*name of device*>`_INSTANCE` to allocate storage for the device.
- Call the macro <*name of device*>`_INIT` inside the `alt_sys_init()` function to initialize the device.

In the user-managed design flow, the Quartus II component discovery mechanism performs header file discovery.

You must define the `*_INSTANCE` and `*_INIT` macros in the associated device header file. For example, **altera_avalon_jtag_uart.h** must define the macros `ALTERA_AVALON_JTAG_UART_INSTANCE` and `ALTERA_AVALON_JTAG_UART_INIT`. The purpose of these macros is as follows:

- The `*_INSTANCE` macro performs any per-device static memory allocation that the driver requires.
- The `*_INIT` macro performs runtime initialization of the device.

Both macros take two input arguments:

- The first argument, `name`, is the capitalized name of the device instance.
- The second argument, `dev`, is the lower case version of the device name. `dev` is the name given to the component in SOPC Builder at system generation time.

You can use these input parameters to extract device-specific configuration information from the **system.h** file.

For a complete example, see any of the Altera-supplied device drivers, such as the JTAG UART driver in <*Altera installation*>**\ip\sopc_builder_ip\altera_avalon_jtag_uart**.

For optimal project rebuild time, do not include the peripheral header in **system.h**. It is included in **alt_sys_init.c**.

To publish a device driver for an SOPC builder component, you provide the file <*Altera installation*>**/ip/sopc_builder_ip/**<*component name*>**/HAL/inc/**<*component name*>**.h**. This file is then required to define the macros <*component name*>`_INSTANCE` and <*component name*>`_INIT`. With this infrastructure in place for your device, the HAL instantiates and registers your device driver before calling `main()`.

Device Driver Source Code

In addition to the header, the component driver needs to provide executable source code, to be built into the BSP.

Source Code Discovery in the IDE Design Flow

Place any required source code in the **HAL/src** directory. In addition, you should include a makefile fragment, **component.mk**. The **component.mk** file lists the source files to include in the system library. You can list multiple files by separating filenames with a space. Example 7–2 on page 7–22 shows an example makefile fragment for Altera's JTAG UART device.

Example 7–2. component.mk for a UART Driver

```
C_LIB_SRCS   += altera_avalon_uart.c
ASM_LIB_SRCS +=
INCLUDE_PATH +=
```

The Nios II IDE includes the **component.mk** file into the top-level makefile when compiling system library projects and application projects. **component.mk** can only modify the make variables listed in Table 7–5

Table 7–5. Make Variables Defined in component.mk

Make Variable	Meaning
`C_LIB_SRCS`	The list of C source files to build into the system library.
`ASM_LIB_SRCS`	The list of assembler source files to build into the system library (these are preprocessed with the C preprocessor).
`INCLUDE_PATH`	A list of directories to add to the include search path. The directory <*component*>**/HAL/inc** is added automatically and so does not need to be explicitly defined by the component.

component.mk can add additional make rules and macros as required, but interoperability macro names should conform to the namespace rules. See "Namespace Allocation" on page 7–25

Source Code Discovery in the Build Tools Design Flow

In the build tools design flow, you use Tcl scripts to specify the location of driver source files.

For more information about driver development in the build tools design flow, refer to *"Device Drivers and Software Packages"* in the *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

Reducing Code Footprint

The HAL provides several options for reducing the size, or footprint, of the BSP code. Some of these options require explicit support from device drivers. If you need to minimize the size of your software, consider using one or both of the following techniques in your custom device driver:

- Provide reduced footprint drivers. This technique usually reduces driver functionality.

- Support the lightweight device driver API. This technique reduces driver overhead. It need not reduce functionality, but it might restrict your flexibility in using the driver.

These techniques are discussed in the following sections.

Provide Reduced Footprint Drivers

The HAL defines a C preprocessor macro named `ALT_USE_SMALL_DRIVERS` that you can use in driver source code to provide alternate behavior for systems that require minimal code footprint. If `ALT_USE_SMALL_DRIVERS` is not defined, driver source code implements a fully featured version of the driver. If the macro is defined, the source code might provide a driver with restricted functionality. For example a driver might implement interrupt-driven operation by default, but polled (and presumable smaller) operation if `ALT_USE_SMALL_DRIVERS` is defined.

When writing a device driver, if you choose to ignore the value of `ALT_USE_SMALL_DRIVERS`, the same version of the driver is used regardless of the definition of this macro.

You can enable `ALT_USE_SMALL_DRIVERS` in a BSP as follows:

- In the Nios II IDE, use the **Reduced Device Drivers** option in the system library settings. For further information, refer to the Nios II IDE help system.
- With the Nios II software build tools, use the `hal.enable_reduced_device_drivers` BSP setting. For further information, refer to the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook.*

Support the Lightweight Device Driver API

The lightweight device driver API allows you to minimize the overhead of character-mode device drivers. It does this by removing the need for the `alt_fd` file descriptor table, and the `alt_dev` data structure required by each driver instance.

If you want to support the lightweight device driver API on a character-mode device, you need to write at least one of the lightweight character-mode functions listed in Table 7–6. Implement the functions needed by your software. For example, if you only use the device for `stdout`, you only need to implement the <*component class*>`_write()` function.

To support the lightweight device driver API, name your driver functions based on the component class name, as shown in Table 7–6.

Table 7–6. Driver Functions for Lightweight Device Driver API

Function	Purpose	Example *(1)*
<component class>`_read()`	Implements character-mode read functions	`altera_avalon_jtag_uart_read()`
<component class>`_write()`	Implements character-mode write functions	`altera_avalon_jtag_uart_write()`
<component class>`_ioctl()`	Implements device-dependent functions	`altera_avalon_jtag_uart_ioctl()`

(1) Based on component **altera_avalon_jtag_uart**

When you build your BSP with `ALT_USE_DIRECT_DRIVERS` enabled, instead of using file descriptors, the HAL accesses your drivers with the following macros:

- `ALT_DRIVER_READ(instance, buffer, len, flags)`
- `ALT_DRIVER_WRITE(instance, buffer, len, flags)`
- `ALT_DRIVER_IOCTL(instance, req, arg)`

These macros are defined in *<Nios II EDS install path>***/components/altera_hal/HAL/inc/sys/alt_driver.h**.

These macros, together with the system-specific macros that the Nios II design flow creates in **system.h**, generate calls to your driver functions. For example, with the **Lightweight Device Driver API** options turned on, `printf()` calls the HAL `write()` function, which directly calls your driver's *<component class>*`_write()` function, bypassing file descriptors.

You can enable `ALT_USE_DIRECT_DRIVERS` in a BSP as follows:

- In the Nios II IDE, use the **Lightweight Device Driver API** option in the system library settings. For further information, refer to the Nios II IDE help system.
- With the Nios II software build tools, use the `hal.enable_lightweight_device_driver_api` BSP setting. For further information, refer to the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook.*

You can also take advantage of the lightweight device driver API by invoking `ALT_DRIVER_READ()`, `ALT_DRIVER_WRITE()` and `ALT_DRIVER_IOCTL()` in your application software. To use these macros, include the header file **sys/alt_driver.h**. Replace the `instance` argument with the device instance name macro from **system.h**; or if you are confident that the device instance name will never change, you can use a literal string, e.g. `"custom_uart_0"`.

Another way to use your driver functions is to call them directly, without macros. If your driver includes functions other than <*component class*>`_read()`, <*component class*>`_write()` and <*component class*>`_ioctl()`, you must invoke those functions directly from your application.

Namespace Allocation

To avoid conflicting names for symbols defined by devices in the SOPC Builder system, all global symbols need a defined prefix. Global symbols include global variable and function names. For device drivers, the prefix is the name of the SOPC Builder component followed by an underscore. Because this naming can result in long strings, an alternate short form is also permitted. This short form is based on the vendor name, for example `alt_` is the prefix for components published by Altera. It is expected that vendors test the interoperability of all components they supply.

For example, for the `altera_avalon_jtag_uart` component, the following function names are valid:

- `altera_avalon_jtag_uart_init()`
- `alt_jtag_uart_init()`

The following names are invalid:

- `avalon_jtag_uart_init()`
- `jtag_uart_init()`

As source files are located using search paths, these namespace restrictions also apply to filenames for device driver source and header files.

Overriding the Default Device Drivers

All SOPC Builder components can elect to provide a HAL device driver. See "Integrating a Device Driver into the HAL" on page 7–18. However, if the driver supplied with a component is inappropriate for your application, you can override the default driver by supplying a different driver.

The Nios II IDE locates all include and source files using search paths. The system library project directory is always searched first. If you place an alternative driver in the system library project directory, it overrides drivers installed with the Nios II EDS. For example, if a component provides the header file **alt_my_component.h**, and the system library project directory also contains a file **alt_my_component.h**, the version provided in the system library project directory is used at compile time. This same mechanism can override C and assembler source files.

In the Nios II software build tools design flow, the Quartus II component discovery mechanism finds the driver source files and copies it into the BSP. If you choose to edit or replace these files, your BSP is built with the updated files.

For further details on BSP source files, refer to "*Generated and Copied Files*" in the *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

Referenced Documents

This chapter references the following documents:

- *Overview* chapter of the *Nios II Software Developer's Handbook*
- *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*
- *Overview of the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.
- *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*
- *Exception Handling* chapter of the Nios II Software Developer's Handbook
- *Cache and Tightly-Coupled Memory* chapter of the Nios II Software Developer's Handbook
- *Ethernet and the NicheStack TCP/IP Stack - Nios II Edition* chapter of the Nios II Software Developer's Handbook
- *HAL API Reference* chapter in the Nios II Software Developer's Handbook
- *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*
- *NicheStack Technical Reference Manual*, available at **www.altera.com/literature/lit-nio2.jsp**

Document Revision History

Table 7–7 shows the revision history for this document.

Table 7–7. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Added documentation for HAL device driver development with the Nios II software build tools.	—
May 2007 v7.1.0	● Added table of contents to Introduction section. ● Added Referenced Documents section.	—
March 2007 v7.0.0	No change from previous release.	
November 2006 v6.1.0	● Add section "Reducing Code Footprint" ● Replace lwIP driver section with NicheStack TCP/IP Stack driver section	Lightweight device driver API and minimal file I/O API; NicheStack TCP/IP Stack support.
May 2006 v6.0.0	No change from previous release.	
October 2005 v5.1.0	Added IOADDR_* macro details to section "Accessing Hardware ".	
May 2005 v5.0.0	Updated reference to version of lwIP from 0.7.2 to 1.1.0.	
December 2004 v1.1	Updated reference to version of lwIP from 0.6.3 to 0.7.2.	
May 2004 v1.0	Initial Release.	

Section III. Advanced Programming Topics

This section provides information on advanced programming topics.

This section 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

8. Exception Handling

NII52006-7.2.0

Introduction

This chapter discusses how to write programs to handle exceptions in the Nios® II processor architecture. Emphasis is placed on how to process hardware interrupt requests by registering a user-defined interrupt service routine (ISR) with the hardware abstraction layer (HAL).

This chapter contains the following sections:

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

Nios II Exceptions Overview

Nios II exception handling is implemented in classic RISC fashion, i.e., all exception types are handled by a single exception handler. As such, all exceptions (hardware and software) are handled by code residing at a single location called the "exception address".

The Nios II processor provides the following exception types:

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

Exception Handling Concepts

The following list outlines basic exception handling concepts, with the HAL terms used for each one:

- **application context** — the status of the Nios II processor and the HAL during normal program execution, outside of the exception handler.
- **context switch** — the process of saving the Nios II processor's registers on an exception, and restoring them on return from the interrupt service routine.
- **exception** — any condition or signal that interrupts normal program execution.
- **exception handler** — the complete system of software routines, which service all exceptions and pass control to ISRs as necessary.
- **exception overhead** — additional processing required by exception processing. The exception overhead for a program is the sum of all the time occupied by all context switches.
- **hardware interrupt** — an exception caused by a signal from a hardware device.
- **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 context** — the status of the Nios II processor and the HAL when the exception handler is executing.
- **interrupt request (IRQ)** — a signal from a peripheral requesting a 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.
- **software exception** — an exception caused by a software condition. 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.

- **other exception** — an exception which is not a hardware interrupt nor a trap.

How the Hardware Works

The Nios II processor can respond to software exceptions and hardware interrupts. Thirty-two independent hardware interrupt signals are available. These interrupt signals allow software to prioritize interrupts, although the interrupt signals themselves have no inherent priority.

When the Nios II processor responds to an exception, it does the following things:

1. Saves the `status` register in `estatus`. This means that if hardware interrupts are enabled, the `EPIE` bit of `estatus` is set.

2. Disables hardware interrupts.

3. Saves the next execution address in `ea` (`r29`).

4. Transfers control to the Nios II processor exception address.

☞ Nios II exceptions and interrupts are not vectored. Therefore, the same exception address receives control for all types of interrupts and exceptions. The exception handler at that address must determine the type of exception or interrupt.

For details about the Nios II processor exception and interrupt controller, see the *Processor Architecture* chapter of the *Nios II Processor Reference Handbook*.

ISRs

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

When you use the Nios II IDE to create a system library project, the IDE includes all needed ISRs. 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 the HAL system library for handling hardware interrupts.

You can also look at existing handlers for Altera® SOPC Builder components for examples of how to write HAL ISRs.

For more details about the Altera-provided HAL handlers, see the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook.*

HAL API for ISRs

The HAL system library provides an API to help ease the creation and maintenance of ISRs. This API also applies to programs based on a real-time operating system (RTOS) such as MicroC/OS-II, because the full HAL API is available to RTOS-based programs. The HAL 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 on these functions, see the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

Using the HAL API to implement ISRs entails the following steps:

1. Write your ISR that handles interrupts for a specific device.

2. Your program must register the ISR with the HAL by calling the `alt_irq_register()` function. `alt_irq_register()` enables interrupts for you, by calling `alt_irq_enable_all()`.

Writing an ISR

The ISR you write must match the prototype that `alt_irq_register()` expects to see. The prototype for your ISR function must match the prototype:

```
void isr (void* context, alt_u32 id)
```

The parameter definitions of `context` and `id` are the same as for the `alt_irq_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 interrupt condition is specific to the peripheral.

For details, see the relevant chapter in the *Quartus® II Handbook, Volume 5: Altera Embedded Peripherals*.

When the ISR has finished servicing the interrupt, it must return to the HAL exception handler.

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

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 waiting for an 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, i.e., the system can become permanently blocked within 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 an interrupt that never occurs because interrupts are disabled.

Registering an ISR

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

```
int alt_irq_register (alt_u32 id,
                      void*   context,
                      void    (*isr)(void*, alt_u32));
```

The prototype has the following parameters:

- id is the hardware interrupt number for the device, as defined in **system.h**. Interrupt priority corresponds inversely to the IRQ number. Therefore, IRQ$_0$ represents the highest priority interrupt and IRQ$_{31}$ is the lowest.
- context is a pointer 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 function that is called in response to IRQ number id. The two input arguments provided to this function are the context pointer and id. Registering a null pointer for isr results in the interrupt being disabled.

The HAL registers the ISR by the storing the function pointer, isr, in a lookup table. The return code from alt_irq_register() is zero if the function succeeded, and nonzero if it failed.

If the HAL registers your ISR successfully, the associated Nios II interrupt (as defined by id) is enabled on return from alt_irq_register().

☞ Hardware-specific initialization might also be required.

When a specific IRQ occurs, the HAL looks up the IRQ in the lookup table and dispatches the registered ISR.

For details of interrupt initialization specific to your peripheral, see the relevant chapter in the *Quartus II Handbook, Volume 5: Altera Embedded Peripherals*. For details on alt_irq_register(), see the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

Enabling and Disabling ISRs

The HAL provides the functions alt_irq_disable(), alt_irq_enable(), alt_irq_disable_all(), alt_irq_enable_all(), and alt_irq_enabled() to allow a program to disable interrupts for certain sections of code, and re-enable them later. alt_irq_disable() and alt_irq_enable() allow you to disable and enable individual interrupts. alt_irq_disable_all() disables all interrupts, and returns a context value. To re-enable 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 non-zero if interrupts are enabled, allowing a program to check on the status of interrupts.

☞ Disable interrupts for as short a time as possible. Maximum interrupt latency increases with the amount of time interrupts are disabled. For more information about disabled interrupts, see "Keep Interrupts Enabled" on page 8–11.

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

C Example

The following code illustrates an ISR that services an interrupt from a button PIO. 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 via the context pointer.

Example: An ISR to Service a Button PIO IRQ

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

static void handle_button_interrupts(void* context, alt_u32 id)
{
  /* cast the context pointer to an integer pointer. */
  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);

  /* reset interrupt capability for the Button PIO. */
  IOWR_ALTERA_AVALON_PIO_IRQ_MASK(BUTTON_PIO_BASE, 0xf);
}
```

The following code shows an example of the code for the main program that registers the ISR with the HAL.

Example: 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. */
    alt_irq_register( BUTTON_PIO_IRQ,
                        edge_capture_ptr,
                        handle_button_interrupts );
}
```

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

1. Button is pressed, generating an IRQ.

2. The HAL exception handler is invoked and dispatches the `handle_button_interrupts()` ISR.

3. `handle_button_interrupts()` services the interrupt and returns.

4. Normal program operation continues with an updated value of `edge_capture`.

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

ISR Performance Data

This section provides performance data related to ISR processing on the Nios II processor. The following three key metrics determine ISR performance:

- Interrupt latency—the time from when an interrupt is first generated to when the processor runs the first instruction at the exception address.
- Interrupt response time—the time from when an interrupt is first generated to when the processor runs the first instruction in the ISR.
- Interrupt recovery time—the time taken from the last instruction in the ISR to return to normal processing.

Because the Nios II processor is highly configurable, there is no single typical number for each metric. This section provides data points for each of the Nios II cores under the following assumptions:

- All code and data is stored in on-chip memory.
- The ISR code does not reside in the instruction cache.
- The software under test is based on the Altera-provided HAL exception handler system.
- The code is compiled using compiler optimization level "–O3", or high optimization.

Table 8–1 lists the interrupt latency, response time, and recovery time for each Nios II core.

Table 8–1. Interrupt Performance Data (1)

Core	Latency	Response Time	Recovery Time
Nios II/f	10	105	62
Nios II/s	10	128	130
Nios II/e	15	485	222

Note to Table 8–1:

(1) The numbers indicate time measured in CPU clock cycles.

The results you experience in a specific application can vary significantly based on several factors discussed in the next section.

Improving ISR Performance

If your software uses 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, see "Hardware Performance Improvements" on page 8–13.

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

Move Lengthy Processing to Application Context

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

If lengthy processing is needed, design your software to perform this processing outside of the interrupt 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.

Move Lengthy Processes to Hardware

Processor-intensive tasks must often transfer large amounts of data to and from peripherals. A general-purpose CPU 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 the *Using DMA Devices* section of the *Developing Programs using the HAL* 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 IRQs, 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 events. Buffers and queues can fill or overflow. Even in the absence of overflow, maximum interrupt processing time can increase after interrupts are disabled, because the ISRs must process data backlogs.

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

Instead of disabling all interrupts, call `alt_irq_disable()` and `alt_irq_enable()` to enable and disable individual IRQs.

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 re-enable interrupts immediately.

Use Fast Memory

ISR performance depends upon memory speed.

Place the ISRs and the stack in the fastest available memory.

For best performance, place the stack in on-chip memory, preferably tightly-coupled memory, if available.

If it is not possible to place the main stack in fast memory, you can use a private exception stack, mapped to a fast memory section. However, the private exception stack entails some additional context switch overhead, so use it only if you are able to place it in significantly faster memory. You can specify a private exception stack on the **System properties** page of the Nios II IDE.

For more information about mapping memory, see the "Memory Usage" section of the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*. For more information on tightly-coupled memory, refer to the *Cache and Tightly-Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.

Use Nested ISRs

The HAL system library 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, because the ISR does not need to be reentrant.

However, first-come first-served execution means that the HAL interrupt priorities only have effect if two IRQs are asserted on the same application-level instruction cycle. A low-priority interrupt occurring before a higher-priority IRQ 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 interrupt prioritization by using nested ISRs. With nested ISRs, higher priority interrupts are allowed to interrupt lower-priority ISRs.

This technique can improve the interrupt latency of higher priority ISRs.

☞ Nested ISRs increase the processing time for lower priority interrupts.

If your ISR is very short, it might not be worth the overhead to re-enable higher-priority interrupts. Enabling nested interrupts in a short ISR can actually increase the interrupt latency of higher priority interrupts.

☞ If you use a private exception stack, you cannot nest interrupts. For more information about private exception stacks, see "Use Fast Memory" on page 8–11.

To implement nested interrupts, use the `alt_irq_interruptible()` and `alt_irq_non_interruptible()` functions to bracket code within a processor-intensive ISR. After the call to `alt_irq_interruptible()`, higher priority IRQs can interrupt the running ISR. When your ISR calls `alt_irq_non_interruptible()`, interrupts are disabled as they were before `alt_irq_interruptible()`.

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

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 interrupt response time.

For further information about compiler optimizations, refer to the *Reducing Code Footprint* section in the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*.

Hardware Performance Improvements

There are several simple hardware changes that 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, see "Software Performance Improvements" on page 8–9.

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

Add Fast Memory

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

For further information about tightly-coupled memory, refer to the *Cache and Tightly-Coupled Memory* chapter in the *Nios II Processor Reference 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, see the *DMA Controller Core* chapter of the *Quartus II Handbook, Volume 5: Embedded Peripherals*.

Place the Exception Handler Address in Fast Memory

For the fastest execution of exception code, place the exception address in a fast memory device. For example, an on-chip RAM with zero waitstates 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 interrupt 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 Interrupt Priorities

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

Use the Interrupt Vector Custom Instruction

The Nios II processor core offers an interrupt vector custom instruction which accelerates interrupt vector dispatch in the Hardware Abstraction Layer (HAL). You can choose to 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.

For further information about the interrupt vector custom instruction, see the *Interrupt Vector Custom Instruction* section in the chapter entitled *Instantiating the Nios II Processor in SOPC Builder* in the *Nios II Processor Reference Handbook.*

Debugging ISRs

You can debug an ISR with the Nios II IDE by setting breakpoints within the ISR. The debugger completely halts the processor upon reaching a breakpoint. In the meantime, however, the other hardware in your system continues to operate. Therefore, it is inevitable that other IRQs 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 have to reset the processor to return the system to a known state.

The `ipending` register (`ctl4`) is masked to all zeros during single stepping. This masking prevents the processor from servicing IRQs that are asserted while you single-step through code. As a result, if you try to single step through a part of the exception handler code (e.g. `alt_irq_entry()` or `alt_irq_handler()`) that reads the `ipending` register, the code does not detect any pending IRQs. This issue does not affect debugging software exceptions. You can set breakpoints within your ISR code (and single step through it), because the exception handler has already used `ipending` to determine which IRQ caused the exception.

Summary of Guidelines for Writing ISRs

This section summarizes guidelines for writing ISRs for the HAL framework:

- Write your ISR function to match the prototype: `void isr (void* context, alt_u32 id).`
- Register your ISR using the `alt_irq_register()` function provided by the HAL API.
- Do not use the C standard library I/O functions, such as `printf()`, inside of an ISR.

HAL Exception Handler Implementation

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

This information is for your reference. You can take advantage of the HAL exception services without a complete understanding of the HAL implementation. For details of how to install ISRs using the HAL application programming interface (API), see "ISRs" on page 8–3.

Exception Handler Structure

The exception handling system consists of the following components:

- The top-level exception handler
- The hardware interrupt handler
- The software exception handler
- An ISR for each peripheral that generates interrupts.

When the Nios II processor generates an exception, the top-level exception handler receives control. The top-level exception handler passes control to either the hardware interrupt handler or the software exception handler. The hardware interrupt handler passes control to one or more ISRs.

Each time an exception occurs, the exception handler services either a software exception or hardware interrupts, with hardware interrupts having a higher priority. The HAL does not support nested exceptions, but can handle multiple hardware interrupts per context switch. For details, see "Hardware Interrupt Handler" on page 8–18.

Top-Level Exception Handler

The top-level exception handler provided with the HAL system library is located at the Nios II processor's exception address. When an exception occurs and control transfers to the exception handler, it does the following:

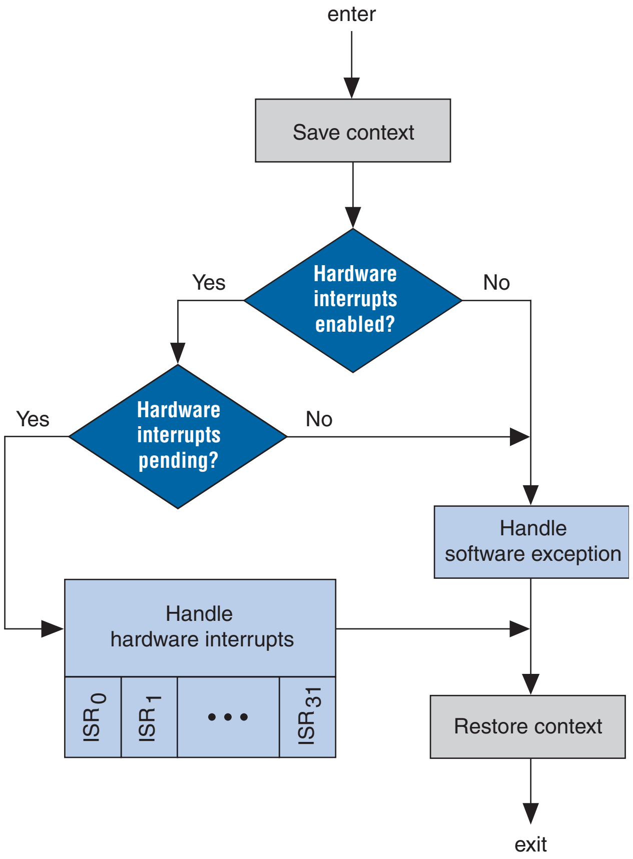
1. Creates the private exception stack (if specified)

2. Stores register values onto the stack

3. Determines the type of exception, and passes control to the correct handler

Figure 8–1 shows the algorithm that HAL top-level exception handler uses to distinguish between hardware interrupts and software exceptions.

Figure 8–1. HAL Top-Level Exception Handler


enter
Save context
Hardware interrupts enabled?
Yes
No
Hardware interrupts pending?
Yes
No
Handle software exception
Handle hardware interrupts
ISR 0
ISR 1
• • •
ISR 31
Restore context
exit

The top-level exception handler looks at the `estatus` register to determine the interrupt enable status. If the `EPIE` bit is set, hardware interrupts were enabled at the time the exception happened. If so, the exception handler looks at the IRQ bits in `ipending`. If any IRQs are asserted, the exception handler calls the 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 top-level exception handler calls the software exception handler.

All hardware interrupts are higher priority than software exceptions.

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

Upon return from the hardware interrupt or software exception handler, the top-level exception handler does the following:

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

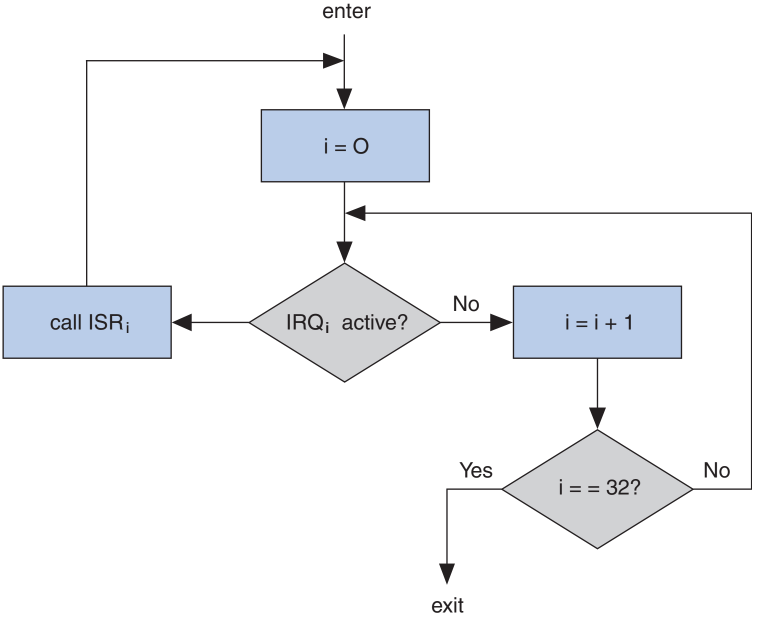
Hardware Interrupt Handler

The Nios II processor supports thirty-two hardware interrupts. In the HAL exception handler, hardware interrupt 0 has the highest priority, and 31 the lowest. This prioritization is a feature of the HAL exception handler, and is not inherent in the Nios II exception and interrupt controller.

The hardware interrupt handler 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 exception handler begins scanning the IRQs again, starting at IRQ_0. In this way, higher priority exceptions are always processed before lower-priority exceptions. When all IRQs are clear, the hardware interrupt handler returns to the top level. Figure 8–2 shows a flow diagram of the HAL hardware interrupt handler.

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, see "Use the Interrupt Vector Custom Instruction" on page 8–14.

Figure 8–2. HAL Hardware Interrupt Handler



Software Exception Handler

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

Software exception handling depends on options selected in the Nios II IDE. If you have enabled unimplemented instruction emulation, the exception handler first checks to see if an unimplemented instruction caused the exception. If so, it emulates the instruction. Otherwise, it handles traps and other 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 on unimplemented instructions, see 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 8–23.

When to Use the Unimplemented Instruction Handler

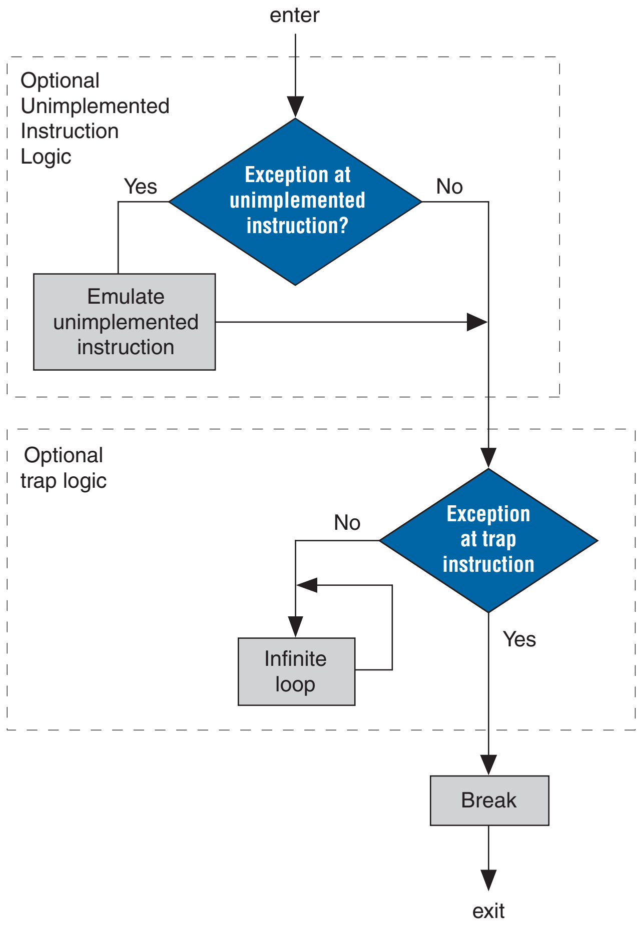
You do not normally need the unimplemented instruction handler, because the Nios II IDE includes software emulation for unimplemented instructions from its run-time libraries if you are compiling for a Nios II processor that does not support the instructions.

Here are the circumstances under which you might need the unimplemented instruction handler:

- 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 handler, including the optional instruction emulation logic. If instruction emulation is not enabled, this logic is omitted.

Figure 8–3. HAL Software Exception Handler


enter
Optional Unimplemented Instruction Logic
Exception at unimplemented instruction?
Yes
No
Emulate unimplemented instruction
Optional trap logic
Exception at trap instruction
No
Yes
Infinite loop
Break
exit

If unimplemented instruction emulation is disabled, but the processor encounters an unimplemented instruction, the exception handler treats resulting exception as an other exception. Other exceptions are described in "Other Exceptions" on page 8–22.

Using the Unimplemented Instruction Handler

The unimplemented instruction handler defines an emulation routine for each of the implementation-dependent instructions. In this way, the full Nios II instruction set is always supported, even if a particular Nios II core does not implement all instructions in hardware.

To include the unimplemented instruction handler, turn on **Emulate multiply and divide instructions** on the **System properties** page of the Nios II IDE. The emulation routines are small (less than ¾ KBytes of memory), so it is usually safe to include them even when targeting a Nios II core that does not require them. If a Nios II core implements a particular instruction in hardware, its corresponding exception never occurs.

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

Software Trap Handling

If the cause of the software exception is not an unimplemented instruction, the HAL software exception handler checks for a `trap` instruction. The HAL is not designed to handle software traps. If it finds one, it executes a `break`.

If your software is compiled for release, the exception handler makes a distinction between traps and other exceptions. If your software is compiled for debug, traps and other exceptions are handled identically, by executing a `break` instruction. Figure 8–3 shows a flowchart of the HAL software exception handler, including the optional trap logic. If your software is compiled for debug, the trap logic is omitted.

In the Nios II IDE, you can select debug or release compilation in the **Project Properties** dialog box, under **C/C++ Build**.

Other Exceptions

If the exception is not caused by an unimplemented instruction or a trap, it is an other exception. In a debugging environment, the processor executes a `break`, allowing the debugger to take control. In a non-debugging environment, the processor goes into an infinite loop.

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

Other exceptions can occur for these reasons:

- You need to include the unimplemented instruction handler, discussed in "Unimplemented Instructions" on page 8–19.

- A peripheral is generating spurious 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 exception handler cannot detect or respond to an invalid instruction.

☞ Invalid instructions are different from unimplemented instructions, which are described in "Unimplemented Instructions" on page 8–19.

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

HAL Exception Handler Files

The HAL exception handling code is in the following files:

- Source files:
  - alt_exception_entry.S
  - alt_exception_muldiv.S
  - alt_exception_trap.S
  - alt_irq_entry.S
  - alt_irq_handler.c
  - alt_software_exception.S
  - alt_irq_vars.c
  - alt_irq_register.c
- Header files:
  - alt_irq.h
  - alt_irq_entry.h

Referenced Documents

This chapter references the following documents:

- *Programming Model* chapter of the *Nios II Processor Reference Handbook*
- *Processor Architecture* chapter of the *Nios II Processor Reference Handbook*
- *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*

- *HAL API Reference* chapter in the *Nios II Software Developer's Handbook*
- *Quartus II Handbook, Volume 5: Embedded Peripherals*
- *Cache and Tightly Coupled Memory* chapter of the *Nios II Software Developer's Handbook*
- *Using Nios II Tightly Coupled Memory Tutorial*
- *DMA Controller Core* chapter of the *Quartus II Handbook, Volume 5: Embedded Peripherals*
- *Instantiating the Nios II Processor in SOPC Builder* chapter of the *Nios II Processor Reference Handbook*
- *Instruction Set Reference* chapter of the *Nios II Processor Reference Handbook*
- *Nios II Core Implementation Details* chapter of the *Nios II Processor Reference Handbook*

Document Revision History

Table 8–2 shows the revision history for this document.

Table 8–2. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	No change from previous release.	
May 2007 v7.1.0	● Chapter 7 was formerly chapter 6. ● Added table of contents to Introduction section. ● Added Referenced Documents section.	
March 2007 v7.0.0	No change from previous release.	
November 2006 v6.1.0	● Describes support for the interrupt vector custom instruction.	Interrupt vector custom instruction added.
May 2006 v6.0.0	● Corrected error in `alt_irq_enable_all()` usage ● Added illustrations ● Revised text on optimizing ISRs ● Expanded and revised text discussing HAL exception handler code structure.	
October 2005 v5.1.0	● Updated references to HAL exception-handler assembly source files in section "HAL Exception Handler Files". ● Added description of `alt_irq_disable()` and `alt_irq_enable()` in section "ISRs".	
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.	
September 2004 v1.1	● Changed examples. ● Added ISR performance data.	
May 2004 v1.0	Initial Release.	

9. Cache and Tightly-Coupled Memory

NII52007-7.2.0

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 HAL system library 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 when you need to 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:

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.

In all current Nios II cores, there is no hardware cache coherency mechanism. Therefore, if there are multiple masters accessing shared memory, software must explicitly maintain coherency across all masters.

For complete details on the features of each Nios II core implementation, see 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. The following code 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: 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 API provides the following functions for managing cache memory.:

- `alt_dcache_flush()`
- `alt_dcache_flush_all()`
- `alt_icache_flush()`
- `alt_icache_flush_all()`
- `alt_uncached_malloc()`
- `alt_uncached_free()`
- `alt_remap_uncached()`
- `alt_remap_cached()`

For details on API functions, see 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. *The Cache Memory Book* by Jim Handy is a good text that covers 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 the reset handler address is required to 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 `initi` instruction is used to initialize one instruction cache line. Do not use the `flushi` instruction because it may cause undesired effects when used to initialize the instruction cache in future Nios II implementations.

Place the initi instruction in a loop that executes `initi` for each instruction cache line address. The following code shows an example of assembly code to initialize the instruction cache.

Example: 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:
        initi   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 `initd` instruction is used to initialize one data cache line. Do not use the `flushd` 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 `flushd` can cause unexpected writes of random data to random addresses. The `initd` instruction does not write back dirty data.

Place the `initd` instruction in a loop that executes `initd` for each data cache line address. The following code shows an example of assembly code to initialize the data cache:

Example: 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:
        initd   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 don't implement one or both of the caches. The initi and initd instructions are simply treated as nop instructions if there is no cache of the corresponding type present.

For HAL System Library Users

Programs based on the HAL do not have to 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() are 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.

For device drivers, the data cache should be bypassed by using 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.

For C programmers, note that declaring a pointer as volatile does not cause accesses using that volatile pointer to bypass the data cache. The volatile keyword only 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 System Library 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
IORD(BASE, REGNUM)	Read the value of the register at offset REGNUM within a device with base address BASE. Registers are assumed to be offset by the address width of the bus.
IOWR(BASE, REGNUM, DATA)	Write the value DATA to the register at offset REGNUM within a device with base address BASE. Registers are assumed to be offset by the address width of the bus.
IORD_32DIRECT(BASE, OFFSET)	Make a 32-bit read access at the location with address BASE+OFFSET.
IORD_16DIRECT(BASE, OFFSET)	Make a 16-bit read access at the location with address BASE+OFFSET.
IORD_8DIRECT(BASE, OFFSET)	Make an 8-bit read access at the location with address BASE+OFFSET.
IOWR_32DIRECT(BASE, OFFSET, DATA)	Make a 32-bit write access to write the value DATA at the location with address BASE+OFFSET.
IOWR_16DIRECT(BASE, OFFSET, DATA)	Make a 16-bit write access to write the value DATA at the location with address BASE+OFFSET.
IOWR_8DIRECT(BASE, OFFSET, DATA)	Make an 8-bit write access to write the value DATA at the location with address BASE+OFFSET.

Writing Program Loaders or Self-Modifying Code

Software that writes instructions into memory, such as program loaders or self-modifying code, needs to ensure that old instructions are flushed from the instruction cache and CPU 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 into memory.

The following code shows assembly code that writes a new instruction to memory.

Example: 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 CPU pipeline has not prefetched the old instruction at the address specified by `r5`.

Notice that the above code sequence used the `stw/flushd` pair instead of the `stwio` instruction. Using a `stwio` instruction doesn't flush the data cache so could leave stale data in the data cache.

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

For Users of the HAL System Library

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

Managing Cache in Multi-Master/Multi-CPU 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 is used to ensure 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 of utmost importance. 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 actions 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 CPU; 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 within 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 is dependent 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 System Library 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. See Table 9–1. These macros are provided to access non-cacheable 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 isn't 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, put 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 the "Assigning Code and Data to Memory Partitions" section of the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*.

Referenced Documents

This chapter references the following documents:

- *Nios II Core Implementation Details* chapter of the *Nios II Processor Reference Handbook*
- *HAL API Reference* chapter in the *Nios II Software Developer's Handbook*

Document Revision History

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

Table 9–2. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	No change from previous release.	
May 2007 v7.1.0	● Chapter 8 was formerly chapter 7. ● Added table of contents to Introduction section. ● Added Referenced Documents section.	
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	Added detail to section “Tightly-Coupled Memory”.	
May 2005 v5.0.0	Added tightly-coupled memory section.	
May 2004 v1.0	Initial Release.	

10. MicroC/OS-II Real-Time Operating System

NII52008-7.2.0

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 10–8

As described in the *Overview* chapter of the *Nios II Software Developer's Handbook*, the Nios II EDS offers the following two distinct design flows:

- The Nios II IDE design flow
- The Nios II software build tools design flow

Most of the information in this chapter applies to both design flows. Design flow differences are noted explicitly.

☞ Both design flows include board support packages (BSPs). However, the Nios II IDE design flow refers to a BSP as a system library.

For more detailed information about developing MicroC/OS-II programs in the Nios II software build tools design flow, refer to the *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

Overview

MicroC/OS-II is a popular real-time kernel produced by Micrium Inc., and is documented in the book *MicroC/OS-II - The Real Time Kernel* by Jean J. Labrosse (CMP Books). The book describes MicroC/OS-II as a portable, ROMable, scalable, preemptive, 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 advanced programming interface (API).
- It is easy to implement interrupt service routines (ISRs).

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*. Further information is also available on the Micrium website, **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 contact Micrium to obtain a license at **Licensing@Micrium.com** or **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 real-time operating system (RTOS). In addition to MicroC/OS-II, many other RTOSes are available from third-party vendors.

For a complete list of RTOSes that support the Nios II processor, visit the Nios II home page at **www.altera.com/nios2**.

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 fact, Nios development boards are pre-programmed with a web server reference design based on MicroC/OS-II and the Lightweight IP TCP/IP stack.

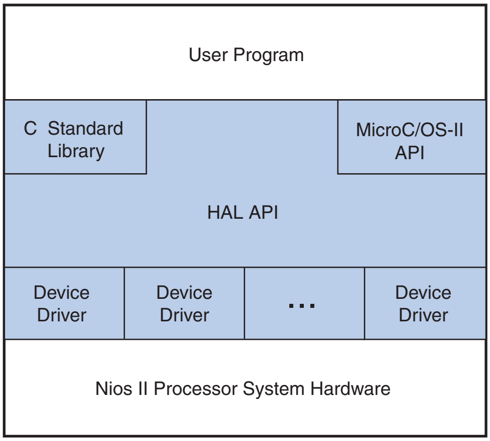
The Altera implementation of MicroC/OS-II is designed to be easy to use. Using the Nios II project settings, you can control the configuration for all the RTOS's modules. You need not modify source files directly to enable or disable kernel features. Nonetheless, Altera provides the Nios II processor-specific source code if you ever wish to examine it. The code is provided in directory <*Nios II EDS install path*>**/components/altera_nios2/UCOSII**. The processor-independent code resides in <*Nios II EDS install path*>**/components/micrium_uc_osii**. The MicroC/OS-II software component 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 into the project.

MicroC/OS-II Architecture

The Altera implementation of MicroC/OS-II for the Nios II processor is essentially a superset of the HAL. It is the HAL environment extended by the inclusion of the MicroC/OS-II scheduler and the associated MicroC/OS-II API. The complete HAL API is available from within MicroC/OS-II projects.

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

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



The multi-threaded environment affects certain HAL functions.

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

MicroC/OS-II Thread-Aware Debugging

When debugging a MicroC/OS-II application, the debugger can display the current state of all threads within 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 (i.e., an SOPC Builder component) can provide include files and source files within the **inc** and **src** subdirectories of the component's **HAL** directory.

For more information, refer to the *Developing Device Drivers for the HAL* 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 with the Nios II integrated development environment (IDE), these directories are added to the search paths for source and include files.

The Nios II software build tools copy the files into your BSP's **obj** subdirectory.

☞ For more information about specifying file paths with the Nios II software build tools, refer to *"Board Support Packages"* in the *Using 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 HAL* 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 allow the same driver to be portable across the HAL and MicroC/OS-II environments, Altera defines a set of OS-independent macros that provide access to operating system facilities. When compiled for a MicroC/OS-II project, the macros expand to a MicroC/OS-II API call. 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 drivers with similar portability.

Table 10–1 lists the available macros and their function.

For more information on the functionality in the MicroC/OS-II environment, see *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
`ALT_FLAG_GRP(group)`	**os/alt_flag.h**	Create a pointer to a flag group with the name `group`.	Empty statement.
`ALT_EXTERN_FLAG_GRP(group)`	**os/alt_flag.h**	Create an external reference to a pointer to a flag group with name `group`.	Empty statement.
`ALT_STATIC_FLAG_GRP(group)`	**os/alt_flag.h**	Create a static pointer to a flag group with the name `group`.	Empty statement.

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_CREATE(group, flags)`	**os/alt_flag.h**	Call `OSFlagCreate()` to initialize the flag group pointer, `group`, with the flags value `flags`. The error code is the return value of the macro.	Return 0 (success).
`ALT_FLAG_PEND(group, flags, wait_type, timeout)`	**os/alt_flag.h**	Call `OSFlagPend()` with the first four input arguments set to `group`, `flags`, `wait_type`, and `timeout` respectively. The error code is the return value of the macro.	Return 0 (success).
`ALT_FLAG_POST(group, flags, opt)`	**os/alt_flag.h**	Call `OSFlagPost()` with the first three input arguments set to `group`, `flags`, and `opt` respectively. The error code is the return value of the macro.	Return 0 (success).
`ALT_SEM(sem)`	**os/alt_sem.h**	Create an OS_EVENT pointer with the name `sem`.	Empty statement.
`ALT_EXTERN_SEM(sem)`	**os/alt_sem.h**	Create an external reference to an `OS_EVENT` pointer with the name `sem`.	Empty statement.
`ALT_STATIC_SEM(sem)`	**os/alt_sem.h**	Create a static `OS_EVENT` pointer with the name `sem`.	Empty statement.
`ALT_SEM_CREATE(sem, value)`	**os/alt_sem.h**	Call `OSSemCreate()` with the argument `value` to initialize the `OS_EVENT` pointer `sem`. The return value is zero upon success, or negative otherwise.	Return 0 (success).
`ALT_SEM_PEND(sem, timeout)`	**os/alt_sem.h**	Call `OSSemPend()` with the first two argument set to `sem` and `timeout` respectively. The error code is the return value of the macro.	Return 0 (success).
`ALT_SEM_POST(sem)`	**os/alt_sem.h**	Call `OSSemPost()` with the input argument `sem`.	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 within 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. Upon 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. Therefore, you need to make allowance 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 on the contents of the `_reent` structure, refer to the newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II <version>, **Nios II Documentation**.

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

Interrupt Service Routines for MicroC/OS-II

Implementing interrupt service routines (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 does the following on behalf of your ISR:

- Saving and restoring processor registers
- Calling `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 system library settings in the Nios II IDE, or BSP settings with the Nios II software build tools.

Traditionally, you had to configure MicroC/OS-II using `#define` directives in the file **OS_CFG.h**. Instead, the Nios II IDE provides a GUI that allows you to configure each option. Therefore, you do not need to edit header files or source code to configure the MicroC/OS-II features. The GUI settings are reflected in the BSP's **system.h** file; **OS_CFG.h** simply includes **system.h**.

The Nios II software build tools provide access to the same settings as the Nios II IDE.

For further information about system library settings, refer to the Nios II IDE help system. For further information about BSP settings, refer to the *Using the Nios II Software Build Tools* and *Nios II Software Build Tools Reference* chapters of the *Nios II Software Developer's Handbook*.

The following sections define the MicroC/OS-II settings available in Nios II projects. The meaning of each setting is defined fully in *MicroC/OS-II – The Real-Time Kernel*.

For step-by-step instructions on how to create a MicroC/OS-II project in the Nios II IDE, refer to *Using the MicroC/OS-II RTOS with the Nios II Processor Tutorial*.

MicroC/OS-II General Options

Table 10–2 shows the general options.

Table 10–2. General Options (Part 1 of 2)

Option	Description
Maximum number of tasks	Maps onto the `#define OS_MAX_TASKS`. Must be at least 2
Lowest assignable priority	Maps on the `#define OS_LOWEST_PRIO`. Maximum allowable value is 63.
Enable code generation for event flags	Maps onto the `#define OS_FLAG_EN`. When disabled, event flag settings are also disabled. See "Event Flags Settings" on page 10–9.
Enable code generation for mutex semaphores	Maps onto the #define `OS_MUTEX_EN`. When disabled, mutual exclusion semaphore settings are also disabled. See "Mutex Settings" on page 10–9

Table 10–2. General Options (Part 2 of 2)

Option	Description
Enable code generation for semaphores	Maps onto the `#define OS_SEM_EN`. When disabled, semaphore settings are also disabled. See "Semaphores Settings" on page 10–10.
Enable code generation for mailboxes	Maps onto the `#define OS_MBOX_EN`. When disabled, mailbox settings are also disabled. See "Mailboxes Settings" on page 10–10.
Enable code generation for queues	Maps onto the `#define OS_Q_EN`. When disabled, queue settings are also disabled. See "Queues Settings" on page 10–10.
Enable code generation for memory management	Maps onto the `#define OS_MEM_EN`. When disabled, memory management settings are also disabled. See "Memory Management Settings" on page 10–11.

Event Flags Settings

Table 10–3 shows the event flag settings.

Table 10–3. Event Flags Settings

Setting	Description
Include code for wait on clear event flags	Maps on `#define OS_FLAG_WAIT_CLR_EN`.
Include code for `OSFlagAccept()`	Maps on `#define OS_FLAG_ACCEPT_EN`.
Include code for `OSFlagDel()`	Maps on `#define OS_FLAG_DEL_EN`.
Include code for `OSFlagQuery()`	Maps onto the `#define OS_FLAG_QUERY_EN`.
Maximum number of event flag groups	Maps onto the `#define OS_MAX_FLAGS`.
Size of name of event flags group	Maps onto the `#define OS_FLAG_NAME_SIZE`.

Mutex Settings

Table 10–4 shows the mutex settings.

Table 10–4. Mutex Settings

Setting	Description
Include code for `OSMutexAccept()`	Maps onto the `#define OS_MUTEX_ACCEPT_EN`.
Include code for `OSMutexDel()`	Maps onto the `#define OS_MUTEX_DEL_EN`.
Include code for `OSMutexQuery()`	Maps onto the `#define OS_MUTEX_QUERY_EN`.

Semaphores Settings

Table 10–5 shows the semaphores settings.

Table 10–5. Semaphores Settings

Setting	Description
Include code for `OSSemAccept()`	Maps onto the `#define OS_SEM_ACCEPT_EN`.
Include code for `OSSemSet()`	Maps onto the `#define OS_SEM_SET_EN`.
Include code for `OSSemDel()`	Maps onto the `#define OS_SEM_DEL_EN`.
Include code for `OSSemQuery()`	Maps onto the `#define OS_SEM_QUERY_EN`.

Mailboxes Settings

Table 10–6 shows the mailbox settings.

Table 10–6. Mailboxes Settings

Setting	Description
Include code for `OSMboxAccept()`	Maps onto `#define OS_MBOX_ACCEPT_EN`.
Include code for `OSMBoxDel()`	Maps onto `#define OS_MBOX_DEL_EN`.
Include code for `OSMboxPost()`	Maps onto `#define OS_MBOX_POST_EN`.
Include code for `OSMboxPostOpt()`	Maps onto `#define OS_MBOX_POST_OPT_EN`.
Include code fro `OSMBoxQuery()`	Maps onto `#define OS_MBOX_QUERY_EN`.

Queues Settings

Table 10–7 shows the queues settings.

Table 10–7. Queues Settings (Part 1 of 2)

Setting	Description
Include code for `OSQAccept()`	Maps onto `#define OS_Q_ACCEPT_EN`.
Include code for `OSQDel()`	Maps onto `#define OS_Q_DEL_EN`.
Include code for `OSQFlush()`	Maps onto `#define OS_Q_FLUSH_EN`.
Include code for `OSQPost()`	Maps onto `#define OS_Q_POST_EN`.
Include code for `OSQPostFront()`	Maps onto `#define OS_Q_POST_FRONT_EN`.
Include code for `OSQPostOpt()`	Maps onto `#define OS_Q_POST_OPT_EN`.

Table 10–7. Queues Settings (Part 2 of 2)

Setting	Description
Include code for `OSQQuery()`	Maps onto `#define OS_Q_QUERY_EN`.
Maximum number of Queue Control blocks	Maps onto `#define OS_MAX_QS`.

Memory Management Settings

Table 10–8 shows the memory management settings.

Table 10–8. Memory Management Settings

Setting	Description
Include code for `OSMemQuery()`	Maps onto `#define OS_MEM_QUERY_EN`.
Maximum number of memory partitions	Maps onto `#define OS_MAX_MEM_PART`.
Size of memory partition name	Maps onto `#define OS_MEM_NAME_SIZE`.

Miscellaneous Settings

Table 10–9 shows the miscellaneous settings.

Table 10–9. Miscellaneous Settings (Part 1 of 2)

Setting	Description
Enable argument checking	Maps onto `#define OS_ARG_CHK_EN`.
Enable `uCOS-II` hooks	Maps onto `#define OS_CPU_HOOKS_EN`.
Enable debug variables	Maps onto `#define OS_DEBUG_EN`.
Include code for `OSSchedLock()` and `OSSchedUnlock()`	Maps onto `#define OS_SCHED_LOCK_EN`.
Enable tick stepping feature for `uCOS-View`	Maps onto `#define OS_TICK_STEP_EN`.
Enable statistics task	Maps onto `#define OS_TASK_STAT_EN`.
Check task stacks from statistics task	Maps onto `#define OS_TASK_STAT_STK_CHK_EN`.
Statistics task stack size	Maps onto `#define OS_TASK_STAT_STK_SIZE`.
Idle task stack size	Maps onto `#define OS_TASK_IDLE_STK_SIZE`.

Table 10–9. Miscellaneous Settings (Part 2 of 2)

Setting	Description
Maximum number of event control blocks	Maps onto `#define OS_MAX_EVENTS 60`.
Size of semaphore, mutex, mailbox, or queue name	Maps onto `#define OS_EVENT_NAME_SIZE`.

Task Management Settings

Table 10–10 shows the task management settings.

Table 10–10. Task Management Settings

Setting	Description
Include code for `OSTaskChangePrio()`	Maps onto `#define OS_TASK_CHANGE_PRIO_EN`.
Include code for `OSTaskCreate()`	Maps onto `#define OS_TASK_CREATE_EN`.
Include code for `OSTaskCreateExt()`	Maps onto `#define OS_TASK_CREATE_EXT_EN`.
Include code for `OSTaskDel()`	Maps onto `#define OS_TASK_DEL_EN`.
Include variables in `OS_TCB` for profiling	Maps onto `#define OS_TASK_PROFILE_EN`.
Include code for `OSTaskQuery()`	Maps onto `#define OS_TASK_QUERY_EN`.
Include code for `OSTaskSuspend()` and `OSTaskResume()`	Maps onto `#define OS_TASK_SUSPEND_EN`.
Include code for `OSTaskSwHook()`	Maps onto `#define OS_TASK_SW_HOOK_EN`.
Size of task name	Maps onto `#define OS_TASK_NAME_SIZE`.

Time Management Settings

Table 10–11 shows the time management settings.

Table 10–11. Time Management Settings (Part 1 of 2)

Setting	Description
Include code for `OSTimeDlyHMSM()`	Maps onto `#define OS_TIME_DLY_HMSM_EN`.
Include code `OSTimeDlyResume()`	Maps onto `#define OS_TIME_DLY_RESUME_EN`.

Table 10–11. Time Management Settings (Part 2 of 2)

Setting	Description
Include code for `OSTimeGet()` and `OSTimeSet()`	Maps onto `#define OS_TIME_GET_SET_EN`.
Include code for `OSTimeTickHook()`	Maps onto `#define OS_TIME_TICK_HOOK_EN`.

Referenced Documents

This chapter references the following documents:

- *Overview* chapter of the *Nios II Software Developer's Handbook*
- *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*
- *Developing Device Drivers for the HAL* 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

Document Revision History

Table 10–12 shows the revision history for this document.

Table 10–12. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	● Added documentation for MicroC/OS-II development with the Nios II software build tools. ● Added description of HAL ISR support	
May 2007 v7.1.0	● Added table of contents to Introduction section. ● Added Referenced Documents section.	
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.	

11. Ethernet and the NicheStack TCP/IP Stack - Nios II Edition

NII52013-7.2.0

Overview

The NicheStack® TCP/IP Stack - Nios® II Edition is a small-footprint implementation of the transmission control protocol/Internet protocol (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 component, available through the Nios II Integrated Development Environment (IDE), and the Nios II board support package (BSP) generator, which you can add to your system library or BSP. 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 programming 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

To make the best use of information in this chapter, you need have basic familiarity with these topics:

- Sockets. There are a number of books 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 *Nios II Software Developer's Handbook* for full information on the Nios II EDS.
- The MicroC/OS-II real time operating system (RTOS). To learn about MicroC/OS-II, refer 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. Naturally, the Nios II processor system must also contain an Ethernet interface, or media access controller (MAC). The Altera-provided NicheStack TCP/IP Stack includes driver support for the SMSC lan91c111 MAC/PHY device and Altera Triple Speed Ethernet MegaCore function. The Nios II Embedded Design Suite includes hardware for both MACs, plus an evaluation copy of the Triple Speed Ethernet MegaCore.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. By virtue of 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 on writing an Ethernet device driver, refer to the *Developing Device Drivers for the HAL* 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 C/C++ program using the Nios II IDE. 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*>.

The directory format of the stack tries to maintain the original code as much as possible under the <*iniche path*>**/src/downloads** directory for ease of 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 at **www.altera.com/literature/lit-nio2.jsp**, under **Other Related Documentation**.

Altera's implementation of the NicheStack TCP/IP Stack is based on version 3.0 of the protocol stack, with wrappers placed around the code to integrate it to the HAL system library.

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 *www.altera.com/nichestack*.

You can license other protocol stacks directly from InterNiche. Refer to the InterNiche website, *www.interniche.com*, for details.

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 on products available from third party providers, visit Altera's Embedded Software Partners page at: **www.altera.com/products/software/partners/embedded/emb-partners.html**.

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 system library must be based on MicroC/OS-II
- The MicroC/OS-II RTOS must be configured to have the following enabled:
  - 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. These tasks run continuously in addition to the tasks that your program creates.

1. 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.

2. 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()" on page 11–5.

☞ You can modify the task priority and stack sizes by using `#define` statements in the configuration file **ipport.h**. Additional system tasks might be created if you enable other options in the NicheStack TCP/IP Stack by editing **ipport.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 the your code does not attempt further initialization until 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.

☞ Make sure that your code does not use the sockets interface until `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 on page 11–6.

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 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 through **MAC interface** in the **NicheStack TCP/IP Stack** tab of the **Software Components** dialog box. 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 onchip 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]);
```

Inside the function, you must fill in `mac_addr` with the MAC address.

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()` returns -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 need to 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 for 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*>**/inc/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 this example. In a real application, you might need to return -1 on error.

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;
}
```

`INICHE_DEFAULT_IF`, defined in system.h, identifies the network interface that you defined in SOPC Builder. In the Nios II IDE, you can set `INICHE_DEFAULT_IF` through the **MAC interface** control in the **NicheStack TCP/IP Stack** tab of the **Software Components** dialog box. In the Nios II BSP generator, use 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 setting in the Nios II IDE (with the **Use DHCP to automatically assign IP address** check box), or through the Nios II BSP generator (with the `dhcp_client` setting).

Calling the Sockets Interface

After initializing 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 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);
```

It is 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 on 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 the Nios II IDE

The NicheStack TCP/IP Stack has many options that you can configure using `#define` directives in the file **ipport.h**. The Nios II integrated development environment (IDE) allows you to configure certain options (i.e. modify the `#defines` in **system.h**) without editing source code. The most commonly accessed options are available through the **NicheStack TCP/IP Stack** tab of the **Software Components** dialog box.

There are some less frequently used options that are not accessible through the IDE. If you need to modify these options, you must use the Nios II BSP Generator, or edit the **ipport.h** file manually.

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

You can find **ipport.h** in the **debug/system_description** directory for your system library project.

> If you modify the `ipport.h` file directly, be careful not to select the **Clean Project** build option in the Nios II IDE. Selecting **Clean Project** results in your modified `ipport.h` file being replaced with the starting template version of this file.

The following sections describe the features that you can configure via the Nios II IDE. The IDE provides 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 transmission control protocol (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	When on, the component use DHCP to acquire an IP address. When off, you must assign a static IP address.
Enable statistics	When this option is turned on, the stack keeps counters of packets received, errors, etc. The counters are defined in `mib` structures defined in various header files in directory *<iniche path>***/src/downloads/30src/h.** For details on `mib` 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. See "Known Limitations" on page 11–12.

IP Options

Table 11–4 shows the IP options.

Table 11–3. IP Options

Option	Description
Forward IP packets	When there is more than one network interface, if this option is turned on, and the IP stack for one interface receives packets not addressed to it, it forwards the packet out of the other interface. See "Known Limitations" on page 11–12.
Reassemble IP packet fragments	If this option is turned on, the NicheStack TCP/IP Stack reassembles IP packet fragments into full IP packets. Otherwise, it discards IP packet fragments. This topic is explained in *Unix Network Programming* by Richard Stevens.

TCP Options

Table 11–4 shows the TCP options, which are only available with 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, see 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, see the NicheStack TCP/IP Stack reference manual, available at **www.altera.com/literature/lit-nio2.jsp**, under **Other Related Documentation**.

Known Limitations

Although the NicheStack code contains features intended to support multiple network interfaces, these features are not tested. See 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:

- *Developing Device Drivers for the HAL* chapter of the *Nios II Software Developer's Handbook*
- *NicheStack TCP/IP Stack documentation* available at *Literature: Nios II Processor, Other Related Documentation*
- *Using the NicheStack TCP/IP Stack - Nios II Edition Tutorial*
- *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*

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
October 2007 v7.2.0	No change from previous release.	
May 2007 v7.1.0	● Chapter 10 was formerly chapter 9. ● Minor clarifications added to content. ● Added table of contents to Overview section. ● Added Referenced Documents section.	
March 2007 v7.0.0	No change from previous release.	
November 2006 v6.1.0	Initial Release.	

Section IV. Appendices

This section provides appendix information.

This section includes the following chapters:

12. HAL API Reference

NII52010-7.2.0

Introduction

This chapter provides an alphabetically ordered list of all the functions within the hardware abstraction layer (HAL) application programming interface (API). Each function is listed with its C prototype and a short description. Indication is also given as to whether the function is thread safe when running in a multi-threaded environment, and whether it can be called from an interrupt service routine (ISR).

This appendix only lists the functionality provided by the HAL. You should be aware that the complete newlib API is also available from within HAL systems. For example, newlib provides `printf()`, and other standard I/O functions, which are not described here.

For more details of the newlib API, refer to the newlib documentation. On the Windows **Start** menu, click **Programs**, **Altera**, **Nios II** <*version*>, **Nios II Documentation**.

This chapter contains the following sections:

- "HAL API Functions" on page 12–1

HAL API Functions

The HAL API functions are as shown below.

_exit()

Prototype: `void _exit (int exit_code)`

Commonly called by: Newlib C library

Thread-safe: Yes.

Available from ISR: No.

Include: <**unistd.h**>

Description: The newlib `exit()` function calls the `_exit()` function to terminate the current process. Typically, when `main()` completes. Because there is only a single process within HAL systems, the HAL implementation blocks forever.

Note that interrupts are not disabled, so ISRs continue to execute.

The input argument, `exit_code`, is ignored.

Return: –

See also: Newlib documentation. On the Windows **Start** menu, click **Programs**, **Altera**, **Nios II** *<version>*, **Nios II Documentation**.

_rename()

Prototype:	`int _rename(char *existing, char* new)`
Commonly called by:	Newlib C library
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	<**stdio.h**>
Description:	The `_rename()` function is provided for newlib compatibility.
Return:	It always fails with return code –1, and with `errno` set to `ENOSYS`.
See also:	Newlib documentation. On the Windows **Start** menu, click **Programs**, **Altera**, **Nios II** <*version*>, **Nios II Documentation**.

alt_alarm_start()

Prototype:

```
int alt_alarm_start (alt_alarm* alarm,
                     alt_u32    nticks,
                    alt_u32 (*callback) (void* context),
                     void*      context)
```

Commonly called by: C/C++ programs
Device drivers

Thread-safe: Yes.

Available from ISR: Yes.

Include: <**sys/alt_alarm.h**>

Description: The `alt_alarm_start()` function schedules an alarm callback. See the "Alarms" section of the *Developing Programs using the HAL* chapter of the *Nios® II Software Developer's Handbook*. The input argument, `ntick`, is the number of system clock ticks that elapse until the call to the `callback` function. The input argument `context` is passed as the input argument to the `callback` function, when the callback occurs.

The input `alarm` is a pointer to a structure that represents this alarm. You must create it, and it must have a lifetime that is at least as long as that of the alarm. However, you are not responsible for initializing the contents of the structure pointed to by `alarm`. This action is done by the call to `alt_alarm_start()`.

Return: The return value for `alt_alarm_start()` is zero upon success, and negative otherwise. This function fails if there is no system clock available.

See also:

```
alt_alarm_stop()
alt_nticks()
alt_sysclk_init()
alt_tick()
alt_ticks_per_second()
gettimeofday()
settimeofday()
times()
usleep()
```

alt_alarm_stop()

Prototype: `void alt_alarm_stop (alt_alarm* alarm)`

Commonly called by: C/C++ programs
Device drivers

Thread-safe: Yes.

Available from ISR: Yes.

Include: <**sys/alt_alarm.h**>

Description: You can call the `alt_alarm_stop()` function to cancel an alarm previously registered by a call to `alt_alarm_start()`. The input argument is a pointer to the alarm structure in the previous call to `alt_alarm_start()`.

Upon return the alarm is canceled, if it is still active.

Return: –

See also:

```
alt_alarm_start()
alt_nticks()
alt_sysclk_init()
alt_tick()
alt_ticks_per_second()
gettimeofday()
settimeofday()
times()
usleep()
```

alt_dcache_flush()

Prototype:	`void alt_dcache_flush (void* start, alt_u32 len)`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	<**sys/alt_cache.h**>
Description:	The `alt_dcache_flush()` function flushes (i.e. writes back dirty data and then invalidates) the data cache for a memory region of length `len` bytes, starting at address `start`. In processors without data caches, it has no effect.
Return:	–

See also:	`alt_dcache_flush_all()`
	`alt_icache_flush()`
	`alt_icache_flush_all()`
	`alt_remap_cached()`
	`alt_remap_uncached()`
	`alt_uncached_free()`
	`alt_uncached_malloc()`

alt_dcache_flush_all()

Prototype:	`void alt_dcache_flush_all (void)`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	<**sys/alt_cache.h**>
Description:	The `alt_dcache_flush_all()` function flushes, i.e., writes back dirty data and then invalidates, the entire contents of the data cache. In processors without data caches, it has no effect.
Return:	–

See also:	`alt_dcache_flush()`
	`alt_icache_flush()`
	`alt_icache_flush_all()`
	`alt_remap_cached()`
	`alt_remap_uncached()`
	`alt_uncached_free()`
	`alt_uncached_malloc()`

alt_dev_reg()

Prototype:	`int alt_dev_reg(alt_dev* dev)`
Commonly called by:	Device drivers
Thread-safe:	No.
Available from ISR:	No.
Include:	<**sys/alt_dev.h**>
Description:	The `alt_dev_reg()` function registers a device with the system. Once registered you can access a device using the standard I/O functions. See the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*. The system behavior is undefined in the event that a device is registered with a name that conflicts with an existing device or file system. The `alt_dev_reg()` function is not thread safe in the sense that there should be no other thread using the device list at the time that `alt_dev_reg()` is called. In practice `alt_dev_reg()` should only be called while operating in a single threaded mode. The expectation is that it is only called by the device initialization functions invoked by `alt_sys_init()`, which in turn should only be called by the single threaded C startup code.
Return:	A return value of zero indicates success. A negative return value indicates failure.
See also:	`alt_fs_reg()`

alt_dma_rxchan_close()

Prototype: `int alt_dma_rxchan_close (alt_dma_rxchan rxchan)`

Commonly called by: C/C++ programs
Device drivers

Thread-safe: Yes.

Available from ISR: No.

Include: <**sys/alt_dma.h**>

Description: The `alt_dma_rxchan_close()` function notifies the system that the application has finished with the direct memory access (DMA) receive channel, `rxchan`. The current implementation always succeeds.

Return: The return value is zero upon success and negative otherwise.

See also:

```
alt_dma_rxchan_depth()
alt_dma_rxchan_ioctl()
alt_dma_rxchan_open()
alt_dma_rxchan_prepare()
alt_dma_rxchan_reg()
alt_dma_txchan_close()
alt_dma_txchan_ioctl()
alt_dma_txchan_open()
alt_dma_txchan_reg()
alt_dma_txchan_send()
alt_dma_txchan_space()
```

alt_dma_rxchan_depth()

Prototype: `alt_u32 alt_dma_rxchan_depth(alt_dma_rxchan dma)`

Commonly called by: C/C++ programs
Device drivers

Thread-safe: Yes.

Available from ISR: No.

Include: <**sys/alt_dma.h**>

Description: The `alt_dma_rxchan_depth()` function returns the maximum number of receive requests that can be posted to the specified DMA transmit channel, `dma`.

Whether this function is thread-safe, or can be called from an ISR is dependent on the underlying device driver. In general it should be assumed this is not the case.

Return: Returns the maximum number of receive requests that can be posted.

See also:

```
alt_dma_rxchan_close()
alt_dma_rxchan_ioctl()
alt_dma_rxchan_open()
alt_dma_rxchan_prepare()
alt_dma_rxchan_reg()
alt_dma_txchan_close()
alt_dma_txchan_ioctl()
alt_dma_txchan_open()
alt_dma_txchan_reg()
alt_dma_txchan_send()
alt_dma_txchan_space()
```

alt_dma_rxchan_ioctl()

Prototype:

```
int alt_dma_rxchan_ioctl (alt_dma_rxchan dma,
                          int            req,
                          void*          arg)
```

Commonly called by: C/C++ programs
Device drivers

Thread-safe: See description.

Available from ISR: See description.

Include: <**sys/alt_dma.h**>

Description: The `alt_dma_rxchan_ioctl()` function performs DMA I/O operations on the DMA receive channel, `dma`. The I/O operations are device specific. For example, some DMA drivers support options to control the width of the transfer operations. The input argument, `req`, is an enumeration of the requested operation; `arg` is an additional argument for the request. The interpretation of `arg` is request dependent.

Table 12–1 shows generic requests defined in <**sys/alt_dma.h**>, which a DMA device might support.

Whether a call to `alt_dma_rxchan_ioctl` is thread safe, or can be called from an ISR, is device dependent. In general it should be assumed it is not the case.

The `alt_dma_rxchan_ioctl()` function should not be called while DMA transfers are pending, otherwise unpredictable behavior might result.

For device-specific information about the Altera® DMA controller core, see the *DMA Controller Core with Avalon Interface* chapter in volume 5 of the *Quartus® II Handbook*.

Return: A negative return value indicates failure, otherwise the interpretation of the return value is request specific.

See also:

```
alt_dma_rxchan_close()
alt_dma_rxchan_depth()
alt_dma_rxchan_open()
alt_dma_rxchan_prepare()
alt_dma_rxchan_reg()
alt_dma_txchan_close()
alt_dma_txchan_ioctl()
alt_dma_txchan_open()
alt_dma_txchan_reg()
alt_dma_txchan_send()
alt_dma_txchan_space()
```

Table 12–1. Generic Requests

Request	Meaning
`ALT_DMA_SET_MODE_8`	Transfer data in units of 8 bits. The value of `arg` is ignored.
`ALT_DMA_SET_MODE_16`	Transfer data in units of 16 bits. The value of `arg` is ignored.
`ALT_DMA_SET_MODE_32`	Transfer data in units of 32 bits. The value of `arg` is ignored.
`ALT_DMA_SET_MODE_64`	Transfer data in units of 64 bits. The value of `arg` is ignored.
`ALT_DMA_SET_MODE_128`	Transfer data in units of 128 bits. The value of `arg` is ignored.
`ALT_DMA_GET_MODE`	Return the transfer width. The value of `arg` is ignored.
`ALT_DMA_TX_ONLY_ON` *(1)*	The `ALT_DMA_TX_ONLY_ON` request causes a DMA channel to operate in a mode where only the transmitter is under software control. The other side writes continuously from a single location. The address to write to is the argument to this request.
`ALT_DMA_TX_ONLY_OFF` *(1)*	Return to the default mode where both the receive and transmit sides of the DMA can be under software control.
`ALT_DMA_RX_ONLY_ON` *(1)*	The `ALT_DMA_RX_ONLY_ON` request causes a DMA channel to operate in a mode where only the receiver is under software control. The other side reads continuously from a single location. The address to read is the argument to this request.
`ALT_DMA_RX_ONLY_OFF` *(1)*	Return to the default mode where both the receive and transmit sides of the DMA can be under software control.

Notes to Table 12–1:

(1) These macro names changed in version 1.1 of the Nios II Embedded Design Suite (EDS). The old names (`ALT_DMA_TX_STREAM_ON`, `ALT_DMA_TX_STREAM_OFF`, `ALT_DMA_RX_STREAM_ON`, and `ALT_DMA_RX_STREAM_OFF`) are still valid, but new designs should use the new names.

alt_dma_rxchan_open()

Prototype:	`alt_dma_rxchan alt_dma_rxchan_open (const char* name)`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**sys/alt_dma.h**>
Description:	The `alt_dma_rxchan_open()` function obtains an `alt_dma_rxchan` descriptor for a DMA receive channel. The input argument, `name`, is the name of the associated physical device, e.g., `/dev/dma_0`.
Return:	The return value is null on failure and non-null otherwise. If there is an error, `errno` is set to `ENODEV`.

See also:	`alt_dma_rxchan_close()`
	`alt_dma_rxchan_depth()`
	`alt_dma_rxchan_ioctl()`
	`alt_dma_rxchan_prepare()`
	`alt_dma_rxchan_reg()`
	`alt_dma_txchan_close()`
	`alt_dma_txchan_ioctl()`
	`alt_dma_txchan_open()`
	`alt_dma_txchan_reg()`
	`alt_dma_txchan_send()`
	`alt_dma_txchan_space()`

alt_dma_rxchan_prepare()

Prototype:

```
int alt_dma_rxchan_prepare (alt_dma_rxchan   dma,
                            void*            data,
                            alt_u32          length,
                            alt_rxchan_done* done,
                            void*            handle)
```

Commonly called by: C/C++ programs
Device drivers

Thread-safe: See description.

Available from ISR: See description.

Include: **<sys/alt_dma.h>**

Description: The `alt_dma_rxchan_prepare()` posts a receive request to a DMA receive channel. The input arguments are: `dma`, the channel to use; `data`, a pointer to the location that data is to be received to; `length`, the maximum length of the data to receive in bytes; `done`, callback function that is called once the data is received; `handle`, an opaque value passed to `done`.

Whether this function is thread-safe, or can be called from an ISR is dependent on the underlying device driver. In general it should be assumed it is not the case.

Return: The return value is negative if the request cannot be posted, and zero otherwise.

See also:

```
alt_dma_rxchan_close()
alt_dma_rxchan_depth()
alt_dma_rxchan_ioctl()
alt_dma_rxchan_open()
alt_dma_rxchan_reg()
alt_dma_txchan_close()
alt_dma_txchan_ioctl()
alt_dma_txchan_open()
alt_dma_txchan_reg()
alt_dma_txchan_send()
alt_dma_txchan_space()
```

alt_dma_rxchan_reg()

Prototype: `int alt_dma_rxchan_reg (alt_dma_rxchan_dev* dev)`

Commonly called by: C/C++ programs
Device drivers

Thread-safe: No.

Available from ISR: No.

Include: <**sys/alt_dma_dev.h**>

Description: The `alt_dma_rxchan_reg()` function registers a DMA receive channel with the system. Once registered a device can be accessed using the functions described in the "DMA Receive Channels" section of the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*.

System behavior is undefined in the event that a channel is registered with a name that conflicts with an existing channel.

The `alt_dma_rxchan_reg()` function is not thread safe if other threads are using the channel list at the time that `alt_dma_rxchan_reg()` is called. In practice, only call `alt_dma_rxchan_reg()` while operating in a single threaded mode. Only call it by the device initialization functions invoked by `alt_sys_init()`, which in turn should only be called by the single threaded C startup code.

Return: A return value of zero indicates success. A negative return value indicates failure.

See also:
`alt_dma_rxchan_close()`
`alt_dma_rxchan_depth()`
`alt_dma_rxchan_ioctl()`
`alt_dma_rxchan_open()`
`alt_dma_rxchan_prepare()`
`alt_dma_txchan_close()`
`alt_dma_txchan_ioctl()`
`alt_dma_txchan_open()`
`alt_dma_txchan_reg()`
`alt_dma_txchan_send()`
`alt_dma_txchan_space()`

alt_dma_txchan_close()

Prototype:	`int alt_dma_txchan_close (alt_dma_txchan txchan)`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**sys/alt_dma.h**>
Description:	The `alt_dma_txchan_close` function notifies the system that the application has finished with the DMA transmit channel, `txchan`. The current implementation always succeeds.
Return:	The return value is zero upon success and negative otherwise.

See also:	`alt_dma_rxchan_close()`
	`alt_dma_rxchan_depth()`
	`alt_dma_rxchan_ioctl()`
	`alt_dma_rxchan_open()`
	`alt_dma_rxchan_prepare()`
	`alt_dma_rxchan_reg()`
	`alt_dma_txchan_ioctl()`
	`alt_dma_txchan_open()`
	`alt_dma_txchan_reg()`
	`alt_dma_txchan_send()`
	`alt_dma_txchan_space()`

alt_dma_txchan_ioctl()

Prototype:

```
int alt_dma_txchan_ioctl (alt_dma_txchan dma,
                          int            req,
                          void*          arg)
```

Commonly called by: C/C++ programs
Device drivers

Thread-safe: See description.

Available from ISR: See description.

Include: <**sys/alt_dma.h**>

Description: The `alt_dma_txchan_ioctl()` function performs device specific I/O operations on the DMA transmit channel, `dma`. For example, some drivers support options to control the width of the transfer operations. The input argument, `req`, is an enumeration of the requested operation; `arg` is an additional argument for the request. The interpretation of `arg` is request dependent.

See Table 12–1 for the generic requests a device might support.

Whether a call to `alt_dma_txchan_ioctl()` is thread safe, or can be called from an ISR, is device dependent. In general it should be assumed this is not the case.

The `alt_dma_rxchan_ioctl()` function should not be called while DMA transfers are pending, otherwise unpredictable behavior might result.

Return: A negative return value indicates failure; otherwise the interpretation of the return value is request specific.

See also:

```
alt_dma_rxchan_close()
alt_dma_rxchan_depth()
alt_dma_rxchan_ioctl()
alt_dma_rxchan_open()
alt_dma_rxchan_prepare()
alt_dma_rxchan_reg()
alt_dma_txchan_close()
alt_dma_txchan_open()
alt_dma_txchan_reg()
alt_dma_txchan_send()
alt_dma_txchan_space()
```

alt_dma_txchan_open()

Prototype: `alt_dma_txchan alt_dma_txchan_open (const char* name)`

Commonly called by: C/C++ programs
Device drivers

Thread-safe: Yes.

Available from ISR: No.

Include: **<sys/alt_dma.h>**

Description: The `alt_dma_txchan_open()` function obtains an `alt_dma_txchan()` descriptor for a DMA transmit channel. The input argument, `name`, is the name of the associated physical device, e.g., `/dev/dma_0`.

Return: The return value is null on failure and non-null otherwise. If there is an error, `errno` is set to `ENODEV`.

See also:
`alt_dma_rxchan_close()`
`alt_dma_rxchan_depth()`
`alt_dma_rxchan_ioctl()`
`alt_dma_rxchan_open()`
`alt_dma_rxchan_prepare()`
`alt_dma_rxchan_reg()`
`alt_dma_txchan_close()`
`alt_dma_txchan_ioctl()`
`alt_dma_txchan_reg()`
`alt_dma_txchan_send()`
`alt_dma_txchan_space()`

alt_dma_txchan_reg()

Prototype: `int alt_dma_txchan_reg (alt_dma_txchan_dev* dev)`

Commonly called by: C/C++ programs
Device drivers

Thread-safe: No.

Available from ISR: No.

Include: <**sys/alt_dma_dev.h**>

Description: The `alt_dma_txchan_reg()` function registers a DMA transmit channel with the system. Once registered, a device can be accessed using the functions described in the DMA Transmit Channels section of the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*.

System behavior is undefined in the event that a channel is registered with a name that conflicts with an existing channel.

The `alt_dma_txchan_reg()` function is not thread safe if other threads are using the channel list at the time that `alt_dma_txchan_reg()` is called. Only call `alt_dma_txchan_reg()` while operating in a single-threaded mode. Only call it by the device initialization functions invoked by `alt_sys_init()`, which in turn should only be called by the single threaded C startup code.

Return: A return value of zero indicates success. A negative return value indicates failure.

See also:

```
alt_dma_rxchan_close()
alt_dma_rxchan_depth()
alt_dma_rxchan_ioctl()
alt_dma_rxchan_open()
alt_dma_rxchan_prepare()
alt_dma_rxchan_reg()
alt_dma_txchan_close()
alt_dma_txchan_ioctl()
alt_dma_txchan_open()
alt_dma_txchan_send()
alt_dma_txchan_space()
```

alt_dma_txchan_send()

Prototype:

```
int alt_dma_txchan_send (alt_dma_txchan   dma,
                         const void*      from,
                         alt_u32          length,
                         alt_txchan_done* done,
                         void*            handle)
```

Commonly called by: C/C++ programs
Device drivers

Thread-safe: See description.

Available from ISR: See description.

Include: <**sys/alt_dma.h**>

Description: The `alt_dma_txchan_send()` function posts a transmit request to a DMA transmit channel. The input arguments are: `dma`, the channel to use; from, a pointer to the start of the data to send; `length`, the length of the data to send in bytes; `done`, a callback function that is called once the data is sent; and `handle`, an opaque value passed to done.

Whether this function is thread-safe, or can be called from an ISR is dependent on the underlying device driver. In general it should be assumed this is not the case.

Return: The return value is negative if the request cannot be posted, and zero otherwise.

See also:

```
alt_dma_rxchan_close()
alt_dma_rxchan_depth()
alt_dma_rxchan_ioctl()
alt_dma_rxchan_open()
alt_dma_rxchan_prepare()
alt_dma_rxchan_reg()
alt_dma_txchan_close()
alt_dma_txchan_ioctl()
alt_dma_txchan_open()
alt_dma_txchan_reg()
alt_dma_txchan_space()
```

alt_dma_txchan_space()

Prototype:	`int alt_dma_txchan_space (alt_dma_txchan dma)`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	See description.
Available from ISR:	See description.
Include:	<**sys/alt_dma.h**>
Description:	The `alt_dma_txchan_space()` function returns the number of transmit requests that can be posted to the specified DMA transmit channel, `dma`. A negative value indicates that the value cannot be determined. Whether this function is thread-safe, or can be called from an ISR is dependent on the underlying device driver. In general it should be assumed this is not the case.
Return:	Returns the number of transmit requests that can be posted.

See also:	`alt_dma_rxchan_close()`
	`alt_dma_rxchan_depth()`
	`alt_dma_rxchan_ioctl()`
	`alt_dma_rxchan_open()`
	`alt_dma_rxchan_prepare()`
	`alt_dma_rxchan_reg()`
	`alt_dma_txchan_close()`
	`alt_dma_txchan_ioctl()`
	`alt_dma_txchan_open()`
	`alt_dma_txchan_reg()`
	`alt_dma_txchan_send()`

alt_erase_flash_block()

Prototype:

```
int alt_erase_flash_block(alt_flash_fd* fd,
                          int           offset,
                          int           length)
```

Commonly called by: C/C++ programs
Device drivers

Thread-safe: No.

Available from ISR: No.

Include: <**sys/alt_flash.h**>

Description: The `alt_erase_flash_block()` function erases an individual flash erase block. The parameter `fd` specifies the flash device; `offset` is the offset within the flash of the block to erase; `length` is the size of the block to erase. No error checking is performed to check that this is a valid block, or that the length is correct. See the "Fine-Grained Flash Access" section of the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*.

Only call the `alt_erase_flash_block()` function when operating in single threaded mode.

The only valid values for the `fd` parameter are those returned from the `alt_flash_open_dev` function. If any other value is passed the behavior of this function is undefined.

Return: A return value of zero indicates success. A negative return value indicates failure.

See also:

```
alt_flash_close_dev()
alt_flash_open_dev()
alt_get_flash_info()
alt_read_flash()
alt_write_flash()
alt_write_flash_block()
```

alt_flash_close_dev()

Prototype: `void alt_flash_close_dev(alt_flash_fd* fd)`

Commonly called by: C/C++ programs
Device drivers

Thread-safe: No.

Available from ISR: No.

Include: <**sys/alt_flash.h**>

Description: The `alt_flash_close_dev()` function closes a flash device. All subsequent calls to `alt_write_flash()`, `alt_read_flash()`, `alt_get_flash_info()`, `alt_erase_flash_block()`, or `alt_write_flash_block()` for this flash device fail.

Only call the `alt_flash_close_dev()` function when operating in single-threaded mode.

The only valid values for the `fd` parameter are those returned from the `alt_flash_open_dev` function. If any other value is passed the behavior of this function is undefined.

Return: –

See also:

```
alt_erase_flash_block()
alt_flash_open_dev()
alt_get_flash_info()
alt_read_flash()
alt_write_flash()
alt_write_flash_block()
```

alt_flash_open_dev()

Prototype: `alt_flash_fd* alt_flash_open_dev(const char* name)`

Commonly called by: C/C++ programs
Device drivers

Thread-safe: No.

Available from ISR: No.

Include: <**sys/alt_flash.h**>

Description: The `alt_flash_open_dev()` function opens a flash device. Once opened a flash device can be written to using `alt_write_flash()`, read from using `alt_read_flash()`, or you can control individual flash blocks using the `alt_get_flash_info()`, `alt_erase_flash_block()`, or `alt_write_flash_block()` function.

Only call the `alt_flash_open_dev` function when operating in single threaded mode.

Return: A return value of zero indicates failure. Any other value is success.

See also:

```
alt_erase_flash_block()
alt_flash_close_dev()
alt_get_flash_info()
alt_read_flash()
alt_write_flash()
alt_write_flash_block()
```

alt_fs_reg()

Prototype: `int alt_fs_reg (alt_dev* dev)`

Commonly called by: Device drivers

Thread-safe: No.

Available from ISR: No.

Include: <**sys/alt_dev.h**>

Description: The `alt_fs_reg()` function registers a file system with the HAL. Once registered, a file system can be accessed using the standard I/O functions. See the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*.

System behavior is undefined in the event that a file system is registered with a name that conflicts with an existing device or file system.

`alt_fs_reg()` is not thread safe if other threads are using the device list at the time that `alt_fs_reg()` is called. In practice `alt_fs_reg()` should only be called while operating in a single threaded mode. The expectation is that it is only called by the device initialization functions invoked by `alt_sys_init()`, which in turn should only be called by the single threaded C startup code.

Return: A return value of zero indicates success. A negative return value indicates failure.

See also: `alt_dev_reg()`

alt_get_flash_info()

Prototype:

```
int alt_get_flash_info(alt_flash_fd*   fd,
                       flash_region** info,
                       int*          number_of_regions)
```

Commonly called by: C/C++ programs
Device drivers

Thread-safe: No.

Available from ISR: No.

Include: <**sys/alt_flash.h**>

Description: The `alt_get_flash_info()` function gets the details of the erase region of a flash part. The flash part is specified by the descriptor `fd`, a pointer to the start of the `flash_region` structures is returned in the `info` parameter, and the number of flash regions are returned in number of regions.

Only call this function when operating in single threaded mode.

The only valid values for the `fd` parameter are those returned from the `alt_flash_open_dev` function. If any other value is passed the behavior of this function is undefined.

Return: A return value of zero indicates success. A negative return value indicates failure.

See also:

`alt_erase_flash_block()`
`alt_flash_close_dev()`
`alt_flash_open_dev()`
`alt_read_flash()`
`alt_write_flash()`
`alt_write_flash_block()`

alt_icache_flush()

Prototype:	`void alt_icache_flush (void* start, alt_u32 len)`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	<**sys/alt_cache.h**>
Description:	The `alt_icache_flush()` function invalidates the instruction cache for a memory region of length `len` bytes, starting at address `start`. In processors without instruction caches, it has no effect.
Return:	–

See also:	`alt_dcache_flush()`
	`alt_dcache_flush_all()`
	`alt_icache_flush_all()`
	`alt_remap_cached()`
	`alt_remap_uncached()`
	`alt_uncached_free()`
	`alt_uncached_malloc()`

alt_icache_flush_all()

Prototype: `void alt_icache_flush_all (void)`

Commonly called by: C/C++ programs
Device drivers

Thread-safe: Yes.

Available from ISR: Yes.

Include: <**sys/alt_cache.h**>

Description: The `alt_icache_flush_all()` function invalidates the entire contents of the instruction cache.

In processors without instruction caches, it has no effect.

Return: –

See also:

```
alt_dcache_flush()
alt_dcache_flush_all()
alt_icache_flush()
alt_remap_cached()
alt_remap_uncached()
alt_uncached_free()
alt_uncached_malloc()
```

alt_irq_disable()

Prototype: `int alt_irq_disable (alt_u32 id)`

Commonly called by: C/C++ programs
Device drivers

Thread-safe: Yes.

Available from ISR: No.

Include: <**sys/alt_irq.h**>

Description: The `alt_irq_disable()` function disables a single interrupt.

Return: The return value is zero.

See also:
`alt_irq_disable_all()`
`alt_irq_enable()`
`alt_irq_enable_all()`
`alt_irq_enabled()`
`alt_irq_register()`

alt_irq_disable_all()

Prototype:	`alt_irq_context alt_irq_disable_all (void)`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**sys/alt_irq.h**>
Description:	The `alt_irq_disable_all()` function disables all interrupts.
Return:	Pass the return value as the input argument to a subsequent call to `alt_irq_enable_all()`.

See also:	`alt_irq_disable()`
	`alt_irq_enable()`
	`alt_irq_enable_all()`
	`alt_irq_enabled()`
	`alt_irq_register()`

alt_irq_enable()

Prototype: `int alt_irq_enable (alt_u32 id)`

Commonly called by: C/C++ programs
Device drivers

Thread-safe: Yes.

Available from ISR: No.

Include: <**sys/alt_irq.h**>

Description: The `alt_irq_enable()` function enables a single interrupt.

Return: The return value is zero.

See also:
`alt_irq_disable()`
`alt_irq_disable_all()`
`alt_irq_enable_all()`
`alt_irq_enabled()`
`alt_irq_register()`

alt_irq_enable_all()

Prototype: `void alt_irq_enable_all (alt_irq_context context)`

Commonly called by: C/C++ programs
Device drivers

Thread-safe: Yes.

Available from ISR: Yes.

Include: <**sys/alt_irq.h**>

Description: The `alt_irq_enable_all()` function enables all interrupts that were previously disabled by `alt_irq_disable_all()`. The input argument, `context`, is the value returned by a previous call to `alt_irq_disable_all()`. Using `context` allows nested calls to `alt_irq_disable_all()` and `alt_irq_enable_all()`. As a result, `alt_irq_enable_all()` does not necessarily enable all interrupts, such as interrupts explicitly disabled by `alt_irq_disable()`.

Return: –

See also:
`alt_irq_disable()`
`alt_irq_disable_all()`
`alt_irq_enable()`
`alt_irq_enabled()`
`alt_irq_register()`

alt_irq_enabled()

Prototype: `int alt_irq_enabled (void)`

Commonly called by: Device drivers

Thread-safe: Yes.

Available from ISR: Yes.

Include: <**sys/alt_irq.h**>

Description: The `alt_irq_enabled()` function.

Return: Returns zero if interrupts are disabled, and non-zero otherwise.

See also:

```
alt_irq_disable()
alt_irq_disable_all()
alt_irq_enable()
alt_irq_enable_all()
alt_irq_register()
```

alt_irq_register()

Prototype:

```
int alt_irq_register (alt_u32 id,
                      void*   context,
                      void    (*isr)(void*, alt_u32))
```

Commonly called by: Device drivers

Thread-safe: Yes.

Available from ISR: No.

Include: <**sys/alt_irq.h**>

Description: The alt_irq_register() function registers an ISR. If the function is successful, the requested interrupt is enabled upon return.
The input argument, id is the interrupt to enable, isr is the function that is called when the interrupt is active, context and id are the two input arguments to isr.

Calls to alt_irq_register() replace previously registered handlers for interrupt id.

If irq_handler is set to null, the interrupt is disabled.

Return: The alt_irq_register() function returns zero if successful, or non-zero otherwise.

See also:

alt_irq_disable()
alt_irq_disable_all()
alt_irq_enable()
alt_irq_enable_all()
alt_irq_enabled()

alt_llist_insert()

Prototype:	`void alt_llist_insert(alt_llist* list,`
	`                      alt_llist* entry)`

Commonly called by:	C/C++ programs Device drivers
Thread-safe:	No.
Available from ISR:	Yes.
Include:	<**sys/alt_llist.h**>
Description:	The `alt_llist_insert()` function inserts the doubly linked list entry `entry` into the list `list`. This operation is not re-entrant. For example, if a list can be manipulated from different threads, or from within both application code and an ISR, some mechanism is required to protect access to the list. Interrupts can be locked, or in MicroC/OS-II, a `mutex` can be used.
Return:	–
See also:	`alt_llist_remove()`

alt_llist_remove()

Prototype: `void alt_llist_remove(alt_llist* entry)`

Commonly called by: C/C++ programs
Device drivers

Thread-safe: No.

Available from ISR: Yes.

Include: <**sys/alt_llist.h**>

Description: The `alt_llist_remove()` function removes the doubly linked list entry `entry` from the list it is currently a member of. This operation is not re-entrant. For example if a list can be manipulated from different threads, or from within both application code and an ISR, some mechanism is required to protect access to the list. Interrupts can be locked, or in MicroC/OS-II, a `mutex` can be used.

Return: –

See also: `alt_llist_insert()`

alt_load_section()

Prototype:

```
void alt_load_section(alt_u32* from,
                      alt_u32* to,
                      alt_u32* end)
```

Commonly called by: C/C++ programs

Thread-safe: No.

Available from ISR: No.

Include: <**sys/alt_load.h**>

Description: When operating in run-from-flash mode, the sections .exceptions, .rodata, and .rwdata are automatically loaded from the boot device to RAM at boot time. However, if there are any additional sections that require loading, the alt_load_section() function loads them manually before these sections are used.

The input argument from is the start address in the boot device of the section; to is the start address in RAM of the section, and end is the end address in RAM of the section.

To load one of the additional memory sections provided by the default linker script, use the macro ALT_LOAD_SECTION_BY_NAME rather than calling alt_load_section() directly. For example, to load the section .onchip_ram, use the following code:

```
ALT_LOAD_SECTION_BY_NAME(onchip_ram);
```

The leading '.' is omitted in the section name. This macro is defined in the header **sys/alt_load.h**.

Return: –

See also: –

alt_nticks()

Prototype:	`alt_u32 alt_nticks (void)`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	<**sys/alt_alarm.h**>
Description:	The `alt_nticks()` function.
Return:	Returns the number of elapsed system clock tick since reset. It returns zero if there is no system clock available.

See also:	`alt_alarm_start()`
	`alt_alarm_stop()`
	`alt_sysclk_init()`
	`alt_tick()`
	`alt_ticks_per_second()`
	`gettimeofday()`
	`settimeofday()`
	`times()`
	`usleep()`

alt_read_flash()

Prototype:

```
int alt_read_flash(alt_flash_fd* fd,
                   int           offset,
                   void*         dest_addr,
                   int           length)
```

Commonly called by: C/C++ programs
Device drivers

Thread-safe: No.

Available from ISR: No.

Include: <**sys/alt_flash.h**>

Description: The `alt_read_flash()` function reads data from flash. Length bytes are read from the flash `fd`, offset bytes from the beginning of the flash and are written to the location `dest_addr`.

Only call this function when operating in single threaded mode.

The only valid values for the `fd` parameter are those returned from the `alt_flash_open_dev` function. If any other value is passed the behavior of this function is undefined.

Return: The return value is zero upon success and non-zero otherwise.

See also:

```
alt_erase_flash_block()
alt_flash_close_dev()
alt_flash_open_dev()
alt_get_flash_info()
alt_write_flash()
alt_write_flash_block()
```

alt_remap_cached()

Prototype:

```
void* alt_remap_cached (volatile void* ptr,
                        alt_u32        len);
```

Commonly called by: C/C++ programs
Device drivers

Thread-safe: Yes.

Available from ISR: No.

Include: <**sys/alt_cache.h**>

Description: The `alt_remap_cached()` function remaps a region of memory for cached access. The memory to map is `len` bytes, starting at address `ptr`.

Processors that do not have a data cache return uncached memory.

Return: The return value for this function is the remapped memory region.

See also:

```
alt_dcache_flush()
alt_dcache_flush_all()
alt_icache_flush()
alt_icache_flush_all()
alt_remap_uncached()
alt_uncached_free()
alt_uncached_malloc()
```

alt_remap_uncached()

Prototype:

```
volatile void* alt_remap_uncached (void*   ptr,
                                   alt_u32 len);
```

Commonly called by: C/C++ programs
Device drivers

Thread-safe: Yes.

Available from ISR: No.

Include: <**sys/alt_cache.h**>

Description: The `alt_remap_uncached()` function remaps a region of memory for uncached access. The memory to map is `len` bytes, starting at address `ptr`.

Processors that do not have a data cache return uncached memory.

Return: The return value for this function is the remapped memory region.

See also:

```
alt_dcache_flush()
alt_dcache_flush_all()
alt_icache_flush()
alt_icache_flush_all()
alt_remap_cached()
alt_uncached_free()
alt_uncached_malloc()
```

alt_sysclk_init()

Prototype:	`int alt_sysclk_init (alt_u32 nticks)`
Commonly called by:	Device drivers
Thread-safe:	No.
Available from ISR:	No.
Include:	<**sys/alt_alarm.h**>
Description:	The `alt_sysclk_init()` function registers the presence of a system clock driver. The input argument is the number of ticks per second at which the system clock is run. The expectation is that this function is only called from within `alt_sys_init()`, i.e., while the system is running in single-threaded mode. Concurrent calls to this function might lead to unpredictable results.
Return:	This function returns zero upon success, otherwise it returns a negative value. The call can fail if a system clock driver has already been registered.

See also:	`alt_alarm_start()`
	`alt_alarm_stop()`
	`alt_nticks()`
	`alt_tick()`
	`alt_ticks_per_second()`
	`gettimeofday()`
	`settimeofday()`
	`times()`
	`usleep()`

alt_tick()

Prototype: void alt_tick (void)

Commonly called by: Device drivers

Thread-safe: No.

Available from ISR: Yes.

Include: <**sys/alt_alarm.h**>

Description: Only the system clock driver should call the alt_tick() function. The driver is responsible for making periodic calls to this function at the rate indicated in the call to alt_sysclk_init(). This function provides notification to the system that a system clock tick has occurred. This function runs as a part of the ISR for the system clock driver.

Return: –

See also:

alt_alarm_start()
alt_alarm_stop()
alt_nticks()
alt_sysclk_init()
alt_ticks_per_second()
gettimeofday()
settimeofday()
times()
usleep()

alt_ticks_per_second()

Prototype:	`alt_u32 alt_ticks_per_second (void)`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	<**sys/alt_alarm.h**>
Description:	The `alt_ticks_per_second()` function returns the number of system clock ticks that elapse per second. If there is no system clock available, the return value is zero.
Return:	Returns the number of system clock ticks that elapse per second.

See also:	`alt_alarm_start()`
	`alt_alarm_stop()`
	`alt_nticks()`
	`alt_sysclk_init()`
	`alt_tick()`
	`gettimeofday()`
	`settimeofday()`
	`times()`
	`usleep()`

alt_timestamp()

Prototype: `alt_u32 alt_timestamp (void)`

Commonly called by: C/C++ programs

Thread-safe: See description.

Available from ISR: See description.

Include: <**sys/alt_timestamp.**h>

Description: The `alt_timestamp()` function returns the current value of the timestamp counter. See the “High Resolution Time Measurement” section of the *Developing Programs using the HAL* chapter of the *Nios II Software Developer’s Handbook*. The implementation of this function is provided by the timestamp driver. Therefore, whether this function is thread-safe and or available at interrupt level is dependent on the underlying driver.

Always call the `alt_timestamp_start()` function before any calls to `alt_timestamp()`. Otherwise the behavior of `alt_timestamp()` is undefined.

Return: Returns the current value of the timestamp counter.

See also: `alt_timestamp_freq()`
`alt_timestamp_start()`

alt_timestamp_freq()

Prototype: `alt_u32 alt_timestamp_freq (void)`

Commonly called by: C/C++ programs

Thread-safe: See description.

Available from ISR: See description.

Include: <**sys/alt_timestamp.h**>

Description: The `alt_timestamp_freq()` function returns the rate at which the timestamp counter increments. See the “High Resolution Time Measurement” section of the *Developing Programs using the HAL* chapter of the *Nios II Software Developer’s Handbook*. The implementation of this function is provided by the timestamp driver. Therefore, whether this function is thread-safe and or available at interrupt level is dependent on the underlying driver.

Return: The returned value is the number of counter ticks per second.

See also: `alt_timestamp()`
`alt_timestamp_start()`

alt_timestamp_start()

Prototype:	`int alt_timestamp_start (void)`
Commonly called by:	C/C++ programs
Thread-safe:	See description.
Available from ISR:	See description.
Include:	<**sys/alt_timestamp.h**>
Description:	The `alt_timestamp_start()` function starts the system timestamp counter. See the “High Resolution Time Measurement” section of the *Developing Programs using the HAL* chapter of the *Nios II Software Developer’s Handbook*. The implementation of this function is provided by the timestamp driver. Therefore, whether this function is thread-safe and or available at interrupt level is dependent on the underlying driver. This function resets the counter to zero, and starts the counter running.
Return:	The return value is zero upon success and non-zero otherwise.

See also:	`alt_timestamp()`
	`alt_timestamp_freq()`

alt_uncached_free()

Prototype: `void alt_uncached_free (volatile void* ptr)`

Commonly called by: C/C++ programs
Device drivers

Thread-safe: Yes.

Available from ISR: No.

Include: <**sys/alt_cache.h**>

Description: The `alt_uncached_free()` function causes the memory pointed to by `ptr` to be de-allocated, i.e., made available for future allocation through a call to `alt_uncached_malloc()`. The input pointer, `ptr`, points to a region of memory previously allocated through a call to `alt_uncached_malloc()`. Behavior is undefined if this is not the case.

Return: –

See also: `alt_dcache_flush()`
`alt_dcache_flush_all()`
`alt_icache_flush()`
`alt_icache_flush_all()`
`alt_remap_cached()`
`alt_remap_uncached()`
`alt_uncached_malloc()`

alt_uncached_malloc()

Prototype: `volatile void* alt_uncached_malloc (size_t size)`

Commonly called by: C/C++ programs
Device drivers

Thread-safe: Yes.

Available from ISR: No.

Include: <**sys/alt_cache.h**>

Description: The `alt_uncached_malloc()` function allocates a region of uncached memory of length `size` bytes. Regions of memory allocated in this way can be released using the `alt_uncached_free()` function.

Processors that do not have a data cache return uncached memory.

Return: If sufficient memory cannot be allocated, this function returns null, otherwise a pointer to the allocated space is returned.

See also:

```
alt_dcache_flush()
alt_dcache_flush_all()
alt_icache_flush()
alt_icache_flush_all()
alt_remap_cached()
alt_remap_uncached()
alt_uncached_free()
```

alt_write_flash()

Prototype:

```
int alt_write_flash(alt_flash_fd* fd,
                    int           offset,
                    const void*   src_addr,
                    int           length)
```

Commonly called by: C/C++ programs
Device drivers

Thread-safe: No.

Available from ISR: No.

Include: <**sys/alt_flash.h**>

Description: The `alt_write_flash()` function writes data into flash. The data to be written is at `src_addr` address, length bytes are written into the flash `fd`, offset bytes from the beginning of the flash.

Only call this function when operating in single threaded mode. This function does not preserve any non written areas of any flash sectors affected by this write. See the "Simple Flash Access" section of the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*.

The only valid values for the `fd` parameter are those returned from the `alt_flash_open_dev` function. If any other value is passed the behavior of this function is undefined.

Return: The return value is zero upon success and non-zero otherwise.

See also:

```
alt_erase_flash_block()
alt_flash_close_dev()
alt_flash_open_dev()
alt_get_flash_info()
alt_read_flash()
alt_write_flash_block()
```

alt_write_flash_block()

Prototype:

```
int alt_write_flash_block(alt_flash_fd* fd,
                          int           block_offset,
                          int           data_offset,
                          const void    *data,
                          int           length)
```

Commonly called by: C/C++ programs
Device drivers

Thread-safe: No.

Available from ISR: No.

Include: <**sys/alt_flash.h**>

Description: The `alt_write_flash_block()` function writes one erase block of flash. The flash device is specified by `fd`, the block offset is the offset within the flash of the start of this block, `data_offset` is the offset within the flash at which to start writing data, `data` is the data to write, `length` is how much data to write. Note, no check is made on any of the parameters. See the "Fine-Grained Flash Access" section of the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*.

Only call this function when operating in single threaded mode.

The only valid values for the `fd` parameter are those returned from the `alt_flash_open_dev` function. If any other value is passed the behavior of this function is undefined.

Return: The return value is zero upon success and non-zero otherwise.

See also:

```
alt_erase_flash_block()
alt_flash_close_dev()
alt_flash_open_dev()
alt_get_flash_info()
alt_read_flash()
alt_write_flash()
```

close()

Prototype: `int close (int fd)`

Commonly called by: C/C++ programs
Newlib C library

Thread-safe: See description.

Available from ISR: No.

Include: <**unistd.h**>

Description: The `close()` function is the standard UNIX style `close()` function, which closes the file descriptor `fd`.

Calls to `close()` are only thread-safe, if the implementation of `close()` provided by the driver that is manipulated is thread-safe.

Valid values for the `fd` parameter are: `stdout`, `stdin` and `stderr`, or any value returned from a call to `open()`.

Return: The return value is zero upon success, and –1 otherwise. If an error occurs, `errno` is set to indicate the cause.

See also:

`fcntl()`
`fstat()`
`ioctl()`
`isatty()`
`lseek()`
`open()`
`read()`
`stat()`
`write()`
Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II *<version>*, **Nios II Documentation**.

execve()

Prototype:	`int execve(const char   *path,`
	`            char *const   argv[],`
	`            char *const   envp[])`

Commonly called by:	Newlib C library
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	<**unistd.h**>
Description:	The `execve()` function is only provided for compatibility with newlib.
Return:	Calls to `execve()` always fail with the return code –1 and `errno` set to `ENOSYS`.
See also:	Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II <*version*>, **Nios II Documentation**.

fcntl()

Prototype:	`int fcntl(int fd, int cmd)`
Commonly called by:	C/C++ programs
Thread-safe:	No.
Available from ISR:	No.
Include:	<**unistd.h**> <**fcntl.h**>
Description:	The `fcntl()` function a limited implementation of the standard `fcntl()` system call, which can change the state of the flags associated with an open file descriptor. Normally these flags are set during the call to `open()`. The main use of this function is to change the state of a device from blocking to non-blocking (for device drivers that support this feature). The input argument `fd` is the file descriptor to be manipulated. `cmd` is the command to execute, which can be either `F_GETFL` (return the current value of the flags) or `F_SETFL` (set the value of the flags).
Return:	If `cmd` is `F_SETFL`, the argument `arg` is the new value of flags, otherwise `arg` is ignored. Only the flags `O_APPEND` and `O_NONBLOCK` can be updated by a call to fcntl(). All other flags remain unchanged. The return value is zero upon success, or –1 otherwise. If `cmd` is `F_GETFL`, the return value is the current value of the flags. If there is an error, –1 is returned. In the event of an error, `errno` is set to indicate the cause.

See also:	`close()`
	`fstat()`
	`ioctl()`
	`isatty()`
	`lseek()`
	`read()`
	`stat()`
	`write()`
	Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II *<version>*, **Nios II Documentation**.

fork()

Prototype:	`pid_t fork (void)`
Commonly called by:	Newlib C library
Thread-safe:	Yes.
Available from ISR:	no
Include:	<**unistd.h**>
Description:	The `fork()` function is only provided for compatibility with newlib.
Return:	Calls to `fork()` always fails with the return code –1 and `errno` set to `ENOSYS`.
See also:	Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II *<version>*, **Nios II Documentation**.

fstat()

Prototype: `int fstat (int fd, struct stat *st)`

Commonly called by: C/C++ programs
Newlib C library

Thread-safe: See description.

Available from ISR: No.

Include: <**sys/stat.h**>

Description: The `fstat()` function obtains information about the capabilities of an open file descriptor. The underlying device driver fills in the input `st` structure with a description of its functionality. See the header file **sys/stat.h** provided with the compiler for the available options.

By default file descriptors are marked as character devices, if the underlying driver does not provide its own implementation of the `fsat()` function.

Calls to `fstat()` are only thread-safe, if the implementation of `fstat()` provided by the driver that is manipulated is thread-safe.

Valid values for the `fd` parameter are: `stdout`, `stdin` and `stderr`, or any value returned from a call to `open()`.

Return: The return value is zero upon success, or –1 otherwise. If the call fails, `errno` is set to indicate the cause of the error.

See also: `close()`
`fcntl()`
`ioctl()`
`isatty()`
`lseek()`
`open()`
`read()`
`stat()`
`write()`
Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II *<version>*, **Nios II Documentation**.

getpid()

Prototype:	`pid_t getpid (void)`
Commonly called by:	Newlib C library
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**unistd.h**>
Description:	The `getpid()` function is provided for newlib compatibility and obtains the current process `id`.
Return:	Because HAL systems cannot contain multiple processes, `getpid()` always returns the same `id` number.
See also:	Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II *<version>*, **Nios II Documentation**.

gettimeofday()

Prototype:

```
int gettimeofday(struct timeval *ptimeval,
                 struct timezone *ptimezone)
```

Commonly called by: C/C++ programs
Newlib C library

Thread-safe: See description.

Available from ISR: Yes.

Include: <**sys/time.h**>

Description: The gettimeofday() function obtains a time structure that indicates the current wall clock time. This time is calculated using the elapsed number of system clock ticks, and the current time value set through the last call to settimeofday().

If this function is called concurrently with a call to settimeofday(), the value returned by gettimeofday() is unreliable; however, concurrent calls to gettimeofday() are legal.

Return: The return value is zero upon success, or –1 otherwise. If the call fails, errno is set to indicate the cause of the error.

See also:
alt_alarm_start()
alt_alarm_stop()
alt_nticks()
alt_sysclk_init()
alt_tick()
alt_ticks_per_second()
settimeofday()
times()
usleep()
Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II *<version>*, **Nios II Documentation**.

ioctl()

Prototype: `int ioctl (int fd, int req, void* arg)`

Commonly called by: C/C++ programs

Thread-safe: See description.

Available from ISR: No.

Include: <**sys/ioctl.h**>

Description: The `ioctl()` function allows application code to manipulate the I/O capabilities of a device driver in driver specific ways. This function is equivalent to the standard UNIX `ioctl()` function. The input argument `fd` is an open file descriptor for the device to manipulate, `req` is an enumeration defining the operation request, and the interpretation of `arg` is request specific.

In general, this implementation vectors requests to the appropriate drivers `ioctl()` function (as registered in the drivers `alt_dev` structure). However, in the case of devices (as opposed to file subsystems), the `TIOCEXCL` and `TIOCNXCL` requests are handled without reference to the driver. These requests lock and release a device for exclusive access.

Calls to `ioctl()` are only thread-safe if the implementation of `ioctl()` provided by the driver that is manipulated is thread-safe.

Valid values for the `fd` parameter are: `stdout`, `stdin` and `stderr`, or any value returned from a call to `open()`.

Return: The interpretation of the return value is request specific. If the call fails, `errno` is set to indicate the cause of the error.

See also:
`close()`
`fcntl()`
`fstat()`
`isatty()`
`lseek()`
`open()`
`read()`
`stat()`
`write()`
Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II *<version>*, **Nios II Documentation**.

isatty()

Prototype: `int isatty(int fd)`

Commonly called by: C/C++ programs
Newlib C library

Thread-safe: See description.

Available from ISR: No.

Include: <**unistd.h**>

Description: The `isatty()` function determines whether the device associated with the open file descriptor `fd` is a terminal device. This implementation uses the drivers `fstat()` function to determine its reply.

Calls to `isatty()` are only thread-safe, if the implementation of `fstat()` provided by the driver that is manipulated is thread-safe.

Return: The return value is 1 if the device is a character device, and zero otherwise. If an error occurs, `errno` is set to indicate the cause.

See also:
`close()`
`fcntl()`
`fstat()`
`ioctl()`
`lseek()`
`open()`
`read()`
`stat()`
`write()`
Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II *<version>*, **Nios II Documentation**.

kill()

Prototype: `int kill(int pid, int sig)`

Commonly called by: Newlib C library

Thread-safe: Yes.

Available from ISR: Yes.

Include: <**signal.h**>

Description: The `kill()` function is used by newlib to send signals to processes. The input argument `pid` is the `id` of the process to signal, and `sig` is the signal to send. As there is only a single process in the HAL, the only valid values for `pid` are either the current process `id`, as returned by `getpid()`, or the broadcast values, i.e., `pid` must be less than or equal to zero.

The following signals result in an immediate shutdown of the system, without call to `exit()`: `SIGABRT`, `SIGALRM`, `SIGFPE`, `SIGILL`, `SIGKILL`, `SIGPIPE`, `SIGQUIT`, `SIGSEGV`, `SIGTERM`, `SIGUSR1`, `SIGUSR2`, `SIGBUS`, `SIGPOLL`, `SIGPROF`, `SIGSYS`, `SIGTRAP`, `SIGVTALRM`, `SIGXCPU`, and `SIGXFSZ`.

The following signals are ignored: `SIGCHLD` and `SIGURG`.

All the remaining signals are treated as errors.

Return: The return value is zero upon success, or –1 otherwise. If the call fails, `errno` is set to indicate the cause of the error.

See also: Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II *<version>*, **Nios II Documentation**.

link()

Prototype:

```
int link(const char *_path1,
         const char *_path2)
```

Commonly called by: Newlib C library

Thread-safe: Yes.

Available from ISR: Yes.

Include: <**unistd.h**>

Description: The `link()` function is only provided for compatibility with newlib.

Return: Calls to `link()` always fails with the return code –1 and `errno` set to `ENOSYS`.

See also: Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II *<version>*, **Nios II Documentation**.

lseek()

Prototype: `off_t lseek(int fd, off_t ptr, int whence)`

Commonly called by: C/C++ programs
Newlib C library

Thread-safe: See description.

Available from ISR: No.

Include: <**unistd.h**>

Description: The `lseek()` function moves the read/write pointer associated with the file descriptor `fd`. This function vectors the call to the `lseek()` function provided by the driver associated with the file descriptor. If the driver does not provide an implementation of `lseek()`, an error is indicated.

`lseek()` corresponds to the standard UNIX `lseek()` function.

You can use the following values for the input parameter, `whence`:

- Value of `whence`
- Interpretation
- `SEEK_SET`—the offset is set to `ptr` bytes.
- `SEEK_CUR`—the offset is incremented by `ptr` bytes.
- `SEEK_END`—the offset is set to the end of the file plus `ptr` bytes.

Calls to `lseek()` are only thread-safe if the implementation of `lseek()` provided by the driver that is manipulated is thread-safe.

Valid values for the `fd` parameter are: `stdout`, `stdin` and `stderr`, or any value returned from a call to `open()`.

Return: Upon success, the return value is a non-negative file pointer. The return value is –1 in the event of an error. If the call fails, `errno` is set to indicate the cause of the error.

See also: `close()`
`fcntl()`
`fstat()`
`ioctl()`
`isatty()`
`open()`
`read()`
`stat()`
`write()`
Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II *<version>*, **Nios II Documentation**.

open()

Prototype: `int open (const char* pathname, int flags, mode_t mode)`

Commonly called by: C/C++ programs

Thread-safe: See description.

Available from ISR: No.

Include: <**unistd.h**>
<**fcntl.h**>

Description: The `open()` function opens a file or device and returns a file descriptor (a small, non-negative integer for use in read, write, etc.)

`flags` is one of: `O_RDONLY`, `O_WRONLY`, or `O_RDWR`, which request opening the file read-only, write-only or read/write, respectively.

You can also bitwise-`OR` `flags` with `O_NONBLOCK`, which causes the file to be opened in non-blocking mode. Neither `open()` nor any subsequent operations on the returned file descriptor causes the calling process to wait.

Note that not all file systems/devices recognize this option.

`mode` specifies the permissions to use, if a new file is created. It is unused by current file systems, but is maintained for compatibility.

Calls to `open()` are only thread-safe if the implementation of `open()` provided by the driver that is manipulated is thread-safe.

Return: The return value is the new file descriptor, and –1 otherwise. If an error occurs, `errno` is set to indicate the cause.

See also: `close()`
`fcntl()`
`fstat()`
`ioctl()`
`isatty()`
`lseek()`
`read()`
`stat()`
`write()`
Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II *<version>*, **Nios II Documentation**.

read()

Prototype: `int read(int fd, void *ptr, size_t len)`

Commonly called by: C/C++ programs
Newlib C library

Thread-safe: See description.

Available from ISR: No.

Include: <**unistd.h**>

Description: The `read()` function reads a block of data from a file or device. This function vectors the request to the device driver associated with the input open file descriptor `fd`. The input argument, `ptr`, is the location to place the data read and `len` is the length of the data to read in bytes.

Calls to `read()` are only thread-safe if the implementation of `read()` provided by the driver that is manipulated is thread-safe.

Valid values for the `fd` parameter are: `stdout`, `stdin` and `stderr`, or any value returned from a call to `open()`.

Return: The return argument is the number of bytes read, which might be less than the requested length.

A return value of –1 indicates an error. In the event of an error, `errno` is set to indicate the cause.

See also: `close()`
`fcntl()`
`fstat()`
`ioctl()`
`isatty()`
`lseek()`
`open()`
`stat()`
`write()`
Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II *<version>*, **Nios II Documentation**.

sbrk()

Prototype: `caddr_t sbrk(int incr)`

Commonly called by: Newlib C library

Thread-safe: No.

Available from ISR: No.

Include: <**unistd.h**>

Description: The `sbrk()` function dynamically extends the data segment for the application. The input argument `incr` is the size of the block to allocate. Do not call `sbrk()` directly–if you wish to dynamically allocate memory, use the newlib `malloc()` function.

Return: –

See also: Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II *<version>*, **Nios II Documentation**.

settimeofday()

Prototype:

```
int settimeofday (const struct timeval  *t,
                  const struct timezone *tz)
```

Commonly called by: C/C++ programs

Thread-safe: No.

Available from ISR: Yes.

Include: <**sys/time.h**>

Description: If the `settimeofday()` function is called concurrently with a call to `gettimeofday()`, the value returned by `gettimeofday()` is unreliable.

Return: The return value is zero upon success, or –1 otherwise. The current implementation always succeeds.

See also:

`alt_alarm_start()`
`alt_alarm_stop()`
`alt_nticks()`
`alt_sysclk_init()`
`alt_tick()`
`alt_ticks_per_second()`
`gettimeofday()`
`times()`
`usleep()`

stat()

Prototype:

```
int stat(const char  *file_name,
         struct stat *buf);
```

Commonly called by:

C/C++ programs
Newlib C library

Thread-safe: See description.

Available from ISR: No.

Include: <**sys/stat.h**>

Description:

The `stat()` function is similar to the `fstat()` function—it obtains status information about a file. Instead of using an open file descriptor, like `fstat()`, `stat()` takes the name of a file as an input argument.

Calls to `stat()` are only thread-safe, if the implementation of `stat()` provided by the driver that is manipulated is thread-safe.

Internally, the `stat()` function is implemented as a call to `fstat()`. See "fstat()" on page 12–56.

Return: –

See also:

`close()`
`fcntl()`
`fstat()`
`ioctl()`
`isatty()`
`lseek()`
`open()`
`read()`
`write()`
Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II *<version>*, **Nios II Documentation**.

times()

Prototype: `clock_t times (struct tms *buf)`

Commonly called by: C/C++ programs
Newlib C library

Thread-safe: Yes.

Available from ISR: Yes.

Include: <**sys/times.h**>

Description: This `times()` function is provided for compatibility with newlib. It returns the number of clock ticks since reset. It also fills in the structure pointed to by the input parameter `buf` with time accounting information. The definition of the `tms` structure is:

```
typedef struct
{
  clock_t tms_utime;
  clock_t tms_stime;
  clock_t tms_cutime;
  clock_t tms_cstime;
};
```

The structure has the following elements:

- `tms_utime`: the CPU time charged for the execution of user instructions
- `tms_stime`: the CPU time charged for execution by the system on behalf of the process
- `tms_cutime`: the sum of all the `tms_utime` and `tms_cutime` of the child processes
- `tms_cstime`: the sum of the `tms_stimes` and `tms_cstimes` of the child processes

In practice, all elapsed time is accounted as system time. No time is ever attributed as user time. In addition, no time is allocated to child processes, as child processes can not be spawned by the HAL.

Return: If there is no system clock available, the return value is zero.

See also:

```
alt_alarm_start()
alt_alarm_stop()
alt_nticks()
alt_sysclk_init()
alt_tick()
alt_ticks_per_second()
gettimeofday()
settimeofday()
usleep()
```

Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II <*version*>, **Nios II Documentation**.

unlink()

Prototype: `int unlink(char *name)`

Commonly called by: Newlib C library

Thread-safe: Yes.

Available from ISR: Yes.

Include: <**unistd.h**>

Description: The `unlink()` function is only provided for compatibility with newlib.

Return: Calls to `unlink()` always fails with the return code –1 and `errno` set to `ENOSYS`.

See also: Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II <*version*>, **Nios II Documentation**.

usleep()

Prototype: `int usleep (int us)`

Commonly called by: C/C++ programs
Device drivers

Thread-safe: Yes.

Available from ISR: No.

Include: <**unistd.h**>

Description: The `usleep()` function blocks until at least `us` microseconds have elapsed.

Return: The `usleep()` function returns zero upon success, or –1 otherwise. If an error occurs, `errno` is set to indicate the cause. The current implementation always succeeds.

See also:

`alt_alarm_start()`
`alt_alarm_stop()`
`alt_nticks()`
`alt_sysclk_init()`
`alt_tick()`
`alt_ticks_per_second()`
`gettimeofday()`
`settimeofday()`
`times()`

wait()

Prototype:	`int wait(int *status)`
Commonly called by:	Newlib C library
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	<**sys/wait.h**>
Description:	Newlib uses the `wait()` function to wait for all child processes to exit. Because the HAL does not support spawning child processes, this function returns immediately.
Return:	Upon return, the content of `status` is set to zero, which indicates there is no child processes. The return value is always –1 and `errno` is set to `ECHILD`, which indicates that there are no child processes to wait for.
See also:	Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II *<version>*, **Nios II Documentation**.

write()

Prototype: `int write(int fd, const void *ptr, size_t len)`

Commonly called by: C/C++ programs
Newlib C library

Thread-safe: See description.

Available from ISR: no

Include: <**unistd.h**>

Description: The `write()` function writes a block of data to a file or device. This function vectors the request to the device driver associated with the input file descriptor `fd`. The input argument `ptr` is the data to write and `len` is the length of the data in bytes.

Calls to `write()` are only thread-safe if the implementation of `write()` provided by the driver that is manipulated is thread-safe.

Valid values for the `fd` parameter are: `stdout`, `stdin` and `stderr`, or any value returned from a call to `open()`.

Return: The return argument is the number of bytes written, which might be less than the requested length.

A return value of –1 indicates an error. In the event of an error, `errno` is set to indicate the cause.

See also: `close()`
`fcntl()`
`fstat()`
`ioctl()`
`isatty()`
`lseek()`
`open()`
`read()`
`stat()`
Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II *<version>*, **Nios II Documentation**.

Standard Types

In the interest of portability, the HAL uses a set of standard type definitions in place of the ANSI C built-in types. Table 12–2 describes these types that are defined in the header **alt_types.h**.

Table 12–2. Standard Types

Type	Description
`alt_8`	Signed 8-bit integer.
`alt_u8`	Unsigned 8-bit integer.
`alt_16`	Signed 16-bit integer.
`alt_u16`	Unsigned 16-bit integer.
`alt_32`	Signed 32-bit integer.
`alt_u32`	Unsigned 32-bit integer.
`alt_64`	Signed 64-bit integer.
`alt_u64`	Unsigned 64-bit integer.

Referenced Documents

This chapter references the following documents:

- *Newlib ANSI C standard library documentation* installed with the Nios II EDS

Document Revision History

Table 12–3 shows the revision history for this document.

Table 12–3. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	No change from previous release.	
May 2007 v7.1.0	● Chapter 11 was formerly chapter 10. ● Added table of contents to Introduction section. ● Added Referenced Documents section.	
March 2007 v7.0.0	No change from previous release.	
November 2006 v6.1.0	Function `open()` requires `fcntl.h`.	
May 2006 v6.0.0	No change from previous release.	
October 2005 v5.1.0	Added API entries for "alt_irq_disable()" and "alt_irq_enable()", which were previously omitted by error.	
May 2005 v5.0.0	● Added `alt_load_section()` function ● Added `fcntl()` function	
December 2004 v1.2	Updated names of DMA generic requests.	
September 2004 v1.1	● Added `open()`. ● Added `ERRNO` information to `alt_dma_txchan_open()`. ● Corrected `ALT_DMA_TX_STREAM_ON` definition. ● Corrected `ALT_DMA_RX_STREAM_ON` definition. ● Added information to `alt_dma_rxchan_ioctl()` and `alt_dma_txchan_ioctl()`.	
May 2004 v1.0	Initial Release.	

13. Altera-Provided Development Tools

NII520011-7.2.0

Introduction

This chapter summarizes the development tools that Altera® provides for the Nios® II processor. This chapter does not describe detailed usage of any of the tools, but it refers you to the most appropriate documentation. This chapter contains the following sections:

- "The Nios II IDE Tools" on page 13–1
- "Altera Nios II Build Tools" on page 13–2
- "GNU Compiler Tool Chain" on page 13–5
- "Libraries and Embedded Software Packages" on page 13–6

The Nios II IDE Tools

Table 13–1 describes the tools provided by the Nios II IDE user interface.

Table 13–1. The Nios II IDE and Associated Tools

Tools	Description
The Nios II IDE	The Nios II IDE is the software development user interface for the Nios II processor. All software development tasks can be accomplished within the IDE, including editing, building, and debugging programs. For more information, refer to the Nios II IDE help system.
Flash programmer	The Nios II IDE includes a flash programmer utility that allows you to program flash memory chips on a target board. The flash programmer supports programming flash on any board, including Altera development boards and your own custom boards. The flash programmer facilitates programming flash for the following purposes: ● Executable code and data ● Bootstrap code to copy code from flash to RAM, and then run from RAM. ● HAL file subsystems ● FPGA hardware configuration data For more information, refer to the *Nios II Flash Programmer User Guide*.
Instruction set simulator	Altera provides an instruction set simulator (ISS) for the Nios II processor. The ISS is available within the Nios II IDE, and the process for running and debugging programs on the ISS is the same as for running and debugging on target hardware. For more information, refer to the Nios II IDE help system.
Quartus® II Programmer	The Quartus II programmer is part of the Quartus II Complete Design Suite, however the Nios II IDE can launch the Quartus II programmer directly. The Quartus II programmer allows you to download new FPGA configuration files to the board. For more information, refer to the Nios II IDE help system, or press the F1 key while the Quartus II programmer is open.

Altera Nios II Build Tools

This section describes the Altera Nios II build tools. Under Windows, you can run these tools from a *Nios II Command Shell* command prompt. Under Linux, use the command shell of your preference.

Each tool provides its own documentation in the form of help pages accessible from the command line. To view the help, open a *Nios II Command Shell*, and type the following command:

```
<name of tool> --help
```

Nios II Software Build Tools

The Nios II software build tools are utilities and scripts that provide similar functionality to the **New Project** wizard and the **System Library** properties page in the Nios II IDE. You can create, modify and build Nios II programs with commands typed at a command line or embedded in a script.

Table 13–2 summarizes the command line utilities and scripts included in the software build tools. You can invoke these utilities on the command line or from a scripting language of your choice (such as **perl** or **bash**). On Windows, these utilities have a **.exe** extension.

Table 13–2. Nios II Software Build Tools Utilities and Scripts

Command	Summary
nios2-app-generate-makefile	Creates an application makefile
nios2-lib-generate-makefile	Creates a library makefile
nios2-bsp-create-settings	Creates a board support package (BSP) settings file
nios2-bsp-update-settings	Updates the contents of a BSP settings file
nios2-bsp-query-settings	Queries the contents of a BSP settings file
nios2-bsp-generate-files	Generates all files for a given BSP settings file
nios2-bsp	Creates or updates a BSP
nios2-c2h-generate-makefile	Creates an application makefile fragment for the Nios II C2H compiler

The Nios II software build tools reside in the *<Nios II EDS install path>***/sdk2/bin** directory.

For further information about the Nios II software build tools, refer to the *Introduction to the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

File Format Conversion Tools

File format conversion is sometimes necessary when passing data from one utility to another. Table 13–3 shows the Altera-provided utilities for converting file formats.

Table 13–3. File Conversion Utilities

Utility	Description
bin2flash	Converts binary files to a Motorola S-record file (**.flash**) for programming into flash memory.
elf2dat	Converts an executable and linking format file (**.elf)** to a **.dat** file format appropriate for Verilog HDL hardware simulators.
elf2flash	Converts an executable and linking format file to an S-record file for programming into flash memory.
elf2hex	Converts an executable and linking format file to the Intel hexadecimal file (**.hex**) format.
elf2mem	Generates the memory contents for the memory devices in a specific Nios II system.
elf2mif	Converts an executable and linking format file to the Quartus II memory initialization file **(.mif**) format
flash2dat	Converts an S-record file to the **.dat** file format appropriate for Verilog HDL hardware simulators.
sof2flash	Converts an SRAM object file to an S-record file for programming into flash memory.

The file format conversion tools are in the *<Nios II EDS install path>***/bin/** directory.

Other Command-Line Tools

Table 13–4 shows other Altera-provided command-line tools for developing Nios II programs.

Table 13–4. Altera Command-Line Tools

Tool	Description
nios2-download	Downloads code to a target processor for debugging or running.
nios2-flash-programmer	Programs data to flash memory on the target board.
nios2-gdb-server	Translates GNU debugger (GDB) remote serial protocol packets over TCP to joint test action group (JTAG) transactions with a target Nios II processor.
nios2-terminal	Performs terminal I/O with a JTAG universal asynchronous receiver-transmitter (UART) in a Nios II system
validate_zip	Verifies if a specified zip file is compatible with Altera's read-only zip file system.
nios2-debug	Downloads a program to a Nios II processor and launches the Insight debugger.
nios2-console	Opens the FS2 command-line interface (CLI), connects to the Nios II processor, and (optionally) downloads code.
nios2-configure-sof	Configures an Altera configurable part. If no explicit SRAM object file (**.sof**) is specified, it tries to determine the correct file to use.
jtagconfig	Allows you configure the JTAG server on the host machine. It can also detect a JTAG chain and set up the download hardware configuration.

The command-line tools described in this section are in the <*Nios II EDS install path*>**/bin/** directory.

Nios II IDE Command-Line Tools

Table 13–5 on page 13–5 shows the command-line utilities that form the basis of the Nios II IDE. These tools can create and build Nios II IDE projects without launching the Nios II IDE graphical user interface (GUI). However, Altera recommends that you use the Nios II software build tools for new projects.

For detailed information about the Nios II software build tools, refer to the *Introduction to the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

Each of the Nios II IDE command-line tools launches the Nios II IDE in the background, without displaying the GUI. You cannot use these utilities while the IDE is running, because only one instance of the Nios II IDE can be active at a time.

Table 13–5. Nios II IDE Command-Line Tools

Tool	Description
nios2-create-system-library	Creates a new system library project.
nios2-create-application-project	Creates a new C/C++ application project.
nios2-build-project	Builds a project using the Nios II IDE managed-make facilities. Creates or updates the makefiles to build the project, and optionally runs make. **nios2-build-project** operates only on projects that exist in the current Nios II IDE workspace.
nios2-import-project	Imports a previously-created Nios II IDE project into the current workspace.
nios2-delete-project	Removes a project from the Nios II IDE workspace, and optionally deletes files from the file system.

The Nios II IDE command-line tools are in the <*Nios II EDS install path*>**/bin/** directory.

GNU Compiler Tool Chain

GNU Tool Chain

Altera provides and supports the standard GNU compiler tool chain for the Nios II processor. Complete HTML documentation for the GNU tools resides in the Nios II Embedded Design Suite (EDS) directory. The GNU tools are in the <*Nios II EDS install path*>**/bin/nios2-gnutools** directory.

GNU tools for the Nios II processor are generally named **nios2-elf-**<*tool name*>. The following list shows some examples:

- `nios2-elf-gcc`
- `nios2-elf-as`
- `nios2-elf-ld`
- `nios2-elf-objdump`
- `nios2-elf-size`

The exception is the **make** utility, which is simply named `make`.

For a comprehensive list of GNU tools, refer to the GNU HTML documentation, installed with the Nios II EDS.

Libraries and Embedded Software Packages

Table 13–6 shows the Nios II libraries and software packages.

Table 13–6. Libraries and Software Packages

Name	Description
Hardware abstraction layer (HAL) system library	See the *Overview of the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.
MicroC/OS-II RTOS	See the *MicroC/OS-II Real Time Operating System* chapter of the *Nios II Software Developer's Handbook*.
NicheStack TCP/IP Stack - Nios II Edition	See the *Ethernet and the NicheStack TCP/IP Stack - Nios II Edition* chapter of the *Nios II Software Developer's Handbook*.
newlib ANSI C standard library	The complete HTML documentation for newlib resides in the Nios II EDS directory. Also see the *Overview of the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.
Read-only zip file system	See the *Read-Only Zip File System* chapter of the *Nios II Software Developer's Handbook*.
Host file system	See the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*.

Example Designs

The Nios II EDS provides documented software examples to demonstrate all prominent features of the Nios II processor and the development environment.

Referenced Documents

This chapter references the following documents:

- *Introduction to the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*
- *Overview of 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*.
- *Ethernet and the NicheStack TCP/IP Stack - Nios II Edition* chapter of the *Nios II Software Developer's Handbook*.
- *Read-Only Zip File System* chapter of the *Nios II Software Developer's Handbook*.
- *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*
- *GNU documentation* installed with the Nios II EDS

Document Revision History

Table 13–7 shows the revision history for this document.

Table 13–7. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	● **mk-nios2-signaltap-mnemonic-table** deprecated ● Add **jtagconfig** ● Add Host File System	
May 2007 v7.1.0	● Discuss Nios II software build tools ● Added table of contents to Introduction section. ● Added Referenced Documents section.	Nios II software build tools
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	● Added **nios2-configure-sof** tool. ● Removed utilities for the legacy SDK flow, because it is no longer supported.	
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 Nios II command line tools information.	
May 2004 v1.0	Initial Release.	

14. Nios II Software Build Tools Reference

NII52016-7.2.0

Introduction

This chapter provides a complete reference of all available commands, options and settings for the Nios® II software build tools. This reference is useful for developing your own software projects, packages, or device drivers.

Before using this chapter, read the *Introduction to the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*, and familiarize yourself with the parts of the *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook* that are relevant to your tasks.

This chapter includes the following sections:

- "Nios II Software Build Tools Utilities" on page 14–1
- "Settings" on page 14–21
- "Tcl Commands for BSP Settings" on page 14–160
- "Tcl Commands for Drivers and Packages" on page 14–206
- "Path Names" on page 14–218

Nios II Software Build Tools Utilities

The build tools utilities are the entry point into the Nios® II software build tools. Everything you can do with the tools, such as specifying settings, creating makefiles, and building projects, is made available by the utilities.

Each build tools utility shares the following behaviors:

- Sends error messages and warning messages to `stderr`.
- Sends normal messages (other than errors and warnings) to `stdout`.
- Displays one error message for each error.
- Returns an exit value of 1 if it detects any errors.
- Returns an exit value of 0 if it does not detect any errors. (Warnings are not errors.)
- If the help or version command line option is specified, returns an exit value of 0, and takes no other action. Sends the output (help or version number) to `stdout`.
- If no command-line arguments are specified, returns an exit value of 1 and sends a help message to `stderr`. All commands require at least one argument.
- When an error is detected, suppresses all subsequent operations (such as writing files).

Logging Levels

All the utilities support multiple status logging levels. You specify the logging level on the command line. Table 14–1 shows the logging levels supported. At each level, the utilities report the status as listed under **Description**. Each level includes the messages from all lower levels.

Table 14–1. Nios II Software Build Tools Logging Levels

Logging Level	Description
silent (lowest)	No information is provided except for errors and warnings (sent to `stderr`).
default	Minimal information is provided (for example, start and stop operation of software build tools phases).
verbose	Detailed information is provided (for example, lists of files written).
debug (highest)	Debug information is provided (for example, stack backtraces on errors). This level is for reporting problems to Altera®.

Table 14–2 shows the command line options used to select each logging level. Only one logging level is possible at a time, so these options are all mutually exclusive.

Table 14–2. Selecting Logging Level

Command Line Option	Logging Level	Comments
none	default	No command line option selects the default level.
`--silent`	silent	Selects silent level of logging.
`--verbose`	verbose	Selects verbose level of logging.
`--debug`	debug	Selects debug level of logging.
`--log` <*fname*>	debug	All information is written to <*fname*> in addition to being sent to the `stdout` and `stderr` devices.

Setting Formats

The format in which you specify the setting value depend on the setting type. Several settings types are supported. Table 14–3 shows the allowed formats for each setting type.

The value of a setting is specified with the `--set` command line option to **nios2-bsp-create-settings** and **nios2-bsp-update-settings** or with the `set_setting` Tcl command. The value of a setting is obtained with the `--get` command line option to **nios2-bsp-query-settings** or with the get_setting Tcl command.

Table 14–3. Setting Formats

Setting Type	Format When Setting	Format When Getting
boolean	0/1 or false/true	0/1
number	0x prefix for hexadecimal or no prefix for a decimal number	decimal
string	Quoted string. Use "`none`" to set string to empty (do not use "")	Quoted string

Utility Summary

The command line utilities are as follows:

nios2-app-generate-makefile

Usage

```
nios2-app-generate-makefile
    [--app-dir <directory>] --bsp-dir <directory>
    [--c2h] [--debug] [--elf-name <filename>]
    [--extended-help] [--help] [--log <filename>]
    [--set <name value>] [--silent] [--src-dir
    <directory>] [--src-files <filenames>] [--src-rdir
    <directory>] [--use-lib-dir <directory>]
    [--verbose] [--version]
```

Options

- --app-dir <directory>: Destination directory for the application makefile and ELF. If omitted, it defaults to the current directory.
- --bsp-dir <directory>: Path to the BSP generated files directory (populated using the nios2-bsp-generate-files command).
- --c2h: Enables C2H support. Includes a static C2H makefile fragment in the application makefile. Also copies a null c2h.mk to the makefile directory.
- --debug: Outputs debug, exception traces, verbose, and default information about the command's operation to stdout.
- --elf-name <filename>: Name of the executable file (.elf) to create. If omitted, it defaults to the first source file specified with the file name extension replaced with .elf and placed in the application directory.
- --extended-help: Displays full information about this command and its options.
- --help: Displays basic information about this command and its options.
- --log <filename>: Creates a debug log and write to specified file. Also logs debug information to stdout.
- --set <name value>: Sets the makefile variable called <name> to <value>. If the variable exists in the managed section of the makefile, <value> replaces the default settings. If the variable does not already exist, it is added. Multiple --set options are allowed.
- --silent: Suppresses information about the command's operation normally sent to stdout.
- --src-dir <directory>: Searches for source files in <filepath>. Use . to look in the current directory. Multiple --src-dir options are allowed.
- --src-files <filenames>: A list of space-separated source file names added to the makefile. The list of file names is terminated by the next option or the end of the command line. Multiple --src-files options are allowed.

- --src-rdir <directory>: Same as --src-dir option but recursively searches for source files in or under <filepath>. Multiple --src-rdir options are allowed. You can mix --src-rdir with --src-dir options.
- --use-lib-dir <directory>: Path to a dependent library directory. The library directory must contain a makefile fragment called public.mk. Multiple --use-lib-dir options are allowed.
- --verbose: Outputs verbose, and default information about the command's operation to stdout.
- --version: Displays the version of this command and exits with a zero exit status.

Description

The nios2-app-generate-makefile command generates an application makefile (called Makefile). The path to a BSP created by nios2-bsp-generate-files is a mandatory command line option.

You can enable support for the Nios II C2H compiler with the c2h option, which creates a null C2H makefile fragment in your project, and includes it in the application make file. This makefile fragment, c2h.mk, contains comments to help you fill in the make file variables by hand. NOTE: this c2h.mk will overwrite any existing c2h.mk.

You can use the command line tool nios2-c2h-generate-makefile to generate a populated C2H make file fragment.

For more details about this command, use the --extended-help option to display comprehensive usage information.

nios2-bsp-create-settings

Usage

```
nios2-bsp-create-settings
    [--cmd <tcl command>] [--cpu-name <cpu name>]
    [--debug] [--extended-help] [--help]
    [--librarian-factory-path <directory>]
    [--librarian-path <directory>] [--log <filename>]
    [--script <filename>] [--set <name value>]
    --settings <filename> [--silent] --sopc <filename>
    --type <bsp type> [--verbose] [--version]
```

Options

- --cmd <tcl command>: Runs the specified Tcl command. Multiple --cmd options are allowed.
- --cpu-name <cpu name>: The name of the Nios II processor that the BSP supports. Optional for a single-processor SOPC Builder system.
- --debug: Outputs debug, exception traces, verbose, and default information about the command's operation to stdout.
- --extended-help: Displays full information about this command and its options.
- --help: Displays basic information about this command and its options.
- --librarian-factory-path <directory>: Comma separated librarian search path. Use '$' for default factory search path.
- --librarian-path <directory>: Comma separated librarian search path. Use '$' for default search path.
- --log <filename>: Creates a debug log and write to specified file. Also logs debug information to stdout.
- --script <filename>: Runs the specified Tcl script with optional arguments. Multiple --script options are allowed.
- --set <name value>: Sets the setting called <name> to <value>. Multiple --set options are allowed.
- --settings <filename>: File name of the BSP settings file to create. The Nios II software build tools create this file with a .bsp file extension. It overwrites any existing settings file.
- --silent: Suppresses information about the command's operation normally sent to stdout.
- --sopc <filename>: The SOPC Builder design file used to create the BSP.
- --type <bsp type>: BSP type. If this argument is missing, an error message lists all available BSP types.
- --verbose: Outputs verbose, and default information about the command's operation to stdout.

- --version: Displays the version of this command and exits with a zero exit status.

Description

If you use nios2-bsp-create-settings to create a settings file without any command line options, Tcl commands, or Tcl scripts to modify the default settings, it creates a settings file that fails when running nios2-bsp-generate-files. Failure occurs because the nios2-bsp-create-settings command is able to create reasonable defaults for most settings, but the command requires additional information for system-dependent settings. The default Tcl scripts set the required system-dependent settings. Therefore it is better to use default Tcl scripts when calling nios2-bsp-create-settings directly. For an example of how to use the default Tcl scripts, refer to the nios2-bsp bash script.

For more details about this command, use the --extended-help option to display comprehensive usage information.

Example

```
nios2-bsp-create-settings --settings my_settings.bsp --sopc \
    ../my_sopc.sopc --type hal --script default_settings.tcl
```

nios2-bsp-generate-files

Usage

```
nios2-bsp-generate-files
   --bsp-dir <directory> [--debug] [--extended-help]
   [--help] [--librarian-factory-path <directory>]
   [--librarian-path <directory>] [--log <filename>]
   --settings <filename> [--silent] [--verbose]
   [--version]
```

Options

- --bsp-dir <directory>: Path to the directory where nios2-bsp-generate-files places the BSP files. Use . for the current directory. The directory <filepath> must exist. This command overwrites pre-existing files in <filepath> without warning.
- --debug: Sends debug, exception trace, verbose, and default information about the command's operation to stdout.
- --extended-help: Displays full information about this command and its options.
- --help: Displays basic information about this command and its options.
- --librarian-factory-path <directory>: Comma separated librarian search path. Use '$' for default factory search path.
- --librarian-path <directory>: Comma separated librarian search path. Use '$' for default search path.
- --log <filename>: Creates a debug log and writes to specified file. Also logs debug information to stdout.
- --settings <filename>: File name of an existing BSP settings file (.bsp) to generate files from.
- --silent: Suppresses information about the command's operation normally sent to stdout.
- --verbose: Sends verbose and default information about the command's operation to stdout.
- --version: Displays the version of this command and exits with a zero exit status.

Description

The nios2-bsp-generate-files command creates a board support package (BSP). The path to an existing BSP settings file (.bsp file) and the path to the BSP directory are mandatory command line options. nios2-bsp-generate-files writes generated files into the specified BSP directory.

For more details about this command, use the --extended-help option to display comprehensive usage information.

nios2-bsp-query-settings

Usage

```
nios2-bsp-query-settings
    [--cmd <tcl command>] [--debug] [--extended-help]
    [--get <name>] [--get-all] [--help]
    [--librarian-factory-path <directory>]
    [--librarian-path <directory>] [--log <filename>]
    [--script <filename>] --settings <filename>
    [--show-descriptions] [--show-names] [--silent]
    [--verbose] [--version]
```

Options

- --cmd <tcl command>: Runs the specified Tcl command. Multiple --cmd options are allowed.
- --debug: Outputs debug, exception traces, verbose, and default information about the command's operation to stdout.
- --extended-help: Displays full information about this command and its options.
- --get <name>: Displays the value of the setting called <name>. Multiple --get options are allowed. Each value appears on its own line, in the order in which you specify the --get options. This option is mutually exclusive with the --get-all option.
- --get-all: Displays the value of all BSP settings in order sorted by option name. Each option appears on its own line. Mutually exclusive with the --get option.
- --help: Displays basic information about this command and its options.
- --librarian-factory-path <directory>: Comma separated librarian search path. Use '$' for default factory search path.
- --librarian-path <directory>: Comma separated librarian search path. Use '$' for default search path.
- --log <filename>: Creates a debug log and write to specified file. Also logs debug information to stdout.
- --script <filename>: Runs the specified Tcl script with optional arguments. Multiple --script options are allowed.
- --settings <filename>: File name of an existing BSP settings file to query settings from.
- --show-descriptions: Displays the description of each option after the value.
- --show-names: Displays the name of each option before the value.
- --silent: Suppresses information about the command's operation normally sent to stdout.
- --verbose: Outputs verbose, and default information about the command's operation to stdout.

- --version: Displays the version of this command and exits with a zero exit status.

Description

The nios2-bsp-query-settings command provides information from a Nios II board support package (BSP) settings file. The path to an existing BSP settings file (.bsp file) is a mandatory command line option. The command does not modify the settings file. This command only displays information requested by the user on stdout. It does not display informational messages.

For more details about this command, use the --extended-help option to display comprehensive usage information.

nios2-bsp-update-settings

Usage

```
nios2-bsp-update-settings
    [--cmd <tcl command>] [--cpu-name <cpu name>]
    [--debug] [--extended-help] [--help]
    [--librarian-factory-path <directory>]
    [--librarian-path <directory>] [--log <filename>]
    [--script <filename>] [--set <name value>]
    --settings <filename> [--silent] [--sopc
    <filename>] [--verbose] [--version]
```

Options

- --cmd <tcl command>: Runs the specified Tcl command. Multiple --cmd options are allowed.
- --cpu-name <cpu name>: The name of the Nios II processor that the BSP supports. This argument is useful if the SOPC Builder design contains multiple Nios II processors. Optional for single-processor SOPC Builder design.
- --debug: Outputs debug, exception traces, verbose, and default information about the command's operation to stdout.
- --extended-help: Displays full information about this command and its options.
- --help: Displays basic information about this command and its options.
- --librarian-factory-path <directory>: Comma separated librarian search path. Use '$' for default factory search path.
- --librarian-path <directory>: Comma separated librarian search path. Use '$' for default search path.
- --log <filename>: Creates a debug log and write to specified file. Also logs debug information to stdout.
- --script <filename>: Runs the specified Tcl script with optional arguments. Multiple --script options are allowed.
- --set <name value>: Sets the setting called <name> to <value>. Multiple --set options are allowed.
- --settings <filename>: File name of an existing BSP settings file to update.
- --silent: Suppresses information about the command's operation normally sent to stdout.
- --sopc <filename>: The SOPC Builder design file to <filename> update the BSP with. This argument is useful if the path to the original SOPC Builder system file has changed.
- --verbose: Outputs verbose, and default information about the command's operation to stdout.

- --version: Displays the version of this command and exits with a zero exit status.

Description

The nios2-bsp-update-settings command updates an existing Nios II board support package (BSP) settings file. The path to an existing BSP settings file (.bsp file) is a mandatory command line option. The command modifies the settings file so the file must have write permissions. You might want to pass the default Tcl script to the nios2-bsp-update-settings command to make sure that your BSP is consistent with your SOPC Builder system. The nios2-bsp command uses the default Tcl script this way.

For more details about this command, use the --extended-help option to display comprehensive usage information.

nios2-lib-generate-makefile

Usage

```
nios2-lib-generate-makefile
    [--bsp-dir <directory>] [--debug]
    [--extended-help] [--help] [--lib-dir <directory>]
    [--lib-name <filename>] [--log <filename>]
    [--public-inc-dir <directory>] [--set <name value>]
    [--silent] [--src-dir <directory>] [--src-files
    <filenames>] [--src-rdir <directory>]
    [--use-lib-dir <directory>] [--verbose]
    [--version]
```

Options

- --bsp-dir <directory>: Path to the BSP generated files directory (populated using the nios2-bsp-generate-files command).
- --debug: Outputs debug, exception traces, verbose, and default information about the command's operation to stdout.
- --extended-help: Displays full information about this command and its options.
- --help: Displays basic information about this command and its options.
- --lib-dir <directory>: Destination directory for the library makefile, public.mk, and .a. If omitted, it defaults to the current directory.
- --lib-name <filename>: Name of the library being created. The library file name is the library name with a "lib" prefix and ".a" suffix added. Do not include the prefix and suffix in the argument value. If you omit the library name option, the library name defaults to the name of the first source file (minus the source filename extension).
- --log <filename>: Creates a debug log and write to specified file. Also logs debug information to stdout.
- --public-inc-dir <directory>: Path to a directory that contains C-language header files (.h files) that need to be available (public) to users of the library. nios2-lib-generate-makefile adds this directory to the appropriate variable in public.mk. Multiple --public-inc-dir options are allowed.
- --set <name value>: Sets the makefile variable called <name> to <value>. If the variable exists in the managed section of the makefile, <value> replaces the default settings. It adds the makefile variable if it does not already exist. Multiple --set options are allowed.
- --silent: Suppresses information about the command's operation normally sent to stdout.
- --src-dir <directory>: Searches for source files in <filepath>. Use . to look in the current directory. Multiple --src-dir options are allowed.

- --src-files <filenames>: A list of space-separated source file names added to the makefile. The list of file names is terminated by the next option or the end of the command line. Multiple --src-files options are allowed.
- --src-rdir <directory>: Same as --src-dir option but recursively searches for source files in or under <filepath>. Multiple --src-rdir options are allowed. You can mix --src-rdir with --src-dir options.
- --use-lib-dir <directory>: Path to a dependent library directory. The library directory must contain a makefile fragment called public.mk. Multiple --use-lib-dir options are allowed.
- --verbose: Outputs verbose, and default information about the command's operation to stdout.
- --version: Displays the version of this command and exits with a zero exit status.

Description

The nios2-lib-generate-makefile command generates a private library makefile called Makefile, and a public makefile, called public.mk. The path to a BSP created by nios2-bsp-generate-files is an optional command line option.

For more details about this command, use the --extended-help option to display comprehensive usage information.

nios2-c2h-generate-makefile

Usage

```
nios2-c2h-generate-makefile --sopc=<sopc> [OPTIONS]
```

Options

- --sopc: The path to the SOPC Builder system file (**.sopc**).
- --app-dir: Directory to place the application Makefile and executable file. If omitted, it defaults to the current directory.
- --accelerator: Specifies a function to be accelerated
- --enable_quartus: Building the app compiles the associated Quartus® II project. Defaults to 0.
- --analyze_only: Disables hardware generation, SOPC Builder system generation, and Quartus II compilation for all accelerators in the app. Building the project with this option only updates the report files. Defaults to 0.
- --use_existing_accelerators: Disables all hardware generation steps. The build behaves as if c2h.mk did not exist, with the exception of possible accelerator linking as specified in the --accelerator option. Defaults to 0.

Description

The **nios2-c2h-generate-makefile** command creates a C2H makefile fragment, **c2h.mk**, that specifies all accelerators and accelerator options for an application.

This command creates a new **c2h.mk** each time it is called, overwriting the existing **c2h.mk**

The --accelerator argument specifies a function to be accelerated. This argument accepts up to four comma-separated values:

- Target function name
- Target function file
- Link hardware accelerator instead of original software. 1 or 0. Defaults to 1.
- Flush data cache before each call. 1 or 0. Defaults to 1.

Example

```
nios2-c2h-generate-makefile \
    --sopc=../../NiosII_stratix_1s40_standard.sopc \
    --app_dir=./ \
    --accelerator=filter,filter.c \
    --accelerator=xmath,../../../xmath.c,1,0 \
    --use_existing_accelerators
```

nios2-bsp

Usage

```
nios2-bsp <bsp type> <bsp dir> [<sopc>] [OPTION]...
```

Options

- <bsp-type>: `hal` or `ucosii`
- <bsp-dir>: Path to the board support package (BSP) directory.
- <sopc>: The path to the SOPC Builder system file or its directory.
- <option>: Options to override defaults.

Description

The **nios2-bsp** bash script calls **nios2-bsp-create-settings** or **nios2-bsp-update-settings** to create or update a BSP settings file, and the **nios2-bsp-generate-files** command to create the BSP files. The Nios II Embedded Design Suite (EDS) supports the following BSP types:

- `hal`
- `ucosii`

BSP type names are case insensitive.

This utility produces a BSP of type <bsp-type> in <bsp-dir>. If the BSP does not exist, it is created. If the BSP already exists, it is updated to be consistent with the associated SOPC Builder system.

The default Tcl script is used to set the following system-dependent settings:

- `stdio` character device
- System timer device
- Default linker memory
- Boot loader status (enabled or disabled)

If the BSP already exists, **nios2-bsp** overwrites these system-dependent settings.

The default Tcl script resides at:

*<Nios II EDS install path>***/sdk2/bin/bsp-set-defaults.tcl**

When creating a new BSP, this utility runs **nios2-bsp-create-settings**, which creates **settings.bsp** in <bsp-dir>.

When updating an existing BSP, this utility runs **nios2-bsp-update-settings**, which updates **settings.bsp** in <bsp-dir>.

After creating or updating the **settings.bsp** file, this utility runs **nios2-bsp-generate-files**, which generates files in <bsp-dir>

Required arguments:

- <bsp-type>: Specifies the type of BSP. This argument is ignored when updating a BSP. This argument is case insensitive. **nios2-bsp** supports the following values of <bsp-type>:
  - `hal`
  - `ucosii`
- <bsp-dir>: Path to the BSP directory. Use "." to specify the current directory.

Optional arguments:

- <sopc>: The path name of the SOPC Builder system file. Alternatively, specify a directory containing an SOPC Builder system file. In the latter case, the tool finds a file with the extension **.sopc**. This argument is ignored when updating a BSP. If you omit this argument, it defaults to the current directory.
- <option>: Options to override defaults. **nios2-bsp** passes most options to **nios2-bsp-create-settings** or **nios2-bsp-update-settings**. It also passes the `--silent`, `--verbose`, `--debug`, and `--log` options to **nios2-bsp-generate-files**.

  nios2-bsp passes the following options to the default Tcl script:

  - `--default_stdio <device>|none|DONT_CHANGE`
    Specifies stdio device.

  - `--default_sys_timer <device>|none|DONT_CHANGE`
    Specifies system timer device.

  - `--default_memory_regions DONT_CHANGE`
    Suppresses creation of new default memory regions when updating a BSP. Do not use this option when creating a new BSP.

  - `--default_sections_mapping <region>|DONT_CHANGE`
    Specifies the memory region for the default sections.

  - `--use_bootloader 0|1|DONT_CHANGE`
    Specifies whether a boot loader is required.

  On a pre-existing BSP, the value `DONT_CHANGE` prevents associated settings from changing their current value.

☞ The "- -" prefix is stripped when the option is passed to the underlying utility.

Settings

Settings are central to how you create and work with BSPs, software packages and device drivers. You control the characteristics of your project by controlling the settings. The settings determine things like whether or not an operating system is supported, and what device drivers and other packages are included.

Every example in the *Introduction to the Nios II Software Build Tools* and *Using the Nios II Software Build Tools* chapters of the *Nios II Software Developer's Handbook* involves specifying or manipulating settings. Sometimes these settings are specified automatically, by scripts such as **create-this-bsp**, and sometimes explicitly, with Tcl commands. Either way, settings are always involved.

This section contains a complete list of available settings for BSPs and for Altera-supported device drivers and software components. It does not include settings for device drivers or software components furnished by Altera partners or other third parties. If you are using a third-party driver or component, refer to the supplier's documentation.

Settings used in the Nios II software build tools are organized hierarchically, for logical grouping and to avoid name space conflicts. Each setting's position in the hierarchy is indicated by one or more prefixes. A prefix is an identifier followed by a dot (`.`). For example, `hal.enable_c_plus_plus` is a hardware abstraction layer (HAL) setting, while `ucosii.event_flag.os_flag_accept_en` is a member of the event flag subgroup of MicroC/OS-II settings.

Setting names are case-insensitive.

Overview of BSP Settings

There are several types of BSP settings, as shown in Table 14–4.

Table 14–4. Types of BSP Settings

Setting Type	Description
Altera HAL	Settings available with the Altera HAL BSP or any BSP based on it (for example, Micrium MicroC/OS-II).
Micrium MicroC/OS-II	Settings available if using the Micrium MicroC/OS-II BSP. All Altera HAL BSP settings are also available because MicroC/OS-II is based on the Altera HAL BSP.
Altera BSP Makefile Generator	Settings available if using the Altera BSP makefile generator (generates the **Makefile** and **public.mk** files). These settings control the contents of makefile variables. This generator is always present in Altera HAL BSPs or any BSPs based on the Altera HAL.
Altera BSP Linker Script Generator	Settings available if using the Altera BSP linker script generator (generates the **linker.x** and **linker.h** files). This generator is always present in Altera HAL BSPs or any BSPs based on the Altera HAL.

A BSP setting consist of a name/value pair.

Do not confuse BSP settings with BSP Tcl commands. This section covers BSP settings, including their types, meanings, and legal values. The Tcl commands, which are the tools for manipulating the settings, are covered in "Tcl Commands for BSP Settings" on page 14–160.

Overview of Component and Driver Settings

The Nios II EDS includes a number of standard software components and device drivers. All of the software components, and several drivers, have settings that you can manipulate when creating a BSP. This section lists the packages and drivers that have settings.

Altera Host-Based File System Settings

The Altera host-based file system has one setting. If the Altera host-based file system is enabled, you must debug (not run) applications based on the BSP for the host-based file system to function. The host-based file system relies on the GNU debugger running on the host to provide host-based file operations.

Use the following BSP Tcl command to enable the host-based file system software package:

```
enable_sw_package altera_hostfs
```

Altera Read-Only Zip File System Settings

The Altera read-only Zip file system has several settings. If the read-only Zip file system is enabled, it adds `-DUSE_RO_ZIPFS` to `ALT_CPPFLAGS` in **public.mk**.

Use the following BSP Tcl command to enable the read-only Zip file system software package:

```
enable_sw_package altera_ro_zipfs
```

Altera NicheStack® TCP/IP - Nios II Edition Stack Settings

The Altera NicheStack® TCP/IP - Nios II Edition transmission control protocol/Internet protocol (TCP/IP) networking stack has several settings. The stack is only available in MicroC/OS-II BSPs. If the NicheStack TCP/IP stack is enabled, it adds `-DALT_INICHE` to `ALT_CPPFLAGS` in **public.mk**.

Use the following BSP Tcl command to enable the NicheStack TCP/IP networking stack software package:

```
enable_sw_package altera_iniche
```

Altera Avalon-MM JTAG UART Driver Settings

The Altera Avalon Memory-Mapped Joint Test Action Group (JTAG) universal asynchronous receiver/transmitter (UART) driver settings are available if the `altera_avalon_jtag_uart` driver is present. By default, this driver is used if your SOPC Builder system has an `altera_avalon_jtag_uart` module connected to it.

Altera Avalon-MM UART Driver Settings

The Altera Avalon-MM UART driver settings are available if the `altera_avalon_uart` driver is present. By default, this driver is used if your SOPC Builder system has an `altera_avalon_uart` module connected to it.

Settings Reference

This section lists all settings for BSPs, software packages and device drivers.

hal.sys_clk_timer

Identifier	none
Type	Unquoted string
Default Value	none
Destination File	none
Description	Slave descriptor of the system clock timer device. This device provides a periodic interrupt ("tick") and is typically required for RTOS use. This setting defines the value of ALT_SYS_CLK in system.h.
Restrictions	none

hal.timestamp_timer

Identifier	none
Type	Unquoted string
Default Value	none
Destination File	none
Description	Slave descriptor of timestamp timer device. This device is used by Altera HAL timestamp drivers for high-resolution time measurement. This setting defines the value of ALT_TIMESTAMP_CLK in system.h.
Restrictions	none

hal.max_file_descriptors

Identifier	none
Type	Decimal number
Default Value	32
Destination File	none
Description	Determines the number of file descriptors statically allocated.
Restrictions	If hal.enable_lightweight_device_driver_api is true, there are no file descriptors so this setting is ignored. If hal.enable_lightweight_device_driver_api is false, this setting must be at least 4 because HAL needs a file descriptor for /dev/null, /dev/stdin, /dev/stdout, and /dev/stderr. This setting defines the value of ALT_MAX_FD in system.h.

ucosii.os_max_tasks

Identifier	OS_MAX_TASKS
Type	Decimal number
Default Value	10
Destination File	system_h_define
Description	Maximum number of tasks
Restrictions	none
Enabled	false

ucosii.os_lowest_prio

Identifier	OS_LOWEST_PRIO
Type	Decimal number
Default Value	20
Destination File	system_h_define
Description	Lowest assignable priority
Restrictions	none
Enabled	false

ucosii.os_thread_safe_newlib

Identifier	OS_THREAD_SAFE_NEWLIB
Type	Boolean assignment
Default Value	1
Destination File	system_h_define
Description	Thread safe C library
Restrictions	none
Enabled	false

ucosii.miscellaneous.os_arg_chk_en

Identifier	OS_ARG_CHK_EN
Type	Boolean assignment
Default Value	1
Destination File	system_h_define
Description	Enable argument checking
Restrictions	none
Enabled	false

ucosii.miscellaneous.os_cpu_hooks_en

Identifier	OS_CPU_HOOKS_EN
Type	Boolean assignment
Default Value	1
Destination File	system_h_define
Description	Enable uCOS-II hooks
Restrictions	none
Enabled	false

ucosii.miscellaneous.os_debug_en

Identifier	OS_DEBUG_EN
Type	Boolean assignment
Default Value	1
Destination File	system_h_define
Description	Enable debug variables
Restrictions	none
Enabled	false

ucosii.miscellaneous.os_sched_lock_en

Identifier	OS_SCHED_LOCK_EN
Type	Boolean assignment
Default Value	1
Destination File	system_h_define
Description	Include code for OSSchedLock() and OSSchedUnlock()
Restrictions	none
Enabled	false

ucosii.miscellaneous.os_task_stat_en

Identifier	OS_TASK_STAT_EN
Type	Boolean assignment
Default Value	1
Destination File	system_h_define
Description	Enable statistics task
Restrictions	none
Enabled	false

ucosii.miscellaneous.os_task_stat_stk_chk_en

Identifier	OS_TASK_STAT_STK_CHK_EN
Type	Boolean assignment
Default Value	1
Destination File	system_h_define
Description	Check task stacks from statistics task
Restrictions	none
Enabled	false

ucosii.miscellaneous.os_tick_step_en

Identifier	OS_TICK_STEP_EN
Type	Boolean assignment
Default Value	1
Destination File	system_h_define
Description	Enable tick stepping feature for uCOS-View
Restrictions	none
Enabled	false

ucosii.miscellaneous.os_event_name_size

Identifier	OS_EVENT_NAME_SIZE
Type	Decimal number
Default Value	32
Destination File	system_h_define
Description	Size of name of Event Control Block groups
Restrictions	none
Enabled	false

ucosii.miscellaneous.os_max_events

Identifier	OS_MAX_EVENTS
Type	Decimal number
Default Value	60
Destination File	system_h_define
Description	Maximum number of event control blocks
Restrictions	none
Enabled	false

ucosii.miscellaneous.os_task_idle_stk_size

Identifier	OS_TASK_IDLE_STK_SIZE
Type	Decimal number
Default Value	512
Destination File	system_h_define
Description	Idle task stack size
Restrictions	none
Enabled	false

ucosii.miscellaneous.os_task_stat_stk_size

Identifier	OS_TASK_STAT_STK_SIZE
Type	Decimal number
Default Value	512
Destination File	system_h_define
Description	Statistics task stack size
Restrictions	none
Enabled	false

ucosii.task.os_task_change_prio_en

Identifier	OS_TASK_CHANGE_PRIO_EN
Type	Boolean assignment
Default Value	1
Destination File	system_h_define
Description	Include code for OSTaskChangePrio()
Restrictions	none
Enabled	false

ucosii.task.os_task_create_en

Identifier	OS_TASK_CREATE_EN
Type	Boolean assignment
Default Value	1
Destination File	system_h_define
Description	Include code for OSTaskCreate()
Restrictions	none
Enabled	false

ucosii.task.os_task_create_ext_en

Identifier	OS_TASK_CREATE_EXT_EN
Type	Boolean assignment
Default Value	1
Destination File	system_h_define
Description	Include code for OSTaskCreateExt()
Restrictions	none
Enabled	false

ucosii.task.os_task_del_en

Identifier	OS_TASK_DEL_EN
Type	Boolean assignment
Default Value	1
Destination File	system_h_define
Description	Include code for OSTaskDel()
Restrictions	none
Enabled	false

ucosii.task.os_task_name_size

Identifier	OS_TASK_NAME_SIZE
Type	Decimal number
Default Value	32
Destination File	system_h_define
Description	Size of task name
Restrictions	none
Enabled	false

ucosii.task.os_task_profile_en

Identifier	OS_TASK_PROFILE_EN
Type	Boolean assignment
Default Value	1
Destination File	system_h_define
Description	Include data structure for run-time task profiling
Restrictions	none
Enabled	false

ucosii.task.os_task_query_en

Identifier	OS_TASK_QUERY_EN
Type	Boolean assignment
Default Value	1
Destination File	system_h_define
Description	Include code for OSTaskQuery
Restrictions	none
Enabled	false

ucosii.task.os_task_suspend_en

Identifier	OS_TASK_SUSPEND_EN
Type	Boolean assignment
Default Value	1
Destination File	system_h_define
Description	Include code for OSTaskSuspend() and OSTaskResume()
Restrictions	none
Enabled	false

ucosii.task.os_task_sw_hook_en

Identifier	OS_TASK_SW_HOOK_EN
Type	Boolean assignment
Default Value	1
Destination File	system_h_define
Description	Include code for OSTaskSwHook()
Restrictions	none
Enabled	false

ucosii.time.os_time_tick_hook_en

Identifier	OS_TIME_TICK_HOOK_EN
Type	Boolean assignment
Default Value	1
Destination File	system_h_define
Description	Include code for OSTimeTickHook()
Restrictions	none
Enabled	false

ucosii.time.os_time_dly_resume_en

Identifier	OS_TIME_DLY_RESUME_EN
Type	Boolean assignment
Default Value	1
Destination File	system_h_define
Description	Include code for OSTimeDlyResume()
Restrictions	none
Enabled	false

ucosii.time.os_time_dly_hmsm_en

Identifier	OS_TIME_DLY_HMSM_EN
Type	Boolean assignment
Default Value	1
Destination File	system_h_define
Description	Include code for OSTimeDlyHMSM()
Restrictions	none
Enabled	false

ucosii.time.os_time_get_set_en

Identifier	OS_TIME_GET_SET_EN
Type	Boolean assignment
Default Value	1
Destination File	system_h_define
Description	Include code for OSTimeGet and OSTimeSet()
Restrictions	none
Enabled	false

ucosii.os_flag_en

Identifier	OS_FLAG_EN
Type	Boolean
Default Value	1
Destination File	system_h_define
Description	Enable code for Event Flags (used by UART and JTAG UART drivers)
Restrictions	none
Enabled	false

ucosii.event_flag.os_flag_wait_clr_en

Identifier	OS_FLAG_WAIT_CLR_EN
Type	Boolean
Default Value	1
Destination File	system_h_define
Description	Include code for Wait on Clear Event Flags
Restrictions	Requires os_flag_en set to true
Enabled	false

ucosii.event_flag.os_flag_accept_en

Identifier	OS_FLAG_ACCEPT_EN
Type	Boolean
Default Value	1
Destination File	system_h_define
Description	Include code for OSFlagAccept()
Restrictions	Requires os_flag_en set to true
Enabled	false

ucosii.event_flag.os_flag_del_en

Identifier	OS_FLAG_DEL_EN
Type	Boolean
Default Value	1
Destination File	system_h_define
Description	Include code for OSFlagDel()
Restrictions	Requires os_flag_en set to true
Enabled	false

ucosii.event_flag.os_flag_query_en

Identifier	OS_FLAG_QUERY_EN
Type	Boolean
Default Value	1
Destination File	system_h_define
Description	Include code for OSFlagQuery()
Restrictions	Requires os_flag_en set to true
Enabled	false

ucosii.event_flag.os_flag_name_size

Identifier	OS_FLAG_NAME_SIZE
Type	Decimal number
Default Value	32
Destination File	system_h_define
Description	Size of name of Event Flags group
Restrictions	Requires os_flag_en set to true
Enabled	false

ucosii.event_flag.os_flags_nbits

Identifier	OS_FLAGS_NBITS
Type	Decimal number
Default Value	16
Destination File	system_h_define
Description	Event Flag bits (8,16,32)
Restrictions	Requires os_flag_en set to true
Enabled	false

ucosii.event_flag.os_max_flags

Identifier	OS_MAX_FLAGS
Type	Decimal number
Default Value	20
Destination File	system_h_define
Description	Maximum number of Event Flags groups
Restrictions	Requires os_flag_en set to true
Enabled	false

ucosii.os_mutex_en

Identifier	OS_MUTEX_EN
Type	Boolean
Default Value	1
Destination File	system_h_define
Description	Enable code for Mutex Semaphores
Restrictions	none
Enabled	false

ucosii.mutex.os_mutex_accept_en

Identifier	OS_MUTEX_ACCEPT_EN
Type	Boolean
Default Value	1
Destination File	system_h_define
Description	Include code for OSMutexAccept()
Restrictions	Requires os_mutex_en set to true
Enabled	false

ucosii.mutex.os_mutex_del_en

Identifier	OS_MUTEX_DEL_EN
Type	Boolean
Default Value	1
Destination File	system_h_define
Description	Include code for OSMutexDel()
Restrictions	Requires os_mutex_en set to true
Enabled	false

ucosii.mutex.os_mutex_query_en

Identifier	OS_MUTEX_QUERY_EN
Type	Boolean
Default Value	1
Destination File	system_h_define
Description	Include code for OSMutexQuery
Restrictions	Requires os_mutex_en set to true
Enabled	false

ucosii.os_sem_en

Identifier	OS_SEM_EN
Type	Boolean
Default Value	1
Destination File	system_h_define
Description	Enable code for semaphores (This must be enabled, it is required by the HAL)
Restrictions	none
Enabled	false

ucosii.semaphore.os_sem_accept_en

Identifier	OS_SEM_ACCEPT_EN
Type	Boolean
Default Value	1
Destination File	system_h_define
Description	Include code for OSSemAccept()
Restrictions	Requires os_sem_en set to true
Enabled	false

ucosii.semaphore.os_sem_set_en

Identifier	OS_SEM_SET_EN
Type	Boolean
Default Value	1
Destination File	system_h_define
Description	Include code for OSSemSet()
Restrictions	Requires os_sem_en set to true
Enabled	false

ucosii.semaphore.os_sem_del_en

Identifier	OS_SEM_DEL_EN
Type	Boolean
Default Value	1
Destination File	system_h_define
Description	Include code for OSSemDel()
Restrictions	Requires os_sem_en set to true
Enabled	false

ucosii.semaphore.os_sem_query_en

Identifier	OS_SEM_QUERY_EN
Type	Boolean
Default Value	1
Destination File	system_h_define
Description	Include code for OSSemQuery()
Restrictions	Requires os_sem_en set to true
Enabled	false

ucosii.os_mbox_en

Identifier	OS_MBOX_EN
Type	Boolean
Default Value	1
Destination File	system_h_define
Description	Enable code for mailboxes
Restrictions	none
Enabled	false

ucosii.mailbox.os_mbox_accept_en

Identifier	OS_MBOX_ACCEPT_EN
Type	Boolean
Default Value	1
Destination File	system_h_define
Description	Include code for OSMboxAccept()
Restrictions	Requires os_mbox_en set to true
Enabled	false

ucosii.mailbox.os_mbox_del_en

Identifier	OS_MBOX_DEL_EN
Type	Boolean
Default Value	1
Destination File	system_h_define
Description	Include code for OSMboxDel()
Restrictions	Requires os_mbox_en set to true
Enabled	false

ucosii.mailbox.os_mbox_post_en

Identifier	OS_MBOX_POST_EN
Type	Boolean
Default Value	1
Destination File	system_h_define
Description	Include code for OSMboxPost()
Restrictions	Requires os_mbox_en set to true
Enabled	false

ucosii.mailbox.os_mbox_post_opt_en

Identifier	OS_MBOX_POST_OPT_EN
Type	Boolean
Default Value	1
Destination File	system_h_define
Description	Include code for OSMboxPostOpt()
Restrictions	Requires os_mbox_en set to true
Enabled	false

ucosii.mailbox.os_mbox_query_en

Identifier	OS_MBOX_QUERY_EN
Type	Boolean
Default Value	1
Destination File	system_h_define
Description	Include code for OSMboxQuery()
Restrictions	Requires os_mbox_en set to true
Enabled	false

ucosii.os_q_en

Identifier	OS_Q_EN
Type	Boolean
Default Value	1
Destination File	system_h_define
Description	Enable code for Queues
Restrictions	none
Enabled	false

ucosii.queue.os_q_accept_en

Identifier	OS_Q_ACCEPT_EN
Type	Boolean
Default Value	1
Destination File	system_h_define
Description	Include code for OSQAccept()
Restrictions	Requires os_q_en set to true
Enabled	false

ucosii.queue.os_q_del_en

Identifier	OS_Q_DEL_EN
Type	Boolean
Default Value	1
Destination File	system_h_define
Description	Include code for OSQDel()
Restrictions	Requires os_q_en set to true
Enabled	false

ucosii.queue.os_q_flush_en

Identifier	OS_Q_FLUSH_EN
Type	Boolean
Default Value	1
Destination File	system_h_define
Description	Include code for OSQFlush()
Restrictions	Requires os_q_en set to true
Enabled	false

ucosii.queue.os_q_post_en

Identifier	OS_Q_POST_EN
Type	Boolean
Default Value	1
Destination File	system_h_define
Description	Include code of OSQFlush()
Restrictions	Requires os_q_en set to true
Enabled	false

ucosii.queue.os_q_post_front_en

Identifier	OS_Q_POST_FRONT_EN
Type	Boolean
Default Value	1
Destination File	system_h_define
Description	Include code for OSQPostFront()
Restrictions	Requires os_q_en set to true
Enabled	false

ucosii.queue.os_q_post_opt_en

Identifier	OS_Q_POST_OPT_EN
Type	Boolean
Default Value	1
Destination File	system_h_define
Description	Include code for OSQPostOpt()
Restrictions	Requires os_q_en set to true
Enabled	false

ucosii.queue.os_q_query_en

Identifier	OS_Q_QUERY_EN
Type	Boolean
Default Value	1
Destination File	system_h_define
Description	Include code for OSQQuery()
Restrictions	Requires os_q_en set to true
Enabled	false

ucosii.queue.os_max_qs

Identifier	OS_MAX_QS
Type	Decimal number
Default Value	20
Destination File	system_h_define
Description	Maximum number of Queue Control Blocks
Restrictions	Requires os_q_en set to true
Enabled	false

ucosii.os_mem_en

Identifier	OS_MEM_EN
Type	Boolean
Default Value	1
Destination File	system_h_define
Description	Enable code for memory management
Restrictions	none
Enabled	false

ucosii.memory.os_mem_query_en

Identifier	OS_MEM_QUERY_EN
Type	Boolean
Default Value	1
Destination File	system_h_define
Description	Include code for OSMemQuery()
Restrictions	Requires os_mem_en set to true
Enabled	false

ucosii.memory.os_mem_name_size

Identifier	OS_MEM_NAME_SIZE
Type	Decimal number
Default Value	32
Destination File	system_h_define
Description	Size of memory partition name
Restrictions	Requires os_mem_en set to true
Enabled	false

ucosii.memory.os_max_mem_part

Identifier	OS_MAX_MEM_PART
Type	Decimal number
Default Value	60
Destination File	system_h_define
Description	Maximum number of memory partitions
Restrictions	Requires os_mem_en set to true
Enabled	false

ucosii.os_tmr_en

Identifier	OS_TMR_EN
Type	Boolean
Default Value	0
Destination File	system_h_define
Description	Enable code for timers
Restrictions	none
Enabled	false

ucosii.timer.os_task_tmr_stk_size

Identifier	OS_TASK_TMR_STK_SIZE
Type	Decimal number
Default Value	512
Destination File	system_h_define
Description	Stack size for timer task
Restrictions	Requires os_tmr_en set to true
Enabled	false

ucosii.timer.os_task_tmr_prio

Identifier	OS_TASK_TMR_PRIO
Type	Decimal number
Default Value	2
Destination File	system_h_define
Description	Priority of timer task (0=highest)
Restrictions	Requires os_tmr_en set to true
Enabled	false

ucosii.timer.os_tmr_cfg_max

Identifier	OS_TMR_CFG_MAX
Type	Decimal number
Default Value	16
Destination File	system_h_define
Description	Maximum number of timers
Restrictions	Requires os_tmr_en set to true
Enabled	false

ucosii.timer.os_tmr_cfg_name_size

Identifier	OS_TMR_CFG_NAME_SIZE
Type	Decimal number
Default Value	16
Destination File	system_h_define
Description	Size of timer name
Restrictions	Requires os_tmr_en set to true
Enabled	false

ucosii.timer.os_tmr_cfg_ticks_per_sec

Identifier	OS_TMR_CFG_TICKS_PER_SEC
Type	Decimal number
Default Value	10
Destination File	system_h_define
Description	Rate at which timer management task runs (Hz)
Restrictions	Requires os_tmr_en set to true
Enabled	false

ucosii.timer.os_tmr_cfg_wheel_size

Identifier	OS_TMR_CFG_WHEEL_SIZE
Type	Decimal number
Default Value	2
Destination File	system_h_define
Description	Size of timer wheel (number of spokes)
Restrictions	Requires os_tmr_en set to true
Enabled	false

altera_avalon_jtag_uart_driver.enable_small_driver

Identifier	ALTERA_AVALON_JTAG_UART_SMALL
Type	Boolean definition
Default Value	false
Destination File	public_mk_define
Description	Small-footprint (polled mode) driver
Restrictions	none
Enabled	false

altera_avalon_uart_driver.enable_small_driver

Identifier	ALTERA_AVALON_UART_SMALL
Type	Boolean definition
Default Value	false
Destination File	public_mk_define
Description	Small-footprint (polled mode) driver
Restrictions	none
Enabled	false

altera_avalon_uart_driver.enable_ioctl

Identifier	ALTERA_AVALON_UART_USE_IOCTL
Type	Boolean definition
Default Value	false
Destination File	public_mk_define
Description	Enable driver ioctl() support
Restrictions	none
Enabled	false

altera_iniche.iniche_default_if

Identifier	INICHE_DEFAULT_IF
Type	Quoted string
Default Value	NEEDS_SPECIFICATION
Destination File	system_h_define
Description	Default MAC interface (This must be assigned before building)
Restrictions	none
Enabled	false

altera_iniche.enable_dhcp_client

Identifier	DHCP_CLIENT
Type	Boolean definition
Default Value	true
Destination File	system_h_define
Description	Use DHCP to automatically assign IP address
Restrictions	none
Enabled	false

altera_iniche.enable_ip_fragments

Identifier	IP_FRAGMENTS
Type	Boolean definition
Default Value	true
Destination File	system_h_define
Description	Reassemble IP packet fragments
Restrictions	none
Enabled	false

altera_iniche.enable_include_tcp

Identifier	INCLUDE_TCP
Type	Boolean definition
Default Value	true
Destination File	system_h_define
Description	Enable TCP protocol
Restrictions	none
Enabled	false

altera_iniche.enable_tcp_zerocopy

Identifier	TCP_ZEROCOPY
Type	Boolean definition
Default Value	false
Destination File	system_h_define
Description	Use TCP zero-copy
Restrictions	none
Enabled	false

altera_iniche.enable_net_stats

Identifier	NET_STATS
Type	Boolean definition
Default Value	false
Destination File	system_h_define
Description	Enable statistics
Restrictions	none
Enabled	false

altera_ro_zipfs.ro_zipfs_name

Identifier	ALTERA_RO_ZIPFS_NAME
Type	Quoted string
Default Value	/mnt/rozipfs
Destination File	system_h_define
Description	Mount point
Restrictions	none
Enabled	false

altera_ro_zipfs.ro_zipfs_offset

Identifier	ALTERA_RO_ZIPFS_OFFSET
Type	Hexadecimal number
Default Value	0x100000
Destination File	system_h_define
Description	Offset of file system from base of flash
Restrictions	none
Enabled	false

altera_ro_zipfs.ro_zipfs_base

Identifier	ALTERA_RO_ZIPFS_BASE
Type	Hexadecimal number
Default Value	0x0
Destination File	system_h_define
Description	Base address of flash memory device
Restrictions	none
Enabled	false

altera_hostfs.hostfs_name

Identifier	ALTERA_HOSTFS_NAME
Type	Quoted string
Default Value	/mnt/host
Destination File	system_h_define
Description	Mount point
Restrictions	none
Enabled	false

hal.linker.exception_stack_memory_region_name

Identifier	none
Type	Unquoted string
Default Value	none
Destination File	none
Description	Name of the memory region that is divided up to create the exception_stack region.
Restrictions	Only used if hal.linker.enable_exception_stack is true.

hal.linker.allow_code_at_reset

Identifier	none
Type	Boolean assignment
Default Value	0
Destination File	none
Description	Indicates if initialization code is allowed at the reset address. If true, defines the macro ALT_ALLOW_CODE_AT_RESET in linker.h.
Restrictions	This setting is typically false if an external bootloader (e.g. flash bootloader) is present.

hal.linker.enable_alt_load

Identifier	none
Type	Boolean assignment
Default Value	0
Destination File	none
Description	Enables the alt_load() facility. The alt_load() facility copies sections from the .text memory into RAM. If true, this setting sets up the VMA/LMA of sections in linker.x to allow them to be loaded into the .text memory.
Restrictions	This setting is typically false if an external bootloader (e.g. flash bootloader) is present.

hal.linker.enable_alt_load_copy_rwdata

Identifier	none
Type	Boolean assignment
Default Value	0
Destination File	none
Description	Causes the alt_load() facility to copy the .rwdata section. If true, this setting defines the macro ALT_LOAD_COPY_RWDATA in linker.h.
Restrictions	none

hal.linker.enable_alt_load_copy_rodata

Identifier	none
Type	Boolean assignment
Default Value	0
Destination File	none
Description	Causes the alt_load() facility to copy the .rodata section. If true, this setting defines the macro ALT_LOAD_COPY_RODATA in linker.h.
Restrictions	none

hal.linker.enable_alt_load_copy_exceptions

Identifier	none
Type	Boolean assignment
Default Value	0
Destination File	none
Description	Causes the alt_load() facility to copy the .exceptions section. If true, this setting defines the macro ALT_LOAD_COPY_EXCEPTIONS in linker.h.
Restrictions	none

hal.linker.enable_exception_stack

Identifier	none
Type	Boolean assignment
Default Value	0
Destination File	none
Description	Enables use of a separate exception stack. If true, defines the macro ALT_EXCEPTION_STACK in linker.h, adds a memory region called exception_stack to linker.x, and provides the symbols __alt_exception_stack_pointer and __alt_exception_stack_limit in linker.x.
Restrictions	The hal.linker.exception_stack_size and hal.linker.exception_stack_memory_region_name settings must also be valid. This setting must be false for MicroC/OS-II BSPs.

hal.linker.exception_stack_size

Identifier	none
Type	Decimal number
Default Value	1024
Destination File	none
Description	Size of the exception stack in bytes.
Restrictions	Only used if hal.linker.enable_exception_stack is true.

hal.make.build_pre_process

Identifier	BUILD_PRE_PROCESS
Type	Unquoted string
Default Value	none
Destination File	makefile_variable
Description	Command executed before BSP built.
Restrictions	none

hal.make.ar_pre_process

Identifier	AR_PRE_PROCESS
Type	Unquoted string
Default Value	none
Destination File	makefile_variable
Description	Command executed before archiver execution.
Restrictions	none

hal.make.bsp_cflags_defined_symbols

Identifier	BSP_CFLAGS_DEFINED_SYMBOLS
Type	Unquoted string
Default Value	-DALT_DEBUG
Destination File	makefile_variable
Description	Preprocessor macros to define. A macro definition in this setting has the same effect as a "#define" in source code. Adding "-DALT_DEBUG" to this setting has the same effect as "#define ALT_DEBUG" in a souce file. Adding "-DFOO=1" to this setting is equivalent to the macro "#define FOO 1" in a source file. Macros defined with this setting are applied to all .S, .c, and C++ files in the BSP. This setting defines the value of BSP_CFLAGS_DEFINED_SYMBOLS in the BSP Makefile.
Restrictions	none

hal.make.ar_post_process

Identifier	AR_POST_PROCESS
Type	Unquoted string
Default Value	none
Destination File	makefile_variable
Description	Command executed after archiver execution.
Restrictions	none

hal.make.as

Identifier	AS
Type	Unquoted string
Default Value	nios2-elf-gcc
Destination File	makefile_variable
Description	Assembler command. Note that CC is used for .S files.
Restrictions	none

hal.make.build_post_process

Identifier	BUILD_POST_PROCESS
Type	Unquoted string
Default Value	none
Destination File	makefile_variable
Description	Command executed after BSP built.
Restrictions	none

hal.make.bsp_cflags_debug

Identifier	BSP_CFLAGS_DEBUG
Type	Unquoted string
Default Value	-g
Destination File	makefile_variable
Description	C/C++ compiler debug level. "-g" provides the default set of debug symbols typically required to debug a typical application. Omitting "-g" removes debug symbols from the ELF. This setting defines the value of BSP_CFLAGS_DEBUG in Makefile.
Restrictions	none

hal.make.ar

Identifier	AR
Type	Unquoted string
Default Value	nios2-elf-ar
Destination File	makefile_variable
Description	Archiver command. Creates library files.
Restrictions	none

hal.make.rm

Identifier	RM
Type	Unquoted string
Default Value	rm -f
Destination File	makefile_variable
Description	Command used to remove files during 'clean' target.
Restrictions	none

hal.make.cxx_pre_process

Identifier	CXX_PRE_PROCESS
Type	Unquoted string
Default Value	none
Destination File	makefile_variable
Description	Command executed before each C++ file is compiled.
Restrictions	none

hal.make.bsp_cflags_warnings

Identifier	BSP_CFLAGS_WARNINGS
Type	Unquoted string
Default Value	-Wall
Destination File	makefile_variable
Description	C/C++ compiler warning level. "-Wall" is commonly used.This setting defines the value of BSP_CFLAGS_WARNINGS in Makefile.
Restrictions	none

hal.make.bsp_arflags

Identifier	BSP_ARFLAGS
Type	Unquoted string
Default Value	-src
Destination File	makefile_variable
Description	Custom flags only passed to the archiver. This content of this variable is directly passed to the archiver rather than the more standard "ARFLAGS". The reason for this is that GNU Make assumes some default content in ARFLAGS.This setting defines the value of BSP_ARFLAGS in Makefile.
Restrictions	none

hal.make.bsp_cflags_optimization

Identifier	BSP_CFLAGS_OPTIMIZATION
Type	Unquoted string
Default Value	-O0
Destination File	makefile_variable
Description	C/C++ compiler optimization level. "-O0" = no optimization,"-O2" = "normal" optimization, etc. "-O0" is recommended for code that you want to debug since compiler optimization can remove variables and produce non-sequential execution of code while debugging. This setting defines the value of BSP_CFLAGS_OPTIMIZATION in Makefile.
Restrictions	none

hal.make.as_post_process

Identifier	AS_POST_PROCESS
Type	Unquoted string
Default Value	none
Destination File	makefile_variable
Description	Command executed after each assembly file is compiled.
Restrictions	none

hal.make.cc_pre_process

Identifier	CC_PRE_PROCESS
Type	Unquoted string
Default Value	none
Destination File	makefile_variable
Description	Command executed before each .c/.S file is compiled.
Restrictions	none

hal.make.bsp_asflags

Identifier	BSP_ASFLAGS
Type	Unquoted string
Default Value	-Wa,-gdwarf2
Destination File	makefile_variable
Description	Custom flags only passed to the assembler. This setting defines the value of BSP_ASFLAGS in Makefile.
Restrictions	none

hal.make.as_pre_process

Identifier	AS_PRE_PROCESS
Type	Unquoted string
Default Value	none
Destination File	makefile_variable
Description	Command executed before each assembly file is compiled.
Restrictions	none

hal.make.bsp_cflags_undefined_symbols

Identifier	BSP_CFLAGS_UNDEFINED_SYMBOLS
Type	Unquoted string
Default Value	none
Destination File	makefile_variable
Description	Preprocessor macros to undefine. Undefined macros are similar to defined macros, but replicate the "#undef" directive in source code. To undefine the macro FOO use the syntax "-u FOO" in this setting. This is equivalent to "#undef FOO" in a source file. Note: the syntax differs from macro definition (there is a space, i.e. "-u FOO" versus "-DFOO"). Macros defined with this setting are applied to all .S, .c, and C++ files in the BSP. This setting defines the value of BSP_CFLAGS_UNDEFINED_SYMBOLS in the BSP Makefile.
Restrictions	none

hal.make.cc_post_process

Identifier	CC_POST_PROCESS
Type	Unquoted string
Default Value	none
Destination File	makefile_variable
Description	Command executed after each .c/.S file is compiled.
Restrictions	none

hal.make.cxx_post_process

Identifier	CXX_POST_PROCESS
Type	Unquoted string
Default Value	none
Destination File	makefile_variable
Description	Command executed before each C++ file is compiled.
Restrictions	none

hal.make.cc

Identifier	CC
Type	Unquoted string
Default Value	nios2-elf-gcc -xc
Destination File	makefile_variable
Description	C compiler command
Restrictions	none

hal.make.bsp_cxx_flags

Identifier	BSP_CXXFLAGS
Type	Unquoted string
Default Value	none
Destination File	makefile_variable
Description	Custom flags only passed to the C++ compiler. This setting defines the value of BSP_CXXFLAGS in Makefile.
Restrictions	none

hal.make.bsp_inc_dirs

Identifier	BSP_INC_DIRS
Type	Unquoted string
Default Value	none
Destination File	makefile_variable
Description	Space separated list of extra include directories to scan for header files. Directories are relative to the top-level BSP directory. The -I prefix's added by the makefile so don't add it here. This setting defines the value of BSP_INC_DIRS in Makefile.
Restrictions	none

hal.make.cxx

Identifier	CXX
Type	Unquoted string
Default Value	nios2-elf-gcc -xc++
Destination File	makefile_variable
Description	C++ compiler command
Restrictions	none

hal.make.bsp_cflags_user_flags

Identifier	BSP_CFLAGS_USER_FLAGS
Type	Unquoted string
Default Value	none
Destination File	makefile_variable
Description	Custom flags passed to the compiler when compiling C, C++, and .S files. This setting defines the value of BSP_CFLAGS_USER_FLAGS in Makefile.
Restrictions	none

hal.enable_exit

Identifier	ALT_NO_EXIT
Type	Boolean assignment
Default Value	1
Destination File	public_mk_define
Description	Add exit() support. This option increases code footprint if your "main()" routine does "return" or call "exit()". If false, adds -DALT_NO_EXIT to ALT_CPPFLAGS in public.mk, and reduces footprint.
Restrictions	none

hal.enable_small_c_library

Identifier	none
Type	Boolean assignment
Default Value	0
Destination File	public_mk_define
Description	Causes the small newlib (C library) to be used. This reduces code and data footprint at the expense of reduced functionality. Several newlib features are removed such as floating-point support in printf(), stdin input routines, and buffered I/O. If true, adds -msmallc to ALT_LDFLAGS in public.mk.
Restrictions	none

hal.enable_clean_exit

Identifier	ALT_NO_CLEAN_EXIT
Type	Boolean assignment
Default Value	1
Destination File	public_mk_define
Description	When your application exits, close file descriptors, call C++ destructors, etc. Code footprint can be reduced by disabling clean exit. If disabled, adds -DALT_NO_CLEAN_EXIT to ALT_CPPFLAGS and -Wl,--defsym, exit=_exit to ALT_LDFLAGS in public.mk.
Restrictions	none

hal.enable_runtime_stack_checking

Identifier	ALT_STACK_CHECK
Type	Boolean assignment
Default Value	0
Destination File	public_mk_define
Description	Turns on HAL runtime stack checking feature. Enabling this setting causes additional code to be placed into each subroutine call to generate an exception if a stack collision occurs with the heap or statically allocated data. If true, adds -DALT_STACK_CHECK and -mstack-check to ALT_CPPFLAGS in public.mk.
Restrictions	none

hal.enable_gprof

Identifier	ALT_PROVIDE_GMON
Type	Boolean assignment
Default Value	0
Destination File	public_mk_define
Description	Causes code to be compiled with gprof profiling enabled and the application ELF to be linked with the GPROF library. If true, adds -DALT_PROVIDE_GMON to ALT_CPPFLAGS and -pg to ALT_CFLAGS in public.mk.
Restrictions	none

hal.enable_c_plus_plus

Identifier	ALT_NO_C_PLUS_PLUS
Type	Boolean assignment
Default Value	1
Destination File	public_mk_define
Description	Add C++ support. This option increases code footprint by adding support for C++ constructors. If false, adds -DALT_NO_C_PLUS_PLUS to ALT_CPPFLAGS in public.mk, and reduces code footprint.
Restrictions	none

hal.enable_reduced_device_drivers

Identifier	ALT_USE_SMALL_DRIVERS
Type	Boolean assignment
Default Value	0
Destination File	public_mk_define
Description	The drivers are compiled with reduced functionality to reduce code footprint. Not all drivers observe this setting. The altera_avalon_uart and altera_avalon_jtag_uart drivers switch to a polled-mode of operation. The altera_avalon_cfi_flash, altera_avalon_epcs_flash_controller, and altera_avalon_lcd_16207 drivers are removed. You can define a symbol provided by each driver to prevent it from being removed. If true, adds -DALT_USE_SMALL_DRIVERS to ALT_CPPFLAGS in public.mk.
Restrictions	none

hal.enable_lightweight_device_driver_api

Identifier	ALT_USE_DIRECT_DRIVERS
Type	Boolean assignment
Default Value	0
Destination File	public_mk_define
Description	Enables lightweight device driver API. This reduces code and data footprint by removing the HAL layer that maps device names (e.g. /dev/uart0) to file descriptors. Instead, driver routines are called directly. The open(), close(), and lseek() routines will always fail if called. The read(), write(), fstat(), ioctl(), and isatty() routines only work for the stdio devices. If true, adds -DALT_USE_DIRECT_DRIVERS to ALT_CPPFLAGS in public.mk.
Restrictions	The Altera Host and read-only ZIP file systems can't be used if hal.enable_lightweight_device_driver_api is true.

hal.enable_mul_div_emulation

Identifier	ALT_NO_INSTRUCTION_EMULATION
Type	Boolean assignment
Default Value	0
Destination File	public_mk_define
Description	Adds code to emulate multiply and divide instructions in case they are executed but aren't present in the CPU. Normally this isn't required because the compiler won't use multiply and divide instructions that aren't present in the CPU. If false, adds -DALT_NO_INSTRUCTION_EMULATION to ALT_CPPFLAGS in public.mk.
Restrictions	none

hal.enable_sim_optimize

Identifier	ALT_SIM_OPTIMIZE
Type	Boolean assignment
Default Value	0
Destination File	public_mk_define
Description	The BSP is compiled with optimizations to speedup HDL simulation such as initializing the cache, clearing the .bss section, and skipping long delay loops. If true, adds -DALT_SIM_OPTIMIZE to ALT_CPPFLAGS in public.mk.
Restrictions	When this setting is true, the BSP shouldn't be used to build applications that are expected to run real hardware.

hal.enable_sopc_sysid_check

Identifier	none
Type	Boolean assignment
Default Value	1
Destination File	public_mk_define
Description	Enable SOPC Builder System ID. If a System ID SOPC Builder component is connected to the CPU associated with this BSP, it will be enabled in the creation of command-line arguments to download an ELF to the target. Otherwise, system ID and timestamp values are left out of public.mk for application Makefile "download-elf" target definition. If false, adds --accept-bad-sysid to SOPC_SYSID_FLAG in public.mk.
Restrictions	none

hal.custom_newlib_flags

Identifier	CUSTOM_NEWLIB_FLAGS
Type	Unquoted string
Default Value	none
Destination File	public_mk_define
Description	Build a custom version of newlib with the specified space-separated compilerflags.
Restrictions	The custom newlib build will be placed in the <bsp root>/newlib directory, and will be used only for applications that utilize this BSP.

hal.stdin

Identifier	STDIN
Type	Unquoted string
Default Value	none
Destination File	system_h_define
Description	Slave descriptor of STDIN character-mode device. This setting is used by the ALT_STDIN family of defines in system.h.
Restrictions	This device must be different than the LOG_PORT device.

hal.stdout

Identifier	STDOUT
Type	Unquoted string
Default Value	none
Destination File	system_h_define
Description	Slave descriptor of STDOUT character-mode device. This setting is used by the ALT_STDOUT family of defines in system.h.
Restrictions	This device must be different than the LOG_PORT device.

hal.stderr

Identifier	STDERR
Type	Unquoted string
Default Value	none
Destination File	system_h_define
Description	Slave descriptor of STDERR character-mode device. This setting is used by the ALT_STDERR family of defines in system.h.
Restrictions	This device must be different than the LOG_PORT device.

hal.log_port

Identifier	LOG_PORT
Type	Unquoted string
Default Value	none
Destination File	system_h_define
Description	Slave descriptor of debug logging character-mode device. If defined, it enables extra debug messages in the HAL source. This setting is used by the ALT_LOG_PORT family of defines in system.h.
Restrictions	This device must be different than the STDIN/STDOUT/STDERR devices.

Tcl Commands for BSP Settings

"Settings" on page 14–21 describes settings that are available in a user-managed Nios II project. This section describes the tools that you use to specify and manipulate these settings.

You manipulate project settings with BSP Tcl commands. The commands in this section are used with the utilities **nios2-bsp-create-settings**, **nios2-bsp-update-settings**, and **nios2-bsp-query-settings**. You can invoke the Tcl commands directly on a utility command line using the `--cmd` option, or you can put them in a Tcl script, specified with the `--script` option. For details of how to invoke Tcl commands from utilities, see "Nios II Software Build Tools Utilities" on page 14–1.

Refer to "*Tcl Scripts for Board Support Package Settings*" in the *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook* for a discussion of the default Tcl script, which provides excellent usage examples of many of the Tcl commands described in this section.

The following commands are available to manipulate BSP settings:

add_memory_region

Usage

```
add_memory_region <name> <slave_desc> <offset> <span>
```

Options

- <name>: String with the name of the memory region to create.
- <slave_desc>: String with the slave descriptor of the memory device for this region.
- <offset>: String with the byte offset of the memory region from the memory device base address.
- <span>: String with the span of the memory region in bytes.

Description

Creates a new memory region for the linker script. This memory region must not overlap with any other memory region and must be within the memory range of the associated slave descriptor. The offset and span are decimal numbers unless prefixed with 0x.

Example

```
add_memory_region onchip_ram0 onchip_ram0 0 0x100000
```

add_section_mapping

Usage

```
add_section_mapping
    <section_name> <memory_region_name>
```

Options

- <section_name>: String with the name of the linker section.
- <memory_region_name>: String with the name of the memory region to map.

Description

Maps the specified linker section to the specified linker memory region. If the section does not already exist, add_section_mapping creates it. If it already exists, add_section_mapping overrides the existing mapping with the new one. The linker creates the section mappings in the order in which they appear in the linker script.

Example

```
add_section_mapping .text onchip_ram0
```

are_same_resource

Usage

```
are_same_resource <slave_desc1> <slave_desc2>
```

Options

- <slave_desc1>: String with the first slave descriptor to compare.
- <slave_desc2>: String with the second slave descriptor to compare.

Description

Returns a boolean value that indicates whether the two slave descriptors are connected to the same resource. To connect to the same resource, the two slave descriptors have to be associated with the same module. The module specifies whether two slaves access the same resource or different resources within that module. For example, a dual-port memory has two slaves that access the same resource (the memory). However, you could create a module that has two slaves that access two different resources such as a memory and a control port.

delete_memory_region

Usage

```
delete_memory_region <region_name>
```

Options

- <region_name>: String with the name of the memory region to delete.

Description

Deletes the specified memory region. The region must exist to avoid an error condition.

delete_section_mapping

Usage

```
delete_section_mapping <section_name>
```

Options

- <section_name>: String with the name of the section.

Description

Deletes the specified section mapping.

Example

```
delete_section_mapping .text
```

disable_sw_package

Usage

```
disable_sw_package <software_package_name>
```

Options

- <software_package_name>: String with the name of the software package.

Description

Disables the specified software package. Settings belonging to the package are no longer available in the BSP, and associated source files are not included in the BSP makefile. It is an error to disable a software package that is not enabled.

enable_sw_package

Usage

```
enable_sw_package <software_package_name>
```

Options

- <software_package_name>: String with the name of the software package, with the version number optionally appended with a ':'.

Description

Enables a software package. Adds its associated source files and settings to the BSP. Specify the desired version in the form <sw_package_name>:<version>. If you do not specify the version, enable_sw_package selects the latest available version.

Example

```
Example 1:

enable_sw_package altera_hostfs:7.2

Example 2:

enable_sw_package my_sw_package
```

get_addr_span

Usage

```
get_addr_span <slave_desc>
```

Options

- <slave_desc>: String with the slave descriptor to query.

Description

Returns the address span (length in bytes) of the slave descriptor as an integer decimal number.

Example

```
puts [get_addr_span onchip_ram_64_kbytes]

Returns:

65536
```

get_available_drivers

Usage

```
get_available_drivers <module_name>
```

Options

- <module_name>: String with the name of the module to query.

Description

Returns a list of available device driver names that are compatible with the specified module instance. The list is empty if there are no drivers available for the specified slave descriptor. The format of each entry in the list is the driver name followed by a colon and the version number (if provided).

Example

```
puts [get_available_drivers jtag_uart]

Returns:

altera_avalon_jtag_uart_driver:7.2 altera_avalon_jtag_uart_driver:6.1
```

get_available_sw_packages

Usage

```
get_available_sw_packages
```

Options

None

Description

Returns a list of software package names that are available for the current BSP. The format of each entry in the list is a string containing the package name followed by a colon and the version number (if provided).

Example

```
puts [get_available_sw_packages]

Returns:

altera_hostfs:7.2 altera_ro_zipfs:7.2
```

get_base_addr

Usage

```
get_base_addr <slave_desc>
```

Options

- <slave_desc>: String with the slave descriptor to query.

Description

Returns the base byte address of the slave descriptor as an integer decimal number.

Example

```
puts [get_base_addr jtag_uart]

Returns:

67616
```

get_current_memory_regions

Usage

```
get_current_memory_regions
```

Options

None

Description

Returns a sorted list of records representing the existing linker script memory regions. Each record in the list represents a memory region. Each record is a list containing the region name, associated memory device slave descriptor, offset, and span, in that order.

Example

```
puts [get_current_memory_regions]

Returns:

{reset onchip_ram0 0 32} {onchip_ram0 onchip_ram0 32 1048544}
```

get_current_section_mappings

Usage

```
get_current_section_mappings
```

Options

None

Description

Returns a list of lists for all the current section mappings. Each list represents a section mapping with the format {section_name memory_region}. The order of the section mappings matches their order in the linker script.

Example

```
puts [get_current_section_mappings]

Returns:

{.text onchip_ram0} {.rodata onchip_ram0} {.rwdata onchip_ram0} {.bss \
    onchip_ram0} {.heap onchip_ram0} {.stack onchip_ram0}
```

get_default_memory_regions

Usage

```
get_default_memory_regions
```

Options

None

Description

Returns a sorted list of records representing the default linker script memory regions. The default linker script memory regions are the best guess for memory regions based on the reset address and exception address of the processor associated with the BSP, and all other processors in the system that share memories with the processor associated with the BSP. Each record in the list represents a memory region. Each record is a list containing the region name, associated memory device slave descriptor, offset, and span, in that order.

Example

```
puts [get_default_memory_regions]

Returns:

{reset onchip_ram0 0 32} {onchip_ram0 onchip_ram0 32 1048544}
```

get_driver

Usage

```
get_driver <module_name>
```

Options

- <module_name>: String with the name of the module instance to query.

Description

Returns the driver name associated with the specified module instance. The format is <driver name> followed by a colon and the version (if provided). Returns the string "none" if there is no driver associated with the specified module instance name.

Example

```
Example 1:

puts [get_driver jtag_uart]

Returns:

altera_avalon_jtag_uart_driver:7.2

Example 2:

puts [get_driver onchip_ram_64_kbytes]

Returns:

none
```

get_enabled_sw_packages

Usage

```
get_enabled_sw_packages
```

Options

None

Description

Returns a list of currently enabled software packages. The format of each entry in the list is the software package name followed by a colon and the version number (if provided).

Example

```
puts [get_enabled_sw_packages]

Returns:

altera_hostfs:7.2
```

get_exception_offset

Usage

```
get_exception_offset
```

Options

None

Description

Returns the byte offset of the processor exception address.

Example

```
puts [get_exception_offset]

Returns:

32
```

get_exception_slave_desc

Usage

```
get_exception_slave_desc
```

Options

None

Description

Returns the slave descriptor associated with the processor exception address.

Example

```
puts [get_exception_slave_desc]

Returns:

onchip_ram_64_kbytes
```

get_fast_tlb_miss_exception_offset

Usage

```
get_fast_tlb_miss_exception_offset
```

Options

None

Description

Returns the byte offset of the CPU fast TLB miss exception address. Only a CPU with an MMU has such an exception address.

Example

```
puts [get_fast_tlb_miss_exception_offset]

Returns:

32
```

get_fast_tlb_miss_exception_slave_desc

Usage

```
get_fast_tlb_miss_exception_slave_desc
```

Options

None

Description

Returns the slave descriptor associated with the CPU fast TLB miss exception address. Only a CPU with an MMU has such an exception address.

Example

```
puts [get_fast_tlb_miss_exception_slave_desc]

Returns:

onchip_ram_64_kbytes
```

get_irq_number

Usage

```
get_irq_number <slave_desc>
```

Options

- <slave_desc>: String with the slave descriptor to query.

Description

Returns the interrupt request number of the slave descriptor (-1 if none).

get_memory_region

Usage

```
get_memory_region <name>
```

Options

- <name>: String with the name of the memory region.

Description

Returns the linker script region information for the specified region. The format of the region is a list containing the region name, associated memory device slave descriptor, offset, and span in that order.

Example

```
puts [get_memory_region reset]

Returns:

reset onchip_ram0 0 32
```

get_module_class_name

Usage

```
get_module_class_name <module_name>
```

Options

- <module_name>: String with the module instance name to query.

Description

Returns the name of the module class associated with the module instance.

Example

```
puts [get_module_class_name jtag_uart0]
```

```
Returns:
```

```
altera_avalon_jtag_uart
```

get_module_name

Usage

```
get_module_name <slave_desc>
```

Options

- <slave_desc>: String with the slave descriptor to query.

Description

Returns the name of the module instance associated with the slave descriptor. If a module with one slave, or if it has multiple slaves connected to the same resource, the slave descriptor is the same as the module name. If a module has multiple slaves that do not connect to the same resource, the slave descriptor consists of the module name followed by an underscore and the slave name.

Example

```
puts [get_module_name multi_jtag_uart0_s1]

Returns:

multi_jtag_uart0
```

get_module_parameter_value

Usage

```
get_module_parameter_value
    <module_name> <parameter_name>
```

Options

- <module_name>: String with the name of the module in the SOPC system of interest.
- <parameter_name>: String with the name of the parameter to query.

Description

Given the name of a module and a parameter name to query, this function command locates the module by name, and returns the value of the parameter as a string. Returns "null" if any error occurs, such as a module that cannot be found or a parameter name that does not exist.

get_reset_offset

Usage

```
get_reset_offset
```

Options

None

Description

Returns the byte offset of the processor reset address.

Example

```
puts [get_reset_offset]

Returns:

0
```

get_reset_slave_desc

Usage

```
get_reset_slave_desc
```

Options

None

Description

Returns the slave descriptor associated with the processor reset address.

Example

```
puts [get_reset_slave_desc]

Returns:

onchip_ram_64_kbytes
```

get_section_mapping

Usage

```
get_section_mapping <section_name>
```

Options

- <section_name>: String with the section name to query.

Description

Returns the name of the memory region for the specified linker section. Returns null if the linker section does not exist.

Example

```
puts [get_section_mapping .text]

Returns:

onchip_ram0
```

get_setting

Usage

```
get_setting <name>
```

Options

- <name>: String with the name of the setting to get.

Description

Returns the value of the specified BSP setting. get_setting returns boolean settings with the value 1 or 0. If the value of the setting is an empty string, get_setting returns "none".

The get_setting command is equivalent to the --get command line option.

Example

```
puts [get_setting hal.enable_gprof]

Returns:

0
```

get_setting_desc

Usage

```
get_setting_desc <name>
```

Options

- <name>: String with the name of the setting to get the description for.

Description

Returns a string describing the BSP setting.

Example

```
puts [get_setting_desc hal.enable_gprof]

Returns:

"This example compiles the code with gprof profiling enabled and links \
    the application ELF with the GPROF library. If true, adds \
   -DALT_PROVIDE_GMON to ALT_CPPFLAGS and -pg to ALT_CFLAGS in public.mk."
```

get_slave_descs

Usage

```
get_slave_descs
```

Options

None

Description

Returns a sorted list of all the slave descriptors connected to the Nios II processor.

Example

```
puts [get_slave_descs]

Returns:

jtag_uart0 onchip_ram0
```

is_char_device

Usage

```
is_char_device <slave_desc>
```

Options

- <slave_desc>: String with the slave descriptor to query.

Description

Returns a boolean value that indicates whether the slave descriptor is a character device.

Example

```
Example 1:

puts [is_char_device jtag_uart]

Returns:

1

Example 2:

puts [is_char_device onchip_ram_64_kbytes]

Returns:

0
```

is_connected_to_data_master

Usage

```
is_connected_to_data_master <slave_desc>
```

Options

- <slave_desc>: String with the slave descriptor to query.

Description

Returns a boolean value that indicates whether the slave descriptor is connected to a data master.

is_connected_to_instruction_master

Usage

```
is_connected_to_instruction_master <slave_desc>
```

Options

- <slave_desc>: String with the slave descriptor to query.

Description

Returns a boolean value that indicates whether the slave descriptor is connected to an instruction master.

is_flash

Usage

```
is_flash <slave_desc>
```

Options

- <slave_desc>: String with the slave descriptor to query.

Description

Returns a boolean value that indicates whether the slave descriptor is a flash memory device.

is_memory_device

Usage

```
is_memory_device <slave_desc>
```

Options

- <slave_desc>: String with the slave descriptor to query.

Description

Returns a boolean value that indicates whether the slave descriptor is a memory device.

Example

```
Example 1:

puts [is_memory_device jtag_uart]

Returns:

0

Example 2:

puts [is_memory_device onchip_ram_64_kbytes]

Returns:

1
```

is_non_volatile_storage

Usage

```
is_non_volatile_storage <slave_desc>
```

Options

- <slave_desc>: String with the slave descriptor to query.

Description

Returns a boolean value that indicates whether the slave descriptor is a non-volatile storage device.

log_debug

Usage

```
log_debug <message>
```

Options

- <message>: String with message to log.

Description

Displays a message to the host's stdout when the logging level is "debug".

log_default

Usage

```
log_default <message>
```

Options

- <message>: String with message to log.

Description

Displays a message to the host's stdout when the logging level is "default" or higher.

Example

```
log_default "This is a default message."

Displays:

INFO: Tcl message: "This is a default message."
```

log_error

Usage

```
log_error <message>
```

Options

- <message>: String with message to log.

Description

Displays a message to the host's stderr, regardless of logging level.

log_verbose

Usage

```
log_verbose <message>
```

Options

- <message>: String with message to log.

Description

Displays a message to the host's stdout when the logging level is "verbose" or higher.

set_driver

Usage

```
set_driver <driver_name> <module_name>
```

Options

- <driver_name>: String with the name of the device driver to use.
- <module_name>: String with the name of the module instance to set.

Description

Selects the specified device driver for the specified module instance. The "driver_name" argument includes a version number, delimited by a colon (:). If you omit the version number, set_driver uses the latest available version of the driver which is compatible with the SOPC Builder module specified by the "module_name" argument.

If driver_name is "none", the specified module instance does not use a driver. If driver_name is not "none", it must be the name of the associated SOPC Builder module class.

Example

```
Example 1:

set_driver altera_avalon_jtag_uart_driver:7.2 jtag_uart

Example 2:

set_driver none jtag_uart
```

set_setting

Usage

```
set_setting <name> <value>
```

Options

- <name>: String with the name of the setting.
- <value>: String with the value of the setting.

Description

Sets the value for the specified BSP setting. Legal values for boolean settings are true, false, 1, and 0. Use the keyword none instead of an empty string to set a string to an empty value. The set_setting command is equivalent to the --set command line option.

Example

```
set_setting hal.enable_gprof true
```

Tcl Commands for Drivers and Packages

This section describes the tools that you use to specify and manipulate the settings and characteristics of a custom software package or driver. Typically, when creating a custom software package or device driver, or importing a package or driver from another development environment, you need these more powerful tools. To manipulate settings on existing software packages and device drivers, see "Settings" on page 14–21 and "Tcl Commands for BSP Settings" on page 14–160.

A device driver and a software package are both collections of source files added to the BSP. A device driver is associated with a particular SOPC Builder module class (for example, `altera_avalon_jtag_uart`). A software package is not associated with any particular SOPC Builder module class, but implements a functionality such as TCP/IP.

To define a device driver or software package, you create a Tcl script defining its characteristics. This section describes the Tcl commands available to define device drivers and software packages.

The following commands are available for device driver and software package creation:

add_sw_property

Usage

```
add_sw_property <property> <value>
```

Options

- <property>: Name of property.
- <value>: Value assigned or appended to the value.

Description

This command defines a property for a device driver or software package. A property is a list of values (for example, a list of file names). The add_sw_property command defines a property if it has not already been defined. The command appends a new value to the list of values if the property is already defined.

In the case of a property consisting of a file name or directory name, use a relative path. Specify the path relative to the directory containing the Tcl script.

This command supports the following properties:

`asm_source`

Adds an assembly language file (.s or .S) to BSPs containing your package. nios2-bsp-generate-files copies assembly source files into a BSP and adds them to the source build list in the BSP makefile. This property is optional.

`c_source`

Adds a C source file (.c) to BSPs containing your package. nios2-bsp-generate-files copies C source files into a BSP and adds them to the source build list in the BSP makefile. This property is optional.

`include_source`

Adds an include file (typically .h) to BSPs containing your package. nios2-bsp-generate-files copies include files into a BSP, but does not add them to the generated makefile. This property is optional.

`include_directory`

Adds a directory to the ALT_INCLUDE_DIRS variable in the BSP's public.mk file. Adding a directory to ALT_INCLUDE_DIRS allows all source files to find include files in this directory. add_sw_property adds the path to the generated public makefile shared by the BSP and applications or libraries referencing it. add_sw_property compiles all files with the include directory listed in the compiler arguments.

This property is optional.

`lib_source`

Adds a precompiled library (typically .a) to each BSP containing the driver or package. nios2-bsp-generate-files copies the precompiled library into the BSP and adds both the library and the path (required to locate the library) into to the BSP's public.mk file. Applications using the BSP link with the library.

The library filename must conform to the following pattern:

```
lib<name>.a
```

`where <name> is a nonempty string.`

Example:

```
add_sw_property lib_source HAL/lib/libcomponent.a
```

This property is optional.

`specific_compatible_hw_version`

Specifies that the device driver only supports the specified SOPC Builder module hardware version. See the "set_sw_property version" command help for information on version strings. This property applies only to device drivers (see "create_driver" command), not to software packages. If your driver supports all versions of a peripheral after a specific release, use the "set_property min_compatible_hw_version" command instead. This property is optional.

`supported_bsp_type`

Adds a specific BSP type (operating system) to the list of supported operating systems. Valid operating system names are HAL and UCOSII. If your software is OS-neutral and works on multiple operating systems, use successive "supported_os" commands to state compatibility. You must make sure this property is set for each BSP type your software or driver supports.

`alt_cppflags_addition`

Adds a line of arbitrary text to the ALT_CPPFLAGS variable in the BSP public.mk file. This technique can be useful if you wish to have a static compilation flag or definition that all BSP, application, and library files receive during software build. This property is optional.

add_sw_setting

Usage

```
add_sw_setting
    <type> <destination> <displayName> <identifier>
    <value> <description>
```

Options

- <type>: Setting type - Boolean, QuotedString, UnquotedString.
- <destination>: The destination BSP file associated with the setting, or the module generator that processes this setting.
- <displayName>: Setting name.
- <identifier>: Name of the macro created for a generated destination file.
- <value>: Default value of the setting.
- <description>: Setting description.

Description

This command creates a BSP setting associated with a software package or device driver. The setting is available whenever the software package or device driver is present in the BSP. nios2-bsp-generate-files converts the setting and its value into either a C preprocessor macro or BSP makefile variable. add_sw_setting passes macro definitions to the compiler using the "-D" command line option, or adds them to the system.h file as #defines.

The setting only exists once even if there are multiple instances of a software package. Set or get the setting with the --set and --get command line options of the nios-bsp, nios2-bsp-create-settings, nios2-bsp-query-settings, and nios2-bsp-update-settings commands. You can also use the BSP Tcl commands set_setting and get_setting to set or get the setting. The value of the setting persists in the BSP settings file.

To create a setting, you must define each of the following parameters:

```
type
```

This parameter formats the setting value during BSP generation. The following supported types and usage restrictions apply:

```
boolean_define_only
```

Defines a macro if the setting's value is "1" or "true". Example: "#define LCD_PRESENT". No macro is defined if the setting's value is "0" or "false". This setting type supports the "sustem_h_define" and "public_mk_define" generators.

`boolean`

Defines a macro or makefile variable to "1" (if the value is "1" or "true") or "0" (if the value is "0" or "false"). Example: "#define LCD_PRESENT 1". This type supports all generators.

`character`

Defines a macro with a single character with single-quotes around the character. Example: "#define DELIMITER ':'". This type supports the "system_h_define" destination.

`decimal_number`

Decimal numbers define a macro or makefile variable with an unquoted decimal (integer) number. Example: "#define NUM_COPROCESSORS 3". This type supports all destinations.

`double`

Double numbers have a macro name and setting value in the destination file including decimal point. Example: "#define PI 3.1416". This type supports the "system_h_define" destination.

`float`

Float numbers have a macro name and setting value in the destination file including decimal point and 'f' character. Example: "#define PI 3.1416f". This type supports the "system_h_define" destination.

`hex_number`

Hex numbers have a macro name and setting value in the destination file with "0x" prepended to the value. Example: "#define LCD_SIZE 0x1000". This type supports the "system_h_define" destination.

`quoted_string`

Quoted strings always have the macro name and setting value added to the destination files. In the destination, the setting value is enclosed in quotation marks. Example:

```
#define DFLT_ERR "General error"
```

If the setting value contains white space, you must also place quotation marks around the value string in the Tcl script.

This type supports the "system_h_define" destination.

`unquoted_string`

Unquoted strings define a macro or makefile variable with setting name and value in the destination file. In the destination file, the setting value is not enclosed in quotation marks. Example:

```
#define DFLT_ERROR Error
```

This type supports all destinations.

`destination`

The destination parameter specifies where add_sw_setting puts the setting in the generated BSP. add_sw_settings supports the following destinations:

`system_h_define`

With this destionation, add_sw_settings formats settings as "#define <setting name> [optional: <setting value>]" macros in the system.h file

`public_mk_define`

With this destionation, add_sw_settings formats settings as "-D<setting name>[optional: =<setting value>] additions to the ALT_CPPFLAGS variable in the BSP public.mk file. public.mk passes the flag to the C preprocessor for each source file in the BSP, and in applications and libraries using the BSP.

`makefile_variable`

With this destionation, add_sw_settings formats settings as makefile variable additions to the BSP makefile. The variable name must be unique in the makefile.

`displayName`

The name of the setting. Settings exist in a hierarchical namespace. A period separates levels of the hierarchy. Settings created in your Tcl script are located in the hierarchy under the driver or software package name you specified in the "create_driver" or "create_sw_package" command. Example: "my_driver.my_setting". The Nios II software build tools add the hierarchical prefix to the setting name.

`identifier`

The name of the macro or makefile variable being defined. In a setting added to the system.h file at generation time, this parameter corresponds to the text immediately following the "#define" statement.

`value`

The default value associated with the setting. If the user does not assign a value to the option, its value is this default value. Valid initial values are "true", "1", "false", and "0" for boolean and boolean_define_only setting types, a single character for the character type, integer numbers for the decimal_number setting type, integer numbers with or without a "0x" prefix for the hex_number type, numbers with decimals for float_number and double_number types, or an arbitrary string of text for quoted and unquoted string setting types. For string types, if the value contains any white space, you must enclose it in quotation marks.

`description`

Descriptive text that is inserted along with the setting value and name in the summary.html file. You must enclose the description in quotation marks if it contains any spaces. If the description includes any special characters (such as quotation marks), you must escape them with the backslash (\) character. The description field is mandatory, but can be an empty string ("").

create_driver

Usage

```
create_driver <name>
```

Options

- <name>: Name of device driver.

Description

This command creates a new device driver instance available for the Nios II BSP generator. This command must precede all others that describe the device driver in its Tcl script. You can only have one create_driver command in each Tcl script. If the create_driver command appears in the Tcl script, the create_sw_package command cannot appear.

The name argument is usually distinct from all other device drivers and software packages that the BSP generator might locate. You can specify driver name identical to another driver if the driver you are describing has a unique version number assignment.

If your driver differs for different BSP (OS) types, you need to provide a unique name for each BSP type.

This command is required, unless you use the create_sw_package command.

create_sw_package

Usage

```
create_sw_package <name>
```

Options

- <name>: Name of the software package.

Description

This command creates a new software package instance available for the Nios II BSP generator. This command must precede all others that describe the software package in its Tcl script. You can only have one create_sw_package command in each Tcl script. If the create_sw_package command appears in the Tcl script, the command create_driver cannot appear.

The name argument is usually distinct from all other device drivers and software packages that the BSP generator might locate. You can specify a name identical to another software package if the software package you are describing has a unique version number assignment.

If your software package differs for different BSP (OS) types, you need to provide a unique name for each BSP type.

This command is required, unless you use the create_driver command.

set_sw_property

Usage

```
set_sw_property <property> <value>
```

Options

- <property>: Type of software property being set.
- <value>: Value assigned to the property.

Description

Sets the specified value to the specified property. The properties this command supports can only hold a single value. This command overwrites the existing (or default) contents of a particular property with the specified value. This command applies to device drivers and software packages.

This command supports the following properties:

```
hw_class_name
```

The name of the hardware class which your device driver supports. The hardware class name is also the "Component Name" in SOPC Builder Component Editor. Example: altera_avalon_uart. This property is only available for device drivers.

NOTE: if your driver supports a user-defined component created with SOPC Builder 7.0 or earlier, you must append "_classic" to the class name. If you create (or update) your component with the SOPC Builder 7.1 (or later) component editor, there is no need to append "_classic".

This property is required for all drivers.

```
version
```

The version number of this package. set_sw_property uses version numbers to determine compatibility between hardware (peripherals) and their software (drivers), as well as to choose the most recent software or driver if multiple compatible versions are available. A version can be any alphanumeric string, but is usually a major and one or more minor revision integers. The dot (".") character separates major and minor revision numbers. Examples: "7.2", "5.0sp1", "3.2.11". This property is optional, but recommended. If you do not specify a version, the newest version of the package is used.

`min_compatible_hw_version`

Specifies that the device driver you are describing supports the specified hardware version, or all greater versions. This property is only available for device drivers. If your device driver supports only one or more specific versions of a hardware class, use the "add_sw_property specific_compatible_hw_version" command instead. See the "version" property help for information on version strings. This property is optional, but recommended (for device drivers only).

`auto_initialize`

Boolean value that specifies alt_sys_init.c needs to initialize your package. If enabled, you must provide a header file containing "INSTANCE" and "INIT" macros per the instructions in the "Nios II Software Developer's Handbook". This property is optional; if unspecified, alt_sys_init.c does not contain references to your driver or software.

`bsp_subdirectory`

Specifies the top-level directory where nios2-bsp-generate-files copies all source files for this package. This property is a path relative to the top-level BSP directory. This property is optional; if unspecified, nios2-bsp-generate-files copies the driver or software package into the "drivers" subdirectory of any BSP including this software.

`alt_sys_init_priority`

This property assigns a priority to the software package or device driver. The value of this property must be a positive integer. Customize the order in which alt_sys_init.c initializes software and drivers by specifying a priority assignment. Specifying the priority is useful if your software or driver must be initialized before or after other software in the system. For example, your driver might depend on another driver already having been initialized.

This property is optional. The default priority is "1000".

Path Names

There are some restrictions on how you can specify file paths when working with the Nios II software build tools. The tools are designed for the maximum possible compatibility with a variety of computing environments. By following the restrictions in this section, you can ensure that the build tools work smoothly with other tools in your tool chain.

Command Arguments

Many Nios II software build tool commands take file name and directory path arguments. You can provide these arguments in any of several supported cross-platform formats. The Nios II software build tools support the following path name formats:

- **Quoted Windows** — A drive letter followed by a colon, followed by directory names delimited with backslashes, surrounded by double quotes. Example of a quoted Windows absolute path:

```
“C:\altera\72\nios2eds\examples\verilog\niosII_cyclone_1c20\standard”
```

  Quoted Windows relative paths omit the drive letter, and begin with two periods followed by a backslash. Example:

```
“..\niosII_cyclone_1c20\standard”
```

- **Escaped Windows** — The same as quoted Windows, except that each backslash is replaced by a double backslash, and the double quotes are omitted. Examples:

```
C:\\altera\\72\\nios2eds\\examples\\verilog\\niosII_cyclone_1c20\\standard

..\\niosII_cyclone_1c20\\standard
```

- **Linux** — An optional forward slash, followed by directory names delimited with forward slashes. Examples:

```
/altera/72/nios2eds/examples/verilog/niosII_cyclone_1c20/standard

verilog/niosII_cyclone_1c20/standard
```

  Linux relative paths begin with two periods followed by a forward slash. Example:

```
../niosII_cyclone_1c20/standard
```

- **Mixed** — The same as quoted Windows, except that each backslash is replaced by a forward slash, and the double quotes are omitted. Examples:

```
C:/altera/72/nios2eds/examples/verilog/niosII_cyclone_1c20/standard
```

```
../niosII_cyclone_1c20/standard
```

- **Cygwin** — An absolute Cygwin path consists of the pseudo-directory name "/cygdrive/", followed by the lower case Windows drive name, followed by directory names delimited with forward slashes. Example:

```
/cygdrive/c/altera/72/nios2eds/examples/verilog/niosII_cyclone_1c20/standard
```

Cygwin relative paths are the same as Linux relative paths. Example:

```
../niosII_cyclone_1c20/standard
```

The Nios II software build tools accept both relative and absolute path names.

Table 14–5 shows the supported path name formats for each platform, for Nios II software build tools utilities and makefiles.

Table 14–5. Path Name Format Support

Context	Formats supported on Linux *(1)*	Formats supported on Windows with Cygwin
Utilities and scripts	Linux	● Quoted Windows *(2)* ● Mixed *(2)* ● Escaped Windows *(2)* ● Cygwin
Makefiles	Linux	● Mixed *(3)* ● Cygwin *(3)*

Notes to Table 14–5

(1) These rules apply to any Unix-like platform.

(2) These rules apply to other Unix-like shells running on Windows. The Nios II Command Shell, provided with the Nios II Embedded Design Suite (EDS), is based on Cygwin. Examples in this chapter are designed for the Nios II Command Shell.

(3) The build tools automatically convert path names to Cygwin format

Object File Directory Tree

When the Nios II software build tools create a makefile, the makefile is designed to create a new directory tree for generated object files. As far as possible, the object file directory tree retains the structure of the corresponding source directory.

However, there is a restriction, to avoid creating directories outside the project directory root. If the source file path you specify is a relative path, beginning with "..", the Nios II software build tools "flatten" the path name prior to creating the object directory structure.

For example, if source file **tools.c** is located in **src/util/special**, the makefile puts the object file in **obj/util/special/tools.obj**.

The object file directory structure is illustrated in *"HAL BSP Files and Folders"* in the *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

However, if you specify the path to a source file as

```
../special/tools.c
```

the Nios II software build tools place the corresponding object code in

```
obj/tools.o
```

If you specify an absolute path to source files under Cygwin, the Nios II software build tools create the `obj` directory structure as if you had used the Cygwin form of the path name. For example, if you specify the path to a source file as

```
c:/dev/app/special/tools.c
```

the Nios II software build tools place the corresponding object code in

```
obj/cygdrive/c/dev/app/special/tools.o
```

Referenced Documents

This chapter references the following documents:

- *Introduction to the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*
- *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*

Document Revision History

Table 14–6 shows the revision history for this document.

Table 14–6. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	Initial release. Reference material moved here from former *Nios II Software Build Tools* chapter.	

15. Read-Only Zip File System

NII520012-7.2.0

Introduction

Altera® provides a read-only zip file system for use with the hardware abstraction layer (HAL) system library. 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 component for use in the Nios II integrated development environment (IDE). All source and header files for the HAL drivers are located in the directory <*Nios II EDS install path*>**/components/altera_ro_zipfs/HAL/**.

This chapter contains the following sections:

- "Using the Zip File System in a Project" on page 15–1

Using the Zip File System in a Project

The read-only zip file system is supported under the Nios II IDE user interface. You do not have to edit any source code to include and configure the file system. To use the zip file system, you use the Nios II IDE graphical user interface (GUI) to include it as a software component for the system library project.

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

- The name of the flash device you wish to program the file system into
- The offset with this flash.
- The name of the mount point for this file subsystem within the HAL file system. For example, if you name the mount point **/mnt/zipfs**, the following code called from within a HAL-based program opens the file **hello** within the zip file:
  `fopen("/mnt/zipfs/hello", "r")`
- The name of the zip file you wish to use. Before you can specify the zip filename, you must first import it into the Nios II IDE system library project.

For details on importing, see the Nios II IDE help system.

The next time you build your project after you make these settings, the Nios II IDE includes and links the file subsystem in the project. After rebuilding, the **system.h** file reflects the presence of this software component 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 any file decompression features that zip utilities are famous for.

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 into flash. As part of the project build process, the Nios II IDE creates a **.flash** file that includes the data for the zip file system. This file is in the **Release** directory of your project.

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

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

Referenced Documents

This chapter references the following documents:

- *Nios II Flash Programmer User Guide*

Document Revision History

Table 15–1 shows the revision history for this document.

Table 15–1. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	No change from previous release.	
May 2007 v7.1.0	● Chapter 13 was formerly chapter 12. ● Added table of contents to Introduction section. ● Added Referenced Documents section.	
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.	
May 2004 v1.0	Initial Release.	

16. Ethernet and Lightweight IP

NII52009-7.2.0

Usage Note

☞ Do not incorporate the Lightweight IP (lwIP) transmission control protocol/Internet protocol (TCP/IP) suite in new software projects. lwIP is an older networking solution, provided for compatibility with existing customer networking designs. lwIP will be removed from the Nios II EDS in a future release.

This chapter is included for reference in case you are maintaining an existing lwIP-based software application. If you are developing a new networking application, use the NicheStack® TCP/IP Stack - Nios II Edition.

The NicheStack TCP/IP Stack is discussed in the *Ethernet and the NicheStack TCP/IP Stack - Nios II Edition* chapter of the *Nios II Software Developer's Handbook*.

☞ lwIP is incompatible with the NicheStack TCP Stack - Nios II Edition software component.

Introduction

Lightweight IP (lwIP) is a small-footprint implementation of the TCP/IP suite. The focus of the lwIP TCP/IP implementation is to reduce resource usage while providing a full scale TCP/IP. lwIP is designed for use in embedded systems with small memory footprints, making it suitable for Nios® II processor systems. This chapter contains the following sections:

- "Other TCP/IP Stack Providers" on page 16–3
- "Using the lwIP Protocol Stack" on page 16–4
- "Configuring lwIP in the Nios II IDE" on page 16–10
- "Known Limitations" on page 16–13

lwIP includes the following features:

- IP including packet forwarding over multiple network interfaces
- Internet control message protocol (ICMP) for network maintenance and debugging
- User datagram protocol (UDP)
- TCP with congestion control, RTT estimation and fast recovery and fast retransmit
- Dynamic host configuration protocol (DHCP)
- Address resolution protocol (ARP) for Ethernet

- Standard sockets for application programming interface (API)

lwIP Port for the Nios II Processor

Altera provides the Nios II port of lwIP, including source code, in the Nios II Embedded Design Suite (EDS). lwIP provides you with immediate, open-source access to a stack for Ethernet connectivity for the Nios II processor. The Altera® port of lwIP includes a sockets API wrapper, providing the standard, well-documented socket API.

The Nios II EDS include several working examples of programs using lwIP for your reference. In fact, Nios development boards are pre-programmed with a web server reference design based on lwIP and the MicroC/OS-II real-time operating system (RTOS). Full source code is provided.

Altera's port of lwIP uses the MicroC/OS-II RTOS multi-threaded environment. Therefore, to use lwIP, you must base your C/C++ project on the MicroC/OS-II RTOS. Naturally, the Nios II processor system must also contain an Ethernet interface. At present, the Altera-provided lwIP driver supports only the SMSC lan91c111 MAC/PHY device, which is the same device that is provided on Nios development boards. The lwIP driver is interrupt-driven, so you must ensure that interrupts for the Ethernet component are connected.

Altera's port of lwIP is based on the hardware abstraction layer (HAL) generic Ethernet device model. By virtue of the generic device model, you can write a new driver to support any target Ethernet media access controller (MAC), and maintain the consistent HAL and sockets API to access the hardware.

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

This chapter discusses the details of how to use lwIP for the Nios II processor only.

The standard sockets interface is well-documented, and there are a number of books on the topic of programming with sockets. Two good texts are *Unix Network Programming* by Richard Stevens or *Internetworking with TCP/IP Volume 3* by Douglas Comer.

lwIP Files and Directories

You need not edit the source code to use lwIP in a C/C++ program using the Nios II IDE. 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_lwip/UCOSII** directory.

The directory format of the stack tries to maintain the original open-source code as much as possible under the **UCOSII/src/downloads** directory to make upgrades smoother to a more recent version of lwIP. The **UCOSII/src/downloads/lwip-1.1.0** directory contains the original lwIP v1.1.0 source code; the **UCOSII/src/downloads/lwip4ucosii** directory contains the source code of the port for MicroC/OS-II.

Altera's port of lwIP is based on version 1.1.0 of the protocol stack, with wrappers placed around the code to integrate it to the HAL system library. More recent versions of lwIP are available, but newer versions have not been tested with the HAL system library wrappers.

Licensing

lwIP is an open-source TCP/IP protocol stack created by Adam Dunkels at the Computer and Networks Architectures (CNA) lab at the Swedish Institute of Computer Science (SICS), and is available under a modified BSD license. The lwIP project is hosted by Savannah at **http://savannah.nongnu.org/projects/lwip/**. Refer to the Savannah website for complete background information on lwIP and licensing details.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the copyright notice and disclaimer shown in the file *<lwIP component path>***/UCOSII/src/downloads/lwIP-1.1.0/COPYING**.
- Redistributions in binary form must reproduce the copyright notice shown in the file *<lwIP component path>***/UCOSII/src/downloads/lwIP-1.1.0/COPYING**.

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 framework.

For up-to-date information on products available from third-party providers, visit the Nios II homepage at **www.altera.com/nios2**.

Using the lwIP Protocol Stack

This section discusses how to include the lwIP protocol stack in a Nios II program.

The primary interface to the lwIP protocol stack is the standard sockets interface. In addition to the sockets interface, you call the following functions to initialize the stack and drivers:

- `lwip_stack_init()`
- `lwip_devices_init()`

You must also provide the following simple functions that are called by HAL system code to set the MAC address and IP address:

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

Nios II System Requirements

To use lwIP, your Nios II system must meet the following requirements:

- The system hardware must include an Ethernet interface with interrupts enabled
- The system library must be based on MicroC/OS-II

The lwIP Tasks

The Altera-provided lwIP protocol uses the following two fundamental tasks. These tasks run continuously in addition to the tasks that your program creates.

1. The main task is used by the protocol stack. There is a task for receiving packets. The main function of this task blocks waiting for a message box. When a new packet arrives, an interrupt request (IRQ) is generated and an interrupt service routine (ISR) clears the IRQ and posts a message to the message box.

2. The new message then activates the receive task. This design allows the ISR to execute as quickly as possible, reducing the impact on system latency.

These tasks are started automatically when the initialization process succeeds. You set the task priorities, based on the criticality compared to other tasks in the system.

Initializing the Stack

To initialize the stack, call the function `lwip_stack_init()` before calling `OSStart` to start the MicroC/OS-II scheduler. The following code shows an example of a `main()`.

Example: Instantiating the lwIP Stack in main()

```
#include <includes.h>
#include <alt_lwip_dev.h>

int main ()
{
...
  lwip_stack_init(TCPIP_THREAD_PRIO, init_done_func, 0);
...
  OSStart();
...
  return 0;
}
```

lwip_stack_init()

`lwip_stack_init()` performs setup for the protocol stack. The prototype for `lwip_stack_init()` is:

```
void lwip_stack_init(int thread_prio,
        void (* init_done_func)(void *), void *arg)
```

`lwip_stack_init()` returns nothing and has the following parameters:

- `thread_prio`—the priority of the main TCP/IP thread
- `init_done_func`—a pointer to a function that is called once the stack is initialized
- `arg`—an argument to pass to `init_done_func()`. `arg` is usually set to zero.

init_done_func()

You must provide the function `init_done_func()`, which is called after the stack has been initialized. The `init_done_func()` function must call `lwip_devices_init()`, which initializes all the installed Ethernet device drivers, and then creates the receive task.

The prototype for `init_done_func()` is:

```
void init_done_func(void* arg)
```

The following code shows an example of the `tcpip_init_done()` function, which is an example of an implementation of an `init_done_func()` function.

Example: An implementation of init_done_func()

```
#include <stdio.h>
#include <lwip/sys.h>
#include <alt_lwip_dev.h>
#include <includes.h>
/*
 * This function is called once the IP stack is alive
 */
static void tcpip_init_done(void *arg)
{
  int temp;

  if (lwip_devices_init(ETHER_PRIO))
  {
    /* If initialization succeeds, start a user task */
    temp = sys_thread_new(user_thread_func,
                          NULL,
                          USER_THREAD_PRIO);
    if (!temp)
    {
      perror("Can't add the application threads
      OSTaskDel(OS_PRIO_SELF);
    }
  }
  else
  {
    /*
     * May not be able to add an Ethernet interface if:
     * 1. There is no Ethernet hardware
     * 2. Your hardware cannot initialize (e.g.
     * not connected to a network, or can’t get
     * a mac address)
     */
    perror("Can't initialize any interface. Closing down.\n");
    OSTaskDel(OS_PRIO_SELF);
  }

  return;
}
```

You must use `sys_thread_new()` to create any new task that talks to the IP stack using the sockets protocol.

For more information, see “Calling the Sockets Interface” on page 16–9.

lwip_devices_init()

`lwip_devices_init()` iterates through the list of all installed Ethernet device drivers defined in **system.h**, and registers each driver with the stack. `lwip_devices_init()` returns a non-zero value to indicate success. Upon success, the TCP/IP stack is available, and you can then create the task(s) for your program.

The prototype for lwip_devices_init() is:

```
int lwip_devices_init(int rx_thread_prio)
```

The parameter to this function is the priority of the receive thread. `lwip_devices_init()` calls the functions `get_mac_addr()` and `get_ip_address()`, which you must provide.

get_mac_addr() and get_ip_addr()

`get_mac_addr()` and `get_ip_addr()` are called by the lwIP system code during the devices initialization process. These functions are necessary for the lwIP system code to set the MAC and IP addresses for a particular device. By writing 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 may store the MAC address in flash memory, while others may have the MAC address in on-chip embedded memory.

Both functions take as parameters device structures used internally by the lwIP. 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:

```
err_t gat_mac_addr(alt_lwip_dev* lwip_dev);
```

Inside the function, you must fill in the following fields of the `alt_lwip_dev` structure that define the MAC address:

- `unsigned char lwip_dev->netif->hwaddr_len`—the length of the MAC address, which should be 6
- `unsigned char lwIP_dev->netif->hwaddr[0-5]`—the MAC address of the device.

Your code can also verify the name of the device being initialized.

The prototype for `get_mac_addr()` is in the header file **UCOSII/inc/alt_lwip_dev.h**. The `netif` structure is defined in the **UCOSII/src/downloads/lwip-1.1.0/src/include/lwip/netif.h** file.

The following code shows an example implementation of `get_mac_addr()`. For demonstration purposes only, the MAC address is stored at address `0x7f0000` in this example.

Example: An implementation of get_mac_addr()

```
#include <alt_lwip_dev.h>
#include <lwip/netif.h>
#include <io.h>
err_t get_mac_addr(alt_lwip_dev* lwip_dev)
{
  err_t ret_code = ERR_IF;
  /*
   * The name here is the device name defined in system.h
   */
  if (!strcmp(lwip_dev->name, "/dev/lan91c111"))
  {
    /* Read the 6-byte MAC address from wherever it is stored */
    lwip_dev->netif->hwaddr[0] = IORD_8DIRECT(0x7f0000, 4);
    lwip_dev->netif->hwaddr[1] = IORD_8DIRECT(0x7f0000, 5);
    lwip_dev->netif->hwaddr[2] = IORD_8DIRECT(0x7f0000, 6);
    lwip_dev->netif->hwaddr[3] = IORD_8DIRECT(0x7f0000, 7);
    lwip_dev->netif->hwaddr[4] = IORD_8DIRECT(0x7f0000, 8);
    lwip_dev->netif->hwaddr[5] = IORD_8DIRECT(0x7f0000, 9);
    ret_code = ERR_OK;
  }
  return ret_code;
}
```

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

```
int get_ip_addr(alt_lwip_dev*   lwip_dev,
                struct ip_addr* ipaddr,
                struct ip_addr* netmask,
                struct ip_addr* gw,
                int*            use_dhcp);
```

To enable DHCP, include the line:

```
*use_dhcp = 1;
```

To assign a static IP address, include the lines:

```
IP4_ADDR(ipaddr, IPADDR0,IPADDR1,IPADDR2,IPADDR3);
IP4_ADDR(gw, GWADDR0,GWADDR1,GWADDR2,GWADDR3);
IP4_ADDR(netmask, MSKADDR0,MSKADDR1,MSKADDR2,MSKADDR3);
*use_dhcp = 0;
```

`IP_ADDR0-3` are the bytes 0-3 of the IP address. `GWADDR0-3` are the bytes of the gateway address. `MSKADDR0-3` are the bytes of the network mask.

The prototype for get_ip_addr() is in the header file **UCOSII/inc/alt_lwip_dev.h**.

The following code shows an example implementation of get_ip_addr() and shows a list of the necessary include files.

Example: An implementation of get_ip_addr()

```
#include <lwip/tcpip.h>
#include <alt_lwip_dev.h>
int get_ip_addr(alt_lwip_dev*   lwip_dev,
                struct ip_addr* ipaddr,
                struct ip_addr* netmask,
                struct ip_addr* gw,
                int*            use_dhcp)
{
  int ret_code = 0;
  /*
   * The name here is the device name defined in system.h
   */
  if (!strcmp(lwip_dev->name, "/dev/lan91c111"))
  {
#if LWIP_DHCP == 1
    *use_dhcp = 1;
#else
    /* Assign Static IP Addresses */
    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);
    *use_dhcp = 0;
#endif /* LWIP_DHCP */

    ret_code = 1;
  }
  return ret_code;
}
```

Calling the Sockets Interface

Once your Ethernet device has been initialized, the remainder of your program should use the sockets API 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 sys_thread_new(). The sys_thread_new() function is part of the lwIP OS porting layer to create threads. sys_thread_new() calls the MicroC/OS-II OSTaskCreate() function and performs some other lwIP-specific actions.

The prototype for sys_thread_new() is:

```
sys_thread_t sys_thread_new(void (* thread)(void *arg),
                            void *arg,
                             int  prio);
```

It is in **ucosII/src/downloads/lwIP-1.1.0/src/include/lwIP/sys.h**. You can include this as `#include "lwIP/sys.h"`.

You can find other details of the OS porting layer in the **sys_arch.c** file in the lwIP component directory, **UCOSII/src/downloads/lwip4ucosii/ucos-ii/**.

Configuring lwIP in the Nios II IDE

The lwIP protocol stack has many configuration options that are configured using `#define` directives in the file **lwipopts.h**. The Nios II integrated development environment (IDE) provides a graphical user interface (GUI) that enables you to configure lwIP options (i.e. modify the `#defines` in **lwipopts.h**) without editing source code. The most commonly accessed options are available through the GUI. However, there are some options that cannot be changed via the GUI, so you have to edit the **lwipopts.h** file manually.

The following sections describe the features that can be configured via the Nios II IDE. The GUI provides 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.

Lightweight TCP/IP Stack General Settings

The ARP and IP protocols are always enabled. Table 16–1 shows the protocol options.

Table 16–1. Protocol Options

Option	Description
UDP	Enables and disables the user datagram protocol (UDP).
TCP	Enables and disables the transmission control protocol (TCP).

Table 16–2 shows the global options, which affect the overall behavior of the TCP/IP stack.

Table 16–2. Global Options

Option	Description
Use DHCP to automatically assign an IP address	Enables and disables DHCP. DHCP requires that the UDP protocol is enabled.
Enable statistics	When this option is turned on, the stack keeps counters of packets received, errors, etc. The counters are defined in a structure variable `lwip_stats` in the **UCOSII/src/downloads/lwIP-1.1.0/src/core** file. The structure definition is in **UCOSII/src/downloads/lwIP-1.1.0/src/include/lwIP/stats.h**.
Number of packet buffers	The number of buffers for the network driver to receive packets into.
Time to live	The number of seconds that a datagram can remain in the system before being discarded.
Maximum packet size	The maximum size of the packets on the network interface.
Stack size of the LWIP tasks (32-bit words)	The stack size of the lwIP tasks. For more information on the task, see "The lwIP Tasks" on page 16–4.
Default MAC interface	If the IP stack has more than one network interface, this parameter indicates which interface to use when sending packets to an IP address without a known route. See "Known Limitations" on page 16–13.

IP Options

If the forward IP packets option is turned on, when there is more than one network interface, and the IP stack for one interface receives packets not addressed to it, it forwards the packet out of the other interface.

ARP Options

The size of ARP table is the number of entries that can be stored in the ARP cache.

UDP Options

You can enter the maximum number of UDP sockets that the application uses.

TCP Options

Table 16–3 shows the TCP options.

Table 16–3. TCP Options

Option	Description
Max number of listening sockets	Maximum number of TCP sockets that can be listening for a client to connect.
Max number of active sockets	Maximum number of TCP sockets that the program uses, excluding listening sockets.
Max retransmissions	The maximum number of times that the TCP protocol tries to retransmit a packet which is not acknowledged.
Max retransmissions of SYN frames	The maximum number of times that the TCP protocol tries to retransmit a SYN packet, which is not acknowledged.
Max segment size	Maximum TCP segment size.
Max send buffer space	The maximum amount of data TCP buffers up for transmission.
Max window size	The maximum amount of data for each receiving socket that TCP buffers up

DHCP Options

You can specify that the ARP checks the assigned address is not in use, so once the DHCP protocol has assigned an IP address, it send out an APR packet to check that no-one else is using the assigned address.

Memory Options

Table 16–4 shows the memory options.

Table 16–4. Memory Options (Part 1 of 2)

Option	Description
Maximum number of buffers sent without copying	The maximum number of buffers that the stack attempts to transmit without copying. Only use this option for sending UDP packets and fragmented IP packets. This option maps onto the lwIP `#define memp_num_pbuf`.
Maximum number of packet buffers passed between the application and stack threads	The maximum number of buffers that can be passed between the application thread and the protocol stack thread (in either direction) at any one time.This option maps onto the lwIP `#define memp_num_netbuf`.
Maximum number of pending API calls from the application to the stack thread	The size of the message box that sends API calls from the application thread to the protocol thread. This option maps onto the lwIP `#define memp_num_api_msg`.

Table 16–4. Memory Options (Part 2 of 2)

Option	Description
Maximum number of messages passed from the protocol stack thread to the application	The combination of API calls passed from the application thread to the stack thread, and packets being passed the other way. This option maps onto the lwIP `#define memp_num_tcpip_msg`.
TCP/IP Heap size	The size of the memory pool for copying buffers into temporary locations, which is not the total memory size. This option maps onto the lwIP `#define mem_size`.

Known Limitations

The following limitations of Altera's current implementation of the lwIP stack are known:

- lwIP does not implement the shutdown socket call correctly. The shutdown call maps directly on to the close socket call
- Multiple network interfaces features are present in the code, but have not been tested.

Referenced Documents

This chapter references the following documents:

- *Ethernet and the NicheStack TCP/IP Stack - Nios II Edition* chapter of the *Nios II Software Developer's Handbook*
- *Developing Device Drivers for the HAL the HAL* chapter of the *Nios II Software Developer's Handbook*

Document Revision History

Table 16–5 shows the revision history for this document.

Table 16–5. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
October 2007 v7.2.0	No change from previous release.	
May 2007 v7.1.0	● Chapter 14 was formerly chapter 13. ● Added table of contents to Introduction section. ● Added Referenced Documents section.	
March 2007 v7.0.0	No change from previous release.	
November 2006 v6.1.0	Moved to Appendix section	NicheStack TCP/IP Stack - Nios II Edition is the preferred network package.
May 2006 v6.0.0	● Corrected error in `alt_irq_enable_all()` usage ● Added illustrations ● Revised text on optimizing ISRs ● Expanded and revised text discussing HAL exception handler code structure.	
October 2005 v5.1.0	● Updated references to HAL exception-handler assembly source files in section "HAL Exception Handler Files". ● Added description of `alt_irq_disable()` and `alt_irq_enable()` in section "ISRs".	
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.	
September 2004 v1.1	● Changed examples. ● Added ISR performance data.	
May 2004 v1.0	Initial Release.	