Nios II Software Developer's Handbook

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

NII5V2-13.1

Contents

Section I. Nios II Software Development

Chapter 1. Overview of Nios II Embedded Development

Chapter 2. Getting Started with the Graphical User Interface

Chapter 4. Nios II Software Build Tools

Section II. 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

Chapter 12. Read-Only Zip File System

Chapter 13. Publishing Component Information to Embedded Software

Section IV. Reference Material

Chapter 14. HAL API Reference

Chapter 15. Nios II Software Build Tools Reference

Additional Information

Chapter Revision Dates

The chapters in this document, 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 of Nios II Embedded Development
Revised: *January 2014*
Part Number: *NII52001-13.1.0*

Chapter 2. Getting Started with the Graphical User Interface
Revised: *January 2014*
Part Number: *NII52017-13.1.0*

Chapter 3. Getting Started from the Command Line
Revised: *January 2014*
Part Number: *NII52014-13.1.0*

Chapter 4. Nios II Software Build Tools
Revised: *January 2014*
Part Number: *NII52015-13.1.0*

Chapter 5. Overview of the Hardware Abstraction Layer
Revised: *May 2011*
Part Number: *NII52003-11.0.0*

Chapter 6. Developing Programs Using the Hardware Abstraction Layer
Revised: *January 2014*
Part Number: *NII52004-13.1.0*

Chapter 7. Developing Device Drivers for the Hardware Abstraction Layer
Revised: *May 2011*
Part Number: *NII52005-11.0.0*

Chapter 8. Exception Handling
Revised: *May 2011*
Part Number: *NII52006-11.0.0*

Chapter 9. Cache and Tightly-Coupled Memory
Revised: *May 2011*
Part Number: *NII52007-11.0.0*

Chapter 10. MicroC/OS-II Real-Time Operating System
Revised: *May 2011*
Part Number: *NII52008-11.0.0*

Chapter 11. Ethernet and the NicheStack TCP/IP Stack - Nios II Edition
Revised: *May 2011*
Part Number: *NII52013-11.0.0*

Chapter 12. Read-Only Zip File System

Revised: *May 2011*
Part Number: *NII52012-11.0.0*

Chapter 13. Publishing Component Information to Embedded Software
Revised: *May 2011*
Part Number: *NII52018-11.0.0*

Chapter 14. HAL API Reference
Revised: *May 2011*
Part Number: *NII52010-11.0.0*

Chapter 15. Nios II Software Build Tools Reference
Revised: *January 2014*
Part Number: *NII52016-13.1.0*

Section I. Nios II Software Development

This section introduces Nios® II embedded software development, including the available tools and tool flows.

This section includes the following chapters:

1. Overview of Nios II Embedded Development

NII52001-13.1.0

The *Nios® II Software Developer's Handbook* provides the basic information needed to develop embedded software for the Altera® Nios II processor. This handbook describes 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 provides a high-level overview of the embedded software development environments for the Nios II processor, and contains the following sections:

Prerequisites for Understanding the Nios II Embedded Design Suite

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 create and debug applications without further knowledge of Altera technology.

Finding Nios II EDS Files

When you install the Nios II EDS, you specify a root directory for the EDS file structure. This root directory must be adjacent to the Quartus® II installation. For example, if the Nios II EDS 10.0 is installed on the Windows operating system, the root directory might be **c:\altera\100\nios2eds**.

For simplicity, this handbook refers to this directory as <*Nios II EDS install path*>.

Nios II Software Development Environment

The Nios II EDS provides a consistent software development environment that works for all Nios II processor systems. With the Nios II EDS running on a host computer, an Altera FPGA, and a 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 connect to 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 includes 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 processor-based systems, eliminating the need to spend time manually creating BSPs. The BSP provides a C/C++ runtime environment, insulating you from the hardware in your embedded system. Altera BSPs contain the Altera hardware abstraction layer (HAL), an optional RTOS, and device drivers.

Nios II EDS Development Flows

A development flow is a way of using a set of development tools together to create a software project. The Nios II EDS provides the following development flows for creating Nios II programs:

- The Nios II Software Build Tools (SBT), which provides two user interfaces:
  - The Nios II SBT command line
  - The Nios II SBT for Eclipse™

The Nios II SBT Development Flow

The Nios II SBT allows you to create Nios II software projects, with detailed control over the software build process. The same Nios II SBT utilities, scripts and Tcl commands are available from both the command line and the Nios II SBT for Eclipse graphical user interface (GUI).

The SBT allows you to create and manage single-threaded programs as well as complex applications based on an RTOS and middleware libraries available from Altera and third-party vendors.

The SBT provides powerful Tcl scripting capabilities. In a Tcl script, you can query project settings, specify project settings conditionally, and incorporate the software project creation process in a scripted software development flow. Tcl scripting is supported both in Eclipse and at the command line.

For information about Tcl scripting, refer to the *Nios II Software Build Tools* chapter of the *Nios II Software Developer’s Handbook.*

The Nios II SBT for Eclipse

The Nios II SBT for Eclipse is a thin GUI layer that runs the Nios II SBT utilities and scripts behind the scenes, presenting a unified development environment. The SBT for Eclipse provides a consistent development platform that works for all Nios II processor systems. You can accomplish all software development tasks within Eclipse, including creating, editing, building, running, debugging, and profiling programs.

The Nios II SBT for Eclipse is based on the popular Eclipse framework and the Eclipse C/C++ development toolkit (CDT) plugins. The Nios II SBT creates your project makefiles for you, and Eclipse provides extensive capabilities for interactive debugging and management of source files.

The SBT for Eclipse also allows you to import and debug projects you created in the Nios II Command Shell.

For details about the Nios II SBT for Eclipse, refer to the *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook.* For details about Eclipse, visit the Eclipse Foundation website (www.eclipse.org).

The Nios II SBT Command Line

In the Nios II SBT command line development flow, you create, modify, build, and run Nios II programs with Nios II SBT commands typed at a command line or embedded in a script. You run the Nios II SBT commands from the Nios II Command Shell.

For further information about the Nios II SBT in command-line mode, refer to the *Getting Started from the Command Line* chapter of the *Nios II Software Developer's Handbook.*

To debug your command-line program, import your SBT projects to Eclipse. You can further edit, rebuild, run, and debug your imported project in Eclipse.

Nios II Programs

Each Nios II program you develop consists of an application project, optional user library projects, and a BSP project. You build your Nios II program to create an Executable and Linking Format File (**.elf)** which runs on a Nios II processor.

The Nios II SBT creates software projects for you. Each project is based on a makefile.

Makefiles and the SBT

The makefile is the central component of a Nios II software project, whether the project is created with the Nios II SBT for Eclipse, or on the command line. 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.

As a key part of creating a software project, the SBT creates a makefile for you. Nios II projects are sometimes called “user-managed,” because you, the user, are responsible for the content of the project makefile. You use the Nios II SBT to control what goes in the makefile.

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

Nios II Software Project Types

The following sections describe the project types that constitute a Nios II program.

Application Project

A Nios II C/C++ application project consists of a collection of source code, plus a makefile. 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. The makefile compiles the source code and links it with a BSP and one or more optional libraries, to create one **.elf** file.

User Library Project

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

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 a Nios II hardware system. The Nios II EDS provides tools to modify settings that control the behavior of the BSP.

A BSP contains the following elements:

- Hardware abstraction layer—For information, refer to the *Overview of the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.
- Optional custom newlib C standard library—For information, refer to the *Overview of the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*. The complete HTML documentation for newlib resides in the Nios II EDS directory.
- Device drivers—For information, refer to "Nios II Embedded Software Projects" in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.
- Optional software packages—For information, refer to "Altera Software Packages for Embedded Systems".
- Optional real-time operating system—For information, refer to the *MicroC/OS-II Real-Time Operating System* chapter of the *Nios II Software Developer's Handbook*.

Altera Software Packages for Embedded Systems

The Nios II EDS includes software packages to extend the capabilities of your software. You can include these software packages in your BSP. Table 1–1 shows those Altera Nios II software packages that are distributed with the Nios II EDS.

Table 1–1. Software Packages

Name	Description
NicheStack TCP/IP Stack - Nios II Edition	Refer to the *Ethernet and the NicheStack TCP/IP Stack - Nios II Edition* chapter of the *Nios II Software Developer's Handbook*.
Read-only zip file system	Refer to the *Read-Only Zip File System* chapter of the *Nios II Software Developer's Handbook*.
Host file system	Refer to the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

Additional software packages are available from Altera's partners. For a complete list, refer to the Embedded Software page of the Altera website.

Nios II Embedded Design Examples

The Nios II EDS includes documented hardware design examples and software examples to demonstrate all prominent features of the Nios II processor and the development environment. The examples can help 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.

Hardware Examples

You can run Nios II hardware designs on many Altera development boards. The hardware examples for each Altera development board can be found in the kit installation provided with the board, and on this website:

http://www.altera.com/products/devkits/kit-dev_platforms.jsp

Alternatively, you can use the Nios II Ethernet Standard design located at *http://www.altera.com/support/examples/nios2/exm-net-std-de.html* or Nios II with MMU design located at *http://www.altera.com/support/examples/nios2/exm-mmu.html*.

The Nios II with MMU design is intended to demonstrate Linux. This design does not work with the SBT, because the SBT does not support the Nios II MMU.

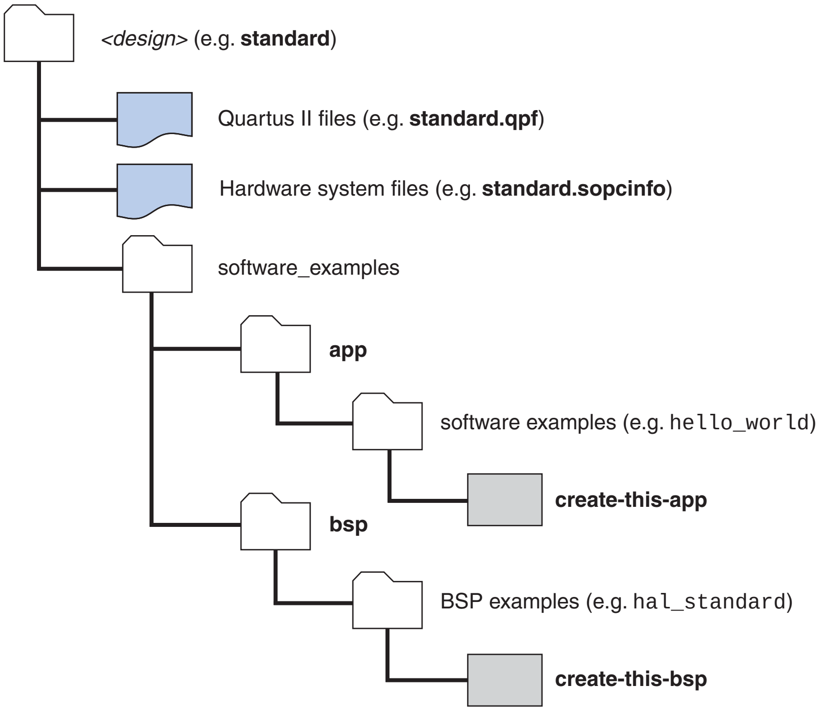
Software Examples

You can run Nios II software examples that run on many of the hardware design examples described in the previous section.

The Nios II software examples include scripts and templates to create the software projects using the Nios II SBT. These scripts and templates do everything necessary to create a BSP and an application project for each software example.

Figure 1–1 shows the directory structure under each hardware design example. 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 "Getting Started with Eclipse" in the *Getting Started from the Command Line* chapter of the *Nios II Software Developer's Handbook*.

Figure 1–1. Software Design Example Directory Structure


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

Third-Party Embedded Tools 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 about third-party support for the Nios II processor, visit the Nios II Processor page of the Altera website.

Additional 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 *Embedded Peripherals IP User Guide* 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 *Embedded Design Handbook* describes how to use Altera software development tools effectively, and recommends design styles and practices for developing, debugging, and optimizing embedded systems.
- The Altera Knowledge Database is an Internet resource that offers solutions to frequently asked questions with an easy-to-use search engine. Visit the Knowledge Database page of the Altera website.
- 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 of the Altera website.
- The Nios II EDS documentation launchpad. The launchpad is an HTML page installed with the Nios II EDS, which provides links to Nios II documentation, examples, and other resources. The way you open the launchpad depends on your software platform.
  - In the Windows operating system, on the Start menu, point to **Programs** > **Altera** > **Nios II EDS**, and click **Nios II** <*version*> **Documentation**.
  - In the Linux operating system, open <*Nios II EDS install path*>**/documents/index.html** in a web browser.

Document Revision History

Table 1–2 shows the revision history for this document.

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

Date	Version	Changes
January 2014	13.1.0	■ Removed references to Nios II IDE. ■ Removed references to Nios II C2H. ■ Updated the “Hardware Examples” section.
May 2011	11.0.0	Introduced Qsys system integration tool
February 2011	10.1.0	Removed “Referenced Documents” section.
July 2010	10.0.0	Installation method changed; Nios II EDS always installed in a directory adjacent to Quartus II tools.
November 2009	9.1.0	■ Described the Nios II Software Build Tools for Eclipse. ■ Nios II IDE information moved to Appendix A. ■ Detailed Nios II Software Build Tools utility information moved to *Nios II Software Build Tools*.
March 2009	9.0.0	■ Incorporate information formerly in *Altera-Provided Development Tools* chapter. ■ Describe BSP Editor. ■ Reorganize and update information and terminology to clarify role of Nios II Software Build Tools. ■ Describe `-data` argument for IDE command-line tools. ■ Correct minor typographical errors.

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

Date	Version	Changes
May 2008	8.0.0	■ Add “What’s New” section. ■ SOPC Information File (**.sopcinfo**). ■ Design examples removed from EDS. ■ Memory management unit (MMU) added to Nios II core.
October 2007	7.2.0	Maintenance release.
May 2007	7.1.0	■ Revise entire chapter to introduce Nios II EDS design flows, Nios II programs, Nios II Software Build Tools, and Nios II BSPs. ■ Add table of contents to Introduction section. ■ Add “Referenced Documents” section.
March 2007	7.0.0	Maintenance release.
November 2006	6.1.0	Maintenance release.
May 2006	6.0.0	Maintenance release.
October 2005	5.1.0	Maintenance release.
May 2005	5.0.0	Maintenance release.
May 2004	1.0	Initial release.

2. Getting Started with the Graphical User Interface

NII52017-13.1.0

The Nios® II Software Build Tools (SBT) for Eclipse™ is a set of plugins based on the Eclipse™ framework and the Eclipse C/C++ development toolkit (CDT) plugins. The Nios II SBT for Eclipse provides a consistent development platform that works for all Nios II embedded processor systems. You can accomplish all Nios II software development tasks within Eclipse, including creating, editing, building, running, debugging, and profiling programs.

This chapter familiarizes you with the features of the Nios II SBT for Eclipse. This chapter contains the following sections:

- "Getting Started with Nios II Software in Eclipse"
- "Makefiles and the Nios II SBT for Eclipse" on page 2–9
- "Using the BSP Editor" on page 2–12
- "Run Configurations in the SBT for Eclipse" on page 2–20
- "Optimizing Project Build Time" on page 2–22
- "Importing a Command-Line Project" on page 2–22
- "Packaging a Library for Reuse" on page 2–25
- "Creating a Software Package" on page 2–26
- "Programming Flash in Altera Embedded Systems" on page 2–30
- "Creating Memory Initialization Files" on page 2–32
- "Running a Nios II System with ModelSim" on page 2–34
- "Eclipse Usage Notes" on page 2–37

Getting Started with Nios II Software in Eclipse

Writing software for the Nios II processor is similar to writing software 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, a getting-started reference design, and all the development tools necessary to write Nios II programs.

Modifying existing code is a common, easy way to learn to start writing software in a new environment. The Nios II Embedded Design Suite (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 RTOS example, to a full TCP/IP stack running a web server. Each example is documented and ready to compile.

Subscribe

This section guides you through the most fundamental operations in the Nios II SBT for Eclipse in a tutorial-like fashion. It shows how to create an application project for the Nios II processor, along with the board support package (BSP) project required to interface with your hardware. It also shows how to build the application and BSP projects in Eclipse, and how to run the software on an Altera® development board.

The Nios II SBT for Eclipse Workbench

The term “workbench” refers to the Nios II SBT for Eclipse desktop development environment. The workbench is where you edit, compile and debug your programs in Eclipse.

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.

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, and provide alternative presentations and ways to navigate the information in your workbench.

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.

For detailed information about the Eclipse workbench, perspectives, and views, refer to the Eclipse help system.

Before you create a Nios II project, you must ensure that the Nios II perspective is visible. To open the Nios II perspective, on the Window menu, point to **Open Perspective**, then **Other**, and click **Nios II**.

The Altera Bytestream Console

The workbench in Eclipse for Nios II includes a bytestream console, available through the Eclipse **Console** view. The Altera bytestream console enables you to see output from the processor’s `stdout` and `stderr` devices, and send input to its `stdin` device. For information about the Altera bytestream console, see “Using the Altera Bytestream Console” on page 2–8.

Creating a Project

In the Nios II perspective, on the File menu, point to **Nios II Application and BSP from Template**. The **Nios II Application and BSP from Template** wizard appears. This wizard provides a quick way to create an application and BSP at the same time.

Alternatively, you can create separate application, BSP and user library projects.

Specifying the Application

In the first page of the **Nios II Application and BSP from Template** wizard, you specify a hardware platform, a project name, and a project template. You optionally override the default location for the application project, and specify a processor name if you are targeting a multiprocessor hardware platform.

You specify a BSP in the second page of the wizard.

Specifying the Hardware Platform

You specify the target hardware design by selecting a SOPC Information File (**.sopcinfo**) in the **SOPC Information File name** box.

Specifying the Project Name

Select a descriptive name for your project. The SBT creates a folder with this name to contain the application project files.

Letters, numbers, and the underscore (_) symbol are the only valid project name characters. Project names cannot contain spaces or special characters. The first character in the project name must be a letter or underscore. The maximum filename length is 250 characters.

The SBT also creates a folder to contain BSP project files, as described in "Specifying the BSP".

Specifying the Project Template

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

You select the project template from the **Templates** list.

The hello_world template provides an easy way to create your first Nios II project and verify that it builds and runs correctly.

Specifying the Project Location

The project location is the parent directory in which the SBT creates the project folder. By default, the project location is under the directory containing the .**sopcinfo** file, in a folder named **software**.

To place your application project in a different folder, turn off **Use default location**, and specify the path in the **Project location** box.

Specifying the Processor

If your target hardware contains multiple Nios II processors, **CPU name** contains a list of all available processors in your design. Select the processor on which your software is intended to run.

Specifying the BSP

When you have finished specifying the application project in the first page of the **Nios II Application and BSP from Template** wizard, you proceed to the second page by clicking **Next**.

On the second page, you specify the BSP to link with your application. You can create a new BSP for your application, or select an existing BSP. Creating a new BSP is often the simplest way to get a project running the first time.

You optionally specify the name and location of the BSP.

Specifying the BSP Project Name

By default, if your application project name is <*project*>, the BSP is named <*project*>**_bsp**. You can type in a different name if you prefer. The SBT creates a directory with this name, to contain the BSP project files. BSP project names are subject to the same restrictions as application project names, as described in "Specifying the Project Name".

Specifying the BSP Project Location

The BSP project location is the parent directory in which the SBT creates the folder. The default project location is the same as the default location for an application project. To place your BSP in a different folder, turn off **Use default location**, and specify the BSP location in the **Project location** box.

Selecting an Existing BSP

As an alternative to creating a BSP automatically from a template, you can associate your application project with a pre-existing BSP. Select **Select an existing BSP project from your workspace**, and select a BSP in the list. The **Create** and **Import** buttons to the right of the existing BSP list provide convenient ways to add BSPs to the list.

Creating the Projects

When you have specified your BSP, you click **Finish** to create the projects. The SBT copies required source files to your project directories, and creates makefiles and other generated files. Finally, the SBT executes a **make clean** command on your BSP.

For details about what happens when Nios II projects are created, refer to "Nios II Software Projects" in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*. For details about the **make clean** command, refer to "Makefiles" in the same chapter.

Navigating the Project

When you have created a Nios II project, it appears in the **Project Explorer** view, which is typically displayed at the left side of the Nios II perspective. You can expand each project to examine its folders and files.

For an explanation of the folders and files in a Nios II BSP, refer to "Nios II Software Projects" in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

Building the Project

To build a Nios II project in the Nios II SBT for Eclipse, right-click the project name and click **Build Project**. A progress bar shows you the build status. The build process can take a minute or two for a simple project, depending on the speed of the host machine. Building a complex project takes longer.

During the build process, you view the build commands and command-line output in the Eclipse **Console** view.

For details about Nios II SBT commands and output, refer to the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

When the build process is complete, the following message appears in the **Console** view, under the **C-Build [*<project name>*]** title:

```
[<project name> build complete]
```

If the project has a dependency on another project, such as a BSP or a user library, the SBT builds the dependency project first. This feature allows you to build an application and its BSP with a single command.

Configuring the FPGA

Before you can run your software, you must ensure that the correct hardware design is running on the FPGA. To configure the FPGA, you use the Quartus® II Programmer.

In the Windows operating system, you start the Quartus II Programmer from the Nios II SBT for Eclipse, through the Nios II menu. In the Linux operating system, you start Quartus II Programmer from the Quartus II software.

The project directory for your hardware design contains an SRAM Object File (**.sof**) along with the **.sopcinfo** file. The **.sof** file contains the hardware design to be programmed in the FPGA.

For details about programming an FPGA with Quartus II Programmer, refer to the *Quartus II Programmer* chapter in *Volume 3: Verification* of the *Quartus II Handbook*.

Running the Project on Nios II Hardware

This section describes how to run a Nios II program using the Nios II SBT for Eclipse on Nios II hardware, such as an Altera development board.

☞ If your project was created with version 10.1 or earlier of the Nios II SBT, you must re-import it to create the Nios II launch configuration correctly.

A Nios II instruction set simulator is available through the Lauterbach GmbH website (www.lauterbach.com).

To run a software project, right-click the application project name, point to **Run As**, and click **Nios II Hardware**. This command carries out the following actions:

- Creates a Nios II run configuration. For details about run configurations, refer to "Run Configurations in the SBT for Eclipse" on page 2–20.
- Builds the project executable. If all target files are up to date, nothing is built.
- Establishes communications with the target, and verifies that the FPGA is configured with the correct hardware design.
- Downloads the Executable and Linking Format File (**.elf)** to the target memory
- Starts execution at the **.elf** entry point.

Program output appears in the Nios II Console view. The Nios II Console view maintains a terminal I/O connection with a communication device connected to the Nios II processor in the hardware system, such as a JTAG UART. When the Nios II program writes to `stdout` or `stderr`, the Nios II Console view displays the text. The Nios II Console view can also accept character input from the host keyboard, which is sent to the processor and read as `stdin`.

To disconnect the terminal from the target, click the **Terminate** icon in the Nios II Console view. Terminating only disconnects the host from the target. The target processor continues executing the program.

Debugging the Project on Nios II Hardware

This section describes how to debug a Nios II program using the Nios II SBT for Eclipse on Nios II hardware, such as an Altera development board.

If your project was created with version 10.1 or earlier of the Nios II SBT, you must re-import it to create the Nios II launch configuration correctly.

To debug a software project, right-click the application project name, point to **Debug As**, and click **Nios II Hardware**. This command carries out the following actions:

- Creates a Nios II run configuration. For details about run configurations, refer to "Run Configurations in the SBT for Eclipse" on page 2–20.
- Builds the project executable. If all target files are up to date, nothing is built.
- If debugging on hardware, establishes communications with the target, and verifies that the FPGA is configured with the correct hardware design.
- Downloads the **.elf** to the target memory.
- Sets a breakpoint at the top of `main()`.
- Starts execution at the **.elf** entry point.

The Eclipse debugger with the Nios II plugins provides a Nios II perspective, allowing you to perform many common debugging tasks. Debugging a Nios II program with the Nios II plugins is generally the same as debugging any other C/C++ program with Eclipse and the CDT plugins.

For information about debugging with Eclipse and the CDT plugins, refer to the Eclipse help system.

The debugging tasks you can perform with the Nios II SBT for Eclipse include the following tasks:

- Controlling program execution with commands such as:
  - Suspend (pause)
  - Resume
  - Terminate
  - Step Into
  - Step Over
  - Step Return
- Setting breakpoints and watchpoints
- Viewing disassembly
- Instruction stepping mode
- Displaying and changing the values of local and global variables in the following formats:
  - Binary
  - Decimal
  - Hexadecimal
- Displaying watch expressions
- Viewing and editing registers in the following formats:
  - Binary
  - Decimal
  - Hexadecimal
- Viewing and editing memory in the following formats:
  - Hexadecimal
  - ASCII
  - Signed integer
  - Unsigned integer
- Viewing stack frames in the **Debug** view

Just as when running a program, Eclipse displays program output in the Console view of Eclipse. The Console view maintains a terminal I/O connection with a communication device connected to the Nios II processor in the hardware system, such as a JTAG UART. When the Nios II program writes to stdout or stderr, the Console view displays the text. The Console view can also accept character input from the host keyboard, which is sent to the processor and read as stdin.

To disconnect the terminal from the target, click the **Terminate** icon in the Console view. Terminating only disconnects the host from the target. The target processor continues executing the program.

If your project was created with version 10.1 or earlier of the Nios II SBT, you must re-import it to create the Nios II launch configuration correctly.

Using the Altera Bytestream Console

The Altera bytestream console enables you to see output from the processor's `stdout` and `stderr` devices, and send input to its `stdin` device. The function of the Altera bytestream console is similar to the **nios2-terminal** command-line utility.

Open the Altera bytestream console in the Eclipse **Console** view the same way as any other Eclipse console, by clicking the **Open Console** button.

When you open the Altera bytestream console, the **Bytestream Console Selection** dialog box shows you a list of available bytestreams. This is the same set of bytestreams recognized by System Console. Select the bytestream connected to the processor you are debugging.

For information about how System Console recognizes bytestreams, refer to the *Analyzing and Debugging Designs with the System Console* chapter in *Volume 3: Verification* of the *Quartus II Handbook*.

You can send characters to the processor's `stdin` device by typing in the bytestream console. Be aware that console input in buffered on a line-by-line basis. Therefore, the processor does not receive any characters until you press the Enter key.

A bytestream device can support only one connection at a time. You must close the Altera bytestream console before attempting to connect to the processor with the **nios2-terminal** utility, and vice versa.

Creating a Simple BSP

You create a BSP with default settings using the **Nios II Board Support Package** wizard. To start the wizard, on the File menu, point to **New** and click **Nios II Board Support Package**.

The **Nios II Board Support Package** wizard enables you to specify the following BSP parameters:

- The name
- The underlying hardware design
- The location

- The operating system and version

  You can select the operating system only at the time you create the BSP. To change operating systems, you must create a new BSP.

- Additional arguments to the **nios2-bsp** script

  If you intend to run the project in the Nios II ModelSim® simulation environment, use the **Additional arguments** parameter to specify the location of the testbench Simulation Package Descriptor File (**.spd**). The **.spd** file is located in the Quartus II project directory. Specify the path as follows:

  ```
  --set QUARTUS_PROJECT_DIR=<relative path>
  ```

  Altera recommends that you use a relative path name, to ensure that the location of your project is independent of the installation directory.

  For details about **nios2-bsp** command arguments, refer to "Details of BSP Creation" in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

After you have created the BSP, you have the following options for GUI-based BSP editing:

- To access and modify basic BSP properties, right-click the BSP project, point to **Properties** and click **Nios II BSP Properties**.
- To modify parameters and settings in detail, use the Nios II BSP Editor, described in "Using the BSP Editor".

Makefiles and the Nios II SBT for Eclipse

The Nios II SBT for Eclipse creates and manages the makefiles for Nios II software projects. When you create a project, the Nios II SBT creates a makefile based on the source content you specify and the parameters and settings you select. When you modify the project in Eclipse, the Nios II SBT updates the makefile to match.

Details of how each makefile is created and maintained vary depending on the project type, and on project options that you control. The authoritative specification of project contents is always the makefile, regardless how it is created or updated.

By default, the Nios II SBT manages the list of source files in your makefile, based on actions you take in Eclipse. However, in the case of applications and libraries, you have the option to manage sources manually. Both styles of source management are discussed in the following sections.

Eclipse Source Management

Nios II application and user library makefiles are based on source files and properties that you specify directly. Eclipse source management allows you to add and remove source files with standard Eclipse actions, such as dragging a source file into and out of the Project Explorer view and adding a new source file through the File menu.

You can examine and modify many makefile properties in the **Nios II Application Properties** or **Nios II Library Properties** dialog box. To open the dialog box, right-click the project, click **Properties**, and click **Nios II Application Properties** or **Nios II Library Properties**.

Table 2–1 lists GUI actions that make changes to an application or user library makefile under Eclipse source management.

Table 2–1. Modifying a Makefile with Eclipse Source Management

Modification	Where Modified
Specifying the application or user library name	**Nios II Application Properties** or **Nios II Library Properties** dialog box.
Adding or removing source files	Refer to the Eclipse help system.
Specifying a path to an associated BSP	**Project References** dialog box.
Specifying a path to an associated user library	**Project References** dialog box.
Enabling, disabling or modifying compiler options	**Nios II Application Properties** or **Nios II Library Properties** dialog box.

After the SBT has created a makefile, you can modify the makefile in the following ways:

- With the Nios II SBT for Eclipse, as described in Table 2–1.
- With Nios II SBT commands from the Nios II Command Shell.

When modifying a makefile, the SBT preserves any previous nonconflicting modifications, regardless how those modifications were made.

After you modify a makefile with the Nios II Command Shell, in Eclipse you must right-click the project and click **Update linked resource** to keep the Eclipse project view in step with the makefile.

When the Nios II SBT for Eclipse modifies a makefile, it locks the makefile to prevent corruption by other processes. You cannot edit the makefile from the command line until the SBT has removed the lock.

If you want to exclude a resource (a file or a folder) from the Nios II makefile temporarily, without deleting it from the project, you can use the **Remove from Nios II Build** command. Right-click the resource and click **Remove from Nios II Build**. When a resource is excluded from the build, it does not appear in the makefile, and Eclipse ignores it. However, it is still visible in the Project Explorer, with a modified icon. To add the resource back into the build, right-click the resource and click **Add to Nios II Build**.

☞ Do not use the Eclipse **Exclude from build** command. With Nios II software projects, you must use the **Remove from Nios II Build** and **Add to Nios II Build** commands instead.

Absolute Source Paths and Linked Resources

By default, the source files for an Eclipse project are stored under the project directory. If your project must incorporate source files outside the project directory, you can add them as linked resources.

An Eclipse linked resource can be either a file or a folder. With a linked folder, all source files in the folder and its subfolders are included in the build.

When you add a linked resource (file or folder) to your project, the SBT for Eclipse adds the file or folder to your makefile with an absolute path name. You might use a linked resource to refer to common source files in a fixed location. In this situation, you can move the project to a different directory without disturbing the common source file references.

A linked resource appears with a modified icon (green dot) in the Project Explorer, to distinguish it from source files and folders that are part of the project. You can use the Eclipse debugger to step into a linked source file, exactly as if it were part of the project.

You can reconfigure your project to refer to any linked resource either as an individual file, or through its parent folder. Right-click the linked resource and click **Update Linked Resource**.

You can use the **Remove from Nios II Build** and **Add to Nios II Build** commands with linked resources. When a linked resource is excluded from the build, its icon is modified with a white dot.

You can use Eclipse to create a path variable, defining the location of a linked resource. A path variable makes it easy to modify the location of one or more files in your project.

For information about working with path variables and creating linked resources, refer to the Eclipse help system.

User Source Management

You can remove a makefile from source management control through the **Nios II Application Properties** or **Nios II Library Properties** dialog box. Simply turn off **Enable source management** to convert the makefile to user source management. When **Enable source management** is off, you must update your makefile manually to add or remove source files to or from the project. The SBT for Eclipse makes no changes to the list of source files, but continues to manage all other project parameters and settings in the makefile.

Editing a makefile manually is an advanced technique. Altera recommends that you avoid manual editing. The SBT provides extensive capabilities for manipulating makefiles while ensuring makefile correctness.

In a makefile with user-managed sources, you can refer to source files with an absolute path. You might use an absolute path to refer to common source files in a fixed location. In this situation, you can move the project to a different directory without disturbing the common source file references.

Projects with user-managed sources do not support the following features:

- Linked resources
- The **Add to Nios II Build** command
- The **Remove from Nios II Build** command

Table 2–2 lists GUI actions that make changes to an application or user library makefile under user source management.

Table 2–2. Modifying a Makefile with User Source Management

Modification	Where Modified
Specifying the application or user library name	**Nios II Application Properties** or **Nios II Library Properties** dialog box
Specifying a path to an associated BSP	**Project References** dialog box
Specifying a path to an associated user library	**Project References** dialog box
Enabling, disabling or modifying compiler options	**Nios II Application Properties** or **Nios II Library Properties** dialog box

With user source management, the source files shown in the Eclipse Project Explorer view do not necessarily reflect the sources built by the makefile. To update the Project Explorer view to match the makefile, right-click the project and click **Sync from Nios II Build**.

BSP Source Management

Nios II BSP makefiles are handled differently from application and user library makefiles. BSP makefiles are based on the operating system, BSP settings, selected software packages, and selected drivers. You do not specify BSP source files directly.

BSP makefiles must be managed by the SBT, either through the BSP Editor or through the SBT command-line utilities.

For further details about specifying BSPs, refer to "Using the BSP Editor".

Using the BSP Editor

Typically, you create a BSP with the Nios II SBT for Eclipse. The Nios II plugins provide the basic tools and settings for defining your BSP. For more advanced BSP editing, use the Nios II BSP Editor. The BSP Editor provides all the tools you need to create even the most complex BSPs.

Tcl Scripting and the Nios II BSP Editor

The Nios II BSP Editor provides support for Tcl scripting. When you create a BSP in the BSP Editor, the editor can run a Tcl script that you specify to supply BSP settings.

You can also export a Tcl script from the BSP Editor, containing all the settings in an existing BSP. By studying such a script, you can learn about how BSP Tcl scripts are constructed.

Starting the Nios II BSP Editor

You start the Nios II BSP Editor in one of the following ways:

- Right-click an existing project, point to **Nios II**, and click **BSP Editor**. The editor loads the BSP Settings File (**.bsp**) associated with your project, and is ready to update it.
- On the Nios II menu, click **Nios II BSP Editor**. The editor starts without loading a **.bsp** file.
- Right-click an existing BSP project and click **Properties**. In the **Properties** dialog box, select **Nios II BSP Properties**, and click **BSP Editor**. The editor loads your **.bsp** file for update.

The Nios II BSP Editor Screen Layout

The Nios II BSP Editor screen is divided into two areas. The top area is the command area, and the bottom is the console area. The details of the Nios II BSP Editor screen areas are described in this section.

Below the console area is the **Generate** button. This button is enabled when the BSP settings are valid. It generates the BSP target files, as shown in the **Target BSP Directory** tab.

The Command Area

In the command area, you specify settings and other parameters defining the BSP. The command area contains several tabs:

- The **Main** tab
- The **Software Packages** tab
- The **Drivers** tab
- The **Linker Script** tab
- The **Enable File Generation** tab
- The **Target BSP Directory** tab

Each tab allows you to view and edit a particular aspect of the **.bsp**, along with relevant command line parameters and Tcl scripts.

The settings that appear on the **Main**, **Software Packages** and **Drivers** tabs are the same as the settings you manipulate on the command line.

For detailed descriptions of settings defined for Altera-provided operating systems, software packages, and drivers, refer to the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer’s Handbook*.

The Main Tab

The **Main** tab presents general settings and parameters, and operating system settings, for the BSP. The BSP includes the following settings and parameters:

- The path to the **.sopcinfo** file specifying the target hardware
- The processor name
- The operating system and version

  You cannot change the operating system in an existing BSP. You must create a new BSP based on the desired operating system.

- The BSP target directory—the destination for files that the SBT copies and creates for your BSP.
- BSP settings

BSP settings appear in a tree structure. Settings are organized into **Common** and **Advanced** categories. Settings are further organized into functional groups. The available settings depend on the operating system.

When you select a group of settings, the controls for those settings appear in the pane to the right of the tree. When you select a single setting, the pane shows the setting control, the full setting name, and the setting description.

Software package and driver settings are presented separately, as described in "The Software Packages Tab" and "The Drivers Tab".

The Software Packages Tab

The **Software Packages** tab allows you to insert and remove software packages in your BSP, and control software package settings.

At the top of the **Software Packages** tab is the software package table, listing each available software package. The table allows you to select the software package version, and enable or disable the software package.

The operating system determines which software packages are available.

Many software packages define settings that you can control in your BSP. When you enable a software package, the available settings appear in a tree structure, organized into **Common** and **Advanced** settings.

When you select a group of settings, the controls for those settings appear in the pane to the right of the tree. When you select a single setting, the pane shows the setting control, the full setting name, and the setting description.

Enabling and disabling software packages and editing software package settings can have a profound impact on BSP behavior. Refer to the documentation for the specific software package for details.

For the read-only zip file system, refer to the *Read-Only Zip File System* chapter of the *Nios II Software Developer's Handbook*. For the NicheStack TCP/IP Stack - Nios II Edition, refer to the *Ethernet and the NicheStack TCP/IP Stack - Nios II Edition* chapter of the *Nios II Software Developer's Handbook*.

General settings, operating system settings, and driver settings are presented separately, as described in "The Main Tab" and "The Drivers Tab".

The Drivers Tab

The **Drivers** tab allows you to select, enable, and disable drivers for devices in your system, and control driver settings.

At the top of the **Drivers** tab is the driver table, mapping components in the hardware system to drivers. The driver table shows components with driver support. Each component has a module name, module version, module class name, driver name, and driver version, determined by the contents of the hardware system. The table allows you to select the driver by name and version, as well as to enable or disable each driver.

When you select a driver version, all instances of that driver in the BSP are set to the version you select. Only one version of a given driver can be used in an individual BSP.

Many drivers define settings that you can control in your BSP. Available driver settings appear in a tree structure below the driver table, organized into **Common** and **Advanced** settings.

When you select a group of settings, the controls for those settings appear in the pane to the right of the tree. When you select a single setting, the pane shows the setting control, the full setting name, and the setting description.

☞ Enabling and disabling device drivers, changing drivers and driver versions, and editing driver settings, can have a profound impact on BSP behavior. Refer to the relevant component documentation and driver information for details. For Altera components, refer to the *Embedded Peripherals IP User Guide*.

General settings, operating system settings, and software package settings are presented separately, as described in "The Main Tab" and "The Software Packages Tab".

The Linker Script Tab

The **Linker Script** tab allows you to view available memory in your hardware system, and examine and modify the arrangement and usage of linker regions in memory.

When you make a change to the memory configuration, the SBT validates your change. If there is a problem, a message appears in the **Problems** tab in the console area, as described in "The Problems Tab" on page 2–18.

☞ Rearranging linker regions and linker section mappings can have a very significant impact on BSP behavior.

Linker Section Mappings

At the top of the **Linker Script** tab, the **Linker Section Mappings** table shows the mapping from linker sections to linker regions. You can edit the BSP linker section mappings using the following buttons located next to the linker section table:

- **Add**—Adds a linker section mapping to an existing linker region. The **Add** button opens the **Add Section Mapping** dialog box, where you specify a new section name and an existing linker region.
- **Remove**—Removes a mapping from a linker section to a linker region.
- **Restore Defaults**—Restores the section mappings to the default configuration set up at the time of BSP creation.

Linker Regions

At the bottom of the **Linker Script** tab, the **Linker Memory Regions** table shows all defined linker regions. Each row of the table shows one linker region, with its address range, memory device name, size, and offset into the selected memory device.

You reassign a defined linker region to a different memory device by selecting a different device name in the **Memory Device Name** column. The **Size** and **Offset** columns are editable. You can also edit the list of linker regions using the following buttons located next to the linker region table:

- **Add**—Adds a linker region in unused space on any existing device. The **Add** button opens the **Add Memory Region** dialog box, where you specify the memory device, the new memory region name, the region size, and the region's offset from the device base address.
- **Remove**—Removes a linker region definition. Removing a region frees the region's memory space to be used for other regions.
- **Add Memory Device**—Creates a linker region representing a memory device that is outside the hardware system. The button launches the **Add Memory Device** dialog box, where you can specify the device name, memory size and base address. After you add the device, it appears in the linker region table, the **Memory Device Usage Table** dialog box, and the **Memory Map** dialog box.

  This functionality is equivalent to the `add_memory_device` Tcl command.

  ☞ Ensure that you specify the correct base address and memory size. If the base address or size of an external memory changes, you must edit the BSP manually to match. The SBT does not automatically detect changes in external memory devices, even if you update the BSP by creating a new settings file.

  For information about `add_memory_device` and other SBT Tcl commands, refer to "Software Build Tools Tcl Commands" in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook.*

- **Restore Defaults**—restores the memory regions to the default configuration set up at the time of BSP creation.

- **Memory Usage**—Opens the **Memory Device Usage Table**. The **Memory Device Usage Table** allows you to view memory device usage by defined memory region. As memory regions are added, removed, and adjusted, each device's free memory, used memory, and percentage of available memory are updated. The rightmost column is a graphical representation of the device's usage, according to the memory regions assigned to it.
- **Memory Map**—Opens the **Memory Map** dialog box. The memory map allows you to view a map of system memory in the processor address space. The **Device** table is a read-only reference showing memories in the hardware system that are mastered by the selected processor. Devices are listed in memory address order.

  To the right of the **Device** table is a graphical representation of the processor's memory space, showing the locations of devices in the table. Gaps indicate unmapped address space.

  This representation is not to scale.

Enable File Generation Tab

The **Enable File Generation** tab allows you to take ownership of specific BSP files that are normally generated by the SBT. When you take ownership of a BSP file, you can modify it, and prevent the SBT from overwriting your modifications. The **Enable File Generation** tab shows a tree view of all target files to be generated or copied when the BSP is generated. To disable generation of a specific file, expand the software component containing the file, expand any internal directory folders, select the file, and right-click. Each disabled file appears in a list at the bottom of the tab.

This functionality is equivalent to the `set_ignore_file` Tcl command.

☞ If you take ownership of a BSP file, the SBT can no longer update it to reflect future changes in the underlying hardware. If you change the hardware, be sure to update the file manually.

For information about `set_ignore_file` and other SBT Tcl commands, refer to "Software Build Tools Tcl Commands" in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook.*

Target BSP Directory Tab

The **Target BSP Directory** tab is a read-only reference showing you what output to expect when the BSP is generated. It does not depict the actual file system, but rather the files and directories to be created or copied when the BSP is generated. Each software component, including the operating system, drivers, and software packages, specifies source code to be copied into the BSP target directory. The files are generated in the directory specified on the **Main** tab.

When you generate the BSP, existing BSP files are overwritten, unless you disable generation of the file in the **Enable File Generation** tab.

The Console Area

The console area shows results of settings and commands that you select in the command area. The console area consists of the following tabs:

- The **Information** tab
- The **Problems** tab
- The **Processing** tab

The following sections describe each tab.

The Information Tab

The **Information** tab shows a running list of high-level changes you make to your BSP, such as adding a software package or changing a setting value.

The Problems Tab

The **Problems** tab shows warnings and errors that impact or prohibit BSP creation. For example, if you inadvertently specify an invalid linker section mapping, a message appears in the **Problems** tab.

The Processing Tab

When you generate your BSP, the **Processing** tab shows files and folders created and copied in the BSP target directory.

Exporting a Tcl Script

When you have configured your BSP to your satisfaction, you can export the BSP settings as a Tcl script. This feature allows you to perform the following tasks:

- Regenerate the BSP from the command line
- Recreate the BSP as a starting point for a new BSP
- Recreate the BSP on a different hardware platform
- Examine the Tcl script to improve your understanding of Tcl command usage

The exported Tcl script captures all BSP settings that you have changed since the previous time the BSP settings file was saved. If you export a Tcl script after creating a new BSP, the script captures all nondefault settings in the BSP. If you export a Tcl script after editing a pre-existing BSP, the script captures your changes from the current editing session.

To export a Tcl script, in the Tools menu, click **Export Tcl Script**, and specify a filename and destination path. The file extension is **.tcl**.

You can later run your exported script as a part of creating a new BSP.

To run a Tcl script during BSP creation, refer to "Using a Tcl Script in BSP Creation". For details about default BSP settings, refer to "Tcl Scripts for BSP Settings" in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.For information about recreating and regenerating BSPs, refer to "Revising Your BSP" in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

Creating a New BSP

To create a BSP in the Nios II BSP Editor, use the **New BSP** command in the File menu to open the **New BSP** dialog box. This dialog box controls the creation of a new BSP settings file. The BSP Editor loads this new BSP after the file is created.

In this dialog box, you specify the following parameters:

- The .**sopcinfo** file defining the hardware platform.
- The CPU name of the targeted processor.
- The BSP type and version.

☞ You can select the operating system only at the time you create the BSP. To change operating systems, you must create a new BSP.

- The operating system version.
- The name of the BSP settings file. It is created with file extension .**bsp**.
- Absolute or relative path names in the BSP settings file. By default, relative paths are enabled for filenames in the BSP settings file.
- An optional Tcl script that you can run to supply additional settings.

Normally, you specify the path to your .**sopcinfo** file relative to the BSP directory. This enables you to move, copy and archive the hardware and software files together. If you browse to the .**sopcinfo** file, or specify an absolute path, the Nios II BSP Editor offers to convert your path to the relative form.

Using a Tcl Script in BSP Creation

When you create a BSP, the **New BSP Settings File** dialog box allows you to specify the path and filename of a Tcl script. The Nios II BSP Editor runs this script after all other BSP creation steps are done, to modify BSP settings. This feature allows you to perform the following tasks:

- Recreate an existing BSP as a starting point for a new BSP
- Recreate a BSP on a different hardware platform
- Include custom settings common to a group of BSPs

The Tcl script can be created by hand, or exported from another BSP.

"Exporting a Tcl Script" describes how to create a Tcl script from an existing BSP. Refer to "Tcl Scripts for BSP Settings" in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook.*

BSP Validation Errors

If you modify a hardware system after basing a BSP on it, some BSP settings might no longer be valid. This is a very common cause of BSP validation errors. Eliminating these errors usually requires correcting a large number of interrelated settings.

If your modifications to the underlying hardware design result in BSP validation errors, the best practice is to update or recreate the BSP. Updating and recreating BSPs is very easy with the BSP Editor.

For complete information about updating and recreating BSPs, refer to "Revising Your BSP" in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

If you recreate your BSP, you might find it helpful to capture your old BSP settings by exporting them to a Tcl script. You can edit the Tcl script to remove any settings that are incompatible with the new hardware design.

For details about exporting and using Tcl scripts, refer to "Exporting a Tcl Script" and "Using a Tcl Script in BSP Creation". For a detailed discussion of updating BSPs for modified hardware systems, refer to "Revising Your BSP" in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

Configuring Component Search Paths

By default, the SBT discovers system components using the same search algorithm as SOPC Builder or Qsys. You can define additional search paths to be used for locating components.

You define additional search paths through the **Edit Custom Search Paths** dialog box. In the Tools menu, click **Options**, select **BSP Component Search Paths**, and click **Custom Component Search Paths**. You can specify multiple search paths. Each path can be recursive.

Run Configurations in the SBT for Eclipse

Eclipse uses run configurations to control how it runs and debugs programs. Run configurations in the Nios II SBT for Eclipse have several features that help you debug Nios II software running on FPGA platforms.

You can open the run configuration dialog box two ways:

- You can right-click an application, point to **Run As**, and click **Run Configurations**.
- You can right-click an application, point to **Debug As**, and click **Debug Configurations**.

Depending on which way you opened the run configuration dialog box, the title is either **Run Configuration** or **Debug Configuration**. However, both views show the same run configurations.

☞ If your project was created with version 10.1 or earlier of the Nios II SBT, you must re-import it to create the Nios II launch configuration correctly.

Each run configuration is presented on several tabs. This section describes each tab.

The Project Tab

On this tab, you specify the application project to run. The **Advanced** button opens the **Nios II ELF Section Properties** dialog box. In this dialog box, you can control the runtime parameters in the following ways:

- Specify the processor on which to execute the program (if the hardware design provides multiple processors)
- Specify the device to use for standard I/O
- Specify the expected location, timestamp and value of the system ID
- Specify the path to the Quartus II JTAG Debugging Information File (**.jdi**)
- Enable or disable profiling

The Nios II SBT for Eclipse sets these parameters to reasonable defaults. Do not modify them unless you have a clear understanding of their effects.

The Target Connection Tab

This tab allows you to control the connection between the host machine and the target hardware in the following ways:

- Select the cable, if more than one cable is available
- Allow software to run despite a system ID value or timestamp that differs from the hardware
- Reset the processor when the software is downloaded

The **System ID Properties** button allows you to examine the system ID and timestamp in both the .**elf** file and the hardware. This can be helpful when you need to analyze the cause of a system ID or timestamp mismatch.

The Debugger Tab

In this tab, you optionally enable the debugger to halt at a specified entry point.

Nios II Hardware v2 (beta)

Starting with version 13.1, run configurations and debug configurations have a launch type called Nios II Hardware v2 (beta). To create this launch type, in the Run menu select either **Run Configurations** or **Debug Configurations**. In the **Run/Debug Configurations** dialog box, select **Nios II Hardware v2 (beta)** and click the **New** button to create a new launch configuration.

Nios II Hardware v2(beta) has options below.

The Main Tab

This tab allows you to select the following options:

- Specify the application project to run and the ELF File location
- Specify the processor and the JTAG UART connection to use
- Enable or disable system ID and timestamp checks
- Enable or disable processor controls such as download ELF, reset processor or start processor

The Debugger Tab

In this tab, you optionally enable the debugger to halt at a specified entry point.

Multi-Core Launches

If you have multiple run configurations, create an Eclipse launch group. Launch groups are an Eclipse feature that allows multiple run configurations to be started at the same time. You choose which run configurations are added to the group. You can use the launch group in any place where you can use a run configuration.

For details about Eclipse launch groups, refer to the Eclipse help system.

Optimizing Project Build Time

When you build a Nios II project, the project makefile builds any components that are unbuilt or out of date. For this reason, the first time you build a project is normally the slowest. Subsequent builds are fast, only rebuilding sources that have changed.

To further optimize your project build time, disable generation of the objdump linker map.

Nios II software build performance is generally better on Linux platforms than on Windows platforms.

Importing a Command-Line Project

If you have software projects that were created with the Nios II SBT command line, you can import the projects into the Nios II SBT for Eclipse for debugging and further development. This section discusses the import process.

Your command-line C/C++ application, and its associated BSP, is created on the command line. Any Nios II SBT command-line project is ready to import into the Nios II SBT for Eclipse. No additional preparation is necessary.

The Nios II SBT for Eclipse imports the following kinds of Nios II command-line projects:

- Command-line C/C++ application project
- Command-line BSP project
- Command-line user library project

You can edit, build, debug, and manage the settings of an imported project exactly the same way you edit, build, debug, and manage the settings of a project created in Nios II SBT for Eclipse.

The Nios II SBT for Eclipse 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 SBT project after importing the project into Eclipse. You can edit source files and rebuild the project, using the SBT either in Eclipse or on the command line.

For information about creating projects with the command line, refer to the *Getting Started from the Command Line* 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 command-line C/C++ application
- Import a supporting project
- Debug a command-line C/C++ application
- Edit command-line C/C++ application code

When importing a project, the SBT for Eclipse might make some minor changes to your makefile. If the makefile refers to a source file located outside the project directory tree, the SBT for Eclipse treats that file as a linked resource. However, it does not add or remove any source files to or from your makefile.

When you import an application or user library project, the Nios II SBT for Eclipse allows you to choose Eclipse source management or user source management. Unless your project has an unusual directory structure, choose Eclipse source management, to allow the SBT for Eclipse to automatically maintain your list of source files.

You debug and edit an imported project exactly the same way you debug and edit a project created in Eclipse.

Import a Command-Line C/C++ Application

To import a command-line C/C++ application, perform the following steps:

1. Start the Nios II SBT for Eclipse.
2. On the File menu, click **Import**. The **Import** dialog box appears.
3. Expand the **Nios II Software Build Tools Project** folder, and select **Import Nios II Software Build Tools Project**.
4. Click **Next**. The **File Import** wizard appears.
5. Click **Browse** and locate the directory containing the C/C++ application project to import.
6. Click **OK**. The wizard fills in the project path.

7. Specify the project 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 Eclipse project. Either it is an Eclipse project, or it is a command-line project that is already imported into Eclipse.

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

8. Click **Finish**. The wizard imports the application project.

After you complete these steps, the Nios II SBT for Eclipse can build, debug, and run the complete program, including the BSP and any libraries. The Nios II SBT for Eclipse builds the project using the SBT makefiles in your imported C/C++ application project. Eclipse displays and steps through application source code exactly as if the project were created in the Nios II SBT for Eclipse. However, Eclipse does not have direct information about where BSP or user library code resides. If you need to view, debug or step through BSP or user library source code, you need to import the BSP or user library. The process of importing supporting projects, such as BSPs and libraries, is described in "Import a Supporting Project".

Importing a Project with Absolute Source Paths

If your project uses an absolute path to refer to a source file, the SBT for Eclipse imports that source file as a linked resource. In this case, the import wizard provides a page where you can manage how Eclipse refers to the source: as a file, or through a parent directory.

For information about managing linked resources, refer to "Absolute Source Paths and Linked Resources" on page 2–10.

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 user library. To make supporting project source code visible in the Eclipse debug perspective, you need to import the supporting project.

If you do not need BSP or user library source code visible in the debugger, you can skip this task, and proceed to debug your project exactly as if you had created it in Eclipse.

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

The steps for importing a supporting project are exactly the same as those shown in "Import a Command-Line C/C++ Application".

User-Managed Source Files

When you import a Nios II application or user library project, the Nios II SBT for Eclipse offers the option of user source management. User source management is helpful if you prefer to update your makefile manually to reflect source files added to or removed from the project.

With user source management, Eclipse never makes any changes to the list of source files in your makefile. However, the SBT for Eclipse manages all other project parameters and settings, just as with any other Nios II software project.

If your makefile refers to a source file with an absolute path, when you import with user source management, the absolute path is untouched, like any other source path. You might use an absolute path to refer to common source files in a fixed location. In this situation, you can move the project to a different directory without disturbing the common source file references.

User source management is not available with BSP projects. BSP makefiles are based on the operating system, BSP settings, selected software packages, and selected drivers. You do not specify BSP source files directly.

For details about how the SBT for Eclipse handles makefiles with user-managed sources, refer to "User Source Management" on page 2–11.

Packaging a Library for Reuse

This section shows how to create and use a library archive file (**.a**) in the Nios II Software Build Tools for Eclipse. This technique enables you to provide a library to another engineer or organization without providing the C source files. This process entails two tasks:

1. Create a Nios II user library
2. Create a Nios II application project based on the user library

Creating the User Library

To create a user library, perform the following steps:

1. In the File menu, point to **New** and click **Nios II Library**.
2. Type a project name, for example `test_lib`.
3. For **Location**, browse to the directory containing your library source files (**.c** and **.h**).
4. Click **Finish**.
5. Build the project to create the **.a** file (in this case **libtest_lib.a**)

Using the Library

To use the library in a Nios II application project, perform the following steps:

1. Create your Nios II application project as described in "Creating a Project" on page 2–2.
2. To set the library path in the application project, right-click the project, and click **Properties**.
3. Expand **Nios II Application Properties**. In **Nios II Application Paths**, next to **Application include directories**, click **Add** and browse to the directory containing your library header files.
4. Next to **Application library directories**, click **Add** and browse to the directory containing your **.a** file.
5. Next to **Library name**, click **Add** and type the library project name you selected in "Creating the User Library".
6. Click **OK**.
7. Build your application.

As this example shows, the **.c** source files are not required to build the application project. To hand off the library to another engineer or organization for reuse, you provide the following files:

- Nios II library archive file (**.a**)
- Software header files (**.h**)

Creating a Software Package

This section shows how you can build a custom library into a BSP as a software package. The software package can be linked to any BSP through the BSP Editor.

This section contains an example illustrating the steps necessary to include any software package into a Nios II BSP.

To create and exercise the example software package, perform the following steps:

1. Locate the **ip** directory in your Altera Complete Design Suite installation. For example, if the Altera Complete Design Suite version 11.0 is installed on the Windows operating system, the directory might be **c:\altera\11.0\ip**. Under the **ip** directory, create a directory for the software package. For simplicity, this section refers to this directory as <*example package*>.
2. In <*example package*>, create a subdirectory named **EXAMPLE_SW_PACKAGE**. In <*example package*>/**EXAMPLE_SW_PACKAGE**, create two subdirectories named **inc** and **lib**.
3. In <*example package*>/**EXAMPLE_SW_PACKAGE/inc**, create a new header file named **example_sw_package.h**. Insert the code shown in Example 2–1.

Example 2–1. Contents of example_sw_package.h

```
/* Example Software Package */

void example_sw_package(void);
```

4. In <*example package*>/**EXAMPLE_SW_PACKAGE/lib**, create a new C source file named **example_sw_package.c**. Insert the code shown in Example 2–2.

Example 2–2. Contents of example_sw_package.c

```
/* Example Software Package  */
#include <stdio.h>
#include "..\inc\example_sw_package.h"

void example_sw_package(void)
{
    printf ("Example Software Package. \n");
}
```

5. In <*example package*>, create a new Tcl script file named **example_sw_package_sw.tcl**. Insert the code shown in Example 2–3.
6. In the SBT for Eclipse, create a Nios II application and BSP project based on the Hello World template. Set the application project name to `hello_example_sw_package`.

Example 2–3. Contents of example_sw_package_sw.tcl

```
#
# example_sw_package_sw.tcl
#

# Create a software package known as "example_sw_package"
create_sw_package example_sw_package

# The version of this software
set_sw_property version 11.0

# Location in generated BSP that sources should be copied into
set_sw_property bsp_subdirectory Example_SW_Package

#
# Source file listings...
#

# C/C++ source files
#add_sw_property c_source EXAMPLE_SW_PACKAGE/src/my_source.c

# Include files
add_sw_property include_source
EXAMPLE_SW_PACKAGE/inc/example_sw_package.h

# Lib files
add_sw_property lib_source
EXAMPLE_SW_PACKAGE/lib/libexample_sw_package_library.a

# Include paths for headers which define the APIs for this package
# to share w/ app & bsp
# Include paths are relative to the location of this software
# package tcl file

add_sw_property include_directory EXAMPLE_SW_PACKAGE/inc

# This driver supports HAL & UCOSII BSP (OS) types
add_sw_property supported_bsp_type HAL
add_sw_property supported_bsp_type UCOSII

# Add example software package system.h setting to the BSP:
add_sw_setting quoted_string system_h_define \
  example_sw_package_system_value EXAMPLE_SW_PACKAGE_SYSTEM_VALUE 1 \
  "Example software package system value"

# End of file
```

7. Create a new C file named **hello_example_sw_package.c** in the new application project. Insert the code shown in Example 2–4.

Example 2–4. Contents of hello_example_sw_package.c

```
/*
 * "Hello World" example.
 *
 * This example prints 'Hello from Nios II' to the STDOUT stream. It also
 * tests inclusion of a user software package.
 */

#include <stdio.h>
#include "example_sw_package.h"

int main()
{
  printf("Hello from Nios II!\n");
  example_sw_package();
  return 0;
}
```

8. Delete **hello_world.c** from the hello_example_sw_package application project.
9. In the File menu, point to **New** and click **Nios II Library**
10. Set the project name to `example_sw_package_library`.
11. For **Location**, browse to <*example package*>**\EXAMPLE_SW_PACKAGE\lib**

    Building the library here is required, because the resulting **.a** is referenced here by **example_sw_package_sw.tcl**.

12. Click **Finish**.
13. Build the example_sw_package_library project to create the **libexample_sw_package_library.a** library archive file.
14. Right-click the BSP project, point to **Nios II**, and click **BSP Editor** to open the BSP Editor.
15. In the **Software Packages** tab, find example_sw_package in the software package table, and enable it.

    If there are any errors in a software package's ***_sw.tcl** file, such as an incorrect path that causes a file to not be found, the software package does not appear in the BSP Editor.

16. Click the **Generate** button to regenerate the BSP. On the File menu, click **Save** to save your changes to **settings.bsp**.
17. In the File menu, click **Exit** to exit the BSP Editor.
18. Build the hello_example_sw_package_bsp BSP project.
19. Build the hello_example_sw_package application project.

hello_example_sw_package.elf is ready to download and execute.

Programming Flash in Altera Embedded Systems

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 SBT for Eclipse provides flash programmer utilities to help you manage and program the contents of flash memory. The flash programmer allows you to program any combination of software, hardware, and binary data into flash memory in one operation.

Starting the Flash Programmer

You start the flash programmer by clicking **Flash Programmer** in the Nios II menu.

When you first open the flash programmer, no controls are available until you open or create a Flash Programmer Settings File (**.flash-settings**).

Creating a Flash Programmer Settings File

The .**flash-settings** file describes how you set up the flash programmer GUI to program flash. This information includes the files to be programmed to flash, a **.sopcinfo** file describing the hardware configuration, and the file programming locations. You must create or open a flash programmer settings file before you can program flash.

You create a flash programmer settings file through the File menu. When you click **New**, the **New Flash Programmer Settings File** dialog box appears.

Specifying the Hardware Configuration

You specify the hardware configuration by opening a .**sopcinfo** file. You can locate the .**sopcinfo** file in either of two ways:

- Browse to a BSP settings file. The flash programmer finds the .**sopcinfo** file associated with the BSP.
- Browse directly to a .**sopcinfo** file.

Once you have identified a hardware configuration, details about the target hardware appear at the top of the Nios II flash programmer screen.

Also at the top of the Nios II flash programmer screen is the **Hardware Connections** button, which opens the **Hardware Connections** dialog box. This dialog box allows you to select a download cable, and control system ID behavior, as described in "The Target Connection Tab" on page 2–21.

The Flash Programmer Screen Layout

The flash programmer screen is divided into two areas. The top area is the command area, and the bottom is the console area. The details of the flash programmer screen areas are described in this section.

Below the console area is the **Start** button. This button is enabled when the flash programmer parameters are valid. It starts the process of programming flash.

The Command Area

In the command area, you specify settings and other parameters defining the flash programmer settings file. The command area contains one or more tabs. Each tab represents a flash memory component available in the target hardware. Each tab allows you to view the parameters of the memory component, and view and edit the list of files to be programmed in the component.

The **Add** and **Remove** buttons allow you to create and edit the list of files to be programmed in the flash memory component.

The **File generation command** box shows the commands used to generate the Motorola S-record Files (**.flash**) used to program flash memory.

The **File programming command** box shows the commands used to program the .**flash** files to flash memory.

The **Properties** button opens the **Properties** dialog box, which allows you to view and modify information about an individual file. In the case of a .**elf**, the **Properties** button provides access to the project reset address, the flash base and end addresses, and the boot loader file (if any).

The flash programmer determines whether a boot loader is required based on the load and run locations of the `.text` section. You can use the **Properties** dialog box to override the default boot loader configuration.

The Console Area

The console area shows results of settings and commands that you select in the command area. The console area consists of the following tabs:

- The **Information** tab
- The **Problems** tab
- The **Processing** tab

This section describes each tab.

The Information Tab

The **Information** tab shows the high-level changes you make to your flash programmer settings file.

The Problems Tab

The **Problems** tab shows warnings and error messages about the process of flash programmer settings file creation.

The Processing Tab

When you program flash, the **Processing** tab shows the individual programming actions as they take place.

Saving a Flash Programmer Settings File

When you have finished configuring the input files, locations, and other settings for programming your project to flash, you can save the settings in a **.flash-settings** file. With a **.flash-settings** file, you can program the project again without reconfiguring the settings. You save a **.flash-settings** file through the File menu.

Flash Programmer Options

Through the Options menu, you can control several global aspects of flash programmer behavior, as described in this section.

For details about these features, refer to the *Nios II Flash Programmer User's Guide*.

Staging Directories

Through the **Staging Directories** dialog box, you control where the flash programmer creates its script and **.flash-settings** files.

Generate Files

If you disable this option, the flash programmer does not generate programming files, but programs files already present in the directory. You might use this feature to reprogram a set of files that you have previously created.

Program Files

If you disable this option, the flash programmer generates the programming files and the script, but does not program flash. You can use the files later to program flash by turning off the **Generate Files** option.

Erase Flash Before Programming

When enabled, this option erases flash memory before programming.

Run From Reset After Programming

When enabled, this option resets and starts the Nios II processor after programming flash.

Creating Memory Initialization Files

Sometimes it is useful to generate memory initialization files. For example, to program your FPGA with a complete, running Nios II system, you must include the memory contents in your **.sof** file. In this configuration, the processor can boot directly from internal memory without downloading.

Creating a Hexadecimal (Intel-Format) File (**.hex**) is a necessary intermediate step in creating such a **.sof** file. The Nios II SBT for Eclipse can create **.hex** files and other memory initialization formats.

To generate correct memory initialization files, the Nios II SBT needs details about the physical memory configuration and the types of files required. Typically, this information is specified when the hardware system is generated.

☞ If your system contains a user-defined memory, you must specify these details manually. For information, see "Memory Initialization Files for User-Defined Memories".

To generate memory initialization files, perform the following steps:

1. Right-click the application project.
2. Point to **Make targets** and click **Build** to open the **Make Targets** dialog box.
3. Select **mem_init_generate**.
4. Click **Build**. The makefile generates a separate file (or files) for each memory device. It also generates a Quartus II IP File (**.qip**). The **.qip** file tells the Quartus II software where to find the initialization files.
5. Add the **.qip** file to your Quartus II project.
6. Recompile your Quartus II project.

If your hardware system was generated with SOPC Builder, you can alternatively use the legacy method to generate memory initialization files. However, this method is not preferred. To generate memory initialization files by the legacy method, perform the following steps:

1. Right-click the application project.
2. Point to **Make targets** and click **Build** to open the **Make Targets** dialog box.
3. Select **mem_init_install**.
4. Click **Build**. The makefile generates a separate file (or files) for each memory device. The makefile inserts the memory initialization files directly in the Quartus II project directory for you.
5. Recompile your Quartus II project.

For more information about creating memory initialization files, refer to "Common BSP Tasks" in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

Memory Initialization Files for User-Defined Memories

Generating memory initialization files requires detailed information about the physical memory devices, such as device names and data widths. Normally, the Nios II SBT extracts this information from the **.sopcinfo** file. However, in the case of a user-defined memory, the **.sopcinfo** file does not contain information about the data memory, which is outside the system. Therefore, you must provide this information manually.

You specify memory device information when you add the user-defined memory device to your BSP. The device information persists in the BSP settings file, allowing you to regenerate memory initialization files at any time, exactly as if the memory device were part of the hardware system.

Specify the memory device information in the **Advanced** tab of the **Add Memory Device** dialog box. Settings in this tab control makefile variables in **mem_init.mk**.

On the **Advanced** tab, you can control the following memory characteristics:

- The physical memory width.
- The device's name in the hardware system.
- The memory initialization file parameter name. Every memory device can have an HDL parameter specifying the name of the initialization file. The Nios II ModelSim launch configuration overrides the HDL parameter to specify the memory initialization filename. When available, this method is preferred for setting the memory initialization filename.

  ☞ For further information about this parameter, refer to "Embedded Software Assignments" in the *Publishing Component Information to Embedded Software* chapter of the *Nios II Software Developer's Handbook*.

- The **Mem init filename** parameter can be used in Nios II systems as an alternative method of specifying the memory initialization filename. The **Mem init filename** parameter directly overrides any filename specified in the HDL.
- Connectivity to processor master ports. These parameters are used when creating the linker script.
- The memory type: volatile, CFI flash or EPCS flash.
- Byte lanes.

You can also enable and disable generation of the following memory initialization file types:

- **.hex** file
- **.dat** and **.sym** files
- **.flash** file

Running a Nios II System with ModelSim

You can run a Nios II program on Nios II hardware, such as an Altera development board, or you can run it in the Nios II ModelSim® simulation environment.

☞ If your project was created with version 10.1 or earlier of the Nios II SBT, you must re-import it to create the Nios II launch configuration correctly.

Using ModelSim with an SOPC Builder-Generated System

If your hardware system was generated by SOPC Builder, running a software project in ModelSim is very similar to running it on Nios II hardware. Follow the instructions in "Running the Project on Nios II Hardware" on page 2–5, except that when you right-click the application project name, point to **Run As**, and click **Nios II ModelSim**.

Similarly, to debug a software project in ModelSim, right-click the application project name, point to **Debug As**, and click **Nios II ModelSim**.

Using ModelSim with a Qsys-Generated System

To run a Qsys-generated Nios II system with ModelSim, you must first create a simulation model and testbench, and specify memory initialization files. You create your Nios II simulation model and testbench using the steps that apply to any Qsys design.

Refer to "Qsys Design Flow" in the *Creating a System with Qsys* chapter in Volume 1 of the *Quartus II Handbook*.

Creating the software projects is nearly the same as when you run the project on hardware. To prepare your software for ModelSim simulation, perform the following steps:

1. Create your software project, as described in "Creating a Project" on page 2–2.

   Be sure to specify the Quartus II project path, as described in "Creating a Simple BSP" on page 2–8.

   If you need to initialize a user-defined memory, you must take special steps to create memory initialization files correctly. These steps are described in "Memory Initialization Files for User-Defined Memories" on page 2–33.

2. Build your software project, as described in "Building the Project" on page 2–5.
3. Create a ModelSim launch configuration with the following steps:
   a. Right-click the application project name, point to **Run As**, and click **Run Configurations**. In the **Run Configurations** dialog box, select **Nios II ModelSim**, and click the **New** button.
   b. In the **Main** tab, ensure that the correct software project name and **.elf** file are selected.
   c. Click **Apply** to save the launch configuration.
   d. Click **Close** to close the dialog box.

   If you are simulating multiple processors, create a launch configuration for each processor, and create a launch group, as described in "Multi-Core Launches" on page 2–22.

4. Open the run configuration you previously created. Click **Run**. The Nios II SBT for Eclipse performs a `make mem_init_generate` command to create memory initialization files, and launches ModelSim.
5. At the ModelSim command prompt, type `ld`↵.

☞ When you create the launch configuration, you might see the following error message:

SEVERE: The Quartus II project location has not been set in the ELF section. You can manually override this setting in the launch configuration's ELF file 'Advanced' properties page.

To correct this error, perform the following steps:

1. Click the **Advanced** button.
2. In the **Quartus II project directory** box, browse to locate the directory containing your Quartus II project **.spd** file.
3. Click **Close**.

To avoid this error condition, specify the Quartus II project directory when you create your application project, as described in "Creating a Simple BSP" on page 2–8.

☞ Starting with version 13.1, run configurations has the launch type Nios II Hardware v2 (beta). To create this launch type, in the Run menu select **Run Configurations**. In the **Run Configurations** dialog box, select **Nios II Hardware v2 (beta)** and click the **New** button to create a new launch configuration. Nios II Hardware v2 (beta) has the following options:

- Specify the application project to run and the ELF file location
- Specify the SPD file location and Modelsim path
- Specify the SPD file

Nios II GCC Tool chain upgrade from GCC 4.1.2 to GCC 4.7.3

In Nios II EDS version 13.1, the Nios® II GNU tool chain is upgraded from GCC 4.1.2 to GCC 4.7.3. When upgrading to the new tool chain you should note the following changes.

Nios II specific changes:

Use __buildin_custom_* instead of -mcustom-* or #pragma to reliably generate Nios II Floating Point Custom Instructions (FPCI), independent of compiler optimization level and command line flags.

To use -mcustom-* or #pragma for Nios II Floating Point Custom Instructions (FPCI):

the -ffinite-math-only flag must be used to generate fmins and fmax FPCI

the optimization (non -O0 flag) must be used to generate fsqrts FPCI

Users implementing transcendental functions in hardware must use the -funsafe-math-optimizations flag to generate the FPCI for the transcendental functions fsins(), fcoss(), ftans(), fatans(), fexps(), flogs() and corresponding double-precision functions

The Pragma format has changed from eg. #pragma custom_fadds 253 to #pragma GCC target("custom-fadds=253") and function attributes provide an alternative format __attribute__((target("custom-fadds=253"))).

Use the -mel/-meb flags instead of -EL/-EB for endian settings. Software Build Tool for Eclipse (SBTE) users must regenerate the BSP for this setting to take effect.

The -mreverse-bitfields flag and reverse_bitfields pragma are no longer supported.

The -fstack-check flag must be used instead of -mstack-check to enable stack checking.

GCC changes and enhancements:

The -Wa,-relax-all flag in nios2-elf-gcc GCC 4.7.3 supports function calls and programs exceeding the 256MB limit.

When used with optimization, inline assembly code with the asm operator needs to declare values imported from C and exported back to C, using the mechanisms described in "http://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html" \l "Extended-Asm", http://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html#Extended-Asm.

Pre-standard C++ headers are not supported in GCC 4.7.3. Replace pre-standard C++ with standard C++ eg. #include <iostream.h>, cout, endl with #include <iostream>, std::cout and std::endl respectively.

The compile flag -Wl,--defsym foo=bar where bar is an undefined symbol, will generate error at the linker level in GCC 4.7.3. GCC 4.1.2 does not include this check.

GNU also provides a porting guide to GCC4.7 to document common issues at :*http://gcc.gnu.org/gcc-4.7/porting_to.html*

Full GCC release notes are available at *http://gcc.gnu.org/releases.html*.

For general information about the GCC toolchains, refer to “Altera-Provided Development Tools” in the *Nios II Software Build Tools* chapter in the *Nios II Software Developer’s Handbook*. For information about selecting the toolchain on the command line, refer to the *Getting Started from the Command Line* chapter of the *Nios II Software Developer’s Handbook*. For detailed information about installing the Altera Complete Design Suite, refer to the *Altera Software Installation and Licensing Manual*.

Eclipse Usage Notes

The behavior of certain Eclipse and CDT features is modified by the Nios II SBT for Eclipse. If you attempt to use these features the same way you would with a non-Nios II project, you might have problems configuring or building your project. This section discusses such features.

Configuring Application and Library Properties

To configure project properties specific to Nios II SBT application and library projects, use the **Nios II Application Properties** and **Nios II Library Properties** tabs of the **Properties** dialog box. To open the appropriate properties tab, right-click the application or library project and click **Properties**. Depending on the project type, **Nios II Application Properties** or **Nios II Library Properties** tab appears in the list of tabs. Click the appropriate Properties tab to open it.

The **Nios II Application Properties** and **Nios II Library Properties** tabs are nearly identical. These tabs allow you to control the following project properties:

- The name of the target **.elf** file (application project only)
- The library name (library project only)
- A list of symbols to be defined in the makefile
- A list of symbols to be undefined in the makefile
- A list of assembler flags
- Warning level flags
- A list of user flags
- Generation of debug symbols
- Compiler optimization level
- Generation of object dump file (application project only)
- Source file management
- Path to associated BSP (required for application, optional for library)

Configuring BSP Properties

To configure BSP settings and properties, use the Nios II BSP Editor.

For detailed information about the BSP Editor, refer to "Using the BSP Editor" on page 2–12.

Exclude from Build Not Supported

The **Exclude from Build** command is not supported. You must use the **Remove from Nios II Build** and **Add to Nios II Build** commands instead.

This behavior differs from the behavior of the Nios II SBT for Eclipse in version 9.1.

Selecting the Correct Launch Configuration Type

If you try to debug a Nios II software project as a CDT Local C/C++ Application launch configuration type, you see an error message, and the Nios II Debug perspective fails to open. This is expected CDT behavior in the Eclipse platform. Local C/C++ Application is the launch configuration type for a standard CDT project. To invoke the Nios II plugins, you must use a Nios II launch configuration type.

☞ If your project was created with version 10.1 or earlier of the Nios II SBT, you must re-import it to create the Nios II launch configuration correctly.

Target Connection Options

The Nios II launch configurations offer the following Nios II-specific options in the **Target Connection** tab:

- Disable 'Nios II Console' view
- Ignore mismatched system ID

- Ignore mismatched system timestamp
- Download ELF to selected target system
- Start processor
- Reset the selected target system

Renaming Nios II Projects

To rename a project in the Nios II SBT for Eclipse, perform the following steps:

1. Right-click the project and click **Rename**.
2. Type the new project name.
3. Right-click the project and click **Refresh**.

If you neglect to refresh the project, you might see the following error message when you attempt to build it:

```
Resource <original_project_name> is out of sync with the system
```

Running Shell Scripts from the SBT for Eclipse

Many SBT utilities are implemented as shell scripts. You can use Eclipse external tools configurations to run shell scripts. However, you must ensure that the shell environment is set up correctly.

To run shell scripts from the SBT for Eclipse, execute the following steps:

1. Start the Nios II Command Shell, as described in the *Getting Started from the Command Line* chapter of the *Nios II Software Developer's Handbook*.
2. Start the Nios II SBT for Eclipse by typing the following command:

   ```
   eclipse-nios2↵
   ```

   You must start the SBT for Eclipse from the command line in both the Linux and Windows operating systems, to set up the correct shell environment.
3. From the Eclipse Run menu, point to **External Tools**, and click **External Tools Configurations**.
4. Create a new tools configuration, or open an existing tools configuration.

5. On the **Main** tab, set **Location** and **Argument** as shown in Table 2–3.

Table 2–3. Location and Argument to Run Shell Script from Eclipse

Platform	Location	Argument
Windows	`${env_var:QUARTUS_ROOTDIR}\bin\cygwin\bin\sh.exe`	`-c "`*<script name> <script args>*`"`
Linux	`${env_var:SOPC_KIT_NIOS2}/bin/`*<script name>*	*<script args>*

For example, to run the command `elf2hex --help`, set **Location** and **Argument** as shown in Table 2–4.

Table 2–4. Location and Argument to Run elf2hex --help from Eclipse

Platform	Location	Argument
Windows	`${env_var:QUARTUS_ROOTDIR}\bin\cygwin\bin\sh.exe`	`-c "elf2hex --help"`
Linux	`${env_var:SOPC_KIT_NIOS2}/bin/elf2hex`	`--help`

6. On the **Build** tab, ensure that **Build before launch** and its related options are set appropriately.

   By default, a new tools configuration builds all projects in your workspace before executing the command. This might not be the desired behavior.

7. Click **Run**. The command executes in the Nios II Command Shell, and the command output appears in the Eclipse **Console** tab.

Must Use Nios II Build Configuration

Although Eclipse can support multiple build configurations, you must use the Nios II build configuration for Nios II projects.

☞ If your project was created with version 10.1 or earlier of the Nios II SBT, you must re-import it to create the Nios II launch configuration correctly.

CDT Limitations

The features listed in the left column of Table 2–5 are supported by the Eclipse CDT plugins, but are not supported by Nios II plugins. The right column lists alternative features supported by the Nios II plugins.

Table 2–5. Eclipse CDT Features Not Supported by the Nios II Plugins (Part 1 of 3)

Unsupported CDT Feature	Alternative Nios II Feature
New Project Wizard	
C/C++ ■ C Project ■ C++ Project	To create a new project, use one of the following Nios II wizards: ■ Nios II Application ■ Nios II Application and BSP from Template ■ Nios II Board Support Package ■ Nios II Library

Table 2–5. Eclipse CDT Features Not Supported by the Nios II Plugins (Part 2 of 3)

Unsupported CDT Feature	Alternative Nios II Feature
■ Convert to a C/C++ Project	
■ Source Folder	
Build configurations	
■ Right-click project and point to **Build Configurations**	The Nios II plugins only support a single build configuration.
■ **Debugger** tab	
■ **Stop on startup**	This feature is supported only at the top of `main()`.
Exclude from Build (from version 10.0 onwards)	
Right-click source files	Use **Remove from Nios II Build** and **Add to Nios II Build**.
Project Properties	
C/C++ Build	
■ Builder Settings	
■ Makefile generation	By default, the Nios II SBT generates makefiles automatically.
■ Build location	The build location is determined with the **Nios II Application Properties** or **Nios II BSP Properties** dialog box.
■ Behavior	
■ Build on resource save (Auto build)	
■ Build Variables	
■ Discovery Options	
■ Environment	
■ Settings	
■ Tool Chain Editor	
■ Current builder	
■ Used tools	To change the toolchain, use the **Current tool chain** option
Project Properties, continued	
C/C++ General	
■ Enable project specific settings	
■ Documentation tool comments	
■ Documentation	
■ File Types	
■ Indexer	
■ Build configuration for the indexer	The Nios II plugins only support a single build configuration.
■ Language Mappings	
■ Paths and Symbols	Use **Nios II Application Properties** and **Nios II Application Paths**

Table 2–5. Eclipse CDT Features Not Supported by the Nios II Plugins (Part 3 of 3)

Unsupported CDT Feature	Alternative Nios II Feature
Window Preferences	
C/C++	
■ Build scope	
■ Build project configurations	The Nios II plugins only support a single build configuration.
■ Build Variables	
■ Environment	
■ File Types	
■ Indexer	
■ Build configuration for the indexer	The Nios II plugins only support a single build configuration.
■ Language Mappings	
■ New CDT project wizard	

Document Revision History

Table 2–6 shows the revision history for this document.

Table 2–6. Document Revision History

Date	Version	Changes
January 2014	13.1.0	■ Added section on Nios II Hardware v2 beta ■ Updated GCC4 toolchain from 4.1.2 to GCC 4.7.3 ■ Removed “Managing Toolchains in Eclipse” section.
May 2011	11.0.0	■ Introduction of Qsys system integration tool impacts ModelSim flow ■ Launch configuration change requires re-importation of existing projects ■ Using variables to link to external resources ■ The GCC 3 toolchain is an optional feature ■ Minor corrections to Table 2–5 on page 2–54
February 2011	10.1.0	■ Do not mix versions of GCC. ■ How to create and use a library archive file (**.a**). ■ How to create a software package. ■ Describe Eclipse launch groups. ■ Removed “Referenced Documents” section.
July 2010	10.0.0	■ Document how to import and use projects with user-managed source files. ■ Document how to import and use projects with linked resources. ■ Document **Remove from Nios II Build** command. ■ Update BSP Editor documentation. ■ Document **Add Memory Device** command. ■ Document **Enable File Generation** tab.
November 2009	9.1.0	Initial release.

3. Getting Started from the Command Line

NII52014-13.1.0

The Nios® II Software Build Tools (SBT) allows you to construct a wide variety of complex embedded software systems using a command-line interface. From this interface, you can execute Software Built Tools command utilities, and use scripts (or other tools) to combine the command utilities in many useful ways.

This chapter introduces you to project creation with the SBT at the command line.

This chapter includes the following sections:

- "Advantages of Command-Line Software Development"
- "Outline of the Nios II SBT Command-Line Interface"
- "Getting Started in the SBT Command Line"
- "Software Build Tools Scripting Basics" on page 3–7
- "Running make" on page 3–10

Advantages of Command-Line Software Development

The Nios II SBT command line offers the following advantages over the Nios II SBT for Eclipse™:

- You can invoke the command line tools from custom scripts or other tools that you might already use in your development flow.
- On a command line, you can run several Tcl scripts to control the creation of a board support package (BSP).
- You can use command line tools in a bash script to build several projects at once.

The Nios II SBT command-line interface is designed to work in the Nios II Command Shell.

For details about the Nios II Command Shell, refer to "The Nios II Command Shell" on page 3–2.

Outline of the Nios II SBT Command-Line Interface

The Nios II SBT command-line interface consists of:

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

These elements work together in the Nios II Command Shell to create software projects.

Utilities

The Nios II SBT command-line utilities enable you to create software projects. You can call these utilities from 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 SBT resides in the <*Nios II EDS install path*>**/sdk2/bin** directory.

Refer to "Altera-Provided Development Tools" in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook* for a summary of the command-line utilities provided by the Nios II SBT.

Scripts

Nios II SBT scripts implement complex behavior that extends the capabilities provided by the utilities.

Table 3–1 summarizes the scripts provided with the Nios II SBT.

Table 3–1. Nios II SBT Scripts

Command	Summary
nios2-bsp	Creates or updates a BSP
create-this-app *(1)*	Creates a software example and builds it
create-this-bsp *(1)*	Creates a BSP for a specific hardware design example and builds it

Note to Table 3–1:

(1) There are **create-this-app** scripts for each software example and several **create-this-bsp** scripts for each hardware design example. For more details, refer to "Nios II Design Example Scripts" in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Tcl Commands

Tcl commands are a crucial component of the Nios II SBT. Tcl commands allow you to exercise detailed control over BSP generation, as well as to define drivers and software packages.

Tcl Scripts

The SBT provides powerful Tcl scripting capabilities. In a Tcl script, you can query project settings, specify project settings conditionally, and incorporate the software project creation process in a scripted software development flow. The SBT uses Tcl scripting to customize your BSP according to your hardware and the settings you select. You can also write custom Tcl scripts for detailed control over the BSP.

The Nios II Command Shell

The Nios II Command Shell is a bash command-line environment initialized with the correct settings to run Nios II command-line tools. The Command Shell supports the GCC toolchain.

For general information about GCC toolchains, refer to "Altera-Provided Development Tools" in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*

Starting the Nios II Command Shell

To open the Nios II Command Shell, perform the following steps, depending on your environment:

- In the Windows operating system, on the Start menu, point to **Programs** > **Altera** > **Nios II EDS** <*version*>,and click **Nios II** <*version*> **Command Shell**:.
- In the Linux operating system, in a command shell, change directories to <*Nios II EDS install path*>, and type the command `nios2_command_shell.sh`.

Auto-Executing a Command in the Nios II Command Shell

In certain situations, you might need to run a command or a script automatically after the Nios II Command Shell is initialized. When you start the Nios II Command Shell environment, to automatically execute a command perform one of the following steps, depending on your environment:

- In the Windows operating system, execute the following command:

```
"<Nios II EDS install path>/Nios II Command Shell.bat" <command>↵
```

- In the Linux operating system, execute the following command:

```
<Nios II EDS install path>/nios2_command_shell.sh <command>↵
```

For example, in Windows, to run an automated build, you might execute the following command:

```
"<Nios II EDS install path>/Nios II Command Shell.bat" custom_build.sh↵
```

The Nios II Command Shell startup script (`Nios II Command Shell.bat` or `nios2_command_shell.sh`) makes no special assumptions about its initial environment. You can use the Nios II Command Shell with auto-execution from any environment that accepts commands native to your host operating system. For example, in Linux you can use **crontab** to schedule a job to run in the Nios II Command Shell at a later time.

Getting Started in the SBT Command Line

Using the Nios II SBT on the command line is the best way to learn about it. 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 provide more of the underlying details, allowing you to take more control of the process. The goal of this chapter is to show you that the basic process is simple and straightforward.

The Nios II SBT includes a number of scripts that demonstrate how to combine command utilities to obtain the results you need. This tutorial uses a **create-this-app** script as an example.

What You Need

To complete this tutorial, you must have the following:

- Altera Quartus® II development software, version 8.0 or later. The software must be installed on a Windows or Linux computer that meets the Quartus II minimum requirements.
- The Altera Nios II Embedded Design Suite (EDS), version 8.0 or later.
- An Altera development board.
- A download cable such as the Altera USB-Blaster™ cable.

You run the Nios II SBT commands from the Nios II Command Shell.

For details about the Nios II Command Shell, refer to "The Nios II Command Shell".

Creating hello_world for an Altera Development Board

In this section you create a simple "Hello World" project. To create and build the `hello_world` example for an Altera development board, perform the following steps:

1. Start the Nios II Command Shell, as described in "The Nios II Command Shell".
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 by typing the following command:

   ```
   cd <projects>↵
   ```

4. Locate a Nios II hardware example for your Altera development board. For example, if you have a Stratix® IV GX FPGA Development Kit, you might select <*Nios II EDS install path*>**/examples/verilog/niosII_stratixIV_4sgx230/triple_speed_ethernet_design**.
5. Copy the hardware example to your <*projects*> working directory, using a command such as the following:

```
cp -R /altera/100/nios2eds/examples/verilog/niosII_stratixIV_4sgx230/triple_speed_ethernet_design .↵
```

6. Ensure that the working directory and all subdirectories are writable by typing the following command:

   ```
   chmod -R +w .r
   ```

7. The <*projects*> directory contains a subdirectory named **software_examples/app/hello_world**. The following steps refer to this directory as <*application*>.
8. Change to the <*application*> directory by typing the following command:

   ```
   cd <application>↵
   ```

9. Type the following command to create and build the application:

   ```
   ./create-this-app↵
   ```

The **create-this-app** script copies the application source code to the <*application*> directory, runs **nios2-app-generate-makefile** to create a makefile (named **Makefile**), and then runs **make** to create an Executable and Linking Format File (**.elf**). 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 an Altera Development Board

To run the `hello_world` example on an Altera development board, perform the following steps:

1. Start the Nios II Command Shell.
2. Download the SRAM Object File (**.sof**) for the Quartus II project to the Altera development board. This step configures the FPGA on the development board with your project's associated SOPC Builder system.

   The .**sof** file resides in <*projects*>, along with your Quartus II Project File (**.qpf)**. You download it by typing the following commands:

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

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

   The **nios2-configure-sof** utility runs the Quartus II Programmer to download the .**sof** file. You can also run the `quartus_pgm` command directly.

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

3. Start 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 Altera development board through the JTAG UART port by typing the following command:

   ```
   nios2-terminal↵
   ```

5. Return to the original command shell, and ensure that <*projects*>**/ software_examples/app/hello_world** is the current working directory.
6. Download and run the `hello_world` executable program as follows:

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

   The following output appears in the second command shell:

   ```
   Hello from Nios II!
   ```

Debugging hello_world

An integrated development environment is the most powerful environment for debugging a software project. You debug a command-line project by importing it to the Nios II SBT for Eclipse. After you import the project, Eclipse uses your makefiles to build the project. This two-step process combines the advantages of the SBT command line development flow with the convenience of a 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 Nios II SBT for Eclipse.

2. On the File menu, click **Import**. The **Import** dialog box appears.
3. Expand the **Nios II Project** folder, and select **Import Nios II project**.
4. Click **Next**. The **File Import** wizard appears.
5. Click **Browse** and navigate to the <*application*> directory, containing the **hello_world** application project.
6. Click **OK**. The wizard fills in the project path.
7. Type the project name `hello_world` in the **Project name** box.
8. Click **Finish**. The wizard imports the application project.

☞ If you want to view the BSP source files while debugging, you also need to import the BSP project into the Nios II SBT for Eclipse.

For a description of importing BSPs into Eclipse, refer to “Importing a Command-Line Project” in the *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook.*

Download Executable Code and Start the Debugger

To debug the software project, perform the following steps:

1. Right-click the `hello_world` project, point to **Debug As**, and click **Nios II Hardware**.
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
   - Optionally verifies that the expected SOPC Builder system is configured in the FPGA.
   - Downloads the **.elf** file to memory on the target board.

   a. 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 Eclipse, you can also pause, stop, and single-step the program, set breakpoints, examine variables, and perform many other common debugging tasks.

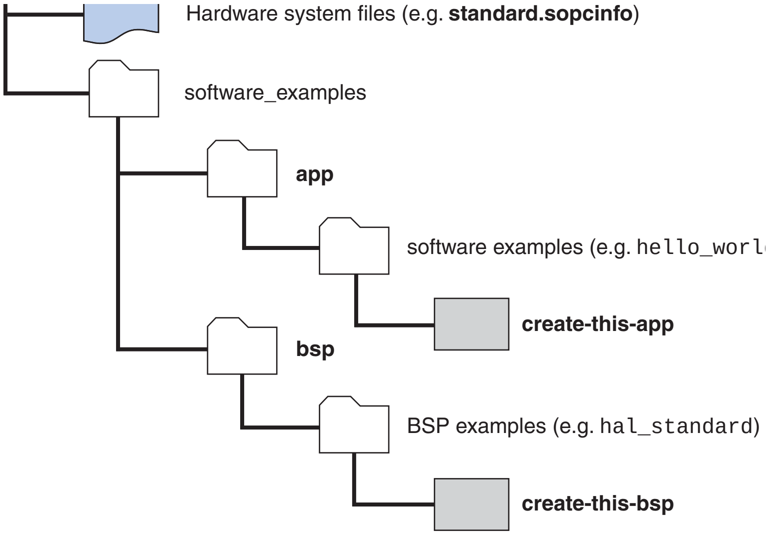
For more detailed information about debugging projects in the Nios II SBT for Eclipse, refer to “Importing a Command-Line Project” and “Getting Started with Eclipse” in the *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook*.

Software Build Tools Scripting Basics

This section provides an example to teach you how you can create a software application using a command line script.

In this section, assume that 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, assume that you have organized the hardware design files and the software source files as shown in Figure 3–1.

Figure 3–1. Simple Software Project Directory Structure


Hardware system files (e.g. **standard.sopcinfo**)
software_examples
app
software examples (e.g. hello_worl
create-this-app
bsp
BSP examples (e.g. hal_standard)
create-this-bsp

Creating a BSP with a Script

One easy method for creating a BSP is to use the **nios2-bsp** script. The script in Example 3–1 creates a BSP and then builds it.

Example 3–1. nios2-bsp

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

Table 3–2 shows the meaning of each argument to the **nios2-bsp** script in Example 3–1.

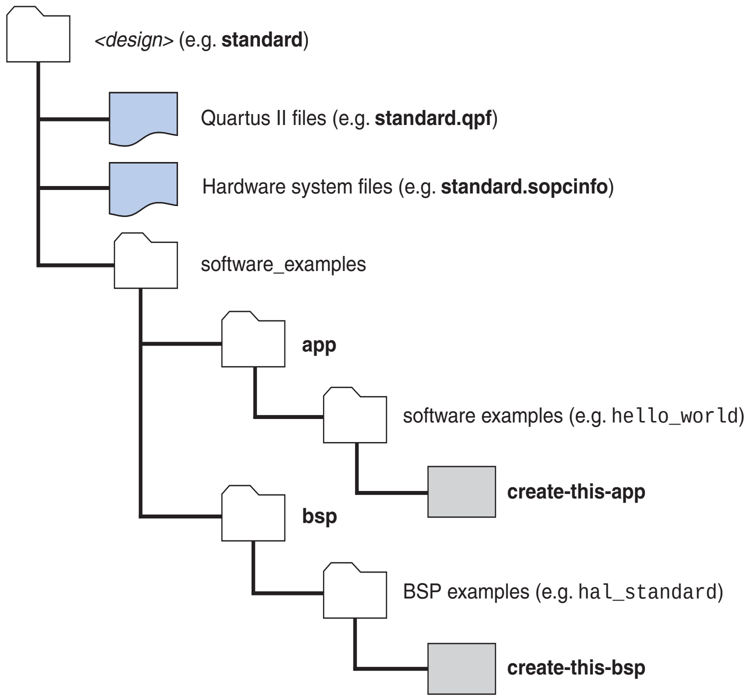
For additional information about the **nios2-bsp** command, refer to “Nios II Software Build Tools Utilities” in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer’s Handbook*.

Table 3–2. nios2-bsp Example Arguments

Argument	Purpose	Further Information
`ucosii`	Sets the operating system to MicroC/OS-II	“Settings Managed by the Software Build Tools” in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer’s Handbook*
`.`	Specifies the directory in which the BSP is to be created	—
`../SOPC/`	Points to the location of the hardware project	—
`--cmd enable_sw_package altera_iniche`	Adds the NicheStack TCP/IP stack software package to the BSP	“Settings Managed by the Software Build Tools” and “Software Build Tools Tcl Commands” in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer’s Handbook*
`--set altera_iniche.iniche_default_if lan91c111`	Specifies the default hardware interface for the NicheStack TCP/IP Stack - Nios II Edition	“Settings Managed by the Software Build Tools” in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer’s Handbook*

Figure 3–2 shows the flow to create a BSP using the **nios2-bsp** script. The **nios2-bsp** script uses the **.sopcinfo** 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 3–2. nios2-bsp Command Flow


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

Creating an Application Project with a Script

You use **nios2-app-generate-makefile** to create application projects. The script in Example 3–2 creates an application project and builds it.

Example 3–2. nios2-app-generate-makefile

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

Table 3–3 shows the meaning of each argument in Example 3–2.

Table 3–3. nios2-app-generate-makefile Example Arguments

Argument	Purpose
`--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

For further information about each command argument in Table 3–3, refer to "Nios II Software Build Tools Utilities" in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook.* For more detail about the software example scripts, refer to "Nios II Design Example Scripts" in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Running make

nios2-bsp places all BSP files in the BSP directory, specified on the command line with argument `--bsp-dir`. 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 linked with your application.

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↵
```

You can modify an application or user library makefile with the **nios2-lib-update-makefile** and **nios2-app-update-makefile** utilities. With these utilities, you can execute the following tasks:

- Add source files to a project
- Remove source files from a project
- Add compiler options to a project's make rules
- Modify or remove compiler options in a project's make rules

Creating Memory Initialization Files

To create memory initialization files for a Nios II system, you can use the Nios II Command Shell. Change to the software application folder, and type:

```
make mem_init_generate↵
```

This command creates the memory initialization and simulation files for all memory devices. It also generates a Quartus II IP File (**.qip**). The **.qip** file tells the Quartus II software where to find the initialization files. Add the **.qip** file to your Quartus II project.

Document Revision History

Table 3–4 shows the revision history for this document.

Table 3–4. Document Revision History

Date	Version	Changes
January 2014	13.1.0	■ Updated GCC4 toolchain from 4.1.2 to GCC 4.7.3. ■ Removed references to the Nios II IDE. ■ Removed references to GCC 3. ■ Removed the “Using the Nios II C2H Compiler” section.
May 2011	11.0.0	■ Can auto-execute a Command in the Nios II Command Shell ■ The GCC 3 toolchain is an optional feature
February 2011	10.1.0	■ Do not mix versions of GCC. ■ Removed “Referenced Documents” section.
July 2010	10.0.0	■ Introduction of GCC 4. ■ Discuss usage of GCC 3 and GCC 4 command shells.
November 2009	9.1.0	■ Repurpose and retitle this chapter as an introduction to Nios II Software Build Tools command-line usage. ■ Information about the BSP Editor moved to the *Getting Started with the Graphical User Interface* chapter.
March 2009	9.0.0	■ Describe BSP Editor. ■ Reorganize and update information and terminology to clarify role of Nios II Software Build Tools. ■ Correct minor typographical errors.
May 2008	8.1.0	Maintenance release.
October 2007	7.2.0	Repurpose this chapter as a “getting started” guide. Move descriptive and reference material to separate chapters.
May 2007	7.1.0	Initial Release.

4. Nios II Software Build Tools

NII52015-13.1.0

This chapter describes the Nios® II Software Build Tools (SBT), a set of utilities and scripts that creates and builds embedded C/C++ application projects, user library projects, and board support packages (BSPs). The Nios II SBT supports a repeatable, scriptable, and archivable process for creating your software product.

You can invoke the Nios II SBT through either of the following user interfaces:

- The Eclipse™ GUI
- The Nios II Command Shell

The purpose of this chapter is to make you familiar with the internal functionality of the Nios II SBT, independent of the user interface employed.

☞ Before reading this chapter, consider getting an introduction to the Nios II SBT by first reading one of the following chapters:

- *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook*
- *Getting Started from the Command Line* chapter of the *Nios II Software Developer's Handbook*

This chapter contains the following sections:

This chapter assumes you are familiar with the following topics:

- The GNU **make** utility. Altera recommends you use version 3.80 or later. On the Windows platform, GNU **make** version 3.80 is provided with the Nios II EDS.

  You can obtain general information about GNU **make** from the Free Software Foundation, Inc. (www.gnu.org).

- 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.

Road Map for the SBT

Before you start using the Nios II SBT, it is important to understand its scope. This section helps you understand their purpose, what they include, and what each tool does. Understanding these points helps you determine how each tool fits in with your development process, what parts of the tools you need, and what features you can disregard for now.

What the Build Tools Create

The purpose of the build tools is to create and build Nios II software projects. A Nios II project is a makefile with associated source files.

The SBT creates the following types of 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 UARTs and other I/O devices. A BSP provides a software runtime environment customized for one processor in a hardware system. A BSP optionally also includes the operating system, and other basic system software packages such as communications protocol stacks.
- User library—A library implementing a collection of reusable functions, such as graphics algorithms.

Comparing the Command Line with Eclipse

Aside from the Eclipse GUI, there are very few differences between the SBT command line and the Nios II SBT for Eclipse. Table 4–1 lists the differences.

Table 4–1. Differences between Nios II SBT for Eclipse and the Command Line

Feature	Eclipse	Command Line
Project source file management	Specify sources automatically, e.g. by dragging and dropping into project	Specify sources manually using command arguments
Debugging	Yes	Import project to Eclipse environment
Integrates with custom shell scripts and tool flows	No	Yes

The Nios II SBT for Eclipse provides access to a large, useful subset of SBT functionality. Any project you create in Eclipse can also be created using the SBT from the command line or in a script. Create your software project using the interface that is most convenient for you. Later, it is easy to perform additional project tasks in the other interface if you find it advantageous to do so.

Makefiles

Makefiles are a key element of Nios II C/C++ projects. The Nios II SBT includes powerful tools to create makefiles. An understanding of how these tools work can help you make the most optimal use of them.

The Nios II SBT creates two kinds of makefiles:

- Application or user library makefile—A simple makefile that builds the application or user library with user-provided source files
- BSP makefile—A more complex makefile, generated to conform to user-specified settings and the requirements of the target hardware system

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

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

The generated makefiles have a straightforward structure, and each makefile has in-depth comments explaining how it works. Altera recommends that you study these makefiles for further information about how they work. 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 user library has one makefile, located in the application or user library directory.

Modifying Makefiles

It is not necessary to edit makefiles by hand. The Nios II SBT for Eclipse offers GUI tools for makefile management.

For further information, refer to the *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook*.

On the command line, the project type determines the correct utility or utilities to update your makefile, as shown in Table 4–2.

Table 4–2. Command-Line Utilities for Updating Makefiles

Project Type	Utilities
Application	**nios2-app-update-makefile**
Library	**nios2-lib-update-makefile**
BSP *(1)*	**nios2-bsp-update-settings** **nios2-bsp-generate-files**

Note to Table 4–2:

(1) For details about updating BSP makefiles, refer to "Updating Your BSP" on page 4–32.

After making changes to a makefile, run **make clean** before rebuilding your project. If you are using the Nios II SBT for Eclipse, this happens automatically.

Makefile Targets

Table 4–3 shows the application makefile targets. Altera recommends that you study the generated makefiles for further details about these targets.

Table 4–3. 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`	Performs a clean build of the application. Deletes all application-related generated files. Leaves associated BSP and libraries alone.
`clean_all`	Performs a clean build of the application, and associated BSP and libraries (if any).
`clean_bsp`	Performs a clean build of the BSP.
`clean_libs`	Performs a clean build of 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.

Note to Table 4–3:

(1) You can use the `download-elf` makefile target if the host system is connected to a single USB-Blaster™ download cable. If you have more than one download cable, you must download your executable with a separate command. Set up a run configuration in the Nios II SBT for Eclipse, or use **nios2-download**, with the `--cable` option to specify the download cable.

Nios II Embedded Software Projects

The Nios II SBT supports the following kinds of software projects:

- C/C++ application projects
- C/C++ user library projects
- BSP projects

This section discusses each type of project in detail.

Applications and Libraries

The Nios II SBT has nearly identical support for C/C++ applications and libraries. The support for applications and libraries is very simple. For each case, the SBT generates a private makefile (named **Makefile**). The private makefile is used to build the application or user library.

The private makefile builds one of two types of files:

- A **.elf** file—For an application
- A library archive file (**.a**)—For a user library

For a user library, the SBT also generates a public makefile, called **public.mk**. The public makefile is included in the private makefile for any application (or other user library) that uses the user library.

When you create a makefile for an application or user library, you provide the SBT with a list of source files and a reference to a BSP directory. The BSP directory is mandatory for applications and optional for libraries.

The Nios II SBT examines the extension of each source file to determine the programming language. Table 4–4 shows the supported programming languages with the corresponding file extensions.

Table 4–4. Supported Source File Types

Programming Language	File Extensions *(1)*
C	`.c`
C++	`.cpp, .cxx, .cc`
Nios II assembly language; sources are built directly by the Nios II assembler without preprocessing	`.s`
Nios II assembly language; sources are preprocessed by the Nios II C preprocessor, allowing you to include header files	`.S`

Note to Table 4–4:

(1) All file extensions are case-sensitive.

Board Support Packages

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 a hardware system. The BSP isolates your application from system-specific details such as the memory map, available devices, and processor configuration.

A BSP includes a **.a** 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 SBT supports two types of BSPs: Altera® Hardware Abstraction Layer (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

The Nios II SBT creates your BSP for you. The tools provide a great deal of power and flexibility, enabling you to control details of your BSP implementation while maintaining compatibility with a hardware system that might change.

By default, the tools generate a basic BSP for a Nios II system. If you require more detailed control over the characteristics of your BSP, the Nios II SBT provides that control, as described in the remaining sections of this chapter.

Parts of a Nios II BSP

Hardware Abstraction Layer

The HAL provides a single-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 interfaces to HAL device drivers, which access peripheral registers directly, abstracting hardware details from the software application. This abstraction minimizes or eliminates the need to access hardware registers directly to connect to and control peripherals.

For complete details about the HAL, refer to 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 hardware 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 hardware 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.

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

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.

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 in the **UCOSII** directory. The **UCOSII** directory is contained in 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.

Software Build Process

To create a software project with the Nios II SBT, you perform several high-level steps:

1. Obtain the hardware design on which the software is to run. When you are learning about the build tools, this might be a Nios II design example. When you are developing your own design, it is probably a design developed by someone in your organization. Either way, you need to have the SOPC Information File (**.sopcinfo**).
2. Decide what features the BSP requires. For example, does it need to support an 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 the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

3. Define a BSP. Use the Nios II SBT 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, refer to "Board Support Packages" on page 4–5.
4. Create a BSP makefile using the Nios II build tools.
5. Optionally create a user library. If you need to include a custom software user library, you collect the user library source files in a single directory, and create a user library makefile. The Nios II build tools can create a makefile for you. 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, refer to "Applications and Libraries" on page 4–5.
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, refer to "Applications and Libraries" on page 4–5.
7. Create an application makefile. The easiest approach is to let the Nios II build tools create the makefile for you. 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, refer to "Makefiles" on page 4–3.

Common BSP Tasks

The Nios II SBT creates a BSP for you with useful default settings. However, for many tasks you must manipulate the BSP explicitly. This section describes the following common BSP tasks, and how you carry them out.

- "Using Version Control" on page 4–9
- "Copying, Moving, or Renaming a BSP" on page 4–10
- "Handing Off a BSP" on page 4–10
- "Creating Memory Initialization Files" on page 4–11
- "Modifying Linker Memory Regions" on page 4–11
- "Creating a Custom Linker Section" on page 4–12
- "Changing the Default Linker Memory Region" on page 4–16
- "Changing a Linker Section Mapping" on page 4–16
- "Creating a BSP for an Altera Development Board" on page 4–17
- "Querying Settings" on page 4–18
- "Managing Device Drivers" on page 4–18
- "Creating a Custom Version of newlib" on page 4–18
- "Controlling the stdio Device" on page 4–19
- "Configuring Optimization and Debugger Options" on page 4–19

Although this section describes tasks in terms of the SBT command line flow, you can also carry out most of these tasks with the Nios II SBT for Eclipse, described in the *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook*.

Adding the Nios II SBT to Your Tool Flow

A common reason for using the SBT 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 in which you can incorporate the build tools in an existing tool chain.

Using Version Control

One common tool flow requirement 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 which files to add to your version control database. With the Nios II SBT, the version control requirements depend on what you are trying to do and 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.

Refer to "Revising Your BSP" on page 4–30 for a discussion of BSP regeneration with Nios II EDS updates.

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 recreate 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 SBT places 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 a **.sopcinfo** file, you must usually store the **.sopcinfo** file in source control along with the BSP. The BSP settings file stores the **.sopcinfo** file path as a relative or absolute path, according to the definition on the **nios2-bsp** or **nios2-bsp-create-settings** command line. You must take the path into account when retrieving the BSP and the **.sopcinfo** file from source control.

Copying, Moving, or Renaming a BSP

BSP makefiles have only relative path references to project source files. Therefore 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 must ensure that the **.sopcinfo** file is still accessible from the new location of the BSP. This **.sopcinfo** file path is stored in the BSP settings file.

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

For information about **.d** files, refer to the GNU `make` documentation, available from the Free Software Foundation, Inc. (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, you must manually revise any references to the BSP name or location in application or user library makefiles.

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 simply run `make` to build the BSP.

Linking and Locating

When autogenerating a HAL BSP, the SBT makes some reasonable assumptions about how you want to use memory, as described in “Specifying the Default Memory Map” on page 4–38. 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 specify which memory device contains your interrupt service routines (ISRs).

This section describes several common scenarios in which the SBT allows 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 4–5 shows the **mem_init.mk** targets. Although the application makefile provides all these targets, it does not build any of them by default. The SBT creates the memory initialization files in the application directory (under a directory named **mem_init**). The SBT optionally copies them to your Quartus II project directory and HDL simulation directory, as described in Table 4–5.

☞ The Nios II SBT does not generate a definition of `QUARTUS_PROJECT_DIR` in your application makefile. If you have an on-chip RAM, and require that a compiled software image be inserted in your SRAM Object File (**.sof**) at Quartus II compilation, you must manually specify the value of `QUARTUS_PROJECT_DIR` in your application makefile. You must define `QUARTUS_PROJECT_DIR` before the **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)
```

Table 4–5. 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 to your Quartus II project directory named `$(QUARTUS_PROJECT_DIR)`. If the `SOPC_NAME` variable is defined, **mem_init.mk** copies memory initialization files to your HDL simulation directory named `$(QUARTUS_PROJECT_DIR)/$(SOPC_NAME)_sim`.
`mem_init_generate`	Generates all memory initialization files in the application **mem_init** directory. This target also generates a Quartus II IP File (**.qip**). The **.qip** file tells the Quartus II software where to find the initialization files.
`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.

Modifying Linker Memory Regions

If the linker memory regions that are created by default do not meet your needs, BSP Tcl commands 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 the top 1024 bytes of `onchip_ram` to create a new region named `onchip_special`.

For an explanation of each Tcl command used in this example, refer to 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↵
```

Take care that one of the new memory regions has the same name as the original memory region.

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 that you use the `--script` approach when updating your BSP. This approach allows the default Tcl script to update memory regions if memories are added, removed, renamed, or resized. 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.

For details about using the `--script` argument, refer to "Calling a Custom BSP Tcl Script" on page 4–27.

Creating a Custom Linker Section

The Nios II SBT provides a Tcl command, `add_section_mapping`, to create a linker section.

Table 4–6 lists the default section names. The default Tcl script creates these default sections for you using the add_section_mapping Tcl command.

Table 4–6. Nios II Default Section Names

.entry
.exceptions
.text
.rodata
.rwdata
.bss
.heap
.stack

Creating a Linker Section for an Existing Region

To create your own section named special_section that is mapped to the linker region named onchip_special, use the following command to run **nios2-bsp**:

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

When the **nios2-bsp-generate-files** utility (called 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 4–6.

Dividing a Linker Region to Create a New Region and Section

Example 4–2 creates a section named .isrs in the tightly_coupled_instruction_memory on-chip memory. This example works with any hardware design containing an on-chip memory named tightly_coupled_instruction_memory connected to a Nios II instruction master.

Example 4–2. Tcl Script to Create New Region and Section

```
# 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 Tcl script in Example 4–2 script splits off 1 KB of RAM from the region named `tightly_coupled_instruction_memory`, gives it the name `isrs_region`, and then calls `add_section_mapping` to add the `.isrs` section to `isrs_region`.

To use such a Tcl script, you would execute the following steps:

1. Create the Tcl script as shown in Example 4–2.
2. Edit your **create-this-bsp** script, and add the following argument to the **nios2-bsp** command line:

   ```
   --script <script name>.tcl
   ```

3. In the BSP project, edit **timer_interrupt_latency.h**. In the `timer_interrupt_latency_irq()` function, change the `.section` directive from `.exceptions` to `.isrs`.
4. Rebuild the application by running `make`.

After make completes successfully, you can examine the object dump file, <*project name*>**.objdump**, illustrated in Example 4–3. The object dump file shows that the new .isrs section is located in the tightly coupled instruction memory. This object dump file excerpt shows a hardware design with an on-chip memory whose base address is 0x04000000.

Example 4–3. Excerpts from Object Dump File

```
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
 00013778  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

    .
    .
    .
```

If you examine the linker script file, **linker.x**, illustrated in Example 4–4, you can see that **linker.x** places the new region isrs_region in tightly-coupled instruction memory, adjacent to the tightly_coupled_instruction_memory region.

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 processor as the default region. All default memory sections specified in Table 4–6 on page 4–13 are mapped to this default region. You can pass in a command-line option to the default Tcl script to override this default mapping. 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 add_section_mapping 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 resized.

Other BSP Tasks

This section covers some other common situations in which the SBT is useful.

Creating a BSP for an Altera 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, perform the following steps:

1. Start the Nios II Command Shell.

   For details about the Nios II Command Shell, refer to the *Getting Started from the Command Line* chapter of the *Nios II Software Developer's Handbook*.

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 Altera development board. For example, if you have a Stratix® IV development board, you might select <*Nios II EDS install path*>**/examples/verilog/niosII_stratixIV_4sgx230/triple_speed_ethernet_design**.
5. Copy the hardware example to your working directory, using a command such as the following:

   ```
   cp -R /altera/100/nios2eds/examples/verilog\
       /niosII_stratixIV_4sgx230/triple_speed_ethernet_design .↵
   ```

6. Ensure that the working directory and all subdirectories are writable by typing the following command:

   ```
   chmod -R +w .↵
   ```

   The <*projects*> directory contains a subdirectory named **software_examples/bsp**. The **bsp** directory contains several BSP example directories, such as **hal_default**. Select the directory containing an appropriate BSP, and make it the current working directory.

   For a description of the example BSPs, refer to "Nios II Design Example Scripts" in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

7. Create and build the BSP with the **create-this-bsp** script by typing the following command:

   ```
   ./create-this-bsp↵
   ```

Now 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 is 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** utility. To maintain 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. This command sends the values of the settings you requested to `stdout`. Just capture the output of `stdout` in 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 about the **nios2-bsp-query-settings** command-line options, refer to the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Managing Device Drivers

The Nios II SBT creates an **alt_sys_init.c** file. By default, the SBT assumes that if a device is connected to the Nios II processor, and a driver is available, the BSP must include the most recent version of the driver. However, you might want to use a different version of the driver, or you might not want a driver at all (for example, if your application accesses the device directly).

The SBT 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. The **nios2-bsp-create-settings** command also supports the `--cmd` option. Alternatively, you can put the `set_driver` command in a Tcl script and pass the 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 precompiled libraries. These libraries include the newlib libraries (**libc.a** and **libm.a**). The Nios II SBT allows 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**, type the following command:

```
nios2-bsp hal my_bsp --set hal.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 runs the **configure** utility. The **configure** utility creates a makefile in the build directory, which compiles the newlib source. The newlib library files are copied to the BSP directory named newlib. The newlib source files are not copied to the BSP.

☞ The Nios II SBT recompiles newlib whenever you introduce new compiler flags. For example, if you use compiler flags to add floating point math hardware support, newlib is recompiled to use the hardware. Recompiling newlib might take several minutes.

Controlling the stdio Device

The build tools offer several ways to control the details of your `stdio` device configuration, such as the following:

- 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 must provide the original command-line options again to prevent the default Tcl script from choosing 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.

Configuring Optimization and Debugger Options

By default, the Nios II SBT creates your project with the correct compiler options for debugging environments. These compiler options turn off code optimization, and generate a symbol table for the debugger.

You can control the optimization and debug level through the project makefile, which determines the compiler options. Example 4–5 illustrates how a typical application makefile specifies the compiler options.

Example 4–5. Default Application Makefile Settings

```
APP_CFLAGS_OPTIMIZATION := -O0
APP_CFLAGS_DEBUG_LEVEL  := -g
```

When your project is fully debugged and ready for release, you might want to enable optimization and omit the symbol table, to achieve faster, smaller executable code. To enable optimization and turn off the symbol table, edit the application makefile to contain the symbol definitions shown in Example 4–6. The absence of a value on the right hand side of the `APP_CFLAGS_DEBUG_LEVEL` definition causes the compiler to omit generating a symbol table.

Example 4–6. Application Makefile Settings with Optimization

```
APP_CFLAGS_OPTIMIZATION := -O3
APP_CFLAGS_DEBUG_LEVEL  :=
```

☞ When you change compiler options in a makefile, before building the project, run `make clean` to ensure that all sources are recompiled with the correct flags. For further information about makefile editing and `make clean`, refer to “Applications and Libraries” on page 4–5.

You individually specify the optimization and debug level for the application and BSP projects, and any user library projects you might be using. You use the BSP settings `hal.make.bsp_cflags_debug` and `hal.make.bsp_cflags_optimization` to specify the optimization and debug level in a BSP, as shown in Example 4–7.

Example 4–7. Configuring a BSP for Debugging

```
nios2-bsp hal my_bsp --set hal.make.bsp_cflags_debug -g \
  --set hal.make.bsp_cflags_optimization -O0↵
```

Alternatively, you can manipulate the BSP settings with a Tcl script.

You can easily copy an existing BSP and modify it to create a different build configuration. For details, refer to “Copying, Moving, or Renaming a BSP” on page 4–10.

To change the optimization and debug level for a user library, use the same procedure as for an application.

☞ Normally you must set the optimization and debug levels the same for the application, the BSP, and all user libraries in a software project. If you mix settings, you cannot debug those components which do not have debug settings. For example, if you compile your BSP with the `-O0` flag and without the `-g` flag, you cannot step into the newlib `printf()` function.

Details of BSP Creation

BSP creation is the same in the Nios II SBT for Eclipse as at the command line. Figure 4–1 shows how the SBT creates a BSP. The **nios2-bsp-create-settings** utility creates a new BSP settings file. For detailed information about BSP settings files, refer to “BSP Settings File Creation” on page 4–22.

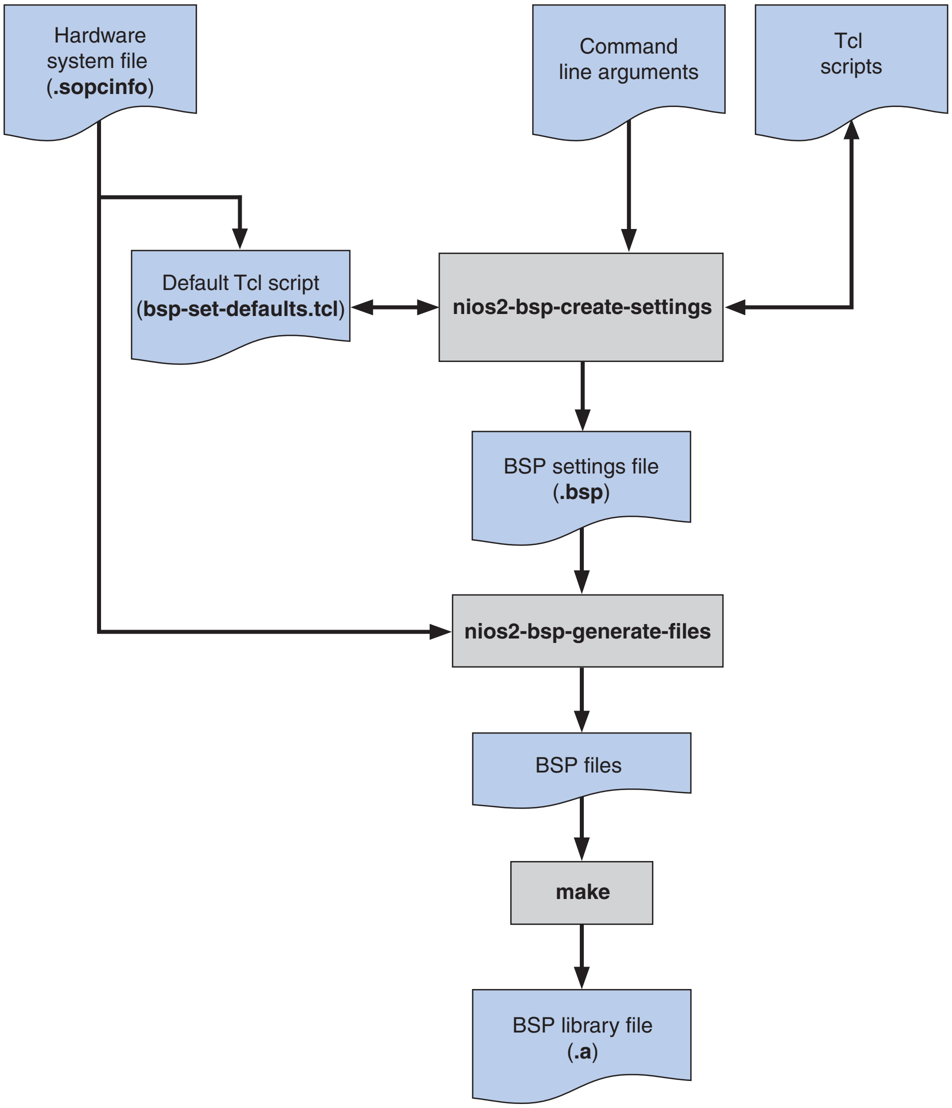
nios2-bsp-generate-files creates the BSP files. The **nios2-bsp-generate-files** utility places all source files in your BSP directory. It copies some files from the Nios II EDS installation directory. Others, such as **system.h** and **Makefile**, it generates dynamically.

The SBT manages copied files slightly differently from generated files. If a copied file (such as a HAL source file) already exists, the tools check the file timestamp against the timestamp of the file in the Nios II EDS installation. The tools do not replace the BSP file unless it differs from the distribution file. The tools normally overwrite generated files, such as the BSP **Makefile**, **system.h**, and **linker.x**, unless you have disabled generation of the individual file with the `set_ignore_file` Tcl command or the **Enable File Generation** tab in the BSP Editor. A comment at the top of each generated file warns you not to edit it.

For information about `set_ignore_file` and other SBT Tcl commands, refer to "Software Build Tools Tcl Commands" in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Avoid modifying BSP files. Use BSP settings, or custom device drivers or software packages, to customize your BSP.

Figure 4–1. Nios II SBT BSP Creation



Nothing prevents you from modifying a BSP generated file. However, after you do so, it becomes difficult to update your BSP to match changes in your hardware system. If you regenerate your BSP, your previous changes to the generated file are destroyed.

For information about regenerating your BSP, refer to "Revising Your BSP" on page 4–30.

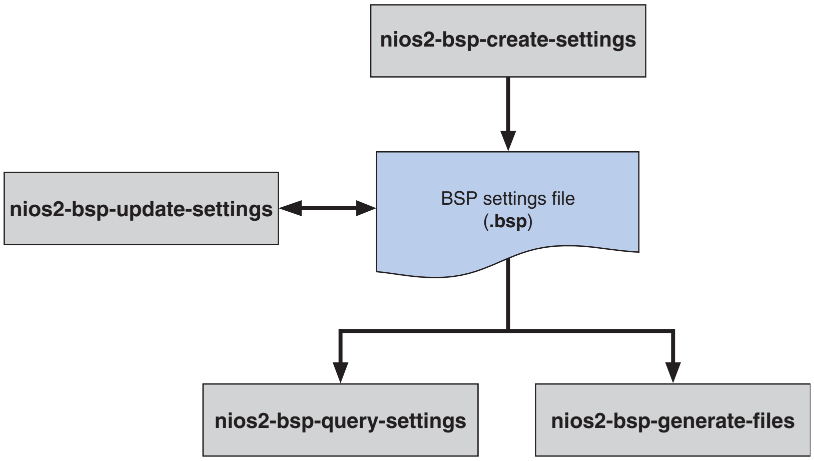
BSP Settings File Creation

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 Nios II SBT writes the value of all settings to the settings file.

Figure 4–1 on page 4–21 shows that the default Tcl script and **nios2-bsp-generate-files** both use the **.sopcinfo** file. The BSP settings file does not need to duplicate system information (such as base addresses of devices), because the **nios2-bsp-generate-files** utility has access to the **.sopcinfo** file.

Figure 4–2 shows how the Nios II SBT interacts with the BSP settings file. The **nios2-bsp-create-settings** utility creates a new BSP settings file. The **nios2-bsp-update-settings** utility updates an existing BSP settings file. The **nios2-bsp-query-settings** utility reports the setting values in an existing BSP settings file. The **nios2-bsp-generate-files** utility generates a BSP from the BSP settings file.

Figure 4–2. BSP Settings File and BSP Utilities



Generated and Copied Files

To understand how to build and modify Nios II C/C++ 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 to your BSP directory when you create your BSP. It does not replace the BSP file unless it differs from the distribution file.

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

You can disable generation of any BSP file in the BSP Editor, or on the command line with the `set_ignore_file` Tcl command. Otherwise, if you modify a BSP file, it is destroyed when you regenerate the BSP.

HAL BSP Files and Folders

The Nios II SBT creates the HAL BSP directory in the location you specify. Figure 4–3 shows a BSP directory after the SBT creates a BSP and generates BSP files. The SBT places generated files in the top-level BSP directory, and copied files in the **HAL** and **drivers** directories.

Figure 4–3. HAL BSP After Generating Files

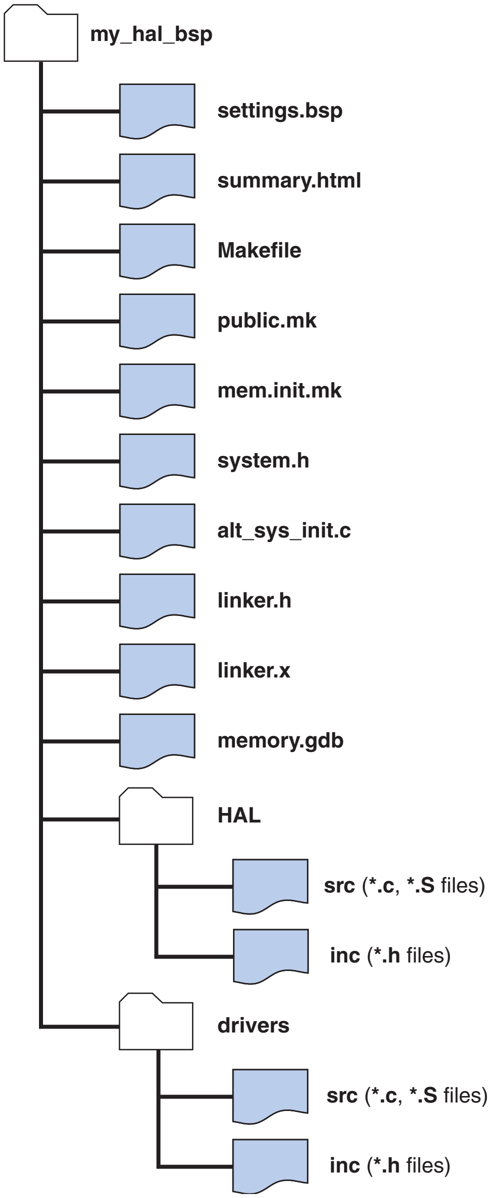


Table 4–7 details all the generated BSP files shown in Figure 4–3.

Table 4–7. Generated BSP Files

File Name	Function
settings.bsp	Contains all BSP settings. This file is coded in XML. On the command line, **settings.bsp** is created by the **nios2-bsp-create-settings** command, and optionally updated by the **nios2-bsp-update-settings** command. 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**. The Nios II SBT for Eclipse provides equivalent functionality.
summary.html	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, the SBT updates the **summary.html** file the next time you regenerate the BSP.
Makefile	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	A 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**.
mem_init.mk	A makefile fragment that defines targets and rules to convert an application executable file to 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, refer to any application makefile generated when you run the SBT. For more information, refer to "Creating Memory Initialization Files" on page 4–11.
alt_sys_init.c	Used to initialize device driver instances and software packages. *(1)*
system.h	Contains the C declarations describing the BSP memory map and other system information needed by software applications. *(1)*
linker.h	Contains information about the linker memory layout. **system.h** includes the **linker.h** file.
linker.x	Contains a linker script for the GNU linker.
memory.gdb	Contains memory region declarations for the GNU debugger.
obj Directory	Contains the object code files for all source files in the BSP. The hierarchy of the BSP source files is preserved in the **obj** directory.
libhal_bsp.a Library	Contains the HAL BSP library. All object files are combined in the library file. The HAL BSP library file is always named **libhal_bsp.a**.

Note to Table 4–7:

(1) For further details about this file, refer to the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

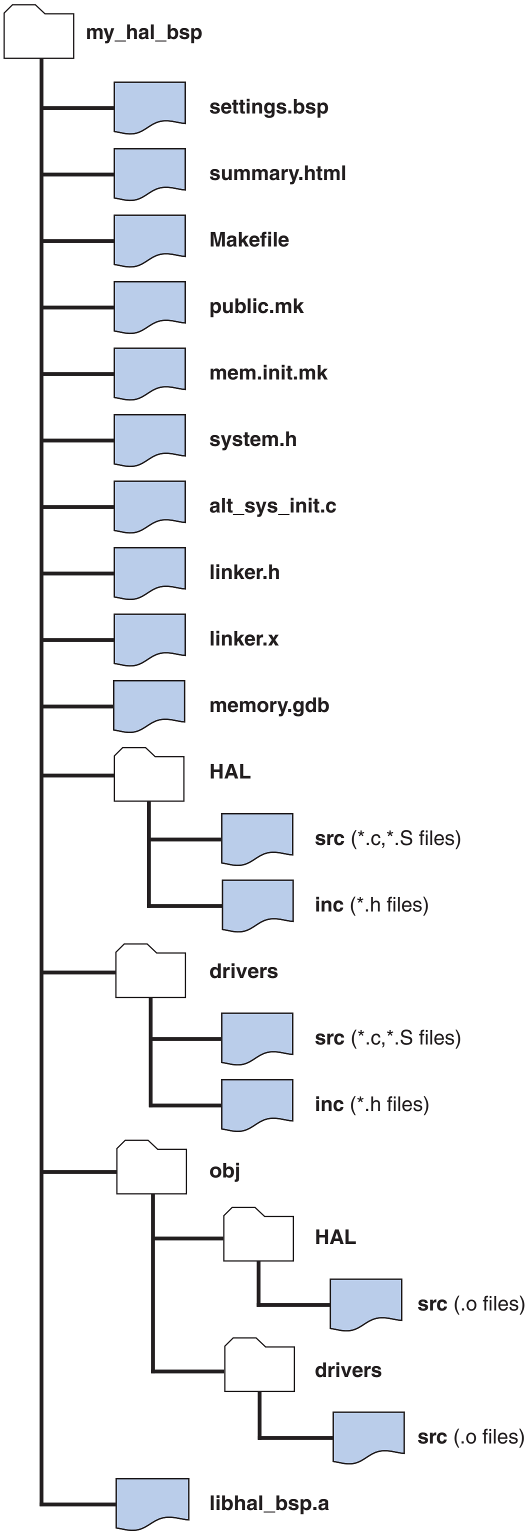
Table 4–8 details all the copied BSP files shown in Figure 4–3.

Table 4–8. Copied BSP Files

File Name	Function
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	Contains all driver source code. The files in this directory are all copied files. The **drivers** directory has **src** and **inc** subdirectories like the **HAL** directory.

Figure 4–4 shows a BSP directory after executing **make.**

Figure 4–4. HAL BSP After Build



Linker Map Validation

When a BSP is generated, the SBT validates the linker region and section mappings, to ensure that they are valid for a HAL project. The tools display an error in each of the following cases:

- The `.entry` section maps to a nonexistent region.
- The `.entry` section maps to a memory region that is less than 32 bytes in length.
- The `.entry` section maps to a memory region that does not start on the reset vector base address.
- The `.exceptions` section maps to a nonexistent region.
- The `.exceptions` section maps to a memory region that does not start on the exception vector base address.
- The `.entry` section and `.exceptions` section map to the same device, and the memory region associated with the `.exceptions` section precedes the memory region associated with the `.entry` section.
- The `.entry` section and `.exceptions` section map to the same device, and the base address of the memory region associated with the `.exceptions` section is less than 32 bytes above the base address of the memory region associated with the `.entry` section.

Tcl Scripts for BSP Settings

In many cases, you can fully specify your Nios II BSP with the Nios II SBT settings and defaults. However, in some cases you might need to create some simple Tcl scripts to customize your BSP.

You control the characteristics of your BSP by manipulating BSP settings, using Tcl commands. The most powerful way of using Tcl commands is by combining them in Tcl scripts.

Tcl scripting gives you maximum control over the contents of your BSP. One advantage of Tcl scripts over command-line arguments is that a Tcl script can obtain information from the hardware system or pre-existing BSP settings, and then use it later in script execution.

For descriptions of the Tcl commands used to manipulate BSPs, refer to "Software Build Tools Tcl Commands" in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Calling a Custom BSP Tcl Script

From the Nios II Command Shell, you can call a custom BSP Tcl script with any of the following commands:

```
nios2-bsp --script custom_bsp.tcl
nios2-bsp-create-settings --script custom_bsp.tcl
nios2-bsp-query-settings --script custom_bsp.tcl
nios2-bsp-update-settings --script custom_bsp.tcl
```

In the Nios II BSP editor, you can execute a Tcl script when generating a BSP, through the **New BSP Settings File** dialog box.

For information about using Tcl scripts in the SBT for Eclipse, refer to “Using the BSP Editor” in the *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook.*

For an example of custom Tcl script usage, refer to “Creating Memory Initialization Files” on page 4–11.

Any settings you specify in your script override the BSP default values. For further information about BSP defaults, refer to “Specifying BSP Defaults” on page 4–35.

When you update an existing BSP, you must include any scripts originally used to create it. Otherwise, your project's settings revert to the defaults.

When you use a custom Tcl script to create your BSP, you must include the script in the set of files archived in your version control system. For further information, refer to “Using Version Control” on page 4–9.

The Tcl script in Example 4–8 is a very simple example that sets `stdio` to a device with the name `my_uart`.

Example 4–8. 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
```

Example 4–9 illustrates how you might use more powerful scripting capabilities to customize a BSP based on the contents of the hardware system.

The Nios II SBT uses slave descriptors to refer to components connected to the Nios II processor. A slave descriptor is the unique name of a hardware component's slave port.

If a component has only one slave port connected to the Nios II processor, 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`).

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

The script shown in Example 4–9 is similar to **bsp-stdio-utils.tcl**, which examines the hardware system and determines what device to use for `stdio`. For details, refer to “Specifying BSP Defaults” on page 4–35.

Example 4–9. Tcl Script to Examine Hardware and Choose Settings

```
# Select a device connected to the processor as the default STDIO devi
ce.

# 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 processor.
    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 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
```

Revising Your BSP

Your BSP is customized to your hardware design and your software requirements. If your hardware design or software requirements change, you usually need to revise your BSP.

Every BSP is based on a Nios II processor in a hardware system. The BSP settings file does not duplicate information available in the **.sopcinfo** file, but it does contain system-dependent settings that reference system information. Because of these system-dependent settings, a BSP settings file can become inconsistent with its system if the system changes.

You can revise a BSP at several levels. This section describes each level, and provides guidance about when to use it.

Rebuilding Your BSP

Rebuilding a BSP is the most superficial way to revise a BSP.

What Happens

Rebuilding the BSP simply recreates all BSP object files and the **.a** library file. BSP settings, source files, and compiler options are unchanged.

How to Rebuild Your BSP

In the Nios II SBT for Eclipse, right-click the BSP project and click **Build**.

On the command line, change to the BSP directory and type `make`.

Regenerating Your BSP

Regenerating the BSP refreshes the BSP source files without updating the BSP settings.

What Happens

Regenerating a BSP has the following effects:

- Reads the **.sopcinfo** file for basic system parameters such as module base addresses and clock frequencies.
- Retrieves the current system identification (ID) from the **.sopcinfo** file. Ensures that the correct system ID is inserted in the **.elf** file the next time the BSP is built.
- Adjusts the default memory map to correspond to changes in memory sizes. If you are using a custom memory map, it is untouched.
- Retains all other existing settings in the BSP settings file.

> ☞ Except for adjusting the default memory map, the SBT does not ensure that the settings are consistent with the hardware design in the **.sopcinfo** file.

- Ensures that the correct set of BSP files is present, as follows:
  - Copies all required source files to the BSP directory tree. Copied BSP files are listed in Table 4–8 on page 4–25.

    If a copied file (such as a HAL source file) already exists, the SBT checks the file timestamp against the timestamp of the file in the Nios II EDS installation. The tools do not replace the BSP file unless it differs from the distribution file.
  - Recreates all generated files. Generated BSP files are listed in Table 4–7 on page 4–24.

  ☞ You can disable generation of any BSP file in the BSP Editor, or on the command line with the `set_ignore_file` Tcl command. Otherwise, changes you make to a BSP file are lost when you regenerate the BSP. Whenever possible, use BSP settings, or custom device drivers or software packages, to customize your BSP.

  - Removes any files that are not required, for example, source files for drivers that are no longer in use.

When to Regenerate Your BSP

Regenerating your BSP is required (and sufficient) in the following circumstances:

- You change your hardware design, but all BSP system-dependent settings remain consistent with the new **.sopcinfo** file. The following are examples of system changes that do not affect BSP system-dependent settings:
  - Changing a component’s base address
  - With the internal interrupt controller (IIC), adding or removing hardware interrupts
  - With the IIC, changing a hardware interrupt number
  - Changing a clock frequency
  - Changing a simple processor option, such as cache size or core type
  - Changing a simple component option, other than memory size.
  - Adding a bridge
  - Adding a new component
  - Removing or renaming a component, other than a memory component, the `stdio` device, or the system timer device
  - Changing the size of a memory component when you are using the default memory map

  ☞ Unless you are sure that your modified hardware design remains consistent with your BSP settings, update your BSP as described in “Updating Your BSP” on page 4–32.

- You want to eliminate any customized source files and revert to the distributed BSP code.

  ☞ To revert to the distributed BSP code, you must ensure that you have not disabled generation on any BSP files.

- You have installed a new version of the Nios II EDS, and you want the updated BSP software implementations.
- When you attempt to rebuild your project, an error message indicates that the BSP must be updated.
- You have updated or recreated the BSP settings file.

How to Regenerate Your BSP

You can regenerate your BSP in the Nios II SBT for Eclipse, or with SBT commands at the command line.

Regenerating Your BSP in Eclipse

In the Nios II SBT for Eclipse, right-click the BSP project, point to **Nios II**, and click **Generate BSP**.

☞ For information about generating a BSP with the SBT for Eclipse, refer to the *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook*.

Regenerating Your BSP from the Command Line

From the command line, use the **nios2-bsp-generate-files** command.

☞ For information about the **nios2-bsp-generate-files** command, refer to the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Updating Your BSP

When you update a BSP, you recreate the BSP settings file based on the current hardware definition and previous BSP settings.

☞ You must always regenerate your BSP after updating the BSP settings file.

What Happens

Updating a BSP has the following effects:

- System-dependent settings are derived from the original BSP settings file, but adjusted to correspond with any changes in the hardware system.
- Non-system-dependent BSP settings persist from the original BSP settings file.

Also refer to "Regenerating Your BSP" on page 4–30 for actions taken when you regenerate the BSP after updating it.

When to Update Your BSP

Updating your BSP is necessary in the following circumstances:

- A change to your BSP settings is required.
- Changes to your **.sopcinfo** file make it inconsistent with your BSP. The following are examples of system changes that affect BSP system-dependent settings:
  - Renaming the processor
  - Renaming or removing a memory, the `stdio` device, or the system timer device
  - Changing the size of a memory component when using a custom memory map
  - Changing the processor reset or exception slave port or offset
  - Adding or removing an external interrupt controller (EIC)
  - Changing the parameters of an EIC
- When you attempt to rebuild your project, an error message indicates that you must update the BSP.

How to Update Your BSP

You can update your BSP at the command line. You have the option to use a Tcl script to control your BSP settings.

From the command line, use the **nios2-bsp-update-settings** command. You can use the `--script` option to define the BSP with a Tcl script.

For details about the **nios2-bsp-update-settings** command, refer to the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

nios2-bsp-update-settings does not reapply default settings unless you explicitly call the top-level default Tcl script with the `--script` option.

For information about using the default Tcl script, refer to "Specifying BSP Defaults" on page 4–35.

Alternatively, you can update your BSP with the **nios2-bsp** script. **nios2-bsp** determines that your BSP already exists, and uses the **nios2-bsp-update-settings** command to update the BSP settings file.

The **nios2-bsp** script 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 Tcl Script for BSP Defaults" on page 4–36. Alternatively, you can provide **nios2-bsp** with command-line options or Tcl scripts to override the default settings.

For information about using the **nios2-bsp** script, refer to the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Recreating Your BSP

When you recreate your BSP, you start over as if you were creating a new BSP.

☞ After you recreate your BSP, you must always regenerate it.

What Happens

Recreating a BSP has the following effects:

- System-dependent settings are created based on the current hardware system.
- Non-system-dependent settings can be selected by the default Tcl script, by values you specify, or both.

Also refer to "Regenerating Your BSP" on page 4–30 for actions taken when you generate the BSP after recreating it.

When to Recreate Your BSP

If you are working exclusively in the Nios II SBT for Eclipse, and you modify the underlying hardware design, the best practice is to create a new BSP. Creating a BSP is very easy with the SBT for Eclipse. Manually correcting a large number of interrelated settings, on the other hand, can be difficult.

How to Recreate Your BSP

You can recreate your BSP in the Nios II SBT for Eclipse, or using the SBT at the command line. Regardless which method you choose, you can use Tcl scripts to control and reproduce your BSP settings. This section describes the options for recreating BSPs.

Using Tcl Scripts When Recreating Your BSP

A Tcl script automates selection of BSP settings. This automation ensures that you can reliably update or recreate your BSP with its original settings. Except when creating very simple BSPs, Altera recommends specifying all BSP settings with a Tcl script.

To use Tcl scripts most effectively, it is best to create a Tcl script at the time you initially create the BSP. However, the BSP Editor enables you to export a Tcl script from an existing BSP.

For details about exporting Tcl scripts, refer to "Using the BSP Editor" in the *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook*.

By recreating the BSP settings file with a Tcl script that specifies all BSP settings, you can reproduce the original BSP while ensuring that system-dependent settings are adjusted correctly based on any changes in the hardware system.

For information about Tcl scripting with the SBT, refer to "Tcl Scripts for BSP Settings" on page 4–27.

Recreating Your BSP in Eclipse

The process for recreating a BSP is the same as the process for creating a new BSP. The Nios II SBT for Eclipse provides an option to import a Tcl script when creating a BSP.

For details, refer to "Getting Started with Eclipse" and "Using the BSP Editor" in the *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook.*

Recreating Your BSP at the Command Line

Recreate your BSP using the **nios2-bsp-create-settings** command. You can use the `--script` option to define the BSP with a Tcl script.

The **nios2-bsp-create-settings** command does not apply default settings to your BSP. However, you can use the `--script` command-line option to run the default Tcl script. For information about the default Tcl script, refer to "Specifying BSP Defaults".

For information about using the **nios2-bsp-create-settings** command, refer to the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Specifying BSP Defaults

The Nios II SBT sets BSP defaults using a set of Tcl scripts. Table 4–9 lists the components of the BSP default Tcl scripts included in the Nios II SBT. These scripts specify default BSP settings. The scripts are located in the following directory:

*<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.

For more information about Tcl scripting with the SBT, refer to "Tcl Scripts for BSP Settings" on page 4–27.

The Nios II SBT uses the default Tcl scripts to specify default values for system-dependent settings. System-dependent settings are BSP settings that reference system information in the .**sopcinfo** file.

The SBT executes the default Tcl script before any user-specified Tcl scripts. As a result, user input overrides settings made by the default Tcl script.

You can pass command-line options to the default Tcl script to override the choices it makes or to prevent it from making changes to settings. For details, refer to "Top Level Tcl Script for BSP Defaults".

The default Tcl script makes the following choices for you based on your hardware 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.

Top Level Tcl Script for BSP Defaults

The top level Tcl script for setting BSP defaults is **bsp-set-defaults.tcl**. This script specifies BSP system-dependent settings, which depend on the hardware system. The **nios2-bsp-create-settings** and **nios2-bsp-update-settings** utilities 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. Both the Nios II SBT for Eclipse and the **nios2-bsp** script call the default Tcl script by invoking 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 **.sopcinfo** 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 on the **nios2-bsp-create-settings**, **nios2-bsp-query-settings**, or **nios2-bsp-update-settings** command line, by using the `--script` argument. It has the following usage:

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

Table 4–10 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, refer to the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

The script searches the **.sopcinfo** 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 processor. 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, refer to the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

The script searches the **.sopcinfo** file for a timer component to use as the default system timer. To be an appropriate system timer, the component must have the following characteristics:

- It must be a timer, that is, `is_timer_device` must return true.
- It must have a slave port connected to the Nios II processor.

When the script finds an appropriate system timer component, it sets `hal.sys_clk_timer` to the timer slave port descriptor. The script prefers a slave port whose descriptor contains the string `sys_clk`, if one exists. If no appropriate system timer component is found, the script sets `hal.sys_clk_timer` to `none`.

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, refer to 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 assigns 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, refer to 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, refer to 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, refer to the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Using Individual Default Tcl Procedures

The default Tcl script consists of the top-level **bsp-call-proc.tcl** script plus the helper scripts listed in Table 4–9 on page 4–35. The procedure call Tcl script allows you to call 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 view the available functions 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 Nios II SBT can incorporate device drivers and software packages supplied by Altera, supplied by other third-party developers, or created by you.

For details about integrating device drivers and software packages with the Nios II SBT, refer to the *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

Boot Configurations for Altera Embedded Software

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

For detailed information about the HAL boot loader process, refer to the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

Table 4–12 shows the memory types that the default Tcl script recognizes when 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 **.sopcinfo** file) indicates whether the **.sopcinfo** file identifies the memory as a flash memory device. The `IsNonVolatileStorage` property indicates whether the **.sopcinfo** 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 4–12. 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 ROM, HardCopy ROM	false	true
RAM	On-chip memory configured as RAM, HardCopy RAM, SDRAM, synchronous static RAM (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 to 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 modifies 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 processor 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 memory in the Nios II EDS, precompiled to Motorola S-record Files (**.srec**). 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 chooses this configuration only if 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** script, call 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 processor comes out of reset. The initialization might be implemented by using a non-volatile reset memory (for example, flash, ROM, initialized FPGA RAM) or by an external master (for example, another processor) 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 4–39.

`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.

In this boot configuration, when the processor core resets, by default the `.rwdata` section is not reinitialized. Reinitialization would normally be done by a boot loader. However, this configuration has no boot loader, because the software is running out of memory that is assumed to be preinitialized before startup.

If your software has a `.rwdata` section that must be reinitialized at processor reset, turn on the `hal.linker.enable_alt_load_copy_rwdata` setting in the BSP.

Run-time Configurable Reset Configuration

The reset address points to a memory that contains code that executes before the normal reset code. When the processor 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 processor 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 processor 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.

Altera-Provided Embedded Development Tools

This section lists the components of the Nios II SBT, and other development tools that Altera provides for use with the SBT. This section does not describe detailed usage of the tools, but refers you to the most appropriate documentation.

Nios II Software Build Tool GUIs

The Nios II EDS provides the following SBT GUIs for software development:

- The Nios II SBT for Eclipse
- The Nios II BSP Editor
- The Nios II Flash Programmer

Each GUI is primarily a thin layer providing graphical control of the command-line tools described in "The Nios II Command-Line Commands" on page 4–44.

Table 4–13 outlines the correlation between GUI features and the SBT command line.

Table 4–13. Summary of Nios II GUI Tasks

Task	Tool	Feature	Nios II SBT Command Line
Creating an example Nios II program	Nios II SBT for Eclipse	**Nios II Application and BSP from Template** wizard	**create-this-app** script
Creating an application	Nios II SBT for Eclipse	**Nios II Application** wizard	**nios2-app-generate-makefile** utility
Creating a user library	Nios II SBT for Eclipse	**Nios II Library** wizard	**nios2-lib-generate-makefile** utility
Creating a BSP	Nios II SBT for Eclipse	**Nios II Board Support Package** wizard	■ Simple: ■ **nios2-bsp** script ■ Detailed: ■ **nios2-bsp-create-settings** utility ■ **nios2-bsp-generate-files** utility
	BSP Editor	**New BSP Setting File** dialog box	
Modifying an application	Nios II SBT for Eclipse	**Nios II Application Properties** page	**nios2-app-update-makefile** utility
Modifying a user library	Nios II SBT for Eclipse	**Nios II Library Properties** page	**nios2-lib-update-makefile** utility
Updating a BSP	Nios II SBT for Eclipse	**Nios II BSP Properties** page	**nios2-bsp-update-settings** utility **nios2-bsp-generate-files** utility
	BSP Editor	—	
Examining properties of a BSP	Nios II SBT for Eclipse	**Nios II BSP Properties** page	**nios2-bsp-query-settings** utility
	BSP Editor	—	
Programming flash memory	Nios II Flash Programmer	—	**nios2-flash-programmer**
Importing a command-line project	Nios II SBT for Eclipse	**Import** dialog box	—

The Nios II SBT for Eclipse

The Nios II SBT for Eclipse is a configuration of the popular Eclipse development environment, specially adapted to the Nios II family of embedded processors. The Nios II SBT for Eclipse includes Nios II plugins for access to the Nios II SBT, enabling you to create applications based on the Altera HAL, and debug them using the JTAG debugger.

You can launch the Nios II SBT for Eclipse either of the following ways:

- In the Windows operating system, on the Start menu, point to **Programs** > **Altera** > **Nios II EDS** <*version*>, and click **Nios II** <*version*> **Software Build Tools for Eclipse**.
- From the Nios II Command Shell, by typing `eclipse-nios2`.

For more information about the Nios II SBT for Eclipse, refer to the *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook.*

The Nios II BSP Editor

You can create or modify a Nios II BSP project with the Nios II BSP Editor, a standalone GUI that also works with the Nios II SBT for Eclipse. You can launch the BSP Editor either of the following ways:

- From the Nios II menu in the Nios II SBT for Eclipse
- From the Nios II Command Shell, by typing **nios2-bsp-editor**.

The Nios II BSP Editor enables you to edit settings, linker regions, and section mappings, and to select software packages and device drivers.

The capabilities of the Nios II BSP Editor constitute a large subset of the capabilities of the **nios2-bsp-create-settings**, **nios2-bsp-update-settings**, and **nios2-bsp-generate-files** utilities. Any project created in the BSP Editor can also be created using the command-line utilities.

For more information about the BSP Editor, refer to "Using the BSP Editor" in the *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook.*

The Nios II Flash Programmer

The Nios II flash programmer allows you to program flash memory devices 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

You can launch the flash programmer either of the following ways:

- From the Nios II menu in the Nios II SBT for Eclipse
- From the Nios II Command Shell, by typing:

```
nios2-flash-programmer-generate↵
```

The Nios II Command Shell

The Nios II Command Shell is a **bash** command-line environment initialized with the correct settings to run Nios II command-line tools. The Nios II EDS includes two versions of the Nios II Command Shell, for the two supported GCC toolchain versions, described in "GNU Compiler Tool Chain".

For information about launching the Nios II Command Shell, refer to the *Getting Started from the Command Line* chapter of the *Nios II Software Developer's Handbook*.

The Nios II Command-Line Commands

This section describes the Altera Nios II command-line tools. You can run these tools from the Nios II Command Shell.

Each tool provides its own documentation in the form of help accessible from the command line. To view the help, open the Nios II Command Shell, and type the following command:

```
<name of tool> --help↵
```

GNU Compiler Tool Chain

The Nios II compiler tool chain is based on the standard GNU **GCC** compiler, assembler, linker, and make facilities. Altera provides and supports the standard GNU compiler tool chain for the Nios II processor.

The Nios II EDS includes version GCC 4.7.3 of the GCC toolchain.

For detailed information about installing the Altera Complete Design Suite, refer to the *Altera Software Installation and Licensing Manual*.

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**.

The Nios II GNU tools reside in the following location:

- <*Nios II EDS install path*>**/bin/gnu** directory

Refer to the following additional sources of information:

- For information about managing GCC toolchains in the SBT for Eclipse—"Managing Toolchains in Eclipse" in the *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook*
- For information about selecting the toolchain on the command line—the *Getting Started from the Command Line* chapter of the *Nios II Software Developer's Handbook*
- For a comprehensive list of Nios II GNU tools—the GNU HTML documentation, available at the Nios II Embedded Design Suite Support page of the Altera website
- For further information about GNU from the Free Software Foundation website (www.gnu.org).

Nios II Software Build Tools

The Nios II SBT utilities and scripts provide the functionality underlying the Nios II SBT for Eclipse. You can create, modify, and build Nios II programs with commands typed at a command line or embedded in a script.

Table 4–14 summarizes the command-line utilities and scripts included in the Nios II SBT. You can call these utilities and scripts on the command line or from the scripting language of your choice (such as **perl** or **bash**).

Table 4–14. Nios II SBT Utilities and Scripts

Command	Summary	Utility	Script
nios2-app-generate-makefile	Creates an application makefile	✓	
nios2-lib-generate-makefile	Creates a user library makefile	✓	
nios2-app-update-makefile	Modifies an existing application makefile	✓	
nios2-lib-update-makefile	Modifies an existing user 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	✓	
nios2-bsp	Creates or updates a BSP		✓
create-this-app	Creates an example application project		✓
create-this-bsp	Creates an example BSP project		✓

The Nios II SBT utilities reside in the <*Nios II EDS install path*>**/sdk2/bin** directory.

For further information about the Nios II SBT, refer to the *Getting Started from the Command Line* 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 4–15 shows the Altera-provided utilities for converting file formats.

Table 4–15. File Conversion Utilities

Utility	Description
bin2flash	Converts binary files to a Nios II Flash Programmer File (**.flash**) for programming to flash memory.
elf2dat	Converts a **.elf** file to a **.dat** file format appropriate for Verilog HDL hardware simulators.
elf2flash	Converts a **.elf** file to a **.flash** file for programming to flash memory.
elf2hex	Converts a **.elf** file to a Hexadecimal (Intel-format) File (**.hex**).
elf2mem	Generates the memory contents for the memory devices in a specific Nios II system.
elf2mif	Converts a **.elf** file to a Quartus® II Memory Initialization File (**.mif**).
flash2dat	Converts a **.flash** file to the **.dat** file format appropriate for Verilog HDL hardware simulators.
sof2flash	Converts an SRAM Object File (**.sof**) to a **.flash** file.

The file format conversion tools are in the <*Nios II EDS install path*>**/bin/** directory.

Other Command-Line Tools

Table 4–16 shows other Altera-provided command-line tools for developing Nios II programs.

Table 4–16. Altera Command-Line Tools

Tool	Description
nios2-download	Downloads code to a target processor for debugging or running.
nios2-flash-programmer-generate	Allows multiple files to be converted to **.flash** files, and optionally programs each file to the specified location on a flash device.
nios2-flash-programmer	Programs data to flash memory on the target board.
nios2-gdb-server	Translates GNU debugger (GDB) remote serial protocol packets over Transmission Control Protocol (TCP) to JTAG transactions with a target Nios II processor.
nios2-terminal	Performs terminal I/O with a JTAG 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-configure-sof	Configures an Altera configurable part. If no explicit **.sof** file 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.

Restrictions

The Nios II SBT supports BSPs incorporating the Altera HAL and Micrium MicroC/OS-II only.

Document Revision History

Table 4–17 shows the revision history for this document.

Table 4–17. Document Revision History (Part 1 of 2)

Date	Version	Changes
January 2014	13.1.0	■ Updated GCC4 toolchain from 4.1.2 to GCC 4.7.3. ■ Removed references to Nios II IDE. ■ Removed references to Nios II GCC3.
May 2011	11.0.0	■ Introduction of Qsys system integration tool ■ The GCC 3 toolchain is an optional feature
February 2011	10.1.0	Removed "Referenced Documents" section.

Table 4–17. Document Revision History (Part 2 of 2)

Date	Version	Changes
July 2010	10.0.0	■ Added explanation of the effects of disabled BSP file generation. ■ Described regeneration of BSP with changed memory sizes. ■ Described GCC 4. ■ Described GCC 3 and GCC 4 command shells
November 2009	9.1.0	■ Chapter repurposed and retitled to cover Nios II Software Build Tools functionality applicable to both command line and Eclipse. ■ Describe the Nios II Flash Programmer
March 2009	9.0.0	■ Moved information about Tcl-based device drivers and software packages, formerly in this chapter, to *Developing device Drivers for the Hardware Abstraction Layer*. ■ Described how to work with compiler optimization and debugger settings. ■ Described newlib recompilation. ■ Corrected minor typographical errors.
May 2008	8.0.0	■ Advanced exceptions added to Nios II core. ■ Added instructions for writing instruction-related exception handler. ■ Design examples removed from list.
October 2007	7.2.0	Initial release. Material moved here from former *Nios II Software Build Tools* chapter.

Section II. Hardware Abstraction Layer

This section describes the Nios® II hardware abstraction layer (HAL). It includes the following chapters:

- 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

5. Overview of the Hardware Abstraction Layer

NII52003-11.0.0

This chapter introduces the hardware abstraction layer (HAL) for the Nios® II processor. This chapter contains the following sections:

- "Getting Started with the Hardware Abstraction Layer" on page 5–1
- "HAL Architecture for Embedded Software Systems" on page 5–2
- "Supported Peripherals" on page 5–4

The HAL is a lightweight embedded runtime environment that provides a simple device driver interface for programs to connect to 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. The Nios II software development tools extract system information from your SOPC Information File (**.sopcinfo**). The Nios II Software Build Tools (SBT) generate a custom HAL board support package (BSP) specific to your hardware configuration. Changes in the hardware configuration automatically propagate to the HAL device driver configuration. As a result, changes in the underlying hardware are prevented 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 with the Hardware Abstraction Layer

The easiest way to get started using the HAL is to create a software project. In the process of creating a new project, you also create a HAL BSP. You need not create or copy HAL files, and you need not edit any of the HAL source code. The Nios II SBT generates the HAL BSP for you.

For an exercise in creating a simple Nios II HAL software project, refer to "Getting Started with Eclipse" in the *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook.*

In the Nios II SBT command line, 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 hardware system. A Nios II system consists of a Nios II processor core integrated with peripherals and memory. Nios II systems are generated by Qsys or SOPC Builder.

If you do not have a custom Nios II system, you can base your project on an Altera-provided example hardware system. In fact, you can first start developing projects targeting an Altera® development board, and later re-target the project to a custom board. You can easily change the target hardware system later.

For information about creating a new project with the Nios II SBT, refer to the *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook*, or to the *Getting Started from the Command Line* chapter of the *Nios II Software Developer's Handbook*.

HAL Architecture for Embedded Software Systems

This section describes the fundamental elements of the HAL architecture.

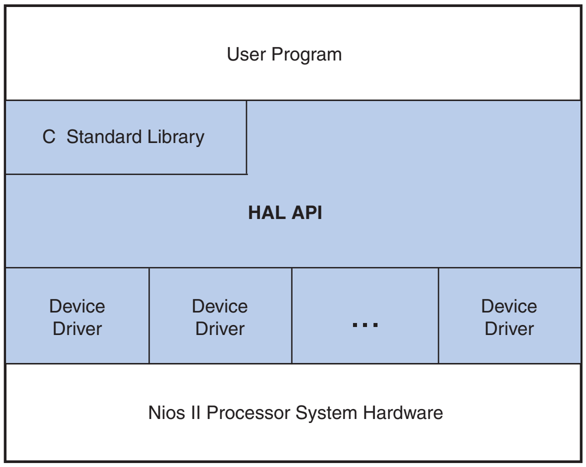
Services

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()` runs

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 versus Drivers

Application developers 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 about the HAL, refer to the following chapters:

- The *Developing Programs Using the Hardware Abstraction Layer* 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 Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook* describes how to communicate directly with hardware and how to make hardware resources available with the 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 models 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—A mechanism for accessing files stored in 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—Devices that 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.
- Direct memory access (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 need not 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 need only provide this set of driver functions. As a result, your driver development task is predefined 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 in 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**.

Embedded Hardware Supported by the HAL

This section summarizes Nios II HAL support for Nios II hardware.

Nios II Processor Core Support

The Nios II HAL supports all available Nios II processor core implementations.

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 with 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 erasable programmable configurable serial (EPCS) serial configuration device controller
- File subsystems
  - Altera host based file system
  - Altera read-only zip file system
- Timer devices
  - Timer core
- DMA devices
  - DMA controller core
  - Scatter-gather DMA controller core
- Ethernet devices
  - Triple Speed Ethernet MegaCore® function
  - 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 the Embedded Software page of the Altera website.

All peripherals (both from Altera and third party vendors) must provide a header file that defines the peripheral's low-level interface to hardware. Therefore, 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 use unnamed constants, such as hard-coded addresses, to access a peripheral.

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 with the Nios II processor system. The PIO peripheral does not fit in 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 about Altera-provided peripherals, refer to the *Embedded Peripherals IP User Guide*.

MPU Support

The HAL does not include explicit support for the optional memory protection unit (MPU) hardware. However, it does support an advanced exception handling system that can handle Nios II MPU exceptions.

For details about handling MPU and other advanced exceptions, refer to the *Exception Handling* chapter of the *Nios II Software Developer's Handbook*. For details about the MPU hardware implementation, refer to the *Programming Model* chapter of the *Nios II Processor Reference Handbook*.

MMU Support

The HAL does not support the optional memory management unit (MMU) hardware. To use the MMU, you need to implement a full-featured operating system.

For details about the Nios II MMU, refer to the *Programming Model* chapter of the *Nios II Processor Reference Handbook*.

Document Revision History

Table 5–1 shows the revision history for this document.

Table 5–1. Document Revision History

Date	Version	Changes
May 2011	11.0.0	Introduction of Qsys system integration tool
February 2011	10.1.0	Removed "Referenced Documents" section.
July 2010	10.0.0	Maintenance release.
November 2009	9.1.0	Maintenance release.
March 2009	9.0.0	■ Reorganized and updated information and terminology to clarify role of Nios II Software Build Tools. ■ Corrected minor typographical errors.
May 2008	8.0.0	Maintenance release.
October 2007	7.2.0	Maintenance release.
May 2007	7.1.0	■ Scatter-gather DMA core. ■ Triple-speed Ethernet MAC. ■ Refer to HAL generation with Nios II Software Build Tools. ■ Added table of contents to "Introduction" section. ■ Added Referenced Documents section.
March 2007	7.0.0	Maintenance release.
November 2006	6.1.0	NicheStack TCP/IP Stack - Nios II Edition.
May 2006	6.0.0	Maintenance release.
October 2005	5.1.0	Maintenance release.
May 2005	5.0.0	Maintenance release.
May 2004	1.0	Initial release

6. Developing Programs Using the Hardware Abstraction Layer

NII52004-13.1.0

This chapter discusses how to develop embedded programs for the Nios® II embedded processor based on the Altera® hardware abstraction layer (HAL). This chapter contains the following sections:

The application program interface (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 with 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).

HAL BSP Settings

Every Nios II board support package (BSP) has settings that determine the BSP's characteristics. For example, HAL BSPs have settings to identify the hardware components associated with standard devices such as `stdout`. Defining and manipulating BSP settings is an important part of Nios II project creation. You manipulate BSP settings with the Nios II BSP Editor, with command-line options, or with Tcl scripts.

For details about how to control BSP settings, refer to one or more of the following documents:

- For the Nios II SBT for Eclipse, refer to the *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook*.
- For the Nios II SBT command line, refer to the *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

For detailed descriptions of available BSP settings, refer to the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Many HAL settings are reflected in the **system.h** file, which provides a helpful reference for details about your BSP. For information about **system.h**, refer to "The system.h System Description File" on page 6–4.

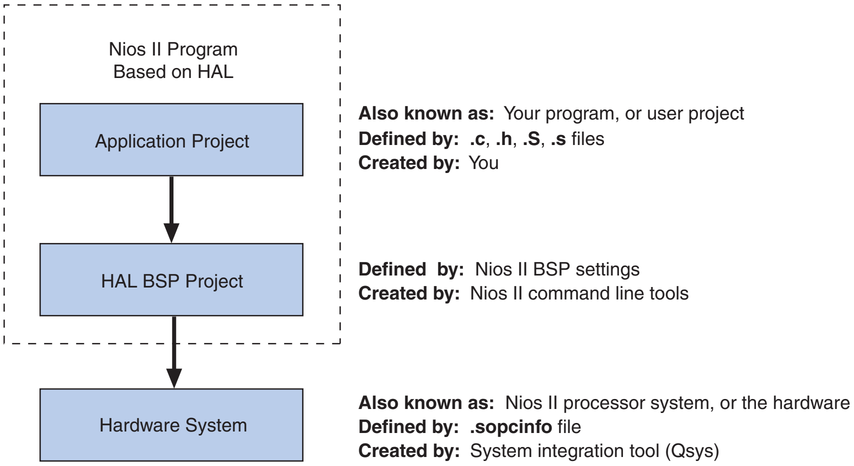
Do not edit **system.h**. The Nios II EDS provides tools to manipulate system settings.

The Nios II Embedded Project Structure

The creation and management of software projects based on the HAL is integrated tightly with the Nios II SBT. 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**, **.s** files
Created by: You

Defined by: Nios II BSP settings
Created by: Nios II command line tools

Also known as: Nios II processor system, or the hardware
Defined by: **.sopcinfo** file
Created by: System integration tool (Qsys)

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.

With the Nios II SBT for Eclipse, the tools create the HAL BSP project when you create your application project. In the Nios II SBT command line flow, you create the BSP using **nios2-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 hardware system are incorporated in the BSP project.

The BSP project depends on the hardware system, defined by a SOPC Information File (**.sopcinfo**). The Nios II SBT can keep your BSP up-to-date with the hardware 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.

You can use the Nios II SBT to update your BSP to match updated hardware. You control whether and when these updates occur.

For details about how the SBT keeps your BSP up-to-date with your hardware system, refer to “Revising Your BSP” in 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 with a few simple SBT commands.

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 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
- Interrupt request (IRQ) information (if any)
- A symbolic name for the peripheral

The Nios II SBT generates 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**. The SBT provides facilities to manipulate system settings.

For details about how to control BSP settings, refer to "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 this file 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.
`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 in the HAL environment. The HAL uses these functions primarily 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 contains all 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. Refer to "File System" on page 6–6.

For details about 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, refer to the *Read-Only Zip File System* chapter of the *Nios II Software Developer's Handbook*.

You can access files in 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 about 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 in 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 with UNIX-style path names. The HAL registers character mode devices as nodes in 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 at system generation time. For example, a UART peripheral that appears as **uart1** in Qsys or SOPC builder is named **/dev/uart1** in **system.h**.

The code in Example 6–2 reads 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 about 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** > **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 UART. Character mode devices are registered as nodes in 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 shows the classic Hello World program. This program sends characters to whatever device is associated with `stdout` when the program is compiled.

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 other than stdin, stdout, or stderr is as easy as opening and writing to a file. The code in Example 6–4 writes 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 can 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 in the system.

Lightweight Character-Mode I/O

The HAL offers several methods of reducing the code footprint of character-mode device drivers. For details, refer to "Reducing Code Footprint in Embedded Systems" on page 6–30.

Altera Logging Functions

The Altera logging functions provide a separate channel for sending logging and debugging information to a character-mode device, supplementing stdout and stderr. The Altera logging information can be printed in response to several conditions. Altera logging can be enabled and disabled independently of any normal stdio output, making it a powerful debugging tool.

When Altera logging is enabled, your software can print extra messages to a specified port with HAL function calls. The logging port, specified in the BSP, can be a UART or a JTAG UART device. In its default configuration, Altera logging prints out boot messages, which trace each step of the boot process.

☞ Avoid setting the Altera logging device to the device used for `stdout` or `stderr`. If Altera logging output is sent to `stdout` or `stderr`, the logging output might appear interleaved with the `stdout` or `stderr` output

Several logging options are available, controlled by C preprocessor symbols. You can also choose to add custom logging messages.

☞ Altera logging changes system behavior. The logging implementation is designed to be as simple as possible, loading characters directly to the transmit register. It can have a negative impact on software performance.

Altera logging functions are conditionally compiled. When logging is disabled, it has no impact on code footprint or performance.

☞ The Altera reduced device drivers do not support Altera logging.

Enabling Altera Logging

The Nios II SBT has a setting to enable Altera logging. The setting is called **hal.log_port**. It is similar to **hal.stdout**, **hal.stdin**, and **hal.stderr**. To enable Altera logging, you set **hal.log_port** to a JTAG UART or a UART device. The setting allows the HAL to send log messages to the specified device when a logging macro is invoked.

When Altera logging is enabled, the Nios II SBT defines `ALT_LOG_ENABLE` in **public.mk** to enable log messages. The build tools also set the `ALT_LOG_PORT_TYPE` and `ALT_LOG_PORT_BASE` values in **system.h** to point to the specified device.

When Altera logging is enabled without special options, the HAL prints out boot messages to the selected port. For typical software that uses the standard **alt_main.c** (such as the Hello World software example), the messages appear as in Example 6–5.

Example 6–5. Default Boot Logging Output

```
[crt0.S] Inst & Data Cache Initialized.
[crt0.S] Setting up stack and global pointers.
[crt0.S] Clearing BSS
[crt0.S] Calling alt_main.
[alt_main.c] Entering alt_main, calling alt_irq_init.
[alt_main.c] Done alt_irq_init, calling alt_os_init.
[alt_main.c] Done OS Init, calling alt_sem_create.
[alt_main.c] Calling alt_sys_init.
[alt_main.c] Done alt_sys_init.  Redirecting IO.
[alt_main.c] Calling C++ constructors.
[alt_main.c] Calling main.
[alt_exit.c] Entering _exit() function.
[alt_exit.c] Exit code from main was 0.
[alt_exit.c] Calling ALT_OS_STOP().
[alt_exit.c] Calling ALT_SIM_HALT().
[alt_exit.c] Spinning forever.
```

A write operation to the Altera logging device stalls in `ALT_LOG_PRINTF()` until the characters are read from the Altera logging device's output buffer. To ensure that the Nios II application completes initialization, run the **nios2-terminal** command from the Nios II Command Shell to accept the Altera logging output.

Extra Logging Options

In addition to the default boot messages, logging options are incorporated in Altera logging. Each option is controlled by a C preprocessor symbol. The details of each option are outlined in Table 6–3.

Table 6–3. Altera Logging Options (Part 1 of 2)

Name	Description	
System clock log	Purpose	Prints out a message from the system clock interrupt handler at a specified interval. This indicates that the system is still running. The default interval is every 1 second.
	Preprocessor symbol	`ALT_LOG_SYS_CLK_ON_FLAG_SETTING`
	Modifiers	The system clock log has two modifiers, providing two different ways to specify the logging interval. ■ `ALT_LOG_SYS_CLK_INTERVAL`—Specifies the logging interval in system clock ticks. The default is *<clock ticks per second>*, that is, one second. ■ `ALT_LOG_SYS_CLK_INTERVAL_MULTIPLIER`—Specifies the logging interval in seconds. The default is 1. When you modify `ALT_LOG_SYS_CLK_INTERVAL_MULTIPLIER`, `ALT_LOG_SYS_CLK_INTERVAL` is recalculated.

	Sample Output	`System Clock On 0`
		`System Clock On 1`

Write echo	Purpose	Every time `alt_write()` is called (normally, whenever characters are sent to `stdout`), the first *<n>* characters are echoed to a logging message. The message starts with the string `"Write Echo:"`. *<n>* is specified with `ALT_LOG_WRITE_ECHO_LEN`. The default is 15 characters.
	Preprocessor symbol	`ALT_LOG_WRITE_ON_FLAG_SETTING`
	Modifiers	`ALT_LOG_WRITE_ECHO_LEN`—Number of characters to echo. Default is 15.
	Sample Output	`Write Echo: Hello from Nio`
JTAG startup log	Purpose	At JTAG UART driver initialization, print out a line with the number of characters in the software transmit buffer followed by the JTAG UART control register contents. The number of characters, prefaced by the string `"SW CirBuf"`, might be negative, because it is computed as (*<tail_pointer>* – *<head_pointer>*) on a circular buffer. For more information about the JTAG UART control register fields, refer to the *Off-Chip Interface Peripherals* section in the *Embedded Peripherals IP User Guide*.
	Preprocessor symbol	`ALT_LOG_JTAG_UART_STARTUP_INFO_ON_FLAG_SETTING`
	Modifiers	None
	Sample Output	`JTAG Startup Info: SW CirBuf = 0, HW FIFO wspace=64 AC=0 WI=0 RI=0 WE=0 RE=1`

Table 6–3. Altera Logging Options (Part 2 of 2)

Name	Description	
JTAG interval log	Purpose	Creates an alarm object to print out the same JTAG UART information as the JTAG startup log, but at a repeated interval. Default interval is 0.1 second, or 10 messages a second.
	Preprocessor symbol	`ALT_LOG_JTAG_UART_ALARM_ON_FLAG_SETTING`
	Modifiers	The JTAG interval log has two modifiers, providing two different ways to specify the logging interval. ■ `ALT_LOG_JTAG_UART_TICKS`—Logging interval in ticks. Default is *<ticks_per_second>* / 10. ■ `ALT_LOG_JTAG_UART_TICKS_DIVISOR`—Specifies the number of logs per second. The default is 10. When you modify `ALT_LOG_JTAG_UART_TICKS_DIVISOR`, `ALT_LOG_JTAG_UART_TICKS` is recalculated.
	Sample Output	`JTAG Alarm: SW CirBuf = 0, HW FIFO wspace=45 AC=0 WI=0 RI=0 WE=0 RE=1`
JTAG interrupt service routine (ISR) log	Purpose	Prints out a message every time the JTAG UART near-empty interrupt triggers. Message contains the same JTAG UART information as in the JTAG startup log.
	Preprocessor symbol	`ALT_LOG_JTAG_UART_ISR_ON_FLAG_SETTING`
	Modifiers	None
	Sample Output	`JTAG IRQ: SW CirBuf = -20, HW FIFO wspace=64 AC=0 WI=1 RI=0 WE=1 RE=1`
Boot log	Purpose	Prints out messages tracing the software boot process. The boot log is turned on by default when Altera logging is enabled.
	Preprocessor symbol	`ALT_LOG_BOOT_ON_FLAG_SETTING`
	Modifiers	None
	Sample Output	Refer to “Enabling Altera Logging” on page 6–10.

Setting a preprocessor flag to 1 enables the corresponding option. Any value other than 1 disables the option.

Several options have modifiers, which are additional preprocessor symbols controlling details of how the options work. For example, the system clock log’s modifiers control the logging interval. Option modifiers are also listed in Table 6–3. An option’s modifiers are meaningful only when the option is enabled.

Logging Levels

An additional preprocessor symbol, ALT_LOG_FLAGS, can be set to provide some grouping for the extra logging options. ALT_LOG_FLAGS implements logging levels based on performance impact. With higher logging levels, the Altera logging options take more processor time. ALT_LOG_FLAGS levels are defined in Table 6–4.

Table 6–4. Altera Logging Levels

Logging Level	Logging
0	Boot log (default)
1	Level 0 plus system clock log and JTAG startup log
2	Level 1 plus JTAG interval log and write echo
3	Level 2 plus JTAG ISR log
-1	Silent mode—No Altera logging

Note to Table 6–4:

(1) You can use logging level -1 to turn off logging without changing the program footprint. The logging code is still present in your executable image, as determined by other logging options chosen. This is useful when you wish to switch the log output on or off without disturbing the memory map.

Because each logging option is controlled by an independent preprocessor symbol, individual options in the logging levels can be overridden.

Example: Creating a BSP with Logging

Example 6–6 creates a HAL BSP with Altera logging enabled and the following options in addition to the default boot log:

- System clock log
- JTAG startup log
- JTAG interval log, logging twice a second
- No write echo

Example 6–6. BSP With Logging

```
nios2-bsp hal my_bsp ../my_hardware.sopcinfo \
  --set hal.log_port uart1 \
  --set hal.make.bsp_cflags_user_flags \
  -DALT_LOG_FLAGS=2 \
  -DALT_LOG_WRITE_ON_FLAG_SETTING=0 \
  -DALT_LOG_JTAG_UART_TICKS_DIVISOR=2↵
```

The -DALT_LOG_FLAGS=2 argument adds -DALT_LOG_FLAGS=2 to the ALT_CPP_FLAGS make variable in **public.mk**.

Custom Logging Messages

You can add custom messages that are sent to the Altera logging device. To define a custom message, include the header file **alt_log_printf.h** in your C source file as follows:

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

Then use the following macro function:

```
ALT_LOG_PRINTF(const char *format, ...)
```

This C preprocessor macro is a pared-down version of `printf()`. The `format` argument supports most `printf()` options. It supports `%c`, `%d %I %o %s %u %x`, and `%X`, as well as some precision and spacing modifiers, such as `%-9.3o`. It does not support floating point formats, such as `%f` or `%g`. This function is not compiled if Altera logging is not enabled.

If you want your custom logging message be controlled by Altera logging preprocessor options, use the appropriate Altera logging option preprocessor flags from Table 6–4, or Table 6–3 on page 6–11. Example 6–7 illustrates two ways to implement logging options with custom logging messages.

Example 6–7. Using Preprocessor Flags

```
/* The following example prints "Level 2 logging message" if
   logging is set to level 2 or higher */
#if ( ALT_LOG_FLAGS >= 2 )
    ALT_LOG_PRINTF ( "Level 2 logging message" );
#endif

/* The following example prints "Boot logging message" if boot logging
   is turned on */
#if ( ALT_LOG_BOOT_ON_FLAG_SETTING == 1)
    ALT_LOG_PRINTF ( "Boot logging message" );
#endif
```

Altera Logging Files

Table 6–5 lists HAL source files which implement Altera logging functions.

Table 6–5. HAL Implementation Files for Altera Logging

Location *(1)*	File Name
components/altera_hal/HAL/inc/sys/	alt_log_printf.h
components/altera_hal/HAL/src/	alt_log_printf.c
components/altera_nios2/HAL/src/	alt_log_macro.S

Note to Table 6–5:

(1) All file locations are relative to *<Nios II EDS install path>*.

Table 6–6 lists HAL source files which use Altera logging functions. These files implement the logging options listed in table Table 6–3 on page 6–11. They also serve as examples of logging usage.

Table 6–6. HAL Example Files for Altera Logging

Location *(1)*	File Name
components/altera_avalon_jtag_uart/HAL/src/	altera_avalon_jtag_uart.c
components/altera_avalon_timer/HAL/src/	altera_avalon_timer_sc.c
components/altera_hal/HAL/src/	alt_exit.c

Note to Table 6–6:

(1) All file locations are relative to *<Nios II EDS install path>*.

Table 6–6. HAL Example Files for Altera Logging

Location *(1)*	File Name
components/altera_hal/HAL/src/	alt_main.c
components/altera_hal/HAL/src/	alt_write.c
components/altera_nios2/HAL/src/	crt0.S

Note to Table 6–6:

(1) All file locations are relative to *<Nios II EDS install path>*.

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 read-only zip 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 in a file subsystem using the C file I/O functions defined in **file.h**, such as `fopen()` and `fread()`.

For more information about the use of file I/O 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 EDS <*version*> Documentation**.

Host-Based File System

The host-based file system enables programs executing on a target board to read and write files stored on the host computer. The Nios II SBT for Eclipse transmits file data over the Altera download cable. Your program accesses the host based file system using the ANSI C standard library I/O functions, such as `fopen()` and `fread()`. The host-based file system is a software package which you add to your BSP.

The following features and restrictions apply to the host based file system:

- The host-based file system makes the Nios II C/C++ application project directory and its subdirectories available to the HAL file system on the target hardware.
- The target processor can access any file in the project directory. Be careful not to corrupt project source files.
- The host-based file system only operates while debugging a project. It cannot be used for run sessions.
- Host file data travels between host and target serially through the Altera download cable, and therefore file access time is relatively slow. Depending on your host and target system configurations, it can take several milliseconds per call to the host. For higher performance, use buffered I/O function such as `fread()` and `fwrite()`, and increase the buffer size for large files.

You configure the host-based file system using the Nios II BSP Editor. The host-based file system has one setting: the mount point, which specifies the mount point within the HAL file system. For example, if you name the mount point **/mnt/host** and the project directory on you host computer is **/software/project1**, in a HAL-based program, the following code opens the file **/software/project1/datafile.dat.**:

```
fopen("/mnt/host/datafile.dat", "r");
```

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—Supports alarms, such as you would use in a scheduler.
- Timestamp 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 about how to control BSP settings, refer to "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 clock 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 about the number of elapsed ticks. The prototypes for these functions appear in **times.h**.

For more information about 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 run simultaneously.

Alarm callback functions execute in an exception context. This imposes functional restrictions which you must observe when writing an alarm callback.

For more information about the use of these functions, refer to the *Exception Handling* chapter of the *Nios II Software Developer's Handbook*.

The code fragment in Example 6–8 demonstrates registering an alarm for a periodic callback every second.

Example 6–8. 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 is called once per 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 of the Nios II processor system—usually millions of cycles per second. The timestamp drivers are defined in the **alt_timestamp.h** header file.

For more information about 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 6–9 shows how you can use the timestamp facility to measure code execution time.

Example 6–9. 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 memory. 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 erasable programmable configurable serial (EPCS) 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 two different levels of access to the flash:

- Simple flash access—Functions that write buffers to flash and read them back at the block level. In writing, if the buffer is less than a full block, these functions erase preexisting flash data above and below the newly written data.
- Fine-grained flash access—Functions that write buffers to flash and read them back at the buffer level. In writing, if the buffer is less than a full block, these functions preserve preexisting 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 about the use of these functions, refer to the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*. You can get details about the Common Flash Interface, including the organization of common flash interface (CFI) erase regions and blocks, from JEDEC (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 6–22 shows the use 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**.

After you obtain 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`, and 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 6–10 shows the use of simple flash API functions to access a flash device named **/dev/ext_flash**, as defined in **system.h**.

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 the 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. These are discussed in the following section.

Table 6–7 on page 6–23 shows how you can cause unexpected data corruption by writing using the simple flash access functions. Table 6–7 shows the example of an 8-kilobyte (KB) flash memory comprising two 4-KB blocks. First write 5 KB of all `0xAA` to flash memory at address `0x0000`, and then write 2 KB of all `0xBB` to address `0x1400`. After the first write succeeds (at time t(2)), the flash memory contains 5 KB of `0xAA`, and the rest is empty (that is, `0xFF`). Then the second write begins, but before writing to the second block, the block is erased. At this point, t(3), the flash contains 4 KB of `0xAA` and 4 KB of `0xFF`. After the second write finishes, at time t(4), the 2 KB of `0xFF` at address `0x1000` is corrupted.

Fine-Grained Flash Access

Three additional functions provide complete control for writing flash contents at the highest granularity:

- `alt_get_flash_info()`
- `alt_erase_flash_block()`
- `alt_write_flash_block()`

By the nature of flash memory, you cannot erase a single address in a block. You must erase (that is, 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 in 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.

Example 6–10. 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
         * have read that all back to dest
         */
      }
    }
    alt_flash_close_dev(fd);
  }
  else
  {
    printf("Cannot open flash device\n");
  }
  return 0;
}
```

`alt_get_flash_info()` gets the number of erase regions, the number of erase blocks in each region, and the size of each erase block. The function prototype is as follows:

```
int alt_get_flash_info (
    alt_flash_fd*  fd,
    flash_region** info,
    int*           number_of_regions )
```

If the call is successful, on 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.

Table 6–7. 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)
			First Write		Second Write	
		Before First 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

Note to Table 6–7:

(1) Unintentionally cleared to FF during erasure for second write.

The flash_region structure is defined in **sys/alt_flash_types.h**. The data structure is defined as follows:

```
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 device.

alt_erase_flash() erases a single block in the flash memory. The function prototype is as follows:

```
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 device. 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 6–11 on page 6–24 demonstrates the use of the fine-grained flash access functions.

Example 6–11. 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 to
          * the first block of the flash
          */
         ret_code = alt_write_flash_block (
            fd,
            regions->offset,
            regions->offset+0x100,
            write_data,
            BUF_SIZE );
      }
    }
  }
  return 0;
}
```

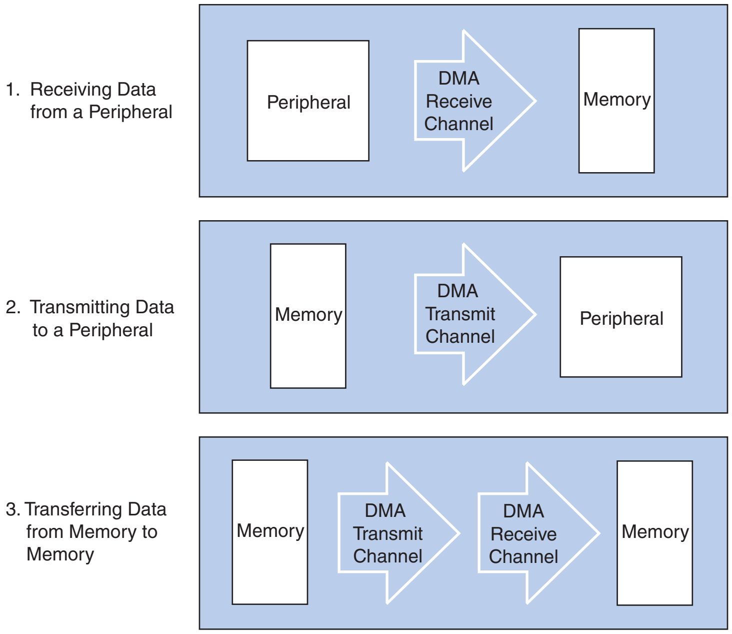
Using DMA Devices

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, there are two categories of DMA transactions: transmit and receive. 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 in 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 about 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 about 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 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 6–12 shows how to obtain a handle for a DMA transmit device `dma_0`.

Example 6–12. 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 Avalon Memory-Mapped® (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 the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

DMA Receive Channels

DMA receive channels operate similarly 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 (that is, 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 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, refer to the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

The code in Example 6–13 shows a complete example application that posts a DMA receive request, and blocks in `main()` until the transaction completes.

Example 6–13. 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 6–14 shows the process of queuing up a receive request followed by a transmit request to achieve a memory-to-memory DMA transaction.

Example 6–14. Copying Data from Memory to Memory (Part 1 of 2)

```
#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
* is 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);
  }

/* Continued... */
```

Example 6–14. Copying Data from Memory to Memory (Part 2 of 2)

```
  /* 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;
}
```

Using Interrupt Controllers

The HAL supports two types of interrupt controllers:

- The Nios II internal interrupt controller
- An external interrupt controller component

For information about working with interrupt controllers, refer to the *Exception Handling* chapter of the *Nios II Software Developer's Handbook.*

Reducing Code Footprint in Embedded Systems

Code size is always a concern for embedded systems 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.

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.

Implementing the options in the following sections entails making changes to BSP settings. For detailed information about manipulating BSP settings, refer to "HAL BSP Settings" on page 6–2.

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 through a BSP setting.

With this option turned on, the Nios II compiler compiles code with the maximum optimization available, for both size and speed.

☞ You must set this option for both the BSP and the application project.

Use Reduced Device Drivers

Some devices provide two driver variants, a fast variant and a small variant. The feature sets provided by these two variants are device specific. The fast variant is full-featured, and the small variant provides a reduced code footprint.

By default the HAL always uses the fast driver variants. You can select the reduced device driver for all hardware components, or for an individual component, through HAL BSP settings.

Table 6–8 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 6–8. 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 excluded in small footprint mode
LCD module controller	Driver excluded in small footprint mode
EPCS serial configuration device	Driver 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. You can change the size of the file descriptor pool through a BSP setting. The default is 32.

Use /dev/null

At boot time, standard input, standard output, and standard error are all directed towards the null device, that is, **/dev/null**. This direction ensures that calls to `printf()` during driver initialization do nothing and therefore are harmless. After all drivers are installed, these streams are redirected to 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 through `stdin`. You can control the assignment of `stdin`, `stdout`, and `stderr` channels by manipulating BSP settings.

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. When you use **nios2-elf-gcc** at the command line, the `-msmallc` command-line option enables the small C library.

You can select the small newlib library through BSP settings. Table 6–9 summarizes the limitations of the Nios II small newlib C library implementation.

Table 6–9. 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()`

Table 6–9. Limitations of the Nios II Small newlib C Library (Part 2 of 2)

Limitation	Functions Affected

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()`

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 functions listed in this table are not supported.	

Note to Table 6–9:

(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 about 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** > **Nios II Documentation**.

The Nios II implementation of the small newlib C library differs slightly from GCC. Table 6–9 provides details about 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. For details, refer to "UNIX-Style Interface" on page 6–5.

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 6–15 shows a simple, unbuffered implementation of `getchar()`.

Example 6–15. 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 read-only zip file system are not available with the reduced device driver API. You can select the reduced device drivers through BSP settings.

By default, the lightweight device driver API is disabled.

For further details about the lightweight device driver API, refer to the *Developing Device Drivers for the Hardware Abstraction Layer* 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 6–34.

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 1 KBKB 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 about 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 development 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.

You can selectively omit any individual driver, select a specific driver version, or substitute your own driver.

For further information about controlling driver configurations, refer to 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 to use the free-standing environment. For details, refer to "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 on 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 select the `_exit()` function through a BSP setting.

Eliminate All Exit Code

Many embedded systems never exit at all. In such cases, exit code is unnecessary. You can eliminate all exit code through a BSP setting.

☞ If you enable this option, ensure that 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 disable C++ support through a BSP setting.

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 Versus 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 need not 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 initializes any needed hardware explicitly. The `alt_main()` function provides a free-standing environment, giving you complete control over the initialization of the system. The free-standing environment places on the programmer the responsibility to initialize any system features 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 free-standing 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, use the suggestions described in "Reducing Code Footprint in Embedded Systems" on page 6–30. It is easier to reduce the HAL BSP footprint by using BSP settings, than to use the free-standing mode.

The Nios II EDS provides examples of both free-standing and hosted programs.

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.
- Initializes the block started by symbol (BSS) region to zeroes 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`. Refer to "Global Pointer Register" on page 6–43.
- Calls `alt_main()`.

The HAL provides a default implementation of the `alt_main()` function, which performs the following steps:

- Calls the `alt_irq_init()` function, located in **alt_sys_init.c.** `alt_irq_init()` **initializes the hardware interrupt controller.** The Nios II development flow creates the file **alt_sys_init.c** for each HAL BSP.
- Calls `ALT_OS_INIT()` to perform any necessary operating system specific initialization. For a system that does not include an operating system (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.
- Enables interrupts.
- Calls the `alt_sys_init()` function, also located in **alt_sys_init.c.** `alt_sys_init()` initializes all device drivers and software packages in the system.
- 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. The SBT copies **alt_main.c** to 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 allows you 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 following line of code is the prototype for `alt_main()`:

```
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.

alt_sys_init.c is a generated file, which you must not modify. However, the Nios II SBT enables 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 and how to control the contents of **alt_sys_init.c**, refer to 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 Hardware Abstraction Layer* 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 how the HAL uses memory and arranges code, data, stack, and other logical memory sections, in physical memory.

Memory Sections

By default, HAL-based systems are linked using a generated linker script that is created by the Nios II SBT. This linker script controls the mapping of code and data to the available memory sections. The autogenerated 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 a memory component named `sdram` is defined in the **system.h** file, there is a memory section named `.sdram`. Figure 6–3 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.

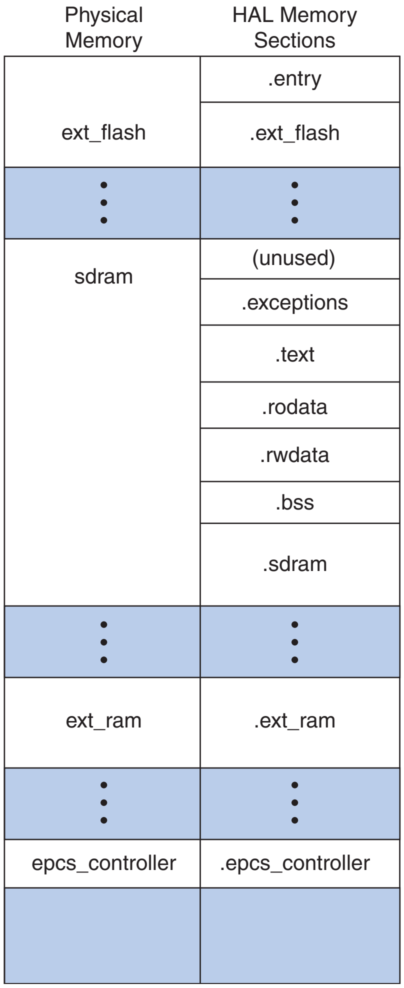
In a memory device containing the reset or exception address, the linker creates a normal (nonreserved) memory section above the `.entry` or `.exceptions` section. If there is a region of memory below the `.entry` or `.exceptions` section, it is unavailable to the Nios II software. Figure 6–3 illustrates an unavailable memory region below the `.exceptions` section.

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 development flow 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 (that is, 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 must specify manually which code belongs in which section.

Figure 6–3. Sample HAL Link Map



Simple Placement Options

The reset handler code is always placed at the base of the .reset partition. The general exception funnel code is always the first code in 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 about how to control BSP settings, refer to "HAL BSP Settings" on page 6–2.

The Nios II BSP Editor is a very convenient way to manipulate the linker's memory map. The BSP Editor displays memory section and region assignments graphically, allowing you to see overlapping or unused sections of memory. The BSP Editor is available either through the Nios II SBT for Eclipse, or at the command line of the Nios II SBT.

For details, refer to the *Getting Started from the Command Line* chapter of the *Nios II Software Developer's Handbook*.

Advanced Placement Options

In 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 6–16 places a variable foo in the memory named ext_ram, and the function bar() in the memory named sdram.

Example 6–16. 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 about the use 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 EDS documentation launchpad, scroll down to **Software Development,** and click **Using the GNU Compiler Collection (GCC).**

A powerful way to manipulate the linker memory map is by using the Nios II BSP Editor. With the BSP Editor, you can assign linker sections to specific physical regions, and then review a graphical representation of memory showing unused or overlapping regions. You start the BSP Editor from the Nios II Command Shell. For details about using the BSP Editor, refer to the editor's tool tips.

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 in 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_BYTES=1048576` sets the heap size limit to 0x100000. You can specify this command-line option through a BSP setting. For more information about manipulating BSP settings, refer to "HAL BSP Settings" on page 6–2.

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.

Refer to the *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook* for details about selecting stack and heap placement, and setting up stack checking.

For details about how to control BSP settings, refer to "HAL BSP Settings" on page 6–2.

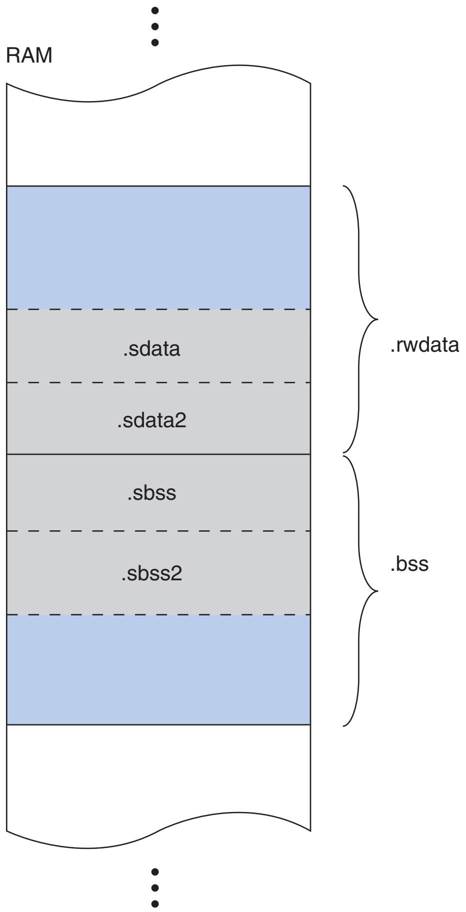
Global Pointer Register

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 64 KB. 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 8 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, 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 is more than 64 KB, these data structures 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 4 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 through a project setting. For information about manipulating project settings, refer to "HAL BSP Settings" on page 6–2.

☞ 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 in it. The HAL provides a small boot loader program that 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. This boot loader 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 in the HAL executable program 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 program 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 about `alt_load_section()`, refer to the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

Working with HAL Source Files

You might wish to view files in the HAL, especially header files, for reference. This section describes how to find and use HAL source files.

Finding HAL Files

You determine the location of HAL source files when you create the BSP. HAL source files (and other BSP files) are copied to the BSP directory.

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 to your BSP directory when you create your BSP. If you regenerate a BSP, any HAL source files that differ from the installation files are copied. Avoid modifying BSP files. To override default HAL code, use BSP settings, or custom device drivers or software packages.

For information about what happens when you regenerate a BSP, refer to "Revising your BSP" in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook.*

Avoid modifying HAL source files. If you modify a HAL source file, you cannot regenerate the BSP without losing your changes. This makes it difficult to keep the BSP coordinated with changes to the underlying hardware system.

For more information, refer to "Nios II Embedded Software Projects" in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

Document Revision History

Table 6–10 shows the revision history for this document.

Table 6–10. Document Revision History (Part 1 of 2)

Date	Version	Changes
January 2014	13.1.0	Removed "Nios II Development Flows" section.
May 2011	11.0.0	Introduction of Qsys system integration tool
February 2011	10.1.0	Removed "Referenced Documents" section.
July 2010	10.0.0	Maintenance release.
November 2009	9.1.0	■ Introduced external interrupt controller. ■ BSP generation file-copy behavior changed. ■ Described `alt_irq_init()` function. ■ Inserted host-based file system description. ■ Removed IDE-specific information. ■ Updated information about overriding HAL functions.
March 2009	9.0.0	■ Reorganized and updated information and terminology to clarify role of Nios II Software Build Tools. ■ Add documentation for Altera logging. ■ Corrected minor typographical errors.
May 2008	8.0.0	Maintenance release.
October 2007	7.2.0	■ Added documentation for HAL program development with the Nios II Software Build Tools. ■ Additional documentation of alarms functions. ■ Correct `alt_erase_flash_block()` example.

Table 6–10. Document Revision History (Part 2 of 2)

Date	Version	Changes
May 2007	7.1.0	■ Added table of contents to “Introduction” section. ■ Added Referenced Documents section.
March 2007	7.0.0	Maintenance release.
November 2006	6.1.0	■ **Program never exits** system library option. ■ **Support C++** system library option. ■ **Lightweight device driver API** system library option. ■ Minimal character-mode API.
May 2006	6.0.0	■ Revised text on instruction emulation. ■ Added section on global pointers.
October 2005	5.1.0	■ Added `alt_64` and `alt_u64 types` to Table 6–1 on page 6–5. ■ Made changes to section “Placement of the Heap and Stack”.
May 2005	5.0.0	Added `alt_load_section()` function information.
December 2004	1.2	■ Added boot modes information. ■ Amended compiler optimizations. ■ Updated Reducing Code Footprint section.
September 2004	1.1	Corrected DMA receive channels example code.
May 2004	1.0	Initial release.

7. Developing Device Drivers for the Hardware Abstraction Layer

NII52005-11.0.0

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 also describes how to develop software packages for use with HAL board support packages (BSPs). The process of integrating a software package with the HAL is nearly identical with the process for integrating a device driver.

This chapter contains the following sections:

Confine direct interaction with the hardware to device driver code. In general, the best practice is to keep most of your program code free of low-level access to the hardware. Wherever possible, use the high-level HAL application program 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 with the HAL framework at one of the following two levels:

- Integration in the HAL API
- Peripheral-specific API

☞ As an alternative to creating a driver, you can compile the device-specific code as a user library, 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.

Driver Integration in the HAL API

Integration in 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 in the HAL API, you write device access functions as specified in this chapter, and the device becomes accessible to software through 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.

The HAL 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. In this case, 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 in the HAL API is higher, but you gain the benefit of the HAL and C standard library API to manipulate devices.

For details about integration in the HAL API, refer to "Integrating a Device Driver in the HAL" on page 7–18.

All the other sections in this chapter apply to integrating drivers in the HAL API and creating drivers with a peripheral-specific API.

Although C++ is supported for programs based on the HAL, HAL drivers can not be written in C++. Restrict your driver code to either C or assembly language. C is preferred for portability.

Preparing for HAL Driver Development

This chapter assumes that you are familiar with C programming for the HAL.

Refer to the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook* for information you need before reading this chapter.

This chapter uses the variable <*Altera installation*> to represent the location where the Altera® Complete Design Suite is installed. On a Windows system, by default, that location is **c:/altera/**<*version number*>.

Development Flow for Creating Device Drivers

The steps to develop a new driver for the HAL depend 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 in the HAL environment.
5. Integrate the device driver in the HAL framework.

Nios II Hardware Design Concepts

This section discusses some basic concepts behind the Altera Qsys and SOPC Builder system integration tools. These concepts can enhance your understanding of the driver development process. You do not normally need to use a system integration tool when developing Nios II device drivers.

The Relationship Between the .sopcinfo File and system.h

The **system.h** header file provides a complete software description of the Nios II system hardware. The **system.h** system description is a fundamental part of developing drivers. Because drivers interact with hardware at the lowest level, it is worth understanding the relationship between the **.sopcinfo** file and **system.h**.

The system generation tool, Qsys or SOPC Builder, generates the Nios II processor system hardware. Hardware designers use the system generation tool 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 the system generation tool. These design choices are encapsulated in the **.sopcinfo** file. **system.h** is derived from the **.sopcinfo** file.

For more information about the **system.h** header file, refer to the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

Using the System Generation Tool to Optimize Hardware

If you find less-than-optimal definitions in **system.h**, remember that you can modify the contents of **system.h** by changing the underlying hardware with the system generation tool, Qsys or SOPC Builder. Before you write a device driver to accommodate imperfect hardware, it is worth considering whether the hardware can be improved easily with the system generation tool.

Components, Devices, and Peripherals

The Qsys and SOPC Builder system generation tools use the term “component” to describe hardware modules included in the system. In the context of Nios II software development, 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 the system generation tool.

Accessing Hardware

Software accesses the hardware with macros that abstract the memory-mapped interface to the device. This section describes the macros that define the hardware interface for each device.

All 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/altera/sopc_builder_ip** directory. Many components provide a header file that defines their hardware interface. The header file is named <*component name*>**_regs.h**, 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/altera/sopc_builder_ip/altera_avalon_jtag_uart/inc/altera_avalon_jtag_uart_regs.h**.

The **_regs.h** header file defines the following access macros for the component:

- Register access macros that provide a read and/or write macro for each register in 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 in 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 in 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.

Access a device's registers only with the macros defined in the **_regs.h** file. 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 must prepare the **_regs.h** header file.

For detailed information about developing device drivers for HAL BSPs, refer to *AN 459: Guidelines for Developing a Nios II HAL Device Driver*. For a complete example of the **_regs.h** file, refer to the component directory for any of the Altera-supplied components, such as <*Altera installation*>**/ip/sopc_builder_ip/altera_avalon_jtag_uart/inc**. For more information about the effects of cache management and device access, refer to the *Cache and Tightly-Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.

Creating Embedded Drivers for HAL Device Classes

The HAL supports a number of generic device model classes. 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.

Generic device model classes are defined in the *Overview of the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

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 in 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 code in Example 7–1 defines the `alt_dev` structure.

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.

Example 7–1. 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;
```

For more information about open(), close(), read(), write(), lseek(), fstat(), and ioctl(), refer to the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

None of these functions directly modifies 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 on 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, ensure that 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 in 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 must always be set to the value ALT_LLIST_ENTRY. name is the location of the device in the HAL file system and is the name of the device as defined in **system.h**.

Register a Character Device

After you create 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. The return value is zero upon success. A negative return value indicates that the device cannot be registered.

After a device is registered with the HAL file system, you can access it through 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 Hardware Abstraction Layer* 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 in 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 (refer to "Character-Mode Device Drivers" on page 7–5). 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 cannot be registered.

After a file subsystem is registered with the HAL file system, you can access it through 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 Hardware Abstraction Layer* 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 clock 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 exception 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 on success, and nonzero otherwise.

For more information about writing interrupt service routines, refer to 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 about using these functions, refer to the *Developing Programs Using the Hardware Abstraction Layer* 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 must always be set to the value ALT_LLIST_ENTRY. `name` is the location of the device in 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 the following functions:

- `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 about the format of the `flash_region` structure, refer to "Using Flash Devices" in the *Developing Programs Using the Hardware Abstraction Layer* 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. For all other flash devices, 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. The return value is zero upon success. A negative return value indicates that the device cannot 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 "Using DMA Devices" the *Developing Programs Using the Hardware Abstraction Layer* 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, shown in Example 7–2.

Example 7–2. 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.

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)
```

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, refer to the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.
`ioctl`	This function provides device specific I/O control. Refer to **sys/alt_dma_dev.h** for a list of the generic options that you might want your device to support.

The input argument `dev` is the device to register. The return value is zero on 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, shown in Example 7–3.

Example 7–3. 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.

The `prepare()` function must 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 on success, or negative if the device cannot be registered.

Table 7–3. Fields in the alt_dma_rxchan Structure

Field	Function
`llist`	This function 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_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, refer to 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. Refer to **sys/alt_dma_dev.h** for a list of the generic options that a device might wish to support.

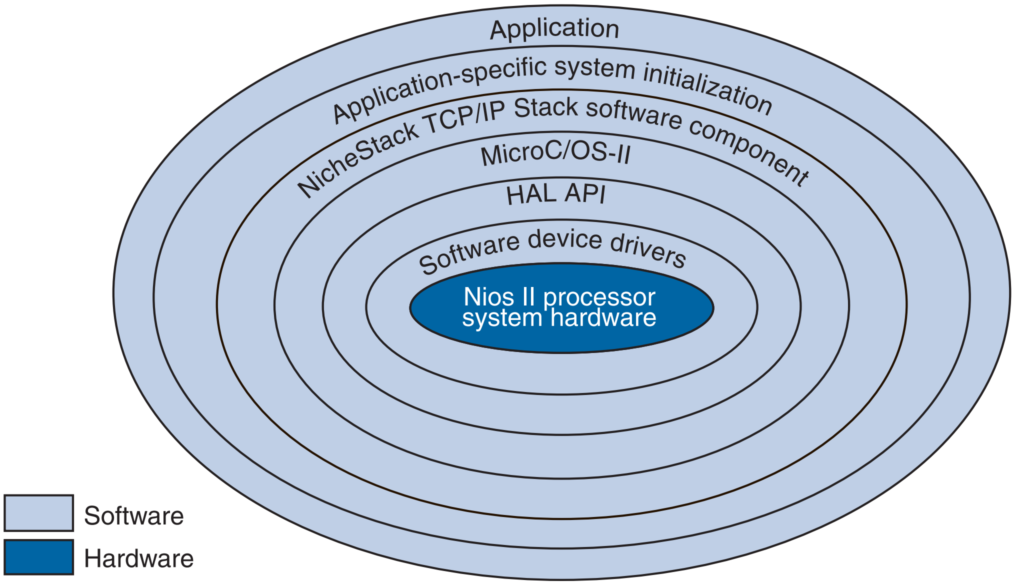
Ethernet Device Drivers

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.

The onion diagram in Figure 7–1 shows the architectural layers of a Nios II MicroC/OS-II software application.

Figure 7–1. Layered Software Model


Application
Application-specific system initialization
NicheStack TCP/IP Stack software component
MicroC/OS-II
HAL API
Software device drivers
Nios II processor system hardware
Software
Hardware

Each layer encapsulates the specific implementation details of that layer, abstracting the data for the next outer layer. However, the hierarchy of layers is not absolute. For example, the application makes system calls directly to the MicroC/OS-II or HAL API layers for services that do not require networking.

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/altera/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 Embedded Design Suite (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 about the NicheStack TCP/IP Stack implementation, refer to the *NicheStack Technical Reference Manual*, available on the Literature: Nios II Processor page of the Altera website.

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 functions in the SMSC lan91c111 device driver are shown in Table 7–4.

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 Internet protocol (IP) layer 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

To enable the HAL to 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, refer to `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 "Header Files and alt_sys_init.c" on page 7–16. 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. Refer to "Header Files and alt_sys_init.c" on page 7–16 for further detail about the `*INSTANCE` and `*INIT` macros.

Provide a Software Initialization Function

The `*INSTANCE()` macro inserts a pointer to your initialization function in the `alt_iniche_dev` structure, as described in "Provide *INSTANCE and *INIT Macros" on page 7–15. Your software initialization function must perform at least the following three tasks:

- 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, refer to 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, refer to `alt_avalon_lan91c111_init()` in <*SMSC path*>**/src/iniche/smsc91x.c**.

Creating a Custom Device Driver for the HAL

This section describes how to provide appropriate files to integrate your device driver in the HAL. The "Integrating a Device Driver in the HAL" section on page 7–18 describes the correct locations for the files.

Header Files and alt_sys_init.c

At the heart of the HAL is the autogenerated source file, **alt_sys_init.c**. This file 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 device drivers software packages, and make them available to the program.

When you create the driver or software package, you specify in a Tcl script whether you want the `alt_sys_init()` function to invoke your `INSTANCE` and `INIT` macros. Refer to "Enabling Software Initialization" on page 7–25 for details.

Example 7–4 shows excerpts from an **alt_sys_init.c** file.

☞ The remainder of this section assumes that you are using the `alt_sys_init()` HAL initialization mechanism.

The Software Build Tools (SBT) creates **alt_sys_init.c** based on the header files associated with each device driver and software package. For a device driver, the header file must define the macros *`<component name>_INSTANCE`* and *`<component name>_INIT`*.

Like a device driver, a software package provides an INSTANCE macro, which alt_sys_init() invokes once. A software package header file can optionally provide an INIT macro.

Example 7–4. 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 );

/*
* Initialize the devices
*/
void alt_sys_init( void )
{
    ALTERA_AVALON_UART_INIT( UART1, uart1 );
    ALTERA_AVALON_TIMER_INIT( SYSCLK, sysclk );
}
```

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 required static memory allocation. For drivers, *_INSTANCE is invoked once per device instance, so that memory can be initialized on a per-device basis. For software packages, *_INSTANCE is invoked once.
- The *_INIT macro performs runtime initialization of the device driver or software package.

In the case of a device driver, 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 at system generation time.

You can use these input parameters to extract device-specific configuration information from the **system.h** file.

The name of the header file must be as follows:

- Device driver: <*hardware component class*>**.h**. For example, if your driver targets the **altera_avalon_uart** component, the file name is **altera_avalon_uart.h**.
- Software packages <*package name*>**.h**. For example, if you create the software package with the following command:

  ```
  create_sw_package my_sw_package
  ```

  the header file is called **my_sw_package.h**.

For a complete example, refer to 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**.

Device Driver Source Code

In addition to the header file, the component driver might need to provide compilable source code, to be incorporated in the BSP. This source code is specific to the hardware component, and resides in one or more C files (or assembly language files).

Integrating a Device Driver in the HAL

The Nios II SBT can incorporate device drivers and software packages supplied by Altera, supplied by other third-party developers, or created by you. This section describes how to prepare device drivers and software packages so the BSP generator recognizes and adds them to a generated BSP.

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.

☞ The process required to integrate a device driver is nearly identical to that required to develop a software package. The following sections describe the process for both. Certain steps are not needed for software packages, as noted in the text.

Overview

To publish a device driver or a software package, you provide the following items:

- A header file defining the package or driver interface
- A Tcl script specifying how to add the package or driver to a BSP

The header file and Tcl script are described in the following sections.

Assumptions and Requirements

This section assumes that you are developing a device driver or software package for eventual incorporation in a BSP. The driver or package is to be incorporated in 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 must rely on the driver or package settings that you create with the tools described in this section.

For a device driver or software package to work with the Nios II SBT, it must meet the following criteria:

- It must have a defining Tcl script. The Tcl script for each driver or software package provides the Nios II SBT 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
  - Version and compatibility information—The driver or package version, and (for drivers) information about what device core versions it supports.
  - BSP type(s)—The supported operating system(s)
  - Settings—The 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 included in the `SOPC_BUILDER_PATH` environment variable, or in any directory located one level beneath such a directory. This approach is recommended if your driver or software packages are installed in a distribution you create.
  - In a directory named **ip**, 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 certain conventions, described in "File Names and Locations" on page 7–21.
- If your driver or software package uses the HAL autoinitialization mechanism (`alt_sys_init()`), certain macros must be defined in a header file. For details about this header file, refer to "Header Files and alt_sys_init.c" on page 7–16.

For details about integrating a HAL device driver, refer to *AN 459: Guidelines for Developing a Nios II HAL Device Driver*. For details of the commands you can use in a driver Tcl script, refer to the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

The Nios II BSP Generator

This section describes the process by which the Nios II BSP generator adds device drivers and software packages to your BSP. The Nios II BSP generator, a subset of the Nios II SBT, is a combination of command utilities and scripts that enable you to create and manage BSPs and their settings.

For an overview of the Nios II SBT, refer to the *Overview* and *Getting Started from the Command Line* chapters of the *Nios II Software Developer's Handbook*.

Component Discovery

When you run 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.

☞ For run-time efficiency, the BSP generator only looks at driver files that conform to the criteria listed in this section.

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

Device Driver Versions

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, refer to the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Device Driver and Software Package Inclusion

The BSP generator adds software packages to the BSP if they are specifically requested during BSP generation, with the `enable_sw_package` command.

For further details, refer to "Software Build Tools Tcl Commands" 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, 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.

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

With the Tcl software definition files in place, the SBT reads in the Tcl file and populate the makefiles and other support files accordingly.

When the Nios II SBT adds each driver or software package to the system, it uses 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.

When you create a new software project, the Nios II SBT generates the contents of **alt_sys_init.c** to match the specific hardware contents of the system.

File Names and Locations

As described in "The Nios II BSP Generator" on page 7–20, the Nios II build tools find a device driver or software package by locating a Tcl script with the file name ending in **_sw.tcl**, and sourcing it.

Each peripheral in a Nios II system is associated with a specific component directory. This directory contains a file defining the software interface to the peripheral. Refer to "Accessing Hardware" on page 7–3.

To enable the SBT to find your component device driver, place the Tcl script in a directory named **ip** under your hardware project directory.

Figure 7–2 illustrates a file hierarchy suitable for the Nios II SBT. This file hierarchy is located in the <*Altera installation*>**/ip/altera/sopc_builder_ip** directory. This example assumes a device driver supporting a hardware component named `custom_component`.

Source Code Discovery

You use Tcl scripts to specify the location of driver source files. For further details, refer to "The Nios II BSP Generator" on page 7–20.

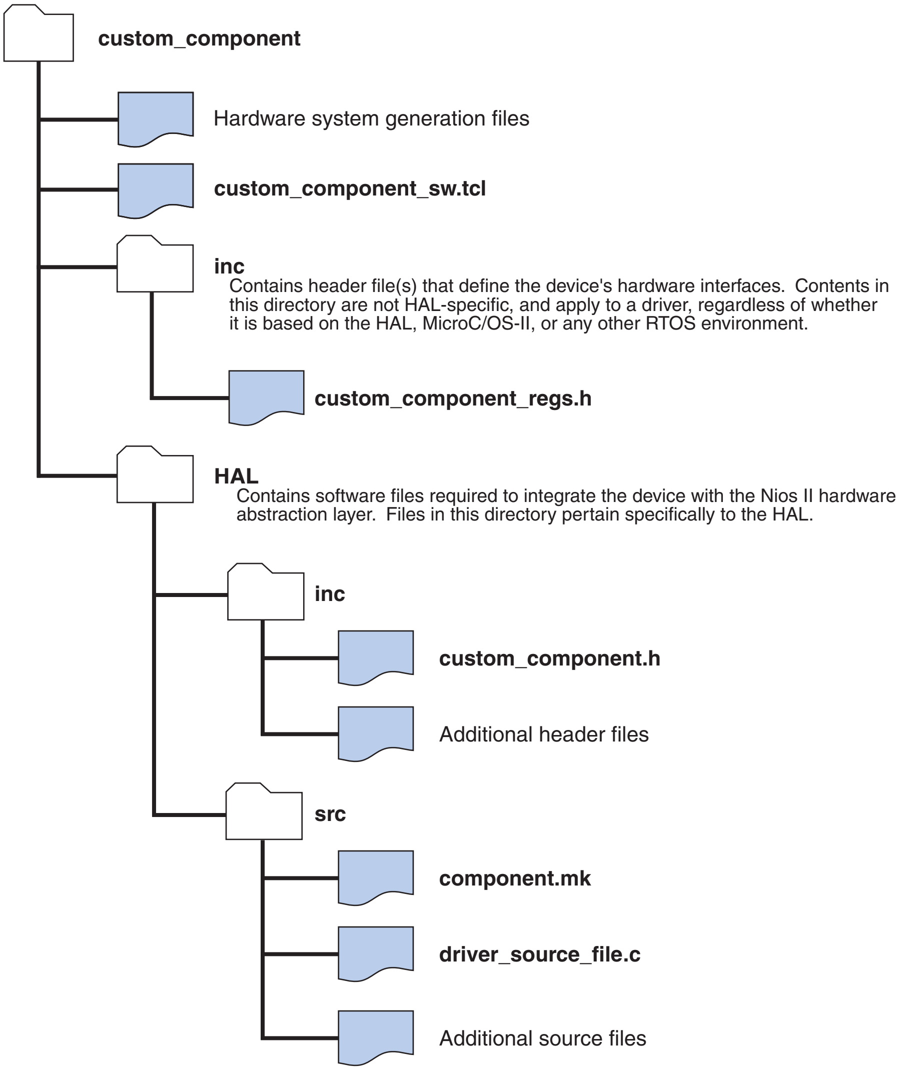
Driver and Software Package Tcl Script Creation

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 need only include a few commands. For more complex software, the Nios II SBT provides powerful features that give the BSP end user control of your software or driver's operation.

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

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 the 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.

Figure 7–2. Example Device Driver File Hierarchy and Naming


custom_component
Hardware system generation files
custom_component_sw.tcl
inc
Contains header file(s) that define 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.
custom_component_regs.h
HAL
Contains software files required to integrate the device with the Nios II hardware abstraction layer. Files in this directory pertain specifically to the HAL.
inc
custom_component.h
Additional header files
src
component.mk
driver_source_file.c
Additional source files

Tcl Command Walkthrough for a Typical Driver or Software Package

The Tcl script excerpts in this section describe a typical device driver or software package.

The example in this section creates a device driver for a hardware peripheral whose 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 as in the example in Figure 7–2.

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. The following example illustrates this convention.

```
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 the set_sw_property command, refer to 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 SBT uses your driver only if you specify it explicitly.

For further details, refer to 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 SBT finds multiple compatible drivers or software packages with the same name, it 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 SBT.

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 operating system (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 language source file (**.s** or **.S**) 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 to BSPs that incorporate the driver, using the path information specified, and adds them to source file list in the generated BSP makefile. When you build 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 to 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 to 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.

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 details, refer to "Provide *INSTANCE and *INIT Macros" on page 7–15.

To support this functionality in Nios II BSPs, you must set the set_sw_property command's auto_initialize argument to true using the following Tcl command:

```
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
```

☞ 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.

Additional include paths are added to the preprocessor flags in the BSP **public.mk** file. These preprocessor flags allow BSP source files, as well as application and user 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 with 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 in which 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 are `8.0`, `5.1.1`, `6.1`, and `6.1sp1`. The . character is 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, as follows:

```
set_sw_property min_compatible_hw_version 5.0.1add_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:

- Settings are added either to the BSP **system.h** or **public.mk**, or to the BSP makefile as a variable.
- 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. To specify the details, `add_sw_setting` requires each of the following arguments, in the order shown:

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 a 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 7–5.

Table 7–5. Data Type Settings (Part 1 of 2)

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 (')

Table 7–5. Data Type Settings (Part 2 of 2)

Data Type	Setting Value	Notes
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 the file to which the setting is saved in the BSP. The BSP generator formats the setting's assigned value based on the definition file and type of setting. Table 7–6 shows possible values of the `destination` argument.

Table 7–6. 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 build 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, refer to 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 namespace.

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, suppose your software creates a buffer as follows:

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

You can 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, refer to 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 7–5 implements a setting for a driver that has two variants of a function, one implementing a small driver (minimal code footprint) and the other a fast driver (efficient execution).

Example 7–5. 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 7–5, 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 incorporated in the object file. When a user disables the setting, `MY_CUSTOM_DRIVER_SMALL` is not defined.

You add the `MY_CUSTOM_DRIVER_SMALL` setting to your driver 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.

Each argument has several variants. For detailed usage and restrictions, refer to the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Reducing Code Footprint in HAL Embedded Drivers

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 a 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 with 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 7–7. 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 7–7.

Table 7–7. 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 SBT creates in **system.h**, generate calls to your driver functions. For example, with lightweight drivers 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 with 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, for example `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 call those functions directly from your application.

HAL Namespace Allocation

To avoid conflicting names for symbols defined by devices in the hardware 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 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 file names for device driver source and header files.

Overriding the HAL Default Device Drivers

All components can elect to provide a HAL device driver. Refer to "Integrating a Device Driver in 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.

In the Nios II SBT for Eclipse, you can use the BSP Editor to specify a custom driver.

For information about selecting device drivers, refer to "Using the BSP Editor" in the *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook*

On the command line, you specify a custom driver with the following BSP Tcl command:

```
set_driver <driver name> <component name>
```

For example, if you are using the **nios2-bsp** command, you replace the default driver for `uart0` with a driver called `custom_driver` as follows:

```
nios2-bsp hal my_bsp --cmd set_driver custom_driver uart0↵
```

Document Revision History

Table 7–8 shows the revision history for this document.

Table 7–8. Document Revision History (Part 1 of 2)

Date	Version	Changes
May 2011	11.0.0	■ Introduction of Qsys system integration tool ■ Added figure illustrating NicheStack implementation layers
February 2011	10.1.0	Removed "Referenced Documents" section.
July 2010	10.0.0	Maintenance release.
November 2009	9.1.0	■ Introduced the Nios II Software Build Tools for Eclipse™. ■ Removed Nios II IDE-specific information.
March 2009	9.0.0	■ Reorganized and updated information and terminology to clarify role of Nios II Software Build Tools. ■ Incorporated information about Tcl-based device drivers and software packages, formerly in *Using the Nios II Software Build Tools*. ■ Described use of the `INSTANCE` macro in software packages. ■ Corrected minor typographical errors.
May 2008	8.0.0	Maintenance release.
October 2007	7.2.0	Added documentation for HAL device driver development with the Nios II Software Build Tools.
May 2007	7.1.0	■ Added table of contents to "Introduction" section. ■ Added Referenced Documents section.

Table 7–8. Document Revision History (Part 2 of 2)

Date	Version	Changes
March 2007	7.0.0	Maintenance release.
November 2006	6.1.0	■ Add section “Reducing Code Footprint in HAL Embedded Drivers”. ■ Replace lwIP driver section with NicheStack TCP/IP Stack driver section.
May 2006	6.0.0	Maintenance release.
October 2005	5.1.0	Added IOADDR_* macro details to section “Accessing Hardware”.
May 2005	5.0.0	Updated reference to version of lwIP from 0.7.2 to 1.1.0.
December 2004	1.1	Updated reference to version of lwIP from 0.6.3 to 0.7.2.
May 2004	1.0	Initial release.

Section III. Advanced Programming Topics

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

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

8. Exception Handling

NII52006-11.0.0

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

This chapter contains the following sections:

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

Nios II Exception Handling Overview

The Nios II processor provides the following exception types:

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

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

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

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

Exception Handling Terminology

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

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

- Invalid instruction—An instruction that is not defined for any implementation of the Nios II processor.
- Maskable exceptions—Exceptions that can be disabled with the `status.PIE` flag, including internal hardware interrupts, maskable external hardware interrupts, and software exceptions, but not including nonmaskable external interrupts.
- Maximum disabled time—The maximum amount of continuous time that the system spends with maskable exceptions disabled.
- Maximum masked time—The maximum amount of continuous time that the system spends with a single interrupt masked.
- Miscellaneous exception—A software exception which is neither an unimplemented instruction nor a `trap` instruction. For further information, refer to "Miscellaneous Exceptions" on page 8–32.
- Nested interrupts—See pre-emption.
- Pre-emption—The process of a high-priority interrupt taking control when a lower-priority ISR is already running. Also: nested interrupts.
- Software exception—An exception caused by a software condition; that is, any exception other than a hardware interrupt. This includes unimplemented instructions and `trap` instructions.
- Unimplemented instruction—An implementation-dependent instruction that is not supported on the particular Nios II core implementation that is in your system. For example, in the Nios II/e core, `mul` and `div` are unimplemented.
- Worst-case exception (or interrupt) latency—The value of the exception (or interrupt) latency, including the maximum disabled time or maximum masked time. Including the maximum disabled or masked time accounts for the case when the exception (or interrupt) occurs at the beginning of the masked or disabled time.

Interrupt Controllers

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

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

Internal Interrupt Concepts

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

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

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

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

External Interrupt Concepts

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

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

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

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

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

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

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

An EIC provides the following information about each hardware interrupt:

Requested Handler Address

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

Requested Interrupt Level

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

Requested Register Set

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

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

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

Requested NMI Mode

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

Shadow Register Sets

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

Latency and Response Time

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

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

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

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

Internal or External Interrupt Controller

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

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

Shadow Register Sets

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

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

How the Hardware Works

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

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

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

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

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

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

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

How the Internal Interrupt Controller Works

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

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

How an External Interrupt Controller Works

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

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

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

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

Nios II Interrupt Service Routines

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

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

For examples of HAL ISRs, refer to existing handlers for Altera components.

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

HAL APIs for Hardware Interrupts

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

Altera also supports a legacy hardware interrupt API. This API supports only the IIC. If you have a custom driver written prior to Nios II version 9.1, it uses the legacy API.

Both interrupt APIs include the following types of routines:

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

Both interrupt APIs support the following types of BSPs:

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

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

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

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

Selecting an Interrupt API

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

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

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

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

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

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

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

The Enhanced HAL Interrupt API

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

Table 8–1. Enhanced HAL Interrupt API Functions

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

Note ~~to Table 8–1~~:

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

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

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

Using the Enhanced HAL Interrupt API to Implement ISRs

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

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

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

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

The External Interrupt Controller Driver

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

Using the HAL Interrupt API with the VIC

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

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

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

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

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

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

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

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

The Legacy HAL Interrupt API

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

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

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

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

Using the Legacy HAL API to implement ISRs

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

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

Supporting Multiple Interrupt APIs

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

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

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

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

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

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

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

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

HAL ISR Restrictions

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

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

Writing an ISR

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

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

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

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

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

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

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

Using Interrupt Funnels

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

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

The HAL includes the following interrupt funnels:

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

The HAL funnel code is called from the vector table.

Running in a Restricted Environment

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

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

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

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

Managing Pre-Emption

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

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

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

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

Operating systems can also publish the `isr_preemption_supported` property.

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

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

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

Registering an ISR with the Enhanced Interrupt API

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

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

The function has the following parameters:

- `ic_id` is the interrupt controller identifier (ID) as defined in **system.h**. With daisy-chained EICs, `ic_id` identifies the EIC in the daisy chain. With the IIC, `ic_id` is not significant.
- `irq` is the hardware interrupt number for the device, as defined in **system.h**.
  - For the IIC, `irq` is the IRQ number. Interrupt priority corresponds inversely to the IRQ number. Therefore, IRQ$_0$ represents the highest priority interrupt and IRQ$_{31}$ is the lowest.
  - For an EIC, `irq` is the interrupt port ID.
- `isr_context` points to a data structure associated with the device driver instance. `isr_context` is passed as the input argument to the isr function. It is used to pass context-specific information to the ISR, and can point to any ISR-specific information. The context value is opaque to the HAL; it is provided entirely for the benefit of the user-defined ISR.
- `isr` is a pointer to the ISR function that is called in response to IRQ number `irq`. The ISR function prototype is:

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

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

  ☞ Registering a null pointer for `isr` results in the interrupt being disabled.
- `flags` is reserved.

Methods the HAL Uses to Register the ISR

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

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

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

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

☞ Hardware-specific initialization might also be required.

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

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

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

Enabling and Disabling Interrupts

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

Disable hardware interrupts for as short a time as possible. Maximum interrupt latency increases with the longest amount of time interrupts are disabled. For more information about disabled interrupts, refer to "Keep Interrupts Enabled" on page 8–19.

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

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

Configuring an External Interrupt Controller

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

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

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

C Example

An ISR to Service a Button PIO Interrupt

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

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

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

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

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

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

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

Registering the Button PIO ISR with the HAL

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

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

1. Button is pressed, generating an IRQ.

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

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

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

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

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

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

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

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

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

Upgrading to the Enhanced HAL Interrupt API

If you have custom device drivers, Altera recommends that you upgrade them to use the enhanced HAL interrupt API. The enhanced API maintains compatibility with the IIC, while supporting external interrupt controllers. The legacy HAL interrupt API is deprecated.

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

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

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

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

Table 8–2. HAL Interrupt API Functions to Upgrade

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

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

Improving Nios II ISR Performance

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

Software Performance Improvements

In improving your ISR performance, you probably consider software changes first. However, in some cases it might require less effort to implement hardware design changes that increase system efficiency. For a discussion of hardware optimizations, refer to "Hardware Performance Improvements" on page 8–23.

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

Execute Time-Intensive Algorithms in the Application Context

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

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

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

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

Implement Time-Intensive Algorithms in Hardware

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

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

Increase Buffer Size

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

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

Use Double Buffering

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

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

Keep Interrupts Enabled

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

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

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

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

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

Use Fast Memory

ISR performance depends on memory speed.

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

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

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

Use a Separate Exception Stack

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

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

Separate General Exception Stack

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

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

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

Separate Hardware Interrupt Stack

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

Controlling the Separate Hardware Interrupt Stack

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

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

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

Table 8–3. Separate Exception Stack Usage

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

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

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

Use Nested Hardware Interrupts

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

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

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

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

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

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

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

Nested Hardware Interrupts with the Internal Interrupt Controller

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

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

Nested Hardware Interrupts with an External Interrupt Controller

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

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

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

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

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

Locate ISR Body in Vector Table

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

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

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

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

Use Compiler Optimization

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

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

Hardware Performance Improvements

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

In some cases, these changes also require changes in the software architecture or implementation. For a discussion of these and other software optimizations, refer to "Software Performance Improvements" on page 8–18.

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

Use Vectored Hardware Interrupts

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

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

Using the Interrupt Vector Custom Instruction

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

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

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

☞ The interrupt vector custom instruction is only available in hardware systems generated by SOPC Builder.

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

Using an External Interrupt Controller

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

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

Add Fast Memory

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

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

Add a DMA Controller

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

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

Place the Handler in Fast Memory

For the fastest execution of exception handler code, place the handler in a fast memory device. For example, an on-chip RAM with zero wait states is preferable to a slow SDRAM. For best performance, store exception handling code and data in tightly-coupled memory.

Use a Fast Nios II Core

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

Select Hardware Interrupt Priorities

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

Hardware Interrupt Priorities with the Internal Interrupt Controller

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

Hardware Interrupt Priorities with an External Interrupt Controller

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

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

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

Debugging Nios II ISRs

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

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

HAL Exception Handling System Implementation

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

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

Exception Handling System Structure

The exception handling system consists of the following components:

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

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

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

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

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

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

General Exception Funnel

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

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

Hardware Interrupt Dispatch with the Internal Interrupt Controller

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

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

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

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

All hardware interrupts are higher priority than software exceptions.

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

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

Returning from Exceptions

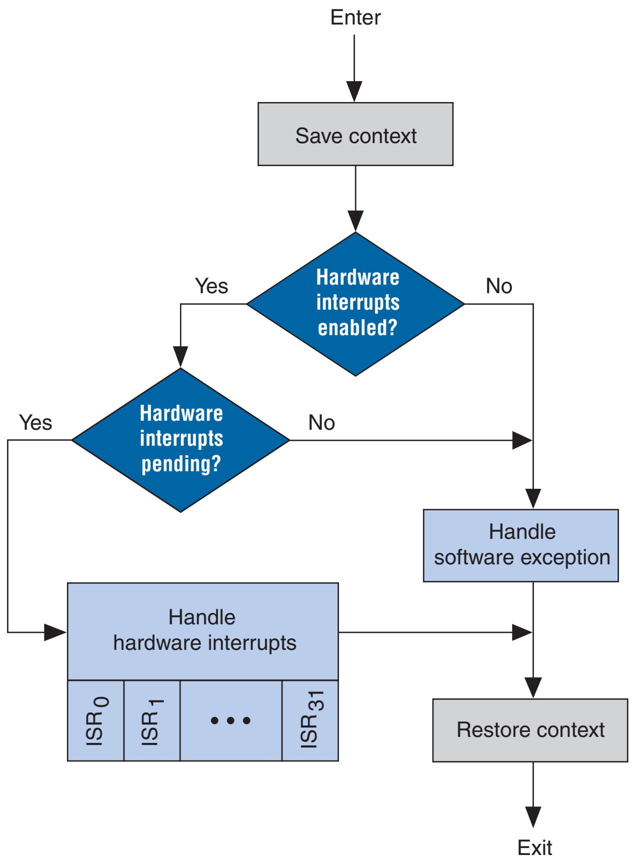
After returning from the ISR or software exception handler, the general exception funnel performs the following tasks:

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

Hardware Interrupt Funnel

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

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


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

Interrupt Funnel for the Internal Interrupt Controller

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

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

When the interrupt vector custom instruction is present in the Nios II processor, the HAL source detects it at compile time and generates code using the custom instruction. For further information, refer to "Using the Interrupt Vector Custom Instruction" on page 8–24.

Interrupt Funnels for External Interrupt Controllers

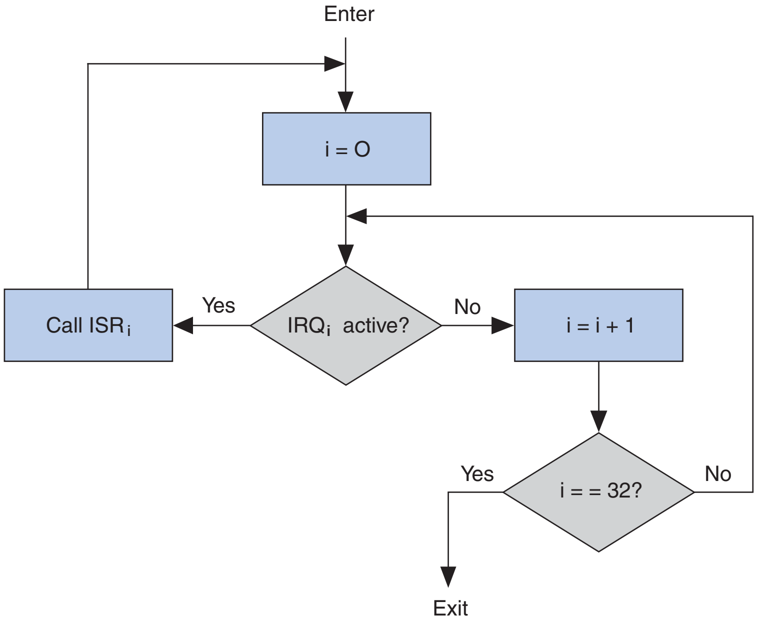
With the EIC interface, the Nios II processor supports a potentially unlimited number of hardware interrupts on daisy-chained EICs. The interrupt priority level can be software-configurable. Details of setting interrupt priorities depend on the particular EIC implementation. The hardware ensures that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

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

For details, refer to "Exception Handling System Structure" on page 8–26.

Interrupt Funnels for Internal Interrupt Controllers

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



The HAL provides the following interrupt funnels:

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

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

Software Exception Funnel

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

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

Unimplemented Instructions

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

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

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

Unimplemented instructions are different from invalid instructions, which are described in "Invalid Instructions" on page 8–32.

When to Use the Unimplemented Instruction Handler

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

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

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

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

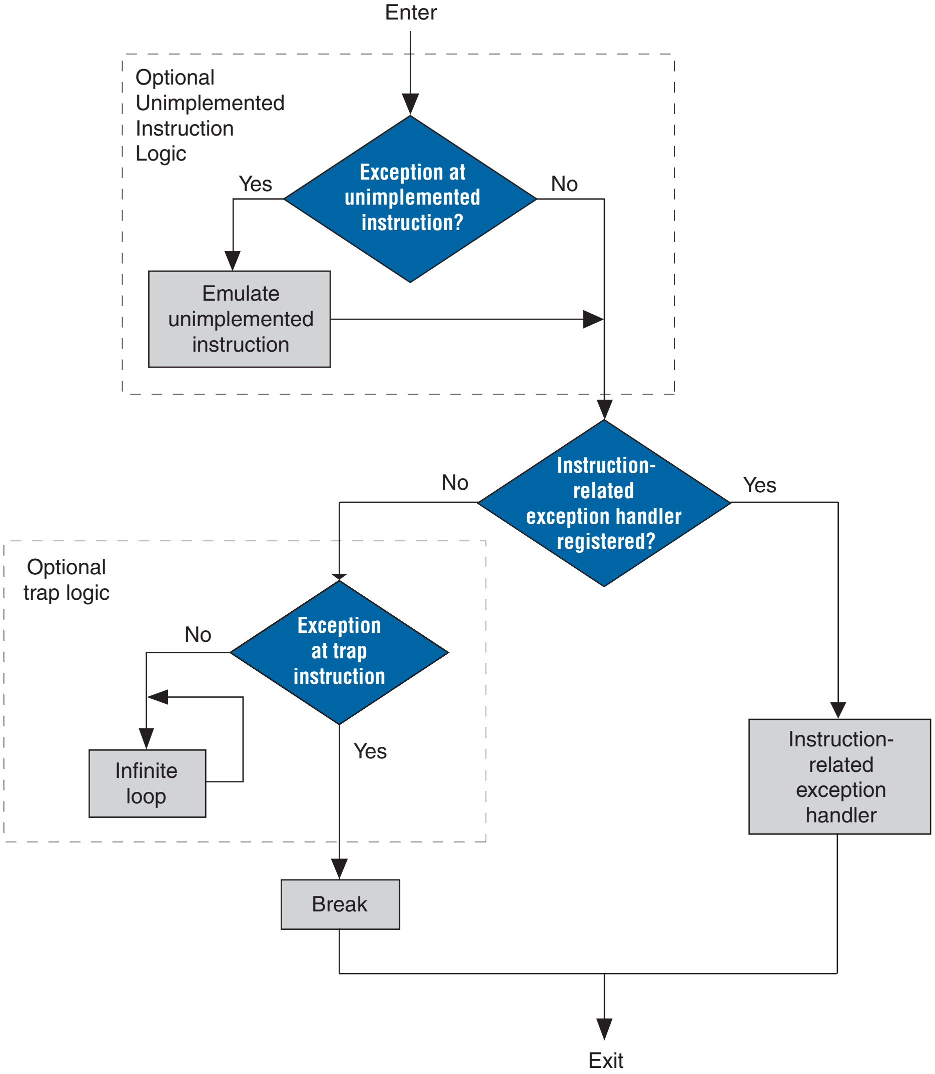
If unimplemented instruction emulation is disabled, but the processor encounters an unimplemented instruction, the software exception funnel treats the exception as a miscellaneous exception. Miscellaneous exceptions are described in "Miscellaneous Exceptions" on page 8–32.

Using the Unimplemented Instruction Handler

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

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

Figure 8–3. HAL Software Exception Funnel


Enter
Optional Unimplemented Instruction Logic
Exception at unimplemented instruction?
Yes
No
Emulate unimplemented instruction
Instruction-related exception handler registered?
No
Yes
Optional trap logic
Exception at trap instruction
No
Yes
Infinite loop
Instruction-related exception handler
Break
Exit

Instruction-Related Exceptions

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

Software Trap Handling

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

Miscellaneous Exceptions

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

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

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

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

Miscellaneous exceptions can occur for these reasons:

- Advanced exceptions, the memory protection unit (MPU), or the memory management unit (MMU) are implemented in the Nios II processor core. To handle advanced and MPU exceptions, refer to "The Nios II Instruction-Related Exception Handler". To handle MMU exceptions, you need to implement a full-featured operating system, as mentioned in the *Programming Model* chapter of the *Nios II Processor Reference Handbook*.
- You need to include the unimplemented instruction handler, discussed in "Unimplemented Instructions" on page 8–30.
- A peripheral is generating spurious hardware interrupts. This is a symptom of a serious hardware problem. A peripheral might generate spurious hardware interrupts if it deasserts its interrupt output before an ISR has explicitly serviced it.

Invalid Instructions

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

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

☞ Invalid instructions are different from unimplemented instructions, which are described in "Unimplemented Instructions" on page 8–30.

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

The Nios II Instruction-Related Exception Handler

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

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

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

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

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

Writing an Instruction-Related Exception Handler

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

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

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

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

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

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

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

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

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

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

Exception Cause Codes

Table 8–4. Nios II Exception Cause Codes

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

Notes ~~to Table 8–4~~:

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

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

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

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

Registering an Instruction-Related Exception Handler

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

The function prototype is as follows:

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

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

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

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

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

For details, refer to “Settings Managed by the Software Build Tools” in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer’s Handbook*.

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

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

Removing an Instruction-Related Exception Handler

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

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

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

Document Revision History

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

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

Date	Version	Changes
May 2011	11.0.0	■ Introduction of Qsys system integration tool ■ Interrupt vector custom instruction only available with SOPC Builder
February 2011	10.1.0	Removed “Referenced Documents” section.
July 2010	10.0.0	Maintenance release.

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

Date	Version	Changes
November 2009	9.1.0	■ Described HAL support for external interrupt controller interface. ■ Described HAL support for shadow register sets with external interrupt controller interface. ■ Described enhanced HAL interrupt API. ■ Legacy HAL interrupt API deprecated. ■ Removed information specific to the Nios II Integrated Development Environment (IDE).
March 2009	9.0.0	■ Reorganized and updated information and terminology to clarify role of Nios II Software Build Tools. ■ Corrected minor typographical errors.
May 2008	8.0.0	Maintenance release.
October 2007	7.2.0	Maintenance release.
May 2007	7.1.0	■ Added table of contents to “Introduction” section. ■ Added Referenced Documents section.
March 2007	7.0.0	Maintenance release.
November 2006	6.1.0	■ Describes support for the interrupt vector custom instruction.
May 2006	6.0.0	■ 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	5.1.0	■ 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 “Nios II Interrupt Service Routines”.
May 2005	5.0.0	Added tightly-coupled memory information.
December 2004	1.2	Corrected the “Registering the Button PIO ISR with the HAL” example.
September 2004	1.1	■ Changed examples. ■ Added ISR performance data.
May 2004	1.0	Initial release.

9. Cache and Tightly-Coupled Memory

NII52007-11.0.0

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

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

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

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

This chapter contains the following sections:

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

Nios II Cache Implementation

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

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

Subscribe

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

Defining Cache Properties

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

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

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

This system has a 4-kilobyte (KB) instruction cache with 32 byte lines, and no data cache.

HAL API Functions for Managing Cache

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

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

Initializing the Nios II Cache after Reset

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

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

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

Assembly Code to Initialize the Instruction Cache

Place the `initi` instruction in a loop that executes `initi` for each instruction cache line address. ~~Example 9–2 shows an example of assembly code to initialize the instruction cache~~.

Example 9–2. Assembly Code to Initialize the Instruction Cache

```
    mov     r4, r0
    movhi   r5, %hi(NIOS2_ICACHE_SIZE)
    ori     r5, r5, %lo(NIOS2_ICACHE_SIZE)
icache_init_loop:
    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 initializes a single 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.

Assembly Code to Initialize the Data Cache

Place the `initd` instruction in a loop that executes `initd` for each data cache line address. ~~Example 9–3 shows an example of assembly code to initialize the data cache:~~

Example 9–3. Assembly Code to Initialize the Data Cache

```
    mov     r4, r0
    movhi   r5, %hi(NIOS2_DCACHE_SIZE)
    ori     r5, r5, %lo(NIOS2_DCACHE_SIZE)
dcache_init_loop:
    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 do not 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 Users

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

Nios II Device Driver Cache Considerations

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

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

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

For HAL Users

The HAL provides the C-language macros `IORD` and `IOWR` that expand to the appropriate assembly instructions to bypass the data cache. The `IORD` macro expands to the `ldwio` instruction, and the `IOWR` macro expands to the `stwio` instruction. These macros are provided to enable 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` in 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` in 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`.

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

Macro	Use
`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`.

Cache Considerations for Writing Program Loaders

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

~~Example 9–4 shows assembly code that writes a new instruction to memory.~~

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

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

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

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

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

For Users of the HAL

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

Managing Cache in Multi-Master and Multi-Processor Systems

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

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

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

Bit-31 Cache Bypass

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

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

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

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

For HAL Users

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

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

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

Nios II Tightly-Coupled Memory

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

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

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

Document Revision History

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

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

Date	Version	Changes
May 2011	11.0.0	No change
February 2011	10.1.0	Removed “Referenced Documents” section.
July 2010	10.0.0	Maintenance release.
November 2009	9.1.0	Maintenance release.
March 2009	9.0.0	■ Reorganized and updated information and terminology to clarify role of Nios II Software Build Tools. ■ Corrected minor typographical errors.
May 2008	8.0.0	Maintenance release.
October 2007	7.2.0	Maintenance release.
May 2007	7.1.0	■ Added table of contents to “Introduction” section. ■ Added Referenced Documents section.
March 2007	7.0.0	Maintenance release.
November 2006	6.1.0	Maintenance release.
May 2006	6.0.0	Maintenance release.
October 2005	5.1.0	Added detail to section “Nios II Tightly-Coupled Memory”.

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

Date	Version	Changes
May 2005	5.0.0	Added tightly-coupled memory section.
May 2004	1.0	Initial release.

10. MicroC/OS-II Real-Time Operating System

NII52008-11.0.0

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

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

Overview of the MicroC/OS-II RTOS

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

MicroC/OS-II provides the following services:

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

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

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

Further Information

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

Licensing

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

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

Other RTOS Providers

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

For a complete list of RTOSs that support the Nios II processor, visit the Embedded Software page of the Altera® website.

The Nios II Implementation of MicroC/OS-II

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

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

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

- Processor-specific code: <*Nios II EDS install path*>**/components/altera_nios2/UCOSII**
- Processor-independent code: <*Nios II EDS install path*>**/components/micrium_uc_osii**

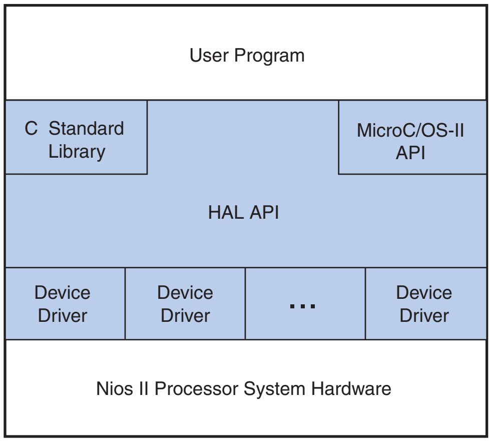
The MicroC/OS-II software package behaves like the drivers for hardware components: When MicroC/OS-II is included in a Nios II project, the header and source files from **components/micrium_uc_osii** are included in the project path, causing the MicroC/OS-II kernel to compile and link as part of the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

MicroC/OS-II Architecture

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

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

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



The multi-threaded environment affects certain HAL functions.

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

MicroC/OS-II Thread-Aware Debugging

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

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

MicroC/OS-II Device Drivers

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

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

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

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

The Nios II Software Build Tools (SBT) copies the files to your BSP's **obj** subdirectory.

For more information about specifying file paths with the Nios II SBT, refer to "Nios II Embedded Software Projects" in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

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

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

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

Thread-Safe HAL Drivers

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

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

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

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

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

Macro	Defined in Header	MicroC/OS-II Implementation	Single-Threaded HAL Implementation
`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.
`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).

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

Macro	Defined in Header	MicroC/OS-II Implementation	Single-Threaded HAL Implementation
ALT_FLAG_PEND(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 on 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 in a single structure referenced through the pointer _impure_ptr. However, the Altera MicroC/OS-II implementation creates a new instance of the structure for each task. During a context switch, the value of _impure_ptr is updated to point to the current task's version of this structure. In this way, the contents of the structure pointed to by _impure_ptr are treated as thread local. For example, through this mechanism each task has its own version of errno.

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

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

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

Interrupt Service Routines for MicroC/OS-II

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

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

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

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

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

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

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

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

Document Revision History

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

Table 10–2. Document Revision History (Part 1 of 2)

Date	Version	Changes
May 2011	11.0.0	Introduction of Qsys system integration tool
February 2011	10.1.0	Removed "Referenced Documents" section.
July 2010	10.0.0	Maintenance release.

Table 10–2. Document Revision History (Part 2 of 2)

Date	Version	Changes
November 2009	9.1.0	■ Introduced the Nios II Software Build Tools for Eclipse. ■ Remove tables of Nios II IDE-specific setting names. Refer solely to BSP setting names.
March 2009	9.0.0	■ Reorganized and updated information and terminology to clarify role of Nios II Software Build Tools. ■ Corrected minor typographical errors.
May 2008	8.0.0	Maintenance release.
October 2007	7.2.0	■ Added documentation for MicroC/OS-II development with the Nios II Software Build Tools. ■ Added description of HAL ISR support.
May 2007	7.1.0	■ Added table of contents to “Introduction” section. ■ Added Referenced Documents section.
March 2007	7.0.0	Maintenance release.
November 2006	6.1.0	Maintenance release.
May 2006	6.0.0	Maintenance release.
October 2005	5.1.0	Maintenance release.
May 2005	5.0.0	Maintenance release.
December 2004	1.2	
September 2004	1.1	Added thread-aware debugging paragraph.
May 2004	1.0	Initial release.

11. Ethernet and the NicheStack TCP/IP Stack - Nios II Edition

NII52013-11.0.0

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

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

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

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

Prerequisites for Understanding the NicheStack TCP/IP Stack

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

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

Introduction to the NicheStack TCP/IP Stack - Nios II Edition

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. Altera's implementation of the NicheStack TCP/IP Stack includes an API wrapper, providing the standard, well documented socket API.

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

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

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

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

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

The NicheStack TCP/IP Stack Files and Directories

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

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

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

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

Licensing

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

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

Other TCP/IP Stack Providers for the Nios II Processor

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

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

Using the NicheStack TCP/IP Stack - Nios II Edition

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

The NicheStack TCP/IP Stack Tasks

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

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

These tasks are started when the initialization process succeeds in the `netmain()` function, as described in "netmain()".

☞ You can modify the task priority and stack sizes using `#define` statements in the configuration file **ipport.h**. You can create additional system tasks by enabling 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 the RTOS is running and I/O drivers are available.

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

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

alt_iniche_init()

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

```
void alt_iniche_init(void)
```

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

netmain()

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

```
void netmain(void)
```

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

iniche_net_ready

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

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

Example 11–1 illustrates the use of `iniche_net_ready` to wait until the network stack has completed initialization.

Example 11–1. Instantiating the NicheStack TCP/IP Stack

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

  alt_iniche_init();
  netmain();

  while (!iniche_net_ready)
    TK_SLEEP(1);

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

    .
    .
    .

}
```

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

get_mac_addr() and get_ip_addr()

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

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

Prototype for get_mac_addr()

The prototype for `get_mac_addr()` is:

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

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

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

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

Example 11–2. An Implementation of get_mac_addr()

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

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

  return ret_code;
}
```

Prototype for get_ip_addr()

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

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

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

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

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

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

To enable DHCP, include the line:

```
*use_dhcp = 1;
```

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

To assign a static IP address, include the lines:

```
*use_dhcp = 0;
```

The prototype for `get_ip_addr()` is in the header file <*iniche path*>**/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 Example 11–3. In a real application, you might need to return -1 on error.

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

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

Calling the Sockets Interface

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

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

The prototype for `TK_NEWTASK()` is:

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

Example 11–3. An Implementation of get_ip_addr()

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

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

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

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

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

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

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

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

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

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

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

You can find **ipport.h** in the **debug/system_description** directory for your BSP project.

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

NicheStack TCP/IP Stack General Settings

The ARP, UDP, and IP protocols are always enabled. ~~Table 11–1 shows the protocol options~~.

Table 11–1. Protocol Options

Option	Description
TCP	Enables and disables the TCP.

~~Table 11–2 shows the global options, which affect the overall behavior of the TCP/IP stack.~~

Table 11–2. Global Options

Option	Description
Use DHCP to automatically assign IP address	If this option is turned on, the component uses DHCP to acquire an IP address. If this option is turned off, you must assign a static IP address.
Enable statistics	If this option is turned on, the stack keeps counters of packets received, errors, etc. The counters are defined in `mib` structures defined in various header files in directory *<iniche path>*/**src/downloads/30src/h**. For details about `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. Refer to "Known Limitations" on page 11–10.

IP Options

~~Table 11–3 shows the IP options.~~

Table 11–3. IP Options

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

TCP Options

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

Table 11–4. TCP Options

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

Further Information

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

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

Known Limitations

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

Document Revision History

~~Table 11–5 shows the revision history for this document.~~

Table 11–5. Document Revision History

Date	Version	Changes
May 2011	11.0.0	No change
February 2011	10.1.0	Removed "Referenced Documents" section.
July 2010	10.0.0	Maintenance release.
November 2009	9.1.0	■ Introduced the Nios II Software Build Tools for Eclipse™. ■ Nios II IDE information removed to *Appendix A. Using the Nios II Integrated Development Environment*.
March 2009	9.0.0	■ Reorganized and updated information and terminology to clarify role of Nios II Software Build Tools. ■ Corrected minor typographical errors.
May 2008	8.0.0	Maintenance release.
October 2007	7.2.0	Maintenance release.
May 2007	7.1.0	■ Minor clarifications added to content. ■ Added table of contents to Overview section. ■ Added Referenced Documents section.
March 2007	7.0.0	Maintenance release.
November 2006	6.1.0	Initial release.

12. Read-Only Zip File System

NII52012-11.0.0

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

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

Using the Read-Only Zip File System in a Project

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

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

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

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

  This code, called from within a HAL-based program, opens the file **hello** for reading.
- The name of the zip file you wish to use.

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

Preparing the Zip File

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

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

Programming the Zip File to Flash

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

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

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

Document Revision History

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

Table 12–1. Document Revision History

Date	Version	Changes
May 2011	11.0.0	No change
February 2011	10.1.0	Removed "Referenced Documents" section.
July 2010	10.0.0	Maintenance release.
November 2009	9.1.0	Maintenance release.
March 2009	9.0.0	■ Reorganized and updated information and terminology to clarify role of Nios II Software Build Tools. ■ Corrected minor typographical errors.
May 2008	8.0.0	Maintenance release.
October 2007	7.2.0	Maintenance release.
May 2007	7.1.0	■ Added table of contents to "Introduction" section. ■ Added Referenced Documents section.
March 2007	7.0.0	Maintenance release.
November 2006	6.1.0	Maintenance release.
May 2006	6.0.0	Maintenance release.
October 2005	5.1.0	Maintenance release.
May 2005	5.0.0	Maintenance release.
May 2004	1.0	Initial release.

13. Publishing Component Information to Embedded Software

NII52018-11.0.0

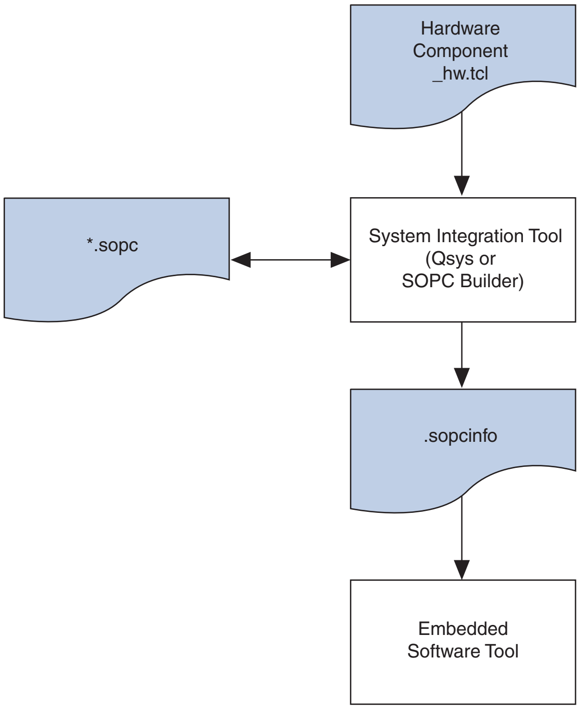
This document describes how to publish hardware component information for embedded software tools. You can publish component information for use by software, such as a C compiler and a board support package (BSP) generator. Information used by a C compiler might be a set of `#define` statements that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

This chapter contains the following sections:

Embedded Component Information Flow

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

Figure 13–1. Embedded Component Information Flow



Hardware Component _hw.tcl
*.sopc
System Integration Tool (Qsys or SOPC Builder)
.sopcinfo
Embedded Software Tool

A component publishes information by including Tcl assignment statements in its component description file, <*component_name*>**_hw.tcl**. Each assignment is a name-value pair that can be associated with the entire component, or with a single interface. When the assignment statement applies to the entire component, it is set using the `set_module_assignment` command. Assignment statements that apply to an interface are set using the `set_interface_assignment` command. Example 13–1 shows the syntax for these assignment statements.

Example 13–1. Syntax of Assignment Statements

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

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


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

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

For more information about the **_hw.tcl** file and using Tcl to define hardware components, refer to the *System Design with Qsys* section in *Volume 1: Design and Synthesis* of the *Quartus® II Handbook*.

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

Embedded Software Assignments

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

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

C Macro Namespace

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

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

Example 13–2 illustrates a Tcl assignment statement for the `BAUD_RATE` of uart_0 in a hardware system.

Example 13–2. C Macro Example

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

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

Example 13–3 illustrates the corresponding C or C++ `#define`. The string `BAUD_RATE` is appended to the name of the component. This `#define` is included in the `system.h` file.

Example 13–3. Generated Macro in system.h

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

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

For complete details on formatting constants, refer to the GNU web page.

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

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

Configuration Namespace

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

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

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

Table 13–2. Configuration Data Types

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

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

Table 13–3. Component Configuration Information - Assign with set_module_assignment

Configuration Name	Type	Default	Meaning	Example
`cpuArchitecture`	string	—	Processor instruction set architecture. Provide this assignment if you want your component to be considered a processor.	My 8051
`requiredDriver`	boolean	0	If this configuration is 1 (true), the component requires a software driver. Software tools are expected to generate a warning if no driver is found.	1

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

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

Configuration Name	Type	Default	Meaning	Examples
`isMemoryDevice`	boolean	0	The slave port provides access to a memory device.	Altera® On-Chip Memory Component, DDR Controller, erasable programmable configurable serial (EPCS) Controller
`isPrintableDevice`	boolean	0	The slave port provides access to a character-based device.	Altera UART, Altera JTAG UART, Altera LCD

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

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

Note to Table 13–4:

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

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

Table 13–5. Streaming Source Information - Assign with set_interface_assignment

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

Note to Table 13–5:

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

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

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

Table 13–6. Streaming Sink Information - Assign with set_interface_assignment

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

Note to Table 13–6:

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

Memory Initialization Namespace

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

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

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

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

Table 13–7. Memory Initialization Data Types

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

Note to Table 13–7:

(1) Quotation marks are not required.

Memory initialization assignments are associated with an entire component. Table 13–8 shows the embedded software memory initialization names.

Table 13–8. Memory Initialization Information - Assign with set_module_assignment Command

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

Document Revision History

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

Table 13–9. Document Revision History (Part 1 of 2)

Date	Version	Changes
May 2011	11.0.0	Introduction of Qsys system integration tool
February 2011	10.1.0	Removed “Referenced Documents” section.
July 2010	10.0.0	Maintenance release.

Table 13–9. Document Revision History (Part 2 of 2)

Date	Version	Changes
November 2009	9.1.0	Add the following Avalon-ST interface settings, to support external interrupt controllers: ■ `embeddedsw.configuration.isInterruptControllerReceiver` ■ `embeddedsw.configuration.isInterruptControllerSender` ■ `embeddedsw.configuration.transportsInterruptsFromReceivers`
March 2009	9.0.0	Initial release.

Section IV. Reference Material

This section provides a comprehensive reference to the Nios® II hardware abstraction layer (HAL) application program interface (API) and the utilities, scripts, and settings that constitute the Nios II Software Build Tools. This section includes the following chapters:

- Chapter 14, HAL API Reference
- Chapter 15, Nios II Software Build Tools Reference

14. HAL API Reference

NII52010-11.0.0

This chapter provides an alphabetically ordered list of all the functions in the hardware abstraction layer (HAL) application program interface (API). Each function is listed with its C prototype and a short description. Each listing provides information about 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 chapter only lists the functionality provided by the HAL. 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.

☞ Each function description lists the C header file that your code must include to access the function. Because header files include other header files, the function prototype might not be defined in the listed header file. However, you must include the listed header file in order to include all definitions on which the function depends.

For more details about the newlib API, refer to the newlib documentation. On the Windows **Start** menu, click **Programs** > **Altera** > **Nios II** *<version>* > **Nios II** *<version>* **Documentation**.

This chapter contains the following sections:

- "HAL API Functions" on page 14–1
- "HAL Standard Types" on page 14–79

HAL API Functions

The HAL API functions are shown on the following pages.

_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, `exit()` calls this function when `main()` completes. Because there is only a single process in HAL systems, the HAL implementation blocks forever.

Interrupts are not disabled, so ISRs continue to execute.

The input argument, `exit_code`, is ignored.

Return: –

See also: newlib 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

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. Refer to "Using Timer Devices" in the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*. The HAL waits `nticks` system clock ticks before calling the `callback()` function. When the HAL calls `callback()`, it passes it the input argument context.

The `alarm` argument 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 on 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()`.

On 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 the data cache for a memory region of length len bytes, starting at address start. Flushing the cache consists of writing back dirty data and then invalidating the cache. 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, that is, 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. After it is registered, you can access a device using the standard I/O functions. Refer to the *Developing Programs Using the Hardware Abstraction Layer* 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 no other thread can use the device list at the time that `alt_dev_reg()` is called. Call `alt_dev_reg()` only in the following circumstances:

- When running in single-threaded mode.
- From a device initialization function called by `alt_sys_init()`. `alt_sys_init()` may only be called by the single-threaded C startup code.

Return: The return value is zero upon 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 using the direct memory access (DMA) receive channel, `rxchan`. The current implementation always succeeds.

Return: The return value is zero on 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, depends on the underlying device driver. In general it safest to assume that it is not thread-safe.

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 14–1 shows generic requests defined in **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 safest to assume that it is not thread-safe.

Do not call the `alt_dma_rxchan_ioctl()` function while DMA transfers are pending, or unpredictable behavior could result.

For device-specific information about the Altera® DMA controller core, refer to the *DMA Controller Core* chapter in the *Embedded Peripherals IP User Guide*.

Return: A negative return value indicates failure. The interpretation of nonnegative return values 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 14–1. 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.

Table 14–1. Generic Requests

Request	Meaning
ALT_DMA_TX_ONLY_ON	The ALT_DMA_TX_ONLY_ON request causes a DMA channel to operate in a mode in which only the transmitter is under software control. The other side writes continuously from a single location. The address to which to write is the argument to this request.
ALT_DMA_TX_ONLY_OFF	Return to the default mode, in which both the receive and transmit sides of the DMA can be under software control.
ALT_DMA_RX_ONLY_ON	The ALT_DMA_RX_ONLY_ON request causes a DMA channel to operate in a mode in which 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	Return to the default mode, in which both the receive and transmit sides of the DMA can be under software control.

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, for example, `/dev/dma_0`.
Return:	The return value is null on failure and non-null otherwise. If an error occurs, `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 after the data is received; `handle`, an opaque value passed to `done`.

Whether this function is thread-safe, or can be called from an ISR, depends on the underlying device driver. In general it safest to assume that it is not thread-safe.

Return: The return value is zero upon success. A negative return value indicates that the request cannot be posted.

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. After it is registered, a device can be accessed using the functions described in "Using DMA Devices" in the *Developing Programs Using the Hardware Abstraction Layer* 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. Call `alt_dma_rxchan_reg()` only in the following circumstances:

- When running in single-threaded mode.
- From a device initialization function called by `alt_sys_init()`. `alt_sys_init()` may only be called by the single-threaded C startup code.

Return: The return value is zero upon 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 using the DMA transmit channel, `txchan`. The current implementation always succeeds.
Return:	The return value is zero on 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.

Refer to Table 14–1 on page 14–11 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 safest to assume that it is not thread-safe.

Do not call the `alt_dma_txchan_ioctl()` function while DMA transfers are pending, or unpredictable behavior could 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, for example, `/dev/dma_0`.

Return: The return value is null on failure and non-null otherwise. If an error occurs, `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. After it is registered, a device can be accessed using the functions described in “Using DMA Devices” in the *Developing Programs Using the Hardware Abstraction Layer* 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. Call `alt_dma_txchan_reg()` only in the following circumstances:

- When running in single-threaded mode.
- From a device initialization function called by `alt_sys_init()`. `alt_sys_init()` may only be called by the single-threaded C startup code.

Return: The return value is zero upon 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 after 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, depends on the underlying device driver. In general it safest to assume that it is not thread-safe.

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, depends on the underlying device driver. In general it safest to assume that it is not thread-safe.
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. Refer to "Using Flash Devices" in the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

Call the `alt_erase_flash_block()` function only 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. 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_exception_cause_generated_bad_addr()

Prototype:

```
int alt_exception_cause_generated_bad_addr
                              ( alt_exception_cause cause)
```

Commonly called by: Instruction-related exception handlers

Thread-safe:

Available from ISR:

Include: `<sys/alt_exceptions.h>`

Description: This function validates the `bad_addr` argument to an instruction-related exception handler. The function parses the handler’s `cause` argument to determine whether the `bad_addr` register contains the exception-causing address.

If the exception is of a type that generates a valid address in `bad_addr`, this function returns a nonzero value. Otherwise, it returns zero.

If the `cause` register is unimplemented in the Nios® II processor core, this function always returns zero.

Return: A nonzero value means `bad_addr` contains the exception-causing address.

Zero means the value of `bad_addr` is to be ignored.

See also: alt_instruction_exception_register()

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. Call the `alt_flash_close_dev()` function only 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. After it is opened, you can perform the following operations:

- Write to a flash device using `alt_write_flash()`
- Read from a flash device using `alt_read_flash()`
- Control individual flash blocks using `alt_get_flash_info()`, `alt_erase_flash_block()`, or `alt_write_flash_block()`.

Call the `alt_flash_open_dev` function only when operating in single-threaded mode.

Return: The return value is zero upon failure. Any other value indicates 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. After it is registered, a file system can be accessed using the standard I/O functions. Refer to the *Developing Programs Using the Hardware Abstraction Layer* 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. Call `alt_fs_reg()` only in the following circumstances: ■ When running in single-threaded mode. ■ From a device initialization function called by `alt_sys_init()`. `alt_sys_init()` may only be called by the single-threaded C startup code.
Return:	The return value is zero upon 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. Call this function only 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. 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_ic_irq_disable()

Prototype: `int alt_ic_irq_disable (alt_u32 ic_id, alt_u32 irq)`

Commonly called by: C/C++ programs

Device drivers

Thread-safe: Yes.

Available from ISR: No.

Include: `<sys/alt_irq.h>`

Description: The `alt_ic_irq_disable()` function disables a single interrupt.

The function arguments are as follows:

- `ic_id` is the interrupt controller identifier (ID) as defined in **system.h**, identifying the external interrupt controller in the daisy chain. This argument is ignored if the external interrupt controller interface is not implemented.
- irq is the interrupt request (IRQ) number, as defined in system.h, identifying the interrupt to enable.
- A driver for an external interrupt controller (EIC) must implement this function.

Return: This function returns zero if successful, or nonzero otherwise. The function fails if the `irq` parameter is greater than the maximum interrupt port number supported by the external interrupt controller.

See also: `alt_irq_disable_all()`

`alt_irq_enable()`

`alt_irq_enable_all()`

`alt_irq_enabled()`

`alt_irq_register()`

`alt_irq_disable()`

alt_ic_irq_enable()

alt_ic_irq_enabled()

alt_ic_isr_register()

alt_ic_irq_enable()

Prototype: `int alt_ic_irq_enable (alt_u32 ic_id, alt_u32 irq)`

Commonly called by: C/C++ programs

Device drivers

Thread-safe: Yes.

Available from ISR: No.

Include: `<sys/alt_irq.h>`

Description: The `alt_ic_irq_enable()` function enables a single interrupt.

The function arguments are as follows:

- `ic_id` is the interrupt controller ID as defined in **system.h**, identifying the external interrupt controller in the daisy chain. This argument is ignored if the external interrupt controller interface is not implemented.
- `irq` is the IRQ number, as defined in **system.h**, identifying the interrupt to enable.
- A driver for an EIC must implement this function.

Return: This function returns zero if successful, or nonzero otherwise. The function fails if the `irq` parameter is greater than the maximum interrupt port number supported by the external interrupt controller.

See also: `alt_irq_disable()`

`alt_irq_disable_all()`

`alt_irq_enable_all()`

`alt_irq_enabled()`

`alt_irq_register()`

`alt_irq_enable()`

alt_ic_irq_disable()

alt_ic_irq_enabled()

alt_ic_isr_register()

alt_ic_irq_enabled()

Prototype:	`int alt_ic_irq_enabled (alt_u32 ic_id, alt_u32 irq)`
Commonly called by:	Device drivers
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	`<sys/alt_irq.h>`
Description:	This function determines whether a specified interrupt is enabled. The function arguments are as follows: ■ `ic_id` is the interrupt controller ID as defined in **system.h**, identifying the external interrupt controller in the daisy chain. This argument is ignored if the external interrupt controller interface is not implemented. ■ `irq` is the IRQ number, as defined in **system.h**, identifying the interrupt to enable. ■ A driver for an EIC must implement this function.
Return:	Returns zero if the specified interrupt is disabled, and nonzero otherwise.

See also:	`alt_irq_disable()`
	`alt_irq_disable_all()`
	`alt_irq_enable()`
	`alt_irq_enable_all()`
	`alt_irq_register()`
	`alt_irq_enabled()`
	alt_ic_irq_disable()
	alt_ic_irq_enable()
	alt_ic_isr_register()

alt_ic_isr_register()

Prototype:

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

Commonly called by: Device drivers

Thread-safe: Yes.

Available from ISR: No.

Include: `<sys/alt_irq.h>`

Description: The `alt_ic_isr_register()` function registers an ISR. If the function is successful, the requested interrupt is enabled on return, and `isr` and `isr_context` are inserted in the vector table.

The function arguments are as follows:

- `ic_id` is the interrupt controller ID as defined in **system.h**, identifying the external interrupt controller in the daisy chain. This argument is ignored if the external interrupt controller interface is not implemented.
- `irq` is the IRQ number, as defined in **system.h**, identifying the interrupt to register.
- `isr` is the function that is called when the interrupt is accepted.
- `isr_context` is the input argument to `isr`. `isr_context` points to a data structure associated with the device driver instance.
- `flags` is reserved.

The ISR function prototype is defined as follows:

```
typedef void (*alt_isr_func) (void* isr_context);
```

Calls to `alt_ic_isr_register()` replace previously registered handlers for interrupt `irq`.

If `isr` is set to null, the interrupt is disabled.

- A driver for an EIC must implement this function.

Return: This function returns zero if successful, or nonzero otherwise. The function fails if the `irq` parameter is greater than the maximum interrupt port number supported by the external interrupt controller.

See also:

`alt_irq_disable()`

`alt_irq_disable_all()`

`alt_irq_enable()`

`alt_irq_enable_all()`

`alt_irq_enabled()`

`alt_irq_register()`

alt_ic_irq_disable()

alt_ic_irq_enable()

alt_ic_irq_enabled()

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_instruction_exception_register()

Prototype:

```
void alt_instruction_exception_register (
              alt_exception_result (*handler)
                    ( alt_exception_cause cause,
                      alt_u32            exception_pc,
                      alt_u32            bad_addr ))
```

Commonly called by: C/C++ programs

Device drivers

Thread-safe: No.

Available from ISR: Yes.

Include: `<sys/alt_exceptions.h>`

Description: The HAL API function `alt_instruction_exception_register()` registers an instruction-related exception handler. The `handler` argument is a pointer to the instruction-related exception handler.

You can only use this API function if you have enabled the `hal.enable_instruction_related_exceptions_api` setting in the board support package (BSP). For details, refer to "Settings Managed by the Software Build Tools" in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Register the instruction-related exception handler as early as possible in function `main()`. This allows you to handle abnormal conditions during startup.

You can register an exception handler from the `alt_main()` function.

A call to `alt_instruction_exception_register()` replaces the previously registered exception handler, if any. If `handler` is set to null, the instruction-related exception handler is removed.

For further usage details, refer to the *Exception Handling* chapter of the *Nios II Software Developer's Handbook*.

Return: —

See also: `alt_irq_register()`

alt_exception_cause_generated_bad_addr()

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.

☞ This function is part of the legacy HAL interrupt API, which is deprecated. Altera recommends using the enhanced HAL interrupt API.

For details about using the enhanced HAL interrupt API, refer to “Nios II Interrupt Service Routines” in the *Exception Handling* chapter of the *Nios II Software Developer’s Handbook*.

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_ic_irq_disable()
alt_ic_irq_enable()
alt_ic_irq_enabled()
alt_ic_isr_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 maskable interrupts. Nonmaskable interrupts (NMIs) are unaffected.
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_ic_irq_disable()
	alt_ic_irq_enable()
	alt_ic_irq_enabled()
	alt_ic_isr_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_ic_irq_disable()
	alt_ic_irq_enable()
	alt_ic_irq_enabled()
	alt_ic_isr_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_ic_irq_disable()
	alt_ic_irq_enable()
	alt_ic_irq_enabled()
	alt_ic_isr_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: Determines whether maskable exceptions (`status.PIE`) are enabled.

☞ This function is part of the legacy HAL interrupt API, which is deprecated. Altera recommends using the enhanced HAL interrupt API.

For details about using the enhanced HAL interrupt API, refer to “Nios II Interrupt Service Routines” in the *Exception Handling* chapter of the *Nios II Software Developer’s Handbook*.

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_ic_irq_disable()

alt_ic_irq_enable()

alt_ic_irq_enabled()

alt_ic_isr_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 on 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.

☞ This function is part of the legacy HAL interrupt API, which is deprecated. Altera recommends using the enhanced HAL interrupt API.

For details about using the enhanced HAL interrupt API, refer to "Nios II Interrupt Service Routines" in the *Exception Handling* chapter of the *Nios II Software Developer's Handbook*.

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_ic_irq_disable()

alt_ic_irq_enable()

alt_ic_irq_enabled()

alt_ic_isr_register()

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` in the list `list`. This operation is not reentrant. 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 reentrant. 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`, starting `offset` bytes from the beginning of the flash and are written to the location `dest_addr`. Call this function only 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 on success and nonzero 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()`, that is, while the system is running in single-threaded mode. Concurrent calls to this function might lead to unpredictable results.
Return:	This function returns zero on success; otherwise it returns a negative value. The call can fail if a system clock driver is already registered, or if no system clock device is available.

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 may call the `alt_tick()` function. The driver is responsible for making periodic calls to this function at the rate specified 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. Refer to “Using Timer Devices” in the *Developing Programs Using the Hardware Abstraction Layer* 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 depends 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. Refer to “Using Timer Devices” in the *Developing Programs Using the Hardware Abstraction Layer* 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 depends 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. Refer to "Using Timer Devices" in the *Developing Programs Using the Hardware Abstraction Layer* 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 depends on the underlying driver.

This function resets the counter to zero, and starts the counter running.

Return: The return value is zero on success and nonzero 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 deallocated, that is, 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 to flash. The data to be written is at address src_addr. length bytes are written to the flash fd, offset bytes from the beginning of the flash device address space.

Call this function only when operating in single-threaded mode. This function does not preserve any unwritten areas of any flash sectors affected by this write. Refer to "Using Flash Devices" in the *Developing Programs Using the Hardware Abstraction Layer* 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 on success and nonzero 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 block of data of flash. The data to be written is at address `data`. `length` bytes are written to the flash `fd`, into the block starting at offset `block_offset` from the beginning of the flash address space. The data starts at offset `data_offset` from the beginning of the flash address space.

No check is performed on any of the parameters. Refer to "Using Flash Devices" in the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

Call this function only 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 on success and nonzero 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 thread-safe only 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 on 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

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

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 is 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 nonblocking (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 on success, or –1 otherwise.

If `cmd` is `F_GETFL`, the return value is the current value of the flags. If an error occurs, –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

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

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. Refer to the header file **sys/stat.h** provided with the compiler for the available options.

By default, file descriptors are marked as character devices, unless the underlying driver provides its own implementation of the `fstat()` function.

Calls to `fstat()` are thread-safe only 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 on 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

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

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 time. This time is calculated using the elapsed number of system clock ticks, and the current time value set by the most recent 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 on success. If no system clock is available, the return value is -ENOTSUP.

See also: alt_alarm_start()

alt_alarm_stop()

alt_nticks()

alt_sysclk_init()

alt_tick()

alt_ticks_per_second()

settimeofday()

times()

usleep()

newlib 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.

For file subsystems, `ioctl()` is wrapper function that passes control directly to the appropriate device driver's `ioctl()` function (as registered in the driver's `alt_dev` structure).

For devices, `ioctl()` handles `TIOCEXCL` and `TIOCNXCL` requests internally, without calling the device driver. These requests lock and release a device for exclusive access. For requests other than `TIOCEXCL` and `TIOCNXCL`, `ioctl()` passes control to the device driver's `ioctl()` function.

Calls to `ioctl()` are thread-safe only 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

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 driver's `fstat()` function to determine its reply.

Calls to `isatty()` are thread-safe only 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

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, that is, `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 on success, or –1 otherwise. If the call fails, `errno` is set to indicate the cause of the error.

See also: newlib 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

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`. `lseek()` is wrapper function that passes control directly to the `lseek()` function registered for the driver associated with the file descriptor. If the driver does not provide an implementation of `lseek()`, an error is reported.

`lseek()` corresponds to the standard UNIX `lseek()` function.

You can use the following values for the input parameter, `whence`:

- `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 thread-safe only 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: On success, the return value is a nonnegative 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

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, nonnegative integer for use in read, write, etc.)

`flags` is one of: `O_RDONLY`, `O_WRONLY`, or `O_RDWR`, which request opening the file in read-only, write-only, or read/write mode, respectively.

You can also bitwise-`OR` `flags` with `O_NONBLOCK`, which causes the file to be opened in nonblocking mode. Neither `open()` nor any subsequent operation on the returned file descriptor causes the calling process to wait.

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 thread-safe only 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

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. `read()` is wrapper function that passes control directly to the `read()` function registered for the device driver associated with the 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 thread-safe only 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

The return value is –1 upon 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

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

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 on success. If no system clock is available, the return value is -1, and `errno` is set to `ENOSYS`.

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 thread-safe only 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()`. Refer to "fstat()" on page 14–62.

Return: –

See also: `close()`

`fcntl()`

`fstat()`

`ioctl()`

`isatty()`

`lseek()`

`open()`

`read()`

`write()`

newlib 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 processor time charged for the execution of user instructions
- `tms_stime`: the processor time charged for execution by the system on behalf of the process
- `tms_cutime`: the sum of the values of `tms_utime` and `tms_cutime` for all child processes
- `tms_cstime`: the sum of the values of `tms_stime` and `tms_cstime` for all 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 cannot be spawned by the HAL.

Return: If there is no system clock available, the return value is zero, and `errno` is set to `ENOSYS`.

See also: `alt_alarm_start()`

`alt_alarm_stop()`

`alt_nticks()`

`alt_sysclk_init()`

`alt_tick()`

`alt_ticks_per_second()`

`gettimeofday()`

`settimeofday()`

`usleep()`

newlib 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

usleep()

Prototype: `int usleep (unsigned 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 on 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: On 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

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. `write()` is wrapper function that passes control directly to the `write()` function registered for the device driver associated with the 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 thread-safe only 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. The return value is –1 upon 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

HAL 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 14–2 describes these types, which are defined in the header file **alt_types.h**.

Table 14–2. HAL 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.

Table 14–2. HAL Standard Types

Type	Description
alt_64	Signed 64-bit integer.
alt_u64	Unsigned 64-bit integer.

Document Revision History

Table 14–3 shows the revision history for this document.

Table 14–3. Document Revision History

Date	Version	Changes
May 2011	11.0.0	No change
February 2011	10.1.0	Removed “Referenced Documents” section.
July 2010	10.0.0	■ Clarify purpose of listed C header file for functions. ■ Correction: `alt_irq_enabled()` is not a legacy function.
November 2009	9.1.0	■ Document new enhanced HAL interrupt API functions: `alt_ic_irq_disable()`, `alt_ic_irq_enable()`, `alt_ic_irq_enabled()`, and `alt_ic_isr_register()`. ■ Deprecate legacy HAL interrupt API functions `alt_irq_disable()`, `alt_irq_enable()`, `alt_irq_enabled()`, and `alt_irq_register()`.
March 2009	9.0.0	■ Corrected minor typographical errors.
May 2008	8.0.0	■ Advanced exceptions added to Nios II core. ■ Instruction-related exception handling added to HAL. ■ Added `alt_instruction_exception_register()` and `alt_exception_cause_generated_bad_addr()` for instruction-related exception handlers.
October 2007	7.2.0	Maintenance release.
May 2007	7.1.0	■ Added table of contents to “Introduction” section. ■ Added Referenced Documents section.
March 2007	7.0.0	Maintenance release.
November 2006	6.1.0	Function `open()` requires `fcntl.h`.
May 2006	6.0.0	Maintenance release.
October 2005	5.1.0	Added API entries for “alt_irq_disable()” and “alt_irq_enable()”, which were previously omitted by error.
May 2005	5.0.0	■ Added `alt_load_section()` function. ■ Added `fcntl()` function.
December 2004	1.2	Updated names of DMA generic requests.
September 2004	1.1	■ 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	1.0	Initial release.

15. Nios II Software Build Tools Reference

NII52016-13.1.0

This chapter provides a complete reference of all available commands, options, and settings for the Nios® II Software Build Tools (SBT). This reference is useful for developing your own embedded software projects, packages, or device drivers.

Before using this chapter, read the *Getting Started from the Command Line* chapter of the *Nios II Software Developer's Handbook*, and familiarize yourself with the parts of 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 15–1
- "Nios II Design Example Scripts" on page 15–31
- "Settings Managed by the Software Build Tools" on page 15–34
- "Application and User Library Makefile Variables" on page 15–73
- "Software Build Tools Tcl Commands" on page 15–76
- "Software Build Tools Path Names" on page 15–122

Nios II Software Build Tools Utilities

The build tools utilities are an entry point to the Nios II SBT. Everything you can do with the tools, such as specifying settings, creating makefiles, and building projects, is made available by the utilities.

All Nios II SBT utilities share the following behavior:

- 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`.
- 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 15–1 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 15–1. Nios II SBT 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 SBT 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 15–2 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 15–2. Selecting Logging Level

Command-Line Option	Logging Level	Comments
none	default	If there is no command-line option, the default level is selected.
`--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 Values

The value of a setting is specified with the `--set` command-line option to **nios2-bsp-create-settings** or **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.

For more information about settings values and formats, refer to “Settings Managed by the Software Build Tools” on page 15–34.

Utility and Script Summary

The following command-line utilities and scripts are available:

- "nios2-app-generate-makefile"
- "nios2-bsp-create-settings" on page 15–6
- "nios2-bsp-generate-files" on page 15–8
- "nios2-bsp-query-settings" on page 15–9
- "nios2-bsp-update-settings" on page 15–11
- "nios2-lib-generate-makefile" on page 15–13
- "nios2-bsp-editor" on page 15–15
- "nios2-app-update-makefile" on page 15–16
- "nios2-lib-update-makefile" on page 15–19
- "nios2-swexample-create" on page 15–22
- "nios2-elf-insert" on page 15–23
- "nios2-elf-query" on page 15–24
- "nios2-bsp" on page 15–28
- "nios2-bsp-console" on page 15–30

nios2-app-generate-makefile

Usage

```
nios2-app-generate-makefile [--app-dir <directory>]
  --bsp-dir <directory> [--debug]
  [--elf-name <filename>] [--extended-help] [--help]
  [--log <filename>] [--no-src] [--set <name> <value>]
  [--silent] [--src-dir <directory>]
  [--src-files <filenames>] [--src-rdir <directory>]
  [--use-lib-dir <directory>] [--verbose]
  [--version]
```

Options

- `--app-dir` *<directory>*: Directory to place the application makefile and executable and linking format file (.elf). If omitted, it defaults to the current directory.
- `--bsp-dir` *<directory>*: Specifies the path to the BSP generated files directory (populated using the **nios2-bsp-generate-files** command).
- `--debug`: Output debug, exception traces, verbose, and default information about the command’s operation to `stdout`.
- `--elf-name` *<filename>*: Name of the .elf file 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>*: Create a debug log and write to specified file. Also logs debug information to `stdout`.
- `--no-src`: Allows no sources files to be set in the Makefile. You must add source files in manually before compiling
- `--set` *<name> <value>*: Set 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`: Suppress information about the command’s operation normally sent to `stdout`.
- `--src-dir` *<directory>*: Searches for source files in <*directory*>. Use . to look in the current directory. Multiple `--src-dir` options are allowed.
- `--src-files` *<filenames>*: Adds a list of space-separated source file names 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 search for source files in or under <*directory*>. Multiple `--src-rdir` options are allowed and can be freely mixed with `--src-dir` options.
- `--use-lib-dir` *<directory>*: Specifies the path to a dependent user library directory. The user library directory must contain a makefile fragment called **public.mk**. Multiple `--use-lib-dir` options are allowed.

- `--verbose`: Output 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. If no command-line arguments are specified, this command returns an exit value of 1 and sends a help message to `stderr`.

For detailed information about installing the Altera Complete Design Suite, refer to the *Altera Software Installation and Licensing Manual*.

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 [--bsp-dir <directory>]
  [--cmd <tcl command>] [--cpu-name <cpu name>]
  [--debug] [--extended-help] [--get-cpu-arch]
  [--help] [--jdi <filename>]
  [--librarian-factory-path <directory>]
  [--librarian-path <directory>] [--log <filename>]
  [--script <filename>] [--set <name> <value>]
  --settings <filename> [--silent]
  --sopc <filename> --type <OS name> [--type-version <version>]
  [--verbose] [--version]
```

Options

- `--bsp-dir` <*directory*>: Path to the directory where the BSP files are generated. Use . for the current directory. The directory <*directory*> must exist. This command overwrites preexisting files in <*directory*> without warning.
- `--cmd` <*tcl command*>: Runs the specified Tcl command. Multiple `--cmd` options are allowed. Available Tcl commands are described in "Tcl Commands for BSP Settings" on page 15–76.
- `--cpu-name` <*cpu name*>: The name of the Nios II processor that the BSP supports. Optional for a single-processor system. Use `?` to list available Nios II processor names.
- `--debug`: Sends debug information, exception traces, verbose output, and default information about the command's operation, to `stdout`.
- `--extended-help`: Displays full information about this command and its options. Also displays Tcl command help for the `--cmd` and `--script` options.
- `--get-cpu-arch`: Queries for processor architecture from the processor specified. Does not create a BSP.
- `--help`: Displays basic information about this command and its options.
- `--jdi` <*filename*>: The location of the JTAG Debugging Information File (**.jdi**) generated by the Quartus® II software. The **.jdi** file specifies the name-to-node mappings for the JTAG chain elements. The tool inserts the JTAG Debugging Information File (**.jdi**) path in **public.mk**. If no **.jdi** path is specified, the command searches the directory containing the SOPC Information File (**.sopcinfo**), and uses the first **.jdi** file found.
- `--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*>: Run the specified Tcl script with optional arguments. Multiple `--script` options are allowed. Scripts can use the Tcl commands described in "Tcl Commands for BSP Settings" on page 15–76.

- `--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. This file is created 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 **.sopcinfo file** used to create the BSP.
- `--type` *<OS name>*: BSP type. Use `?` or `types` to list available BSP types for this option. Use `names` to list the display names of available BSP types. For a Nios II DPX system, always set this argument to `lwhal`.
- `--type-version` *<version>*: BSP software component version. By default the latest version is used. `default` value can be used to reset to this default behavior. Use `?` to list available BSP types and versions.
- `--verbose`: Sends verbose output, 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** script.

If no command-line arguments are specified, this command returns an exit value of 1 and sends a help message to `stderr`.

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.sopcinfo --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 the BSP files are generated. Use . for the current directory. The directory <*directory*> must exist. This command overwrites preexisting files in <*directory*> 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 populates the files in a BSP directory. The path to an existing .**bsp** file and the path to the BSP directory are mandatory command-line options. Files are written to the specified BSP directory. Generated files are created unconditionally. Copied files are copied from the Nios II EDS installation folder only if they are not present in the BSP directory, or if the existing files differ from the installation files.

If no command-line arguments are specified, this command returns an exit value of 1 and sends a help message to `stderr`.

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>*: Run the specified Tcl command. Multiple `--cmd` options are allowed.
- `--debug`: Output 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>*: Display the value of the setting called *<name>*. Multiple `--get` options are allowed. Each value appears on its own line in the order the `--get` options are specified. Mutually exclusive with the `--get-all` option.
- `--get-all`: Display 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>*: Create a debug log and write to specified file. Also logs debug information to `stdout`.
- `--script` *<filename>*: Run 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`: Suppress information about the command's operation normally sent to `stdout`.
- `--verbose`: Output 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 .bsp file. The path to an existing .bsp file is a mandatory command-line option. The command does not modify the settings file. Only requested information is displayed on `stdout`; no informational messages are displayed.

If no command-line arguments are specified, this command returns an exit value of 1 and sends a help message to `stderr`.

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 [--bsp-dir <directory>]
  [--cmd <tcl command>] [--cpu-name <cpu name>]
  [--debug] [--extended-help] [--help] [--jdi <filename>]
  [--librarian-factory-path <directory>]
  [--librarian-path <directory>] [--log <filename>]
  [--script <filename>] [--set <name> <value>]
  --settings <filename> [--silent]
  [--sopc <filename>] [--verbose] [--version]
```

Options

- `--bsp-dir` *<directory>*: Path to the directory where the BSP files are generated. Use . for the current directory. The directory <*directory*> must exist.
- `--cmd` *<tcl command>*: Run 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 hardware design contains multiple Nios II processors. Optional for a single-processor design.
- `--debug`: Output 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.
- `--jdi` *<filename>*: The location of the **.jdi** file generated by the Quartus II software. The **.jdi** file specifies the name-to-node mappings for the JTAG chain elements. The tool inserts the **.jdi** path in **public.mk**. If no **.jdi** path is specified, the command searches the directory containing the **.sopcinfo** file, and uses the first **.jdi** file found.
- `--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>*: Create a debug log and write to specified file. Also logs debug information to `stdout`.
- `--script` *<filename>*: Run the specified Tcl script with optional arguments. Multiple `--script` options are allowed.
- `--set` *<name> <value>*: Set the setting called <*name*> to <*value*>. Multiple `--set` options are allowed.
- `--settings` *<filename>*: File name of an existing BSP settings file to update.
- `--silent`: Suppress information about the command's operation normally sent to `stdout`.
- `--sopc` *<filename>*: The **.sopcinfo** file to update the BSP with. It is recommended to create a new BSP if the design has changed significantly. This argument is useful if the path to the original **.sopcinfo** file has changed.

- `--verbose`: Output 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 .**bsp** file. The path to an existing .**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 use the `--script` option to pass the default Tcl script to the **nios2-bsp-update-settings** command to make sure that your BSP is consistent with your system (this is what the **nios2-bsp** command does).

If no command-line arguments are specified, this command returns an exit value of 1 and sends a help message to `stderr`.

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>] [--no-src]
  [--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`: Output 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 user library archive file (**.a**), the user library makefile, and **public.mk**. If omitted, it defaults to the current directory.
- `--lib-name` *<filename>*: Name of the user library being created. The user library file name is the user library name with a **lib** prefix and **.a** suffix added. Do not include these in the user library name itself. If the user library name option is omitted, the user library name defaults to the name of the first source file with its extension removed.
- `--log` *<filename>*: Create a debug log and write to specified file. Also logs debug information to `stdout`.
- `--no-src`: Allows no sources files to be set in the Makefile. You must add source files manually before compiling.
- `--public-inc-dir` *<directory>*: Path to a directory that contains C header files (**.h)** that are to be made available (that is, public) to users of the user library. This directory is added to the appropriate variable in **public.mk**. Multiple `--public-inc-dir` options are allowed.
- `--set` *<name> <value>*: Set 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`: Suppress information about the command's operation normally sent to `stdout`.
- `--src-dir` *<directory>*: Search for source files in <*directory*>. 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 search for source files in or under <*directory*>. Multiple `--src-rdir` options are allowed and can be freely mixed with `--src-dir` options.
- `--use-lib-dir` <*directory*>: Path to a dependent user library directory. The user library directory must contain a makefile fragment called **public.mk**. Multiple `--use-lib-dir` options are allowed.
- `--verbose`: Output 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 user library makefile (called Makefile). The path to a BSP created by **nios2-bsp-generate-files** is an optional command-line option.

If no command-line arguments are specified, this command returns an exit value of 1 and sends a help message to `stderr`.

For more details about this command, use the `--extended-help` option to display comprehensive usage information.

nios2-bsp-editor

Usage

```
nios2-bsp-editor [--extended-help]
  [--fontsize <point size>] [--help]
  [--librarian-factory-path <directory>]
  [--librarian-path <directory>] [--log <filename>]
  [--settings <filename>] [--version]
```

Options

- `--extended-help`: Displays full information about this command and its options.
- `--fontsize` *`<point size>`*: The default point size for GUI fonts is 11. Use this option to adjust the point size.
- `--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>`*: Create a debug log and write to specified file.
- `--settings` *`<filename>`*: File name of an existing BSP settings file to update.
- `--version`: Displays the version of this command and exits with a zero exit status.

Description

The **nios2-bsp-editor** command is a GUI application for creating and editing board support packages for Nios II designs.

For more details about this command, use the `--extended-help` option to display comprehensive usage information.

nios2-app-update-makefile

Usage

```
nios2-app-update-makefile --app-dir <directory>
  [--add-lib-dir <directory>] [--add-src-dir <directory>]
  [--add-src-files <filenames>] [--add-src-rdir <directory>] [--debug]
  [--extended-help] [--force] [--get <name>] [--get-all]
  [--get-asflags] [--get-bsp-dir] [--get-debug-level]
  [--get-defined-symbols] [--get-elf-name] [--get-optimization]
  [--get-undefined-symbols] [--get-user-flags] [--get-warnings]
  [--help] [--list-lib-dir] [--list-src-files] [--lock]
  [--log <filename>] [--no-src] [--remove-lib-dir <directory>]
  [--remove-src-dir <directory>] [--remove-src-files <filenames>]
  [--remove-src-rdir <directory>] [--set <name>]
  [--set-asflags <value>] [--set-bsp-dir <directory>]
  [--set-debug-level <value>] [--set-defined-symbols <value>]
  [--set-elf-name <name>] [--set-optimization <value>]
  [--set-undefined-symbols <value>] [--set-user-flags <value>]
  [--set-warnings <value>] [--show-managed-section] [--show-names]
  [--silent] [--unlock] [--verbose] [--version]
```

Options

- `--app-dir` *<directory>*: Path to the Application Directory with the generated makefile.
- `--add-lib-dir` *<directory>*: Add a path to dependent user library directory
- `--add-src-dir` *<directory>*: Add source files in *<directory>*. Use . to look in the current directory. Multiple `--add-src-dir` options are allowed.
- `--add-src-files` *<filenames>*: A list of space-separated source file names to be 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.
- `--add-src-rdir` *<directory>*: Same as `--add-src-dir` option but recursively search for source files in or under *<directory>*. Multiple `--add-src-rdir` options are allowed and can be freely mixed with `--src-dir` options.
- `--debug`: Output 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.
- `--force`: Update the Makefile even if it's locked
- `--get` *<name>*: Get the values of Makefile variables
- `--get-all`: Get all variables in the managed section of the Makefile
- `--get-asflags`: Get user assembler flags
- `--get-bsp-dir`: Get the BSP generated files directory
- `--get-debug-level`: Get debug level flag
- `--get-defined-symbols`: Get defined symbols flag
- `--get-elf-name`: Get the name of .elf file
- `--get-optimization`: Get optimization flag
- `--get-undefined-symbols`: Get undefined symbols flag

- `--get-user-flags`: Get user flags
- `--get-warnings`: Get warnings flag
- `--help:` Displays basic information about this command and its options.
- `--list-lib-dir`: List all paths to dependent user library directories
- `--list-src-files`: List all source files in the makefile.
- `--lock:` Lock the Makefile to prevent it from being updated
- `--log` *<filename>*: Create a debug log and write to specified file. Also logs debug information to `stdout`.
- `--no-src`: Remove all source files in the makefile
- `--remove-lib-dir` *<directory>*: Remove a path to dependent user library directory
- `--remove-src-dir` *<directory>*: Remove source files in *<directory>*. Use . to look in the current directory. Multiple `--remove-src-dir` options are allowed.
- `--remove-src-files` *<filenames>*: A list of space-separated source file names to be removed from 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.
- `--remove-src-rdir` *<directory>*: Same as `--remove-src-dir` option but recursively search for source files in or under *<directory>*. Multiple `--remove-src-rdir` options are allowed and can be freely mixed with `--src-dir` options.
- `--set` *<name> <value>*: Set the value of a Makefile variable called *<name>*
- `--set-asflags` *<value>*: Set user assembler flags
- `--set-bsp-dir` *<directory>*: Set the BSP generated files directory
- `--set-debug-level` *<value>*: Set debug level flag
- `--set-defined-symbols` *<value>*: Set defined symbols flag
- `--set-elf-name` *<name>*: Set the name of .elf file
- `--set-optimization` *<value>*: Set optimization flag
- `--set-undefined-symbols` *<value>*: Set undefined symbols flag
- `--set-user-flags` *<value>*: Set user flags
- `--set-warnings` *<value>*: Set warnings flag
- `--show-managed-section`: Show the managed section in the Makefile
- `--show-names`: Show name of the variables
- `--silent`: Suppress information about the command's operation normally sent to `stdout`.
- `--unlock`: Unlock the Makefile
- `--verbose`: Output 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-update-makefile** command updates an application makefile to add or remove source files.

If no command-line arguments are specified, this command returns an exit value of 1 and sends a help message to `stderr`.

For more details about this command, use the `--extended-help` option to display comprehensive usage information.

☞ The `--add-src-dir`, `--add-src-rdir`, `--remove-src-dir`, and `--remove-src-rdir` options add and remove files found in <*directory*> at the time the command is executed. Files subsequently added to or removed from the directory are not reflected in the makefile.

nios2-lib-update-makefile

Usage

```
nios2-lib-update-makefile --lib-dir <directory>
  [--add-lib-dir <directory>] [--add-public-inc-dir <directory>]
  [--add-src-dir <directory>] [--add-src-files <filenames>]
  [--add-src-rdir <directory>] [--debug] [--extended-help] [--force]
  [--get <name>] [--get-all] [--get-asflags] [--get-bsp-dir]
  [--get-debug-level] [--get-defined-symbols] [--get-lib-name]
  [--get-optimization] [--get-undefined-symbols] [--get-user-flags]
  [--get-warnings] [--help] [--list-lib-dir] [--list-public-inc-dir]
  [--list-src-files] [--lock] [--log <filename>] [--no-src]
  [--remove-lib-dir <directory>] [--remove-public-inc-dir <directory>]
  [--remove-src-dir <directory>] [--remove-src-files <filenames>]
  [--remove-src-rdir <directory>] [--set <name> <value>]
  [--set-asflags <value>] [--set-bsp-dir <directory>]
  [--set-debug-level <value>] [--set-defined-symbols <value>]
  [--set-lib-name <name>] [--set-optimization <value>]
  [--set-undefined-symbols <value>] [--set-user-flags <value>]
  [--set-warnings <value>] [--show-managed-section] [--show-names]
  [--silent] [--unlock] [--verbose] [--version]
```

Options

- `--add-lib-dir` *<directory>*: Add a path to dependent user library directory
- `--add-public-inc-dir` *<directory>*: Add a directory that contains C-language header files
- `--add-src-dir` *<directory>*: Add source files in <*directory*>. Use . to look in the current directory. Multiple `--add-src-dir` options are allowed.
- `--add-src-files` *<filenames>*: A list of space-separated source file names to be 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.
- `--add-src-rdir` *<directory>*: Same as `--add-src-dir` option but recursively search for source files in or under <*directory*>. Multiple `--add-src-rdir` options are allowed and can be freely mixed with --src-dir options.
- `--debug`: Output 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.
- `--force`: Update the Makefile even if it is locked
- `--get` *<name>*: Get the values of Makefile variables
- `--get-all`: Get all variables in the managed section of the Makefile
- `--get-asflags`: Get user assembler flags
- `--get-bsp-dir`: Get the BSP generated files directory
- `--get-debug-level`: Get debug level flag
- `--get-defined-symbols`: Get defined symbols flag
- `--get-lib-name`: Get the name of user library
- `--get-optimization`: Get optimization flag

- `--get-undefined-symbols`: Get undefined symbols flag
- `--get-user-flags`: Get user flags
- `--get-warnings:` Get warnings flag
- `--help:` Displays basic information about this command and its options.
- `--list-lib-dir`: List all paths to dependent user library directories
- `--list-public-inc-dir`: List all public include directories
- `--list-src-files`: List all source files in the makefile.
- `--lock:` Lock the Makefile to prevent it from being updated
- `--log` *`<filename>`*: Create a debug log and write to specified file. Also logs debug information to `stdout`.
- `--no-src:` Remove all source files
- `--remove-lib-dir` *`<directory>`*: Remove a path to dependent user library directory
- `--remove-public-inc-dir` *`<directory>`*: Remove a include directory
- `--remove-src-dir` *`<directory>`*: Remove source files in <*directory*>. Use . to look in the current directory. Multiple `--remove-src-dir` options are allowed.
- `--remove-src-files` *`<filenames>`*: A list of space-separated source file names to be removed from 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.
- `--remove-src-rdir` *`<directory>`*: Same as `--remove-src-dir` option but recursively search for source files in or under <*directory*>. Multiple `--remove-src-rdir` options are allowed and can be freely mixed with --src-dir options.
- `--set` *`<name>`* *`<value>`*: Set the value of a Makefile variable called <*name*>
- `--set-asflags` *`<value>`*: Set user assembler flags
- `--set-bsp-dir` *`<directory>`*: Set the BSP generated files directory
- `--set-debug-level` *`<value>`*: Set debug level flag
- `--set-defined-symbols` *`<value>`*: Set defined symbols flag
- `--set-lib-name` *`<name>`*: Set the name of user library
- `--set-optimization` *`<value>`*: Set optimization flag
- `--set-undefined-symbols` *`<value>`*: Set undefined symbols flag
- `--set-user-flags` *`<value>`*: Set user flags
- `--set-warnings` *`<value>`*: Set warnings flag
- `--show-managed-section`: Show the managed section in the Makefile
- `--show-names:` Show name of the variables
- `--silent:` Suppress information about the command’s operation normally sent to `stdout`.
- `--unlock:` Unlock the Makefile

- `--verbose`: Output 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-update-makefile** command updates a user library makefile to add or remove source files.

If no command-line arguments are specified, this command returns an exit value of 1 and sends a help message to `stderr`.

For more details about this command, use the `--extended-help` option to display comprehensive usage information.

☞ The `--add-src-dir`, `--add-src-rdir`, `--remove-src-dir`, and `--remove-src-rdir` options add and remove files found in <*directory*> at the time the command is executed. Files subsequently added to or removed from the directory are not reflected in the makefile.

nios2-swexample-create

Usage

```
nios2-create-swexample [--name] --sopc-dir --type [--list] [--app-dir]
  [--bsp-dir] [--no-app] [--no-bsp] [--elf-name] [--describe]
  [--describeX] [--describeAll] [--search] [--help] [--silent]
  [--version]
```

Options

- `--name`: Specify the name of the software project to create.
- `--sopc-dir`: Specify the hardware design directory. Required.
- `--type`: Specify the software design example template type. Required.
- `--list`: List all valid software design example template types.
- `--app-dir`: Specify the application directory to create. Default: <*sopc-dir*>**/software_examples/app/**<*name*>
- `--bsp-dir`: Specify the bsp directory to create. Default: <*sopc-dir*>**/software_examples/bsp/**<*bsp-type*>
- `--no-app`: Do not generate the **create-this-app** script
- `--no-bsp`: Do not generate the **create-this-bsp** script
- `--elf-name`: Name of the .elf file to create.
- `--describe`: Describe the software example type specified and exit.
- `--describeX`: Verbosely describe the software example type specified and exit.
- `--describeAll`: Describe all the software example types and exit.
- `--search`: Search for software example templates in the specified directory. Default: <*Nios II EDS install path*>**/examples/software**
- `--help`: Displays basic information about this command and its options.
- `--silent`: Do not echo commands.
- `--version`: Print the version number of nios2-create-swexample and exit.

Description

This utility creates a template software example for a given SOPC system.

If no command-line arguments are specified, this command returns an exit value of 1 and sends a help message to `stderr`.

nios2-elf-insert

Usage

```
nios2-elf-insert <filename> [--cpu_name <cpuName>]
  [--stderr_dev <stderrDev>] [--stdin_dev <stdinDev>]
  [--stdout_dev <stdoutDev>] [--sidp <sysidBase>] [--id <sysidHash>]
  [--timestamp <sysidTime>] [--sof <sofFile>]
  [--sopcinfo <sopcinfoFile>] [--jar <jarFile>] [--jdi <jdiFile>]
  [--quartus_project_dir <quartusProjectDir>]
  [--sopc_system_name <sopcSystemName>]
  [--profiling_enabled <profilingEnabled>]
  [--simulation_enabled <simulationEnabled>]
```

Options

- *<filename>*: the ELF filename
- `--cpu_name` *<cpuName>*
- `--stderr_dev` *<stderrDev>*
- `--stdin_dev` *<stdinDev>*
- `--stdout_dev` *<stdoutDev>*
- `--sidp` *<sysidBase>*
- `--id` *<sysidHash>*
- `--timestamp` *<sysidTime>*
- `--sof` *<sofFile>*
- `--sopcinfo` *<sopcinfoFile>*
- `--jar` *<jarFile>*
- `--jdi` *<jdiFile>*
- `--quartus_project_dir` *<quartusProjectDir>*
- `--sopc_system_name` *<sopcSystemName>*
- `--profiling_enabled` *<profilingEnabled>*
- `--simulation_enabled` *<simulationEnabled>*

If no command-line arguments are specified, this command returns an exit value of 1 and sends a help message to `stderr`.

nios2-elf-query

Usage

```
nios2-elf-query <filename> [--cpu_name] [--stderr_dev] [--stdin_dev]
  [--stdout_dev] [--sidp] [--id] [--timestamp] [--sof] [--sopcinfo]
  [--jar] [--jdi] [--quartus_project_dir] [--sopc_system_name]
  [--profiling_enabled] [--simulation_enabled]
```

Options

- *<filename>*: the ELF filename
- --cpu_name
- --stderr_dev
- --stdin_dev
- --stdout_dev
- --sidp
- --id
- --timestamp
- --sof
- --sopcinfo
- --jar
- --jdi
- --quartus_project_dir
- --sopc_system_name
- --profiling_enabled
- --simulation_enabled

If no command-line arguments are specified, this command returns an exit value of 1 and sends a help message to stderr.

nios2-flash-programmer-generate

Usage

```
nios2-flash-programmer-generate [--accept-bad-sysid]
  [--add-bin <fname> <flash-slave-desc> <offset>]
  [--add-elf <fname> <flash-slave-desc> <extra-elf2flash-arguments>]
  [--add-sof <fname> <flash-slave-desc> <offset>
   <extra-sof2flash-arguments>]
  [--cable <cable name>] [--cpu <processor_name>] [--debug]
  [--device <device name>] [--erase-first] [--extended-help]
  --flash-dir <directory> [--go] [--help] [--id <address>]
  [--instance <instance value>] [--log <filename>] [--mmu]
  [--program-flash] [--script-dir <directory>] [--sidp <address>]
  [--silent] --sopcinfo <filename> [--verbose] [--version]
```

Options

- `--accept-bad-sysid`: Continue even if the system identifier (ID) comparison fails.
- `--add-bin` *<fname> <flash-slave-desc> <offset>*: Specify a binary file to convert and program. The filename, target flash slave descriptor, and target offset amount are required. This option can be used multiple times for SRAM Object Files (.**sof)**.
- `--add-elf` *<fname> <flash-slave-desc> <extra-elf2flash-arguments>*: Specify a **.elf** file to convert and program. The filename and target flash slave descriptor are required. This option can be used multiple times for **.elf** files. *<extra-elf2flash-arguments>* can be any of the following options supported by **elf2flash**:
  - `save`
  - `sim_optimize`

  The following **elf2flash** options have default values computed, but are also supported as *<extra-elf2flash-arguments>* for manual override of those defaults:
  - `base`
  - `boot`
  - `end`
  - `reset`
- `--add-sof` *<fname> <flash-slave-desc> <offset> <extra-sof2flash-arguments>*: Specify a **.sof** file to convert and program. The filename, target flash slave descriptor, and target offset arguments are required. This option can be used multiple times for **.sof** files. *<extra-sof2flash-arguments>* can be any of the following options supported by **sof2flash**:
  - `activeparallel`
  - `compress`
  - `save`
  - `timestamp`
  - `options`
- `--cable` *<cable name>*: Specifies which JTAG cable to use (not needed if you only have one cable). Not used without `--program-flash` option.

- `--cpu <processor_name>`: The Nios II processor name from the **.sopcinfo** file. Not required if only one Nios II processor in the system.
- `--debug`: Sends debug information, exception traces, verbose output, and default information about the command's operation, to stdout.
- `--device <device name>`: Specifies in which device you want to look for the Nios II debug core. Device 1 is the device nearest TDI. Not used without `--program-flash` option.
- `--erase-first`: Erase entire flash targets before programming them. Not used without `--program-flash` option.
- `--extended-help`: Displays full information about this command and its options.
- `--flash-dir <directory>`: Path to the directory where the flash files are generated. Use . for the current directory. This command overwrites pre-existing files in <*directory*> without warning.
- `--go`: Run processor from reset vector after program.
- `--help`: Displays basic information about this command and its options.
- `--id <address>`: Unique ID code for target system. Not used without `--program-flash` option.
- `--instance <instance value>`: Specifies the INSTANCE value of the debug core (not needed if there is exactly one on the chain). Not used without `--program-flash` option.
- `--log <filename>`: Creates a debug log and write to specified file. Also logs debug information to stdout.
- `--mmu`: Specifies if the processor with the corresponding INSTANCE value has an MMU (not needed if there is exactly one processor in the system). Not used without `--program-flash` option.
- `--program-flash`: Providing this flag causes calls to **nios2-flash-programmer** to be generated and executed. This results in flash targets being programmed.
- `--script-dir <directory>`: Path to the directory where a shell script of this tool's executed command lines is generated. This script can be used in place of this **nios2-flash-programmer-generate** command. Use . for the current directory. This command overwrites pre-existing files in <*directory*> without warning.
- `--sidp <address>`: Base address of system ID peripheral on the target. Not used without `--program-flash` option.
- `--silent`: Suppresses information about the command's operation normally sent to stdout.
- `--sopcinfo <filename>`: The **.sopcinfo** file.
- `--verbose`: Sends verbose output, 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-flash-programmer-generate** command converts multiple files to a **.flash** in Motorola S-record format, and programs them to the designated target flash devices (`--program-flash`). This is a convenience utility that manages calls to the following command line utilities

- **bin2flash**
- **elf2flash**
- **sof2flash**
- **nios2-flash-programmer**

This utility also generates a script that captures the sequence of conversion and flash programmer commands.

If no command-line arguments are specified, this command returns an exit value of 1 and sends a help message to `stderr`.

Example

```
nios2-flash-programmer-generate --sopcinfo=C:\my_design.sopcinfo \
  --flash-dir=. \
  --add-sof C:\my_design\test.sof 0x0C000000 memory_0 compress save \
  --add-elf C:\my_app\my_app.elf 0x08000000 memory_0 \
  --program-flash
```

nios2-bsp

Usage

```
nios2-bsp <bsp-type> <bsp-dir> [<sopc>] [<override>]...
```

Options

- `<bsp-type>`: `hal` or `ucosii`.
- `<bsp-dir>`: Path to the BSP directory.
- `<sopc>`: The path to the **.sopcinfo** file or its directory.
- `<override>`: Options to override defaults.

Description

The **nios2-bsp** 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 <*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 hardware 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 is installed 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. The **nios2-bsp** script 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 **.sopcinfo** file. Alternatively, specify a directory containing a **.sopcinfo** file. In the latter case, the tool finds a file with the extension **.sopcinfo**. This argument is ignored when updating a BSP. If you omit this argument, it defaults to the current directory.
- *<override>*: Options to override defaults. The **nios2-bsp** script passes most overrides 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 overrides 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 preexisting 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.

If no command-line arguments are specified, this command returns an exit value of 1 and sends a help message to `stderr`.

nios2-bsp-console

Usage

```
nios2-bsp-console [--cmd <tcl> <command>] [--extended-help] [--gui]
  [--help] [--script <filename>] [--version]
```

Options

- `--cmd` *`<tcl> <command>`*: Runs the specified Tcl command. Multiple `--cmd` options are allowed. Available Tcl commands are listed in "Tcl Commands for BSP Settings" on page 15–76.
- `--extended-help`: Displays full information about this command and its options. Lists Altera BSP Tcl command help for the `--cmd` and `--script` options
- `--gui`: This option opens a Tcl console for creating, editing, and generating Altera BSPs.
- `--help`: Displays basic information about this command and its options.
- `--script` *`<filename>`*: Run the specified Tcl script with optional arguments. Multiple `--script` options are allowed. Available Tcl commands are listed in "Tcl Commands for BSP Settings" on page 15–76.
- `--version`: Displays the version of this command and exits with a zero exit status.

Description

When invoked with no arguments, **nios2-bsp-console** starts an interactive command-line Tcl interpreter for creating, editing, and generating Altera BSPs. Available Tcl commands are listed in "Tcl Commands for BSP Settings" on page 15–76.

Nios II Design Example Scripts

The Nios II SBT includes scripts that allow you to create sample BSPs and applications. This section describes each script and its location in the design example directory structure. Each hardware design example 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 15–3 lists all potential BSP examples provided in the **bsp** directory. The **bsp** directory for each hardware example only includes BSP examples supported by the associated hardware example.

Table 15–3. BSP Examples

Example BSP *(1)*	Summary
`hal_reduced_footprint`	Hardware abstraction layer (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 15–3:

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

The **app** subdirectory contains a separate subdirectory for each software example supported by the hardware example, as listed in Table 15–4.

Table 15–4. 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
Hello Freestanding	Prints `"Hello from Nios II"` from a free-standing application
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
Memory Test	Runs diagnostic tests on both volatile and flash memory
Memory Test Small	Runs diagnostic tests on volatile memory from a small footprint
Simple Socket Server	Runs a TCP/IP socket server
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 15–4:

(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. The **create-this-bsp** script 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 .**sopcinfo** file. The .**sopcinfo** 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. The **create-this-app** script copies the C/C++ application source code to the current directory, runs **nios2-app-generate-makefile** to create a makefile (named **Makefile**), and then runs `make` to build the Executable and Linking Format File (**.elf)** for your application. Each **create-this-app** script uses a particular example BSP. For further information, refer to the script to determine the associated example BSP. If the BSP does not exist when **create-this-app** runs, **create-this-app** calls 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 design examples. You can easily find them from the following information:

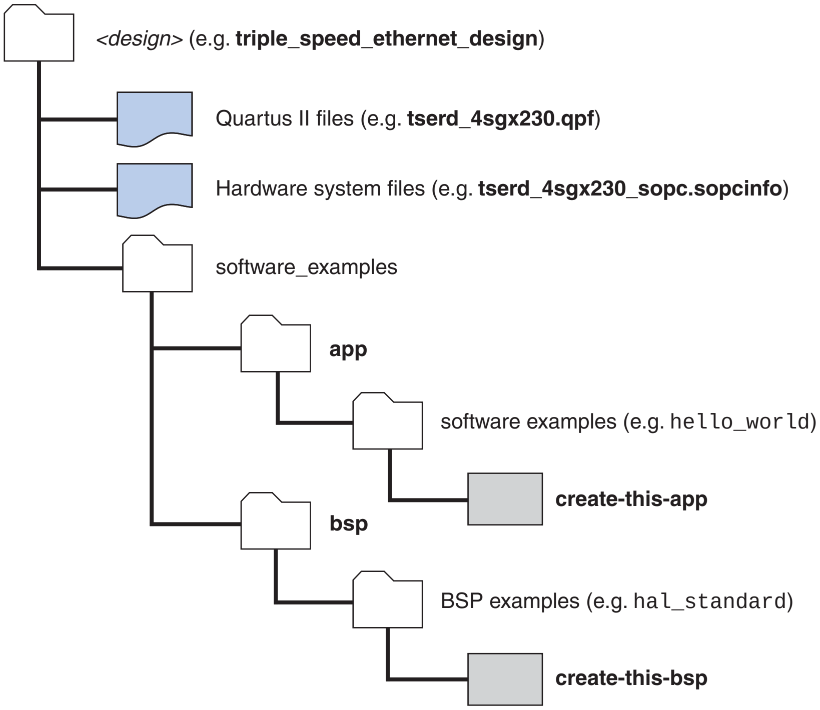
- Where the Nios II EDS is installed
- Which Altera development board you are using
- Which HDL you are using
- Which Nios II hardware design example you are using
- The name of the Nios II software example

The **create-this-app** script for each software design example 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 triple-speed ethernet design example for the Stratix® IV GX FPGA Development Kit might be located in **c:/altera/100/nios2eds/examples/verilog/niosII_stratixIV_4sgx230/triple_speed_ethernet_design/software_examples/app/hello_world**.

Similarly, the **create-this-bsp** script for each software design example 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 triple-speed ethernet design example for the Stratix IV GX FPGA Development Kit might be located in **c:/altera/100/nios2eds/examples/verilog/niosII_stratixIV_4sgx230/triple_speed_ethernet_design/software_examples/bsp/hal_default**.

Figure 15–1 shows the directory structure under each hardware design example.

Figure 15–1. Software Design Example Directory Structure


<design> (e.g. triple_speed_ethernet_design)
Quartus II files (e.g. tserd_4sgx230.qpf)
Hardware system files (e.g. tserd_4sgx230_sopc.sopcinfo)
software_examples
app
software examples (e.g. hello_world)
create-this-app
bsp
BSP examples (e.g. hal_standard)
create-this-bsp

Settings Managed by the Software Build Tools

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 the device drivers and other packages that are included.

Every example in the *Getting Started from the Command Line* and *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 packages. It does not include settings for device drivers or software packages 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 SBT are organized hierarchically, for logical grouping and to avoid name 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

A BSP setting consists of a name/value pair.

The format in which you specify the setting value depends on the setting type. Several settings types are supported. Table 15–5 shows the allowed formats for each setting type.

Table 15–5. 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 empty string. In the SBT command line, to specify a string value with embedded spaces, surround the string with a backslash-double-quote sequence (`\"`). For example: `--set APP_INCLUDE_DIRS \"lcd board\"`	Quoted string

There are several contexts for BSP settings, as shown in Table 15–6.

Table 15–6. BSP Setting Contexts

Setting Context	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.

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 include tools for manipulating the settings, are covered in "Tcl Commands for BSP Settings" on page 15–76.

Overview of Component and Driver Settings

The Nios II EDS includes a number of standard software packages and device drivers. All of the software packages, and several drivers, have settings that you can manipulate when creating a BSP. This section lists the packages and drivers that have settings.

You can enable a software package or driver in the Nios II BSP Editor. In the SBT command line flow, use the `enable_sw_package` Tcl command, described in "Tcl Commands for BSP Settings" on page 15–76.

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.

The host-based file system enables debugging information in your project, by setting `APP_CFLAGS_OPTIMIZATION` to `-g` in the makefile.

To include the host-based file system in your BSP, enable the `altera_hostfs` software package.

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**.

To include the read-only zip file system in your BSP, enable the `altera_ro_zipfs` software package.

Altera NicheStack TCP/IP - Nios II Edition Stack Settings

The Altera NicheStack® TCP/IP Network Stack - Nios II Edition 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**.

To include the NicheStack TCP/IP networking stack in your BSP, enable the `altera_iniche` software package.

Altera Avalon-MM JTAG UART Driver Settings

The Altera Avalon Memory-Mapped® (Avalon-MM) JTAG UART driver settings are available if the `altera_avalon_jtag_uart` driver is present. By default, this driver is used if your hardware 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 hardware 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.enable_instruction_related_exceptions_api

Identifier:	none
Type:	Boolean definition
Default Value:	false
Destination File:	none
Description:	Enables application program interface (API) for registering handlers to service instruction-related exceptions. These exception types can be generated if various processor options are enabled, such as the memory management unit (MMU), memory protection unit (MPU), or other advanced exception types. Enabling this setting increases the size of the exception entry code.
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**.

hal.exclude_default_exception

Identifier:	ALT_EXCLUDE_DEFAULT_EXCEPTION
Type:	Boolean definition
Default Value:	false
Destination File:	**system.h**
Description:	Excludes default exception vector. If true, this setting defines the macro ALT_EXCLUDE_DEFAULT_EXCEPTION in system.h.
Restrictions:	none

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

ucosii.os_max_tasks

Identifier:	OS_MAX_TASKS
Type:	Decimal number
Default Value:	10
Destination File:	**system.h**
Description:	Maximum number of tasks
Restrictions:	none

ucosii.os_lowest_prio

Identifier:	OS_LOWEST_PRIO
Type:	Decimal number
Default Value:	20
Destination File:	**system.h**
Description:	Lowest assignable priority
Restrictions:	none

ucosii.os_thread_safe_newlib

Identifier:	OS_THREAD_SAFE_NEWLIB
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Thread safe C library
Restrictions:	none

ucosii.miscellaneous.os_arg_chk_en

Identifier:	OS_ARG_CHK_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Enable argument checking
Restrictions:	none

ucosii.miscellaneous.os_cpu_hooks_en

Identifier:	OS_CPU_HOOKS_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Enable MicroC/OS-II hooks
Restrictions:	none

ucosii.miscellaneous.os_debug_en

Identifier:	OS_DEBUG_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Enable debug variables
Restrictions:	none

ucosii.miscellaneous.os_sched_lock_en

Identifier:	OS_SCHED_LOCK_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code for OSSchedLock() and OSSchedUnlock()
Restrictions:	none

ucosii.miscellaneous.os_task_stat_en

Identifier:	OS_TASK_STAT_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Enable statistics task
Restrictions:	none

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**
Description:	Check task stacks from statistics task
Restrictions:	none

ucosii.miscellaneous.os_tick_step_en

Identifier:	OS_TICK_STEP_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Enable tick stepping feature for uCOS-View
Restrictions:	none

ucosii.miscellaneous.os_event_name_size

Identifier:	OS_EVENT_NAME_SIZE
Type:	Decimal number
Default Value:	32
Destination File:	**system.h**
Description:	Size of name of Event Control Block groups
Restrictions:	none

ucosii.miscellaneous.os_max_events

Identifier:	OS_MAX_EVENTS
Type:	Decimal number
Default Value:	60
Destination File:	**system.h**
Description:	Maximum number of event control blocks
Restrictions:	none

ucosii.miscellaneous.os_task_idle_stk_size

Identifier:	OS_TASK_IDLE_STK_SIZE
Type:	Decimal number
Default Value:	512
Destination File:	**system.h**
Description:	Idle task stack size
Restrictions:	none

ucosii.miscellaneous.os_task_stat_stk_size

Identifier:	OS_TASK_STAT_STK_SIZE
Type:	Decimal number
Default Value:	512
Destination File:	**system.h**
Description:	Statistics task stack size
Restrictions:	none

ucosii.task.os_task_change_prio_en

Identifier:	OS_TASK_CHANGE_PRIO_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code for OSTaskChangePrio()
Restrictions:	none

ucosii.task.os_task_create_en

Identifier:	OS_TASK_CREATE_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code for OSTaskCreate()
Restrictions:	none

ucosii.task.os_task_create_ext_en

Identifier:	OS_TASK_CREATE_EXT_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code for OSTaskCreateExt()
Restrictions:	none

ucosii.task.os_task_del_en

Identifier:	OS_TASK_DEL_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code for OSTaskDel()
Restrictions:	none

ucosii.task.os_task_name_size

Identifier:	OS_TASK_NAME_SIZE
Type:	Decimal number
Default Value:	32
Destination File:	**system.h**
Description:	Size of task name
Restrictions:	none

ucosii.task.os_task_profile_en

Identifier:	OS_TASK_PROFILE_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include data structure for run-time task profiling
Restrictions:	none

ucosii.task.os_task_query_en

Identifier:	OS_TASK_QUERY_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code for OSTaskQuery
Restrictions:	none

ucosii.task.os_task_suspend_en

Identifier:	OS_TASK_SUSPEND_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code for OSTaskSuspend() and OSTaskResume()
Restrictions:	none

ucosii.task.os_task_sw_hook_en

Identifier:	OS_TASK_SW_HOOK_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code for OSTaskSwHook()
Restrictions:	none

ucosii.time.os_time_tick_hook_en

Identifier:	OS_TIME_TICK_HOOK_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code for OSTimeTickHook()
Restrictions:	none

ucosii.time.os_time_dly_resume_en

Identifier:	OS_TIME_DLY_RESUME_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code for OSTimeDlyResume()
Restrictions:	none

ucosii.time.os_time_dly_hmsm_en

Identifier:	OS_TIME_DLY_HMSM_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code for OSTimeDlyHMSM()
Restrictions:	none

ucosii.time.os_time_get_set_en

Identifier:	OS_TIME_GET_SET_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code for OSTimeGet and OSTimeSet()
Restrictions:	none

ucosii.os_flag_en

Identifier:	OS_FLAG_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Enable code for Event Flags. This setting is enabled by default in MicroC-OS/II BSPs, because it is required for correct functioning of Altera device drivers and the HAL in a multithreaded environment. Avoid disabling it.
Restrictions:	none

ucosii.event_flag.os_flag_wait_clr_en

Identifier:	OS_FLAG_WAIT_CLR_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code for Wait on Clear Event Flags. This setting is enabled by default in MicroC-OS/II BSPs, because it is required for correct functioning of Altera device drivers and the HAL in a multithreaded environment. Avoid disabling it.
Restrictions:	none

ucosii.event_flag.os_flag_accept_en

Identifier:	OS_FLAG_ACCEPT_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code for OSFlagAccept(). This setting is enabled by default in MicroC-OS/II BSPs, because it is required for correct functioning of Altera device drivers and the HAL in a multithreaded environment. Avoid disabling it.
Restrictions:	none

ucosii.event_flag.os_flag_del_en

Identifier:	OS_FLAG_DEL_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code for OSFlagDel(). This setting is enabled by default in MicroC-OS/II BSPs, because it is required for correct functioning of Altera device drivers and the HAL in a multithreaded environment. Avoid disabling it.
Restrictions:	none

ucosii.event_flag.os_flag_query_en

Identifier:	OS_FLAG_QUERY_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code for OSFlagQuery(). This setting is enabled by default in MicroC-OS/II BSPs, because it is required for correct functioning of Altera device drivers and the HAL in a multithreaded environment. Avoid disabling it.
Restrictions:	none

ucosii.event_flag.os_flag_name_size

Identifier:	OS_FLAG_NAME_SIZE
Type:	Decimal number
Default Value:	32
Destination File:	**system.h**
Description:	Size of name of Event Flags group. CAUTION: This is required by the HAL and many Altera device drivers; use caution in reducing this value.
Restrictions:	none

ucosii.event_flag.os_flags_nbits

Identifier:	OS_FLAGS_NBITS
Type:	Decimal number
Default Value:	16
Destination File:	**system.h**
Description:	Event Flag bits (8,16,32). CAUTION: This is required by the HAL and many Altera device drivers; use caution in changing this value.
Restrictions:	none

ucosii.event_flag.os_max_flags

Identifier:	OS_MAX_FLAGS
Type:	Decimal number
Default Value:	20
Destination File:	**system.h**
Description:	Maximum number of Event Flags groups. CAUTION: This is required by the HAL and many Altera device drivers; use caution in reducing this value.
Restrictions:	none

ucosii.os_mutex_en

Identifier:	OS_MUTEX_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Enable code for Mutex Semaphores
Restrictions:	none

ucosii.mutex.os_mutex_accept_en

Identifier:	OS_MUTEX_ACCEPT_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code for OSMutexAccept()
Restrictions:	none

ucosii.mutex.os_mutex_del_en

Identifier:	OS_MUTEX_DEL_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code for OSMutexDel()
Restrictions:	none

ucosii.mutex.os_mutex_query_en

Identifier:	OS_MUTEX_QUERY_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code for OSMutexQuery
Restrictions:	none

ucosii.os_sem_en

Identifier:	OS_SEM_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Enable code for semaphores. This setting is enabled by default in MicroC-OS/II BSPs, because it is required for correct functioning of Altera device drivers and the HAL in a multithreaded environment. Avoid disabling it.
Restrictions:	none

ucosii.semaphore.os_sem_accept_en

Identifier:	OS_SEM_ACCEPT_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code for OSSemAccept(). This setting is enabled by default in MicroC-OS/II BSPs, because it is required for correct functioning of Altera device drivers and the HAL in a multithreaded environment. Avoid disabling it.
Restrictions:	none

ucosii.semaphore.os_sem_set_en

Identifier:	OS_SEM_SET_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code for OSSemSet(). This setting is enabled by default in MicroC-OS/II BSPs, because it is required for correct functioning of Altera device drivers and the HAL in a multithreaded environment. Avoid disabling it.
Restrictions:	none

ucosii.semaphore.os_sem_del_en

Identifier:	OS_SEM_DEL_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code for OSSemDel(). This setting is enabled by default in MicroC-OS/II BSPs, because it is required for correct functioning of Altera device drivers and the HAL in a multithreaded environment. Avoid disabling it.
Restrictions:	none

ucosii.semaphore.os_sem_query_en

Identifier:	OS_SEM_QUERY_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code for OSSemQuery(). This setting is enabled by default in MicroC-OS/II BSPs, because it is required for correct functioning of Altera device drivers and the HAL in a multithreaded environment. Avoid disabling it.
Restrictions:	none

ucosii.os_mbox_en

Identifier:	OS_MBOX_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Enable code for mailboxes
Restrictions:	none

ucosii.mailbox.os_mbox_accept_en

Identifier:	OS_MBOX_ACCEPT_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code for OSMboxAccept()
Restrictions:	none

ucosii.mailbox.os_mbox_del_en

Identifier:	OS_MBOX_DEL_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code for OSMboxDel()
Restrictions:	none

ucosii.mailbox.os_mbox_post_en

Identifier:	OS_MBOX_POST_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code for OSMboxPost()
Restrictions:	none

ucosii.mailbox.os_mbox_post_opt_en

Identifier:	OS_MBOX_POST_OPT_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code for OSMboxPostOpt()
Restrictions:	none

ucosii.mailbox.os_mbox_query_en

Identifier:	OS_MBOX_QUERY_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code for OSMboxQuery()
Restrictions:	none

ucosii.os_q_en

Identifier:	OS_Q_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Enable code for Queues
Restrictions:	none

ucosii.queue.os_q_accept_en

Identifier:	OS_Q_ACCEPT_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code for OSQAccept()
Restrictions:	none

ucosii.queue.os_q_del_en

Identifier:	OS_Q_DEL_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code for OSQDel()
Restrictions:	none

ucosii.queue.os_q_flush_en

Identifier:	OS_Q_FLUSH_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code for OSQFlush()
Restrictions:	none

ucosii.queue.os_q_post_en

Identifier:	OS_Q_POST_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code of OSQFlush()
Restrictions:	none

ucosii.queue.os_q_post_front_en

Identifier:	OS_Q_POST_FRONT_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code for OSQPostFront()
Restrictions:	none

ucosii.queue.os_q_post_opt_en

Identifier:	OS_Q_POST_OPT_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code for OSQPostOpt()
Restrictions:	none

ucosii.queue.os_q_query_en

Identifier:	OS_Q_QUERY_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code for OSQQuery()
Restrictions:	none

ucosii.queue.os_max_qs

Identifier:	OS_MAX_QS
Type:	Decimal number
Default Value:	20
Destination File:	**system.h**
Description:	Maximum number of Queue Control Blocks
Restrictions:	none

ucosii.os_mem_en

Identifier:	OS_MEM_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Enable code for memory management
Restrictions:	none

ucosii.memory.os_mem_query_en

Identifier:	OS_MEM_QUERY_EN
Type:	Boolean assignment
Default Value:	1
Destination File:	**system.h**
Description:	Include code for OSMemQuery()
Restrictions:	none

ucosii.memory.os_mem_name_size

Identifier:	OS_MEM_NAME_SIZE
Type:	Decimal number
Default Value:	32
Destination File:	**system.h**
Description:	Size of memory partition name
Restrictions:	none

ucosii.memory.os_max_mem_part

Identifier:	OS_MAX_MEM_PART
Type:	Decimal number
Default Value:	60
Destination File:	**system.h**
Description:	Maximum number of memory partitions
Restrictions:	none

ucosii.os_tmr_en

Identifier:	OS_TMR_EN
Type:	Boolean assignment
Default Value:	0
Destination File:	**system.h**
Description:	Enable code for timers
Restrictions:	none

ucosii.timer.os_task_tmr_stk_size

Identifier:	OS_TASK_TMR_STK_SIZE
Type:	Decimal number
Default Value:	512
Destination File:	**system.h**
Description:	Stack size for timer task
Restrictions:	none

ucosii.timer.os_task_tmr_prio

Identifier:	OS_TASK_TMR_PRIO
Type:	Decimal number
Default Value:	2
Destination File:	**system.h**
Description:	Priority of timer task (0=highest)
Restrictions:	none

ucosii.timer.os_tmr_cfg_max

Identifier:	OS_TMR_CFG_MAX
Type:	Decimal number
Default Value:	16
Destination File:	**system.h**
Description:	Maximum number of timers
Restrictions:	none

ucosii.timer.os_tmr_cfg_name_size

Identifier:	OS_TMR_CFG_NAME_SIZE
Type:	Decimal number
Default Value:	16
Destination File:	**system.h**
Description:	Size of timer name
Restrictions:	none

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**
Description:	Rate at which timer management task runs (Hz)
Restrictions:	none

ucosii.timer.os_tmr_cfg_wheel_size

Identifier:	OS_TMR_CFG_WHEEL_SIZE
Type:	Decimal number
Default Value:	2
Destination File:	**system.h**
Description:	Size of timer wheel (number of spokes)
Restrictions:	none

altera_avalon_uart_driver.enable_small_driver

Identifier:	ALTERA_AVALON_UART_SMALL
Type:	Boolean definition
Default Value:	false
Destination File:	**public.mk**
Description:	Small-footprint (polled mode) driver
Restrictions:	none

altera_avalon_uart_driver.enable_ioctl

Identifier:	ALTERA_AVALON_UART_USE_IOCTL
Type:	Boolean definition
Default Value:	false
Destination File:	**public.mk**
Description:	Enable driver ioctl() support. This feature is not compatible with the small driver; ioctl() support is not compiled if either the UART `enable_small_driver` or the HAL `enable_reduced_device_drivers` setting is enabled.
Restrictions:	none

altera_avalon_jtag_uart_driver.enable_small_driver

Identifier:	ALTERA_AVALON_JTAG_UART_SMALL
Type:	Boolean definition
Default Value:	false
Destination File:	**public.mk**
Description:	Small-footprint (polled mode) driver
Restrictions:	none

altera_hostfs.hostfs_name

Identifier:	ALTERA_HOSTFS_NAME
Type:	Quoted string
Default Value:	/mnt/host
Destination File:	**system.h**
Description:	Mount point
Restrictions:	none

altera_iniche.iniche_default_if

Identifier:	INICHE_DEFAULT_IF
Type:	Quoted string
Default Value:	NOT_USED
Destination File:	**system.h**
Description:	Deprecated setting: Default media access control (MAC) interface. This setting is used in some legacy Altera networking examples. It is not needed in new projects. If this setting appears in an existing project, Altera recommends that you make any necessary changes to remove it. This setting might be removed in a future release.
Restrictions:	none

altera_iniche.enable_dhcp_client

Identifier:	DHCP_CLIENT
Type:	Boolean definition
Default Value:	true
Destination File:	**system.h**
Description:	Use dynamic host configuration protocol (DHCP) to automatically assign Internet protocol (IP) address
Restrictions:	none

altera_iniche.enable_ip_fragments

Identifier:	IP_FRAGMENTS
Type:	Boolean definition
Default Value:	true
Destination File:	**system.h**
Description:	Reassemble IP packet fragments
Restrictions:	none

altera_iniche.enable_include_tcp

Identifier:	INCLUDE_TCP
Type:	Boolean definition
Default Value:	true
Destination File:	**system.h**
Description:	Enable Transmission Control Protocol (TCP)
Restrictions:	none

altera_iniche.enable_tcp_zerocopy

Identifier:	TCP_ZEROCOPY
Type:	Boolean definition
Default Value:	false
Destination File:	**system.h**
Description:	Use TCP zero-copy
Restrictions:	none

altera_iniche.enable_net_stats

Identifier:	NET_STATS
Type:	Boolean definition
Default Value:	false
Destination File:	**system.h**
Description:	Enable statistics
Restrictions:	none

altera_ro_zipfs.ro_zipfs_name

Identifier:	ALTERA_RO_ZIPFS_NAME
Type:	Quoted string
Default Value:	/mnt/rozipfs
Destination File:	**system.h**
Description:	Mount point
Restrictions:	none

altera_ro_zipfs.ro_zipfs_offset

Identifier:	ALTERA_RO_ZIPFS_OFFSET
Type:	Hexadecimal number
Default Value:	0x100000
Destination File:	**system.h**
Description:	Offset of file system from base of flash
Restrictions:	none

altera_ro_zipfs.ro_zipfs_base

Identifier:	ALTERA_RO_ZIPFS_BASE
Type:	Hexadecimal number
Default Value:	0x0
Destination File:	**system.h**
Description:	Base address of flash memory device
Restrictions:	none

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: 1

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 (virtual memory address/low memory address) 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_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_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_rwdata

Identifier: none

Type: Boolean assignment

Default Value: 0

Destination File: none

Description: Causes the initialization code to copy the `.rwdata` section. If true, this setting defines the macro ALT_LOAD_COPY_RWDATA 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. The exception stack can be used to improve interrupt and other exception performance if an EIC is not implemented.

hal.linker.exception_stack_memory_region_name

Identifier: none

Type: Unquoted string

Default Value: none

Destination File: none

Description: Name of the existing memory region to be divided up to create the `exception_stack` memory region. The selected region name is adjusted automatically when the BSP is generated to create the `exception_stack` memory region.

Restrictions: Only used if hal.linker.enable_exception_stack is true.

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.linker.enable_interrupt_stack

Identifier:	none
Type:	Boolean assignment
Default Value:	0
Destination File:	none
Description:	Enables use of a separate interrupt stack. If true, defines the macro `ALT_INTERRUPT_STACK` in **linker.h**, adds a memory region called interrupt_stack to **linker.x**, and provides the symbols `__alt_interrupt_stack_pointer` and `__alt_interrupt_stack_limit` in **linker.x**.
Restrictions:	The `hal.linker.interrupt_stack_size` and `hal.linker.interrupt_stack_memory_region_name` settings must also be valid. This setting must be false for MicroC/OS-II BSPs. Only enable this setting for systems with an EIC. If an EIC is not implemented, use the separate exception stack to improve interrupt and other exception performance.

hal.linker.interrupt_stack_memory_region_name

Identifier:	none
Type:	Unquoted String
Default Value:	none
Destination File:	none
Description:	Name of the existing memory region that is divided up to create the `interrupt_stack` memory region. The selected region name is adjusted automatically when the BSP is generated to create the `interrupt_stack` memory region.
Restrictions:	Only used if `hal.linker.enable_interrupt_stack` is true.

hal.linker.interrupt_stack_size

Identifier:	none
Type:	Decimal Number
Default Value:	1024
Destination File:	none
Description:	Size of the interrupt stack in bytes.
Restrictions:	Only used if `hal.linker.enable_interrupt_stack` is true.

hal.make.ar

Identifier:	AR
Type:	Unquoted string
Default Value:	nios2-elf-ar
Destination File:	BSP makefile
Description:	Archiver command. Creates library files.
Restrictions:	none

hal.make.ar_post_process

Identifier:	AR_POST_PROCESS
Type:	Unquoted string
Default Value:	none
Destination File:	BSP makefile
Description:	Command executed after archiver execution.
Restrictions:	none

hal.make.ar_pre_process

Identifier:	AR_PRE_PROCESS
Type:	Unquoted string
Default Value:	none
Destination File:	BSP makefile
Description:	Command executed before archiver execution.
Restrictions:	none

hal.make.as

Identifier:	AS
Type:	Unquoted string
Default Value:	nios2-elf-gcc
Destination File:	BSP makefile
Description:	Assembler command. Note that CC is used for Nios II assembly language source files (**.S**).
Restrictions:	none

hal.make.as_post_process

Identifier:	AS_POST_PROCESS
Type:	Unquoted string
Default Value:	none
Destination File:	BSP makefile
Description:	Command executed after each assembly file is compiled.
Restrictions:	none

hal.make.as_pre_process

Identifier:	AS_PRE_PROCESS
Type:	Unquoted string
Default Value:	none
Destination File:	BSP makefile
Description:	Command executed before each assembly file is compiled.
Restrictions:	none

hal.make.bsp_arflags

Identifier:	BSP_ARFLAGS
Type:	Unquoted string
Default Value:	-src
Destination File:	BSP makefile
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_asflags

Identifier:	BSP_ASFLAGS
Type:	Unquoted string
Default Value:	-Wa,-gdwarf2
Destination File:	BSP makefile
Description:	Custom flags only passed to the assembler. This setting defines the value of BSP_ASFLAGS in Makefile.
Restrictions:	none

hal.make.bsp_cflags_debug

Identifier:	BSP_CFLAGS_DEBUG
Type:	Unquoted string
Default Value:	-g
Destination File:	BSP makefile
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.bsp_cflags_defined_symbols

Identifier:	BSP_CFLAGS_DEFINED_SYMBOLS
Type:	Unquoted string
Default Value:	none
Destination File:	BSP makefile
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 source 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 source (**.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.bsp_cflags_optimization

Identifier: BSP_CFLAGS_OPTIMIZATION

Type: Unquoted string

Default Value: -O0

Destination File: BSP makefile

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 nonsequential execution of code while debugging. This setting defines the value of BSP_CFLAGS_OPTIMIZATION in Makefile.

Restrictions: none

hal.make.bsp_cflags_undefined_symbols

Identifier: BSP_CFLAGS_UNDEFINED_SYMBOLS

Type: Unquoted string

Default Value: none

Destination File: BSP makefile

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.bsp_cflags_user_flags

Identifier: BSP_CFLAGS_USER_FLAGS

Type: Unquoted string

Default Value: none

Destination File: BSP makefile

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.make.bsp_cflags_warnings

Identifier: BSP_CFLAGS_WARNINGS

Type: Unquoted string

Default Value: -Wall

Destination File: BSP makefile

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_cxx_flags

Identifier:	BSP_CXXFLAGS
Type:	Unquoted string
Default Value:	none
Destination File:	BSP makefile
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:	BSP makefile
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 is added by the makefile, therefore you must not include it in the setting value. This setting defines the value of `BSP_INC_DIRS` in the makefile.
Restrictions:	none

hal.make.build_post_process

Identifier:	BUILD_POST_PROCESS
Type:	Unquoted string
Default Value:	none
Destination File:	BSP makefile
Description:	Command executed after BSP built.
Restrictions:	none

hal.make.build_pre_process

Identifier:	BUILD_PRE_PROCESS
Type:	Unquoted string
Default Value:	none
Destination File:	BSP makefile
Description:	Command executed before BSP built.
Restrictions:	none

hal.make.cc

Identifier:	CC
Type:	Unquoted string
Default Value:	nios2-elf-gcc -xc
Destination File:	BSP makefile
Description:	C compiler command
Restrictions:	none

hal.make.cc_post_process

Identifier:	CC_POST_PROCESS
Type:	Unquoted string
Default Value:	none
Destination File:	BSP makefile
Description:	Command executed after each **.c** or **.S** file is compiled.
Restrictions:	none

hal.make.cc_pre_process

Identifier:	CC_PRE_PROCESS
Type:	Unquoted string
Default Value:	none
Destination File:	BSP makefile
Description:	Command executed before each **.c** or **.S** file is compiled.
Restrictions:	none

hal.make.cxx

Identifier:	CXX
Type:	Unquoted string
Default Value:	nios2-elf-gcc -xc++
Destination File:	BSP makefile
Description:	C++ compiler command
Restrictions:	none

hal.make.cxx_post_process

Identifier:	CXX_POST_PROCESS
Type:	Unquoted string
Default Value:	none
Destination File:	BSP makefile
Description:	Command executed before each C++ file is compiled.
Restrictions:	none

hal.make.cxx_pre_process

Identifier:	CXX_PRE_PROCESS
Type:	Unquoted string
Default Value:	none
Destination File:	BSP makefile
Description:	Command executed before each C++ file is compiled.
Restrictions:	none

hal.make.ignore_system_derived.big_endian

Identifier:	none
Type:	Boolean assignment
Default Value:	0
Destination File:	**public.mk**
Description:	Enable BSP generation to query if SOPC system is big endian. If true ignores export of 'ALT_CFLAGS += -meb' to public.mk if big endian system. If true ignores export of 'ALT_CFLAGS += -mel' if little endian system.
Restrictions:	none

hal.make.ignore_system_derived.fpu_present

Identifier:	none
Type:	Boolean assignment
Default Value:	0
Destination File:	**public.mk**
Description:	Enable BSP generation to query if SOPC system has FPU present. If true ignores export of 'ALT_CFLAGS += -mhard-float' to public.mk if FPU is found in the system. If true ignores export of 'ALT_CFLAGS += -mhard-soft' if FPU is not found in the system.
Restrictions:	none

hal.make.ignore_system_derived.hardware_divide_present

Identifier:	none
Type:	Boolean assignment
Default Value:	0
Destination File:	**public.mk**
Description:	Enable BSP generation to query if SOPC system has hardware divide present. If true ignores export of 'ALT_CFLAGS += -mno-hw-div' to public.mk if no division is found in system. If true ignores export of 'ALT_CFLAGS += -mhw-div' if division is found in the system.
Restrictions:	none

hal.make.ignore_system_derived.hardware_fp_cust_inst_divider_present

Identifier: none

Type: Boolean assignment

Default Value: 0

Destination File: **public.mk**

Description: Enable BSP generation to query if SOPC system floating point custom instruction with a divider is present. If true ignores export of 'ALT_CFLAGS += -mcustom-fpu-cfg=60-2' and 'ALT_LDFLAGS += -mcustom-fpu-cfg=60-2' to public.mk if the custom instruction is found in the system.

Restrictions: none

hal.make.ignore_system_derived.hardware_fp_cust_inst_no_divider_present

Identifier: none

Type: Boolean assignment

Default Value: 0

Destination File: **public.mk**

Description: Enable BSP generation to query if SOPC system floating point custom instruction without a divider is present. If true ignores export of 'ALT_CFLAGS += -mcustom-fpu-cfg=60-1' and 'ALT_LDFLAGS += -mcustom-fpu-cfg=60-1' to public.mk if the custom instruction is found in the system.

Restrictions: none

hal.make.ignore_system_derived.sopc_simulation_enabled

Identifier: none

Type: Boolean assignment

Default Value: 0

Destination File: **public.mk**

Description: Enable BSP generation to query if SOPC system has simulation enabled. If true ignores export of 'ELF_PATCH_FLAG += --simulation_enabled' to public.mk.

Restrictions: none

hal.make.ignore_system_derived.debug_core_present

Identifier: none

Type: Boolean assignment

Default Value: 0

Destination File: **public.mk**

Description: Enable BSP generation to query if SOPC system has a debug core present. If true ignores export of 'CPU_HAS_DEBUG_CORE = 1' to public.mk if a debug core is found in the system. If true ignores export of 'CPU_HAS_DEBUG_CORE = 0' if no debug core is found in the system.

Restrictions: none

hal.make.ignore_system_derived.hardware_multiplier_present

Identifier:	none
Type:	Boolean assignment
Default Value:	0
Destination File:	**public.mk**
Description:	Enable BSP generation to query if SOPC system has multiplier present. If true ignores export of 'ALT_CFLAGS += -mno-hw-mul' to public.mk if no multiplier is found in the system. If true ignores export of 'ALT_CFLAGS += -mhw-mul' if multiplier is found in the system.
Restrictions:	none

hal.make.ignore_system_derived.hardware_mulx_present

Identifier:	none
Type:	Boolean assignment
Default Value:	0
Destination File:	**public.mk**
Description:	Enable BSP generation to query if SOPC system has hardware mulx present. If true ignores export of 'ALT_CFLAGS += -mno-hw-mulx' to public.mk if no mulx is found in the system. If true ignores export of 'ALT_CFLAGS += -mhw-mulx' if mulx is found in the system.
Restrictions:	none

hal.make.ignore_system_derived.sopc_system_base_address

Identifier:	none
Type:	Boolean assignment
Default Value:	0
Destination File:	**public.mk**
Description:	Enable BSP generation to query SOPC system for system ID base address. If true ignores export of 'SOPC_SYSID_FLAG += --sidp=<address>' and 'ELF_PATCH_FLAG += --sidp=<address>' to public.mk.
Restrictions:	none

hal.make.ignore_system_derived.sopc_system_id

Identifier:	none
Type:	Boolean assignment
Default Value:	0
Destination File:	**public.mk**
Description:	Enable BSP generation to query SOPC system for system ID. If true ignores export of 'SOPC_SYSID_FLAG += --id=<sysid>' and 'ELF_PATCH_FLAG += --id=<sysid>' to public.mk.
Restrictions:	none

hal.make.ignore_system_derived.sopc_system_timestamp

Identifier:	none
Type:	Boolean assignment
Default Value:	0
Destination File:	**public.mk**
Description:	Enable BSP generation to query SOPC system for system timestamp. If true ignores export of 'SOPC_SYSID_FLAG += --timestamp=<timestamp>' and 'ELF_PATCH_FLAG += --timestamp=<timestamp>' to public.mk.
Restrictions:	none

hal.make.rm

Identifier:	RM
Type:	Unquoted string
Default Value:	rm -f
Destination File:	BSP makefile
Description:	Command used to remove files when building the `clean` target.
Restrictions:	none

hal.custom_newlib_flags

Identifier:	CUSTOM_NEWLIB_FLAGS
Type:	Unquoted string
Default Value:	none
Destination File:	**public.mk**
Description:	Build a custom version of newlib with the specified space-separated compiler flags.
Restrictions:	The custom newlib build is placed in the *<bsp root>***/newlib** directory, and is used only for applications that utilize this BSP.

hal.enable_c_plus_plus

Identifier:	ALT_NO_C_PLUS_PLUS
Type:	Boolean assignment
Default Value:	1
Destination File:	**public.mk**
Description:	Enable support for a subset of the C++ language. This option increases code footprint by adding support for C++ constructors. Certain features, such as multiple inheritance and exceptions are not supported. If false, adds `-DALT_NO_C_PLUS_PLUS` to `ALT_CPPFLAGS` in **public.mk**, and reduces code footprint.
Restrictions:	none

hal.enable_clean_exit

Identifier:	ALT_NO_CLEAN_EXIT
Type:	Boolean assignment
Default Value:	1
Destination File:	**public.mk**
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_exit

Identifier:	ALT_NO_EXIT
Type:	Boolean assignment
Default Value:	1
Destination File:	**public.mk**
Description:	Add exit() support. This option increases code footprint if your `main()` routine returns or calls `exit()`. If false, adds `-DALT_NO_EXIT` to `ALT_CPPFLAGS` in **public.mk**, and reduces footprint.
Restrictions:	none

hal.enable_gprof

Identifier:	ALT_PROVIDE_GMON
Type:	Boolean assignment
Default Value:	0
Destination File:	**public.mk**
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_lightweight_device_driver_api

Identifier:	ALT_USE_DIRECT_DRIVERS
Type:	Boolean assignment
Default Value:	0
Destination File:	**public.mk**
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 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 cannot 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**

Description: Adds code to the BSP to emulate multiply and divide instructions. This code is independent of any emulation code added by the C/C++ compiler.

If false, adds `-DALT_NO_INSTRUCTION_EMULATION` to `ALT_CPPFLAGS` in **public.mk**.

You do not normally need to enable this option, because the C/C++ compiler detects whether the target Nios II processor core supports the multiply and divide instructions directly. If you compile for a core that lacks support for the instructions, the HAL includes the required software emulation in its run-time libraries.

However, you might need to enable `hal.enable_mul_div_emulation` under the following circumstances:

- You expect to run the Nios II software on an implementation of the Nios II processor other than the one you compiled for. The best solution is to build your program for the correct Nios II processor implementation. Resort to the `hal.enable_mul_div_emulation` if it is not possible to determine the processor implementation at compile time.
- You have assembly language code that uses an implementation-dependent instruction.

Restrictions: none

hal.enable_reduced_device_drivers

Identifier: ALT_USE_SMALL_DRIVERS

Type: Boolean assignment

Default Value: 0

Destination File: **public.mk**

Description: Certain drivers are compiled with reduced functionality to reduce code footprint. Not all drivers observe this setting.

If true, adds `-DALT_USE_SMALL_DRIVERS` to `ALT_CPPFLAGS` in **public.mk**.

Typically, drivers support this setting with a polled mode. For example, the altera_avalon_uart and altera_avalon_jtag_uart reduced drivers operate in polled mode.

Several device drivers are disabled entirely in reduced drivers mode. These include the altera_avalon_cfi_flash, altera_avalon_epcs_flash_controller, and altera_avalon_lcd_16207 drivers. As a result, certain API routines fail (HAL flash access routines). You can define a symbol provided by each driver to prevent it from being removed.

Restrictions: none

hal.enable_runtime_stack_checking

Identifier: ALT_STACK_CHECK

Type: Boolean assignment

Default Value: 0

Destination File: **public.mk**

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 `-fstack-check` 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**

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 cannot run on hardware.

hal.enable_small_c_library

Identifier: none

Type: Boolean assignment

Default Value: 0

Destination File: **public.mk**

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. The small C library is not compatible with Micrium MicroC/OS-II. If true, adds `-msmallc` to `ALT_LDFLAGS` and adds `-DSMALL_C_LIB` to `ALT_CPPFLAGS` in **public.mk**.

Restrictions: none

hal.enable_sopc_sysid_check

Identifier: none

Type: Boolean assignment

Default Value: 1

Destination File: **public.mk**

Description: Enables system ID check. If a System ID component is connected to the processor associated with this BSP, the system ID check is 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 the application makefile `download-elf` target definition. With the system ID check disabled, the Nios II EDS tools do not automatically ensure that the application **.elf** file (and BSP it is linked against) corresponds to the hardware design on the target. If false, adds `--accept-bad-sysid` to `SOPC_SYSID_FLAG` in **public.mk**.

Altera strongly recommends leaving hal.enable_sopc_sysid_check enabled. This setting is exposed to support rare cases in which FPGA logic resources are in extremely short supply. When the system ID check is disabled, the software is unable to detect whether the software is running on the correct hardware version. This situation can lead to subtle errors that are difficult to diagnose.

Restrictions: none

hal.log_port

Identifier: LOG_PORT

Type: Unquoted string

Default Value: none

Destination File: **system.h**

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 Altera logging functions.

hal.log_flags

Identifier: ALT_LOG_FLAGS

Type: Decimal Number

Default Value: 0

Destination File: **public.mk**

Description: The value is assigned to ALT_LOG_FLAGS in the generated **public.mk**. Refer to **hal.log_port** for further details. The valid range of this setting is 1 through 4.

hal.stderr

Identifier: STDERR

Type: Unquoted string

Default Value: none

Destination File: **public.mk**

Description: Slave descriptor of STDERR character-mode device. This setting is used by the ALT_STDERR family of defines in **system.h**.

hal.stdin

Identifier: STDIN

Type: Unquoted string

Default Value: none

Destination File: **system.h**

Description: Slave descriptor of STDIN character-mode device. This setting is used by the ALT_STDIN family of defines in **system.h**.

hal.stdout

Identifier: STDOUT

Type: Unquoted string

Default Value: none

Destination File: **system.h**

Description: Slave descriptor of STDOUT character-mode device. This setting is used by the ALT_STDOUT family of defines in **system.h**.

Application and User Library Makefile Variables

The Nios II SBT constructs application and makefile libraries for you, inserting makefile variables appropriate to your project configuration. You can control project build characteristics by manipulating makefile variables at the time of project generation. You control a variable with the `--set` command line option, as in the following example:

```
nios2-bsp hal my_bsp --set APP_CFLAGS_WARNINGS "-Wall"↵
```

The following utilities and scripts support modifying makefile variables with the `--set` option:

- **nios2-app-generate-makefile**
- **nios2-lib-generate-makefile**
- **nios2-app-update-makefile**
- **nios2-lib-update-makefile**
- **nios2-bsp**

Application Makefile Variables

You can modify the following application makefile variables on the command line:

- `CREATE_OBJDUMP`—Assign 1 to this variable to enable creation of an object dump file (**.objdump**) after linking the application. The **nios2-elf-objdump** utility is called to create this file. An object dump contains information about all object files linked into the **.elf** file. It provides a complete view of all code linked into your application. An object dump contains a disassembly view showing each instruction and its address.
- `OBJDUMP_INCLUDE_SOURCE`—Assign 1 to this variable to include source code inline with disassembled instructions in the object dump. When enabled, this includes the `--source` switch when calling the object dump executable. This is useful for debugging and examination of how the preprocessor and compiler generate instructions from higher level source code (such as C) or from macros.
- `OBJDUMP_FULL_CONTENTS`—Assign 1 to this variable to include a raw display of the contents of the `.text` linker section. When enabled, this variable includes the `--full-contents` switch when calling the object dump executable.
- `CREATE_ELF_DERIVED_FILES`—Setting this variable to 1 creates the HDL simulation and onchip memory initialization files when you invoke the makefile with the `all` target. When this variable is 0 (the default), these files are only created when you make the `mem_init_generate` or `mem_init_install` target.

  ☞ Creating the HDL simulation and onchip memory initialization files increases project build time.

- `CREATE_LINKER_MAP`—Assign 1 to this variable to enable creation of a link map file (**.map**) after linking the application. A link map file provides information including which object files are included in the executable, the path to each object file, where objects and symbols are located in memory, and how the common symbols are allocated.

- APP_CFLAGS_DEFINED_SYMBOLS—This variable allows you to define macros using the -D argument, for example -D *<macro name>*. The contents of this variable are passed to the compiler and linker without modification.
- APP_CFLAGS_UNDEFINED_SYMBOLS—This variable allows you to remove macro definitions using the -U argument, for example -U *<macro name>*. The contents of this variable are passed to the compiler and linker without modification.
- APP_CFLAGS_OPTIMIZATION—The C/C++ compiler optimization level. For example, -O0 provides no optimization and -O2 provides standard optimization. -O0 is recommended for debugging code, because compiler optimization can remove variables and produce non-sequential execution of code while debugging.
- APP_CFLAGS_DEBUG_LEVEL—The C/C++ compiler debug level. -g provides the default set of debug symbols typically required to debug an application. Omitting -g omits debug symbols from the **.elf**.
- APP_CFLAGS_WARNINGS—The C/C++ compiler warning level. -Wall is commonly used, enabling all warning messages.
- APP_CFLAGS_USER_FLAGS
- APP_INCLUDE_DIRS—Use this variable to specify paths for the preprocessor to search. These paths commonly contain C header files (**.h**) that application code requires. Each path name is formatted and passed to the preprocessor with the -I option.

  You can add multiple directories by enclosing them in double quotes, for example --set APP_INCLUDE_DIRS "../my_includes ../../other_includes".
- APP_LIBRARY_DIRS—Use this variable to specify paths for additional libraries that your application links with.

  ☞ When you specify a user library path with APP_LIBRARY_DIRS, you also need to specify the user library names with the APP_LIBRARY_NAMES variable.

  APP_LIBRARY_DIRS specifies only the directory where the user library file(s) are located, not the library archive file (**.a**) name.

  ☞ Do not use this variable to specify the path to a BSP or user library created with the SBT. The paths to these libraries are specified in **public.mk** files included in the application makefile.

  You can add multiple directories by enclosing them in double quotes, for example --set APP_LIBRARY_DIRS "../my_libs ../../other_libs".
- APP_LIBRARY_NAMES—Use this variable to specify the names of additional libraries that your application must link with. Library files are **.a** files.

You do not specify the full name of the **.a** file. Instead, you specify the user library name <*name*>, and the SBT constructs the filename **lib**<*name*>**.a**. For example, if you add the string `"math"` to `APP_LIBRARY_NAMES`, the SBT assumes that your library file is named **libmath.a**.

Each specified user library name is passed to the linker with the `-l` option. The paths to locate these libraries must be specified in the `APP_LIBRARY_DIRS` variable.

You cannot use this variable to specify a BSP or user library created with the SBT. The paths to these libraries are specified in **public.mk** file included in the application makefile.

- `BUILD_PRE_PROCESS`—This variable allows you to specify a command to be executed prior to building the application, for example, `cp *.elf ../lastbuild`.
- `BUILD_POST_PROCESS`—This variable allows you to specify a command to be executed after building the application, for example, `cp *.elf //production/test/nios2executables`.

User Library Makefile Variables

You can modify the following user library makefile variables on the command line:

- `LIB_CFLAGS_DEFINED_SYMBOLS`—This variable allows you to define macros using the `-D` argument, for example `-D <macro name>`. The contents of this variable are passed to the compiler and linker without modification.
- `LIB_CFLAGS_UNDEFINED_SYMBOLS`—This variable allows you to remove macro definitions using the `-U` argument, for example `-U <macro name>`. The contents of this variable are passed to the compiler and linker without modification.
- `LIB_CFLAGS_OPTIMIZATION`—The C/C++ compiler optimization level. For example, `-O0` provides no optimization and `-O2` provides standard optimization. `-O0` is recommended for debugging code, because compiler optimization can remove variables and produce non-sequential execution of code while debugging.
- `LIB_CFLAGS_DEBUG_LEVEL`—The C/C++ compiler debug level. `-g` provides the default set of debug symbols typically required to debug an application. Omitting `-g` omits debug symbols from the **.elf**.
- `LIB_CFLAGS_WARNINGS`—The C/C++ compiler warning level. `-Wall` is commonly used, enabling all warning messages.
- `LIB_CFLAGS_USER_FLAGS`—
- `LIB_INCLUDE_DIRS`—You can add multiple directories by enclosing them in double quotes, for example `--set LIB_INCLUDE_DIRS "../my_includes ../../other_includes"`

Standard Build Flag Variables

The SBT creates makefiles supporting the following standard makefile command-line variables:

- `CFLAGS`
- `CPPFLAGS`
- `ASFLAGS`
- `CXXFLAGS`

You can define flags in these variables on the makefile command line, or in a script that invokes the makefile. The makefile passes these flags on to the corresponding GCC tool.

Software Build Tools Tcl Commands

Tcl commands are a crucial component of the Nios II SBT. Tcl commands allow you to exercise detailed control over BSP generation, as well as to define drivers and software packages. This section describes the Tcl commands, the environments in which they run, and how the commands work together.

Tcl Command Environments

The Nios II SBT supports Tcl commands in the following environments:

- BSP setting specification—In this environment, you manipulate BSP settings to control the static characteristics of the BSP. BSP setting commands are executed before the BSP is generated.
- BSP generation callbacks—In this environment, you exercise further control over BSP details, managing settings that interact with one another and with the hardware design. BSP callbacks run at BSP generation time.
- Device driver and software package definition—In this environment, you bundle source files into a custom driver or package. This process prepares the driver or package so that a BSP developer can include it in a BSP using the SBT.

The following sections describe each Tcl environment in detail, listing the available commands.

Tcl Commands for BSP Settings

"Settings Managed by the Software Build Tools" on page 15–34 describes settings that are available in a 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 call 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 about how to call Tcl commands from utilities, refer to "Nios II Software Build Tools Utilities" on page 15–1.

For more information about creating Tcl scripts, refer to "Tcl Scripts for BSP Settings" in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*. This chapter includes 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_device" on page 15–79
- "add_memory_region" on page 15–79
- "add_section_mapping" on page 15–80
- "are_same_resource" on page 15–80
- "delete_memory_region" on page 15–81
- "delete_section_mapping" on page 15–81
- "disable_sw_package" on page 15–81
- "enable_sw_package" on page 15–81
- "get_addr_span" on page 15–82
- "get_assignment" on page 15–82
- "get_available_drivers" on page 15–83
- "get_available_sw_packages" on page 15–83
- "get_base_addr" on page 15–83
- "get_break_offset" on page 15–84
- "get_break_slave_desc" on page 15–84
- "get_cpu_name" on page 15–84
- "get_current_memory_regions" on page 15–85
- "get_current_section_mappings" on page 15–85
- "get_default_memory_regions" on page 15–86
- "get_driver" on page 15–86
- "get_enabled_sw_packages" on page 15–87
- "get_exception_offset" on page 15–87
- "get_exception_slave_desc" on page 15–87
- "get_fast_tlb_miss_exception_offset" on page 15–88
- "get_fast_tlb_miss_exception_slave_desc" on page 15–88
- "get_interrupt_controller_id" on page 15–88
- "get_irq_interrupt_controller_id" on page 15–89
- "get_irq_number" on page 15–89
- "get_memory_region" on page 15–89
- "get_module_class_name" on page 15–90
- "get_module_name" on page 15–90

- "get_cpu_architecture" on page 15–102
- "get_nios2_dpx_thread_num" on page 15–102
- "get_sopcinfo_file" on page 15–103
- "get_supported_bsp_types" on page 15–103
- "is_bsp_hal_extension" on page 15–103
- "is_bsp_lwhal_extension" on page 15–103
- "open_bsp" on page 15–103
- "save_bsp" on page 15–104
- "set_bsp_version" on page 15–104
- "set_logging_mode" on page 15–104

add_memory_device

Usage

```
add_memory_device <device name> <base address> <span>
```

Options

- *<device name>*: String with the name of the memory device.
- *<base address>*: The base address of the memory device. Hexadecimal or decimal string.
- *<span>*: The size (span) of the memory device. Hexadecimal or decimal string.

Description

This command is provided to define a user-defined external memory device, outside the hardware system. Such a device would typically be mapped through a bridge component. This command adds an external memory device to the BSP's memory map, allowing the BSP to define memory regions and section mappings for the memory as if it were part of the system. The external memory device parameters are stored in the BSP settings file.

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 must 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 `<software_package_name>:<version>`. If you do not specify the version, `enable_sw_package` selects the latest available version.

Examples

- 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_assignment

Usage

```
get_assignment <module_name> <assignment_name>
```

Options

- *<module_name>*: Module instance name to query for assignment
- *<assignment_name>*: Module instance assignment name to query for

Description

Returns the name of the value of the assignment for a specified module instance name.

Example

```
puts [get_assignment "cpu0" "embeddedsw.configuration.breakSlave"]
```

Returns:

```
memory_0.s0
```

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_break_offset

Usage

```
get_break_offset
```

Options

None

Description

Returns the byte offset of the processor break address.

Example

```
puts [get_break_offset]
```

Returns:

```
32
```

get_break_slave_desc

Usage

```
get_break_slave_desc
```

Options

None

Description

Returns the slave descriptor associated with the processor break address. If null, then the break device is internal to the processor (debug module).

Example

```
puts [get_break_slave_desc]
```

Returns:

```
onchip_ram_64_kbytes
```

get_cpu_name

Usage

```
get_cpu_name
```

Options

None

Description

Returns the name of the BSP specific processor.

Example

```
puts [get_cpu_name]
```

Returns:

```
cpu_0
```

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.

Examples

- 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 processor fast translation lookaside buffer (TLB) miss exception address. Only a processor 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 processor fast TLB miss exception address. Only a processor with an MMU has such an exception address.

Example

```
puts [get_fast_tlb_miss_exception_slave_desc]
```

Returns:

```
onchip_ram_64_kbytes
```

get_interrupt_controller_id

Usage

get_interrupt_controller_id *<slave_desc>*

Options

- *<slave_desc>*: String with the slave descriptor to query.

Description

Returns the interrupt controller ID of the slave descriptor (-1 if not a connected interrupt controller).

get_irq_interrupt_controller_id

Usage

```
get_irq_interrupt_controller_id <slave_desc>
```

Options

- `<slave_desc>`: String with the slave descriptor to query.

Description

Returns the interrupt controller ID connected to the IRQ associated with the slave descriptor (-1 if none).

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, or -1 if no interrupt request number is found.

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_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.

Examples

- Example 1:

  ```
  puts [is_char_device jtag_uart]
  ```

  Returns:

  ```
  1
  ```

- Example 2:

  ```
  puts [is_char_device onchip_ram_64_kbytes]
  ```

  Returns:

  ```
  0
  ```

is_connected_interrupt_controller_device

Usage

```
is_connected_interrupt_controller_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 an interrupt controller device that is connected to the processor so that the interrupt controller sends interrupts to the processor.

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_ethernet_mac_device

Usage

```
is_ethernet_mac_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 an Ethernet MAC device.

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.

Examples

- 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.

is_timer_device

Usage

```
is_timer_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 timer 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 that is compatible with the component 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 component class.

Examples

- Example 1:

  ```
  set_driver altera_avalon_jtag_uart_driver:7.2 jtag_uart
  ```

- Example 2:

  ```
  set_driver none jtag_uart
  ```

set_ignore_file

Usage

`set_ignore_file <software_component_name> <file_name> <ignore>`

Options

- *`<software_component_name>`*: Name of the driver, software package, or operating system to which the file belongs.
- *`<file_name>`*: Name of the file.
- *`<ignore>`*: Set to true to ignore (not generate or copy) the file, false to generate or copy the file normally.

Description

You can use this command to have a specific BSP file ignored (not generated or copied) during BSP generation. This command allows you to take ownership of a specific file, modify it, and prevent the SBT from overwriting your modifications.

<*software_component_name*> can have one of the following values:

- *`<driver_name>`*—The name of a driver, as specified with the `create_driver` command in the ***_sw.tcl** file that defines the driver. Specifies that <*file_name*> is a copied file associated with a device driver.
- *`<software_package_name>`*—The name of a software package, specified with the `create_sw_package` command in the ***_sw.tcl** file that defines the package. Specifies that <*file_name*> is a copied file associated with a software package.
- *`<OS_name>`*—The name of an OS, specified with the `create_os` command in the ***_sw.tcl** file that defines the OS, and is used in the **nios2-bsp-create-settings** to specify the BSP type. Specifies that <*file_name*> is a copied file associated with an OS.

- `generated`—Specifies that <*file_name*> is a generated top-level BSP file. The list of generated BSP files depends on the BSP type. For a list of generated files associated with HAL and MicroC/OS-II BSPs, refer to "Details of BSP Creation" in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*. For a list of generated files associated with a third-party OS, refer to the OS supplier's documentation.

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
```

update_memory_region

Usage

```
update_memory_region <name> <slave_desc> <offset> <span>
```

Options

- *<name>*: String with the name of the memory region to update.
- *<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

Updates an existing 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

```
update_memory_region onchip_ram0 onchip_ram0 0 0x100000
```

update_section_mapping

Usage

```
update_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

Updates the specified linker section. The linker creates the section mappings in the order in which they appear in the linker script.

Example

```
update_section_mapping .text onchip_ram0
```

add_default_memory_regions

Usage

```
add_default_memory_regions
```

Description

Defaults the BSP to use 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.

create_bsp

Usage

```
create_bsp <bspType> <bsp version> <processor name> <sopcinfo>
```

Options

- `<bspType>`: Type of BSP to create.
- `<bsp version>`: Version of BSP software element to utilize.
- `<processor name>`: Name of processor instance for BSP
- `<sopcinfo>`: .sopcinfo generated file that describes the system the BSP is for.

Description

Creates a new BSP.

generate_bsp

Usage

```
generate_bsp <bspDir>
```

Options

- `<bspDir>`: BSP directory to generate files to.

Description

Generates a new BSP.

get_available_bsp_type_versions

Usage

```
get_available_bsp_type_versions <bsp_type> <sopcinfo_path>
```

Options

- `<bsp_type>`: BSP type identifier (e.g. hal, ucosii).
- `<sopcinfo_path>`: SOPC Information File path. Its parent folder might include custom BSP IP software components (*_sw.tcl).

Description

Gets the available BSP type versions.

get_available_bsp_types

Usage

```
get_available_bsp_types <sopcinfo_path>
```

Options

- `<sopcinfo_path>`: SOPC Information File path. Its parent folder might include custom BSP IP software components (*_sw.tcl).

Description

Gets the available BSP type identifiers.

get_available_cpu_architectures

Usage

```
get_available_cpu_architectures
```

Description

Gets the available processor architectures.

get_available_cpu_names

Usage

```
get_available_cpu_names <sopcinfo_path>
```

Options

- `<sopcinfo_path>`: SOPC Information File path that contains processor instances

Description

Gets the processor names given a SOPC system.

get_available_software

Usage

```
get_available_software <bsp_type> <filter> <sopcinfo_path>
```

Options

- `<bsp_type>`: BSP type identifier (e.g. hal, ucosii).
- `<sopcinfo_path>`: SOPC Information File path. Its parent folder might include custom BSP IP software components (*_sw.tcl).
- `<filter>`: A filter can be applied to restrict results. The following filters are available:
  - `all`
  - `drivers`
  - `sw_packages`
  - `os_elements`

  Comma-separated tokens are acceptable.

Description

Gets the available software (drivers, software packages, and bsp components) for a given BSP type.

get_available_software_setting_properties

Usage

```
get_available_software_setting_properties <setting_name> \
  <software_name> <software_version> <sopcinfo_path>
```

Options

- `<software_name>`: Name of a software component (e.g. `"altera_avalon_uart_driver"`, or `"hal"`).
- `<software_version>`: Enter `"default"` for latest version, or a specific version number.
- `<setting_name>`: Name of a selected software component setting to get properties for(e.g. `hal.linker.allow_code_at_reset`).
- `<sopcinfo_path>`: SOPC Information File path. Its parent folder might include custom BSP IP software components (***_sw.tcl**).

Description

Gets the available setting names for a software component.

get_available_software_settings

Usage

```
get_available_software_settings <software_name> <software_version> \
  <sopcinfo_path>
```

Options

- `<software_name>`: Name of a software component (e.g. altera_avalon_uart_driver).
- `<software_version>`: Enter `"default"` for latest version, or a specific version number.
- `<sopcinfo_path>`: SOPC Information File path. Its parent folder can include custom BSP IP software components (***_sw.tcl**).

Description

Gets the available setting names for a software component.

get_bsp_version

Usage

```
get_bsp_version
```

Description

Gets the version of the BSP operating system software element.

get_cpu_architecture

Usage

```
get_cpu_architecture <processor_name> <sopcinfo_path>
```

Options

- `<processor_name>`: processor instance name
- `<sopcinfo_path>`: SOPC Information File path that contains processor_name instance

Description

Gets the processor architecture (e.g. nios2) of a specified processor instance given a SOPC system.

get_nios2_dpx_thread_num

Usage

```
get_nios2_dpx_thread_num
```

Description

If the BSP is mastered by a Nios II DPX processor, then this function returns the number of threads the processor supports. Otherwise it returns `null`.

get_sopcinfo_file

Usage

```
get_sopcinfo_file
```

Description

Returns the path of the BSP specific SOPC Information File.

get_supported_bsp_types

Usage

```
get_supported_bsp_types <processor_name> <sopcinfo_path>
```

Options

- `<processor_name>`: processor instance name
- `<sopcinfo_path>`: SOPC Information File path. Its parent folder can include custom BSP IP software components (***_sw.tcl**).

Description

Gets the BSP types supported for a given processor and SOPC system.

is_bsp_hal_extension

Usage

```
is_bsp_hal_extension
```

Description

Returns a boolean value that indicates whether the BSP instantiated is of a type based on Altera HAL.

is_bsp_lwhal_extension

Usage

```
is_bsp_lwhal_extension
```

Description

Returns a boolean value that indicates whether the BSP instantiated is of a type based on Altera Lightweight HAL.

open_bsp

Usage

```
open_bsp <settingsFile>
```

Options

- `<settingsFile>`: .bsp settings file to open.

Description

Opens an existing BSP.

save_bsp

Usage

```
save_bsp <settingsFile>
```

Options

- `<settingsFile>`: .bsp settings file to save BSP to.

Description

Saves a new BSP.

set_bsp_version

Usage

```
set_bsp_version <version>
```

Options

- `<version>`: Version of BSP type software element to use.

Description

Sets the version of the BSP operating system software element to a specific value. The value `"default'` uses the latest version available. If this call is not used, the BSP is created using the 'default' BSP software element version.

set_logging_mode

Usage

```
set_logging_mode <mode>
```

Options

- `<mode>`: Logging Mode: 'silent', 'default', 'verbose', 'debug'

Description

Sets the verbosity level of the logger. Useful to eliminate tool informative messages

Tcl Commands for BSP Generation Callbacks

If you are defining a device driver or a software package, you can define Tcl callback functions to run whenever a BSP is generated containing your driver or package. Tcl callback functions enable you to create settings dynamically for the driver or package. This capability is essential when the driver or package settings must be customized to the hardware configuration, or to other BSP settings.

Tcl callback scripts are defined and controlled from the ***_sw.tcl** file associated with the driver or package. In ***_sw.tcl**, you can specify where the Tcl functions come from, when function runs, and the scope of each Tcl function's operation.

When the BSP is generated with your driver or software package, the settings you define in the callback scripts are inserted in **settings.bsp**.

You specify the source of the callback functions with the `set_sw_property` command, using the `callback_source_file` property.

A Tcl callback function can run at one of the following times:

- BSP initialization
- BSP generation
- BSP validation

Although you can specify a new setting's value when you create the setting at BSP initialization, the setting's value can change between initialization and generation. For example, the BSP developer might edit the setting in the BSP Editor.

A Tcl callback can function in either of the following scopes:

- Component class
- Component instance

You specify each callback function's runtime environment by using the appropriate property in the `set_sw_property` command, as shown in Table 15–7.

Table 15–7. Callback Properties

Property as specified in set_sw_property	Run time	Scope	Callback Arguments
`initialization_callback`	Initialization	Component instance	Component instance name
`validation_callback`	Validation	Component instance	Component instance name
`generation_callback`	Generation	Component instance	Component instance name, BSP generate target directory, driver BSP subdirectory *(1)*
`class_initialization_callback`	Initialization	Component class	Driver class name
`class_validation_callback`	Validation	Component class	Driver class name
`class_generation_callback`	Generation	Component class	Driver class name, BSP generate target directory, driver BSP subdirectory *(1)*

Note to Table 15–7:

(1) The BSP subdirectory into which the driver or package files are copied

Tcl callbacks have access to a specialized set of commands, described in this section. In addition, Tcl callbacks can use any read-only BSP setting Tcl command.

Refer to "Tcl Commands for BSP Settings" on page 15–76 for details about BSP setting Tcl commands.

When a Tcl callback creates a setting, it can specify the value. However, callbacks cannot change the value of a pre-existing setting.

add_class_sw_setting

Usage

```
add_class_sw_setting <setting-name> <setting-type>
```

Options

- *<setting-name>*: Name of the setting to persist in the BSP settings file. This is prepended with the driver class name associated with this callback script
- *<setting-type>*: Type of the setting to persist in the BSP settings file.

Description

Creates a BSP setting that is associated with a particular software driver element class. The `set_class_sw_setting_property` command is required to set the values for fields pertaining to a BSP software setting definition. This command is only valid for a callback script. A callback script is set in the driver's ***_sw.tcl** file, using the command `set_sw_property callback_source_file` *`<filename>`*.

Example

```
add_class_sw_setting MY_FAVORITE_SETTING String
```

add_class_systemh_line

Usage

```
add_class_systemh_line <macro-name> <macro-value>
```

Options

- *<macro-name>*: Macro to be added to the system.h file for the generated BSP
- *<macro-value>*: Value associated with the macro-name to be added to the system.h file for the generated BSP

Description

This adds a system.h assignment or macro during a driver callback execution. The BSP typically uses this during the generate phase depending on the generator. This command is only valid for a callback script. A callback script is set in the driver's ***_sw.tcl** file, using the command `set_sw_property callback_source_file` *`<filename>`*.

Example

```
add_class_systemh_line MY_MACRO "Macro_Value";
```

add_module_sw_property

Usage

```
add_module_sw_property <property-name> <property-value>
```

Options

- *<property-name>*: Name of the property to add to the BSP for a module instance
- *<property-value>*: Value of the property to add to the BSP for a module instance

Description

This adds a software property to the BSP driver of this module instance. The BSP typically uses this during the generate phase depending on the generator. This command is only valid for a callback script. A callback script is set in the driver's ***_sw.tcl** file, using the command `set_sw_property callback_source_file <filename>`.

Example

```
add_module_sw_setting MY_FAVORITE_SETTING String
```

add_module_sw_setting

Usage

```
add_module_sw_setting <setting-name> <setting-type>
```

Options

- `<setting-name>`: Name of the setting to persist in the BSP settings file. This is prepended with the module name associated with this callback script
- `<setting-type>`: Type of the setting to persist in the BSP settings file.

Description

Creates a BSP setting that is associated with a particular instance of hardware module in a SOPC system. The `set_module_sw_setting_property` command is required to set the values for fields pertaining to a BSP software setting definition. This command is only valid for a callback script. A callback script is set in the driver's ***_sw.tcl** file, using the command `set_sw_property callback_source_file <filename>`.

Example

```
add_module_sw_setting MY_FAVORITE_SETTING String
```

add_module_systemh_line

Usage

```
add_module_systemh_line <macro-name> <macro-value>
```

Options

- `<macro-name>`: Macro to be added to the system.h file for the generated BSP
- `<macro-value>`: Value associated with the macro-name to be added to the system.h file for the generated BSP

Description

This adds a system.h assignment or macro during a driver callback execution. The BSP typically uses this during the generate phase depending on the generator. This command is only valid for a callback script. A callback script is set in the driver's ***_sw.tcl** file, using the command `set_sw_property callback_source_file <filename>`.

Example

```
add_module_systemh_line MY_MACRO "Macro_Value";
```

add_systemh_line

Usage

```
add_systemh_line <sw> <name> <value
```

- *<sw>*: The software (OS) that the **system.h** text is associated with
- *<name>*: Name of macro to write into **system.h** (left-hand side of `#define`)
- *<value>*: Name of value to assign to macro in **system.h** (right-hand side of `#define`)

Description

Adds a line of text to the **system.h** file. The `<sw>` argument is the name of the software type (typically an operating system name) that the **system.h** text applies to. In the context of an operating system Tcl script, the name in the `create_os` *<name>* command must be used. The text is a name-value pair that creates a macro (`#define` statement) in the **system.h** file.

☞ This command can only be used by Tcl scripts that are registered to run at BSP generation time by an operating system.

Example

```
add_systemh_line UCOSII OS_TICKS_PER_SEC 100
```

get_class_peripheral

Usage

```
get_class_peripheral <instance-name> <irq-number>
```

Options

- *<instance-name>*: Name of EIC module instance to find connected peripheral for.
- *<irq-number>*: IRQ number to locate connected peripheral device

Description

This command is used on an EIC instance callback to obtain a peripheral slave descriptor connected to a specific IRQ port number. This command is only valid for a callback script.

Example

```
get_class_peripheral eic_1 $irq_2;
```

get_module_assignment

Usage

```
get_module_assignment <assignment-name>
```

Options

- *<assignment-name>*: Name of the module assignment to retrieve the value for, as defined for the module instance in the **.sopcinfo** file

Description

Given a module assignment key, return the assignment value of a module associated with the callback script using this command. The callback script must be set in the ***_sw.tcl** file using the following command:

```
set_sw_property callback_source_file <filename>
```

Example

```
puts [get_module_assignment embeddedsw.configuration.isMemoryDevice]
```

Returns:

```
true
```

get_module_name

Usage

```
get_module_name
```

Options

None

Description

Returns the name of the module associated with the callback script using this command. The callback script must be set in the ***_sw.tcl** file using the following command:

```
set_sw_property callback_source_file <filename>
```

Example

```
puts [get_module_name]
```

Returns:

```
jtag_uart
```

get_module_peripheral

Usage

```
get_module_peripheral <irq-number>
```

Options

- *<irq-number>*: IRQ number to locate connected peripheral device

Description

This command is used on an EIC instance callback to obtain a peripheral slave descriptor connected to a specific IRQ port number. This command is only valid for a callback script.

Example

```
get_module_peripheral 2;
```

get_module_sw_setting_value

Usage

```
get_module_sw_setting_value <setting-name>
```

Options

- *<setting-name>*: Name of the module software setting to retrieve the value for, as defined by the `add_module_sw_setting` command.

Description

Given a module software setting name, return the setting value. The callback script using this command must be set in the ***_sw.tcl** file using the following command:

```
set_sw_property callback_source_file <filename>
```

You can use this command in a generation or validation callback to retrieve the current value of a setting created in an initialization callback.

Example

```
puts [get_module_sw_setting_value MY_SETTING]
```

Returns:

```
"My setting value"
```

get_peripheral_property

Usage

```
get_peripheral_property <slave-descriptor> <property-name>
```

Options

- *<slave-descriptor>*: Slave descriptor of a connected peripheral device
- *<property-name>*: Property name to query from the connected peripheral device

Description

This command is used on an EIC instance callback to obtain a connected peripheral property value. This command is only valid for a callback script. A callback script is set in the driver's ***_sw.tcl** file, using the command `set_sw_property callback_source_file <filename>`.

Example

```
get_peripheral_property jtag_uart supports_preemption;
```

remove_class_systemh_line

Usage

```
remove_class_systemh_line <macro-name>
```

Options

- *<macro-name>*: Macro to be removed to the system.h file for the generated BSP

Description

This removes a system.h assignment or macro during a driver callback execution. The BSP typically uses this during the generate phase depending on the generator. This command is only valid for a callback script. A callback script is set in the driver's ***_sw.tcl** file, using the command `set_sw_property callback_source_file` *<filename>*.

Example

```
remove_class_systemh_line MY_MACRO;
```

remove_module_systemh_line

Usage

```
remove_module_systemh_line <macro-name>
```

Options

- *<macro-name>*: Macro to be removed to the system.h file for the generated BSP

Description

This removes a system.h assignment or macro during a driver callback execution. The BSP typically uses this during the generate phase depending on the generator. This command is only valid for a callback script. A callback script is set in the driver's ***_sw.tcl** file, using the command `set_sw_property callback_source_file` *<filename>*.

Example

```
remove_module_systemh_line MY_MACRO;
```

set_class_sw_setting_property

Usage

```
set_class_sw_setting_property <setting-name> <property> <value>
```

Options

- *<setting-name>*: Name of the setting to persist in the BSP settings file associated with the driver class of this callback script
- *<property>*: Name of the setting property to update
- *<value>*: Value of the setting property to update

Description

Update a driver class software setting property. The setting must be added using the `add_class_sw_setting` command before calling this method. This command is only valid for a callback script. A callback script is set in the driver's ***_sw.tcl** file, using the command `set_sw_property callback_source_file` *<filename>*.

You can set the following setting properties:

- `destination`
- `identifier`
- `value`
- `default_value`
- `description`
- `restrictions`
- `group`

Example

```
set_class_sw_setting_property MY_FAVORITE_SETTING default-value '42'
```

set_module_sw_setting_property

Usage

`set_module_sw_setting_property <`*`setting-name`*`> <`*`property`*`> <`*`value`*`>`

Options

- <*`setting-name`*>: Name of the setting to persist in the BSP settings file associated with the SOPC module of this callback script
- <*`property`*>: Name of the setting property to update
- <*`value`*>: Value of the setting property to update

Description

Update a module’s software setting property. The setting must be added using the `add_module_sw_setting` command before calling this method. This command is only valid for a callback script. A callback script is set in the driver’s ***_sw.tcl** file, using the command `set_sw_property callback_source_file <`*`filename`*`>`.

You can set the following setting properties:

- `destination`
- `identifier`
- `value`
- `default_value`
- `description`
- `restrictions`
- `group`

Example

```
set_module_sw_setting_property MY_FAVORITE_SETTING default-value '42'
```

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, refer to "Settings Managed by the Software Build Tools" on page 15–34 and "Tcl Commands for BSP Settings" on page 15–76.

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 component class (for example, `altera_avalon_jtag_uart`). A software package is not associated with any particular component 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.

For more information about creating Tcl scripts, refer to "Tcl Scripts for BSP Settings" in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

The following commands are available for device driver and software package creation:

- "add_sw_property" on page 15–113
- "add_sw_setting" on page 15–115
- "create_driver" on page 15–117
- "create_os" on page 15–118
- "create_sw_package" on page 15–118
- "set_sw_property" on page 15–119

add_sw_property

Usage

`add_sw_property <property> <value>`

Options

- `<property>`: Name of property.
- `<value>`: Value assigned, or appended to the current 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 is not already 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 a Nios II assembly language source 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.
- `cpp_source`—Adds a C++ source file (**.cpp**, **.cc**, or **.cxx**) to BSPs containing your package. **nios2-bsp-generate-files** copies the 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 file (typically .a) to each BSP containing the driver or package. **nios2-bsp-generate-files** copies the precompiled library file into the BSP directory and adds both the library file name and the path (required to locate the library file) into to the BSP's **public.mk** file. Applications using the BSP link with the library file.
  The library file name 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 component hardware version. See the `version` property of the `set_sw_property` command for information about version strings. This property applies only to device drivers (see the `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.
  This property is only available for device drivers.
- `supported_bsp_type`—Adds a specific BSP type (operating system) to the list of supported operating systems that the driver or software package supports. Specify `HAL` if the software supports the Altera HAL, or operating systems that extend it. If your software is operating system-neutral and works on multiple HAL-based operating systems, state HAL only. If your software or driver contains code that depends on a particular operating system, state compatibility with that operating system only, but not HAL.
  The name of another operating system to support must match the name of the operating system exactly. This operating system name string is the same as that

used to create a BSP with the nios2-bsp-* commands, as well as in the **.tcl** script that describes the operating system, in its create_os command.
When you create a BSP with an operating system that extends HAL, such as UCOSII, and the BSP tools select a driver for a particular hardware module, precedence is given to drivers which state compatibility with a that specific operating system (OS) before a more generic driver stating HAL compatibility.
This property is only available for device drivers and software packages. This property must be set to at least one operating system.

- 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.
- excluded_hal_source—Specifies a file to exclude from the a BSP generated with an operating system that extends HAL. The value is the path to a BSP file to exclude, with respect to the BSP root. This property is optional.
- systemh_generation_script—Specifies a **.tcl** script to execute during generation of the BSP **system.h** file. This script runs with the tcl commands available to other BSP settings tcl scripts, and allow you to influence the contents of the **system.h** file. This property is available only to operating systems, created with the create_os command. 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 #define statements.

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 **nios2-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 `system_h_define` and `public_mk_define` destinations, defined below.
  - `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 destinations.
  - `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, defined below.
  - `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, defined below.
  - `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, defined below.
  - `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, defined below.
  - `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, defined below.
  - `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 destination, `add_sw_settings` formats settings as `#define <`*`setting name`*`> [<`*`setting value`*`>]` macros in the **system.h** file
  - `public_mk_define`—With this destination, `add_sw_settings` formats settings as `-D<`*`setting name`*`>[=<`*`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 destination, `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 SBT adds 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 you do 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 SBT. 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` and `create_os` commands cannot appear.

The name argument is usually distinct from all other device drivers and software packages that the SBT 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 operating systems, you need to provide a unique name for each BSP type.

This command is required, unless you use the `create_sw_package` or `create_os` commands, as appropriate.

create_os

Usage

```
create_os <name>
```

Options

- *<name>*: Name of operating system (BSP type).

Description

This command creates a new operating system (OS) instance (also known as a BSP type) available for the Nios II BSP tools. This command must precede all others that describe the OS in its Tcl script. You can only have one `create_os` command in each Tcl script. If the `create_os` command appears in the Tcl script, the `create_driver` or `create_sw_package` commands cannot appear.

The name argument is usually distinct from all other operating systems that the SBT might locate. You can specify an OS name identical to OS if the OS you are describing has a unique version number assignment.

This command is required, unless you use the `create_driver` or `create_sw_package` commands, as appropriate.

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 SBT. 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 `create_driver` or `create_os` commands cannot appear.

The name argument is usually distinct from all other device drivers and software packages that the SBT 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 operating systems, you need to provide a unique name for each BSP type.

This command is required, unless you use the `create_driver` or `create_os` commands, as appropriate.

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** shown in the Component Editor. Example: `altera_avalon_uart`. This property is only available for device drivers. 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: `9.0`, `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 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 documentation for information about version strings. This property is optional. This property is only available for device drivers.
- `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. This property is only available for device drivers and software packages.
- `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. Use this property to customize the order of macro calls in the BSP **alt_sys_init.c** file. 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 being initialized.
  This property is optional. The default priority is `1000`.
  This property is only available for device drivers and software packages.
- `display_name`—This property is used for user interfaces and other tools that wish to show a human-readable name to identify the software being described in the **.tcl** script. `display_name` is set to a few words of text (in quotes) that name your software. For example: `Altera Nios II driver`.
  This property is optional. If not set, tools that attempt to use the display name use the package name created with the appropriate `create_` command.
- `extends_bsp_type`—This property specifies which BSP type that an operating system (created with the `create_os` command) extends (if any). Currently, only the Altera HAL (`HAL`) is supported.
  This command is required for all operating systems that wish to use HAL-compatible generators in the Nios II BSP tools. It is also required for operating systems that require the Altera HAL, device driver, or software package source files that are HAL compatible in BSPs created with that operating system. An operating system that extends HAL is presumed to be compatible with device drivers that support HAL.
  This command is only available for operating systems.
- `callback_source_file`—This property specifies a Tcl source file containing callback functions.

- `initialization_callback`—This property specifies the name of a Tcl callback function which is intended to run in the following environment:
  - Run time: initialization
  - Scope: component instance
  - Function argument(s): component instance name
- `validation_callback`—This property specifies the name of a Tcl callback function which is intended to run in the following environment:
  - Run time: validation
  - Scope: component instance
  - Function argument(s): component instance name
- `generation_callback`—This property specifies the name of a callback function which is intended to run in the following environment:
  - Run time: generation
  - Scope: component instance
  - Function argument(s): component instance name, BSP generate target directory, driver BSP subdirectory
- `class_initialization_callback`—This property specifies the name of a callback function which is intended to run in the following environment:
  - Run time: initialization
  - Scope: component instance
  - Function argument(s): driver class name
- `class_validation_callback`—This property specifies the name of a callback function which is intended to run in the following environment:
  - Run time: validation
  - Scope: component instance
  - Function argument(s): driver class name
- `class_generation_callback`—This property specifies the name of a callback function which is intended to run in the following environment:
  - Run time: generation
  - Scope: component instance
  - Function argument(s): driver class name, BSP generate target directory, driver BSP subdirectory
- `supported_interrupt_apis`—Specifies the interrupt API that the device driver supports. Specify `legacy_interrupt_api` if the device driver supports the legacy API only or `enhanced_interrupt_api` if the device driver supports the enhanced API only. Specify both using a quoted list if the device driver supports both APIs.

  If you do not specify which API your device driver supports, the Nios II SBT assumes that only the legacy interrupt API is supported. The Nios II SBT analyzes this property for each driver in the system to determine the appropriate API to be used in the system.

☞ This property is only available for device drivers.

For more information about the legacy and enhanced APIs, refer to "Exception Handling" in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

- `isr_preemption_supported`—Specify true if your device driver ISR can be preempted by a higher priority ISR. If you do not specify whether ISR preemption is supported, the Nios II SBT assumes that your device driver does not support preemption. If your driver does not have an ISR, but the associated device has an interrupt port, you can set this property to true.

  ☞ This property is valid for operating systems and device drivers.

Software Build Tools Path Names

There are some restrictions on how you can specify file paths when working with the Nios II SBT. 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 SBT supports 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 SBT accepts both relative and absolute path names.

Table 15–8 shows the supported path name formats for each platform, for Nios II SBT utilities and makefiles.

Table 15–8. 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 15–8:

(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 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

The makefile created by the Nios II SBT creates a new directory tree for generated object files. To the extent possible, the object file directory tree retains the structure of the corresponding source directory.

For example, if you specify the path to a source file as

```
src/util/special/tools.c
```

the makefile places the corresponding object code in

```
obj/util/special/tools.o
```

The object file directory structure is illustrated in "Nios II Embedded Software Projects" in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

The makefile does not create object directories outside the project directory root. If the source file path you specify is a relative path beginning with "..", the Nios II SBT flattens the path name prior to creating the object directory structure.

For example, if you specify the path to a source file as

```
../special/tools.c
```

the makefile places the corresponding object code in

```
obj/tools.o
```

If you specify an absolute path to source files under Cygwin, the Nios II SBT creates 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 SBT places the corresponding object code in

```
obj/cygdrive/c/dev/app/special/tools.o
```

Document Revision History

Table 15–9 shows the revision history for this document.

Table 15–9. Document Revision History (Part 1 of 2)

Date	Version	Changes
January 2014	13.1.0	■ Replaced the -EL/-EB flags with -mel/-meb flags for endian settings. ■ Replaced -mstack-check with -fstack-check to enable stack checking. ■ Removed references to Nios II C2H. ■ Removed references to Nios II IDE. ■ Removed the "nios2-convert-ide2sbt" and "nios2-c2h-generate-makefile" commands.
May 2011	11.0.0	■ Introduction of Qsys system integration tool ■ New Tcl commands added ■ Recommend leaving `hal.enable_sopc_sysid_check` enabled
February 2011	10.1.0	■ Correction to `add_memory_device` Tcl command arguments. ■ New functionality in `nios2-bsp-create-settings` command. ■ Removed "Referenced Documents" section.
July 2010	10.0.0	■ Update documentation of `hal.enable_small_c_library` setting. ■ Describe new BSP Tcl commands: 　■ `add_memory_device` 　■ `set_ignore_file` ■ Correct missing properties in `set_sw_property` Tcl command: 　■ `supported_interrupt_apis` 　■ `isr_preemption_supported`

Table 15–9. Document Revision History (Part 2 of 2)

Date	Version	Changes
November 2009	9.1.0	■ Support for external interrupt controller. ■ Add documentation for the following utilities: 　■ **nios2-lib-update-makefile** 　■ **nios2-app-update-makefile** 　■ **nios2-convert-ide2sbt** 　■ **nios2-swexample-create** 　■ **nios2-elf-insert** 　■ **nios2-elf-query** 　■ **nios2-flash-programmer** ■ Add documentation for `--jdi` command-line argument. ■ Add documentation for example design scripts. ■ Add documentation for **hal.log_flags** setting. ■ Add documentation for interrupt stack settings. ■ Clarify information about default settings for MicroC/OS-II. ■ Clarify information about default settings for host file system. ■ Add documentation for makefile variables. ■ Add documentation for the following BSP Tcl commands: 　■ add_systemh_line 　■ get_assignment 　■ get_cpu_name 　■ get_interrupt_controller_id 　■ get_irq_interrupt_controller_id 　■ is_connected_interrupt_controller_device ■ Describe Tcl callback functions.
March 2009	9.0.0	■ Reorganized and updated information and terminology to clarify role of Nios II Software Build Tools. ■ Described usage of custom Tcl scripts. ■ Corrected minor typographical errors.
May 2008	8.0.0	■ Advanced exceptions added to Nios II core. ■ Instruction-related exception handling added to HAL. ■ Describe new BSP setting `hal.enable_instruction_related_exceptions_api`.
October 2007	7.2.0	Initial release. Reference material moved here from former *Nios II Software Build Tools* chapter.

Additional Information

This chapter provides additional information about the document and Altera.

How to Find Further Information

This handbook is one part of the complete Nios II processor documentation. The following references are also available.

- The *Nios II Processor Reference Handbook* describes the Nios II processor from a high-level conceptual description to the low-level details of implementation.
- The *Quartus II Handbook, Volume 5: Embedded Peripherals* discusses Altera-provided peripherals and Nios II drivers which are included with the Quartus II software.
- Altera's on-line solutions database is an internet resource that offers solutions to frequently asked questions via an easy-to-use search engine. You can access the database from the Knowledge Database page of the Altera website.
- Altera application notes and tutorials offer step-by-step instructions on using the Nios II processor for a specific application or purpose. You can obtain these documents from the Literature: Nios II Processor page of the Altera website.

How to Contact Altera

To locate 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
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

The following table shows the typographic conventions this document uses.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Indicate command names, dialog box titles, dialog box options, and other GUI labels. For example, **Save As** dialog box. For GUI elements, capitalization matches the GUI.
bold type	Indicates directory names, project names, disk drive names, file names, file name extensions, software utility names, and GUI labels. For example, **\qdesigns** directory, **D:** drive, and **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Indicate document titles. For example, *Stratix IV Design Guidelines*.
italic type	Indicates variables. For example, *n* + 1. Variable names are enclosed in angle brackets (< >). For example, *<file name>* and *<project name>*.**pof** file.
Initial Capital Letters	Indicate keyboard keys and menu names. For example, the Delete key and the Options menu.
“Subheading Title”	Quotation marks indicate references to sections within a document and titles of Quartus II Help topics. For example, “Typographic Conventions.”
`Courier type`	Indicates signal, port, register, bit, block, and primitive names. For example, `data1`, `tdi`, and `input`. The suffix `n` denotes an active-low signal. For example, `resetn`. Indicates command line commands and anything that must be typed exactly as it appears. For example, `c:\qdesigns\tutorial\chiptrip.gdf`. Also indicates sections of an actual file, such as a Report File, references to parts of files (for example, the AHDL keyword `SUBDESIGN`), and logic function names (for example, `TRI`).
↵	An angled arrow instructs you to press the Enter key.
1., 2., 3., and a., b., c., and so on	Numbered steps indicate a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ■ ■	Bullets indicate a list of items when the sequence of the items is not important.
☞	The hand points to information that requires special attention.
?	A question mark directs you to a software help system with related information.
👣	The feet direct you to another document or website with related information.
⚠ CAUTION	A caution calls attention to a condition or possible situation that can damage or destroy the product or your work.
⚠ WARNING	A warning calls attention to a condition or possible situation that can cause you injury.
✉	The envelope links to the Email Subscription Management Center page of the Altera website, where you can sign up to receive update notifications for Altera documents.