2. Developing Nios II Software

ED51002-1.4

Introduction

This chapter provides in-depth information about software development for the Altera® Nios® II processor. It complements the *Nios II Software Developer's Handbook* by providing the following additional information:

- **Recommended design practices**—Best practice information for Nios II software design, development, and deployment.
- **Implementation information**—Additional in-depth information about the implementation of application programming interfaces (APIs) and source code for each topic, if available.
- **Pointers to topics**—Informative background and resource information for each topic, if available.

Before reading this document, you should be familiar with the process of creating a simple board support package (BSP) and an application project using the Nios II Software Build Tools development flow. The Software Build Tools flow is supported by Nios II Software Build Tools for Eclipse™ as well as the Nios II Command Shell. This document focuses on the Nios II Software Build Tools for Eclipse, but most information is also applicable to project development in the Command Shell.

The following resources provide training on the Nios II Software Build Tools development flow:

- Online training demonstrations located on the Embedded SW Designer Curriculum page of the Altera website:
  - Developing Software for the Nios II Processor: Tools Overview
  - Developing Software for the Nios II Processor: Design Flow
  - Developing Software for the Nios II Processor: Software Build Flow (Part 1)
  - Developing Software for the Nios II Processor: Software Build Flow (Part 2)
- Documentation located on the Literature: Nios II Processor page of the Altera website, especially the *Getting Started from the Command Line* and *Getting Started with the Graphical User Interface* chapters of the *Nios II Software Developer's Handbook*.
- Example designs provided with the Nios II Embedded Design Suite (EDS). The online training demonstrations describe these software design examples, which you can use as-is or as the basis for your own more complex designs.

This chapter is structured according to the Nios II software development process. Each section describes Altera's recommended design practices to accomplish a specific task.

This chapter contains the following sections:

- "Software Development Cycle"
- "Software Project Mechanics" on page 2–5
- "Developing With the Hardware Abstraction Layer" on page 2–24
- "Optimizing the Application" on page 2–43
- "Linking Applications" on page 2–49
- "Application Boot Loading and Programming System Memory" on page 2–51

When you install the Nios II EDS, it is installed in the same directory with the Quartus II software. For example, if the Quartus II software is installed on the Windows operating system, and the root directory of the Quartus II software is **c:\altera\<***version***>\quartus**, then the root directory of the Nios II EDS is **c:\altera\<***version***>\nios2eds**. For simplicity, this handbook refers to the **nios2eds** directory as <*Nios II EDS install dir*>.

Software Development Cycle

The Nios II EDS includes a complete set of C/C++ software development tools for the Nios II processor. In addition, a set of third-party embedded software tools is provided with the Nios II EDS. This set includes the MicroC/OS-II real-time operating system and the NicheStack TCP/IP networking stack. This chapter focuses on the use of the Altera-created tools for Nios II software generation. It also includes some discussion of third-party tools.

The Nios II EDS is a collection of software generation, management, and deployment tools for the Nios II processor. The toolchain includes tools that perform low-level tasks and tools that perform higher-level tasks using the lower-level tools.

This section contains the following subsections:

- "Altera System on a Programmable Chip (SOPC) Solutions"
- "Nios II Software Development Process" on page 2–3

Altera System on a Programmable Chip (SOPC) Solutions

To understand the Nios II software development process, you must understand the definition of an SOPC Builder system. SOPC Builder is a system development tool for creating systems including processors, peripherals, and memories. The tool enables you to define and generate a complete SOPC very efficiently. SOPC Builder does not require that your system contain a Nios II processor, although it provides complete support for integrating Nios II processors with your system.

An SOPC Builder system is similar in many ways to a conventional embedded system; however, the two kinds of system are not identical. An in-depth understanding of the differences increases your efficiency when designing your SOPC Builder system.

In Altera SOPC Builder solutions, the hardware design is implemented in an FPGA device. An FPGA device is volatile—contents are lost when the power is turned off—and reprogrammable. When an FPGA is programmed, the logic cells inside it are configured and connected to create an SOPC system, which can contain Nios II processors, memories, peripherals, and other structures. The system components are connected with Avalon® interfaces. After the FPGA is programmed to implement a Nios II processor, you can download, run, and debug your system software on the system.

Understanding the following basic characteristics of FPGAs and Nios II processors is critical for developing your Nios II software application efficiently:

- FPGA devices and SOPC Builder—basic properties:
  - **Volatility**—The FPGA is functional only after it is configured, and it can be reconfigured at any time.
  - **Design**—Many Altera SOPC systems are designed using SOPC Builder and the Quartus® II software, and may include multiple peripherals and processors.
  - **Configuration**—FPGA configuration can be performed through a programming cable, such as the USB-Blaster™ cable, which is also used for Nios II software debugging operations.
  - **Peripherals**—Peripherals are created from FPGA resources and can appear anywhere in the Avalon memory space. Most of these peripherals are internally parameterizable.
- Nios II processor—basic properties:
  - **Volatility**—The Nios II processor is volatile and is only present after the FPGA is configured. It must be implemented in the FPGA as a system component, and, like the other system components, it does not exist in the FPGA unless it is implemented explicitly.
  - **Parameterization**—Many properties of the Nios II processor are parameterizable in SOPC Builder, including core type, cache memory support, and custom instructions, among others.
  - **Processor Memory**—The Nios II processor must boot from and run code loaded in an internal or external memory device.
  - **Debug support**—To enable software debug support, you must configure the Nios II processor with a debug core. Debug communication is performed through a programming cable, such as the USB-Blaster cable.
  - **Reset vector**—The reset vector address can be configured to any memory location.
  - **Exception vector**—The exception vector address can be configured to any memory location.

Nios II Software Development Process

This section provides an overview of the Nios II software development process and introduces terminology. The rest of the chapter elaborates the description in this section.

The Nios II software generation process includes the following stages and main hardware configuration tools:

1. Hardware configuration
   - SOPC Builder
   - Quartus II software
2. Software project management
   - BSP configuration
   - Application project configuration
   - Editing and building the software project
   - Running, debugging, and communicating with the target
   - Ensuring hardware and software coherency
   - Project management
3. Software project development
   - Developing with the Hardware Abstraction Layer (HAL)
   - Programming the Nios II processor to access memory
   - Writing exception handlers
   - Optimizing the application for performance and size
4. Application deployment
   - Linking (run-time memory)
   - Boot loading the system application
   - Programming flash memory

In this list of stages and tools, the subtopics under the topics Software project management, Software project development, and Application deployment correspond closely to sections in the chapter.

You create the hardware for the system using the Quartus II and SOPC Builder software. The main output produced by generating the hardware for the system is the SRAM Object File (**.sof**), which is the hardware image of the system, and the SOPC Information File (**.sopcinfo**), which describes the hardware components and connections.

The key file required to generate the application software is the **.sopcinfo** file.

The software generation tools use the **.sopcinfo** file to create a BSP project. The BSP project is a collection of C source, header and initialization files, and a makefile for building a custom library for the hardware in the system. This custom library is the BSP library file (**.a**). The BSP library file is linked with your application project to create an executable binary file for your system, called an application image. The combination of the BSP project and your application project is called the software project.

The application project is your application C source and header files and a makefile that you can generate by running Altera-provided tools. You can edit these files and compile and link them with the BSP library file using the makefile. Your application sources can reference all resources provided by the BSP library file. The BSP library file contains services provided by the HAL, which your application sources can reference. After you build your application image, you can download it to the target system, and communicate with it through a terminal application.

You can access the makefile in the Eclipse **Project Explorer** view after you have created your project in the Nios II Software Build Tools for Eclipse framework.

The software project is flexible: you can regenerate it if the system hardware changes, or modify it to add or remove functionality, or tune it for your particular system. You can also modify the BSP library file to include additional Altera-supplied software packages, such as the read-only zip file system or TCP/IP networking stack (the NicheStack TCP/IP Stack). Both the BSP library file and the application project can be configured to build with different parameters, such as compiler optimizations and linker settings.

If you change the hardware system, you must recreate, update or regenerate the BSP project to keep the library header files up-to-date.

For information about how to keep your BSP up-to-date with your hardware, refer to "Revising Your BSP" in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

Software Project Mechanics

This section describes the recommended ways to edit, build, download, run, and debug your software application, primarily using the Nios II Software Build Tools for Eclipse.

The Nios II Software Build Tools flow is the recommended design flow for hardware designs that contain a Nios II processor. This section describes how to configure BSP and application projects, and the process of developing a software project for a system that contains a Nios II processor, including ensuring coherency between the software and hardware designs.

This section contains the following subsections:

Software Tools Background

The Nios II EDS provides a sophisticated set of software project generation tools to build your application image. Two separate software-development methodologies are available for project creation: the Nios II Software Build Tools flow and the Nios II Integrated Development Environment (IDE) flow. The Nios II Software Build Tools flow includes the Software Build Tools command-line interface and the Nios II Software Build Tools for Eclipse.

The Nios II Software Build Tools for Eclipse is the recommended flow. However, the Nios II Software Build Tools for Eclipse does not support the following Nios II IDE features:

- Trace features available in the FS2 console do not work correctly with the System Console debugger.
- `stdio` output to an RS-232 UART cannot display on the System Console. To display `stdio` output on the System Console, configure your BSP to use a JTAG UART peripheral for `stdout`, using the `hal.stdout` BSP setting. If no JTAG UART is available in your hardware system, you can run **nios2-terminal** in a separate Nios II Command Sheell to capture `stdio` output.

The Nios II Software Build Tools development flow provides an easily controllable development environment for creating, managing, and configuring software applications. The Nios II Software Build Tools include command-line utilities, scripts, and tools for Tcl scripting. Starting in version 9.1 of the Nios II EDS, Altera provides Nios II Software Build Tools for Eclipse, which is a user-friendly, graphical user interface (GUI)-based version of the Software Build Tools flow.

Altera recommends that you use the Nios II Software Build Tools for Eclipse to create new software projects. The Nios II Software Build Tools are the basis for Altera's future development.

For information about migrating existing Nios II IDE projects to the Nios II Software Build Tools flow, refer to "Porting Nios II IDE Projects to the Software Build Tools" in the *Using the Nios II Integrated Development Environment* appendix to the *Nios II Software Developer's Handbook*.

In the Nios II IDE design flow, the BSP project is called a system library.

A graphical user interface for configuring BSP libraries, called the Nios II BSP Editor, is also available. The BSP Editor is integrated with the Nios II Software Build Tools for Eclipse, and can also be used independently.

Development Flow Guidelines

The Nios II Software Build Tools flow provides many services and functions for your use. Until you become familiar with these services and functions, Altera recommends that you adhere to the following guidelines to simplify your development effort:

- **Begin with a known hardware design**—The Nios II Embedded Processor Design Examples web page of the Altera website includes a set of known working designs, called hardware example designs, which are excellent starting points for your own design. In addition, the *Nios II Hardware Development Tutorial* walks through some example designs.
- **Begin with a known software example design**—The Nios II EDS includes a set of preconfigured application projects for you to use as the starting point of your own application. Use one of these designs and parameterize it to suit your application goals.
- **Follow pointers to documentation**—Many of the application and BSP project source files include inline comments that provide additional information.
- **Make incremental changes**—Regardless of your end-application goals, develop your software application by making incremental, testable changes, to compartmentalize your software development process. Altera recommends that you use a version control system to maintain distinct versions of your source files as you develop your project.

The following section describes how to implement these guidelines.

Nios II Software Build Tools Flow

The Nios II Software Build Tools are a collection of command-line utilities and scripts. These tools allow you to build a BSP project and an application project to create an application image. The BSP project is a parameterizable library, customized for the hardware capabilities and peripherals in your system. When you create a BSP library file from the BSP project, you create it with a specific set of parameter values. The application project consists of your application source files and the application makefile. The source files can reference services provided by the BSP library file.

For the full list of utilities and scripts in the Nios II Software Build Tools flow, refer to "Altera-Provided Embedded Development Tools" in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

The Nios II Software Build Tools for Eclipse

The Nios II Software Build Tools for Eclipse provide a consistent development platform that works for all Nios II processor systems. You can accomplish most software development tasks in the Nios II Software Build Tools for Eclipse, including creating, editing, building, running, debugging, and profiling programs.

The Nios II Software Build Tools for Eclipse are based on the popular Eclipse™ framework and the Eclipse C/C++ development toolkit (CDT) plug-ins. Simply put, the Nios II Software Build Tools for Eclipse provides a GUI that runs the Nios II Software Build Tools utilities and scripts behind the scenes.

For detailed information about the Nios II Software Build Tools 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 Software Build Tools Command Line

In the Nios II Software Build Tools command line development flow, you create, modify, build, and run Nios II programs with Nios II Software Build Tools commands typed at a command line or embedded in a script.

To debug your program, import your Software Build Tools projects to Eclipse. You can further edit, rebuild, run, and debug your imported project in Eclipse.

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

Configuring BSP and Application Projects

This section describes some methods for configuring the BSP and application projects that comprise your software application, while encouraging you to begin your software development with a software example design.

For information about using version control, copying, moving and renaming a BSP project, and transferring a BSP project to another person, refer to "Common BSP Tasks" in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

Software Example Designs

The best way to become acquainted with the Nios II Software Build Tools flow and begin developing software for the Nios II processor is to use one of the pre-existing software example designs that are provided with the Nios II EDS. The software example designs are preconfigured software applications that you can use as the basis for your own software development. The software examples can be found in the Nios II installation directory.

For more information about the software example designs provided in the Nios II EDS, refer to "Nios II Embedded Design Examples" in the *Overview of Nios II Embedded Development* chapter of the *Nios II Software Developer's Handbook*.

To use a software example design, follow these steps:

1. Set up a working directory that contains your system hardware, including the system **.sopcinfo** file.

   Ensure that you have compiled the system hardware with the Quartus II software to create up-to-date **.sof** and **.sopcinfo** files.

2. Start the Nios II Software Build Tools for Eclipse as follows:
   - 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**.
   - In the Linux operating system, in a command shell, type `eclipse-nios2↵`.
3. Right-click anywhere in the **Project Explorer** view, point to **New** and click **Nios II Application and BSP from Template**.

4. Select an appropriate software example from the **Templates** list.

   You must ensure that your system hardware satisfies the requirements for the software example design listed under **Template description**. If you use an Altera Nios II development kit, the software example designs supplied with the kit are guaranteed to work with the hardware examples included with the kit.

5. Next to **SOPC Information File Name**, browse to your working directory and select the **.sopcinfo** file associated with your system.
6. In a multiprocessor design, you must select the processor on which to run the software project.

   If your design contains a single Nios II processor, the processor name is automatically filled in.

7. Fill in the project name.
8. Click **Next**.
9. Select **Create a new BSP project based on the application project template**.
10. Click **Finish**. The Nios II Software Build Tools generate an Altera HAL BSP for you.

If you do not want the Software Build Tools for Eclipse to automatically create a BSP for you, at Step 9, select **Select an existing BSP project from your workspace**. You then have several options:

- You can import a pre-existing BSP by clicking **Import**.
- You can create a HAL or MicroC/OS-II BSP as follows:
  a. Click **Create**. The **Nios II Board Support Package** dialog box appears.
  b. Next to **Operating System**, select either **Altera HAL** or **Micrium MicroC/OS-II**.

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

Selecting the Operating System (HAL versus MicroC/OS-II RTOS)

You have a choice of the following run-time environments (operating systems) to incorporate in your BSP library file:

- The Nios II HAL—A lightweight, POSIX-like, single-threaded library, sufficient for many applications.
- The MicroC/OS-II RTOS—A real-time, multi-threaded environment. The Nios II implementation of MicroC/OS-II is based on the HAL, and includes all HAL services.

After you select HAL or MicroC/OS-II, you cannot change the operating system for this BSP project.

Configuring the BSP Project

The BSP project is a configurable library. You can configure your BSP project to incorporate your optimization preferences—size, speed, or other features—in the custom library you create. This custom library is the BSP library file (**.a**) that is used by the application project.

Creating the BSP project populates the target directory with the BSP library file source and build file scripts. Some of these files are copied from other directories and are not overwritten when you recreate the BSP project. Others are generated when you create the BSP project.

The most basic tool for configuring BSPs is the BSP setting. Throughout this chapter, many of the project modifications you can make are based on BSP settings. In each case, this chapter presents the names of the relevant settings, and explains how to select the correct setting value. You can control the value of BSP settings several ways: on the command line, with a Tcl script, by directly adjusting the settings with the BSP Editor, or by importing a Tcl script to the BSP Editor.

Another powerful tool for configuring a BSP is the software package. Software packages add complex capabilities to your BSP. As when you work with BSP settings, you can add and remove software packages on the command line, with a Tcl script, directly with the BSP Editor, or by importing a Tcl script to the BSP Editor.

Altera recommends that you use the Nios II BSP Editor to configure your BSP project. To start the Nios II BSP Editor from the Nios II Software Build Tools for Eclipse, right-click an existing BSP, point to **Nios II**, and click **BSP Editor**.

For detailed information about how to manipulate BSP settings and add and remove software packages with 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*. This chapter also discusses how to use Tcl scripts in the BSP Editor. For information about manipulating BSP settings and controlling software packages at the command line, 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 details about available 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*. For a discussion of Tcl scripting, refer to "Software Build Tools Tcl Commands" in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Do not edit BSP files, because they are overwritten by the Software Build Tools the next time the BSP is generated.

MicroC/OS-II RTOS Configuration Tips

If you use the MicroC/OS-II RTOS environment, be aware of the following properties of this environment:

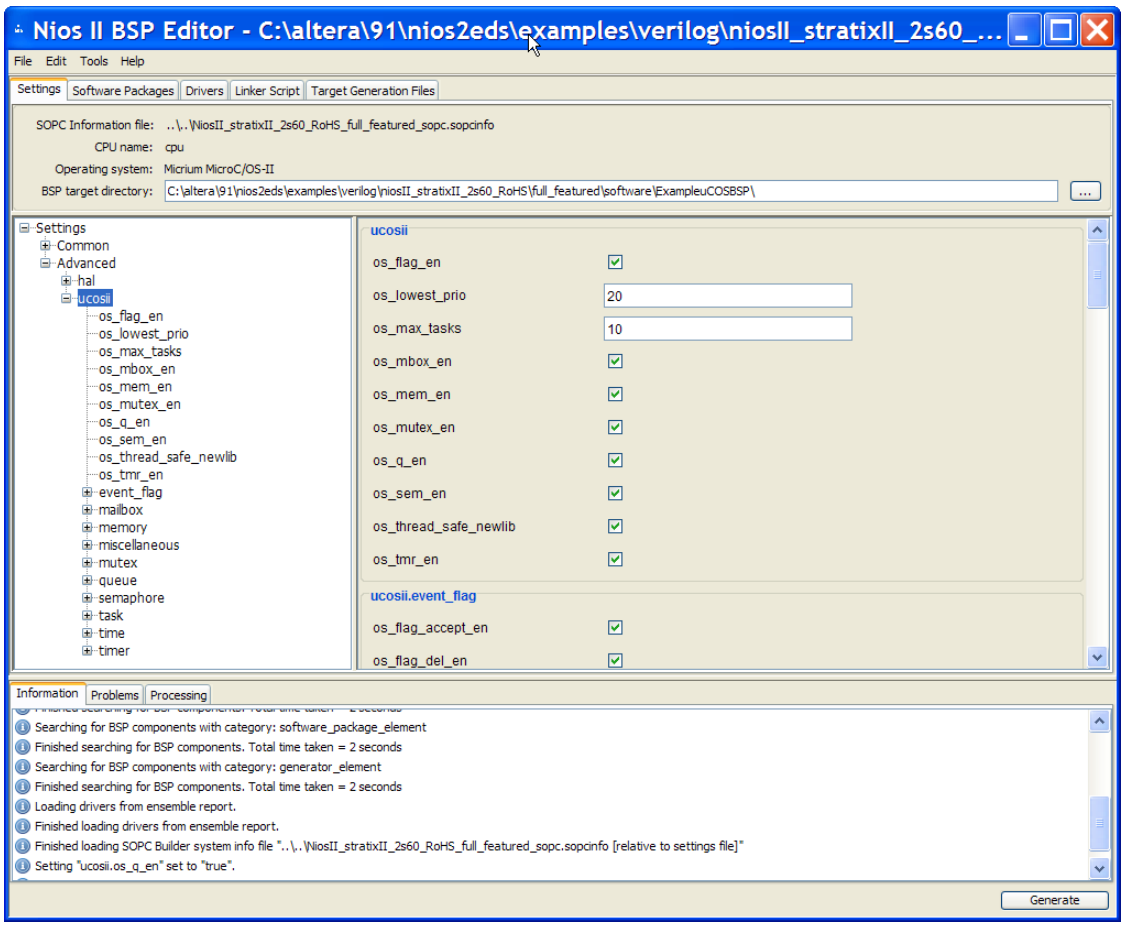
- **MicroC/OS-II BSP settings**—The MicroC/OS-II RTOS supports many configuration options. All of these options can be enabled and disabled with BSP settings. Some of the options are enabled by default. A comprehensive list of BSP settings for MicroC/OS-II is shown in the **Settings** tab of the Nios II BSP Editor.

The MicroC/OS-II BSP settings are also described in "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 setting modification**—Modifying the MicroC/OS-II options modifies the **system.h** file, which is used to compile the BSP library file.
- **MicroC/OS-II initialization**—The core MicroC/OS-II RTOS is initialized during the execution of the C run-time initialization (`crt0`) code block. After the `crt0` code block runs, the MicroC/OS-II RTOS resources are available for your application to use. For more information, refer to "crt0 Initialization" on page 2–26.

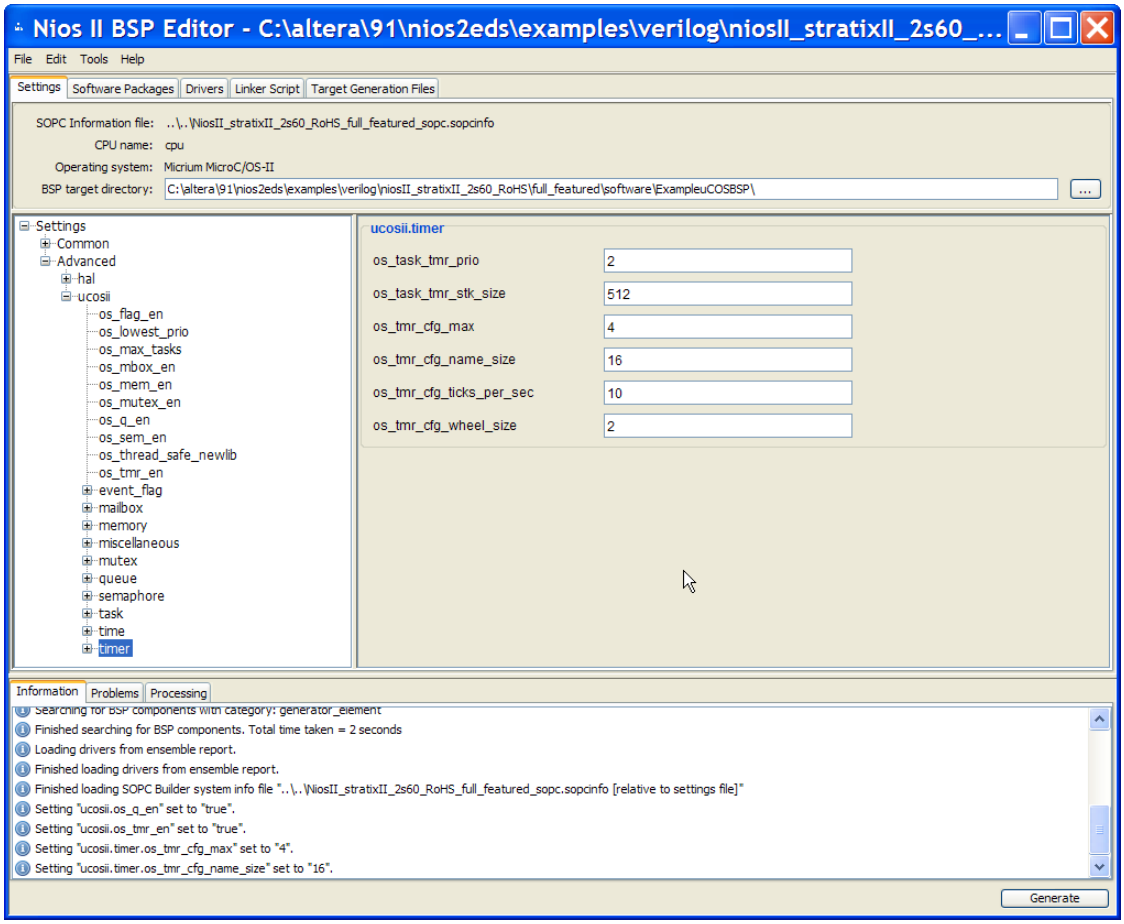
You can configure MicroC/OS-II with the BSP Editor. Figure 2–1 shows how you enable the MicroC/OS-II timer and queue code. Figure 2–2 on page 2–12 shows how you specify a maximum of four timers for use with MicroC/OS-II.

Figure 2–1. Enabling MicroC/OS-II Timers and Queues in BSP Editor



The MicroC/OS-II configuration script in Example 2–1 on page 2–12 performs the same MicroC/OS-II configuration as Figure 2–1 and Figure 2–2: it enables the timer and queue code, and specifies a maximum of four timers.

Figure 2–2. Configuring MicroC/OS-II for Four Timers in BSP Editor



Example 2–1. MicroC/OS-II Tcl Configuration Script Example (ucosii_conf.tcl)

```
#enable code for UCOSII timers
set_setting ucosii.os_tmr_en 1

#enable a maximum of 4 UCOSII timers
set_setting ucosii.timer.os_tmr_cfg_max 4

#enable code for UCOSII queues
set_setting ucosii.os_q_en 1
```

HAL Configuration Tips

If you use the HAL environment, be aware of the following properties of this environment:

- **HAL BSP settings**—A comprehensive list of options is shown in the **Settings** tab in the Nios II BSP Editor. These options include settings to specify a pre- and post-process to run for each C or C++ file compiled, and for each file assembled or archived.

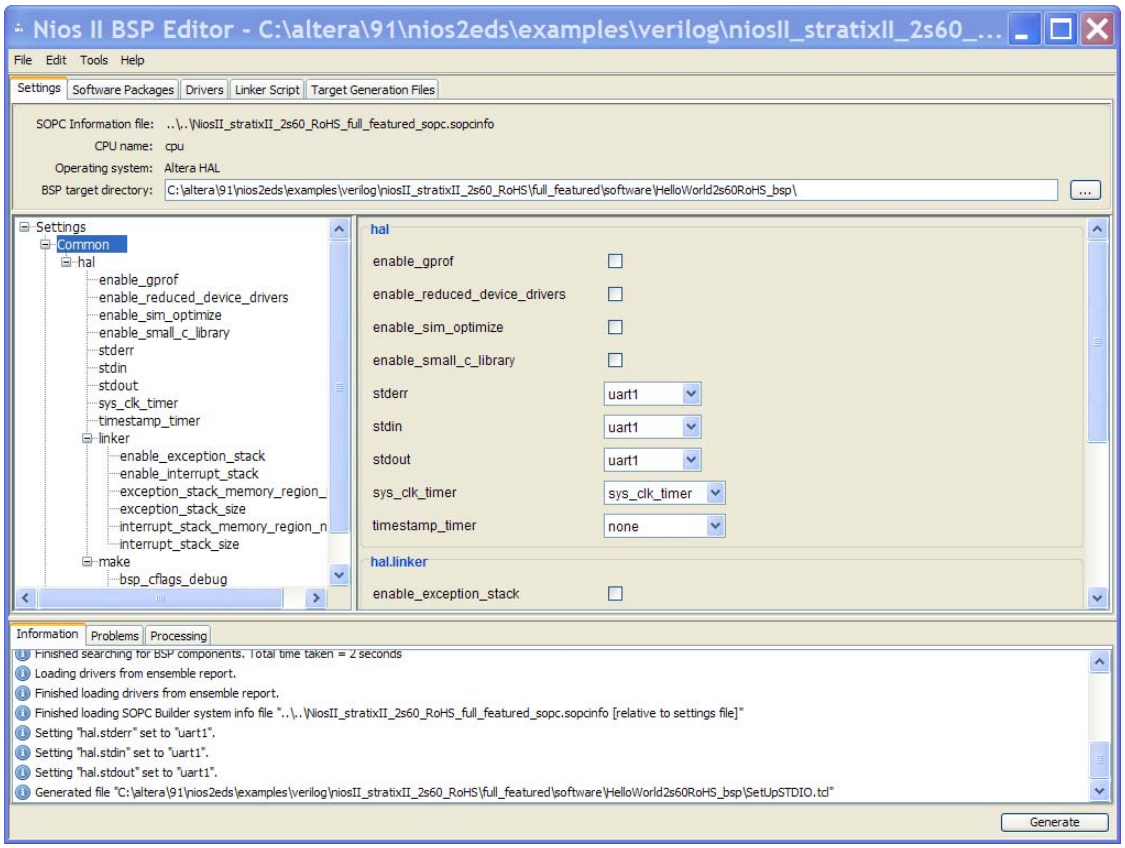
  For more information about BSP settings, refer to "Settings Managed by the Software Build Tools" in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

- **HAL setting modification**—Modifying the HAL options modifies the **system.h** file, which is used to compile the BSP library file.
- **HAL initialization**—The HAL is initialized during the execution of the C run-time initialization (`crt0`) code block. After the `crt0` code block runs, the HAL resources are available for your application to use. For more information, refer to "crt0 Initialization" on page 2–26.

You can configure the HAL in the BSP Editor. Figure 2–3 on page 2–14 shows how you specify a UART to be used as the `stdio` device.

The Tcl script in Example 2–2 on page 2–14 performs the same configuration as Figure 2–3: it specifies a UART to be used as the `stdio` device.

Figure 2–3. Configuring HAL stdio Device in BSP Editor



Example 2–2. HAL Tcl Configuration Script Example (hal_conf.tcl)

```
#set up stdio file handles to point to a UART
set default_stdio uart1
set_setting hal.stdin $default_stdio
set_setting hal.stdout $default_stdio
set_setting hal.stderr $default_stdio
```

Adding Software Packages

Altera supplies several add-on software packages in the Nios II EDS. These software packages are available for your application to use, and can be configured in the BSP Editor from the **Software Packages** tab. The **Software Packages** tab allows you to insert and remove software packages in your BSP, and control software package settings. The software package table at the top of this tab lists 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.

The following software packages are provided with the Nios II EDS:

- **Host File System**—Allows a Nios II system to access a file system that resides on the workstation. For more information, refer to "The Host-Based File System" on page 2–38.
- **Read-Only Zip File System**—Provides access to a simple file system stored in flash memory. For more information, refer to "Read-Only Zip File System" on page 2–38.
- **NicheStack TCP/IP Stack – Nios II Edition**—Enables support of the NicheStack TCP/IP networking stack.

For more information about the NicheStack TCP/IP networking stack, refer to the *Ethernet and the TCP/IP Networking Stack - Nios II Edition* chapter of the *Nios II Software Developer's Handbook*.

Using Tcl Scripts with the Nios II BSP Editor

The Nios II BSP Editor supports Tcl scripting. Tcl scripting in the Nios II BSP Editor is a simple but powerful tool that allows you to easily migrate settings from one BSP to another. This feature is especially useful if you have multiple software projects utilizing similar BSP settings. Tcl scripts in the BSP editor allow you to perform the following tasks:

- Regenerate the BSP from the command line
- Export a TCL script from an existing 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 and BSP settings

You can configure a BSP either by importing your own manually-created Tcl script, or by using a Tcl script exported from the Nios II BSP Editor.

You can apply a Tcl script only at the time that you create the BSP.

Exporting a Tcl Script

To export a Tcl script, follow these steps:

1. Use the Nios II BSP Editor to configure the BSP settings in an existing BSP project.
2. In the Tools menu, click **Export Tcl Script.**
3. Navigate to the directory where you wish to store your Tcl script.
4. Select a file name for the Tcl script.

When creating a Tcl script, the Nios II BSP Editor only exports settings that differ from the BSP defaults. For example, if the only nondefault settings in the BSP are those shown in Figure 2–3 on page 2–14, the BSP Editor exports the script shown in Example 2–3.

Example 2–3. Tcl Script Exported by BSP Editor

```
#############################################################################
#
#  This is a generated Tcl script exported
#  by a user of the Altera Nios II BSP Editor.
#
#  It can be used with the Altera 'nios2-bsp' shell script '--script'
#  option to customize a new or existing BSP.
#
#############################################################################

#############################################################################
#
# Exported Setting Changes
#
#############################################################################

set_setting hal.stdout uart1
set_setting hal.stderr uart1
set_setting hal.stdin uart1
```

For details about default BSP settings, refer to "Specifying BSP Defaults" in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook.*

Importing a Tcl Script to Create a New BSP

The following example illustrates how to configure a new BSP with an imported Tcl script. You import the Tcl script with the Nios II BSP Editor, when you create a new BSP settings file.

In this example, you create the Tcl script by hand, with a text editor. You can also use a Tcl script exported from another BSP, as described in "Exporting a Tcl Script".

To configure a new BSP with a Tcl script, follow these steps:

1. With any text editor, create a new file called **example.tcl**.
2. Insert the contents of Example 2–4 in the file.

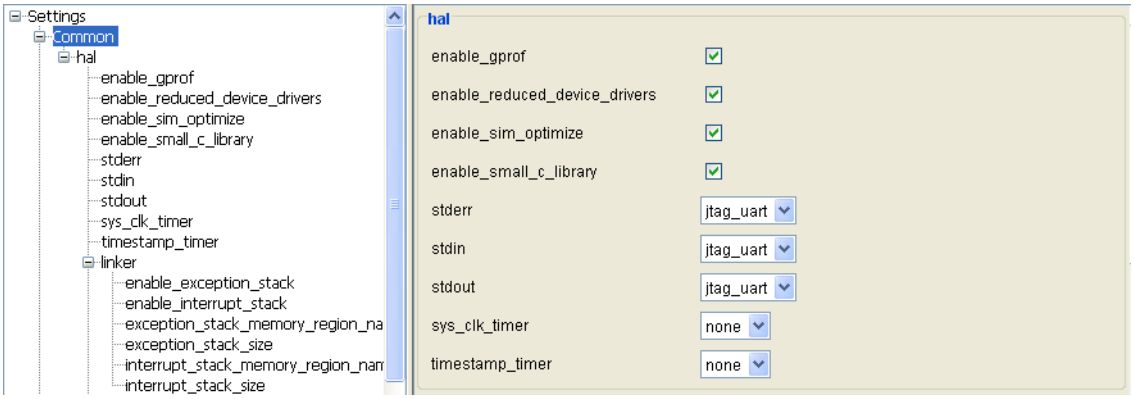
Example 2–4. BSP Configuration Tcl Script example.tcl

```
set_setting hal.enable_reduced_device_drivers true
set_setting hal.enable_sim_optimize true
set_setting hal.enable_small_c_library true
set_setting hal.enable_gprof true
```

3. In the Nios II BSP Editor, in the File menu, click **New BSP.**
4. In the **BSP Settings File Name** box, select a folder in which to save your new BSP settings file. Accept the default settings file name, **settings.bsp.**
5. In the **Operating System** list, select **Altera HAL**.
6. In the **Additional Tcl script** box, navigate to **example.tcl**.

7. In the **SOPC Information File Name** box, select the **.sopcinfo** file.
8. Click **OK**. The BSP Editor creates the new BSP. The settings modified by **example.tcl** appear as in Figure 2–4.

Figure 2–4. Nios II BSP Settings Configured with example.tcl



Do not attempt to import an Altera HAL Tcl script to a MicroC/OS-II BSP or vice-versa. Doing so could result in unpredictable behavior, such as lost settings. Some BSP settings are OS-specific, making scripts from different OSes incompatible.

For more information about commands that can appear in BSP Tcl scripts, refer to "Software Build Tools Tcl Commands" in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook.*

Configuring the Application Project

You configure the application project by specifying source files and a valid BSP project, along with other command-line options to the `nios2-app-generate-makefile` or `nios2-app-update-makefile` commands.

Application Configuration Tips

Use the following tips to increase your efficiency in designing your application project:

- **Source file inclusion**—To add source files to your project, drag them from a file browser, such as Windows Explorer, and drop them in the **Project Explorer** view in the Nios II Software Build Tools for Eclipse.

  From the command line, several options are available for specifying the source files in your application project. If all your source files are in the same directory, use the `--src-dir` command-line option. If all your source files are contained in a single directory and its subdirectories, use the `--src-rdir` command-line option.

- **Makefile variables**—When a new project is created in the Nios II Software Build Tools for Eclipse, a makefile is automatically generated in the software project directory. You can modify application makefile variables with the Nios II Application Wizard.

  From the command line, set makefile variables with the `--set <var> <value>` command-line option during configuration of the application project. The variables you can set include the pre- and post-processing settings `BUILD_PRE_PROCESS` and `BUILD_POST_PROCESS` to specify commands to be executed before and after building the application. Examine a generated application makefile to ensure you understand the current and default settings.

- **Creating top level generation script**—From the command line, simplify the parameterization of your application project by creating a top level shell script to control the configuration. The **create-this-app** scripts in the embedded processor design examples available from the Nios II Embedded Processor Design Examples web page are good models for your configuration script.

Linking User Libraries

You can create and use your own user libraries in the Nios II Software Build Tools. The Nios II Software Build Tools for Eclipse includes the Nios II Library wizard, which enables you to create a user library in a GUI environment.

You can also create user libraries in the Nios II Command Shell, as follows:

1. Create the library using the **nios2-lib-generate-makefile** command. This command generates a **public.mk** file.
2. Configure the application project with the new library by running the **nios2-app-generate-makefile** command with the `--use-lib-dir` option. The value for the option specifies the path to the library's **public.mk** file.

Makefiles and the Nios II Software Build Tools for Eclipse

The Nios II Software Build Tools for Eclipse create and manage the makefiles for Nios II software projects. When you create a project, the Nios II Software Build Tools create a makefile based on parameters and settings you select. When you modify parameters and settings, the Nios II Software Build Tools update the makefile to match. BSP makefiles are based on the operating system, BSP settings, selected software packages, and selected drivers.

Nios II BSP makefiles are handled differently from application and user library makefiles. Nios II application and user library makefiles are based on source files that you specify directly. The following changes to an application or user library change the contents of the corresponding makefile:

- Change the application or user library name
- Add or remove source files
- Specify a path to an associated BSP
- Specify a path to an associated user library
- Enable, disable or modify compiler options

For information about BSPs and makefiles, refer to "Makefiles and the Nios II Software Build Tools for Eclipse" in the *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook*.

Building and Running the Software in Nios II Software Build Tools for Eclipse

Building the Project

After you edit the BSP settings and properties, and generate the BSP (including the makefile), you can build your project. Right-click your project in the **Project Explorer** view and click **Build Project**.

Downloading and Running the Software

To download and run or debug your program, right-click your project in the **Project Explorer** view. To run the program, point to **Run As** and click **Nios II Hardware**.

Before you run your target application, ensure that your FPGA is configured with the target hardware image in your **.sof** file.

Communicating with the Target

The Nios II Software Build Tools for Eclipse provide a console window through which you can communicate with your system. When you use the Nios II Software Build Tools for Eclipse to communicate with the target, characters you input are transmitted to the target line by line. Characters are visible to the target only after you press the Enter key on your keyboard.

If you configured your application to use the `stdio` functions in a UART or JTAG UART interface, you can use the **nios2-terminal** application to communicate with your target subsystem. However, the Nios II Software Build Tools for Eclipse and the **nios2-terminal** application handle input characters very differently.

On the command line, you must use the **nios2-terminal** application to communicate with your target. To start the application, type the following command:

```
nios2-terminal ↵
```

When you use the **nios2-terminal** application, characters you type in the shell are transmitted, one by one, to the target.

Software Debugging in Nios II Software Build Tools for Eclipse

This section describes how to debug a Nios II program using the Nios II Software Build Tools for Eclipse. You can debug a Nios II program on Nios II hardware such as a Nios development board. To debug a software project, right-click the application project name, point to **Debug As** and click **Nios II Hardware**.

Do not select **Local C/C++ Application**. Nios II projects can only be run and debugged with Nios II run configurations.

For more information about using the Nios II Software Build Tools for Eclipse to debug your application, refer to the *Debugging Nios II Designs* chapter of the *Embedded Design Handbook*.

For debugging purposes, it is useful to enable run-time stack checking, using the `hal.enable_runtime_stack_checking` BSP setting. When properly used, this setting enables the debugger to take control if the stack collides with the heap or with statically allocated data in memory.

For information about how to use run-time stack checking, refer to "Run-Time Analysis Debug Techniques" in the *Debugging Nios II Designs* chapter of the *Embedded Design Handbook*. For more information about this and other BSP configuration 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*.

Ensuring Software Project Coherency

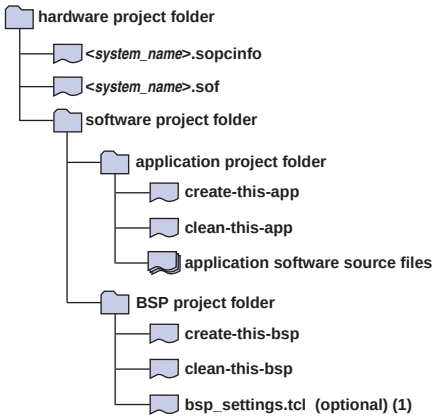
In some engineering environments, maintaining coherency between the software and system hardware projects is difficult. For example, in a mixed team environment in which a hardware engineering team creates new versions of the hardware, independent of the software engineering team, the potential for using the incorrect version of the software on a particular version of the system hardware is high. Such an error may cause engineers to spend time debugging phantom issues. This section discusses several design and software architecture practices that can help you avoid this problem.

Recommended Development Practice

The safest software development practice for avoiding the software coherency problem is to follow a strict hardware and software project hierarchy, and to use scripts to generate your application and BSP projects.

One best practice is to structure your application hierarchy with parallel application project and BSP project folders. In Figure 2–5, a top-level hardware project folder includes the Quartus II project file, the SOPC Builder-generated files, and the software project folder. The software project folder contains a subfolder for the application project and a subfolder for the BSP project. The application project folder contains a **create-this-app** script, and the BSP project folder contains a **create-this-bsp** script.

Figure 2–5. Recommended Directory Structure



Note for Figure 2–5:

(1) **bsp_settings.tcl** is a Tcl configuration file. For more information about the Tcl configuration file, refer to "Configuring the BSP Project" on page 2–10.

To build your own software project from the command line, create your own **create-this-app** and **create-this-bsp** scripts. Altera recommends that you also create **clean-this-app** and **clean-this-bsp** scripts. These scripts perform the following tasks:

- **create-this-app**—This **bash** script uses the **nios2-app-generate-makefile** command to create the application project, using the application software source files for your project. The script verifies that the BSP project is properly configured (a **settings.bsp** file is present in the BSP project directory), and runs the **create-this-bsp** script if necessary. The Altera-supplied **create-this-app** scripts that are included in the embedded design examples on the Nios II Embedded Processor Design Examples web page of the Altera website provide good models for this script.
- **clean-this-app**—This **bash** script performs all necessary clean-up tasks for the whole project, including the following:
  - Call the application makefile with the clean-all target.
  - Call the **clean-this-bsp** shell script.
- **create-this-bsp**—This **bash** script generates the BSP project. The script uses the **nios2-bsp** command, which can optionally call the configuration script **bsp_settings.tcl**. The **nios2-bsp** command references the **<***system_name***>.sopcinfo** file located in the hardware project folder. Running this script creates the BSP project, and builds the BSP library file for the system.
- **clean-this-bsp**—This **bash** script calls the clean target in the BSP project makefile and deletes the **settings.bsp** file.

The complete system generation process, from hardware to BSP and application projects, must be repeated every time a change is made to the system in SOPC Builder. Therefore, defining all your settings in your **create-this-bsp** script is more efficient than using the Nios II BSP Editor to customize your project. The system generation process follows:

1. **Hardware files generation**—Using SOPC Builder, write the updated system description to the <*system_name*>**.sopcinfo** file.
2. **Regenerate BSP project**—Generate the BSP project with the **create-this-bsp** script.
3. **Regenerate application project**—Generate the application project with the **create-this-app** script. This script typically runs the **create-this-bsp** script, which builds the BSP project by creating and running the makefile to generate the BSP library file.
4. **Build the system**—Build the system software using the application and BSP makefile scripts. The **create-this-app** script runs `make` to build both the application project and the BSP library.

To implement this system generation process, Altera recommends that you use the following checklists for handing off responsibility between the hardware and software groups.

This method assumes that the hardware engineering group installs the Nios II EDS. If so, the hardware and software engineering groups must use the same version of the Nios II EDS toolchain.

To hand off the project from the hardware group to the software group, perform the following steps:

1. **Hardware project hand-off**—The hardware group provides copies of the <*system_name*>**.sopcinfo** and <*system_name*>**.sof** files. The software group copies these files to the software group's hardware project folder.
2. **Recreate software project**—The software team recreates the software application for the new hardware by running the **create-this-app** script. This script runs the **create-this-bsp** script.
3. **Build**—The software team runs `make` in its application project directory to regenerate the software application.

To hand off the project from the software group to the hardware group, perform the following steps:

1. **Clean project directories**—The software group runs the **clean-this-app** script.
2. **Software project folder hand-off**—The software group provides the hardware group with the software project folder structure it generated for the latest hardware version. Ideally, the software project folder contains only the application project files and the application project and BSP generation scripts.
3. **Reconfigure software project**—The hardware group runs the **create-this-app** script to reconfigure the group's application and BSP projects.
4. **Build**—The hardware group runs `make` in the application project directory to regenerate the software application.

Recommended Architecture Practice

Many of the hardware and software coherency issues that arise during the creation of the application software are problems of misplaced peripheral addresses. Because of the flexibility provided by SOPC Builder, almost any peripheral in the system can be assigned an arbitrary address, or have its address modified during system creation. Implement the following practices to prevent this type of coherency issue during the creation of your software application:

- **Peripheral and Memory Addressing**—The Nios II Software Build Tools automatically generate a system header file, **system.h**, that defines a set of `#define` symbols for every peripheral in the system. These definitions specify the peripheral name, base address location, and address span. If the Memory Management Unit (MMU) is enabled in your Nios II system, verify that the address span for all peripherals is located in direct-mapped memory, outside the memory address range managed by the MMU.

  To protect against coherency issues, access all system peripherals and memory components with their **system.h** name and address span symbols. This method guarantees successful peripheral register access even after a peripheral's addressable location changes.

  For example, if your system includes a UART peripheral named UART1, located at address 0x1000, access the UART1 registers using the **system.h** address symbol (`iowr_32(UART1_BASE, 0x0, 0x10101010)`) rather than using its address (`iowr_32(0x1000, 0x0, 0x10101010)`).

- **Checking peripheral values with the preprocessor**—If you work in a large team environment, and your software has a dependency on a particular hardware address, you can create a set of C preprocessor `#ifdef` statements that validate the hardware during the software compilation process. These `#ifdef` statements validate the `#define` values in the **system.h** file for each peripheral.

  For example, for the peripheral UART1, assume the `#define` values in **system.h** appear as follows:

  ```
  #define UART1_NAME "/dev/uart1"
  #define UART1_BASE 0x1000
  #define UART1_SPAN 32
  #define UART1_IRQ 6
  . . .
  ```

  In your C/C++ source files, add a preprocessor macro to verify that your expected peripheral settings remain unchanged in the hardware configuration. For example, the following code checks that the base address of UART1 remains at the expected value:

  ```
  #if (UART1_BASE != 0x1000)
     #error UART should be at 0x1000, but it is not
  #endif
  ```

- **Ensuring coherency with the System ID core**—Use the System ID core. The System ID core is an SOPC Builder peripheral that provides a unique identifier for a generated hardware system. This identifier is stored in a hardware register readable by the Nios II processor. This unique identifier is also stored in the **.sopcinfo** file, which is then used to generate the BSP project for the system. You can use the system ID core to ensure coherency between the hardware and software by either of the following methods:

  - The first method is optionally implemented during system software development, when the Executable and Linking Format (**.elf**) file is downloaded to the Nios II target. During the software download process, the value of the system ID core is checked against the value present in the BSP library file. If the two values do not match, this condition is reported. If you know that the system ID difference is not relevant, the system ID check can be overridden to force a download. Use this override with extreme caution, because a mismatch between hardware and software can lead you to waste time trying to resolve nonexistent bugs.
  - The second method for using the system ID peripheral is useful in systems that do not have a Nios II debug port, or in situations in which running the Nios II software download utilities is not practical. In this method you use the C function `alt_avalon_sysid_test()`. This function reports whether the hardware and software system IDs match.

  
  For more information about the System ID core, refer to the *System ID Core* chapter of the *Embedded Peripherals IP User Guide*.

Developing With the Hardware Abstraction Layer

The HAL for the Nios II processor is a lightweight run-time environment that provides a simple device driver interface for programs to communicate with the underlying hardware. The HAL API is integrated with the ANSI C standard library. The HAL API allows you to access devices and files using familiar C library functions.

This section contains the following subsections:

Overview of the HAL

This section describes how to use HAL services in your Nios II software. It provides information about the HAL configuration options, and the details of system startup and HAL services in HAL-based applications.

HAL Configuration Options

To support the Nios II software development flow, the HAL BSP library is self-configuring to some extent. By design, the HAL attempts to enable as many services as possible, based on the peripherals present in the system hardware. This approach provides your application with the least restrictive environment possible—a useful feature during the product development and board bringup cycle.

The HAL is configured with a group of settings whose values are determined by Tcl commands, which are called during the creation of the BSP project. As mentioned in "Configuring the BSP Project" on page 2–10, Altera recommends you create a separate Tcl file that contains your HAL configuration settings.

HAL configuration settings control the boot loading process, and provide detailed control over the initialization process, system optimization, and the configuration of peripherals and services. For each of these topics, this section provides pointers to the relevant material elsewhere in this chapter.

Configuring the Boot Environment

Your particular system may require a boot loader to configure the application image before it can begin execution. For example, if your application image is stored in flash memory and must be copied to volatile memory for execution, a boot loader must configure the application image in the volatile memory. This configuration process occurs before the HAL BSP library configuration routines execute, and before the `crt0` code block executes. A boot loader implements this process. For more information, refer to "Linking Applications" on page 2–49 and "Application Boot Loading and Programming System Memory" on page 2–51.

Controlling HAL Initialization

As noted in "HAL Initialization" on page 2–27, although most application debugging begins in the `main()` function, some tasks, such as debugging device driver initialization, require the ability to control overall system initialization after the `crt0` initialization routine runs and before `main()` is called.

For an example of this kind of application, refer to the hello_alt_main software example design supplied with the Nios II EDS installation.

Minimizing the Code Footprint and Increasing Performance

For information about increasing your application's performance, or minimizing the code footprint, refer to "Optimizing the Application" on page 2–43.

Configuring Peripherals and Services

For information about configuring and using HAL services, refer to "HAL Peripheral Services" on page 2–28.

System Startup in HAL-Based Applications

System startup in HAL-based applications is a three-stage process. First, the system initializes, then the `crt0` code section runs, and finally the HAL services initialize. The following sections describe these three system-startup stages.

System Initialization

The system initialization sequence begins when the system powers up. The initialization sequence steps for FPGA designs that contain a Nios II processor are the following:

1. **Hardware reset event**—The board receives a power-on reset signal, which resets the FPGA.
2. **FPGA configuration**—The FPGA is programmed with a **.sof** file, from a specialized configuration memory or an external hardware master. The external hardware master can be a CPLD device or an external processor.
3. **System reset**—The SOPC Builder system, composed of one or more Nios II processors and other peripherals, receives a hardware reset signal and enters the components' combined reset state.
4. **Nios II processor(s)**—Each Nios II processor jumps to its preconfigured reset address, and begins running instructions found at this address.
5. **Boot loader or program code**—Depending on your system design, the reset address vector contains a packaged boot loader, called a boot image, or your application image. Use the boot loader if the application image must be copied from non-volatile memory to volatile memory for program execution. This case occurs, for example, if the program is stored in flash memory but runs from SDRAM. If no boot loader is present, the reset vector jumps directly to the `.crt0` section of the application image. Do not use a boot loader if you wish your program to run in-place from non-volatile or preprogrammed memory. For additional information about both of these cases, refer to "Application Boot Loading and Programming System Memory" on page 2–51.

6. **crt0 execution**—After the boot loader executes, the processor jumps to the beginning of the program's initialization block—the `.crt0` code section. The function of the `crt0` code block is detailed in the next section.

crt0 Initialization

The `crt0` code block contains the C run-time initialization code—software instructions needed to enable execution of C or C++ applications. The `crt0` code block can potentially be used by user-defined assembly language procedures as well. The Altera-provided `crt0` block performs the following initialization steps:

1. **Calls alt_load macros**—If the application is designed to run from flash memory (the `.text` section runs from flash memory), the remaining sections are copied to volatile memory. For additional information, refer to "Configuring the Boot Environment" on page 2–24.
2. **Initializes instruction cache**—If the processor has an instruction cache, this cache is initialized. All instruction cache lines are zeroed (without flushing) with the `initi` instruction.

   
   SOPC Builder determines the processors that have instruction caches, and configures these caches at system generation. The Nios II Software Build Tools insert the instruction-cache initialization code block if necessary.

3. **Initializes data cache**—If the processor has a data cache, this cache is initialized. All data cache lines are zeroed (without flushing) with the `initd` instruction. As for the instruction caches, this code is enabled if the processor has a data cache.
4. **Sets the stack pointer**—The stack pointer is initialized. You can set the stack pointer address. For additional information refer to "HAL Linking Behavior" on page 2–50.
5. **Clears the `.bss` section**—The `.bss` section is initialized to all zeroes. You can set the `.bss` section address. For additional information refer to "HAL Linking Behavior" on page 2–50.
6. **Initializes stack overflow protection**—Stack overflow checking is initialized. For additional information, refer to "Software Debugging in Nios II Software Build Tools for Eclipse" on page 2–19.
7. **Jumps to `alt_main()`**—The processor jumps to the `alt_main()` function, which begins initializing the HAL BSP run-time library.

   
   If you use a third-party RTOS or environment for your BSP library file, the `alt_main()` function could be different than the one provided by the Nios II EDS.

If you use a third-party compiler or library, the C run-time initialization behavior may differ from this description.

The `crt0` code includes initialization short-cuts only if you perform hardware simulations of your design. You can control these optimizations by turning `hal.enable_sim_optimize` on or off.

For information about the `hal.enable_sim_optimize` BSP setting, 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*.

The **crt0.S** source file is located in the *<Altera tools installation>***/ip/altera/nios2_ip/altera_nios2/HAL/src** directory.

HAL Initialization

As for any other C program, the first part of the HAL's initialization is implemented by the Nios II processor's `crt0.S` routine. For more information, see "crt0 Initialization" on page 2–26. After `crt0.S` completes the C run-time initialization, it calls the HAL `alt_main()` function, which initializes the HAL BSP run-time library and subsystems.

The HAL `alt_main()` function performs the following steps:

1. **Initializes interrupts**—Sets up interrupt support for the Nios II processor (with the `alt_irq_init()` function).
2. **Starts MicroC/OS-II**—Starts the MicroC/OS-II RTOS, if this RTOS is configured to run (with the `ALT_OS_INIT` and `ALT_SEM_CREATE` functions). For additional information about MicroC/OS-II use and initialization, refer to "Selecting the Operating System (HAL versus MicroC/OS-II RTOS)" on page 2–9.
3. **Initializes device drivers**—Initializes device drivers (with the `alt_sys_init()` function). The Nios II Software Build Tools automatically find all peripherals supported by the HAL, and automatically insert a call to a device configuration function for each peripheral in the `alt_sys_init()` code. To override this behavior, you can disable a device driver with the Nios II BSP Editor, in the **Drivers** tab.

   For information about enabling and disabling 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.*

   To disable a driver from the Nios II Command Shell, use the following option to the **nios2-bsp** script:

   ```
   --cmd set_driver <peripheral_name> none
   ```

   For information about removing a device configuration function, and other methods of reducing the BSP library size, refer to Table 2–1 on page 2–49.
4. **Configures `stdio` functions**—Initializes `stdio` services for `stdin`, `stderr`, and `stdout`. These services enable the application to use the GNU newlib `stdio` functions and maps the file pointers to supported character devices. For more information about configuring the `stdio` services, refer to "Character Mode Devices" on page 2–31.
5. **Initializes C++ CTORS and DTORS**—Handles initialization of C++ constructor and destructor functions. These function calls are necessary if your application is written in the C++ programming language. By default, the HAL configuration mechanism enables support for the C++ programming language. Disabling this feature reduces your application's code footprint, as noted in "Optimizing the Application" on page 2–43.

The Nios II C++ language support depends on the GCC tool chain. The Nios II GCC 4 C++ tool chain supports polymorphism, friendship and inheritance, multiple inheritance, virtual base classes, run-time type information (`typeid`), the `mutable` type qualifier, namespaces, templates, new-and-delete style dynamic memory allocation, operator overloading, and the Standard Template Library (STL). Exceptions and new-style dynamic casts are not supported.

6. **Calls main()**—Calls function `main()`, or application program. Most applications are constructed using a `main()` function declaration, and begin execution at this function.

If you use a BSP that is not based on the HAL and need to initialize it after the `crt0.S` routine runs, define your own `alt_main()` function. For an example, see the `main()` and `alt_main()` functions in the **hello_alt_main.c** file at *<Nios II EDS install dir>***\examples\software\hello_alt_main**.

After you generate your BSP project, the **alt_main.c** source file is located in the **HAL/src** directory.

HAL Peripheral Services

The HAL provides your application with a set of services, typically relying on the presence of a hardware peripheral to support the services. By default, if you configure your HAL BSP project from the command-line by running the **nios2-bsp** script, each peripheral in the system is initialized, operational, and usable as a service at the entry point of your C/C++ application (`main()`).

This section describes the core set of Altera-supplied, HAL-accessible peripherals and the services they provide for your application. It also describes application design guidelines for using the supplied service, and background and configuration information, where appropriate.

For more information about the HAL peripheral services, refer to the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*. For more information about HAL BSP configuration settings, refer to the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Timers

The HAL provides two types of timer services, a system clock timer and a timestamp timer. The system clock timer is used to control, monitor, and schedule system events. The timestamp variant is used to make high performance timing measurements. Each of these timer services is assigned to a single Altera Avalon Timer peripheral.

For more information about this peripheral, refer to the *Interval Timer Core* chapter of the *Embedded Peripherals IP User Guide*.

System Clock Timer

The system clock timer resource is used to trigger periodic events (alarms), and as a timekeeping device that counts system clock ticks. The system clock timer service requires that a timer peripheral be present in the SOPC Builder system. This timer peripheral must be dedicated to the HAL system clock timer service.

Only one system clock timer service may be identified in the BSP library. This timer should be accessed only by HAL supplied routines.

The `hal.sys_clk_timer` setting controls the BSP project configuration for the system clock timer. This setting configures one of the timers available in your SOPC Builder design as the system clock timer.

Altera provides separate APIs for application-level system clock functionality and for generating alarms.

Application-level system clock functionality is provided by two separate classes of APIs, one Nios II specific and the other Unix-like. The Altera function `alt_nticks` returns the number of clock ticks that have elapsed. You can convert this value to seconds by dividing by the value returned by the `alt_ticks_per_second()` function. For most embedded applications, this function is sufficient for rudimentary time keeping.

The POSIX-like `gettimeofday()` function behaves differently in the HAL than on a Unix workstation. On a workstation, with a battery backed-up, real-time clock, this function returns an absolute time value, with the value zero representing 00:00 Coordinated Universal Time (UTC), January 1, 1970, whereas in the HAL, this function returns a time value starting from system power-up. By default, the function assumes system power-up to have occurred on January 1, 1970. Use the `settimeofday()` function to correct the HAL `gettimeofday()` response. The `times()` function exhibits the same behavior difference.

Consider the following common issues and important points before you implement a system clock timer:

- **System Clock Resolution**—The timer's period value specifies the rate at which the HAL BSP project increments the internal variable for the system clock counter. If the system clock increments too slowly for your application, you can decrease the timer's period in SOPC Builder.
- **Rollover**—The internal, global variable that stores the number of system clock counts (since reset) is a 32-bit unsigned integer. No rollover protection is offered for this variable. Therefore, you should calculate when the rollover event will occur in your system, and plan the application accordingly.
- **Performance Impact**—Every clock tick causes the execution of an interrupt service routine. Executing this routine leads to a minor performance penalty. If your system hardware specifies a short timer period, the cumulative interrupt latency may impact your overall system performance.

The alarm API allows you to schedule events based on the system clock timer, in the same way an alarm clock operates. The API consists of the `alt_alarm_start()` function, which registers an alarm, and the `alt_alarm_stop()` function, which disables a registered alarm.

Consider the following common issues and important points before you implement an alarm:

- **Interrupt Service Routine (ISR) context**—A common mistake is to program the alarm callback function to call a service that depends on interrupts being enabled (such as the `printf()` function). This mistake causes the system to deadlock, because the alarm callback function occurs in an interrupt context, while interrupts are disabled.
- **Resetting the alarm**—The callback function can reset the alarm by returning a nonzero value. Internally, the `alt_alarm_start()` function is called by the callback function with this value.
- **Chaining**—The `alt_alarm_start()` function is capable of handling one or more registered events, each with its own callback function and number of system clock ticks to the alarm.
- **Rollover**—The alarm API handles clock rollover conditions for registered alarms seamlessly.

A good timer period for most embedded systems is 50 ms. This value provides enough resolution for most system events, but does not seriously impact performance nor roll over the system clock counter too quickly.

Timestamp Timer

The timestamp timer service provides applications with an accurate way to measure the duration of an event in the system. The timestamp timer service requires that a timer peripheral be present in the SOPC Builder system. This timer peripheral must be dedicated to the HAL timestamp timer service.

Only one timestamp timer service may be identified in the BSP library file. This timer should be accessed only by HAL supplied routines.

The `hal.timestamp_timer` setting controls the BSP configuration for the timer. This setting configures one of the timers available in the SOPC Builder design as the timestamp timer.

Altera provides a timestamp API. The timestamp API is very simple. It includes the `alt_timestamp_start()` function, which makes the timer operational, and the `alt_timestamp()` function, which returns the current timer count.

Consider the following common issues and important points before you implement a timestamp timer:

- **Timer Frequency**—The timestamp timer decrements at the clock rate of the clock that feeds it in the SOPC Builder system. You can modify this frequency in SOPC Builder.
- **Rollover**—The timestamp timer has no rollover event. When the `alt_timestamp()` function returns the value 0, the timer has run down.
- **Maximum Time**—The timer peripheral has 32 bits available to store the timer value. Therefore, the maximum duration a timestamp timer can count is ((1/*timer frequency*) × 2^{32}) seconds.

For more information about the APIs that control the timestamp and system clock timer services, refer to the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

Character Mode Devices

stdin, stdout, and stderr

The HAL can support the `stdio` functions provided in the GNU newlib library. Using the `stdio` library allows you to communicate with your application using functions such as `printf()` and `scanf()`.

Currently, Altera supplies two system components that can support the `stdio` library, the UART and JTAG UART components. These devices can function as standard I/O devices.

To enable this functionality, use the `--default_stdio <device>` option during Nios II BSP configuration. The `stdin` character input file variable and the `stdout` and `stderr` character output file variables can also be individually configured with the HAL BSP settings `hal.stdin`, `hal.stdout`, and `hal.stderr`.

Make sure that you assign values individually for each of the `stdin`, `stdout`, and `stderr` file variables that you use.

After your target system is configured to use the `stdin`, `stdout`, and `stderr` file variables with either the UART or JTAG UART peripheral, you can communicate with the target Nios II system with the Nios II EDS development tools. For more information about performing this task, refer to "Communicating with the Target" on page 2–19.

For more information about the `--default_stdio <device>` option, 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*.

Blocking versus Non-Blocking I/O

Character mode devices can be configured to operate in blocking mode or non-blocking mode. The mode is specified in the device's file descriptor. In blocking mode, a function call to read from the device waits until the device receives new data. In non-blocking mode, the function call to read new data returns immediately and reports whether new data was received. Depending on the function you use to read the file handle, an error code is returned, specifying whether or not new data arrived.

The UART and JTAG UART components are initialized in blocking mode. However, each component can be made non-blocking with the `fnctl` or the `ioctl()` function, as seen in the following `open` system call, which specifies that the device being opened is to function in non-blocking mode:

```
fd = open ("/dev/<your uart name>", O_NONBLOCK | O_RDWR);
```

The `fnctl()` system call shown in Example 2–5 specifies that a device that is already open is to function in non-blocking mode:

Example 2–5. fnctl() System Call

```
/* You can specify <file_descriptor> to be
 * STDIN_FILENO, STDOUT_FILENO, or STDERR_FILENO
 * if you are using STDIO
*/
fnctl(<file_descriptor>, F_SETFL, O_NONBLOCK);
```

The code fragment in Example 2–6 illustrates the use of a nonblocking device:

Example 2–6. Non-Blocking Device Code Fragment

```
input_chars[128];

return_chars = scanf("%128s", &input_chars);
if(return_chars == 0)
{
  if(errno != EWOULDBLOCK)
  {
    /* check other errnos */
  }
}
else
{
  /* process received characters */
}
```

The behavior of the UART and JTAG UART peripherals can also be modified with an `ioctl()` function call. The `ioctl()` function supports the following parameters:

- For UART peripherals:
  - TIOCMGET (reports baud rate of UART)
  - TIOCMSET (sets baud rate of UART)
- For JTAG UART peripherals:
  - TIOCSTIMEOUT (timeout value for connecting to workstation)
  - TIOCGCONNECTED (find out whether host is connected)

The `altera_avalon_uart_driver.enable_ioctl` BSP setting enables and disables the `ioctl()` function for the UART peripherals. The `ioctl()` function is automatically enabled for the JTAG UART peripherals.

The `ioctl()` function is not compatible with the `altera_avalon_uart_driver.enable_small_driver` and `hal.enable_reduced_driver` BSP settings. If either of these settings is enabled, `ioctl()` is not implemented.

Adding Your Own Character Mode Device

If you have a custom device capable of character mode operation, you can create a custom device driver that the `stdio` library functions can use.

For information about how to develop the device driver, refer to *AN459: Guidelines for Developing a Nios II HAL Device Driver*.

Flash Memory Devices

The HAL BSP library supports parallel common flash interface (CFI) memory devices and Altera erasable, programmable, configurable serial (EPCS) flash memory devices. A uniform API is available for both flash memory types, providing read, write, and erase capabilities.

Memory Initialization, Querying, and Device Support

Every flash memory device is queried by the HAL during system initialization to determine the kind of flash memory and the functions that should be used to manage it. This process is automatically performed by the `alt_sys_init()` function, if the device drivers are not explicitly omitted and the small driver configuration is not set.

After initialization, you can query the flash memory for status information with the `alt_flash_get_flash_info()` function. This function returns a pointer to an array of flash region structures—C structures of type `struct flash_region`—and the number of regions on the flash device.

For additional information about the `struct flash_region` structure, refer to the source file **HAL/inc/sys/alt_flash_types.h** in the BSP project directory.

Accessing the Flash Memory

The `alt_flash_open()` function opens a flash memory device and returns a descriptor for that flash memory device. After you complete reading and writing the flash memory, call the `alt_flash_close()` function to close it safely.

The HAL flash memory device model provides you with two flash access APIs, one simple and one fine-grained The simple API takes a buffer of data and writes it to the flash memory device, erasing the sectors if necessary. The fine-grained API enables you to manage your flash device on a block-by-block basis.

Both APIs can be used in the system. The type of data you store determines the most useful API for your application. The following general design guidelines help you determine which API to use for your data storage needs:

Simple API—This API is useful for storing arbitrary streams of bytes, if the exact flash sector location is not important. Examples of this type of data are log or data files generated by the system during run-time, which must be accessed later in a continuous stream somewhere in flash memory.

Fine-Grained API—This API is useful for storing units of data, or data sets, which must be aligned on absolute sector boundaries. Examples of this type of data include persistent user configuration values, FPGA hardware images, and application images, which must be stored and accessed in a given flash sector (or sectors).

For examples that demonstrate the use of APIs, refer to the "Using Flash Devices" section in the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

Configuration and Use Limitations

If you use flash memories in your system, be aware of the following properties of this memory:

- **Code Storage**—If your application runs code directly from the flash memory, the flash manipulation functions are disabled. This setting prevents the processor from erasing the memory that holds the code it is running. In this case, the symbols `ALT_TEXT_DEVICE`, `ALT_RODATA_DEVICE`, and `ALT_EXCEPTIONS_DEVICE` must all have values different from the flash memory peripheral. (Note that each of these `#define` symbols names a memory device, not an address within a memory device).
- **Small Driver**—If the small driver flag is set for the software—the `hal.enable_reduced_device_drivers` setting is enabled—then the flash memory peripherals are not automatically initialized. In this case, your application must call the initialization routines explicitly.
- **Thread safety**—Most of the flash access routines are not thread-safe. If you use any of these routines, construct your application so that only one thread in the system accesses these function.
- **EPCS flash memory limitations**—The Altera EPCS memory has a serial interface. Therefore, it cannot run Nios II instructions and is not visible to the Nios II processor as a standard random-access memory device. Use the Altera-supplied flash memory access routines to read data from this device.
- **File System**—The HAL flash memory API does not support a flash file system in which data can be stored and retrieved using a conventional file handle. However, you can store your data in flash memory before you run your application, using the read-only zip file system and the Nios II flash programmer utility. For information about the read-only zip file system, refer to "Read-Only Zip File System" on page 2–38.

For more information about the configuration and use limitations of flash memory, refer to the "Using Flash Devices" section in the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*. For more information about the API for the flash memory access routines, refer to the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

Direct Memory Access Devices

The HAL Direct Memory Access (DMA) model uses DMA transmit and receive channels. A DMA operation places a transaction request on a channel. A DMA peripheral can have a transmit channel, a receive channel, or both. This section describes three possible hardware configurations for a DMA peripheral, and shows how to activate each kind of DMA channel using the HAL memory access functions.

The DMA peripherals are initialized by the `alt_sys_init()` function call, and are automatically enabled by the **nios2-bsp** script.

DMA Configuration and Use Model

The following examples illustrate use of the DMA transmit and receive channels in a system. The information complements the information available in "Using DMA Devices" in the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

Regardless of the DMA peripheral connections in the system, initialize a transmit channel by running the `alt_dma_txchan_open()` function, and initialize a receive DMA channel by running the `alt_dma_rxchan_open()` function. The following sections describe the use model for some specific cases.

RX-Only DMA Component

A typical RX-only DMA component moves the data it receives from another component to memory. In this case, the receive channel of the DMA peripheral reads continuously from a fixed location in memory, which is the other peripheral's data register. The following sequence of operations directs the DMA peripheral:

1. Open the DMA peripheral—Call the `alt_dma_rxchan_open()` function to open the receive DMA channel.
2. Enable DMA `ioctl` operations—Call the `alt_dma_rxchan_ioctl()` function to set the `ALT_DMA_RX_ONLY_ON` flag. Use the `ALT_DMA_SET_MODE_<n>` flag to set the data width to match that of the other peripheral's data register.
3. Configure the other peripheral to run—The Nios II processor configures the other peripheral to begin loading new data in its data register.
4. Queue the DMA transaction requests—Call the `alt_avalon_dma_prepare()` function to begin a DMA operation. In the function call, you specify the DMA receive channel, the other peripheral's data register address, the number of bytes to transfer, and a callback function to run when the transaction is complete.

TX-Only DMA Component

A typical TX-only DMA component moves data from memory to another component. In this case, the transmit channel of the DMA peripheral writes continuously to a fixed location in memory, which is the other peripheral's data register. The following sequence of operations directs the DMA peripheral:

1. Open the DMA peripheral—Call the `alt_dma_txchan_open()` function to open the transmit DMA channel.
2. Enable DMA `ioctl` operations—Call the `alt_dma_txchan_ioctl()` function to set the `ALT_DMA_TX_ONLY_ON` flag. Use the `ALT_DMA_SET_MODE_<n>` flag to set the data width to match that of the other peripheral's data register.
3. Configure the other peripheral to run—The Nios II processor configures the other peripheral to begin receiving new data in its data register.
4. Queue the DMA transaction requests—Call the `alt_avalon_dma_send()` function to begin a DMA operation. In the function call, you specify the DMA transmit channel, the other peripheral's data register address, the number of bytes to transfer, and a callback function to run when the transaction is complete.

RX and TX DMA Component

A typical RX and TX DMA component performs memory-to-memory copy operations. The application must open, configure, and assign transaction requests to both DMA channels explicitly. The following sequence of operations directs the DMA peripheral:

1. Open the DMA RX channel—Call the `alt_dma_rxchan_open()` function to open the DMA receive channel.
2. Enable DMA RX `ioctl` operations—Call the `alt_dma_rxchan_ioctl()` function to set the `ALT_DMA_RX_ONLY_OFF` flag. Use the `ALT_DMA_SET_MODE_<n>` flag to set the data width to the correct value for the memory transfers.
3. Open the DMA TX channel—Call the `alt_dma_txchan_open()` function to open the DMA transmit channel.
4. Enable DMA TX `ioctl` operations—Call the `alt_dma_txchan_ioctl()` function to set the `ALT_DMA_TX_ONLY_OFF` flag. Use the `ALT_DMA_SET_MODE_<n>` flag to set the data width to the correct value for the memory transfers.
5. Queue the DMA RX transaction requests—Call the `alt_avalon_dma_prepare()` function to begin a DMA RX operation. In the function call, you specify the DMA receive channel, the address from which to begin reading, the number of bytes to transfer, and a callback function to run when the transaction is complete.
6. Queue the DMA TX transaction requests—Call the `alt_avalon_dma_send()` function to begin a DMA TX operation. In the function call, you specify the DMA transmit channel, the address to which to begin writing, the number of bytes to transfer, and a callback function to run when the transaction is complete.

The DMA peripheral does not begin the transaction until the DMA TX transaction request is issued.

For examples of DMA device use, refer to "Using DMA Devices" in the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

DMA Data-Width Parameter

The DMA data-width parameter is configured in SOPC Builder to specify the widths that are supported. In writing the software application, you must specify the width to use for a particular transaction. The width of the data you transfer must match the hardware capability of the component.

Consider the following points about the data-width parameter before you implement a DMA peripheral:

- **Peripheral width**—When a DMA component moves data from another peripheral, the DMA component must use a single-operation transfer size equal to the width of the peripheral's data register.
- **Transfer length**—The byte transfer length specified to the DMA peripheral must be a multiple of the data width specified.

- **Odd transfer sizes**—If you must transfer an uneven number of bytes between memory and a peripheral using a DMA component, you must divide up your data transfer operation. Implement the longest allowed transfer using the DMA component, and transfer the remaining bytes using the Nios II processor. For example, if you must transfer 1023 bytes of data from memory to a peripheral with a 32-bit data register, perform 255 32-bit transfers with the DMA and then have the Nios II processor write the remaining 3 bytes.

Configuration and Use Limitations

If you use DMA components in your system, be aware of the following properties of these components:

- **Hardware configuration**—The following aspects of the hardware configuration of the DMA peripheral determine the HAL service:
  - DMA components connected to peripherals other than memory support only half of the HAL API (receive or transmit functionality). The application software should not attempt to call API functions that are not available.
  - The hardware parameterization of the DMA component determines the data width of its transfers, a value which the application software must take into account.
- **IOCTL control**—The DMA `ioctl()` function call enables the setting of a single flag only. To set multiple flags for a DMA channel, you must call `ioctl()` multiple times.
- **DMA transaction slots**—The current driver is limited to four transaction slots. If you must increase the number of transaction slots, you can specify the number of slots using the macro `ALT_AVALON_DMA_NSLOTS`. The value of this macro must be a power of two.
- **Interrupts**—The HAL DMA service requires that the DMA peripheral's interrupt line be connected in the system.
- **User controlled DMA accesses**—If the default HAL DMA access routines are too unwieldy for your application, you can create your own access functions. For information about how to remove the default HAL DMA driver routines, refer to "Reducing Code Size" on page 2–48.

For more information about the HAL API for accessing DMA devices, refer to "Using DMA Devices" in the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook* and to the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

Files and File Systems

The HAL provides two simple file systems and an API for dealing with file data. The HAL uses the GNU newlib library's file access routines, found in **file.h**, to provide access to files. In addition, the HAL provides the following file systems:

- Host-based file system—Enables a Nios II system to access the host workstation's file system
- Read-only zip file system—Enables simple access to preconfigured data in the Nios II system memory

Several more conventional file systems that support both read and write operations are available through third-party vendors. For up-to-date information about the file system solutions available for the Nios II processor, visit the Embedded Processing page of the Altera website, and click **Altera Embedded Alliance Partners**.

To make either of these software packages visible to your application, you must enable it in the BSP. You can enable a software package either in the BSP Editor, or from the command line. The names that specify the host-based file system and read-only zip file system packages are `altera_hostfs` and `altera_ro_zipfs`, respectively.

The Host-Based File System

The host-based file system enables the Nios II system to manipulate files on a workstation through a JTAG connection. The API is a transparent way to access data files. The system does not require a physical block device.

Consider the following points about the host-based file system before you use it:

- **Communication speed**—Reading and writing large files to the Nios II system using this file system is slow.
- **Debug use mode**—The host-based file system is only available during debug sessions from the Nios II debug perspective. Therefore, you should use the host-based file system only during system debugging and prototyping operations.
- **Incompatibility with direct drivers**—The host-based file system only works if the HAL BSP library is configured with direct driver mode disabled. However, enabling this mode reduces the size of the application image. For more information, refer to "Optimizing the Application" on page 2–43.

For more information about the host file system, refer to "Using File Subsystems" in the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

Read-Only Zip File System

The read-only zip file system is a lightweight file system for the Nios II processor, targeting flash memory.

Consider the following points about the read-only zip file system before you use it:

- **Read-Only**—The read-only zip file system does not implement writes to the file system.
- **Configuring the file system**—To create the read-only zip file system you must create a binary file on your workstation and use the Nios II flash programmer utility to program it in the Nios II system.
- **Incompatibility with direct drivers**—The read-only zip file system only works if the HAL BSP library is configured with direct driver mode disabled. However, enabling this mode reduces the size of the application image. For more information, refer to "Optimizing the Application" on page 2–43.

For more information, refer to the *Read-Only Zip File System* and *Developing Programs Using the Hardware Abstraction Layer* chapters of the *Nios II Software Developer's Handbook*, and the read-only zip file system Nios II software example design listed in "Nios II Design Example Scripts" in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Ethernet Devices

Ethernet devices are a special case for the HAL service model. To make them accessible to the application, these devices require an additional software library, a TCP/IP stack. Altera supplies a TCP/IP networking stack called NicheStack, which provides your application with a socket-based interface for communicating over Ethernet networks.

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*.

To enable your application to use networking, you enable the NicheStack software package in the BSP library. The Tcl command argument that specifies the NicheStack software package is `altera_iniche`.

Unsupported Devices

The HAL provides a wide variety of native device support for Altera-supplied peripherals. However, your system may require a device or peripheral that Altera does not provide. In this case, one or both of the following two options may be available to you:

- Obtain a device through Altera's third-party program
- Incorporate your own device

Altera's third-party program information is available on the Nios II embedded software partners page. Refer to the Embedded Processing page of the Altera website, and click **Altera Embedded Alliance Partners**.

Incorporating your own custom peripheral is a two-stage process. First you must incorporate the peripheral in the hardware, and then you must develop a device driver.

For more information about how to incorporate a new peripheral in the hardware, refer to the *Nios II Hardware Development Tutorial*. For more information about how to develop a device driver, refer to the *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook* and to *AN459: Guidelines for Developing a Nios II HAL Device Driver*.

Accessing Memory With the Nios II Processor

It can be difficult to create software applications that program the Nios II processor to interact correctly with data and instruction caches when it reads and writes to peripherals and memories. There are also subtle differences in how the different Nios II processor cores handle these operations, that can cause problems when you migrate from one Nios II processor core to another.

This section helps you avoid the most common pitfalls. It provides background critical to understanding how the Nios II processor reads and writes peripherals and memories, and describes the set of software utilities available to you, as well as providing sets of instructions to help you avoid some of the more common problems in programming these read and write operations.

Creating General C/C++ Applications

You can write most C/C++ applications without worrying about whether the processor's read and write operations bypass the data cache. However, you do need to make sure the operations do not bypass the data cache in the following cases:

- Your application must guarantee that a read or write transaction actually reaches a peripheral or memory. This guarantee is critical for the correct functioning of a device driver interrupt service routine, for example.
- Your application shares a block of memory with another processor or Avalon interface master peripheral.

Accessing Peripherals

If your application accesses peripheral registers, or performs only a small set of memory accesses, Altera recommends that you use the default HAL I/O macros, IORD and IOWR. These macros guarantee that the accesses bypass the data cache.

Two types of cache-bypass macros are available. The HAL access routines whose names end in _32DIRECT, _16 DIRECT, and _8 DIRECT interpret the offset as a byte address. The other routines treat this offset as a count to be multiplied by four bytes, the number of bytes in the 32-bit connection between the Nios II processor and the system interconnect fabric. The _32DIRECT, _16DIRECT, and _8DIRECT routines are designed to access memory regions, and the other routines are designed to access peripheral registers.

Example 2–7 shows how to write a series of half-word values into memory. Because the target addresses are not all on a 32-bit boundary, this code sample uses the IOWR_16DIRECT macro.

Example 2–7. Writing Half-Word Locations

```
/* Loop across 100 memory locations, writing 0xdead to */
/* every half word location... */
for(i=0, j=0;i<100;i++, j+=2)
{
  IOWR_16DIRECT(MEM_START, j, (unsigned short)0xdead);
}
```

Example 2–8 shows how to access a peripheral register. In this case, the write is to a 32-bit boundary address, and the code sample uses the IOWR macro.

Example 2–8. Peripheral Register Access

```
 unsigned int control_reg_val = 0;
/* Read current control register value */
control_reg_val = IORD(BAR_BASE_ADDR, CONTROL_REG);

/* Enable "start" bit */
control_reg_val |= 0x01;

/* Write "start" bit to control register to start peripheral */
IOWR(BAR_BASE_ADDR, CONTROL_REG, control_reg_val);
```

Altera recommends that you use the HAL-supplied macros for accessing external peripherals and memory.

Sharing Uncached Memory

If your application must allocate some memory, operate on that memory, and then share the memory region with another peripheral (or processor), use the HAL-supplied `alt_uncached_malloc()` and `alt_uncached_free()` functions. Both of these functions operate on pointers to bypass cached memory.

To share uncached memory between a Nios II processor and a peripheral, perform the following steps:

1. **malloc memory**—Run the `alt_uncached_malloc()` function to claim a block of memory from the heap. If this operation is successful, the function returns a pointer that bypasses the data cache.
2. **Operate on memory**—Have the Nios II processor read or write the memory using the pointer. Your application can perform normal pointer-arithmetic operations on this pointer.
3. **Convert pointer**—Run the `alt_remap_cached()` function to convert the pointer to a memory address that is understood by external peripherals.
4. **Pass pointer**—Pass the converted pointer to the external peripheral to enable it to perform operations on the memory region.

Sharing Memory With Cache Performance Benefits

Another way to share memory between a data-cache enabled Nios II processor and other external peripherals safely without sacrificing processor performance is the delayed data-cache flush method. In this method, the Nios II processor performs operations on memory using standard C or C++ operations until it needs to share this memory with an external peripheral.

Your application can share non-cache-bypassed memory regions with external masters if it runs the `alt_dcache_flush()` function before it allows the external master to operate on the memory.

To implement delayed data-cache flushing, the application image programs the Nios II processor to follow these steps:

1. **Processor operates on memory**—The Nios II processor performs reads and writes to a memory region. These reads and writes are C/C++ pointer or array based accesses or accesses to data structures, variables, or a malloc'ed region of memory.
2. **Processor flushes cache**—After the Nios II processor completes the read and write operations, it calls the `alt_dcache_flush()` instruction with the location and length of the memory region to be flushed. The processor can then signal to the other memory master peripheral to operate on this memory.
3. **Processor operates on memory again**—When the other peripheral has completed its operation, the Nios II processor can operate on the memory once again. Because the data cache was previously flushed, any additional reads or writes update the cache correctly.

Example 2–9 shows an implementation of delayed data-cache flushing for memory accesses to a C array of structures. In the example, the Nios II processor initializes one field of each structure in an array, flushes the data cache, signals to another master that it may use the array, waits for the other master to complete operations on the array, and then sums the values the other master is expected to set.

Example 2–9. Data-Cache Flushing With Arrays of Structures

```
struct input foo[100];

for(i=0;i<100;i++)
  foo[i].input = i;
alt_dcache_flush(&foo, sizeof(struct input)*100);
signal_master(&foo);
for(i=0;i<100;i++)
  sum += foo[i].output;
```

Example 2–10 shows an implementation of delayed data-cache flushing for memory accesses to a memory region the Nios II processor acquired with `malloc()`.

Example 2–10. Data-Cache Flushing With Memory Acquired Using malloc()

```
char * data = (char*)malloc(sizeof(char) * 1000);

write_operands(data);
alt_dcache_flush(data, sizeof(char) * 1000);
signal_master(data);
result = read_results(data);
free(data);
```

The `alt_dcache_flush_all()` function call flushes the entire data cache, but this function is not efficient. Altera recommends that you flush from the cache only the entries for the memory region that you make available to the other master peripheral.

Handling Exceptions

The HAL infrastructure provides a robust interrupt handling service routine and an API for exception handling. The Nios II processor can handle exceptions caused by hardware interrupts, unimplemented instructions, and software traps.

This section discusses exception handling with the Nios II internal interrupt controller. The Nios II processor also supports an external interrupt controller (EIC), which you can use to prioritize interrupts and make other performance improvements.

For information about the EIC, refer to the *Programming Model* chapter of the *Nios II Processor Reference Handbook*. For information about the exception handler software routines, HAL-provided services, API, and software support for the EIC, refer to the *Exception Handling* chapter of the *Nios II Software Developer's Handbook*.

Consider the following common issues and important points before you use the HAL-provided exception handler:

- **Prioritization of interrupts**—The Nios II processor does not prioritize its 32 interrupt vectors, but the HAL exception handler assigns higher priority to lower numbered interrupts. You must modify the interrupt request (IRQ) prioritization of your peripherals in SOPC Builder.
- **Nesting of interrupts**—The HAL infrastructure allows interrupts to be nested—higher priority interrupts can preempt processor control from an exception handler that is servicing a lower priority interrupt. However, Altera recommends that you not nest your interrupts because of the associated performance penalty.
- **Exception handler environment**—When creating your exception handler, you must ensure that the handler does not run interrupt-dependent functions and services, because this can cause deadlock. For example, an exception handler should not call the IRQ-driven version of the `printf()` function.

Modifying the Exception Handler

In some very special cases, you may wish to modify the existing HAL exception handler routine or to insert your own interrupt handler for the Nios II processor. However, in most cases you need not modify the interrupt handler routines for the Nios II processor for your software application.

Consider the following common issues and important points before you modify or replace the HAL-provided exception handler:

- **Interrupt vector address**—The interrupt vector address for each Nios II processor is set during compilation of the FPGA design. You can modify it during hardware configuration in SOPC Builder.
- **Modifying the exception handler**—The HAL-provided exception handler is fairly robust, reliable, and efficient. Modifying the exception handler could break the HAL-supplied interrupt handling API, and cause problems in the device drivers for other peripherals that use interrupts, such as the UART and the JTAG UART.

You may wish to modify the behavior of the exception handler to increase overall performance. For guidelines for increasing the exception handler's performance, refer to "Accelerating Interrupt Service Routines" on page 2–47.

Optimizing the Application

This section examines techniques to increase your software application's performance and decrease its size.

This section contains the following subsections:

Performance Tuning Background

Software performance is the speed with which a certain task or series of tasks can be performed in the system. To increase software performance, you must first determine the sections of the code in which the processing time is spent.

An application's tasks can be divided into interrupt tasks and system processing tasks. Interrupt task performance is the speed with which the processor completes an interrupt service routine to handle an external event or condition. System processing task performance is the speed with which the system performs a task explicitly described in the application code.

A complete analysis of application performance examines the performance of the system processing tasks and the interrupt tasks, as well as the footprint of the software image.

Speeding Up System Processing Tasks

To increase your application's performance, determine how you can speed up the system processing tasks it performs. First analyze the current performance and identify the slowest tasks in your system, then determine whether you can accelerate any part of your application by increasing processor efficiency, creating a hardware accelerator, or improving the applications's methods for data movement.

Analyzing the Problem

The first step to accelerate your system processing is to identify the slowest task in your system. Altera provides the following tools to profile your application:

- **GNU Profiler**—The Nios II EDS toolchain includes a method for profiling your application with the GNU Profiler. This method of profiling reports how long various functions run in your application.
- **High resolution timer**—The interval timer peripheral is a simple time counter that can determine the amount of time a given subroutine runs.
- **Performance counter peripheral**—The performance counter unit can profile several different sections of code with a collection of counters. This peripheral includes a simple software API that enables you to print out the results of these counters through the Nios II processor's `stdio` services.

Use one or more of these tools to determine the tasks in which your application is spending most of its processing time.

For more information about how to profile your software application, refer to *AN391: Profiling Nios II Systems*.

Accelerating your Application

This section describes several techniques to accelerate your application. Because of the flexible nature of the FPGA, most of these techniques modify the system hardware to improve the processor's execution performance. This section describes the following performance enhancement methods:

- Methods to increase processor efficiency
- Methods to accelerate select software algorithms using hardware accelerators
- Using a DMA peripheral to increase the efficiency of sequential data movement operations

Increasing Processor Efficiency

An easy way to increase the software application's performance is to increase the rate at which the Nios II processor fetches and processes instructions, while decreasing the number of instructions the application requires. The following techniques can increase processor efficiency in running your application:

- **Processor clock frequency**—Modify the processor clock frequency using SOPC Builder. The faster the execution speed of the processor, the more quickly it is able to process instructions.
- **Nios II processor improvements**—Select the most efficient version of the Nios II processor and parameterize it properly. The following processor settings can be modified using SOPC Builder:
  - **Processor type**—Select the fastest Nios II processor core possible. In order of performance, from fastest to slowest, the processors are the Nios II/f, Nios II/s, and Nios II/e cores.
  - **Instruction and data cache**—Include an instruction or data cache, especially if the memory you select for code execution—where the application image and the data are stored—has high access time or latency.
  - **Multipliers**—Use hardware multipliers to increase the efficiency of relevant mathematical operations.

  For more information about the processor configuration options, refer to the *Instantiating the Nios II Processor* chapter of the *Nios II Processor Reference Handbook*.

- **Nios II instruction and data memory speed**—Select memory with low access time and latency for the main program execution. The memory you select for main program execution impacts overall performance, especially if the Nios II caches are not enabled. The Nios II processor stalls while it fetches program instructions and data.
- **Tightly coupled memories**—Select a tightly coupled memory for the main program execution. A tightly coupled memory is a fast general purpose memory that is connected directly to the Nios II processor's instruction or data paths, or both, and bypasses any caches. Access to tightly coupled memory has the same speed as access to cache memory. A tightly coupled memory must guarantee a single-cycle access time. Therefore, it is usually implemented in an FPGA memory block.

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

- **Compiler Settings**—More efficient code execution can be attained with the use of compiler optimizations. Increase the compiler optimization setting to `-O3`, the fastest compiler optimization setting, to attain more efficient code execution. You set the C-compiler optimization settings for the BSP project independently of the optimization settings for the application.

  For information about configuring the compiler optimization level for the BSP project, refer to the `hal.make.bsp_cflags_optimization` BSP setting in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Accelerating Hardware

Slow software algorithms can be accelerated with the use of custom instructions, dedicated hardware accelerators, and use of the C-to-Hardware (C2H) compiler tool. The following techniques can increase processor efficiency in running your application:

- **Custom instructions**—Use custom instructions to augment the Nios II processor's arithmetic and logic unit (ALU) with a block of dedicated, user-defined hardware to accelerate a task-specific, computational operation. This hardware accelerator is associated with a user-defined operation code, which the application software can call.

  For information about how to create a custom instruction, refer to the *Using Nios II Floating-Point Custom Instructions Tutorial*.

- **Hardware accelerators**—Use hardware accelerators for bulk processing operations that can be performed independently of the Nios II processor. Hardware accelerators are custom, user-defined peripherals designed to speed up the processing of a specific system task. They increase the efficiency of operations that are performed independently of the Nios II processor.

  For more information about hardware acceleration, refer to the *Hardware Acceleration and Coprocessing* chapter of the *Embedded Design Handbook*.

- **C2H Compiler**—Use the C2H Compiler to accelerate standard ANSI C functions by converting them to dedicated hardware blocks.

  For more information about the C2H Compiler, refer to the *Nios II C2H Compiler User Guide* and to the *Optimizing Nios II C2H Compiler Results* chapter of the *Embedded Design Handbook*.

Improving Data Movement

If your application performs many sequential data movement operations, a DMA peripheral might increase the efficiency of these operations. Altera provides the following two DMA peripherals for your use:

- **DMA**—Simple DMA peripheral that can perform single operations before being serviced by the processor. For more information about using the DMA peripheral, refer to "HAL Peripheral Services" on page 2–28.

  For information about the DMA peripheral, refer to the *DMA Controller Core* chapter in the *Embedded Peripherals IP User Guide*.

- **Scatter-Gather DMA (SGDMA)**—Descriptor-based DMA peripheral that can perform multiple operations before being serviced by processor.

  For more information, refer to the *Scatter-Gather DMA Controller Core* chapter in the *Embedded Peripherals IP User Guide*.

Accelerating Interrupt Service Routines

To increase the efficiency of your interrupt service routines, determine how you can speed up the tasks they perform. First analyze the current performance and identify the slowest parts of your interrupt dispatch and handler time, then determine whether you can accelerate any part of your interrupt handling.

Analyzing the Problem

The total amount of time consumed by an interrupt service routine is equal to the latency of the HAL interrupt dispatcher plus the interrupt handler running time. Use the following methods to profile your interrupt handling:

- **Interrupt dispatch time**—Calculate the interrupt handler entry time using the method found in design files that accompany the *Using Nios II Tightly Coupled Memory Tutorial* on the Altera literature pages.

  You can download the design files from the Literature: Tutorials page of the Altera website.

- **Interrupt service routine time**—Use a timer to measure the time from the entry to the exit point of the service routine.

Accelerating the Interrupt Service Routine

The following techniques can increase interrupt handling efficiency when running your application:

- **General software performance enhancements**—Apply the general techniques for improving your application's performance to the ISR and ISR handler. Place the `.exception` code section in a faster memory region, such as tightly coupled memory.

- **IRQ priority**—Assign an appropriate priority to the hardware interrupt. The method for assigning interrupt priority depends on the type of interrupt controller.
  - With the internal interrupt controller, set the interrupt priority of your hardware device to the lowest number available. The HAL ISR service routine uses a priority based system in which the lowest number interrupt has the highest priority.
  - With an external interrupt controller (EIC), the method for priority configuration depends on the hardware. Refer to the EIC documentation for details.
- **Custom instruction and tightly coupled memories**—Decrease the amount of time spent by the interrupt handler by using the interrupt-vector custom instruction and tightly coupled memory regions.

For more information about how to improve the performance of the Nios II exception handler, refer to the *Exception Handling* chapter of the *Nios II Software Developer's Handbook*.

Reducing Code Size

Reducing the memory space required by your application image also enhances performance. This section describes how to measure and decrease your code footprint.

Analyzing the Problem

The easiest way to analyze your application's code footprint is to use the GNU Binary Utilities tool **nios2-elf-size**. This tool analyzes your compiled **.elf** binary file and reports the total size of your application, as well as the subtotals for the `.text`, `.data`, and `.bss` code sections. Example 2–11 shows a **nios2-elf-size** command response.

Example 2–11. Example Use of nios2-elf-size Command

```
> nios2-elf-size -d application.elf
text data bss dec hex filename
203412 8288 4936 216636 34e3c application.elf
```

Reducing the Code Footprint

The following methods help you to reduce your code footprint:

- **Compiler options**—Setting the `-Os` flag for the GCC causes the compiler to apply size optimizations for code size reduction. Use the `hal.make.bsp_cflags_optimization` BSP setting to set this flag.

- **Reducing the HAL footprint**—Use the HAL BSP library configuration settings to reduce the size of the HAL component of your BSP library file. However, enabling the size-reduction settings for the HAL BSP library often impacts the flexibility and performance of the system.

  Table 2–1 lists the configuration settings for size optimization. Use as many of these settings as possible with your system to reduce the size of BSP library file.

Table 2–1. BSP Settings to Reduce Library Size

BSP Setting Name	Value
hal.max_file_descriptors	`4`
hal.enable_small_c_library	`true`
hal.sys_clk_timer	`none`
hal.timestamp_timer	`none`
hal.enable_exit	`false`
hal.enable_c_plus_plus	`false`
hal.enable_lightweight_device_driver_api	`true`
hal.enable_clean_exit	`false`
hal.enable_sim_optimize	`false`
hal.enable_reduced_device_drivers	`true`
hal.make.bsp_cflags_optimization	`\"-Os\"`

  You can reduce the HAL footprint by adjusting BSP settings as shown in Table 2–1.

- **Removing unused HAL device drivers**—Configure the HAL with support only for system peripherals your application uses.
  - By default, the HAL configuration mechanism includes device driver support for all system peripherals present. If you do not plan on accessing all of these peripherals using the HAL device drivers, you can elect to have them omitted during configuration of the HAL BSP library by using the `set_driver` command when you configure the BSP project.
  - The HAL can be configured to include various software modules, such as the NicheStack networking stack and the read-only zip file system, whose presence increases the overall footprint of the application. However, the HAL does not enable these modules by default.

Linking Applications

This section discusses how the Nios II software development tools create a default linker script, what this script does, and how to override its default behavior. The section also includes instructions to control some common linker behavior, and descriptions of the circumstances in which you may need them.

This section contains the following subsections:

- "Background"
- "Linker Sections and Application Configuration"
- "HAL Linking Behavior"

Background

When you generate your project, the Nios II Software Build Tools generate two linker-related files, **linker.x** and **linker.h**. **linker.x** is the linker command file that the generated application's makefile uses to create the **.elf** binary file. All linker setting modifications you make to the HAL BSP project affect the contents of these two files.

Linker Sections and Application Configuration

Every Nios II application contains `.text`, `.rodata`, `.rwdata`, `.bss`, `.heap`, and `.stack` sections. Additional sections can be added to the **.elf** file to hold custom code and data.

These sections are placed in named memory regions, defined to correspond with physical memory devices and addresses. By default, these sections are automatically generated by the HAL. However, you can control them for a particular application.

HAL Linking Behavior

This section describes the default linking behavior of the BSP generation tools and how to control the linking explicitly.

Default BSP Linking

During BSP configuration, the tools perform the following steps automatically:

1. **Assign memory region names**—Assign a name to each system memory device, and add each name to the linker file as a memory region.
2. **Find largest memory**—Identify the largest read-and-write memory region in the linker file.
3. **Assign sections**—Place the default sections (`.text`, `.rodata`, `.rwdata`, `.bss`, `.heap`, and `.stack`) in the memory region identified in the previous step.
4. **Write files**—Write the **linker.x** and **linker.h** files.

Usually, this section allocation scheme works during the software development process, because the application is guaranteed to function if the memory is large enough.

The rules for the HAL default linking behavior are contained in the Altera-generated Tcl scripts **bsp-set-defaults.tcl** and **bsp-linker-utils.tcl** found in the *<Nios II EDS install dir>*/**sdk2/bin** directory. These scripts are called by the **nios2-bsp-create-settings** configuration application. Do not modify these scripts directly.

User-Controlled BSP Linking

You can manage the default linking behavior in the **Linker Script** tab of the Nios II BSP Editor. You can manipulate the linker script in the following ways:

- Add a memory region—Maps a memory region name to a physical memory device.
- Add a section mapping—Maps a section name to a memory region. The Nios II BSP Editor allows you to view the memory map before and after making changes.

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

Application Boot Loading and Programming System Memory

Most Nios II systems require some method to configure the hardware and software images in system memory before the processor can begin executing your application program. This section describes various possible memory topologies for your system (both volatile and non-volatile), their use, their requirements, and their configuration. The Nios II software application requires a boot loader application to configure the system memory if the system software is stored in flash memory, but is configured to run from volatile memory. If the Nios II processor is running from flash memory—the `.text` section is in flash memory—a copy routine, rather than a boot loader, loads the other program sections to volatile memory. In some cases, such as when your system application occupies internal FPGA memory, or is preloaded into external memory by another processor, no configuration of the system memory is required.

This section contains the following subsections:

- "Default BSP Boot Loading Configuration"
- "Boot Configuration Options" on page 2–51
- "Generating and Programming System Memory Images" on page 2–55

Default BSP Boot Loading Configuration

The **nios2-bsp** script determines whether the system requires a boot loader and whether to enable the copying of the default sections.

By default, the **nios2-bsp** script makes these decisions using the following rules:

- **Boot loader**—The **nios2-bsp** script assumes that a boot loader is being used if the following conditions are met:
  - The Nios II processor's reset address is not in the `.text` section.
  - The Nios II processor's reset address is in flash memory.
- **Copying default sections**—The **nios2-bsp** script enables the copying of the default volatile sections if the Nios II processor's reset address is set to an address in the `.text` section.

If the default boot loader behavior is appropriate for your system, you do not need to intervene in the boot loading process.

Boot Configuration Options

You can modify the default **nios2-bsp** script behavior for application loading by using the following settings:

- **hal.linker.allow_code_at_reset**
- **hal.linker.enable_alt_load**
- **hal.linker.enable_alt_load_copy_rwdata**

- **hal.linker.enable_alt_load_copy_exceptions**
- **hal.linker.enable_alt_load_copy_rodata**

If you enable these settings, you can override the BSP's default behavior for boot loading. You can modify the application loading behavior in the **Settings** tab of the Nios II BSP Editor.

Alternatively, you can list the settings in a Tcl script that you import to the BSP Editor.

For information about using an imported Tcl script, refer to *"Using Tcl Scripts with the Nios II BSP Editor" on page 2–15*.

These settings are created in the **settings.bsp** configuration file whether or not you override the default BSP generation behavior. However, you may override their default values.

For more information about BSP configuration settings, refer to the "Settings Managed by the Software Build Tools" section in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*. For more information about boot loading options and for advanced boot loader examples, refer to *AN458: Alternative Nios II Boot Methods*.

Booting and Running From Flash Memory

If your program is loaded in and runs from flash memory, the application's `.text` section is not copied. However, during C run-time initialization—execution of the `crt0` code block—some of the other code sections may be copied to volatile memory in preparation for running the application.

For more information about the behavior of the `crt0` code, refer to "crt0 Initialization" on page 2–26.

Altera recommends that you avoid this configuration during the normal development cycle because downloading the compiled application requires reprogramming the flash memory. In addition, software breakpoint capabilities require that hardware breakpoints be enabled for the Nios II processor when using this configuration.

Prepare for BSP configuration by following these steps to configure your application to boot and run from flash memory:

1. **Nios II processor reset address**—Ensure that the Nios II processor's reset address is in flash memory. Configure the reset address and flash memory addresses in SOPC Builder.
2. **Text section linker setting**—Ensure that the `.text` section maps to the flash memory address region. You can examine and modify section mappings in the **Linker Script** tab in the BSP Editor. Alternatively, use the following Tcl command:

   ```
   add_section_mapping .text ext_flash
   ```

3. **Other sections linker setting**—Ensure that all of the other sections, with the possible exception of the `.rodata` section, are mapped to volatile memory regions. The `.rodata` section can map to a flash-memory region.

4. **HAL C run-time configuration settings**—Configure the BSP settings as shown in Table 2–2.

Table 2–2. BSP Settings to Boot and Run from Flash Memory

BSP Setting Name	Value
hal.linker.allow_code_at_reset	1
hal.linker.enable_alt_load	1
hal.linker.enable_alt_load_copy_rwdata	1
hal.linker.enable_alt_load_copy_exceptions	1
hal.linker.enable_alt_load_copy_rodata	1

If your application contains custom memory sections, you must manually load the custom sections. Use the `alt_load_section()` HAL library function to ensure that these sections are loaded before your program runs.

The HAL BSP library disables the flash memory write capability to prevent accidental overwrite of the application image.

Booting From Flash Memory and Running From Volatile Memory

If your application image is stored in flash memory, but executes from volatile memory with assistance from a boot loader program, prepare for BSP configuration by following these steps:

1. **Nios II processor reset address**—Ensure that the Nios II processor's reset address is an address in flash memory. Configure this option using SOPC Builder.
2. **Text section linker setting**—Ensure that the `.text` section maps to a volatile region of system memory, and not to the flash memory.
3. **Other sections linker setting**—Ensure that all of the other sections, with the possible exception of the `.rodata` section, are mapped to volatile memory regions. The `.rodata` section can map to a flash-memory region.
4. **HAL C run-time configuration settings**—Configure the BSP settings as shown in Table 2–3.

Table 2–3. BSP Settings to Boot from Flash Memory and Run from Volatile Memory

BSP Setting Name	Value
hal.linker.allow_code_at_reset	0
hal.linker.enable_alt_load	0
hal.linker.enable_alt_load_copy_rwdata	0
hal.linker.enable_alt_load_copy_exceptions	0
hal.linker.enable_alt_load_copy_rodata	0

Booting and Running From Volatile Memory

This configuration is use in cases where the Nios II processor's memory is loaded externally by another processor or interconnect switch fabric master port. In this case, prepare for BSP configuration by performing the same steps as in "Booting From Flash Memory and Running From Volatile Memory", except that the Nios II processor reset address should be changed to the memory that holds the code that the processor executes initially. Prepare for BSP configuration by following these steps:

1. **Nios II processor reset address**—Ensure that the Nios II processor's reset address is in volatile memory. Configure this option using SOPC Builder.
2. **Text section linker setting**—Ensure that the `.text` section maps to the reset address memory.
3. **Other sections linker setting**—Ensure that all of the other sections, including the `.rodata` section, also map to the reset address memory.
4. **HAL C run-time configuration settings**—Configure the BSP settings as shown in Table 2–4.

Table 2–4. BSP Settings to Boot and Run from Volatile Memory

BSP Setting Name	Value
hal.linker.allow_code_at_reset	1
hal.linker.enable_alt_load	0
hal.linker.enable_alt_load_copy_rwdata	0
hal.linker.enable_alt_load_copy_exceptions	0
hal.linker.enable_alt_load_copy_rodata	0

This type of boot loading and sequencing requires additional supporting hardware modifications, which are beyond the scope of this chapter.

Booting From Altera EPCS Memory and Running From Volatile Memory

This configuration is a special case of the configuration described in "Booting From Flash Memory and Running From Volatile Memory" on page 2–53. However, in this configuration, the processor does not perform the initial boot loading operation. The EPCS flash memory stores the FPGA hardware image and the application image. During system power up, the FPGA configures itself from EPCS memory. Then the Nios II processor resets control to a small FPGA memory resource in the EPCS memory controller, and executes a small boot loader application that copies the application from EPCS memory to the application's run-time location.

To make this configuration work, you must instantiate the EPCS device controller core in your system hardware. Add the component using SOPC Builder.

Prepare for BSP configuration by following these steps:

1. **Nios II processor reset address**—Ensure that the Nios II processor's reset address is in the EPCS memory controller. Configure this option using SOPC Builder.
2. **Text section linker setting**—Ensure that the `.text` section maps to a volatile region of system memory.

3. **Other sections linker setting**—Ensure that all of the other sections, including the `.rodata` section, map to volatile memory.
4. **HAL C run-time configuration settings**—Configure the BSP settings as shown in Table 2–5.

Table 2–5. BSP Settings to Boot from EPCS and Run from Volatile Memory

BSP Setting Name	Value
hal.linker.allow_code_at_reset	0
hal.linker.enable_alt_load	0
hal.linker.enable_alt_load_copy_rwdata	0
hal.linker.enable_alt_load_copy_exceptions	0
hal.linker.enable_alt_load_copy_rodata	0

Booting and Running From FPGA Memory

In this configuration, the program is loaded in and runs from internal FPGA memory resources. The FPGA memory resources are automatically configured when the FPGA device is configured, so no additional boot loading operations are required.

Prepare for BSP configuration by following these steps:

1. **Nios II processor reset address**—Ensure that the Nios II processor's reset address is in the FPGA internal memory. Configure this option using SOPC Builder.
2. **Text section linker setting**—Ensure that the `.text` section maps to the internal FPGA memory.
3. **Other sections linker setting**—Ensure that all of the other sections map to the internal FPGA memory.
4. **HAL C run-time configuration settings**—Configure the BSP settings as shown in Table 2–6.

Table 2–6. BSP Settings to Boot and Run from FPGA Memory

BSP Setting Name	Value
hal.linker.allow_code_at_reset	1
hal.linker.enable_alt_load	0
hal.linker.enable_alt_load_copy_rwdata	0
hal.linker.enable_alt_load_copy_exceptions	0
hal.linker.enable_alt_load_copy_rodata	0

This configuration requires that you generate FPGA memory Hexadecimal (Intel-format) Files (**.hex**) for compilation to the FPGA image. This step is described in the following section.

Generating and Programming System Memory Images

After you configure your linker settings and boot loader configuration and build the application image **.elf** file, you must create a memory programming file. The flow for creating the memory programming file depends on your choice of FPGA, flash, or EPCS memory.

The easiest way to generate the memory files for your system is to use the application-generated makefile targets.

The available **mem_init.mk** targets are listed in the "Common BSP Tasks" section in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*. You can also perform the same process manually, as shown in the following sections.

Generating memory programming files is not necessary if you want to download and run the application on the target system, for example, during the development and debug cycle.

Programming FPGA Memory with the Nios II Software Build Tools Command Line

If your software application is designed to run from an internal FPGA memory resource, you must convert the application image **.elf** file to one or more **.hex** memory files. The Quartus II software compiles these **.hex** memory files to a **.sof** file. When this image is loaded in the FPGA it initializes the internal memory blocks.

To create a **.hex** memory file from your **.elf** file, type the following command:

```
elf2hex <myapp>.elf <start_addr> <end_addr> --width=<data_width> <hex_filename>.hex ↵
```

This command creates a **.hex** memory file from application image <*myapp*>**.elf**, using data between <*start_addr*> and <*end_addr*>, formatted for memory of width <*data_width*>. The command places the output in the file <*hex_filename*>**.hex**. For information about **elf2hex** command-line arguments, type `elf2hex --help`.

Compile the **.hex** memory files to an FPGA image using the Quartus II software. Initializing FPGA memory resources requires some knowledge of SOPC Builder and the Quartus II software.

Configuring and Programming Flash Memory in Nios II Software Build Tools for Eclipse

The Nios II Software Build Tools for Eclipse provide 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.

"Configuring and Programming Flash Memory from the Command Line" describes several common tasks that you can perform in the Flash Programmer in command-line mode. Most of these tasks can also be performed with the Flash Programmer GUI in the Nios II Software Build Tools for Eclipse.

For information about using the Flash Programmer, refer to "Programming Flash in Altera Embedded Systems" in the *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook.*

Configuring and Programming Flash Memory from the Command Line

After you configure and build your BSP project and your application image **.elf** file, you must generate a flash programming file. The **nios2-flash-programmer** tool uses this file to configure the flash memory device through a programming cable, such as the USB-Blaster cable.

Creating a Flash Image File

If a boot loader application is required in your system, then you must first create a flash image file for your system. This section shows some standard commands in the Nios II Software Build Tools command line to create a flash image file. The section does not address the case of programming and configuring the FPGA image from flash memory.

The following standard commands create a flash image file for your flash memory device:

- **Boot loader required and EPCS flash device used**—To create an EPCS flash device image, type the following command:

  ```
  elf2flash --epcs --after=<standard>.flash --input=<myapp>.elf \
  --output=<myapp>.flash ↵
  ```

  This command converts the application image in the file <*myapp*>**.elf** to a flash record format, and creates the new file <*myapp*>**.flash** that contains the new flash record appended to the FPGA hardware image in <*standard*>**.flash**.

- **Boot loader required and CFI flash memory used**—To create a CFI flash memory image, type the following command:

  ```
  elf2flash --base=0x0 --reset=0x0 --end=0x1000000 \
            --boot=<boot_loader_cfi>.srec \
            --input=<myapp>.elf --output=<myapp>.flash ↵
  ```

  This command converts the application image in the file <*myapp*>**.elf** to a flash record format, and creates the new file <*myapp*>**.flash** that contains the new flash record appended to the CFI boot loader in <*boot_loader_cfi*>**.srec**. The flash record is to be downloaded to the reset address of the Nios II processor, `0x0`, and the base address of the flash device is `0x0`. If you use the Altera-supplied boot loader, your user-created program sections are also loaded from the flash memory to their run-time locations.

- **No boot loader required and CFI flash memory used**—To create a CFI flash memory image, if no boot loader is required, type the following command:

  ```
  elf2flash --base=0x0 --reset=0x0 --end=0x1000000 \
            --input=<myapp>.elf --output=<myapp>.flash ↵
  ```

  This command and its effect are almost identical to those of the command to create a CFI flash memory image if a boot loader is required. In this case, no boot loader is required, and therefore the `--boot` command-line option is not present.

The Nios II EDS includes two precompiled boot loaders for your use, one for CFI flash devices and another for EPCS flash devices. The source code for these boot loaders can be found in the
<*Nios II EDS install dir***>/components/altera_nios2/boot_loader_sources/** directory.

Programming Flash Memory

The easiest way to program your system flash memory is to use the application-generated makefile target called **program-flash**. This target automatically downloads the flash image file to your development board through a JTAG download cable. You can also perform this process manually, using the **nios2-flash-programmer** utility. This utility takes a flash file and some command line arguments, and programs your system's flash memory. The following command-line examples illustrate use of the **nios2-flash-programmer** utility to program your system flash memory:

- **Programming CFI Flash Memory**—To program CFI flash memory with your flash image file, type the following command:

  ```
  nios2-flash-programmer --base=0x0 <myapp>.flash ↵
  ```

  This command programs a flash memory located at base address `0x0` with a flash image file called *<myapp>***.flash**.

- **Programming EPCS Flash Memory**—To program EPCS flash memory with your flash image file, type the following command:

  ```
  nios2-flash-programmer --epcs --base=0x0 <myapp>.flash ↵
  ```

  This command programs an EPCS flash memory located at base address `0x0` with a flash image file called *<myapp>***.flash**.

The **nios2-flash-programmer** utility requires that your FPGA is already configured with your system hardware image. You must download your **.sof** file with the **nios2-configure-sof** command before running the **nios2-flash-programmer** utility.

For more information about how to configure, program, and manage your flash memory devices, refer to the *Nios II Flash Programmer User Guide*.

Conclusion

Altera recommends that you use the Nios II Software Build Tools flow to develop software for hardware designs containing a Nios II processor. The easiest way to use the Software Build Tools is with the Nios II Software Build Tools for Eclipse and the Nios II BSP Editor.

This chapter provides information about the Nios II Software Build Tools flow that complements the *Nios II Software Developer's Handbook*. It discusses recommended design practices and implementation information, and provides pointers to related topics for more in-depth information.

Document Revision History

Table 2–7 shows the revision history for this document.

Table 2–7. Document Revision History

Date	Version	Changes
July 2011	1.4	■ Updated information about GCC toolchain, Nios II SBT for Eclipse. ■ Added availability of pre- and post-processing settings in application and BSP makefiles. ■ Updated references.
December 2009	1.3	■ Updated for Nios II Software Build Tools for Eclipse. ■ Removed all Nios II IDE instructions. ■ Replaced all instances of Nios II IDE instructions with instructions for Nios II Software Build Tools for Eclipse.
July 2009	1.2	Added Nios II BSP Editor.
June 2008	1.1	Corrected Table of Contents.
March 2008	1.0	Initial release.