

This section of the Embedded Design Handbook describes how to most effectively use the Altera® tools for embedded system software development, and recommends design styles and practices for developing, debugging, and optimizing the software for embedded systems using Altera-provided tools. The section introduces concepts to new users of Altera's embedded solutions, and helps to increase the design efficiency of the experienced user.

This section includes the following chapters:

- [Chapter 2, Developing Nios II Software](#)
- [Chapter 3, Debugging Nios II Designs](#)
- [Chapter 4, Nios II Command-Line Tools](#)
- [Chapter 5, Optimizing Nios II C2H Compiler Results](#)



For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.



## 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 2–23
- “Optimizing the Application” on page 2–43
- “Linking Applications” on page 2–50
- “Application Boot Loading and Programming System Memory” on page 2–51



When you install the Nios II EDS, you specify a root directory for the EDS file structure. For example, if the Nios II EDS 9.1 is installed on the Windows operating system, the root directory might be `c:\altera\91\nios2eds`. The root directory can be any accessible location in your file system. For simplicity, this handbook refers to this directory as `<Nios II EDS install path>`. The Nios II EDS defines the environment variable `SOPC_KIT_NIOS2` to represent `<Nios II EDS install path>`.

## 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**—Most 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”
- “Development Flow Guidelines” on page 2-6
- “Nios II Software Build Tools Flow” on page 2-7
- “Configuring BSP and Application Projects” on page 2-8
- “Ensuring Software Project Coherency” on page 2-19

## 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.

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 EDS includes a set of known working designs, called hardware example designs, which are excellent starting points for your own design.
- **Begin with a known software example design**—The Nios II EDS includes a set of preconfigured application and BSP 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 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](http://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 “Example Designs” in the *Overview* chapter of the *Nios II Software Developer’s Handbook*.

To use a software example design, perform the following 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” 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 “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:

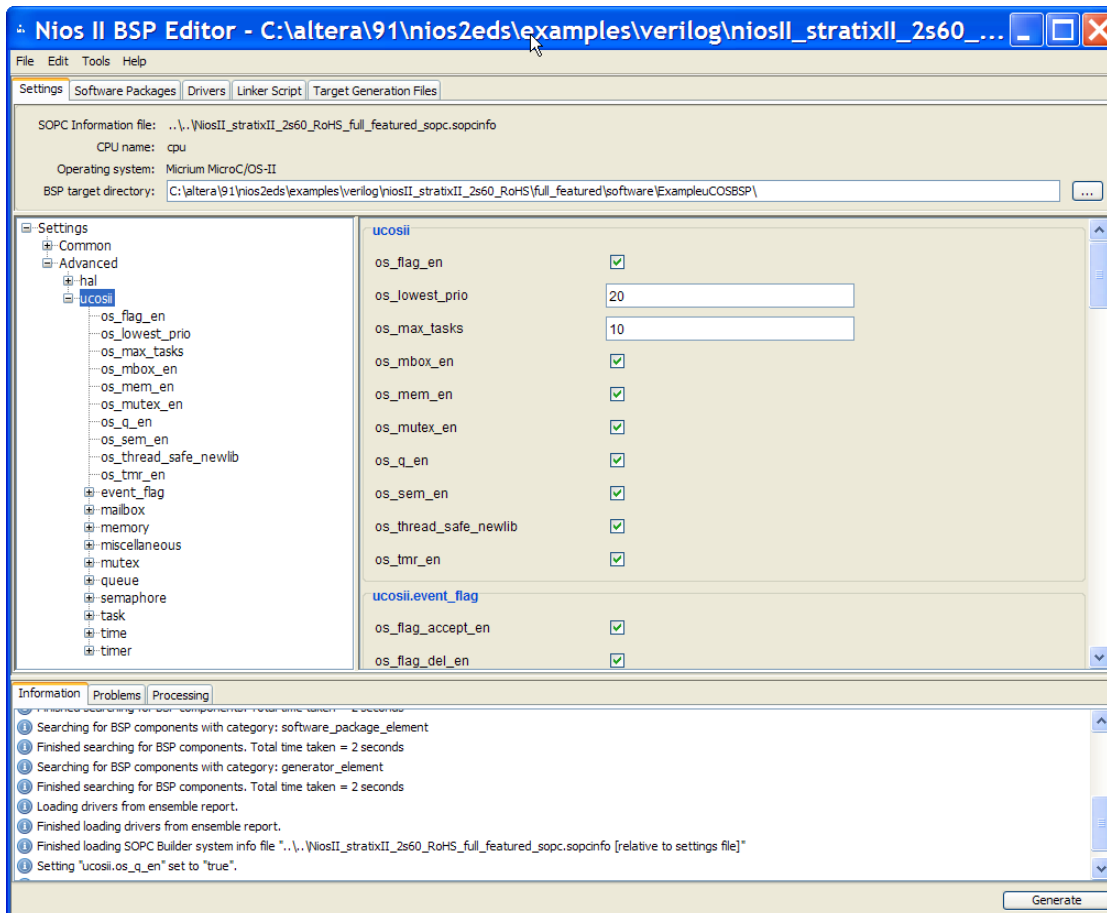
- **MicroC/OS-II BSP settings**—The MicroC/OS-II RTOS supports many configuration options. All of these options can be enabled and disabled with BSP settings. Some of the options are enabled by default. A comprehensive list of BSP settings for MicroC/OS-II is shown in the **Settings** tab of the Nios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

 The MicroC/OS-II BSP settings are also described in “Settings” in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer’s Handbook*.

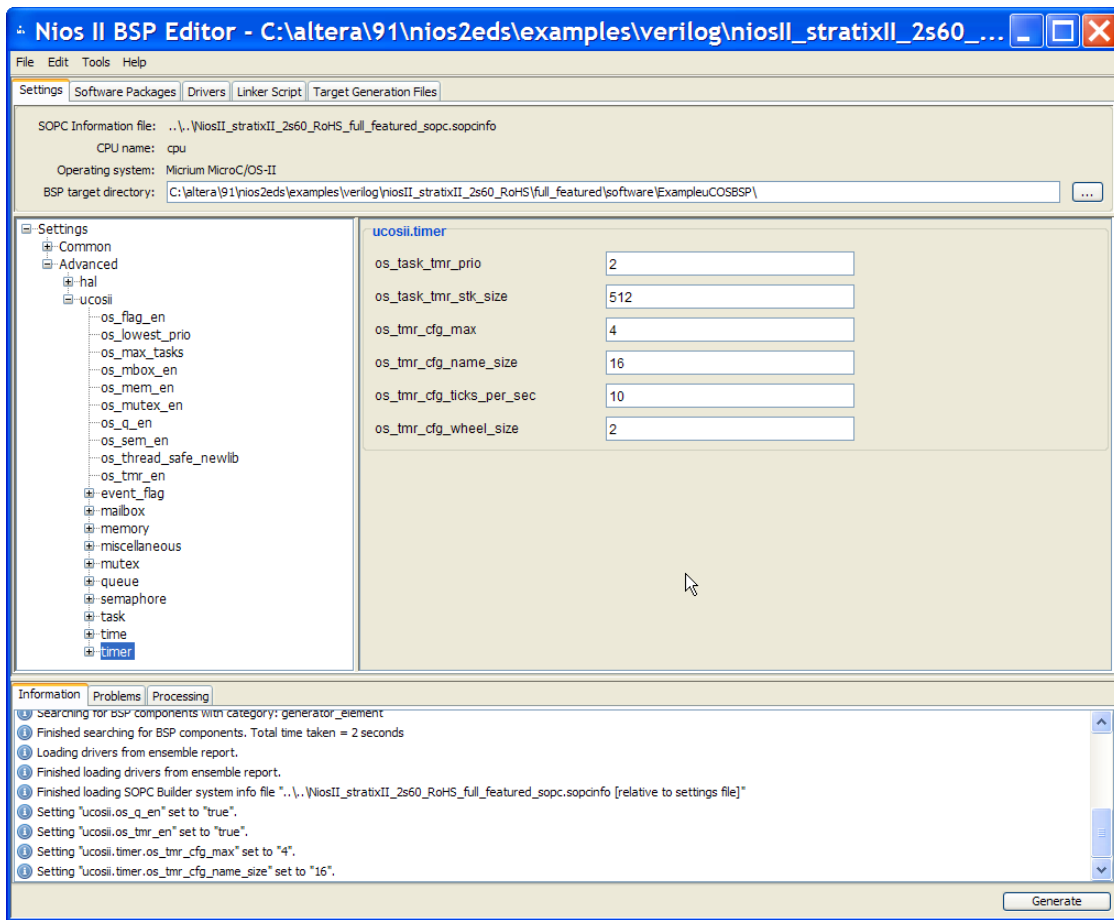
- **MicroC/OS-II setting modification**—Modifying the MicroC/OS-II options modifies the **system.h** file, which is used to compile the BSP library file.
- **MicroC/OS-II initialization**—The core MicroC/OS-II RTOS is initialized during the execution of the C run-time initialization (`crt0`) code block. 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 2–25.

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.

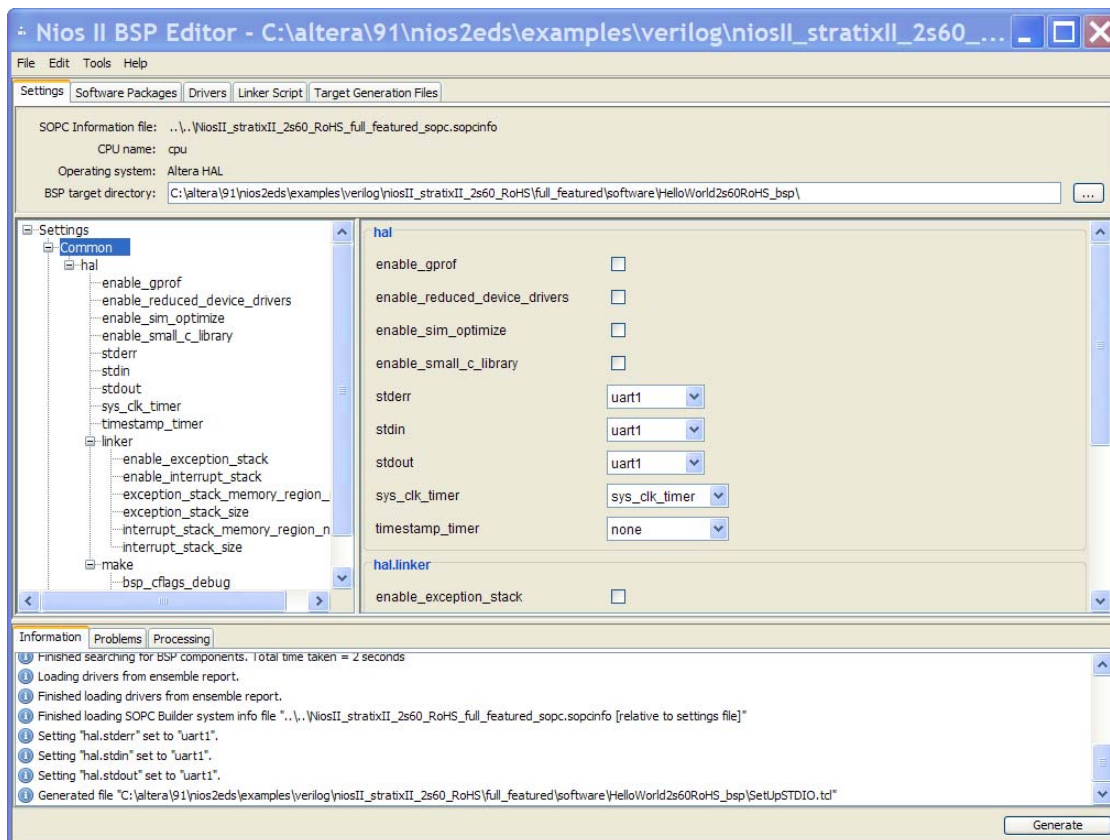
 For more information about BSP settings, refer to “Settings” in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer’s Handbook*.

- **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 (`cr0`) code block. After the `cr0` code block runs, the HAL resources are available for your application to use. For more information, refer to “[cr0 Initialization](#)” on page 2-25.

You can configure the HAL in the BSP Editor. [Figure 2-3 on page 2-13](#) 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, execute the following 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 2-13](#), 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, execute the following steps:

1. With any text editor, create a new file called **example.tcl**.
2. Insert the contents of [Example 2-4](#) in the file.

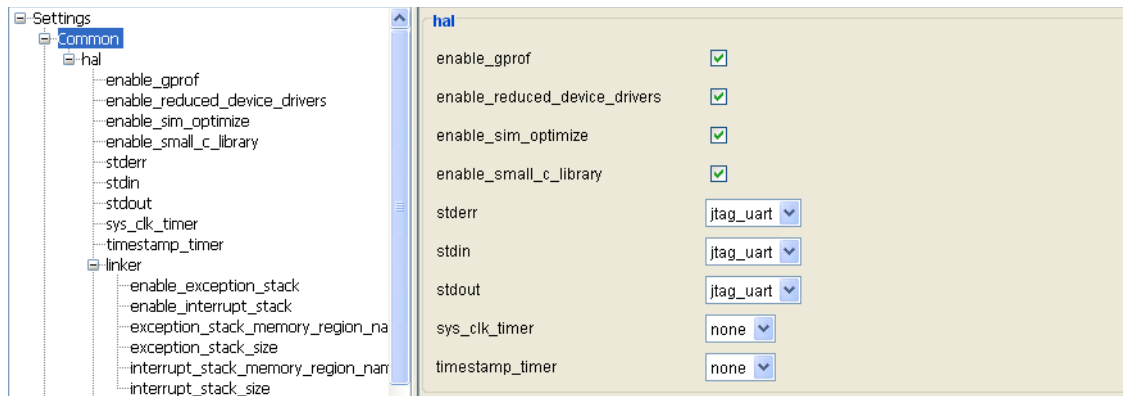
**Example 2-4.** BSP Configuration Tcl Script example.tcl


```


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 “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.

## 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. 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 mentioned in “[Software Example Designs](#)” on page 2-8 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” in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer’s Handbook*.

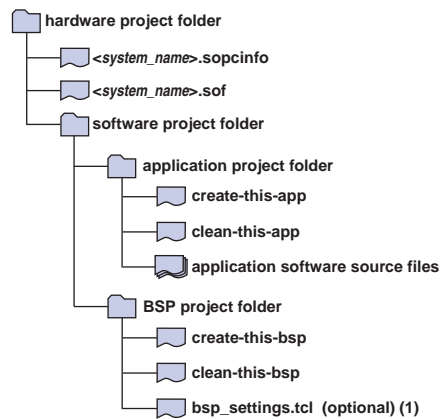
## Ensuring Software Project Coherency

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, as in the Nios II installation **software\_examples** directories. 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 2-9.

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 software project example designs 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 in volume 5 of the *Quartus II Handbook*.

## 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” on page 2-23
- “System Startup in HAL-Based Applications” on page 2-24
- “HAL Peripheral Services” on page 2-28
- “Accessing Memory With the Nios II Processor” on page 2-39
- “Handling Exceptions” on page 2-42
- “Modifying the Exception Handler” on page 2-43

### 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 2-9](#), 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 2-50](#) and [“Application Boot Loading and Programming System Memory” on page 2-51](#).

### Controlling HAL Initialization

As noted in [“HAL Initialization” on page 2-26](#), 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 `init_i` 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. 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” 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 2-25. 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 2-30.
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 HAL supports only the standard Embedded C++ subset of the full C++ language. C++ programs that use features beyond this subset fail in the HAL environment. C++ features not available in Embedded C++ include polymorphism, templates, and single and multiple object inheritance. In general, features that consume a large amount of memory are not included in Embedded C++. Catch/throw exceptions fail in the MicroC/OS-II environment.

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 path>\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 *Timer Core* chapter in volume 5 of the *Quartus II Handbook*.

### 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  $((1/\text{timer frequency}) \times 2^{32})$  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 2-18.



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 4 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 multiple 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, refer to the *Embedded Processing* page of the Altera website, and click **Embedded Software 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, refer to the host file system Nios II software example design listed in “Nios II Example Design Scripts” in the *Nios II Software Build Tools Reference* 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 Example Design 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 **Embedded Software 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`, `_16DIRECT`, and `_8DIRECT` 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_dcachel_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 perform the following 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_dcachel_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”](#)
- [“Speeding Up System Processing Tasks” on page 2-44](#)
- [“Accelerating Interrupt Service Routines” on page 2-47](#)
- [“Reducing Code Size” on page 2-48](#)

## 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 in SOPC Builder* 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 Nios II Tightly Coupled Memory 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 volume 5 of the *Quartus II Handbook*.

- **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 volume 5 of the *Quartus II Handbook*.

## 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: Nios II Processor](#) 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
<code>hal.max_file_descriptors</code>	4
<code>hal.enable_small_c_library</code>	true
<code>hal.sys_clk_timer</code>	none
<code>hal.timestamp_timer</code>	none
<code>hal.enable_exit</code>	false
<code>hal.enable_c_plus_plus</code>	false
<code>hal.enable_lightweight_device_driver_api</code>	true
<code>hal.enable_clean_exit</code>	false
<code>hal.enable_sim_optimize</code>	false
<code>hal.enable_reduced_device_drivers</code>	true
<code>hal.make.bsp_cflags_optimization</code>	<code>\ "-Os\"</code>

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

 For an example, refer to the BSP project `hal_reduced_footprint`, included in your Quartus II installation, in the hardware project directory of your Altera Nios development board type, in `software_examples/bsp/hal_reduced_footprint`.

- **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`, `.rdata`, `.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`, `.rdata`, `.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 `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 2-52
- “Generating and Programming System Memory Images” on page 2-56

### 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 2-14*.



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” 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 2-25*.

 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 performing the following 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
<code>hal.linker.allow_code_at_reset</code>	1
<code>hal.linker.enable_alt_load</code>	1
<code>hal.linker.enable_alt_load_copy_rwdata</code>	1
<code>hal.linker.enable_alt_load_copy_exceptions</code>	1
<code>hal.linker.enable_alt_load_copy_rodata</code>	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 performing the following 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
<code>hal.linker.allow_code_at_reset</code>	0
<code>hal.linker.enable_alt_load</code>	0
<code>hal.linker.enable_alt_load_copy_rwdata</code>	0
<code>hal.linker.enable_alt_load_copy_exceptions</code>	0
<code>hal.linker.enable_alt_load_copy_rodata</code>	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 performing the following 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
<code>hal.linker.allow_code_at_reset</code>	1
<code>hal.linker.enable_alt_load</code>	0
<code>hal.linker.enable_alt_load_copy_rwdata</code>	0
<code>hal.linker.enable_alt_load_copy_exceptions</code>	0
<code>hal.linker.enable_alt_load_copy_rodata</code>	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 performing the following 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
<code>hal.linker.allow_code_at_reset</code>	0
<code>hal.linker.enable_alt_load</code>	0
<code>hal.linker.enable_alt_load_copy_rwdata</code>	0
<code>hal.linker.enable_alt_load_copy_exceptions</code>	0
<code>hal.linker.enable_alt_load_copy_rodata</code>	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 performing the following 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
<code>hal.linker.allow_code_at_reset</code>	1
<code>hal.linker.enable_alt_load</code>	0
<code>hal.linker.enable_alt_load_copy_rwdata</code>	0
<code>hal.linker.enable_alt_load_copy_exceptions</code>	0
<code>hal.linker.enable_alt_load_copy_rodata</code>	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 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 *<nios2eds 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.

## Referenced Documents

This chapter references the following documents:

- *AN391: Profiling Nios II Systems*
- *AN458: Alternative Nios II Boot Methods*
- *AN459: Guidelines for Developing a Nios II HAL Device Driver*
- *Cache and Tightly-Coupled Memory* chapter of the *Nios II Software Developer's Handbook*
- *Debugging Nios II Designs* chapter of the *Embedded Design Handbook*
- *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*
- *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*
- *DMA Controller Core* chapter in volume 5 of the *Quartus II Handbook*
- *Ethernet and the TCP/IP Networking Stack - Nios II Edition* chapter of the *Nios II Software Developer's Handbook*
- *Exception Handling* chapter of the *Nios II Software Developer's Handbook*
- *Getting Started from the Command Line* chapter of the *Nios II Software Developer's Handbook*
- *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook*
- *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*
- *Hardware Acceleration and Coprocessing* chapter of the *Embedded Design Handbook*
- *Instantiating the Nios II Processor in SOPC Builder* chapter of the *Nios II Processor Reference Handbook*
- *Nios II C2H Compiler User Guide*
- *Nios II Flash Programmer User Guide*
- *Nios II Hardware Development Tutorial*

- *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*
- *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*
- *Nios II Software Developer's Handbook*
- *Optimizing Nios II C2H Compiler Results* chapter of the *Embedded Design Handbook*
- *Overview* chapter of the *Nios II Software Developer's Handbook*
- *Programming Model* chapter of the *Nios II Processor Reference Handbook*
- *Read-Only Zip File System* chapter of the *Nios II Software Developer's Handbook*
- *Scatter-Gather DMA Controller Core* chapter in volume 5 of the *Quartus II Handbook*
- *System ID Core* chapter in volume 5 of the *Quartus II Handbook*
- *Timer Core* chapter in volume 5 of the *Quartus II Handbook*
- *Using Nios II Floating-Point Custom Instructions* Tutorial
- *Using Nios II Tightly Coupled Memory* Tutorial
- *Using the Nios II Integrated Development Environment* appendix of the *Nios II Software Developer's Handbook*

## Document Revision History

Table 2-7 shows the revision history for this chapter.

**Table 2-7.** Document Revision History

<b>Date and Document Version</b>	<b>Changes Made</b>	<b>Summary of Changes</b>
December 2009 v1.3	<ul style="list-style-type: none"> <li>■ Updated for Nios II Software Build Tools for Eclipse</li> <li>■ Removed all Nios II IDE instructions</li> <li>■ Replaced all instances of Nios II IDE instructions with instructions for Nios II Software Build tools for Eclipse.</li> </ul>	Updated for Nios II Software Build Tools for Eclipse
July 2009 v1.2	Added Nios II BSP Editor	Added Nios II BSP Editor
June 2008 v1.1	Corrected Table of Contents	—
March 2008 v1.0	Initial release.	—

This chapter describes best practices for debugging Nios® II processor software designs. Debugging these designs involves debugging both hardware and software, which requires familiarity with multiple disciplines. Successful debugging requires expertise in board layout, FPGA configuration, and Nios II software tools and application software. This chapter includes the following sections that discuss debugging techniques and tools to address difficult embedded design problems:

- “Debuggers”
- “Run-Time Analysis Debug Techniques” on page 3–10

## Debuggers

The Nios II development environments offer several tools for debugging Nios II software systems. This section describes the debugging capabilities available in the following development environments:

- “Nios II Software Development Tools”
- “FS2 Console” on page 3–9
- “SignalTap II Embedded Logic Analyzer” on page 3–9
- “Lauterbach Trace32 Debugger and PowerTrace Hardware” on page 3–10
- “Insight and Data Display Debuggers” on page 3–10

## Nios II Software Development Tools

The Nios II Software Build Tools for Eclipse™ is a graphical user interface (GUI) that supports creating, modifying, building, running, and debugging Nios II programs. The Nios II Software Build Tools for the command line are command-line utilities available from a Nios II Command Shell. The Nios II Software Build Tools for Eclipse use the same underlying utilities and scripts as the Nios II Software Build Tools for the command line. Using the Software Build Tools provides fine control over the build process and project settings.

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 FPGA system very efficiently. SOPC Builder does not require that your system contain a Nios II processor. However, it provides complete support for integrating Nios II processors in your system, including some critical debugging features.

The following sections describe debugging tools and support features available in the Nios II software development tools:

- “Nios II System ID”
- “Project Templates”
- “Configuration Options” on page 3–3
- “Nios II GDB Console and GDB Commands” on page 3–6

- “Nios II Console View and stdio Library Functions” on page 3-6
- “Importing Projects Created Using the Nios II Software Build Tools” on page 3-7
- “Selecting a Processor Instance in a Multiple Processor Design” on page 3-7

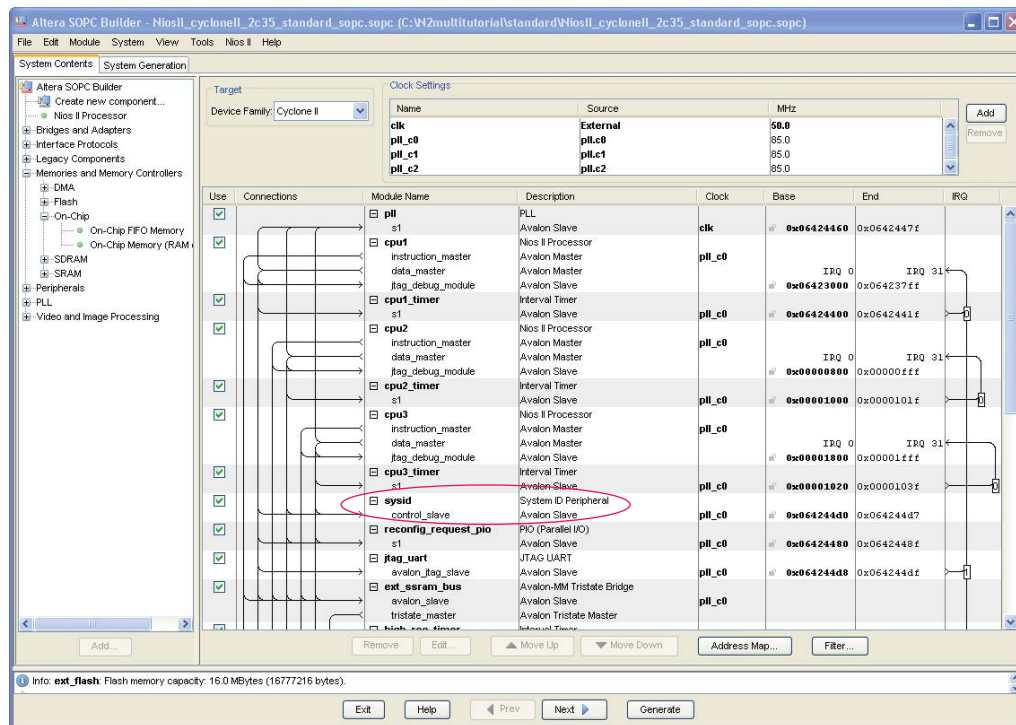
### Nios II System ID

The system identifier (ID) feature is available as a system component in SOPC Builder. The component allows the debugger to identify attempts to download software projects with BSP projects that were generated for a different SOPC Builder system. This feature protects you from inadvertently using an Executable and Linking Format (.elf) file built for a Nios II hardware design that is not currently loaded in the FPGA. If your application image does not run on the hardware implementation for which it was compiled, the results are unpredictable.

To start your design with this basic safety net, in the Nios II Software Build Tools for Eclipse **Debug Configurations** dialog box, in the **Target Connection** tab, ensure that **Ignore mismatched system ID** is not turned on.

The system ID feature requires that the SOPC Builder design include a system ID component. [Figure 3-1](#) shows an SOPC Builder system with a system ID component.

**Figure 3-1.** SOPC Builder System With System ID Component



For more information about the System ID component, refer to the [System ID Core](#) chapter in volume 5 of the *Quartus II Handbook*.

### Project Templates

The Nios II Software Build Tools for Eclipse help you to create a simple, small, and pretested software project to test a new board.

The Nios II Software Build Tools for Eclipse provide a mechanism to create new software projects using project templates. To create a simple test program to test a new board, perform the following steps:

1. In the Nios II perspective, on the File menu, point to **New**, and click **Nios II Application and BSP from Template**.

The New Project wizard for Nios II C/C++ application projects appears.

2. Specify the SOPC information (**.sopcinfo**) file for your design. The folder in which this file is located is your project directory.
3. If your hardware design contains multiple Nios II processors, in the **CPU** list, click the processor you wish to run this application software.
4. Specify a project name.
5. In the **Templates** list, click **Hello World Small**.
6. Click **Next**.
7. Click **Finish**.

The Hello World Small template is a very simple, small application. Using a simple, small application minimizes the number of potential failures that can occur as you bring up a new piece of hardware.

To create a new project for which you already have source code, perform the preceding steps with the following exceptions:

- In step 5, click **Blank Project**.
- After you perform step 7, perform the following steps:
  - a. Create the new directory  
`<your_project_directory>/software/<project_name>/source`, where  
`<project_name>` is the project name you specified in step 4.
  - b. Copy your source code files to the new project by copying them to the new  
`<your_project_directory>/software/<project_name>/source` directory.
  - c. In the **Project Explorer** tab, right-click your application project name, and click **Refresh**. The new **source** folder appears under your application project name.

### Configuration Options

The following Nios II Software Build Tools for Eclipse configuration options increase the amount of debugging information available for your application image **.elf** file:

- **Objdump File**
- **Show Make Commands**
- **Show Line Numbers**

### Objdump File

You can direct the Nios II build process to generate helpful information about your `.elf` file in an object dump text file (`.objdump`). The `.objdump` file contains information about the memory sections and their layout, the addresses of functions, and the original C source code interleaved with the assembly code. [Example 3-1](#) shows part of the C and assembly code section of an `.objdump` file for the Nios II built-in Hello World Small project.

#### Example 3-1. Piece of Code in .objdump File From Hello World Small Project

---

```
06000170 <main>:

include "sys/alt_stdio.h"

int main()
{
6000170:deffff04 addisp,sp,-4
alt_putstr("Hello from Nios II!\n");
6000174:01018034 movhir4,1536
6000178:2102ba04 addir4,r4,2792
600017c:dfc00015 stwra,0(sp)
6000180:60001c00 call60001c0 <alt_putstr>
6000184:003fff06 br6000184 <main+0x14>

06000188 <alt_main>:
* the users application, i.e. main().
*/

void alt_main (void)
{
6000188:deffff04 addisp,sp,-4
600018c:dfc00015 stwra,0(sp)

static ALT_INLINE void ALT_ALWAYS_INLINE
alt_irq_init (const void* base)
{
NIOS2_WRITE_IENABLE (0);
6000190:000170fa wrctlisable,zero
NIOS2_WRITE_STATUS (NIOS2_STATUS_PIE_MSK);
6000194:00800044 movir2,1
6000198:1001703a wrctlstatus,r2
```

---

To enable this option in the Nios II Software Build Tools for Eclipse, perform the following steps:

1. In the Project Explorer window, right-click your application project and click **Properties**.
2. On the list to the left, click **Nios II Application Properties**.
3. On the **Nios II Application Properties** page, turn on **Create object dump**.
4. Click **Apply**.
5. Click **OK**.

After the next build, the `.objdump` file is found in the same directory as the newly built `.elf` file.

After the next build generates the `.elf` file, the build runs the `nios2-elf-objdump` command with the options `--disassemble-all`, `--source`, and `--all-headers` on the generated `.elf` file.

In the Nios II user-managed tool flow, you can edit the settings in the application makefile that determine the options with which the `nios2-elf-objdump` command runs. Running the `create-this-app` script, or the `nios2-app-generate-makefile` script, creates the following lines in the application makefile:

```
#Options to control objdump.  
CREATE_OBJDUMP := 1  
OBJDUMP_INCLUDE_SOURCE := 0  
OBJDUMP_FULL_CONTENTS := 0
```

Edit these options to control the `.objdump` file according to your preferences for the project:

- `CREATE_OBJDUMP`—The value 1 directs `nios2-elf-objdump` to run with the options `--disassemble`, `--syms`, `--all-header`, and `--source`.
- `OBJDUMP_INCLUDE_SOURCE`—The value 1 adds the option `--source` to the `nios2-elf-objdump` command line.
- `OBJDUMP_FULL_CONTENTS`—The value 1 adds the option `--full-contents` to the `nios2-elf-objdump` command line.



For detailed information about the information each command-line option generates, in a Nios II Command Shell, type the following command:

```
nios2-elf-objdump --help ←
```

### Show Make Commands

To enable a verbose mode for the `make` command, in which the individual Makefile commands appear in the display as they are run, in the Nios II Software Build Tools for Eclipse, perform the following steps:

1. In the Project Explorer window, right-click your application project and click **Properties**.
2. On the list to the left, click **C/C++ Build**.
3. On the **C/C++ Build** page, turn off **Use default build command**.
4. For **Build command**, type `make -d`.
5. Click **Apply**.
6. Click **OK**.

### Show Line Numbers

To enable display of C source-code line numbers in the Nios II Software Build Tools for Eclipse, follow these steps:

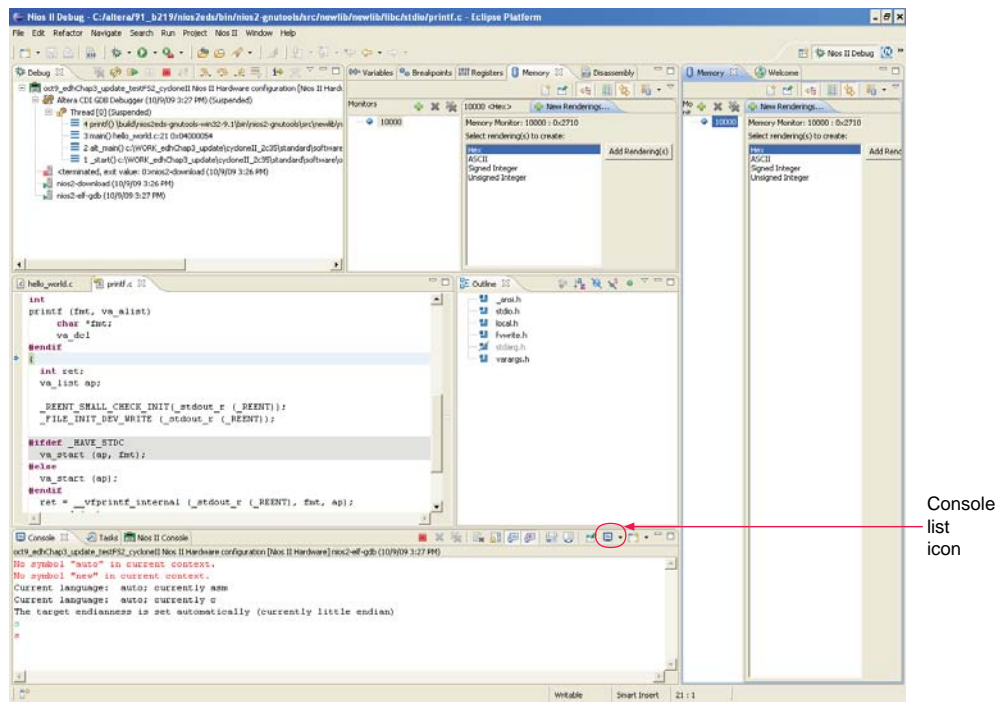
1. On the Window menu, click **Preferences**.
2. On the list to the left, under **General**, under **Editors**, select **Text Editors**.
3. On the **Text Editors** page, turn on **Show line numbers**.
4. Click **Apply**.
5. Click **OK**.

## Nios II GDB Console and GDB Commands

The Nios II GNU Debugger (GDB) console allows you to send GDB commands to the Nios II processor directly.

To display this console, which allows you to enter your own GDB commands, click the blue monitor icon on the lower right corner of the Nios II Debug perspective. (If the Nios II Debug perspective is not displayed, on the Window menu, click **Open Perspective**, and click **Other** to view the available perspectives). If multiple consoles are connected, click the black arrow next to the blue monitor icon to list the available consoles. On the list, select the GDB console. **Figure 3-2** shows the console list icon—the blue monitor icon and black arrow—that allow you to display the GDB console.

**Figure 3-2.** Console List Icon



An example of a useful command you can enter in the GDB console is `dump binary memory <file> <start_addr> <end_addr> ←`

This command dumps the contents of a specified address range in memory to a file on the host computer. The file type is binary. You can view the generated binary file using the HexEdit hexadecimal-format editor that is available from the HexEdit website ([www.expertcomsoft.com](http://www.expertcomsoft.com)).

## Nios II Console View and stdio Library Functions

When debugging I/O behavior, you should be aware of whether your Nios II software application outputs characters using the `printf()` function from the `stdio` library or the `alt_log_printf()` function. The two functions behave slightly differently, resulting in different system and I/O blocking behavior.

The `alt_log_printf()` function bypasses HAL device drivers and writes directly to the component device registers. The behavior of the two functions may also differ depending on whether you enable the reduced-driver option, whether you set your `nios2-terminal` session or the Nios II Console view in the Nios II Software Build Tools for Eclipse to use a UART or a `jtag_uart` as the standard output device, and whether the `O_NONBLOCK` control code is set. In general, enabling the reduced-driver option disables interrupts, which can affect blocking in `jtag_uart` devices.

To enable the reduced-drivers option, perform the following steps:

1. In the Nios II Software build Tools for Eclipse, in the Project Explorer window, right-click your BSP project.
2. Point to **Nios II** and click **BSP Editor**. The BSP Editor appears.
3. In the BSP Editor, in the **Settings** tab, under **Common**, under **hal**, click **enable\_reduced\_device\_drivers**.
4. Click **Generate**.

 For more information about the `alt_log_printf()` function, refer to "Using Character-Mode Devices" in the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

### Importing Projects Created Using the Nios II Software Build Tools

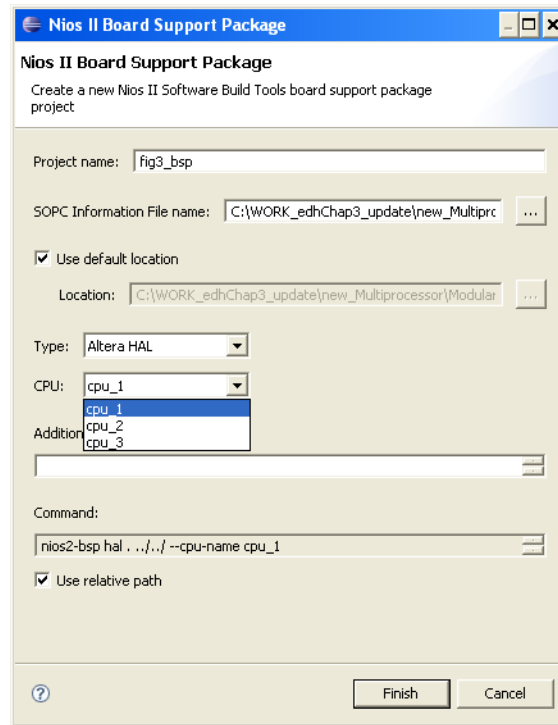
Whether a project is created and built using the Nios II Software Build Tools for Eclipse or using the Nios II Software Build Tools command line, you can debug the resulting `.elf` image file in the Nios II Software Build Tools for Eclipse.

 For information about how to import a project created with the Nios II Software Build Tools command line to the Nios II Software Build Tools for Eclipse, refer to "Importing a Command-Line Project" in the *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook*.

### Selecting a Processor Instance in a Multiple Processor Design

In a design with multiple Nios II processors, you must create a different software project for each processor. When you create an application project, the Nios II Software Build Tools for Eclipse require that you specify a Board Support Package (BSP) project. If a BSP for your application project does not yet exist, you can create one. For BSP generation, you must specify the CPU to which the application project is targeted.

**Figure 3-3** shows how you specify the CPU for the BSP in the Nios II Software Build Tools for Eclipse. The **Nios II Board Support Package** page of the New Project wizard collects the information required for BSP creation. This page derives the list of available CPU choices from the `.sopcinfo` file for the system.

**Figure 3-3.** Nios II Software Build Tools for Eclipse Board Support Package Page— CPU Selection

From the Nios II Command Shell, the **jtagconfig -n** command identifies available JTAG devices and the number of CPUs in the subsystem connected to each JTAG device. [Example 3-2](#) shows the system response to a **jtagconfig -n** command.

#### **Example 3-2.** Two-FPGA System Response to jtagconfig Command

```
[SOPC Builder]$ jtagconfig -n
1) USB-Blaster [USB-0]
  120930DD EP2S60
  Node 19104600
  Node 0C006E00
2) USB-Blaster [USB-1]
  020B40DD EP2C35
  Node 19104601
  Node 19104602
  Node 19104600
  Node 0C006E00
```

The response in [Example 3-2](#) lists two different FPGAs, connected to the running JTAG server through different USB-Blaster™ cables. The cable attached to the USB-0 port is connected to a JTAG node in an SOPC Builder subsystem with a single Nios II core. The cable attached to the USB-1 port is connected to a JTAG node in an SOPC Builder subsystem with three Nios II cores. The node numbers represent JTAG nodes inside the FPGA. The appearance of the node number 0x191046xx in the response confirms that your FPGA implementation has a Nios II processor with a JTAG debug

module. The appearance of a node number `0x0C006Exx` in the response confirms that the FPGA implementation has a JTAG UART component. The CPU instances are identified by the least significant byte of the nodes beginning with 191. The JTAG UART instances are identified by the least significant byte of the nodes beginning with 0C. Instance IDs begin with 0.

Only the CPUs that have JTAG debug modules appear in the listing. Use this listing to confirm you have created JTAG debug modules for the Nios II processors you intended.

## FS2 Console

On Windows platforms, you can use a Nios II-compatible version of the First Silicon Solutions, Inc. (FS2) console. The FS2 console is very helpful for low-level system debug, especially when bringing up a system or a new board. It provides a TCL-based scripting environment and many features for testing your system, from low-level register and memory access to debugging your software (trace, breakpoints, and single-stepping).

To run the FS2 console using the Nios II Software Build Tools command line, use the **nios2-console** command.



For more details about the Nios II-compatible version of the FS2 console, refer to the FS2-provided documentation in your Nios II installation, at `<Nios II EDS install path>\bin\fs2\doc`.

In the FS2 console, the **sld info** command returns information about the JTAG nodes connected to the system-level debug (SLD) hubs—one SLD hub per FPGA—in your system. If you receive a failure response, refer to the FS2-provided documentation for more information.

Use the **sld info** command to verify your system configuration. After communication is established, you can perform simple memory reads and writes to verify basic system functionality. The FS2 console can write bytes or words, if Avalon® Memory-Mapped (Avalon-MM) interface `byteenable` signals are present. In contrast, the Nios II Software Build Tools for Eclipse memory window can perform only 32-bit reads and writes regardless of the 8- or 16-bit width settings for the values retrieved. If you encounter any issues, you can perform these reads and writes and capture SignalTap® II embedded logic analyzer traces of related hardware signals to diagnose a hardware level problem in the memory access paths.

## SignalTap II Embedded Logic Analyzer

The SignalTap II embedded logic analyzer can help you to catch some software-related problems, such as an interrupt service routine that does not properly clear the interrupt signal.



For information about the SignalTap II embedded logic analyzer, refer to the *Design Debugging Using the SignalTap II Embedded Logic Analyzer* chapter in volume 3 of the *Quartus II Handbook* and *AN323: Using SignalTap II Embedded Logic Analyzers in SOPC Builder Systems*, and the *Verification and Board Bring-Up* chapter of the *Embedded Design Handbook*.

The Nios II plug-in for the SignalTap II embedded logic analyzer enables you to capture a Nios II processor's program execution.

 For more information about the Nios II plug-in for the SignalTap II embedded logic analyzer, refer to *AN446: Debugging Nios II Systems with the SignalTap II Logic Analyzer*.

## Lauterbach Trace32 Debugger and PowerTrace Hardware

Lauterbach Datentechnik GmbH (Lauterbach) ([www.lauterbach.com](http://www.lauterbach.com)) provides the Trace32 ICD-Debugger for the Nios II processor. The product contains both hardware and software. In addition to a connection for the 10-pin JTAG connector that is used for the Altera USB-Blaster cable, the PowerTrace hardware has a 38-pin mictor connection option.

Lauterbach also provides a module for off-chip trace capture and an instruction-set simulator for Nios II systems.

The order in which devices are powered up is critical. The Lauterbach PowerTrace hardware must always be powered when power to the FPGA hardware is applied or terminated. The Lauterbach PowerTrace hardware's protection circuitry is enabled after the module is powered up.

 For more information about the Lauterbach PowerTrace hardware and the required power-up sequence, refer to *AN543: Debugging Nios II Software Using the Lauterbach Debugger*.

## Insight and Data Display Debuggers

The Tcl/Tk-based Insight GDB GUI installs with the Altera-specific GNU GDB distribution that is part of the Nios II Embedded Design Suite (EDS). To launch the Insight debugger from the Nios II Command Shell, type the following command:  
`nios2-debug <file>.elf ↵`

Although the Insight debugger has fewer features than the Nios II Software Build Tools for Eclipse, this debugger supports faster communication between host and target, and therefore provides a more responsive debugging experience.

Another alternative debugger is the Data Display Debugger (DDD). This debugger is compatible with GDB commands—it is a user interface to the GDB debugger—and can therefore be used to debug Nios II software designs. The DDD can display data structures as graphs.


## Run-Time Analysis Debug Techniques

This section discusses methods and tools available to analyze a running software system.

## Software Profiling

Altera provides the following tools to profile the run-time behavior of your software system:

- **GNU profiler**—The Nios II EDS toolchain includes the **gprof** utility for profiling your application. This method of profiling reports how long various functions run in your application.
- **High resolution timer**—The SOPC Builder timer peripheral is a simple time counter that can determine the amount of time a given subroutine or code segment runs. You can read it at various points in the source code to calculate elapsed time between timer samples.
- **Performance counter peripheral**—The SOPC Builder performance counter peripheral can profile several different sections of code with a series of counter peripherals. This peripheral includes a simple software API that enables you to print out the results of these timers through the Nios II processor's `stdio` services.

 For more information about how to profile your software application, refer to [AN391: Profiling Nios II Systems](#).

 For additional information about the SOPC Builder timer peripheral, refer to the [Timer Core](#) chapter in volume 5 of the *Quartus II Handbook*, and to the [Developing Nios II Software](#) chapter of the *Embedded Design Handbook*.

 For additional information about the SOPC Builder performance counter peripheral, refer to the [Performance Counter Core](#) chapter in volume 5 of the *Quartus II Handbook*.

## Watchpoints

Watchpoints provide a powerful method to capture all writes to a global variable that appears to be corrupted. The Nios II Software Build Tools for Eclipse support watchpoints directly.

For more information about watchpoints, refer to the Nios II online Help. In Nios II Software Build Tools for Eclipse, on the Help menu, click **Search**. In the search field, type `watchpoint`, and select the topic **Adding watchpoints**.

To enable watchpoints, you must configure the Nios II processor's debug level in SOPC Builder to debug level 2 or higher. To configure the Nios II processor's debug level in SOPC Builder to the appropriate level, perform the following steps:

1. On the SOPC Builder **System Contents** tab, right-click the desired Nios II processor component. A list of options appears.
2. On the list, click **Edit**. The Nios II processor configuration page appears.
3. Click the **JTAG Debug Module** tab, shown in [Figure 3-4 on page 3-13](#).
4. Select **Level 2**, **Level 3**, or **Level 4**.
5. Click **Finish**.


Depending on the debug level you select, a maximum of four watchpoints, or data triggers, are available. [Figure 3-4 on page 3-13](#) shows the number of data triggers available for each debug level. The higher your debug level, the more logic resources you use on the FPGA.

 For more information about the Nios II processor debug levels, refer to the *Instantiating the Nios II Processor in SOPC Builder* chapter of the *Nios II Processor Reference Handbook*.

## Stack Overflow

Stack overflow is a common problem in embedded systems, because their limited memory requires that your application have a limited stack size. When your system runs a real-time operating system, each running task has its own stack, increasing the probability of a stack overflow condition. As an example of how this condition may occur, consider a recursive function, such as a function that calculates a factorial value. In a typical implementation of this function, `factorial(n)` is the result of multiplying the value `n` by another invocation of the factorial function, `factorial(n-1)`. For large values of `n`, this recursive function causes many call stack frames to be stored on the stack, until it eventually overflows before calculating the final function return value.

Using the Nios II Software Build Tools for Eclipse, you can enable the HAL to check for stack overflow. If you enable stack overflow checking and you register an instruction-related exception handler, on stack overflow, the HAL calls the instruction-related exception handler. If you enable stack overflow checking and do not register an instruction-related exception handler, and you enable a JTAG debug module for your Nios II processor, on stack overflow, execution pauses in the debugger, exactly as it does when the debugger encounters a breakpoint. To enable stack overflow checking, in the BSP Editor, in the **Settings** tab, under **Advanced**, under **hal**, click **enable\_runtime\_stack\_checking**.

 For information about the instruction-related exception handler, refer to "The Instruction-Related Exception Handler" in the *Exception Handling* chapter of the *Nios II Software Developer's Handbook*.

## Hardware Abstraction Layer (HAL)

The Altera HAL provides the interfaces and resources required by the device drivers for most SOPC Builder system peripherals. You can customize and debug these drivers for your own SOPC Builder system. To learn more about debugging HAL device drivers and SOPC Builder peripherals, refer to *AN459: Guidelines for Developing a Nios II HAL Device Driver*.

## Breakpoints

You can set hardware breakpoints on code located in read-only memory such as flash memory. If you set a breakpoint in a read-only area of memory, a hardware breakpoint, rather than a software breakpoint, is selected automatically.

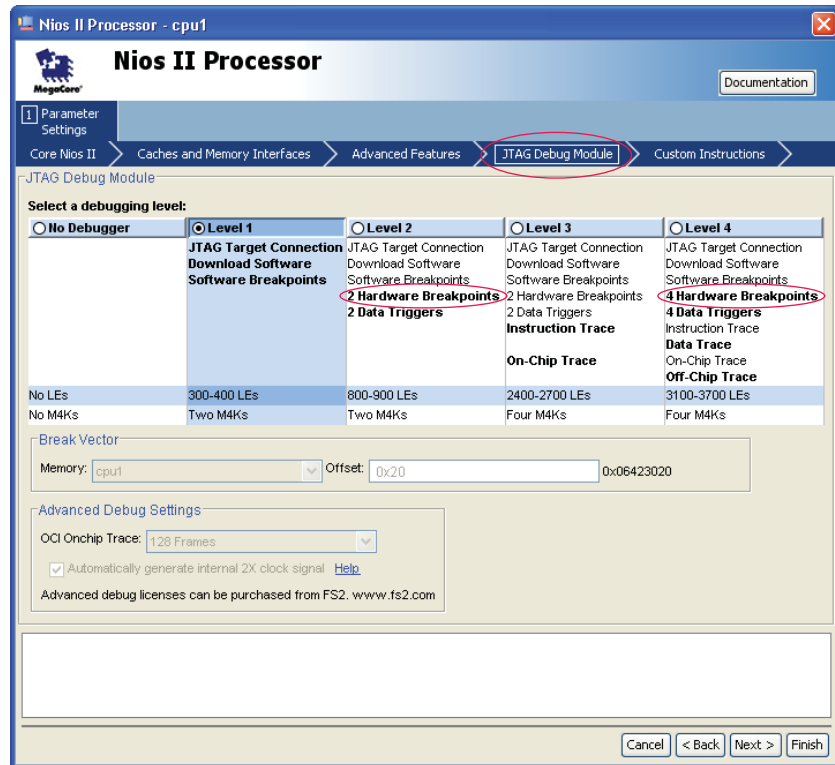
To enable hardware breakpoints, you must configure the Nios II processor's debug level in SOPC Builder to debug level 2 or higher. To configure the Nios II processor's debug level in SOPC Builder to the appropriate level, perform the following steps:

1. On the SOPC Builder **System Contents** tab, right-click the desired Nios II processor component.
2. On the right button pop-up menu, click **Edit**. The Nios II processor configuration page appears.

3. Click the **JTAG Debug Module** tab, shown in [Figure 3-4](#).
4. Select **Level 2**, **Level 3**, or **Level 4**.
5. Click **Finish**.

Depending on the debug level you select, a maximum of four hardware breakpoints are available. [Figure 3-4](#) shows the number of hardware breakpoints available for each debug level. The higher your debug level, the more logic resources you use on the FPGA.

**Figure 3-4.** Nios II Processor — JTAG Debug Module — SOPC Builder Configuration Page



For more information about the Nios II processor debug levels, refer to the *Instantiating the Nios II Processor in SOPC Builder* chapter of the *Nios II Processor Reference Handbook*.

## Debugger Stepping and Using No Optimizations

Use the **None (-O0)** optimization level compiler switch to disable optimizations for debugging. Otherwise, the breakpoint and stepping behavior of your debugger may not match the source code you wrote. This behavior mismatch between code execution and high-level original source code may occur even when you click the **i** button to use the instruction stepping mode at the assembler instruction level. This mismatch occurs because optimization and in-lining by the compiler eliminated some of your original source code.

To set the **None (-O0)** optimization level compiler switch in the Nios II Software Build Tools for Eclipse, perform the following steps:

1. In the Nios II perspective, right-click your application project. A list of options appears.
2. On the list, click **Properties**. The **Properties for <project name>** dialog box appears.
3. In the left pane, click **Nios II Application Properties**.
4. In the **Optimization Level** list, select **Off**.
5. Click **Apply**.
6. Click **OK**.

## Conclusion

Successful debugging of Nios II designs requires expertise in board layout, FPGA configuration, and Nios II software tools and application software. Altera and third-party tools are available to help you debug your Nios II application. This chapter describes debugging techniques and tools to address difficult embedded design problems.

## Referenced Documents

This chapter references the following documents:

- *AN323: Using SignalTap II Embedded Logic Analyzers in SOPC Builder Systems*
- *AN391: Profiling Nios II Systems*
- *AN446: Debugging Nios II Systems with the SignalTap II Logic Analyzer*
- *AN459: Guidelines for Developing a Nios II HAL Device Driver*
- *AN543: Debugging Nios II Software Using the Lauterbach Debugger*
- *Design Debugging Using the SignalTap II Embedded Logic Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Developing Nios II Software* chapter of the *Embedded Design Handbook*
- *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*
- *Exception Handling* chapter of the *Nios II Software Developer's Handbook*
- *Instantiating the Nios II Processor in SOPC Builder* chapter of the *Nios II Processor Reference Handbook*
- *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook*
- *Performance Counter Core* chapter in volume 5 of the *Quartus II Handbook*
- *System ID Core* chapter in volume 5 of the *Quartus II Handbook*
- *Timer Core* chapter in volume 5 of the *Quartus II Handbook*
- *Verification and Board Bring-Up* chapter of the *Embedded Design Handbook*

## Document Revision History

Table 3-1 shows the revision history for this chapter.

**Table 3-1.** Document Revision History

Date and Document Version	Changes Made	Summary of Changes
December 2009 v1.3	<ul style="list-style-type: none"> <li>■ Updated for Nios II Software Build Tools for Eclipse.</li> <li>■ Removed all Nios II IDE instructions. Replaced all instances of Nios II IDE instructions with instructions for Nios II Software Build Tools for Eclipse.</li> </ul>	Updated for Nios II Software Build Tools for Eclipse.
April 2009 v1.2	<ul style="list-style-type: none"> <li>■ Added reference to new application note <i>AN543: Debugging Nios II Software Using the Lauterbach Debugger</i></li> <li>■ Removed information made redundant by the new application note from “Lauterbach Trace32 Debugger and PowerTrace Hardware” section.</li> </ul>	Removed information about the Lauterbach debugging tools now described in new application note.
June 2008 v1.1	Corrected Table of Contents	—
March 2008 v1.0	Initial release.	—



## Introduction

This chapter describes the Nios® II command-line tools that are provided with the Nios II Embedded Development Suite (EDS). The chapter describes both the Altera® tools and the GNU tools. Most of the commands are located in the `$SOPC_KIT_NIOS2\bin` and `$SOPC_KIT_NIOS2\sdk2` subdirectories of your Nios II EDS installation.

The Altera command line tools are useful for a range of activities, from board and system-level debugging to programming an FPGA configuration file (`.sof`). For these tools, the examples expand on the brief descriptions of the Altera-provided command-line tools for developing Nios II programs in the *Overview* chapter of the *Nios II Software Developer's Guide*. The Nios II GCC toolchain contains the GNU Compiler Collection, GNU Binary Utilities (binutils), and newlib C library.



All of the commands described in this chapter are available in the Nios II command shell. For most of the commands, you can obtain help in this shell by typing

```
<command name> --help ↵
```

To start the Nios II command shell on Windows platforms, on the Start menu, click **All Programs**. On the All Programs menu, on the Altera submenu, on the Nios II EDS `<version>` submenu, click **Nios II <version> Command Shell**.

On Linux platforms, type the following command:

```
$SOPC_KIT_NIOS2/sdk_shell ↵
```

The command shell is a Bourne-again shell (bash) with a pre-configured environment.

## Altera Command-Line Tools for Board Bringup and Diagnostics

This section describes Altera command-line tools useful for Nios development board bringup and debugging.

### jtagconfig

This command returns information about the devices connected to your host PC through the JTAG interface, for your use in debugging or programming. Use this command to determine if you configured your FPGA correctly.

Many of the other commands depend on successful JTAG connection. If you are unable to use other commands, check whether your JTAG chain differs from the simple, single-device chain used as an example in this chapter.

Type `jtagconfig --help` from a Nios II command shell to display a list of options and a brief usage statement.

### jtagconfig Usage Example

To use the `jtagconfig` command, perform the following steps:

1. Open a Nios II command shell.
2. In the command shell, type the following command:  
`jtagconfig -n` ←

**Example 4-1** shows a typical system response to the `jtagconfig -n` command.

#### Example 4-1. jtagconfig Example Response

---

```
[SOPC Builder]$ jtagconfig -n
1) USB-Blaster [USB-0]
   020050DD  EP1S40/_HARDCOPY_FPGA_PROTOTYPE
      Node 11104600
      Node 0C006E00
```

---

The information in the response varies, depending on the particular FPGA, its configuration, and the JTAG connection cable type. **Table 4-1** describes the information that appears in the response in **Example 4-1**.

**Table 4-1.** Interpretation of jtagconfig Command Response

Value	Description
USB-Blaster [USB-0]	The type of cable. You can have multiple cables connected to your workstation.
EP1S40/_HARDCOPY_FPGA_PROTOTYPE	The device name, as identified by silicon identification number.
Node 11104600	The node number of a JTAG node inside the FPGA. The appearance of a node number between 11104600 and 11046FF, inclusive, in the response confirms that you have a Nios II processor with a JTAG debug module.
Note 0C006E00	The node number of a JTAG node inside the FPGA. The appearance of a node number between 0C006E00 and 0C006EFF, inclusive, in the response confirms that you have a JTAG UART component.

The device name is read from the text file `pgm_parts.txt` in your Quartus® II installation. In **Example 4-1**, the name is `EP1S40/_HARDCOPY_FPGA_PROTOTYPE` because the silicon identification number on the JTAG chain for the FPGA device is `020050DD`, which maps to the names `EP1S40<device-specific name>`, a couple of which end in the string `_HARDCOPY_FPGA_PROTOTYPE`. The internal nodes are nodes on the system-level debug (SLD) hub. All JTAG communication to an Altera FPGA passes through this hub, including advanced debugging capabilities such as the SignalTap® II embedded logic analyzer and the debugging capabilities in the Nios II Integrated Development Environment (IDE).

**Example 4-1** illustrates a single cable connected to a single-device JTAG chain. However, your computer can have multiple JTAG cables, connected to different systems. Each of these systems can have multiple devices in its JTAG chain. Each device can have multiple JTAG debug modules, JTAG UART modules, and other kinds of JTAG nodes. Use the `jtagconfig -n` command to help you understand the devices with JTAG connections to your host PC and how you can access them.

## nios2-configure-sof

This command downloads the specified `.sof` and configures the FPGA according to its contents. At a Nios II command shell prompt, type `nios2-configure-sof --help` for a list of available command-line options.



You must specify the cable and device when you have more than one JTAG cable (USB-Blaster™ or ByteBlaster™ cable) connected to your computer or when you have more than one device (FPGA) in your JTAG chain. Use the `--cable` and `--device` options for this purpose.

### nios2-configure-sof Usage Example

To use the `nios2-configure-sof` command, perform the following steps:

1. Open a Nios II command shell.
2. In the command shell, change to the directory in which your `.sof` is located. By default, the correct location is the top-level Quartus II project directory.
3. In the command shell, type the following command:

```
nios2-configure-sof ↵
```

The Nios II IDE searches the current directory for a `.sof` and programs it through the specified JTAG cable.

## system-console

The `system-console` command starts a Tcl-based command shell that supports low-level JTAG chain verification and full system-level validation. This tool is available in the Nios II EDS starting in version 8.0.

This application is very helpful for low-level system debug, especially when bringing up a system. It provides a Tcl-based scripting environment and many features for testing your system.

The following important command-line options are available for the `system-console` command:

- The `--script=<your script>.tcl` option directs the System Console to run your Tcl script.
- The `--cli` option directs the System Console to open in your existing shell, rather than opening a new window.
- The `--debug` option directs the System Console to redirect additional debug output to `stderr`.
- The `--project-dir=<project dir>` option directs the System Console to the location of your hardware project. Ensure that you're working with the project you intend—the JTAG chain details and other information depend on the specific project.
- The `--jdi=<JDI file>` option specifies the name-to-node mapping for the JTAG chain elements in your project.



For System Console usage examples and a comprehensive list of system console commands, refer to the *System Console User Guide*. On-line training is available at <http://www.altera.com/training>.

## Altera Command-Line Tools for Hardware Development

This section describes Altera command-line tools useful for hardware project development. They are useful for all projects created with SOPC Builder, whether or not the project includes a Nios II processor.

### quartus\_cmd and socp\_builder

These commands create scripts that automate generation of SOPC Builder systems and compilation of the corresponding Quartus II projects.

You can use these commands to create a flow that maintains only the minimum source files required to build your Quartus II project. If you copy an existing project to use as the basis for development of a new project, you should copy only this minimum set of source files. Similarly, when you check in files to your version control system, you want to check in only the minimum set required to reconstruct the project.

To reconstruct an SOPC Builder system, the following files are required:

- `<project>.qpf` (Quartus II project file)
- `<project>.qsf` (Quartus II settings file)
- `<SOPC Builder system>.socp` (SOPC Builder system description)
- The additional HDL, BDF, or BSF files in your existing project

If you work with the hardware design examples that are provided with the Quartus II installation, Altera recommends that you copy each set of source files to a working directory to avoid modifying the original source files inadvertently. Run the script on the new working directory.

To create a flow that maintains only the minimum source files, perform the following steps:

1. Copy the required source files to a working directory, maintaining a correct copy of each source file elsewhere.
2. Change to this working directory.
3. To generate a `.sof` to configure your FPGA, type the following command sequence:

```
socp_builder --no_splash -s --generate ↵  
quartus_cmd <project>.qpf -c <project>.qsf ↵
```

The shell script in [Example 4-2](#) illustrates these commands. This script automates the process of generating SOPC Builder systems and compiling Quartus II projects across any number of subdirectories. The script is an example only, and may require modification for your project. If you want to compile the Quartus II projects, set the `COMPILE_QUARTUS` variable in the script to 1.

**Example 4-2.** Script to Generate SOPC Builder System and Compile Quartus II Projects (Part 1 of 2)

---

```
#!/bin/sh
COMPILE_QUARTUS=0
#
# Resolve TOP_LEVEL_DIR, default to PWD if no path provided.
#
if [ $# -eq 0 ]; then
    TOP_LEVEL_DIR=$PWD
else
    TOP_LEVEL_DIR=$1
fi
echo "TOP_LEVEL_DIR is $TOP_LEVEL_DIR"
echo
#
# Generate SOPC list...
#
SOPC_LIST=`find $TOP_LEVEL_DIR -name "*.sopc"`
#
# Generate Quartus II project list.
#
PROJ_LIST=`find $TOP_LEVEL_DIR -name "*.qpf" | sed s/\.qpf//g`
#
# Main body of the script. First "generate" all of the SOPC Builder
# systems that are found, then compile the Quartus II projects.
#
#
# Run SOPC Builder to "generate" all of the systems that were found.
#
for SOPC_FN in $SOPC_LIST
do
    cd `dirname $SOPC_FN`
    if [ ! -e `basename $SOPC_FN .sopc`.vhd -a ! -e `basename $SOPC_FN .sopc`.v ]; then
        echo; echo
        echo "INFO: Generating $SOPC_FN SOPC Builder system."
        socp_builder -s --generate=1 --no_splash
        if [ $? -ne 4 ]; then
            echo; echo
            echo "ERROR: SOPC Builder generation for $SOPC_FN has failed!!!"
            echo "ERROR: Please check the SOPC file and data " \
                "in the directory `dirname $SOPC_FN` for errors."
        fi
    else
        echo; echo
        echo "INFO: HDL already exists for $SOPC_FN, skipping Generation!!!"
    fi
    cd $TOP_LEVEL_DIR
done
#
# Continued...
#
```

---

**Example 4-2.** Script to Generate SOPC Builder System and Compile Quartus II Projects (Part 2 of 2)

```

#
# Now, generate all of the Quartus II projects that were found.
#
if [ $COMPILE_QUARTUS ]; then
  for PROJ in $PROJ_LIST
  do
    cd `dirname $PROJ`
    if [ ! -e `basename $PROJ`.sof ]; then
      echo; echo
      echo "INFO: Compiling $PROJ Quartus II Project."
      quartus_cmd `basename $PROJ`.qpf -c `basename $PROJ`.qsf
      if [ $? -ne 4 ]; then
        echo; echo
        echo "ERROR: Quartus II compilation for $PROJ has failed!!!"
        echo "ERROR: Please check the Quartus II project " \
          "in `dirname $PROJ` for details."
      fi
    else
      echo; echo
      echo "INFO: SOF already exists for $PROJ, skipping compilation."
    fi
    cd $TOP_LEVEL_DIR
  done
fi

```



The commands and script in [Example 4-2](#) are provided for example purposes only. Altera does not guarantee the functionality for your particular use.

## Altera Command-Line Tools for Flash Programming

This section describes the command-line tools for programming your Nios II-based design in flash memory.

When you use the Nios II IDE to program flash memory, the Nios II IDE generates a shell script that contains the flash conversion commands and the programming commands. You can use this script as the basis for developing your own command-line flash programming flow.



For more details about the Nios II IDE and command-line usage of the Nios II Flash Programmer and related tools, refer to the [Nios II Flash Programmer User Guide](#).

### nios2-flash-programmer

This command programs common flash interface (CFI) memory. Because the Nios II flash programmer uses the JTAG interface, the `nios2-flash-programmer` command has the same options for this interface as do other commands. You can obtain information about the command-line options for this command with the `--help` option.

#### nios2-flash-programmer Usage Example

You can perform the following steps to program a CFI device:

1. Follow the steps in [“nios2-download” on page 4-9](#), or use the Nios II IDE, to program your FPGA with a design that interfaces successfully to your CFI device.

2. Type the following command to verify that your flash device is detected correctly:

```
nios2-flash-programmer -debug -base=<base address>↵
```

where *<base address>* is the base address of your flash device. The base address of each component is displayed in SOPC Builder. If the flash device is detected, the flash memory's CFI table contents are displayed.

3. Convert your file to flash format (**.flash**) using one of the utilities `elf2flash`, `bin2flash`, or `sof2flash` described in the section “[elf2flash, bin2flash, and sof2flash](#)”.
4. Type the following command to program the resulting **.flash** file in the CFI device:
5. Optionally, type the following command to reset and start the processor at its reset address:

```
nios2-flash-programmer -base=<base address> <file>.flash↵
```

```
nios2-download -g -r↵
```

## elf2flash, bin2flash, and sof2flash

These three commands are often used with the `nios2-flash-programmer` command. The resulting **.flash** file is a standard **.srec** file.

The following two important command-line options are available for the `elf2flash` command:

- The `-boot=<boot copier file>.srec` option directs the `elf2flash` command to prepend a bootloader S-record file to the converted ELF file.
- The `-after=<flash file>.flash` option places the generated **.flash** file—the converted ELF file—immediately following the specified **.flash** file in flash memory.

The `-after` option is commonly used to place the **.elf** file immediately following the **.sof** in an erasable, programmable, configurable serial (EPCS) flash device.



If you use an EPCS device, you must program the hardware image in the device before you program the software image. If you disregard this rule your software image will be corrupted.

Before it writes to any flash device, the Nios II flash programmer erases the entire sector to which it expects to write. In EPCS devices, however, if you generate the software image using the `elf2flash -after` option, the Nios II flash programmer places the software image directly following the hardware image, not on the next flash sector boundary. Therefore, in this case, the Nios II flash programmer does not erase the current sector before placing the software image. However, it does erase the current sector before placing the hardware image.

When you use the flash programmer through the Nios II IDE, you automatically create a script that contains some of these commands. Running the flash programmer creates a shell script (**.sh**) in the **Debug** or **Release** target directory of your project. This script contains the detailed command steps you used to program your flash memory.

[Example 4-3](#) shows a sample auto-generated script.

**Example 4-3. Sample Auto-Generated Script:**


---

```
#!/bin/sh
#
# This file was automatically generated by the Nios II IDE Flash Programmer.
#
# It will be overwritten when the flash programmer options change.
#

cd <full path to your project>/Debug

# Creating .flash file for the FPGA configuration
#"$SOPC_KIT_NIOS2/bin/sof2flash" --offset=0x400000 --input="full path to your SOF" \
  --output="<your design>.flash"

# Programming flash with the FPGA configuration
#"$SOPC_KIT_NIOS2/bin/nios2-flash-programmer" --base=0x00000000 --sidp=0x00810828 \
  --id=1436046714 --timestamp=1169569475 --instance=0 "<your design>.flash"
#
# Creating .flash file for the project
"$SOPC_KIT_NIOS2/bin/elf2flash" --base=0x00000000 --end=0x7ffffff --reset=0x0 \
  --input="<your project name>.elf" --output="ext_flash.flash" \
  --boot="<path to the bootloader>/boot_loader_cfi.srec"

# Programming flash with the project
"$SOPC_KIT_NIOS2/bin/nios2-flash-programmer" --base=0x00000000 --sidp=0x00810828 \
  --id=1436046714 --timestamp=1169569475 --instance=0 "ext_flash.flash"

# Creating .flash file for the read only zip file system
"$SOPC_KIT_NIOS2/bin/bin2flash" --base=0x00000000 --location=0x100000 \
  --input="<full path to your binary file>" --output="<filename>.flash"

# Programming flash with the read only zip file system
"$SOPC_KIT_NIOS2/bin/nios2-flash-programmer" --base=0x00000000 --sidp=0x00810828 \
  --id=1436046714 --timestamp=1169569475 --instance=0 "<filename>.flash"
```

---

The paths, file names, and addresses in the auto-generated script change depending on the names and locations of the files that are converted and on the configuration of your hardware design.

**bin2flash Usage Example**

To program an arbitrary binary file to flash memory, perform the following steps:

1. Type the following command to generate your **.flash** file:

```
bin2flash --location=<offset from the base address> \
  -input=<your file> --output=<your file>.flash ↵
```

2. Type the following command to program your newly created file to flash memory:

```
nios2-flash-programmer -base=<base address> <your file>.flash ↵
```

**Altera Command-Line Tools for Software Development and Debug**

This section describes Altera command-line tools that are useful for software development and debugging.

## nios2-terminal

This command establishes contact with **stdin**, **stdout**, and **stderr** in a Nios II processor subsystem. **stdin**, **stdout**, and **stderr** are routed through a UART (standard UART or JTAG UART) module within this system.

The `nios2-terminal` command allows you to monitor **stdout**, **stderr**, or both, and to provide input to a Nios II processor subsystem through **stdin**. This command behaves the same as the `nios2-configure-sof` command described in “[nios2-configure-sof](#)” on page 4-3 with respect to JTAG cables and devices. However, because multiple JTAG UART modules may exist in your system, the `nios2-terminal` command requires explicit direction to apply to the correct JTAG UART module instance. Specify the instance using the `-instance` command-line option. The first instance in your design is 0 (`-instance "0"`). Additional instances are numbered incrementally, starting at 1 (`-instance "1"`).

## nios2-download

This command parses Nios II **.elf** files, downloads them to a functioning Nios II processor, and optionally runs the **.elf** file.

As for other commands, you can obtain command-line option information with the `--help` option. The `nios2-download` command has the same options as the `nios2-terminal` command for dealing with multiple JTAG cables and Nios II processor subsystems.

### nios2-download Usage Example

To download (and run) a Nios II **.elf** program:

1. Open a Nios II command shell.
2. Change to the directory in which your **.elf** file is located. If you use the Nios II IDE for development, the correct location is often the **Debug** or **Release** subdirectory of your top-level project.
3. In the command shell, type the following command to download and start your program:  

```
nios2-download -g <project name>.elf ←
```
4. Optionally, use the `nios2-terminal` command to connect to view any output or provide any input to the running program.

## nios2-stackreport

This command returns a brief report on the amount of memory still available for stack and heap from your project's **.elf** file.

This command does not help you to determine the amount of stack or heap space your code consumes during runtime, but it does tell you how much space your code has to work in.

[Example 4-4](#) illustrates the information this command provides.

**Example 4-4.** nios2-stackreport Command and Response

---

```
[SOPC Builder]$ nios2-stackreport <your project>.elf
Info: (<your project>.elf) 6312 KBytes program size (code + initialized data).
Info:                10070 KBytes free for stack + heap.
```

---

**nios2-stackreport Usage Example**

To use the `nios2-stackreport` command, perform the following steps:

1. Open a Nios II command shell.
2. Change to the directory in which your `.elf` file is located.
3. In the command shell, type the following command:

```
nios2-stackreport <your project>.elf ←
```

**validate\_zip**

The Nios II IDE uses this command to validate that the files you use for the Read Only Zip Filing System are uncompressed. You can use it for the same purpose.

**validate\_zip Usage Example**

To use the `validate_zip` command, perform the following steps:

1. Open a Nios II command shell.
2. Change to the directory in which your `.zip` file is located.
3. In the command shell, type the following command:

```
validate_zip <file>.zip ←
```

If no response appears, your `.zip` file is not compressed.

**nios2-ide**

On Linux and Windows systems, you can type `nios2-ide` in a command shell to launch the Nios II IDE. On Windows systems, you can also use the Nios II IDE launch icon in SOPC Builder.

The `nios2-ide` command does not call the executable file directly. Instead, it runs a simple Bourne shell wrapper script, which calls the **nios2-ide** executable file. The Linux and Windows platform versions of the wrapper script follow.

**Linux wrapper script**

```
#!/bin/sh
# This is the linux-gtk version of the nios2-ide launcher script
# set the default workspace location for linux
WORKSPACE="$HOME/nios2-ide-workspace-7.2"
WORKSPACE_ARGS="-data $WORKSPACE"
# if -data is already passed in, we can't specify it
# again when calling nios2-ide
for i in $*
do
    if [ "x$i" = "x-data" ]; then
```

```
        WORKSPACE_ARGS=" "  
    fi  
done  
exec $SOPC_KIT_NIOS2/bin/eclipse/nios2-ide -configuration  
$HOME/.nios2-ide-6.1 $WORKSPACE_ARGS "$@"
```

### Windows wrapper script

```
#!/bin/sh  
# This is the win32 version of the nios2-ide launcher script  
# It simply invokes the binary with the same arguments as  
# passed in.  
# By doing this, the user will default to the same workspace as  
# when launched using the Windows shortcut, as "persisted"  
# in the configuration/.settings/org.eclipse.ui.ide.prefs file.  
cd "$SOPC_KIT_NIOS2/bin/eclipse"  
exec ./nios2-ide-console "$@"
```

## nios2-gdb-server

This command starts a GNU Debugger (GDB) JTAG conduit that listens on a specified TCP port for a connection from a GDB client, such as a `nios2-elf-gdb` client.

Occasionally, you may have to terminate a GDB server session. If you no longer have access to the Nios II command shell session in which you started a GDB server session, or if the offending GDB server process results from an errant Nios II IDE debugger session, you should stop the `nios2-gdb-server.exe` process on Windows platforms, or type the following command on Linux platforms:

```
kill -9 -f nios2-gdb-server ←
```

### nios2-gdb-server Usage Example

The Nios II IDE and most of the other available debuggers use the `nios2-gdb-server` and `nios2-elf-gdb` commands for debugging. You should never have to use these tools at this low level. However, in case you prefer to do so, this section includes instructions to start a GDB debugger session using these commands, and an example GDB debugging session.

You can perform the following steps to start a GDB debugger session:

1. Open a Nios II command shell.
2. In the command shell, type the following command to start the GDB server on the machine that is connected through a JTAG interface to the Nios II system you wish to debug:

```
nios2-gdb-server --tcpport 2342 --tcpersist ←
```

If the transfer control protocol port 2342 is already in use, use a different port.

Following is the system response:

```
Using cable "USB-Blaster [USB-0]", device 1, instance 0x00  
Pausing target processor: OK  
Listening on port 2342 for connection from GDB:
```

Now you can connect to your server (locally or remotely) and start debugging.

3. Type the following command to start a GDB client that targets your `.elf` file:  
`nios2-elf-gdb <file>.elf ←`

Example 4-5 shows a sample session.

#### Example 4-5. Sample Debugging Session

---

```
GNU gdb 6.1
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "--host=i686-pc-cygwin --target=nios2-elf"...
(gdb) target remote <your_host>:2342
Remote debugging using <your_host>:2342
OS_TaskIdle (p_arg=0x0) at sys/alt_irq.h:127
127 {
(gdb) load
Loading section .exceptions, size 0x1b0 lma 0x1000020
Loading section .text, size 0x3e4f4 lma 0x10001d0
Loading section .rodata, size 0x4328 lma 0x103e6c4
Loading section .rwddata, size 0x2020 lma 0x10429ec
Start address 0x10001d0, load size 281068
Transfer rate: 562136 bits/sec, 510 bytes/write.
(gdb) step
.
.
.
(gdb) quit
```

---

Possible commands include the standard debugger commands `load`, `step`, `continue`, `run`, and `quit`. Press `Ctrl+c` to terminate your GDB server session.

## nios2-debug

This command is a wrapper around the Tcl/Tk-based Insight GDB GUI, which installs with the Altera-specific GNU GDB distribution that is part of the Nios II EDS.

The command-line option `-save-gdb-script` saves the session script, and the option `-command=<GDB script name>` restores a previous GDB session by executing its previously saved GDB script. Use this option to restore break and watch points.



For more information about the Insight GDB GUI, refer to the Insight documentation available at [sources.redhat.com](http://sources.redhat.com).

### nios2-debug Usage Example

After you generate the `.elf` file manually or using the Nios II IDE, perform the following steps to open an Insight debugger session:

1. Open a Nios II command shell.
2. Change to the directory in which your `.elf` file is located.

If you use the Nios II IDE for development, the correct location is often the **Debug** or **Release** subdirectory of your top-level project.

3. In the command shell, type the following command:

```
nios2-debug <file>.elf ←
```

Your `.elf` file is parsed and downloaded to memory in your Nios II subsystem, and the main debugger window opens, with the first executable line in the `main()` function highlighted. This debugger window displays your Insight debugging session. Simply click on the **Continue** menu item to run your code, or set some breakpoints to experiment.

## Altera Command-Line Nios II Software Build Tools

The Nios II software build tools are command-line utilities available from a Nios II command shell that enable you to create application, board support package (BSP), and library software for a particular Nios II hardware system. Use these tools to create a portable, self-contained makefile-based project that can be easily modified later to suit your build flow.

Unlike the Nios II IDE-based flow, proficient use of these tools requires some expertise with the GNU make-based software build flow. Before you use these tools, refer to the *Introduction to the Nios II Software Build Tools* and the *Using the Nios II Software Build Tools* chapters of the *Nios II Software Developer's Handbook*. The `software_examples` directory for each current Nios II development board contains examples that use the GNU make-based software build flow. The examples for your development board are located in the following location:

```
$SOPC_KIT_NIOS2/examples/[verilog|vhdl]/<dev_board>/  
<design>/software_examples
```

The following sections summarize the commands available for generating a BSP for your hardware design and for generating your application software. Many additional options are available in the Nios II software build tools.



For an overview of the tools summarized in this section, refer to the *Introduction to the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.



For information on the many additional options available to you in the Nios II software build tools, refer to the *Introduction to the Nios II Software Build Tools*, *Using the Nios II Software Build Tools*, and *Nios II Software Build Tools Reference* chapters of the *Nios II Software Developer's Handbook*, and the *Developing Nios II Software* chapter of the *Embedded Design Handbook*.

## BSP Related Tools

Use the following command-line tools to create a BSP for your hardware design:

- `nios2-bsp-create-settings` creates a BSP settings file.
- `nios2-bsp-update-settings` updates a BSP settings file.
- `nios2-bsp-query-settings` queries an existing BSP settings file.
- `nios2-bsp-generate-files` generates all the files related to a given BSP settings file.

- `nios2-bsp` is a script that includes most of the functionality of the preceding commands.
- `create-this-bsp` is a high-level script that creates a BSP for a specific hardware design example.

## Application Related Tools

Use the following commands to create and manipulate Nios II application and library projects:

- `nios2-app-generate-makefile` creates a makefile for your application.
- `nios2-lib-generate-makefile` creates a makefile for your application library.
- `nios2-c2h-generate-makefile` creates a makefile fragment for the C2H compiler.
- `create-this-app` is a high-level script that creates an application for a specific hardware design example.

## GNU Command-Line Tools

The Nios II GCC toolchain contains the GNU Compiler Collection, the GNU binutils, and the newlib C library. You can follow links to detailed documentation from the Nios II EDS documentation launchpad found in your Nios II EDS distribution. To start the launchpad on Windows platforms, on the Start menu, click **All Programs**. On the All Programs menu, on the Altera submenu, on the Nios II EDS *<version>* submenu, click **Literature**. On Linux platforms, run the program in the file `$(SOPC_KIT_NIOS2)/documents/index.htm`. In addition, more information about the GNU GCC toolchain is available on the World Wide Web.

### nios2-elf-addr2line

This command returns a source code line number for a specific memory address. The command is similar to but more specific than the `nios2-elf-objdump` command described in “[nios2-elf-objdump](#)” on page 4-21 and the `nios2-elf-nm` command described in “[nios2-elf-nm](#)” on page 4-20.

Use the `nios2-elf-addr2line` command to help validate code that should be stored at specific memory addresses. [Example 4-6](#) illustrates its usage and results:

#### Example 4-6. nios2-elf-addr2line Utility Usage Example

---

```
[SOPC Builder]$ nios2-elf-addr2line --exe=<your project>.elf 0x1000020  
${SOPC_KIT_NIOS2}/components/altera_nios2/HAL/src/alt_exception_entry.S:99
```

---

### nios2-elf-addr2line Usage Example

To use the `nios2-elf-addr2line` command, perform the following steps:

1. Open a Nios II command shell.
2. In the command shell, type the following command:

```
nios2-elf-addr2line <your project>.elf <your_address_0>,\
<your_address_1>,...,<your_address_n> ↵
```

If your project file contains source code at this address, its line number appears.

### nios2-elf-gdb

This command is a GDB client that provides a simple shell interface, with built-in commands and scripting capability. A typical use of this command is illustrated in the section “[nios2-gdb-server](#)” on page 4-11.

### nios2-elf-readelf

Use this command to parse information from your project's `.elf` file. The command is useful when used with `grep`, `sed`, or `awk` to extract specific information from your `.elf` file.

#### nios2-elf-readelf Usage Example

To display information about all instances of a specific function name in your `.elf` file, perform the following steps:

1. Open a Nios II command shell.
2. In the command shell, type the following command:

```
nios2-elf-readelf -symbols <project>.elf | grep <function name> ↵
```

[Example 4-7](#) shows a search for the `http_read_line()` function in a `.elf` file.

#### Example 4-7. Search for the `http_read_line` Function Using `nios2-elf-readelf`

---

```
[SOPC Builder]$ nios2-elf-readelf.exe -s my_file.elf | grep http_read_line
1106: 01001168 160 FUNC GLOBAL DEFAULT 3 http_read_line
```

---

[Table 4-2](#) lists the meanings of the individual columns in [Example 4-7](#).

**Table 4-2.** Interpretation of `nios2-elf-readelf` Command Response

Value	Description
1106	Symbol instance number
01001168	Memory address, in hexadecimal format
160	Size of this symbol, in bytes
FUNC	Type of this symbol (function)
GLOBAL	Binding (values: GLOBAL, LOCAL, and WEAK)
DEFAULT	Visibility (values: DEFAULT, INTERNAL, HIDDEN, and PROTECTED)
3	Index
http_read_line	Symbol name

You can obtain further information about the ELF file format online. Each of the ELF utilities has its own man page.

## nios2-elf-ar

This command generates an archive (.a) file containing a library of object (.o) files. The Nios II IDE uses this command to archive the System Library project.

### nios2-elf-ar Usage Example

To archive your object files using the `nios2-elf-ar` command, perform the following steps:

1. Open a Nios II command shell.
2. Change to the directory in which your object files are located.
3. In the command shell, type the following command:

```
nios2-elf-ar q <archive_name>.a <object files>
```

**Example 4-8** shows how to create an archive of all of the object files in your current directory. In **Example 4-8**, the `q` option directs the command to append each object file it finds to the end of the archive. After the archive file is created, it can be distributed for others to use, and included as an argument in linker commands, in place of a long object file list.

### Example 4-8. nios2-elf-ar Command Response

---

```
[SOPC Builder]$ nios2-elf-ar q <archive_name>.a *.o
nios2-elf-ar: creating <archive_name>.a
```

---

## Linker

Use the `nios2-elf-g++` command to link your object files and archives into the final executable format, ELF.

### Linker Usage Example

To link your object files and archives into a .elf file, open a Nios II command shell and call `nios2-elf-g++` with appropriate arguments. The following example command line calls the linker:

```
nios2-elf-g++ -T'<linker script>' -msys-crt0='<crt0.o file>' \
-msys-lib=<system library> -L'<The path where your libraries reside>' \
-DALT_DEBUG -O0 -g -Wall -mhw-mul -mhw-mulx -mno-hw-div \
-o <your project>.elf <object files> -lm ↵
```

The exact linker command line to link your executable may differ. When you build a project in the Nios II IDE, you can see the command line used to link your application. To turn on this option in the Nios II IDE, on the Window menu, click **Preferences**, select the **Nios II** tab, and enable **Show command lines when running make**. You can also force the command lines to display by running `make` without the `-s` option from a Nios II command shell.



Altera recommends that you not use the native linker `nios2-elf-ld` to link your programs. For the Nios II processor, as for all softcore processors, the linking flow is complex. The `g++` (`nios2-elf-g++`) command options simplify this flow. Most of the options are specified by the `-m` command-line option, but the options available depend on the processor choices you make.

## nios2-elf-size

This command displays the total size of your program and its basic code sections.

### nios2-elf-size Usage Example

To display the size information for your program, perform the following steps:

1. Open a Nios II command shell.
2. Change to the directory in which your `.elf` file is located.
3. In the command shell, type the following command:  
`nios2-elf-size <project>.elf`

[Example 4-9](#) shows the size information this command provides.

### Example 4-9. nios2-elf-size Command Usage

---

```
[SOPC Builder]$ nios2-elf-size my_project.elf
text    data    bss     dec     hex filename
272904  8224 6183420 6464548 62a424 my_project.elf
```

---

## nios2-elf-strings

This command displays all the strings in a `.elf` file.

### nios2-elf-strings Usage Example

The command has a single required argument:

```
nios2-elf-strings <project>.elf
```

## nios2-elf-strip

This command strips all symbols from object files. All object files are supported, including ELF files, object files (`.o`) and archive files (`.a`).

### nios2-elf-strip Usage Example

```
nios2-elf-strip <options> <project>.elf
```

### nios2-elf-strip Usage Notes

The `nios2-elf-strip` command decreases the size of the `.elf` file.

This command is useful only if the Nios II processor is running an operating system that supports ELF natively. If ELF is the native executable format, the entire `.elf` file is stored in memory, and the size savings matter. If not, the file is parsed and the instructions and data stored directly in memory, without the symbols in any case.

Linux is one operating system that supports ELF natively; uClinux is another. uClinux uses the flat (FLT) executable format, which is translated directly from the ELF.

## nios2-elf-gdbtui

This command starts a GDB session in which a terminal displays source code next to the typical GDB console.

The syntax for the `nios2-elf-gdbtui` command is identical to that for the `nios2-elf-gdb` command described in “[nios2-elf-gdb](#)” on page 4-15.



Two additional GDB user interfaces are available for use with the Nios II GDB Debugger. CGDB, a cursor-based GDB UI, is available at [www.sourceforge.net](http://www.sourceforge.net). The Data Display Debugger (DDD) is highly recommended.

## nios2-elf-gprof

This command allows you to profile your Nios II system.



For details about this command and the Nios II IDE-based results GUI, refer to [AN 391: Profiling Nios II Systems](#).

## nios2-elf-insight

The `nios2-debug` command described in “[nios2-debug](#)” on page 4-12 uses this command to start an Insight debugger session on the supplied `.elf` file.

## nios2-elf-gcc and g++

These commands run the GNU C and C++ compiler, respectively, for the Nios II processor.

### Compilation Command Usage Example

The following simple example shows a command line that runs the GNU C or C++ compiler:

```
nios2-elf-gcc(g++) <options> -o <object files> <C files>
```

## More Complex Compilation Example

Example 4-10 is a Nios II IDE-generated command line that compiles C code in multiple files in many directories.

### Example 4-10. Example nios2-elf-gcc Command Line

```
nios2-elf-gcc -xc -MD -c \  
-DSYSTEM_BUS_WIDTH=32 -DALT_NO_C_PLUS_PLUS -DALT_NO_INSTRUCTION_EMULATION \  
-DALT_USE_SMALL_DRIVERS -DALT_USE_DIRECT_DRIVERS -DALT_PROVIDE_GMON \  
-I.. -I/cygdrive/c/Work/Projects/demo_reg32/Designs/std_2s60_ES/software/  
reg_32_example_0_syslib/Release/system_description \  
-I/cygdrive/c/altera/70_b31/ip/sopc_builder_ip/altera_avalon_timer/HAL/inc \  
-I/cygdrive/c/altera/70_b31/ip/sopc_builder_ip/altera_avalon_timer/inc \  
-I/cygdrive/c/altera/70_b31/ip/sopc_builder_ip/altera_avalon_jtag_uart/HAL/inc \  
-I/cygdrive/c/altera/70_b31/ip/sopc_builder_ip/altera_avalon_jtag_uart/inc \  
-I/cygdrive/c/altera/70_b31/ip/sopc_builder_ip/altera_avalon_pio/inc \  
-I/cygdrive/c/altera/70_b31/ip/sopc_builder_ip/altera_avalon_lcd_16207/HAL/inc \  
-I/cygdrive/c/altera/70_b31/ip/sopc_builder_ip/altera_avalon_lcd_16207/inc \  
-I/cygdrive/c/altera/70_b31/ip/sopc_builder_ip/altera_avalon_sysid/HAL/inc \  
-I/cygdrive/c/altera/70_b31/ip/sopc_builder_ip/altera_avalon_sysid/inc \  
-I/cygdrive/c/altera/70_b31/nios2eds/components/altera_nios2/HAL/inc \  
-I/cygdrive/c/altera/70_b31/nios2eds/components/altera_hal/HAL/inc \  
-DALT_SINGLE_THREADED -D__hal__ -pipe -DALT_RELEASE -O2 -g -Wall\  
-mhw-mul -mhw-mulx -mno-hw-div -o obj/reg_32_buttons.o ../reg_32_buttons.c
```

## nios2-elf-c++filt

This command demangles C++ mangled names. C++ allows multiple functions to have the same name if their parameter lists differ; to keep track of each unique function, the compiler mangles, or decorates, function names. Each compiler mangles functions in a particular way.



For a full explanation, including more details about how the different compilers mangle C++ function names, refer to standard reference sources for the C++ language compilers.

### nios2-elf-c++filt Usage Example

To display the original, demangled function name that corresponds to a particular symbol name, you can type the following command:

```
nios2-elf-c++filt -n <symbol name> ←
```

For example,

```
nios2-elf-c++filt -n _Z11my_functionv ←
```

### More Complex nios2-elf-c++filt Example

The following example command line causes the display of all demangled function names in an entire file:

```
nios2-elf-strings <file>.elf | grep ^_Z | nios2-elf-c++filt -n
```

In this example, the `nios2-elf-strings` operation outputs all strings in the `.elf` file. This output is piped to a `grep` operation that identifies all strings beginning with `_Z`. (GCC always prepends mangled function names with `_Z`). The output of the `grep` command is piped to a `nios2-elf-c++filt` command. The result is a list of all demangled functions in a GCC C++ `.elf` file.

## nios2-elf-nm

This command lists the symbols in a `.elf` file.

### nios2-elf-nm Usage Example

The following two simple examples illustrate the use of the `nios2-elf-nm` command:

- `nios2-elf-nm <project>.elf` ←
- `nios2-elf-nm <project>.elf | sort -n` ←

### More Complex nios2-elf-nm Example

To generate a list of symbols from your `.elf` file in ascending address order, use the following command:

```
nios2-elf-nm <project>.elf | sort -n > <project>.elf.nm
```

The `<project>.elf.nm` file contains all of the symbols in your executable file, listed in ascending address order. In this example, the `nios2-elf-nm` command creates the symbol list. In this text list, each symbol's address is the first field in a new line. The `-n` option for the `sort` command specifies that the symbols be sorted by address in numerical order instead of the default alphabetical order.

## nios2-elf-objcopy

Use this command to copy from one binary object format to another, optionally changing the binary data in the process.

Though typical usage converts from or to ELF files, the `objcopy` command is not restricted to conversions from or to ELF files. You can use this command to convert from, and to, any of the formats listed in [Table 4-3](#).

**Table 4-3.** `-objcopy` Binary Formats

Command (...-objcopy)	Comments
elf32-littlenios2, elf32-little	Header little endian, data little endian, the default and most commonly used format
elf32-bignios2, elf32-big	Header big endian, data big endian
srec	S-Record (SREC) output format
symbolsrec	SREC format with all symbols listed in the file header, preceding the SREC data
tekhex	Tektronix hexadecimal (TekHex) format
binary	Raw binary format Useful for creating binary images for storage in flash on your embedded system
ihex	Intel hexadecimal (ihex) format



You can obtain information about the TekHex, `ihex`, and other text-based binary representation file formats on the World Wide Web. As of the initial publication of this handbook, you can refer to the [www.sbprojects.com](http://www.sbprojects.com) knowledge-base entry on file formats.

### **nios2-elf-objcopy Usage Example**

To create an SREC file from an ELF file, use the following command:

```
nios2-elf-objcopy -O srec <project>.elf <project>.srec
```

ELF is the assumed binary format if none is listed. For information about how to specify a different binary format, in a Nios II command shell, type the following command:

```
nios2-elf-objcopy --help ←
```

### **nios2-elf-objdump**

Use this command to display information about the object file, usually an ELF file.

The `nios2-elf-objdump` command supports all of the binary formats that the `nios2-elf-objcopy` command supports, but ELF is the only format that produces useful output for all command-line options.

### **nios2-elf-objdump Usage Description**

The Nios II IDE uses the following command line to generate object dump files:

```
nios2-elf-objdump -D -S -x <project>.elf > <project>.elf.objdump
```

### **nios2-elf-ranlib**

Calling `nios2-elf-ranlib` is equivalent to calling `nios2-elf-ar` with the `-s` option (`nios2-elf-ar -s`).

For further information about this command, refer to [“nios2-elf-ar” on page 4-16](#) or type `nios2-elf-ar --help` in a Nios II command shell.

## **Referenced Documents**

This chapter references the following documents:

- [AN 391: Profiling Nios II Systems](#)
- [Developing Nios II Software](#) chapter of the *Embedded Design Handbook*
- [Introduction to the Nios II Software Build Tools](#) chapter of the *Nios II Software Developer's Handbook*
- [Nios II Flash Programmer User Guide](#)
- [Nios II Software Build Tools Reference](#) chapter of the *Nios II Software Developer's Handbook*
- [Nios II Software Developer's Handbook](#)
- [Overview](#) chapter of the *Nios II Software Developer's Guide*
- [System Console User Guide](#)
- [Using the Nios II Software Build Tools](#) chapter of the *Nios II Software Developer's Handbook*
- [Verification and Board Bring-Up](#) chapter of the *Embedded Design Handbook*

## Document Revision History

Table 4-4 shows the revision history for this chapter.

**Table 4-4.** Document Revision History

<b>Date and Document Version</b>	<b>Changes Made</b>	<b>Summary of Changes</b>
April 2009 v2.1	Fix outdated reference.	Fix outdated reference.
November 2008 v2.0	Add System Console.	Add System Console.
March 2008 v1.0	Initial release.	—

## Introduction


The Nios® II C2H Compiler is a powerful tool that generates hardware accelerators for software functions. The C2H Compiler enhances design productivity by allowing you to use a compiler to accelerate software algorithms in hardware. You can quickly prototype hardware functional changes in C, and explore hardware-software design tradeoffs in an efficient, iterative process. The C2H Compiler is well suited to improving computational bandwidth, as well as memory throughput. It is possible to achieve substantial performance gains with minimal engineering effort.

The structure of your C code affects the results you get from the C2H Compiler. Although the C2H Compiler can accelerate most ANSI C code, you might need to modify your C code to meet resource usage and performance requirements. This document describes how to improve the performance of hardware accelerators, by refactoring your C code with C2H-specific optimizations.

## Prerequisites

To make effective use of this chapter, you should be familiar with the following topics:

- ANSI C syntax and usage
- Defining and generating Nios II hardware systems with SOPC Builder
- Compiling Nios II hardware systems with the Altera® Quartus® II development software
- Creating, compiling, and running Nios II software projects
- Nios II C2H Compiler theory of operation
- Data caching

 To familiarize yourself with the basics of the C2H Compiler, refer to the *Nios II C2H Compiler User Guide*, especially the *Introduction to the C2H Compiler* and *Getting Started Tutorial* chapters. To learn about defining, generating, and compiling Nios II systems, refer to the *Nios II Hardware Development Tutorial*. To learn about Nios II software projects, refer to the *Nios II Software Development Tutorial*, available in the Nios II IDE help system. To learn about data caching, refer to the *Cache and Tightly-Coupled Memory* in the *Nios II Processor Reference Handbook*.

## Cost and Performance

When writing C code for the C2H Compiler, you can optimize it relative to several optimization criteria. Often you must make tradeoffs between these criteria, which are listed below:

- **Hardware cost**—C2H accelerators consume hardware resources such as LEs, multipliers, and on-chip memory. This document uses the following terms to describe the hardware cost of C language constructs:
  - **Free**—the construct consumes no hardware resources.
  - **Cheap**—the construct consumes few hardware resources. The acceleration obtained is almost always worth the cost.
  - **Moderate**—the construct consumes some hardware resources. The acceleration obtained is usually worth the cost.
  - **Expensive**—the construct consumes substantial hardware resources. The acceleration obtained is sometimes worth the cost, depending on the nature of the application.
- **Algorithm performance**—A C2H accelerator performs the same algorithm as the original C software executed by a Nios II processor. Typically the accelerator uses many fewer clock cycles than the software implementation. This document describes the algorithm performance of C constructs as fast or slow. The concept of algorithm performance includes the concepts of latency and throughput. These concepts are defined under [“Cycles Per Loop Iteration \(CPLI\)” on page 5–12](#).
- **Hardware performance impact**—Certain C language constructs, when converted to logic by the C2H Compiler, can result in long timing paths that can degrade  $f_{MAX}$  for the entire system or for the clock domain containing the C2H accelerator. This document clearly notes such situations and offers strategies for avoiding them.

## Overview of the C2H Optimization Process

It is unlikely that you can meet all of your optimization goals in one iteration. Instead, plan on making the one or two optimizations that appear most relevant to your cost and performance issues. When you profile your system with the optimized accelerator, you can determine whether further optimizations are needed, and then you can identify the next most important optimization issue to address. By optimizing your accelerator one step at a time, you apply only the optimizations needed to achieve your goals.

### Getting Started

The most important first step is to decide on a clear performance goal. Depending on your application, you may require a specific performance level from your algorithm. If you have already selected a target device, and if other hardware in the system is well defined, you might have specific hardware cost limitations. Alternatively, if you are in early phases of development, you might only have some general guidelines for conserving hardware resources. Finally, depending on your design needs and the  $f_{MAX}$  of your existing design, you might be concerned with possible  $f_{MAX}$  degradation. Refer to [“Meeting Your Cost and Performance Goals” on page 5–3](#) for more information about cost and performance criteria.

The next step is to develop your algorithm in C, and, if possible, test it conventionally on the Nios II processor. This step is very helpful in establishing and maintaining correct functionality. If the Nios II processor is not fast enough for in-circuit testing of your unaccelerated algorithm, consider simulation options for testing.


When you are confident of your algorithm's correctness, you are ready to accelerate it. This first attempt provides a set of baseline acceleration metrics. These metrics help you assess the overall success of the optimization process.

Altera recommends that you maintain two copies of your algorithm in parallel: one accelerated and the other unaccelerated. By comparing the results of the accelerated and unaccelerated algorithms, you immediately discover any errors which you might inadvertently introduce while optimizing the code.

## Iterative Optimization

The iteration phase of C2H Compiler optimization consists of these steps:

1. Profile your accelerated system.
2. Identify the most serious performance bottleneck.
3. Identify an appropriate optimization from the “[Optimization Techniques](#)” on [page 5–14](#) section.
4. Apply the optimization and rebuild the accelerated system.

 For instructions on profiling Nios II systems, refer to [AN391: Profiling Nios II Systems](#).

## Meeting Your Cost and Performance Goals

Having a clear set of optimization goals helps you determine when to stop optimization. Each time you profile your accelerated system, compare the results with your goals. You might find that you have reached your cost and performance goals even if you have not yet applied all relevant optimizations.

If your optimization goals are flexible, consider keeping track of your baseline acceleration metrics, and the acceleration metrics achieved at each optimization step. You might wish to stop if you reach a point of diminishing returns.

## Factors Affecting C2H Results

This section describes key differences in the mapping of C constructs by a C compiler and the Nios II C2H Compiler. You must understand these differences to create efficient hardware.

C code originally written to run on a processor does not necessarily produce efficient hardware. A C compiler and the Nios II C2H Compiler both use hardware resources such as adders, multipliers, registers, and memories to execute the C code. However, while a C compiler assumes a sequential model of computing, the C2H Compiler assumes a concurrent model of computing. A C compiler maps C code to instructions which access shared hardware resources. The C2H Compiler maps C code to one or more state machines which access unique hardware resources. The C2H Compiler pipelines the computation as much as possible to increase data throughput.

[Example 5–1](#) illustrates this point.

**Example 5-1.** Pipelined Computation

```
int sumfunc(int a, int b, int c, int d)
{
int sum1 = a + b;
int sum2 = c + d;
int result = sum1 + sum2;
return result;
}
```

The `sumfunc()` function takes four integer arguments and returns their sum. A C compiler maps the function to three add instructions sharing one adder. The processor executes the three additions sequentially. The C2H Compiler maps the function to one state machine and three adders. The accelerator executes the additions for `sum1` and `sum2` concurrently, followed by the addition for `result`. The addition for `result` cannot execute concurrently with the `sum1` and `sum2` additions because of the data dependency on the `sum1` and `sum2` variables.

Different algorithms require different C structures for optimal hardware transformation. This chapter lists possible optimizations to identify in C code. Each C scenario describes the best methods to refactor the C code. The “[Optimization Techniques](#)” section discusses how to address the following potential problem areas:

- [Memory Accesses and Variables](#)
- [Arithmetic and Logical Operations](#)
- [Statements](#)
- [Control Flow](#)
- [Subfunction Calls](#)
- [Resource Sharing](#)
- [Data Dependencies](#)
- [Memory Architecture](#)

## Memory Accesses and Variables

Memory accesses can occur when your C code reads or writes the value of a variable. [Table 5-1](#) provides a summary of the key differences in the mapping of memory accesses between a C compiler and the C2H Compiler.

A C compiler generally allocates many types of variables in your data memory. These include scalars, arrays, and structures that are local, static, or global. When allocated in memory, variables are relatively cheap due to the low cost per bit of memory (especially external memory) and relatively slow due to the overhead of load or store instructions used to access them. In some situations, a C compiler is able to use processor registers for local variables. When allocated in processor registers, these variables are relatively fast and expensive.

The C2H Compiler allocates local scalar variables in registers implemented with logic elements (LEs), which have a moderate cost and are fast.

A C compiler maps pointer dereferences and array accesses to a small number of instructions to perform the address calculation and access to your data memory. Pointer dereferences and array accesses are relatively cheap and slow.

The C2H Compiler maps pointer dereferences and array accesses to a small amount of logic to perform the address calculation and creates a unique Avalon® Memory-Mapped (Avalon-MM) master port to access the addressed memory. This mapping is expensive due to the logic required to create an Avalon-MM master port. It is slow or fast depending on the type of memory connected to the port. Local arrays are fast because the C2H Compiler implements them as on-chip memories.

**Table 5–1.** Memory Accesses

C Construct	C Compiler Implementation	C2H Implementation
Local scalar variables	Allocated in memory (cheap, slow) or allocated in processor registers (expensive, fast)	Allocated in registers based on logic elements (LEs) (moderate cost, fast)
Uninitialized local array variables	Allocated in memory (cheap, slow)	Allocated in on-chip memory. (expensive, fast)
Initialized local array variables	Allocated in memory (cheap, slow)	Allocated in memory (cheap, slow)
All other types of variables	Allocated in memory (cheap, slow)	Allocated in memory (cheap, slow)
Pointer dereferences and nonlocal array accesses	Access normal data memory (cheap, slow)	Avalon-MM master port (expensive, slow or fast)

## Arithmetic and Logical Operations

Table 5–2 provides a summary of the key differences in the mapping of arithmetic and logical operations between a C compiler and the C2H Compiler.

A C compiler maps arithmetic and logical operations into one or more instructions. In many cases, it can map them to one instruction. In other cases, it might need to call a function to implement the operation. An example of the latter occurs when a Nios II processor that does not have a hardware multiplier or divider performs a multiply operation.

The C2H Compiler implements the following logical operations simply as wires without consuming any logic at all.

- Shifts by a constant
- Multiplies and divides by a power of two constant
- Bitwise ANDs and ORs by a constant

As a result, these operations are fast and free. The following is an example of one of these operations:

```
int result = some_int >> 2;
```

A C compiler maps this statement to a right shift instruction. The C2H Compiler maps the statement to wires that perform the shift.

**Table 5-2.** Arithmetic and Logical Operations

C Construct	C Compiler Implementation	C2H Implementation
Shift by constant or multiply or divide by power of 2 constant. (1) Example: $y = x/2;$	Shift instruction (cheap, fast)	Wires (free, fast)
Shift by variable Example: $y = x \gg z;$	Shift instruction (cheap, fast)	Barrel shifter (expensive, fast)
Multiply by a value that is not a power of 2 (constant or variable) Example: $y = x \times z;$	Multiply operation (cheap, slow)	If the Quartus II software can produce an optimized multiply circuit (cheap, fast); otherwise a multiply circuit (expensive, fast)
Divide by a value that is not a power of 2 (constant or variable) Example: $y = x/z;$	Divide operation (cheap, slow)	Divider circuit (expensive, slow)
Bitwise AND or bitwise OR with constant Example: $y = x \mid 0xFFFF;$	AND or OR instruction (cheap, fast)	Wires (free, fast)
Bitwise AND or bitwise OR with variable Example: $y = x \& z;$	AND or OR instruction (cheap, fast)	Logic (cheap, fast)

**Notes to Table 5-2:**

(1) Dividing by a *negative* power of 2 is expensive.

**Statements**

A C compiler maps long expressions (those with many operators) to instructions. The C2H Compiler maps long expressions to logic which could create a long timing path. The following is an example of a long expression:

```
int sum = a + b + c + d + e + f + g + h;
```

A C compiler creates a series of add instructions to compute the result. The C2H Compiler creates several adders chained together. The resulting computation has a throughput of one data transfer per clock cycle and a latency of one cycle.

A C compiler maps a large function to a large number of instructions. The C2H Compiler maps a large function to a large amount of logic which is expensive and potentially degrades  $f_{MAX}$ . If possible, remove from the function any C code that does not have to be accelerated.

A C compiler maps mutually exclusive, multiple assignments to a local variable as store instructions or processor register writes, which are both relatively cheap and fast. However, the C2H Compiler creates logic to multiplex between the possible assignments to the selected variable. [Example 5-2](#) illustrates such a case.

**Example 5-2.** Multiple Assignments to a Single Variable

```
int result;
if (a > 100)
{
result = b;
}
else if (a > 10)
{
result = c;
}
else if (a > 1)
{
result = d;
}
```

A C compiler maps this C code to a series of conditional branch instructions and associated expression evaluation instructions. The C2H Compiler maps this C code to logic to evaluate the conditions and a three-input multiplexer to assign the correct value to `result`. Each assignment to `result` adds another input to the multiplexer. The assignments increase the amount of the logic, and might create a long timing path. [Table 5-3](#) summarizes the key differences between the C compiler and C2H Compiler in handling C constructs.

**Table 5-3.** Statements

C Construct	C Compiler Implementation	C2H Implementation
Long expressions	Several instructions (cheap, slow)	Logic (cheap, degrades $f_{MAX}$ )
Large functions	Many instructions (cheap, slow)	Logic (expensive, degrades $f_{MAX}$ )
Multiple assignments to a local variable	Store instructions or processor register writes (cheap, fast)	Logic (cheap, degrades $f_{MAX}$ )

**Control Flow**

[Table 5-4](#) provides a summary of the differences in the mapping of control flow between a C compiler and the C2H Compiler.

**If Statements**

The C2H compiler maps the expression of the `if` statement to control logic. The statement is controlled by the expression portion of the `if` statement.

## Loops

Loops include `for` loops, `do` loops, and `while` loops. A C compiler and the C2H Compiler both treat the expression evaluation part of a loop just like the expression evaluation in an `if` statement. However, the C2H Compiler attempts to pipeline each loop iteration to achieve a throughput of one iteration per cycle. Often there is no overhead for each loop iteration in the C2H accelerator, because it executes the loop control concurrently with the body of the loop. The data and control paths pipelining allows the control path to control the data path. If the control path (loop expression) is dependent on a variable calculated within the loop, the throughput decreases because the data path must complete before control path can allow another loop iteration.

The expression, `while (++a < 10) { b++ };` runs every cycle because there is no data dependency. On the other hand, `while (a < 10) { a++ };` takes 2 cycles to run because the value of `<a>` is calculated in the loop.

A C compiler maps `switch` statements to the equivalent `if` statements or possibly to a jump table. The C2H Compiler maps `switch` statements to the equivalent `if` statements.

**Table 5-4.** Control Flow

C Construct	C Compiler Implementation	C2H Implementation
<code>if</code> statements	A few instructions (cheap, slow)	Logic (cheap, fast)
Loops	A few instructions of overhead per loop iteration (cheap, slow)	Logic (moderate, fast)
Switch statements	A few instructions (cheap, slow)	Logic (moderate, fast)
Ternary operation	A few instructions (cheap, slow)	Logic (cheap, fast)

## Subfunction Calls

A C compiler maps subfunction calls to a few instructions to pass arguments to or from the subfunction and a few instructions to call the subfunction. A C compiler might also convert the subfunction into inline code. The C2H Compiler maps a subfunction call made in your top-level accelerated function into a new accelerator. This technique is expensive, and stalls the pipeline in the top-level accelerated function. It might result in a severe performance degradation.

However, if the subfunction has a fixed, deterministic execution time, the outer function attempts to pipeline the subfunction call, avoiding the performance degradation. In [Example 5-3](#), the subfunction call is pipelined.

---

**Example 5-3.** Pipeline Stall

---

```
int abs(int a)
{
return (a < 0) ? -a : a;
}
int abs_sum(int* arr, int num_elements)
{
int i;
int result = 0;
for (i = 0; i < num_elements; i++)
{
result += abs(*arr++);
}
return result;
}
```

---

## Resource Sharing

By default, the C2H Compiler creates unique instances of hardware resources for each operation encountered in your C code. If this translation consumes too many resources, you can change your C code to share resources. One mechanism to share resources is to use shared subfunctions in your C code. Simply place the code to be shared in a subfunction and call it from your main accelerated function. The C2H Compiler creates only one instance of the hardware in the function, shared by all function callers.

[Example 5-4](#) uses a subfunction to share one multiplier between two multiplication operations.

---

**Example 5-4.** Shared Multiplier

---

```
int mul2(int x, int y)
{
return x * y;
}
int muladd(int a, int b, int c, int d)
{
int prod1 = mul2(a, b);
int prod2 = mul2(c, d);
int result = prod1 + prod2;
return result;
}
```

---

## Data Dependencies

A data dependency occurs when your C code has variables whose values are dependent on the values of other variables. Data dependency prevents a C compiler from performing some optimizations which typically result in minor performance degradation. When the C2H Compiler maps code to hardware, a data dependency causes it to schedule operations sequentially instead of concurrently, which can cause a dramatic performance degradation.

The algorithm in [Example 5-5](#) shows data dependency.

---

**Example 5-5.** Data Dependency

---

```
int sum3(int a, int b, int c)
{
int sum1 = a + b;
int result = sum1 + c;
return result;
}
```

---

The C2H Compiler schedules the additions for `sum1` and `result` sequentially due to the dependency on `sum1`.

## Memory Architecture

The types of memory and how they are connected to your system, including the C2H accelerator, define the memory system architecture. For many algorithms, appropriate memory architecture is critical to achieving high performance with the C2H Compiler. With an inappropriate memory architecture, an accelerated algorithm can perform more poorly than the same algorithm running on a processor.

Due to the concurrency possible in a C2H accelerator, compute-limited algorithms might become data-limited algorithms. To achieve the highest levels of performance, carefully consider the best memory architecture for your algorithm and modify your C code accordingly to increase memory bandwidth.

For the following discussion, assume that the initial memory architecture is a processor with a data cache connected to an off-chip memory such as DDR SDRAM.

The C code in [Example 5-6](#) is data-limited when accelerated by the C2H Compiler because the `src` and `dst` dereferences both create Avalon-MM master ports that access the same Avalon-MM slave port. An Avalon-MM slave port can only handle one read or write operation at any given time; consequently, the accesses are interleaved, limiting the throughput to the memory bandwidth.

---

**Example 5-6.** Memory Bandwidth Limitation

---

```
void memcpy(char* dst, char* src, int num_bytes)
{
while (num_bytes-- > 0)
{
*dst++ = *src++;
}
}
```

---

The C2H Compiler is able to achieve a throughput of one data transfer per clock cycle if the code is modified and the appropriate memory architecture is available. The changes required to achieve this goal are covered in [“Efficiency Metrics” on page 5-11](#).

## Data Cache Coherency

When a C2H accelerator accesses memory, it uses its own Avalon-MM master port, which bypasses the Nios II data cache. Before invoking the accelerator, if the data is potentially stored in cache, the Nios II processor must write it to memory, thus avoiding the typical cache coherency problem. This cache coherency issue is found in any multimaster system that lacks support for hardware cache coherency protocols.

When you configure the C2H accelerator, you choose whether or not the Nios II processor flushes the data cache whenever it calls the accelerated function. If you enable this option, it adds to the overhead of calling the accelerator and causes the rest of the C code on the processor to temporarily run more slowly because the data cache must be reloaded.

You can avoid flushing the entire data cache. If the processor never shares data accessed by the accelerator, it does not need to flush the data cache. However, if you use memory to pass data between the processor and the accelerator, as is often the case, it might be possible to change the C code running on the processor to use uncacheable accesses to the shared data. In this case, the processor does not need to flush the data cache, but it has slower access to the shared data. Alternatively, if the size of the shared data is substantially smaller than the size of the data cache, the processor only needs to flush the shared data before calling the accelerator.

Another option is to use a processor without a data cache. Running without a cache slows down all processor accesses to memory but the acceleration provided by the C2H accelerator might be substantial enough to result in the overall fastest solution.

## DRAM Architecture

Memory architectures consisting of a single DRAM typically require modification to maximize C2H accelerator performance. One problem with the DRAM architecture is that memory performance degrades if accesses to it are nonsequential. Because the DRAM has only one port, multiple Avalon-MM master ports accessing it concurrently prevent sequential accesses by one Avalon-MM master from occurring.

The default behavior of the arbiter in an SOPC Builder system is round-robin. If the DRAM controller (such as the Altera Avalon SDRAM controller) can only keep one memory bank open at a time, the master ports experience long stalls and do not achieve high throughput. Stalls can cause the performance of any algorithm accelerated using the C2H Compiler to degrade if it accesses memory nonsequentially due to multiple master accesses or nonsequential addressing.



For additional information about optimizing memory architectures in a Nios II system, refer to the *Cache and Tightly-Coupled Memory* in the *Nios II Software Developer's Handbook*.

## Efficiency Metrics

There are several ways to measure the efficiency of a C2H accelerator. The relative importance of these metrics depends on the nature of your application. This section explains each efficiency metric in detail.

## Cycles Per Loop Iteration (CPLI)

The C2H report section contains a CPLI value for each loop in an accelerated function. The CPLI value represents the number of clock cycles each iteration of the loop takes to complete once the initial latency is overcome. The goal is to minimize the CPLI value for each loop to increase the data throughput. It is especially important that the innermost loop of the function have the lowest possible CPLI because it executes the most often.

The CPLI value does not take into account any hardware stalls that might occur. A shared resource such as memory stalls the loop if it is not available. If you nest looping structures the outer loops stall and, as a result, reduce the throughput of the outer loops even if their CPLI equals one. The [“Optimization Techniques” on page 5-14](#) section offers methods for maximizing the throughput of loops accelerated with the C2H Compiler.

Optimizations that can help CPLI are as follows:

- Reducing data dependencies
- Reducing the system interconnect fabric by using the `connect_variable` pragma

$f_{MAX}$  is the maximum frequency at which a hardware design can run. The longest register-to-register delay or critical path determines  $f_{MAX}$ . The Quartus II software reports the  $f_{MAX}$  of a design after each compilation.

Adding accelerated functions to your design can potentially affect  $f_{MAX}$  in two ways: by adding a new critical path, or by adding enough logic to the design that the Quartus II fitter fails to fit the elements of the critical path close enough to each other to maintain the path's previous delay. The optimizations that can help with  $f_{MAX}$  are as follows:

- Pipelined calculations
- Avoiding division
- Reducing system interconnect fabric by using the `connect_variable` pragma
- Reducing unnecessary memory connections to the Nios II processor

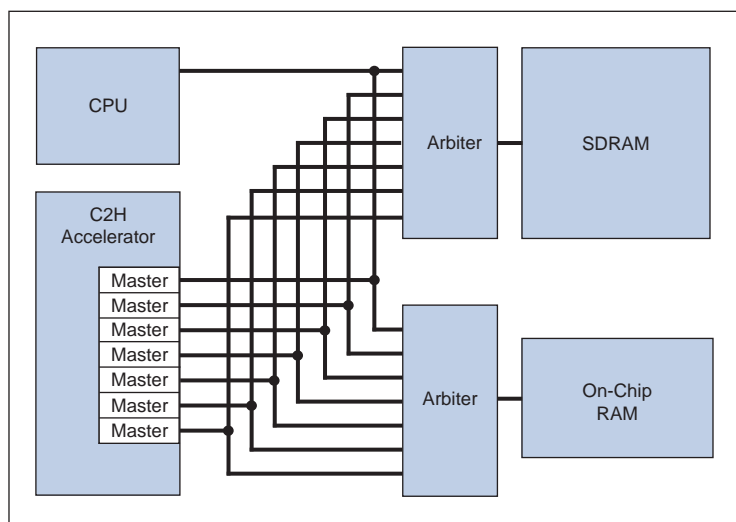
## FPGA Resource Usage

Because an accelerated function is implemented in FPGA hardware, it consumes FPGA resources such as logic elements and memory. Sometimes, an accelerator consumes more FPGA resources than is desired or expected. Unanticipated resource usage has the disadvantage of consuming resources that are needed for other logic and can also degrade system  $f_{MAX}$ .

## Avalon-MM Master Ports

The number of Avalon-MM master ports on the accelerator can heavily influence logic utilization. The C2H report, which the Nios II IDE displays after accelerating the function, reports how many Avalon-MM ports are generated. Multiple master ports can help increase the parallelization of logic when attached to separate memories. However, they have a cost in logic, and can also promote the creation of excessive arbitration logic when connected to the same memory port, as shown in [Figure 5-1](#).

**Figure 5-1.** Too Many Master Ports



## Embedded Multipliers

Multiplication logic is often available on the FPGA as dedicated hardware or created using logic elements. When you use dedicated hardware, be aware that having a large amount of multiplication logic can degrade the routing of your design because the fitter cannot place the multiplier columns to achieve a better fit. When creating multiplication logic from logic elements, be aware that this is expensive in resource usage, and can degrade  $f_{MAX}$ .

If one of the operands in a multiplication is a constant, the Quartus II software determines the most efficient implementation. [Example 5-7](#) shows a optimization the Quartus II software might make:

**Example 5-7.** Quartus II Software Optimization for Multiplication by a Constant

```
/* C code multiplication by a constant */  
c = 7 * a;  
  
/* Quartus II software optimization */  
c = (4 * a) + (2 * a) + a;
```

Because the optimized equation includes multiplications by a constant factor of 2, the Quartus II software turns them into 2 shifts plus a add.

## Embedded Memory

Embedded memory is a valuable resource for many hardware accelerators due to its high speed and fixed latency. Another benefit of embedded memory is that it can be configured with dual ports. Dual-ported memory allows two concurrent accesses to occur, potentially doubling the memory bandwidth. Whenever your code declares an uninitialized local array in an accelerated function, the C2H Compiler instantiates embedded memory to hold its contents. Use embedded memory only when it is appropriate; do not waste it on operations that do not benefit from its high performance.

Optimization tips that can help reduce FPGA resource use are:

- Using wide memory accesses
- Keeping loops rolled up
- Using narrow local variables

## Data Throughput

Data throughput for accelerated functions is difficult to quantify. The Altera development tools do not report any value that directly corresponds to data throughput. The only true data throughput metrics reported are the number of clock cycles and the average number of clock cycles it takes for the accelerated function to complete. One method of measuring the data throughput is to use the amount of data processed and divide by the amount of time required to do so. You can use the Nios II processor to measure the amount of time the accelerator spends processing data to create an accurate measurement of the accelerator throughput.

Before accelerating a function, profile the source code to locate the sections of your algorithm that are the most time-consuming. If possible, leave the profiling features in place while you are accelerating the code, so you can easily judge the benefits of using the accelerator. The following general optimizations can maximize the throughput of an accelerated function:

- Using wide memory accesses
- Using localized data



For more information about profiling Nios II systems, refer to [Application Note 391: Profiling Nios II Systems](#).

## Optimization Techniques

### Pipelining Calculations

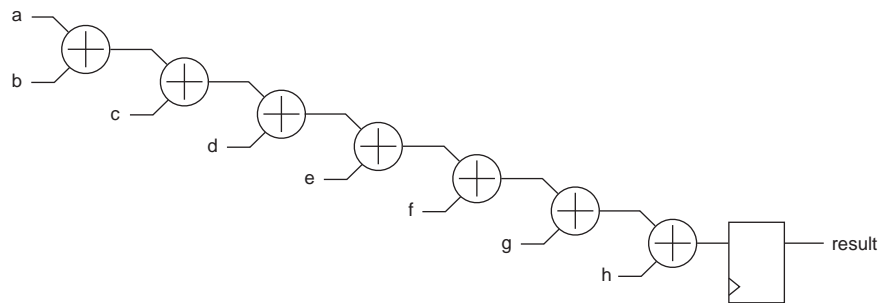
Although condensing multiple mathematical operations to a single line of C code, as in [Example 5-8](#), can reduce the latency of an assignment, it can also reduce the clock speed of the entire design.

**Example 5-8.** Non-Pipelined Calculation (Lower Latency, Degraded  $f_{MAX}$ )

```
int result = a + b + c + d + e + f + g + h;
```

Figure 5-2 shows the hardware generated for Example.

**Figure 5-2.** Non-Pipelined Calculations



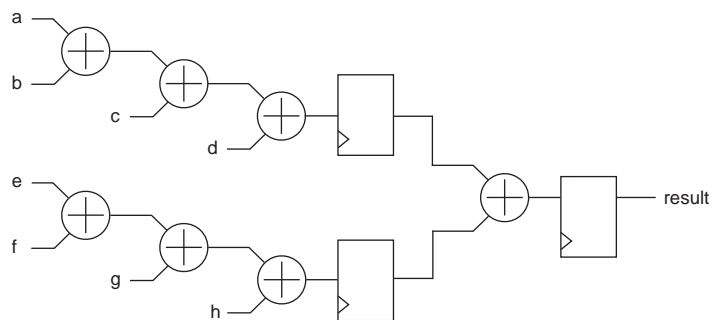
Often, you can break the assignment into smaller steps, as shown in Example 5-9. The smaller steps increase the loop latency, avoiding  $f_{MAX}$  degradation.

**Example 5-9.** Pipelined Calculation (Higher Latency, No  $f_{MAX}$  Degradation)

```
int result_abcd = a + b + c + d;
int result_efgh = e + f + g + h;
int result = result_abcd + result_efgh;
```

Figure 5-3 shows the hardware generated for Example 5-9.

**Figure 5-3.** Pipelined Calculations



## Increasing Memory Efficiency

The following sections discuss coding practices that improve C2H performance.

### Use Wide Memory Accesses

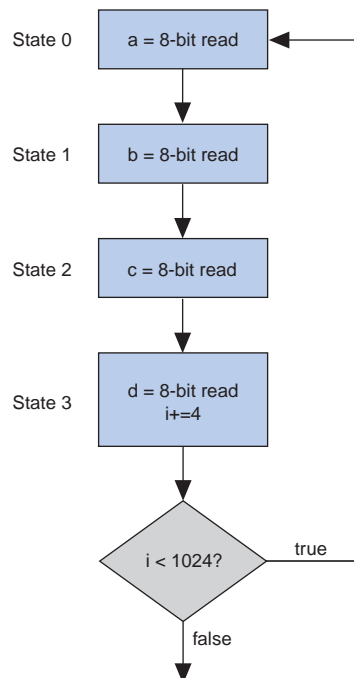
When software runs on a processor with a data cache, byte and halfword accesses to DRAM become full word transfers to and from the cache to guarantee efficient use of memory bandwidth. By contrast, when you make byte and halfword DRAM accesses in a C2H accelerator, as shown in [Example 5-10](#), the Avalon-MM master port connected to the DRAM uses narrow accesses and fails to take advantage of the full data width of the memory.

#### Example 5-10. Narrow Memory Access (Slower Memory Access)

```
unsigned char narrow_array[1024];
char a, b, c, d;
for(i = 0; i < 1024; i+=4)
{
  a = narrow_array[i];
  b = narrow_array[i+1];
  c = narrow_array[i+2];
  d = narrow_array[i+3];
}
```

[Figure 5-4](#) shows the hardware generated for [Example 5-10](#).

**Figure 5-4.** Narrow Memory Access



In a situation where multiple narrow memory accesses are needed, it might be possible to combine those multiple narrow accesses into a single wider access, as shown in [Example 5-11](#). Combining accesses results in the use of fewer memory clock cycles to access the same amount of data. Consolidating four consecutive 8-bit accesses into one 32-bit access effectively increases the performance of those accesses by a factor of four.

**Example 5-11. Wide Memory Access (Faster Memory Access)**

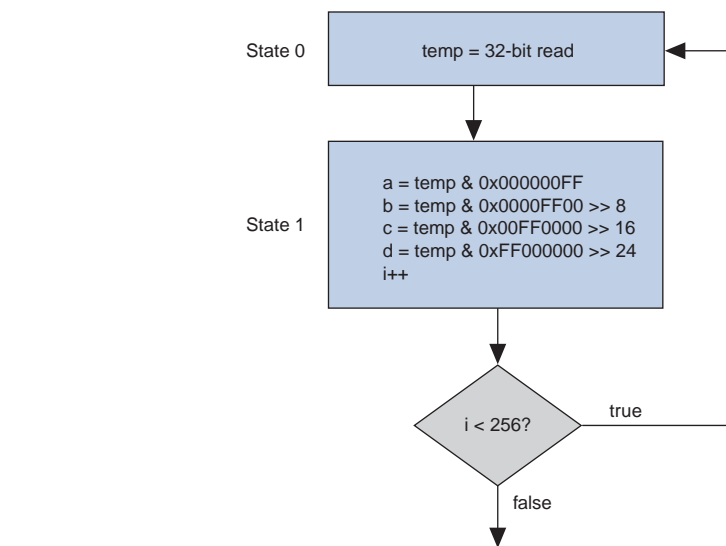
```

unsigned int *wide_array = (unsigned int *) narrow_array;
unsigned int temp;
for(i = 0; i < 256; i++)
{
temp = wide_array[i];
a = (char)( temp and 0x000000FF);
b = (char)( (temp and 0x0000FF00) >> 8);
c = (char)( (temp and 0x00FF0000) >> 16);
d = (char)( (temp and 0xFF000000) >> 24);
}

```

[Figure 5-5](#) shows the hardware generated for [Example 5-11](#).

**Figure 5-5. Wide Memory Access**



## Segment the Memory Architecture

Memory segmentation is an important strategy to increase the throughput of the accelerator. Memory segmentation leads to concurrent memory access, increasing the memory throughput. There are multiple ways to segment your memory and the method used is typically application specific. Refer to [Example 5-12](#) for the following discussions of memory segmentation optimizations.

### Example 5-12. Memory Copy

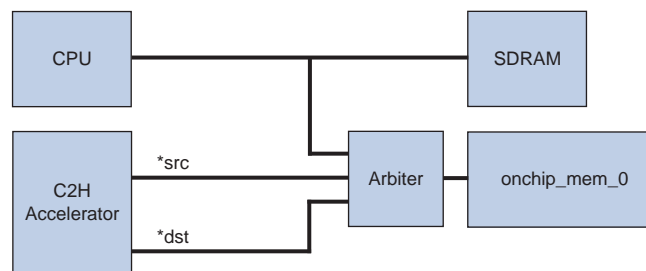
```
void memcpy(char* dst, char* src, int num_bytes)
{
    while (num_bytes-- > 0)
    {
        *dst++ = *src++;
    }
}
```

If the `src` and `dst` memory regions can be moved from the DRAM to an on-chip or off-chip SRAM, better performance is possible. To add on-chip memories, use SOPC Builder to instantiate an on-chip memory component (called `onchip_mem_0` in this example) with a 32-bit wide Avalon-MM slave port. Add the following pragmas to your C code before `memcpy`:

```
#pragma altera_accelerate connect_variable memcpy/dst to onchip_mem_0
#pragma altera_accelerate connect_variable memcpy/src to onchip_mem_0
```

The pragmas state that `dst` and `src` only connect to the `onchip_mem_0` component. This memory architecture offers better performance because SRAMs do not require large bursts like DRAMs to operate efficiently and on-chip memories operate at very low latencies. [Figure 5-6](#) shows the hardware generated for [Example 5-12](#) with the data residing in on-chip RAM.

**Figure 5-6.** Use On-Chip Memory - Partition Memory for Better Bandwidth

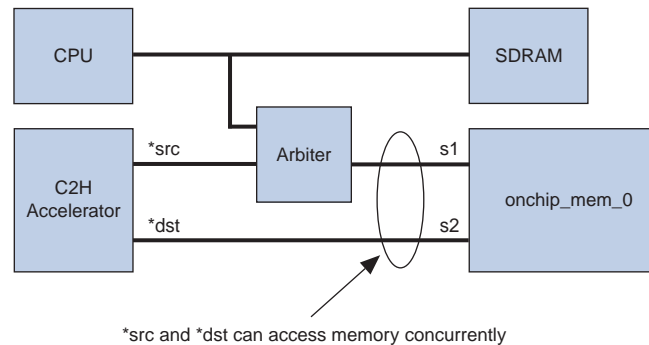


However, both master ports still share the single-port SRAM, `onchip_mem_0`, which can lead to a maximum throughput of one loop iteration every two clock cycles (a CPLI of 2). There are two solutions to this problem: Either create another SRAM so that each Avalon-MM master port has a dedicated memory, or configure the memories with dual-ports. For the latter solution, open your system in SOPC Builder and change the `onchip_mem_0` component to have two ports. This change creates slave ports called `s1` and `s2` allowing the connection pragmas to use each memory port as follows:

```
#pragma altera_accelerate connect_variable memcpy/dst to onchip_mem_0/s1
#pragma altera_accelerate connect_variable memcpy/src to onchip_mem_0/s2
```

These pragmas state that `dst` only accesses slave port `s1` of the `onchip_mem_0` component and that `src` only accesses slave port `s2` of the `onchip_mem_0` component. This new version of `memcpy` along with the improved memory architecture achieves a maximum throughput of one loop iteration per cycle (a CPLI of 1). Figure 5-7 shows the hardware generated for Example 5-12 with the data stored in dual-port on-chip RAM.

**Figure 5-7.** Use Dual-Port On-Chip Memory



### Use Localized Data

Pointer dereferences and array accesses in accelerated functions always result in memory transactions through an Avalon-MM master port. Therefore, if you use a pointer to store temporary data inside an algorithm, as in Example 5-13, there is a memory access every time that temporary data is needed, which might stall the pipeline.

**Example 5-13.** Temporary Data in Memory (Slower)

```
for(i = 0; i < 1024; i++)
{
  for(j = 0; j < 10; j++)
  { /* read and write to the same location */
    AnArray[i] += AnArray[i] * 3;
  }
}
```

Often, storing that temporary data in a local variable, as in Example 5-14, increases the performance of the accelerator by reducing the number of times the accelerator must make an Avalon-MM access to memory. Local variables are the fastest type of storage in an accelerated function and are very effective for storing temporary data.

Although local variables can help performance, too many local variables can lead to excessive resource usage. This is a tradeoff you can experiment with when accelerating a function with the C2H Compiler.

---

**Example 5-14. Temporary Data in Registers (Faster)**

---

```
int temporary;
for(i = 0; i < 1024; i++)
{
    temporary = AnArray[i]; /* read from location i */
    for(j = 0; j < 10; j++)
    {
        /* read and write to a registered value */
        temporary += temporary * 3;
    }
    AnArray[i] = temporary; /* write to location i */
}
```

---

## Reducing Data Dependencies

The following sections provide information on reducing data dependencies.

### Use `__restrict`

By default, the C2H Compiler cannot pipeline read and write pointer accesses because read and write operations may occur at the same memory location. If you know that the `src` and `dst` memory regions do not overlap, add the `__restrict` keyword to the pointer declarations, as shown in [Example 5-15](#).

---

**Example 5-15. `__restrict` Usage**

---

```
void memcpy(char* __restrict__ dst, char* __restrict__ src, int
num_bytes)
{
    while (num_bytes-- > 0)
    {
        *dst++ = *src++;
    }
}
```

---

The `__restrict` declaration on a pointer specifies that accesses via that pointer do not alias any memory addresses accessed by other pointers. Without `__restrict`, the C2H Compiler must schedule accesses to pointers strictly as written which can severely reduce performance.

It is very important that you verify that your algorithm operates correctly when using `__restrict` because this option can cause sequential code to fail when accelerated. The most common error is caused by a read and write pointer causing overlapping accesses to a dual port memory. You might not detect this situation when the function executes in software, because a processor can only perform one access at a time, however by using `__restrict` you are allowing the C2H Compiler to potentially schedule the read and write accesses of two pointers to occur concurrently.

The most common type of data dependency is between scalar data variables. A scalar data dependency occurs when an assignment relies on the result of one or more other assignments. The C code in [Example 5-16](#) shows a data dependency between `sum1` and `result`:

---

**Example 5-16. Scalar Data Dependency**

---

```
int sum3(int a, int b, int c)
{
int sum1 = a + b;
int result = sum1 + c;
return result;
}
```

---

A C compiler attempts to schedule the instructions to prevent the processor pipeline from stalling. There is no limit to the number of concurrent operations which you can exploit with the C2H Compiler. Adding all three integers in one assignment removes the data dependency, as shown in [Example 5-17](#).

---

**Example 5-17. Scalar Data Dependency Removed**

---

```
int sum3(int a, int b, int c)
{
int result = a + b + c;
return result;
}
```

---

The other common type of data dependency is between elements of an array of data. The C2H Compiler treats the array as a single piece of data, assuming that all accesses to the array overlap. For example, the `swap01` function in [Example 5-18](#) swaps the values at index 0 and index 1 in the array pointed to by `<p>`.

---

**Example 5-18. Array Data Dependency**

---

```
void swap01(int* p)
{
int tmp = p[0];
p[0] = p[1];
p[1] = tmp;
}
```

---

The C2H Compiler is unable to detect that the `p[0]` and `p[1]` accesses are to different locations so it schedules the assignments to `p[0]` and `p[1]` sequentially. To force the C2H Compiler to schedule these assignments concurrently, add `__restrict__` to the pointer declaration, as shown in [Example 5-19](#).

---

**Example 5-19. Array Data Dependency Removed**


---

```
void swap01(int* p)
{
int* __restrict__ p0 = andp[0];
int* __restrict__ p1 = andp[1];
int tmp0 = *p0;
int tmp1 = *p1;
*p0 = tmp1;
*p1 = tmp0;
}
```

---

Now, the C2H Compiler attempts to perform the assignments to `p[0]` and `p[1]` concurrently. In this example, there is only a significant performance increase if the memory containing `p[0]` and `p[1]` has two or more write ports. If the memory is single-ported then one access stalls the other and little or no performance is gained.

A form of scalar or array data dependency is the in-scope data dependency. [Example 5-20](#) exhibits an in-scope dependency, because it takes pointers to two arrays and their sizes and returns the sum of the contents of both arrays.

---

**Example 5-20. In-Scope Data Dependency**


---

```
int sum_arrays(int* arr_a, int* arr_b,
int size_a, int size_b)
{
int i;
int sum = 0;
for (i = 0; i < size_a; i++)
{
sum += arr_a[i];
}
for (i = 0; i < size_b; i++)
{
sum += arr_b[i];
}
return sum;
}
```

---

There is a dependency on the `sum` variable which causes C2H accelerator to execute the two loops sequentially. There is no dependency on the loop index variable `<i>` between the two loops because the algorithm reassigns `<i>` to 0 in the beginning of the second loop.

**Example 5-21** shows a new version of `sum_arrays` that removes the in-scope dependency:

---

**Example 5-21. In-Scope Data Dependency Removed**

---

```
int sum_arrays(int* arr_a, int* arr_b,
int size_a, int size_b)
{
int i;
int sum_a = 0;
int sum_b = 0;
for (i = 0; i < size_a; i++)
{
sum_a += arr_a[i];
}
for (i = 0; i < size_b; i++)
{
sum_b += arr_b[i];
}
return sum_a + sum_b;
}
```

---

Using separate sum variables in each loop removes the in-scope dependency. The accelerator adds the two independent sums together at the end of the function to produce the final sum. Each loop runs concurrently although the longest loop determines the execution time. For best performance, connect `arr_a` and `arr_b` to a memory with two read ports or two separate memories.

Sometimes it is not possible to remove data dependencies by simple changes to the C code. Instead, you might need to use a different algorithm to implement the same functionality. In **Example 5-22**, the code searches for a value in a linked list and returns 1 if the value found and 0 if it is not found. Arguments to the search function are the pointer to the head of the linked list and the value to match against.

---

**Example 5-22. Pointer-Based Data Dependency**

---

```
struct item
{
int value;
struct item* next;
};
int search(struct item* head, int match_value)
{
struct item* p;
for (p=head; p != NULL; p=p->next)
{
if (p->value == match_value)
{
return 1; // Found a match
}
}
return 0; // No match found
}
```

---

The C2H Compiler is not able to achieve a throughput of one comparison per cycle due to the `p=p->next` does not occur until the next pointer location has been read from memory, causing a latency penalty to occur each time the loop state machine reaches this line of C code. To achieve better performance, use a different algorithm that supports more parallelism. For example, assume that the values are stored in an array instead of a linked list, as in [Example 5-23](#). Arguments to the new search function are the pointer to the array, the size of the array, and the value to match against.

---

**Example 5-23. Pointer-Based Data Dependency Removed**

---

```
int search(int* arr, int num_elements, int match_value)
{
  for (i = 0; i < num_elements; i++)
  {
    if (arr[i] == match_value)
    {
      return 1; // Found a match
    }
  }
  return 0; // No match found
}
```

---

This new search function achieves a throughput of one comparison per cycle assuming there is no contention for memory containing the `arr` array. Prototype such a change in software before accelerating the code, because the change affects the functionality of the algorithm.

## Reducing Logic Utilization

The following sections discuss coding practices you can adopt to reduce logic utilization.

### Use "do-while" rather than "while"

The overhead of `do` loops is lower than those of the equivalent `while` and `for` loops because the accelerator checks the loop condition after one iteration of the loop has executed. The C2H Compiler treats a `while` loop like a `do` loop nested in an `if` statement. [Example 5-24](#) illustrates code that the C2H Compiler transforms into a `do` loop and nested `if` statement.

---

**Example 5-24. while Loop**

---

```
int sum(int* arr, int num_elements)
{
  int result = 0;
  while (num_elements-- > 0)
  {
    result += *arr++;
  }
  return result;
}
```

---

[Example 5-25](#) is the same function rewritten to show how the C2H Compiler converts a while loop to an if statement and a do loop.

---

**Example 5-25.** Converted while Loop

---

```
int sum(int* arr, int num_elements)
{
  int result = 0;
  if (num_elements > 0)
  {
    do
    {
      result += *arr++;
    } while (--num_elements > 0);
  }
  return result;
}
```

---

Notice that an extra if statement outside the do loop is required to convert the while loop to a do loop. If you know that the sum function is never called with an empty array, that is, the initial value of num\_elements is always greater than zero, the most efficient C2H code uses a do loop instead of the original while loop. [Example 5-26](#) illustrates this optimization.

---

**Example 5-26.** do Loop

---

```
int sum(int* arr, int num_elements)
{
  int result = 0;
  do
  {
    result += *arr++;
  } while (--num_elements > 0);
  return result;
}
```

---

### Use Constants

Constants provide a minor performance advantage in C code compiled for a processor. However, they can provide substantial performance improvements in a C2H accelerator.

[Example 5-27](#) demonstrates a typical add and round function.

---

**Example 5-27.** Add and Round with Variable Shift Value

---

```
int add_round(int a, int b, int sft_amount)
{
  int sum = a + b;
  return sum >> sft_amount;
}
```

---

As written above, the C2H Compiler creates a barrel shifter for the right shift operation. If `add_round` is always called with the same value for `sft_amount`, you can improve the accelerated function's efficiency by changing the `sft_amount` function parameter to a `#define` value and changing all your calls to the function. [Example 5-28](#) is an example of such an optimization.

---

**Example 5-28.** Add and Round with Constant Shift Value

---

```
#define SFT_AMOUNT 1
int add_round(int a, int b)
{
    int sum = a + b;
    return sum >> SFT_AMOUNT;
}
```

---

Alternatively, if `add_round` is called with a few possible values for `sft_amount`, you can still avoid the barrel shifter by using a `switch` statement which just creates a multiplexer and a small amount of control logic. [Example 5-29](#) is an example of such an optimization.

---

**Example 5-29.** Add and Round with a Finite Number of Shift Values

---

```
int add_round(int a, int b, int sft_amount)
{
    int sum = a + b;
    switch (sft_amount)
    {
        case 1:
            return sum >> 1;
        case 2:
            return sum >> 2;
    }
    return 0; // Should never be reached
}
```

---

You can also use these techniques to avoid creating a multiplier or divider. This technique is particularly beneficial for division operations because the hardware responsible for the division is large and relatively slow.

## Leave Loops Rolled Up

Sometimes developers unroll loops to achieve better results using a C compiler. Because the C2H Compiler attempts to pipeline all loops, unrolling loops is unnecessary for C2H code. In fact, unrolled loops tend to produce worse results because the C2H Compiler creates extra logic. It is best to leave the loop rolled up. [Example 5-30](#) shows an accumulator algorithm that was unrolled in order to execute faster on a processor.

### Example 5-30. Unrolled Loop

---

```
int sum(int* arr)
{
  int i;
  int result = 0;
  for (i = 0; i < 100; i += 4)
  {
    result += *arr++;
    result += *arr++;
    result += *arr++;
    result += *arr++;
  }
  return result;
}
```

---

This function is passed an array of 100 integers, accumulates each element, and returns the sum. To achieve higher performance on a processor, the developer has unrolled the inner loop four times, reducing the loop overhead by a factor of four when executed on a processor. When the C2H Compiler maps this code to hardware, there is no loop overhead because the accelerator executes the loop overhead statements concurrently with the loop body.

As a result of unrolling the code, the C2H Compiler creates four times more logic because four separate assignments are used. The C2H Compiler creates four Avalon-MM master ports in the loop. However, an Avalon-MM master port can only perform one read or write operation at any given time. The four master ports must interleave their accesses, eliminating any advantage of having multiple masters.

[Example 5-30](#) shows how resource sharing (memory) can cause parallelism to be nullified. Instead of using four assignments, roll this loop up as shown in [Example 5-31](#).

### Example 5-31. Rolled-Up Loop

---

```
int sum(int* arr)
{
  int i;
  int result = 0;
  for (i = 0; i < 100; i++)
  {
    result += *arr++;
  }
  return result;
}
```

---

This implementation achieves the same throughput as the previous unrolled example because this loop can potentially iterate every clock cycle. The unrolled algorithm iterates every four clock cycles due to memory stalls. Because these two algorithms achieve the same throughput, the added benefit of the rolling optimization is savings on logic resources such as Avalon-MM master ports and additional accumulation logic.

### Use ++ to Sequentially Access Arrays

The unrolled version of the `sum` function in [Example 5-30](#) uses `*arr++` to sequentially access all elements of the array. This procedure is more efficient than the alternative shown in [Example 5-32](#).

#### Example 5-32. Traversing Array with Index

```
int sum(int* arr)
{
  int i;
  int result = 0;
  for (i = 0; i < 100; i++)
  {
    result += arr[i];
  }
  return result;
}
```

The C2H Compiler must create pointer dereferences for both `arr[i]` and `*arr++`. However, the instantiated logic is different for each case. For `*arr++` the value used to address memory is the pointer value itself, which is capable of incrementing. For `arr[i]` the accelerator must add base address `arr` to the counter value `i`. Both require counters, however in the case of `arr[i]` an adder block is necessary, which creates more logic.

### Avoid Excessive Pointer Dereferences

Any pointer dereference via the dereference operator `*` or array indexing might create an Avalon-MM master port. Avoid using excessive pointer dereference operations because they lead to both additional logic which degrades the  $f_{MAX}$  of the design.



Any local arrays within the accelerated function instantiate on-chip memory resources. Do not declare large local arrays because the amount of on-chip memory is limited and excessive use affects the routing of the design.

### Avoid Multipliers

Embedded multipliers have become a standard feature of FPGAs; however, they are still limited resources. When you accelerate source code that uses a multiplication function, the C2H accelerator instantiates a multiplier. Embedded multiplier blocks have various modes that allow them to be segmented into smaller multiplication units depending on the width of the data being used. They also have the ability to perform multiply and accumulate functionality.

When using multipliers in accelerated code validate the data width of the multiplication to reduce the logic. The embedded multiplier blocks handle 9 by 9 (char \*char), 18 by 18 (short \*short), and 36 by 36 (long \*long) modes which are set depending on the size of the largest width input. Reducing the input width of multiplications not only saves resources, but also improves the routing of the design, because multiplier blocks are fixed resources. If multiplier blocks are not available or the design requires too many multiplications, the Quartus II software uses logic elements to create the multiplication hardware. Avoid this situation if possible, because multipliers implemented in logic elements are expensive in terms of resources and design speed.

Multiplications by powers of two do not instantiate multiplier logic because the accelerator can implement them with left shift operations. The C2H Compiler performs this optimization automatically, so it is not necessary to use the << operator. When multiplication is necessary, try to use powers of two in order to save logic resources and to benefit from the fast logic created for this operation. An assignment that uses a multiplication by a power of two becomes a register-to-register path in which the data is shifted in the system interconnect fabric.

When multiplying by a constant the Quartus II software optimizes the LEs either using memory or logic optimizations. [Example 5-33](#) shows an optimization for multiplication by a constant.

---

**Example 5-33. Multiplication by Constants**

```
/* This multiplication by a constant is optimized */  
y = a * 3;  
  
/*The optimization is shift and add: (2*a + a = 3*a) */  
y = a << 1 + a
```

---

C2H offloads the intelligence of multiplies, divides, and modulo to Quartus II synthesis to do the right thing when possible.

**Avoid Arbitrary Division**

If at all possible, avoid using arbitrary division in accelerated functions, including the modulus % operator. Arbitrary division occurs whenever the divisor is unknown at compile time. True division operations in hardware are expensive and slow.

---

**Example 5-34. Arbitrary Division (Expensive, Slow):**

```
z = y / x; /* x can equal any value */
```

---

The exception to this rule is division by denominators which are positive powers of two. Divisions by positive powers of two simply become binary right-shift operations. Dividing by two can be accomplished by shifting the value right one bit. Dividing by four is done by shifting right two bits, and so on. If the accelerated function uses the / division operator, and the right-hand argument is a constant power of two, the C2H Compiler converts the divide into a fixed-bit shift operation. In hardware, fixed-bit shift operations result in only wires, which are free.

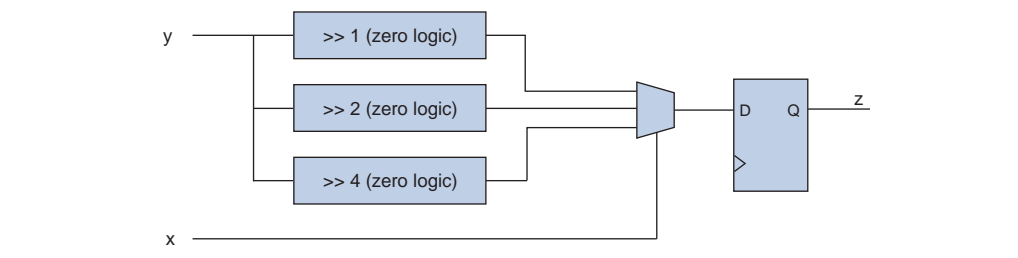
If a division operation in an accelerated function always uses a denominator that is a power of two, but can use various multiples of two, you can use a ternary operation to convert the divides to the appropriate fixed-bit shift, as shown in [Example 5-35](#).

**Example 5-35.** Division using Shifts with a Ternary Operator (Cheap, Fast)

```
z = (x == 2)? y >> 1:(x == 4)? y >> 2: y >> 4);
```

[Figure 5-8](#) shows the hardware generated for Example.

**Figure 5-8.** Ternary Shift Divide



The logic created by this optimization is relatively cheap and fast, consisting of a multiplexer and minimal control logic. Because the assignments to  $z$  are just shifted copies of  $y$  the multiplexer is the only logic in the register-to-register path. If there are many possible denominator values, explore the tradeoff between latency and frequency discussed in the “[Improve Conditional Frequency](#)” on page 5-37.

The other possible optimization to avoid generating an expensive and slow division circuit is to implement a serial divider. Serial dividers have a high latency, but tend not to degrade  $f_{MAX}$ . Another benefit of using serial division is the relatively low cost of the hardware generated because the operations performed are on bits instead of words.

You can use macros in `c2h_division.h` and `c2h_modulo.h` to implement serial division or modulo operations in your own system. These files are available on the Nios II literature page. A hyperlink to the software files appears next to [Optimizing Nios II C2H Compiler Results](#) (this document), at [www.altera.com/literature/lit-nio2.jsp](http://www.altera.com/literature/lit-nio2.jsp). The two header files are distributed in a zip file.

## Use Masks

Both the C compiler for a 32-bit processor and the C2H Compiler convert data types smaller than integers to 32-bit integers. If you want to override this default behavior to save logic and avoid degrading the  $f_{MAX}$  of the design, add a bitwise AND with a mask. In [Example 5-36](#), the C2H Compiler promotes `b1` and `b2` to 32-bit integers when performing the addition so that it instantiates a 32-bit adder in hardware. However, because `b1` and `b2` are unsigned characters, the sum of `b1` and `b2` is guaranteed to fit in nine bits, so you can mask the addition to save bits. The C2H Compiler still instantiates a 32-bit adder but Quartus II synthesis removes the unnecessary bits, resulting in a 9-bit adder in hardware.

### Example 5-36. Use Bitwise And with a Mask

```
unsigned int add_chars(unsigned char b1, unsigned char b2)
{
    return (b1 + b2) & 0x1fff;
}
```



This optimization can cause a failure if you mistakenly reduce the width of the calculation so that needed data resolution is lost. Another common mistake is to use bit masks with signed data. Signed data, stored using 2's complement format, requires that the accelerator preserve and extend the sign bit through the masking operation.

## Use Powers of Two in Multi-Dimensional Arrays

A conventional C compiler implements a multidimensional array as a one-dimensional array stored in row-major order. For example, a two-dimensional array might appear as follows:

```
#define NUM_ROWS 10
#define NUM_COLS 12
int arr2d[NUM_ROWS][NUM_COLS];
```

The first array index is the row and the second is the column. A conventional C compiler implements this as a one-dimensional array with `NUM_ROWS` x `NUM_COLS` elements. The compiled code computes the offset into the one-dimensional array using the following equation:

```
offset = row * NUM_COLS + col;
```

The C2H Compiler follows this implementation of multidimensional arrays. Whenever your C code indexes into a multidimensional array, an implicit multiplication is created for each additional dimension. If the multiplication is by a power of two, the C2H Compiler implements the multiplication with a wired shift, which is free. If you can increase that dimension of the array to a power of two, you save a multiplier. This optimization comes at the cost of some memory, which is cheap. In the example, just make the following change:

```
#define NUM_COLS 16
```

To avoid all multipliers for multidimensional array accesses of  $<n>$  dimensions, you must use an integer power of two array size for each of the final  $<n-1>$  dimensions. The first dimension can have any length because it does not influence the decision made by the C2H Compiler to instantiate multipliers to create the index.

### Use Narrow Local Variables

The use of local variables that are larger data types than necessary can waste hardware resources in an accelerator. [Example 5-37](#) includes a variable that is known to contain only the values 0–229. Using a `long int` variable type for this variable creates a variable that is much larger than needed. This type of optimization is usually not applicable to pointer variables. Pointers always cost 32 bits, regardless of their type. Reducing the type size of a pointer variable affects the size of the data the pointer points to, not the pointer itself. It is generally best to use large pointer types to take advantage of wide memory accesses. Refer to [“Use Wide Memory Accesses” on page 5-16](#) for details.

---

#### Example 5-37. Wide Local Variable `i` Costs 32 Bits

```
int i;
int var;
for(i = 0; i < 230; i++)
{
var += *ptr + i;
}
```

---

An `unsigned char` variable type, as shown in [Example 5-38](#), is large enough because it can store values up to 255, and only costs 8 bits of logic, whereas a `long int` type costs 32 bits of logic. Excessive logic utilization wastes FPGA resources and can degrade system  $f_{MAX}$ .

---

#### Example 5-38. Narrow Local Variable `i` Costs 8 Bits

```
unsigned char i;
int var;
for(i = 0; i < 230; i++)
{
var += *ptr + i;
}
```

---

## Optimizing Memory Connections

The following sections discuss ways to optimize memory connectivity.

### Remove Unnecessary Connections to Memory Slave ports

The Avalon-MM master ports associated with the `src` and `dst` pointers in [Example 5-39](#) are connected to all of the Avalon-MM slave ports that are connected to the processor's data master. Typically, the accelerator does not need to access all these slave ports. This extra connectivity adds unnecessary logic to the system interconnect fabric, which increases the hardware resources and potentially creates long timing paths, degrading  $f_{MAX}$ .

The C2H Compiler supports pragmas added to your C code to inform the C2H Compiler which slave ports each pointer accesses in your accelerator. For example, if the `src` and `dst` pointers can only access the DRAM (assume it is called `dram_0`), add these pragmas before `memcpy` in your C code.

```
#pragma altera_accelerate connect_variable memcpy/dst to dram_0
#pragma altera_accelerate connect_variable memcpy/src to dram_0
```

---

**Example 5-39. Memory Interconnect**

---

```
void memcpy(char* dst, char* src, int num_bytes)
{
while (num_bytes-- > 0)
{
*dst++ = *src++;
}
}
```

---

These pragmas state that `dst` and `src` only access the `dram_0` component. The C2H Compiler connects the associated Avalon-MM ports only to the `dram_0` component.

**Reduce Avalon-MM Interconnect Using #pragma**

Accelerated functions use Avalon-MM ports to access data related to pointers in the C code. By default, each master generated connects to every memory slave port that is connected to the Nios II data master port. This connectivity can result in large amounts of arbitration logic when you generate an SOPC Builder system, which is expensive and can degrade system  $f_{MAX}$ . In most cases, pointers do not need to access every memory in the system.

You can reduce the number of master-slave port connections in your SOPC Builder system by explicitly specifying the memories to which a pointer dereference must connect. You can make connections between pointers and memories with the `connect_variable` pragma directive, as shown in [Example 5-40](#). In [Figure 5-9](#), three pointers, `output_data`, `input_data1`, and `input_data2` are connected to

memories named `sdram`, `onchip_dataram1`, and `onchip_dataram2`, respectively. Using the `connect_variable` pragma directive ensures that each of the accelerated function's three Avalon-MM master ports connects to a single memory slave port. The result is a more efficient overall because it has no unnecessary master-slave port connections.

---

**Example 5-40. Reducing Memory Interconnect**

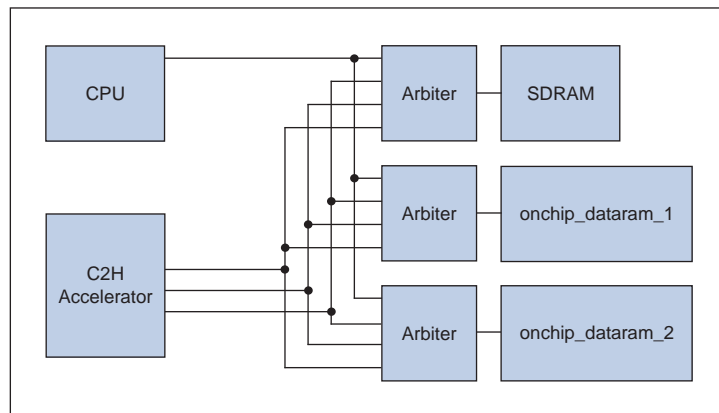
---

```
#pragma altera_accelerate connect_variable my_c2h_function/output_data to sdram
#pragma altera_accelerate connect_variable my_c2h_function/input_data1 to onchip_dataram1
#pragma altera_accelerate connect_variable my_c2h_function/input_data2 to onchip_dataram2

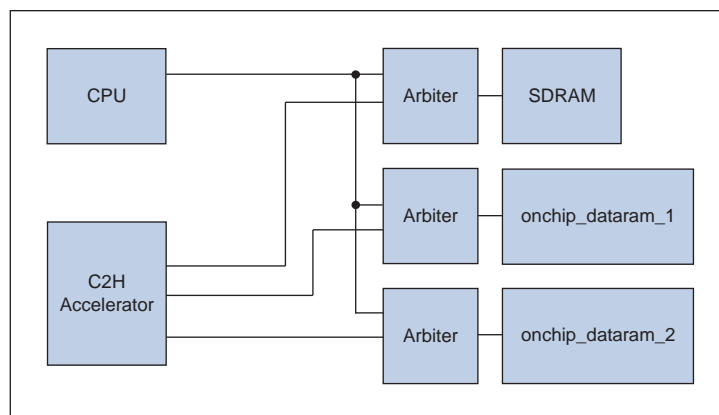
void my_c2h_function( int *input_data1,
                    int *input_data2,
                    int* output_data )
{
    char i;
    for( i = 0; i < 52; i++ )
    {
        *(output_data + i) = *(input_data1 + i) + *(input_data2 + i);
    }
}
```

---

**Figure 5-9.** Pragma Connect



All connections (unnecessary arbitration logic)



Only necessary connections (less arbitration logic)

### Remove Unnecessary Memory Connections to Nios II Processor

As part of your optimization, you might have added on-chip memories to the system to allow an accelerated function access to multiple pointers in parallel, as in “[Segment the Memory Architecture](#)” on page 5-18. During implementation and debug, it is important that these on-chip memories have connections to both the appropriate accelerator Avalon-MM master port and to the Nios II data master port, so the function can run in both accelerated and non-accelerated modes. In some cases however, after you are done debugging, you can remove the memory connections to the Nios II data master if the processor does not access the memory when the function is accelerated. Removing connections lowers the cost and avoids degrading system  $f_{MAX}$ .

### Optimizing Frequency Versus Latency

The following sections describe tradeoffs you can make between frequency and latency to improve performance.

### Improve Conditional Latency

Algorithms that contain `if` or `case` statements use registered control paths when accelerated. The C2H Compiler accelerates the code show in [Example 5-41](#) in this way.

#### Example 5-41. Registered Control Path

```
if(testValue < Threshold)
{
a = x;
}
else
{
a = y;
}
```

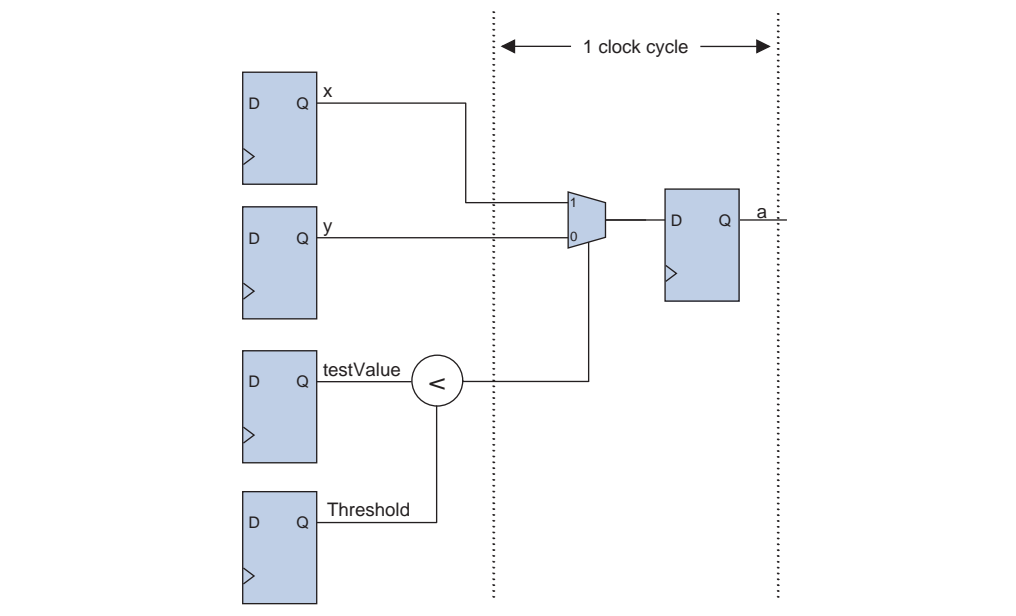
You can modify your software to make use of the ternary operator, (`? :` ), as in [Example 5-42](#), to reduce the latency of the control path. The ternary operator does not register signals on the control path, so this optimization results in lower latency at the expense of  $f_{MAX}$ . This optimization primarily helps reduce the CPLI of the accelerator when a data dependency prevents the conditional statement from becoming fully pipelined. Do not use this optimization if the CPLI of the loop containing the conditional statement is already equal to one.

#### Example 5-42. Unregistered Control Path

```
a = (testValue < Threshold)? x : y;
```

[Figure 5-10](#) shows the hardware generated for [Example 5-42](#).

**Figure 5-10.** Conditional Latency Improvement



### Improve Conditional Frequency

If you wish to avoid degrading  $f_{MAX}$  in exchange for an increase in latency, consider removing ternary operators. By using an `if` or `case` statement to replace the ternary operator the control path of the condition becomes registered and shortens the timing paths in that portion of the accelerator. In the case of the conditional statement being executed infrequently (outside of a loop), this optimization might prove a small price to pay to increase the overall frequency of the hardware design.

[Example 5-43](#) and [Example 5-44](#) show how you can rewrite a ternary operator as an `if` statement.

#### Example 5-43. Unregistered Conditional Statement

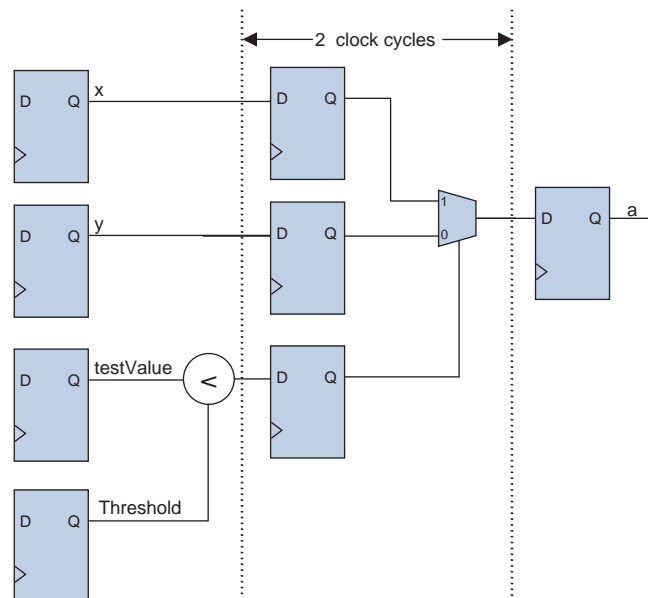
```
a = (testValue < Threshold)? x : y;
```

#### Example 5-44. Registered Conditional Statement

```
if(testValue < Threshold)
{
a = x;
}
else
{
a = y;
}
```

[Figure 5-11](#) shows the hardware the C2H Compiler generates for [Example 5-44](#).

**Figure 5-11.** Conditional Frequency Improvement



## Improve Throughput

To increase the computational throughput, focus on two main areas: achieving a low CPLI, and performing many operations within one loop iteration.

### Avoid Short Nested Loops

Because a loop has a fixed latency before any iteration can occur, nesting looping structures can lead to unnecessary delays. The accelerator incurs the loop latency penalty each time it enters the loop. Rolling software into loops adds the possible benefit of pipelining, and the benefits of this pipelining usually outweigh the latency associated with loop structures. Generally, if the latency is greater than the maximum number of iterations times the CPLI then the looping implementation is slower. You must take into account that leaving a loop unrolled usually increases the resource usage of the hardware accelerator.

#### Example 5-45. Nested Loops

---

```
for(loop1 = 0; loop1 < 10; loop1++) /* Latency = 3, CPLI = 2 */
{
  /* statements requiring two clock cycles per loop1 iteration */
  for(loop2 = 0; loop2 < 5; loop2++) /* Latency = 10, CPLI = 1 */
  {
    /* statements requiring one clock cycle per loop2 iteration */
  }
}
```

---

Assuming no memory stalls occur, the total number of clock cycles is as follows:

$$\text{Innerloop} = \text{latency} + (\text{iterations} - 1)(\text{CPLI} + \text{innerlooptime})$$

$$\text{Innerloop} = 10 + 4(1 + 0)$$

$$\text{Innerloop} = 14 \text{ cycles}$$

$$\text{Outerloop} = 3 + 9(2 + 14)$$

$$\text{Outerloop} = 147 \text{ cycles}$$

Due to the high latency of the inner loop the total time for this example is 147 clock cycles.

#### Example 5-46. Single Loop

---

```
for(loop1 = 0; loop1 < 10; loop1++) /* Latency = 3, CPLI = 7 */
{
  /* statements requiring two clock cycles per loop1 iteration */
  /* statements that were previously contained in loop2 */
}
```

---

Assuming no memory stalls occur, the total number of clock cycles is as follows:

$$\text{Outerloop} = \text{latency} + (\text{iterations} - 1)(\text{CPLI} + \text{innerlooptime})$$

$$\text{Outerloop} = 3 + 9(7 + 0)$$

$$\text{Outerloop} = 66 \text{ cycles}$$

The inner loop (loop2) has been eliminated and consequently is 0 in these equations. Combining the inner loop with the outer loop dramatically decreases the total time to complete the same outer loop. This optimization assumes that unrolling the inner loop resulted in adding five cycles per iteration to the outer loop. The combination of the loops would most likely result in a hardware utilization increase which you must take into consideration.

### Remove In-place Calculations

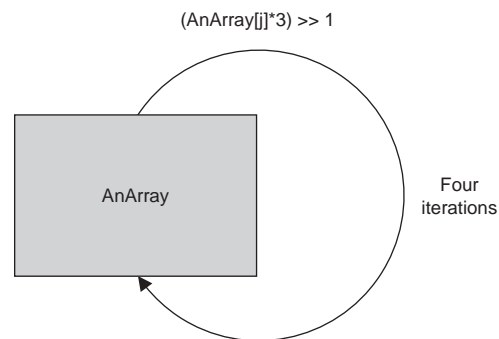
Some software algorithms perform in-place calculations, in which results overwrite the input data as they are calculated. This technique conserves memory, but produces suboptimal performance when compiled to a C2H accelerator. Example 5-47 shows such an algorithm. Unfortunately this approach leads to memory stalls because in-place algorithms read and write to the same memory locations.

#### Example 5-47. Two Avalon-MM Ports Using The Same Memory

```
for(i = 0; i < 4; i++)
{
for(j = 0; j < 1024; j++)
{
AnArray[j] = (AnArray[j] * 3) >> 1;
}
}
```

Figure 5-12 shows the dataflow in hardware generated for Example.

Figure 5-12. In-Place Calculation



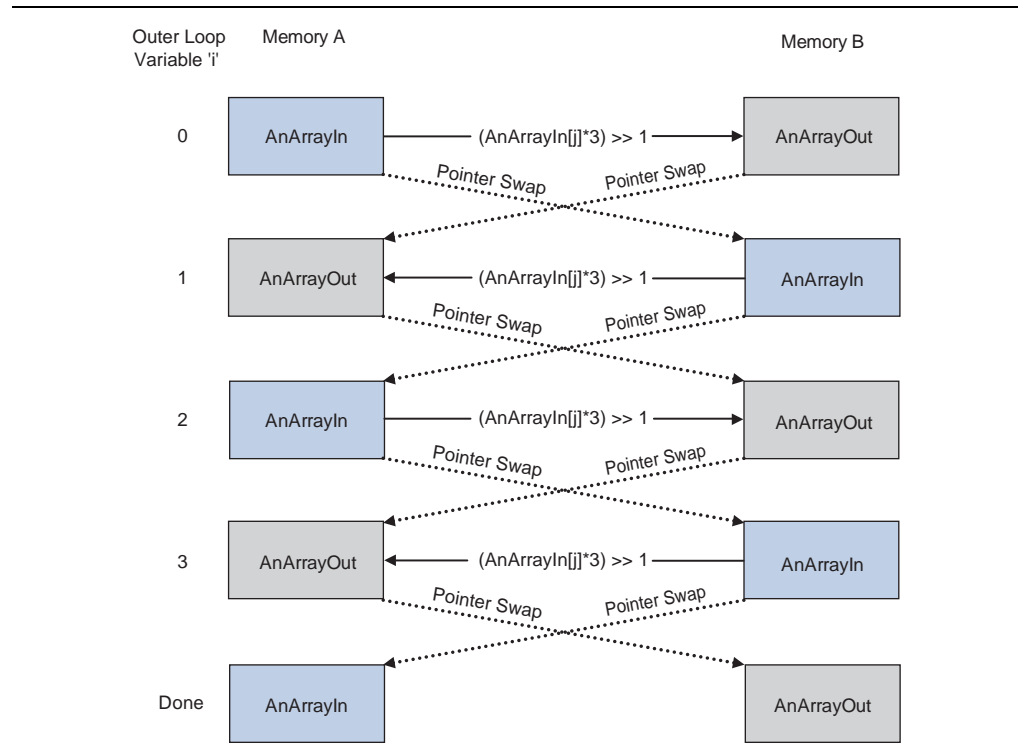
To solve this problem, remove the in-place behavior of the algorithm by adding a "shadow" memory to the system, as shown in [Example 5-48](#). Instead of the input and output residing in the same memory, each uses an independent memory. This optimization prevents memory stalls because the input and output data reside in separate memories.

**Example 5-48.** Two Avalon-MM Ports Using Separate Memories

```
int * ptr;
for(i = 0; i < 4; i++)
{
  for(j = 0; j < 1024; j++)
  {
    /* In from one memory and out to the other */
    AnArrayOut[j] = (AnArrayIn[j] * 3) >> 1;
  }
  /* Swap the input and output pointers and do it all
  over again */
  ptr = AnArrayOut;
  AnArrayOut = AnArrayIn;
  AnArrayIn = ptr;
}
```

[Figure 5-13](#) shows the dataflow of hardware generated for [Example 5-48](#).

**Figure 5-13.** In-Place Calculation



You can also use this optimization if the data resides in on-chip memory. Most on-chip memory can be dual-ported to allow for simultaneous read and write access. With a dual-port memory, the accelerator can read the data from one port without waiting for the other port to be written. When you use this optimization, the read and write addresses must not overlap, because that could lead to data corruption. A method for preventing a read and a write from occurring simultaneously at the same address is to read the data into a variable before the write occurs.

### Replace Arrays

Often software uses data structures that are accessed via a base pointer location and offsets from that location, as shown in [Example 5-49](#). When the hardware accelerator accesses the data in these structures, memory accesses result.

#### Example 5-49. Individual Memory Accesses

---

```
int a = Array[0];  
int b = Array[1];  
int c = Array[2];  
int d = Array[3];
```

---

You can replace these memory accesses using a single pointer and registers, as in [Example 5-50](#). The overall structure of the hardware created resembles a FIFO.

#### Example 5-50. FIFO Memory Accesses

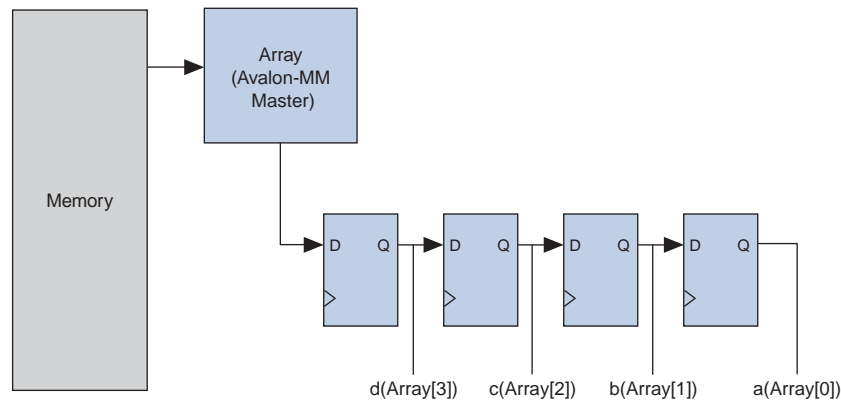
---

```
/* initialize variables */  
int a = 0;  
int b = 0;  
int c = 0;  
int d = 0;  
for(i = 0; i < 4; i++)  
{  
    d = Array[i];  
    c = d;  
    b = c;  
    a = b;  
}
```

---

Figure 5-14 shows the hardware generated for Example 5-50.

**Figure 5-14.** Array Replacement



### Use Polled Accelerators

When you create a hardware accelerator using the C2H Compiler, it creates a wrapper file that is linked at compile time, allowing the main program to call both the software and hardware versions of the algorithm using the same function name. The wrapper file performs the following three tasks:

- Writes the passed parameters to the accelerator
- Polls the accelerator to determine when the computation is complete
- Sends the return value back to the caller

Because the wrapper file is responsible for determining the status of the accelerator, the Nios II processor must wait for the wrapper code to complete. This behavior is called blocking.

The hardware accelerator blocks the Nios II processor from progressing until the accelerator has reached completion. The wrapper file is responsible for this blocking action. Using the same pragma statement to create the interrupt include file, you can access the macros defined in it to implement a custom polling algorithm common in systems that do not use a real time operating system.

Instead of using the interrupt to alert Nios II that the accelerator has completed its calculation, the software polls the busy value associated with the accelerator. The macros necessary to manually poll the accelerator to determine if it has completed are in the include file created under either the **Debug** or **Release** directory of your application project. These macros are shown in Table 5-5.

**Table 5-5.** C2H Accelerator Polling Macros

Purpose	Macro Name
Busy value	ACCELERATOR_<Project Name>_<Function Name>_BUSY ( )
Return value	ACCELERATOR_<Project Name>_<Function Name>_GET_RETURN_VALUE ( )

While the accelerator is busy, the rest of the software must not attempt to read the return value because it might be invalid.

## Use an Interrupt-Based Accelerator

The blocking behavior of a polled accelerator might be undesirable if there are processing tasks which the Nios II processor can carry out while the accelerator is running. In this case, you can create an interrupt-based accelerator.

Create the hardware accelerator with the standard flow first, because interrupts add an extra level of complexity. Before proceeding to the interrupt flow, debug the system to make sure the accelerator behaves correctly. Add enhancements in polled mode, as well.

To use the hardware accelerator in a non-blocking mode, add the following line to your function source code:


```
#pragma altera_accelerate enable_interrupt_for_function<function name>
```

At the next software compilation, the C2H Compiler creates a new wrapper file containing all the macros needed to use the accelerator and service the interrupts it generates. The hardware accelerator does not have an IRQ level so you must open the system in SOPC Builder and manually assign this value. After assigning the IRQ level you must click the **Generate** button to regenerate your SOPC Builder system.

The macros necessary to service the accelerator interrupt are in the **include** file created under either the **Debug** or **Release** directory of your application project. These macros are shown in [Table 5-6](#).

**Table 5-6.** C2H Accelerator Interrupt Macros

Purpose	Macro Name
IRQ level value	ACCELERATOR_<Project Name>_<Function Name>_IRQ ( )
Return value	ACCELERATOR_<Project Name>_<Function Name>_GET_RETURN_VALUE ( )
Interrupt clear	ACCELERATOR_<Project Name>_<Function Name>_CLEAR_IRQ ( )

 Refer to the [Exception Handling](#) chapter in the *Nios II Software Developer's Handbook* for more information about creating interrupt service routines.

## Glossary

This document uses the following terminology:

- **Accelerator throughput**—the throughput achieved by a C2H accelerator during a single invocation. Accelerator throughput might be less than peak throughput if pipeline stalls occur. Accelerator throughput does not include latency. See also CPLI, throughput, peak throughput, application throughput.
- **Application throughput**—the throughput achieved by a C2H accelerator in the context of the application, involving multiple accelerator invocations and including the number of cycles of latency.
- **Barrel shifter** – hardware that shifts a byte or word of data an arbitrary number of bits in one clock cycle. Barrel shifters are fast and expensive, and can degrade  $f_{MAX}$ .

- **Cache coherency**—the integrity of cached data. When a processor accesses memory through a cache and also shares that memory with a coprocessor (such as a C2H accelerator), it must ensure that the data in memory matches the data in cache whenever the coprocessor accesses the data. If the coprocessor can access data in memory that has not been updated from the cache, there is a cache-coherency problem.
- **Compute-limited**—describes algorithms whose speed is restricted by how fast data can be processed. When an algorithm is compute-limited, there is no benefit from increasing the efficiency of memory or other hardware. See also data-limited.
- **Control path**—a chain of logic controlling the output of a multiplexer
- **CPLI**—cycles per loop iteration. The number of clock cycles required to execute one loop iteration. CPLI does not include latency.
- **Critical timing path**—the longest timing path in a clock domain. The critical timing path limits  $f_{MAX}$  or the entire clock domain. See also timing path.
- **Data dependency**—a situation where the result of an assignment depends on the result of one or more other assignments, as in [Example 5-5](#).
- **Data-limited**—describes algorithms whose speed is restricted by how fast data can be transferred to or from memory or other hardware. When an algorithm is data-limited, there is no benefit from increasing processing power. See also compute-limited.
- **DRAM**—dynamic random access memory. It is most efficient to access DRAM sequentially, because there is a time penalty when it is accessed randomly. SDRAM is a common type of DRAM.
- **Latency**—a time penalty incurred each time the accelerator enters a loop.
- **Long timing path**—a critical timing path that degrades  $f_{MAX}$ .
- **Peak throughput**—the throughput achieved by a C2H accelerator, assuming no pipeline stalls and disregarding latency. For a given loop, peak throughput is inversely proportional to CPLI. See also throughput, accelerator throughput, application throughput, CPLI, latency.
- **Rolled-up loop**—A normal C loop, implementing one algorithmic iteration per processor iteration. See also unrolled loop.
- **SDRAM**—synchronous dynamic random access memory. See DRAM.
- **SRAM**—static random access memory. SRAM can be accessed randomly without a timing penalty.
- **Subfunction**—a function called by an accelerated function. If `apple()` is a function, and `apple()` calls `orange()`, `orange()` is a subfunction of `apple()`. If `orange()` calls `banana()`, `banana()` is also a subfunction of `apple()`.
- **Throughput**—the amount of data processed per unit time. See also accelerator throughput, application throughput, and peak throughput.
- **Timing path**—a chain of logic connecting the output of a hardware register to the input of the next hardware register.

- **Unrolled loop**—A C loop that is deconstructed to implement more than one algorithmic iteration per loop iteration, as illustrated in [Example 5-30](#). See also rolled-up loop.

## Referenced Documents

This chapter references the following documents:

- *AN391: Profiling Nios II Systems*
- *Cache and Tightly-Coupled Memory* in the *Nios II Processor Reference Handbook*
- *Exception Handling* chapter in the *Nios II Software Developer's Handbook*
- *Nios II C2H Compiler User Guide*
- *Nios II Hardware Development Tutorial*
- *Nios II Processor Reference Handbook*

## Document Revision History

[Table 5-7](#) shows the revision history for this chapter.

**Table 5-7.** Document Revision History

Date and Document Version	Changes Made	Summary of Changes
June 2008 v1.1	Corrected Table of Contents	—
March 2008 v1.0	Initial release	This chapter was previously released as <i>AN 420: Optimizing Nios II C2H Compiler Results</i> .

