Nios® II

Nios II Software Developer's Handbook

ALTERA®

101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
http://www.altera.com

NII5V2-6.1

I.S. EN ISO 9001

Contents

Section III. Advanced Programming Topics

Section IV. Appendices

Chapter Revision Dates

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

Chapter 1.	Overview	
	Revised:	*November 2006*
	Part number:	*NII52001-6.1.0*
Chapter 2.	Tour of the Nios II IDE	
	Revised:	*November 2006*
	Part number:	*NII52002-6.1.0*
Chapter 3.	Overview of the HAL System Library	
	Revised:	*November 2006*
	Part number:	*NII52003-6.1.0*
Chapter 4.	Developing Programs using the HAL	
	Revised:	*November 2006*
	Part number:	*NII52004-6.1.0*
Chapter 5.	Developing Device Drivers for the HAL	
	Revised:	*November 2006*
	Part number:	*NII52005-6.1.0*
Chapter 6.	Exception Handling	
	Revised:	*November 2006*
	Part number:	*NII52006-6.1.0*
Chapter 7.	Cache & Tightly-Coupled Memory	
	Revised:	*November 2006*
	Part number:	*NII52007-6.1.0*
Chapter 8.	MicroC/OS-II Real-Time Operating System	
	Revised:	*November 2006*
	Part number:	*NII52008-6.1.0*
Chapter 9.	Ethernet & the NicheStack® TCP/IP Stack - Nios II Edition	
	Revised:	*November 2006*
	Part number:	*NII52013-6.1.0*

Chapter 10.	HAL API Reference	
	Revised:	*November 2006*
	Part number:	*NII52010-6.1.0*
Chapter 11.	Altera-Provided Development Tools	
	Revised:	*November 2006*
	Part number:	*NII520011-6.1.0*
Chapter 12.	Read-Only Zip File System	
	Revised:	*November 2006*
	Part number:	*NII520012-6.1.0*
Chapter 13.	Ethernet & Lightweight IP	
	Revised:	*November 2006*
	Part number:	*NII52009-6.1.0*

About this Handbook

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

How to Contact Altera

For the most up-to-date information about Altera products, go to the Altera world-wide web site at www.altera.com. For technical support on this product, go to www.altera.com/mysupport. For additional information about Altera products, consult the sources shown below.

Information Type	USA & Canada	All Other Locations
Technical support	www.altera.com/mysupport/	www.altera.com/mysupport/
	(800) 800-EPLD (3753) (7:00 a.m. to 5:00 p.m. Pacific Time)	+1 408-544-8767 7:00 a.m. to 5:00 p.m. (GMT -8:00) Pacific Time
Product literature	www.altera.com	www.altera.com
Altera literature services	literature@altera.com	literature@altera.com
Non-technical customer service	(800) 767-3753	+ 1 408-544-7000 7:00 a.m. to 5:00 p.m. (GMT -8:00) Pacific Time
FTP site	ftp.altera.com	ftp.altera.com

Typographic Conventions

This document uses the typographic conventions shown below.

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

Section I. Nios II Software Development

This section introduces information for Nios® II software development.

This section includes the following chapters:

1. Overview

NII52001-6.1.0

Introduction

This chapter provides a high-level overview of the Nios® II processor for the software developer. This chapter introduces you to the Nios II software development environment, the tools available to you, and the process for developing software.

Getting Started

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

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

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

Development Environment

This section introduces the Nios II software development environment.

Tools

The Nios II software development environment provided by Altera consists of the following tools:

- Nios II IDE
- GNU Tool Chain

- Instruction Set Simulator
- Hardware Abstraction Layer System Library
- RTOS and TCP/IP stack
- Example Designs

Nios II IDE

The Nios II integrated development environment (IDE) is the software development graphical user interface (GUI) for the Nios II processor. All software development tasks can be accomplished within the Nios II IDE, including editing, building, and debugging programs. The Nios II IDE is the window through which all other tools can be launched.

The Nios II IDE is based on the popular Eclipse IDE framework and the Eclipse C development toolkit (CDT) plug-ins. The Nios II IDE is a thin user interface that manipulates other tools behind the scenes, shields you from the details of command-line tools, and presents a unified development environment. If necessary, software development processes can be scripted and executed independently of the GUI.

GNU Tool Chain

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

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

Instruction Set Simulator

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

Hardware Abstraction Layer System Library

The hardware abstraction layer (HAL) system library provides a UNIX-like hosted C runtime environment based on the standard ANSI C libraries. The HAL provides generic I/O devices, allowing you to write programs that access hardware using the C standard library routines, such as `printf()`. The HAL minimizes (or eliminates) the need to access hardware registers directly to control and communicate with peripherals.

Different individual hardware peripherals have different levels of HAL support, depending on the implementation of the HAL software drivers.

For complete details about the HAL, refer to the *The HAL System Library* section and the *HAL Application Program Interface Reference* chapter of the *Nios II Software Developer's Handbook*.

RTOS and TCP/IP stack

Altera provides ports of the MicroC/OS-II RTOS and the Lightweight IP TCP/IP stack. MicroC/OS-II is built on the thread-safe HAL system library, and implements a simple, well-documented RTOS scheduler. The TCP/IP stack is built on MicroC/OS-II, and implements the standard UNIX Sockets application programming interface (API). Several other operating systems and stacks are available from third-party vendors.

Example Designs

Documented software examples are provided to demonstrate all prominent features of the Nios II processor and the development environment.

Consistent Development Environment

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

Consistent Runtime Environment

The HAL system library provides a consistent, hosted C/C++ runtime environment, regardless of the underlying hardware features in the embedded system. A custom HAL system library, which serves as the board-support package, is generated automatically for each unique Nios II processor system. Therefore, you do not spend time manually writing drivers and board-support packages.

You can easily pare down the HAL runtime environment to bare essentials to achieve minimal code footprint. A freestanding C environment is also available if you want complete control over system initialization and device drivers for hardware interaction.

Third-Party Support

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

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

Migrating from the First-Generation Nios Processor

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

Further Nios II Information

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

- The *Nios II Processor Reference Handbook* defines the processor hardware architecture and features, including the instruction set architecture.
- The *Quartus® II Handbook, Volume 5: Embedded Peripherals* provides a reference for the peripherals distributed with the Nios II processor. This handbook describes the hardware structure and Nios II software drivers for each peripheral.
- The Nios II integrated development environment (IDE) provides tutorials and complete reference for using the features of the graphical user interface (GUI). The help system is available after launching the Nios II IDE.
- Altera's on-line solutions database, Find Answers, is an Internet resource that offers solutions to frequently asked questions via an easy-to-use search engine. Go to the support center on **www.altera.com** and click on **Find Answers.**
- Altera application notes and tutorials offer step-by-step instructions on using the Nios II processor for a specific application or purpose. These documents are often installed with Altera development kits, or are available from **www.altera.com.**

Document Revision History

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

Table 1–1. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
November 2006, v6.1.0	No change from previous release.	
May 2006, v6.0.0	No change from previous release.	
October 2005, v5.1.0	No change from previous release.	
May 2005, v5.0.0	No change from previous release.	
May 2004 v1.0	First publication.	

2. Tour of the Nios II IDE

NII52002-6.1.0

Introduction

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

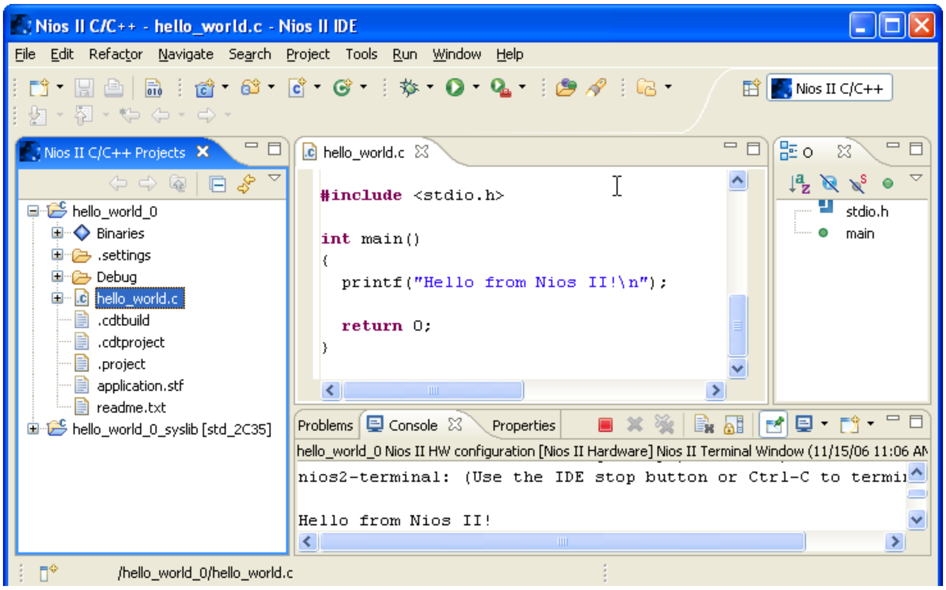
☞ Because of evolution and improvement of the software, the figures in this chapter might not match exactly what you see in the actual software.

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

The Nios II IDE Workbench

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

Figure 2–1. The Nios II IDE Workbench



```
#include <stdio.h>

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

  return 0;
}
```

```
hello_world_0 Nios II HW configuration [Nios II Hardware] Nios II Terminal Window (11/15/06 11:06 AM
nios2-terminal: (Use the IDE stop button or Ctrl-C to termi

Hello from Nios II!
```

Perspectives, Editors & Views

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

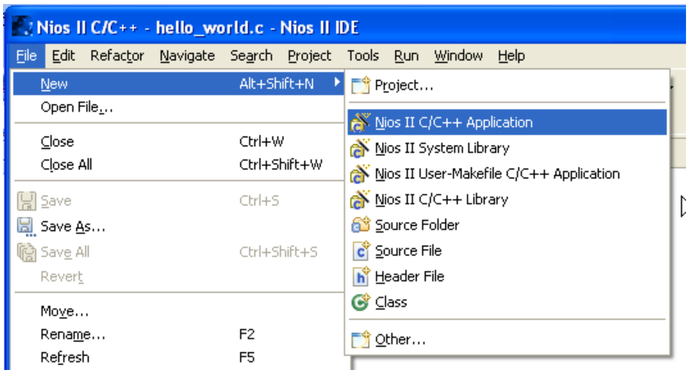
Most perspectives in the workbench comprise an editor area and one or more views. An editor allows you to open and edit a project resource (i.e., a file, folder, or project). Views support editors, provide alternative presentations, and ways to navigate the information in your workbench. Figure 2–1 shows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

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

Creating a New Project

The Nios II IDE provides a **New Project** wizard that guides you through the steps to create a new Nios II C/C++ application project. To start the Nios II C/C++ application **New Project** wizard, click **File**, **New**, **Nios II C/C++ Application**. See Figure 2–2.

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



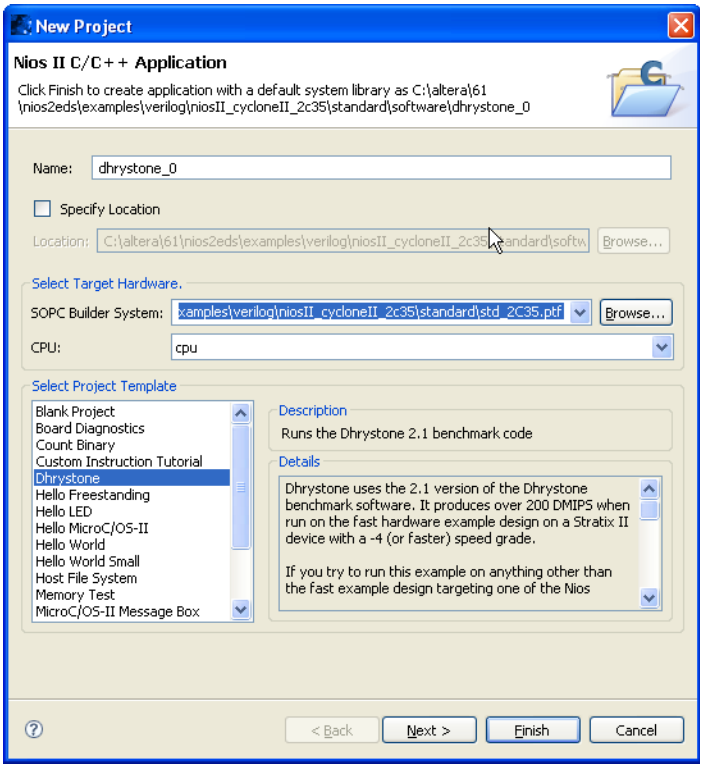
The Nios II C/C++ application **New Project** wizard prompts you to specify:

1. A name for your new Nios II project.
2. The target hardware.
3. A template for the new project.

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

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

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



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

Name: dhrystone_0

☐ Specify Location

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

Select Target Hardware.

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

CPU: cpu

Select Project Template

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

Description

Runs the Dhrystone 2.1 benchmark code

Details

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

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

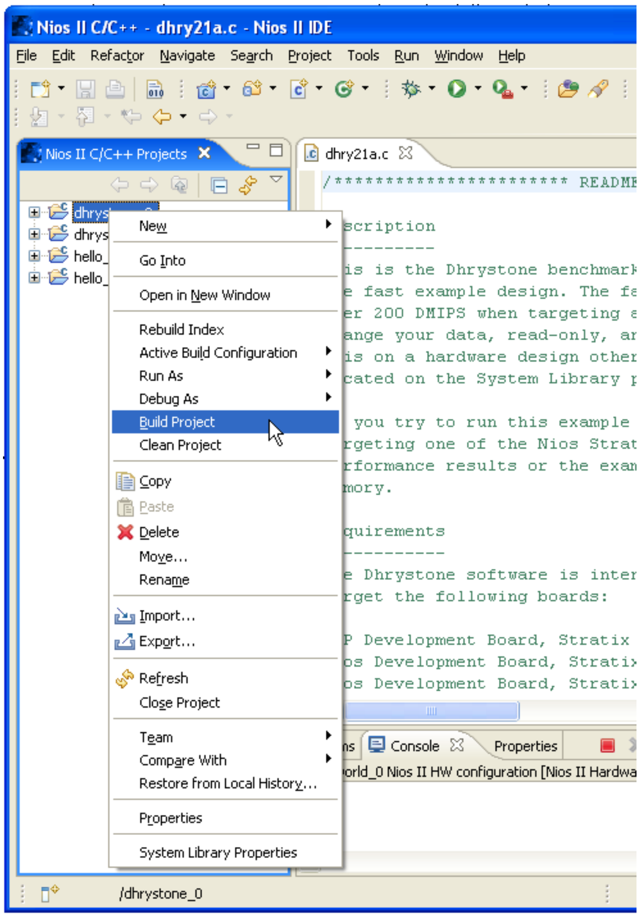
After you click **Finish**, the Nios II IDE creates the new project. The IDE also creates a system library project, ***_syslib** (for example, **dhrystone_0_syslib** for Figure 2–3). These projects show up in the **Nios II C/C++ Projects** view of the workbench.

Building & Managing Projects

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

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

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



```
/*********************** READMI

scription
---------
is is the Dhrystone benchmark
e fast example design. The fa
er 200 DMIPS when targeting a
ange your data, read-only, ar
is on a hardware design other
cated on the System Library p

 you try to run this example
rgeting one of the Nios Strat
rformance results or the exam
mory.

quirements
----------
e Dhrystone software is inter
rget the following boards:

P Development Board, Stratix
os Development Board, Stratix
os Development Board, Stratix
```

```
orld_0 Nios II HW configuration [Nios II Hardwa
```

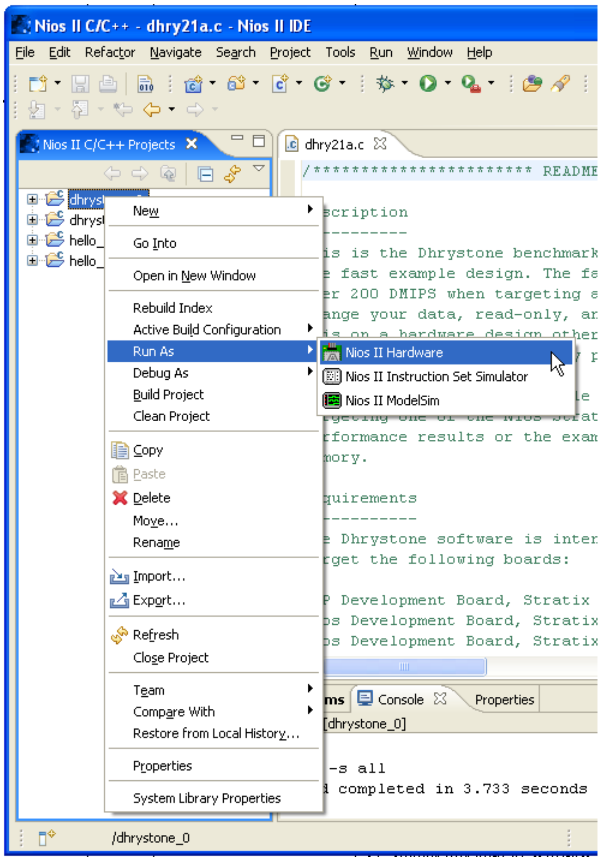
Right-clicking a project in **Nios II C/C++ Projects** also allows you to access the following important options for managing the project:

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

Running & Debugging Programs

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

Figure 2–5. Running a Program on Target Hardware

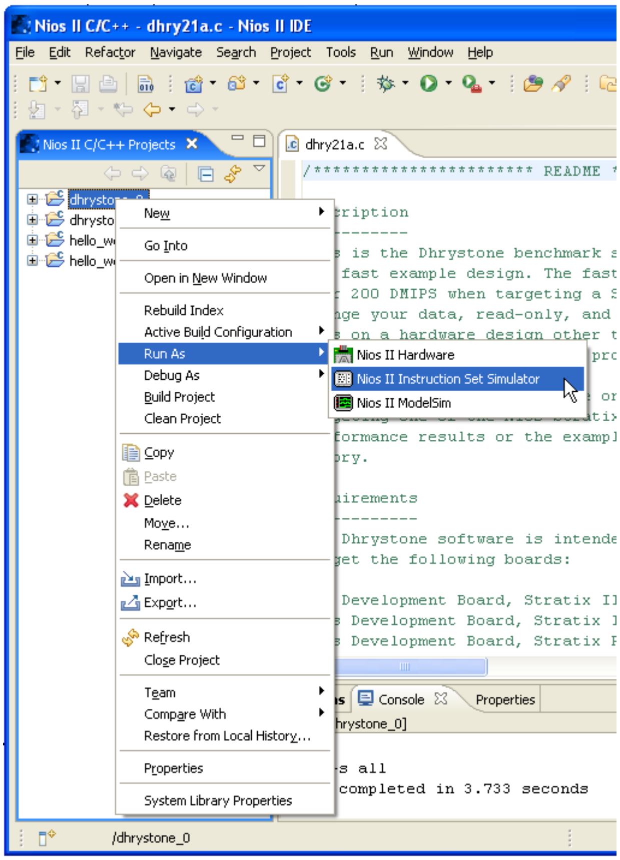


```
/*********************** README
scription
---------
is is the Dhrystone benchmark
e fast example design. The fa
er 200 DMIPS when targeting a
ange your data, read-only, an
is on a hardware design other
rgeting one of the Nios sorat
rformance results or the exam
mory.

quirements
----------
e Dhrystone software is inter
rget the following boards:

P Development Board, Stratix
os Development Board, Stratix
os Development Board, Stratix
```

```
-s all
d completed in 3.733 seconds
```

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

Figure 2–6. Launching the Instruction Set Simulator

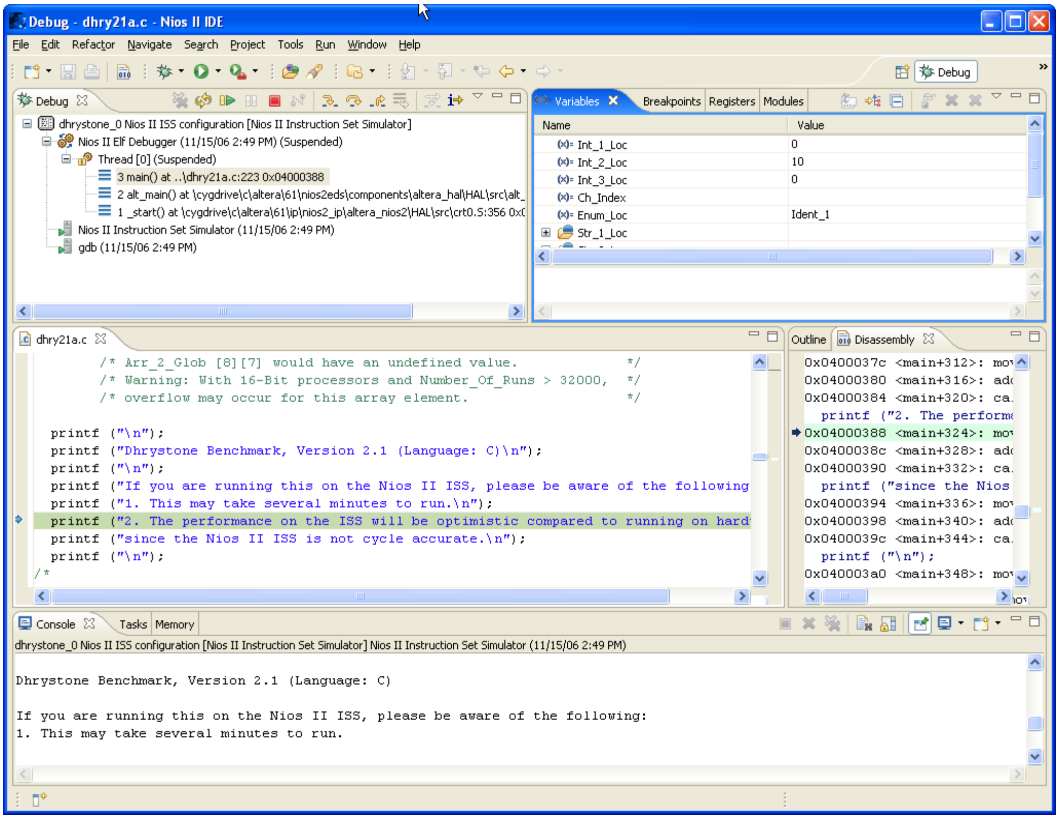


```
/************************ README *
cription
--------
s is the Dhrystone benchmark s
 fast example design. The fast
r 200 DMIPS when targeting a S
nge your data, read-only, and
s on a hardware design other t
```

```
s all
completed in 3.733 seconds
```

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

Figure 2–7. Debugging dhrystone_0 on the ISS


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

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

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

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

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

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

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

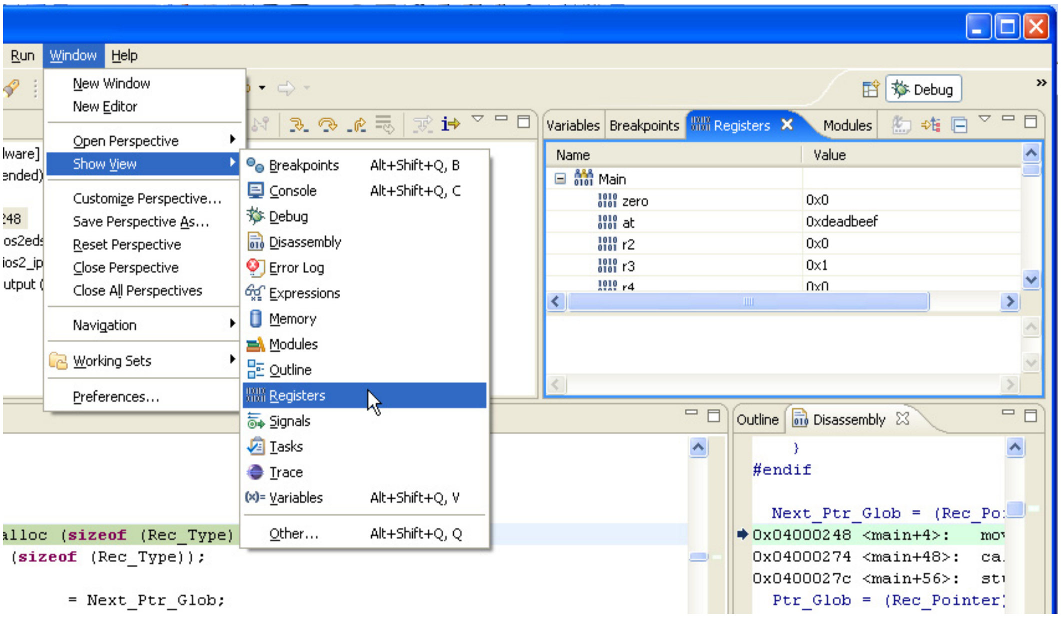
```
Dhrystone Benchmark, Version 2.1 (Language: C)

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

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

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

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

Figure 2–8. The Registers View While Debugging



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

      = Next_Ptr_Glob;
```

```
      }
#endif

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

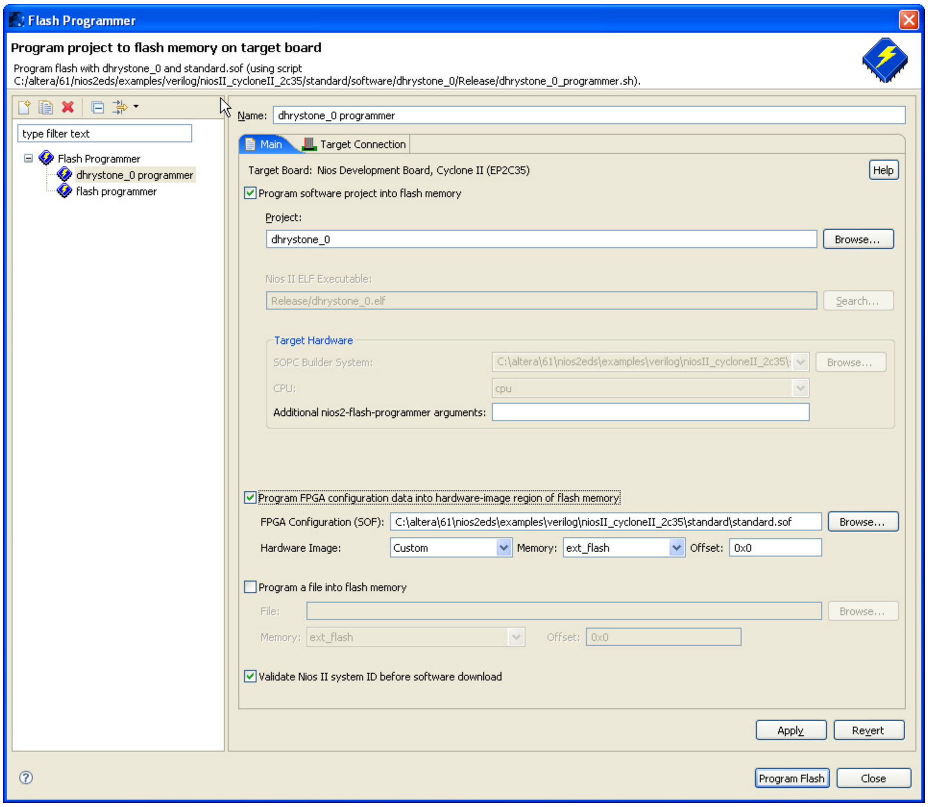
Programming Flash

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

- Program code
- Program data
- FPGA configuration data
- File systems

The Nios II IDE provides a Flash Programmer utility to help you manage and program the contents of flash memory. Figure 2–9 shows the Flash Programmer.

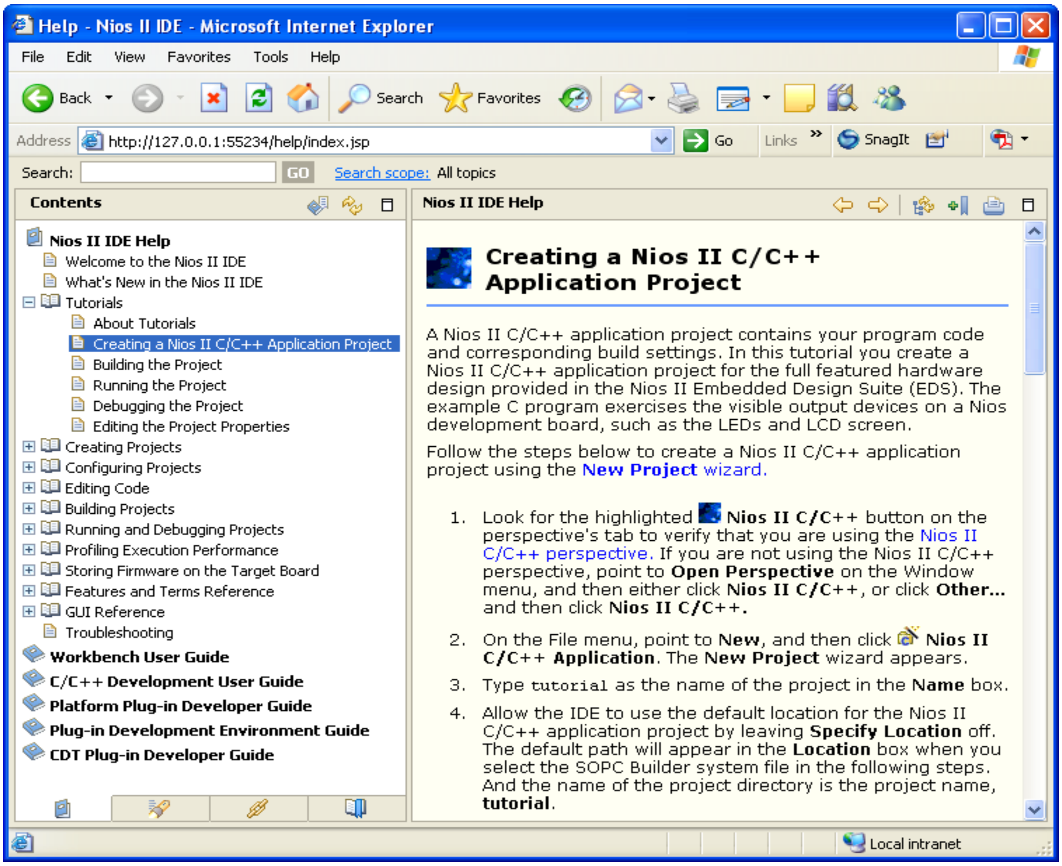
Figure 2–9. The Nios II IDE Flash Programmer


Flash Programmer
Program project to flash memory on target board
Program flash with dhrystone_0 and standard.sof (using script C:/altera/61/nios2eds/examples/verilog/niosII_cycloneII_2c35/standard/software/dhrystone_0/Release/dhrystone_0_programmer.sh).
type filter text
Flash Programmer
dhrystone_0 programmer
flash programmer
Name: dhrystone_0 programmer
Main
Target Connection
Target Board: Nios Development Board, Cyclone II (EP2C35)
Help
Program software project into flash memory
Project: dhrystone_0
Browse...
Nios II ELF Executable: Release/dhrystone_0.elf
Search...
Target Hardware
SOPC Builder System: C:\altera\61\nios2eds\examples\verilog\niosII_cycloneII_2c35\
Browse...
CPU: cpu
Additional nios2-flash-programmer arguments:
Program FPGA configuration data into hardware-image region of flash memory
FPGA Configuration (SOF): C:\altera\61\nios2eds\examples\verilog\niosII_cycloneII_2c35\standard\standard.sof
Browse...
Hardware Image: Custom
Memory: ext_flash
Offset: 0x0
Program a file into flash memory
File:
Browse...
Memory: ext_flash
Offset: 0x0
Validate Nios II system ID before software download
Apply
Revert
Program Flash
Close

Help System

The Nios II IDE help system provides documentation on all IDE-related topics. To launch the help system, click **Help**, **Help Contents**. On Windows, you can also press **F1** at any time for context-sensitive help. The Nios II IDE help system contains hands-on tutorials that guide you step-by-step through the process of creating, building, and debugging a Nios II project. Figure 2–10 shows the Nios II IDE help system displaying a tutorial.

Figure 2–10. Tutorials in the Nios II IDE Help System


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

Creating a Nios II C/C++ Application Project

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

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

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

Document Revision History

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

Table 2–1. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
November 2006, v6.1.0	Describes updated look and feel, including Nios II C/C++ perspective and Nios II C/C++ Projects views, renamed project types.	Updated look and feel based on Eclipse 3.2.
May 2006, v6.0.0	No change from previous release.	
October 2005, v5.1.0	Updated for the Nios II IDE version 5.1.	
May 2005, v5.0.0	No change from previous release.	
September 2004 v1.1	Updated screen shots.	
May 2004 v1.0	First publication.	

Section II. The HAL System Library

This section provides information on the hardware abstraction layer (HAL) system library.

This section includes the following chapters:

3. Overview of the HAL System Library

NII52003-6.1.0

Introduction

This chapter introduces the hardware abstraction layer (HAL) system library for the Nios® II processor.

The HAL system library is a lightweight runtime environment that provides a simple device driver interface for programs to communicate with the underlying hardware. The HAL application program interface (API) is integrated with the ANSI C standard library. The HAL API allows you to access devices and files using familiar C library functions, such as `printf()`, `fopen()`, `fwrite()`, etc.

The HAL serves as a board-support package for Nios II processor systems, providing a consistent interface to the peripherals in your embedded systems. Tight integration between SOPC Builder and Nios II integrated development environment (IDE) allows the IDE to generate the HAL system library for you. After SOPC Builder generates a hardware system, the Nios II IDE can generate a custom HAL system library to match the hardware configuration. Furthermore, changes in the hardware configuration automatically propagate to the HAL device driver configuration, eliminating frustrating bugs that appear due to subtle changes in the underlying hardware.

HAL device driver abstraction provides a clear distinction between application and device driver software. This driver abstraction promotes reusable application code that is resistant to changes in the underlying hardware. In addition, it is easy to write drivers for new hardware peripherals that are consistent with existing peripheral drivers.

Getting Started

The easiest way to get started using the HAL is to perform the tutorials provided with the Nios II IDE. In the process of creating a new project in the Nios II IDE, you also create a HAL system library. You do not have to create or copy HAL files, and you do not have to edit any of the HAL source code. The Nios II IDE generates and manages the HAL system library for you.

You must base the HAL system library on a specific SOPC Builder system. An SOPC Builder system refers to the Nios II processor core integrated with peripherals and memory (which is generated by SOPC Builder). If you do not have a custom SOPC Builder system, you can base your project on an Altera®-provided example hardware system. In fact, you can first start developing projects targeting an Altera® Nios development board, and later re-target the project to a custom board. It is easy to change the target SOPC Builder system later.

For details on starting a new project, refer to the help system in the Nios II IDE.

HAL Architecture

This section describes the fundamental elements of the HAL architecture.

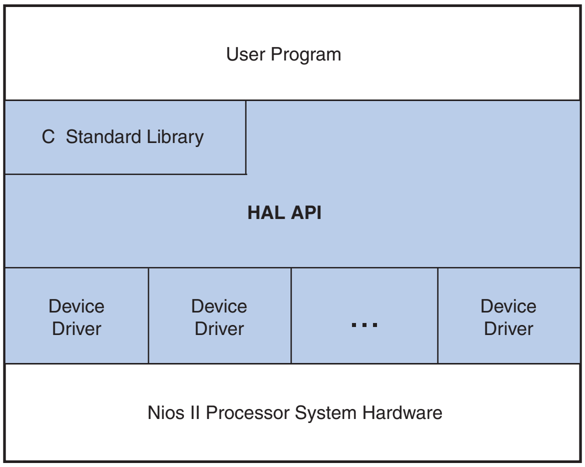
Services

The HAL system library provides the following services:

- *Integration with the newlib ANSI C standard library*—provides the familiar C standard library functions
- *Device drivers*—provides access to each device in the system
- *The HAL API*—provides a consistent, standard interface to HAL services, such as device access, interrupt handling, and alarm facilities
- *System initialization*—performs initialization tasks for the processor and the runtime environment before `main()`
- *Device initialization*—instantiates and initializes each device in the system before `main()`

Figure 3–1 shows the layers of a HAL-based system, from the hardware level up to a user program.

Figure 3–1. The Layers of a HAL-Based System



Applications vs. Drivers

Programmers fall into two distinct groups: application developers and device driver developers. Application developers are the majority of users, and are responsible for writing the system's `main()` routine, among other routines. Applications interact with system resources either through the C standard library, or through the HAL system library API. Device driver developers are responsible for making device resources available to application developers. Device drivers communicate directly with hardware through low-level hardware-access macros.

For this reason, the main HAL documentation is separated into the following two main chapters:

- The *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook* describes how to take advantage of the HAL to write programs without considering the underlying hardware.
- The *Developing Device Drivers for the HAL* chapter of the *Nios II Software Developer's Handbook* describes how to communicate directly with hardware and how to make hardware resources available via the abstracted HAL API.

Generic Device Models

The HAL provides generic device models for classes of peripherals found in embedded systems, such as timers, Ethernet MAC/PHY chips, and I/O peripherals that transmit character data. The generic device models are at the core of the HAL system library's power. The generic device models allow you to write programs using a consistent API, regardless of the underlying hardware.

Device Model Classes

The HAL provides a model for the following classes of devices:

- Character-mode devices—hardware peripherals that send and/or receive characters serially, such as a UART.
- Timer devices—hardware peripherals that count clock ticks and can generate periodic interrupt requests.
- File subsystems—provide a mechanism for accessing files stored within physical device(s). Depending on the internal implementation, the file subsystem driver might access the underlying device(s) directly or use a separate device driver. For example, you can write a flash file subsystem driver that accesses flash using the HAL API for flash memory devices.

- Ethernet devices—provide access to an Ethernet connection for a networking stack such as the Altera-provided NicheStack® TCP/IP Stack - Nios II Edition. You need a networking stack to use an ethernet device.
- DMA devices—peripherals that perform bulk data transactions from a data source to a destination. Sources and destinations can be memory or another device, such as an Ethernet connection.
- Flash memory devices—nonvolatile memory devices that use a special programming protocol to store data.

Benefits to Application Developers

The HAL system library defines a set of functions that you use to initialize and access each class of device. The API is consistent, regardless of the underlying implementation of the device hardware. For example, to access character-mode devices and file subsystems, you can use the C standard library functions, such as `printf()` and `fopen()`. For application developers, you do not have to write low-level routines just to establish basic communication with the hardware for these classes of peripherals.

Benefits to Device Driver Developers

Each device model defines a set of driver functions necessary to manipulate the particular class of device. If you are writing drivers for a new peripheral, you only need to provide this set of driver functions. As a result, your driver development task is pre-defined and well documented. In addition, you can use existing HAL functions and applications to access the device, which saves software development effort. The HAL system library calls driver functions to access hardware. Application programmers call the ANSI C or HAL API to access hardware, rather than calling your driver routines directly. Therefore, the usage of your driver is already documented as part of the HAL API.

C Standard Library—Newlib

The HAL system library integrates the ANSI C standard library into its runtime environment. The HAL uses newlib, an open-source implementation of the C standard library. Newlib is a C library for use on embedded systems, making it a perfect match for the HAL and the Nios II processor. Newlib licensing does not require you to release your source code or pay royalties for projects based on newlib.

The ANSI C standard library is well documented. Perhaps the most well-known reference is *The C Programming Language* by B. Kernighan & D. Ritchie, published by Prentice Hall and available in over 20 languages. Redhat also provides online documentation for newlib at **http://sources.redhat.com/newlib**.

Supported Peripherals

Altera provides many peripherals for use in Nios II processor systems. Most Altera peripherals provide HAL device drivers that allow you to access the hardware via the HAL API. The following Altera peripherals provide full HAL support:

- Character mode devices:
  - UART core
  - JTAG UART core
  - LCD 16207 display controller
- Flash memory devices
  - Common flash interface compliant flash chips
  - Altera's EPCS serial configuration device controller
- File subsystems
  - Altera host based file system
  - Altera zip read-only file system
- Timer devices
  - Timer core
- DMA devices
  - DMA controller core
- Ethernet devices
  - LAN91C111 Ethernet MAC/PHY Controller

The LAN91C111 component requires the MicroC/OS-II runtime environment. For more information, refer to the *Ethernet & the NicheStack® TCP/IP Stack - Nios II Edition* chapter of the *Nios II Software Developer's Handbook.*

Third-party vendors offer additional peripherals not listed here. For a list of other peripherals available for the Nios II processor, visit Altera's Embedded Software Partners page at **http://www.altera.com/products/ip/processors/nios2/tools/embed-partners/ni2-embed-partners.html**.

All peripherals (both from Altera and third party vendors) must provide a header file that defines the peripheral's low-level interface to hardware. By this token, all peripherals support the HAL to some extent. However, some peripherals might not provide device drivers. If drivers are not available, use only the definitions provided in the header files to access the hardware. Do not access a peripheral using hard-coded addresses or other such "magic numbers".

Inevitably certain peripherals have hardware-specific features with usage requirements that do not map well to a general-purpose API. The HAL system library handles hardware-specific requirements by providing the UNIX-style `ioctl()` function. Because the hardware features depend on the peripheral, the `ioctl()` options are documented in the description for each peripheral.

Some peripherals provide dedicated accessor functions that are not based on the HAL generic device models. For example, Altera provides a general-purpose parallel I/O (PIO) core for use in Nios II processor system. The PIO peripheral does not fit into any class of generic device models provided by the HAL, and so it provides a header file and a few dedicated accessor functions only.

For complete details regarding software support for a peripheral, refer to the peripheral's description. For further details on Altera-provided peripherals, see the *Quartus® II Handbook, Volume 5: Embedded Peripherals*.

Document Revision History

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

Table 3–1. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
November 2006, v6.1.0	● NicheStack TCP/IP Stack - Nios II Edition	
May 2006, v6.0.0	No change from previous release.	
October 2005, v5.1.0	No change from previous release.	
May 2005, v5.0.0	No change from previous release.	
May 2004 v1.0	First publication.	

4. Developing Programs using the HAL

NII52004-6.1.0

Introduction

This chapter discusses how to develop programs based on the Altera® hardware abstraction layer (HAL) system library.

The API for HAL-based systems is readily accessible to software developers who are new to the Nios® II processor. Programs based on the HAL use the ANSI C standard library functions and runtime environment, and access hardware resources via the HAL API's generic device models. The HAL API largely conforms to the familiar ANSI C standard library functions, though the ANSI C standard library is separate from the HAL system library. The close integration of the ANSI C standard library and the HAL makes it possible to develop useful programs that never call the HAL system library functions directly. For example, you can manipulate character mode devices and files using the ANSI C standard library I/O functions, such as `printf()` and `scanf()`.

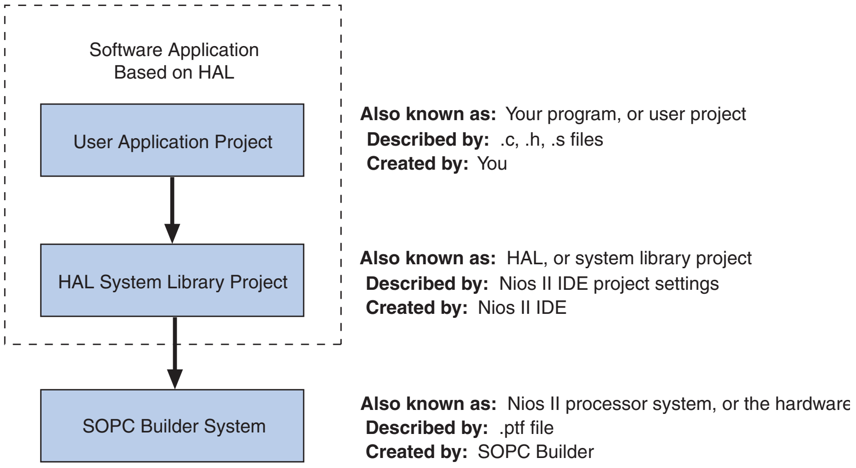
☞ This document does not cover the ANSI C standard library. An excellent reference is *The C Programming Language, Second Edition*, by Kernighan and Ritchie.

The Nios II IDE Project Structure

The creation and management of software projects based on the HAL system library is integrated tightly with the Nios II integrated development environment (IDE). This section discusses the Nios II IDE projects as a basis for understanding the HAL.

Figure 4–1 shows the blocks of a Nios II program with emphasis on how the HAL system library fits in. The label for each block describes what or who generated that block, and an arrow points to each block's dependency.

Figure 4–1. The Nios II IDE Project Structure


Software Application
Based on HAL
User Application Project
HAL System Library Project
SOPC Builder System
Also known as: Your program, or user project
Described by: .c, .h, .s files
Created by: You
Also known as: HAL, or system library project
Described by: Nios II IDE project settings
Created by: Nios II IDE
Also known as: Nios II processor system, or the hardware
Described by: .ptf file
Created by: SOPC Builder

HAL-based Nios II programs consist of two Nios II IDE projects. See Figure 4–1. Your program is contained in one project (the user application project), and it depends on a separate system library project (the HAL system library project).

The application project contains all the code you develop. The executable image for your program ultimately results from building this project.

The Nios II IDE creates the HAL system library project when you create your application project. The HAL system library project contains all information needed to interface your program to the hardware. At build time, all HAL drivers relevant to your SOPC Builder system are added to the system library project. HAL settings are stored as system library properties.

The system library project depends on the SOPC Builder system, defined by a **.ptf** file generated by SOPC Builder. The Nios II IDE manages the HAL system library and updates the driver configurations to accurately reflect the system hardware. If the SOPC Builder system changes (i.e., the **.ptf** file is updated), the IDE rebuilds the HAL system library the next time you build or run your application program.

This project dependency structure isolates your program from changes to the underlying hardware, and you can develop and debug code without having to worry about whether your program matches the target hardware. In short, programs based on a HAL system library are always synchronized with the target hardware.

The system.h System Description File

The **system.h** file is the foundation of the HAL system library. The **system.h** file provides a complete software description of the Nios II system hardware. It serves as the hand-off point between the hardware and software design processes. Not all information in **system.h** is useful to you as a programmer, and it is rarely necessary to include it explicitly in your C source files. Nonetheless, **system.h** holds the answer to the fundamental question, “What hardware is present in this system?”

The **system.h** file describes each peripheral in the system and provides the following details:

- The hardware configuration of the peripheral
- The base address
- The IRQ priority (if any)
- A symbolic name for the peripheral

The Nios II IDE generates the **system.h** file for HAL system library projects. The contents of **system.h** depend on both the hardware configuration and the HAL system library properties you set in the Nios II IDE. Do not edit **system.h**.

See the Nios II IDE help system for details of how to set the system library properties.

The following code from a **system.h** file shows some of the hardware configuration options it defines.

Example: Excerpts from a system.h File

```
/*
 * sys_clk_timer configuration
 *
 */

#define SYS_CLK_TIMER_NAME "/dev/sys_clk_timer"
#define SYS_CLK_TIMER_TYPE "altera_avalon_timer"
#define SYS_CLK_TIMER_BASE 0x00920800
#define SYS_CLK_TIMER_IRQ 0
#define SYS_CLK_TIMER_ALWAYS_RUN 0
#define SYS_CLK_TIMER_FIXED_PERIOD 0

/*
 * jtag_uart configuration
 *
 */

#define JTAG_UART_NAME "/dev/jtag_uart"
#define JTAG_UART_TYPE "altera_avalon_jtag_uart"
#define JTAG_UART_BASE 0x00920820
#define JTAG_UART_IRQ 1
```

Data Widths & the HAL Type Definitions

For embedded processors such as the Nios II processor, it is often important to know the exact width and precision of data. Because the ANSI C data types do not explicitly define data width, the HAL uses a set of standard type definitions instead. The ANSI C types are supported, but their data widths are dependent on the compiler's convention.

The header file **alt_types.h** defines the HAL type definitions; Table 4–1 shows the HAL type definitions.

Table 4–1. The HAL Type Definitions

Type	Meaning
`alt_8`	Signed 8-bit integer.
`alt_u8`	Unsigned 8-bit integer.
`alt_16`	Signed 16-bit integer.
`alt_u16`	Unsigned 16-bit integer.
`alt_32`	Signed 32-bit integer.
`alt_u32`	Unsigned 32-bit integer.
`alt_64`	Signed 64-bit integer.
`alt_u64`	Unsigned 64-bit integer.

Table 4–2 shows the data widths that the Altera-provided GNU toolchain uses.

Table 4–2. GNU Toolchain Data Widths

Type	Meaning
`char`	8 bits.
`short`	16 bits.
`long`	32 bits.
`int`	32 bits.

UNIX-Style Interface

The HAL API provides a number of UNIX-style functions. The UNIX-style functions provide a familiar development environment for new Nios II programmers, and can ease the task of porting existing code to run under the HAL environment. The HAL primarily uses these functions to provide the system interface for the ANSI C standard library. For example, the functions perform device access required by the C library functions defined in **stdio.h**.

The following list is the complete list of the available UNIX-style functions:

- `_exit()`
- `close()`
- `fstat()`
- `getpid()`
- `gettimeofday()`
- `ioctl()`
- `isatty()`
- `kill()`
- `lseek()`
- `open()`
- `read()`
- `sbrk()`
- `settimeofday()`
- `stat()`
- `usleep()`
- `wait()`
- `write()`

The most commonly used functions are those that relate to file I/O. See "File System" on page 4–5.

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

File System

The HAL provides infrastructure for UNIX-style file access. You can use this infrastructure to build a file system on any storage devices available in your hardware.

For an example, see the *Read-Only Zip File System* chapter.

You can access files within a HAL-based file system by using either the C standard library file I/O functions in the newlib C library (for example `fopen()`, `fclose()`, and `fread()`), or using the UNIX-style file I/O provided by the HAL system library.

The HAL provides the following UNIX style functions for file manipulation:

- `close()`
- `fstat()`
- `ioctl()`
- `isatty()`
- `lseek()`

- `open()`
- `read()`
- `stat()`
- `write()`

For more information on these functions, refer to the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

The HAL registers a file subsystem as a mount point within the global HAL file system. Attempts to access files below that mount point are directed to the file subsystem. For example, if a read-only zip file subsystem (**zipfs**) is mounted as **/mount/zipfs0**, the **zipfs** file subsystem handles calls to `fopen()` for **/mount/zipfs0/myfile**.

There is no concept of a current directory. Software must access all files using absolute paths.

The HAL file infrastructure also allows you to manipulate character mode devices via UNIX-style path names. The HAL registers character mode devices as nodes within the HAL file system. By convention, **system.h** defines the name of a device node as the prefix **/dev/** plus the name assigned to the hardware component in SOPC builder. For example, a UART peripheral **uart1** in SOPC builder is **/dev/uart1** in **system.h**.

The following code shows reading characters from a read-only zip file subsystem **rozipfs** that is registered as a node in the HAL file system. The standard header files `stdio.h`, `stddef.h`, and `stdlib.h` are installed with the HAL.

Example: Reading Characters from a File Subsystem

```
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>

#define BUF_SIZE (10)

int main(void)
{
  FILE* fp;
  char buffer[BUF_SIZE];

  fp = fopen ("/mount/rozipfs/test", "r");
  if (fp == NULL)
  {
    printf ("Cannot open file.\n");
    exit (1);
  }

  fread (buffer, BUF_SIZE, 1, fp);
  fclose (fp);

  return 0;
}
```

For more information on the use of these functions, refer to the newlib C library documentation installed with the Nios II Embedded Design Suite (EDS). On the Windows **Start** menu, click **Programs**, **Altera**, **Nios II** *<version>*, **Nios II Documentation**.

Using Character-Mode Devices

A character-mode device is a hardware peripheral that sends and/or receives characters serially. A common example is the universal asynchronous receiver/transmitter (UART). Character mode devices are registered as nodes within the HAL file system. In general, a program associates a file descriptor to a device's name, and then writes and reads characters to or from the file using the ANSI C file operations defined in **file.h**. The HAL also supports the concept of standard input, standard output, and standard error, allowing programs to call the **stdio.h** I/O functions.

Standard Input, Standard Output & Standard Error

Using standard input (`stdin`), standard output (`stdout`), and standard error (`stderr`) is the easiest way to implement simple console I/O. The HAL system library manages `stdin`, `stdout`, and `stderr` behind the scenes, which allows you to send and receive characters through these channels without explicitly managing file descriptors. For example, the system library directs the output of `printf()` to standard out, and `perror()` to standard error.

You associate each channel to a specific hardware device by setting system library properties in the Nios II IDE.

For more information, see the Nios II IDE help system.

The following code shows the classic Hello World program. This program sends characters to whatever device is associated with `stdout` when compiled in Nios II IDE.

Example: Hello World

```
#include <stdio.h>
int main ()
{
  printf ("Hello world!");
  return 0;
}
```

When using the UNIX-style API, you can use the file descriptors `stdin`, `stdout`, and `stderr`, defined in **unistd.h**, to access, respectively, the standard in, standard out, and standard error character I/O streams. **unistd.h** is installed with the Nios II EDS as part of the newlib C library package.

General Access to Character Mode Devices

Accessing a character-mode device (besides `stdin`, `stdout`, or `stderr`) is as easy as opening and writing to a file. The following code demonstrates writing a message to a UART called `uart1`.

Example: Writing Characters to a UART

```
#include <stdio.h>
#include <string.h>

int main (void)
{
  char* msg = "hello world";
  FILE* fp;

  fp = fopen ("/dev/uart1", "w");
  if (fp!=NULL)
  {
    fprintf(fp, "%s",msg);
    fclose (fp);
  }
  return 0;
}
```

C++ Streams

HAL-based systems can use the C++ streams API for manipulating files from C++.

/dev/null

All systems include the device **/dev/null**. Writing to **/dev/null** has no effect, and all data is discarded. **/dev/null** is used for safe I/O redirection during system startup. This device could also be useful for applications that wish to sink unwanted data.

This device is purely a software construct. It does not relate to any physical hardware device within the system.

Lightweight Character-Mode I/O

The HAL offers several methods of reducing the code footprint of character-mode device drivers. For details, see "Reducing Code Footprint" on page 4–25.

Using File Subsystems

The HAL generic device model for file subsystems allows access to data stored in an associated storage device using the C standard library file I/O functions. For example the Altera zip read-only file system provides read-only access to a file system stored in flash memory.

A file subsystem is responsible for managing all file I/O access beneath a given mount point. For example, if a file subsystem is registered with the mount point **/mnt/rozipfs**, all file access beneath this directory, such as `fopen("/mnt/rozipfs/myfile", "r")`, is directed to that file subsystem.

As with character mode devices, you can manipulate files within a file subsystem using the C file I/O functions defined in **file.h**, such as `fopen()` and `fread()`.

For more information on the use of these functions, refer to the newlib C library documentation installed with the Nios II EDS. On the Windows Start menu, click Programs, Altera, Nios II *<version>*, **Nios II Documentation**.

Using Timer Devices

Timer devices are hardware peripherals that count clock ticks and can generate periodic interrupt requests. You can use a timer device to provide a number of time-related facilities, such as the HAL system clock, alarms, the time-of-day, and time measurement. To use the timer facilities, the Nios II processor system must include a timer peripheral in hardware.

The HAL API provides two types of timer device drivers:

- System clock driver. This type of driver supports alarms, such as you would use in a scheduler.
- Timestamp driver. This driver supports high-resolution time measurement.

An individual timer peripheral can behave as either a system clock or a timestamp, but not both.

The HAL-specific API functions for accessing timer devices are defined in **sys/alt_alarm.h** and **sys/alt_timestamp.h**.

System Clock Driver

The HAL system clock driver provides a periodic "heartbeat", causing the system clock to increment on each beat. Software can use the system clock facilities to execute functions at specified times, and to obtain timing information. You select a specific hardware timer peripheral as the system clock device by setting system library properties in the Nios II IDE.

For more information, see the Nios II IDE help system.

The HAL provides implementations of the following standard UNIX functions: `gettimeofday()`, `settimeofday()`, and `times()`. The times returned by these functions are based on the HAL system clock.

The system clock measures time in units of "ticks". For embedded engineers who deal with both hardware and software, do not confuse the HAL system clock with the clock signal driving the Nios II processor hardware. The period of a HAL system clock tick is generally much longer than the hardware system clock. **system.h** defines the clock tick frequency.

At runtime, you can obtain the current value of the system clock by calling the `alt_nticks()` function. This function returns the elapsed time in system clock ticks since reset. You can get the system clock rate, in ticks per second, by calling the function `alt_ticks_per_second()`. The HAL timer driver initializes the tick frequency when it creates the instance of the system clock.

The standard UNIX function `gettimeofday()` is available to obtain the current time. You must first calibrate the time of day by calling `settimeofday()`. In addition, you can use the `times()` function to obtain information on the number of elapsed ticks. The prototypes for these functions appear in **times.h**.

For more information on the use of these functions, refer to the HAL *API Reference* chapter of the *Nios II Software Developer's Handbook*.

Alarms

You can register functions to be executed at a specified time using the HAL alarm facility. A software program registers an alarm by calling the function `alt_alarm_start()`:

```
int alt_alarm_start (alt_alarm* alarm,
                     alt_u32    nticks,
                     alt_u32    (*callback) (void* context),
                     void*      context);
```

The function `callback` is called after `nticks` have elapsed. The input argument `context` is passed as the input argument to `callback` when the call occurs. `alt_alarm_start()` initializes the structure pointed to by the input argument `alarm`. You do not have to initialize it.

The callback function can reset the alarm. The return value of the registered callback function is the number of ticks until the next call to `callback`. A return value of zero indicates that the alarm should be stopped. You can manually cancel an alarm by calling `alt_alarm_stop()`.

Alarm callback functions execute in an interrupt context. This imposes functional restrictions which you must observe when writing an alarm callback.

For more information on the use of these functions, refer to the *Exception Handling* chapter of the *Nios II Software Developer's Handbook*.

The following code fragment demonstrates registering an alarm for a periodic callback every second.

Example: Using a Periodic Alarm Callback Function

```
#include <stddef.h>
#include <stdio.h>
#include "sys/alt_alarm.h"
#include "alt_types.h"

/*
 * The callback function.
 */

alt_u32 my_alarm_callback (void* context)
{
  /* This function will be called once/second */
  return alt_ticks_per_second();
}

...

/* The alt_alarm must persist for the duration of the alarm. */
static alt_alarm alarm;

...

  if (alt_alarm_start (&alarm,
                       alt_ticks_per_second(),
                       my_alarm_callback,
                       NULL) < 0)
  {
    printf ("No system clock available\n");
  }
```

Timestamp Driver

Sometimes you want to measure time intervals with a degree of accuracy greater than that provided by HAL system clock ticks. The HAL provides high resolution timing functions using a timestamp driver. A timestamp driver provides a monotonically increasing counter that you can sample to obtain timing information. The HAL only supports one timestamp driver in the system.

You select a specific hardware timer peripheral as the timestamp device by setting system library properties in the Nios II IDE.

For more information, see the Nios II IDE help system.

If a timestamp driver is present, the functions `alt_timestamp_start()` and `alt_timestamp()` become available. The Altera-provided timestamp driver uses the timer that you select on the system library properties page in the Nios II IDE.

Calling the function `alt_timestamp_start()` starts the counter running. Subsequent calls to `alt_timestamp()` then returns the current value of the timestamp counter. Calling `alt_timestamp_start()` again resets the counter to zero. The behavior of the timestamp driver is undefined when the counter reaches ($2^{32} – 1$).

You can obtain the rate at which the timestamp counter increments by calling the function `alt_timestamp_freq()`. This rate is typically the hardware frequency that the Nios II processor system runs at—usually millions of cycles per second. The timestamp drivers are defined in the **alt_timestamp.h** header file.

For more information on the use of these functions, refer to the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

The following code fragment shows how you can use the timestamp facility to measure code execution time.

Example: Using the Timestamp to Measure Code Execution Time

```
#include <stdio.h>
#include "sys/alt_timestamp.h"
#include "alt_types.h"

int main (void)
{
  alt_u32 time1;
  alt_u32 time2;
  alt_u32 time3;

  if (alt_timestamp_start() < 0)
```

```
    {
      printf ("No timestamp device available\n");
    }
    else
    {
      time1 = alt_timestamp();
      func1(); /* first function to monitor */
      time2 = alt_timestamp();
      func2(); /* second function to monitor */
      time3 = alt_timestamp();

      printf ("time in func1 = %u ticks\n",
              (unsigned int) (time2 - time1));
      printf ("time in func2 = %u ticks\n",
              (unsigned int) (time3 - time2));
      printf ("Number of ticks per second = %u\n",
              (unsigned int)alt_timestamp_freq());
    }
    return 0;
  }
```

Using Flash Devices

The HAL provides a generic device model for nonvolatile flash memory devices. Flash memories use special programming protocols to store data. The HAL API provides functions to write data to flash. For example, you can use these functions to implement a flash-based file subsystem.

The HAL API also provides functions to read flash, although it is generally not necessary. For most flash devices, programs can treat the flash memory space as simple memory when reading, and do not need to call special HAL API functions. If the flash device has a special protocol for reading data, such as the Altera EPCS serial configuration device, you must use the HAL API to both read and write data.

This section describes the HAL API for the flash device model. The following two APIs provide a different level of access to the flash:

- Simple flash access—functions that write buffers into flash and read them back at the block level. In writing, if the buffer is less than a full block, these functions erase pre-existing flash data above and below the newly written data.
- Fine-grained flash access—functions that write buffers into flash and read them back at the buffer level. In writing, if the buffer is less than a full block, these functions preserve pre-existing flash data above and below the newly written data. This functionality is generally required for managing a file subsystem.

The API functions for accessing flash devices are defined in **sys/alt_flash.h**.

For more information on the use of these functions, refer to the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*. For details of the Common Flash Interface, including the organization of CFI erase regions and blocks, see the JEDEC website at *www.jedec.org*. You can find the CFI standard by searching for document JESD68.

Simple Flash Access

This interface consists of the functions `alt_flash_open_dev()`, `alt_write_flash()`, `alt_read_flash()`, and `alt_flash_close_dev()`. The code "Example: Using the Simple Flash API Functions" on page 4–15 shows the usage of all of these functions in one code example. You open a flash device by calling `alt_flash_open_dev()`, which returns a file handle to a flash device. This function takes a single argument that is the name of the flash device, as defined in **system.h**.

Once you have obtained a handle, you can use the `alt_write_flash()` function to write data to the flash device. The prototype is:

```
int alt_write_flash(alt_flash_fd* fd,
                    int           offset,
                    const void*   src_addr,
                    int           length )
```

A call to this function writes to the flash device identified by the handle `fd`. The driver writes the data starting at `offset` bytes from the base of the flash device. The data written comes from the address pointed to by `src_addr`, the amount of data written is `length`.

There is also an `alt_read_flash()` function to read data from the flash device. The prototype is:

```
int alt_read_flash( alt_flash_fd* fd,
                    int           offset,
                    void*         dest_addr,
                    int           length )
```

A call to `alt_read_flash()` reads from the flash device with the handle `fd`, `offset` bytes from the beginning of the flash device. The function writes the data to location pointed to by `dest_addr`, and the amount of data read is `length`. For most flash devices, you can access the contents as standard memory, making it unnecessary to use `alt_read_flash()`.

The function alt_flash_close_dev() takes a file handle and closes the device. The prototype for this function is:

```
void alt_flash_close_dev(alt_flash_fd* fd )
```

The following code shows the usage of simple flash API functions to access a flash device named **/dev/ext_flash**, as defined in **system.h**.

Example: Using the Simple Flash API Functions

```
#include <stdio.h>
#include <string.h>
#include "sys/alt_flash.h"
#define BUF_SIZE 1024

int main ()
{
  alt_flash_fd* fd;
  int           ret_code;
  char          source[BUF_SIZE];
  char          dest[BUF_SIZE];

  /* Initialize the source buffer to all 0xAA */
  memset(source, 0xAA, BUF_SIZE);

  fd = alt_flash_open_dev("/dev/ext_flash");
  if (fd!=NULL)
  {
    ret_code = alt_write_flash(fd, 0, source, BUF_SIZE);
    if (ret_code==0)
    {
      ret_code = alt_read_flash(fd, 0, dest, BUF_SIZE);
      if (ret_code==0)
      {
        /*
         * Success.
         * At this point, the flash is all 0xAA and we
         * should have read that all back into dest
         */
      }
    }
    alt_flash_close_dev(fd);
  }
  else
  {
    printf("Can't open flash device\n");
  }
  return 0;
}
```

Block Erasure or Corruption

Generally, flash memory is divided into blocks. `alt_write_flash()` might need to erase the contents of a block before it can write data to it. In this case, it makes no attempt to preserve the existing contents of a block. This action can lead to unexpected data corruption (erasure), if you are performing writes that do not fall on block boundaries. If you wish to preserve existing flash memory contents, use the fine-grained flash functions. See "Fine-Grained Flash Access" on page 4–16.

Table 4–3 shows how you can cause unexpected data corruption by writing using the simple flash-access functions. Table 4–3 shows the example of an 8 Kbyte flash memory comprising two 4 Kbyte blocks. First write 5 Kbytes of all `0xAA` into flash memory at address `0x0000`, and then write 2 Kbytes of all `0xBB` to address `0x1400`. After the first write succeeds (at time t(2)), the flash memory contains 5 Kbyte of `0xAA`, and the rest is empty (i.e., `0xFF`). Then the second write begins, but before writing into the second block, the block is erased. At this point, t(3), the flash contains 4 Kbyte of `0xA` and 4 Kbyte of `0xFF`. After the second write finishes, at time t(4), the 2 Kbyte of `0xFF` at address `0x1000` is corrupted.

Table 4–3. Example of Writing Flash & Causing Unexpected Data Corruption

Address	Block	Time t(0)	Time t(1)	Time t(2)	Time t(3)	Time t(4)
		Before First Write	First Write		Second Write	
			After Erasing Block(s)	After Writing Data 1	After Erasing Block(s)	After Writing Data 2
`0x0000`	1	??	FF	AA	AA	AA
`0x0400`	1	??	FF	AA	AA	AA
`0x0800`	1	??	FF	AA	AA	AA
`0x0C00`	1	??	FF	AA	AA	AA
`0x1000`	2	??	FF	AA	FF	FF *(1)*
`0x1400`	2	??	FF	FF	FF	BB
`0x1800`	2	??	FF	FF	FF	BB
`0x1C00`	2	??	FF	FF	FF	FF

Notes to Table 4–3:
(1) Unintentionally cleared to FF during erasure for second write.

Fine-Grained Flash Access

There are three additional functions that provide complete control over writing flash contents at the highest granularity: `alt_get_flash_info()`, `alt_erase_flash_block()`, and `alt_write_flash_block()`.

By the nature of flash memory, you cannot erase a single address within a block. You must erase (i.e., set to all ones) an entire block at a time. Writing to flash memory can only change bits from 1 to 0; to change any bit from 0 to 1, you must erase the entire block along with it.

Therefore, to alter a specific location within a block while leaving the surrounding contents unchanged, you must read out the entire contents of the block to a buffer, alter the value(s) in the buffer, erase the flash block, and finally write the whole block-sized buffer back to flash memory. The fine-grained flash access functions automate this process at the flash block level.

`alt_get_flash_info()` gets the number of erase regions, the number of erase blocks within each region, and the size of each erase block. The prototype is:

```
int alt_get_flash_info( alt_flash_fd*  fd,
                        flash_region** info,
                        int*           number_of_regions)
```

If the call is successful, upon return the address pointed to by `number_of_regions` contains the number of erase regions in the flash memory, and `*info` points to an array of `flash_region` structures. This array is part of the file descriptor.

The `flash_region` structure is defined in **sys/alt_flash_types.h**, and the `typedef` is:

```
typedef struct flash_region
{
  int offset;   /* Offset of this region from start of the flash */
  int region_size;       /* Size of this erase region */
  int number_of_blocks;  /* Number of blocks in this region */
  int block_size;    /* Size of each block in this erase region */
}flash_region;
```

With the information obtained by calling `alt_get_flash_info()`, you are in a position to erase or program individual blocks of the flash.

`alt_erase_flash()` erases a single block in the flash memory. The prototype is:

```
int alt_erase_flash_block( alt_flash_fd* fd,
                           int           offset,
                           int          length)
```

The flash memory is identified by the handle `fd`. The block is identified as being `offset` bytes from the beginning of the flash memory, and the block size is passed in `length`.

`alt_write_flash_block()` writes to a single block in the flash memory. The prototype is:

```
int alt_write_flash_block( alt_flash_fd* fd,
                           int           block_offset,
                           int            data_offset,
                           const void     *data,
                           int            length)
```

This function writes to the flash memory identified by the handle `fd`. It writes to the block located `block_offset` bytes from the start of the flash. The function writes `length` bytes of data from the location pointed to by `data` to the location `data_offset` bytes from the start of the flash device.

☞ These program and erase functions do not perform address checking, and do not verify whether a write operation spans into the next block. You must pass in valid information about the blocks to program or erase.

The following code demonstrates the usage of the fine-grained flash access functions.

Example: Using the Fine-Grained Flash Access API Functions

```
#include <string.h>
#include "sys/alt_flash.h"
#define BUF_SIZE 100

int main (void)
{
  flash_region* regions;
  alt_flash_fd* fd;
  int           number_of_regions;
  int           ret_code;
  char          write_data[BUF_SIZE];

  /* Set write_data to all 0xa */
  memset(write_data, 0xA, BUF_SIZE);

  fd = alt_flash_open_dev(EXT_FLASH_NAME);

  if (fd)
  {
    ret_code = alt_get_flash_info(fd,
                                  &regions,
                                  &number_of_regions);

    if (number_of_regions && (regions->offset == 0))
    {
      /* Erase the first block */
      ret_code = alt_erase_flash_block(fd,
                                       regions->offset,
                                       regions->block_size);
      if (ret_code)
      {
         /*
          * Write BUF_SIZE bytes from write_data 100 bytes into
          * the first block of the flash
          */
          ret_code = alt_write_flash_block (
                         fd,
                         regions->offset,
                         regions->offset+0x100,
                         write_data,
                         BUF_SIZE );
        }
      }
    }
  return 0;
}
```

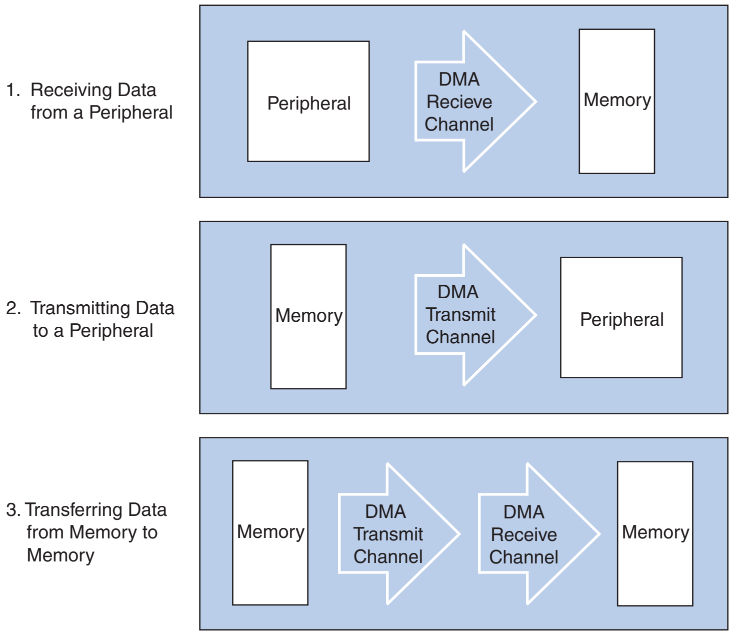
Using DMA Devices

The HAL provides a device abstraction model for direct memory access (DMA) devices. These are peripherals that perform bulk data transactions from a data source to a destination. Sources and destinations can be memory or another device, such as an Ethernet connection.

In the HAL DMA device model, DMA transactions fall into one of two categories: transmit or receive. As a result, the HAL provides two device drivers to implement transmit channels and receive channels. A transmit channel takes data in a source buffer and transmits it to a destination device. A receive channel receives data from a device and deposits it into a destination buffer. Depending on the implementation of the underlying hardware, software might have access to only one of these two endpoints.

Figure 4–2 shows the three basic types of DMA transactions. Copying data from memory to memory involves both receive and transmit DMA channels simultaneously.

Figure 4–2. Three Basic Types of DMA Transactions



The API for access to DMA devices is defined in **sys/alt_dma.h**.

For more information on the use of these functions, refer to the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

DMA devices operate on the contents of physical memory, therefore when reading and writing data you must consider cache interactions.

For more information on cache memory, refer to the *Cache & Tightly-Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.

DMA Transmit Channels

DMA transmit requests are queued up using a DMA transmit device handle. To obtained a handle, use the function `alt_dma_txchan_open()`. This function takes a single argument, the name of a device to use, as defined in **system.h**.

The following code shows how to obtain a handle for a DMA transmit device `dma_0`.

Example: Obtaining a File Handle for a DMA Device

```
#include <stddef.h>
#include "sys/alt_dma.h"

int main (void)
{
  alt_dma_txchan tx;

  tx = alt_dma_txchan_open ("/dev/dma_0");
  if (tx == NULL)
  {
    /* Error */
  }
  else
  {
    /* Success */
  }
  return 0;
}
```

You can use this handle to post a transmit request using `alt_dma_txchan_send()`. The prototype is:

```
typedef void (alt_txchan_done)(void* handle);

int alt_dma_txchan_send (alt_dma_txchan   dma,
                         const void*      from,
                         alt_u32          length,
                         alt_txchan_done* done,
                         void*            handle);
```

Calling `alt_dma_txchan_send()` posts a transmit request to channel `dma`. Argument `length` specifies the number of bytes of data to transmit, and argument `from` specifies the source address. The function returns before the full DMA transaction completes. The return value indicates whether the request is successfully queued. A negative return value indicates that the request failed. When the transaction completes, the user-supplied function `done` is called with argument `handle` to provide notification.

Two additional functions are provided for manipulating DMA transmit channels: `alt_dma_txchan_space()`, and `alt_dma_txchan_ioctl()`. The `alt_dma_txchan_space()` function returns the number of additional transmit requests that can be queued to the device. The `alt_dma_txchan_ioctl()` function performs device-specific manipulation of the transmit device.

☞ If you are using the Altera Avalon DMA device to transmit to hardware (not memory-to-memory transfer), call the `alt_dma_txchan_ioctl()` function with the request argument set to `ALT_DMA_TX_ONLY_ON` (see the "alt_dma_txchan_ioctl()" section of the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*).

DMA Receive Channels

DMA receive channels operate in a similar manner to DMA transmit channels. Software can obtain a handle for a DMA receive channel using the `alt_dma_rxchan_open()` function. You can then use the `alt_dma_rxchan_prepare()` function to post receive requests. The prototype for `alt_dma_rxchan_prepare()` is:

```
typedef void (alt_rxchan_done)(void* handle, void* data);

int alt_dma_rxchan_prepare (alt_dma_rxchan   dma,
                            void*            data,
                            alt_u32          length,
                            alt_rxchan_done* done,
                            void*            handle);
```

A call to this function posts a receive request to channel `dma`, for up to `length` bytes of data to be placed at address `data`. This function returns before the DMA transaction completes. The return value indicates whether the request is successfully queued. A negative return value indicates that the request failed. When the transaction completes, the user-supplied function `done` is called with argument `handle` to provide notification and a pointer to the receive data.

Two additional functions are provided for manipulating DMA receive channels: `alt_dma_rxchan_depth()` and `alt_dma_rxchan_ioctl()`.

☞ If you are using the Altera Avalon DMA device to receive from hardware, (not memory-to-memory transfer), call the `alt_dma_rxchan_ioctl()` function with the request argument set to `ALT_DMA_RX_ONLY_ON` (see the "alt_dma_rxchan_ioctl()" section of the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*).

`alt_dma_rxchan_depth()` returns the maximum number of receive requests that can be queued to the device. `alt_dma_rxchan_ioctl()` performs device-specific manipulation of the receive device.

For further details, see the *HAL API Reference* chapter.

The following code shows a complete example application that posts a DMA receive request, and blocks in `main()` until the transaction completes.

Example: A DMA Transaction on a Receive Channel

```
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include "sys/alt_dma.h"
#include "alt_types.h"

/* flag used to indicate the transaction is complete */
volatile int dma_complete = 0;

/* function that is called when the transaction completes */
void dma_done (void* handle, void* data)
{
  dma_complete = 1;
}

int main (void)
{
  alt_u8 buffer[1024];
  alt_dma_rxchan rx;

  /* Obtain a handle for the device */
  if ((rx = alt_dma_rxchan_open ("/dev/dma_0")) == NULL)
  {
    printf ("Error: failed to open device\n");
    exit (1);
  }
  else
  {
    /* Post the receive request */
   if (alt_dma_rxchan_prepare (rx, buffer, 1024, dma_done, NULL)
        < 0)
    {
      printf ("Error: failed to post receive request\n");
      exit (1);
    }

    /* Wait for the transaction to complete */
    while (!dma_complete);
    printf ("Transaction complete\n");
    alt_dma_rxchan_close (rx);
  }
  return 0;
}
```

Memory-to-Memory DMA Transactions

Copying data from one memory buffer to another buffer involves both receive and transmit DMA drivers. The following code shows the process of queuing up a receive request followed by a transmit request to achieve a memory-to-memory DMA transaction.

Example: Copying Data from Memory to Memory

```
#include <stdio.h>
#include <stdlib.h>

#include "sys/alt_dma.h"
#include "system.h"

static volatile int rx_done = 0;

/*
 * Callback function that obtains notification that the data has
 * been received.
 */

static void done (void* handle, void* data)
{
  rx_done++;
}

/*
 *
 */

int main (int argc, char* argv[], char* envp[])
{
  int rc;

  alt_dma_txchan txchan;
  alt_dma_rxchan rxchan;

  void* tx_data = (void*) 0x901000;  /* pointer to data to send */
  void* rx_buffer = (void*) 0x902000;  /* pointer to rx buffer */

  /* Create the transmit channel */

  if ((txchan = alt_dma_txchan_open("/dev/dma_0")) == NULL)
  {
    printf ("Failed to open transmit channel\n");
    exit (1);
  }

  /* Create the receive channel */

  if ((rxchan = alt_dma_rxchan_open("/dev/dma_0")) == NULL)
  {
    printf ("Failed to open receive channel\n");
    exit (1);
  }

  /* Post the transmit request */

  if ((rc = alt_dma_txchan_send (txchan,
                                 tx_data,
                                 128,
                                 NULL,
                                 NULL)) < 0)
  {
    printf ("Failed to post transmit request, reason = %i\n", rc);
    exit (1);
  }

  /* Post the receive request */

  if ((rc = alt_dma_rxchan_prepare (rxchan,
                                     rx_buffer,
                                     128,
                                     done,
                                     NULL)) < 0)
  {
    printf ("Failed to post read request, reason = %i\n", rc);
    exit (1);
  }

  /* wait for transfer to complete */

  while (!rx_done);

  printf ("Transfer successful!\n");

  return 0;
}
```

Reducing Code Footprint

Code size is always of concern for system developers, because there is a cost associated with the memory device that stores code. The ability to control and reduce code size is important in controlling this cost.

The HAL environment is designed to include only those features that you request, minimizing the total code footprint. If your Nios II hardware system contains exactly the peripherals used by your program, the HAL contains only the drivers necessary to control the hardware, and nothing more.

The following sections describe options to consider when you need to further reduce code size. The **hello_world_small** example project demonstrates the use of some of these options to reduce code size to the absolute minimum.

Enable Compiler Optimizations

To enable compiler optimizations, use the `-O3` compiler optimization level for the `nios2-elf-gcc` compiler. You can specify this command-line option in the project properties; for details, refer to the Nios II IDE help system. Alternatively, you can specify the `-O3` option on the command line. With this option turned on, the Nios II IDE compiles code with the maximum optimization available, for both size and speed. You must set this option for both the system library and the application project.

Use Reduced Device Drivers

Some devices provide two driver variants, a “fast” variant and a “small” variant. Which features are provided by these two variants is device specific. The “fast” variant is full-featured, while the “small” variant provides a reduced code footprint.

By default the HAL system library always uses the fast driver variants. You can choose the small footprint drivers by turning on the **Reduced device drivers** option for your HAL system library in the Nios II IDE. Alternatively, on the command line, you can use the preprocessor option `-DALT_USE_SMALL_DRIVERS` when building the HAL system library.

Table 4–4 lists the Altera Nios II peripherals that currently provide small footprint drivers. The small footprint option might also affect other peripherals. Refer to each peripheral’s data sheet for complete details of its driver’s small footprint behavior.

Table 4–4. Altera Peripherals Offering Small Footprint Drivers

Peripheral	Small Footprint Behavior
UART	Polled operation, rather than IRQ-driven.
JTAG UART	Polled operation, rather than IRQ-driven.
Common flash interface controller	Driver is excluded in small footprint mode.
LCD module controller	Driver is excluded in small footprint mode
EPCS serial configuration device	Driver is excluded in small footprint mode

Reduce the File Descriptor Pool

The file descriptors that access character mode devices and files are allocated from a file descriptor pool. Software can control the size of this pool with the **Max file descriptors** system library property in the Nios II IDE. Alternatively, on the GNU command line, use the compile time constant `ALT_MAX_FD`. The default is 32.

Use /dev/null

At boot time, standard input, standard output and standard error are all directed towards the null device, i.e., **/dev/null**. This direction ensures that calls to `printf()` during driver initialization do nothing and therefore are harmless. Once all drivers have been installed, these streams are then redirected towards the channels configured in the HAL. The footprint of the code that performs this redirection is small, but you can eliminate it entirely by selecting `null` for `stdin`, `stdout`, and `stderr`. This selection assumes that you want to discard all data transmitted on standard out or standard error, and your program never receives input via `stdin`. You can control the `stdin`, `stdout`, and `stderr` channels as a system library property in the Nios II IDE.

Use a Smaller File I/O Library

Use the Small newlib C Library

The full newlib ANSI C standard library is often unnecessary for embedded systems. The GNU Compiler Collection (GCC) provides a reduced implementation of the newlib ANSI C standard library, omitting features of newlib that are often superfluous for embedded systems. The small newlib implementation requires a smaller code footprint. You can control the newlib implementation as a system library property in the Nios II IDE. When you use `nios2-elf-gcc` in command line mode, the `-msmallc` command-line option enables the small C library.

Table 4–5 summarizes the limitations of the Nios II small newlib C library implementation.

Table 4–5. Limitations of the Nios II Small newlib C Library (Part 1 of 2)

Limitation	Functions Affected

No floating-point support for `printf()` family of routines. The functions listed are implemented, but `%f` and `%g` options are not supported. *(1)*	`asprintf()`
	`fiprintf()`
	`fprintf()`
	`iprintf()`
	`printf()`
	`siprintf()`
	`snprintf()`
	`sprintf()`

No floating-point support for `vprintf()` family of routines. The functions listed are implemented, but `%f` and `%g` options are not supported.	`vasprintf()`
	`vfiprintf()`
	`vfprintf()`
	`vprintf()`
	`vsnprintf()`
	`vsprintf()`

No support for `scanf()` family of routines. The functions listed are not supported.	`fscanf()`
	`scanf()`
	`sscanf()`
	`vfscanf()`
	`vscanf()`
	`vsscanf()`

No support for seeking. The functions listed are not supported.	`fseek()`
	`ftell()`

Table 4–5. Limitations of the Nios II Small newlib C Library (Part 2 of 2)

Limitation	Functions Affected

No support for opening/closing `FILE *`. Only pre-opened `stdout`, `stderr`, and `stdin` are available. The functions listed are not supported.	`fopen()`
	`fclose()`
	`fdopen()`
	`fcloseall()`
	`fileno()`

No buffering of **stdio.h** output routines.	functions supported with no buffering:
	`fiprintf()`
	`fputc()`
	`fputs()`
	`perror()`
	`putc()`
	`putchar()`
	`puts()`
	`printf()`
	functions not supported:
	`setbuf()`
	`setvbuf()`

No **stdio.h** input routines. The functions listed are not supported.	`fgetc()`
	`gets()`
	`fscanf()`
	`getc()`
	`getchar()`
	`gets()`
	`getw()`
	`scanf()`

No support for locale.	`setlocale()`
	`localeconv()`

No support for C++, because the above functions are not supported.	

Notes to Table 4–5:

(1) These functions are a Nios II extension. GCC does not implement them in the small newlib C library.

The small newlib C library does not support MicroC/OS II.

For details of the GCC small newlib C library, refer to the newlib documentation installed with the Nios II EDS. On the Windows **Start** menu, click **Programs**, **Altera**, **Nios II** <*version*>, **Nios II Documentation**.

The Nios II implementation of the small newlib C library differs somewhat from GCC. Table 4–5 provides details of the differences.

Use UNIX-Style File I/O

If you need to reduce the code footprint further, you can omit the newlib C library, and use the UNIX-style API. See "UNIX-Style Interface" on page 4–4.

The Nios II EDS provides ANSI C file I/O, in the newlib C library, because there is a per-access performance overhead associated with accessing devices and files using the UNIX-style file I/O functions. The ANSI C file I/O provides buffered access, thereby reducing the total number of hardware I/O accesses performed. Also the ANSI C API is more flexible and therefore easier to use. However, these benefits are gained at the expense of code footprint.

Emulate ANSI C Functions

If you choose to omit the full implementation of newlib, but you need a limited number of ANSI-style functions, you can implement them easily using UNIX-style functions. The following code shows a simple, unbuffered implementation of `getchar()`.

Example: Unbuffered `getchar()`

```
/* getchar: unbuffered single character input */
int getchar ( void )
{
  char c;
  return ( read ( 0, &c, 1 ) == 1 ) ? ( unsigned char ) c : EOF;
}
```

This example is from *The C Programming Language, Second Edition*, by Brian W. Kernighan and Dennis M. Ritchie. This standard textbook contains many other useful functions.

Use the Lightweight Device Driver API

The lightweight device driver API allows you to minimize the overhead of accessing device drivers. It has no direct effect on the size of the drivers themselves, but lets you eliminate driver API features which you might not need, reducing the overall size of the HAL code.

The lightweight device driver API is available for character-mode devices. The following device drivers support the lightweight device driver API:

- JTAG UART
- UART
- Optrex 16207 LCD

For these devices, the lightweight device driver API conserves code space by eliminating the dynamic file descriptor table and replacing it with three static file descriptors, corresponding to `stdin`, `stdout` and `stderr`. Library functions related to opening, closing and manipulating file descriptors are unavailable, but all other library functionality is available. You can refer to `stdin`, `stdout` and `stderr` as you would to any other file descriptor. You can also refer to the following predefined file numbers:

```
#define STDIN 0
#define STDOUT 1
#define STDERR 2
```

This option is appropriate if your program has a limited need for file I/O. The Altera Host Based File System and the Altera Zip Read-only File System are not available with the reduced device driver API.

You can turn on the **Lightweight device driver API** system library property in the Nios II IDE. Alternatively, on the command line, you can use the preprocessor option `-DALT_USE_DIRECT_DRIVERS` when building the HAL system library. By default, the lightweight device driver API is disabled.

For further details about the lightweight device driver API, see the *Developing Device Drivers for the HAL* chapter of the *Nios II Software Developer's Handbook*.

Use the Minimal Character-Mode API

If you can limit your use of character-mode I/O to very simple features, you can reduce code footprint by using the minimal character-mode API. This API includes the following functions:

- `alt_printf()`
- `alt_putchar()`
- `alt_putstr()`
- `alt_getchar()`

These functions are appropriate if your program only needs to accept command strings and send simple text messages. Some of them are helpful only in conjunction with the lightweight device driver API, discussed in "Use the Lightweight Device Driver API" on page 4–29.

To use the minimal character-mode API, include the header file **sys/alt_stdio.h**.

The following sections outline the effects of the functions on code footprint.

alt_printf()

This function is similar to `printf()`, but supports only the `%c`, `%s`, `%x` and `%%` substitution strings. `alt_printf()` takes up substantially less code space than `printf()`, regardless whether you select the lightweight device driver API. `alt_printf()` occupies less than 1Kbyte with compiler optimization level `-O2`.

alt_putchar()

Equivalent to `putchar()`. In conjunction with the lightweight device driver API, this function further reduces code footprint. In the absence of the lightweight API, it calls `putchar()`.

alt_putstr()

Similar to `puts()`, except that it does not append a newline character to the string. In conjunction with the lightweight device driver API, this function further reduces code footprint. In the absence of the lightweight API, it calls `puts()`.

alt_getchar()

Equivalent to `getchar()`. In conjunction with the lightweight device driver API, this function further reduces code footprint. In the absence of the lightweight API, it calls `getchar()`.

For further details on the minimal character-mode functions, refer to the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

Eliminate Unused Device Drivers

If a hardware device is present in the system, the Nios II IDE assumes the device needs drivers, and configures the HAL system library accordingly. If the HAL can find an appropriate driver, it creates an instance of this driver. If your program never actually accesses the device, resources are being used unnecessarily to initialize the device driver.

If the hardware includes a device that your program never uses, consider removing the device from the hardware. This reduces both code footprint and FPGA resource usage.

However, there are cases when a device must be present, but runtime software does not require a driver. The most common example is flash memory. The user program might boot from flash, but not use it at runtime; thus, it does not need a flash driver.

In the Nios II IDE, you can prevent the HAL from including the flash driver by defining the `DALT_NO_CFI_FLASH` preprocessor option in the properties for the system library project. Alternatively, you can specify the `-DALT_NO_CFI_FLASH` option to the preprocessor on the command line.

You can achieve further control of the device driver initialization process by using the free-standing environment. See "Boot Sequence and Entry Point" on page 4–33.

Eliminate Unneeded Exit Code

The HAL calls the `exit()` function at system shutdown to provide a clean exit from the program. `exit()` flushes all of the C library internal I/O buffers and calls any C++ functions registered with `atexit()`. In particular, `exit()` is called upon return from `main()`. Two HAL options allow you to minimize or eliminate this exit code.

Eliminate Clean Exit

To avoid the overhead associated with providing a clean exit, your program can use the function `_exit()` in place of `exit()`. This function does not require you to change source code. You can control exit behavior through the **Clean exit (flush buffers)** system library property in the Nios II IDE. Alternatively, on the command line, you can specify the preprocessor option `-Dexit=_exit`.

Eliminate All Exit Code

Many embedded systems never exit at all. In such cases, exit code is unnecessary.

You can configure the HAL to omit all exit code (`exit()` and `_exit()`) from the system library by turning on **Program never exits** in the system library properties in the Nios II IDE. Alternatively, on the command line, you can use the preprocessor option `-DALT_NO_EXIT` when building the HAL system library.

☞ If you enable this option, make sure your `main()` function (or `alt_main()` function) does not return.

Turn off C++ Support

By default, the HAL provides support for C++ programs, including default constructors and destructors. You can omit this support code by turning off the **Support C++** system library property in the Nios II IDE. Alternatively, on the command line, you can use the preprocessor option `-DALT_NO_C_PLUS_PLUS` when building the HAL system library.

Boot Sequence and Entry Point

Normally, your program's entry point is the function `main()`. There is an alternate entry point, `alt_main()`, that you can use to gain greater control of the boot sequence. The difference between entering at `main()` and entering at `alt_main()` is the difference between hosted and free-standing applications.

Hosted vs. Free-Standing Applications

The ANSI C standard defines a hosted application as one that calls `main()` to begin execution. At the start of `main()`, a hosted application presumes the runtime environment and all system services are initialized and ready to use. This is true in the HAL environment. If you are new to Nios II programming, the HAL's hosted environment helps you come up to speed more easily, because you don't have to consider what devices exist in the system or how to initialize each one. The HAL initializes the whole system.

The ANSI C standard also provides for an alternate entry point that avoids automatic initialization, and assumes that the Nios II programmer manually initializes any needed hardware. The `alt_main()` function provides a free-standing environment, giving you complete control over the initialization of the system. The free-standing environment places upon the programmer the burden of manually initializing any system feature used in the program. For example, calls to `printf()` do not function correctly in the free-standing environment, unless `alt_main()` first instantiates a character-mode device driver, and redirects `stdout` to the device.

☞ Using the freestanding environment increases the complexity of writing Nios II programs, because you assume responsibility for initializing the system. If your main interest in the freestanding environment is to reduce code footprint, you should use the suggestions described in "Reducing Code Footprint" on page 4–25. It is easier to reduce the HAL system library footprint by using options available in the Nios II IDE, than to use the freestanding mode.

The Nios II EDS provides examples of both free-standing and hosted programs.

For more information, refer to the Nios II IDE help system.

Boot Sequence for HAL-Based Programs

The HAL provides system initialization code that performs the following boot sequence:

- Flushes the instruction and data cache
- Configures the stack pointer
- Configures the global pointer register
- Zero initializes the BSS region using the linker supplied symbols `__bss_start` and `__bss_end`. These are pointers to the beginning and the end of the BSS region
- If there is no boot loader present in the system, copies to RAM any linker section whose run address is in RAM, such as `.rwdata`, `.rodata`, and `.exceptions`. See "Global Pointer Register" on page 4–39.
- Calls `alt_main()`

The HAL provides a default implementation of the `alt_main()` function, which performs the following steps:

- Calls `ALT_OS_INIT()` to perform any necessary operating system specific initialization. For a system that does not include an OS scheduler, this macro has no effect
- If you are using the HAL with an operating system, initializes the `alt_fd_list_lock` semaphore, which controls access to the HAL file systems.
- Initializes the interrupt controller, and enable interrupts
- Calls the `alt_sys_init()` function, which initializes all device drivers and software components in the system. The Nios II IDE creates and manages the file `alt_sys_init.c` for each HAL system library
- Redirects the C standard I/O channels (`stdin`, `stdout`, and `stderr`) to use the appropriate devices
- Calls the C++ constructors, using the `_do_ctors()` function
- Register the C++ destructors to be called at system shutdown
- Calls `main()`
- Calls `exit()`, passing the return code of `main()` as the input argument for `exit()`

alt_main.c, installed with the Nios II EDS, provides this default implementation. You can find it in <*Nios II EDS install path*>/**components/altera_hal/HAL/src.**

Customizing the Boot Sequence

You can provide your own implementation of the start-up sequence by simply defining `alt_main()` in your Nios II IDE project. This gives you complete control of the boot sequence, and gives you the power to selectively enable HAL services. If your application requires an `alt_main()` entry point, you can copy the default implementation as a starting point and customize it to your needs.

Function `alt_main()` calls function `main()`. After `main()` returns, the default `alt_main()` enters an infinite loop. Alternatively, your custom `alt_main()` might terminate by calling `exit()`. Do not use a `return` statement.

The prototype for `alt_main()` is:

```
void alt_main (void)
```

A feature of the HAL build environment is that all source and include files are located using a search path. The build system always searches the system library project's paths first. This lets you override the default device drivers and system code with your own implementation. For example, you can supply your own custom **alt_sys_init.c** by placing it in your system project directory. Your custom file is used in preference to the auto-generated version.

For more information on `alt_sys_init()`, see the *Developing Device Drivers for the HAL* chapter of the *Nios II Software Developer's Handbook*.

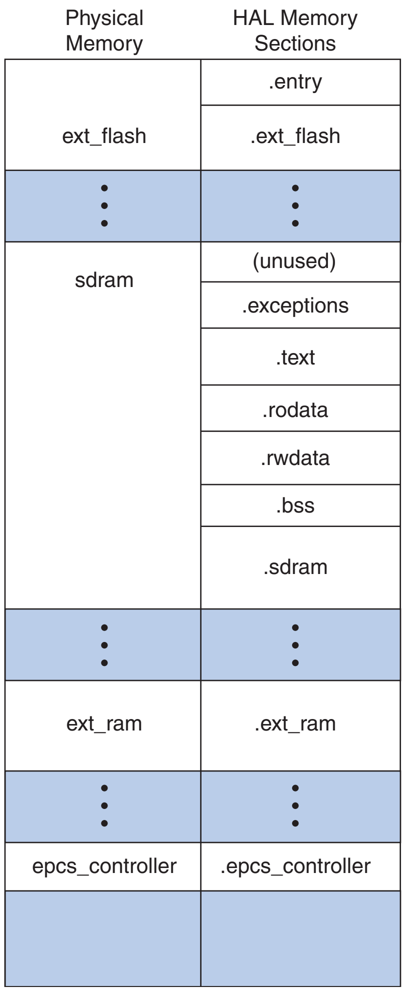
Memory Usage

This section describes the way that the HAL uses memory and how the HAL arranges code, data, stack, and other logical memory sections, in physical memory.

Memory Sections

By default, HAL-based systems are linked using an automatically-generated linker script that is created and managed by the Nios II IDE. This linker script controls the mapping of code and data within the available memory sections. The auto-generated linker script creates standard code and data sections (.text, .rodata, .rwdata, and .bss), plus a section for each physical memory device in the system. For example, if there is a memory component named `sdram` defined in the **system.h** file, there is a memory section named `.sdram`. Figure 4–3 on page 4–36 shows the organization of a typical HAL link map.

Figure 4–3. Sample HAL Link Map



The memory devices that contain the Nios II processor's reset and exception addresses are a special case. The Nios II tools construct the 32-byte `.entry` section starting at the reset address. This section is reserved exclusively for the use of the reset handler. Similarly, the tools construct a `.exceptions` section, starting at the exception address.

In a memory device containing the reset or exception address, the linker creates a normal (non-reserved) memory section above the `.entry` or `.exceptions` section. If there is a region of memory below the `.entry` or `.exceptions` section, it is unavailable to the Nios II software. Figure 4–3 on page 4–36 illustrates an unavailable memory region below the `.exceptions` section.

Assigning Code & Data to Memory Partitions

This section describes how to control the placement of program code and data in specific memory sections. In general, the Nios II IDE automatically specifies a sensible default partitioning. However, you might wish to change the partitioning in special situations.

For example, to enhance performance, it is a common technique to place performance-critical code and data in RAM with fast access time. It is also common during the debug phase to reset (i.e., boot) the processor from a location in RAM, but then boot from flash memory in the released version of the software. In these cases, you have to specify manually which code belongs in which section.

Simple Placement Options

The reset handler code is always placed at the base of the `.reset` partition. The exception handler code is always the first code within the section that contains the exception address. By default, the remaining code and data are divided into the following output sections:

- `.text`—all remaining code
- `.rodata`—the read-only data
- `.rwdata`—read-write data,
- `.bss`—zero-initialized data

You can control the placement of `.text`, `.rodata`, `.rwdata`, and all other memory partitions through the **System Library Properties** page in the Nios II IDE. For more information, in the Nios II IDE help system, search for the **"System Library Properties"** topic.

Advanced Placement Options

Within your program source code, you can specify a target memory section for each piece of code. In C or C++, you can use the `section` attribute. This attribute must be placed in a function prototype; you cannot place it in the function declaration itself. The following code shows placing a variable `foo` within the memory named `ext_ram`, and the function `bar()` in the memory named `sdram`.

Example: Manually Assigning C Code to a Specific Memory Section

```
/* data should be initialized when using the section attribute */
int foo __attribute__ ((section (".ext_ram.rwdata"))) = 0;
```

```
void bar (void) __attribute__ ((section (".sdram.txt")));

void bar (void)
{
  foo++;
}
```

In assembly you do this using the `.section` directive. For example, all code after the following line is placed in the memory device named `ext_ram`:

```
.section .ext_ram.txt
```

The section names `ext_ram` and `sdram` are examples. You need to use section names corresponding to your hardware. When creating section names, use the following extensions:

- `.txt` for code: for example, `.sdram.txt`
- `.rodata` for read-only data: for example, `.cfi_flash.rodata`
- `.rwdata` for read-write data: for example, `.ext_ram.rwdata`

For details of the usage of these features, refer to the GNU compiler and assembler documentation. This documentation is installed with the Nios II IDE. To find it, open the **Nios II Literature** page, scroll down to **Software Development,** and click **Using the GNU Compiler Collection (GCC).**

Placement of the Heap & Stack

By default, the heap and stack are placed in the same memory partition as the `.rwdata` section. The stack grows downwards (toward lower addresses) from the end of the section. The heap grows upwards from the last used memory within the `.rwdata` section. You can control the placement of the heap and stack as a system library property in the Nios II IDE.

By default, the HAL performs no stack or heap checking. This makes function calls and memory allocation faster, but it means that `malloc()` (in C) and `new` (in C++) are unable to detect heap exhaustion. You can enable run-time stack checking, which is a system library property in the Nios II IDE. With stack checking on, `malloc()` and `new` can detect heap exhaustion.

To specify the heap size limit, set the preprocessor symbol `ALT_MAX_HEAP_BYTES` to the maximum heap size in decimal. For example, the preprocessor argument

`-DALT_MAX_HEAP_SIZE=1048576` sets the heap size limit to 0x100000. You can specify this command-line option in the system library properties; for details, refer to the Nios II IDE help system. Alternatively, you can specify the option on the command line.

Stack checking has performance costs. If you choose to leave stack checking turned off, you must code your program so as to ensure that it operates within the limits of available heap and stack memory.

See the Nios II IDE help system for details of selecting stack and heap placement, and setting up stack checking.

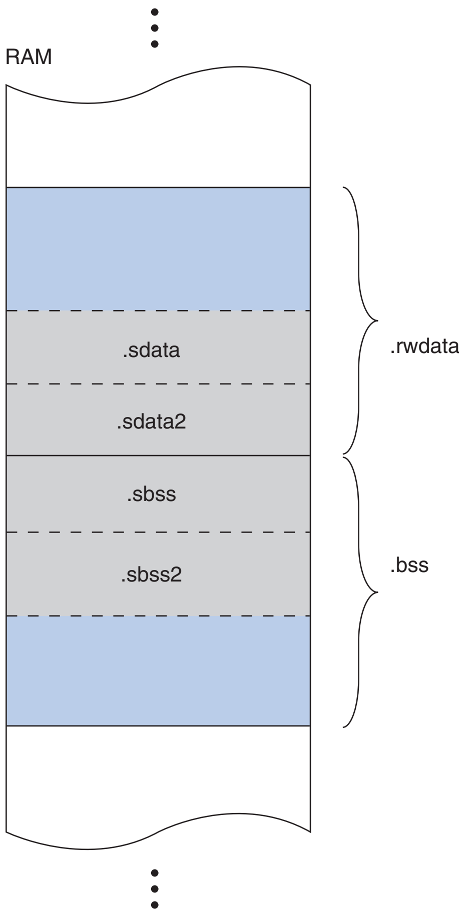
Global Pointer Register

The global pointer register enables fast access to global data structures in Nios II programs. The Nios II compiler implements the global pointer, and determines which data structures to access with it. You do not need to do anything unless you want to change the default compiler behavior.

The global pointer register can access a single contiguous region of 64K bytes. To avoid overflowing this region, the compiler only uses the global pointer with small global data structures. A data structure is considered "small" if its size is less than a specified threshold. By default, this threshold is eight bytes.

The “small” data structures are allocated to the small global data sections, `.sdata`, `.sdata2`, `.sbss`, and `.sbss2`. The small global data sections are subsections of the `.rwdata` and `.bss` sections. They are located together, as shown in Figure 4–4 on page 4–40, to enable the global pointer to access them.

Figure 4–4. Small Global Data sections



If the total size of the small global data structures happens to be more than 64K bytes, they overflow the global pointer region. The linker produces an error message saying `"Unable to reach` *<variable name>* `... from the global pointer ... because the offset ... is out of the allowed range, -32678 to 32767."`

You can fix this with the `-G` compiler option. This option sets the threshold size. For example, `-G 4` restricts global pointer usage to data structures four bytes long or smaller. Reducing the global pointer threshold reduces the size of the small global data sections.

The `-G` option's numeric argument is in decimal. You can specify this compiler option in the project properties; for details, refer to the Nios II IDE help system. Alternatively, you can specify the option on the command line. You must set this option to the same value for both the system library and the application project.

Boot Modes

The processor's boot memory is the memory that contains the reset vector. This device might be an external flash or an Altera EPCS serial configuration device, or it might be an on-chip RAM. Regardless of the nature of the boot memory, HAL-based systems are constructed so that all program and data sections are initially stored within it. The HAL provides a small boot loader program which copies these sections to their run time locations at boot time. You can specify run time locations for program and data memory on the system library properties page.

If the runtime location of the `.text` section is outside of the boot memory, the Altera flash programmer in the Nios II IDE places a boot loader at the reset address, which is responsible for loading all program and data sections before the call to `_start`. When booting from an EPCS device, this loader function is provided by the hardware.

However, if the runtime location of the `.text` section is in the boot memory, the system does not need a separate loader. Instead the `_reset` entry point within the HAL executable is called directly. The function `_reset` initializes the instruction cache and then calls `_start`. This initialization sequence lets you develop applications that boot and execute directly from flash memory.

When running in this mode, the HAL executable must take responsibility for loading any sections that require loading to RAM. The `.rwdata`, `.rodata`, and `.exceptions` sections are loaded before the call to `alt_main()`, as required. This loading is performed by the function `alt_load()`. To load any additional sections, use the `alt_load_section()` function.

For more information, refer to the "alt_load_section()" section of the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

Paths to HAL System Library Files

You might wish to view files in the HAL system library, especially header files, for reference. Do not edit HAL system library files.

Finding HAL Files

HAL system library files are in several directories because of the custom nature of Nios II systems. Each Nios II system can include different peripherals, and therefore the HAL system library for each system is different. You can find HAL-related files in the following locations:

- The <*Nios II EDS install path*>/**components** directory contains most HAL system library files.
- <*Nios II EDS install path*>/**components/altera_hal/HAL/inc/sys** contains header files defining the HAL generic device models. In a `#include` directive, reference these files relative to <*Nios II EDS install path*>/**components/altera_hal/HAL/inc/**. For example, to include the DMA drivers, use `#include sys/alt_dma.h`
- Each Nios II IDE system project directory contains the **system.h** file generated for that system library project**.**
- <*Nios II EDS install path*>/**bin** contains the newlib ANSI C library header files.
- The Altera design suite includes HAL drivers for SOPC Builder components distributed with Quartus II. For example, if the Altera design suite is installed in **c:\altera\61**, you can find the drivers under **c:\altera\61\ip\sopc_builder_ip**.
- These drivers are located in

Overriding HAL Functions

To provide your own implementation of a HAL function, include the file in your Nios II IDE system project. When building the executable, Nios II IDE finds your function, and uses it in place of the HAL version.

Document Revision History

Table 4–6 shows the revision history for this document.

Table 4–6. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
November 2006, v6.1.0	● **Program never exits** system library option ● **Support C++** system library option ● **Lightweight device driver API** system library option ● Minimal character-mode API	
May 2006, v6.0.0	● Revised text on instruction emulation. ● Added section on global pointers.	
October 2005, v5.1.0	● Added `alt_64` and `alt_u64 types` to Table 4–1. ● Made changes to section "Placement of the Heap & Stack".	
May 2005, v5.0.0	Added `alt_load_section()` function information.	
December 2004 v1.2	● Added boot modes information. ● Amended compiler optimizations. ● Updated *Reducing Code Footprint* section.	
September 2004 v1.1	Corrected DMA receive channels example code.	
May 2004 v1.0	First publication.	

5. Developing Device Drivers for the HAL

NII52005-6.1.0

Introduction

Embedded systems typically have application-specific hardware features that require custom device drivers. This chapter describes how to develop device drivers and integrate them with the hardware abstraction layer (HAL) system library.

Direct interaction with the hardware should be confined to device driver code. In general, most of your program code should be free of low-level access to the hardware. Wherever possible, use the high-level HAL application programming interface (API) functions to access hardware. This makes your code more consistent and more portable to other Nios® II systems that might have different hardware configurations.

When you create a new driver, you can integrate the driver into the HAL framework at one of the following two levels:

- Integration into the HAL API
- Peripheral-specific API

Integration into the HAL API

Integration into the HAL API is the preferred option for a peripheral that belongs to one of the HAL generic device model classes, such as character-mode or direct memory access (DMA) devices. For integration into the HAL API, you write device accessor functions as specified in this chapter, and the device becomes accessible to software via the standard HAL API. For example, if you have a new LCD screen device that displays ASCII characters, you write a character-mode device driver. With this driver in place, programs can call the familiar `printf()` function to stream characters to the LCD screen.

Peripheral-Specific API

If the peripheral does not belong to one of the HAL generic device model classes, you need to provide a device driver with an interface that is specific to the hardware implementation, and the API to the device is separate from the HAL API. Programs access the hardware by calling the functions you provide, not the HAL API.

The up-front effort to implement integration into the HAL API is higher, but you gain the benefit of the HAL and C standard library API to manipulate devices.

For details on integration into the HAL API, see "Integrating a Device Driver into the HAL" on page 5–17.

All the other sections in this chapter apply to integrating drivers into the HAL API and creating drivers with a peripheral-specific API.

☞ Although C++ is supported for programs based on the HAL, HAL drivers should not be written in C++. Restrict your driver code to either C or assembler, and preferably C for portability.

Before You Begin

This chapter assumes that you are familiar with C programming for the HAL. You should be familiar with the information in the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*, before reading this chapter.

☞ This document uses the variable <*Altera installation*> to represent the location where the Altera design suite is installed. On a Windows system, by default, that location is **c:\altera**\nn, where *nn* represents the current version number.

Development Flow for Creating Device Drivers

The steps to develop a new driver for the HAL are very much dependent on your device details. However, the following generic steps apply to all device classes.

1. Create the device header file that describes the registers. This header file might be the only interface required.

2. Implement the driver functionality.

3. Test from `main()`.

4. Proceed to the final integration of the driver into the HAL environment.

5. Integrate the device driver into the HAL framework.

SOPC Builder Concepts

This section discusses concepts about the Altera® SOPC Builder hardware design tool that enhance your understanding of the driver development process. You need not use SOPC Builder to develop Nios II device drivers.

The Relationship between system.h & SOPC Builder

The **system.h** header file provides a complete software description of the Nios II system hardware, and is a fundamental part of developing drivers. Because drivers interact with hardware at the lowest level, it is worth mentioning the relationship between **system.h** and SOPC Builder that generates the Nios II processor system hardware. Hardware designers use SOPC Builder to specify the architecture of the Nios II processor system and integrate the necessary peripherals and memory. Therefore, the definitions in **system.h**, such as the name and configuration of each peripheral, are a direct reflection of design choices made in SOPC Builder.

For more information on the **system.h** header file, see the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*.

Using SOPC Builder for Optimal Hardware Configuration

If you find less-than-optimal definitions in **system.h**, remember that the contents of **system.h** can be modified by changing the underlying hardware with SOPC Builder. Before you write a device driver to accommodate imperfect hardware, it is worth considering whether the hardware can be improved easily with SOPC Builder.

Components, Devices & Peripherals

SOPC Builder uses the term "component" to describe hardware modules included in the system. In the context of Nios II software development, SOPC Builder components are devices, such as peripherals or memories. In the following sections, "component" is used interchangeably with "device" and "peripheral" when the context is closely related to SOPC Builder.

Accessing Hardware

Software accesses the hardware via macros that abstract the memory-mapped interface to the device. This section describes the macros that define the hardware interface for each device.

All SOPC Builder components provide a directory that defines the device hardware and software. For example, each component provided in the Quartus® II software has its own directory in the <*Altera installation*>**/ip/sopc_builder_ip** directory. Many components provide a header file that defines their hardware interface. The header file is <*name of component*>**_regs.h** and is included in the **inc** subdirectory for the specific component. For example, the Altera-provided JTAG UART component defines its hardware interface in the file *<Altera installation>***/ip/sopc_builder_ip/altera_avalon_jtag_uart/inc/altera_avalon_jtag_uart_regs.h**.

The **_regs.h** header file defines the following access:

- Register accessor macros that provide a read and/or write macro for each register within the component that supports the operation. The macros are **IORD_***<name_of_component>*_*<name_of_register>* (*component_base_address*), and **IOWR_***<name_of_component>*_*<name_of_register>* (*component_base_address*, *data*). See the *Cache & Tightly-Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.
- Register address accessor macros that return the physical address for each register within a component. The address register returned is the component's base address + the specified register offset value. These macros are named **IOADDR_***<name_of_component>*_*<name_of_register>* (*component_base_address*). Use these macros only as parameters to a function that requires the specific address of a data source or destination. For example, a routine that reads a stream of data from a particular source register in a component might require the physical address of the register as a parameter.
- Bit-field masks and offsets that provide access to individual bit-fields within a register. These macros have the following names:
  - *<name_of_component>*_*<name_of_register>*_*<name_of_field>*`_MSK`, for a bit-mask of the field
  - *<name_of_component>*_*<name_of_register>*_*<name_of_field>*`_OFST`, for the bit offset of the start of the field
  - `ALTERA_AVALON_UART_STATUS_PE_MSK` and `ALTERA_AVALON_UART_STATUS_PE_OFST`, for accessing the PE field of the status register.

Only use the macros defined in the **_regs.h** file to access a device's registers. You must use the register accessor functions to ensure that the processor bypasses the data cache when reading and or writing the device. Furthermore, you should never use hard-coded constants, because this action makes your software susceptible to changes in the underlying hardware.

If you are writing the driver for a completely new hardware device, you have to prepare the **_regs.h** header file.

For more information on the effects of cache management and device access, see the *Cache & Tightly-Coupled Memory* chapter of the *Nios II Software Developer's Handbook*. For a complete example of the **_regs.h** file, see the component directory for any of the Altera-supplied SOPC Builder components.

Creating Drivers for HAL Device Classes

The HAL supports a number of generic device model classes, as defined in the *Overview of the HAL System Library* chapter of the *Nios II Software Developer's Handbook*. By writing a device driver as described in this section, you describe to the HAL an instance of a specific device that falls into one of its known device classes. This section defines a consistent interface for driver functions so that the HAL can access the driver functions uniformly.

The following sections define the API for the following classes of devices:

- Character-mode devices
- File subsystems
- DMA devices
- Timer devices used as system clock
- Timer devices used as timestamp clock
- Flash memory devices
- Ethernet devices

The following sections describe how to implement device drivers for each class of device, and how to register them for use within HAL-based systems.

Character-Mode Device Drivers

This section describes how to create a device instance and register a character device.

Create a Device Instance

For a device to be made available as a character mode device, it must provide an instance of the `alt_dev` structure. The following code defines the `alt_dev` structure:

```
typedef struct {
  alt_llist    llist;     /* for internal use */
  const char*  name;
 int (*open)  (alt_fd* fd, const char* name, int flags, int mode);
  int (*close) (alt_fd* fd);
  int (*read)  (alt_fd* fd, char* ptr, int len);
  int (*write) (alt_fd* fd, const char* ptr, int len);
  int (*lseek) (alt_fd* fd, int ptr, int dir);
  int (*fstat) (alt_fd* fd, struct stat* buf);
  int (*ioctl) (alt_fd* fd, int req, void* arg);
} alt_dev;
```

The `alt_dev` structure, defined in ***<Nios II EDS install path>*/components/altera_hal/HAL/inc/sys/alt_dev.h**, is essentially a collection of function pointers. These functions are called in response to application accesses to the HAL file system. For example, if you call the function `open()` with a file name that corresponds to this device, the result is a call to the `open()` function provided in this structure.

For more information on `open()`, `close()`, `read()`, `write()`, `lseek()`, `fstat()`, and `ioctl()`, see the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

None of these functions directly modify the global error status, `errno`. Instead, the return value is the negation of the appropriate error code provided in **errno.h**.

For example, the `ioctl()` function returns `-ENOTTY` if it cannot handle a request rather than set `errno` to `ENOTTY` directly. The HAL system routines that call these functions ensure that `errno` is set accordingly.

The function prototypes for these functions differ from their application level counterparts in that they each take an input file descriptor argument of type `alt_fd*` rather than `int`.

A new `alt_fd` structure is created upon a call to `open()`. This structure instance is then passed as an input argument to all function calls made for the associated file descriptor.

The following code defines the `alt_fd` structure.

```
typedef struct
{
  alt_dev* dev;
  void* priv;
  int fd_flags;
} alt_fd;
```

where:

- `dev` is a pointer to the device structure for the device being used.
- `fd_flags` is the value of `flags` passed to `open()`.
- `priv` is an opaque value that is unused by the HAL system code. `priv` is available for drivers to store any per file descriptor information that they require for internal use.

A driver is not required to provide all of the functions within the `alt_dev` structure. If a given function pointer is set to `NULL`, a default action is used instead. Table 5–1 shows the default actions for each of the available functions.

Table 5–1. Default Behavior for Functions Defined in alt_dev

Function	Default Behavior
`open`	Calls to `open()` for this device succeed, unless the device has been previously locked by a `TIOCEXCL` `ioctl()` request.
`close`	Calls to `close()` for a valid file descriptor for this device always succeed.
`read`	Calls to `read()` for this device always fail.
`write`	Calls to `write()` for this device always fail.
`lseek`	Calls to `lseek()` for this device always fail.
`fstat`	The device identifies itself as a character mode device.
`ioctl`	`ioctl()` requests that cannot be handled without reference to the device fail.

In addition to the function pointers, the `alt_dev` structure contains two other fields: `llist` and `name`. `llist` is for internal use, and should always be set to the value `ALT_LLIST_ENTRY`. `name` is the location of the device within the HAL file system and is the name of the device as defined in **system.h**.

Register a Character Device

Having created an instance of the `alt_dev` structure, the device must be made available to the system by registering it with the HAL and by calling the following function:

```
int alt_dev_reg (alt_dev* dev)
```

This function takes a single input argument, which is the device structure to register. A return value of zero indicates success. A negative return value indicates that the device can not be registered.

Once a device is registered with the HAL file system, you can access it via the HAL API and the ANSI C standard library. The node name for the device is the name specified in the `alt_dev` structure.

For more information, refer to the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook.*

File Subsystem Drivers

A file subsystem device driver is responsible for handling file accesses beneath a specified mount point within the global HAL file system.

Create a Device Instance

Creating and registering a file system is very similar to creating and registering a character-mode device. To make a file system available, create an instance of the `alt_dev` structure (see "Character-Mode Device Drivers" on page 5–5). The only distinction is that the `name` field of the device represents the mount point for the file subsystem. Of course, you must also provide any necessary functions to access the file subsystem, such as `read()` and `write()`, similar to the case of the character-mode device.

If you do not provide an implementation of `fstat()`, the default behavior returns the value for a character-mode device, which is incorrect behavior for a file subsystem.

Register a File Subsystem Device

You can register a file subsystem using the following function:

```
int alt_fs_reg  (alt_dev* dev)
```

This function takes a single input argument, which is the device structure to register. A negative return value indicates that the file system can not be registered.

Once a file subsystem is registered with the HAL file system, you can access it via the HAL API and the ANSI C standard library. The mount point for the file subsystem is the `name` specified in the `alt_dev` structure.

For more information, refer to the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook.*

Timer Device Drivers

This section describes the system clock and timestamp drivers.

System Clock Driver

A system clock device model requires a driver to generate the periodic "tick". There can be only one system clock driver in a system. You implement a system clock driver as an interrupt service routine (ISR) for a timer peripheral that generates a periodic interrupt. The driver must provide periodic calls to the following function:

```
void alt_tick (void)
```

The expectation is that `alt_tick()` is called in interrupt context.

To register the presence of a system clock driver, call the following function:

```
int alt_sysclk_init (alt_u32 nticks)
```

The input argument `nticks` is the number of system clock ticks per second, which is determined by your system clock driver. The return value of this function is zero upon success, and non-zero otherwise.

For more information on writing interrupt service routines, see the *Exception Handling* chapter of the *Nios II Software Developer's Handbook*.

Timestamp Driver

A timestamp driver provides implementations for the three timestamp functions: `alt_timestamp_start()`, `alt_timestamp()`, and `alt_timestamp_freq()`. The system can only have one timestamp driver.

For more information on using these functions, see the *Developing Programs using the HAL* and *HAL API Reference* chapters of the *Nios II Software Developer's Handbook*.

Flash Device Drivers

This section describes how to create a flash driver and register a flash device.

Create a Flash Driver

Flash device drivers must provide an instance of the `alt_flash_dev` structure, defined in **sys/alt_flash_dev.h**. The following code shows the structure:

```
struct alt_flash_dev
{
  alt_llist                    llist; // internal use only
  const char*                  name;
  alt_flash_open               open;
  alt_flash_close              close;
  alt_flash_write              write;
  alt_flash_read               read;
  alt_flash_get_flash_info     get_info;
  alt_flash_erase_block        erase_block;
  alt_flash_write_block        write_block;
  void*                        base_addr;
  int                          length;
  int                          number_of_regions;
  flash_region    region_info[ALT_MAX_NUMBER_OF_FLASH_REGIONS];
};
```

The first parameter `llist` is for internal use, and should always be set to the value ALT_LLIST_ENTRY. `name` is the location of the device within the HAL file system and is the name of the device as defined in **system.h**.

The seven fields `open` to `write_block` are function pointers that implement the functionality behind the application API calls to:

- `alt_flash_open_dev()`
- `alt_flash_close_dev()`
- `alt_write_flash()`
- `alt_read_flash()`
- `alt_get_flash_info()`
- `alt_erase_flash_block()`
- `alt_write_flash_block()`

where:

- the `base_addr` parameter is the base address of the flash memory
- `length` is the size of the flash in bytes
- `number_of_regions` is the number of erase regions in the flash
- `region_info` contains information about the location and size of the blocks in the flash device

For more information on the format of the `flash_region` structure, refer to the "Using Flash Devices" section of the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*.

Some flash devices such as common flash interface (CFI) compliant devices allow you to read out the number of regions and their configuration at run time. Otherwise, these two fields must be defined at compile time.

Register a Flash Device

After creating an instance of the `alt_flash_dev` structure, you must make the device available to the HAL system by calling the following function:

```
int alt_flash_device_register( alt_flash_fd* fd)
```

This function takes a single input argument, which is the device structure to register. A return value of zero indicates success. A negative return value indicates that the device could not be registered.

DMA Device Drivers

The HAL models a DMA transaction as being controlled by two endpoint devices: a receive channel and a transmit channel. This section describes the drivers for each type of DMA channel separately.

For a complete description of the HAL DMA device model, refer to the "Using DMA Devices" section of the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*.

The DMA device driver interface is defined in **sys/alt_dma_dev.h**.

DMA Transmit Channel

A DMA transmit channel is constructed by creating an instance of the `alt_dma_txchan` structure:

```
typedef struct alt_dma_txchan_dev_s alt_dma_txchan_dev;
struct alt_dma_txchan_dev_s
{
  alt_llist   llist;
  const char* name;
  int         (*space) (alt_dma_txchan  dma);
  int         (*send) (alt_dma_txchan   dma,
                       const void*      from,
                       alt_u32          len,
                       alt_txchan_done* done,
                       void*            handle);
  int         (*ioctl) (alt_dma_txchan dma, int req, void* arg);
};
```

Table 5–2 shows the available fields and their functions.

Table 5–2. Fields in the alt_dma_txchan Structure

Field	Function
`llist`	This field is for internal use, and must always be set to the value ALT_LLIST_ENTRY.
`name`	The name that refers to this channel in calls to `alt_dma_txchan_open()`. `name` is the name of the device as defined in **system.h**.
`space`	A pointer to a function that returns the number of additional transmit requests that can be queued to the device. The input argument is a pointer to the `alt_dma_txchan_dev` structure.
`send`	A pointer to a function that is called as a result of a call to the application API function `alt_dma_txchan_send()`.This function posts a transmit request to the DMA device. The parameters passed to `alt_txchan_send()` are passed directly to `send()`. For a description of parameters and return values, see `alt_dma_txchan_send()` in the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.
`ioctl`	This function provides device specific I/O control. See **sys/alt_dma_dev.h** for a list of the generic options that a device might wish to support.

Both the `space` and `send` functions need to be defined. If the `ioctl` field is set to null, calls to `alt_dma_txchan_ioctl()` return `-ENOTTY` for this device.

After creating an instance of the `alt_dma_txchan` structure, you must register the device with the HAL system to make it available by calling the following function:

```
int alt_dma_txchan_reg (alt_dma_txchan_dev* dev)
```

The input argument `dev` is the device to register. The return value is zero upon success, or negative if the device cannot be registered.

DMA Receive Channel

A DMA receive channel is constructed by creating an instance of the `alt_dma_rxchan` structure:

```
typedef alt_dma_rxchan_dev_s alt_dma_rxchan;
struct alt_dma_rxchan_dev_s
{
  alt_llist   list;
  const char* name;
  alt_u32     depth;
  int         (*prepare) (alt_dma_rxchan   dma,
                           void*            data,
                           alt_u32          len,
                           alt_rxchan_done* done,
                           void*            handle);
  int         (*ioctl) (alt_dma_rxchan dma, int req, void* arg);
};
```

Table 5–3 shows the available fields and their functions.

Table 5–3. Fields in the alt_dma_rxchan Structure

Field	Function
`llist`	This function is for internal use and should always be set to the value `ALT_LLIST_ENTRY`.
`name`	The name that refers to this channel in calls to `alt_dma_rxchan_open()`. `name` is the name of the device as defined in **system.h**.
`depth`	The total number of receive requests that can be outstanding at any given time.
`prepare`	A pointer to a function that is called as a result of a call to the application API function `alt_dma_rxchan_prepare()`. This function posts a receive request to the DMA device. The parameters passed to `alt_dma_rxchan_prepare()` are passed directly to `prepare()`. For a description of parameters and return values, see `alt_dma_rxchan_prepare()` in the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.
`ioctl`	This is a function that provides device specific I/O control. See **sys/alt_dma_dev.h** for a list of the generic options that a device might wish to support.

The `prepare()` function is required to be defined. If the `ioctl` field is set to null, calls to `alt_dma_rxchan_ioctl()` return `-ENOTTY` for this device.

After creating an instance of the `alt_dma_rxchan` structure, you must register the device driver with the HAL system to make it available by calling the following function:

```
int alt_dma_rxchan_reg (alt_dma_rxchan_dev* dev)
```

The input argument dev is the device to register. The return value is zero upon success, or negative if the device cannot be registered.

Ethernet Device Drivers

The HAL generic device model for Ethernet devices provides access to the NicheStack® TCP/IP Stack - Nios II Edition running on the MicroC/OS-II operating system. You can provide support for a new Ethernet device by supplying the driver functions that this section defines.

Before you consider writing a device driver for a new Ethernet device, you need a basic understanding of the Altera implementation of the NicheStack TCP/IP Stack and its usages.

For more information, refer to the *Ethernet & the NicheStack® TCP/IP Stack - Nios II Edition* chapter of the *Nios II Software Developer's Handbook*.

The easiest way to write a new Ethernet device driver is to start with Altera's implementation for the SMSC lan91c111 device, and modify it to suit your Ethernet media access controller (MAC). This section assumes you take this approach. Starting from a known-working example makes it easier for you to learn the most important details of the NicheStack TCP/IP Stack implementation.

The source code for the lan91c111 driver is provided with the Quartus II software in <*Altera installation*>**/ip/sopc_builder_ip/altera_avalon_lan91c111/UCOSII**. For the sake of brevity, this section refers to this directory as ***<SMSC path>***. The source files are in the ***<SMSC path>*****/src/iniche** and ***<SMSC path>*****/inc/iniche** directories.

A number of useful NicheStack TCP/IP Stack files are installed with the Nios II EDS, under the <*Nios II EDS install path*>**/components/altera_iniche/UCOSII** directory. For the sake of brevity, this chapter refers to this directory as ***<iniche path>***.

For more information on the NicheStack TCP/IP Stack implementation, see the *NicheStack Technical Reference*, installed in the ***<iniche path>*****/src/downloads/packages/** directory. The reference manual is in **NicheStackRef.zip**.

You need not edit the NicheStack TCP/IP Stack source code to implement a NicheStack-compatible driver. Nevertheless, Altera provides the source code for your reference. The files are installed with the Nios II EDS in the ***<iniche_path>*** directory. The Ethernet device driver interface is defined in ***<iniche path>*****/inc/alt_iniche_dev.h**.

The following sections describe how to provide a driver for a new Ethernet device.

Provide the NicheStack Hardware Interface Routines

The NicheStack TCP/IP Stack architecture requires several network hardware interface routines:

- Initialize hardware
- Send packet
- Receive packet
- Close
- Dump statistics

These routines are fully documented in the *Porting Engineer Provided Functions* chapter of the *NicheStack Technical Reference*. The corresponding function in the SMSC lan91c111 device driver are:

Table 5–4. SMSC lan91c111 Hardware Interface Routines

Prototype function	lan91c111 function	File	Notes
`n_init()`	`s91_init()`	**smsc91x.c**	The initialization routine can install an ISR if applicable
`pkt_send()`	`s91_pkt_send()`	**smsc91x.c**	
Packet receive mechanism	`s91_isr()`	**smsc91x.c**	Packet receive includes three key actions: ● `pk_alloc()` — allocate a `netbuf` structure ● `putq()` — place `netbuf` structure on `rcvdq` ● `SignalPktDemux()` — notify the IP layer so that it can demux the packet
	`s91_rcv()`	**smsc91x.c**	
	`s91_dma_rx_done()`	**smsc_mem.c**	
`n_close()`	`s91_close()`	**smsc91x.c**	
`n_stats()`	`s91_stats()`	**smsc91x.c**	

The NicheStack TCP/IP Stack system code uses the `net` structure internally to define its interface to device drivers. The `net` structure is defined in **net.h**, in **<*iniche path*>/src/downloads/30src/h**. Among other things, the `net` structure contains the following things:

- A field for the IP address of the interface
- A function pointer to a low-level function to initialize the MAC device
- Function pointers to low-level functions to send packets

Typical NicheStack code refers to type `NET`, which is defined as `*net`.

Provide *INSTANCE and *INIT Macros

So that the HAL can use your driver, you must provide two HAL macros. The names of these macros are based on the name of your network interface component, according to the following templates:

- `ALTERA_AVALON_`<*component name*>`_INSTANCE`
- `ALTERA_AVALON_`<*component name*>`_INIT`

For examples, see ALTERA_AVALON_LAN91C111_INSTANCE and ALTERA_AVALON_LAN91C111_INIT in **<*SMSC path*>/inc/iniche/altera_avalon_lan91c111_iniche.h**, which is included in **<*iniche path*>/inc/altera_avalon_lan91c111.h**.

You can copy **altera_avalon_lan91c111_iniche.h** and modify it for your own driver. The HAL expects to find the *INIT and *INSTANCE macros in **altera_avalon_<*component name*>.h**, as discussed in "Device Driver Files for the HAL" on page 5–18. You can accomplish this with a #include directive as in **altera_avalon_lan91c111.h**, or you can define the macros directly in **altera_avalon_<*component name*>.h.**

Your *INSTANCE macro declares data structures required by an instance of the MAC. These data structures must include an alt_iniche_dev structure. The *INSTANCE macro must initialize the first three fields of the alt_iniche_dev structure, as follows:

- The first field, llist, is for internal use, and must always be set to the value ALT_LLIST_ENTRY.
- The second field, name, must be set to the device name as defined in system.h. For example, **altera_avalon_lan91c111_iniche.h** uses the C preprocessor's ## (concatenation) operator to reference the LAN91C111_NAME symbol defined in **system.h**.
- The third field, init_func, must point to your software initialization function, as described in "Provide a Software Initialization Function". For example, **altera_avalon_lan91c111_iniche.h** inserts a pointer to alt_avalon_lan91c111_init().

Your *INIT macro initializes the driver software. Initialization must include a call to the alt_iniche_dev_reg() macro, defined in **alt_iniche_dev.h**. This macro registers the device with the HAL by adding the driver instance to alt_iniche_dev_list.

When your driver is included in a Nios II system library project, the HAL automatically initializes your driver by invoking the *INSTANCE and *INIT macros from its alt_sys_init() function. See "Device Driver Files for the HAL" on page 5–18 for further detail.

Provide a Software Initialization Function

The *INSTANCE() macro inserts a pointer to your initialization function into the alt_iniche_dev structure, as described in "Provide *INSTANCE and *INIT Macros" on page 5–15. Your software initialization function must do at least the three following things:

- Initialize the hardware and verify its readiness

- Finish initializing the `alt_iniche_dev` structure
- Call `get_mac_addr()`

The initialization function must perform any other initialization your driver needs, such as creation and initialization of custom data structures and ISRs.

For details about the `get_mac_addr()` function, see the *Ethernet & the NicheStack® TCP/IP Stack - Nios II Edition* chapter of the *Nios II Software Developer's Handbook*.

For an example of a software initialization function, see `alt_avalon_lan91c111_init()` in **<*SMSC path*>/src/iniche/ smsc91x.c**.

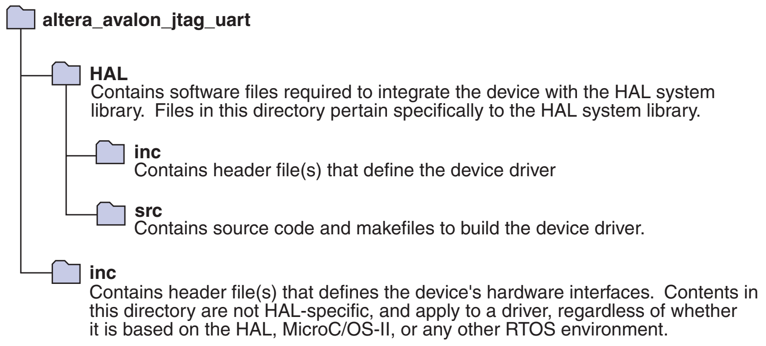
Integrating a Device Driver into the HAL

This section discusses how to take advantage of the HAL's ability to instantiate and register device drivers during system initialization. You can take advantage of this service, whether you created a device driver for one of the HAL generic device models, or you created a peripheral-specific device driver. Taking advantage of the automation provided by the HAL is mainly a process of placing files in the appropriate place in the HAL directory structure.

Directory Structure for HAL Devices

Each peripheral is defined by files provided in a specific SOPC Builder component directory. See "Accessing Hardware " on page 5–3. This section uses the example of Altera's JTAG UART component to demonstrate the location of files. Figure 5–1 shows the directory structure of the JTAG UART component directory, which is located in the <*Altera installation*>**/ip/sopc_builder_ip** directory.

Figure 5–1. HAL Peripheral's Directory Structure


altera_avalon_jtag_uart
HAL
Contains software files required to integrate the device with the HAL system library. Files in this directory pertain specifically to the HAL system library.
inc
Contains header file(s) that define the device driver
src
Contains source code and makefiles to build the device driver.
inc
Contains header file(s) that defines the device's hardware interfaces. Contents in this directory are not HAL-specific, and apply to a driver, regardless of whether it is based on the HAL, MicroC/OS-II, or any other RTOS environment.

Device Driver Files for the HAL

This section describes how to provide appropriate files to integrate your device driver into the HAL.

A Device's HAL Header File & alt_sys_init.c

At the heart of the HAL is the auto-generated source file, **alt_sys_init.c**. **alt_sys_init.c** contains the source code that the HAL uses to initialize the device drivers for all supported devices in the system. In particular, this file defines the `alt_sys_init()` function, which is called before `main()` to initialize all devices and make them available to the program.

The following code shows excerpts from an **alt_sys_init.c** file.

Example: Excerpt from an alt_sys_init.c File Performing Driver Initialization

```
#include "system.h"
#include "sys/alt_sys_init.h"

/*
 * device headers
 */
#include "altera_avalon_timer.h"
#include "altera_avalon_uart.h"

/*
 * Allocate the device storage
 */
ALTERA_AVALON_UART_INSTANCE( UART1, uart1 );
ALTERA_AVALON_TIMER_INSTANCE( SYSCLK, sysclk );

/*
 * Initialise the devices
 */
void alt_sys_init( void )
{
    ALTERA_AVALON_UART_INIT( UART1, uart1 );
    ALTERA_AVALON_TIMER_INIT( SYSCLK, sysclk );
}
```

When you create a new software project, the Nios II integrated development environment (IDE) generates the contents of **alt_sys_init.c** to match the specific hardware contents of the SOPC Builder system. The Nios II IDE calls the generator utility `gtf-generate` to create **alt_sys_init.c**.

☞ You do not need to call `gtf-generate` explicitly; it is mentioned here only because you might find references to `gtf-generate` in the low-level workings of the HAL.

For each device visible to the processor, the generator utility searches for an associated header file in the device's **HAL/inc** directory. The name of the header file depends on the SOPC Builder component name. For example, for Altera's JTAG UART component, the generator finds the file **altera_avalon_jtag_uart/HAL/inc/altera_avalon_jtag_uart.h**. If the generator utility finds such a header file, it inserts code into **alt_sys_init.c** to perform the following actions:

- Include the device's header file. See the `/* device headers */` in "Example: Excerpt from an alt_sys_init.c File Performing Driver Initialization" on page 5–18
- Call the macro <*name of device*>`_INSTANCE` to allocate storage for the device. See the `/* Allocate the device storage */` section in "Example: Excerpt from an alt_sys_init.c File Performing Driver Initialization" on page 5–18
- Call the macro <*name of device*>`_INIT` inside the `alt_sys_init()` function to initialize the device. See the `/* Initialize the devices */` section in "Example: Excerpt from an alt_sys_init.c File Performing Driver Initialization" on page 5–18

These `*_INSTANCE` and `*_INIT` macros must be defined in the associated device header file. For example, **altera_avalon_jtag_uart.h** must define the macros `ALTERA_AVALON_JTAG_UART_INSTANCE` and `ALTERA_AVALON_JTAG_UART_INIT`. The `*_INSTANCE` macro performs any per-device static memory allocation that the driver requires. The `*_INIT` macro performs runtime initialization of the device. Both macros take two input arguments: The first argument is the capitalized name of the device instance; the second is the lower case version of the device name. The name is the name given to the component in SOPC Builder at system generation time. You can use these input parameters to extract device-specific configuration information from the **system.h** file.

For a complete example, see any of the Altera-supplied device drivers.

To improve project rebuild time, the peripheral header file should not include **system.h** directly—it is already included in **alt_sys_init.c**.

To publish a device driver for an SOPC builder component, you provide the file **HAL/inc/**<*component_name*>**.h** within the components directory. This file is then required to define the macros <*COMPONENT_NAME*>`_INSTANCE` and <*COMPONENT_NAME*>`_INIT`, as described above. With this infrastructure in place for your device, the HAL system library instantiates and registers your device driver before calling `main()`.

Device Driver Source Code

In general, a device driver cannot be defined entirely by the header. See "A Device's HAL Header File & alt_sys_init.c" on page 5–18. The component almost certainly also needs to provide additional source code, which is to be built into the system library.

You should place any required source code in the **HAL/src** directory. In addition, you should include a makefile fragment, **component.mk**. The **component.mk** file lists the source files to include in the system library. You can list multiple files by separating filenames with a space. The following code shows an example makefile for Altera's JTAG UART device.

Example: An Example component.mk Makefile

```
C_LIB_SRCS   += altera_avalon_uart.c
ASM_LIB_SRCS +=
INCLUDE_PATH +=
```

The Nios II IDE includes the **component.mk** file into the top-level makefile when compiling system library projects and application projects. **component.mk** can modify any of the available make variables, but is restricted to `C_LIB_SRCS`, `ASM_LIB_SRCS`, and `INCLUDE_PATH`. Table 5–5 shows these variables.

Table 5–5. Make Variables Defined in component.mk

Make Variable	Meaning
`C_LIB_SRCS`	The list of C source files to build into the system library.
`ASM_LIB_SRCS`	The list of assembler source files to build into the system library (these are preprocessed with the C preprocessor).
`INCLUDE_PATH`	A list of directories to add to the include search path. The directory <*component*>**/HAL/inc** is added automatically and so does not need to be explicitly defined by the component.

component.mk can add additional make rules and macros as required, but interoperability macro names should conform to the namespace rules. See "Namespace Allocation" on page 5–23

Summary

In summary, to integrate a device driver into the HAL framework, you must perform the following actions:

- Create an include file that defines the `*_INSTANCE` and `*_INIT` macros and place it in the device's **HAL/inc** directory
- Create source code files that manipulates the device, and place the files into the device's **HAL/src** directory
- Write a makefile fragment, **component.mk**, and place it in the **HAL/src** directory

Reducing Code Footprint

The HAL provides several options for reducing the size, or footprint, of the system library code. Some of these options require explicit support from device drivers. If you need to minimize the size of your software, consider using one or more of the following techniques in your custom device driver:

- Provide reduced footprint drivers. This technique usually reduces driver functionality.
- Support the lightweight device driver API. This technique reduces driver overhead, and might restrict your flexibility in using the driver.

These techniques are discussed in the following sections.

Provide Reduced Footprint Drivers

The HAL defines a C preprocessor macro named `ALT_USE_SMALL_DRIVERS` that you can use in driver source code to provide alternate behavior for systems that require minimal code footprint. An option in the Nios II IDE allows you to enable reduced device drivers. If `ALT_USE_SMALL_DRIVERS` is not defined, driver source code implements a fully featured version of the driver. If the macro is defined, the source code might provide a driver with restricted functionality. For example a driver might implement interrupt-driven operation by default, but polled (and presumable smaller) operation if `ALT_USE_SMALL_DRIVERS` is defined.

When writing a device driver, if you choose to ignore the value of `ALT_USE_SMALL_DRIVERS`, the same version of the driver is used regardless of the definition of this macro.

Support the Lightweight Device Driver API

The lightweight device driver API allows you to minimize the overhead of character-mode device drivers. It does this by removing the need for the `alt_fd` file descriptor table, and the `alt_dev` data structure required by each driver instance.

If you want to support the lightweight device driver API on a character-mode device, you need to write at least one of the lightweight character-mode functions: <*component_class*>`_read()`, <*component_class*>`_write()` and <*component_class*>`_ioctl()`. Implement the functions needed by your software. For example, if you only use the device for `stdout`, you only need to implement the <*component_class*>`_write()` function.

To support the lightweight device driver API, name your driver functions based on the component class name, as shown in Table 5–6.

Table 5–6. Driver Functions for Lightweight Device Driver API

Function	Example (1)
<*component_class*>`_read()`	`altera_avalon_jtag_uart_read()`
<*component_class*>`_write()`	`altera_avalon_jtag_uart_write()`
<*component_class*>`_ioctl()`	`altera_avalon_jtag_uart_ioctl()`

(1) Based on component **altera_avalon_jtag_uart**

When you build your system library in the Nios II IDE with the **Lightweight Device Driver API** option turned on, instead of using file descriptors, the HAL accesses your drivers with the following macros:

- `ALT_DRIVER_READ(instance, buffer, len, flags)`
- `ALT_DRIVER_WRITE(instance, buffer, len, flags)`
- `ALT_DRIVER_IOCTL(instance, req, arg)`

These macros are defined in ***<Nios II EDS install path>*****/components/ altera_hal/HAL/inc/sys/alt_driver.h**.

These macros, together with the system-specific macros that the Nios II IDE creates in **system.h**, generate calls to your driver functions. For example, with the **Lightweight Device Driver API** options turned on, `printf()` calls the HAL `write()` function, which directly calls your driver's <*component_class*>`_write()` function, bypassing file descriptors.

You can also take advantage of the lightweight device driver API by invoking `ALT_DRIVER_READ()`,`ALT_DRIVER_WRITE()` and `ALT_DRIVER_IOCTL()` in your application software. To use these macros, include the header file **sys/alt_driver.h**. Replace the `instance` argument with the device instance name macro from **system.h**; or if you are confident that the device instance name will never change, you can use a literal string, e.g. `"custom_uart_0"`.

Another way to use your driver functions is to call them directly, without macros. If your driver includes functions other than <*component_class*>`_read()`, <*component_class*>`_write()` and <*component_class*>`_ioctl()`, you must invoke those functions directly from your application.

Namespace Allocation

To avoid conflicting names for symbols defined by devices in the SOPC Builder system, all global symbols need a defined prefix. Global symbols include global variable and function names. For device drivers, the prefix is the name of the SOPC Builder component followed by an underscore. Because this naming can result in long strings, an alternate short form is also permitted. This short form is based on the vendor name, for example `alt_` is the prefix for components published by Altera. It is expected that vendors test the interoperability of all components they supply.

For example, for the `altera_avalon_jtag_uart` component, the following function names are valid:

- `altera_avalon_jtag_uart_init()`
- `alt_jtag_uart_init()`

The following names are invalid:

- `avalon_jtag_uart_init()`
- `jtag_uart_init()`

As source files are located using search paths, these namespace restrictions also apply to filenames for device driver source and header files.

Overriding the Default Device Drivers

All SOPC Builder components can elect to provide a HAL device driver. See "Integrating a Device Driver into the HAL" on page 5–17. However, if the driver supplied with a component is inappropriate for your application, you can override the default driver by supplying a different one in the system library project directory in the Nios II IDE.

The Nios II IDE locates all include and source files using search paths, and the system library project directory is always searched first. For example, if a component provides the header file **alt_my_component.h**, and the system library project directory also contains a file **alt_my_component.h**, the version provided in the system library project directory is used at compile time. This same mechanism can override C and assembler source files.

Document Revision History

Table 5–7 shows the revision history for this document.

Table 5–7. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
November 2006, v6.1.0	● Add section "Reducing Code Footprint" ● Replace lwIP driver section with NicheStack TCP/IP Stack driver section	Lightweight device driver API and minimal file I/O API; NicheStack TCP/IP Stack support.
May 2006, v6.0.0	No change from previous release.	
October 2005, v5.1.0	● Added IOADDR_* macro details to section "Accessing Hardware ".	
May 2005, v5.0.0	Updated reference to version of lwIP from 0.7.2 to 1.1.0.	
December 2004 v1.1	Updated reference to version of lwIP from 0.6.3 to 0.7.2.	
May 2004 v1.0	First publication.	

Section III. Advanced Programming Topics

This section provides information on advanced programming topics.

This section includes the following chapters:

6. Exception Handling

NII52006-6.1.0

Introduction

This chapter discusses how to write programs to handle exceptions in the Nios® II processor architecture. Emphasis is placed on how to process hardware interrupt requests by registering a user-defined interrupt service routine (ISR) with the hardware abstraction layer (HAL).

This chapter covers the following topics:

For low-level details of handling exceptions and interrupts on the Nios II architecture, see the *Programming Model* chapter of the *Nios II Processor Reference Handbook.*

Nios II Exceptions Overview

Nios II exception handling is implemented in classic RISC fashion, i.e., all exception types are handled by a single exception handler. As such, all exceptions (hardware and software) are handled by code residing at a single location called the "exception address".

The Nios II processor provides the following exception types:

- Hardware interrupts
- Software exceptions, which fall into the following categories:
  - Unimplemented instructions
  - Software traps
  - Other exceptions

Exception Handling Concepts

The following list outlines basic exception handling concepts, with the HAL terms used for each one:

- **application context** — the status of the Nios II processor and the HAL during normal program execution, outside of the exception handler.
- **context switch** — the process of saving the Nios II processor's registers on an exception, and restoring them on return from the interrupt service routine.
- **exception** — any condition or signal that interrupts normal program execution.
- **exception handler** — the complete system of software routines, which service all exceptions and pass control to ISRs as necessary.
- **exception overhead** — additional processing required by exception processing. The exception overhead for a program is the sum of all the time occupied by all context switches.
- **hardware interrupt** — an exception caused by a signal from a hardware device.
- **implementation-dependent instruction** — a Nios II processor instruction that is not supported on all implementations of the Nios II core. For example, the `mul` and `div` instructions are implementation-dependent, because they are not supported on the Nios II/e core.
- **interrupt context** — the status of the Nios II processor and the HAL when the exception handler is executing.
- **interrupt request (IRQ)** — a signal from a peripheral requesting a hardware interrupt.
- **interrupt service routine (ISR)** — a software routine that handles an individual hardware interrupt.
- **invalid instruction** — an instruction that is not defined for any implementation of the Nios II processor.
- **software exception** — an exception caused by a software condition. This includes unimplemented instructions and `trap` instructions.
- **unimplemented instruction** — an implementation-dependent instruction that is not supported on the particular Nios II core implementation that is in your system. For example, in the Nios II/e core, `mul` and `div` are unimplemented.
- **other exception** — an exception which is not a hardware interrupt nor a trap.

How the Hardware Works

The Nios II processor can respond to software exceptions and hardware interrupts. Thirty-two independent hardware interrupt signals are available. These interrupt signals allow software to prioritize interrupts, although the interrupt signals themselves have no inherent priority.

When the Nios II processor responds to an exception, it does the following things:

1. Saves the `status` register in `estatus`. This means that if hardware interrupts are enabled, the `EPIE` bit of `estatus` is set.

2. Disables hardware interrupts.

3. Saves the next execution address in `ea` (`r29`).

4. Transfers control to the Nios II processor exception address.

Nios II exceptions and interrupts are not vectored. Therefore, the same exception address receives control for all types of interrupts and exceptions. The exception handler at that address must determine the type of exception or interrupt.

For details about the Nios II processor exception and interrupt controller, see the *Processor Architecture* chapter of the *Nios II Processor Reference Handbook*.

ISRs

Software often communicates with peripheral devices using interrupts. When a peripheral asserts its IRQ, it causes an exception to the processor's normal execution flow. When such an IRQ occurs, an appropriate ISR must handle this interrupt and return the processor to its pre-interrupt state upon completion.

When you use the Nios II IDE to create a system library project, the IDE includes all needed ISRs. You do not need to write HAL ISRs unless you are interfacing to a custom peripheral. For reference purposes, this section describes the framework provided by the HAL system library for handling hardware interrupts.

You can also look at existing handlers for Altera® SOPC Builder components for examples of how to write HAL ISRs.

For more details about the Altera-provided HAL handlers, see the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook.*

HAL API for ISRs

The HAL system library provides an API to help ease the creation and maintenance of ISRs. This API also applies to programs based on a real-time operating system (RTOS) such as MicroC/OS-II, because the full HAL API is available to RTOS-based programs. The HAL API defines the following functions to manage hardware interrupt processing:

- `alt_irq_register()`
- `alt_irq_disable()`
- `alt_irq_enable()`
- `alt_irq_disable_all()`
- `alt_irq_enable_all()`
- `alt_irq_interruptible()`
- `alt_irq_non_interruptible()`
- `alt_irq_enabled()`

For details on these functions, see the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

Using the HAL API to implement ISRs entails the following steps:

1. Write your ISR that handles interrupts for a specific device.

2. Your program must register the ISR with the HAL by calling the `alt_irq_register()` function. `alt_irq_register()` enables interrupts for you, by calling `alt_irq_enable_all()`.

Writing an ISR

The ISR you write must match the prototype that `alt_irq_register()` expects to see. The prototype for your ISR function must match the prototype:

```
void isr (void* context, alt_u32 id)
```

The parameter definitions of `context` and `id` are the same as for the `alt_irq_register()` function.

From the point of view of the HAL exception handling system, the most important function of an ISR is to clear the associated peripheral's interrupt condition. The procedure for clearing an interrupt condition is specific to the peripheral.

For details, see the relevant chapter in the *Quartus® II Version 6.0 Handbook, Volume 5: Altera Embedded Peripherals.*

When the ISR has finished servicing the interrupt, it must return to the HAL exception handler.

If you write your ISR in assembly language, use `ret` to return. The HAL exception handler issues an `eret` after restoring the application context.

Restricted Environment

ISRs run in a restricted environment. A large number of the HAL API calls are not available from ISRs. For example, accesses to the HAL file system are not permitted. As a general rule, when writing your own ISR, never include function calls that can block waiting for an interrupt.

The *HAL API Reference* chapter of the *Nios II Software Developer's Handbook* identifies those API functions that are not available to ISRs.

Be careful when calling ANSI C standard library functions inside of an ISR. Avoid using the C standard library I/O API, because calling these functions can result in deadlock within the system, i.e., the system can become permanently blocked within the ISR.

In particular, do not call `printf()` from within an ISR unless you are certain that `stdout` is mapped to a non-interrupt-based device driver. Otherwise, `printf()` can deadlock the system, waiting for an interrupt that never occurs because interrupts are disabled.

Registering an ISR

Before the software can use an ISR, you must register it by calling `alt_irq_register()`. The prototype for `alt_irq_register()` is:

```
int alt_irq_register (alt_u32 id,
                      void*   context,
                      void    (*isr)(void*, alt_u32));
```

The prototype has the following parameters:

- `id` is the hardware interrupt number for the device, as defined in **system.h**. Interrupt priority corresponds inversely to the IRQ number. Therefore, IRQ_0 represents the highest priority interrupt and IRQ_{31} is the lowest.
- `context` is a pointer used to pass context-specific information to the ISR, and can point to any ISR-specific information. The context value is opaque to the HAL; it is provided entirely for the benefit of the user-defined ISR.

- `isr` is a pointer to the function that is called in response to IRQ number id. The two input arguments provided to this function are the `context` pointer and `id`. Registering a null pointer for `isr` results in the interrupt being disabled.

The HAL registers the ISR by the storing the function pointer, `isr`, in a lookup table. The return code from `alt_irq_register()` is zero if the function succeeded, and nonzero if it failed.

If the HAL registers your ISR successfully, the associated Nios II interrupt (as defined by `id`) is enabled on return from `alt_irq_register()`.

Hardware-specific initialization might also be required.

When a specific IRQ occurs, the HAL looks up the IRQ in the lookup table and dispatches the registered ISR.

For details of interrupt initialization specific to your peripheral, see the relevant chapter in the *Quartus II Version 6.0 Handbook, Volume 5: Altera Embedded Peripherals*. For details on `alt_irq_register()`, see the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

Enabling and Disabling ISRs

The HAL provides the functions `alt_irq_disable()`, `alt_irq_enable()`, `alt_irq_disable_all()`, `alt_irq_enable_all()`, and `alt_irq_enabled()` to allow a program to disable interrupts for certain sections of code, and re-enable them later. `alt_irq_disable()` and `alt_irq_enable()` allow you to disable and enable individual interrupts. `alt_irq_disable_all()` disables all interrupts, and returns a context value. To re-enable interrupts, you call `alt_irq_enable_all()` and pass in the context parameter. In this way, interrupts are returned to their state prior to the call to `alt_irq_disable_all()`. `alt_irq_enabled()` returns non-zero if interrupts are enabled, allowing a program to check on the status of interrupts.

Disable interrupts for as short a time as possible. Maximum interrupt latency increases with the amount of time interrupts are disabled. For more information about disabled interrupts, see "Keep Interrupts Enabled" on page 6–11.

For details on these functions, see the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

C Example

The following code illustrates an ISR that services an interrupt from a button PIO. This example is based on a Nios II system with a 4-bit PIO peripheral connected to push-buttons. An IRQ is generated any time a button is pushed. The ISR code reads the PIO peripheral's edge-capture register and stores the value to a global variable. The address of the global variable is passed to the ISR via the context pointer.

Example: An ISR to Service a Button PIO IRQ

```
#include "system.h"
#include "altera_avalon_pio_regs.h"
#include "alt_types.h"

static void handle_button_interrupts(void* context, alt_u32 id)
{
  /* cast the context pointer to an integer pointer. */
  volatile int* edge_capture_ptr = (volatile int*) context;

  /*
   * Read the edge capture register on the button PIO.
   * Store value.
   */
  *edge_capture_ptr =
    IORD_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE);

  /* Write to the edge capture register to reset it. */
  IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE, 0);

  /* reset interrupt capability for the Button PIO. */
  IOWR_ALTERA_AVALON_PIO_IRQ_MASK(BUTTON_PIO_BASE, 0xf);
}
```

The following code shows an example of the code for the main program that registers the ISR with the HAL.

Example: Registering the Button PIO ISR with the HAL

```
#include "sys/alt_irq.h"
#include "system.h"

...
/* Declare a global variable to hold the edge capture value. */
volatile int edge_capture;
...

/* Initialize the button_pio. */
static void init_button_pio()
{
    /* Recast the edge_capture pointer to match the
        alt_irq_register() function prototype. */
    void* edge_capture_ptr = (void*) &edge_capture;

    /* Enable all 4 button interrupts. */
    IOWR_ALTERA_AVALON_PIO_IRQ_MASK(BUTTON_PIO_BASE, 0xf);
    /* Reset the edge capture register. */
    IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE, 0x0);

    /* Register the ISR. */
    alt_irq_register( BUTTON_PIO_IRQ,
                      edge_capture_ptr,
                      handle_button_interrupts );
}
```

Based on this code, the following execution flow is possible:

1. Button is pressed, generating an IRQ.
2. The HAL exception handler is invoked and dispatches the `handle_button_interrupts()` ISR.
3. `handle_button_interrupts()` services the interrupt and returns.
4. Normal program operation continues with an updated value of `edge_capture`.

Further software examples that demonstrate implementing ISRs are installed with the Nios II Embedded Design Suite (EDS), such as the `count_binary` example project template.

ISR Performance Data

This section provides performance data related to ISR processing on the Nios II processor. The following three key metrics determine ISR performance:

- Interrupt latency—the time from when an interrupt is first generated to when the processor runs the first instruction at the exception address.
- Interrupt response time—the time from when an interrupt is first generated to when the processor runs the first instruction in the ISR.
- Interrupt recovery time—the time taken from the last instruction in the ISR to return to normal processing.

Because the Nios II processor is highly configurable, there is no single typical number for each metric. This section provides data points for each of the Nios II cores under the following assumptions:

- All code and data is stored in on-chip memory.
- The ISR code does not reside in the instruction cache.
- The software under test is based on the Altera-provided HAL exception handler system.

- The code is compiled using compiler optimization level "–O3", or high optimization.

Table 6–1 lists the interrupt latency, response time, and recovery time for each Nios II core.

Table 6–1. Interrupt Performance Data (1)

Core	Latency	Response Time	Recovery Time
Nios II/f	10	105	62
Nios II/s	10	128	130
Nios II/e	15	485	222

Note to Table 6–1:
(1) The numbers indicate time measured in CPU clock cycles.

The results you experience in a specific application can vary significantly based on several factors discussed in the next section.

Improving ISR Performance

If your software uses interrupts extensively, the performance of ISRs is probably the most critical determinant of your overall software performance. This section discusses both hardware and software strategies to improve ISR performance.

Software Performance Improvements

In improving your ISR performance, you probably consider software changes first. However, in some cases it might require less effort to implement hardware design changes that increase system efficiency. For a discussion of hardware optimizations, see "Hardware Performance Improvements" on page 6–13.

The following sections describe changes you can make in the software design to improve ISR performance.

Move Lengthy Processing to Application Context

ISRs provide rapid, low latency response to changes in the state of hardware. They do the minimum necessary work to clear the interrupt condition and then return. If your ISR performs lengthy, noncritical processing, it interferes with more critical tasks in the system.

If lengthy processing is needed, design your software to perform this processing outside of the interrupt context. The ISR can use a message-passing mechanism to notify the application code to perform the lengthy processing tasks.

Deferring a task is simple in systems based on an RTOS such as MicroC/OS-II. In this case, you can create a thread to handle the processor-intensive operation, and the ISR can communicate with this thread using any of the RTOS communication mechanisms, such as event flags or message queues.

You can emulate this approach in a single-threaded HAL-based system. The main program polls a global variable managed by the ISR to determine whether it needs to perform the processor-intensive operation.

Move Lengthy Processes to Hardware

Processor-intensive tasks must often transfer large amounts of data to and from peripherals. A general-purpose CPU such as the Nios II processor is not the most efficient way to do this.

Use Direct Memory Access (DMA) hardware if it is available.

For information about programming with DMA hardware, refer to the *Using DMA Devices* section of the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*.

Increase Buffer Size

If you are using DMA to transfer large data buffers, the buffer size can affect performance. Small buffers imply frequent IRQs, which lead to high overhead.

Increase the size of the transaction data buffer(s).

Use Double Buffering

Using DMA to transfer large data buffers might not provide a large performance increase if the Nios II processor must wait for DMA transactions to complete before it can perform the next task.

Double buffering allows the Nios II processor to process one data buffer while the hardware is transferring data to or from another.

Keep Interrupts Enabled

When interrupts are disabled, the Nios II processor cannot respond quickly to hardware events. Buffers and queues can fill or overflow. Even in the absence of overflow, maximum interrupt processing time can increase after interrupts are disabled, because the ISRs must process data backlogs.

Disable interrupts as little as possible, and for the briefest time possible.

Instead of disabling all interrupts, call `alt_irq_disable()` and `alt_irq_enable()` to enable and disable individual IRQs.

To protect shared data structures, use RTOS structures such as semaphores.

Disable all interrupts only during critical system operations. In the code where interrupts are disabled, perform only the bare minimum of critical operations, and re-enable interrupts immediately.

Use Fast Memory

ISR performance depends upon memory speed.

Place the ISRs and the stack in the fastest available memory.

For best performance, place the stack in on-chip memory, preferably tightly-coupled memory, if available.

If it is not possible to place the main stack in fast memory, you can use a private exception stack, mapped to a fast memory section. However, the private exception stack entails some additional context switch overhead, so use it only if you are able to place it in significantly faster memory. You can specify a private exception stack on the **System properties** page of the Nios II IDE.

For more information about mapping memory, see the "Memory Usage" section of the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*. For more information on tightly-coupled memory, refer to the *Cache & Tightly-Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.

Use Nested ISRs

The HAL system library disables interrupts when it dispatches an ISR. This means that only one ISR can execute at any time, and ISRs are executed on a first-come-first-served basis.This reduces the system overhead associated with interrupt processing, and simplifies ISR development, because the ISR does not need to be reentrant.

However, first-come first-served execution means that the HAL interrupt priorities only have effect if two IRQs are asserted on the same application-level instruction cycle. A low-priority interrupt occurring before a higher-priority IRQ can prevent the higher-priority ISR from executing. This is a form of priority inversion, and it can have a significant impact on ISR performance in systems that generate frequent interrupts.

A software system can achieve full interrupt prioritization by using nested ISRs. With nested ISRs, higher priority interrupts are allowed to interrupt lower-priority ISRs.

This technique can improve the interrupt latency of higher priority ISRs.

☞ Nested ISRs increase the processing time for lower priority interrupts.

If your ISR is very short, it might not be worth the overhead to re-enable higher-priority interrupts. Enabling nested interrupts in a short ISR can actually increase the interrupt latency of higher priority interrupts.

☞ If you use a private exception stack, you cannot nest interrupts. For more information about private exception stacks, see "Use Fast Memory" on page 6–11.

To implement nested interrupts, use the `alt_irq_interruptible()` and `alt_irq_non_interruptible()` functions to bracket code within a processor-intensive ISR. After the call to `alt_irq_interruptible()`, higher priority IRQs can interrupt the running ISR. When your ISR calls `alt_irq_non_interruptible()`, interrupts are disabled as they were before `alt_irq_interruptible()`.

☞ If your ISR calls `alt_irq_interruptible()`, it must call `alt_irq_non_interruptible()` before returning. Otherwise, the HAL exception handler might lock up.

Use Compiler Optimization

For the best performance both in exception context and application context, use compiler optimization level `-O3`. Level `-O2` also produces good results. Removing optimization altogether significantly increases interrupt response time.

For further information about compiler optimizations, refer to the *Reducing Code Footprint* section in the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*.

Hardware Performance Improvements

There are several simple hardware changes that can provide a substantial improvement in ISR performance. These changes involve editing and regenerating the SOPC Builder module, and recompiling the Quartus II design.

In some cases, these changes also require changes in the software architecture or implementation. For a discussion of these and other software optimizations, see "Software Performance Improvements" on page 6–9.

The following sections describe changes you can make in the hardware design to improve ISR performance.

Add Fast Memory

Increase the amount of fast on-chip memory available for data buffers. Ideally, implement tightly-coupled memory which the software can use for buffers.

For further information about tightly-coupled memory, refer to the *Cache & Tightly-Coupled Memory* chapter in the *Nios II Processor Reference Handbook*, or to the *Using Nios II Tightly Coupled Memory Tutorial*.

Add a DMA Controller

A DMA controller performs bulk data transfers, reading data from a source address range and writing the data to a different address range. Add DMA controllers to move large data buffers. This allows the Nios II processor to carry out other tasks while data buffers are being transferred.

For information about DMA controllers, see the *DMA Controller with Avalon Interface* chapter in Volume 5 of the *Quartus II Handbook*.

Place the Exception Handler Address in Fast Memory

For the fastest execution of exception code, place the exception address in a fast memory device. For example, an on-chip RAM with zero waitstates is preferable to a slow SDRAM. For best performance, store exception handling code and data in tightly-coupled memory. The Nios II EDS includes example designs that demonstrate the use of tightly-coupled memory for ISRs.

Use a Fast Nios II Core

For processing in both the interrupt context and the application context, the Nios II/f core is the fastest, and the Nios II/e core (designed for small size) is the slowest.

Select Interrupt Priorities

When selecting the IRQ for each peripheral, bear in mind that the HAL hardware interrupt handler treats IRQ_0 as the highest priority. Assign each peripheral's interrupt priority based on its need for fast servicing in the overall system. Avoid assigning multiple peripherals to the same IRQ.

Use the Interrupt Vector Custom Instruction

The Nios II processor core offers an interrupt vector custom instruction which accelerates interrupt vector dispatch in the Hardware Abstraction Layer (HAL). You can choose to include this custom instruction to improve your program's interrupt response time.

When the interrupt vector custom instruction is present in the Nios II processor, the HAL source detects it at compile time and generates code using the custom instruction.

For further information about the interrupt vector custom instruction, see the *Interrupt Vector Custom Instruction* section in the chapter entitled *Implementing the Nios II Processor in SOPC Builder* in the *Nios II Processor Reference Handbook.*

Debugging ISRs

You can debug an ISR with the Nios II IDE by setting breakpoints within the ISR. The debugger completely halts the processor upon reaching a breakpoint. In the meantime, however, the other hardware in your system continues to operate. Therefore, it is inevitable that other IRQs are ignored while the processor is halted. You can use the debugger to step through the ISR code, but the status of other interrupt-driven device drivers is generally invalid by the time you return the processor to normal execution. You have to reset the processor to return the system to a known state.

The `ipending` register (`ctl4`) is masked to all zeros during single stepping. This masking prevents the processor from servicing IRQs that are asserted while you single-step through code. As a result, if you try to single step through a part of the exception handler code (e.g. `alt_irq_entry()` or `alt_irq_handler()`) that reads the `ipending` register, the code does not detect any pending IRQs. This issue does not affect debugging software exceptions. You can set breakpoints within your ISR code (and single step through it), because the exception handler has already used `ipending` to determine which IRQ caused the exception.

Summary of Guidelines for Writing ISRs

This section summarizes guidelines for writing ISRs for the HAL framework:

- Write your ISR function to match the prototype: `void isr (void* context, alt_u32 id)`.
- Register your ISR using the `alt_irq_register()` function provided by the HAL API.
- Do not use the C standard library I/O functions, such as `printf()`, inside of an ISR.

HAL Exception Handler Implementation

This section describes the HAL exception handler implementation. This is one of many possible implementations of an exception handler for the Nios II processor. Some features of the HAL exception handler are constrained by the Nios II hardware, while others are designed to provide generally useful services.

This information is for your reference. You can take advantage of the HAL exception services without a complete understanding of the HAL implementation. For details of how to install ISRs using the HAL application programming interface (API), see "ISRs" on page 6–3.

Exception Handler Structure

The exception handling system consists of the following components:

- The top-level exception handler
- The hardware interrupt handler
- The software exception handler
- An ISR for each peripheral that generates interrupts.

When the Nios II processor generates an exception, the top-level exception handler receives control. The top-level exception handler passes control to either the hardware interrupt handler or the software exception handler. The hardware interrupt handler passes control to one or more ISRs.

Each time an exception occurs, the exception handler services either a software exception or hardware interrupts, with hardware interrupts having a higher priority. The HAL does not support nested exceptions, but can handle multiple hardware interrupts per context switch. For details, see "Hardware Interrupt Handler" on page 6–18.

Top-Level Exception Handler

The top-level exception handler provided with the HAL system library is located at the Nios II processor's exception address. When an exception occurs and control transfers to the exception handler, it does the following:

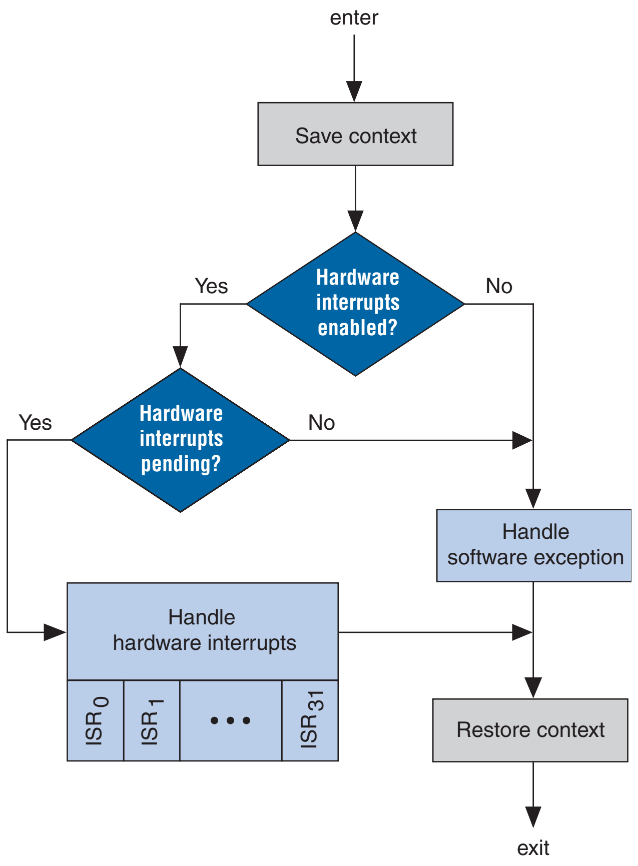
1. Creates the private exception stack (if specified)

2. Stores register values onto the stack

3. Determines the type of exception, and passes control to the correct handler

Figure 6–1 shows the algorithm that HAL top-level exception handler uses to distinguish between hardware interrupts and software exceptions.

Figure 6–1. HAL Top-Level Exception Handler


enter
Save context
Hardware interrupts enabled?
Yes
No
Hardware interrupts pending?
Yes
No
Handle software exception
Handle hardware interrupts
ISR 0
ISR 1
• • •
ISR 31
Restore context
exit

The top-level exception handler looks at the `estatus` register to determine the interrupt enable status. If the `EPIE` bit is set, hardware interrupts were enabled at the time the exception happened. If so, the exception handler looks at the IRQ bits in `ipending`. If any IRQs are asserted, the exception handler calls the hardware interrupt handler.

If hardware interrupts are not enabled at the time of the exception, it is not necessary to look at `ipending`.

If no IRQs are active, there is no hardware interrupt, and the exception is a software exception. In this case, the top-level exception handler calls the software exception handler.

All hardware interrupts are higher priority than software exceptions.

For details on the Nios II processor `estatus` and `ipending` registers, see the *Programming Model* chapter of the *Nios II Processor Reference Handbook.*

Upon return from the hardware interrupt or software exception handler, the top-level exception handler does the following:

1. Restores the stack pointer, if a private exception stack is used
2. Restores the registers from the stack
3. Exits by issuing an `eret` (exception return) instruction

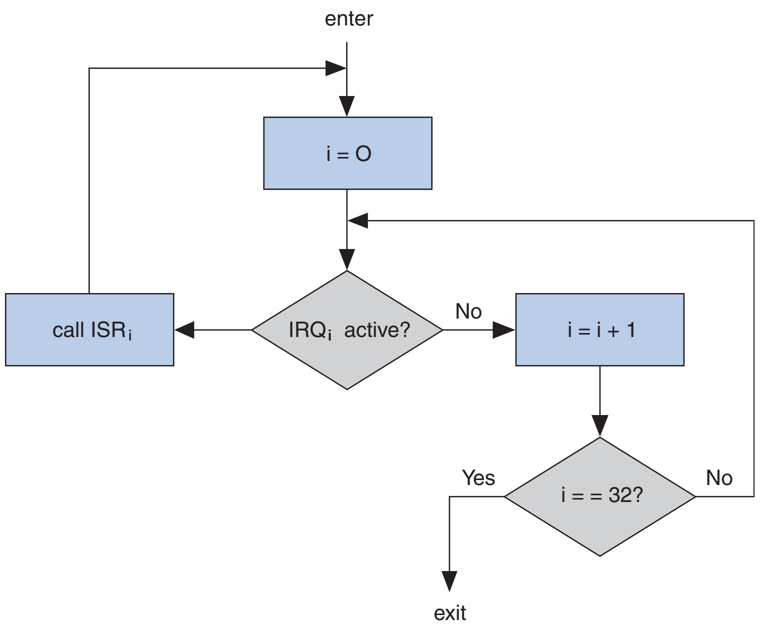
Hardware Interrupt Handler

The Nios II processor supports thirty-two hardware interrupts. In the HAL exception handler, hardware interrupt 0 has the highest priority, and 31 the lowest. This prioritization is a feature of the HAL exception handler, and is not inherent in the Nios II exception and interrupt controller.

The hardware interrupt handler calls the user-registered ISRs. It goes through the IRQs in `ipending` starting at 0, and finds the first (highest priority) active IRQ. Then it calls the corresponding registered ISR. After this ISR executes, the exception handler begins scanning the IRQs again, starting at IRQ_0. In this way, higher priority exceptions are always processed before lower-priority exceptions. When all IRQs are clear, the hardware interrupt handler returns to the top level. Figure 6–2 shows a flow diagram of the HAL hardware interrupt handler.

When the interrupt vector custom instruction is present in the Nios II processor, the HAL source detects it at compile time and generates code using the custom instruction. For further information, see "Use the Interrupt Vector Custom Instruction" on page 6–14.

Figure 6–2. HAL Hardware Interrupt Handler



Software Exception Handler

Software exceptions can include unimplemented instructions, traps, and other exceptions.

Software exception handling depends on options selected in the Nios II IDE. If you have enabled unimplemented instruction emulation, the exception handler first checks to see if an unimplemented instruction caused the exception. If so, it emulates the instruction. Otherwise, it handles traps and other exceptions.

Unimplemented Instructions

You can include a handler to emulate unimplemented instructions. The Nios II processor architecture defines the following implementation-dependent instructions:

- `mul`
- `muli`
- `mulxss`
- `mulxsu`

- `mulxuu`
- `div`
- `divu`

For details on unimplemented instructions, see the *Processor Architecture* chapter of the *Nios II Processor Reference Handbook.*

Unimplemented instructions are different from invalid instructions, which are described in "Invalid Instructions" on page 6–23.

When to Use the Unimplemented Instruction Handler

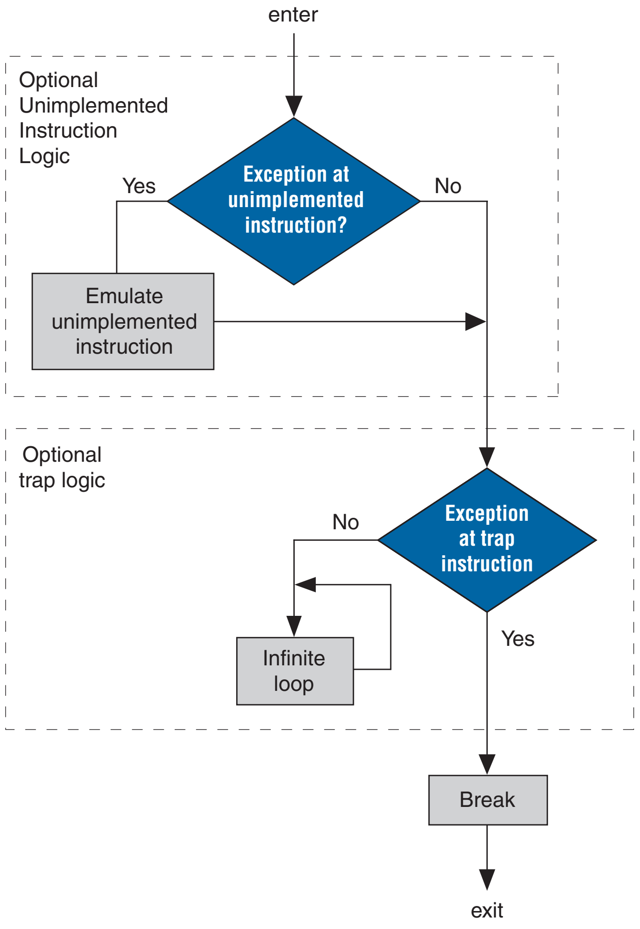
You do not normally need the unimplemented instruction handler, because the Nios II IDE includes software emulation for unimplemented instructions from its run-time libraries if you are compiling for a Nios II processor that does not support the instructions.

Here are the circumstances under which you might need the unimplemented instruction handler:

- You are running a Nios II program on an implementation of the Nios II processor other than the one you compiled for. The best solution is to build your program for the correct Nios II processor implementation. Only if this is not possible should you resort to the unimplemented instruction handler.
- You have assembly language code that uses an implementation-dependent instruction.

Figure 6–3 shows a flowchart of the HAL software exception handler, including the optional instruction emulation logic. If instruction emulation is not enabled, this logic is omitted.

Figure 6–3. HAL Software Exception Handler


enter
Optional Unimplemented Instruction Logic
Exception at unimplemented instruction?
Yes
No
Emulate unimplemented instruction
Optional trap logic
Exception at trap instruction
No
Yes
Infinite loop
Break
exit

If unimplemented instruction emulation is disabled, but the processor encounters an unimplemented instruction, the exception handler treats resulting exception as an other exception. Other exceptions are described in "Other Exceptions" on page 6–22.

Using the Unimplemented Instruction Handler

The unimplemented instruction handler defines an emulation routine for each of the implementation-dependent instructions. In this way, the full Nios II instruction set is always supported, even if a particular Nios II core does not implement all instructions in hardware.

To include the unimplemented instruction handler, turn on **Emulate multiply and divide instructions** on the **System properties** page of the Nios II IDE. The emulation routines are small (less than ¾ KBytes of memory), so it is usually safe to include them even when targeting a Nios II core that does not require them. If a Nios II core implements a particular instruction in hardware, its corresponding exception never occurs.

An exception routine must never execute an unimplemented instruction. The HAL exception handling system does not support nested software exceptions.

Software Trap Handling

If the cause of the software exception is not an unimplemented instruction, the HAL software exception handler checks for a `trap` instruction. The HAL is not designed to handle software traps. If it finds one, it executes a `break`.

If your software is compiled for release, the exception handler makes a distinction between traps and other exceptions. If your software is compiled for debug, traps and other exceptions are handled identically, by executing a `break` instruction. Figure 6–3 shows a flowchart of the HAL software exception handler, including the optional trap logic. If your software is compiled for debug, the trap logic is omitted.

In the Nios II IDE, you can select debug or release compilation in the **Project Properties** dialog box, under **C/C++ Build**.

Other Exceptions

If the exception is not caused by an unimplemented instruction or a trap, it is an other exception. In a debugging environment, the processor executes a `break`, allowing the debugger to take control. In a non-debugging environment, the processor goes into an infinite loop.

For details about the Nios II processor `break` instruction, see the *Programming Model* and *Instruction Set Reference* chapters of the *Nios II Processor Reference Handbook*.

Other exceptions can occur for these reasons:

- You need to include the unimplemented instruction handler, discussed in "Unimplemented Instructions" on page 6–19.

- A peripheral is generating spurious interrupts. This is a symptom of a serious hardware problem. A peripheral might generate spurious hardware interrupts if it deasserts its interrupt output before an ISR has explicitly serviced it.

Invalid Instructions

An invalid instruction word contains invalid codes in the OP or OPX field. For normal Nios II core implementations, the result of executing an invalid instruction is undefined; processor behavior is dependent on the Nios II core.

Therefore, the exception handler cannot detect or respond to an invalid instruction.

☞ Invalid instructions are different from unimplemented instructions, which are described in "Unimplemented Instructions" on page 6–19.

For more information, see the *Nios II Core Implementation Details* chapter of the *Nios II Processor Reference Handbook*.

HAL Exception Handler Files

The HAL exception handling code is in the following files:

- Source files:
  - alt_exception_entry.S
  - alt_exception_muldiv.S
  - alt_exception_trap.S
  - alt_irq_entry.S
  - alt_irq_handler.c
  - alt_software_exception.S
  - alt_irq_vars.c
  - alt_irq_register.c
- Header files:
  - alt_irq.h
  - alt_irq_entry.h

Document Revision History

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

Table 6–2. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
November 2006, v6.1.0	● Describes support for the interrupt vector custom instruction.	Interrupt vector custom instruction added.
May 2006, v6.0.0	● Corrected error in `alt_irq_enable_all()` usage ● Added illustrations ● Revised text on optimizing ISRs ● Expanded and revised text discussing HAL exception handler code structure.	
October 2005, v5.1.0	● Updated references to HAL exception-handler assembly source files in section "HAL Exception Handler Files". ● Added description of `alt_irq_disable()` and `alt_irq_enable()` in section "ISRs".	
May 2005, v5.0.0	Added tightly-coupled memory information.	
December 2004 v1.2	Corrected the "Registering the Button PIO ISR with the HAL" example.	
September 2004 v1.1	● Changed examples. ● Added ISR performance data.	
May 2004 v1.0	First publication.	

7. Cache & Tightly-Coupled Memory

NII52007-6.1.0

Introduction

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

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

This chapter discusses the following common cases when you need to manage the cache:

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

Nios II Cache Implementation

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

In all current Nios II cores, there is no hardware cache coherency mechanism. Therefore, if there are multiple masters accessing shared memory, software must explicitly maintain coherency across all masters.

For complete details on the features of each Nios II core implementation, see the *Nios II Core Implementation Details* chapter of the *Nios II Processor Reference Handbook*.

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

Example: An excerpt from system.h that defines the Cache Structure

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

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

HAL API Functions for Managing Cache

The HAL API provides the following functions for managing cache memory.:

- `alt_dcache_flush()`
- `alt_dcache_flush_all()`
- `alt_icache_flush()`
- `alt_icache_flush_all()`
- `alt_uncached_malloc()`
- `alt_uncached_free()`
- `alt_remap_uncached()`
- `alt_remap_cached()`

For details on API functions, see the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

Further Information

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

Initializing Cache after Reset

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

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

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

Place the initi instruction in a loop that executes `initi` for each instruction cache line address. The following code shows an example of assembly code to initialize the instruction cache.

Example: Assembly code to initialize the instruction cache

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

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

Place the `initd` instruction in a loop that executes `initd` for each data cache line address. The following code shows an example of assembly code to initialize the data cache:

Example: Assembly code to initialize the data cache

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

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

For HAL System Library Users

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

Writing Device Drivers

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

For device drivers, the data cache should be bypassed by using the `ldio`/`stio` family of instructions. On Nios II cores without a data cache, these instructions behave just like their corresponding `ld`/`st` instructions, and therefore are benign.

For C programmers, note that declaring a pointer as `volatile` does not cause accesses using that volatile pointer to bypass the data cache. The `volatile` keyword only prevents the compiler from optimizing out accesses using the pointer.

☞ This `volatile` behavior is different from the methodology for the first-generation Nios processor.

For HAL System Library Users

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

Table 7–1 shows the available macros. All of these macros bypass the data cache when they perform their operation. In general, your program passes values defined in **system.h** as the `BASE` and `REGNUM` parameters. These macros are defined in the file <*Nios II EDS install path*>**/components/altera_nios2/HAL/inc/io.h**.

Table 7–1. HAL I/O Macros to Bypass the Data Cache

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

Writing Program Loaders or Self-Modifying Code

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

The following code shows assembly code that writes a new instruction to memory.

Example: Assembly Code That Writes a New Instruction to Memory

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

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

Notice that the above code sequence used the `stw`/`flushd` pair instead of the `stwio` instruction. Using a `stwio` instruction doesn't flush the data cache so could leave stale data in the data cache.

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

For Users of the HAL System Library

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

Managing Cache in Multi-Master/Multi-CPU Systems

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

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

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

Bit-31 Cache Bypass

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

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

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

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

For HAL System Library Users

The HAL provides the C-language `IORD_*DIRECT` macros that expand to the `ldio` family of instructions and the `IOWR_*DIRECT` macros that expand to the `stio` family of instructions. See Table 7–1. These macros are provided to access non-cacheable memory regions.

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

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

Tightly-Coupled Memory

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

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

For more information, refer to the "Assigning Code & Data to Memory Partitions" section of the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*.

Document Revision History

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

Table 7–2. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
November 2006, v6.1.0	No change from previous release.	
May 2006, v6.0.0	No change from previous release.	
October 2005, v5.1.0	Added detail to section "Tightly-Coupled Memory".	
May 2005, v5.0.0	Added tightly-coupled memory section.	
May 2004 v1.0	First publication.	

8. MicroC/OS-II Real-Time Operating System

NII52008-6.1.0

Introduction

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

Overview

MicroC/OS-II is a popular real-time kernel produced by Micrium Inc., and is documented in the book *MicroC/OS-II - The Real Time Kernel* by Jean J. Labrosse (CMP Books). The book describes MicroC/OS-II as a portable, ROMable, scalable, preemptive, real-time, multitasking kernel. MicroC/OS-II has been used in hundreds of commercial applications since its release in 1992, and has been ported to over 40 different processor architectures in addition to the Nios II processor. MicroC/OS-II provides the following services:

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

The MicroC/OS-II kernel operates on top of the hardware abstraction layer (HAL) system library for the Nios II processor. Because of the HAL, programs based on MicroC/OS-II are more portable to other Nios II hardware systems, and are resistant to changes in the underlying hardware. Furthermore, MicroC/OS-II programs have access to all HAL services, and can call the familiar HAL advanced programming interface (API) functions.

Further Information

This chapter discusses the details of how to use MicroC/OS-II for the Nios II processor only. For complete reference of MicroC/OS-II features and usage, refer to *MicroC/OS-II - The Real-Time Kernel*. Further information is also available on the Micrium website, **www.micrium.com**.

Licensing

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

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

Other RTOS Providers

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

For a complete list of RTOSs that support the Nios II processor, visit the Nios II homepage at **www.altera.com/nios2**.

The Altera Port of MicroC/OS-II

Altera ported MicroC/OS-II to the Nios II processor. Altera distributes MicroC/OS-II in the Nios II EDS, and supports the Nios II port of the MicroC/OS-II kernel. Ready-made, working examples of MicroC/OS-II programs are installed with the Nios II EDS. In fact, Nios development boards are pre-programmed with a web server reference design based on MicroC/OS-II and the Lightweight IP TCP/IP stack.

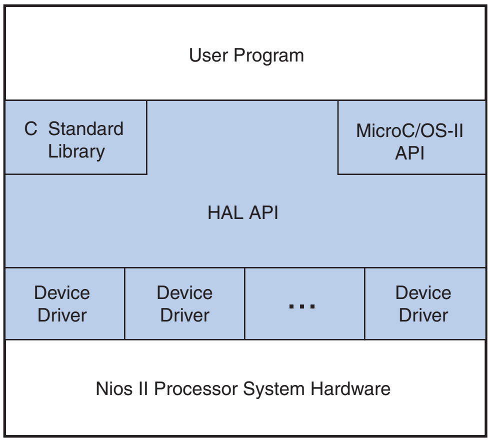
The Altera® port of MicroC/OS-II is designed to be easy-to-use from within the Nios II IDE. Using the Nios II IDE, you can control the configuration for all the RTOS's modules. You need not modify source files directly to enable or disable kernel features. Nonetheless, Altera provides the Nios II processor-specific source code if you ever wish to examine it. The code is provided in directory <*Nios II EDS install path*>**/components/altera_nios/UCOSII**. The processor-independent code resides in <*Nios II EDS install path*>**/components/micrium_uc_osii**. The MicroC/OS-II software component behaves like the drivers for SOPC Builder hardware components: When MicroC/OS-II is included in a Nios II integrated development environment (IDE) project, the header and source files from **components/micrium_uc_osii** are included in the project path, causing the MicroC/OS-II kernel to compile and link into the project.

MicroC/OS-II Architecture

The Altera port of MicroC/OS-II for the Nios II processor is essentially a superset of the HAL. It is the HAL environment extended by the inclusion of the MicroC/OS-II scheduler and the associated MicroC/OS-II API. The complete HAL API is available from within MicroC/OS-II projects.

Figure 8–1 shows the architecture of a program based on MicroC/OS-II and the relationship to the HAL.

Figure 8–1. Architecture of MicroC/OS-II Programs



The multi-threaded environment affects certain HAL functions.

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

MicroC/OS-II Thread-Aware Debugging

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

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

MicroC/OS-II Device Drivers

Each peripheral (i.e., an SOPC Builder component) can provide include files and source files within the **inc** and **src** subdirectories of the component's **HAL** directory.

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

In addition to the **HAL** directory, a component may elect to provide a **UCOSII** directory that contains code specific to the MicroC/OS-II environment. Similar to the **HAL** directory, the **UCOSII** directory contains **inc** and **src** subdirectories. These directories are automatically added to the source and include search paths when building MicroC/OS-II projects in the Nios II IDE.

You can use the **UCOSII** directory to provide code that is used only in a multi-threaded environment. Other than these additional search directories, the mechanism for providing MicroC/OS-II device drivers is identical to the process described in the *Developing Device Drivers for the HAL* chapter of the *Nios II Software Developer's Handbook*.

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

Thread-Safe HAL Drivers

To allow the same driver to be portable across the HAL and MicroC/OS-II environments, Altera defines a set of OS-independent macros that provide access to operating system facilities. When compiled for a MicroC/OS-II project, the macros expand to a MicroC/OS-II API call. When compiled for a single-threaded HAL project, the macros expand to benign empty implementations. These macros are used in Altera-provided device driver code, and you can use them if you need to write a device drivers with similar portability.

Table 8–1 lists the available macros and their function.

For more information on the functionality in the MicroC/OS-II environment, see *MicroC/OS-II – The Real-Time Kernel*.

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

Table 8–1. OS-Independent Macros for Thread-Safe HAL Drivers (Part 1 of 2)

Macro	Defined in Header	MicroC/OS-II Implementation	Single-Threaded HAL Implementation
`ALT_FLAG_GRP(group)`	**os/alt_flag.h**	Create a pointer to a flag group with the name `group`.	Empty statement.
`ALT_EXTERN_FLAG_GRP(group)`	**os/alt_flag.h**	Create an external reference to a pointer to a flag group with name `group`.	Empty statement.
`ALT_STATIC_FLAG_GRP(group)`	**os/alt_flag.h**	Create a static pointer to a flag group with the name `group`.	Empty statement.
`ALT_FLAG_CREATE(group, flags)`	**os/alt_flag.h**	Call `OSFlagCreate()` to initialize the flag group pointer, `group`, with the flags value `flags`. The error code is the return value of the macro.	Return 0 (success).
`ALT_FLAG_PEND(group, flags, wait_type, timeout)`	**os/alt_flag.h**	Call `OSFlagPend()` with the first four input arguments set to `group`, `flags`, `wait_type`, and `timeout` respectively. The error code is the return value of the macro.	Return 0 (success).
`ALT_FLAG_POST(group, flags, opt)`	**os/alt_flag.h**	Call `OSFlagPost()` with the first three input arguments set to `group`, `flags`, and `opt` respectively. The error code is the return value of the macro.	Return 0 (success).
`ALT_SEM(sem)`	**os/alt_sem.h**	Create an OS_EVENT pointer with the name `sem`.	Empty statement.
`ALT_EXTERN_SEM(sem)`	**os/alt_sem.h**	Create an external reference to an `OS_EVENT` pointer with the name `sem`.	Empty statement.
`ALT_STATIC_SEM(sem)`	**os/alt_sem.h**	Create a static `OS_EVENT` pointer with the name `sem`.	Empty statement.

Table 8–1. OS-Independent Macros for Thread-Safe HAL Drivers (Part 2 of 2)

Macro	Defined in Header	MicroC/OS-II Implementation	Single-Threaded HAL Implementation
`ALT_SEM_CREATE(sem, value)`	**os/alt_sem.h**	Call `OSSemCreate()` with the argument `value` to initialize the `OS_EVENT` pointer `sem`. The return value is zero upon success, or negative otherwise.	Return 0 (success).
`ALT_SEM_PEND(sem, timeout)`	**os/alt_sem.h**	Call `OSSemPend()` with the first two argument set to `sem` and `timeout` respectively. The error code is the return value of the macro.	Return 0 (success).
`ALT_SEM_POST(sem)`	**os/alt_sem.h**	Call `OSSemPost()` with the input argument `sem`.	Return 0 (success).

The Newlib ANSI C Standard Library

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

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

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

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

Implementing MicroC/OS-II Projects in the Nios II IDE

To create a program based on MicroC/OS-II, you must first set the properties for the system library to a MicroC/OS-II project. From there, the Nios II IDE offers RTOS options that allow you to control the configuration of the MicroC/OS-II kernel.

Traditionally, you had to configure MicroC/OS-II using `#define` directives in the file **OS_CFG.h**. Instead, the Nios II IDE provides a GUI that allows you to configure each option. Therefore, you do not need to edit header files or source code to configure the MicroC/OS-II features. The GUI settings are reflected in the system library's **system.h** file; **OS_CFG.h** simply includes **system.h**.

The following sections define the MicroC/OS-II settings available from the Nios II IDE. The meaning of each setting is defined fully in the *MicroC/OS-II – The Real-Timer Kernel* chapter of the *MicroC/OS-II Configuration Manual*.

For step-by-step instructions on how to create a MicroC/OS-II project in the Nios II IDE, refer to *Using the MicroC/OS-II RTOS with the Nios II Processor Tutorial*.

MicroC/OS-II General Options

Table 8–2 shows the general options.

Table 8–2. General Options (Part 1 of 2)

Option	Description
Maximum number of tasks	Maps onto the `#define OS_MAX_TASKS`. Must be at least 2
Lowest assignable priority	Maps on the `#define OS_LOWEST_PRIO`. Maximum allowable value is 63.
Enable code generation for event flags	Maps onto the `#define OS_FLAG_EN`. When disabled, event flag settings are also disabled. See "Event Flags Settings" on page 8–8.
Enable code generation for mutex semaphores	Maps onto the #define `OS_MUTEX_EN`. When disabled, mutual exclusion semaphore settings are also disabled. See "Mutex Settings" on page 8–8
Enable code generation for semaphores	Maps onto the `#define OS_SEM_EN`. When disabled, semaphore settings are also disabled. See "Semaphores Settings" on page 8–9.
Enable code generation for mailboxes	Maps onto the `#define OS_MBOX_EN`. When disabled, mailbox settings are also disabled. See "Mailboxes Settings" on page 8–9.

Table 8–2. General Options (Part 2 of 2)

Option	Description
Enable code generation for queues	Maps onto the `#define OS_Q_EN`. When disabled, queue settings are also disabled. See "Queues Settings" on page 8–9.
Enable code generation for memory management	Maps onto the `#define OS_MEM_EN`. When disabled, memory management settings are also disabled. See "Memory Management Settings" on page 8–10.

Event Flags Settings

Table 8–3 shows the event flag settings.

Table 8–3. Event Flags Settings

Setting	Description
Include code for wait on clear event flags	Maps on `#define OS_FLAG_WAIT_CLR_EN`.
Include code for `OSFlagAccept()`	Maps on `#define OS_FLAG_ACCEPT_EN`.
Include code for `OSFlagDel()`	Maps on `#define OS_FLAG_DEL_EN`.
Include code for `OSFlagQuery()`	Maps onto the `#define OS_FLAG_QUERY_EN`.
Maximum number of event flag groups	Maps onto the `#define OS_MAX_FLAGS`.
Size of name of event flags group	Maps onto the `#define OS_FLAG_NAME_SIZE`.

Mutex Settings

Table 8–4 shows the mutex settings.

Table 8–4. Mutex Settings

Setting	Description
Include code for `OSMutexAccept()`	Maps onto the `#define OS_MUTEX_ACCEPT_EN`.
Include code for `OSMutexDel()`	Maps onto the `#define OS_MUTEX_DEL_EN`.
Include code for `OSMutexQuery()`	Maps onto the `#define OS_MUTEX_QUERY_EN`.

Semaphores Settings

Table 8–5 shows the semaphores settings.

Table 8–5. Semaphores Settings

Setting	Description
Include code for `OSSemAccept()`	Maps onto the `#define OS_SEM_ACCEPT_EN`.
Include code for `OSSemSet()`	Maps onto the `#define OS_SEM_SET_EN`.
Include code for `OSSemDel()`	Maps onto the `#define OS_SEM_DEL_EN`.
Include code for `OSSemQuery()`	Maps onto the `#define OS_SEM_QUERY_EN`.

Mailboxes Settings

Table 8–6 shows the mailbox settings.

Table 8–6. Mailboxes Settings

Setting	Description
Include code for `OSMboxAccept()`	Maps onto `#define OS_MBOX_ACCEPT_EN`.
Include code for `OSMBoxDel()`	Maps onto `#define OS_MBOX_DEL_EN`.
Include code for `OSMboxPost()`	Maps onto `#define OS_MBOX_POST_EN`.
Include code for `OSMboxPostOpt()`	Maps onto `#define OS_MBOX_POST_OPT_EN`.
Include code fro `OSMBoxQuery()`	Maps onto `#define OS_MBOX_QUERY_EN`.

Queues Settings

Table 8–7 shows the queues settings.

Table 8–7. Queues Settings (Part 1 of 2)

Setting	Description
Include code for `OSQAccept()`	Maps onto `#define OS_Q_ACCEPT_EN`.
Include code for `OSQDel()`	Maps onto `#define OS_Q_DEL_EN`.
Include code for `OSQFlush()`	Maps onto `#define OS_Q_FLUSH_EN`.
Include code for `OSQPost()`	Maps onto `#define OS_Q_POST_EN`.
Include code for `OSQPostFront()`	Maps onto `#define OS_Q_POST_FRONT_EN`.
Include code for `OSQPostOpt()`	Maps onto `#define OS_Q_POST_OPT_EN`.

Table 8–7. Queues Settings (Part 2 of 2)

Setting	Description
Include code for `OSQQuery()`	Maps onto `#define OS_Q_QUERY_EN`.
Maximum number of Queue Control blocks	Maps onto `#define OS_MAX_QS`.

Memory Management Settings

Table 8–8 shows the memory management settings.

Table 8–8. Memory Management Settings

Setting	Description
Include code for `OSMemQuery()`	Maps onto `#define OS_MEM_QUERY_EN`.
Maximum number of memory partitions	Maps onto `#define OS_MAX_MEM_PART`.
Size of memory partition name	Maps onto `#define OS_MEM_NAME_SIZE`.

Miscellaneous Settings

Table 8–9 shows the miscellaneous settings.

Table 8–9. Miscellaneous Settings (Part 1 of 2)

Setting	Description
Enable argument checking	Maps onto `#define OS_ARG_CHK_EN`.
Enable `uCOS-II` hooks	Maps onto `#define OS_CPU_HOOKS_EN`.
Enable debug variables	Maps onto `#define OS_DEBUG_EN`.
Include code for `OSSchedLock()` and `OSSchedUnlock()`	Maps onto `#define OS_SCHED_LOCK_EN`.
Enable tick stepping feature for `uCOS-View`	Maps onto `#define OS_TICK_STEP_EN`.
Enable statistics task	Maps onto `#define OS_TASK_STAT_EN`.
Check task stacks from statistics task	Maps onto `#define OS_TASK_STAT_STK_CHK_EN`.
Statistics task stack size	Maps onto `#define OS_TASK_STAT_STK_SIZE`.
Idle task stack size	Maps onto `#define OS_TASK_IDLE_STK_SIZE`.

Table 8–9. Miscellaneous Settings (Part 2 of 2)

Setting	Description
Maximum number of event control blocks	Maps onto `#define OS_MAX_EVENTS 60`.
Size of semaphore, mutex, mailbox, or queue name	Maps onto `#define OS_EVENT_NAME_SIZE`.

Task Management Settings

Table 8–10 shows the task management settings.

Table 8–10. Task Management Settings

Setting	Description
Include code for `OSTaskChangePrio()`	Maps onto `#define OS_TASK_CHANGE_PRIO_EN`.
Include code for `OSTaskCreate()`	Maps onto `#define OS_TASK_CREATE_EN`.
Include code for `OSTaskCreateExt()`	Maps onto `#define OS_TASK_CREATE_EXT_EN`.
Include code for `OSTaskDel()`	Maps onto `#define OS_TASK_DEL_EN`.
Include variables in `OS_TCB` for profiling	Maps onto `#define OS_TASK_PROFILE_EN`.
Include code for `OSTaskQuery()`	Maps onto `#define OS_TASK_QUERY_EN`.
Include code for `OSTaskSuspend()` and `OSTaskResume()`	Maps onto `#define OS_TASK_SUSPEND_EN`.
Include code for `OSTaskSwHook()`	Maps onto `#define OS_TASK_SW_HOOK_EN`.
Size of task name	Maps onto `#define OS_TASK_NAME_SIZE`.

Time Management Settings

Table 8–11 shows the time management settings.

Table 8–11. Time Management Settings (Part 1 of 2)

Setting	Description
Include code for `OSTimeDlyHMSM()`	Maps onto `#define OS_TIME_DLY_HMSM_EN`.
Include code `OSTimeDlyResume()`	Maps onto `#define OS_TIME_DLY_RESUME_EN`.

Table 8–11. Time Management Settings (Part 2 of 2)

Setting	Description
Include code for `OSTimeGet()` and `OSTimeSet()`	Maps onto `#define OS_TIME_GET_SET_EN`.
Include code for `OSTimeTickHook()`	Maps onto `#define OS_TIME_TICK_HOOK_EN`.

Document Revision History

Table 8–12 shows the revision history for this document.

Table 8–12. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
November 2006, v6.1.0	No change from previous release.	
May 2006, v6.0.0	No change from previous release.	
October 2005, v5.1.0	No change from previous release.	
May 2005, v5.0.0	No change from previous release.	
December 2004 v1.1	Added thread-aware debugging paragraph.	
May 2004 v1.0	First publication.	

9. Ethernet & the NicheStack® TCP/IP Stack - Nios II Edition

NII52013-6.1.0

Overview

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

Altera® provides the NicheStack TCP/IP Stack as a software component plug-in for the Nios II Integrated Development Environment (IDE), which you can add to your system library. The NicheStack TCP/IP Stack includes these features:

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

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

Prerequisites

To make the best use of information in this chapter, you need have basic familiarity with these topics:

- Sockets. There are a number of books on the topic of programming with sockets. Two good texts are *Unix Network Programming* by Richard Stevens and *Internetworking with TCP/IP Volume 3* by Douglas Comer.
- The Nios II Embedded Design Suite (EDS). Refer to the Nios II Software Development Tutorial, which is available in the Nios II IDE help system.
- The MicroC/OS-II real time operating system (RTOS). To learn about MicroC/OS-II, refer to the *Using MicroC/OS-II RTOS with the Nios II Processor Tutorial*.

Introduction

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

The Nios II EDS includes several working examples of programs using the NicheStack TCP/IP Stack for your reference. In fact, Nios II development boards are preprogrammed with a web server reference design based on the NicheStack TCP/IP Stack and the MicroC/OS-II RTOS. Full source code is provided.

The NicheStack TCP/IP Stack uses the MicroC/OS-II RTOS multithreaded environment. Therefore, to use the NicheStack TCP/IP Stack, you must base your C/C++ project on the MicroC/OS-II RTOS. Naturally, the Nios II processor system must also contain an Ethernet interface, or media access controller (MAC). The Altera-provided NicheStack TCP/IP Stack includes driver support for the SMSC lan91c111 MAC/PHY device. The Nios II Embedded Design Suite includes hardware for the SMSC lan91c111. The NicheStack TCP/IP Stack driver is interrupt-based, so you must ensure that interrupts for the Ethernet component are connected.

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

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

The NicheStack TCP/IP Stack Files & Directories

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

The directory format of the stack tries to maintain the original code as much as possible under the <*iniche path*>**/src/downloads** directory for ease of upgrading to more recent versions of the NicheStack TCP/IP Stack. The <*iniche path*>**/src/downloads/packages** directory contains the original NicheStack TCP/IP Stack source code and documentation; the <*iniche path*>**/src/downloads/30src** directory contains code specific to the Nios II implementation of the NicheStack TCP/IP Stack, including source code supporting MicroC/OS-II.

The reference manual for the NicheStack TCP/IP Stack is installed with the Nios II EDS, in the <*iniche path*>**/src/downloads/packages/** directory. The reference manual is in **NicheStackRef.zip**.

Altera's implementation of the NicheStack TCP/IP Stack is based on version 3.0 of the protocol stack, with wrappers placed around the code to integrate it to the HAL system library.

Licensing

The NicheStack TCP/IP Stack is a TCP/IP protocol stack created by InterNiche Technologies, Inc. You can license the NicheStack TCP/IP Stack from Altera by going to *www.altera.com/nichestack*.

You can license other protocol stacks directly from InterNiche. Refer to the InterNiche website, *http://www.interniche.com*, for details.

Other TCP/IP Stack Providers

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

For up-to-date information on products available from third party providers, visit Altera's Embedded Software Partners page at *http://www.altera.com/products/ip/processors/nios2/tools/embed-partners/ni2-embed-partners.html*.

Using the NicheStack TCP/IP Stack

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

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

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

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

You must provide the following simple functions that are called by HAL system code to obtain the MAC address and IP address:

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

Nios II System Requirements

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

- The system hardware generated in SOPC Builder must include an Ethernet interface with interrupts enabled
- The system library must be based on MicroC/OS-II

The NicheStack TCP/IP Stack Tasks

The NicheStack TCP/IP Stack, in its standard Nios II configuration, consists of two fundamental tasks. These tasks run continuously in addition to the tasks that your program creates.

1. The NicheStack main task — After initialization, this task sleeps until a new packet is available for processing. Packets are received by an interrupt service routine (ISR). When the ISR receives a packet, it places it in the receive queue, and wakes up the main task.

2. The NicheStack tick task — This task wakes up periodically to monitor for time-out conditions.

These tasks are started when the initialization process succeeds in the `netmain()` function, as described in "netmain()" on page 9–5.

☞ Additional system tasks might be created if you enable other options in the NicheStack TCP/IP Stack by editing `ipport.h`.

Initializing the Stack

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

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

Make sure that your code does not use the sockets interface until `iniche_net_ready` is set to `true`. For example, call `alt_iniche_init()` and `netmain()` from the highest priority task, and wait for `iniche_net_ready` before allowing other tasks to run, as shown in "Example: Instantiating the NicheStack TCP/IP Stack".

alt_iniche_init()

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

```
void alt_iniche_init(void)
```

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

netmain()

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

```
void netmain(void)
```

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

iniche_net_ready

When the NicheStack stack has completed initialization, it sets the global variable `iniche_net_ready` to TRUE.

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

The following code shows an example of using `iniche_net_ready` to wait until the network stack has completed initialization:

Example: Instantiating the NicheStack TCP/IP Stack

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

  alt_iniche_init();
  netmain();

  while (!iniche_net_ready)
    TK_SLEEP(1);

  /* Now that the stack is running, perform the application
```

```
    initialization steps */

   .
   .
   .

}
```

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

get_mac_addr() & get_ip_addr()

The NicheStack TCP/IP Stack system code calls `get_mac_addr()` and `get_ip_addr()` during the device initialization process. These functions are necessary for the system code to set the MAC and IP addresses for the network interface, which you select through **MAC interface** in the **NicheStack TCP/IP Stack** tab of the **Software Components** dialog box. Because you write these functions yourself, your system has the flexibility to store the MAC address and IP address in an arbitrary location, rather than a fixed location hard coded in the device driver. For example, some systems might store the MAC address in flash memory, while others might have the MAC address in onchip embedded memory.

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

The prototype for `get_mac_addr()` is:

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

Inside the function, you must fill in `mac_addr` with the MAC address.

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

The following code shows an example implementation of `get_mac_addr()`. For demonstration purposes only, the MAC address is stored at address `CUSTOM_MAC_ADDR` in this example. There is no error checking in this example. In a real application, if there is an error, `get_mac_addr()` returns -1.

Example: An implementation of get_mac_addr()

```
#include <alt_iniche_dev.h>
```

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

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

  return ret_code;
}
```

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

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

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

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

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

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

To enable DHCP, include the line:

```
*use_dhcp = 1;
```

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

To assign a static IP address, include the lines:

```
*use_dhcp = 0;
```

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

The following code shows an example implementation of `get_ip_addr()` and shows a list of the necessary include files.

There is no error checking in this example. In a real application, you might need to return -1 on error.

Example: An implementation of get_ip_addr()

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

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

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

`INICHE_DEFAULT_IF`, defined in system.h, identifies the network interface. In the Nios II IDE, you can set `INICHE_DEFAULT_IF` through the **MAC interface** control in the **NicheStack TCP/IP Stack** tab of the **Software Components** dialog box.

The flag `DHCP_CLIENT`, also defined in system.h, specifies whether to use DHCP. You can set or clear this flag in the Nios II IDE, with the **Use DHCP to automatically assign IP address** check box.

Calling the Sockets Interface

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

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

The prototype for `TK_NEWTASK()` is:

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

It is in <*iniche path*>**/src/downloads/30src/nios2/osport.h**. You can include this header file as follows:

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

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

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

Configuring the NicheStack TCP/IP Stack in the Nios II IDE

The NicheStack TCP/IP Stack has many options that you can configure using `#define` directives in the file **ipport.h**. The Nios II integrated development environment (IDE) allows you to configure certain options (i.e. modify the `#defines` in **system.h**) without editing source code. The most commonly accessed options are available through the **NicheStack TCP/IP Stack** tab of the **Software Components** dialog box.

There are some less frequently used options that are not accessible through the GUI. If you need to modify these options, you must edit the **ipport.h** file manually. You can find **ipport.h** in the **debug/ system_description** directory for your system library project.

The following sections describe the features that you can configure via the Nios II IDE. The GUI provides a default value for each feature. In general, these values provide a good starting point, and you can later fine tune the values to meet the needs of your system.

NicheStack TCP/IP Stack General Settings

The ARP, UDP and IP protocols are always enabled. Table 9–1 shows the protocol options.

Table 9–1. Protocol Options

Option	Description
TCP	Enables and disables the transmission control protocol (TCP).

Table 9–2 shows the global options, which affect the overall behavior of the TCP/IP stack.

Table 9–2. Global Options

Option	Description
Use DHCP to automatically assign IP address	When on, the component use DHCP to acquire an IP address. When off, you must assign a static IP address.
Enable statistics	When this option is turned on, the stack keeps counters of packets received, errors, etc. The counters are defined in `mib` structures defined in various header files in directory *<iniche path>***/src/downloads/30src/h.** For details on `mib` structures, refer to the NicheStack documentation.
MAC interface	If the IP stack has more than one network interface, this parameter indicates which interface to use. See "Known Limitations" on page 9–11.

IP Options

Table 9–4 shows the IP options.

Table 9–3. IP Options

Option	Description
Forward IP packets	When there is more than one network interface, if this option is turned on, and the IP stack for one interface receives packets not addressed to it, it forwards the packet out of the other interface. See "Known Limitations" on page 9–11.
Reassemble IP packet fragments	If this option is turned on, the NicheStack TCP/IP Stack reassembles IP packet fragments into full IP packets. Otherwise, it discards IP packet fragments. This topic is explained in *Unix Network Programming* by Richard Stevens.

TCP Options

Table 9–4 shows the TCP options, which are only available with the TCP option is turned on.

Table 9–4. TCP Options

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

For further details, see the NicheStack TCP/IP Stack reference manual.

Further Information

For further information, refer to the *Using the NicheStack® TCP/IP Stack - Nios II Edition Tutorial*. The tutorial provides in-depth information about the NicheStack TCP/IP Stack, and illustrates how to use it in a networking application.

Known Limitations

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

Document Revision History

Table 9–5 shows the revision history for this document.

Table 9–5. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
November 2006, v6.1.0	First publication.	

Section IV. Appendices

This section provides appendix information.

This section includes the following chapters:

10. HAL API Reference

NII52010-6.1.0

Introduction

This chapter provides an alphabetically ordered list of all the functions within the hardware abstraction layer (HAL) application programming interface (API). Each function is listed with its C prototype and a short description. Indication is also given as to whether the function is thread safe when running in a multi-threaded environment, and whether it can be called from an interrupt service routine (ISR).

This appendix only lists the functionality provided by the HAL. You should be aware that the complete newlib API is also available from within HAL systems. For example, newlib provides `printf()`, and other standard I/O functions, which are not described here.

For more details of the newlib API, refer to the newlib documentation. On the Windows **Start** menu, click **Programs**, **Altera**, **Nios II** <*version*>, **Nios II Documentation**.

_exit()

Prototype:	`void _exit (int exit_code)`
Commonly called by:	Newlib C library
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**unistd.h**>
Description:	The newlib `exit()` function calls the `_exit()` function to terminate the current process. Typically, when `main()` completes. Because there is only a single process within HAL systems, the HAL implementation blocks forever. Note that interrupts are not disabled, so ISRs continue to execute. The input argument, `exit_code`, is ignored.
Return:	–
See also:	Newlib documentation. On the Windows **Start** menu, click **Programs**, **Altera**, **Nios II** *<version>*, **Nios II Documentation**.

_rename()

Prototype:	`int _rename(char *existing, char* new)`
Commonly called by:	Newlib C library
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	<**stdio.h**>
Description:	The `_rename()` function is provided for newlib compatibility.
Return:	It always fails with return code –1, and with `errno` set to `ENOSYS`.
See also:	Newlib documentation. On the Windows **Start** menu, click **Programs**, **Altera**, **Nios II** <*version*>, **Nios II Documentation**.

alt_alarm_start()

Prototype:

```
int alt_alarm_start (alt_alarm* alarm,
                      alt_u32    nticks,
                     alt_u32 (*callback) (void* context),
                      void*      context)
```

Commonly called by: C/C++ programs
Device drivers

Thread-safe: Yes.

Available from ISR: Yes.

Include: <**sys/alt_alarm.h**>

Description: The `alt_alarm_start()` function schedules an alarm callback. See the "Alarms" section of the *Developing Programs using the HAL* chapter of the *Nios® II Software Developer's Handbook*. The input argument, `ntick`, is the number of system clock ticks that elapse until the call to the `callback` function. The input argument `context` is passed as the input argument to the `callback` function, when the callback occurs.

The input `alarm` is a pointer to a structure that represents this alarm. You must create it, and it must have a lifetime that is at least as long as that of the alarm. However, you are not responsible for initializing the contents of the structure pointed to by `alarm`. This action is done by the call to `alt_alarm_start()`.

Return: The return value for `alt_alarm_start()` is zero upon success, and negative otherwise. This function fails if there is no system clock available.

See also:
`alt_alarm_stop()`
`alt_nticks()`
`alt_sysclk_init()`
`alt_tick()`
`alt_ticks_per_second()`
`gettimeofday()`
`settimeofday()`
`times()`
`usleep()`

alt_alarm_stop()

Prototype:	`void alt_alarm_stop (alt_alarm* alarm)`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	<**sys/alt_alarm.h**>
Description:	You can call the `alt_alarm_stop()` function to cancel an alarm previously registered by a call to `alt_alarm_start()`. The input argument is a pointer to the alarm structure in the previous call to `alt_alarm_start()`. Upon return the alarm is canceled, if it is still active.
Return:	–

See also:	`alt_alarm_start()`
	`alt_nticks()`
	`alt_sysclk_init()`
	`alt_tick()`
	`alt_ticks_per_second()`
	`gettimeofday()`
	`settimeofday()`
	`times()`
	`usleep()`

alt_dcache_flush()

Prototype:	`void alt_dcache_flush (void* start, alt_u32 len)`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	<**sys/alt_cache.h**>
Description:	The `alt_dcache_flush()` function flushes (i.e. writes back dirty data and then invalidates) the data cache for a memory region of length `len` bytes, starting at address `start`. In processors without data caches, it has no effect.
Return:	–

See also:	`alt_dcache_flush_all()`
	`alt_icache_flush()`
	`alt_icache_flush_all()`
	`alt_remap_cached()`
	`alt_remap_uncached()`
	`alt_uncached_free()`
	`alt_uncached_malloc()`

alt_dcache_flush_all()

Prototype:	`void alt_dcache_flush_all (void)`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	<**sys/alt_cache.h**>
Description:	The `alt_dcache_flush_all()` function flushes, i.e., writes back dirty data and then invalidates, the entire contents of the data cache. In processors without data caches, it has no effect.
Return:	–

See also:	`alt_dcache_flush()`
	`alt_icache_flush()`
	`alt_icache_flush_all()`
	`alt_remap_cached()`
	`alt_remap_uncached()`
	`alt_uncached_free()`
	`alt_uncached_malloc()`

alt_dev_reg()

Prototype:	`int alt_dev_reg(alt_dev* dev)`
Commonly called by:	Device drivers
Thread-safe:	No.
Available from ISR:	No.
Include:	<**sys/alt_dev.h**>
Description:	The `alt_dev_reg()` function registers a device with the system. Once registered you can access a device using the standard I/O functions. See the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*. The system behavior is undefined in the event that a device is registered with a name that conflicts with an existing device or file system. The `alt_dev_reg()` function is not thread safe in the sense that there should be no other thread using the device list at the time that `alt_dev_reg()` is called. In practice `alt_dev_reg()` should only be called while operating in a single threaded mode. The expectation is that it is only called by the device initialization functions invoked by `alt_sys_init()`, which in turn should only be called by the single threaded C startup code.
Return:	A return value of zero indicates success. A negative return value indicates failure.
See also:	`alt_fs_reg()`

alt_dma_rxchan_close()

Prototype:	`int alt_dma_rxchan_close (alt_dma_rxchan rxchan)`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**sys/alt_dma.h**>
Description:	The `alt_dma_rxchan_close()` function notifies the system that the application has finished with the direct memory access (DMA) receive channel, `rxchan`. The current implementation always succeeds.
Return:	The return value is zero upon success and negative otherwise.

See also:	`alt_dma_rxchan_depth()`
	`alt_dma_rxchan_ioctl()`
	`alt_dma_rxchan_open()`
	`alt_dma_rxchan_prepare()`
	`alt_dma_rxchan_reg()`
	`alt_dma_txchan_close()`
	`alt_dma_txchan_ioctl()`
	`alt_dma_txchan_open()`
	`alt_dma_txchan_reg()`
	`alt_dma_txchan_send()`
	`alt_dma_txchan_space()`

alt_dma_rxchan_depth()

Prototype:	`alt_u32 alt_dma_rxchan_depth(alt_dma_rxchan dma)`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**sys/alt_dma.h**>
Description:	The `alt_dma_rxchan_depth()` function returns the maximum number of receive requests that can be posted to the specified DMA transmit channel, `dma`. Whether this function is thread-safe, or can be called from an ISR is dependent on the underlying device driver. In general it should be assumed this is not the case.
Return:	Returns the maximum number of receive requests that can be posted.

See also:	`alt_dma_rxchan_close()`
	`alt_dma_rxchan_ioctl()`
	`alt_dma_rxchan_open()`
	`alt_dma_rxchan_prepare()`
	`alt_dma_rxchan_reg()`
	`alt_dma_txchan_close()`
	`alt_dma_txchan_ioctl()`
	`alt_dma_txchan_open()`
	`alt_dma_txchan_reg()`
	`alt_dma_txchan_send()`
	`alt_dma_txchan_space()`

alt_dma_rxchan_ioctl()

Prototype:

```
int alt_dma_rxchan_ioctl (alt_dma_rxchan dma,
                          int            req,
                          void*          arg)
```

Commonly called by: C/C++ programs
Device drivers

Thread-safe: See description.

Available from ISR: See description.

Include: <**sys/alt_dma.h**>

Description: The `alt_dma_rxchan_ioctl()` function performs DMA I/O operations on the DMA receive channel, `dma`. The I/O operations are device specific. For example, some DMA drivers support options to control the width of the transfer operations. The input argument, `req`, is an enumeration of the requested operation; `arg` is an additional argument for the request. The interpretation of `arg` is request dependent.

Table 10–1 shows generic requests defined in <**sys/alt_dma.h**>, which a DMA device might support.

Whether a call to `alt_dma_rxchan_ioctl` is thread safe, or can be called from an ISR, is device dependent. In general it should be assumed it is not the case.

The `alt_dma_rxchan_ioctl()` function should not be called while DMA transfers are pending, otherwise unpredictable behavior might result.

For device-specific information about the Altera® DMA controller core, see the *DMA Controller Core with Avalon Interface* chapter in volume 5 of the *Quartus® II Handbook*.

Return: A negative return value indicates failure, otherwise the interpretation of the return value is request specific.

See also:

```
alt_dma_rxchan_close()
alt_dma_rxchan_depth()
alt_dma_rxchan_open()
alt_dma_rxchan_prepare()
alt_dma_rxchan_reg()
alt_dma_txchan_close()
alt_dma_txchan_ioctl()
alt_dma_txchan_open()
alt_dma_txchan_reg()
alt_dma_txchan_send()
alt_dma_txchan_space()
```

Table 10–1. Generic Requests

Request	Meaning
`ALT_DMA_SET_MODE_8`	Transfer data in units of 8 bits. The value of `arg` is ignored.
`ALT_DMA_SET_MODE_16`	Transfer data in units of 16 bits. The value of `arg` is ignored.
`ALT_DMA_SET_MODE_32`	Transfer data in units of 32 bits. The value of `arg` is ignored.
`ALT_DMA_SET_MODE_64`	Transfer data in units of 64 bits. The value of `arg` is ignored.
`ALT_DMA_SET_MODE_128`	Transfer data in units of 128 bits. The value of `arg` is ignored.
`ALT_DMA_GET_MODE`	Return the transfer width. The value of `arg` is ignored.
`ALT_DMA_TX_ONLY_ON` *(1)*	The `ALT_DMA_TX_ONLY_ON` request causes a DMA channel to operate in a mode where only the transmitter is under software control. The other side writes continuously from a single location. The address to write to is the argument to this request.
`ALT_DMA_TX_ONLY_OFF` *(1)*	Return to the default mode where both the receive and transmit sides of the DMA can be under software control.
`ALT_DMA_RX_ONLY_ON` *(1)*	The `ALT_DMA_RX_ONLY_ON` request causes a DMA channel to operate in a mode where only the receiver is under software control. The other side reads continuously from a single location. The address to read is the argument to this request.
`ALT_DMA_RX_ONLY_OFF` *(1)*	Return to the default mode where both the receive and transmit sides of the DMA can be under software control.

Notes to Table 10–1:

(1) These macro names changed in version 1.1 of the Nios II Embedded Design Suite (EDS). The old names (`ALT_DMA_TX_STREAM_ON`, `ALT_DMA_TX_STREAM_OFF`, `ALT_DMA_RX_STREAM_ON`, and `ALT_DMA_RX_STREAM_OFF`) are still valid, but new designs should use the new names.

alt_dma_rxchan_open()

Prototype:	`alt_dma_rxchan alt_dma_rxchan_open (const char* name)`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**sys/alt_dma.h**>
Description:	The `alt_dma_rxchan_open()` function obtains an `alt_dma_rxchan` descriptor for a DMA receive channel. The input argument, `name`, is the name of the associated physical device, e.g., `/dev/dma_0`.
Return:	The return value is null on failure and non-null otherwise. If there is an error, `errno` is set to `ENODEV`.

See also:	`alt_dma_rxchan_close()`
	`alt_dma_rxchan_depth()`
	`alt_dma_rxchan_ioctl()`
	`alt_dma_rxchan_prepare()`
	`alt_dma_rxchan_reg()`
	`alt_dma_txchan_close()`
	`alt_dma_txchan_ioctl()`
	`alt_dma_txchan_open()`
	`alt_dma_txchan_reg()`
	`alt_dma_txchan_send()`
	`alt_dma_txchan_space()`

alt_dma_rxchan_prepare()

Prototype:

```
int alt_dma_rxchan_prepare (alt_dma_rxchan   dma,
                            void*            data,
                            alt_u32          length,
                            alt_rxchan_done* done,
                            void*            handle)
```

Commonly called by: C/C++ programs
Device drivers

Thread-safe: See description.

Available from ISR: See description.

Include: <**sys/alt_dma.h**>

Description: The `alt_dma_rxchan_prepare()` posts a receive request to a DMA receive channel. The input arguments are: `dma`, the channel to use; `data`, a pointer to the location that data is to be received to; `length`, the maximum length of the data to receive in bytes; `done`, callback function that is called once the data is received; `handle`, an opaque value passed to `done`.

Whether this function is thread-safe, or can be called from an ISR is dependent on the underlying device driver. In general it should be assumed it is not the case.

Return: The return value is negative if the request cannot be posted, and zero otherwise.

See also:
`alt_dma_rxchan_close()`
`alt_dma_rxchan_depth()`
`alt_dma_rxchan_ioctl()`
`alt_dma_rxchan_open()`
`alt_dma_rxchan_reg()`
`alt_dma_txchan_close()`
`alt_dma_txchan_ioctl()`
`alt_dma_txchan_open()`
`alt_dma_txchan_reg()`
`alt_dma_txchan_send()`
`alt_dma_txchan_space()`

alt_dma_rxchan_reg()

Prototype:	`int alt_dma_rxchan_reg (alt_dma_rxchan_dev* dev)`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	No.
Available from ISR:	No.
Include:	**<sys/alt_dma_dev.h>**
Description:	The `alt_dma_rxchan_reg()` function registers a DMA receive channel with the system. Once registered a device can be accessed using the functions described in the "DMA Receive Channels" section of the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*. System behavior is undefined in the event that a channel is registered with a name that conflicts with an existing channel. The `alt_dma_rxchan_reg()` function is not thread safe if other threads are using the channel list at the time that `alt_dma_rxchan_reg()` is called. In practice, only call `alt_dma_rxchan_reg()` while operating in a single threaded mode. Only call it by the device initialization functions invoked by `alt_sys_init()`, which in turn should only be called by the single threaded C startup code.
Return:	A return value of zero indicates success. A negative return value indicates failure.

See also:	`alt_dma_rxchan_close()`
	`alt_dma_rxchan_depth()`
	`alt_dma_rxchan_ioctl()`
	`alt_dma_rxchan_open()`
	`alt_dma_rxchan_prepare()`
	`alt_dma_txchan_close()`
	`alt_dma_txchan_ioctl()`
	`alt_dma_txchan_open()`
	`alt_dma_txchan_reg()`
	`alt_dma_txchan_send()`
	`alt_dma_txchan_space()`

alt_dma_txchan_close()

Prototype:	`int alt_dma_txchan_close (alt_dma_txchan txchan)`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**sys/alt_dma.h**>
Description:	The `alt_dma_txchan_close` function notifies the system that the application has finished with the DMA transmit channel, `txchan`. The current implementation always succeeds.
Return:	The return value is zero upon success and negative otherwise.

See also:	`alt_dma_rxchan_close()`
	`alt_dma_rxchan_depth()`
	`alt_dma_rxchan_ioctl()`
	`alt_dma_rxchan_open()`
	`alt_dma_rxchan_prepare()`
	`alt_dma_rxchan_reg()`
	`alt_dma_txchan_ioctl()`
	`alt_dma_txchan_open()`
	`alt_dma_txchan_reg()`
	`alt_dma_txchan_send()`
	`alt_dma_txchan_space()`

alt_dma_txchan_ioctl()

Prototype:

```
int alt_dma_txchan_ioctl (alt_dma_txchan dma,
                          int            req,
                          void*          arg)
```

Commonly called by: C/C++ programs
Device drivers

Thread-safe: See description.

Available from ISR: See description.

Include: <**sys/alt_dma.h**>

Description: The `alt_dma_txchan_ioctl()` function performs device specific I/O operations on the DMA transmit channel, `dma`. For example, some drivers support options to control the width of the transfer operations. The input argument, `req`, is an enumeration of the requested operation; `arg` is an additional argument for the request. The interpretation of `arg` is request dependent.

See Table 10–1 for the generic requests a device might support.

Whether a call to `alt_dma_txchan_ioctl()` is thread safe, or can be called from an ISR, is device dependent. In general it should be assumed this is not the case.

The `alt_dma_rxchan_ioctl()` function should not be called while DMA transfers are pending, otherwise unpredictable behavior might result.

Return: A negative return value indicates failure; otherwise the interpretation of the return value is request specific.

See also:
`alt_dma_rxchan_close()`
`alt_dma_rxchan_depth()`
`alt_dma_rxchan_ioctl()`
`alt_dma_rxchan_open()`
`alt_dma_rxchan_prepare()`
`alt_dma_rxchan_reg()`
`alt_dma_txchan_close()`
`alt_dma_txchan_open()`
`alt_dma_txchan_reg()`
`alt_dma_txchan_send()`
`alt_dma_txchan_space()`

alt_dma_txchan_open()

Prototype:	`alt_dma_txchan alt_dma_txchan_open (const char* name)`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	Yes.
Available from ISR:	No.
Include:	**<sys/alt_dma.h>**
Description:	The `alt_dma_txchan_open()` function obtains an `alt_dma_txchan()` descriptor for a DMA transmit channel. The input argument, `name`, is the name of the associated physical device, e.g., `/dev/dma_0`.
Return:	The return value is null on failure and non-null otherwise. If there is an error, `errno` is set to `ENODEV`.

See also:	`alt_dma_rxchan_close()`
	`alt_dma_rxchan_depth()`
	`alt_dma_rxchan_ioctl()`
	`alt_dma_rxchan_open()`
	`alt_dma_rxchan_prepare()`
	`alt_dma_rxchan_reg()`
	`alt_dma_txchan_close()`
	`alt_dma_txchan_ioctl()`
	`alt_dma_txchan_reg()`
	`alt_dma_txchan_send()`
	`alt_dma_txchan_space()`

alt_dma_txchan_reg()

Prototype: `int alt_dma_txchan_reg (alt_dma_txchan_dev* dev)`

Commonly called by: C/C++ programs
Device drivers

Thread-safe: No.

Available from ISR: No.

Include: <**sys/alt_dma_dev.h**>

Description: The `alt_dma_txchan_reg()` function registers a DMA transmit channel with the system. Once registered, a device can be accessed using the functions described in the DMA Transmit Channels section of the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*.

System behavior is undefined in the event that a channel is registered with a name that conflicts with an existing channel.

The `alt_dma_txchan_reg()` function is not thread safe if other threads are using the channel list at the time that `alt_dma_txchan_reg()` is called. Only call `alt_dma_txchan_reg()` while operating in a single-threaded mode. Only call it by the device initialization functions invoked by `alt_sys_init()`, which in turn should only be called by the single threaded C startup code.

Return: A return value of zero indicates success. A negative return value indicates failure.

See also:

```
alt_dma_rxchan_close()
alt_dma_rxchan_depth()
alt_dma_rxchan_ioctl()
alt_dma_rxchan_open()
alt_dma_rxchan_prepare()
alt_dma_rxchan_reg()
alt_dma_txchan_close()
alt_dma_txchan_ioctl()
alt_dma_txchan_open()
alt_dma_txchan_send()
alt_dma_txchan_space()
```

alt_dma_txchan_send()

Prototype:

```
int alt_dma_txchan_send (alt_dma_txchan   dma,
                          const void*      from,
                          alt_u32          length,
                          alt_txchan_done* done,
                          void*            handle)
```

Commonly called by: C/C++ programs
Device drivers

Thread-safe: See description.

Available from ISR: See description.

Include: <**sys/alt_dma.h**>

Description: The `alt_dma_txchan_send()` function posts a transmit request to a DMA transmit channel. The input arguments are: `dma`, the channel to use; from, a pointer to the start of the data to send; `length`, the length of the data to send in bytes; `done`, a callback function that is called once the data is sent; and `handle`, an opaque value passed to done.

Whether this function is thread-safe, or can be called from an ISR is dependent on the underlying device driver. In general it should be assumed this is not the case.

Return: The return value is negative if the request cannot be posted, and zero otherwise.

See also:

```
alt_dma_rxchan_close()
alt_dma_rxchan_depth()
alt_dma_rxchan_ioctl()
alt_dma_rxchan_open()
alt_dma_rxchan_prepare()
alt_dma_rxchan_reg()
alt_dma_txchan_close()
alt_dma_txchan_ioctl()
alt_dma_txchan_open()
alt_dma_txchan_reg()
alt_dma_txchan_space()
```

alt_dma_txchan_space()

Prototype:	`int alt_dma_txchan_space (alt_dma_txchan dma)`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	See description.
Available from ISR:	See description.
Include:	<**sys/alt_dma.h**>
Description:	The `alt_dma_txchan_space()` function returns the number of transmit requests that can be posted to the specified DMA transmit channel, `dma`. A negative value indicates that the value cannot be determined. Whether this function is thread-safe, or can be called from an ISR is dependent on the underlying device driver. In general it should be assumed this is not the case.
Return:	Returns the number of transmit requests that can be posted.

See also:	`alt_dma_rxchan_close()`
	`alt_dma_rxchan_depth()`
	`alt_dma_rxchan_ioctl()`
	`alt_dma_rxchan_open()`
	`alt_dma_rxchan_prepare()`
	`alt_dma_rxchan_reg()`
	`alt_dma_txchan_close()`
	`alt_dma_txchan_ioctl()`
	`alt_dma_txchan_open()`
	`alt_dma_txchan_reg()`
	`alt_dma_txchan_send()`

alt_erase_flash_block()

Prototype:

```
int alt_erase_flash_block(alt_flash_fd* fd,
                          int           offset,
                          int           length)
```

Commonly called by: C/C++ programs
Device drivers

Thread-safe: No.

Available from ISR: No.

Include: <**sys/alt_flash.h**>

Description: The `alt_erase_flash_block()` function erases an individual flash erase block. The parameter `fd` specifies the flash device; `offset` is the offset within the flash of the block to erase; `length` is the size of the block to erase. No error checking is performed to check that this is a valid block, or that the length is correct. See the "Fine-Grained Flash Access" section of the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*.

Only call the `alt_erase_flash_block()` function when operating in single threaded mode.

The only valid values for the `fd` parameter are those returned from the `alt_flash_open_dev` function. If any other value is passed the behavior of this function is undefined.

Return: A return value of zero indicates success. A negative return value indicates failure.

See also:
`alt_flash_close_dev()`
`alt_flash_open_dev()`
`alt_get_flash_info()`
`alt_read_flash()`
`alt_write_flash()`
`alt_write_flash_block()`

alt_flash_close_dev()

Prototype:	`void alt_flash_close_dev(alt_flash_fd* fd)`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	No.
Available from ISR:	No.
Include:	<**sys/alt_flash.h**>
Description:	The `alt_flash_close_dev()` function closes a flash device. All subsequent calls to `alt_write_flash()`, `alt_read_flash()`, `alt_get_flash_info()`, `alt_erase_flash_block()`, or `alt_write_flash_block()` for this flash device fail. Only call the `alt_flash_close_dev()` function when operating in single-threaded mode. The only valid values for the `fd` parameter are those returned from the `alt_flash_open_dev` function. If any other value is passed the behavior of this function is undefined.
Return:	–

See also:	`alt_erase_flash_block()`
	`alt_flash_open_dev()`
	`alt_get_flash_info()`
	`alt_read_flash()`
	`alt_write_flash()`
	`alt_write_flash_block()`

alt_flash_open_dev()

Prototype:	`alt_flash_fd* alt_flash_open_dev(const char* name)`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	No.
Available from ISR:	No.
Include:	<**sys/alt_flash.h**>
Description:	The `alt_flash_open_dev()` function opens a flash device. Once opened a flash device can be written to using `alt_write_flash()`, read from using `alt_read_flash()`, or you can control individual flash blocks using the `alt_get_flash_info()`, `alt_erase_flash_block()`, or `alt_write_flash_block()` function. Only call the `alt_flash_open_dev` function when operating in single threaded mode.
Return:	A return value of zero indicates failure. Any other value is success.

See also:	`alt_erase_flash_block()`
	`alt_flash_close_dev()`
	`alt_get_flash_info()`
	`alt_read_flash()`
	`alt_write_flash()`
	`alt_write_flash_block()`

alt_fs_reg()

Prototype:	`int alt_fs_reg (alt_dev* dev)`
Commonly called by:	Device drivers
Thread-safe:	No.
Available from ISR:	No.
Include:	<**sys/alt_dev.h**>
Description:	The `alt_fs_reg()` function registers a file system with the HAL. Once registered, a file system can be accessed using the standard I/O functions. See the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*. System behavior is undefined in the event that a file system is registered with a name that conflicts with an existing device or file system. `alt_fs_reg()` is not thread safe if other threads are using the device list at the time that `alt_fs_reg()` is called. In practice `alt_fs_reg()` should only be called while operating in a single threaded mode. The expectation is that it is only called by the device initialization functions invoked by `alt_sys_init()`, which in turn should only be called by the single threaded C startup code.
Return:	A return value of zero indicates success. A negative return value indicates failure.
See also:	`alt_dev_reg()`

alt_get_flash_info()

Prototype:

```
int alt_get_flash_info(alt_flash_fd*  fd,
                       flash_region** info,
                       int*          number_of_regions)
```

Commonly called by: C/C++ programs
Device drivers

Thread-safe: No.

Available from ISR: No.

Include: <**sys/alt_flash.h**>

Description: The `alt_get_flash_info()` function gets the details of the erase region of a flash part. The flash part is specified by the descriptor `fd`, a pointer to the start of the `flash_region` structures is returned in the `info` parameter, and the number of flash regions are returned in number of regions.

Only call this function when operating in single threaded mode.

The only valid values for the `fd` parameter are those returned from the `alt_flash_open_dev` function. If any other value is passed the behavior of this function is undefined.

Return: A return value of zero indicates success. A negative return value indicates failure.

See also:

`alt_erase_flash_block()`
`alt_flash_close_dev()`
`alt_flash_open_dev()`
`alt_read_flash()`
`alt_write_flash()`
`alt_write_flash_block()`

alt_icache_flush()

Prototype:	`void alt_icache_flush (void* start, alt_u32 len)`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	<**sys/alt_cache.h**>
Description:	The `alt_icache_flush()` function invalidates the instruction cache for a memory region of length `len` bytes, starting at address `start`. In processors without instruction caches, it has no effect.
Return:	–

See also:	`alt_dcache_flush()`
	`alt_dcache_flush_all()`
	`alt_icache_flush_all()`
	`alt_remap_cached()`
	`alt_remap_uncached()`
	`alt_uncached_free()`
	`alt_uncached_malloc()`

alt_icache_flush_all()

Prototype:	`void alt_icache_flush_all (void)`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	<**sys/alt_cache.h**>
Description:	The `alt_icache_flush_all()` function invalidates the entire contents of the instruction cache. In processors without instruction caches, it has no effect.
Return:	–

See also:	`alt_dcache_flush()`
	`alt_dcache_flush_all()`
	`alt_icache_flush()`
	`alt_remap_cached()`
	`alt_remap_uncached()`
	`alt_uncached_free()`
	`alt_uncached_malloc()`

alt_irq_disable()

Prototype:	`int alt_irq_disable (alt_u32 id)`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**sys/alt_irq.h**>
Description:	The `alt_irq_disable()` function disables a single interrupt.
Return:	The return value is zero.

See also:	`alt_irq_disable_all()`
	`alt_irq_enable()`
	`alt_irq_enable_all()`
	`alt_irq_enabled()`
	`alt_irq_register()`

alt_irq_disable_all()

Prototype:	`alt_irq_context alt_irq_disable_all (void)`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**sys/alt_irq.h**>
Description:	The `alt_irq_disable_all()` function disables all interrupts.
Return:	Pass the return value as the input argument to a subsequent call to `alt_irq_enable_all()`.

See also:	`alt_irq_disable()`
	`alt_irq_enable()`
	`alt_irq_enable_all()`
	`alt_irq_enabled()`
	`alt_irq_register()`

alt_irq_enable()

Prototype:	`int alt_irq_enable (alt_u32 id)`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**sys/alt_irq.h**>
Description:	The `alt_irq_enable()` function enables a single interrupt.
Return:	The return value is zero.

See also:	`alt_irq_disable()`
	`alt_irq_disable_all()`
	`alt_irq_enable_all()`
	`alt_irq_enabled()`
	`alt_irq_register()`

alt_irq_enable_all()

Prototype:	`void alt_irq_enable_all (alt_irq_context context)`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	<**sys/alt_irq.h**>
Description:	The `alt_irq_enable_all()` function enables all interrupts that were previously disabled by `alt_irq_disable_all()`. The input argument, `context`, is the value returned by a previous call to `alt_irq_disable_all()`. Using `context` allows nested calls to `alt_irq_disable_all()` and `alt_irq_enable_all()`. As a result, `alt_irq_enable_all()` does not necessarily enable all interrupts, such as interrupts explicitly disabled by `alt_irq_disable()`.
Return:	–

See also:	`alt_irq_disable()`
	`alt_irq_disable_all()`
	`alt_irq_enable()`
	`alt_irq_enabled()`
	`alt_irq_register()`

alt_irq_enabled()

Prototype:	`int alt_irq_enabled (void)`
Commonly called by:	Device drivers
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	<**sys/alt_irq.h**>
Description:	The `alt_irq_enabled()` function.
Return:	Returns zero if interrupts are disabled, and non-zero otherwise.

See also:	`alt_irq_disable()`
	`alt_irq_disable_all()`
	`alt_irq_enable()`
	`alt_irq_enable_all()`
	`alt_irq_register()`

alt_irq_register()

Prototype:	`int alt_irq_register (alt_u32 id, void* context, void (*isr)(void*, alt_u32))`
Commonly called by:	Device drivers
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**sys/alt_irq.h**>
Description:	The `alt_irq_register()` function registers an ISR. If the function is successful, the requested interrupt is enabled upon return. The input argument, `id` is the interrupt to enable, `isr` is the function that is called when the interrupt is active, `context` and `id` are the two input arguments to `isr`. Calls to `alt_irq_register()` replace previously registered handlers for interrupt `id`. If `irq_handler` is set to null, the interrupt is disabled.
Return:	The `alt_irq_register()` function returns zero if successful, or non-zero otherwise.

See also:	`alt_irq_disable()`
	`alt_irq_disable_all()`
	`alt_irq_enable()`
	`alt_irq_enable_all()`
	`alt_irq_enabled()`

alt_llist_insert()

Prototype:	`void alt_llist_insert(alt_llist* list,`
	`                      alt_llist* entry)`

Commonly called by:	C/C++ programs Device drivers
Thread-safe:	No.
Available from ISR:	Yes.
Include:	<**sys/alt_llist.h**>
Description:	The `alt_llist_insert()` function inserts the doubly linked list entry `entry` into the list `list`. This operation is not re-entrant. For example, if a list can be manipulated from different threads, or from within both application code and an ISR, some mechanism is required to protect access to the list. Interrupts can be locked, or in MicroC/OS-II, a `mutex` can be used.
Return:	–
See also:	`alt_llist_remove()`

alt_llist_remove()

Prototype:	`void alt_llist_remove(alt_llist* entry)`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	No.
Available from ISR:	Yes.
Include:	<**sys/alt_llist.h**>
Description:	The `alt_llist_remove()` function removes the doubly linked list entry `entry` from the list it is currently a member of. This operation is not re-entrant. For example if a list can be manipulated from different threads, or from within both application code and an ISR, some mechanism is required to protect access to the list. Interrupts can be locked, or in MicroC/OS-II, a `mutex` can be used.
Return:	–
See also:	`alt_llist_insert()`

alt_load_section()

Prototype:	`void alt_load_section(alt_u32* from, alt_u32* to, alt_u32* end)`
Commonly called by:	C/C++ programs
Thread-safe:	No.
Available from ISR:	No.
Include:	<**sys/alt_load.h**>

Description:

When operating in run-from-flash mode, the sections `.exceptions`, `.rodata`, and `.rwdata` are automatically loaded from the boot device to RAM at boot time. However, if there are any additional sections that require loading, the `alt_load_section()` function loads them manually before these sections are used.

The input argument `from` is the start address in the boot device of the section; `to` is the start address in RAM of the section, and `end` is the end address in RAM of the section.

To load one of the additional memory sections provided by the default linker script, use the macro `ALT_LOAD_SECTION_BY_NAME` rather than calling `alt_load_section()` directly. For example, to load the section `.onchip_ram`, use the following code:

```
ALT_LOAD_SECTION_BY_NAME(onchip_ram);
```

The leading '.' is omitted in the section name. This macro is defined in the header **sys/alt_load.h**.

Return: –

See also: –

alt_nticks()

Prototype:	`alt_u32 alt_nticks (void)`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	<**sys/alt_alarm.h**>
Description:	The `alt_nticks()` function.
Return:	Returns the number of elapsed system clock tick since reset. It returns zero if there is no system clock available.

See also:	`alt_alarm_start()`
	`alt_alarm_stop()`
	`alt_sysclk_init()`
	`alt_tick()`
	`alt_ticks_per_second()`
	`gettimeofday()`
	`settimeofday()`
	`times()`
	`usleep()`

alt_read_flash()

Prototype:

```
int alt_read_flash(alt_flash_fd* fd,
                   int           offset,
                   void*         dest_addr,
                   int           length)
```

Commonly called by: C/C++ programs
Device drivers

Thread-safe: No.

Available from ISR: No.

Include: <**sys/alt_flash.h**>

Description: The `alt_read_flash()` function reads data from flash. Length bytes are read from the flash `fd`, offset bytes from the beginning of the flash and are written to the location `dest_addr`.

Only call this function when operating in single threaded mode.

The only valid values for the `fd` parameter are those returned from the `alt_flash_open_dev` function. If any other value is passed the behavior of this function is undefined.

Return: The return value is zero upon success and non-zero otherwise.

See also:

```
alt_erase_flash_block()
alt_flash_close_dev()
alt_flash_open_dev()
alt_get_flash_info()
alt_write_flash()
alt_write_flash_block()
```

alt_remap_cached()

Prototype:	`void* alt_remap_cached (volatile void* ptr, alt_u32 len);`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**sys/alt_cache.h**>
Description:	The `alt_remap_cached()` function remaps a region of memory for cached access. The memory to map is `len` bytes, starting at address `ptr`. Processors that do not have a data cache return uncached memory.
Return:	The return value for this function is the remapped memory region.

See also:	`alt_dcache_flush()`
	`alt_dcache_flush_all()`
	`alt_icache_flush()`
	`alt_icache_flush_all()`
	`alt_remap_uncached()`
	`alt_uncached_free()`
	`alt_uncached_malloc()`

alt_remap_uncached()

Prototype:	`volatile void* alt_remap_uncached (void* ptr, alt_u32 len);`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**sys/alt_cache.h**>
Description:	The `alt_remap_uncached()` function remaps a region of memory for uncached access. The memory to map is `len` bytes, starting at address `ptr`. Processors that do not have a data cache return uncached memory.
Return:	The return value for this function is the remapped memory region.

See also:	`alt_dcache_flush()`
	`alt_dcache_flush_all()`
	`alt_icache_flush()`
	`alt_icache_flush_all()`
	`alt_remap_cached()`
	`alt_uncached_free()`
	`alt_uncached_malloc()`

alt_sysclk_init()

Prototype:	`int alt_sysclk_init (alt_u32 nticks)`
Commonly called by:	Device drivers
Thread-safe:	No.
Available from ISR:	No.
Include:	<**sys/alt_alarm.h**>
Description:	The `alt_sysclk_init()` function registers the presence of a system clock driver. The input argument is the number of ticks per second at which the system clock is run. The expectation is that this function is only called from within `alt_sys_init()`, i.e., while the system is running in single-threaded mode. Concurrent calls to this function might lead to unpredictable results.
Return:	This function returns zero upon success, otherwise it returns a negative value. The call can fail if a system clock driver has already been registered.

See also:	`alt_alarm_start()`
	`alt_alarm_stop()`
	`alt_nticks()`
	`alt_tick()`
	`alt_ticks_per_second()`
	`gettimeofday()`
	`settimeofday()`
	`times()`
	`usleep()`

alt_tick()

Prototype:	`void alt_tick (void)`
Commonly called by:	Device drivers
Thread-safe:	No.
Available from ISR:	Yes.
Include:	<**sys/alt_alarm.h**>
Description:	Only the system clock driver should call the `alt_tick()` function. The driver is responsible for making periodic calls to this function at the rate indicated in the call to `alt_sysclk_init()`. This function provides notification to the system that a system clock tick has occurred. This function runs as a part of the ISR for the system clock driver.
Return:	–

See also:	`alt_alarm_start()`
	`alt_alarm_stop()`
	`alt_nticks()`
	`alt_sysclk_init()`
	`alt_ticks_per_second()`
	`gettimeofday()`
	`settimeofday()`
	`times()`
	`usleep()`

alt_ticks_per_second()

Prototype:	`alt_u32 alt_ticks_per_second (void)`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	<**sys/alt_alarm.h**>
Description:	The `alt_ticks_per_second()` function returns the number of system clock ticks that elapse per second. If there is no system clock available, the return value is zero.
Return:	Returns the number of system clock ticks that elapse per second.

See also:	`alt_alarm_start()`
	`alt_alarm_stop()`
	`alt_nticks()`
	`alt_sysclk_init()`
	`alt_tick()`
	`gettimeofday()`
	`settimeofday()`
	`times()`
	`usleep()`

alt_timestamp()

Prototype:	`alt_u32 alt_timestamp (void)`
Commonly called by:	C/C++ programs
Thread-safe:	See description.
Available from ISR:	See description.
Include:	<**sys/alt_timestamp.**h>
Description:	The `alt_timestamp()` function returns the current value of the timestamp counter. See the "High Resolution Time Measurement" section of the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*. The implementation of this function is provided by the timestamp driver. Therefore, whether this function is thread-safe and or available at interrupt level is dependent on the underlying driver. Always call the `alt_timestamp_start()` function before any calls to `alt_timestamp()`. Otherwise the behavior of `alt_timestamp()` is undefined.
Return:	Returns the current value of the timestamp counter.

See also:	`alt_timestamp_freq()`
	`alt_timestamp_start()`

alt_timestamp_freq()

Prototype:	`alt_u32 alt_timestamp_freq (void)`
Commonly called by:	C/C++ programs
Thread-safe:	See description.
Available from ISR:	See description.
Include:	<**sys/alt_timestamp.h**>
Description:	The `alt_timestamp_freq()` function returns the rate at which the timestamp counter increments. See the "High Resolution Time Measurement" section of the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*. The implementation of this function is provided by the timestamp driver. Therefore, whether this function is thread-safe and or available at interrupt level is dependent on the underlying driver.
Return:	The returned value is the number of counter ticks per second.

See also:	`alt_timestamp()`
	`alt_timestamp_start()`

alt_timestamp_start()

Prototype:	`int alt_timestamp_start (void)`
Commonly called by:	C/C++ programs
Thread-safe:	See description.
Available from ISR:	See description.
Include:	<**sys/alt_timestamp.h**>
Description:	The `alt_timestamp_start()` function starts the system timestamp counter. See the “High Resolution Time Measurement” section of the *Developing Programs using the HAL* chapter of the *Nios II Software Developer’s Handbook*. The implementation of this function is provided by the timestamp driver. Therefore, whether this function is thread-safe and or available at interrupt level is dependent on the underlying driver. This function resets the counter to zero, and starts the counter running.
Return:	The return value is zero upon success and non-zero otherwise.

See also:	`alt_timestamp()`
	`alt_timestamp_freq()`

alt_uncached_free()

Prototype:	`void alt_uncached_free (volatile void* ptr)`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**sys/alt_cache.h**>
Description:	The `alt_uncached_free()` function causes the memory pointed to by `ptr` to be de-allocated, i.e., made available for future allocation through a call to `alt_uncached_malloc()`. The input pointer, `ptr`, points to a region of memory previously allocated through a call to `alt_uncached_malloc()`. Behavior is undefined if this is not the case.
Return:	–

See also:	`alt_dcache_flush()`
	`alt_dcache_flush_all()`
	`alt_icache_flush()`
	`alt_icache_flush_all()`
	`alt_remap_cached()`
	`alt_remap_uncached()`
	`alt_uncached_malloc()`

alt_uncached_malloc()

Prototype:	`volatile void* alt_uncached_malloc (size_t size)`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**sys/alt_cache.h**>
Description:	The `alt_uncached_malloc()` function allocates a region of uncached memory of length `size` bytes. Regions of memory allocated in this way can be released using the `alt_uncached_free()` function. Processors that do not have a data cache return uncached memory.
Return:	If sufficient memory cannot be allocated, this function returns null, otherwise a pointer to the allocated space is returned.

See also:	`alt_dcache_flush()`
	`alt_dcache_flush_all()`
	`alt_icache_flush()`
	`alt_icache_flush_all()`
	`alt_remap_cached()`
	`alt_remap_uncached()`
	`alt_uncached_free()`

alt_write_flash()

Prototype:	`int alt_write_flash(alt_flash_fd* fd,` `int offset,` `const void* src_addr,` `int length)`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	No.
Available from ISR:	No.
Include:	<**sys/alt_flash.h**>
Description:	The `alt_write_flash()` function writes data into flash. The data to be written is at `src_addr` address, length bytes are written into the flash `fd`, offset bytes from the beginning of the flash. Only call this function when operating in single threaded mode. This function does not preserve any non written areas of any flash sectors affected by this write. See the "Simple Flash Access" section of the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*. The only valid values for the `fd` parameter are those returned from the `alt_flash_open_dev` function. If any other value is passed the behavior of this function is undefined.
Return:	The return value is zero upon success and non-zero otherwise.

See also:	`alt_erase_flash_block()`
	`alt_flash_close_dev()`
	`alt_flash_open_dev()`
	`alt_get_flash_info()`
	`alt_read_flash()`
	`alt_write_flash_block()`

alt_write_flash_block()

Prototype:

```
int alt_write_flash_block(alt_flash_fd* fd,
                          int           block_offset,
                          int           data_offset,
                          const void    *data,
                          int           length)
```

Commonly called by: C/C++ programs
Device drivers

Thread-safe: No.

Available from ISR: No.

Include: <**sys/alt_flash.h**>

Description: The `alt_write_flash_block()` function writes one erase block of flash. The flash device is specified by `fd`, the block offset is the offset within the flash of the start of this block, `data_offset` is the offset within the flash at which to start writing data, `data` is the data to write, `length` is how much data to write. Note, no check is made on any of the parameters. See the "Fine-Grained Flash Access" section of the *Developing Programs using the HAL* chapter of the *Nios II Software Developer's Handbook*.

Only call this function when operating in single threaded mode.

The only valid values for the `fd` parameter are those returned from the `alt_flash_open_dev` function. If any other value is passed the behavior of this function is undefined.

Return: The return value is zero upon success and non-zero otherwise.

See also:
`alt_erase_flash_block()`
`alt_flash_close_dev()`
`alt_flash_open_dev()`
`alt_get_flash_info()`
`alt_read_flash()`
`alt_write_flash()`

close()

Prototype:	`int close (int fd)`
Commonly called by:	C/C++ programs Newlib C library
Thread-safe:	See description.
Available from ISR:	No.
Include:	<**unistd.h**>
Description:	The `close()` function is the standard UNIX style `close()` function, which closes the file descriptor `fd`. Calls to `close()` are only thread-safe, if the implementation of `close()` provided by the driver that is manipulated is thread-safe. Valid values for the `fd` parameter are: `stdout`, `stdin` and `stderr`, or any value returned from a call to `open()`.
Return:	The return value is zero upon success, and –1 otherwise. If an error occurs, `errno` is set to indicate the cause.

See also:	`fcntl()`
	`fstat()`
	`ioctl()`
	`isatty()`
	`lseek()`
	`open()`
	`read()`
	`stat()`
	`write()`
	Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II *<version>*, **Nios II Documentation**.

execve()

Prototype:

```
int execve(const char   *path,
           char *const  argv[],
           char *const  envp[])
```

Commonly called by: Newlib C library

Thread-safe: Yes.

Available from ISR: Yes.

Include: <**unistd.h**>

Description: The `execve()` function is only provided for compatibility with newlib.

Return: Calls to `execve()` always fail with the return code –1 and `errno` set to `ENOSYS`.

See also: Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II <*version*>, **Nios II Documentation**.

fcntl()

Prototype:	`int fcntl(int fd, int cmd)`
Commonly called by:	C/C++ programs
Thread-safe:	No.
Available from ISR:	No.
Include:	<**unistd.h**> <**fcntl.h**>
Description:	The `fcntl()` function a limited implementation of the standard `fcntl()` system call, which can change the state of the flags associated with an open file descriptor. Normally these flags are set during the call to `open()`. The main use of this function is to change the state of a device from blocking to non-blocking (for device drivers that support this feature). The input argument `fd` is the file descriptor to be manipulated. `cmd` is the command to execute, which can be either `F_GETFL` (return the current value of the flags) or `F_SETFL` (set the value of the flags).
Return:	If `cmd` is `F_SETFL`, the argument `arg` is the new value of flags, otherwise `arg` is ignored. Only the flags `O_APPEND` and `O_NONBLOCK` can be updated by a call to fcntl(). All other flags remain unchanged. The return value is zero upon success, or –1 otherwise. If `cmd` is `F_GETFL`, the return value is the current value of the flags. If there is an error, –1 is returned. In the event of an error, `errno` is set to indicate the cause.

See also:	`close()`
	`fstat()`
	`ioctl()`
	`isatty()`
	`lseek()`
	`read()`
	`stat()`
	`write()`
	Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II *<version>*, **Nios II Documentation**.

fork()

Prototype:	`pid_t fork (void)`
Commonly called by:	Newlib C library
Thread-safe:	Yes.
Available from ISR:	no
Include:	<**unistd.h**>
Description:	The `fork()` function is only provided for compatibility with newlib.
Return:	Calls to `fork()` always fails with the return code –1 and `errno` set to `ENOSYS`.
See also:	Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II *<version>*, **Nios II Documentation**.

fstat()

Prototype:	`int fstat (int fd, struct stat *st)`
Commonly called by:	C/C++ programs Newlib C library
Thread-safe:	See description.
Available from ISR:	No.
Include:	<**sys/stat.h**>
Description:	The `fstat()` function obtains information about the capabilities of an open file descriptor. The underlying device driver fills in the input `st` structure with a description of its functionality. See the header file **sys/stat.h** provided with the compiler for the available options. By default file descriptors are marked as character devices, if the underlying driver does not provide its own implementation of the `fsat()` function. Calls to `fstat()` are only thread-safe, if the implementation of `fstat()` provided by the driver that is manipulated is thread-safe. Valid values for the `fd` parameter are: `stdout`, `stdin` and `stderr`, or any value returned from a call to `open()`.
Return:	The return value is zero upon success, or –1 otherwise. If the call fails, `errno` is set to indicate the cause of the error.

See also:	`close()`
	`fcntl()`
	`ioctl()`
	`isatty()`
	`lseek()`
	`open()`
	`read()`
	`stat()`
	`write()`
	Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II *<version>*, **Nios II Documentation**.

getpid()

Prototype:	`pid_t getpid (void)`
Commonly called by:	Newlib C library
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**unistd.h**>
Description:	The `getpid()` function is provided for newlib compatibility and obtains the current process `id`.
Return:	Because HAL systems cannot contain multiple processes, `getpid()` always returns the same `id` number.
See also:	Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II *<version>*, **Nios II Documentation**.

gettimeofday()

Prototype:	`int gettimeofday(struct timeval *ptimeval,`
	 `                 struct timezone *ptimezone)`

Commonly called by:	C/C++ programs Newlib C library
Thread-safe:	See description.
Available from ISR:	Yes.
Include:	<**sys/time.h**>
Description:	The `gettimeofday()` function obtains a time structure that indicates the current wall clock time. This time is calculated using the elapsed number of system clock ticks, and the current time value set through the last call to `settimeofday()`. If this function is called concurrently with a call to `settimeofday()`, the value returned by `gettimeofday()` is unreliable; however, concurrent calls to `gettimeofday()` are legal.
Return:	The return value is zero upon success, or –1 otherwise. If the call fails, `errno` is set to indicate the cause of the error.

See also:	`alt_alarm_start()`
	 `alt_alarm_stop()`
	 `alt_nticks()`
	 `alt_sysclk_init()`
	 `alt_tick()`
	 `alt_ticks_per_second()`
	 `settimeofday()`
	 `times()`
	 `usleep()`
	 Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II *<version>*, **Nios II Documentation**.

ioctl()

Prototype:	`int ioctl (int fd, int req, void* arg)`
Commonly called by:	C/C++ programs
Thread-safe:	See description.
Available from ISR:	No.
Include:	<**sys/ioctl.h**>
Description:	The `ioctl()` function allows application code to manipulate the I/O capabilities of a device driver in driver specific ways. This function is equivalent to the standard UNIX `ioctl()` function. The input argument `fd` is an open file descriptor for the device to manipulate, `req` is an enumeration defining the operation request, and the interpretation of `arg` is request specific. In general, this implementation vectors requests to the appropriate drivers `ioctl()` function (as registered in the drivers `alt_dev` structure). However, in the case of devices (as opposed to file subsystems), the `TIOCEXCL` and `TIOCNXCL` requests are handled without reference to the driver. These requests lock and release a device for exclusive access. Calls to `ioctl()` are only thread-safe if the implementation of `ioctl()` provided by the driver that is manipulated is thread-safe. Valid values for the `fd` parameter are: `stdout`, `stdin` and `stderr`, or any value returned from a call to `open()`.
Return:	The interpretation of the return value is request specific. If the call fails, `errno` is set to indicate the cause of the error.

See also:	`close()`
	`fcntl()`
	`fstat()`
	`isatty()`
	`lseek()`
	`open()`
	`read()`
	`stat()`
	`write()`
	Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II <*version*>, **Nios II Documentation**.

isatty()

Prototype:	`int isatty(int fd)`
Commonly called by:	C/C++ programs Newlib C library
Thread-safe:	See description.
Available from ISR:	No.
Include:	<**unistd.h**>
Description:	The `isatty()` function determines whether the device associated with the open file descriptor `fd` is a terminal device. This implementation uses the drivers `fstat()` function to determine its reply. Calls to `isatty()` are only thread-safe, if the implementation of `fstat()` provided by the driver that is manipulated is thread-safe.
Return:	The return value is 1 if the device is a character device, and zero otherwise. If an error occurs, `errno` is set to indicate the cause.

See also:	`close()`
	`fcntl()`
	`fstat()`
	`ioctl()`
	`lseek()`
	`open()`
	`read()`
	`stat()`
	`write()`
	Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II *<version>*, **Nios II Documentation**.

kill()

Prototype:	`int kill(int pid, int sig)`
Commonly called by:	Newlib C library
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	<**signal.h**>
Description:	The `kill()` function is used by newlib to send signals to processes. The input argument `pid` is the `id` of the process to signal, and `sig` is the signal to send. As there is only a single process in the HAL, the only valid values for `pid` are either the current process `id`, as returned by `getpid()`, or the broadcast values, i.e., `pid` must be less than or equal to zero. The following signals result in an immediate shutdown of the system, without call to `exit()`: `SIGABRT`, `SIGALRM`, `SIGFPE`, `SIGILL`, `SIGKILL`, `SIGPIPE`, `SIGQUIT`, `SIGSEGV`, `SIGTERM`, `SIGUSR1`, `SIGUSR2`, `SIGBUS`, `SIGPOLL`, `SIGPROF`, `SIGSYS`, `SIGTRAP`, `SIGVTALRM`, `SIGXCPU`, and `SIGXFSZ`. The following signals are ignored: `SIGCHLD` and `SIGURG`. All the remaining signals are treated as errors.
Return:	The return value is zero upon success, or –1 otherwise. If the call fails, `errno` is set to indicate the cause of the error.
See also:	Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II *<version>*, **Nios II Documentation**.

link()

Prototype:	`int link(const char *_path1,`
	`         const char *_path2)`

Commonly called by:	Newlib C library
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	<**unistd.h**>
Description:	The `link()` function is only provided for compatibility with newlib.
Return:	Calls to `link()` always fails with the return code –1 and `errno` set to `ENOSYS`.
See also:	Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II *<version>*, **Nios II Documentation**.

lseek()

Prototype:	`off_t lseek(int fd, off_t ptr, int whence)`
Commonly called by:	C/C++ programs Newlib C library
Thread-safe:	See description.
Available from ISR:	No.
Include:	<**unistd.h**>

Description:

The `lseek()` function moves the read/write pointer associated with the file descriptor `fd`. This function vectors the call to the `lseek()` function provided by the driver associated with the file descriptor. If the driver does not provide an implementation of `lseek()`, an error is indicated.

`lseek()` corresponds to the standard UNIX `lseek()` function.

You can use the following values for the input parameter, `whence`:

- Value of `whence`
- Interpretation
- `SEEK_SET`—the offset is set to `ptr` bytes.
- `SEEK_CUR`—the offset is incremented by `ptr` bytes.
- `SEEK_END`—the offset is set to the end of the file plus `ptr` bytes.

Calls to `lseek()` are only thread-safe if the implementation of `lseek()` provided by the driver that is manipulated is thread-safe.

Valid values for the `fd` parameter are: `stdout`, `stdin` and `stderr`, or any value returned from a call to `open()`.

Return:

Upon success, the return value is a non-negative file pointer. The return value is –1 in the event of an error. If the call fails, `errno` is set to indicate the cause of the error.

See also:

`close()`
`fcntl()`
`fstat()`
`ioctl()`
`isatty()`
`open()`
`read()`
`stat()`
`write()`
Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II *<version>*, **Nios II Documentation**.

open()

Prototype:	`int open (const char* pathname, int flags, mode_t mode)`
Commonly called by:	C/C++ programs
Thread-safe:	See description.
Available from ISR:	No.
Include:	<**unistd.h**> <**fcntl.h**>
Description:	The `open()` function opens a file or device and returns a file descriptor (a small, non-negative integer for use in read, write, etc.) `flags` is one of: `O_RDONLY`, `O_WRONLY`, or `O_RDWR`, which request opening the file read-only, write-only or read/write, respectively. You can also bitwise-`OR` `flags` with `O_NONBLOCK`, which causes the file to be opened in non-blocking mode. Neither `open()` nor any subsequent operations on the returned file descriptor causes the calling process to wait. Note that not all file systems/devices recognize this option. `mode` specifies the permissions to use, if a new file is created. It is unused by current file systems, but is maintained for compatibility. Calls to `open()` are only thread-safe if the implementation of `open()` provided by the driver that is manipulated is thread-safe.
Return:	The return value is the new file descriptor, and –1 otherwise. If an error occurs, `errno` is set to indicate the cause.

See also:	`close()`
	`fcntl()`
	`fstat()`
	`ioctl()`
	`isatty()`
	`lseek()`
	`read()`
	`stat()`
	`write()`
	Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II *<version>*, **Nios II Documentation**.

read()

Prototype: `int read(int fd, void *ptr, size_t len)`

Commonly called by: C/C++ programs
Newlib C library

Thread-safe: See description.

Available from ISR: No.

Include: <**unistd.h**>

Description: The `read()` function reads a block of data from a file or device. This function vectors the request to the device driver associated with the input open file descriptor `fd`. The input argument, `ptr`, is the location to place the data read and `len` is the length of the data to read in bytes.

Calls to `read()` are only thread-safe if the implementation of `read()` provided by the driver that is manipulated is thread-safe.

Valid values for the `fd` parameter are: `stdout`, `stdin` and `stderr`, or any value returned from a call to `open()`.

Return: The return argument is the number of bytes read, which might be less than the requested length.

A return value of –1 indicates an error. In the event of an error, `errno` is set to indicate the cause.

See also:
`close()`
`fcntl()`
`fstat()`
`ioctl()`
`isatty()`
`lseek()`
`open()`
`stat()`
`write()`
Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II *<version>*, **Nios II Documentation**.

sbrk()

Prototype:	`caddr_t sbrk(int incr)`
Commonly called by:	Newlib C library
Thread-safe:	No.
Available from ISR:	No.
Include:	<**unistd.h**>
Description:	The `sbrk()` function dynamically extends the data segment for the application. The input argument `incr` is the size of the block to allocate. Do not call `sbrk()` directly–if you wish to dynamically allocate memory, use the newlib `malloc()` function.
Return:	–
See also:	Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II *<version>*, **Nios II Documentation**.

settimeofday()

Prototype:	`int settimeofday (const struct timeval  *t,`
	`                   const struct timezone *tz)`

Commonly called by:	C/C++ programs
Thread-safe:	No.
Available from ISR:	Yes.
Include:	<**sys/time.h**>
Description:	If the `settimeofday()` function is called concurrently with a call to `gettimeofday()`, the value returned by `gettimeofday()` is unreliable.
Return:	The return value is zero upon success, or –1 otherwise. The current implementation always succeeds.

See also:	`alt_alarm_start()`
	`alt_alarm_stop()`
	`alt_nticks()`
	`alt_sysclk_init()`
	`alt_tick()`
	`alt_ticks_per_second()`
	`gettimeofday()`
	`times()`
	`usleep()`

stat()

Prototype:	`int stat(const char  *file_name,`
	`         struct stat *buf);`

Commonly called by:	C/C++ programs Newlib C library
Thread-safe:	See description.
Available from ISR:	No.
Include:	<**sys/stat.h**>
Description:	The `stat()` function is similar to the `fstat()` function—it obtains status information about a file. Instead of using an open file descriptor, like `fstat()`, `stat()` takes the name of a file as an input argument. Calls to `stat()` are only thread-safe, if the implementation of `stat()` provided by the driver that is manipulated is thread-safe. Internally, the `stat()` function is implemented as a call to `fstat()`. See "fstat()" on page 10–56.
Return:	–

See also:	`close()`
	`fcntl()`
	`fstat()`
	`ioctl()`
	`isatty()`
	`lseek()`
	`open()`
	`read()`
	`write()`
	Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II *<version>*, **Nios II Documentation**.

times()

Prototype:	`clock_t times (struct tms *buf)`
Commonly called by:	C/C++ programs Newlib C library
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	<**sys/times.h**>

Description:

This `times()` function is provided for compatibility with newlib. It returns the number of clock ticks since reset. It also fills in the structure pointed to by the input parameter `buf` with time accounting information. The definition of the `tms` structure is:

```
typedef struct
{
  clock_t tms_utime;
  clock_t tms_stime;
  clock_t tms_cutime;
  clock_t tms_cstime;
};
```

The structure has the following elements:

- `tms_utime`: the CPU time charged for the execution of user instructions
- `tms_stime`: the CPU time charged for execution by the system on behalf of the process
- `tms_cutime`: the sum of all the `tms_utime` and `tms_cutime` of the child processes
- `tms_cstime`: the sum of the `tms_stimes` and `tms_cstimes` of the child processes

In practice, all elapsed time is accounted as system time. No time is ever attributed as user time. In addition, no time is allocated to child processes, as child processes can not be spawned by the HAL.

Return: If there is no system clock available, the return value is zero.

See also:

`alt_alarm_start()`
`alt_alarm_stop()`
`alt_nticks()`
`alt_sysclk_init()`
`alt_tick()`
`alt_ticks_per_second()`
`gettimeofday()`
`settimeofday()`
`usleep()`
Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II *<version>*, **Nios II Documentation**.

unlink()

Prototype:	`int unlink(char *name)`
Commonly called by:	Newlib C library
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	<**unistd.h**>
Description:	The `unlink()` function is only provided for compatibility with newlib.
Return:	Calls to `unlink()` always fails with the return code –1 and `errno` set to `ENOSYS`.
See also:	Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II <*version*>, **Nios II Documentation**.

usleep()

Prototype:	`int usleep (int us)`
Commonly called by:	C/C++ programs Device drivers
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**unistd.h**>
Description:	The `usleep()` function blocks until at least `us` microseconds have elapsed.
Return:	The `usleep()` function returns zero upon success, or –1 otherwise. If an error occurs, `errno` is set to indicate the cause. The current implementation always succeeds.

See also:	`alt_alarm_start()`
	`alt_alarm_stop()`
	`alt_nticks()`
	`alt_sysclk_init()`
	`alt_tick()`
	`alt_ticks_per_second()`
	`gettimeofday()`
	`settimeofday()`
	`times()`

wait()

Prototype:	`int wait(int *status)`
Commonly called by:	Newlib C library
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	<**sys/wait.h**>
Description:	Newlib uses the `wait()` function to wait for all child processes to exit. Because the HAL does not support spawning child processes, this function returns immediately.
Return:	Upon return, the content of `status` is set to zero, which indicates there is no child processes. The return value is always –1 and `errno` is set to `ECHILD`, which indicates that there are no child processes to wait for.
See also:	Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II *<version>*, **Nios II Documentation**.

write()

Prototype:	`int write(int fd, const void *ptr, size_t len)`
Commonly called by:	C/C++ programs Newlib C library
Thread-safe:	See description.
Available from ISR:	no
Include:	<**unistd.h**>
Description:	The `write()` function writes a block of data to a file or device. This function vectors the request to the device driver associated with the input file descriptor `fd`. The input argument `ptr` is the data to write and `len` is the length of the data in bytes. Calls to `write()` are only thread-safe if the implementation of `write()` provided by the driver that is manipulated is thread-safe. Valid values for the `fd` parameter are: `stdout`, `stdin` and `stderr`, or any value returned from a call to `open()`.
Return:	The return argument is the number of bytes written, which might be less than the requested length. A return value of –1 indicates an error. In the event of an error, `errno` is set to indicate the cause.

See also:	`close()`
	`fcntl()`
	`fstat()`
	`ioctl()`
	`isatty()`
	`lseek()`
	`open()`
	`read()`
	`stat()`
	Newlib documentation. On the Windows Start menu, click Programs, Altera, Nios II *<version>*, **Nios II Documentation**.

Standard Types

In the interest of portability, the HAL uses a set of standard type definitions in place of the ANSI C built-in types. Table 10–2 describes these types that are defined in the header **alt_types.h**.

Table 10–2. Standard Types

Type	Description
`alt_8`	Signed 8-bit integer.
`alt_u8`	Unsigned 8-bit integer.
`alt_16`	Signed 16-bit integer.
`alt_u16`	Unsigned 16-bit integer.
`alt_32`	Signed 32-bit integer.
`alt_u32`	Unsigned 32-bit integer.
`alt_64`	Signed 64-bit integer.
`alt_u64`	Unsigned 64-bit integer.

Document Revision History

Table 10–3 shows the revision history for this document.

Table 10–3. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
November 2006, v6.1.0	Function `open()` requires `fcntl.h`.	
May 2006, v6.0.0	No change from previous release.	
October 2005, v5.1.0	Added API entries for "alt_irq_disable()" and "alt_irq_enable()", which were previously omitted by error.	
May 2005, v5.0.0	● Added `alt_load_section()` function ● Added `fcntl()` function	
December 2004 v1.2	Updated names of DMA generic requests.	
September 2004 v1.1	● Added `open()`. ● Added `ERRNO` information to `alt_dma_txchan_open()`. ● Corrected `ALT_DMA_TX_STREAM_ON` definition. ● Corrected `ALT_DMA_RX_STREAM_ON` definition. ● Added information to `alt_dma_rxchan_ioctl()` and `alt_dma_txchan_ioctl()`.	
May 2004 v1.0	First publication.	

11. Altera-Provided Development Tools

NII520011-6.1.0

Introduction

This chapter introduces all of the development tools that Altera provides for the Nios® II processor. These tools fall into the following categories:

- The Nios II integrated development environment (IDE) and associated tools
- Altera® command-line tools
- GNU compiler tool-chain
- Libraries and embedded software components

This chapter does not describe detailed usage of any of the tools, but it refers you to the most appropriate documentation.

The Nios II IDE Tools

Table 11–1 describes the tools provided by the Nios II IDE user interface.

Table 11–1. The Nios II IDE & Associated Tools (Part 1 of 2)

Tools	Description
The Nios II IDE	The Nios II IDE is the software development user interface for the Nios II processor. All software development tasks can be accomplished within the IDE, including editing, building, and debugging programs. For more information, refer to the Nios II IDE help system.
Flash programmer	The Nios II IDE includes a flash programmer utility that allows you to program flash memory chips on a target board. The flash programmer supports programming flash on any board, including Altera development boards and your own custom boards. The flash programmer facilitates programming flash for the following purposes: ● Executable code and data ● Bootstrap code to copy code from flash to RAM, and then run from RAM. ● HAL file subsystems ● FPGA hardware configuration data For more information, refer to the *Nios II Flash Programmer User Guide*.

Table 11–1. The Nios II IDE & Associated Tools (Part 2 of 2)

Tools	Description
Instruction set simulator	Altera provides an instruction set simulator (ISS) for the Nios II processor. The ISS is available within the Nios II IDE, and the process for running and debugging programs on the ISS is the same as for running and debugging on target hardware. For more information, refer to the Nios II IDE help system.
Quartus II Programmer	The Quartus II programmer is part of the Quartus II software, however the Nios II IDE can launch the Quartus II programmer directly. The Quartus II programmer allows you to download new FPGA configuration files to the board. For more information, refer to the Nios II IDE help system, or press the F1 key while the Quartus II programmer is open.

Altera Command-Line Tools

This section describes the command-line tools provided by Altera. You can run these tools from a *Nios II Command Shell* command prompt, for example, to write a script to automate compilation tasks. The Altera command-line tools are in the <*Nios II EDS install path*>**/bin/** directory.

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

```
<name of tool> --help
```

Table 11–2 shows command-line utilities that create and build Nios II IDE projects without launching the Nios II IDE graphical user interface (GUI). These utilities allow you to automate Nios II IDE operations using command-line scripts. For example, with the help of these utilities, a script can check out a Nios II IDE project from source control, import the project into the Nios II IDE workspace, and build the project.

Each of these utilities launches the Nios II IDE in the background, without displaying the GUI. You cannot use these utilities while the IDE is running, because only one instance of the Nios II IDE can be active at a time.

Table 11–2. Nios II IDE Command Line Tools (Part 1 of 2)

Tool	Description
`nios2-create-system-library`	Creates a new system library project.
`nios2-create-application-project`	Creates a new C/C++ application project.

Table 11–2. Nios II IDE Command Line Tools (Part 2 of 2)

Tool	Description
`nios2-build-project`	Builds a project using the Nios II IDE managed-make facilities. Creates or updates the makefiles to build the project, and optionally runs make. `nios2-build-project` operates only on projects that exist in the current Nios II IDE workspace.
`nios2-import-project`	Imports a previously-created Nios II IDE project into the current workspace.
`nios2-delete-project`	Removes a project from the Nios II IDE workspace, and optionally deletes files from the file system.

Table 11–3 shows other Altera-provided command-line tools for developing Nios II programs.

Table 11–3. Altera Command-Line Tools

Tool	Description
`nios2-download`	Downloads code to a target processor for debugging or running.
`nios2-flash-programmer`	Programs data to flash memory on the target board.
`nios2-gdb-server`	Translates GNU debugger (GDB) remote serial protocol packets over TCP to joint test action group (JTAG) transactions with a target Nios II processor.
`nios2-terminal`	Performs terminal I/O with a JTAG universal asynchronous receiver-transmitter (UART) in a Nios II system
`validate_zip`	Verifies if a specified zip file is compatible with Altera's read-only zip file system.
`nios2-debug`	Downloads a program to a Nios II processor and launches the Insight debugger.
`nios2-console`	Opens the FS2 command-line interface (CLI), connects to the Nios II processor, and (optionally) downloads code.
`nios2-configure-sof`	Configures an Altera configurable part. If no explicit SOF file is specified, it will try to determine the correct file to use.

File format conversion is sometimes necessary when passing data from one utility to another. Table 11–4 shows the Altera-provided utilities for converting file formats.

Table 11–4. File Conversion Utilities

Utility	Description
`bin2flash`	Converts binary files to a **.flash** file for programming into flash memory.
`elf2dat`	Converts an **.elf** executable file format to a **.dat** file format appropriate for Verilog HDL hardware simulators.
`elf2flash`	Converts an **.elf** executable file to a **.flash** file for programming into flash memory.
`elf2hex`	Converts an **.elf** executable file to the Intel **.hex** file format.
`elf2mem`	Generates the memory contents for the memory devices in a specific Nios II system.
`elf2mif`	Converts an **.elf** executable file to the Quartus II memory initialization file **(.mif)** format
`flash2dat`	Converts a **.flash** file to the **.dat** file format appropriate for Verilog HDL hardware simulators.
`mk-nios2-signaltap-mnemonic-table`	Takes an **.elf** file and an SOPC Builder system file (**.ptf**) and creates a **.stp** file containing mnemonic tables for Nios II instructions and symbols for Altera's SignalTap® II logic analyzer.
`sof2flash`	Converts an FPGA configuration file (**.sof**) to a **.flash** file for programming into flash memory.

GNU Compiler Tool-chain

Altera provides and supports the standard GNU compiler tool-chain for the Nios II processor. Complete HTML documentation for the GNU tools resides in the Nios II Embedded Design Suite (EDS) directory. The GNU tools are in the <*Nios II EDS install path*>**/bin/nios2-gnutools** directory.

GNU tools for the Nios II processor are generally named **nios2-elf-**<*tool name*>. The following list shows some examples:

- `nios2-elf-gcc`
- `make`
- `nios2-elf-as`
- `nios2-elf-ld`
- `nios2-elf-objdump`
- `nios2-elf-size`

For a comprehensive list, refer to the GNU HTML documentation.

Libraries & Embedded Software Components

Table 11–5 shows the Nios II libraries and software components.

Table 11–5. Libraries & Software Components

Name	Description
Hardware abstraction layer (HAL) system library	See the *Overview of the HAL System Library* chapter of the *Nios II Software Developer's Handbook*.
MicroC/OS-II RTOS	See the *MicroC/OS-II Real-Time Operating System* chapter of the *Nios II Software Developer's Handbook*.
Lightweight IP TCP/IP stack	See the *Ethernet & Lightweight IP* chapter of the *Nios II Software Developer's Handbook*.
Newlib ANSI C standard library	See the *Overview of the HAL System Library* chapter of the *Nios II Software Developer's Handbook*. The complete HTML documentation for newlib resides in the Nios II EDS directory.
Read-only zip file system	See the *Read-Only Zip File System* chapter of the *Nios II Software Developer's Handbook*.
Example designs	The Nios II EDS provides documented software examples to demonstrate all prominent features of the Nios II processor and the development environment.

Document Revision History

Table 11–6 shows the revision history for this document.

Table 11–6. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
November 2006, v6.1.0	No change from previous release.	
May 2006, v6.0.0	● Added **nios2-configure-sof** tool. ● Removed utilities for the legacy SDK flow, because it is no longer supported.	
October 2005, v5.1.0	No change from previous release.	
May 2005, v5.0.0	No change from previous release.	
December 2004 v1.1	Added Nios II command line tools information.	
May 2004 v1.0	First publication.	

12. Read-Only Zip File System

NII520012-6.1.0

Introduction

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

The Altera® read-only zip file system is provided as a software component for use in the Nios II integrated development environment (IDE). All source and header files for the HAL drivers are located in the directory <*Nios II EDS install path*>**/components/altera_ro_zipfs/HAL/**.

Using the Zip File System in a Project

The read-only zip file system is supported under the Nios II IDE user interface. You do not have to edit any source code to include and configure the file system. To use the zip file system, you use the Nios II IDE graphical user interface (GUI) to include it as a software component for the system library project.

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

- The name of the flash device you wish to program the file system into
- The offset with this flash.
- The name of the mount point for this file subsystem within the HAL file system. For example, if you name the mount point **/mnt/zipfs**, the following code called from within a HAL-based program opens the file **hello** within the zip file:
  `fopen("/mnt/zipfs/hello", "r")`
- The name of the zip file you wish to use. Before you can specify the zip filename, you must first import it into the Nios II IDE system library project.

For details on importing, see the Nios II IDE help system.

The next time you build your project after you make these settings, the Nios II IDE includes and links the file subsystem in the project. After rebuilding, the **system.h** file reflects the presence of this software component in the system.

Preparing the Zip File

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

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

Programming the Zip File to Flash

For your program to access files in the zip file subsystem, you must first program the zip data into flash. As part of the project build process, the Nios II IDE creates a **.flash** file that includes the data for the zip file system. This file is in the **Release** directory of your project.

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

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

Document Revision History

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

Table 12–1. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
November 2006, v6.1.0	No change from previous release.	
May 2006, v6.0.0	No change from previous release.	
October 2005, v5.1.0	No change from previous release.	
May 2005, v5.0.0	No change from previous release.	
May 2004 v1.0	First publication.	

13. Ethernet & Lightweight IP

NII52009-6.1.0

Usage Note

Do not incorporate the Lightweight IP (lwIP) transmission control protocol/Internet protocol (TCP/IP) suite in new software projects. lwIP is an older networking solution, provided for compatibility with existing customer networking designs. lwIP will be removed from the Nios II EDS in a future release.

This chapter is included for reference in case you are maintaining an existing lwIP-based software application. If you are developing a new networking application, use the NicheStack® TCP/IP Stack - Nios II Edition.

The NicheStack TCP/IP Stack is discussed in the *Ethernet & the NicheStack TCP/IP Stack - Nios II Edition* chapter of the *Nios II Software Developer's Handbook*.

lwIP is incompatible with the NicheStack TCP Stack - Nios II Edition software component.

Introduction

Lightweight IP (lwIP) is a small-footprint implementation of the TCP/IP suite. The focus of the lwIP TCP/IP implementation is to reduce resource usage while providing a full scale TCP/IP. lwIP is designed for use in embedded systems with small memory footprints, making it suitable for Nios® II processor systems.

lwIP includes the following features:

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

lwIP Port for the Nios II Processor

Altera provides the Nios II port of lwIP, including source code, in the Nios II Embedded Design Suite (EDS). lwIP provides you with immediate, open-source access to a stack for Ethernet connectivity for the Nios II processor. The Altera® port of lwIP includes a sockets API wrapper, providing the standard, well-documented socket API.

The Nios II EDS include several working examples of programs using lwIP for your reference. In fact, Nios development boards are pre-programmed with a web server reference design based on lwIP and the MicroC/OS-II real-time operating system (RTOS). Full source code is provided.

Altera's port of lwIP uses the MicroC/OS-II RTOS multi-threaded environment. Therefore, to use lwIP, you must base your C/C++ project on the MicroC/OS-II RTOS. Naturally, the Nios II processor system must also contain an Ethernet interface. At present, the Altera-provided lwIP driver supports only the SMSC lan91c111 MAC/PHY device, which is the same device that is provided on Nios development boards. The lwIP driver is interrupt-driven, so you must ensure that interrupts for the Ethernet component are connected.

Altera's port of lwIP is based on the hardware abstraction layer (HAL) generic Ethernet device model. By virtue of the generic device model, you can write a new driver to support any target Ethernet media access controller (MAC), and maintain the consistent HAL and sockets API to access the hardware.

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

This chapter discusses the details of how to use lwIP for the Nios II processor only.

The standard sockets interface is well-documented, and there are a number of books on the topic of programming with sockets. Two good texts are *Unix Network Programming* by Richard Stevens or *Internetworking with TCP/IP Volume 3* by Douglas Comer.

lwIP Files & Directories

You need not edit the source code to use lwIP in a C/C++ program using the Nios II IDE. Nonetheless, Altera provides the source code for your reference. By default the files are installed with the Nios II EDS in the <*Nios II EDS install path*>**/components/altera_lwip/UCOSII** directory.

The directory format of the stack tries to maintain the original open-source code as much as possible under the **UCOSII/src/downloads** directory to make upgrades smoother to a more recent version of lwIP. The **UCOSII/src/downloads/lwip-1.1.0** directory contains the original lwIP v1.1.0 source code; the **UCOSII/src/downloads/lwip4ucosii** directory contains the source code of the port for MicroC/OS-II.

Altera's port of lwIP is based on version 1.1.0 of the protocol stack, with wrappers placed around the code to integrate it to the HAL system library. More recent versions of lwIP are available, but newer versions have not been tested with the HAL system library wrappers.

Licensing

lwIP is an open-source TCP/IP protocol stack created by Adam Dunkels at the Computer and Networks Architectures (CNA) lab at the Swedish Institute of Computer Science (SICS), and is available under a modified BSD license. The lwIP project is hosted by Savannah at **http://savannah.nongnu.org/projects/lwip/**. Refer to the Savannah website for complete background information on lwIP and licensing details.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the copyright notice and disclaimer shown in the file <*lwIP component path*>**/UCOSII/src/downloads/lwIP-1.1.0/COPYING**.
- Redistributions in binary form must reproduce the copyright notice shown in the file <*lwIP component path*>**/UCOSII/src/downloads/lwIP-1.1.0/COPYING**.

Other TCP/IP Stack Providers

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

For up-to-date information on products available from third-party providers, visit the Nios II homepage at **www.altera.com/nios2**.

Using the lwIP Protocol Stack

This section discusses how to include the lwIP protocol stack in a Nios II program.

The primary interface to the lwIP protocol stack is the standard sockets interface. In addition to the sockets interface, you call the following functions to initialize the stack and drivers:

- `lwip_stack_init()`
- `lwip_devices_init()`

You must also provide the following simple functions that are called by HAL system code to set the MAC address and IP address:

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

Nios II System Requirements

To use lwIP, your Nios II system must meet the following requirements:

- The system hardware must include an Ethernet interface with interrupts enabled
- The system library must be based on MicroC/OS-II

The lwIP Tasks

The Altera-provided lwIP protocol uses the following two fundamental tasks. These tasks run continuously in addition to the tasks that your program creates.

1. The main task is used by the protocol stack. There is a task for receiving packets. The main function of this task blocks waiting for a message box. When a new packet arrives, an interrupt request (IRQ) is generated and an interrupt service routine (ISR) clears the IRQ and posts a message to the message box.

2. The new message then activates the receive task. This design allows the ISR to execute as quickly as possible, reducing the impact on system latency.

These tasks are started automatically when the initialization process succeeds. You set the task priorities, based on the criticality compared to other tasks in the system.

Initializing the Stack

To initialize the stack, call the function `lwip_stack_init()` before calling `OSStart` to start the MicroC/OS-II scheduler. The following code shows an example of a `main()`.

Example: Instantiating the lwIP Stack in main()

```
#include <includes.h>
#include <alt_lwip_dev.h>

int main ()
{
...
  lwip_stack_init(TCPIP_THREAD_PRIO, init_done_func, 0);
...
  OSStart();
...
  return 0;
}
```

lwip_stack_init()

`lwip_stack_init()` performs setup for the protocol stack. The prototype for `lwip_stack_init()` is:

```
void lwip_stack_init(int thread_prio,
        void (* init_done_func)(void *), void *arg)
```

`lwip_stack_init()` returns nothing and has the following parameters:

- `thread_prio`—the priority of the main TCP/IP thread
- `init_done_func`—a pointer to a function that is called once the stack is initialized
- `arg`—an argument to pass to `init_done_func()`. `arg` is usually set to zero.

init_done_func()

You must provide the function `init_done_func()`, which is called after the stack has been initialized. The `init_done_func()` function must call `lwip_devices_init()`, which initializes all the installed Ethernet device drivers, and then creates the receive task.

The prototype for `init_done_func()` is:

```
void init_done_func(void* arg)
```

The following code shows an example of the `tcpip_init_done()` function, which is an example of an implementation of an `init_done_func()` function.

Example: An implementation of init_done_func()

```
#include <stdio.h>
#include <lwip/sys.h>
```

```
#include <alt_lwip_dev.h>
#include <includes.h>
/*
 * This function is called once the IP stack is alive
 */
static void tcpip_init_done(void *arg)
{
  int temp;

  if (lwip_devices_init(ETHER_PRIO))
  {
    /* If initialization succeeds, start a user task */
    temp = sys_thread_new(user_thread_func,
                          NULL,
                          USER_THREAD_PRIO);
    if (!temp)
    {
      perror("Can't add the application threads
      OSTaskDel(OS_PRIO_SELF);
    }
  }
  else
  {
    /*
     * May not be able to add an Ethernet interface if:
     * 1. There is no Ethernet hardware
     * 2. Your hardware cannot initialize (e.g.
     * not connected to a network, or can’t get
     * a mac address)
     */
    perror("Can't initialize any interface. Closing down.\n");
    OSTaskDel(OS_PRIO_SELF);
  }

  return;
}
```

You must use `sys_thread_new()` to create any new task that talks to the IP stack using the sockets protocol.

For more information, see “Calling the Sockets Interface” on page 13–9.

lwip_devices_init()

`lwip_devices_init()` iterates through the list of all installed Ethernet device drivers defined in **system.h**, and registers each driver with the stack. `lwip_devices_init()` returns a non-zero value to indicate success. Upon success, the TCP/IP stack is available, and you can then create the task(s) for your program.

The prototype for lwip_devices_init() is:

```
int lwip_devices_init(int rx_thread_prio)
```

The parameter to this function is the priority of the receive thread. `lwip_devices_init()` calls the functions `get_mac_addr()` and `get_ip_address()`, which you must provide.

get_mac_addr() & get_ip_addr()

`get_mac_addr()` and `get_ip_addr()` are called by the lwIP system code during the devices initialization process. These functions are necessary for the lwIP system code to set the MAC and IP addresses for a particular device. By writing these functions yourself, your system has the flexibility to store the MAC address and IP address in an arbitrary location, rather than a fixed location hard-coded in the device driver. For example, some systems may store the MAC address in flash memory, while others may have the MAC address in on-chip embedded memory.

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

The prototype for `get_mac_addr()` is:

```
err_t gat_mac_addr(alt_lwip_dev* lwip_dev);
```

Inside the function, you must fill in the following fields of the `alt_lwip_dev` structure that define the MAC address:

- `unsigned char lwip_dev->netif->hwaddr_len`—the length of the MAC address, which should be 6
- `unsigned char lwIP_dev->netif->hwaddr[0-5]`—the MAC address of the device.

Your code can also verify the name of the device being initialized.

The prototype for `get_mac_addr()` is in the header file **UCOSII/inc/alt_lwip_dev.h**. The `netif` structure is defined in the **UCOSII/src/downloads/lwip-1.1.0/src/include/lwip/netif.h** file.

The following code shows an example implementation of `get_mac_addr()`. For demonstration purposes only, the MAC address is stored at address `0x7f0000` in this example.

Example: An implementation of get_mac_addr()

```
#include <alt_lwip_dev.h>
#include <lwip/netif.h>
#include <io.h>
err_t get_mac_addr(alt_lwip_dev* lwip_dev)
{
  err_t ret_code = ERR_IF;
  /*
   * The name here is the device name defined in system.h
   */
  if (!strcmp(lwip_dev->name, "/dev/lan91c111"))
  {
    /* Read the 6-byte MAC address from wherever it is stored */
    lwip_dev->netif->hwaddr[0] = IORD_8DIRECT(0x7f0000, 4);
    lwip_dev->netif->hwaddr[1] = IORD_8DIRECT(0x7f0000, 5);
    lwip_dev->netif->hwaddr[2] = IORD_8DIRECT(0x7f0000, 6);
    lwip_dev->netif->hwaddr[3] = IORD_8DIRECT(0x7f0000, 7);
    lwip_dev->netif->hwaddr[4] = IORD_8DIRECT(0x7f0000, 8);
    lwip_dev->netif->hwaddr[5] = IORD_8DIRECT(0x7f0000, 9);
    ret_code = ERR_OK;
  }
  return ret_code;
}
```

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

```
int get_ip_addr(alt_lwip_dev*   lwip_dev,
                struct ip_addr* ipaddr,
                struct ip_addr* netmask,
                struct ip_addr* gw,
                int*            use_dhcp);
```

To enable DHCP, include the line:

```
*use_dhcp = 1;
```

To assign a static IP address, include the lines:

```
IP4_ADDR(ipaddr, IPADDR0,IPADDR1,IPADDR2,IPADDR3);
IP4_ADDR(gw, GWADDR0,GWADDR1,GWADDR2,GWADDR3);
IP4_ADDR(netmask, MSKADDR0,MSKADDR1,MSKADDR2,MSKADDR3);
*use_dhcp = 0;
```

`IP_ADDR0-3` are the bytes 0-3 of the IP address. `GWADDR0-3` are the bytes of the gateway address. `MSKADDR0-3` are the bytes of the network mask.

The prototype for `get_ip_addr()` is in the header file **UCOSII/inc/alt_lwip_dev.h**.

The following code shows an example implementation of `get_ip_addr()` and shows a list of the necessary include files.

Example: An implementation of get_ip_addr()

```
#include <lwip/tcpip.h>
#include <alt_lwip_dev.h>
int get_ip_addr(alt_lwip_dev*   lwip_dev,
                struct ip_addr* ipaddr,
                struct ip_addr* netmask,
                struct ip_addr* gw,
                int*            use_dhcp)
{
  int ret_code = 0;
  /*
   * The name here is the device name defined in system.h
   */
  if (!strcmp(lwip_dev->name, "/dev/lan91c111"))
  {
#if LWIP_DHCP == 1
    *use_dhcp = 1;
#else
    /* Assign Static IP Addresses */
    IP4_ADDR(&ipaddr, 10,1 ,1 ,3);
    /* Assign the Default Gateway Address */
    IP4_ADDR(&gw, 10,1 , 1,254);
    /* Assign the Netmask */
    IP4_ADDR(&netmask, 255,255 ,255 ,0);
    *use_dhcp = 0;
#endif /* LWIP_DHCP */

    ret_code = 1;
  }
  return ret_code;
}
```

Calling the Sockets Interface

Once your Ethernet device has been initialized, the remainder of your program should use the sockets API to access the IP stack.

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

The prototype for `sys_thread_new()` is:

```
sys_thread_t sys_thread_new(void (* thread)(void *arg),
                            void *arg,
                             int  prio);
```

It is in **ucosII/src/downloads/lwIP-1.1.0/src/include/lwIP/sys.h**. You can include this as `#include "lwIP/sys.h"`.

You can find other details of the OS porting layer in the **sys_arch.c** file in the lwIP component directory, **UCOSII/src/downloads/lwip4ucosii/ucos-ii/**.

Configuring lwIP in the Nios II IDE

The lwIP protocol stack has many configuration options that are configured using `#define` directives in the file **lwipopts.h**. The Nios II integrated development environment (IDE) provides a graphical user interface (GUI) that enables you to configure lwIP options (i.e. modify the `#defines` in **lwipopts.h**) without editing source code. The most commonly accessed options are available through the GUI. However, there are some options that cannot be changed via the GUI, so you have to edit the **lwipopts.h** file manually.

The following sections describe the features that can be configured via the Nios II IDE. The GUI provides a default value for each feature. In general, these values provide a good starting point, and you can later fine-tune the values to meet the needs of your system.

Lightweight TCP/IP Stack General Settings

The ARP and IP protocols are always enabled. Table 13–1 shows the protocol options.

Table 13–1. Protocol Options

Option	Description
UDP	Enables and disables the user datagram protocol (UDP).
TCP	Enables and disables the transmission control protocol (TCP).

Table 13–2 shows the global options, which affect the overall behavior of the TCP/IP stack.

Table 13–2. Global Options

Option	Description
Use DHCP to automatically assign an IP address	Enables and disables DHCP. DHCP requires that the UDP protocol is enabled.
Enable statistics	When this option is turned on, the stack keeps counters of packets received, errors, etc. The counters are defined in a structure variable `lwip_stats` in the **UCOSII/src/downloads/lwIP-1.1.0/src/core** file. The structure definition is in **UCOSII/src/downloads/lwIP-1.1.0/src/include/lwIP/stats.h**.
Number of packet buffers	The number of buffers for the network driver to receive packets into.
Time to live	The number of seconds that a datagram can remain in the system before being discarded.
Maximum packet size	The maximum size of the packets on the network interface.

Table 13–2. Global Options

Option	Description
Stack size of the LWIP tasks (32-bit words)	The stack size of the lwIP tasks. For more information on the task, see "The lwIP Tasks" on page 13–4.
Default MAC interface	If the IP stack has more than one network interface, this parameter indicates which interface to use when sending packets to an IP address without a known route. See "Known Limitations" on page 13–12.

IP Options

If the forward IP packets option is turned on, when there is more than one network interface, and the IP stack for one interface receives packets not addressed to it, it forwards the packet out of the other interface.

ARP Options

The size of ARP table is the number of entries that can be stored in the ARP cache.

UDP Options

You can enter the maximum number of UDP sockets that the application uses.

TCP Options

Table 13–3 shows the TCP options.

Table 13–3. TCP Options

Option	Description
Max number of listening sockets	Maximum number of TCP sockets that can be listening for a client to connect.
Max number of active sockets	Maximum number of TCP sockets that the program uses, excluding listening sockets.
Max retransmissions	The maximum number of times that the TCP protocol tries to retransmit a packet which is not acknowledged.
Max retransmissions of SYN frames	The maximum number of times that the TCP protocol tries to retransmit a SYN packet, which is not acknowledged.
Max segment size	Maximum TCP segment size.
Max send buffer space	The maximum amount of data TCP buffers up for transmission.
Max window size	The maximum amount of data for each receiving socket that TCP buffers up

DHCP Options

You can specify that the ARP checks the assigned address is not in use, so once the DHCP protocol has assigned an IP address, it send out an APR packet to check that no-one else is using the assigned address.

Memory Options

Table 13–4 shows the memory options.

Table 13–4. Memory Options

Option	Description
Maximum number of buffers sent without copying	The maximum number of buffers that the stack attempts to transmit without copying. Only use this option for sending UDP packets and fragmented IP packets. This option maps onto the lwIP `#define memp_num_pbuf`.
Maximum number of packet buffers passed between the application and stack threads	The maximum number of buffers that can be passed between the application thread and the protocol stack thread (in either direction) at any one time.This option maps onto the lwIP `#define memp_num_netbuf`.
Maximum number of pending API calls from the application to the stack thread	The size of the message box that sends API calls from the application thread to the protocol thread. This option maps onto the lwIP `#define memp_num_api_msg`.
Maximum number of messages passed from the protocol stack thread to the application	The combination of API calls passed from the application thread to the stack thread, and packets being passed the other way. This option maps onto the lwIP `#define memp_num_tcpip_msg`.
TCP/IP Heap size	The size of the memory pool for copying buffers into temporary locations, which is not the total memory size. This option maps onto the lwIP `#define mem_size`.

Known Limitations

The following limitations of Altera’s current implementation of the lwIP stack are known:

- lwIP does not implement the shutdown socket call correctly. The shutdown call maps directly on to the close socket call
- Multiple network interfaces features are present in the code, but have not been tested.

Document Revision History

Table 13–5 shows the revision history for this document.

Table 13–5. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
November 2006, v6.1.0	Moved to Appendix section	NicheStack TCP/IP Stack - Nios II Edition is the preferred network package.
May 2006, v6.0.0	● Corrected error in `alt_irq_enable_all()` usage ● Added illustrations ● Revised text on optimizing ISRs ● Expanded and revised text discussing HAL exception handler code structure.	
October 2005, v5.1.0	● Updated references to HAL exception-handler assembly source files in section "HAL Exception Handler Files". ● Added description of `alt_irq_disable()` and `alt_irq_enable()` in section "ISRs".	
May 2005, v5.0.0	Added tightly-coupled memory information.	
December 2004 v1.2	Corrected the "Registering the Button PIO ISR with the HAL" example.	
September 2004 v1.1	● Changed examples. ● Added ISR performance data.	
May 2004 v1.0	First publication.	