Section II. Hardware Abstraction Layer

This section describes the Nios® II hardware abstraction layer (HAL). It includes the following chapters:

Introduction

This chapter introduces the hardware abstraction layer (HAL) for the Nios® II processor. This chapter contains the following sections:

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

The HAL serves as a device driver package for Nios II processor systems, providing a consistent interface to the peripherals in your system. Tight integration between SOPC Builder and the Nios II software development tools automates the construction of a HAL instance for your hardware. After SOPC Builder generates a hardware system, the Nios II integrated development environment (IDE) or the Nios II software build tools can generate a custom HAL board support package (BSP) to match the hardware configuration. Changes in the hardware configuration automatically propagate to the HAL device driver configuration, preventing changes in the underlying hardware from creating bugs.

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

Getting Started

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

In the Nios II software build tools command-line software build tools flow, you can create an example BSP based on the HAL using one of the `create-this-bsp` scripts supplied with the Nios II embedded design suite.

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

For information about creating a new project with the Nios II IDE, refer to the *Nios II Integrated Development Environment* chapter of the *Nios II Software Developer's Handbook*, or to the Nios II IDE help system. For information about creating a new project with the Nios II software build tools, refer to the *Introduction to the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*. More information about creating a Nios II IDE project is available in the Nios II IDE help system.

HAL Architecture

This section describes the fundamental elements of the HAL architecture.

Services

The HAL provides the following services:

- Integration with the newlib ANSI C standard library—Provides the familiar C standard library functions

- Device drivers—Provides access to each device in the system

- The HAL API—Provides a consistent, standard interface to HAL services, such as device access, interrupt handling, and alarm facilities

- System initialization—Performs initialization tasks for the processor and the runtime environment before `main()`

- Device initialization—Instantiates and initializes each device in the system before `main()` runs

Figure 5–1 shows the layers of a HAL-based system, from the hardware level up to a user program.

Figure 5–1. The Layers of a HAL-Based System
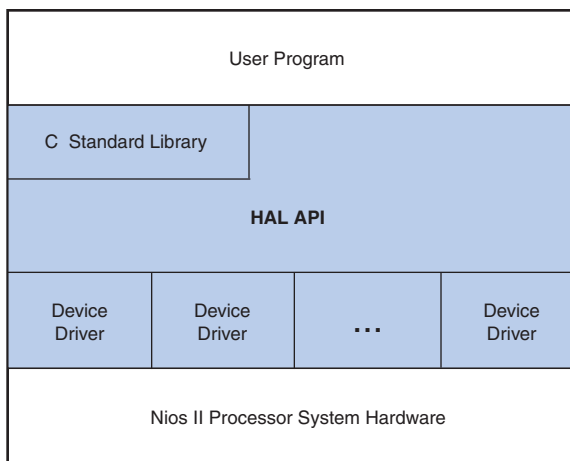
Applications versus Drivers

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

For further details about the HAL, refer to the following chapters:

■ The *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook* describes how to take advantage of the HAL to write programs without considering the underlying hardware.

■ The *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook* describes how to communicate directly with hardware and how to make hardware resources available with the HAL API.

Generic Device Models

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

Device Model Classes

The HAL provides models for the following classes of devices:

■ Character-mode devices—Hardware peripherals that send and/or receive characters serially, such as a UART.

■ Timer devices—Hardware peripherals that count clock ticks and can generate periodic interrupt requests.

■ File subsystems—A mechanism for accessing files stored in physical device(s). Depending on the internal implementation, the file subsystem driver might access the underlying device(s) directly or use a separate device driver. For example, you can write a flash file subsystem driver that accesses flash using the HAL API for flash memory devices.

■ Ethernet devices—Devices that provide access to an Ethernet connection for a networking stack such as the Altera-provided NicheStack® TCP/IP Stack - Nios II Edition. You need a networking stack to use an ethernet device.

■ Direct memory access (DMA) devices—Peripherals that perform bulk data transactions from a data source to a destination. Sources and destinations can be memory or another device, such as an Ethernet connection.

■ Flash memory devices—Nonvolatile memory devices that use a special programming protocol to store data.

Benefits to Application Developers

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

Benefits to Device Driver Developers

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

C Standard Library—Newlib

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

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

Supported Hardware

This section summarizes Nios II HAL support for Nios II hardware.

Nios II Processor Core Support

The Nios II HAL supports all available Nios II processor core implementations.

Supported Peripherals

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

■ Character mode devices

    ■ UART core

    ■ JTAG UART core

    ■ LCD 16207 display controller

- Flash memory devices

    - Common flash interface compliant flash chips

    - Altera's erasable programmable configurable serial (EPCS) serial configuration device controller

- File subsystems

    - Altera host based file system

    - Altera read-only zip file system

- Timer devices

    - Timer core

- DMA devices

    - DMA controller core

    - Scatter-gather DMA controller core

- Ethernet devices

    - Triple Speed Ethernet MegaCore® function

    - LAN91C111 Ethernet MAC/PHY Controller

The LAN91C111 and Triple Speed Ethernet components require the MicroC/OS-II runtime environment.

For more information, refer to the *Ethernet and the NicheStack TCP/IP Stack - Nios II Edition* chapter of the *Nios II Software Developer's Handbook.* Third-party vendors offer additional peripherals not listed here. For a list of other peripherals available for the Nios II processor, visit the Embedded Software page of the Altera website.

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

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

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

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

MPU Support

The HAL does not include explicit support for the optional memory protection unit (MPU) hardware. However, it does support an advanced exception handler that can handle Nios II MPU exceptions.

For details about handling MPU and other advanced exceptions, refer to the *Exception Handling* chapter of the *Nios II Software Developer's Handbook*. For details about the MPU hardware implementation, refer to the *Programming Model* chapter of the *Nios II Processor Reference Handbook*.

MMU Support

The HAL does not support the optional memory management unit (MMU) hardware. To use the MMU, you need to implement a full-featured operating system.

For details about the Nios II MMU, refer to the *Programming Model* chapter of the *Nios II Processor Reference Handbook*.

Referenced Documents

This chapter references the following documents:

- *Nios II Integrated Development Environment* chapter of the *Nios II Software Developer's Handbook*

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

- *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*

- *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*

- *Exception Handling* chapter of the *Nios II Software Developer's Handbook*

- *Ethernet and the NicheStack TCP/IP Stack - Nios II Edition* chapter of the *Nios II Software Developer's Handbook*

- *Programming Model* chapter of the *Nios II Processor Reference Handbook*

- *Volume 5: Embedded Peripherals* of the *Quartus II Handbook*

- The Embedded Software page of the Altera website

Document Revision History

Table 5–1 shows the revision history for this document.

Table 5–1. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	■ Reorganized and updated information and terminology to clarify role of Nios II software build tools. ■ Corrected minor typographical errors.	
May 2008 v8.0.0	No change from previous release.	
October 2007 v7.2.0	No change from previous release.	
May 2007 v7.1.0	■ Scatter-gather DMA core ■ Triple-speed Ethernet MAC ■ Refer to HAL generation with Nios II software build tools. ■ Added table of contents to Introduction section. ■ Added Referenced Documents section.	■ Scatter-gather DMA core ■ Triple-speed Ethernet MAC ■ Nios II software build tools
March 2007 v7.0.0	No change from previous release.	
November 2006 v6.1.0	NicheStack TCP/IP Stack - Nios II Edition	
May 2006 v6.0.0	No change from previous release.	
October 2005 v5.1.0	No change from previous release.	
May 2005 v5.0.0	No change from previous release.	
May 2004 v1.0	Initial Release.	

Introduction

This chapter discusses how to develop programs for the Nios® II processor based on the Altera® hardware abstraction layer (HAL). This chapter contains the following sections:

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

This document does not cover the ANSI C standard library. An excellent reference is *The C Programming Language, Second Edition*, by Brian Kernighan and Dennis M. Ritchie (Prentice-Hall).

Nios II Development Flows

The Nios II Embedded Design Suite (EDS) provides two distinct development flows for creating Nios II programs. You can work entirely in the Nios II integrated development environment (IDE), or you can use the Nios II software build tools in command-line and scripted environments.

HAL BSP Settings

Every Nios II board support package (BSP) has settings that determine the BSP's characteristics. For example, HAL BSPs have settings to identify the hardware components associated with standard devices such as stdout. Defining and manipulating BSP settings is an important part of Nios II project creation.

How you manipulate BSP settings depends on the development flow you use. In the command-line software build tools flow, you manipulate BSP settings with command-line options or Tcl scripts.

For details about how to control BSP settings, refer to one or more of the following documents:

■ For the Nios II IDE development flow, refer to the Nios II IDE help system.

■ For the Nios II command-line software build tools development flow, refer to the *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

For detailed descriptions of available BSP settings, refer to the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Many HAL settings are reflected in the **system.h** file, which provides a helpful reference for details about your BSP. For information about **system.h**, refer to "The system.h System Description File" on page 6–4.

☞ Do not edit **system.h**. The Nios II EDS provides tools to manipulate system settings.

The Nios II Project Structure

The creation and management of software projects based on the HAL is integrated tightly with the Nios II software build tools. This section discusses the Nios II projects as a basis for understanding the HAL.

Figure 6–1 shows the blocks of a Nios II program with emphasis on how the HAL BSP fits in. The label for each block describes what or who generated that block, and an arrow points to each block's dependency.

Figure 6–1. The Nios II HAL Project Structure
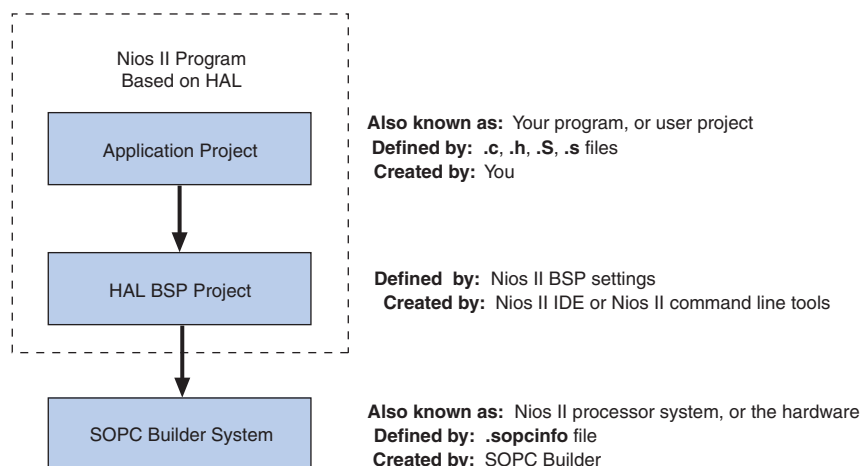
Every HAL-based Nios II program consists of two Nios II projects, as shown in Figure 6–1. Your application-specific code is contained in one project (the user application project), and it depends on a separate BSP project (the HAL BSP).

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

In the Nios II IDE flow, the Nios II IDE creates the HAL BSP project when you create your application project. In the Nios II command-line software build tools flow, you create the BSP using `nios2-bsp` or a related tool.

The HAL BSP project contains all information needed to interface your program to the hardware. The HAL drivers relevant to your SOPC Builder system are incorporated in the BSP project.

The BSP project depends on the SOPC Builder system, defined by an SOPC Builder system (**.sopcinfo**) file.The Nios II software build tools can keep your BSP up-to-date with the SOPC Builder system. This project dependency structure isolates your program from changes to the underlying hardware, and you can develop and debug code without concern about whether your program matches the target hardware.

When you rebuild a BSP, the Nios II software build tools can update it to match the hardware. You control whether and when these updates occur.

For details about how the software build tools keep your BSP up-to-date with your hardware system, refer to the *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

In summary, when your program is based on a HAL BSP, you can always keep it synchronized with the target hardware by simply rebuilding your software.

The system.h System Description File

The **system.h** file provides a complete software description of the Nios II system hardware. Not all information in **system.h** is useful to you as a programmer, and it is rarely necessary to include it explicitly in your C source files. Nonetheless, **system.h** holds the answer to the 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 interrupt request (IRQ) priority (if any)

■ A symbolic name for the peripheral

Both Nios II development flows generate the **system.h** file for HAL BSP projects. The contents of **system.h** depend on both the hardware configuration and the HAL BSP properties.

☞ Do not edit **system.h**. Both development flows provide tools to manipulate system settings.

For details about how to control BSP settings, refer to "HAL BSP Settings" on page 6–2.

The code in Example 6–1 from a **system.h** file shows some of the hardware configuration options this file defines.

Example 6–1. Excerpts from a system.h File

```
/*
 * sys_clk_timer configuration
 *
 */

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

/*
 * jtag_uart configuration
 *
 */

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

Data Widths and the HAL Type Definitions

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

The header file **alt_types.h** defines the HAL type definitions; Table 6–1 shows the HAL type definitions.

Table 6–1. The HAL Type Definitions

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

Table 6–2 shows the data widths that the Altera-provided GNU tool-chain uses.

Table 6–2. GNU Toolchain Data Widths

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

UNIX-Style Interface

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

The following list contains all of the available UNIX-style functions:

- _exit()
- close()
- fstat()
- getpid()
- gettimeofday()

- `ioctl()`
- `isatty()`
- `kill()`
- `lseek()`
- `open()`
- `read()`
- `sbrk()`
- `settimeofday()`
- `stat()`
- `usleep()`
- `wait()`
- `write()`

The most commonly used functions are those that relate to file I/O. Refer to "File System" on page 6–6.

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

File System

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

For an example, refer to the *Read-Only Zip File System* chapter of the *Nios II Software Developer's Handbook*.

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

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

- `close()`
- `fstat()`
- `ioctl()`
- `isatty()`
- `lseek()`
- `open()`
- `read()`
- `stat()`
- `write()`

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

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

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

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

The code in Example 6–2 reads characters from a read-only zip file subsystem **rozipfs** that is registered as a node in the HAL file system. The standard header files stdio.h, stddef.h, and stdlib.h are installed with the HAL.

Example 6–2. Reading Characters from a File Subsystem

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

#define BUF_SIZE (10)

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

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

  fread (buffer, BUF_SIZE, 1, fp);

  fclose (fp);

  return 0;
}
```

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

Using Character-Mode Devices

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

Standard Input, Standard Output and Standard Error

Using standard input (`stdin`), standard output (`stdout`), and standard error (`stderr`) is the easiest way to implement simple console I/O. The HAL manages `stdin`, `stdout`, and `stderr` behind the scenes, which allows you to send and receive characters through these channels without explicitly managing file descriptors. For example, the HAL directs the output of `printf()` to standard out, and `perror()` to standard error. You associate each channel to a specific hardware device by manipulating BSP settings.

The code in Example 6–3 shows the classic Hello World program. This program sends characters to whatever device is associated with `stdout` when the program is compiled.

Example 6–3. Hello World

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

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

General Access to Character Mode Devices

Accessing a character-mode device other than `stdin`, `stdout`, or `stderr` is as easy as opening and writing to a file. The code in Example 6–4 writes a message to a UART called `uart1`.

Example 6–4. Writing Characters to a UART

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

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

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

C++ Streams

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

/dev/null

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

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

Lightweight Character-Mode I/O

The HAL offers several methods of reducing the code footprint of character-mode device drivers. For details, refer to "Reducing Code Footprint" on page 6–29.

Altera Logging Functions

The Altera logging functions provide a separate channel for sending logging and debugging information to a character-mode device, supplementing `stdout` and `stderr`. The Altera logging information can be printed in response to several conditions. Altera logging can be enabled and disabled independently of any normal `stdio` output, making it a powerful debugging tool.

When Altera logging is enabled, your software can print extra messages to a specified port with HAL function calls. The logging port, specified in the BSP, can be a UART or a JTAG UART device. In its default configuration, Altera logging prints out boot messages, which trace each step of the boot process.

☞ Avoid setting the Altera logging device to the device used for `stdout` or `stderr`. If Altera logging output is sent to `stdout` or `stderr`, the logging output might appear interleaved with the `stdout` or `stderr` output

Several logging options are available, controlled by C preprocessor symbols. You can also choose to add custom logging messages.

☞ Altera logging changes system behavior. The logging implementation is designed to be as simple as possible, loading characters directly to the transmit register. It can have a negative impact on software performance.

Altera logging functions are conditionally compiled. When logging is disabled, it has no impact on code footprint or performance.

☞ The Altera reduced device drivers do not support Altera logging.

Enabling Altera Logging

The Nios II software build tools have a setting to enable Altera logging. The setting is called **hal.log_port**. It is similar to **hal.stdout**, **hal.stdin**, and **hal.stderr**. To enable Altera logging, you set **hal.log_port** to a JTAG UART or a UART device. The setting allows the HAL to send log messages to the specified device when a logging macro is invoked.

When Altera logging is enabled, the Nios II software build tools define ALT_LOG_ENABLE in **public.mk** to enable log messages. The build tools also set the ALT_LOG_PORT_TYPE and ALT_LOG_PORT_BASE values in **system.h** to point to the specified device.

When Altera logging is enabled without special options, the HAL prints out boot messages to the selected port. For typical software that uses the standard **alt_main.c** (such as the Hello World software example), the messages appear as in Example 6–5.

Example 6–5. Default Boot Logging Output

```
[crt0.S] Inst & Data Cache Initialized.
[crt0.S] Setting up stack and global pointers.
[crt0.S] Clearing BSS
[crt0.S] Calling alt_main.
[alt_main.c] Entering alt_main, calling alt_irq_init.
[alt_main.c] Done alt_irq_init, calling alt_os_init.
[alt_main.c] Done OS Init, calling alt_sem_create.
[alt_main.c] Calling alt_sys_init.
[alt_main.c] Done alt_sys_init.  Redirecting IO.
[alt_main.c] Calling C++ constructors.
[alt_main.c] Calling main.
[alt_exit.c] Entering _exit() function.
[alt_exit.c] Exit code from main was 0.
[alt_exit.c] Calling ALT_OS_STOP().
[alt_exit.c] Calling ALT_SIM_HALT().
[alt_exit.c] Spinning forever.
```

☞ A write operation to the Altera logging device stalls in ALT_LOG_PRINTF() until the characters are read from the Altera logging device's output buffer. To ensure that the Nios II application completes initialization, run the nios2-terminal command from the Nios II command shell to accept the Altera logging output.

Extra Logging Options

In addition to the default boot messages, logging options are incorporated in Altera logging. Each option is controlled by a C preprocessor symbol. The details of each option are outlined in Table 6–3.

Table 6–3. Altera Logging Options (Part 1 of 2)

Name		Description
System clock log	Purpose	Prints out a message from the system clock interrupt handler at a specified interval. This indicates that the system is still running. The default interval is every 1 second.
	Preprocessor symbol	`ALT_LOG_SYS_CLK_ON_FLAG_SETTING`
	Modifiers	The system clock log has two modifiers, providing two different ways to specify the logging interval. ■ `ALT_LOG_SYS_CLK_INTERVAL`—Specifies the logging interval in system clock ticks. The default is *<clock ticks per second>*, that is, one second. ■ `ALT_LOG_SYS_CLK_INTERVAL_MULTIPLIER`—Specifies the logging interval in seconds. The default is 1. When you modify `ALT_LOG_SYS_CLK_INTERVAL_MULTIPLIER`, `ALT_LOG_SYS_CLK_INTERVAL` is recalculated.

	Sample Output	`System Clock On 0`
		
		 `System Clock On 1`

Write echo	Purpose	Every time `alt_write()` is called (normally, whenever characters are sent to `stdout`), the first *<n>* characters are echoed to a logging message. The message starts with the string `"Write Echo:"`. *<n>* is specified with `ALT_LOG_WRITE_ECHO_LEN`. The default is 15 characters.
	Preprocessor symbol	`ALT_LOG_WRITE_ON_FLAG_SETTING`
	Modifiers	`ALT_LOG_WRITE_ECHO_LEN`—Number of characters to echo. Default is 15.
	Sample Output	`Write Echo: Hello from Nio`
JTAG startup log	Purpose	At JTAG UART driver initialization, print out a line with the number of characters in the software transmit buffer followed by the JTAG UART control register contents. The number of characters, prefaced by the string `"SW CirBuf"`, might be negative, because it is computed as (*<tail_pointer>* - *<head_pointer>*) on a circular buffer. For more information about the JTAG UART control register fields, refer to the *Off-Chip Interface Peripherals* section in *Volume 5: Embedded Peripherals* of the *Quartus II Handbook*.
	Preprocessor symbol	`ALT_LOG_JTAG_UART_STARTUP_INFO_ON_FLAG_SETTING`
	Modifiers	None
	Sample Output	`JTAG Startup Info: SW CirBuf = 0, HW FIFO wspace=64` `AC=0 WI=0 RI=0 WE=0 RE=1`

Table 6–3. Altera Logging Options (Part 2 of 2)

Name	Description	
JTAG interval log	Purpose	Creates an alarm object to print out the same JTAG UART information as the JTAG startup log, but at a repeated interval. Default interval is 0.1 second, or 10 messages a second.
	Preprocessor symbol	`ALT_LOG_JTAG_UART_ALARM_ON_FLAG_SETTING`
	Modifiers	The JTAG interval log has two modifiers, providing two different ways to specify the logging interval. ■ `ALT_LOG_JTAG_UART_TICKS`—Logging interval in ticks. Default is *<ticks_per_second>* / 10. ■ `ALT_LOG_JTAG_UART_TICKS_DIVISOR`—Specifies the number of logs per second. The default is 10. When you modify `ALT_LOG_JTAG_UART_TICKS_DIVISOR`, `ALT_LOG_JTAG_UART_TICKS` is recalculated.
	Sample Output	`JTAG Alarm: SW CirBuf = 0, HW FIFO wspace=45 AC=0 WI=0` `RI=0 WE=0 RE=1`
JTAG interrupt service routine (ISR) log	Purpose	Prints out a message every time the JTAG UART near-empty interrupt triggers. Message contains the same JTAG UART information as in the JTAG startup log.
	Preprocessor symbol	`ALT_LOG_JTAG_UART_ISR_ON_FLAG_SETTING`
	Modifiers	None
	Sample Output	`JTAG IRQ: SW CirBuf = -20, HW FIFO wspace=64 AC=0 WI=1` `RI=0 WE=1 RE=1`
Boot log	Purpose	Prints out messages tracing the software boot process. The boot log is turned on by default when Altera logging is enabled.
	Preprocessor symbol	`ALT_LOG_BOOT_ON_FLAG_SETTING`
	Modifiers	None
	Sample Output	Refer to "Enabling Altera Logging" on page 6–10.

Setting a preprocessor flag to 1 enables the corresponding option. Any value other than 1 disables the option.

Several options have modifiers, which are additional preprocessor symbols controlling details of how the options work. For example, the system clock log's modifiers control the logging interval. Option modifiers are also listed in Table 6–3. An option's modifiers are meaningful only when the option is enabled.

Logging Levels

An additional preprocessor symbol, ALT_LOG_FLAGS, can be set to provide some grouping for the extra logging options. ALT_LOG_FLAGS implements logging levels based on performance impact. With higher logging levels, the Altera logging options take more processor time. ALT_LOG_FLAGS levels are defined in Table 6–4

Table 6–4. Altera Logging Levels

Logging Level	Logging
0	Boot log (default)
1	Level 0 plus system clock log and JTAG startup log
2	Level 1 plus JTAG interval log and write echo
3	Level 2 plus JTAG ISR log
-1	Silent mode—No Altera logging

Note to Table 6–4:

(1) You can use logging level -1 to turn off logging without changing the program footprint. The logging code is still present in your executable image, as determined by other logging options chosen. This is useful when you wish to switch the log output on or off without disturbing the memory map.

Because each logging option is controlled by an independent preprocessor symbol, individual options in the logging levels can be overridden.

Example: Creating a BSP with Logging

Example 6–6 creates a HAL BSP with Altera logging enabled and the following options in addition to the default boot log:

■ System clock log

■ JTAG startup log

■ JTAG interval log, logging twice a second

■ No write echo

Example 6–6. BSP With Logging

```
nios2-bsp hal my_bsp ../my_hardware.sopcinfo \
  --set hal.log_port uart1 \
  --set hal.make.bsp_cflags_user_flags \
  -DALT_LOG_FLAGS=2 \
  -DALT_LOG_WRITE_ON_FLAG_SETTING=0 \
  -DALT_LOG_JTAG_UART_TICKS_DIVISOR=2↵
```

The -DALT_LOG_FLAGS=2 argument adds -DALT_LOG_FLAGS=2 to the ALT_CPP_FLAGS make variable in **public.mk**.

Custom Logging Messages

You can add custom messages that are sent to the Altera logging device. To define a custom message, include the header file **alt_log_printf.h** in your C source file as follows:

```
#include "sys/alt_log_printf.h"
```

Then use the following macro function:

```
ALT_LOG_PRINTF(const char *format, ...)
```

This C preprocessor macro is a pared-down version of `printf()`. The `format` argument supports most `printf()` options. It supports `%c`, `%d`, `%I`, `%o`, `%s`, `%u`, `%x`, and `%X`, as well as some precision and spacing modifiers, such as `%-9.3o`. It does not support floating point formats, such as `%f` or `%g`. This function is not compiled if Altera logging is not enabled.

If you want your custom logging message be controlled by Altera logging preprocessor options, use the appropriate Altera logging option preprocessor flags from Table 6–4, or Table 6–3 on page 6–11. Example 6–7 illustrates two ways to implement logging options with custom logging messages.

Example 6–7. Using Preprocessor Flags

```
/* The following example prints "Level 2 logging message" if
   logging is set to level 2 or higher */
#if ( ALT_LOG_FLAGS >= 2 )
    ALT_LOG_PRINTF ( "Level 2 logging message" );
#endif

/* The following example prints "Boot logging message" if boot logging
   is turned on */
#if ( ALT_LOG_BOOT_ON_FLAG_SETTING == 1)
    ALT_LOG_PRINTF ( "Boot logging message" );
#endif
```

Altera Logging Files

Table 6–5 lists HAL source files which implement Altera logging functions.

Table 6–5. HAL Implementation Files for Altera Logging

Location *(1)*	File Name
components/altera_hal/HAL/inc/sys/	**alt_log_printf.h**
components/altera_hal/HAL/src/	**alt_log_printf.c**
components/altera_nios2/HAL/src/	**alt_log_macro.S**

Note to Table 6–5:

(1) All file locations are relative to `$SOPC_KIT_NIOS2`.

Table 6–6 lists HAL source files which use Altera logging functions. These files implement the logging options listed in table Table 6–3 on page 6–11. They also serve as examples of logging usage.

Table 6–6. HAL Example Files for Altera Logging

Location *(1)*	File Name
components/altera_avalon_jtag_uart/HAL/src/	**altera_avalon_jtag_uart.c**
components/altera_avalon_timer/HAL/src/	**altera_avalon_timer_sc.c**
components/altera_hal/HAL/src/	**alt_exit.c**
components/altera_hal/HAL/src/	**alt_main.c**
components/altera_hal/HAL/src/	**alt_write.c**
components/altera_nios2/HAL/src/	**crt0.S**

Note to Table 6–6:

(1) All file locations are relative to `$SOPC_KIT_NIOS2`.

Using File Subsystems

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

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

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

For more information about the use of 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 EDS *<version>* Documentation**.

Using Timer Devices

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

The HAL API provides two types of timer device drivers:

■ System clock driver—This type of driver supports alarms, such as you would use in a scheduler.

■ Timestamp driver—This driver supports high-resolution time measurement.

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

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

System Clock Driver

The HAL system clock driver provides a periodic heartbeat, causing the system clock to increment on each beat. Software can use the system clock facilities to execute functions at specified times, and to obtain timing information. You select a specific hardware timer peripheral as the system clock device by manipulating BSP settings.

For details about how to control BSP settings, refer to "HAL BSP Settings" on page 6–2.

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

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

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

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

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

Alarms

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

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

The function `callback()` is called after `nticks` have elapsed. The input argument `context` is passed as the input argument to `callback()` when the call occurs. The HAL does not use the `context` parameter. It is only used as a parameter to the `callback()` function.

Your code must allocate the `alt_alarm` structure, pointed to by the input argument `alarm`. This data structure must have a lifetime that is at least as long as that of the alarm. The best way to allocate this structure is to declare it as a static or global. `alt_alarm_start()` initializes *alarm.

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

One alarm is created for each call to alt_alarm_start(). Multiple alarms can run simultaneously.

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

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

The code fragment in Example 6–8 demonstrates registering an alarm for a periodic callback every second.

Example 6–8. Using a Periodic Alarm Callback Function

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

/*
* The callback function.
*/

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

...

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

...

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

Timestamp Driver

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

You specify a hardware timer peripheral as the timestamp device by manipulating BSP settings. The Altera-provided timestamp driver uses the timer that you specify.

If a timestamp driver is present, the following functions are available:

■ `alt_timestamp_start()`

■ `alt_timestamp()`

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

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

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

The code fragment in Example 6–9 shows how you can use the timestamp facility to measure code execution time.

Example 6–9. Using the Timestamp to Measure Code Execution Time

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

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

  if (alt_timestamp_start() < 0)
  {
    printf ("No timestamp device available\n");
  }
  else
  {
    time1 = alt_timestamp();
    func1(); /* first function to monitor */
    time2 = alt_timestamp();
    func2(); /* second function to monitor */
    time3 = alt_timestamp();

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

Using Flash Devices

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

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

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

■ Simple flash access—Functions that write buffers to flash and read them back at the block level. In writing, if the buffer is less than a full block, these functions erase preexisting flash data above and below the newly written data.

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

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

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

Simple Flash Access

This interface consists of the functions `alt_flash_open_dev()`, `alt_write_flash()`, `alt_read_flash()`, and `alt_flash_close_dev()`. The code "Using the Simple Flash API Functions" on page 6–21 shows the use of all of these functions in one code example. You open a flash device by calling `alt_flash_open_dev()`, which returns a file handle to a flash device. This function takes a single argument that is the name of the flash device, as defined in **system.h**.

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

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

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

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

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

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

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

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

The code in Example 6–10 shows the use of simple flash API functions to access a flash device named **/dev/ext_flash**, as defined in **system.h**.

Block Erasure or Corruption

Generally, flash memory is divided into blocks. `alt_write_flash()` might need to erase the contents of a block before it can write data to it. In this case, it makes no attempt to preserve the existing contents of the block. This action can lead to unexpected data corruption (erasure), if you are performing writes that do not fall on block boundaries. If you wish to preserve existing flash memory contents, use the fine-grained flash functions. These are discussed in the following section.

Table 6–7 on page 6–22 shows how you can cause unexpected data corruption by writing using the simple flash-access functions. Table 6–7 shows the example of an 8 Kbyte flash memory comprising two 4 Kbyte blocks. First write 5 Kbytes of all `0xAA` to 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 (that is, `0xFF`). Then the second write begins, but before writing to the second block, the block is erased. At this point, t(3), the flash contains 4 Kbyte of `0xAA` and 4 Kbyte of `0xFF`. After the second write finishes, at time t(4), the 2 Kbyte of `0xFF` at address `0x1000` is corrupted.

Fine-Grained Flash Access

Three additional functions provide complete control for writing flash contents at the highest granularity:

■ `alt_get_flash_info()`

■ `alt_erase_flash_block()`

■ `alt_write_flash_block()`

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

Example 6–10. Using the Simple Flash API Functions

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

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

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

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

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

alt_get_flash_info() gets the number of erase regions, the number of erase blocks in each region, and the size of each erase block. The function prototype is as follows:

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

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

Table 6–7. Example of Writing Flash and Causing Unexpected Data Corruption

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

Note to Table 6–7:

(1) Unintentionally cleared to FF during erasure for second write.

The `flash_region` structure is defined in **sys/alt_flash_types.h**. The data structure is defined as follows:

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

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

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

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

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

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

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

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

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

The code in Example 6–11 on page 6–23 demonstrates the use of the fine-grained flash access functions.

Example 6–11. Using the Fine-Grained Flash Access API Functions

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

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

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

  fd = alt_flash_open_dev(EXT_FLASH_NAME);

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

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

Using DMA Devices

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

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

Figure 6–2 shows the three basic types of DMA transactions. Copying data from memory to memory involves both receive and transmit DMA channels simultaneously.

Figure 6–2. Three Basic Types of DMA Transactions
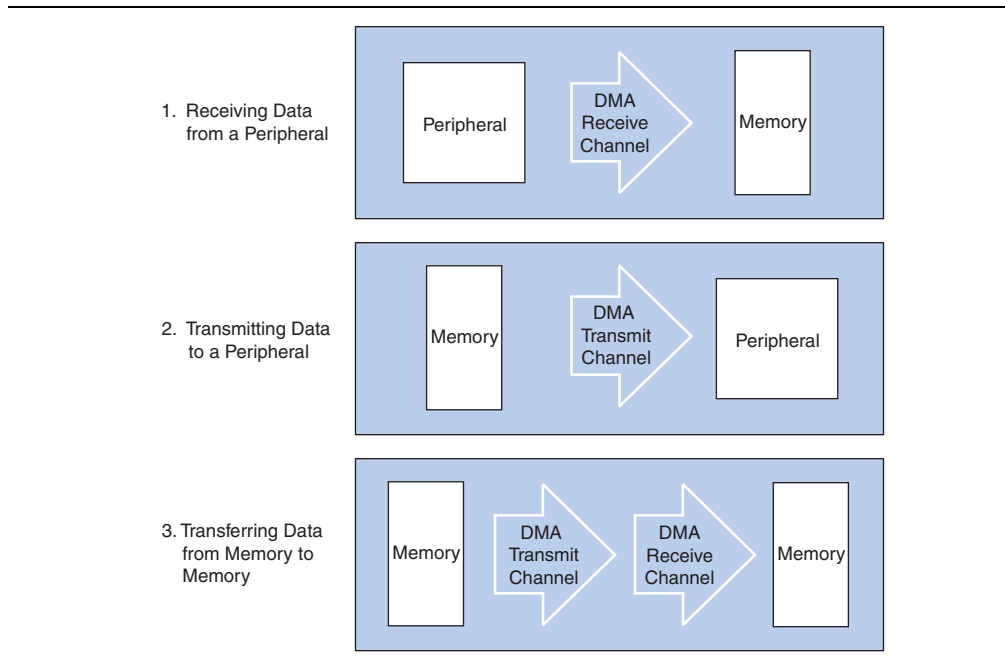
The API for access to DMA devices is defined in **sys/alt_dma.h**.

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

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

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

DMA Transmit Channels

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

The code in Example 6–12 shows how to obtain a handle for a DMA transmit device `dma_0`.

Example 6–12. Obtaining a File Handle for a DMA Device

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

int main (void)
{
  alt_dma_txchan tx;

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

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

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

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

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

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

☞ If you are using the Avalon Memory-Mapped® (Avalon-MM) DMA device to transmit to hardware (not memory-to-memory transfer), call the `alt_dma_txchan_ioctl()` function with the request argument set to `ALT_DMA_TX_ONLY_ON`.

👣 For further information, refer to the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

DMA Receive Channels

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

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

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

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

Certain errors can prevent the DMA transfer from completing. Typically this is caused by a catastrophic hardware failure; for example, if a component involved in the transfer fails to respond to a read or write request. If the DMA transfer does not complete (that is, less than `length` bytes are transferred), function `done()` is never called.

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

☞ If you are using the Avalon-MM DMA device to receive from hardware (not memory-to-memory transfer), call the `alt_dma_rxchan_ioctl()` function with the request argument set to `ALT_DMA_RX_ONLY_ON`.

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

👣 For further details, refer to the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

The code in Example 6–13 shows a complete example application that posts a DMA receive request, and blocks in `main()` until the transaction completes.

Example 6–13. A DMA Transaction on a Receive Channel

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

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

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

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

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

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

Memory-to-Memory DMA Transactions

Copying data from one memory buffer to another buffer involves both receive and transmit DMA drivers. The code in Example 6–14 shows the process of queuing up a receive request followed by a transmit request to achieve a memory-to-memory DMA transaction.

Example 6–14. Copying Data from Memory to Memory (Part 1 of 2)

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

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

static volatile int rx_done = 0;

/*
* Callback function that obtains notification that the data
* is received.
*/

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

/*
*
*/

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

  alt_dma_txchan txchan;
  alt_dma_rxchan rxchan;

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

  /* Create the transmit channel */

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

  /* Create the receive channel */

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

  /* 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);
  }

/* Continued... */
```

Example 6–14. Copying Data from Memory to Memory (Part 2 of 2)

```
/* 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 a concern for embedded systems developers, because there is a cost associated with the memory device that stores code. The ability to control and reduce code size is important in controlling this cost.

The HAL environment is designed to include only those features that you request, minimizing the total code footprint. If your Nios II hardware system contains exactly the peripherals used by your program, the HAL contains only the drivers necessary to control the hardware.

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

Implementing the options in the following sections entails making changes to BSP settings. For detailed information about manipulating BSP settings, refer to "HAL BSP Settings" on page 6–2.

Enable Compiler Optimizations

To enable compiler optimizations, use the -O3 compiler optimization level for the nios2-elf-gcc compiler. You can specify this command-line option through a BSP setting.

With this option turned on, the Nios II compiler compiles code with the maximum optimization available, for both size and speed.

☞ You must set this option for both the BSP and the application project.

Use Reduced Device Drivers

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

By default the HAL always uses the fast driver variants. You can select the reduced device driver for all hardware components, or for an individual component, through HAL BSP settings.

Table 6–8 lists the Altera Nios II peripherals that currently provide small footprint drivers. The small footprint option might also affect other peripherals. Refer to each peripheral's data sheet for complete details of its driver's small footprint behavior.

Table 6–8. Altera Peripherals Offering Small Footprint Drivers

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

Reduce the File Descriptor Pool

The file descriptors that access character mode devices and files are allocated from a file descriptor pool. You can change the size of the file descriptor pool through a BSP setting. The default is 32.

Use /dev/null

At boot time, standard input, standard output, and standard error are all directed towards the null device, that is, **/dev/null**. This direction ensures that calls to `printf()` during driver initialization do nothing and therefore are harmless. After all drivers are installed, these streams are redirected to the channels configured in the HAL. The footprint of the code that performs this redirection is small, but you can eliminate it entirely by selecting `null` for `stdin`, `stdout`, and `stderr`. This selection assumes that you want to discard all data transmitted on standard out or standard error, and your program never receives input through `stdin`. You can control the assignment of `stdin`, `stdout`, and `stderr` channels by manipulating BSP settings.

Use a Smaller File I/O Library

Use the Small newlib C Library

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

You can select the small newlib library through BSP settings. Table 6–9 summarizes the limitations of the Nios II small newlib C library implementation.

Table 6–9. Limitations of the Nios II Small newlib C Library (Part 1 of 2)

Limitation	Functions Affected

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

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

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

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

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

Table 6–9. Limitations of the Nios II Small newlib C Library (Part 2 of 2)

Limitation	Functions Affected

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

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

No support for locale.	`setlocale()` `localeconv()`
No support for C++, because the functions listed in this table are not supported.	

Note to Table 6–9:

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

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

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

☞   The Nios II implementation of the small newlib C library differs slightly from GCC. Table 6–9 provides details about the differences.

Use UNIX-Style File I/O

If you need to reduce the code footprint further, you can omit the newlib C library, and use the UNIX-style API. For details, refer to "UNIX-Style Interface" on page 6–5.

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

Emulate ANSI C Functions

If you choose to omit the full implementation of newlib, but you need a limited number of ANSI-style functions, you can implement them easily using UNIX-style functions. The code in Example 6–15 shows a simple, unbuffered implementation of `getchar()`.

Example 6–15. Unbuffered getchar()

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

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

Use the Lightweight Device Driver API

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

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

■ JTAG UART

■ UART

■ Optrex 16207 LCD

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

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

This option is appropriate if your program has a limited need for file I/O. The Altera host-based file system and the Altera read-only zip file system are not available with the reduced device driver API. You can select the reduced device drivers through BSP settings.

By default, the lightweight device driver API is disabled.

For further details about the lightweight device driver API, refer to the *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

Use the Minimal Character-Mode API

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

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

These functions are appropriate if your program only needs to accept command strings and send simple text messages. Some of them are helpful only in conjunction with the lightweight device driver API, discussed in "Use the Lightweight Device Driver API" on page 6–33.

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

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

alt_printf()

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

alt_putchar()

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

alt_putstr()

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

alt_getchar()

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

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

Eliminate Unused Device Drivers

If a hardware device is present in the system, by default the Nios II development flows assume the device needs drivers, and configure the HAL BSP accordingly. If the HAL can find an appropriate driver, it creates an instance of this driver. If your program never actually accesses the device, resources are being used unnecessarily to initialize the device driver.

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

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

In the Nios II IDE, you can prevent the HAL from including the flash driver by defining the `ALT_EXCLUDE_CFI_FLASH` preprocessor option in the properties for the BSP project.

You can selectively omit any individual driver, select a specific driver version, or substitute your own driver.

For further information about controlling driver configurations, refer to the *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

Another way to control the device driver initialization process is to use the free-standing environment. For details, refer to "Boot Sequence and Entry Point" on page 6–36.

Eliminate Unneeded Exit Code

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

Eliminate Clean Exit

To avoid the overhead associated with providing a clean exit, your program can use the function `_exit()` in place of `exit()`. This function does not require you to change source code. You can select the `_exit()` function through a BSP setting.

Eliminate All Exit Code

Many embedded systems never exit at all. In such cases, exit code is unnecessary. You can eliminate all exit code through a BSP setting.

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

Turn off C++ Support

By default, the HAL provides support for C++ programs, including default constructors and destructors. You can disable C++ support through a BSP setting.

Boot Sequence and Entry Point

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

Hosted versus Free-Standing Applications

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

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

☞ Using the free-standing environment increases the complexity of writing Nios II programs, because you assume responsibility for initializing the system. If your main interest is to reduce code footprint, you should use the suggestions described in "Reducing Code Footprint" on page 6–29. It is easier to reduce the HAL BSP footprint by using BSP settings, than to use the free-standing mode.

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

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

Boot Sequence for HAL-Based Programs

The HAL provides system initialization code in the C runtime library (**crt0.S**). This code performs the following boot sequence:

■ Flushes the instruction and data cache.

■ Configures the stack pointer.

■ Configures the global pointer register.

■ Initializes the BSS region to zeroes using the linker-supplied symbols `__bss_start` and `__bss_end`. These are pointers to the beginning and the end of the BSS region.

■ If there is no boot loader present in the system, copies to RAM any linker section whose run address is in RAM, such as `.rwdata`, `.rodata`, and `.exceptions`. Refer to "Global Pointer Register" on page 6–42.

■ 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 operating system (OS) scheduler, this macro has no effect.

■ If you are using the HAL with an operating system, initializes the `alt_fd_list_lock` semaphore, which controls access to the HAL file systems.

■ Initializes the interrupt controller, and enable interrupts.

■ Calls the `alt_sys_init()` function, which initializes all device drivers and software packages in the system. The Nios II development flow creates the file `alt_sys_init.c` for each HAL BSP.

■ Redirects the C standard I/O channels (`stdin`, `stdout`, and `stderr`) to use the appropriate devices.

■ Calls the C++ constructors, using the `_do_ctors()` function.

■ Registers the C++ destructors to be called at system shutdown.

■ Calls `main()`.

■ Calls `exit()`, passing the return code of `main()` as the input argument for `exit()`.

alt_main.c, installed with the Nios II EDS, provides this default implementation. The software build tools copy **alt_main.c** to your BSP directory.

Customizing the Boot Sequence

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

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

The prototype for `alt_main()` is:

```
void alt_main (void)
```

The HAL build environment includes mechanisms to override default HAL BSP code. This lets you override boot loaders, as well as default device drivers and other system code, with your own implementation.

alt_sys_init.c is a generated file, which you should not modify. However, the Nios II software build tools enable you to control the generated contents of **alt_sys_init.c**. To specify the initialization sequence in **alt_sys_init.c**, you manipulate the `auto_initialize` and `alt_sys_init_priority` properties of each driver, using the `set_sw_property` Tcl command.

For more information about generated files and how to control the contents of **alt_sys_init.c**, refer to the *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*. For general information about **alt_sys_init.c**, refer to the *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*. For details about the `set_sw_property` Tcl command, refer to the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Memory Usage

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

Memory Sections

By default, HAL-based systems are linked using a generated linker script that is created and managed by the Nios II IDE or the Nios II software build tools. This linker script controls the mapping of code and data to 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 a memory component named `sdram` is defined in the **system.h** file, there is a memory section named `.sdram`. Figure 6–3 shows the organization of a typical HAL link map.

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

In a memory device containing the reset or exception address, the linker creates a normal (non-reserved) memory section above the `.entry` or `.exceptions` section. If there is a region of memory below the `.entry` or `.exceptions` section, it is unavailable to the Nios II software. Figure 6–3 illustrates an unavailable memory region below the `.exceptions` section.

Figure 6–3. Sample HAL Link Map

Assigning Code and Data to Memory Partitions

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

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

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 in the section that contains the exception address. By default, the remaining code and data are divided into the following output sections:

- `.text`—All remaining code

- `.rodata`—The read-only data

- `.rwdata`—Read-write data

- `.bss`—Zero-initialized data

You can control the placement of `.text`, `.rodata`, `.rwdata`, and all other memory partitions by manipulating BSP settings. For details about how to control BSP settings, refer to "HAL BSP Settings" on page 6–2.

The BSP editor is a very convenient way to manipulate the linker's memory map. The BSP editor displays memory section and region assignments graphically, allowing you to see overlapping or unused sections of memory. The BSP editor is available at the command line of the Nios II software build tools.

    For details, refer to the *Introduction to the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

Advanced Placement Options

In your program source code, you can specify a target memory section for each piece of code. In C or C++, you can use the `section` attribute. This attribute must be placed in a function prototype; you cannot place it in the function declaration itself. The code in Example 6–16 places a variable `foo` in the memory named `ext_ram`, and the function `bar()` in the memory named `sdram`.

Example 6–16. Manually Assigning C Code to a Specific Memory Section

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

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

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

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

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

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

- `.txt` for code: for example, `.sdram.txt`

- `.rodata` for read-only data: for example, `.cfi_flash.rodata`

- `.rwdata` for read-write data: for example, `.ext_ram.rwdata`

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

☞ A powerful way to manipulate the linker memory map is by using the BSP editor. With the BSP editor, you can assign linker sections to specific physical regions, and then review a graphical representation of memory showing unused or overlapping regions. You start the BSP editor from the Nios II command shell. For details about using the BSP editor, refer to the editor's tool tips.

Placement of the Heap and Stack

By default, the heap and stack are placed in the same memory partition as the `.rwdata` section. The stack grows downwards (toward lower addresses) from the end of the section. The heap grows upwards from the last used memory in the `.rwdata` section. You can control the placement of the heap and stack by manipulating BSP settings.

By default, the HAL performs no stack or heap checking. This makes function calls and memory allocation faster, but it means that `malloc()` (in C) and `new` (in C++) are unable to detect heap exhaustion. You can enable run-time stack checking by manipulating BSP settings. With stack checking on, `malloc()` and `new()` can detect heap exhaustion.

To specify the heap size limit, set the preprocessor symbol `ALT_MAX_HEAP_BYTES` to the maximum heap size in decimal. For example, the preprocessor argument `-DALT_MAX_HEAP_SIZE=1048576` sets the heap size limit to 0x100000. You can specify this command-line option through a BSP setting. For more information about manipulating BSP settings, refer to "HAL BSP Settings" on page 6–2.

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

👣 Refer to the Nios II IDE help system for details about selecting stack and heap placement, and setting up stack checking.

For details about how to control BSP settings, refer to "HAL BSP Settings" on page 6–2.
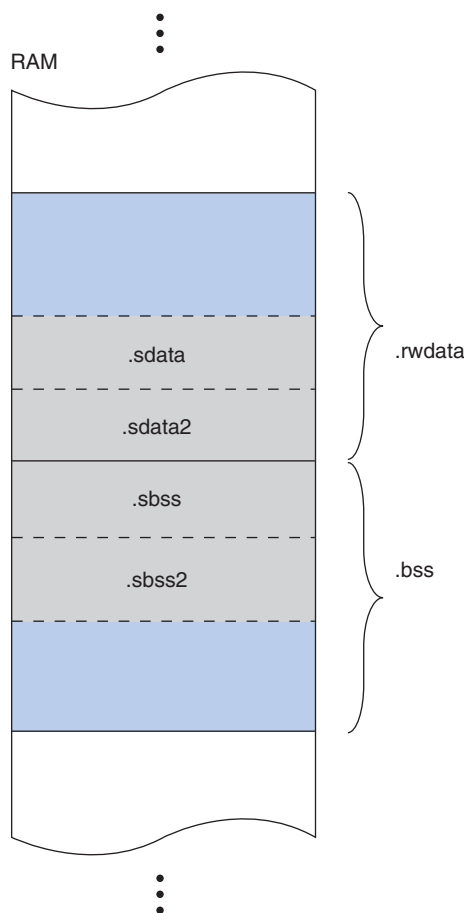
Global Pointer Register

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

The global pointer register can access a single contiguous region of 64 Kbytes. To avoid overflowing this region, the compiler only uses the global pointer with small global data structures. A data structure is considered "small" if its size is less than a specified threshold. By default, this threshold is 8 bytes.

The small data structures are allocated to the small global data sections, `.sdata`, `.sdata2`, `.sbss`, and `.sbss2`. The small global data sections are subsections of the `.rwdata` and `.bss` sections. They are located together, as shown in Figure 6–4, to enable the global pointer to access them.

Figure 6–4. Small Global Data sections

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

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

The `-G` option's numeric argument is in decimal. You can specify this compiler option through a project setting. For information about manipulating project settings, refer to "HAL BSP Settings" on page 6–2.

☞ You must set this option to the same value for both the BSP and the application project.

Boot Modes

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

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

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

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

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

Working with HAL Source Files

You might wish to view files in the HAL, especially header files, for reference. This section describes how to find HAL source files.

Finding HAL Files

You determine the location of HAL source files when you create the BSP.

HAL source files (and other BSP files) are copied to the BSP directory. You are free to modify copied HAL source files.

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

Overriding HAL Functions

HAL source files are copied to your BSP directory when you create your BSP. You can freely modify copied files, without losing your changes when you update your BSP.

For more information, refer to "Board Support Packages" in the *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

Using the HAL in an IDE Project

Like the Nios II software build tools development flow, the Nios II IDE flow can automatically keep your system library up to date with the SOPC Builder system. In an IDE project, the Nios II IDE manages the system library and updates the driver configurations to accurately reflect the system hardware. If the SOPC Builder system changes — i.e., the SOPC Builder system (**.ptf**) file is updated — the IDE rebuilds the system library the next time you build or run your C/C++ application program.

Generated Files

The IDE development flow uses different file name and directory structure conventions for some generated system library files, as described in this section.

generated.x

In an IDE project, the **generated.x** file is the same as the **linker.x** file created by the Nios II build tools.

generated.gdb

In an IDE project, the **generated.gdb** file is the same as the **memory.gdb** file created by the Nios II build tools.

alt_main.c

In an IDE project, you can find **alt_main.c** in *<Nios II EDS install path>*/**components/ altera_hal/HAL/src**.

System Library Settings

In an IDE project, you manage the system library project settings with the **System Library** page of the **Properties** dialog box.

For details of how to control system library settings, refer to the Nios II IDE help system.

Reducing Code Footprint

The basic techniques for reducing code footprint are the same in the Nios II IDE flow as in the software build tools flow, but you use a different procedure to specify the system library options. You control the following system library options through the Nios II IDE system properties dialog box:

Table 6–10. System Library Options for Reducing Code Footprint

Technique	System Library Option Name
Use Reduced Device Drivers	**Reduced device drivers**
Reduce the File Descriptor Pool	**Max file descriptors**
Use a Smaller File I/O Library	**Small C library**
Use the Lightweight Device Driver API	**Lightweight device driver API**
Eliminate Clean Exit	**Clean exit (flush buffers)**
Eliminate All Exit Code	**Program never exits**
Turn off C++ Support	**Support C++**

Paths to Hardware Abstraction Layer Files

In IDE projects, hardware abstraction layer (HAL) source files are in several directories because of the custom nature of Nios II systems. You can find HAL-related files in the following locations:

■ The *<Nios II EDS install path>*/**components** directory contains most HAL source files.

■ *<Nios II EDS install path>*/**components/altera_hal/HAL/inc/sys** contains header files defining the HAL generic device models. In a #include directive, reference these files relative to *<Nios II EDS install path>*/**components/altera_hal/HAL/inc/**. For example, to include the direct memory access (DMA) drivers, use #include sys/alt_dma.h

■ *<Nios II EDS install path>*/**bin** contains the newlib ANSI C library header files.

■ The Quartus® II Complete Design Suite includes HAL drivers for SOPC Builder components distributed with the Quartus II Complete Design Suite. For example, if the design suite is installed in **c:\altera\80**, you can find the drivers under **c:\altera\80\ip\sopc_builder_ip**.

☞ Do not edit HAL files in IDE projects.

Overriding HAL Functions

In the IDE build flow, you can override any HAL source file, including alt_sys_init.c, by placing your own implementation in your system project directory. When building the executable, the Nios II IDE finds your function, and uses it in place of the HAL version.

Referenced Documents

This chapter references the following documents:

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

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

- *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*

- *Exception Handling* chapter of the *Nios II Software Developer's Handbook*

- *Cache and Tightly-Coupled Memory* chapter of the *Nios II Software Developer's Handbook*

- *Read-Only Zip File System* chapter of the *Nios II Software Developer's Handbook*

- *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*

- *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*

- *Off-Chip Interface Peripherals* section in *Volume 5: Embedded Peripherals* of the *Quartus II Handbook*

- *The C Programming Language, Second Edition*, by Brian Kernighan and Dennis M. Ritchie (Prentice-Hall)

- *GNU documentation* on the Nios II EDS documentation launchpad

Document Revision History

Table 6–11 shows the revision history for this document.

Table 6–11. Document Revision History (Part 1 of 2)

Date & Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	■ Reorganized and updated information and terminology to clarify role of Nios II software build tools. ■ Add documentation for Altera logging. ■ Corrected minor typographical errors.	Altera logging
May 2008 v8.0.0	No change from previous release.	
October 2007 v7.2.0	■ Added documentation for HAL program development with the Nios II software build tools. ■ Additional documentation of alarms functions ■ Correct `alt_erase_flash_block()` example	—
May 2007 v7.1.0	■ Added table of contents to Introduction section. ■ Added Referenced Documents section.	—

Table 6–11. Document Revision History (Part 2 of 2)

Date & Document Version	Changes Made	Summary of Changes
March 2007 v7.0.0	No change from previous release.	
November 2006 v6.1.0	■ **Program never exits** system library option ■ **Support C++** system library option ■ **Lightweight device driver API** system library option ■ Minimal character-mode API	
May 2006 v6.0.0	■ Revised text on instruction emulation. ■ Added section on global pointers.	
October 2005 v5.1.0	■ Added `alt_64` and `alt_u64 types` to Table 6–1 on page 6–5. ■ Made changes to section "Placement of the Heap and Stack".	
May 2005 v5.0.0	Added `alt_load_section()` function information.	
December 2004 v1.2	■ Added boot modes information. ■ Amended compiler optimizations. ■ Updated Reducing Code Footprint section.	
September 2004 v1.1	Corrected DMA receive channels example code.	
May 2004 v1.0	Initial Release.	

NII52005-9.0.0

Introduction

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

This chapter also describes how to develop software packages for use with HAL board support packages (BSPs). The process of integrating a software package with the HAL is nearly identical with the process for integrating a device driver.

This chapter contains the following sections:

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 program interface (API) functions to access hardware. This makes your code more consistent and more portable to other Nios® II systems that might have different hardware configurations.

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

■ Integration in the HAL API

■ Peripheral-specific API

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

Integration in the HAL API

Integration in the HAL API is the preferred option for a peripheral that belongs to one of the HAL generic device model classes, such as character-mode or direct memory access (DMA) devices.

7–2

Chapter 7: Developing Device Drivers for the Hardware Abstraction Layer
Development Flow for Creating Device Drivers

For descriptions of the HAL generic device model classes, refer to the *Overview of the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

For integration in the HAL API, you write device access functions as specified in this chapter, and the device becomes accessible to software through the standard HAL API. For example, if you have a new LCD screen device that displays ASCII characters, you write a character-mode device driver. With this driver in place, programs can call the familiar `printf()` function to stream characters to the LCD screen.

Peripheral-Specific API

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

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

For details about integration in the HAL API, refer to "Integrating a Device Driver in the HAL" on page 7–17.

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

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

Before You Begin

This chapter assumes that you are familiar with C programming for the HAL.

Refer to the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook* for information you need before reading this chapter.

☞ This chapter uses the variable *<Altera installation>* to represent the location where the Altera® complete design suite is installed. On a Windows system, by default, that location is **c:/altera/***<version number>*.

Development Flow for Creating Device Drivers

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

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

2. Implement the driver functionality.

3. Test from `main()`.

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

5. Integrate the device driver in the HAL framework.

SOPC Builder Concepts

This section discusses basic concepts of the Altera SOPC Builder hardware design tool that enhance your understanding of the driver development process. You can develop Nios II device drivers without using SOPC Builder.

The Relationship between system.h and SOPC Builder

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

For more information about the **system.h** header file, refer to the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

Using SOPC Builder for Optimal Hardware Configuration

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

Components, Devices, and Peripherals

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

Accessing Hardware

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

All 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/altera/sopc_builder_ip** directory. Many components provide a header file that defines their hardware interface. The header file is named *<component name>*_**regs.h**, included in the **inc** subdirectory for the specific component. For example, the Altera-provided JTAG UART component defines its hardware interface in the file *<Altera installation>*/**ip/altera/sopc_builder_ip/altera_avalon_jtag_uart/inc/altera_avalon_jtag_uart_regs.h**.

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

■ Register access macros that provide a read and/or write macro for each register in the component that supports the operation. The macros are:

   ■ IORD_*<component name>*_*<register name>* (*<component base address>*)

   ■ IOWR_*<component name>*_*<register name>* (*<component base address>*, *<data>*)

   For example, **altera_avalon_jtag_uart_regs.h** defines the following macros:

   ■ IORD_ALTERA_AVALON_JTAG_UART_DATA()

   ■ IOWR_ALTERA_AVALON_JTAG_UART_DATA()

   ■ IORD_ALTERA_AVALON_JTAG_UART_CONTROL()

   ■ IOWR_ALTERA_AVALON_JTAG_UART_CONTROL()

■ Register address macros that return the physical address for each register in a component. The address register returned is the component's base address + the specified register offset value. These macros are named **IOADDR**_*<component name>*_*<register name>* (*<component base address>*).

   For example, **altera_avalon_jtag_uart_regs.h** defines the following macros:

   ■ IOADDR_ALTERA_AVALON_JTAG_UART_DATA()

   ■ IOADDR_ALTERA_AVALON_JTAG_UART_CONTROL()

   Use these macros only as parameters to a function that requires the specific address of a data source or destination. For example, a routine that reads a stream of data from a particular source register in a component might require the physical address of the register as a parameter.

■ Bit-field masks and offsets that provide access to individual bit-fields in a register. These macros have the following names:

   ■ *<component name>*_*<register name>*_*<name of field>*_MSK—A bit-mask of the field

   ■ *<component name>*_*<register name>*_*<name of field>*_OFST—The bit offset of the start of the field

   For example, ALTERA_AVALON_UART_STATUS_PE_MSK and ALTERA_AVALON_UART_STATUS_PE_OFST access the pe field of the status register.

Access a device's registers only with the macros defined in the **_regs.h** file. You must use the register access functions to ensure that the processor bypasses the data cache when reading and or writing the device. Do not use hard-coded constants, because they make your software susceptible to changes in the underlying hardware.

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

For detailed information about developing device drivers for HAL BSPs, refer to *AN 459: Guidelines for Developing a Nios II HAL Device Driver*. For a complete example of the **_regs.h** file, refer to the component directory for any of the Altera-supplied SOPC Builder components, such as *<Altera installation>*/**ip/sopc_builder_ip/altera_avalon_jtag_uart/inc**. For more information about the effects of cache management and device access, refer to the *Cache and Tightly-Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.

Creating Drivers for HAL Device Classes

The HAL supports a number of generic device model classes. By writing a device driver as described in this section, you describe to the HAL an instance of a specific device that falls into one of its known device classes. This section defines a consistent interface for driver functions so that the HAL can access the driver functions uniformly.

Generic device model classes are defined in the *Overview of the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

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

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

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

Character-Mode Device Drivers

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

Create a Device Instance

For a device to be made available as a character mode device, it must provide an instance of the `alt_dev` structure. The 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 about `open()`, `close()`, `read()`, `write()`, `lseek()`, `fstat()`, and `ioctl()`, refer to the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

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

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

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

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

The following code defines the `alt_fd` structure:

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

where:

■ `dev` is a pointer to the device structure for the device being used.

■ `fd_flags` is the value of `flags` passed to `open()`.

■ `priv` is a reserved, implementation-dependent argument, defined by the driver. If the driver requires any special, non-HAL-defined values to be maintained for each file or stream, you can store them in a data structure, and use `priv` maintains a pointer to the structure. The HAL ignores `priv`.

Allocate storage for the data structure in your `open()` function (pointed to by the `alt_dev` structure). Free the storage in your `close()` function.

☞ To avoid memory leaks, ensure that the `close()` function is called when the file or stream is no longer needed.

A driver is not required to provide all of the functions in the `alt_dev` structure. If a given function pointer is set to `NULL`, a default action is used instead. Table 7–1 shows the default actions for each of the available functions.

Table 7–1. Default Behavior for Functions Defined in alt_dev

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

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

Register a Character Device

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

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

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

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

🐾 For more information, refer to the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

File Subsystem Drivers

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

Create a Device Instance

Creating and registering a file system is very similar to creating and registering a character-mode device. To make a file system available, create an instance of the `alt_dev` structure (refer to "Character-Mode Device Drivers" on page 7–5). The only distinction is that the `name` field of the device represents the mount point for the file subsystem. Of course, you must also provide any necessary functions to access the file subsystem, such as `read()` and `write()`, similar to the case of the character-mode device.

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

Register a File Subsystem Device

You can register a file subsystem using the following function:

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

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

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

🐾 For more information, refer to the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

Timer Device Drivers

This section describes the system clock and timestamp drivers.

System Clock Driver

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

```
void alt_tick (void)
```

The expectation is that `alt_tick()` is called in 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 on success, and non-zero otherwise.

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

Timestamp Driver

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

For more information about using these functions, refer to the *Developing Programs Using the Hardware Abstraction Layer* and *HAL API Reference* chapters of the *Nios II Software Developer's Handbook*.

Flash Device Drivers

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

Create a Flash Driver

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

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

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

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

- `alt_flash_open_dev()`

- `alt_flash_close_dev()`

- `alt_write_flash()`

- `alt_read_flash()`

- `alt_get_flash_info()`

- `alt_erase_flash_block()`

- `alt_write_flash_block()`

where:

- the `base_addr` parameter is the base address of the flash memory

- `length` is the size of the flash in bytes

- `number_of_regions` is the number of erase regions in the flash

- `region_info` contains information about the location and size of the blocks in the flash device

For more information about the format of the `flash_region` structure, refer to "Using Flash Devices" in the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

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

Register a Flash Device

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

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

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

DMA Device Drivers

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

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

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

DMA Transmit Channel

A DMA transmit channel is constructed by creating an instance of the `alt_dma_txchan` structure, shown in Example 7–1.

Example 7–1. alt_dma_txchan Structure

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

Table 7–2 shows the available fields and their functions.

Table 7–2. Fields in the alt_dma_txchan Structure

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

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 on success, or negative if the device cannot be registered.

DMA Receive Channel

A DMA receive channel is constructed by creating an instance of the `alt_dma_rxchan` structure, shown in Example 7–2.

Example 7–2. alt_dma_rxchan Structure

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

Table 7–3 shows the available fields and their functions.

Table 7–3. Fields in the alt_dma_rxchan Structure

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

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

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

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

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

Ethernet Device Drivers

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

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

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

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

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

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

For more information about the NicheStack TCP/IP Stack implementation, refer to the *NicheStack Technical Reference Manual*, available on the Literature: Nios II Processor page of the Altera website.

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

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

Provide the NicheStack Hardware Interface Routines

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

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

These routines are fully documented in the *Porting Engineer Provided Functions* chapter of the *NicheStack Technical Reference*. The corresponding functions in the SMSC lan91c111 device driver are shown in Table 7–4.

Table 7–4. SMSC lan91c111 Hardware Interface Routines

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

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

■ A field for the IP address of the interface

■ A function pointer to a low-level function to initialize the MAC device

■ Function pointers to low-level functions to send packets

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

Provide *INSTANCE and *INIT Macros

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

■ *<component name>*_INSTANCE

■ *<component name>*_INIT

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

You can copy **altera_avalon_lan91c111_iniche.h** and modify it for your own driver. The HAL expects to find the `*INIT` and `*INSTANCE` macros in *<component name>***.h**, as discussed in "Header Files and alt_sys_init.c" on page 7–16. You can accomplish this with a `#include` directive as in **altera_avalon_lan91c111.h**, or you can define the macros directly in *<component name>***.h.**

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

■ The first field, llist, is for internal use, and must always be set to the value ALT_LLIST_ENTRY.

■ The second field, name, must be set to the device name as defined in **system.h**. For example, **altera_avalon_lan91c111_iniche.h** uses the C preprocessor's ## (concatenation) operator to reference the LAN91C111_NAME symbol defined in **system.h**.

■ The third field, init_func, must point to your software initialization function, as described in "Provide a Software Initialization Function". For example, **altera_avalon_lan91c111_iniche.h** inserts a pointer to alt_avalon_lan91c111_init().

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

When your driver is included in a Nios II BSP project, the HAL automatically initializes your driver by invoking the *INSTANCE and *INIT macros from its alt_sys_init() function. Refer to "Header Files and alt_sys_init.c" on page 7–16 for further detail about the *INSTANCE and *INIT macros.

Provide a Software Initialization Function

The *INSTANCE() macro inserts a pointer to your initialization function in the alt_iniche_dev structure, as described in "Provide *INSTANCE and *INIT Macros" on page 7–14. Your software initialization function must perform at least the following three tasks:

■ Initialize the hardware and verify its readiness

■ Finish initializing the alt_iniche_dev structure

■ Call get_mac_addr()

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

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

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

Creating a Custom Device Driver for the HAL

This section describes how to provide appropriate files to integrate your device driver in the HAL. The "Integrating a Device Driver in the HAL" section on page 7–17 describes the correct locations for the files.

Header Files and alt_sys_init.c

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

When you create the driver or software package, you specify in a Tcl script whether you want the `alt_sys_init()` function to invoke your INSTANCE and INIT macros. Refer to "Enabling Software Initialization" on page 7–24 for details.

Example 7–3 shows excerpts from an **alt_sys_init.c** file.

Example 7–3. Excerpt from an alt_sys_init.c File Performing Driver Initialization

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

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

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

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

☞   The remainder of this section assumes that you are using the `alt_sys_init()` HAL initialization mechanism.

The software build tools create **alt_sys_init.c** based on the header files associated with each device driver and software package. For a device driver, the header file must define the macros *<component name>*_INSTANCE and *<component name>*_INIT.

Like a device driver, a software package provides an INSTANCE macro, which `alt_sys_init()` invokes once. A software package header file can optionally provide an INIT macro.

For example, **altera_avalon_jtag_uart.h** must define the macros ALTERA_AVALON_JTAG_UART_INSTANCE and ALTERA_AVALON_JTAG_UART_INIT. The purpose of these macros is as follows:

■   The *_INSTANCE macro performs any required static memory allocation. For drivers, *_INSTANCE is invoked once per device instance, so that memory can be initialized on a per-device basis. For software packages, *_INSTANCE is invoked once.

■ The `*_INIT` macro performs runtime initialization of the device driver or software package.

In the case of a device driver, both macros take two input arguments:

■ The first argument, `name`, is the capitalized name of the device instance.

■ The second argument, `dev`, is the lower case version of the device name. `dev` is the name given to the component in SOPC Builder at system generation time.

You can use these input parameters to extract device-specific configuration information from the **system.h** file.

The name of the header file must be as follows:

■ Device driver: *<hardware component class>*.**h**. For example, if your driver targets the **altera_avalon_uart** component, the file name is **altera_avalon_uart.h**.

■ Software packages *<package name>*.**h**. For example, if you create the software package with the following command:

```
create_sw_package my_sw_package
```

the header file is called **my_sw_package.h**.

For a complete example, refer to any of the Altera-supplied device drivers, such as the JTAG UART driver in *<Altera installation>*/**ip/sopc_builder_ip/ altera_avalon_jtag_uart**.

☞ For optimal project rebuild time, do not include the peripheral header in **system.h**. It is included in **alt_sys_init.c**.

Device Driver Source Code

In addition to the header file, the component driver might need to provide compilable source code, to be incorporated in the BSP. This source code is specific to the hardware component, and resides in one or more C files (or assembly language files).

Integrating a Device Driver in the HAL

The Nios II software build tools can incorporate device drivers and software packages supplied by Altera, supplied by other third-party developers, or created by you. This section describes how to prepare device drivers and software packages so the BSP generator recognizes and adds them to a generated BSP.

You can take advantage of this service, whether you created a device driver for one of the HAL generic device models, or you created a peripheral-specific device driver.

☞ The process required to integrate a device driver is nearly identical to that required to develop a software package. The following sections describe the process for both. Certain steps are not needed for software packages, as noted in the text.

Overview

To publish a device driver or a software package, you provide the following items:

■ A header file defining the package or driver interface

■ A Tcl script specifying how to add the package or driver to a BSP

The header file and Tcl script are described in the following sections.

Assumptions and Requirements

This section assumes that you are developing a device driver or software package for eventual incorporation in a BSP. The driver or package is to be incorporated in the BSP by an end user who has limited knowledge of the driver or package internal implementation. To add your driver or package to a BSP, the end user must rely on the driver or package settings that you create with the tools described in this section.

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

■ It must have a defining Tcl script. The Tcl script for each driver or software package provides the Nios II software build tools with a complete description of the driver or software. This description includes the following information:

■ Name—A unique name identifying the driver or software package

■ Source files—The location, name, and type of each C/C++ or assembly language source or header file

■ Associated hardware class (device drivers only)—The name of the hardware peripheral class the driver supports

■ Versioning and compatibility information

■ BSP type(s)—Supported operating system(s)

■ Settings—Visible parameters controlling software build and runtime configuration

■ The Tcl script resides in the driver or software package root directory.

■ The Tcl script's file name ends with **_sw.tcl.** Example: **custom_ip_block_sw.tcl.**

■ The root directory of the driver or software package is in one of the following places:

■ In any directory included in the SOPC_BUILDER_PATH environment variable, or in any directory located one level beneath such a directory. This approach is recommended if your driver or software packages are installed in a distribution you create.

■ In a directory named **ip**, one level beneath the Quartus II project directory containing the design your BSP targets. This approach is recommended if your driver or software package is used only once, in a specific hardware project.

■ File names and directory structures conform to certain conventions, described in "File Names and Locations" on page 7–20.

■ If your driver or software package uses the HAL autoinitialization mechanism (alt_sys_init()), certain macros must be defined in a header file. For details about this header file, refer to "Header Files and alt_sys_init.c" on page 7–16.

■ If your driver needs to be compatible with the Nios II integrated development environment (IDE) development flow, it has a **component.mk** file. (Software packages do not have a **component.mk** file.)

The *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook* describes the commands you can use in the Tcl script.

The Nios II BSP Generator

This section describes the process by which the Nios II BSP generator adds device drivers and software packages to your BSP. The Nios II BSP generator, a subset of the Nios II software build tools, is a combination of command utilities and scripts that enable you to create and manage BSPs and their settings.

For an overview of the Nios II software build tools, refer to the *Overview* and *Introduction to the Nios II Software Build Tools* chapters of the *Nios II Software Developer's Handbook*.

Component Discovery

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

The BSP generator locates software packages and drivers by inspecting a list of known locations determined by the Altera Nios II EDS, Quartus II software, and MegaCore® IP Library installers, as well as searching locations specified in certain system environment variables.

The Nios II BSP tools identify drivers and software packages by locating and sourcing Tcl scripts with file names ending in **_sw.tcl** in these locations.

☞ For run-time efficiency, the BSP generator only looks at driver files that conform to the criteria listed in this section.

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

Device Driver Versions

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

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

Device Driver and Software Package Inclusion

The BSP generator adds software packages to the BSP if they are specifically requested during BSP generation, with the `enable_sw_package` command.

For further details, refer to "Tcl Commands for BSP Settings" in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

If no specific device driver is requested, and no compatible device driver is located for a particular hardware module, the BSP generator issues an informative message visible in either the `debug` or `verbose` generation output. This behavior is normal for many types of hardware in the SOPC Builder system, such as memory devices, that do not have device drivers. If a software package or specific driver is requested and cannot be located, an error is generated and BSP generation or settings update halts.

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

With the Tcl software definition files in place, the software build tools read in the Tcl file and populate the makefiles and other support files accordingly.

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

When you create a new software project, the Nios II software build tools generate the contents of **alt_sys_init.c** to match the specific hardware contents of the SOPC Builder system.

File Names and Locations

As described in "The Nios II BSP Generator" on page 7–19, the Nios II build tools find a device driver or software package by locating a Tcl script with the file name ending in **_sw.tcl**, and sourcing it.

Each peripheral in a Nios II system is associated with a specific SOPC Builder component directory. This directory contains a file defining the software interface to the peripheral. Refer to "Accessing Hardware" on page 7–3.

To enable the software build tools to find your component device driver, place the Tcl script in a directory named **ip** under your hardware project directory.

Figure 7–1 illustrates a file hierarchy suitable for the Nios II software build tools. This file hierarchy is located in the <*Altera installation*>/**ip/altera/sopc_builder_ip** directory. This example assumes a device driver supporting a hardware component named `custom_component`.

Source Code Discovery

You use Tcl scripts to specify the location of driver source files. For further details, refer to "The Nios II BSP Generator" on page 7–19.

Figure 7–1. Example Device Driver File Hierarchy and Naming
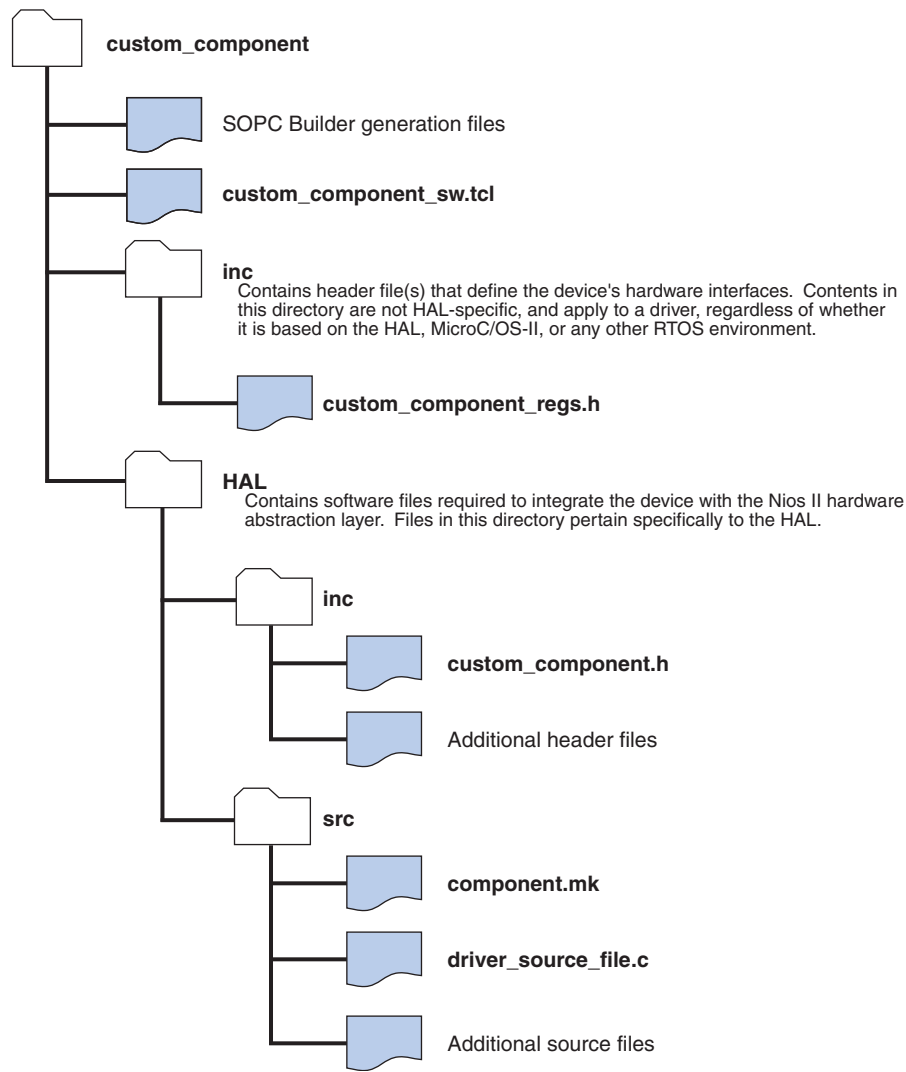
Driver and Software Package Tcl Script Creation

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

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

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

Tcl Command Walkthrough for a Typical Driver or Software Package

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

The example in this section creates a device driver for a hardware peripheral whose SOPC Builder component class name is `my_custom_component`. The driver supports both HAL and MicroC/OS-II BSP types. It has a single C source (**.c**) file and two C header (**.h**) files, organized as in the example in Figure 7–1.

Creating and Naming the Driver or Package

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

```
create_driver my_custom_component_driver
```

Identifying the Hardware Component Class

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

```
set_sw_property hw_class_name my_custom_component
```

☞ The `set_sw_property` command accepts several argument types. Each call to `set_sw_property` sets or overwrites a property to the value specified in the second argument.

👣 For further information about the `set_sw_property` command, refer to the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

The `hw_class_name` argument does not apply to software packages.

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

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

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

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

Setting the BSP Type

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

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

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

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

Specifying an Operating System

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

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

Specifying Source Files

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

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

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

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

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

Specifying a Subdirectory

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

```
set_sw_property bsp_subdirectory my_driver
```

☞ If the path begins with the BSP type (e.g `HAL` or `UCOSII`), the BSP type is removed and replaced with the value of the `bsp_subdirectory` property.

Enabling Software Initialization

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

For further details, refer to "Provide *INSTANCE and *INIT Macros" on page 7–14.

To support this functionality in Nios II BSPs, you must set the `set_sw_property` command's `auto_initialize` argument to `true` using the following Tcl command:

```
set_sw_property auto_initialize true
```

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

Adding Include Paths

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

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

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

☞ If the path begins with the BSP type (e.g `HAL` or `UCOSII`), the BSP type is removed and replaced with the value of the `bsp_subdirectory` property.

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

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

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

Version Compatibility

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

■ You can request a specific version of your driver or software package with BSP settings.

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

The *<version>* argument in each of the following versioning-related commands can be a string containing numbers and characters. Examples of version strings are `8.0`, `5.1.1`, `6.1`, and `6.1sp1`. The `.` character is a separator. The BSP generator compares versions against each other to determine if one is more recent than the other, or if two are equal, by successively comparing the strings between each separator. Thus, `2.1` is greater than `2.0`, and `2.1sp1` is greater than `2.1`. Two versions are equal if their version assignment strings are identical.

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

```
set_sw_property version 7.1
```

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

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

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

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

Creating Settings for Device Drivers and Software Packages

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

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

Settings affect the generated BSP in several ways:

■ Settings are added either to the BSP **system.h** or **public.mk**, or to the BSP **Makefile** as variable.

■ Settings are stored in the BSP settings file, named with hierarchy information to prevent namespace collision.

■ A default value of your choice is assigned to the setting so that the end user of the driver or package does not need to explicitly specify the setting when creating or updating a BSP.

■ Settings are displayed in the BSP **summary.html** document, along with description text of your choice.

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

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

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

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

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

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

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

Data Types

Several setting data types are available, controlled by the `type` argument to
`add_sw_setting`. They correspond to the data types you can express as `#define`
statements or values concatenated to makefile variables. The specific setting type
depends on your software's structure or BSP build needs. The available data types,
and their typical uses, are shown in Table 7–5.

Table 7–5. Data Type Settings

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

Setting Destination Files

The `destination` argument of `add_sw_setting` specifies settings and their
assigned values. This argument controls the file to which the setting is saved in the
BSP. The BSP generator formats the setting's assigned value based on the definition
file and type of setting. Table 7–6 shows possible values of the `destination`
argument.

Table 7–6. Destination File Settings

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

☞ Certain setting types are not compatible with the **public.mk** or **Makefile** destination file types.

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

Setting Display Name

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

Setting Generation Name

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

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

You can enter the exact text, `MY_DRIVER_BUFFER_SIZE`, in the `generationName` argument.

Setting Default Value

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

☞ The value assigned to any setting, whether it is the default value in the driver or software package Tcl script, or entered by the user configuring the BSP, must be compatible with the selected setting.

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

Setting Description

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

Setting Creation Example

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

Example 7–4. Supporting Driver Settings

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

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

You add the `MY_CUSTOM_DRIVER_SMALL` setting to your driver as follows using the `add_sw_setting` Tcl command:

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

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

👣 Each argument has several variants. For detailed usage and restrictions, refer to the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook.*

Reducing Code Footprint

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

■ Provide reduced footprint drivers. This technique usually reduces driver functionality.

■ Support the lightweight device driver API. This technique reduces driver overhead. It need not reduce functionality, but it might restrict your flexibility in using the driver.

These techniques are discussed in the following sections.

Provide Reduced Footprint Drivers

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

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

You can enable `ALT_USE_SMALL_DRIVERS` in a BSP with the `hal.enable_reduced_device_drivers` BSP setting.

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

Support the Lightweight Device Driver API

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

If you want to support the lightweight device driver API on a character-mode device, you need to write at least one of the lightweight character-mode functions listed in Table 7–7. Implement the functions needed by your software. For example, if you only use the device for `stdout`, you only need to implement the *<component class>*`_write()` function.

To support the lightweight device driver API, name your driver functions based on the component class name, as shown in Table 7–7.

Table 7–7. Driver Functions for Lightweight Device Driver API

Function	Purpose	Example *(1)*
*<component class>*_read()	Implements character-mode read functions	altera_avalon_jtag_uart_read()
*<component class>*_write()	Implements character-mode write functions	altera_avalon_jtag_uart_write()
*<component class>*_ioctl()	Implements device-dependent functions	altera_avalon_jtag_uart_ioctl()

(1) Based on component **altera_avalon_jtag_uart**

When you build your BSP with ALT_USE_DIRECT_DRIVERS enabled, instead of using file descriptors, the HAL accesses your drivers with the following macros:

- ALT_DRIVER_READ(instance, buffer, len, flags)

- ALT_DRIVER_WRITE(instance, buffer, len, flags)

- ALT_DRIVER_IOCTL(instance, req, arg)

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

These macros, together with the system-specific macros that the Nios II software build tools create in **system.h**, generate calls to your driver functions. For example, with lightweight drivers turned on, printf() calls the HAL write() function, which directly calls your driver's *<component class>*_write() function, bypassing file descriptors.

You can enable ALT_USE_DIRECT_DRIVERS in a BSP with the hal.enable_lightweight_device_driver_api BSP setting.

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

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

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

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 file names for device driver source and header files.

Overriding the Default Device Drivers

All SOPC Builder components can elect to provide a HAL device driver. Refer to "Integrating a Device Driver in the HAL" on page 7–17. However, if the driver supplied with a component is inappropriate for your application, you can override the default driver by supplying a different driver.

Overriding with Tcl

You specify a custom driver with the following BSP Tcl command:

`set_driver` *<driver name> <component name>*

For example, if you are using the `nios2-bsp` command, you replace the default driver for `uart0` with a driver called `custom_driver` as follows:

`nios2-bsp hal my_bsp --cmd set_driver custom_driver uart0`↵

Overriding by Replacement

The Nios II software build tools find the driver source files and copy them to the **drivers** directory of your BSP, as described in "The Nios II BSP Generator" on page 7–19. If you choose to edit or replace these files, your BSP is built with the updated files.

For further details about BSP source files, refer to "Board Support Packages" in the *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

Device Drivers for IDE Projects

HAL device drivers work the same in the IDE flow as in the software build tools flow. However, there are slight differences in how you create a device driver.

Compared with the Nios II IDE, Nios II software build tools provide a less rigid set of file naming and location requirements for your drivers. However, Altera recommends using the Nios II IDE conventions to maintain build-flow compatibility. Provided you use the file hierarchy described in "File Names and Locations" on page 7–20, your device driver is compatible with the Nios II IDE development flow.

This section describes how to develop device drivers for Nios II IDE projects.

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.

Device Driver Files for the HAL

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

A Device's HAL Header File and alt_sys_init.c

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

Example 7–3 on page 7–16 shows excerpts from an **alt_sys_init.c** file.

A Device's HAL Header File and alt_sys_init.c

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

- Include the device's header file.

- Call the macro *<name of device>*`_INSTANCE` to allocate storage for the device.

- Call the macro *<name of device>*`_INIT` inside the `alt_sys_init()` function to initialize the device.

Device Driver Source Code

Place any required source code in the **HAL/src** directory. In addition, you should include a makefile fragment, **component.mk**. The **component.mk** file lists the source files to include in the system library. You can list multiple files by separating filenames with a space. Example 7–5 shows an example makefile fragment for Altera's JTAG UART device.

Example 7–5. component.mk for a UART Driver

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

The Nios II IDE includes the **component.mk** file into the top-level makefile when compiling system library projects and application projects. **component.mk** can only modify the make variables listed in Table 7–8.

Table 7–8. 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.

For details about namespace rules, refer to "Namespace Allocation" on page 7–32.

Overriding the Default Device Drivers

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

Referenced Documents

This chapter references the following documents:

■ *Overview* chapter of the *Nios II Software Developer's Handbook*

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

■ *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*

- *Overview of the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*

- *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*

- *Exception Handling* chapter of the *Nios II Software Developer's Handbook*

- *Cache and Tightly-Coupled Memory* chapter of the *Nios II Software Developer's Handbook*

- *Ethernet and the NicheStack TCP/IP Stack - Nios II Edition* chapter of the *Nios II Software Developer's Handbook*

- *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*

- *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*

- *AN 459: Guidelines for Developing a Nios II HAL Device Driver*

- *NicheStack Technical Reference Manual*, available on the Literature: Nios II Processor page of the Altera website

Document Revision History

Table 7–9 shows the revision history for this document.

Table 7–9. Document Revision History (Part 1 of 2)

Date & Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	■ Reorganized and updated information and terminology to clarify role of Nios II software build tools. ■ Incorporated information about Tcl-based device drivers and software packages, formerly in *Using the Nios II Software Build Tools*. ■ Described use of the `INSTANCE` macro in software packages. ■ Corrected minor typographical errors.	Tcl-based device drivers and software packages
May 2008 v8.0.0	No change from previous release.	
October 2007 v7.2.0	Added documentation for HAL device driver development with the Nios II software build tools.	—
May 2007 v7.1.0	■ Added table of contents to Introduction section. ■ Added Referenced Documents section.	—
March 2007 v7.0.0	No change from previous release.	
November 2006 v6.1.0	■ Add section "Reducing Code Footprint" ■ Replace lwIP driver section with NicheStack TCP/IP Stack driver section	Lightweight device driver API and minimal file I/O API; NicheStack TCP/IP Stack support.

Table 7–9. Document Revision History (Part 2 of 2)

Date & Document Version	Changes Made	Summary of Changes
May 2006 v6.0.0	No change from previous release.	
October 2005 v5.1.0	Added IOADDR_* macro details to section "Accessing Hardware".	
May 2005 v5.0.0	Updated reference to version of lwIP from 0.7.2 to 1.1.0.	
December 2004 v1.1	Updated reference to version of lwIP from 0.6.3 to 0.7.2.	
May 2004 v1.0	Initial Release.	