6. Developing Programs Using the Hardware Abstraction Layer

NII52004-7.2.0

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 API for HAL-based systems is readily accessible to software developers who are new to the Nios II processor. Programs based on the HAL use the ANSI C standard library functions and runtime environment, and access hardware resources via the HAL API's generic device models. The HAL API largely conforms to the familiar ANSI C standard library functions, though the ANSI C standard library is separate from the HAL. 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 Design Flows

As described in the *Overview* chapter of the *Nios II Software Developer's Handbook*, the Nios II EDS offers the following two distinct design flows:

- The Nios II IDE design flow
- The Nios II software build tools design flow

Most of the information in this chapter applies to both design flows. Design flow differences are noted explicitly.

Both design flows create board support packages (BSPs). However, the Nios II IDE design flow refers to a BSP as a system library.

For more detailed information about developing programs in the Nios II software build tools design flow, refer to the *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

HAL BSP Settings

Every Nios II BSP possesses settings, which determine the BSP's characteristics. For example, HAL BSPs have settings that determine 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 which design flow you are using. In the Nios II IDE, you manipulate BSP (system library) settings through the **System Library Properties** page. With the Nios II software build tools, you manipulate BSP settings with command line options or Tcl scripts.

For details of how to control BSP settings, refer to:

- The Nios II IDE help system — for IDE-managed projects
- The *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook* — for user-managed projects.

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

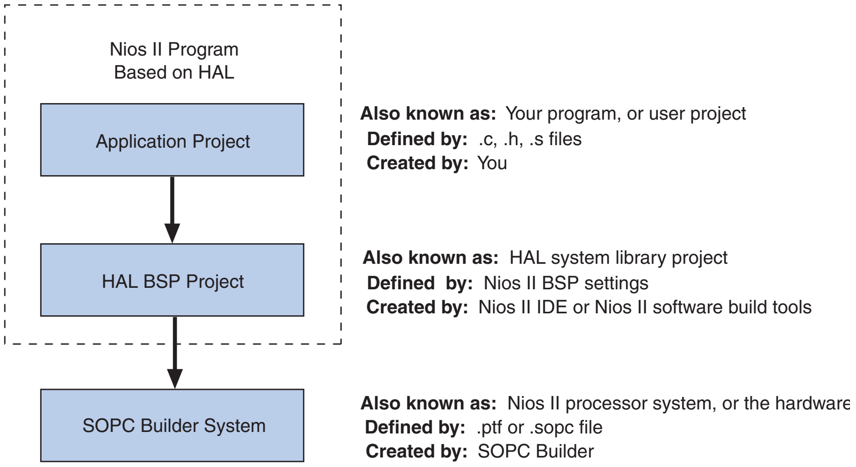
Do not edit **system.h**. Both design flows provide tools to manipulate system settings.

The Nios II Project Structure

The creation and management of software projects based on the HAL is integrated tightly with both Nios II design flows. 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


Nios II Program Based on HAL
Application Project
HAL BSP Project
SOPC Builder System
Also known as: Your program, or user project
Defined by: .c, .h, .s files
Created by: You
Also known as: HAL system library project
Defined by: Nios II BSP settings
Created by: Nios II IDE or Nios II software build tools
Also known as: Nios II processor system, or the hardware
Defined by: .ptf or .sopc file
Created by: SOPC Builder

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 (system library) project when you create your application project. In the Nios II software build tools, you create the BSP using **nios2-create-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 built into the BSP project.

The BSP project depends on the SOPC Builder system, defined by an SOPC Builder system file (**.sopc** or **.ptf**). Both build flows can automatically 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.

In an IDE-managed project, the Nios II IDE manages the HAL BSP (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 file (**.ptf**) is updated — the IDE rebuilds the HAL system library the next time you build or run your application program.

When you rebuild a user-managed project, the Nios II software build tools can automatically update your BSP to match the hardware. You control whether and when you allow these updates to take place.

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 fundamental question, "What hardware is present in this system?"

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

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

Both Nios II design 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 design flows provide tools to manipulate system settings.

For details of how to control BSP settings, see "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 it 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.

Table 6–1. The HAL Type Definitions

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

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

Table 6–2. GNU Toolchain Data Widths

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

UNIX-Style Interface

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

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

- `_exit()`
- `close()`
- `fstat()`
- `getpid()`
- `gettimeofday()`
- `ioctl()`
- `isatty()`
- `kill()`
- `lseek()`

- `open()`
- `read()`
- `sbrk()`
- `settimeofday()`
- `stat()`
- `usleep()`
- `wait()`
- `write()`

The most commonly used functions are those that relate to file I/O. See "File System" on page 6–7.

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

File System

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

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

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

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

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

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

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

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

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

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

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

Using Character-Mode Devices

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

Standard Input, Standard Output 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 on page 6–10 shows the classic Hello World program. This program sends characters to whatever device is associated with `stdout` when compiled in Nios II IDE.

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 (besides `stdin`, `stdout`, or `stderr`) is as easy as opening and writing to a file. The code in Example 6–4 demonstrates writing 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 could also be useful for applications that wish to sink unwanted data.

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

Lightweight Character-Mode I/O

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

Using File Subsystems

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

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

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

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

Using Timer Devices

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

The HAL API provides two types of timer device drivers:

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

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

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

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

System Clock Driver

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

For details of how to control BSP settings, see “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 units of “ticks”. For embedded engineers who deal with both hardware and software, do not confuse the HAL system clock with the clock signal driving the Nios II processor hardware. The period of a HAL system clock tick is generally much longer than the hardware system clock. **system.h** defines the clock tick frequency.

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

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

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

Alarms

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

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

The function `callback()` is called after `nticks` have elapsed. The input argument `context` is passed as the input argument to `callback()` when the call occurs. 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 be running 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 on the use of these functions, refer to the *Exception Handling* chapter of the *Nios II Software Developer's Handbook*.

The code fragment in Example 6–5 demonstrates registering an alarm for a periodic callback every second.

Example 6–5. Using a Periodic Alarm Callback Function

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

```
/*
 * The callback function.
 */

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

...

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

...

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

Timestamp Driver

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

You 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 that the Nios II processor system runs at—usually millions of cycles per second. The timestamp drivers are defined in the **alt_timestamp.h** header file.

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

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

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

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

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

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

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

Using Flash Devices

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

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

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

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

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

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

Simple Flash Access

This interface consists of the functions `alt_flash_open_dev()`, `alt_write_flash()`, `alt_read_flash()`, and `alt_flash_close_dev()`. The code "Using the Simple Flash API Functions" on page 6–17 shows the usage of all of these functions in one code example. You open a flash device by calling `alt_flash_open_dev()`, which returns a file handle to a flash device. This function takes a single argument that is the name of the flash device, as defined in **system.h**.

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

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

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

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

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

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

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

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

The code in Example 6–7 shows the usage of simple flash API functions to access a flash device named **/dev/ext_flash**, as defined in **system.h**.

Example 6–7. Using the Simple Flash API Functions

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

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

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

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

Block Erasure or Corruption

Generally, flash memory is divided into blocks. `alt_write_flash()` might need to erase the contents of a block before it can write data to it. In this case, it makes no attempt to preserve the existing contents of a block. This action can lead to unexpected data corruption (erasure), if you are performing writes that do not fall on block boundaries. If you wish to preserve existing flash memory contents, use the fine-grained flash functions. See "Fine-Grained Flash Access" on page 6–20.

Table 6–3. 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)
		Before First Write	First Write		Second Write	
			After Erasing Block(s)	After Writing Data 1	After Erasing Block(s)	After Writing Data 2
`0x0000`	1	??	FF	AA	AA	AA
`0x0400`	1	??	FF	AA	AA	AA
`0x0800`	1	??	FF	AA	AA	AA
`0x0C00`	1	??	FF	AA	AA	AA
`0x1000`	2	??	FF	AA	FF	FF *(1)*
`0x1400`	2	??	FF	FF	FF	BB
`0x1800`	2	??	FF	FF	FF	BB
`0x1C00`	2	??	FF	FF	FF	FF

Notes to Table 6–3:
(1) Unintentionally cleared to FF during erasure for second write.

Table 6–3 shows how you can cause unexpected data corruption by writing using the simple flash-access functions. Table 6–3 shows the example of an 8 Kbyte flash memory comprising two 4 Kbyte blocks. First write 5 Kbytes of all `0xAA` into flash memory at address `0x0000`, and then write 2 Kbytes of all `0xBB` to address `0x1400`. After the first write succeeds (at time t(2)), the flash memory contains 5 Kbyte of `0xAA`, and the rest is empty (i.e., `0xFF`). Then the second write begins, but before writing into the second block, the block is erased. At this point, t(3), the flash contains 4 Kbyte of `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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The code in Example 6–8 on page 6–21 demonstrates the usage of the fine-grained flash access functions.

Example 6–8. 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 into
          * the first block of the flash
          */
         ret_code = alt_write_flash_block (
                 fd,
                 regions->offset,
                 regions->offset+0x100,
                 write_data,
                 BUF_SIZE );
      }
    }
  }
  return 0;
}
```

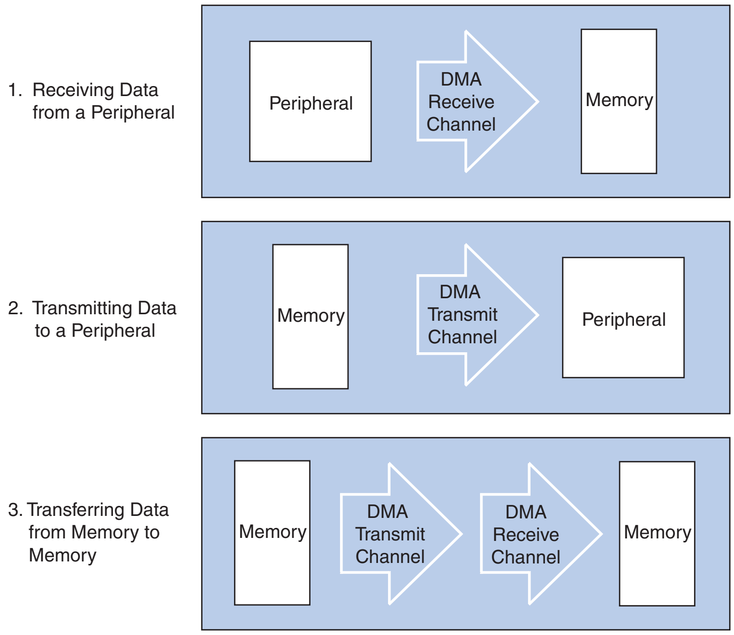
Using DMA Devices

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

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

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

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

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

DMA Transmit Channels

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

The code in Example 6–9 on page 6–24 shows how to obtain a handle for a DMA transmit device `dma_0`.

Example 6–9. Obtaining a File Handle for a DMA Device

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

int main (void)
{
  alt_dma_txchan tx;

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

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

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

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

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

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

If you are using the Altera Avalon-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 *"alt_dma_txchan_ioctl()"* in the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

DMA Receive Channels

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

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

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

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

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 (i.e., 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 Altera 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, see the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

The code in Example 6–10 shows a complete example application that posts a DMA receive request, and blocks in `main()` until the transaction completes.

Example 6–10. A DMA Transaction on a Receive Channel

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

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

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

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

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

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

Memory-to-Memory DMA Transactions

Copying data from one memory buffer to another buffer involves both receive and transmit DMA drivers. The code in Example 6–11 shows the process of queuing up a receive request followed by a transmit request to achieve a memory-to-memory DMA transaction.

Example 6–11. Copying Data from Memory to Memory

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

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

static volatile int rx_done = 0;

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

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

/*
 *
 */

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

  alt_dma_txchan txchan;
  alt_dma_rxchan rxchan;

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

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

  /* Create the receive channel */

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

  /* Post the transmit request */

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

  /* Post the receive request */

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

  /* wait for transfer to complete */

  while (!rx_done);

  printf ("Transfer successful!\n");

  return 0;
}
```

Reducing Code Footprint

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

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

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

Enable Compiler Optimizations

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

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

Use Reduced Device Drivers

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

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

In a user-managed software project, you can select the reduced device driver for an individual component.

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

Table 6–4 lists the Altera Nios II peripherals that currently provide small footprint drivers. The small footprint option might also affect other peripherals. Refer to each peripheral's data sheet for complete details of its driver's small footprint behavior.

Table 6–4. Altera Peripherals Offering Small Footprint Drivers

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

Reduce the File Descriptor Pool

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

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

Use /dev/null

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

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

Use a Smaller File I/O Library

Use the Small newlib C Library

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

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

Table 6–5 summarizes the limitations of the Nios II small newlib C library implementation.

Table 6–5. Limitations of the Nios II Small newlib C Library (Part 1 of 2)

Limitation	Functions Affected

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

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

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

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

Table 6–5. Limitations of the Nios II Small newlib C Library (Part 2 of 2)

Limitation	Functions Affected

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

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

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

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

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

Notes to Table 6–5:

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

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

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

The Nios II implementation of the small newlib C library differs somewhat from GCC. Table 6–5 provides details of the differences.

Use UNIX-Style File I/O

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

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 6–12 shows a simple, unbuffered implementation of `getchar()`.

Example 6–12. Unbuffered getchar()

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

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

Use the Lightweight Device Driver API

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

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

- JTAG UART
- UART
- Optrex 16207 LCD

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

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

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

You can turn on the **Lightweight device driver API** system library property in the Nios II IDE.

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

Alternatively, on the command line, you can use the preprocessor option `-DALT_USE_DIRECT_DRIVERS` when building the HAL BSP. By default, the lightweight device driver API is disabled.

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

Use the Minimal Character-Mode API

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

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

These functions are appropriate if your program only needs to accept command strings and send simple text messages. Some of them are helpful only in conjunction with the lightweight device driver API, discussed in "Use the Lightweight Device Driver API" on page 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 on the minimal character-mode functions, refer to the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

Eliminate Unused Device Drivers

If a hardware device is present in the system, by default the Nios II design 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 (system library) project. Alternatively, you can specify the `-DALT_EXCLUDE_CFI_FLASH` option to the preprocessor on the command line.

In a user-managed project, you can selectively omit any individual driver, select a specific driver version, or substitute your own driver.

For further information on 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 by using the free-standing environment. See "Boot Sequence and Entry Point" on page 6–37.

Eliminate Unneeded Exit Code

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

Eliminate Clean Exit

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

Eliminate All Exit Code

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

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

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

Turn off C++ Support

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

Boot Sequence and Entry Point

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

Hosted vs. Free-Standing Applications

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

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

☞ Using the freestanding environment increases the complexity of writing Nios II programs, because you assume responsibility for initializing the system. If your main interest 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 freestanding mode.

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

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

Boot Sequence for HAL-Based Programs

The HAL provides system initialization code 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
- Zero initializes the BSS region using the linker supplied symbols `__bss_start` and `__bss_end`. These are pointers to the beginning and the end of the BSS region
- If there is no boot loader present in the system, copies to RAM any linker section whose run address is in RAM, such as `.rwdata`, `.rodata`, and `.exceptions`. See "Global Pointer Register" on page 6–44.
- Calls `alt_main()`

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

- Calls `ALT_OS_INIT()` to perform any necessary operating system specific initialization. For a system that does not include an OS scheduler, this macro has no effect.

- If you are using the HAL with an operating system, initializes the `alt_fd_list_lock` semaphore, which controls access to the HAL file systems.
- Initializes the interrupt controller, and enable interrupts.
- Calls the `alt_sys_init()` function, which initializes all device drivers and software components in the system. The Nios II design 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. In an IDE-managed project, you can find it in <*Nios II EDS install path*>**/components/altera_hal/HAL/src**. For user-managed projects, the software build tools copy **alt_main.c** into 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 gives you the power to selectively enable HAL services. If your application requires an `alt_main()` entry point, you can copy the default implementation as a starting point and customize it to your needs.

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

The prototype for `alt_main()` is:

```
void alt_main (void)
```

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.

In the IDE-managed build flow, all source and header files are located using a search path. The build system always searches the BSP (system library) project's paths first. You can override any HAL source file, including **alt_sys_init.c**, by placing your own implementation in your system project directory. Your custom file is used in place of the auto-generated version.

In the user-managed build flow, **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 in user-managed projects, 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 HAL* 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 the way that the HAL uses memory and how the HAL arranges code, data, stack, and other logical memory sections, in physical memory.

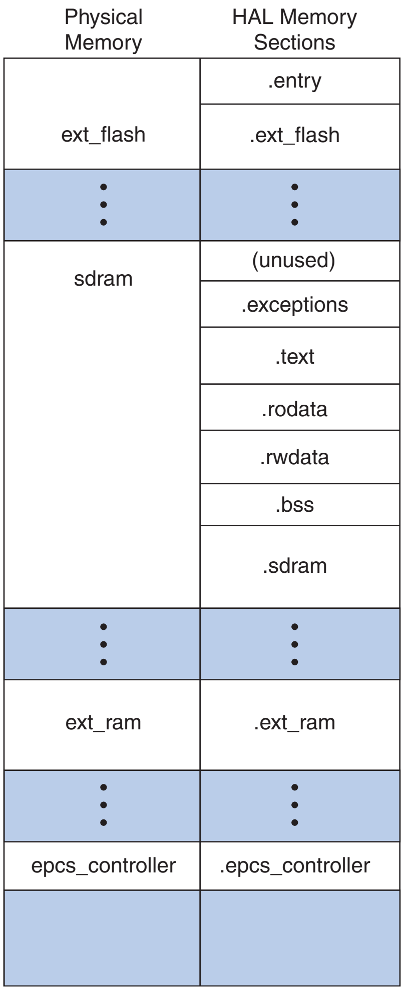
Memory Sections

By default, HAL-based systems are linked using an automatically-generated linker script that is created and managed by the Nios II IDE. This linker script controls the mapping of code and data within the available memory sections. The auto-generated linker script creates standard code and data sections (.text, .rodata, .rwdata, and .bss), plus a section for each physical memory device in the system. For example, if there is a memory component named `sdram` defined in the **system.h** file, there is a memory section named `.sdram`. Figure 6–3 on page 6–41 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 on page 6–41 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 design flow automatically specifies a sensible default partitioning. However, you might wish to change the partitioning in special situations.

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

Simple Placement Options

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

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

You can control the placement of `.text`, `.rodata`, `.rwdata`, and all other memory partitions by manipulating BSP settings.

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

For more information, in the Nios II IDE help system, search for the **"System Library Properties"** topic.

Advanced Placement Options

Within your program source code, you can specify a target memory section for each piece of code. In C or C++, you can use the `section` attribute. This attribute must be placed in a function prototype; you cannot place it in the function declaration itself. The code in Example 6–13 shows placing a variable `foo` within the memory named `ext_ram`, and the function `bar()` in the memory named `sdram`.

Example 6–13. Manually Assigning C Code to a Specific Memory Section

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

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

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

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

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

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

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

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

Placement of the Heap 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 within 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 in the system library properties; for details, refer to the Nios II IDE help system. Alternatively, you can specify the option on the command line.

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

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

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

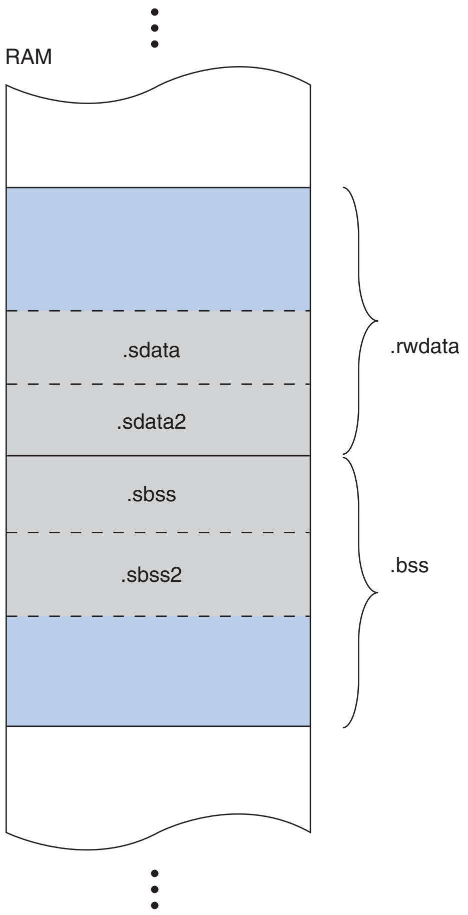
Global Pointer Register

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

The "small" data structures are allocated to the small global data sections, `.sdata`, `.sdata2`, `.sbss`, and `.sbss2`. The small global data sections are subsections of the `.rwdata` and `.bss` sections. They are located together, as shown in Figure 6–4 on page 6–45, 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 happens to be more than 64K bytes, they overflow the global pointer region. The linker produces an error message saying `"Unable to reach` *<variable name>* `... from the global pointer ... because the offset ... is out of the allowed range, -32678 to 32767."`

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

The `-G` option's numeric argument is in decimal. You can specify this compiler option in the project properties; for details, refer to the Nios II IDE help system. Alternatively, you can specify the option on the command line. You must set this option to the same value for both the 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 within it. The HAL provides a small boot loader program which copies these sections to their run time locations at boot time. You can specify run time locations for program and data memory 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, which is responsible for loading all program and data sections before the call to `_start`. When booting from an EPCS device, this loader function is provided by the hardware.

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

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

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

Paths to HAL Files

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

IDE-Managed Projects

In the IDE-managed build flow, HAL source files (and other BSP files) are referred to by path names. Do not edit HAL files in IDE-managed projects.

Finding HAL Files

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

- The <*Nios II EDS install path*>/**components** directory contains most HAL 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 DMA drivers, use `#include sys/alt_dma.h`
- Each Nios II IDE system project directory contains the **system.h** file generated for that BSP (system library)**.**
- <*Nios II EDS install path*>/**bin** contains the newlib ANSI C library header files.
- The Altera design suite includes HAL drivers for SOPC Builder components distributed with the Quartus® II Complete Design Suite. For example, if the Altera design suite is installed in **c:\altera\72**, you can find the drivers under **c:\altera\72\ip\sopc_builder_ip**.

Overriding HAL Functions

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

User-Managed Projects

In the user-managed build flow, HAL source files (and other BSP files) are copied into the BSP directory. You are free to modify copied HAL source files.

Finding HAL Files

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

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 into 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 *"Generated and Copied Files"* in the *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook.*

Referenced Documents

This chapter references the following documents:

- *Overview* 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 HAL* 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*
- *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*
- *Read-Only Zip File System* chapter of the *Nios II Software Developer's Handbook*
- *The C Programming Language, Second Edition*, by Brian Kernighan and Dennis M. Ritchie (Prentice-Hall)
- *GNU documentation* on the Nios II Literature page installed with the Nios II EDS.

Document Revision History

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

Table 6–6. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
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.	—
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. ● 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.	