NII52005-11.0.0

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:

Confine direct interaction with the hardware to device driver code. In general, the best practice is to keep most of your program code 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.

Driver 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.

Subscribe

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.

The HAL 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 7–18.

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 can not be written in C++. Restrict your driver code to either C or assembly language. C is preferred for portability.

Preparing for HAL Driver Development

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.

Nios II Hardware Design Concepts

This section discusses some basic concepts behind the Altera Qsys and SOPC Builder system integration tools. These concepts can enhance your understanding of the driver development process. You do not normally need to use a system integration tool when developing Nios II device drivers.

The Relationship Between the .sopcinfo File and system.h

The **system.h** header file provides a complete software description of the Nios II system hardware. The **system.h** system description is a fundamental part of developing drivers. Because drivers interact with hardware at the lowest level, it is worth understanding the relationship between the **.sopcinfo** file and **system.h**.

The system generation tool, Qsys or SOPC Builder, generates the Nios II processor system hardware. Hardware designers use the system generation tool 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 the system generation tool. These design choices are encapsulated in the **.sopcinfo** file. **system.h** is derived from the **.sopcinfo** file.

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 the System Generation Tool to Optimize Hardware

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 the system generation tool, Qsys or SOPC Builder. Before you write a device driver to accommodate imperfect hardware, it is worth considering whether the hardware can be improved easily with the system generation tool.

Components, Devices, and Peripherals

The Qsys and SOPC Builder system generation tools use the term "component" to describe hardware modules included in the system. In the context of Nios II software development, 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 the system generation tool.

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 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 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 Embedded 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 code in Example 7–1 defines the `alt_dev` structure.

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.

Example 7–1. 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;
```

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 must 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 exception 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 nonzero 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 must 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:

7–10

Chapter 7: Developing Device Drivers for the Hardware Abstraction Layer
Creating Embedded Drivers for HAL Device Classes

- 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 7–2.

Example 7–2. 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.

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)
```

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.

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

7–12

Chapter 7: Developing Device Drivers for the Hardware Abstraction Layer
Creating Embedded Drivers for HAL Device Classes

DMA Receive Channel

A DMA receive channel is constructed by creating an instance of the `alt_dma_rxchan` structure, shown in Example 7–3.

Example 7–3. 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.

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.

Table 7–3. Fields in the alt_dma_rxchan Structure

| Field	Function
llist	This function 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_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.

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.

The onion diagram in Figure 7–1 shows the architectural layers of a Nios II MicroC/OS-II software application.

Figure 7–1. Layered Software Model
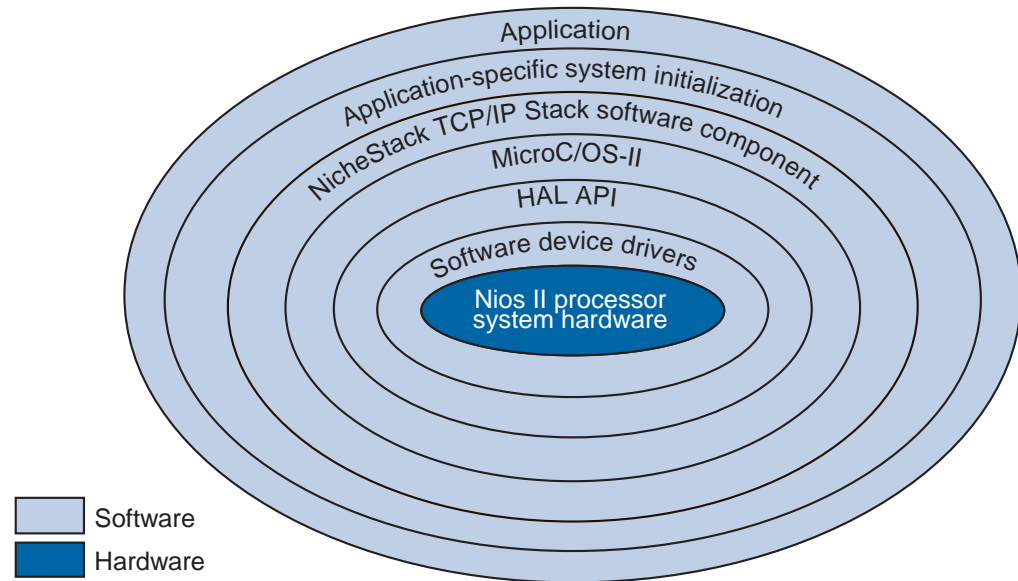
Each layer encapsulates the specific implementation details of that layer, abstracting the data for the next outer layer. However, the hierarchy of layers is not absolute. For example, the application makes system calls directly to the MicroC/OS-II or HAL API layers for services that do not require networking.

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 7–15. 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 7–18 describes the correct locations for the files.

Header Files and alt_sys_init.c

At the heart of the HAL is the autogenerated 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 7–25 for details.

Example 7–4 shows excerpts from an **alt_sys_init.c** file.

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

The Software Build Tools (SBT) creates **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.

Example 7–4. 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 );
}
```

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 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 SBT 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 SBT, 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 SBT 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
  - Version and compatibility information—The driver or package version, and (for drivers) information about what device core versions it supports.
  - BSP type(s)—The supported operating system(s)
  - Settings—The 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 7–21.

- 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.

For details about integrating a HAL device driver, refer to *AN 459: Guidelines for Developing a Nios II HAL Device Driver*. For details of the commands you can use in a driver Tcl script, refer to the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

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 SBT, 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 SBT, refer to the *Overview* and *Getting Started from the Command Line* 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 SBT searches for a suitable driver for each hardware module in the hardware 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 "Software Build Tools Tcl Commands" 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, 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 SBT to control any extra parameters that might exist.

With the Tcl software definition files in place, the SBT reads in the Tcl file and populate the makefiles and other support files accordingly.

When the Nios II SBT adds each driver or software package to the system, it uses 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 SBT generates the contents of **alt_sys_init.c** to match the specific hardware contents of the system.

File Names and Locations

As described in "The Nios II BSP Generator" on page 7–20, 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 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 SBT to find your component device driver, place the Tcl script in a directory named **ip** under your hardware project directory.

Figure 7–2 illustrates a file hierarchy suitable for the Nios II SBT. 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 7–20.

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 SBT provides 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.

Figure 7–2. Example Device Driver File Hierarchy and Naming

```
custom_component
│
├── [  ] Hardware system generation files
│
├── [  ] custom_component_sw.tcl
│
├── inc
│     Contains header file(s) that define the device's hardware interfaces.  Contents in
│     this directory are not HAL-specific, and apply to a driver, regardless of whether
│     it is based on the HAL, MicroC/OS-II, or any other RTOS environment.
│     │
│     └── [  ] custom_component_regs.h
│
└── HAL
      Contains software files required to integrate the device with the Nios II hardware
      abstraction layer.  Files in this directory pertain specifically to the HAL.
      │
      ├── inc
      │     ├── [  ] custom_component.h
      │     └── [  ] Additional header files
      │
      └── src
            ├── [  ] component.mk
            ├── [  ] driver_source_file.c
            └── [  ] Additional source files
```

Tcl Command Walkthrough for a Typical Driver or Software Package

The Tcl script excerpts in this section describe a typical device driver or software package.

The example in this section creates a device driver for a hardware peripheral whose component class name is `my_custom_component`. The driver supports both HAL and MicroC/OS-II BSP types. It has a single C source file (**.c**) and two C header files (**.h**), organized as in the example in Figure 7–2.

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 SBT uses 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 SBT finds multiple compatible
drivers or software packages with the same name, it 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 SBT.

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 file (**.s** or **.S**) 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 7–15.

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 user 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 a 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 (Part 1 of 2)

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 (')

Table 7–5. Data Type Settings (Part 2 of 2)

Data Type	Setting Value	Notes
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 build 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 7–5 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 7–5. 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 7–5, 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 in HAL Embedded Drivers

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 SBT creates 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.

HAL Namespace Allocation

To avoid conflicting names for symbols defined by devices in the hardware 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 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 HAL Default Device Drivers

All components can elect to provide a HAL device driver. Refer to "Integrating a Device Driver in the HAL" on page 7–18. However, if the driver supplied with a component is inappropriate for your application, you can override the default driver by supplying a different driver.

In the Nios II SBT for Eclipse, you can use the BSP Editor to specify a custom driver.

For information about selecting device drivers, refer to "Using the BSP Editor" in the *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook*

On the command line, 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↵
```

Document Revision History

Table 7–8 shows the revision history for this document.

Table 7–8. Document Revision History (Part 1 of 2)

Date	Version	Changes
May 2011	11.0.0	■ Introduction of Qsys system integration tool ■ Added figure illustrating NicheStack implementation layers
February 2011	10.1.0	Removed "Referenced Documents" section.
July 2010	10.0.0	Maintenance release.
November 2009	9.1.0	■ Introduced the Nios II Software Build Tools for Eclipse™. ■ Removed Nios II IDE-specific information.
March 2009	9.0.0	■ 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.
May 2008	8.0.0	Maintenance release.
October 2007	7.2.0	Added documentation for HAL device driver development with the Nios II Software Build Tools.
May 2007	7.1.0	■ Added table of contents to "Introduction" section. ■ Added Referenced Documents section.

Table 7–8. Document Revision History (Part 2 of 2)

Date	Version	Changes
March 2007	7.0.0	Maintenance release.
November 2006	6.1.0	■ Add section "Reducing Code Footprint in HAL Embedded Drivers". ■ Replace lwIP driver section with NicheStack TCP/IP Stack driver section.
May 2006	6.0.0	Maintenance release.
October 2005	5.1.0	Added IOADDR_* macro details to section "Accessing Hardware".
May 2005	5.0.0	Updated reference to version of lwIP from 0.7.2 to 1.1.0.
December 2004	1.1	Updated reference to version of lwIP from 0.6.3 to 0.7.2.
May 2004	1.0	Initial release.