This chapter describes the Nios® II Software Build Tools (SBT), a set of utilities and scripts that creates and builds embedded C/C++ application projects, user library projects, and board support packages (BSPs). The Nios II SBT supports a repeatable, scriptable, and archivable process for creating your software product.

You can invoke the Nios II SBT through either of the following user interfaces:

- The Eclipse™ GUI
- The Nios II Command Shell

The purpose of this chapter is to make you familiar with the internal functionality of the Nios II SBT, independent of the user interface employed.

☞ Before reading this chapter, consider getting an introduction to the Nios II SBT by first reading one of the following chapters:

- *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook*
- *Getting Started from the Command Line* chapter of the *Nios II Software Developer's Handbook*

This chapter contains the following sections:

- "Road Map for the SBT"
- "Makefiles" on page 4–3
- "Nios II Embedded Software Projects" on page 4–5
- "Common BSP Tasks" on page 4–8
- "Details of BSP Creation" on page 4–20
- "Tcl Scripts for BSP Settings" on page 4–27
- "Revising Your BSP" on page 4–30
- "Specifying BSP Defaults" on page 4–35
- "Device Drivers and Software Packages" on page 4–39
- "Boot Configurations for Altera Embedded Software" on page 4–40
- "Altera-Provided Embedded Development Tools" on page 4–42
- "Restrictions" on page 4–47

Subscribe

This chapter assumes you are familiar with the following topics:

■ The GNU **make** utility. Altera recommends you use version 3.80 or later. On the Windows platform, GNU **make** version 3.80 is provided with the Nios II EDS.

> You can obtain general information about GNU **make** from the Free Software Foundation, Inc. (www.gnu.org).

■ Board support packages.

Depending on how you use the tools, you might also need to be familiar with the following topics:

■ Micrium MicroC/OS-II. For information, refer to *MicroC/OS-II - The Real Time Kernel* by Jean J. Labrosse (CMP Books).

■ Tcl scripting language.

Road Map for the SBT

Before you start using the Nios II SBT, it is important to understand its scope. This section helps you understand their purpose, what they include, and what each tool does. Understanding these points helps you determine how each tool fits in with your development process, what parts of the tools you need, and what features you can disregard for now.

What the Build Tools Create

The purpose of the build tools is to create and build Nios II software projects. A Nios II project is a makefile with associated source files.

The SBT creates the following types of projects:

■ Nios II application—A program implementing some desired functionality, such as control or signal processing.

■ Nios II BSP—A library providing access to hardware in the Nios II system, such as UARTs and other I/O devices. A BSP provides a software runtime environment customized for one processor in a hardware system. A BSP optionally also includes the operating system, and other basic system software packages such as communications protocol stacks.

■ User library—A library implementing a collection of reusable functions, such as graphics algorithms.

Comparing the Command Line with Eclipse

Aside from the Eclipse GUI, there are very few differences between the SBT command line and the Nios II SBT for Eclipse. Table 4–1 lists the differences.

Table 4–1. Differences between Nios II SBT for Eclipse and the Command Line

Feature	Eclipse	Command Line
Project source file management	Specify sources automatically, e.g. by dragging and dropping into project	Specify sources manually using command arguments
Debugging	Yes	Import project to Eclipse environment
Integrates with custom shell scripts and tool flows	No	Yes

The Nios II SBT for Eclipse provides access to a large, useful subset of SBT functionality. Any project you create in Eclipse can also be created using the SBT from the command line or in a script. Create your software project using the interface that is most convenient for you. Later, it is easy to perform additional project tasks in the other interface if you find it advantageous to do so.

Makefiles

Makefiles are a key element of Nios II C/C++ projects. The Nios II SBT includes powerful tools to create makefiles. An understanding of how these tools work can help you make the most optimal use of them.

The Nios II SBT creates two kinds of makefiles:

■ Application or user library makefile—A simple makefile that builds the application or user library with user-provided source files

■ BSP makefile—A more complex makefile, generated to conform to user-specified settings and the requirements of the target hardware system

It is not necessary to use to the generated application and user library makefiles if you prefer to write your own. However, Altera recommends that you use the SBT to manage and modify BSP makefiles.

Generated makefiles are platform-independent, calling only utilities provided with the Nios II EDS (such as **nios2-elf-gcc**).

The generated makefiles have a straightforward structure, and each makefile has in-depth comments explaining how it works. Altera recommends that you study these makefiles for further information about how they work. Generated BSP makefiles consist of a single main file and a small number of makefile fragments, all of which reside in the BSP directory. Each application and user library has one makefile, located in the application or user library directory.

Modifying Makefiles

It is not necessary to edit makefiles by hand. The Nios II SBT for Eclipse offers GUI tools for makefile management.

4–4

For further information, refer to the *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook.*

On the command line, the project type determines the correct utility or utilities to update your makefile, as shown in Table 4–2.

Table 4–2. Command-Line Utilities for Updating Makefiles

Project Type	Utilities
Application	**nios2-app-update-makefile**
Library	**nios2-lib-update-makefile**
BSP *(1)*	**nios2-bsp-update-settings**
	nios2-bsp-generate-files

Note to Table 4–2:

(1) For details about updating BSP makefiles, refer to "Updating Your BSP" on page 4–32.

☞ After making changes to a makefile, run **make clean** before rebuilding your project. If you are using the Nios II SBT for Eclipse, this happens automatically.

Makefile Targets

Table 4–3 shows the application makefile targets. Altera recommends that you study the generated makefiles for further details about these targets.

Table 4–3. Application Makefile Targets

Target	Operation
help	Displays all available application makefile targets.
all (default)	Builds the associated BSP and libraries, and then builds the application executable file.
app	Builds only the application executable file.
bsp	Builds only the BSP.
libs	Builds only the libraries and the BSP.
clean	Performs a clean build of the application. Deletes all application-related generated files. Leaves associated BSP and libraries alone.
clean_all	Performs a clean build of the application, and associated BSP and libraries (if any).
clean_bsp	Performs a clean build of the BSP.
clean_libs	Performs a clean build of the libraries and the BSP.
download-elf	Builds the application executable file and then downloads and runs it.
program-flash	Runs the Nios II flash programmer to program your flash memory.

Note to Table 4–3:

(1) You can use the download-elf makefile target if the host system is connected to a single USB-Blaster™ download cable. If you have more than one download cable, you must download your executable with a separate command. Set up a run configuration in the Nios II SBT for Eclipse, or use **nios2-download**, with the --cable option to specify the download cable.

Nios II Embedded Software Projects

The Nios II SBT supports the following kinds of software projects:

■ C/C++ application projects

■ C/C++ user library projects

■ BSP projects

This section discusses each type of project in detail.

Applications and Libraries

The Nios II SBT has nearly identical support for C/C++ applications and libraries. The support for applications and libraries is very simple. For each case, the SBT generates a private makefile (named **Makefile**). The private makefile is used to build the application or user library.

The private makefile builds one of two types of files:

■ A **.elf** file—For an application

■ A library archive file (**.a**)—For a user library

For a user library, the SBT also generates a public makefile, called **public.mk**. The public makefile is included in the private makefile for any application (or other user library) that uses the user library.

When you create a makefile for an application or user library, you provide the SBT with a list of source files and a reference to a BSP directory. The BSP directory is mandatory for applications and optional for libraries.

The Nios II SBT examines the extension of each source file to determine the programming language. Table 4–4 shows the supported programming languages with the corresponding file extensions.

Table 4–4. Supported Source File Types

Programming Language	File Extensions *(1)*
C	**.c**
C++	**.cpp, .cxx, .cc**
Nios II assembly language; sources are built directly by the Nios II assembler without preprocessing	**.s**
Nios II assembly language; sources are preprocessed by the Nios II C preprocessor, allowing you to include header files	**.S**

Note to Table 4–4:

(1)   All file extensions are case-sensitive.

Board Support Packages

A Nios II BSP project is a specialized library containing system-specific support code. A BSP provides a software runtime environment customized for one processor in a hardware system. The BSP isolates your application from system-specific details such as the memory map, available devices, and processor configuration.

A BSP includes a **.a** file, header files (for example, **system.h**), and a linker script (**linker.x**). You use these BSP files when creating an application.

The Nios II SBT supports two types of BSPs: Altera® Hardware Abstraction Layer (HAL) and Micrium MicroC/OS-II. MicroC/OS-II is a layer on top of the Altera HAL and shares a common structure.

Overview of BSP Creation

The Nios II SBT creates your BSP for you. The tools provide a great deal of power and flexibility, enabling you to control details of your BSP implementation while maintaining compatibility with a hardware system that might change.

By default, the tools generate a basic BSP for a Nios II system. If you require more detailed control over the characteristics of your BSP, the Nios II SBT provides that control, as described in the remaining sections of this chapter.

Parts of a Nios II BSP

Hardware Abstraction Layer

The HAL provides a single-threaded UNIX-like C/C++ runtime environment. The HAL provides generic I/O devices, allowing you to write programs that access hardware using the newlib C standard library routines, such as `printf()`. The HAL interfaces to HAL device drivers, which access peripheral registers directly, abstracting hardware details from the software application. This abstraction minimizes or eliminates the need to access hardware registers directly to connect to and control peripherals.

For complete details about the HAL, refer to the *Hardware Abstraction Layer* section and the *HAL API Reference* chapter of the *Nios II Software Developer's Handbook*.

newlib C Standard Library

newlib is an open source implementation of the C standard library intended for use on embedded systems. It is a collection of common routines such as `printf()`, `malloc()`, and `open()`.

Device Drivers

Each device driver manages a hardware component. By default, the HAL instantiates a device driver for each component in your hardware system that needs a device driver. In the Nios II software development environment, a device driver has the following properties:

- A device driver is associated with a specific hardware component.

- A device driver might have settings that impact its compilation. These settings become part of the BSP settings.

Optional Software Packages

A software package is source code that you can optionally add to a BSP project to provide additional functionality. The NicheStack® TCP/IP - Nios II Edition is an example of a software package.

In the Nios II software development environment, a software package typically has the following properties:

■ A software package is not associated with specific hardware.

■ A software package might have settings that impact its compilation. These settings become part of the BSP settings.

☞ In the Nios II software development environment, a software package is distinct from a library project. A software package is part of the BSP project, not a separate library project.

Optional Real-Time Operating System

The Nios II EDS includes an implementation of the third-party MicroC/OS-II RTOS that you can optionally include in your BSP. MicroC/OS-II is built on the HAL, and implements a simple, well-documented RTOS scheduler. You can modify settings that become part of the BSP settings. Other operating systems are available from third-party vendors.

The Micrium MicroC/OS-II is a multi-threaded run-time environment. It is built on the Altera HAL.

The MicroC/OS-II directory structure is a superset of the HAL BSP directory structure. All HAL BSP generated files also exist in the MicroC/OS-II BSP.

The MicroC/OS-II source code resides in the **UCOSII** directory. The **UCOSII** directory is contained in the BSP directory, like the **HAL** directory, and has the same structure (that is, **src** and **inc** directories). The **UCOSII** directory contains only copied files.

The MicroC/OS-II BSP library archive is named **libucosii_bsp.a**. You use this file the same way you use **libhal_bsp.a** in a HAL BSP.

Software Build Process

To create a software project with the Nios II SBT, you perform several high-level steps:

1. Obtain the hardware design on which the software is to run. When you are learning about the build tools, this might be a Nios II design example. When you are developing your own design, it is probably a design developed by someone in your organization. Either way, you need to have the SOPC Information File (**.sopcinfo**).

2. Decide what features the BSP requires. For example, does it need to support an RTOS? Does it need other specialized software support, such as a TCP/IP stack? Does it need to fit in a small memory footprint? The answers to these questions tell you what BSP features and settings to use.

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

3. Define a BSP. Use the Nios II SBT to specify the components in the BSP, and the values of any relevant settings. The result of this step is a BSP settings file, called **settings.bsp**. For more information about creating BSPs, refer to "Board Support Packages" on page 4–5.

4. Create a BSP makefile using the Nios II build tools.

5. Optionally create a user library. If you need to include a custom software user library, you collect the user library source files in a single directory, and create a user library makefile. The Nios II build tools can create a makefile for you. You can also create a makefile by hand, or you can autogenerate a makefile and then customize it by hand. For more information about creating user library projects, refer to "Applications and Libraries" on page 4–5.

6. Collect your application source code. When you are learning, this might be a Nios II software example. When you are developing a product, it is probably a collection of C/C++ source files developed by someone in your organization. For more information about creating application projects, refer to "Applications and Libraries" on page 4–5.

7. Create an application makefile. The easiest approach is to let the Nios II build tools create the makefile for you. You can also create a makefile by hand, or you can autogenerate a makefile and then customize it by hand. For more information about creating makefiles, refer to "Makefiles" on page 4–3.

Common BSP Tasks

The Nios II SBT creates a BSP for you with useful default settings. However, for many tasks you must manipulate the BSP explicitly. This section describes the following common BSP tasks, and how you carry them out.

- "Using Version Control" on page 4–9

- "Copying, Moving, or Renaming a BSP" on page 4–10

- "Handing Off a BSP" on page 4–10

- "Creating Memory Initialization Files" on page 4–11

- "Modifying Linker Memory Regions" on page 4–11

- "Creating a Custom Linker Section" on page 4–12

- "Changing the Default Linker Memory Region" on page 4–16

- "Changing a Linker Section Mapping" on page 4–16

- "Creating a BSP for an Altera Development Board" on page 4–17

- "Querying Settings" on page 4–18

- "Managing Device Drivers" on page 4–18

- "Creating a Custom Version of newlib" on page 4–18

- "Controlling the stdio Device" on page 4–19

- "Configuring Optimization and Debugger Options" on page 4–19

Although this section describes tasks in terms of the SBT command line flow, you can also carry out most of these tasks with the Nios II SBT for Eclipse, described in the *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook*.

Adding the Nios II SBT to Your Tool Flow

A common reason for using the SBT is to enable you to integrate your software build process with other tools that you use for system development, including non-Altera tools. This section describes several scenarios in which you can incorporate the build tools in an existing tool chain.

Using Version Control

One common tool flow requirement is version control. By placing an entire software project, including both source and makefiles, under version control, you can ensure reproducible results from software builds.

When you are using version control, it is important to know which files to add to your version control database. With the Nios II SBT, the version control requirements depend on what you are trying to do and how you create the BSP.

If you create a BSP by running your own script that calls **nios2-bsp**, you can put your script under version control. If your script provides any Tcl scripts to **nios2-bsp** (using the `--script` option), you must also put these Tcl scripts under version control. If you install a new release of Nios II EDS and run your script to create a new BSP or to update an existing BSP, the internal implementation of your BSP might change slightly due to improvements in Nios II EDS.

Refer to "Revising Your BSP" on page 4–30 for a discussion of BSP regeneration with Nios II EDS updates.

If you create a BSP by running **nios2-bsp** manually on the command line or by running your own script that calls **nios2-bsp-generate-files**, you can put your BSP settings file (typically named **settings.bsp**) under version control. As in the scripted **nios2-bsp** case, if you install a new release of Nios II EDS and recreate your BSP, the internal implementation might change slightly.

If you want the exact same BSP after installing a new release of Nios II EDS, create your BSP and then put the entire BSP directory under version control before running `make`. If you have already run `make`, run `make clean` to remove all built files before adding the directory contents to your version control database. The SBT places all the files required to build a BSP in the BSP directory. If you install a new release of Nios II EDS and run `make` on your BSP, the implementation is the same, but the binary output might not be identical.

If you create a script that uses the command-line tools **nios2-bsp-create-settings** and **nios2-bsp-generate-files** explicitly, or you use these tools directly on the command line, it is possible to create the BSP settings file in a directory different from the directory where the generated BSP files reside. However, in most cases, when you want to store a BSP's generated files directory under source control, you also want to store the BSP settings file. Therefore, it is best to keep the settings file with the other BSP files. You can rebuild the project without the BSP settings file, but the settings file allows you to update and query the BSP.

☞ Because the BSP depends on a .**sopcinfo** file, you must usually store the .**sopcinfo** file in source control along with the BSP. The BSP settings file stores the .**sopcinfo** file path as a relative or absolute path, according to the definition on the **nios2-bsp** or **nios2-bsp-create-settings** command line. You must take the path into account when retrieving the BSP and the .**sopcinfo** file from source control.

Copying, Moving, or Renaming a BSP

BSP makefiles have only relative path references to project source files. Therefore you are free to copy, move, or rename the entire BSP. If you specify a relative path to the SOPC system file when you create the BSP, you must ensure that the **.sopcinfo** file is still accessible from the new location of the BSP. This **.sopcinfo** file path is stored in the BSP settings file.

Run `make clean` when you copy, move, or rename a BSP. The `make` dependency files (**.d**) have absolute path references. `make clean` removes the **.d** files, as well as linker object files (**.o**) and **.a** files. You must rebuild the BSP before linking an application with it. You can use the `make clean_bsp` command to combine these two operations.

👣 For information about **.d** files, refer to the GNU `make` documentation, available from the Free Software Foundation, Inc. (www.gnu.org).

Another way to copy a BSP is to run the **nios2-bsp-generate-files** command to populate a BSP directory and pass it the path to the BSP settings file of the BSP that you wish to copy.

If you rename or move a BSP, you must manually revise any references to the BSP name or location in application or user library makefiles.

Handing Off a BSP

In some engineering organizations, one group (such as systems engineering) creates a BSP and hands it off to another group (such as applications software) to use while developing an application. In this situation, Altera recommends that you as the BSP developer generate the files for a BSP without building it (that is, do not run `make`) and then bundle the entire BSP directory, including the settings file, with a utility such as **tar** or **zip**. The software engineer who receives the BSP can simply run `make` to build the BSP.

Linking and Locating

When autogenerating a HAL BSP, the SBT makes some reasonable assumptions about how you want to use memory, as described in "Specifying the Default Memory Map" on page 4–38. However, in some cases these assumptions might not work for you. For example, you might implement a custom boot configuration that requires a bootloader in a specific location; or you might want to specify which memory device contains your interrupt service routines (ISRs).

This section describes several common scenarios in which the SBT allows you to control details of memory usage.

Creating Memory Initialization Files

The **mem_init.mk** file includes targets designed to help you create memory initialization files (**.dat**, **.hex**, **.sym**, and **.flash**). The **mem_init.mk** file is designed to be included in your application makefile. Memory initialization files are used for HDL simulation, for Quartus® II compilation of initializable FPGA on-chip memories, and for flash programming. Initializable memories include M512 and M4K, but not MRAM.

Table 4–5 shows the **mem_init.mk** targets. Although the application makefile provides all these targets, it does not build any of them by default. The SBT creates the memory initialization files in the application directory (under a directory named **mem_init**). The SBT optionally copies them to your Quartus II project directory and HDL simulation directory, as described in Table 4–5.

☞ The Nios II SBT does not generate a definition of QUARTUS_PROJECT_DIR in your application makefile. If you have an on-chip RAM, and require that a compiled software image be inserted in your SRAM Object File (**.sof**) at Quartus II compilation, you must manually specify the value of QUARTUS_PROJECT_DIR in your application makefile. You must define QUARTUS_PROJECT_DIR before the **mem_init.mk** file is included in the application makefile, as in the following example:

```
QUARTUS_PROJECT_DIR = ../my_hw_design
MEM_INIT_FILE := $(BSP_ROOT_DIR)/mem_init.mk
include $(MEM_INIT_FILE)
```

Table 4–5. mem_init.mk Targets

Target	Operation
mem_init_install	Generates memory initialization files in the application **mem_init** directory. If the QUARTUS_PROJECT_DIR variable is defined, **mem_init.mk** copies memory initialization files to your Quartus II project directory named $(QUARTUS_PROJECT_DIR). If the SOPC_NAME variable is defined, **mem_init.mk** copies memory initialization files to your HDL simulation directory named $(QUARTUS_PROJECT_DIR)/$(SOPC_NAME)_sim.
mem_init_generate	Generates all memory initialization files in the application **mem_init** directory. This target also generates a Quartus II IP File (**.qip**). The **.qip** file tells the Quartus II software where to find the initialization files.
mem_init_clean	Removes the memory initialization files from the application **mem_init** directory.
hex	Generates all hex files.
dat	Generates all dat files.
sym	Generates all sym files.
flash	Generates all flash files.
<memory name>	Generates all memory initialization files for *<memory name>* component.

Modifying Linker Memory Regions

If the linker memory regions that are created by default do not meet your needs, BSP Tcl commands let you modify the memory regions as desired.

Suppose you have a memory region named onchip_ram. Example 4–1 shows a Tcl script named **reserve_1024_onchip_ram.tcl** that separates the top 1024 bytes of onchip_ram to create a new region named onchip_special.

For an explanation of each Tcl command used in this example, refer to the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Example 4–1. Reserved Memory Region

```
# Get region information for onchip_ram memory region.
# Returned as a list.
set region_info [get_memory_region onchip_ram]
# Extract fields from region information list.
set region_name [lindex $region_info 0]
set slave_desc [lindex $region_info 1]
set offset [lindex $region_info 2]
set span [lindex $region_info 3]
# Remove the existing memory region.
delete_memory_region $region_name
# Compute memory ranges for replacement regions.
set split_span 1024
set new_span [expr $span-$split_span]
set split_offset [expr $offset+$new_span]
# Create two memory regions out of the original region.
add_memory_region onchip_ram $slave_desc $offset $new_span
add_memory_region onchip_special $slave_desc $split_offset $split_span
```

If you pass this Tcl script to **nios2-bsp**, it runs after the default Tcl script runs and sets up a linker region named onchip_ram0. You pass the Tcl script to **nios2-bsp** as follows:

```
nios2-bsp hal my_bsp --script reserve_1024_onchip_ram.tcl↵
```

☞ Take care that one of the new memory regions has the same name as the original memory region.

If you run **nios2-bsp** again to update your BSP without providing the --script option, your BSP reverts to the default linker memory regions and your onchip_special memory region disappears. To preserve it, you can either provide the --script option to your Tcl script or pass the DONT_CHANGE keyword to the default Tcl script as follows:

```
nios2-bsp hal my_bsp --default_memory_regions DONT_CHANGE↵
```

Altera recommends that you use the --script approach when updating your BSP. This approach allows the default Tcl script to update memory regions if memories are added, removed, renamed, or resized. Using the DONT_CHANGE keyword approach does not handle any of these cases because the default Tcl script does not update the memory regions at all.

For details about using the --script argument, refer to "Calling a Custom BSP Tcl Script" on page 4–27.

Creating a Custom Linker Section

The Nios II SBT provides a Tcl command, add_section_mapping, to create a linker section.

Table 4–6 lists the default section names. The default Tcl script creates these default sections for you using the `add_section_mapping` Tcl command.

Table 4–6. Nios II Default Section Names

`.entry`
`.exceptions`
`.text`
`.rodata`
`.rwdata`
`.bss`
`.heap`
`.stack`

Creating a Linker Section for an Existing Region

To create your own section named `special_section` that is mapped to the linker region named `onchip_special`, use the following command to run **nios2-bsp**:

```
nios2-bsp hal my_bsp --cmd add_section_mapping special_section onchip_special↵
```

When the **nios2-bsp-generate-files** utility (called by **nios2-bsp**) generates the linker script **linker.x**, the linker script has a new section mapping. The order of section mappings in the linker script is determined by the order in which the `add_section_mapping` command creates the sections. If you use **nios2-bsp**, the default Tcl script runs before the `--cmd` option that creates the `special_section` section.

If you run **nios2-bsp** again to update your BSP, you do not need to provide the `add_section_mapping` command again because the default Tcl script only modifies section mappings for the default sections listed in Table 4–6.

Dividing a Linker Region to Create a New Region and Section

Example 4–2 creates a section named `.isrs` in the `tightly_coupled_instruction_memory` on-chip memory. This example works with any hardware design containing an on-chip memory named `tightly_coupled_instruction_memory` connected to a Nios II instruction master.

Example 4–2. Tcl Script to Create New Region and Section

```
# Get region information for tightly_coupled_instruction_memory memory region.
# Returned as a list.
set region_info [get_memory_region tightly_coupled_instruction_memory]
# Extract fields from region information list.
set region_name [lindex $region_info 0]
set slave [lindex $region_info 1]
set offset [lindex $region_info 2]
set span [lindex $region_info 3]
# Remove the existing memory region.
delete_memory_region $region_name
# Compute memory ranges for replacement regions.
set split_span 1024
set new_span [expr $span-$split_span]
set split_offset [expr $offset+$new_span]
# Create two memory regions out of the original region.
add_memory_region tightly_coupled_instruction_memory $slave $offset $new_span
add_memory_region isrs_region $slave $split_offset $split_span
add_section_mapping .isrs isrs_region
```

The Tcl script in Example 4–2 script splits off 1 KB of RAM from the region named
`tightly_coupled_instruction_memory`, gives it the name `isrs_region`, and then calls
`add_section_mapping` to add the `.isrs` section to `isrs_region`.

To use such a Tcl script, you would execute the following steps:

1. Create the Tcl script as shown in Example 4–2.

2. Edit your **create-this-bsp** script, and add the following argument to the **nios2-bsp**
   command line:

   ```
   --script <script name>.tcl
   ```

3. In the BSP project, edit **timer_interrupt_latency.h**. In the
   `timer_interrupt_latency_irq()` function, change the `.section` directive
   from `.exceptions` to `.isrs`.

4. Rebuild the application by running `make`.

After `make` completes successfully, you can examine the object dump file, *<project name>*.**objdump**, illustrated in Example 4–3. The object dump file shows that the new `.isrs` section is located in the tightly coupled instruction memory. This object dump file excerpt shows a hardware design with an on-chip memory whose base address is 0x04000000.

Example 4–3. Excerpts from Object Dump File

```
Sections:
Idx Name             Size     VMA       LMA       File off  Algn

    .
    .
    .

  6 .isrs            000000c0  04000c00  04000c00  000000b4  2**2
                     CONTENTS, ALLOC, LOAD, READONLY, CODE

    .
    .
    .

  9 .tightly_coupled_instruction_memory 00000000  04000000  04000000
00013778  2**0
                     CONTENTS
    .
    .
    .

SYMBOL TABLE:
00000000 l    d  .entry  00000000
30000020 l    d  .exceptions  00000000
30000150 l    d  .text  00000000
30010e14 l    d  .rodata  00000000
30011788 l    d  .rwdata  00000000
30013624 l    d  .bss  00000000
04000c00 l    d  .isrs  00000000
00000020 l    d  .ext_flash  00000000
03200000 l    d  .epcs_controller  00000000
04000000 l    d  .tightly_coupled_instruction_memory  00000000
04004000 l    d  .tightly_coupled_data_memory  00000000

    .
    .
    .
```

If you examine the linker script file, **linker.x**, illustrated in Example 4–4, you can see that **linker.x** places the new region isrs_region in tightly-coupled instruction memory, adjacent to the tightly_coupled_instruction_memory region.

Example 4–4. Excerpt From linker.x

```
MEMORY
{
reset : ORIGIN = 0x0, LENGTH = 32
tightly_coupled_instruction_memory : ORIGIN = 0x4000000, LENGTH = 3072
isrs_region : ORIGIN = 0x4000c00, LENGTH = 1024


    .
    .
    .


}
```

Changing the Default Linker Memory Region

The default Tcl script chooses the largest memory region connected to your Nios II processor as the default region. All default memory sections specified in Table 4–6 on page 4–13 are mapped to this default region. You can pass in a command-line option to the default Tcl script to override this default mapping. To map all default sections to onchip_ram, type the following command:

```
nios2-bsp hal my_bsp --default_sections_mapping onchip_ram↵
```

If you run **nios2-bsp** again to update your BSP, the default Tcl script overrides your default sections mapping. To prevent your default sections mapping from being changed, provide **nios2-bsp** with the original --default_sections_mapping command-line option or pass it the DONT_CHANGE value for the memory name instead of onchip_ram.

Changing a Linker Section Mapping

If some of the default section mappings created by the default Tcl script do not meet your needs, you can use a Tcl command to override the section mappings selectively. To map the .stack and .heap sections into a memory region named ram0, use the following command:

```
nios2-bsp hal my_bsp --cmd add_section_mapping .stack ram0 \
    --cmd add_section_mapping .heap ram0↵
```

The other section mappings (for example, .text) are still mapped to the default linker memory region.

If you run **nios2-bsp** again to update your BSP, the default Tcl script overrides your section mappings for .stack and .heap because they are default sections. To prevent your section mappings from being changed, provide **nios2-bsp** with the original add_section_mapping command-line options or pass the --default_sections_mapping DONT_CHANGE command line to **nios2-bsp**.

Altera recommends using the --cmd add_section_mapping approach when updating your BSP because it allows the default Tcl script to update the default sections mapping if memories are added, removed, renamed, or resized.

Other BSP Tasks

This section covers some other common situations in which the SBT is useful.

Creating a BSP for an Altera Development Board

In some situations, you need to create a BSP separate from any application. Creating a BSP is similar to creating an application. To create a BSP, perform the following steps:

1. Start the Nios II Command Shell.

   For details about the Nios II Command Shell, refer to the *Getting Started from the Command Line* chapter of the *Nios II Software Developer's Handbook*.

2. Create a working directory for your hardware and software projects. The following steps refer to this directory as *<projects>*.

3. Make *<projects>* the current working directory.

4. Find a Nios II hardware example corresponding to your Altera development board. For example, if you have a Stratix® IV development board, you might select *<Nios II EDS install path>*/**examples**/**verilog/niosII_stratixIV_4sgx230/ triple_speed_ethernet_design**.

5. Copy the hardware example to your working directory, using a command such as the following:

   ```
   cp -R /altera/100/nios2eds/examples/verilog\
       /niosII_stratixIV_4sgx230/triple_speed_ethernet_design .↵
   ```

6. Ensure that the working directory and all subdirectories are writable by typing the following command:

   ```
   chmod -R +w .↵
   ```

   The *<projects>* directory contains a subdirectory named **software_examples**/**bsp**. The **bsp** directory contains several BSP example directories, such as **hal_default**. Select the directory containing an appropriate BSP, and make it the current working directory.

   For a description of the example BSPs, refer to "Nios II Design Example Scripts" in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

7. Create and build the BSP with the **create-this-bsp** script by typing the following command:

   ```
   ./create-this-bsp↵
   ```

Now you have a BSP, with which you can create and build an application.

☞ Altera recommends that you examine the contents of the **create-this-bsp** script. It is a helpful example if you are creating your own script to build a BSP. **create-this-bsp** calls **nios2-bsp** with a few command-line options to create a customized BSP, and then calls `make` to build the BSP.

Querying Settings

If you need to write a script that gets some information from the BSP settings file, use the **nios2-bsp-query-settings** utility. To maintain compatibility with future releases of the Nios II EDS, avoid developing your own code to parse the BSP settings file.

If you want to know the value of one or more settings, run **nios2-bsp-query-settings** with the appropriate command-line options. This command sends the values of the settings you requested to `stdout`. Just capture the output of `stdout` in some variable in your script when you call **nios2-bsp-query-settings**. By default, the output of **nios2-bsp-query-settings** is an ordered list of all option values. Use the `-show-names` option to display the name of the setting with its value.

For details about the **nios2-bsp-query-settings** command-line options, refer to the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Managing Device Drivers

The Nios II SBT creates an **alt_sys_init.c** file. By default, the SBT assumes that if a device is connected to the Nios II processor, and a driver is available, the BSP must include the most recent version of the driver. However, you might want to use a different version of the driver, or you might not want a driver at all (for example, if your application accesses the device directly).

The SBT includes BSP Tcl commands to manage device drivers. With these commands you can control which driver is used for each device. When the **alt_sys_init.c** file is generated, it is set up to initialize drivers as you have requested.

If you are using **nios2-bsp**, you disable the driver for the `uart0` device as follows:

```
nios2-bsp hal my_bsp --cmd set_driver none uart0↵
```

Use the `--cmd` option to call a Tcl command on the command line. The **nios2-bsp-create-settings** command also supports the `--cmd` option. Alternatively, you can put the `set_driver` command in a Tcl script and pass the script to **nios2-bsp** or **nios2-bsp-create-settings** with the `--script` option.

You replace the default driver for `uart0` with a specific version of a driver as follows:

```
nios2-bsp hal my_bsp --cmd set_driver altera_avalon_uart:6.1 uart0↵
```

Creating a Custom Version of newlib

The Nios II EDS comes with a number of precompiled libraries. These libraries include the newlib libraries (**libc.a** and **libm.a**). The Nios II SBT allows you to create your own custom compiled version of the newlib libraries.

To create a custom compiled version of newlib, set a BSP setting to the desired compiler flags. If you are using **nios2-bsp**, type the following command:

```
nios2-bsp hal my_bsp --set hal.custom_newlib_flags "-O0 -pg"↵
```

Because newlib uses the open source **configure** utility, its build flow differs from other files in the BSP. When **Makefile** builds the BSP, it runs the **configure** utility. The **configure** utility creates a makefile in the build directory, which compiles the newlib source. The newlib library files are copied to the BSP directory named newlib. The newlib source files are not copied to the BSP.

☞ The Nios II SBT recompiles newlib whenever you introduce new compiler flags. For example, if you use compiler flags to add floating point math hardware support, newlib is recompiled to use the hardware. Recompiling newlib might take several minutes.

Controlling the stdio Device

The build tools offer several ways to control the details of your stdio device configuration, such as the following:

■ To prevent a default stdio device from being chosen, use the following command:

    nios2-bsp hal my_bsp --default_stdio none↵

■ To override the default stdio device and replace it with uart1, use the following command:

    nios2-bsp hal my_bsp --default_stdio uart1↵

■ To override the stderr device and replace it with uart2, while allowing the default Tcl script to choose the default stdout and stdin devices, use the following command:

    nios2-bsp hal my_bsp --set hal.stderr uart2↵

In all these cases, if you run **nios2-bsp** again to update your BSP, you must provide the original command-line options again to prevent the default Tcl script from choosing its own default stdio devices. Alternatively, you can call --default_stdio with the DONT_CHANGE keyword to prevent the default Tcl script from changing the stdio device settings.

Configuring Optimization and Debugger Options

By default, the Nios II SBT creates your project with the correct compiler options for debugging environments. These compiler options turn off code optimization, and generate a symbol table for the debugger.

You can control the optimization and debug level through the project makefile, which determines the compiler options. Example 4–5 illustrates how a typical application makefile specifies the compiler options.

Example 4–5. Default Application Makefile Settings

```
APP_CFLAGS_OPTIMIZATION := -O0
APP_CFLAGS_DEBUG_LEVEL := -g
```

When your project is fully debugged and ready for release, you might want to enable optimization and omit the symbol table, to achieve faster, smaller executable code. To enable optimization and turn off the symbol table, edit the application makefile to contain the symbol definitions shown in Example 4–6. The absence of a value on the right hand side of the APP_CFLAGS_DEBUG_LEVEL definition causes the compiler to omit generating a symbol table.

Example 4–6. Application Makefile Settings with Optimization

```
APP_CFLAGS_OPTIMIZATION := -O3
APP_CFLAGS_DEBUG_LEVEL :=
```

☞ When you change compiler options in a makefile, before building the project, run `make clean` to ensure that all sources are recompiled with the correct flags. For further information about makefile editing and `make clean`, refer to "Applications and Libraries" on page 4–5.

You individually specify the optimization and debug level for the application and BSP projects, and any user library projects you might be using. You use the BSP settings `hal.make.bsp_cflags_debug` and `hal.make.bsp_cflags_optimization` to specify the optimization and debug level in a BSP, as shown in Example 4–7.

Example 4–7. Configuring a BSP for Debugging

```
nios2-bsp hal my_bsp --set hal.make.bsp_cflags_debug -g \
  --set hal.make.bsp_cflags_optimization -O0↵
```

Alternatively, you can manipulate the BSP settings with a Tcl script.

You can easily copy an existing BSP and modify it to create a different build configuration. For details, refer to "Copying, Moving, or Renaming a BSP" on page 4–10.

To change the optimization and debug level for a user library, use the same procedure as for an application.

☞ Normally you must set the optimization and debug levels the same for the application, the BSP, and all user libraries in a software project. If you mix settings, you cannot debug those components which do not have debug settings. For example, if you compile your BSP with the `-O0` flag and without the `-g` flag, you cannot step into the newlib `printf()` function.

Details of BSP Creation

BSP creation is the same in the Nios II SBT for Eclipse as at the command line. Figure 4–1 shows how the SBT creates a BSP. The **nios2-bsp-create-settings** utility creates a new BSP settings file. For detailed information about BSP settings files, refer to "BSP Settings File Creation" on page 4–22.

nios2-bsp-generate-files creates the BSP files. The **nios2-bsp-generate-files** utility places all source files in your BSP directory. It copies some files from the Nios II EDS installation directory. Others, such as **system.h** and **Makefile**, it generates dynamically.

The SBT manages copied files slightly differently from generated files. If a copied file (such as a HAL source file) already exists, the tools check the file timestamp against the timestamp of the file in the Nios II EDS installation. The tools do not replace the BSP file unless it differs from the distribution file. The tools normally overwrite generated files, such as the BSP **Makefile**, **system.h**, and **linker.x**, unless you have disabled generation of the individual file with the `set_ignore_file` Tcl command or the **Enable File Generation** tab in the BSP Editor. A comment at the top of each generated file warns you not to edit it.

> For information about `set_ignore_file` and other SBT Tcl commands, refer to "Software Build Tools Tcl Commands" in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

> Avoid modifying BSP files. Use BSP settings, or custom device drivers or software packages, to customize your BSP.

Figure 4–1. Nios II SBT BSP Creation
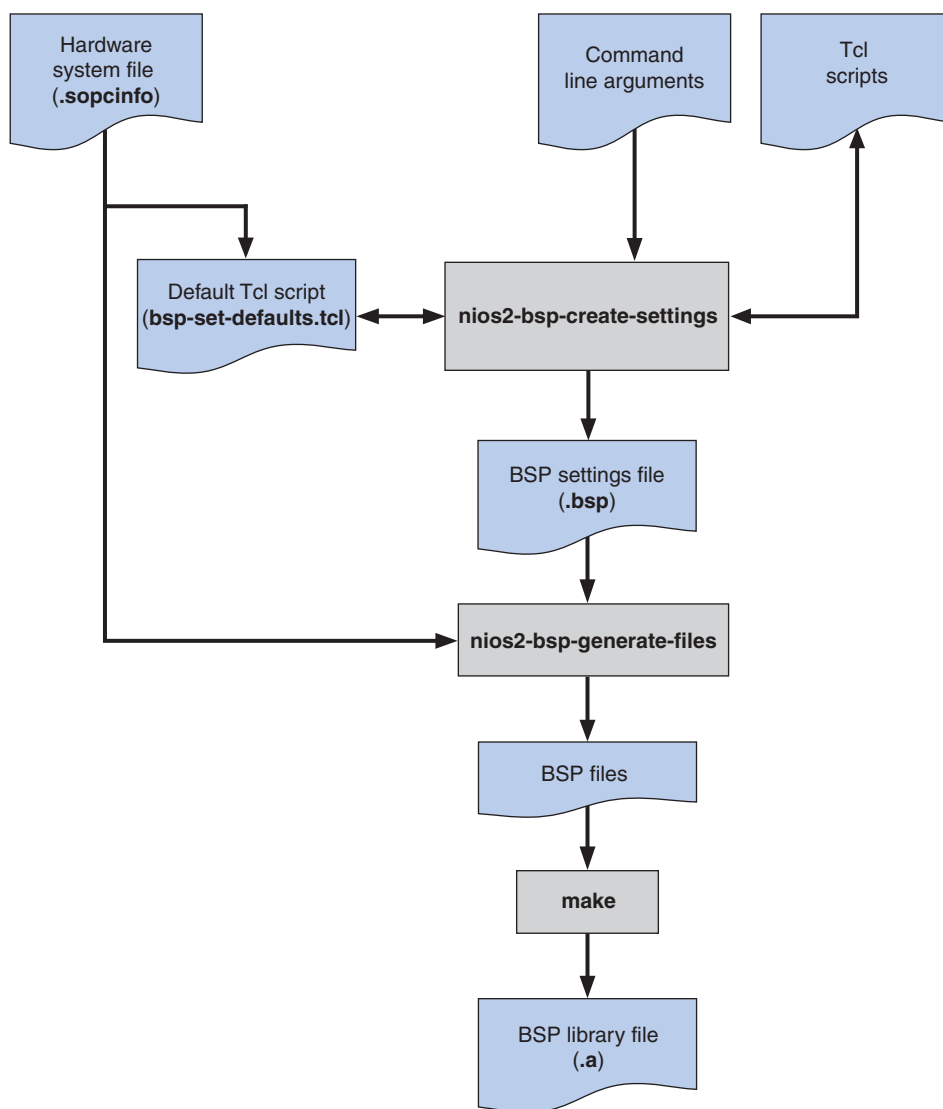
> **CAUTION** Nothing prevents you from modifying a BSP generated file. However, after you do so, it becomes difficult to update your BSP to match changes in your hardware system. If you regenerate your BSP, your previous changes to the generated file are destroyed.

> For information about regenerating your BSP, refer to "Revising Your BSP" on page 4–30.

BSP Settings File Creation

Each BSP has an associated settings file that saves the values of all BSP settings. The BSP settings file is in extensible markup language (XML) format and has a **.bsp** extension by convention. When you create or update your BSP, the Nios II SBT writes the value of all settings to the settings file.

Figure 4–1 on page 4–21 shows that the default Tcl script and **nios2-bsp-generate-files** both use the **.sopcinfo** file. The BSP settings file does not need to duplicate system information (such as base addresses of devices), because the **nios2-bsp-generate-files** utility has access to the .**sopcinfo** file.

Figure 4–2 shows how the Nios II SBT interacts with the BSP settings file. The **nios2-bsp-create-settings** utility creates a new BSP settings file. The **nios2-bsp-update-settings** utility updates an existing BSP settings file. The **nios2-bsp-query-settings** utility reports the setting values in an existing BSP settings file. The **nios2-bsp-generate-files** utility generates a BSP from the BSP settings file.

Figure 4–2. BSP Settings File and BSP Utilities
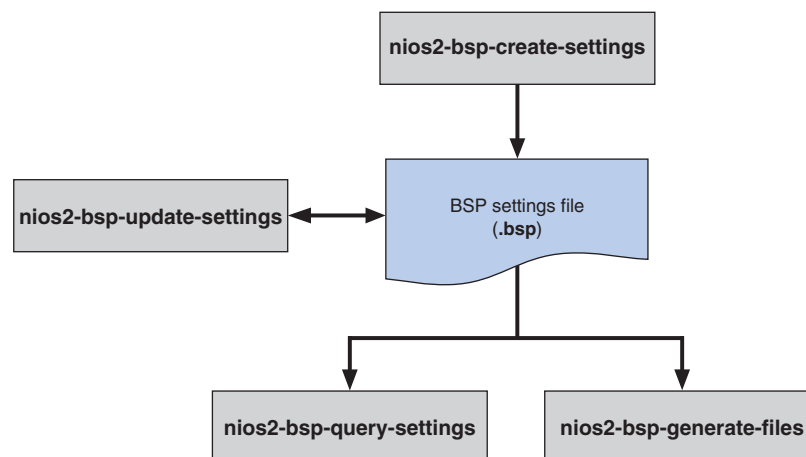
Generated and Copied Files

To understand how to build and modify Nios II C/C++ projects, it is important to understand the difference between copied and generated files.

A copied file is installed with the Nios II EDS, and copied to your BSP directory when you create your BSP. It does not replace the BSP file unless it differs from the distribution file.

A generated file is dynamically created by the **nios2-bsp-generate-files** utility. Generated files reside in the top-level BSP directory. BSP files are normally written every time **nios2-bsp-generate-files** runs.

You can disable generation of any BSP file in the BSP Editor, or on the command line with the `set_ignore_file` Tcl command. Otherwise, if you modify a BSP file, it is destroyed when you regenerate the BSP.

HAL BSP Files and Folders

The Nios II SBT creates the HAL BSP directory in the location you specify. Figure 4–3 shows a BSP directory after the SBT creates a BSP and generates BSP files. The SBT places generated files in the top-level BSP directory, and copied files in the **HAL** and **drivers** directories.

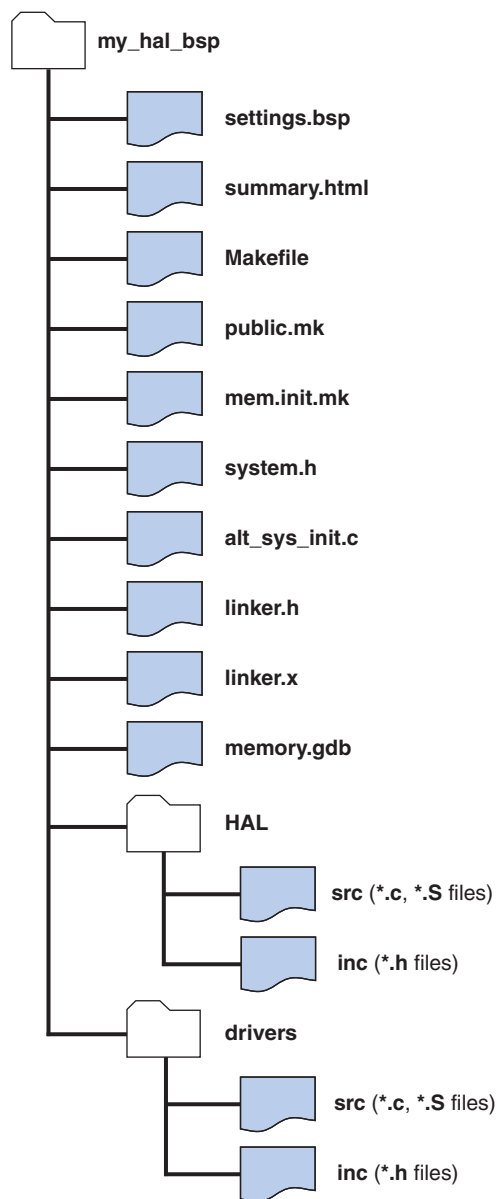**Figure 4–3. HAL BSP After Generating Files**

Table 4–7 details all the generated BSP files shown in Figure 4–3.

Table 4–7. Generated BSP Files

File Name	Function
settings.bsp	Contains all BSP settings. This file is coded in XML. On the command line, **settings.bsp** is created by the **nios2-bsp-create-settings** command, and optionally updated by the **nios2-bsp-update-settings** command. The **nios2-bsp-query-settings** command is available to parse information from the settings file for your scripts. The **settings.bsp** file is an input to **nios2-bsp-generate-files**. The Nios II SBT for Eclipse provides equivalent functionality.
summary.html	Provides summary documentation of the BSP. You can view **summary.html** with a hypertext viewer or browser, such as **Internet Explorer** or **Firefox**. If you change the **settings.bsp** file, the SBT updates the **summary.html** file the next time you regenerate the BSP.
Makefile	Used to build the BSP. The targets you use most often are `all` and `clean`. The `all` target (the default) builds the **libhal_bsp.a** library file. The `clean` target removes all files created by a `make` of the `all` target.
public.mk	A makefile fragment that provides public information about the BSP. The file is designed to be included in other makefiles that use the BSP, such as application makefiles. The BSP **Makefile** also includes **public.mk**.
mem_init.mk	A makefile fragment that defines targets and rules to convert an application executable file to memory initialization files (**.dat**, .**hex**, and .**flash**) for HDL simulation, flash programming, and initializable FPGA memories. The **mem_init.mk** file is designed to be included by an application makefile. For usage, refer to any application makefile generated when you run the SBT. For more information, refer to "Creating Memory Initialization Files" on page 4–11.
alt_sys_init.c	Used to initialize device driver instances and software packages. *(1)*
system.h	Contains the C declarations describing the BSP memory map and other system information needed by software applications. *(1)*
linker.h	Contains information about the linker memory layout. **system.h** includes the **linker.h** file.
linker.x	Contains a linker script for the GNU linker.
memory.gdb	Contains memory region declarations for the GNU debugger.
obj Directory	Contains the object code files for all source files in the BSP. The hierarchy of the BSP source files is preserved in the **obj** directory.
libhal_bsp.a Library	Contains the HAL BSP library. All object files are combined in the library file. The HAL BSP library file is always named **libhal_bsp.a**.

Note to Table 4–7:

(1) For further details about this file, refer to the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

Table 4–8 details all the copied BSP files shown in Figure 4–3.

Table 4–8. Copied BSP Files

File Name	Function
HAL Directory	Contains HAL source code files. These are all copied files. The **src** directory contains the C-language and assembly-language source files. The **inc** directory contains the header files. The **crt0.S** source file, containing HAL C run-time startup code, resides in the **HAL/src** directory.
drivers Directory	Contains all driver source code. The files in this directory are all copied files. The **drivers** directory has **src** and **inc** subdirectories like the **HAL** directory.

Figure 4–4 shows a BSP directory after executing **make.**

Figure 4–4. HAL BSP After Build
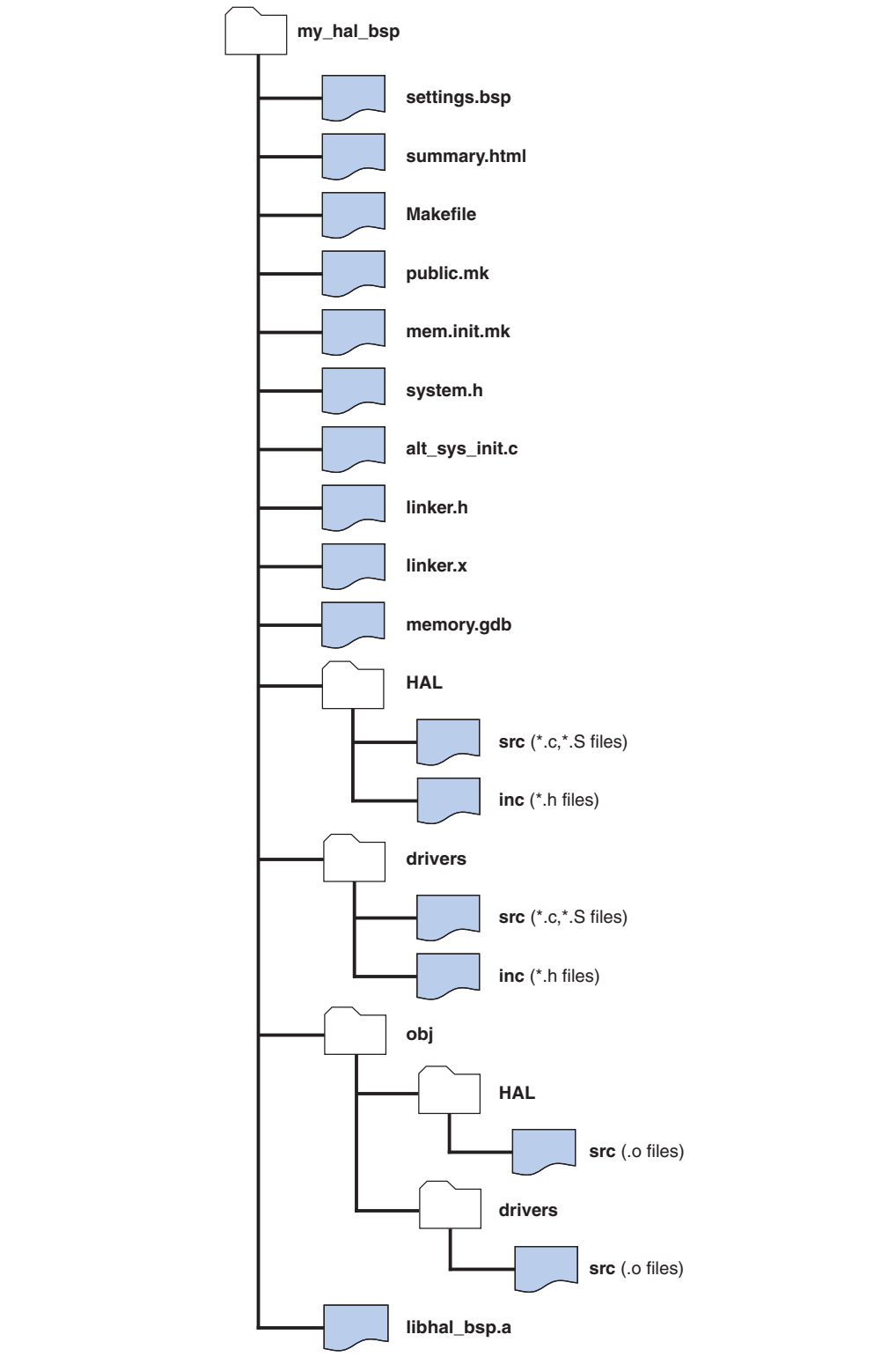
Linker Map Validation

When a BSP is generated, the SBT validates the linker region and section mappings, to ensure that they are valid for a HAL project. The tools display an error in each of the following cases:

- The .entry section maps to a nonexistent region.

- The .entry section maps to a memory region that is less than 32 bytes in length.

- The .entry section maps to a memory region that does not start on the reset vector base address.

- The .exceptions section maps to a nonexistent region.

- The .exceptions section maps to a memory region that does not start on the exception vector base address.

- The .entry section and .exceptions section map to the same device, and the memory region associated with the .exceptions section precedes the memory region associated with the .entry section.

- The .entry section and .exceptions section map to the same device, and the base address of the memory region associated with the .exceptions section is less than 32 bytes above the base address of the memory region associated with the .entry section.

Tcl Scripts for BSP Settings

In many cases, you can fully specify your Nios II BSP with the Nios II SBT settings and defaults. However, in some cases you might need to create some simple Tcl scripts to customize your BSP.

You control the characteristics of your BSP by manipulating BSP settings, using Tcl commands. The most powerful way of using Tcl commands is by combining them in Tcl scripts.

Tcl scripting gives you maximum control over the contents of your BSP. One advantage of Tcl scripts over command-line arguments is that a Tcl script can obtain information from the hardware system or pre-existing BSP settings, and then use it later in script execution.

For descriptions of the Tcl commands used to manipulate BSPs, refer to "Software Build Tools Tcl Commands" in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Calling a Custom BSP Tcl Script

From the Nios II Command Shell, you can call a custom BSP Tcl script with any of the following commands:

```
nios2-bsp --script custom_bsp.tcl

nios2-bsp-create-settings --script custom_bsp.tcl

nios2-bsp-query-settings --script custom_bsp.tcl

nios2-bsp-update-settings --script custom_bsp.tcl
```

In the Nios II BSP editor, you can execute a Tcl script when generating a BSP, through the **New BSP Settings File** dialog box.

For information about using Tcl scripts in the SBT for Eclipse, refer to "Using the BSP Editor" in the *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook.*

For an example of custom Tcl script usage, refer to "Creating Memory Initialization Files" on page 4–11.

Any settings you specify in your script override the BSP default values. For further information about BSP defaults, refer to "Specifying BSP Defaults" on page 4–35.

When you update an existing BSP, you must include any scripts originally used to create it. Otherwise, your project's settings revert to the defaults.

When you use a custom Tcl script to create your BSP, you must include the script in the set of files archived in your version control system. For further information, refer to "Using Version Control" on page 4–9.

The Tcl script in Example 4–8 is a very simple example that sets stdio to a device with the name my_uart.

Example 4–8. Simple Tcl script

```
set default_stdio my_uart
set_setting hal.stdin $default_stdio
set_setting hal.stdout $default_stdio
set_setting hal.stderr $default_stdio
```

Example 4–9 illustrates how you might use more powerful scripting capabilities to customize a BSP based on the contents of the hardware system.

The Nios II SBT uses slave descriptors to refer to components connected to the Nios II processor. A slave descriptor is the unique name of a hardware component's slave port.

If a component has only one slave port connected to the Nios II processor, the slave descriptor is the same as the name of the component (for example, onchip_mem_0). If a component has multiple slave ports connecting the Nios II to multiple resources in the component, the slave descriptor is the name of the component followed by an underscore and the slave port name (for example, onchip_mem_0_s1).

For further information about slave descriptors, refer to the *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

The script shown in Example 4–9 is similar to **bsp-stdio-utils.tcl**, which examines the hardware system and determines what device to use for stdio. For details, refer to "Specifying BSP Defaults" on page 4–35.

Example 4–9. Tcl Script to Examine Hardware and Choose Settings

```
# Select a device connected to the processor as the default STDIO devi
ce.

# It returns the slave descriptor of the selected device.
# It gives first preference to devices with stdio in the name.
# It gives second preference to JTAG UARTs.
# If no JTAG UARTs are found, it uses the last character device.
# If no character devices are found, it returns "none".

# Procedure that does all the work of determining the stdio device
proc choose_default_stdio {} {
    set last_stdio "none"
    set first_jtag_uart "none"

    # Get all slaves attached to the processor.
    set slave_descs [get_slave_descs]

    foreach slave_desc $slave_descs {
        # Lookup module class name for slave descriptor.
        set module_name [get_module_name $slave_desc]
        set module_class_name [get_module_class_name $module_name]

        # If the module_name contains "stdio", we choose it
        # and return immediately.
        if { [regexp .*stdio.* $module_name] } {
            return $slave_desc
        }

        # Assume it is a JTAG UART if the module class name contains
        # the string "jtag_uart".  In that case, return the first one
        # found.
        if { [regexp .*jtag_uart.* $module_class_name] } {
            if {$first_jtag_uart == "none"} {
                set first_jtag_uart $slave_desc
            }
        }

        # Track last character device in case no JTAG UARTs found.
        if { [is_char_device $slave_desc] } {
            set last_stdio $slave_desc
        }
    }

    if {$first_jtag_uart != "none"} {
        return $first_jtag_uart
    }

    return $last_stdio
}

# Call routine to determine stdio
set default_stdio [choose_default_stdio]

# Set stdio settings to use results of above call.
set_setting hal.stdin $default_stdio
set_setting hal.stdout $default_stdio
set_setting hal.stderr $default_stdio
```

Revising Your BSP

Your BSP is customized to your hardware design and your software requirements. If your hardware design or software requirements change, you usually need to revise your BSP.

Every BSP is based on a Nios II processor in a hardware system. The BSP settings file does not duplicate information available in the .**sopcinfo** file, but it does contain system-dependent settings that reference system information. Because of these system-dependent settings, a BSP settings file can become inconsistent with its system if the system changes.

You can revise a BSP at several levels. This section describes each level, and provides guidance about when to use it.

Rebuilding Your BSP

Rebuilding a BSP is the most superficial way to revise a BSP.

What Happens

Rebuilding the BSP simply recreates all BSP object files and the .**a** library file. BSP settings, source files, and compiler options are unchanged.

How to Rebuild Your BSP

In the Nios II SBT for Eclipse, right-click the BSP project and click **Build**.

On the command line, change to the BSP directory and type `make`.

Regenerating Your BSP

Regenerating the BSP refreshes the BSP source files without updating the BSP settings.

What Happens

Regenerating a BSP has the following effects:

■ Reads the .**sopcinfo** file for basic system parameters such as module base addresses and clock frequencies.

■ Retrieves the current system identification (ID) from the .**sopcinfo** file. Ensures that the correct system ID is inserted in the .**elf** file the next time the BSP is built.

■ Adjusts the default memory map to correspond to changes in memory sizes. If you are using a custom memory map, it is untouched.

■ Retains all other existing settings in the BSP settings file.

☞ Except for adjusting the default memory map, the SBT does not ensure that the settings are consistent with the hardware design in the .**sopcinfo** file.

■ Ensures that the correct set of BSP files is present, as follows:

■ Copies all required source files to the BSP directory tree. Copied BSP files are listed in Table 4–8 on page 4–25.

If a copied file (such as a HAL source file) already exists, the SBT checks the file timestamp against the timestamp of the file in the Nios II EDS installation. The tools do not replace the BSP file unless it differs from the distribution file.

■ Recreates all generated files. Generated BSP files are listed in Table 4–7 on page 4–24.

☞ You can disable generation of any BSP file in the BSP Editor, or on the command line with the `set_ignore_file` Tcl command. Otherwise, changes you make to a BSP file are lost when you regenerate the BSP. Whenever possible, use BSP settings, or custom device drivers or software packages, to customize your BSP.

■ Removes any files that are not required, for example, source files for drivers that are no longer in use.

When to Regenerate Your BSP

Regenerating your BSP is required (and sufficient) in the following circumstances:

■ You change your hardware design, but all BSP system-dependent settings remain consistent with the new **.sopcinfo** file. The following are examples of system changes that do not affect BSP system-dependent settings:

■ Changing a component's base address

■ With the internal interrupt controller (IIC), adding or removing hardware interrupts

■ With the IIC, changing a hardware interrupt number

■ Changing a clock frequency

■ Changing a simple processor option, such as cache size or core type

■ Changing a simple component option, other than memory size.

■ Adding a bridge

■ Adding a new component

■ Removing or renaming a component, other than a memory component, the `stdio` device, or the system timer device

■ Changing the size of a memory component when you are using the default memory map

☞ Unless you are sure that your modified hardware design remains consistent with your BSP settings, update your BSP as described in "Updating Your BSP" on page 4–32.

- You want to eliminate any customized source files and revert to the distributed BSP code.

  ☞ To revert to the distributed BSP code, you must ensure that you have not disabled generation on any BSP files.

- You have installed a new version of the Nios II EDS, and you want the updated BSP software implementations.

- When you attempt to rebuild your project, an error message indicates that the BSP must be updated.

- You have updated or recreated the BSP settings file.

How to Regenerate Your BSP

You can regenerate your BSP in the Nios II SBT for Eclipse, or with SBT commands at the command line.

Regenerating Your BSP in Eclipse

In the Nios II SBT for Eclipse, right-click the BSP project, point to **Nios II**, and click **Generate BSP**.

☞ For information about generating a BSP with the SBT for Eclipse, refer to the *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook.*

Regenerating Your BSP from the Command Line

From the command line, use the **nios2-bsp-generate-files** command.

☞ For information about the **nios2-bsp-generate-files** command, refer to the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook.*

Updating Your BSP

When you update a BSP, you recreate the BSP settings file based on the current hardware definition and previous BSP settings.

☞ You must always regenerate your BSP after updating the BSP settings file.

What Happens

Updating a BSP has the following effects:

- System-dependent settings are derived from the original BSP settings file, but adjusted to correspond with any changes in the hardware system.

- Non-system-dependent BSP settings persist from the original BSP settings file.

👣 Also refer to "Regenerating Your BSP" on page 4–30 for actions taken when you regenerate the BSP after updating it.

When to Update Your BSP

Updating your BSP is necessary in the following circumstances:

■ A change to your BSP settings is required.

■ Changes to your .**sopcinfo** file make it inconsistent with your BSP. The following are examples of system changes that affect BSP system-dependent settings:

   ■ Renaming the processor

   ■ Renaming or removing a memory, the `stdio` device, or the system timer device

   ■ Changing the size of a memory component when using a custom memory map

   ■ Changing the processor reset or exception slave port or offset

   ■ Adding or removing an external interrupt controller (EIC)

   ■ Changing the parameters of an EIC

■ When you attempt to rebuild your project, an error message indicates that you must update the BSP.

How to Update Your BSP

You can update your BSP at the command line. You have the option to use a Tcl script to control your BSP settings.

From the command line, use the **nios2-bsp-update-settings** command. You can use the `--script` option to define the BSP with a Tcl script.

For details about the **nios2-bsp-update-settings** command, refer to the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

nios2-bsp-update-settings does not reapply default settings unless you explicitly call the top-level default Tcl script with the `--script` option.

For information about using the default Tcl script, refer to "Specifying BSP Defaults" on page 4–35.

Alternatively, you can update your BSP with the **nios2-bsp** script. **nios2-bsp** determines that your BSP already exists, and uses the **nios2-bsp-update-settings** command to update the BSP settings file.

The **nios2-bsp** script executes the default Tcl script every time it runs, overwriting previous default settings. If you want to preserve all settings, including the default settings, use the `DONT_CHANGE` keyword, described in "Top Level Tcl Script for BSP Defaults" on page 4–36. Alternatively, you can provide **nios2-bsp** with command-line options or Tcl scripts to override the default settings.

For information about using the **nios2-bsp** script, refer to the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Recreating Your BSP

When you recreate your BSP, you start over as if you were creating a new BSP.

☞ After you recreate your BSP, you must always regenerate it.

What Happens

Recreating a BSP has the following effects:

- System-dependent settings are created based on the current hardware system.

- Non-system-dependent settings can be selected by the default Tcl script, by values you specify, or both.

Also refer to "Regenerating Your BSP" on page 4–30 for actions taken when you generate the BSP after recreating it.

When to Recreate Your BSP

If you are working exclusively in the Nios II SBT for Eclipse, and you modify the underlying hardware design, the best practice is to create a new BSP. Creating a BSP is very easy with the SBT for Eclipse. Manually correcting a large number of interrelated settings, on the other hand, can be difficult.

How to Recreate Your BSP

You can recreate your BSP in the Nios II SBT for Eclipse, or using the SBT at the command line. Regardless which method you choose, you can use Tcl scripts to control and reproduce your BSP settings. This section describes the options for recreating BSPs.

Using Tcl Scripts When Recreating Your BSP

A Tcl script automates selection of BSP settings. This automation ensures that you can reliably update or recreate your BSP with its original settings. Except when creating very simple BSPs, Altera recommends specifying all BSP settings with a Tcl script.

To use Tcl scripts most effectively, it is best to create a Tcl script at the time you initially create the BSP. However, the BSP Editor enables you to export a Tcl script from an existing BSP.

👣 For details about exporting Tcl scripts, refer to "Using the BSP Editor" in the *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook*.

By recreating the BSP settings file with a Tcl script that specifies all BSP settings, you can reproduce the original BSP while ensuring that system-dependent settings are adjusted correctly based on any changes in the hardware system.

👣 For information about Tcl scripting with the SBT, refer to "Tcl Scripts for BSP Settings" on page 4–27.

Recreating Your BSP in Eclipse

The process for recreating a BSP is the same as the process for creating a new BSP. The Nios II SBT for Eclipse provides an option to import a Tcl script when creating a BSP.

For details, refer to "Getting Started with Eclipse" and "Using the BSP Editor" in the *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook.*

Recreating Your BSP at the Command Line

Recreate your BSP using the **nios2-bsp-create-settings** command. You can use the `--script` option to define the BSP with a Tcl script.

The **nios2-bsp-create-settings** command does not apply default settings to your BSP. However, you can use the `--script` command-line option to run the default Tcl script. For information about the default Tcl script, refer to "Specifying BSP Defaults".

For information about using the **nios2-bsp-create-settings** command, refer to the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

Specifying BSP Defaults

The Nios II SBT sets BSP defaults using a set of Tcl scripts. Table 4–9 lists the components of the BSP default Tcl scripts included in the Nios II SBT. These scripts specify default BSP settings. The scripts are located in the following directory:

<Nios II EDS install path>/**sdk2/bin**

Table 4–9. Default Tcl Script Components

Script	Level	Summary
bsp-set-defaults.tcl	Top-level	Sets system-dependent settings to default values.
bsp-call-proc.tcl	Top-level	Calls a specified procedure in one of the helper scripts.
bsp-stdio-utils.tcl	Helper	Specifies `stdio` device settings.
bsp-timer-utils.tcl	Helper	Specifies system timer device setting.
bsp-linker-utils.tcl	Helper	Specifies memory regions and section mappings for linker script.
bsp-bootloader-utils.tcl	Helper	Specifies boot loader-related settings.

For more information about Tcl scripting with the SBT, refer to "Tcl Scripts for BSP Settings" on page 4–27.

The Nios II SBT uses the default Tcl scripts to specify default values for system-dependent settings. System-dependent settings are BSP settings that reference system information in the .**sopcinfo** file.

The SBT executes the default Tcl script before any user-specified Tcl scripts. As a result, user input overrides settings made by the default Tcl script.

You can pass command-line options to the default Tcl script to override the choices it makes or to prevent it from making changes to settings. For details, refer to "Top Level Tcl Script for BSP Defaults".

The default Tcl script makes the following choices for you based on your hardware system:

■ `stdio` character device

■ System timer device

■ Default linker memory regions

■ Default linker sections mapping

■ Default boot loader settings

The default Tcl scripts use slave descriptors to assign devices.

Top Level Tcl Script for BSP Defaults

The top level Tcl script for setting BSP defaults is **bsp-set-defaults.tcl**. This script specifies BSP system-dependent settings, which depend on the hardware system. The **nios2-bsp-create-settings** and **nios2-bsp-update-settings** utilities do not call the default Tcl script when creating or updating a BSP settings file. The `--script` option must be used to specify **bsp-set-defaults.tcl** explicitly. Both the Nios II SBT for Eclipse and the **nios2-bsp** script call the default Tcl script by invoking either **nios2-bsp-create-settings** or **nios2-bsp-update-settings** with the `--script bsp-set-defaults.tcl` option.

The default Tcl script consists of a top-level Tcl script named **bsp-set-defaults.tcl** plus the helper Tcl scripts listed in Table 4–9. The helper Tcl scripts do the real work of examining the **.sopcinfo** file and choosing appropriate defaults.

The **bsp-set-defaults.tcl** script sets the following defaults:

■ `stdio` character device (**bsp-stdio-utils.tcl**)

■ System timer device (**bsp-timer-utils.tcl**)

■ Default linker memory regions (**bsp-linker-utils.tcl**)

■ Default linker sections mapping (**bsp-linker-utils.tcl**)

■ Default boot loader settings (**bsp-bootloader-utils.tcl**)

You run the default Tcl script on the **nios2-bsp-create-settings**, **nios2-bsp-query-settings**, or **nios2-bsp-update-settings** command line, by using the `--script` argument. It has the following usage:

```
bsp-set-defaults.tcl [<argument name> <argument value>]*
```

Table 4–10 lists default Tcl script arguments in detail. All arguments are optional. If present, each argument must be in the form of a name and argument value, separated by white space. All argument values are strings. For any argument not specified, the corresponding helper script chooses a suitable default value. In every case, if the argument value is `DONT_CHANGE`, the default Tcl scripts leave the setting unchanged. The `DONT_CHANGE` value allows fine-grained control of what settings the default Tcl script changes and is useful when updating an existing BSP.

Table 4–10. Default Tcl Script Command-Line Options

Argument Name	Argument Value
`default_stdio`	Slave descriptor of default `stdio` device (`stdin`, `stdout`, `stderr`). Set to `none` if no `stdio` device desired.
`default_sys_timer`	Slave descriptor of default system timer device. Set to `none` if no system timer device desired.
`default_memory_regions`	Controls generation of memory regions By default, **bsp-linker-utils.tcl** removes and regenerates all current memory regions. Use the `DONT_CHANGE` keyword to suppress this behavior.
`default_sections_mapping`	Slave descriptor of the memory device to which the default sections are mapped. This argument has no effect if `default_memory_regions == DONT_CHANGE`.
`enable_bootloader`	Boolean: 1 if a boot loader is present; 0 otherwise.

Specifying the Default stdio Device

The **bsp-stdio-utils.tcl** script provides procedures to choose a default `stdio` slave descriptor and to set the `hal.stdin`, `hal.stdout`, and `hal.stderr` BSP settings to that value.

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

The script searches the .**sopcinfo** file for a slave descriptor with the string `stdio` in its name. If **bsp-stdio-utils.tcl** finds any such slave descriptors, it chooses the first as the default `stdio` device. If the script finds no such slave descriptor, it looks for a slave descriptor with the string `jtag_uart` in its component class name. If it finds any such slave descriptors, it chooses the first as the default `stdio` device. If the script finds no slave descriptors fitting either description, it chooses the last character device slave descriptor connected to the Nios II processor. If **bsp-stdio-utils.tcl** does not find any character devices, there is no `stdio` device.

Specifying the Default System Timer

The **bsp-timer-utils.tcl** script provides procedures to choose a default system timer slave descriptor and to set the `hal.sys_clk_timer` BSP setting to that value.

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

The script searches the **.sopcinfo** file for a timer component to use as the default system timer. To be an appropriate system timer, the component must have the following characteristics:

■ It must be a timer, that is, `is_timer_device` must return true.

■ It must have a slave port connected to the Nios II processor.

When the script finds an appropriate system timer component, it sets `hal.sys_clk_timer` to the timer slave port descriptor. The script prefers a slave port whose descriptor contains the string `sys_clk`, if one exists. If no appropriate system timer component is found, the script sets `hal.sys_clk_timer` to `none`.

Specifying the Default Memory Map

The **bsp-linker-utils.tcl** script provides procedures to add the default linker script memory regions and map the default linker script sections to a default region. The **bsp-linker-utils.tcl** script uses the `add_memory_region` and `add_section_mapping` BSP Tcl commands.

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

The script chooses the largest volatile memory region as the default memory region. If there is no volatile memory region, **bsp-linker-utils.tcl** chooses the largest non-volatile memory region. The script assigns the `.text`, `.rodata`, `.rwdata`, `.bss`, `.heap`, and `.stack` section mappings to this default memory region. The script also sets the `hal.linker.exception_stack_memory_region` BSP setting to the default memory region. The setting is available in case the separate exception stack option is enabled (this setting is disabled by default).

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

Specifying Default Bootloader Parameters

The **bsp-bootloader-utils.tcl** script provides procedures to specify the following BSP boolean settings:

■ `hal.linker.allow_code_at_reset`

■ `hal.linker.enable_alt_load_copy_rodata`

■ `hal.linker.enable_alt_load_copy_rwdata`

■ `hal.linker.enable_alt_load_copy_exceptions`

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

The script examines the `.text` section mapping and the Nios II reset slave port. If the `.text` section is mapped to the same memory as the Nios II reset slave port and the reset slave port is a flash memory device, the script assumes that a boot loader is being used. You can override this behavior by passing the `enable_bootloader` option to the default Tcl script.

Table 4–11 shows how the **bsp-bootloader-utils.tcl** script specifies the value of boot loader-dependent settings. If a boot loader is enabled, the assumption is that the boot loader is located at the reset address and handles the copying of sections on reset. If there is no boot loader, the BSP might need to provide code to handle these functions. You can use the `alt_load()` function to implement a boot loader.

Table 4–11. Boot Loader-Dependent Settings

Setting name *(1)*	Value When Boot Loader Enabled	Value When Boot Loader Disabled
`hal.linker.allow_code_at_reset`	0	1
`hal.linker.enable_alt_load_copy_rodata`	0	1 if `.rodata` memory different than `.text` memory and `.rodata` memory is volatile; 0 otherwise
`hal.linker.enable_alt_load_copy_rwdata`	0	1 if `.rwdata` memory different than `.text` memory; 0 otherwise
`hal.linker.enable_alt_load_copy_exceptions`	0	1 if `.exceptions` memory different than `.text` memory and `.exceptions` memory is volatile; 0 otherwise

Notes to Table 4–11:

(1) For further information about these settings, refer to the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook.*

Using Individual Default Tcl Procedures

The default Tcl script consists of the top-level **bsp-call-proc.tcl** script plus the helper scripts listed in Table 4–9 on page 4–35. The procedure call Tcl script allows you to call a specific procedure in the helper scripts, if you want to invoke some of the default Tcl functionality without running the entire default Tcl script.

The procedure call Tcl script has the following usage:

```
bsp-call-proc.tcl <proc-name> [<args>]*
```

bsp-call-proc.tcl calls the specified procedure with the specified (optional) arguments. Refer to the default Tcl scripts to view the available functions and their arguments. The **bsp-call-proc.tcl** script includes the same files as the **bsp-set-defaults.tcl** script, so any function in those included files is available.

Device Drivers and Software Packages

The Nios II SBT can incorporate device drivers and software packages supplied by Altera, supplied by other third-party developers, or created by you.

For details about integrating device drivers and software packages with the Nios II SBT, refer to the *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook.*

Boot Configurations for Altera Embedded Software

The HAL and MicroC/OS-II BSPs support several boot configurations. The default Tcl script configures an appropriate boot configuration based on your hardware system and other settings.

For detailed information about the HAL boot loader process, refer to the *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

Table 4–12 shows the memory types that the default Tcl script recognizes when making decisions about your boot configuration. The default Tcl script uses the `IsFlash` and `IsNonVolatileStorage` properties to determine what kind of memory is in the system.

The `IsFlash` property of the memory module (defined in the .**sopcinfo** file) indicates whether the .**sopcinfo** file identifies the memory as a flash memory device. The `IsNonVolatileStorage` property indicates whether the .**sopcinfo** file identifies the memory as a non-volatile storage device. The contents of a non-volatile memory device are fixed and always present.

☞ Some FPGA memories can be initialized when the FPGA is configured. They are not considered non-volatile because the default Tcl script has no way to determine whether they are actually initialized in a particular system.

Table 4–12. Memory Types

Memory Type	Examples	IsFlash	IsNonVolatileStorage
Flash	Common flash interface (CFI), erasable programmable configurable serial (EPCS) device	true	true
ROM	On-chip memory configured as ROM, HardCopy ROM	false	true
RAM	On-chip memory configured as RAM, HardCopy RAM, SDRAM, synchronous static RAM (SSRAM)	false	false

The following sections describe each supported build configuration in detail. The `alt_load()` facility is HAL code that optionally copies sections from the boot memory to RAM. You can set an option to enable the boot copy. This option only adds the code to your BSP if it needs to copy boot segments. The `hal.enable_alt_load` setting enables `alt_load()` and there are settings for each of the three sections it can copy (such as `hal.enable_alt_load_copy_rodata`). Enabling `alt_load()` also modifies the memory layout specified in your linker script.

Boot from Flash Configuration

The reset address points to a boot loader in a flash memory. The boot loader initializes the instruction cache, copies each memory section to its virtual memory address (VMA), and then jumps to `start`.

This boot configuration has the following characteristics:

- `alt_load()` not called
- No code at reset in executable file

The default Tcl script chooses this configuration when the memory associated with the processor reset address is a flash memory and the `.text` section is mapped to a different memory (for example, SDRAM).

Altera provides example boot loaders for CFI and EPCS memory in the Nios II EDS, precompiled to Motorola S-record Files (**.srec**). You can use one of these example boot loaders, or provide your own.

Boot from Monitor Configuration

The reset address points to a monitor in a nonvolatile ROM or initialized RAM. The monitor initializes the instruction cache, downloads the application memory image (for example, using a UART or Ethernet connection), and then jumps to the entry point provided in the memory image.

This boot configuration has the following characteristics:

- `alt_load()` not called
- No code at reset in executable file

The default Tcl script assumes no boot loader is in use, so it chooses this configuration only if you enable it. To enable this configuration, pass the following argument to the default Tcl script:

`enable_bootloader 1`

If you are using the **nios2-bsp** script, call it as follows:

`nios2-bsp hal my_bsp --use_bootloader 1↵`

Run from Initialized Memory Configuration

The reset address points to the beginning of the application in memory (no boot loader). The reset memory must have its contents initialized before the processor comes out of reset. The initialization might be implemented by using a non-volatile reset memory (for example, flash, ROM, initialized FPGA RAM) or by an external master (for example, another processor) that writes the reset memory. The HAL C run-time startup code (`crt0`) initializes the instruction cache, uses `alt_load()` to copy select sections to their VMAs, and then jumps to `_start`. For each associated section (`.rwdata`, `.rodata`, `.exceptions`), boolean settings control this behavior. The default Tcl scripts set these to default values as described in Table 4–11 on page 4–39.

`alt_load()` must copy the `.rwdata` section (either to another RAM or to a reserved area in the same RAM as the `.text` RAM) if `.rwdata` needs to be correct after multiple resets.

This boot configuration has the following characteristics:

- `alt_load()` called
- Code at reset in executable file

The default Tcl script chooses this configuration when the reset and `.text` memory are the same.

In this boot configuration, when the processor core resets, by default the `.rwdata` section is not reinitialized. Reinitialization would normally be done by a boot loader. However, this configuration has no boot loader, because the software is running out of memory that is assumed to be preinitialized before startup.

If your software has a `.rwdata` section that must be reinitialized at processor reset, turn on the `hal.linker.enable_alt_load_copy_rwdata` setting in the BSP.

Run-time Configurable Reset Configuration

The reset address points to a memory that contains code that executes before the normal reset code. When the processor comes out of reset, it executes code in the reset memory that computes the desired reset address and then jumps to it. This boot configuration allows a processor with a hard-wired reset address to appear to reset to a programmable address.

This boot configuration has the following characteristics:

■ `alt_load()` might be called (depends on boot configuration)

■ No code at reset in executable file

Because the processor reset address points to an additional memory, the algorithms used by the default Tcl script to select the appropriate boot configuration might make the wrong choice. The individual BSP settings specified by the default Tcl script need to be explicitly controlled.

Altera-Provided Embedded Development Tools

This section lists the components of the Nios II SBT, and other development tools that Altera provides for use with the SBT. This section does not describe detailed usage of the tools, but refers you to the most appropriate documentation.

Nios II Software Build Tool GUIs

The Nios II EDS provides the following SBT GUIs for software development:

■ The Nios II SBT for Eclipse

■ The Nios II BSP Editor

■ The Nios II Flash Programmer

Each GUI is primarily a thin layer providing graphical control of the command-line tools described in .

Table 4–13 outlines the correlation between GUI features and the SBT command line.

Table 4–13. Summary of Nios II GUI Tasks

Task	Tool	Feature	Nios II SBT Command Line
Creating an example Nios II program	Nios II SBT for Eclipse	**Nios II Application and BSP from Template** wizard	**create-this-app** script
Creating an application	Nios II SBT for Eclipse	**Nios II Application** wizard	**nios2-app-generate-makefile** utility
Creating a user library	Nios II SBT for Eclipse	**Nios II Library** wizard	**nios2-lib-generate-makefile** utility
Creating a BSP	Nios II SBT for Eclipse	**Nios II Board Support Package** wizard	■ Simple: 　■ **nios2-bsp** script ■ Detailed: 　■ **nios2-bsp-create-settings** utility 　■ **nios2-bsp-generate-files** utility
	BSP Editor	**New BSP Setting File** dialog box	
Modifying an application	Nios II SBT for Eclipse	**Nios II Application Properties** page	**nios2-app-update-makefile** utility
Modifying a user library	Nios II SBT for Eclipse	**Nios II Library Properties** page	**nios2-lib-update-makefile** utility
Updating a BSP	Nios II SBT for Eclipse	**Nios II BSP Properties** page	**nios2-bsp-update-settings** utility **nios2-bsp-generate-files** utility
	BSP Editor	—	
Examining properties of a BSP	Nios II SBT for Eclipse	**Nios II BSP Properties** page	**nios2-bsp-query-settings** utility
	BSP Editor	—	
Programming flash memory	Nios II Flash Programmer	—	**nios2-flash-programmer**
Importing a command-line project	Nios II SBT for Eclipse	**Import** dialog box	—

The Nios II SBT for Eclipse

The Nios II SBT for Eclipse is a configuration of the popular Eclipse development environment, specially adapted to the Nios II family of embedded processors. The Nios II SBT for Eclipse includes Nios II plugins for access to the Nios II SBT, enabling you to create applications based on the Altera HAL, and debug them using the JTAG debugger.

You can launch the Nios II SBT for Eclipse either of the following ways:

■ In the Windows operating system, on the Start menu, point to **Programs** > **Altera** > **Nios II EDS** *<version>*, and click **Nios II** *<version>* **Software Build Tools for Eclipse**.

■ From the Nios II Command Shell, by typing `eclipse-nios2`.

For more information about the Nios II SBT for Eclipse, refer to the *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook.*

The Nios II BSP Editor

You can create or modify a Nios II BSP project with the Nios II BSP Editor, a standalone GUI that also works with the Nios II SBT for Eclipse. You can launch the BSP Editor either of the following ways:

■ From the Nios II menu in the Nios II SBT for Eclipse

■ From the Nios II Command Shell, by typing **nios2-bsp-editor**.

The Nios II BSP Editor enables you to edit settings, linker regions, and section mappings, and to select software packages and device drivers.

The capabilities of the Nios II BSP Editor constitute a large subset of the capabilities of the **nios2-bsp-create-settings**, **nios2-bsp-update-settings**, and **nios2-bsp-generate-files** utilities. Any project created in the BSP Editor can also be created using the command-line utilities.

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

The Nios II Flash Programmer

The Nios II flash programmer allows you to program flash memory devices on a target board. The flash programmer supports programming flash on any board, including Altera development boards and your own custom boards. The flash programmer facilitates programming flash for the following purposes:

■ Executable code and data

■ Bootstrap code to copy code from flash to RAM, and then run from RAM

■ HAL file subsystems

■ FPGA hardware configuration data

You can launch the flash programmer either of the following ways:

■ From the Nios II menu in the Nios II SBT for Eclipse

■ From the Nios II Command Shell, by typing:

```
nios2-flash-programmer-generate↵
```

The Nios II Command Shell

The Nios II Command Shell is a **bash** command-line environment initialized with the correct settings to run Nios II command-line tools. The Nios II EDS includes two versions of the Nios II Command Shell, for the two supported GCC toolchain versions, described in "GNU Compiler Tool Chain".

For information about launching the Nios II Command Shell, refer to the *Getting Started from the Command Line* chapter of the *Nios II Software Developer's Handbook*.

The Nios II Command-Line Commands

This section describes the Altera Nios II command-line tools. You can run these tools from the Nios II Command Shell.

Each tool provides its own documentation in the form of help accessible from the command line. To view the help, open the Nios II Command Shell, and type the following command:

<name of tool> --help↵

GNU Compiler Tool Chain

The Nios II compiler tool chain is based on the standard GNU **GCC** compiler, assembler, linker, and make facilities. Altera provides and supports the standard GNU compiler tool chain for the Nios II processor.

The Nios II EDS includes version GCC 4.7.3 of the GCC toolchain.

For detailed information about installing the Altera Complete Design Suite, refer to the *Altera Software Installation and Licensing Manual*.

GNU tools for the Nios II processor are generally named **nios2-elf-**<*tool name*>. The following list shows some examples:

■ **nios2-elf-gcc**

■ **nios2-elf-as**

■ **nios2-elf-ld**

■ **nios2-elf-objdump**

■ **nios2-elf-size**

The exception is the **make** utility, which is simply named **make**.

The Nios II GNU tools reside in the following location:

■ *<Nios II EDS install path>***/bin/gnu** directory

Refer to the following additional sources of information:

■ For information about managing GCC toolchains in the SBT for Eclipse— "Managing Toolchains in Eclipse" in the *Getting Started with the Graphical User Interface* chapter of the *Nios II Software Developer's Handbook*

■ For information about selecting the toolchain on the command line—the *Getting Started from the Command Line* chapter of the *Nios II Software Developer's Handbook*

■ For a comprehensive list of Nios II GNU tools—the GNU HTML documentation, available at the Nios II Embedded Design Suite Support page of the Altera website

■ For further information about GNU from the Free Software Foundation website (www.gnu.org).

Nios II Software Build Tools

The Nios II SBT utilities and scripts provide the functionality underlying the Nios II SBT for Eclipse. You can create, modify, and build Nios II programs with commands typed at a command line or embedded in a script.

Table 4–14 summarizes the command-line utilities and scripts included in the
Nios II SBT. You can call these utilities and scripts on the command line or from the
scripting language of your choice (such as **perl** or **bash**).

Table 4–14. Nios II SBT Utilities and Scripts

Command	Summary	Utility	Script
nios2-app-generate-makefile	Creates an application makefile	✓	
nios2-lib-generate-makefile	Creates a user library makefile	✓	
nios2-app-update-makefile	Modifies an existing application makefile	✓	
nios2-lib-update-makefile	Modifies an existing user library makefile	✓	
nios2-bsp-create-settings	Creates a BSP settings file	✓	
nios2-bsp-update-settings	Updates the contents of a BSP settings file	✓	
nios2-bsp-query-settings	Queries the contents of a BSP settings file	✓	
nios2-bsp-generate-files	Generates all files for a given BSP settings file	✓	
nios2-bsp	Creates or updates a BSP		✓
create-this-app	Creates an example application project		✓
create-this-bsp	Creates an example BSP project		✓

The Nios II SBT utilities reside in the *<Nios II EDS install path>*/**sdk2/bin** directory.

For further information about the Nios II SBT, refer to the *Getting Started from the Command Line* chapter of the *Nios II Software Developer's Handbook*.

File Format Conversion Tools

File format conversion is sometimes necessary when passing data from one utility to
another. Table 4–15 shows the Altera-provided utilities for converting file formats.

Table 4–15. File Conversion Utilities

Utility	Description
bin2flash	Converts binary files to a Nios II Flash Programmer File (**.flash**) for programming to flash memory.
elf2dat	Converts a .**elf** file to a .**dat** file format appropriate for Verilog HDL hardware simulators.
elf2flash	Converts a .**elf** file to a .**flash** file for programming to flash memory.
elf2hex	Converts a .**elf** file to a Hexadecimal (Intel-format) File (**.hex**).
elf2mem	Generates the memory contents for the memory devices in a specific Nios II system.
elf2mif	Converts a .**elf** file to a Quartus® II Memory Initialization File (**.mif**).
flash2dat	Converts a .**flash** file to the .**dat** file format appropriate for Verilog HDL hardware simulators.
sof2flash	Converts an SRAM Object File (**.sof**) to a .**flash** file.

The file format conversion tools are in the *<Nios II EDS install path>*/**bin/** directory.

Other Command-Line Tools

Table 4–16 shows other Altera-provided command-line tools for developing Nios II programs.

Table 4–16. Altera Command-Line Tools

Tool	Description
nios2-download	Downloads code to a target processor for debugging or running.
nios2-flash-programmer-generate	Allows multiple files to be converted to **.flash** files, and optionally programs each file to the specified location on a flash device.
nios2-flash-programmer	Programs data to flash memory on the target board.
nios2-gdb-server	Translates GNU debugger (GDB) remote serial protocol packets over Transmission Control Protocol (TCP) to JTAG transactions with a target Nios II processor.
nios2-terminal	Performs terminal I/O with a JTAG UART in a Nios II system
validate_zip	Verifies if a specified zip file is compatible with Altera's read-only zip file system.
nios2-debug	Downloads a program to a Nios II processor and launches the Insight debugger.
nios2-configure-sof	Configures an Altera configurable part. If no explicit **.sof** file is specified, it tries to determine the correct file to use.
jtagconfig	Allows you configure the JTAG server on the host machine. It can also detect a JTAG chain and set up the download hardware configuration.

The command-line tools described in this section are in the *<Nios II EDS install path>*/ **bin/** directory.

Restrictions

The Nios II SBT supports BSPs incorporating the Altera HAL and Micrium MicroC/OS-II only.

Document Revision History

Table 4–17 shows the revision history for this document.

Table 4–17. Document Revision History (Part 1 of 2)

Date	Version	Changes
January 2014	13.1.0	■ Updated GCC4 toolchain from 4.1.2 to GCC 4.7.3. ■ Removed references to Nios II IDE. ■ Removed references to Nios II GCC3.
May 2011	11.0.0	■ Introduction of Qsys system integration tool ■ The GCC 3 toolchain is an optional feature
February 2011	10.1.0	Removed "Referenced Documents" section.

Table 4–17. Document Revision History (Part 2 of 2)

Date	Version	Changes
July 2010	10.0.0	■ Added explanation of the effects of disabled BSP file generation. ■ Described regeneration of BSP with changed memory sizes. ■ Described GCC 4. ■ Described GCC 3 and GCC 4 command shells
November 2009	9.1.0	■ Chapter repurposed and retitled to cover Nios II Software Build Tools functionality applicable to both command line and Eclipse. ■ Describe the Nios II Flash Programmer
March 2009	9.0.0	■ Moved information about Tcl-based device drivers and software packages, formerly in this chapter, to *Developing device Drivers for the Hardware Abstraction Layer*. ■ Described how to work with compiler optimization and debugger settings. ■ Described newlib recompilation. ■ Corrected minor typographical errors.
May 2008	8.0.0	■ Advanced exceptions added to Nios II core. ■ Added instructions for writing instruction-related exception handler. ■ Design examples removed from list.
October 2007	7.2.0	Initial release. Material moved here from former *Nios II Software Build Tools* chapter.