

Introduction

This chapter describes how to use the Nios[®] II software build tools at the command line to create and build software projects. The software build tools are a set of command utilities and scripts that create and build C/C++ application projects, library projects, and board support packages (BSPs). They are helpful if you need a repeatable, scriptable, and archivable process for creating your software product.

The purpose of this chapter is to tell you how to use the Nios II software build tools to create and build your software project. This chapter provides what you need to know to develop the most common kinds of software projects.

The chapter contains the following sections:

- “Overview of the Nios II Software Build Tools Development Flow” on page 4-2
- “Using Nios II Example Design Scripts” on page 4-4
- “Makefiles” on page 4-7
- “Applications and Libraries” on page 4-8
- “Board Support Packages” on page 4-9
- “Common BSP Tasks” on page 4-16
- “Using the Nios II C2H Compiler” on page 4-28
- “Details of BSP Creation” on page 4-29
- “Tcl Scripts for Board Support Package Settings” on page 4-30
- “Calling a Custom BSP Tcl Script” on page 4-30
- “Specifying BSP Defaults” on page 4-36
- “Device Drivers and Software Packages” on page 4-39
- “Boot Configurations” on page 4-40
- “Restrictions” on page 4-42



Read the *Introduction to the Nios II Software Build Tools* chapter of the *Nios II Software Developer’s Handbook* before starting this chapter.

This chapter also assumes you are familiar with the following topics:

- The GNU make utility. Altera recommends you use version 3.79 or later, provided with the Nios II Embedded Design Suite (EDS).
- 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.



For an overview of both Nios II EDS development flows, including the Nios II integrated development environment (IDE) and the Nios II command-line software build tools flow, refer to the *Overview* chapter of the *Nios II Software Developer's Handbook*. This chapter includes an overview of all command-line software build tools provided with the Nios II EDS. You can obtain general information about GNU make from the Free Software Foundation, Inc. (www.gnu.org).

Overview of the Nios II Software Build Tools Development Flow

This section provides an overview of the software build tools development flow.

Before you start using the Nios II software build tools, it is important to understand their scope. You need to 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.

Software Build Process

To create a software project with the Nios II software build tools, 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 example design. 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 Builder system (**.sopcinfo**) file.
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 some of the Nios II software build tools 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 4–9.
4. Create a BSP 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 4–7.

5. Optionally create a user library. If you need to include a custom software library, you collect the library source files in a single directory, and create a 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 4-8](#).
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 4-8](#).
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 4-7](#).

Generators

Generators are sets of tools that create specific parts of your software project. The command utilities and scripts included in the Nios II software build tools combine to form the following generators:

- Nios II BSP generator—A set of tools to create and manage settings for a BSP
- Nios II makefile generator—A set of tools to create makefiles for BSPs, C/C++ applications and libraries

For more information about the generators, refer to [“Applications and Libraries” on page 4-8](#) and [“Board Support Packages” on page 4-9](#).

Utilities and Scripts

The Nios II software build tools consist of utilities and scripts. This section discusses each of these portions of the build tools.

Command-Line Utilities

“Altera Nios II Build Tools” in the [Overview](#) chapter of the *Nios II Software Developer’s Handbook* summarizes the command-line utilities provided by the Nios II software build tools, and their relationships to the generators. You can call these utilities from the command line or from a scripting language of your choice (such as **perl** or **bash**). On Windows, these utilities have a **.exe** extension. The Nios II software build tools reside in the **sdk2/bin** directory under *<Nios II EDS install path>*.




In the Nios II command shell, *<Nios II EDS install path>* is specified by the **SOPC_KIT_NIOS2** environment variable.

The Nios II BSP Editor

Another way to create a BSP project is with Nios II BSP editor. The BSP editor provides a graphical front end that drives the software build tools. To start the Nios II BSP editor, type the following command:

```
nios2-bsp-editor↵
```

 For details about how to use the BSP editor, start the tool and refer to the tool tips.

Scripts

Nios II software build tools scripts implement complex behavior that extends the capabilities provided by the utilities.

Table 4-1 summarizes the scripts provided with the Nios II software build tools, and their relationships to the generators.

Table 4-1. Nios II Software Build Tools Scripts

Command	Summary	Generators	
		BSP	Makefile
<code>nios2-bsp</code>	Creates or updates a BSP	✓	
<code>nios2-c2h-generate-makefile</code>	Creates application makefile fragment for the Nios II G2H Compiler		✓
<code>create-this-app (1)</code>	Creates a software example and builds it	✓	✓
<code>create-this-bsp (1)</code>	Creates a BSP for a specific hardware example design and builds it	✓	✓

Note to Table 4-1:

- (1) There are `create-this-app` scripts for each software example and several `create-this-bsp` scripts for each hardware example design. For more details, refer to “Using Nios II Example Design Scripts” on page 4-4.

Using Nios II Example Design Scripts

The Nios II software build tools include scripts that allow you to create sample BSPs and applications. This section describes each script and its location in the example design directory structure. Each hardware example design in the Nios II EDS includes a **software_examples** directory with **app** and **bsp** subdirectories.

The **bsp** subdirectory contains a variety of example BSPs. Table 4-2 lists all potential BSP examples provided in the **bsp** directory. The **bsp** directory for each hardware example only includes BSP examples supported by the associated hardware example.

Table 4-2. BSP Examples

Example BSP (1)	Summary
<code>hal_reduced_footprint</code>	Hardware abstraction layer (HAL) BSP configured to minimize memory footprint
<code>hal_default</code>	HAL BSP configured with all defaults
<code>hal_zipfs</code>	HAL BSP configured with the Altera® read-only zip file system
<code>ucosii_net</code>	MicroC/OS-II BSP configured with networking
<code>ucosii_net_zipfs</code>	MicroC/OS-II BSP configured with networking and the Altera read-only zip file system
<code>ucosii_net_tse</code>	MicroC/OS-II BSP configured with networking support for the Altera triple-speed Ethernet media access control (MAC)
<code>ucosii_net_tse_zipfs</code>	MicroC/OS-II BSP configured with networking support for the Altera triple-speed Ethernet MAC and the Altera read-only zip file system
<code>ucosii_default</code>	MicroC/OS-II BSP configured with all defaults

Note to Table 4-2:

- (1) Some BSP examples might not be available on some hardware examples.

The `app` subdirectory contains a separate subdirectory for each software example supported by the hardware example, as listed in Table 4-3.

Table 4-3. Application Examples (1)

Application Name	Summary
Hello World	Prints 'Hello from Nios II'
Board Diagnostics	Tests peripherals on the development boards
Count Binary	Displays a running count of 0x00 to 0xff
Hello Freestanding	Prints 'Hello from Nios II' from a free-standing application
Hello MicroC/OS-II	Prints 'Hello from Nios II' using the MicroC/OS-II RTOS
Hello World Small	Prints 'Hello from Nios II' from a small footprint program
Memory Test	Runs diagnostic tests on both volatile and flash memory
Simple Socket Server	Runs a TCP/IP socket server
Web Server	Runs a web server from a file system in flash memory
Zip File System	Reads from a file system in flash memory

Note to Table 4-3:

(1) Some application examples might not be available on some hardware examples, depending on BSP support.

create-this-bsp

Each BSP subdirectory contains a `create-this-bsp` script. The `create-this-bsp` script calls the `nios2-bsp` script to create a BSP in the current directory. The `create-this-bsp` script has a relative path to the directory containing the `.sopcinfo` file. The `.sopcinfo` file resides two directory levels above the directory containing the `create-this-bsp` script.

The `create-this-bsp` script takes no command-line arguments. Your current directory must be the same directory as the `create-this-bsp` script. The exit value is zero on success and one on error.

create-this-app

Each application subdirectory contains a `create-this-app` script. The `create-this-app` script copies the C/C++ application source code to the current directory, runs `nios2-app-generate-makefile` to create a makefile (named **Makefile**), and then runs `make` to build the executable and linking format (`.elf`) file for your application. Each `create-this-app` script uses a particular example BSP. For further information, refer to the script to determine the associated example BSP. If the BSP does not exist when `create-this-app` runs, `create-this-app` calls the associated `create-this-bsp` script to create the BSP.

The `create-this-app` script takes no command-line arguments. Your current directory must be the same directory as the `create-this-app` script. The exit value is zero on success and one on error.

Finding create-this-app and create-this-bsp

The `create-this-app` and `create-this-bsp` scripts are installed with your Nios II example designs. You can easily find them from the following information:

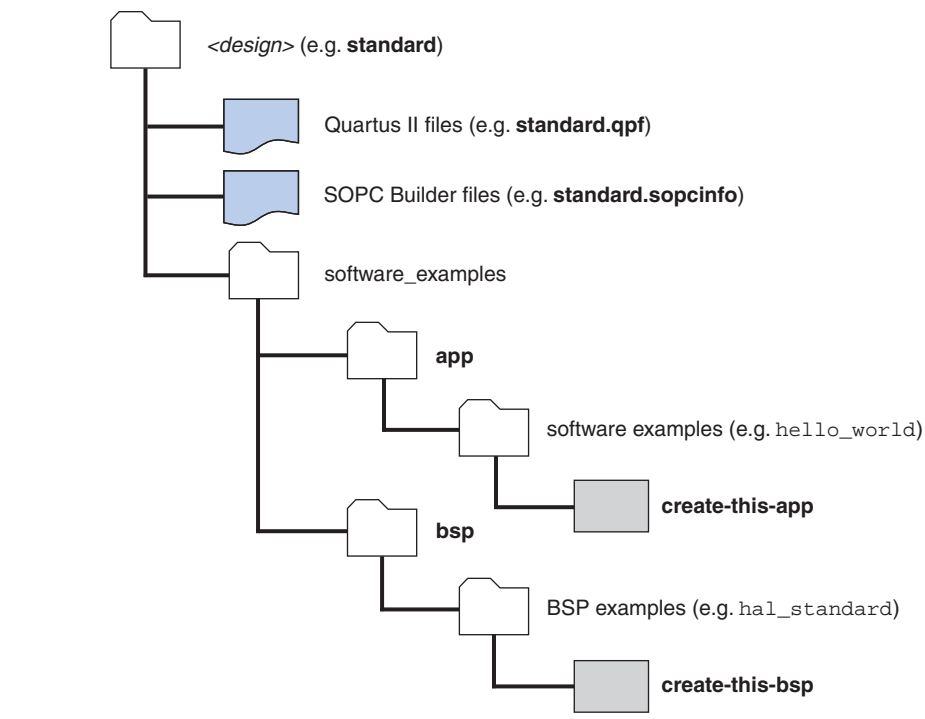
- Where the Nios II EDS is installed
- Which Nios development board you are using
- Which HDL you are using
- Which Nios II hardware example design you are using
- The name of the Nios II software example

The `create-this-app` script for each software example design is in `<Nios II EDS install path>/examples/<HDL>/niosII_<board type>/<design name>/software_examples/app/<example name>`. For example, the `create-this-app` script for the **Hello World** software example running on the Verilog HDL full-featured example design for the Nios II Development Kit, Cyclone® II Edition, might be located in `c:/altera/71/nios2eds/examples/verilog/niosII_cycloneII_2c35/full_featured/software_examples/app/hello_world`.

Similarly, the `create-this-bsp` script for each software example design is in `<Nios II EDS install path>/examples/<HDL>/niosII_<board type>/<design name>/software_examples/bsp/<BSP_type>`. For example, the `create-this-bsp` script for the basic HAL BSP to run on the Verilog HDL full-featured example design for the Nios II Development Kit, Cyclone II Edition, might be located in `c:/altera/71/nios2eds/examples/verilog/niosII_cycloneII_2c35/full_featured/software_examples/bsp/hal_default`.

Figure 4–1 shows the directory structure under each hardware example design.

Figure 4–1. Software Example Design Directory Structure



Makefiles

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

If you choose to create your makefiles by hand, you might still find it helpful to understand how makefile generation works. Letting the software build tools generate makefiles is an excellent way to see examples of powerful makefile usage.

The makefile generators (incorporated in Nios II software build tools) create two kinds of makefiles:

- Application or library makefile—A simple makefile that you can edit by hand with a text editor.
- BSP makefile—A more complex makefile, generated to conform to user-specified settings and the requirements of the target SOPC Builder system.

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

Generated makefiles are platform-independent, calling only commands 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 hints about how to use the makefile generator. 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 library has one makefile, located in the application or library directory.

Makefile Targets

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

Table 4-4. Application Makefile Targets

Target	Operation
<code>help</code>	Displays all available application makefile targets.
<code>all</code> (default)	Builds the associated BSP and libraries, and then builds the application executable file.
<code>app</code>	Builds only the application executable file.
<code>bsp</code>	Builds only the BSP.
<code>libs</code>	Builds only the libraries and the BSP.
<code>clean</code>	Performs a clean build of the application. Deletes all application-related generated files. Leaves associated BSP and libraries alone.
<code>clean_all</code>	Performs a clean build of the application, and associated BSP and libraries (if any).
<code>clean_bsp</code>	Performs a clean build of the BSP.
<code>clean_libs</code>	Performs a clean build of the libraries and the BSP.

Table 4-4. Application Makefile Targets

Target	Operation
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.


You can specify multiple targets on a make command line. For example, the following command removes existing object files in the current project directory, builds the project, downloads the project to a board, and runs it:

```
make clean download-elf
```


Nios II C2H Makefiles

The Nios II software build tools support the Nios II C2H Compiler with the `nios2-c2h-generate-makefile` command. This command creates the C2H makefile fragment, **c2h.mk**, which specifies all accelerators and accelerator options for an application.

`nios2-c2h-generate-makefile` creates a new **c2h.mk** each time it is executed, overwriting the existing **c2h.mk**.

 You must use the `--c2h` flag when calling `nios2-app-generate-makefile` to build your application with the C2H Compiler. This flag causes your application makefile to include the static C2H make rules. These rules in turn include the **c2h.mk** fragment generated by `nios2-c2h-generate-makefile`.

For more details about using the C2H Compiler with the software build tools, refer to [“Using the Nios II C2H Compiler” on page 4-28](#).

 For more details about `nios2-c2h-generate-makefile`, refer to “Nios II Software Build Tools Utilities” in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer’s Handbook*.

Applications and Libraries

The Nios II software build tools have nearly identical support for C/C++ applications and libraries. The support for applications and libraries is very simple. The `nios2-app-generate-makefile` and `nios2-lib-generate-makefile` commands each generate a private makefile (named **Makefile**). The private makefile is used to build the application or library.

The `nios2-lib-generate-makefile` command also generates a public makefile, called **public.mk**. The public makefile is included in the private makefile for any application (or other library) that uses the library.

The private makefile builds one of two types of files:

- A **.elf** file—For an application
- A library archive (**.a**) file—For a library

When you run `nios2-app-generate-makefile` or `nios2-lib-generate-makefile`, you pass it a list of source files and a reference to a BSP directory. The BSP directory, specified with the `--bsp-dir` command-line argument, is mandatory for applications and optional for libraries.

The `nios2-app-generate-makefile` and `nios2-lib-generate-makefile` commands examine the extension of each source file to determine the programming language. Table 4-5 shows the supported programming languages with the corresponding file extensions.

Table 4-5. Supported Source File Types

Programming Language	File Extensions
C	.c
C++	.cpp, .cxx, .cc
Nios II assembly language	.s, .S

If you need to change an application or library makefile after generation, you can edit it using a text editor, or utilities such as `perl` and `sed`.



Normally, after making changes to a makefile, you must run `make clean` before rebuilding your project. However, you can skip this step if you know that your particular Makefile modification does not require a rebuild of every project file. For example, if you simply add a source file or a library, then `make clean` is not necessary. However, if you are unsure whether a partial rebuild is adequate, it is safest to run `make clean`.

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 an SOPC Builder system. The BSP isolates your application from system-specific details such as the memory map, available devices, and processor configuration.

A BSP consists of 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 software build tools support two types of BSPs: Altera 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

You create a BSP with the Nios II BSP generator. This tool provides a great deal of power and flexibility, enabling you to control details of your BSP implementation while maintaining compatibility with an SOPC Builder system which might change.

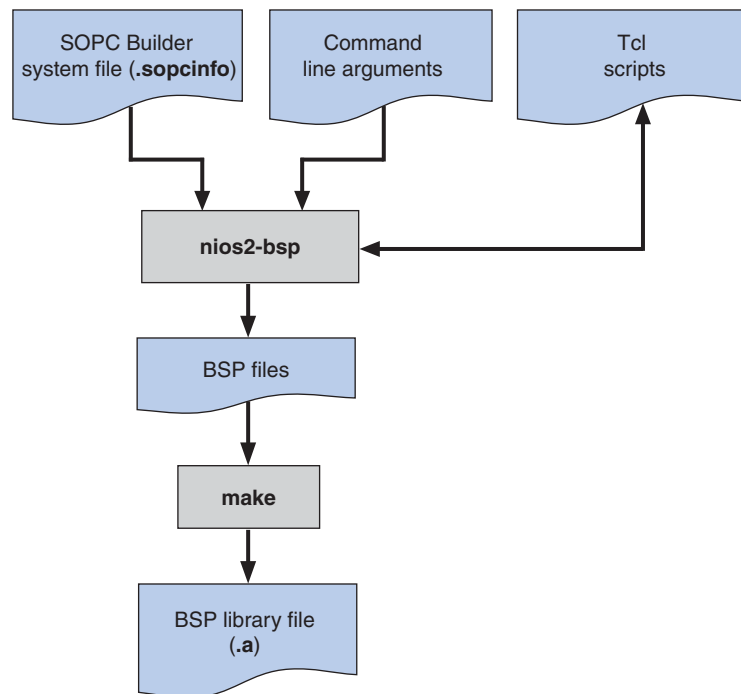
By default, the tools generate a basic BSP for a Nios II system. If you do not need any special features, you can use the default BSP, and skip the remainder of this chapter.

If you require more detailed control over the characteristics of your BSP, Nios II software build tools provide that control. The more features you use, the more complex your commands and scripts become.

The remainder of this section describes how to get the most out of the Nios II software build tools.

Figure 4-2 shows the flow to create a BSP using the `nios2-bsp` command. The `nios2-bsp` script uses the `.sopcinfo` file to create the BSP files. You can override default settings chosen by `nios2-bsp` by supplying command-line arguments, Tcl scripts, or both.

Figure 4-2. `nios2-bsp` Command Flow



`nios2-bsp` places all BSP files in the BSP directory, specified on the command line with argument `--bsp-dir`. After running `nios2-bsp`, you run `make`, which compiles the source code. The result of compilation is the BSP library file, also in the BSP directory. The BSP is ready for incorporation in your application.

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. A copied file is only written if the file does not already exist in your BSP directory. Thus you can freely modify copied files, without losing your changes when you update your BSP.

A generated file is dynamically created by the `nios2-bsp-generate-files` command. A generated file is written every time `nios2-bsp-generate-files` is run. Generated files reside in the top-level BSP directory.

Coordinating with Hardware Changes

If you change your SOPC Builder system, you almost always need to update your BSP. How you update the BSP depends on the nature of the system change. 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. For example, if the `stdio` device is set up to use a module named `uart0` and you rename it to `uart1`, you must update the BSP settings file.

If you are not sure whether the change to your `.sopcinfo` file has introduced inconsistencies with your BSP settings, you can simply rerun `nios2-bsp` to recreate your settings file.

Altera HAL BSP

The Altera HAL is a basic single-threaded run-time environment.

 For more information about the Altera HAL, refer to the *Overview of the Hardware Abstraction Layer* and *Developing Programs Using the Hardware Abstraction Layer* chapters of the *Nios II Software Developer's Handbook*.

HAL BSP Files and Folders

`nios2-bsp-create-settings` creates the HAL BSP directory in the location specified in the `--bsp-dir` argument. [Figure 4-3](#) shows the HAL BSP directory after the `nios2-bsp-create-settings` command creates a settings file, `settings.bsp`, in an empty directory.

Figure 4-3. HAL BSP After Creating Settings



[Figure 4-4](#) shows the `my_hal_bsp` directory after the `nios2-bsp-generate-files` command has generated BSP files. The `nios2-bsp-generate-files` command programmatically generates top-level files from `settings.bsp`. It also copies files to the HAL and `drivers` directories from their installed locations.

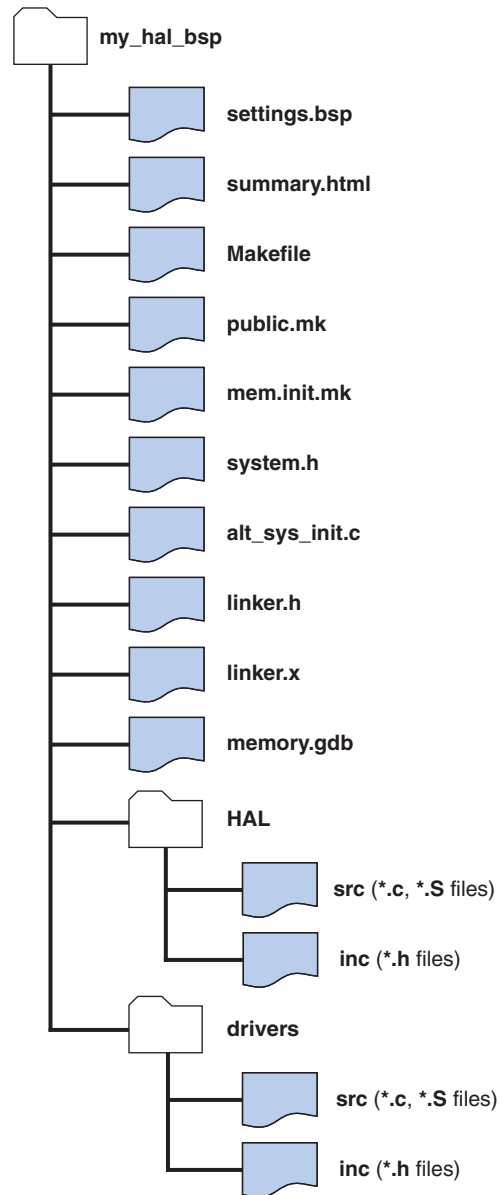
Figure 4-4. HAL BSP After Generating Files

Table 4-6 details all the generated BSP files shown in [Figure 4-4](#).

Table 4-6. Generated BSP Files

File Name	Function
settings.bsp	Contains all BSP settings. This file is coded in extensible markup language (XML). It is created by the <code>nios2-bsp-create-settings</code> command, and optionally updated by the <code>nios2-bsp-update-settings</code> command. It also can be copied from another BSP directory. The <code>nios2-bsp-query-settings</code> command is available to parse information from the settings file for your scripts. The settings.bsp file is an input to <code>nios2-bsp-generate-files</code> .
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 (manually or by running <code>nios2-bsp-update-settings</code>), <code>nios2-bsp-generate-files</code> updates the summary.html file the next time you run it.
Makefile	Used to build the BSP. The targets you use most often are <code>all</code> and <code>clean</code> . The <code>all</code> target (the default) builds the libhal_bsp.a library file. The <code>clean</code> target removes all files created by a <code>make</code> of the <code>all</code> 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 the example application makefile generated when you run <code>nios2-app-generate-makefile</code> .
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.
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.
Note to Table 4-6: (1) For further details about this file, refer to the <i>Developing Programs Using the Hardware Abstraction Layer</i> chapter of the <i>Nios II Software Developer's Handbook</i> .	

Table 4-6. Generated BSP Files

File Name	Function
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 4-6: (1) For further details about this file, refer to the <i>Developing Programs Using the Hardware Abstraction Layer</i> chapter of the <i>Nios II Software Developer's Handbook</i> .	

Figure 4-5 shows the **my_hal_bsp** directory after executing make.

Micrium MicroC/OS-II BSP

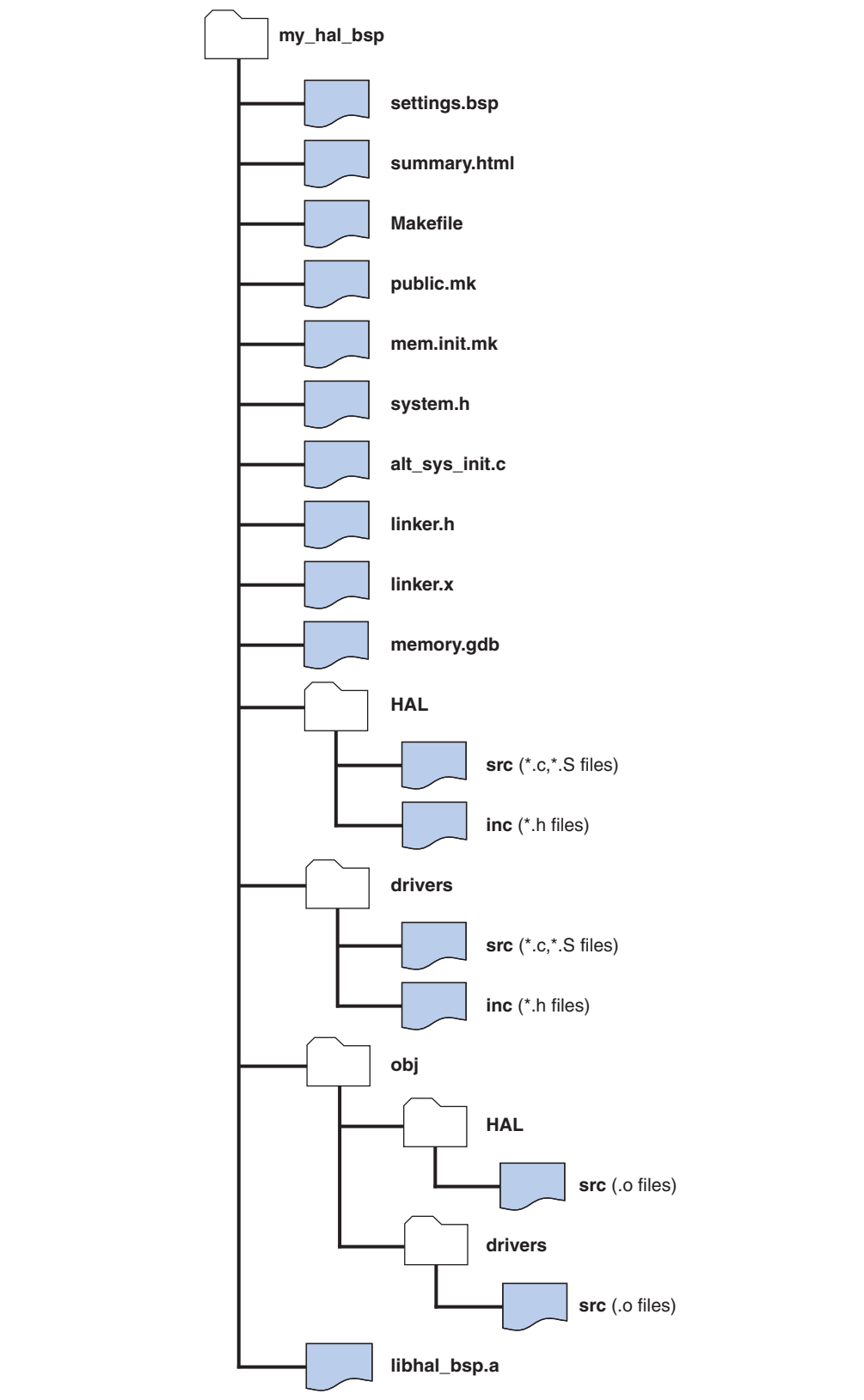
The Micrium MicroC/OS-II is a multi-threaded run-time environment. It is built on the Altera HAL.

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.

Figure 4-5. HAL BSP After Build



Common BSP Tasks

`nios2-bsp` 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 4-16](#)
- [“Copying, Moving, or Renaming a BSP” on page 4-17](#)
- [“Handing Off a BSP” on page 4-18](#)
- [“Creating Memory Initialization Files” on page 4-18](#)
- [“Modifying Linker Memory Regions” on page 4-19](#)
- [“Creating a Custom Linker Section” on page 4-20](#)
- [“Changing the Default Linker Memory Region” on page 4-24](#)
- [“Changing a Linker Section Mapping” on page 4-24](#)
- [“Creating a BSP for a Nios Development Board” on page 4-24](#)
- [“Querying Settings” on page 4-25](#)
- [“Managing Device Drivers” on page 4-26](#)
- [“Creating a Custom Version of Newlib” on page 4-26](#)
- [“Controlling the stdio Device” on page 4-26](#)
- [“Configuring Optimization and Debugger Options” on page 4-27](#)

Although this section describes tasks in terms of the command-line software build tools flow, you can also carry out most of these tasks with the Nios II BSP editor, described in [“The Nios II BSP Editor” on page 4-3](#).

Adding the Nios II Software Build Tools to Your Tool Flow

A common reason for using the software build tools 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 software build tools, 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.

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 BSP generator 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 (**.d**) files have absolute path references. `make clean` removes the **.d** files, as well as linker object (**.o**) files 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 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 then modify the BSP files as needed, or simply run `make` to build the BSP.

Linking and Locating

When autogenerating a HAL BSP, the software build tools make 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 software build tools allow 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 4-7](#) shows the `mem_init.mk` targets. Although the application makefile provides all these targets, it does not build any of them by default. The makefile generator creates the memory initialization files in the application directory (under a directory named `mem_init`). It optionally copies them to your Quartus II project directory and HDL simulation directory, as described in [Table 4-7](#).



The BSP generator 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 (`.sof`) file 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 4-7. mem_init.mk Targets

Target	Operation
mem_init_install	Generates memory initialization files in the application mem_init directory. If the <code>QUARTUS_PROJECT_DIR</code> variable is defined, mem_init.mk copies memory initialization files to your Quartus II project directory named <code>\$(QUARTUS_PROJECT_DIR)</code> . If the <code>SOPC_NAME</code> variable is defined, mem_init.mk copies memory initialization files to your HDL simulation directory named <code>\$(QUARTUS_PROJECT_DIR)/\$(SOPC_NAME)_sim</code> .
mem_init_generate	Generates all memory initialization files in the application mem_init directory.
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 re-sized. 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 4-30](#).

Creating a Custom Linker Section

The Nios II software build tools provide a Tcl command to create a linker section. [Table 4-8](#) lists the default section names.

The default Tcl script creates these default sections for you using the `add_section_mapping` Tcl command.

Table 4-8. Nios II Default Section Names

<code>.entry</code>
<code>.exceptions</code>
<code>.text</code>
<code>.rodata</code>
<code>.rwdata</code>
<code>.bss</code>
<code>.heap</code>
<code>.stack</code>

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

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

When the `nios2-bsp-generate-files` command (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 4-8](#).

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.

Example 4-2. Create `hal_isrs_section.tcl` script

```
# 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 following steps describe the use of this script:

1. Create a working directory for your hardware and software projects. The following steps refer to this directory as *<projects>*.
2. Make *<projects>* the current working directory.
3. Find the full-featured Nios II hardware example corresponding to your Nios development board. For example, if you have a Cyclone II development board, select *<Nios II EDS install path>/examples/verilog/niosII_cycloneII_2c35/full_featured*.

This example uses the Verilog HDL full-featured hardware example design. You can select the language you prefer (Verilog HDL or VHDL)

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

```
cp -R $SOPC_KIT_NIOS2/examples/verilog/niosII_cycloneII_2c35/full_featured .
```

5. 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`. Make this directory the current working directory by typing the following command:

```
cd full_featured/software_examples/bsp
```

- In the `bsp` directory, a subdirectory named `hal_default` contains the `create-this-bsp` script for a default HAL-based BSP. Copy this directory, name the copy `hal_isrs_section`, and change directories to the new directory, by typing the following commands:

```
cp -R hal_default hal_isrs_section
cd hal_isrs_section
```

- Create `isrs_section_script.tcl`, shown in [Example 4-2](#). This script splits off 1 KByte 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`.
- The `<projects>` directory contains a subdirectory named `software_examples/app/tcm`. Make this directory the current working directory by typing the following command:

```
cd ../../app/tcm
```

- Edit the `create-this-app` script. Change occurrences of `hal_default` to `hal_isrs_section`.
- Create and build the application with the `create-this-app` script as follows:

```
./create-this-app
```

- Edit `timer_interrupt_latency.h`. In the `timer_interrupt_latency_irq()` function, change the `.section` directive from `.exceptions` to `.isrs`.
- Rebuild the application by running `make`, as follows:

```
make
```

- After `make` completes successfully, examine the object dump file, `tcm.objdump`, shown in [Example 4-3](#). `tcm.objdump` shows that the new `.isrs` section is located in the tightly coupled instruction memory.
- Examine the linker script file, `linker.x`, shown in [Example 4-4](#). `linker.x` places the new region `isrs_region` in tightly-coupled instruction memory, adjacent to the `tightly_coupled_instruction_memory` region.

You can run the example by carrying out the following steps:

- Open another shell and run `nios2-terminal`.
- If your hardware is not already configured with the correct `.sof` file, type the following command:

```
nios2-configure-sof ../../../../*.sof
```

- In your original shell, type the following command:

```
nios2-download -g tcm.elf
```

Example 4-3. Excerpts from tcm.objdump

```

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
.
.
.

```

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 4-8 on page 4-20](#) 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 re-sized.

Other BSP Tasks

This section covers some other common situations in which the software build tools are useful.

Creating a BSP for a Nios 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 “Altera-Provided Development Tools” in the [Overview](#) 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 Nios development board. For example, if you have a 2C35 development board, you might select *<Nios II EDS install path>/examples/verilog/niosII_cycloneII_2c35/standard*.

This example uses the Verilog HDL standard hardware example design. You can select the language you prefer (Verilog HDL or VHDL), and any type of example design except `small`.

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

```
cp -R $SOPC_KIT_NIOS2/examples/verilog\  
/niosII_cycloneII_2c35/standard .
```

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

```
chmod -R +w .
```

7. The *<projects>* directory contains a subdirectory named `software_examples/bsp`. The `bsp` directory contains several BSP example directories, such as `hal_default`. For a description of the example BSPs, refer to [Table 4-2 on page 4-4](#). Select the directory containing an appropriate BSP, and make it the current working directory.
8. 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` command. 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 software build tools create an `alt_sys_init.c` file. By default, the build tools assume 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 BSP generator 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 software build tools allow 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 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 software build tools recompile 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 software build tools create 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 setting flag 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 4-8](#).

You individually specify the optimization and debug level for the application and BSP projects, and any 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 4-17](#).

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.

Using the Nios II C2H Compiler

The Nios II software build tools support the Nios II C2H Compiler with the `nios2-c2h-generate-makefile` command. The C2H Compiler implements hardware acceleration in the Nios II processor. Perform the following steps to create and build a software project with a C2H accelerator:

1. Create a working directory for your hardware and software projects. The following steps refer to this directory as *<projects>*.
2. Locate a Nios II hardware example corresponding to your Nios development board, and copy the hardware example to your *<projects>* working directory.
3. Select an application in a subdirectory of **software_examples/app** in the *<projects>* directory. The following steps refer to the application directory as *<application>*.
4. Select a BSP appropriate to your application. The following steps refer to the BSP directory as *<BSP>*. Create and build the BSP with the `create-this-bsp` script.
5. Create the application project by typing the following command:

```
nios2-app-generate-makefile --c2h --bsp-dir <BSP> --src-dir <application>
```

The `--c2h` command-line option causes **Makefile** to include the C2H makefile fragment, **c2h.mk**.

6. Create the C2H makefile fragment by typing the following command:

```
nios2-c2h-generate-makefile \
  --sopc=../c2h_tutorial_hw/NiosII_<board name>_standard_sopc.sopcinfo\
  --accelerator=do_dma,dma_c2h_tutorial.c --enable_quartus=1
```

When `nios2-c2h-generate-makefile` completes, you can find the makefile fragment, **c2h.mk**, in the *<application>* directory.

7. Build the application project by typing `make`. To build the project, the makefiles perform the following tasks:
 - a. Start the C2H Compiler to analyze the accelerated function, generate the hardware accelerator, and generate the C wrapper function.
 - b. Run SOPC Builder to incorporate the accelerator in the SOPC Builder system. The build process modifies the `.sopcinfo` file to include the new accelerator as a component in the system.
 - c. Run the Quartus II software to compile the hardware project and regenerate the `.sof` file.
 - d. Rebuild the C/C++ application project and link the accelerator wrapper function to the application.



Close SOPC Builder while building Nios II C/C++ projects with accelerated functions. The C2H Compiler modifies the SOPC Builder system in the background. If SOPC Builder is open while you build a Nios II IDE project with C2H accelerators, the system displayed in the SOPC Builder window can become out-of-date. If you inadvertently leave SOPC Builder open while building an accelerator with the C2H Compiler, be sure to close the `.sopcinfo` file without saving it. If you save the out-of-date file, you overwrite your accelerator-enhanced system file.



For more details about `nios2-c2h-generate-makefile`, refer to the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*. For more details about the C2H acceleration example given here, refer to the *Getting Started Tutorial* chapter of the *Nios II C2H Compiler User Guide*.

Details of BSP Creation

Figure 4-6 on page 4-33 shows more details about how `nios2-bsp` creates a BSP. The `nios2-bsp` command determines whether a BSP already exists, and uses the `nios2-bsp-create-settings` command to create a new BSP settings file or the `nios2-bsp-update-settings` command to update an existing BSP settings file. For detailed information about BSP settings files, refer to “[BSP Settings File Creation](#)” on page 4-33. The `nios2-bsp` script assumes that the BSP settings file is named `settings.bsp` and resides in the BSP directory, which you specify on the `nios2-bsp` command line.

`nios2-bsp` uses the `nios2-bsp-generate-files` command to create the BSP files. The `nios2-bsp-generate-files` command 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.

`nios2-bsp` manages copied files differently from generated files. If copied files, such as source files, already exist, it does not overwrite them. Subsequent executions of `nios2-bsp-generate-files` do not overwrite these files. Preserving existing copied files allows you to directly modify C source files in any BSP, for example to customize a device driver.

By contrast, `nios2-bsp` always overwrites generated files, such as the BSP `Makefile`, `system.h`, and `linker.x`. A comment at the top of each generated file warns you not to edit it.



Nothing prevents you from modifying a generated file. However, after you do so, you can no longer update your BSP to match changes in your SOPC Builder system. If you update your BSP (by running `nios2-bsp` or `nios2-bsp-update-settings`), your changes to the generated file are destroyed.

Tcl Scripts for Board Support Package Settings

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.

The `nios2-bsp-create-settings`, `nios2-bsp-query-settings`, and `nios2-bsp-update-settings` utilities, and the `nios2-bsp` script, all support Tcl scripts, with the `--script` command-line argument.

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 `nios2-bsp` and then use it later in script execution.



For descriptions of the Tcl commands used to manipulate BSPs, refer to “Tcl Commands for BSP Settings” in the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer’s Handbook*.

Calling a Custom BSP Tcl Script

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
```

For an example of custom Tcl script usage, refer to “[Creating Memory Initialization Files](#)” on page 4-18.

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 4-36.



If you update your BSP, for example by rerunning `nios2-bsp`, you must include the script, or 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 your version control system. For further information, refer to “[Using Version Control](#)” on page 4-16.

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 SOPC Builder system.

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

```

# Select a device connected to the CPU as the default STDIO device.

# 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 CPU.
    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'll 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

```

 The Nios II software build tools use slave descriptors to refer to components connected to the Nios II processor. A slave descriptor is the unique name of an SOPC Builder component's slave port.

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 4-36](#).


`nios2-bsp` uses a Tcl script (named `bsp-set-defaults.tcl`) to specify default values for system-dependent settings. System-dependent settings are BSP settings that reference system information in the `.sopcinfo` file. For details about the default Tcl script, refer to [“Specifying BSP Defaults” on page 4-36](#).

The path to the default Tcl script is passed to `nios2-bsp-create-settings` or `nios2-bsp-update-settings` before any user input. As a result, user input overrides settings made by the default Tcl script. You can also 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 Script for BSP Defaults” on page 4-36](#).

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

- `stdio` character device
- System timer device
- Default linker memory regions
- Default linker sections mapping
- Default boot loader settings

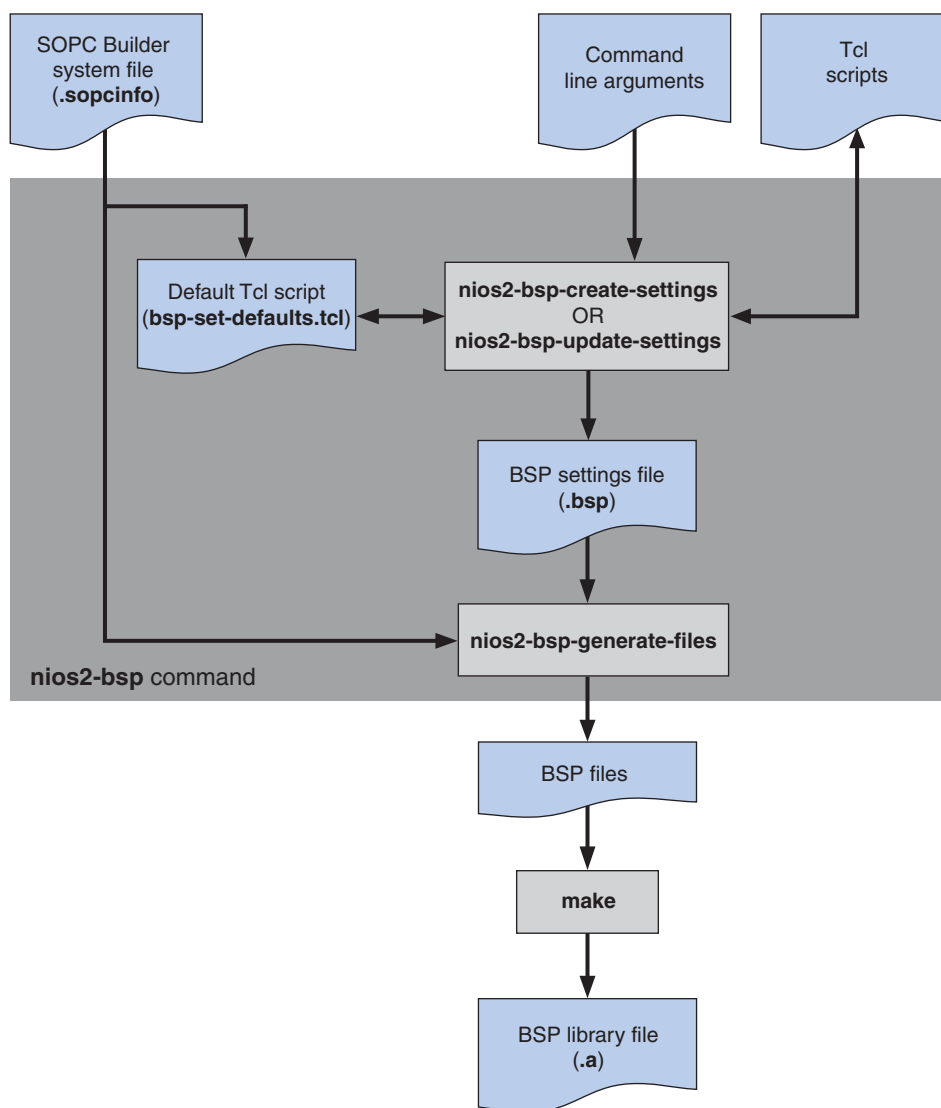
The default Tcl scripts use slave descriptors to assign devices.

 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*.

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`).

[Figure 4-6](#) 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` command has access to the `.sopcinfo` file.

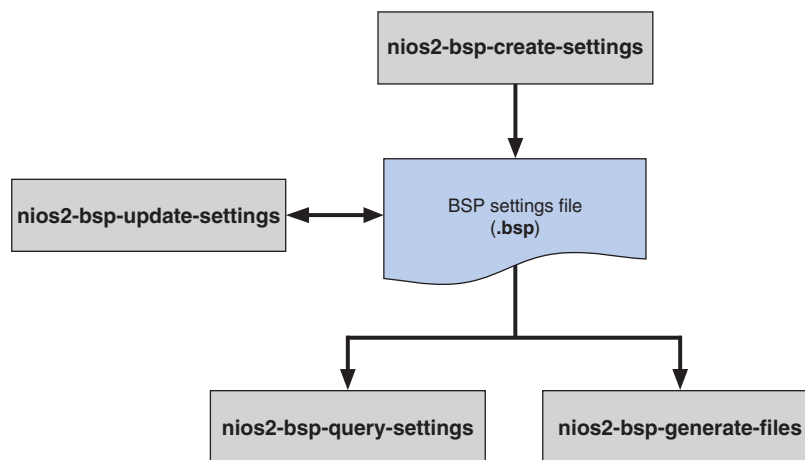
Figure 4-6. nios2-bsp Command Expanded Flow



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 BSP generator writes the value of all settings to the settings file.

Figure 4-7 shows how the BSP generator interacts with the BSP settings file. The `nios2-bsp-create-settings` command creates a new BSP settings file. The `nios2-bsp-update-settings` command updates an existing BSP settings file. The `nios2-bsp-query-settings` command reports the setting values in an existing BSP settings file. The `nios2-bsp-generate-files` command generates a BSP from the BSP settings file.

Figure 4-7. BSP Settings File and BSP Commands

Modifying the BSP

You might need to update an existing BSP if your requirements change (for example, you change the compiler optimization level), or because of changes to the SOPC Builder system to which it refers. There are three approaches to updating a BSP. In order of preference, these approaches are:

- Recreate the BSP using a Tcl script
- Run `nios2-bsp`
- Run `nios2-bsp-update-settings` and `nios2-bsp-generate-files`

The following sections discuss each approach.

Recreate the BSP Using a Tcl Script

This approach gives you the maximum control. When you initially create the BSP, create a Tcl script specifying all BSP settings. Use the same Tcl script to recreate the BSP.

The Tcl script specifies all the contents of the settings file. Because you are recreating the settings file as well as all generated files, you can guarantee that system-dependent settings are adjusted correctly based on any changes in the SOPC Builder system.

Run `nios2-bsp`

This approach keeps your settings up-to-date with your SOPC Builder system in most circumstances.

`nios2-bsp` runs `nios2-bsp-update-settings` to update the settings file as needed and then runs `nios2-bsp-generate-files` to update your BSP files. You can then run `make` to build a new BSP library file.

When you run `nios2-bsp` on an existing BSP, you generally do not need to supply the command-line arguments and Tcl scripts. Most of the original BSP settings persist in the BSP settings file.

The exception is default settings specified by the default Tcl script. 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 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.

Run `nios2-bsp-update-settings` and `nios2-bsp-generate-files`

You can use this approach if you are certain that your settings file needs updating.

Coordinating with SOPC Builder System Changes

Every BSP is based on a Nios II processor in an SOPC Builder system. If the SOPC Builder system changes after you generate your BSP, you usually must update the BSP.

If all BSP system-dependent settings are still consistent with the new `.sopcinfo` file, you can just run `nios2-bsp-generate-files` with the existing BSP settings file to create an updated BSP. The `nios2-bsp-generate-files` command reads the `.sopcinfo` file for basic system parameters such as module base addresses and clock frequencies.

The following list shows examples of system changes that do not affect BSP system-dependent settings. Although these changes do not require you to regenerate your settings file, you must run the `nios2-bsp-generate-files` command to regenerate files such as the `Makefile`, `system.h`, and the linker script. The following are examples of system changes that do not affect BSP system-dependent settings:

- Changing base addresses
- Changing interrupt numbers
- Changing clock frequencies
- Changing most processor options (for example, cache size or core type)
- Changing most component options (except for the size of memories)
- Adding bridges
- Adding new components
- Removing or renaming non-memory components other than the `stdio` device or system timer device
- Adding or removing interrupts

The following are examples of system changes that *do* affect BSP system-dependent settings:

- Renaming the processor
- Renaming or removing memories, the `stdio` device, or the system timer device
- Changing memory sizes
- Changing the processor reset and exception slave ports or offsets

If changes to your `.sopcinfo` file make it inconsistent with your BSP, you must update or recreate the BSP. Use one of the methods described in “Modifying the BSP” on page 4-34.

Specifying BSP Defaults

Table 4-9 lists the components of the BSP default Tcl scripts included in the BSP generator. 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
<code>bsp-set-defaults.tcl</code>	Top-level	Sets system-dependent settings to default values.
<code>bsp-call-proc.tcl</code>	Top-level	Calls a specified procedure in one of the helper scripts.
<code>bsp-stdio-utils.tcl</code>	Helper	Specifies <code>stdio</code> device settings.
<code>bsp-timer-utils.tcl</code>	Helper	Specifies system timer device setting.
<code>bsp-linker-utils.tcl</code>	Helper	Specifies memory regions and section mappings for linker script.
<code>bsp-bootloader-utils.tcl</code>	Helper	Specifies boot loader-related settings.

Top Level 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 SOPC Builder system. The `nios2-bsp-create-settings` and `nios2-bsp-update-settings` commands 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. The `nios2-bsp` command calls the default Tcl script by calling 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
<code>default_stdio</code>	Slave descriptor of default <code>stdio</code> device (<code>stdin</code> , <code>stdout</code> , <code>stderr</code>). Set to <code>none</code> if no <code>stdio</code> device desired.
<code>default_sys_timer</code>	Slave descriptor of default system timer device. Set to <code>none</code> if no system timer device desired.
<code>default_memory_regions</code>	Controls generation of memory regions. By default, bsp-linker-utils.tcl removes and regenerates all current memory regions. Use the <code>DONT_CHANGE</code> keyword to suppress this behavior.
<code>default_sections_mapping</code>	Slave descriptor of the memory device to which the default sections are mapped. This argument has no effect if <code>default_memory_regions == DONT_CHANGE</code> .
<code>enable_bootloader</code>	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`, `.rwddata`, `.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
<code>hal.linker.allow_code_at_reset</code>	0	1
<code>hal.linker.enable_alt_load_copy_rodata</code>	0	1 if <code>.rodata</code> memory different than <code>.text</code> memory and <code>.rodata</code> memory is volatile; 0 otherwise
<code>hal.linker.enable_alt_load_copy_rwdata</code>	0	1 if <code>.rwdata</code> memory different than <code>.text</code> memory; 0 otherwise
<code>hal.linker.enable_alt_load_copy_exceptions</code>	0	1 if <code>.exceptions</code> memory different than <code>.text</code> memory and <code>.exceptions</code> 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*.

Calling Procedures in the Default Tcl Script

The procedure call Tcl script consists of the top-level `bsp-call-proc.tcl` script plus the helper scripts listed in Table 4-9 on page 4-36. 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 software build tools 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 software build tools, refer to the *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

Boot Configurations

The HAL and MicroC/OS-II BSPs support several boot configurations. The default Tcl script configures an appropriate boot configuration based on your SOPC Builder 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 CPU 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 (`.srec`) files. 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` command, 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 CPU 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 CPU) 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 (`.rdata`, `.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 `.rdata` section (either to another RAM or to a reserved area in the same RAM as the `.text` RAM) if `.rdata` 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.

Run-time Configurable Reset Configuration

The reset address points to a memory that contains code that executes before the normal reset code. When the CPU 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 CPU 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 CPU 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.

Restrictions

The Nios II software build tools have the following restrictions:

- The Nios II software build tools are only supported by SOPC Builder release 7.1 or later. The Nios II software build tools require an SOPC Builder System File (**.sopcinfo**) for the system description.
 - If you have a legacy hardware design based on an SOPC Builder system (**.ptf**) file, SOPC Builder can convert your **.ptf** to a **.sopcinfo** file.
 - If your hardware design was generated with SOPB Builder release 7.1 or 7.2, regenerate it with SOPC Builder release 8.0 or later before creating a BSP.
- The Nios II software build tools support BSPs incorporating the Altera HAL and Micrium MicroC/OS-II only.

Referenced Documents

This chapter references the following documents:

- *Overview* chapter of the *Nios II Software Developer's Handbook*
- *Introduction to the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*
- *Overview of the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*
- *Developing Programs Using the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*
- *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*
- *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*
- *Nios II C2H Compiler User Guide*

Document Revision History

Table 4-13 shows the revision history for this document.

Table 4-13. Document Revision History

Date & Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	<ul style="list-style-type: none"> ■ Moved information about Tcl-based device drivers and software packages, formerly in this chapter, to <i>Developing device Drivers for the Hardware Abstraction Layer</i>. ■ Described how to work with compiler optimization and debugger settings. ■ Described newlib recompilation. ■ Corrected minor typographical errors. 	<ul style="list-style-type: none"> ■ Compiler optimization and debugger settings ■ newlib recompilation
May 2008 v8.0.0	<ul style="list-style-type: none"> ■ Add instructions for writing instruction-related exception handler ■ Example designs removed from list 	<ul style="list-style-type: none"> ■ Advanced exceptions added to Nios II core ■ Instruction-related exception handling added to HAL ■ Example designs removed from EDS
October 2007 v7.2.0	Initial release. Material moved here from former <i>Nios II Software Build Tools</i> chapter.	—

