Explore more resources
Altera® Design Hub

altera
An Intel Company

Nios® V Processor Software Developer Handbook

Updated for Quartus® Prime Design Suite: **25.1**

Online Version **743810**
Send Feedback **2025.04.07**

Contents

1. Overview of Nios® V Embedded Processor Development....8
1.1. Nios® V Processor Software Development Environment....8
1.1.1. Quartus Prime Software Support....9
1.2. Nios V Processor Software Project Types....10
1.3. Nios V Processor Development Tools....10
1.3.1. Nios V Processor Command Line Utilities Tools....10
1.3.2. Board Support Package Editor....10
1.3.3. Ashling RiscFree IDE for Altera FPGAs....11
1.4. Nios V Processor Software Programs....11
1.4.1. CMakeLists.txt and Nios V Processor Tools....11

2. Getting Started with the Graphical User Interface....12
2.1. Using Ashling RiscFree IDE for Altera® FPGAs....12
2.2. Using BSP Editor....12
2.3. Using Quartus Programmer....12

3. Getting Started from the Command Line....13
3.1. Advantages of Command Line Software Development....13
3.2. Outline of the Nios V Processor Tools Command Line....13
3.2.1. Command Line Utilities....13

4. Nios V Processor Software Development and Implementation....18
4.1. Nios V Processor Tools....18
4.1.1. Nios V Processor Tools Overview....18
4.1.2. Nios V Processor Software Build Tools....19
4.2. Nios V Processor Software File Tree....25
4.2.1. Building Single Application and Single BSP....25
4.2.2. Building Single Application, Single Library, and BSP....25
4.2.3. Building Multiple Applications and Single BSP....26
4.2.4. Building BSP Library....27
4.3. Nios V Processor Software Development Flow....28
4.3.1. Getting Started....29
4.3.2. Configuring BSP Projects....30
4.3.3. Configuring the Application Project....37
4.3.4. Building and Running Nios V Processor Software....38
4.4. Developing with the Hardware Abstraction Layer....41
4.4.1. Hardware Abstraction Layer Services....42
4.4.2. System Startup in HAL-based Application....42
4.4.3. HAL Peripheral Services....45
4.4.4. Handling Exceptions....52

5. Nios V Processor Board Support Package Editor....53
5.1. Board Support Packages....53
5.1.1. Overview of BSP Creation....53
5.1.2. Nios V Processor BSP Components....54
5.2. Common BSP Tasks....55
5.2.1. Creating a BSP for an Intel FPGA Development Board....55
5.2.2. Adding the BSP Editor to Tool Flow....55

5.2.3. Linking and Locating....57
5.2.4. Other BSP Tasks....61
5.3. Details of BSP Creation....64
5.3.1. BSP Settings File Creation....64
5.3.2. BSP Files and Folders....65
5.3.3. Linker Map Validation....67
5.3.4. Hierarchical Platform Designer Systems....67
5.4. Tcl Scripts for BSP Settings....68
5.4.1. Calling a Custom BSP Tcl Script....68
5.5. Revising Your BSP....70
5.5.1. Regenerating Your BSP....70
5.5.2. Updating Your BSP....71
5.5.3. Recreating Your BSP....72
5.6. Specifying BSP Defaults....72
5.6.1. Top Level Tcl Script for BSP Defaults....73
5.7. Device Drivers and Software Packages....75
5.8. Boot Configurations for Altera FPGA Embedded Software....75
5.8.1. Memory Types....76
5.8.2. Boot from Flash Configuration....76
5.8.3. Run from Initialized Memory Configuration....77
5.8.4. Run-time Configurable Reset Configuration....77

6. Overview of the Hardware Abstraction Layer....79

6.1. Getting Started with the Hardware Abstraction Layer....79
6.2. HAL Architecture for Embedded Software Systems....80
6.2.1. Services....80
6.2.2. Applications versus Drivers....81
6.2.3. Generic Device Models....81
6.2.4. C Standard Library—Newlib, Picolibc....82
6.3. Embedded Hardware Supported by the HAL....82
6.3.1. Nios V Processor Core Support....82
6.3.2. Supported Peripherals....82

7. Developing Programs Using the Hardware Abstraction Layer....84

7.1. HAL BSP Settings....84
7.2. The Nios V Processor Embedded Project Structure....85
7.3. The system.h System Description File....85
7.4. Data Widths and the HAL Type Definitions....86
7.5. UNIX-Style Interface....87
7.6. File System....88
7.7. Using Character-Mode Devices....89
7.7.1. Standard Input, Standard Output and Standard Error....89
7.7.2. General Access to Character Mode Devices....89
7.7.3. C++ Streams....90
7.7.4. /dev/null....90
7.7.5. Lightweight Character-Mode I/O....90
7.7.6. Intel FPGA Logging Functions....90
7.8. Using File System....95
7.9. Using Timer Devices....96
7.9.1. System Clock Driver....96
7.9.2. Alarms....97

7.9.3. Timestamp Driver..... 98
7.10. Using Flash Devices.....99
7.10.1. Simple Flash Access..... 99
7.10.2. Block Erasure or Corruption..... 100
7.10.3. Fine-Grained Flash Access..... 101
7.11. Using DMA Devices.....104
7.11.1. DMA Transmit Channels.....105
7.11.2. DMA Receive Channels..... 106
7.12. Reducing Code Footprint in Embedded Systems.....108
7.12.1. Apply Compiler Flags..... 109
7.12.2. Use Small Variant Device Drivers.....110
7.12.3. Reduce the File Descriptor Pool.....110
7.12.4. Use /dev/null.....111
7.12.5. Use a Smaller File I/O Library.....111
7.12.6. Use the Minimal Character-Mode API.....114
7.12.7. Eliminate Unused Device Drivers..... 114
7.12.8. Use the Picolibc Library..... 115
7.12.9. Eliminate Unused `alt_load()` 115
7.12.10. Eliminate Unneeded Exit Code.....116
7.13. Interrupt Controllers.....117
7.14. Boot Sequence and Entry Point.....117
7.14.1. Hosted Versus Free-Standing Applications.....117
7.14.2. Boot Sequence for HAL-Based Programs..... 117
7.14.3. Customizing the Boot Sequence..... 119
7.15. Memory Usage.....119
7.15.1. Memory Sections.....119
7.15.2. Assigning Code and Data to Memory Partitions..... 120
7.15.3. Placement of the Heap and Stack..... 123
7.15.4. Global Pointer Register.....123
7.15.5. Boot Modes..... 125
7.16. Working with HAL Source Files..... 125
7.16.1. Finding HAL Files.....125
7.16.2. Overriding HAL Functions..... 126

8. Developing Device Drivers for the Hardware Abstraction Layer..... 127
8.1. Driver Integration in the HAL API..... 127
8.2. The HAL Peripheral-Specific API..... 128
8.3. Preparing for HAL Driver Development..... 128
8.4. Development Flow for Creating Device Drivers..... 128
8.5. Nios V Processor Hardware Design Concepts.....129
8.5.1. The Relationship Between the `.qsys` File and `system.h`.....129
8.5.2. Using the System Generation Tool to Optimize Hardware..... 129
8.5.3. Components, Devices, and Peripherals..... 129
8.6. Accessing Hardware..... 129
8.7. Creating Embedded Drivers for HAL Device Classes..... 131
8.7.1. Character-Mode Device Drivers..... 131
8.7.2. File Subsystem Drivers..... 133
8.7.3. Timer Device Drivers..... 134
8.7.4. Flash Device Drivers.....135
8.7.5. DMA Device Drivers.....136
8.8. Integrating a Device Driver in the HAL..... 137

8.8.1. Overview..... 138
8.8.2. Assumptions and Requirements.....138
8.8.3. Nios V Processor BSP Generation Flow..... 139
8.8.4. File Names and Locations..... 140
8.8.5. Driver and Software Package Tcl Script Creation..... 140
8.9. Creating a Custom Device Driver for the HAL..... 149
8.9.1. Header Files and alt_sys_init.c..... 149
8.9.2. Device Driver Source Code..... 150
8.10. Reducing Code Footprint in HAL Embedded Drivers..... 152
8.11. HAL Namespace Allocation.....152
8.12. Overriding the HAL Default Device Drivers..... 153

9. Trap Handling..... 154
9.1. Nios V Processor Trap Handling Overview..... 154
9.1.1. Trap Handling Terminology..... 154
9.2. Hardware Interrupt Controllers..... 155
9.2.1. CLINT-Direct Mode..... 156
9.2.2. CLINT-Vectored Mode..... 161
9.2.3. CLIC Mode..... 167
9.3. Interrupt Service Routines and Exception Handler.....170
9.3.1. Platform Interrupt Service Routines..... 170
9.3.2. Software Interrupt Service Routines.....175
9.3.3. Timer Interrupt Service Routine..... 178
9.3.4. Instruction-Related Exception Handler.....178
9.4. Latency and Response Time.....184
9.4.1. Direct or Vectored Mode.....184
9.5. Improving Nios V Processor ISR Performance.....184
9.5.1. Software Performance Improvements..... 184
9.5.2. Hardware Performance Improvements..... 187
9.6. Debugging Nios V Processor ISRs..... 188

10. Cache and Tightly-Coupled Memory..... 189
10.1. Nios V Processor Cache Implementation..... 189
10.2. Initializing Cache after Reset.....190
10.3. Device Driver Cache Considerations..... 190
10.4. Managing Cache in Multi-Master and Multi-Processor Systems..... 190
10.5. Tightly-Coupled Memory.....190

11. MicroC/OS-II Real-Time Operating System.....191
11.1. Overview of the MicroC/OS-II RTOS.....191
11.1.1. Support and Licensing.....192
11.2. Other RTOS Providers..... 192
11.3. The Nios V Processor Implementation of MicroC/OS-II.....192
11.3.1. MicroC/OS-II Architecture.....192
11.3.2. MicroC/OS-II Device Drivers..... 193
11.3.3. Thread-Safe Drivers.....193
11.3.4. The newlib ANSI C Standard Library.....195
11.3.5. Interrupt Service Routines for MicroC/OS-II..... 195
11.4. Implementing MicroC/OS-II Projects for the Nios V Processor.....196

12. MicroC/TCP-IP Protocol Stack.....197
12.1. Overview of the MicroC/TCP-IP Protocol Stack..... 197

12.2. Support and Licensing....197
12.3. Prerequisites for Understanding the MicroC/TCP-IP Protocol Stack....197
12.4. Introduction to the MicroC/TCP-IP Protocol Stack - Nios V Processor Edition....198
12.4.1. Nios V Processor System Requirements....198
12.5. The MicroC/TCP-IP Protocol Stack Files and Directories....198
12.6. Enabling MicroC/TCP-IP Protocol Stack....199
12.7. Using the MicroC/TCP-IP Protocol Stack....200
12.7.1. Initializing the Stack....201
12.7.2. Network IP configuration....202
12.7.3. Example applications....202

13. FreeRTOS Real-Time Operating System....203
13.1. Overview of the FreeRTOS....203
13.1.1. Support and Licensing....204
13.2. The Nios V Processor Implementation of FreeRTOS....204
13.2.1. FreeRTOS Architecture....204
13.2.2. FreeRTOS Device Drivers....205
13.2.3. Thread-Safe HAL Drivers....206
13.2.4. The newlib ANSI C Standard Library....207
13.3. Implementing FreeRTOS Project for the Nios V Processor....208
13.3.1. Create FreeRTOS BSP Project....208
13.3.2. Configure FreeRTOS Features....209
13.3.3. Write FreeRTOS-based Source Codes....209
13.3.4. Generate Application CMake File....211
13.3.5. Build FreeRTOS BSP and Application Projects....211
13.3.6. Integrating Device Drivers....211

14. Read-Only Zip File System....215
14.1. Read-Only Zip File System in BSP Editor....215
14.2. Preparing the Zip File....216
14.3. Programming the Zip File to Flash....217
14.3.1. General Purpose QSPI Flash....217
14.3.2. Active Serial Configuration Flash in Control-block Based Devices....217
14.3.3. Active Serial Configuration Flash in SDM Based Devices....217
14.4. Programming the Zip File to On-Chip ROM....218

15. Publishing Component Information to Embedded Software....219
15.1. Embedded Component Information Flow....219
15.2. Embedded Software Assignments....220
15.2.1. C Macro Namespace....220
15.2.2. Configuration Namespace....221

16. Nios V Processor Appendix....224
16.1. HAL API....224
16.1.1. HAL API Reference....224
16.1.2. HAL Standard Types....278
16.2. Nios V Processor Tools Reference....281
16.2.1. Nios V Processor Command Line Utilities....281
16.2.2. File Format Conversion Tools Reference....287
16.2.3. Other Command Line Utilities Tools Reference....290
16.3. GNU RISC-V Embedded GCC Toolchain Reference....295
16.4. Settings Managed by Nios V Processor Board Support Package Editor....296

16.4.1. Overview of BSP Settings..... 297
16.4.2. Overview of Component and Driver Settings..... 298
16.4.3. BSP Settings Reference..... 298
16.5. Board Support Package Tcl Commands..... 333
16.5.1. Tcl Command Environments..... 333
16.5.2. Tcl Commands for BSP Settings..... 333
16.5.3. Tcl Commands for BSP Generation Callbacks..... 359
16.5.4. Tcl Commands for Drivers and Packages..... 367
16.6. Nios V Processor Design Example..... 378
16.6.1. Intel FPGA Design Store..... 378
16.6.2. Quartus Prime Software..... 378
16.6.3. Migrating to Other Intel FPGA Device 380
16.7. Manual System PATH Variable Setup..... 380

17. Nios V Processor Software Developer Handbook Archives..... 382

18. Revision History for the Nios V Processor Software Developer Handbook..... 383

1. Overview of Nios® V Embedded Processor Development

1.1. Nios® V Processor Software Development Environment

Nios® V processor tools refer to the development tools ecosystem enabled by Altera which comprises hardware and software development, debugging, and simulation. The Nios V processor tools allow you to perform the following tasks:

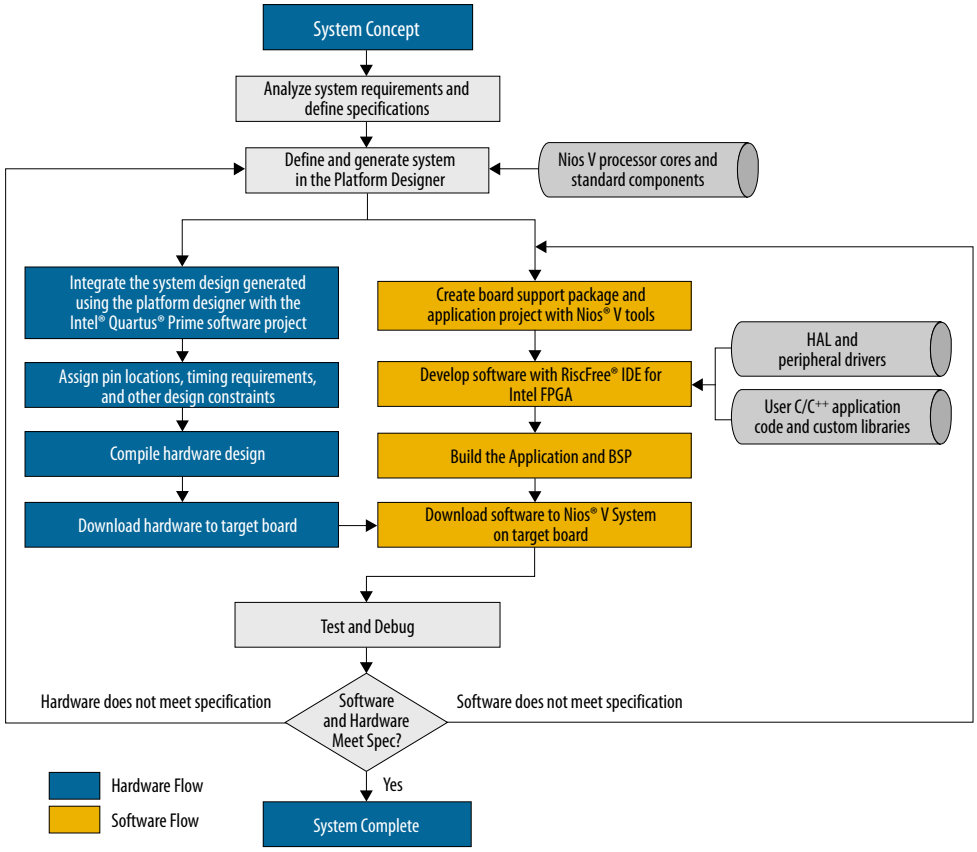
- Create software projects, with detailed control over the software build process.
- Create and manage single-threaded programs and complex applications using RTOS and middleware libraries available from Intel® FPGA IP and third-party vendors.

Install the latest release of the Quartus® Prime software on the Windows*or Linux* operating system. Following that, you can find the Nios V processor tools package in the directory: `<Intel Quartus Prime installation directory>\niosv`. You need to acquire the free license for the Nios V processor to compile the design in Quartus Prime software.

© Altera Corporation. Altera, the Altera logo, the 'a' logo, and other Altera marks are trademarks of Altera Corporation. Altera and Intel warrant performance of its FPGA and semiconductor products to current specifications in accordance with Altera's or Intel's standard warranty as applicable, but reserves the right to make changes to any products and services at any time without notice. Altera and Intel assume no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to inwriting by Altera or Intel. Altera and Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

Figure 1. Hardware and Software Development with Nios V Processor Tools



Related Information

- Nios® V Embedded Processor Design Handbook
  Describes how to most effectively use the tools, recommends design styles, and practices for developing, debugging, and optimizing embedded systems using the Nios® V processor and Intel-provided tools.
- Nios® V Processor Reference Manual
  Provides information about the Nios® V processor performance benchmarks, processor architecture, the programming model, and the core implementation.
- Ashling* RiscFree* Integrated Development Environment (IDE) for Altera® FPGAs User Guide
  Describes the RiscFree integrated development environment (IDE) for Altera® FPGAs Arm-based HPS and Nios® V core processor.

1.1.1. Quartus Prime Software Support

Nios V processor build flow is different for Quartus Prime Pro Edition software and Quartus Prime Standard Edition software. Refer to *AN 980: Nios V Processor Quartus Prime Software Support* for more information about the differences.

Related Information

AN 980: Nios® V Processor Quartus® Prime Software Support

1.2. Nios V Processor Software Project Types

Table 1. Nios V Processor Software Project Types

Software Project Types	Description
Application	• A Nios V processor C/C++ application project consists of a collection of source code, including a `CMakeLists.txt`. • The `CMakeLists.txt` compiles the source code and links the code with a BSP, and one or more optional libraries to create one `.elf` file. • An application project includes code that calls functions in libraries and BSPs. • A typical characteristic of an application project is that one of the source files contains function `main()`.
User Library	• A user library project is a collection of source code compiled to create a single library archive file (`.a`). • Libraries often contain reusable, general purpose functions that multiple application projects can share. One example is a collection of common arithmetical functions. • A user library does not contain a `main()` function.
Board Support Package (BSP)	A BSP project contains the following elements: • Hardware abstraction layer • Device drivers • Optional software packages • Optional real-time operating system *Note:* The toolchain provides the newlib C Library.

1.3. Nios V Processor Development Tools

Altera uses the following applications to build Nios V processor software program:

- Nios V processor command line tools
- BSP Editor in Platform Designer (part of Quartus Prime software)
- Ashling* RiscFree* IDE for Altera FPGAs

1.3.1. Nios V Processor Command Line Utilities Tools

Nios V processor command line tools is a set of command line programs for building Nios V processor programs. You can use them by typing into Nios V Command Shell, or to group them in a script that can be run in the said shell.

1.3.2. Board Support Package Editor

The BSP Editor is for creating, modifying and generating the BSP for Nios V processor-based projects. The BSP Editor is for BSP package generation only. You need the generated BSP package to build Nios V processor software. BSP Editor is the alternative to using the Nios V processor command line utilities tool (niosv-bsp).

Related Information

- Configuring BSP Projects on page 30
  For more information about configuring the BSP project.
- Configuring the Application Project on page 37
  For more information about configuring the application project

1.3.3. Ashling RiscFree IDE for Altera FPGAs

The Ashling RiscFree IDE for Altera FPGAs provides an integrated development environment (IDE) to build, load, and debug applications running on Nios V processor. Follow these steps to use the tool:

1. Configure your Nios V processor BSP package using the BSP Editor.
2. Create a Nios V processor BSP and application project.
3. Import the Nios V processor BSP and application project into Ashling RiscFree IDE for Altera FPGAs.

Note: Before importing the project to Ashling RiscFree IDE for Altera FPGAs, you need the generated BSP package and application package.

Related Information

Ashling* RiscFree* Integrated Development Environment (IDE) for Altera FPGAs User Guide

1.4. Nios V Processor Software Programs

Nios V processor programs are Executable and Linking Format File (ELF). The standard process for creating the ELF file is to create CMake projects that combine user code, BSP and other libraries to build this ELF file. The CMake combines user application project with information from BSP project, and other libraries as necessary to build the Nios V processor program.

The Nios V processor tools create software projects for you. Each project is based on a `CMakeLists.txt`.

1.4.1. CMakeLists.txt and Nios V Processor Tools

The following describes CMakeLists.txt:

- Act as the central component of a Nios V processor software project, whether the project is created with the RiscFree IDE for Altera FPGAs, or with the Nios V Command Shell.
- Describes all the components of a software project and how they are compiled and linked.

With a `CMakeLists.txt` and a complete set of C/C++ source files, your Nios V processor software project is fully defined. As a key part of creating a software project, the Nios V processor tools creates a `CMakeLists.txt` for you. Nios V processor software projects are sometimes called user-managed, because you are responsible for the content of the project in the `CMakeLists.txt`. Use the Nios V processor tools to control what goes in the `CMakeLists.txt`.

Related Information

- CMake
- CMake commands
  For more various actions the tool can perform.

2. Getting Started with the Graphical User Interface

2.1. Using Ashling RiscFree IDE for Altera® FPGAs

RiscFree is Ashling's Eclipse CDT™ C/C++ (Development Toolkit) based integrated development environment (IDE) for Altera® FPGAs Arm*-based HPS and RISC-V-based Nios V processors.

Related Information

Ashling* RiscFree* Integrated Development Environment (IDE) for Altera® FPGAs User Guide

2.2. Using BSP Editor

Platform Designer includes the BSP Editor, a board support package (BSP) editing tool. A BSP provides a software runtime environment for embedded systems, such as Nios V processor systems.

Table 2. Using the Board Support Package Editor in the Quartus Prime Software

Task	Quartus Prime Standard Edition Software	Quartus Prime Pro Edition Software
Invoking the BSP Editor	Run the `niosv-bsp-editor` command.	Use the Platform Designer.
Creating the BSP project	Provide the SOPCINFO file of the processor system into the BSP Editor.	Provide the QSYS file of the processor system into the BSP Editor.

Related Information

- AN 980: Nios® V Processor Quartus® Prime Software Support
- Quartus® Prime Pro Edition User Guide: Platform Designer

2.3. Using Quartus Programmer

The Quartus Prime Programmer allows you to program and configure Altera CPLD, FPGA, and configuration devices. Following full design compilation, generate the primary device programming files in the Assembler, and then use the Programmer to load the programming file to a device.

Related Information

Quartus® Prime Pro Edition User Guide: Programmer

© Altera Corporation. Altera, the Altera logo, the 'a' logo, and other Altera marks are trademarks of Altera Corporation. Altera and Intel warrant performance of its FPGA and semiconductor products to current specifications in accordance with Altera's or Intel's standard warranty as applicable, but reserves the right to make changes to any products and services at any time without notice. Altera and Intel assume no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to inwriting by Altera or Intel. Altera and Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

3. Getting Started from the Command Line

The Nios V processor tools allows you to construct a wide variety of complex embedded software systems using a command line interface. From this interface, you can execute the command utilities, and use scripts (or other tools) to combine the command utilities in many useful ways.

Related Information

- Getting Started with the Hardware Abstraction Layer on page 79
- Simple Placement Options on page 121

3.1. Advantages of Command Line Software Development

The Nios V processor tools command line interface is designed to work in the Nios V Command Shell. The Nios V processor tools command line offers the following advantages over the GUI method:

- You can invoke the command line tools from custom scripts or other tools that you might already use in your development flow.
- On a command line, you can run `niosv-bsp` command with specific options to control the creation of a BSP.
- You can use command line tools in a script to build several projects at once.

3.2. Outline of the Nios V Processor Tools Command Line

3.2.1. Command Line Utilities

The Nios V processor tools command line utilities enable you to create software projects. On Windows, these utilities have `.exe` extension.

Table 3. Nios V Processor Tools Utilities Description

Nios V Processor Tool Utilities	Description
`niosv-bsp`	Create, update, generate or query the BSP project.
`niosv-app`	Generate and configure an application project.
`niosv-download`	Download the application `.elf` file into the Nios V processor.

Related Information

Nios V Processor Tools Reference on page 281
For more information about the command line utilities provided by the Nios V processor software build tools.

© Altera Corporation. Altera, the Altera logo, the 'a' logo, and other Altera marks are trademarks of Altera Corporation. Altera and Intel warrant performance of its FPGA and semiconductor products to current specifications in accordance with Altera's or Intel's standard warranty as applicable, but reserves the right to make changes to any products and services at any time without notice. Altera and Intel assume no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to inwriting by Altera or Intel. Altera and Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

3.2.1.1. niosv-bsp

Usage

```
niosv-bsp <FUNCTION> [OPTIONS] settings.bsp
```

Description

Create a BSP `CMakeLists.txt` that can compile the BSP library file. A `toolchain.cmake` file is also created, and it contains toolchain options such as C compiler, C++ compiler, and link flags to build the Nios V processor software projects. `CMakeLists.txt` uses both files when generating the application project.

The default BSP Tcl scripts run unless specified otherwise during BSP creation. The scripts are available in `<Intel Quartus Prime installation directory>/niosv/scripts/bsp-defaults` folder.

Table 4. Use Cases

Use Case	Function	Example
Quartus Prime Pro Edition: Creates a new settings file and generates the BSP files in the BSP directory.	`-c, --create`	`niosv-bsp -c -s=<path to .qsys file> -t=<bsp type> [OPTIONS] settings.bsp`
Quartus Prime Standard Edition: Creates a new settings file and generates the BSP files in the BSP directory.	`-c, --create`	`niosv-bsp -c -s=<path to .sopcinfo file> -t=<bsp type> [OPTIONS] settings.bsp`
Updates the existing settings file and generates the BSP files in the BSP directory.	`-u, --update`	`niosv-bsp -u [OPTIONS] settings.bsp`
Generates the BSP files in the BSP directory according to the existing settings file.	`-g, --generate`	`niosv-bsp -g [OPTIONS] settings.bsp`
Queries the BSP settings file. Requires extra options to query individual BSP settings.	`-q, --query`	`niosv-bsp -q -E=<exported tcl script name> [OPTIONS] settings.bsp`

Table 5. Basic Options

Options	Description
`-p, --quartus-project=<path to .qpf file>`	The Quartus Prime Project File.
`-s, --qsys=<path to .qsys file>`	The Platform Designer.
`-b, --bsp-dir=<bsp directory>`	• The BSP directory. • If not specified, this option is the current directory which stores the settings file.
`-t, --type=<bsp type>`	• The type of BSP to create. For example – hal, ucosii, or freertos. • Use this option when you are using `-c, --create` function, and the BSP type names are case-insensitive.
`-E, --export-as-tcl=<exported tcl script name>`	The exported Tcl script file name based on the queried BSP project.

Related Information

- Nios V Processor Command Line Utilities on page 281
  For more information about the rest of the provided Nios V Utilities options.
- Nios V Processor Board Support Package Editor on page 53
  For more information about the BSP Package Editor.

3.2.1.2. niosv-app

Usage

```
niosv-app [OPTIONS]
```

Description

The `niosv-app` utility is used to create an application project, and you must run make manually to build the executable (`.elf`) for your application.

In addition to the application `.elf` file, running `make` generates a `.hex` file for your application by running the `elf2hex` automatically. The generated hex files are located in the same directory as the ELF file.

Table 6. Use Cases

Use case	Example
Create application project which links to a specified BSP	`niosv-app -a=<application directory> -b=<bsp directory>-s=<source files directory> [OPTIONS]`

Create a user library	`niosv-app -l=<library directory>`
	`-s=<source files directory>`
	`-p=<public includes directory> [OPTIONS]`

Table 7. Basic Options

Options	Description
`-a, --app-dir=<application directory>`	The application directory.
`-l, --lib-dir=<library directory>`	The library directory.
`-b, --bsp-dir=<bsp directory>`	The BSP directory.
`-L, --link-lib=<linked library directory>`	Links the application or library to the specified library directory. Can be specified multiple times.
`-e, --elf-name=<.elf output file name>`	The name of the `.elf` output file. If omitted, the default `.elf` name is `<application directory>.elf`.
`-s, --srcs=<source files directory>`	Comma separated list of sources. Can be files or directories. Directories have direct descendant source files included. Specified sources are not copied. They show up with relative path in the CMakeList.txt.
`-S, --srcs-recursive=<directory>`	• Same as `--srcs` option but recursively search for source files in or under *<directory>*. • Allows multiple `--srcs-recursive` options and can be freely mixed with `--srcs` options.

continued...

Options	Description
`-i, --incs=<includes directory>`	Comma separated list of include directories.
`-I, --incs-recursive=<directory>`	• Same as `--incs` option but recursively search for source include files in or under `<directory>`. • Allows multiple `--incs-recursive` options.
`-p, --public-incs=<public includes directory>`	Comma separated list of public include directories. This is for libraries to expose public includes (for dependents to #include).

Related Information

Nios V Processor Command Line Utilities on page 281
For more information about the rest of the provided Nios V Utilities options.

3.2.1.3. niosv-download

Usage

```
niosv-download [OPTIONS] [ELF_FILE.elf]
```

Description

The `niosv-download` utility is used to download the Nios V processor application (`.elf`) file to the program memory. The processor is left in a halted state by default.

Table 8. Use Cases

Use case	Example
Download `.elf` file to the only Nios V processor instance	`niosv-download ELF_file.elf`
Download `.elf` file to the only Nios V processor instance, and run from entry point	`niosv-download -g ELF_file.elf`
Download `.elf` file to the Nios V processor which is located at cable 1, device 2 and instance 2	`niosv-download ELF_file.elf -c 1 -d 2 -i 2`
Issue debug reset for Nios V processor	`niosv-download -r`
Download `app.elf` to the only Nios V processor instance, issue a debug reset, then run from the entry point.	`niosv-download -g -r app.elf`

Table 9. Basic Options

Options	Description
`-g, --go`	Run processor from entry point after downloading.
`-c <cable number>, --cable=<cable number>`	To select the cable number that the device is connected.
`-d <device number>, --device=<device number>`	To select the device.
`-i <instance number>, --instance=<instance number>`	To select the instance.
`-r, --reset`	Reset Nios V processor target
`<ELF_file.elf>`	Specify the `.elf` file to download to Nios V processor.
`-o, --use_openocd`	Use OpenOCD instead of Ashling RiscFree GDB Server.

Related Information

Nios V Processor Command Line Utilities on page 281
For more information about the rest of the provided Nios V Utilities options.

4. Nios V Processor Software Development and Implementation

4.1. Nios V Processor Tools

Understanding the Nios V processor tools 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.

Related Information

- Intel FPGA Design Store on page 378
  For more information about example designs.
- Altera FPGA Technical Training
  Online training demonstrations located on the Embedded Design Curriculum page of the Altera FPGA Technical Training website.
- Nios V Processor Embedded Design Handbook
- Ashling* RiscFree* Integrated Development Environment (IDE) for Altera FPGAs User Guide

4.1.1. Nios V Processor Tools Overview

The Nios V processor software development flow uses the Nios V processor tools to provide a flexible, portable, and scriptable software build environment. Altera recommends that you use the Nios V Command Shell and Ashling RiscFree IDE for Altera FPGA when developing the Nios V processor software project. The Nios V processor tools include a command line environment and fits easily in your preferred software or system development environment.

The Nios V processor tools provide the following tools for software development:

- Nios V processor specific port of the newlib C library
- Altera Hardware Abstraction Layer (HAL)
- Nios V processor software build tools — based on a scriptable development flow that includes the following user interfaces:
  - Nios V processor software build tools Graphical User Interface (GUI) — Supports creating, modifying, building, running, and debugging Nios V processor software programs. The GUI is based on the BSP Editor and the RiscFree IDE for Altera FPGAs.
  - Nios V processor software build tools Command Line Interface (CLI) — Executes Nios V processor build tools commands and uses scripts (or other tools) to combine the command utilities.

© Altera Corporation. Altera, the Altera logo, the 'a' logo, and other Altera marks are trademarks of Altera Corporation. Altera and Intel warrant performance of its FPGA and semiconductor products to current specifications in accordance with Altera's or Intel's standard warranty as applicable, but reserves the right to make changes to any products and services at any time without notice. Altera and Intel assume no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to inwriting by Altera or Intel. Altera and Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

To use the Nios V processor tools , you need a `.qsys` file for your system. The `.qsys` file comprised of the Nios V processor hardware design using the Platform Designer development tools. Refer to *Nios V Processor Hardware System Design with Quartus Prime Software and Platform Designer* in the *Nios V Embedded Processor Design Handbook* to understand more on designing Nios V processor hardware. After the hardware system is ready, you can create the Nios V processor software according to the following step:

1. Generate a Board Support Package (BSP) for your system.
2. Generate and build your application software.
3. Iterate through one or both steps until your design is complete.

Related Information

Nios® V Embedded Processor Design Handbook: Nios® V Processor Hardware System Design with Quartus® Prime Software and Platform Designer

4.1.2. Nios V Processor Software Build Tools

The Nios V processor software build tools are a collection of command-line utilities. These tools allow you to build a BSP project and an application project to create an application image. You can invoke these tools in the RiscFree IDE using **External Tools Configurations**.

The BSP project is a parameterizable library, customized for the hardware capabilities and peripherals in your system. When you create a BSP library file from the BSP project, you create it with a specific set of parameter values.

The application project consists of your application source files and the application `CMakeLists.txt`. The source files refer to the services provided by the BSP library file.

Related Information

Performing External Tools Configuration
For more information about configuring juart-terminal in the Ashling RiscFree IDE for Altera FPGAs.

4.1.2.1. Nios V Processor Software Build Tools Graphical User Interface

You can accomplish most software development tasks using the Nios V processor software build tools including creating, editing, building, running, and debugging programs. The Nios V processor tools provides the following GUIs for software development:

- BSP Editor in Platform Designer
- Ashling RiscFree IDE for Altera FPGAs

Table 10. Summary of the Correlation Between GUI and the Command Line Features

Task	GUI	Command Line
Creating a BSP	BSP Editor	`niosv-bsp -c -s=<.qsys file> -t=<bsp type> [OPTIONS] settings.bsp` `niosv-bsp -g [OPTIONS] settings.bsp`
Updating a BSP	BSP Editor	`niosv-bsp -u [OPTIONS] settings.bsp`
Examining a BSP	BSP Editor	`niosv-bsp -q -E=<tcl script> [OPTIONS] settings.bsp`
Creating an application	-	`niosv-app -a=<application directory> -b=<bsp directory> -s=<source files directory> [OPTIONS]`

Creating a user library	-	`niosv-app -l=<library directory>`
		`-s=<source files directory>`
		`-p=<public includes directory> [OPTIONS]`

Modifying an application	• Any source code editor • RiscFree IDE for Altera FPGAs	Any command line source code editor
Modifying a user library	• Any source code editor • RiscFree IDE for Altera FPGAs	Any command line source code editor
Generate an application	RiscFree IDE for Altera FPGAs	`cmake` and `make`
Generate a user library	RiscFree IDE for Altera FPGAs	`cmake` and `make`

4.1.2.1.1. BSP Editor

You can use the BSP Editor to perform the following tasks:

- Create or modify a Nios V processor BSP project that has been integrated into Platform Designer.
- Edit settings, linker regions, and section mappings, and to select software packages and device drivers.

To launch the BSP Editor, click **File ➤ Open... ➤ BSP Editor** tab in the Platform Designer. Select the BSP setting file to open.

4.1.2.1.2. Ashling RiscFree IDE for Altera FPGAs

You can modify, build, run, and debug an application or a user library with the Ashling RiscFree IDE for Altera FPGAs.

4.1.2.1.3. Nios V Command Shell

The Nios V Command Shell is a command line environment initialized with the correct system PATH variable to run the Nios V processor command line tools.

Depending on the host system operating system, the Nios V Command Shell is based on the default shell or command prompt offered, such as BASH in Linux distributions or DOS in Windows operating systems.

If you are running `niosv-shell` command prompt on Windows, you can use the `powershell` or `-p` option as below:

```
C:\Users\<user_name> niosv-shell --powershell
```

Optionally, you can run a Linux environment on your Windows machine, without a virtual machine, by invoking any version of Windows Subsystem for Linux (WSL) within the Nios V Command Shell.

Figure 2. Invoke WSL within the Command Shell



Alternatively, you can manually configure your system environment for Nios V processor development instead Nios V Command Shell.

Related Information

- niosv-shell on page 282
- Nios V Processor Command Line Utilities on page 281
  For more information about launching the Nios V Command Shell.
- Manual System PATH Variable Setup on page 380
  For more information about manually configuring your system environment for Nios V processor development.

4.1.2.2. Nios V Processor Software Build Tools Command Line Interface

In the Nios V processor software build tools command line development flow, you can create, modify, build, and run Nios V processor programs. Type the commands at a command line or embedded in a script.

To debug your program, import your software projects to Ashling RiscFree IDE for Altera FPGAs. You can then further edit, rebuild, run, and debug your imported project.

Each tool provides its own documentation in the form of help accessible from the command line. To view the help menu, open the Nios V Command Shell, and type the following command: `<name of tool> --help`.

Note: Application file cannot be generated in Ashling RiscFree IDE for Altera FPGAs, you must generate the application file using **niosv-app** utilities.

For more information about the Nios V processor tools and the Nios V Command Shell, refer to Getting Started from the Command Line on page 13.

4.1.2.2.1. Nios V Processor Command Line Utilities

The Nios V processor command line tools described in this section are in the `<Intel Quartus Prime software installation directory>/niosv/bin` directory.

Table 11. Nios V Command Line Utilities Tools

Command Line Utilities Tool	Summary
`niosv-app`	To generate and configure an application project.
`niosv-bsp`	To create or update a BSP settings file and create the BSP files.
`niosv-download`	To download the `.elf` file to a Nios V processor.
`niosv-shell`	To open the Nios V Command Shell.
`niosv-stack-report`	To report memory available for stack and heap from an `.elf` file.

For more information about the Nios V processor command line commands, refer to the Nios V Processor Tools Reference.

4.1.2.2.2. File Format Conversion Tools

File format conversion is sometimes necessary when passing data from one utility to another. The file format conversion tools are in the `<Intel Quartus Prime software installation directory>/niosv/bin` directory.

The file conversion utilities are:

- **elf2flash**: To translate the `.elf` file to `.srec` format for flash memory programming.
- **elf2hex**: To translate the `.elf` file to `.hex` format for memory initialization.

Related Information

File Format Conversion Tools Reference on page 287
For more information about the file format conversion tools.

4.1.2.2.3. Other Utilities Tools

You might require the following command line tools when building a Nios V processor-based system. These command line tools are either provided by Altera in `<Intel Quartus Prime installation directory>/quartus/bin` or acquired from open-source tools.

Table 12. Other Command Line Tools

Command Line Tools	Provider	Summary
`juart-terminal`	Altera	To monitor stdout and stderr, and to provide input to a Nios V processor subsystem through stdin. This tool only applies to the JTAG UART IP when it is connected to the Nios V processor.
`cmake`	RiscFree IDE for Altera FPGAs	To generate a project build-system using CMake.
`make`	RiscFree IDE for Altera FPGAs	To generate the project `.elf` file using GNU Make.
`riscv32-unknown-elf-*`	RiscFree IDE for Altera FPGAs	GNU RISC-V Embedded GCC toolchain commands.
`openocd`	Altera	To execute OpenOCD.

continued...

Command Line Tools	Provider	Summary
`openocd-cfg-gen`	Altera	To generate the OpenOCD configuration file.
`elf-insert`	Altera	To inject additional sections into an ELF intended for the Nios V processor.
`elf-query`	Altera	To extract metadata inserted into an ELF intended for the Nios V processor.

Related Information

- Nios V Processor Embedded Design Handbook
  For open-source tools installation guide.
- Other Command Line Utilities Tools Reference on page 290
- GNU RISC-V Embedded GCC Toolchain Reference on page 295

4.1.2.3. What the Build Tools Create

The purpose of the build tools is to create and build Nios V processor software projects. A Nios V processor project build is based on `CMakeLists.txt` file with the associated source files.

The Nios V processor software build tools creates the following types of projects:

- Nios V processor Application: A program running on the Nios V processor implementing some desired functionality, such as control or signal processing.
- Nios V processor Board Support Package (BSP)
  - A library providing access to hardware in the Nios V processor system, such as UARTs and other I/O devices.
  - Provides a software runtime environment customized for one processor in a hardware system.
  - Includes the operating system, and other basic system software packages such as communications protocol stacks.
- User library: A library implementing a collection of reusable functions, such as graphics algorithms.

4.1.2.3.1. Applications and Libraries

The Nios V processor tools have nearly identical support for C/C++ applications and libraries. For each case, the Nios V processor tools generate a `CMakeLists.txt`. The `CMakeLists.txt` is used to build the application or user library.

The `CMakeLists.txt` builds one of two types of files:

- For an application: `.elf` file
- For a user library: `.a` library archive file

When you create a `CMakeLists.txt` for an application or user library, provide the Nios V processor tools with a list of source files. The BSP directory is mandatory for applications.

For more information, refer to section Nios V Processor Software File Tree.

The Nios V processor tools examines the case-sensitive extension of each source file to determine the programming language.

Table 13. Supported Programming Languages with the Corresponding File Extensions

Programming Language	File extensions
C	`.c`
C++	`.cpp`, `.cxx`, `.cc`
Nios V processor assembly language; sources are built directly by the Nios V processor assembler without preprocessing.	`.s`
Nios V processor assembly language; sources are preprocessed by the Nios V C preprocessor, allowing you to leverage macros and other preprocessor features.	`.S`

4.1.2.3.2. Board Support Packages

A Nios V processor BSP project is a specialized library containing system-specific support code. A BSP:

- Provides a software runtime environment customized for one processor in a hardware system.
- Isolates your application from system-specific details such as the memory map, available devices, and processor configuration .
- Includes header files (for example, `system.h`), and a linker script (`linker.x`). Use the BSP files when creating an application.

Related Information

Nios V Processor Board Support Package Editor on page 53
For more information about the BSP Package Editor.

4.1.2.3.3. CMakeLists.txt

The Nios V processor software development flow generates `CMakeLists.txt` files and passses the files to CMake to generate build files for Nios V processor software projects.

There are two types of `CMakeLists.txt`:

- BSP `CMakeLists.txt`: Generated from **niosv-bsp**, is a CMake recipe that describes how to generate build files for the BSP project. The `CMakeLists.txt` lists the files and directories targeted by the build and included `toolchain.cmake` file which consist of compiler configurations for the build.
- APP `CMakeLists.txt`: Generated from **niosv-app,** describes how to generate build files for user application or user library with user-provided source files and links with the specific BSP.

Altera recommends you use the BSP Editor in Platform Designer to manage and customize generated BSP files. As for the application or user library, you can use the generated `CMakeLists.txt` or create your own `CMakeLists.txt`.

4.2. Nios V Processor Software File Tree

4.2.1. Building Single Application and Single BSP

This option is to generate a single application (**app**) and BSP (**bsp**). The Nios V processor application, **app** inherit all toolchain and compile options settings and be able to link against the BSP as a library (this includes toolchain executables, compile flags, and definitions).

4.2.1.1. Build Commands

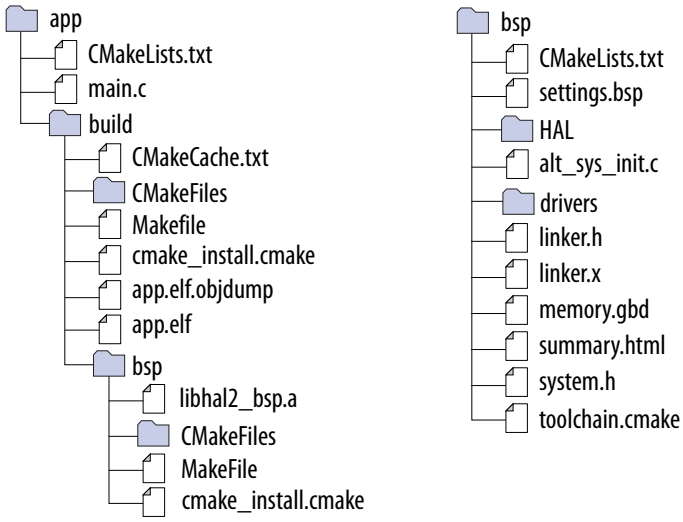
Run the commands below to generate single BSP with single application:

```
#Generate BSP files
niosv-bsp --create --quartus-project=top.qpf --qsys=sys.qsys --type=hal
software/bsp/settings.bsp

#Generate App files
niosv-app --bsp-dir=software/bsp --app-dir=software/app --srcs=software/app

#Build App
cmake -B software/app/build -S software/app -G ”Unix Makefiles”
make -C software/app/build
```

4.2.1.2. File Tree Directory



4.2.2. Building Single Application, Single Library, and BSP

The Nios V processor applications (app) inherit all toolchain or compile options settings and able to link against the BSP as a library (this includes toolchain executables, compile flags, and definitions).

4.2.2.1. Build Commands

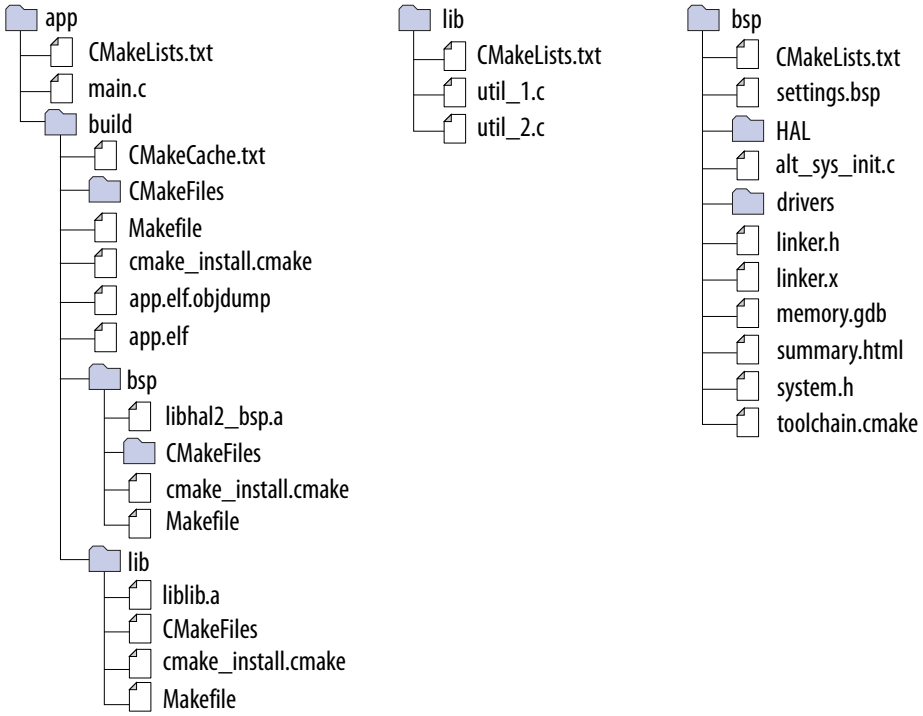
```
#Generate BSP files
niosv-bsp --create --quartus-project=top.qpf --qsys=sys.qsys --type=hal
software/bsp/settings.bsp
```

```
#Generate Library files
niosv-app --lib-dir=software/lib --public-incs=software/lib
--srcs=software/lib/util1.c,software/lib/util2.c

#Generate App files
niosv-app --bsp-dir=software/bsp --link-lib=software/lib
--app-dir=software/app --srcs=software/app

#Build App
cmake -B software/app/build -S software/app -G ”Unix Makefiles”
make -C software/app/build
```

4.2.2.2. File Tree Directory



4.2.3. Building Multiple Applications and Single BSP

This section briefly describes on how to generate multiple applications (called **app1** and **app2**) with a single BSP (**bsp**). The Nios V processor applications, **app1** and **app2** inherit all toolchain/compile options settings and be able to link against the BSP as a library (this includes toolchain executables, compile flags, and definitions).

4.2.3.1. Build Commands

```
#Generate BSP files
niosv-bsp --create --quartus-project=top.qpf --qsys=sys.qsys --type=hal
software/bsp/settings.bsp

#Generate App1 and App2 files
niosv-app --bsp-dir=software/bsp --app-dir=software/app1 --srcs=software/app1
niosv-app --bsp-dir=software/bsp --app-dir=software/app2 --srcs=software/app2

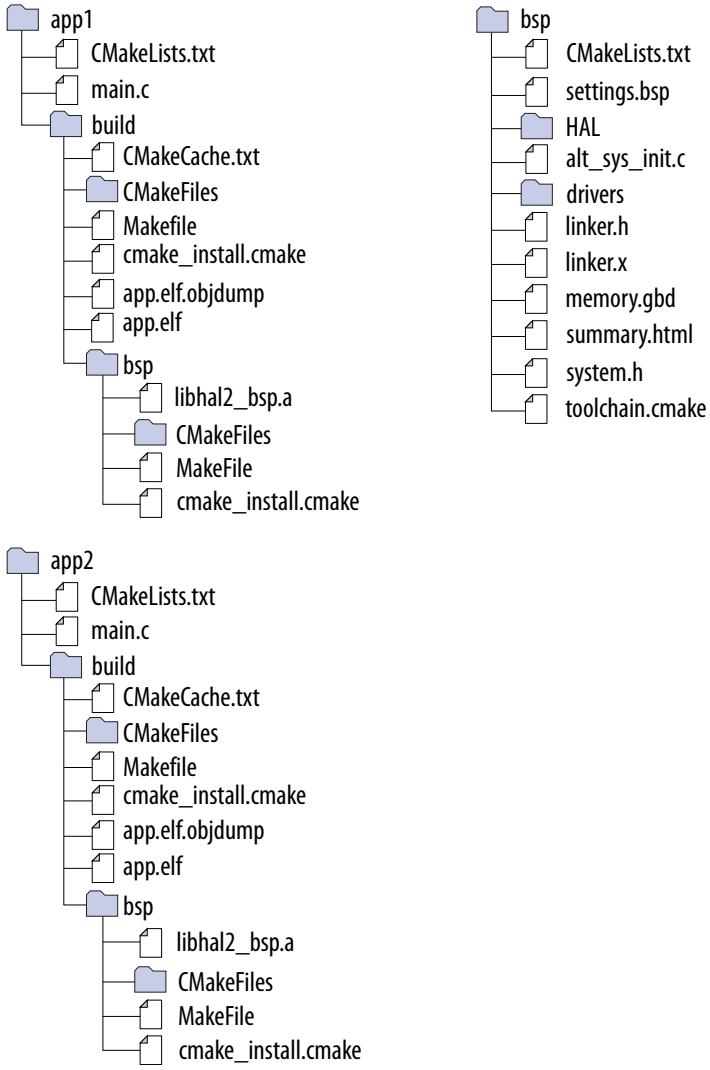
#Build App1 and App2
```

```
cmake -B software/app1/build -S software/app1 -G ”Unix Makefiles”
cmake -B software/app2/build -S software/app2 -G ”Unix Makefiles”

make -C software/app1/build
make -C software/app2/build
```

Note: You can also build multiple application and single BSP using Ashling RiscFree IDE for Altera FPGAs after generating multiple application files using Nios V Command Shell.

4.2.3.2. File Tree Directory



4.2.4. Building BSP Library

The `lib<os>_bsp.a` file is built independently in every Nios V processor application's build directory. If you want to use the `lib<os>_bsp.a` file directly, you may generate the BSP library file in the `bsp` directory using the provided commands.

4.2.4.1. Build Commands

Run the commands below to generate the BSP library only:

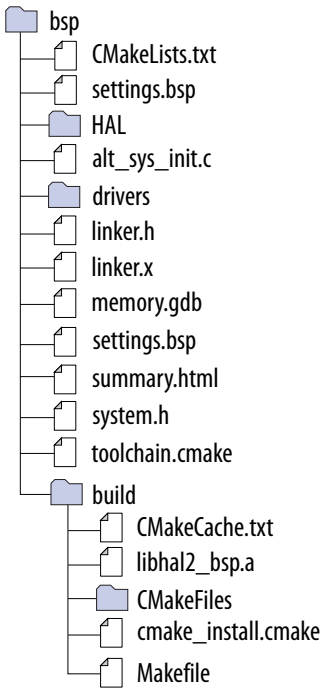
```
#Generate BSP
niosv-bsp -c --quartus-project=top.qpf --qsys=sys.qsys --type=hal software/bsp/
settings.bsp

cmake -B software/bsp/build -S software/bsp -G ”Unix Makefiles”

make -C software/bsp/build
```

Note: When BSP library file is built, it does not include `_sw.tcl` packages which defines pre-compiled libraries. The libraries are exposed only to the Nios V processor application via the CMake variable and linked in during the `.elf` creation.

4.2.4.2. File Tree Directory



4.3. Nios V Processor Software Development Flow

The Nios V processor software generation process includes the following stages:

Table 14. Nios V Processor Software Generation Stages

Stage	Description
Hardware configuration	• Obtain the hardware design on which the Nios V processor software runs. You need to have the Platform Designer System File (`.qsys`) of the Nios V processor-based system. • You can obtain the `.qsys` file from: — A Nios V processor design example, please refer to Nios V Design Example Scripts for more information. — Your design from Platform Designer
BSP configuration	1. Decide what features the BSP requires. (e.g., RTOS, specialized software support, small memory footprint, etc) 2. Define BSP by specifying the components in the BSP and relevant settings, then generate the `settings.bsp` file 3. Create a BSP `CMakeLists.txt` using the `niosv-bsp` utility
Optional user library configuration	1. Include the custom software user library source files in a single directory 2. Create a user library `CMakeLists.txt` using the `niosv-app` utility
Application project configuration	1. Develop the application code based on the hardware design. There are some example codes in Nios V Design Example Scripts for your reference. 2. Create an application `CMakeLists.txt` using the `niosv-app` utility
Software System Design	• Developing with the Hardware Abstraction Layer (HAL) • Programme the Nios V processor • Writing exception handlers • Optimize the application for performance and size • Real-time operating system (RTOS) integration
Application deployment	• Linking (run-time memory) • Boot loading the system application • Programming flash memory

If you change the hardware system, you must recreate, update, or regenerate the BSP project to keep the library header files up to date. For information about how to keep your BSP up to date with your hardware, refer to Revising your BSP.

4.3.1. Getting Started

The Nios V processor tools provide many services and functions for your use. Until you become familiar with these services and functions, Altera recommends that you follow the following guidelines to simplify your development effort:

- Begin with a known hardware design— Refer to Nios V Design Example Scripts that includes a set of known working designs, which are excellent starting points for your own design.
- Begin with a known software example design—The Nios V Design Example Scripts includes a set of preconfigured application projects for you to use as the starting point of your own application. Use one of these designs and parameterize it to suit your application goals.
- Follow pointers to documentation—Many of the application and BSP project source files include inline comments that provide additional information.
- Make incremental changes—Regardless of your end-application goals, develop your software application by making incremental, testable changes, to compartmentalize your software development process. Altera recommends that you use a version control system to maintain distinct versions of your source files as you develop your project.

4.3.2. Configuring BSP Projects

The purpose of the BSP project is to allow you to customize Nios V processor software for your purpose, be it size, speed or other features. This customization is compiled into a BSP library file (`.a`) that you can then use with your application projects.

Creating the BSP project populates the target directory with the BSP library file source and build file. Some of these files are copied from other directories and are overwritten when you recreate the BSP project.

The most basic tool for configuring BSPs is the BSP setting. Throughout this section, many of the project modifications that you can make are based on BSP settings. In each case, this section presents the names of the relevant settings, and explains how to select the correct setting value. You can control the value of BSP settings several ways: on the command line, or by directly adjusting the settings with the BSP Editor.

Another powerful tool for configuring a BSP is the software package. Software packages add complex capabilities to your BSP. As when you work with BSP settings, you can add and remove software packages on the command line, or with the BSP Editor in Platform Designer. Altera recommends that you use the BSP Editor to configure your BSP project.

For detailed information about how to manipulate BSP settings and add and remove software packages with the BSP Editor, refer to Board Support Package Editor.

Note: Do not edit BSP files directly. Use the BSP Editor or Nios V processor utilities instead. If you edit directly, you lose your edit as BSP files are overwritten and regenerated by the BSP Editor.

Related Information

Board Support Package Editor on page 10

4.3.2.1. Selecting the Operating System

You have a choice of the following operating systems to incorporate in your BSP library file:

- Intel Hardware Abstraction Layer (HAL)
- Micrium MicroC/OS-II
- FreeRTOS*

Note: You cannot change the operating system for the BSP project once you have selected it.

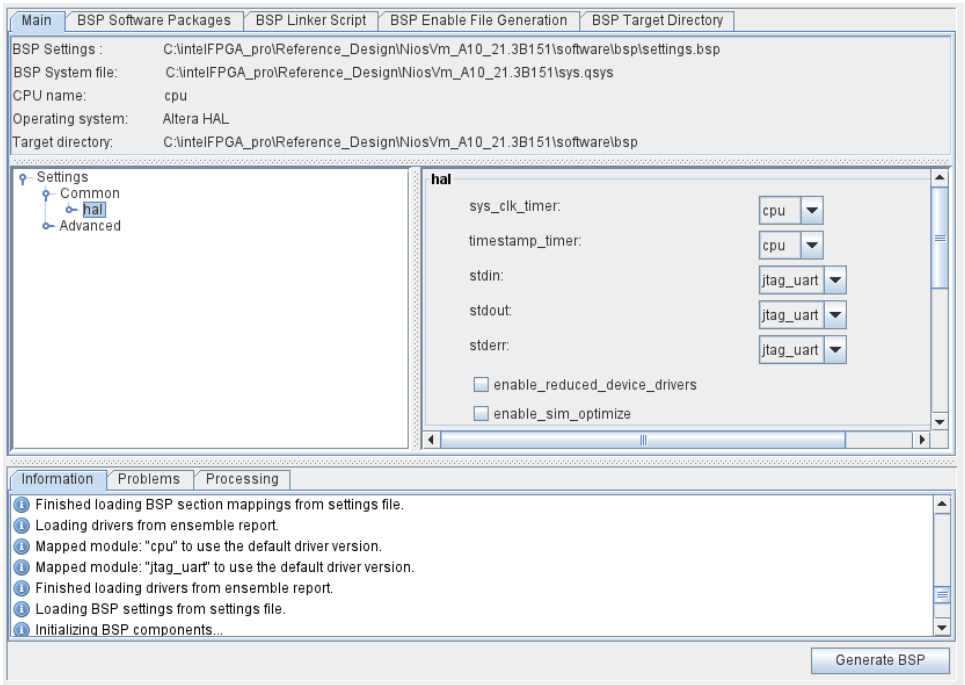
4.3.2.2. Intel HAL Configuration Tips

If you use the HAL environment, be aware of the following properties of this environment:

- **HAL BSP settings**: The **Settings** tab in the BSP Editor shows a comprehensive list of options.
- **HAL setting modification**: Modifying the HAL options modifies the `system.h` file, which is used to compile the BSP library file.
- **HAL initialization**: The HAL is initialized during the execution of the C run-time initialization (crt0) code block. After the crt0 code block runs, the HAL resources are available for your application to use.

You can configure the HAL in the BSP Editor. The figure and Tcl script below shows how you specify a `jtag_uart` to be used as the stdio device.

Figure 3. Configuring HAL stdio Device in BSP Editor



Example 1. Configuring HAL stdio Device using Tcl Script

```
#set up stdio file handles to point to the jtag_uart component
set default_stdio jtag_uart
set_setting hal.stdin $default_stdio
set_setting hal.stdout $default_stdio
set_setting hal.stderr $default_stdio
```

Related Information

Settings Managed by Nios V Processor Board Support Package Editor on page 296
For more comprehensive list of BSP settings for HAL.

4.3.2.3. Micrium MicroC/OS-II Configuration tips

If you use the MicroC/OS-II environment, be aware of the following properties:

- **MicroC/OS-II BSP settings**: The MicroC/OS-II supports many configuration options. All these options can be enabled and disabled with BSP settings.
- **MicroC/OS-II setting modification**: Modifying the MicroC/OS-II options modify the `system.h` file, which is used to compile the BSP library file.
- **MicroC/OS-II initialization**: The core MicroC/OS-II is initialized during the execution of the C run-time initialization (crt0) code block. After the crt0 code block runs, the MicroC/OS-II RTOS resources are available for your application to use.

You can configure MicroC/OS-II with the BSP Editor. Figure Enabling MicroC/OS-II Timers and Queues in BSP Editor shows how you enable the MicroC/OS-II timer and queue code. Figure Configuring MicroC/OS-II for Four Timers in BSP Editor shows how you specify a maximum of four timers for use with MicroC/OS-II. The MicroC/OS-II Tcl script below performs the same MicroC/OS-II in these two figures.

Figure 4. Enabling MicroC/OS-II Timers and Queues in BSP Editor

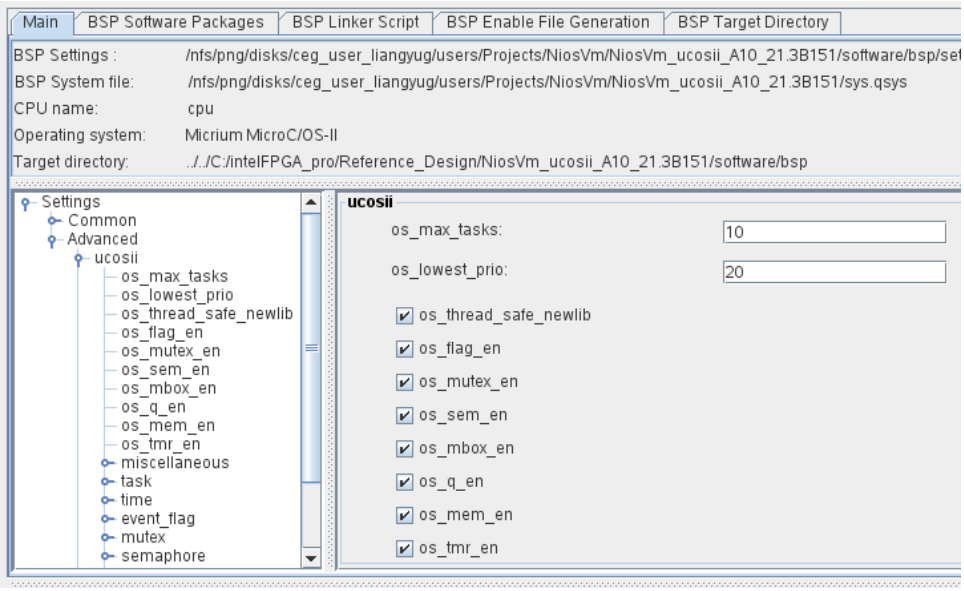
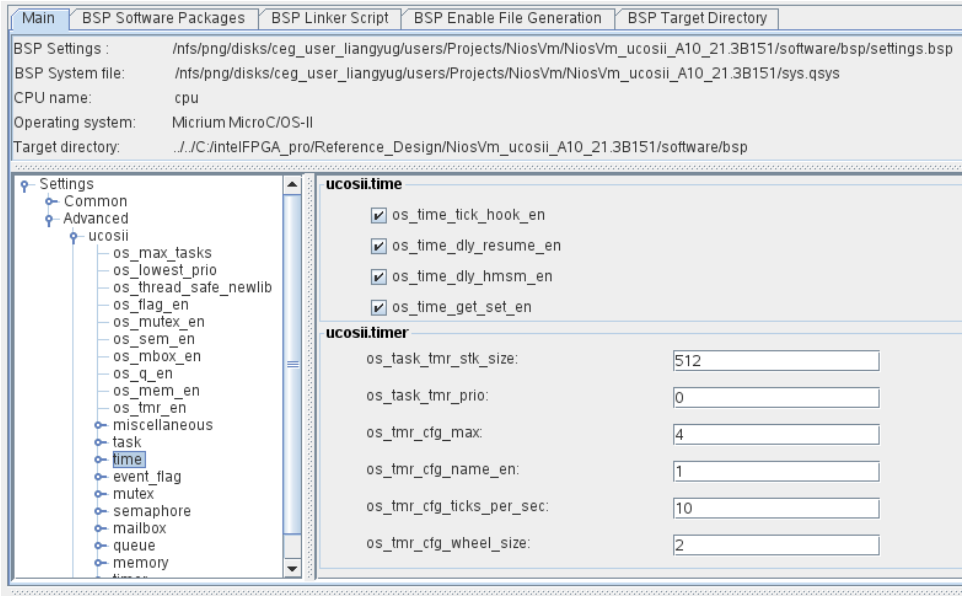


Figure 5. Configuring MicroC/OS-II for Four Timers in BSP Editor



Example 2. Configuring MicroC/OS-II timer Device using Tcl Script

```
#enable code for UCOSII timers
set_setting ucosii.os_tmr_en 1

#enable code for UCOSII queues
set_setting ucosii.os_q_en 1

#enable a maximum of 4 UCOSII timers
set_setting ucosii.timer.os_tmr_cfg_max 4
```

Related Information

Settings Managed by Nios V Processor Board Support Package Editor on page 296
For more comprehensive list of BSP settings for HAL.

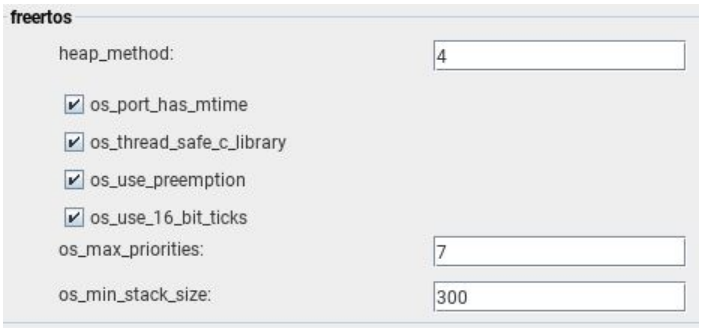
4.3.2.4. Configuring FreeRTOS

Consider the following properties if you are using the FreeRTOS environment:

- **FreeRTOS BSP settings**: The FreeRTOS supports advanced configuration options. All these options can be enabled and disabled with BSP settings.
- **FreeRTOS setting modification**: Modifying the FreeRTOS options modify the `system.h` file, which is used to compile the BSP library file.
- **FreeRTOS initialization**: The core FreeRTOS is initialized during the execution of `alt_main()`, which is called from the C run-time initialization (`crt0`) code block. After the `crt0` code block runs, the FreeRTOS resources are available for your application to use.

You can configure FreeRTOS with the BSP Editor as shown in the following figure :

Figure 6. FreeRTOS Configuration in the BSP Editor



- Specify `heap_4.c` source file
- Maximum number of priorities at 7
- Minimum stack size at 300
- Enable the FreeRTOS features.

Alternatively, you can use the following FreeRTOS Tcl script to perform the same setting as shown in the figure above.

Configuring FreeRTOS using Tcl script

```
#Specify heap method
set_setting freertos.heap_method 4

#Enable FreeRTOS features
set_setting freertos.os_port_has_mtime 1
set_setting freertos.os_thread_safe_c_library 1
set_setting freertos.os_use_preemption 1
set_setting freertos.os_use_16_bit_ticks 1

#Specify maximum number of priorities supported
set_setting freertos.os_max_priorities 7

#Specify minimum stack size
set_setting freertos.os_min_stack_size 300
```

Related Information

Settings Managed by Nios V Processor Board Support Package Editor on page 296
For more comprehensive list of BSP settings for HAL.

4.3.2.5. Adding Software Package

Intel supplies several add-on software packages with the Quartus Prime. The operating system determines which software packages are available.

These software packages are available for your application to use, and can be configured in the BSP Editor from the **Software Packages** tab. The **Software Packages** tab allows you to insert and remove software packages in your BSP, and control software package settings. The software package table at the top of this tab lists each available software package. The table allows you to select the software package version and enable or disable the software package.

4.3.2.6. Using Tcl Script with BSP Editor

The BSP Editor supports Tcl scripting. Tcl scripting in the BSP Editor is a simple but powerful tool that allows you to easily migrate settings from one BSP to another. This feature is especially useful if you have multiple software projects utilizing similar BSP settings.

Tcl scripts in the BSP editor allow you to perform the following tasks:

- Regenerate the BSP from the command line
- Export a TCL script from an existing BSP as a starting point for a new BSP
- Recreate the BSP on a different hardware platform

You can configure a BSP either by importing your own manually created Tcl script, or by using a Tcl script exported from the BSP Editor.

You can apply a Tcl script only at the time that you create the BSP.

4.3.2.7. Exporting Tcl Scripts with BSP Editor

To export a Tcl script, follow these steps:

1. Use the BSP Editor to configure the BSP settings in an existing BSP project.
2. In the **Files** menu in the Platform Designer, click **Export BSP as Tcl Script**.
3. Navigate to the directory where you want to store your Tcl script.
4. Select a file name for the Tcl script.

BSP Editor export all settings including software packages, drivers, linker scripts settings to the TCL script.

Example 3. Tcl Script Exported by BSP Editor

```
# Remove existing memory regions and section mappings
foreach region_info [get_current_memory_regions] {
    delete_memory_region [lindex $region_info 0]
}

foreach mapping_info [get_current_section_mappings] {
    delete_section_mapping [lindex $mapping_info 0]
}

# Settings
set_setting
altera_avalon_jtag_uart_driver.enable_jtag_uart_ignore_fifo_full_error {false}
set_setting altera_avalon_jtag_uart_driver.enable_small_driver {false}
set_setting hal.enable_c_plus_plus {true}
set_setting hal.enable_clean_exit {true}
set_setting hal.enable_exit {true}
set_setting hal.enable_instruction_related_exceptions_api {false}
set_setting hal.enable_lightweight_device_driver_api {false}
set_setting hal.enable_reduced_device_drivers {false}
set_setting hal.enable_runtime_stack_checking {false}
set_setting hal.enable_sim_optimize {false}
set_setting hal.linker.allow_code_at_reset {true}
set_setting hal.linker.enable_alt_load {true}
set_setting hal.linker.enable_alt_load_copy_exceptions {false}
set_setting hal.linker.enable_alt_load_copy_rodata {false}
set_setting hal.linker.enable_alt_load_copy_rwdata {true}
set_setting hal.linker.enable_exception_stack {false}
set_setting hal.linker.exception_stack_memory_region_name {ram}
set_setting hal.linker.exception_stack_size {1024}
set_setting hal.log_flags {0}
```

```
set_setting hal.log_port {none}
set_setting hal.make.asflags {-Wa,-gdwarf2}
set_setting hal.make.cflags_debug {-g}
set_setting hal.make.cflags_defined_symbols {none}
set_setting hal.make.cflags_optimization {-O2}
set_setting hal.make.cflags_undefined_symbols {none}
set_setting hal.make.cflags_user_flags {none}
set_setting hal.make.cflags_warnings {-Wall -Wformat-security}
set_setting hal.make.cxx_flags {none}
set_setting hal.make.enable_cflag_fstack_protector_strong {false}
set_setting hal.make.enable_cflag_wformat_security {false}
set_setting hal.make.link_flags {none}
set_setting hal.make.objdump_flags {-Sdtx}
set_setting hal.max_file_descriptors {32}
set_setting hal.stderr {niosv_jtag_uart_ver2}
set_setting hal.stdin {niosv_jtag_uart_ver2}
set_setting hal.stdout {niosv_jtag_uart_ver2}
set_setting hal.sys_clk_timer {cpu}
set_setting hal.timestamp_timer {cpu}
set_setting hal.toolchain.ar {riscv32-unknown-elf-ar}
set_setting hal.toolchain.as {riscv32-unknown-elf-gcc}
set_setting hal.toolchain.cc {riscv32-unknown-elf-gcc}
set_setting hal.toolchain.cxx {riscv32-unknown-elf-g++}
set_setting hal.toolchain.enable_executable_overrides {false}
set_setting hal.toolchain.objdump {riscv32-unknown-elf-objdump}
set_setting hal.toolchain.prefix {riscv32-unknown-elf-}
set_setting intel_niosv_m_hal_driver.internal_timer_ticks_per_sec {1000}

# Software packages

# Drivers
set_driver intel_niosv_m_hal_driver cpu
set_driver altera_avalon_jtag_uart_driver niosv_jtag_uart_ver2

# User devices

# Linker memory regions
add_memory_region reset ram 0 32
add_memory_region ram ram 32 163808

# Linker section mappings
add_section_mapping .text ram
add_section_mapping .rodata ram
add_section_mapping .rwdata ram
add_section_mapping .bss ram
add_section_mapping .heap ram
add_section_mapping .stack ram
add_section_mapping .exceptions ram
```

4.3.2.8. Importing Tcl Script to Create a New BSP

The following example illustrates how to configure a new BSP with an imported Tcl script. You import the Tcl script with the BSP Editor when you create a new BSP settings file.

In this example, you create the Tcl script by hand, with a text editor. You can also use a Tcl script exported from another BSP, as described in *Exporting a Tcl Script*.

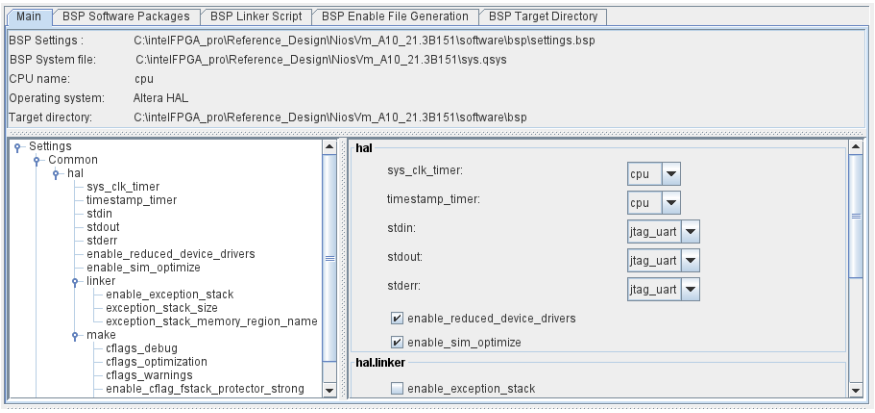
To configure a new BSP with a Tcl script, follow these steps:

1. With any text editor, create a new file called `example.tcl`.
2. Insert the below contents in the file.

```
set_setting hal.enable_reduced_device_drivers true
set_setting hal.enable_sim_optimize true
```

3. In the **Platform Designer**, in the File menu, click **New BSP**.
4. In the **BSP Settings File Name** box, select a folder in which to save your new BSP settings file. Accept the default settings file name, `settings.bsp`.
5. In the **Operating System** list, select **Intel HAL**.
6. In the **Additional Tcl script** box, navigate to `example.tcl`.
7. In the **Platform Designer Information File Name box**, select the `.qsys` file.
8. Click **OK**. The BSP Editor creates the new BSP. The settings modified by `example.tcl` appear as in the following figure.

Figure 7. Nios V Processor BSP Settings (Configured with `example.tcl`)



Note: Do not attempt to import an Intel HAL Tcl script to a RTOS BSP or vice versa. Doing so can result in errors. Some BSP settings are OS-specific, making scripts from different OSes incompatible.

For more information about commands that can appear in BSP Tcl scripts, refer to Board Support Package Tcl Commands on page 333.

4.3.3. Configuring the Application Project

You can configure the application project by specifying source files and a valid BSP project, along with other command line options to the `niosv-app` command.

Related Information

Board Support Package Editor on page 10

4.3.3.1. Application Configuration Tips

Use the following tips to increase your efficiency in designing your application project:

- **Source file inclusion**—To add source files to your project, drag them from a file browser, such as Windows Explorer, and drop them in the **Project Explorer** view in the Ashling RiscFree IDE for Altera FPGAs.

From the command line with `niosv-app`, you may use the `--srcs` argument to specify the source files in your application project command line option.

4.3.3.2. Linking User Libraries

You can create and use your own user libraries in the Nios V processor tools. The Nios V processor tools enables you to create user libraries in the Nios V Command Shell, as follows:

- Create the library using the `niosv-app -l=libdir` command. This command generates a `CMakeLists.txt` file.

Configure the application project with the new library by running the `niosv-app -a=appdir --link-lib=libdir` command to link library to application. The value for the option specifies the path to the library's directory

4.3.4. Building and Running Nios V Processor Software

4.3.4.1. Building the Nios V Processor Software

After editing the BSP settings, and generating the BSP, you can build your project. Below is the CLI flow to build the project once the BSP is generated:

1. Use niosv-app utility to generate and configure an application project.
2. Based in user interface:
   a. CLI (Nios V Command Shell)
      i. Use `cmake` command to generate a project build-system.
      ii. Use `make` command to generate the project `.elf` file.
   b. GUI (Ashling RiscFree IDE for Altera FPGAs)
      i. Import the project to Ashling RiscFree IDE for Altera FPGAs
      ii. Use the **Build** option to build the project.

Note: You need to install the necessary open-source tools before running `cmake` and `make`. For more information, refer to Other Utilities Tools.

4.3.4.2. Downloading and Running the Nios V Processor Software

Download the `.elf` file into a Nios V processor using one of the following method:

- `niosv-download` command in the Nios V Command Shell. Then, reset the Nios V processor system to run your program.
- **Debug As** or **Run As** option using Ashling RiscFree IDE for Altera FPGAs

4.3.4.2.1. Communicating with the Target Device

If you configured your application to use the `stdin` function via a JTAG UART interface, you can use the `juart-terminal` application to communicate with your target processor subsystem.

On the command line, you must use the `juart-terminal` application to communicate with your target. Ensure that your Intel FPGA is configured with the target hardware image. To start the application, type the following command:

```
juart-terminal
```

Characters you type in the shell are transmitted to the target. You can also configure the `juart-terminal` in the Ashling RiscFree IDE for Altera FPGAs.

Related Information

Performing External Tools Configuration
For more information about configuring juart-terminal in the Ashling RiscFree IDE for Altera FPGAs.

4.3.4.3. Nios V Processor Software Debugging

This section describes how to debug a Nios V program using the Ashling RiscFree IDE for Altera FPGAs. You can debug a Nios V processor program on Nios V processor hardware such as a Nios V processor development board.

Note: Do not select **Local C/C++ Application**. Nios V processor projects can only be run and debugged with Nios V processor run configurations.

Related Information

Ashling* RiscFree* Integrated Development Environment (IDE) for Altera FPGAs User Guide

4.3.4.4. Run Time Stack Checking

For debugging purposes, it is useful to enable run-time stack checking, using the `hal.enable_runtime_stack_checking` BSP setting. When properly used, this setting enables the debugger to take control if Nios V processor detects a stack collision has occurred in `sbrk`.

Related Information

Settings Managed by Nios V Processor Board Support Package Editor on page 296
For more comprehensive list of BSP settings for HAL.

4.3.4.5. Ensuring Software Project Coherency

This section discusses several designs and software architecture practices that can help you avoid software project coherency. For example, a new version of a system hardware project is not coherent with the previous version of the software project.

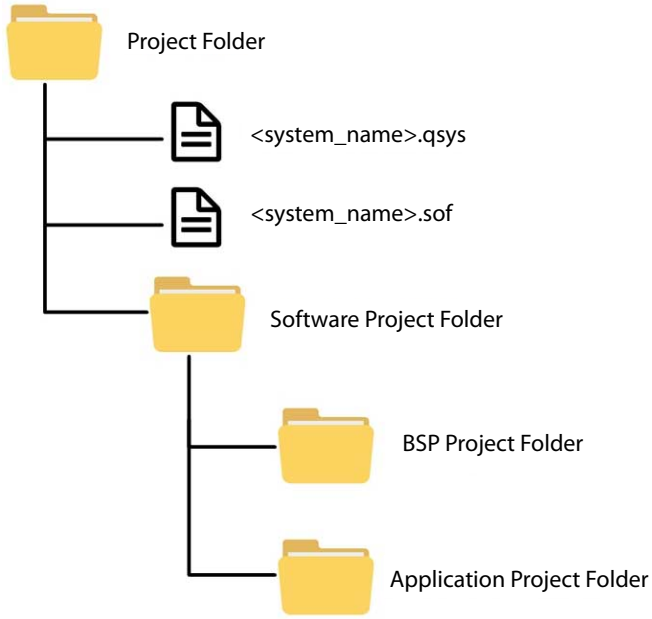
4.3.4.5.1. Recommended Development Practice

The safest software development practice for avoiding the software coherency problem is to follow a strict hardware and software project hierarchy and to use Nios V utilities to generate your application and BSP projects.

One best practice is structuring your application hierarchy with parallel application projects and BSP project folders. In the following recommended directory structure, a top-level hardware project folder includes:

- The Quartus Prime project file.
- The Platform Designer-generated files.
- The Software Project Folder, which contains the following subfolders:
  - Application project
  - BSP project.

Figure 8. Recommended Directory Structure



The complete system generation process, from hardware to BSP and application projects, must be repeated every time a change is made to the system in Platform Designer. Therefore, defining all your settings in your **niosv-bsp** command Tcl script is more efficient than using the BSP Editor to customize your project.

The system generation process follows:

1. **Hardware files generation**: Using Platform Designer, write the updated system description to the `<system_name>.qsys` file.
2. **Regenerate BSP project**: Generates the BSP project with the **niosv-bsp** command.
3. **Regenerate application project**: Generates the application project with the **niosv-app** command.
4. **Build the system**: Builds the system software using the application and BSP `CMakeLists.txt` file.

To implement this system generation process, Altera recommends that you use the following checklists for handing off responsibility between the hardware and software groups.

Note: This method assumes that the hardware and software engineering groups use the same Quartus Prime version.

Table 15. Checklists for Handing Off Responsibility between Groups

To hand off the project from the hardware group to the software group, follow this checklist:	
Hardware project hand-off	The hardware group provides copies of the `<system_name>.qsys` and `<system_name>.sof` files. The software group copies these files to the software group's hardware project folder.
Create software project	The software group creates the software application for the new hardware by running the niosv-bsp and niosv-app utilities.
Build software project	The software group runs `cmake` and `make` commands in its application project directory to build the software application.
Reconfigure software project	The hardware hands over the updated `<system_name>.qsys` and `<system_name>.sof` files. The software group runs the `niosv-app` and `niosv-bsp` command to reconfigure the group's application and BSP projects.
Rebuild software project	The software group runs cmake and make commands in its application project directory to build the software application.
To hand off the project from the software group to the hardware group, follow this checklist:	
Software project hand-off	The software group builds and provides the hardware group with a copy of the `<app_name>.elf` software binary. The hardware group then converts the software binary into the relevant format according to Nios V processor boot methods.

4.3.4.5.2. Recommended Architecture Practice

Many of the hardware and software coherency issues that arise during the creation of the application software are problems of misplaced peripheral addresses. Because of the flexibility provided by Platform Designer, almost any peripheral in the system can be assigned an arbitrary address, or have its address modified during system creation.

Implement the following practices to prevent this type of coherency issue during the creation of your software application:

- **Peripheral and Memory Addressing** —The Nios V processor tools automatically generate a system header file, `system.h`, that defines a set of `#define` symbols for every peripheral in the system. These definitions specify the peripheral name, base address location, and address span.

To protect against coherency issues, access all system peripherals and memory components with their `system.h` name and address span symbols. This method guarantees successful peripheral register access even after a peripheral's addressable location changes.

For example, if your system includes a UART peripheral named UART1 (located at address 0x1000), access the UART1 registers using the `system.h` address symbol `(IOWR_32(UART1_BASE, 0x0,0x10101010))` rather than using its address `(IOWR_32(0x1000, 0x0,0x10101010))`.

4.4. Developing with the Hardware Abstraction Layer

The HAL for the Nios V processor is a lightweight run-time environment that provides a simple device driver interface for programs to communicate with the underlying hardware. The HAL API is integrated with the ANSI C standard library. The HAL API allows you to access devices and files using C library functions.

Related Information

- Overview of the Hardware Abstraction Layer on page 79
- HAL Peripheral Services on page 45
  For information about configuring and using HAL services.
- Reducing Code Footprint in Embedded Systems on page 108
  For information about increasing your application's performance, or minimizing the code footprint.

4.4.1. Hardware Abstraction Layer Services

4.4.1.1. HAL Configuration Options

The HAL BSP library is self-configuring to some extent to support the Nios V software development flow. By design, the HAL attempts to enable as many services as possible based on the peripherals present in the system hardware. This approach provides your application with the least restrictive environment possible, a helpful feature during the product development and board brings-up cycle.

The HAL is configured with a group of settings whose values are determined by Tcl commands, which are called during the creation of the BSP project. Altera recommends you create a separate Tcl file that contains your HAL configuration settings.

HAL configuration settings control the boot loading process and provide detailed control over the initialization process, system optimization, and the configuration of peripherals and services. For each of these topics, this section provides pointers to the relevant material elsewhere in this section.

4.4.1.2. Configuring the Boot Environment

Your particular system may require a bootloader to configure the application image before execution. For example, suppose your application image is stored in flash memory and copied to volatile memory for performance. In that case, a bootloader must configure the application image in the volatile memory. This configuration process occurs before the HAL BSP library configuration routines execute and before the crt0 code block executes. A bootloader implements this process.

Related Information

- crt0 Initialization on page 43
- Nios® V Processor Configuration and Booting Solutions

4.4.2. System Startup in HAL-based Application

System startup in HAL-based applications is a three-stage process.

1. The hardware system initializes
2. The `crt0` code runs
3. The HAL services initialize

The following sections describe these three system-startup stages.

4.4.2.1. System Initialization

The system initialization sequence begins when the system powers up. The initialization sequence steps for FPGA designs that contain a Nios V processor are the following:

1. **Hardware reset event**: The board receives a power-on reset signal, which resets the FPGA.
2. **FPGA configuration**: The FPGA is programmed with a `.sof` file, from a specialized configuration memory or an external hardware master.
3. **System reset**: The Platform Designer system, composed of one or more Nios V processors and other peripherals, receives a hardware reset signal and enters the components' combined reset state.
4. **Nios V processor**: Nios V processor jumps to its preconfigured reset address and begins running instructions found at this address.
5. **Bootloader (optional)**: Refer to bootloader section for more information.
6. **crt0 execution**: If you used a bootloader, the processor jumps to the `_entry` symbol within the crt0 code. If you did not use a bootloader, the CPU starts from the `_reset` symbol which jumps to the`_entry` symbol, which is the beginning of the crt0 code.

4.4.2.2. crt0 Initialization

The crt0 code block contains the C run-time initialization code—initialization instructions needed to enable execution of C or C++ applications. The Altera provided crt0 block performs the following initialization steps:

1. **Initializes instruction cache**—If the processor has an instruction cache, this cache is initialized.

   Note: Platform Designer determines the processors that have instruction caches and configures these caches at system generation. The Nios V processor tools insert the instruction-cache initialization code block if necessary.
2. **Initializes data cache**—If the processor has a data cache, this cache is initialized. As with the instruction caches, this code is enabled if the processor has a data cache.
3. **Initializes exception vector (mtvec) and interrupt controller settings**—The exception vector and interrupt controller settings are initialized as configured in the hardware design.
4. **Set the global pointer**—The global pointer register is initialized.
5. **Sets the stack pointer**—The stack pointer is initialized. You can configure the stack pointer address.
6. **Clears the .bss section**—The .bss section is initialized to all zeros. You can set the .bss section address.
7. **Calls alt_load macros**—If the application is designed to run from flash memory (the .text section runs from flash memory), the remaining sections are copied to volatile memory.
8. **Jumps to alt_main()**—The processor jumps to the `alt_main()` function, which begins initializing the HAL BSP run-time library.

Note: If you use a third-party RTOS or environment for your BSP library file, the `alt_main()` function can be different than the one provided by Intel.

If you use a third-party compiler or library, the C run-time initialization behavior may differ from this description.

The `crt0.S` source file is generated when generating the Nios V processor BSP under `<bsp_dir>/HAL/src` directory. The crt0 code includes initialization short-cuts only if you perform hardware simulations of your design. You can control these optimizations by the `hal.enable_sim_optimize` setting.

For more information about the `hal.enable_sim_optimize` BSP setting, refer to *Settings Managed by Nios V Processor Board Support Package Editor*.

Related Information

- Settings Managed by Nios V Processor Board Support Package Editor on page 296
  For more comprehensive list of BSP settings for HAL.
- Configuring the Boot Environment on page 42
  For more information about Calls alt_load macros.

4.4.2.3. HAL Initialization

As for any other C program, the first part of the HAL's initialization is implemented by the Nios V processor's `crt0.S` routine. After `crt0.S` completes the C run-time initialization, it calls the HAL `alt_main()` function, which initializes the HAL BSP run-time library and subsystems.

The HAL `alt_main()` function performs the following steps:

1. Initializes interrupts—Sets up interrupt support for the Nios V processor with the `alt_irq_init()` function.
2. Starts RTOS—Starts the RTOS, if this OS is configured to run (with the `ALT_OS_INIT` and `ALT_SEM_CREATE` functions). For additional information about using and initializing RTOS, refer to Selecting the Operating System.
3. Initializes device drivers—Initializes device drivers (with the `alt_sys_init()` function). The Nios V processor tools automatically find all peripherals supported by the HAL, and automatically insert a call to a device configuration function for each peripheral in the `alt_sys_init()` code. To override this behavior, you can disable a device driver with the BSP Editor, in the Drivers tab. For information about enabling and disabling device drivers, refer to Managing Device Drivers.
4. Configures stdio functions—Initializes stdio services for stdin, stderr, and stdout. These services enable the application to use the GNU newlib stdio functions and map the file pointers to supported character devices. For more information about configuring the stdio services, refer to Character Mode Devices.
5. Initializes C++ global constructors and destructors—Handles initialization of global C++ constructor and destructor functions. These function calls are necessary if your application is written in the C++ programming language.
6. Calls main()—Calls function `main()`, or application program. Most applications are constructed using a `main()` function declaration and begin execution at this function.

By default, the HAL configuration mechanism enables support for the C++ programming language. Disabling this feature reduces your application's code footprint, as noted in Reducing Code Footprint in Embedded System. The Nios V processor C++ language support depends on the GCC tool chain.

4.4.3. HAL Peripheral Services

The HAL provides your application with a set of services, typically relying on the presence of a hardware peripheral to support the services. By default, if you configure your HAL BSP project from the command line by running the **niosv-bsp** command, each peripheral in the system is initialized, and usable as a service at the entry point of your application (`main()`).

This section describes the core set of Intel-supplied, HAL-accessible peripherals and the services they provide for your application. It also describes application design guidelines for using the supplied service, and background and configuration information, where appropriate.

Related Information

- Settings Managed by Nios V Processor Board Support Package Editor on page 296
  For more comprehensive list of BSP settings for HAL.
- Developing Programs Using the Hardware Abstraction Layer on page 84
  For more information about the HAL peripheral services.

4.4.3.1. Timer Devices

The HAL provides two types of timer services, a system clock timer, and a timestamp timer.

- The system clock timer is used to control, monitor, and schedule system events.
- The timestamp variant is used to make high performance timing measurements.

You can assign each of these timer services to the Nios V processor internal timer or an Intel Avalon Timer peripheral.

Note: Nios V processor core includes a timer that provides both the system clock timer and timestamp services.

Related Information

Embedded Peripherals IP User Guide: Interval Timer Core

4.4.3.1.1. System Clock Timer

The system clock timer resource is used to trigger periodic events (alarms), and as a timekeeping device that counts system clock ticks. The default system clock timer in a Nios V processor subsystem is the Nios V processor built-in internal timer. You may configure the system clock timer to other timer peripherals in the Platform Designer system.

Note: Only one system clock timer service may be identified in the BSP library. Only HAL supplied routines access the timer.

The `hal.sys_clk_timer` setting controls the BSP project configuration for the system clock timer. This setting configures one of the timers available in your Platform Designer design as the system clock timer.

Intel provides separate APIs for application-level system clock functionality and for generating alarms.

Application-level system clock functionality is provided by two separate classes of APIs, one HAL specific and the other Unix-like. The Intel function `alt_nticks()` returns the number of clock ticks that have elapsed. You can convert this value to seconds by dividing by the value returned by the `alt_ticks_per_second()` function. For most embedded applications, this function is sufficient for rudimentary time keeping.

The POSIX-like `gettimeofday()`function behaves differently in the HAL than on a Unix workstation. On a workstation, with a battery backed-up, real-time clock, this function returns an absolute time value, with the value zero representing 00:00 Coordinated Universal Time (UTC), January 1, 1970, whereas in the HAL, this function returns a time value starting from system power-up. By default, the function assumes system power-up to have occurred on January 1, 1970. Use the `settimeofday()` function to correct the HAL `gettimeofday()` response. The `times()` function exhibits the same behavior difference.

Consider the following common issues and important points before you implement a system clock timer:

- **System Clock Resolution**—The timer's period value specifies the rate at which the HAL BSP project increments the internal variable for the system clock counter. If the system clock increases too slowly for your application, you can decrease the timer's period in Platform Designer.
- **Rollover**—The internal, global variable that stores the number of system clock counts (since reset) is a 64-bit unsigned integer. No rollover protection is offered for this variable. Therefore, calculate when the rollover event occurs in your system and plan the application accordingly.
- **Performance Impact**—Every clock tick causes the execution of an interrupt service routine. Executing this routine leads to a minor performance penalty. If your system hardware specifies a short timer period, the cumulative interrupt latency impacts your overall system performance.

The alarm API allows you to schedule events based on the system clock timer, in the same way an alarm clock operates. The API consists of the `alt_alarm_start()` function, which registers an alarm, and the `alt_alarm_stop()` function, which disables a registered alarm.

Consider the following common issues and important points before you implement an alarm:

- **Interrupt Service Routine (ISR) context**—A common mistake is to program the alarm call-back function to call a service that depends on interrupts being enabled (such as the `printf()` function). This mistake causes the system to deadlock, because the alarm call-back function occurs in an interrupt context, while interrupts are disabled.
- **Resetting the alarm**—The callback function can reset the alarm by returning a nonzero value. Internally, the `alt_alarm_start()` function is called by the call-back function with this value.
- **Chaining**—The `alt_alarm_start()` function is capable of handling one or more registered events, each with its own call-back function and number of system clock ticks to the alarm.
- **Rollover**—The alarm API handles clock rollover conditions for registered alarms seamlessly.

A good timer period for most embedded systems is 50 milliseconds. This value provides enough resolution for most system events but does not seriously impact performance nor roll over the system clock counter too quickly. Choosing a timer period that is too short might overwhelm the Nios V processor to continuously servicing the timer interrupt handler, and neglect other tasks.

4.4.3.1.2. Timestamp Timer

The timestamp timer service provides applications with an accurate way to measure the duration of an event in the system. The default timestamp timer in a Nios V processor subsystem is the Nios V processor built-in internal counter. You may alternatively configure the system clock timer to other timer peripherals in the Platform Designer system.

Note: Only one timestamp timer service may be identified in the BSP library file. Only HAL supplied routines access the timer.

The `hal.timestamp_timer` setting controls the BSP configuration for the timer. This setting configures one of the timers available in the Platform Designer design as the timestamp timer.

Intel provides a timestamp API. The timestamp API is very simple. It includes the `alt_timestamp_start()` function, which makes the timer operational, and the `alt_timestamp()` function, which returns the current timer count.

Consider the following common issues and important points before you implement a timestamp timer:

- **Timer Frequency**—The timestamp timer decrements at the clock rate of the clock that feeds it in the Platform Designer system. You can modify this frequency in Platform Designer.
- **Rollover**—The timestamp timer has no rollover event. When the `alt_timestamp()` function returns the value 0, the timer has run down.
- **Maximum Time**—The default Nios V processor internal timer have 64 bits available to store the timer value. For Avalon timer peripherals, it can be 32 or 64 bits depending on the parametrization. Therefore, the maximum duration a timestamp timer can count is $[(1/\text{timer frequency}) \times 2^{bits_value}]$ seconds. (`bits_value` = 32 or 64)

Related Information

HAL API Reference on page 224
For more information about the APIs that control the timestamp and system clock timer services.

4.4.3.2. Character Mode Devices

4.4.3.2.1. stdin, stdout and stderr

The HAL can support the stdio functions provided in the GNU newlib library. Using the stdio library allows you to communicate with your application using functions such as `printf()` and `scanf()`.

Currently, Intel supplies two system components that can support the stdio library, the **UART** and **JTAG UART** components. These devices can function as standard I/O devices.

To enable this functionality, use the `--cmd="set_setting hal.stdin <value>"` arguments during Nios V processor BSP configuration. The stdin character input file variable and the stdout and stderr character output file variables can also be individually configured with the **HAL BSP** settings `hal.stdin`, `hal.stdout`, and `hal.stderr`.

Make sure that you assign values individually for each of the stdin, stdout, and stderr file variables that you use.

4.4.3.2.2. Blocking versus Non-Blocking I/O

Character mode devices can be configured to operate in blocking mode or nonblocking mode. The mode is specified in the device's file descriptor.

- In blocking mode, a function call to read from the device waits until the device receives new data.
- In non-blocking mode, the function calls to read new data returns immediately and reports whether new data was received.

Depending on the function you use to read the file handle, an error code is returned, specifying whether new data arrived.

The UART and JTAG UART components are initialized in blocking mode. However, each component can be made non-blocking with the `fnctl()` or the `ioctl()` function, as seen in the following open system call, which specifies that the device being opened is to function in non-blocking mode:

```
fd = open ("/dev/<your uart name>", O_NONBLOCK | O_RDWR);
```

The `fnctl()` system call shown in the example below specifies that a device that is already open is to function in non-blocking mode.

The behavior of the UART and JTAG UART peripherals can also be modified with an `ioctl()` function call. The `ioctl()` function supports the following parameters:

- For UART peripherals:
  - TIOCMGET (reports baud rate of UART)
  - TIOCMSET (sets baud rate of UART)
- For JTAG UART peripherals:
  - TIOCSTIMEOUT (timeout value for connecting to workstation)
  - TIOCGCONNECTED (find out whether host is connected)

The `altera_avalon_uart_driver.enable_ioctl` BSP setting enables and disables the `ioctl()` function for the UART peripherals. The `ioctl()` function is automatically enabled for the JTAG UART peripherals.

The `ioctl()` function is not compatible with the `altera_avalon_uart_driver.enable_small_driver` and `hal.enable_reduced_driver` BSP settings. If either of these settings is enabled, `ioctl()` is not implemented.

Example 4. fnctl() System Call

```
/* You can specify <file_descriptor> to be
* STDIN_FILENO, STDOUT_FILENO, or STDERR_FILENO
* if you are using STDIO
*/
fnctl(<file_descriptor>, F_SETFL, O_NONBLOCK
```

Example 5. Non-Blocking Device Code Fragment

```
input_chars[128];
return_chars = scanf("%128s", &input_chars);
if(return_chars == 0)
{
if(errno != EWOULDBLOCK)
{
/* check other errnos */
}
}
else
{
/* process received characters */
}
```

4.4.3.2.3. Adding Your Own Character Mode Device

If you have a custom device capable of character mode operation, you can create a custom device driver that the stdio library functions can use.

4.4.3.3. Flash Memory Devices

4.4.3.3.1. Memory Initialization, Querying, and Device Support

Every flash memory device is queried by the HAL during system initialization to determine the kind of flash memory and the functions to manage it. This process is automatically performed by the `alt_sys_init()` function, if the device drivers are not explicitly omitted and the small driver configuration is not set.

After initialization, you can query the flash memory for status information with the `alt_flash_get_flash_info()` function. This function returns a pointer to an array of flash region structures—C structures of type struct flash_region—and the number of regions on the flash device.

For additional information about the struct flash_region structure, refer to the source file **HAL/inc/sys/alt_flash_types.h** in the BSP project directory.

4.4.3.3.2. Accessing Flash Memory

The `alt_flash_open()` function opens a flash memory device and returns a descriptor for that flash memory device. After you complete reading and writing the flash memory call the `alt_flash_close()` function to close it safely.

The HAL flash memory device model provides you with two flash access APIs, one simple and one fine-grained. The simple API takes a buffer of data and writes it to the flash memory device, erasing the sectors if necessary. The fine-grained API enables you to manage your flash device on a block-by-block basis.

Both APIs can be used in the system. The type of data you store determines the most useful API for your application. The following general design guidelines help you determine which API to use for your data storage needs:

- **Simple API**—This API is useful for storing arbitrary streams of bytes if the exact flash sector location is not important. Examples of this type of data are log or data files generated by the system during run-time, which must be accessed later in a continuous stream somewhere in flash memory.
- **Fine-Grained API**—This API is useful for storing units of data, or data sets, which must be aligned on absolute sector boundaries. Examples of this type of data include persistent user configuration values, FPGA hardware images, and application images, which must be stored and accessed in each flash sector (or sectors).

4.4.3.3.3. Configuration and Use Limitations

If you use flash memories in your system, be aware of the following properties of this memory:

- **Code Storage**—If your application runs code directly from the flash memory, the flash manipulation functions are disabled. This setting prevents the processor from erasing the memory that holds the code it is running. In this case, the symbols `ALT_TEXT_DEVICE`, `ALT_RODATA_DEVICE`, and `ALT_EXCEPTIONS_DEVICE` must all have values different from the flash memory peripheral. (Note that each of these `#define` symbols names a memory device, not an address within a memory device).
- **Small Driver**—If the small driver flag is set for the software—the `hal.enable_reduced_device_drivers` setting is enabled—then the flash memory peripherals are not automatically initialized. In this case, your application must call the initialization routines explicitly.
  - **Thread safety**—Most of the flash access routines are not thread-safe. If you use any of these routines, construct your application so that only one thread in the system accesses these functions.

4.4.3.3.4. Direct Memory Access Devices

The HAL Direct Memory Access (DMA) model uses DMA transmit and receive channels. A DMA operation places a transaction request on a channel. A DMA peripheral can have a transmit channel, a receive channel, or both. This section describes three hardware configurations for a DMA peripheral and shows how to activate each kind of DMA channel using the HAL memory access functions.

The DMA peripherals are initialized by the `alt_sys_init()` function call and are automatically enabled by the **niosv-bsp** command.

DMA Configuration and Use Model

The following examples illustrate use of the DMA transmit and receive channels in a system. The information complements the information available in Using DMA Devices. Regardless of the DMA peripheral connections in the system, initialize a transmit channel by running the `alt_dma_txchan_open()` function, and initialize a receive DMA channel by running the `alt_dma_rxchan_open()` function. The following sections describe the use model for some specific cases.

4.4.3.3.5. Unsupported Devices

The HAL provides a wide variety of native devices support for Intel-supplied peripherals. However, your system may require a device or peripheral that Intel does not offer. In this case, one or both of the following two options is available for you:

- Obtain a device through Intel third-party program
- Incorporate your own device

Incorporating your own custom peripheral is a two-stage process. First you must incorporate the peripheral in the hardware using Platform Designer, and then you must develop a device driver.

4.4.4. Handling Exceptions

The HAL infrastructure provides a robust interrupt handling service routine and an API for exception handling. The Nios V processor can handle exceptions caused by hardware interrupts, unimplemented instructions, and software traps.

This section discusses exception handling with the Nios V processor internal interrupt controller.

Consider the following common issues and important points before you use the HAL-provided exception handler:

- **Prioritization of interrupts**: The Nios V processor does not prioritize its 16 platform interrupt, but the HAL exception handler assigns higher priority to lower numbered interrupts. You must modify the interrupt request (IRQ) prioritization of your peripherals in Platform Designer.
- **Nesting of interrupts**: The HAL infrastructure allows interrupts to be nested—higher priority interrupts can preempt processor control from an exception handler that is servicing a lower priority interrupt. However, Altera recommends that you not nest your interrupts because of the associated performance penalty.
- **Exception handler environment**: When creating your exception handler, you must ensure that the handler does not run interrupt-dependent functions and services, because this can cause deadlock. For example, an exception handler should not call the IRQ-driven version of the `printf()` function.

Related Information

Trap Handling on page 154
For information about the exception handler software routines, HAL-provided services, and API.

4.4.4.1. Modifying the Exception Handler

In some very special cases, you can modify the existing HAL exception handler routine or to insert your own interrupt handler for the Nios V processor. However, in most cases you need not modify the interrupt handler routines for the Nios V processor for your software application.

Consider the following common issues and important points before you modify or replace the HAL-provided exception handler:

- **Interrupt vector address**: The interrupt vector address for each Nios V processor is set by the Board Support Package (BSP) files. You can modify it during BSP configuration in the BSP Editor.
- **Modifying the exception handler**: The HAL-provided exception handler is robust, reliable, and efficient. Modifying the exception handler could break the HAL-supplied interrupt handling API, and cause problems in the device drivers for other peripherals that use interrupts, such as the UART and the JTAG UART.

You can modify the behavior of the exception handler to increase overall performance.

5. Nios V Processor Board Support Package Editor

The BSP Editor supports a repeatable and scriptable process for creating your software product. You can invoke the BSP Editor through either of the following user interfaces:

- Platform Designer
- Nios V Command Shell

Before reading this chapter, consider getting an introduction to the build tools from the following references:

- Overview of Nios V Processor Embedded Development
- Getting Started from the Command Line
- Nios V Processor Tools

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

- Micrium MicroC/OS-II.

  Refer to *MicroC/OS-II - The Real Time Kernel* by Jean J. Labrosse (CMP Books) for more information.
- FreeRTOS.

  Refer to *Mastering the FreeRTOS Real Time Kernel* for more information.
- Tcl scripting language.

Related Information

HAL BSP Settings on page 84
For more information about how to control BSP settings.

5.1. Board Support Packages

The BSP Editor supports the following types of BSPs:

- Intel Hardware Abstraction Layer (HAL)
- Micrium MicroC/OS-II.
- FreeRTOS

5.1.1. Overview of BSP Creation

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

© Altera Corporation. Altera, the Altera logo, the 'a' logo, and other Altera marks are trademarks of Altera Corporation. Altera and Intel warrant performance of its FPGA and semiconductor products to current specifications in accordance with Altera's or Intel's standard warranty as applicable, but reserves the right to make changes to any products and services at any time without notice. Altera and Intel assume no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to inwriting by Altera or Intel. Altera and Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

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

5.1.2. Nios V Processor BSP Components

5.1.2.1. Hardware Abstraction Layer

The Hardware Abstraction Layer (HAL) provides a single-threaded UNIX-like C/C++ runtime environment. For more information, refer to Overview of the Hardware Abstraction Layer.

5.1.2.2. newlib C Standard Library

newlib C Standard Library performs the following functions:

- An open source implementation of the C standard library intended for use on embedded systems.
- A collection of common routines such as `printf()`, `malloc()`, and `open()`.

5.1.2.3. Device Drivers

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

- A device driver is associated with a specific hardware component.
- A device driver might have settings that impact its compilation. These settings become part of the BSP settings.

5.1.2.4. Optional Software Packages

A software package is source code that you can optionally add to a BSP project to provide additional functionality.

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

- Not associated with specific hardware.
- Might have settings that impact its compilation. These settings become part of the BSP settings.
- Distinct from a library project. A software package is part of the BSP project, not a separate library project.

5.1.2.5. Optional Real-Time Operating System

The Nios V processor software development flow includes an implementation of third-party RTOS kernels that you can optionally include in your BSP.

The following are some descriptions for Micrium MicroC/OS-II and FreeRTOS:

- MicroC/OS-II and FreeRTOS are built on the HAL and implements a simple, well-documented RTOS scheduler. You can modify settings that become part of the BSP settings. Other operating systems are available from third-party vendors.
- A multi-threaded run-time environment built on the Intel HAL.
- MicroC/OS-II and FreeRTOS directory structures is a superset of the HAL BSP directory structure.
- MicroC/OS-II BSP and FreeRTOS store all HAL BSP generated files.
- MicroC/OS-II source code resides in the `UCOSII` directory. FreeRTOS source code resides in the FREERTOS and `FreeRTOS-Kernel` directories. These directories are contained in the BSP directory, like the HAL directory, and has the same structure (`src` and `inc` directories).

5.2. Common BSP Tasks

The BSP Editor 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 to carry them out.

Although this section describes tasks in terms of the BSP Editor command line flow, you can also carry out most of these tasks from the BSP Editor user interface in Platform Designer.

5.2.1. Creating a BSP for an Intel FPGA Development Board

To create a BSP, perform the following steps:

1. Start the Nios V Command Shell.
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. Create a Nios V processor hardware example corresponding to your Intel FPGA development board. You can refer to the Nios V Processor Design Example Scripts.
5. Create and build the BSP with **niosv-bsp --create** command:

```
niosv-bsp --create --qsys=<Platform Designer system> --type=<os name>
settings.bsp
```

For details about the Nios V Command Shell and **niosv-bsp**, refer to the Getting Started from the Command Line.

5.2.2. Adding the BSP Editor to Tool Flow

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

5.2.2.1. Using Version Control

One common tool flow requirement is version control. When using version control, it is important to know which files to add to your version control database. To ensure reproducible results from software builds, Altera recommends placing the build scripts (such as Platform Designer system script and BSP Tcl script) and application source codes under version control. The build scripts incorporate all information on the system and can be generated using the commands in the following table.

Table 16. Generating Build Scripts

Task	Command
Generate the Platform Designer script	`qsys-generate <Platform Designer system>.qsys --export-qsys-script`
Export the BSP Tcl script	`niosv-bsp --query --export-as-tcl=<BSP Tcl script> settings.bsp`

5.2.2.1.1. Creating BSP by Running User-defined Script to Call niosv-bsp

If you create a BSP by running your own script that calls **niosv-bsp**, you can put your script under version control. If your script provides any BSP TCL scripts to **niosv-bsp**(using the `--script=<Tcl script>` option), you must also put these TCL scripts under version control. If you install a new release of Quartus Prime software 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 BSP Editor.

5.2.2.1.2. Creating Script that Uses the Command Line Tools

If you create a script that uses the command line tools **niosv-bsp** explicitly, or you use these tools directly on the command line, you can create the BSP settings file in a directory different from the directory where the generated BSP files reside. Altera recommends you 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.

5.2.2.2. Copying, Moving, or Renaming a BSP

BSP `CMakeLists.txt` has 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 Platform Designer system file when you create the BSP, you must ensure that the `.qsys` file is still accessible from the new location of the BSP. This file path is stored in the BSP settings file.

You must rebuild the BSP when copying, moving, or renaming a BSP before linking an application with the BSP.

Another way to copy a BSP is to run the **niosv-bsp --generate** command to populate an empty BSP directory and pass it the path to the BSP settings file of the BSP that you want to copy.

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

5.2.2.3. Passing the BSP to Another Developer

If you need to pass a BSP to another developer, save the processor subsystem as a Platform Designer script and export the BSP settings as a BSP TCL script.

Table 17. Generating and Restoring Build Scripts

Task	Command
Generate the Platform Designer script	`qsys-generate <Platform Designer system>.qsys --export-qsys-script`
Export the BSP Tcl script	`niosv-bsp --query --export-as-tcl=<BSP Tcl script> settings.bsp`
Restore the processor subsystem	`qsys-script --script=<Platform Designer script> / --quartus-project=<Quartus Project File>`
Regenerate the BSP *Note:* The person receiving the scripts can regenerate the BSP after restoring the processor subsystem.	`niosv-bsp --create --qsys=<processor subsystem> / --type=<OS type> --script=<BSP TCL script>`

Related Information

Quartus® Prime Pro Edition User Guide: Platform Designer
For more information about creating and restoring a Platform Designer script.

5.2.3. Linking and Locating

When auto-generating a HAL BSP, the BSP Editor makes some reasonable assumptions about how you want to use memory. 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 BSP Editor allows you to control details of a memory usage.

5.2.3.1. Creating Memory Initialization Files

Memory initialization files are used for HDL simulation, for Quartus Prime compilation of initializable FPGA on-chip memories, and for flash programming. Memories that can be initialized include M512, M4K and M20K, but not MRAM.

The BSP Editor does not build the memory initialization files by default. You can generate memory initialization using `elf2hex` command. For more information, refer to File Format Conversion Tools Reference: elf2flash

5.2.3.2. Modifying Linker Memory Regions

If the default linker memory regions do not meet your needs, BSP Tcl commands let you modify the memory regions as you prefer.

Suppose you have a memory region named `onchip_ram`. The following Tcl script example separates the top 1024 bytes of `onchip_ram` to create a new region named `onchip_special`.

For more information about an explanation of each Tcl command used in this example, refer to the Board Support Package Tcl Commands .

```
# 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 **niosv-bsp**, it runs after the default Tcl script runs and sets up two memory regions named `onchip_ram` and `onchip_special`. Pass the Tcl script to **niosv-bsp** as follows:

When creating a new BSP:

```
niosv-bsp --create –-qsys=<Platform Designer system> --type=hal
--script=reserve_1024_onchip_ram.tcl settings.bsp
```

When updating an existing BSP:

```
niosv-bsp --update --script=reserve_1024_onchip_ram.tcl settings.bsp
```

If you update your BSP without providing the `--script` option, your BSP reverts to the default linker memory regions and the `onchip_special` memory region disappears. To preserve it, please provide the `--script` option to your Tcl script.

For more information about using the --script argument, refer to the Calling a Custom BSP Tcl Script.

5.2.3.3. Creating a Custom Linker Section

The BSP Editor provides a Tcl command, `add_section_mapping`, to create a linker section. The default Tcl script creates the default sections (`.entry`, `.exceptions`, `.text`, `.rodata`, `.rwdata`, `.bss`, `.heap`, and `.stack`) using the `add_section_mapping` Tcl command.

5.2.3.3.1. Creating a Linker Section for an Existing Region

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

When creating a new BSP:

```
niosv-bsp –-create –-qsys=<Platform Designer system> --type=hal
--cmd=”add_section_mapping special_section onchip_special” settings.bsp
```

When updating an existing BSP (first update):

```
niosv-bsp –-update --cmd=”add_section_mapping special_section onchip_special”
settings.bsp
```

When the **niosv-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 **niosv-bsp**, the default Tcl script runs before the `--cmd` option that creates the `special_section` section.

If you 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 (`.entry`, `.exceptions`, `.text`, `.rodata`, `.rwdata`, `.bss`, `.heap`, and `.stack`).

5.2.3.3.2. Dividing a Linker Region to Create a New Region and Section

This example works with any hardware design containing an on-chip memory named on_chip_mem connected to a Nios V processor instruction manager.

```
# Get region information for on_chip_mem memory region.
# Returned as a list.
set region_info [get_memory_region on_chip_mem]

# 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 on_chip_mem $slave $offset $new_span
add_memory_region isrs_region $slave $split_offset $split_span
add_section_mapping .isrs isrs_region
```

The above Tcl script splits off 1 KB of RAM from the region named `on_chip_mem`, gives it the name `isrs_region`, and then calls `add_section_mapping` to add the `.isrs` section to `isrs_region`.

Using the Create a New Region and Section Tcl Scrip

To use the Tcl script, execute the following steps:

1. Create the Tcl script as shown in the example, above.
2. Add the following argument to the **niosv-bsp** command line:

   `--script=<script name>.tcl`
3. Rebuild the application by running `cmake` and `make`.

Excerpts from Object Dump Files

After `cmake` and make complete successfully, you can examine the object dump file, *<project_name>***.objdump**. The object dump file shows that the new .isrs section is in the `on_chip_mem`. This object dump file excerpt shows a hardware design with an on-chip memory whose address range is 0x00010000 to 0x00019fff.

The following are the customized linker sections:

- Default linker sections (Span from 0x00010000 to 0x000192bf)
- `on_chip_mem` remaining `on_chip_mem` (Span from 0x000192c0 to 0x00019bff)
- `isrs` customized linker region (Span from 0x00019c00 to 0x00019fff)

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  .
  .
  2 .isrs         00000000  00019c00  00019c00  00008770  2**0
                  CONTENTS
  .
  .
  8 .on_chip_mem  00000000  000192c0  000192c0  00008770  2**0
                  CONTENTS
  .
  .

SYMBOL TABLE:
00010000 l    d  .entry    00000000 .entry
00010020 l    d  .exceptions    00000000 .exceptions
00019c00 l    d  .isrs    00000000 .isrs
000102a0 l    d  .text    00000000 .text
00015cc0 l    d  .rodata    00000000 .rodata
00015cfc l    d  .rwdata    00000000 .rwdata
0001776c l    d  .init_array.00000    00000000 .init_array.00000
000191dc l    d  .bss    00000000 .bss
000192c0 l    d  .on_chip_mem    00000000 .on_chip_mem
```

Excerpt from Linker.x

If you examine the linker script file, `linker.x`, you can see that `linker.x` places the new region `isrs_region` in `MEMORY`, adjacent to the `on_chip_mem` region.

Example 6. Excerpt From linker.x

```
MEMORY
{
    reset : ORIGIN = 0x10000, LENGTH = 32
    on_chip_mem : ORIGIN = 0x10020, LENGTH = 39904
    isrs_region : ORIGIN = 0x19c00, LENGTH = 1024
}
```

5.2.3.4. Changing the Linker Section Mapping

If 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 `ram_0`, use the following command:

When creating a new BSP:

```
niosv-bsp –-create –-qsys=<Platform Designer system> --type=hal
--cmd=”add_section_mapping <.stack or .heap> ram_0” settings.bsp
```

When updating an existing BSP:

```
niosv-bsp –-update --cmd=”add_section_mapping <.stack or .heap> ram0”
settings.bsp
```

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

If you 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 `niosv-bsp --update` with the `add_section_mapping` command line options.

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

5.2.4. Other BSP Tasks

5.2.4.1. Querying Settings

If you need to write a script that gets some information from the BSP settings file, use the `niosv-bsp --query` utility. To maintain compatibility with future releases of the Quartus Prime software, avoid developing your own code to parse the BSP settings file.

```
niosv-bsp --query --export-as-tcl=<BSP Tcl script> settings.bsp
```

This command exports the values of the BSP settings to a BSP Tcl script. By default, the output of `niosv-bsp --query` is an ordered list of all settings values.

For more information about the `niosv-bsp` command line options, refer to the Getting Started from the Command Line.

5.2.4.2. Managing Device Drivers

The BSP Editor creates an `alt_sys_init.c` file. By default, the BSP Editor assumes that if a device is connected to the Nios V 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 Editor 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 `niosv-bsp`, you can disable the driver for the uart0 device as follows:

When creating a new BSP:

```
niosv-bsp --create --qsys=<Platform Designer system> --type=hal
--cmd=”set_driver none uart0” settings.bsp
```

When updating an existing BSP:

```
niosv-bsp --update --cmd=”set_driver none uart0” settings.bsp
```

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

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

When creating a new BSP:

```
niosv-bsp --create --qsys=<Platform Designer system> --type=hal
--cmd=”set_driver altera_avalon_uart:6.1 uart0” settings.bsp
```

When updating an existing BSP:

```
niosv-bsp --update --cmd=”set_driver altera_avalon_uart:6.1 uart0” settings.bsp
```

If you update your BSP, you must provide the original command line options again to prevent the default Tcl script from choosing its own default device drivers.

5.2.4.3. Controlling the stdio Device

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

To override the `stderr` device and replace it with `uart2`:

- When creating a new BSP:

  ```
  niosv-bsp --create --qsys=<Platform Designer system> --type=hal
  --cmd=”set_setting hal.stderr {uart2}” settings.bsp
  ```

- When updating an existing BSP:

  ```
  niosv-bsp --update --cmd=”set_setting hal.stderr {uart2}” settings.bsp
  ```

If you 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.

5.2.4.4. Configuring Optimization and Debugger Options

You can specify the optimization and debug level for the application and BSP projects, and any user library projects you might be using. Use the BSP Editor settings `hal.make.cflags_debug` and `hal.make.cflags_optimization` to specify the optimization and debug level for the whole project.

Configuring the `hal.make.cflags_debug` and `hal.make.cflags_optimization` sets the optimization and debug levels to be the same for the application, the BSP, and all user libraries in a software project. This is to prevent mix settings that disable debuggability to those components which do not have debug settings. For example, if you compile your BSP only with the `-O0` flag and without the `-g` flag, you cannot step into the newlib `printf()` function.

- To configure the whole software project for debugging:

  When creating a new BSP:

  ```
  niosv-bsp --create --qsys=<Platform Designer system> --type=hal
  --cmd=”set_setting hal.make.cflags_debug {-g}” --cmd=”set_setting
  hal.make.cflags_optimization {-O0}” settings.bsp
  ```

  When updating an existing BSP:

  ```
  niosv-bsp --update --cmd=”set_setting hal.make.cflags_debug {-g}”
  --cmd=”set_setting hal.make.cflags_optimization {-O0}” settings.bsp
  ```

  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 BSP Editor settings as shown in the following example. The absence of a value in `hal.make.cflags_debug` causes the compiler to omit generating a symbol table.

- To configure the whole software project for release:

  When creating a new BSP:

  ```
  niosv-bsp --create --qsys=<Platform Designer system> --type=hal
  --cmd=”set_setting hal.make.cflags_debug {none}” --cmd=”set_setting
  hal.make.cflags_optimization {-O3}” settings.bsp
  ```

  When updating an existing BSP:

  ```
  niosv-bsp --update --cmd=”set_setting hal.make.cflags_debug {none}”
  --cmd=”set_setting hal.make.cflags_optimization {-O3}” settings.bsp
  ```

5.2.4.5. Configuring RISC-V Compiler

You can specify the RISC-V compiler using the hal.toolchain.prefix. By default, the `hal.toolchain.prefix` is configured as riscv32-unknown-elf- which is provided by the Ashling RiscFree IDE for Intel FPGAs.

When creating a new BSP:

```
niosv-bsp --create --qsys=<Platform Designer system> --type=hal
--cmd=”set setting hal.toolchain.prefix {riscv32-unknown-elf-}”
```

When updating an existing BSP:

```
niosv-bsp --update --cmd="set setting hal.toolchain.prefix {riscv32-unknown-elf-}"
```

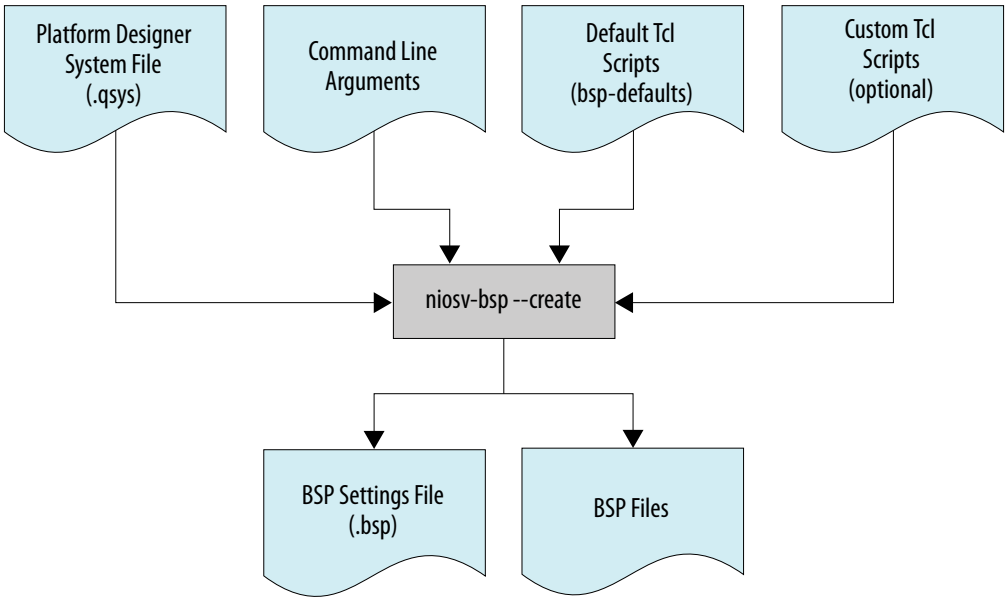
5.3. Details of BSP Creation

The **niosv-bsp --create** utility creates a new BSP settings file and generates the BSP files. It places all source files in your BSP directory. It copies some files from the Quartus Prime installation directory. Others, such as `system.h`, `toolchain.cmake` and `CMakeLists.txt`, it generates dynamically.

The BSP Editor manages copied files slightly differently from generated files. If a copied file (such as a HAL source file) already exists, the tools check the file timestamp against the timestamp of the file in the Quartus Prime installation directory. The tools do not replace the BSP file unless it differs from the distribution file. The tools normally overwrite generated files, such as the `system.h`, `toolchain.cmake` and `CMakeLists.txt`, and `linker.x`, unless you have disabled generation of the individual file with the **BSP Enable File Generation** tab in the BSP Editor.

Note: Avoid modifying BSP files. Use BSP settings, or custom device drivers or software packages from the BSP Editor, to customize your BSP. Manual modification (without using the BSP Editor) to the BSP files are destroyed upon BSP regeneration.

Figure 9. BSP Creation using niosv-bsp --create



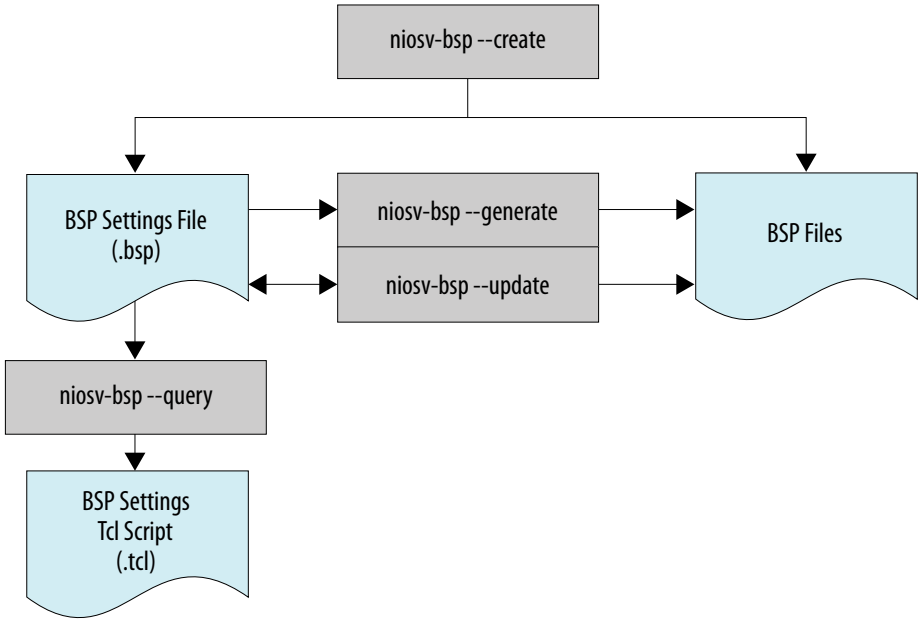
5.3.1. BSP Settings File Creation

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 Editor writes the value of all settings to the settings file.

The BSP settings file does not need to duplicate system information (such as base addresses of devices), because the **niosv-bsp** utility has access to the .**qsys** file.

The **niosv-bsp --create** utility creates a new BSP settings file and generate the BSP files. The **niosv-bsp --update** utility updates an existing BSP settings file and regenerate the BSP files. The **niosv-bsp --query** utility reports the setting values in an existing BSP settings file. The **niosv-bsp --generate** utility generates a BSP from an existing BSP settings file.

Figure 10. Interaction between the niosv-bsp utility



5.3.2. BSP Files and Folders

The BSP Editor creates the BSP directory in the location you specify using `--bsp-dir=<BSP directory>`. If omitted, the BSP Editor creates the BSP directory in the parent directory of BSP settings file by default.

5.3.2.1. Generated and Copied Files

To understand how to build and modify Nios V processor C/C++ projects, understand the difference between copied and generated files.

A copied file is installed with the Quartus Prime software and copied to your BSP directory when you create your BSP. The copied file does not replace the BSP file unless it differs from the distribution file.

The niosv-bsp utility dynamically creates a generated file. Generated files reside in the top-level BSP directory. BSP files are normally written every time **niosv-bsp --create**, **niosv-bsp --generate** or **niosv-bsp --update** runs.

You can disable the generation of any BSP file in the BSP Editor. Otherwise, if you modify a BSP file, the file is deleted when you regenerate the BSP.

5.3.2.2. BSP Files

The BSP Editor places generated files in the top-level BSP directory, and copied files in the HAL and `drivers` directories.

- If you select the MicroC/OS-II operating system, the BSP Editor creates an additional `UCOSII` directory.
- If you select the FreeRTOS operating system, the BSP Editor creates additional `FREERTOS` and `FreeRTOS-Kernel` directories.

Table 18. BSP Files After Generating Files

File Name	File Type	Function
`drivers Directory`	Copied File	• Contains all driver source code. • The `src` directory contains the C-language and assembly-language source files. • The `inc` directory contains the header files.
`HAL Directory`	Copied File	• Contains HAL source code files. • The `HAL directory` has `src` and `inc` subdirectories like the `drivers` directory. • The `crt0.S` source file, containing HAL C run-time start-up code, resides in the `HAL/src` directory.
`UCOSII Directory`	Copied File	• Contains MicroC/OS-II source code files. • Only available when you select MicroC/OS-II operating system. • The `UCOSII` directory has `src` and `inc` subdirectories such as the `drivers` directory.
`FREERTOS Directory`	Copied File	• Contains FreeRTOS OS-HAL contribution-related source code files. • Only available when you select FreeRTOS operating system. • The `FREERTOS directory` has `src` and `inc` subdirectories such as the drivers directory.
`FreeRTOS-Kernel Directory`	Copied File	• Contains FreeRTOS OS services-related and RISC-V port support source code files. • Only available when you select FreeRTOS operating system.
`alt_sys_init.c`	Generated File	Used to initialize device driver instances and software packages.
`CMakeLists.txt`	Generated File	Used to build the BSP by listing the files and directories targeted by the build and other build settings.
`linker.h`	Generated File	Contains information about the linker memory layout. `system.h` includes the `linker.h` file.
`linker.x`	Generated File	Contains a linker script for the GNU linker.
`memory.gdb`	Generated File	Contains memory region declarations for the GNU debugger.
`settings.bsp`	Generated File	Contains all BSP settings. This file is coded in XML.

continued...

File Name	File Type	Function
summary.html	Generated File	• Provides summary documentation of the BSP. • You can view summary.html with a hypertext viewer or browser. • If you change the settings.bsp file, the BSP Editor updates the summary.html file the next time you regenerate the BSP.
system.h	Generated File	Contains the C declarations describing the BSP memory map and other system information needed by software applications.
toolchain.cmake	Generated File	Lists the compiler configurations for the build.

5.3.3. Linker Map Validation

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

- The .entry section maps to a non-existent region
- The .entry section maps to a memory region that is less than 32 bytes in length.
- The .entry section maps to a memory region that does not start on the reset vector base address.
- The .exceptions section maps to a non-existent region.
- The .entry section and .exceptions section map to the same device, and the memory region associated with the .exceptions section precedes the memory region associated with the .entry section.
- The .entry section and .exceptions section map to the same device, and the base address of the memory region associated with the .exceptions section is less than 32 bytes above the base address of the memory region associated with the .entry section.

5.3.4. Hierarchical Platform Designer Systems

Platform Designer supports hierarchical systems that include one or more Platform Designer subsystems within another Platform Designer system. Platform Designer allows you to create, explore, and edit systems and subsystems together in the same Platform Designer window.

The BSP Editor supports BSP generation on hierarchical Platform Designer systems. The supported hierarchical components are:

- Subsystems
- Packaged subsystems
- Composed components

Related Information

- Platform Designer - Create a Composed Component or Subsystem
- Platform Designer - Managing Hierarchical Platform Designer Systems
- Platform Designer - Sharing Platform Designer Packaged Subsystems

5.3.4.1. Handling Hierarchical Systems

You need to perform the following setting when handling a Nios V processor hierarchical system.

Table 19. Handling Hierarchical Systems

Hierarchical System	Description	Action
Reset memory in lower-level subsystems. (Connected through a bridge to the Nios V processor in a top or higher-level subsystem)	Reset memory is not visible to the Nios V processor in higher-level subsystem, prompting errors during linker map validation. *Note:* The reset memory affects packaged subsystems and composed components only.	• Apply **Absolute** as **Reset Agent**. • Specify the exact offset to the reset memory as **Reset Offset** (from the **System View** of the Nios V processor).
Reset memory in top or higher-level subsystem. (Connected through a bridge to the Nios V processor in lower-level subsystem)	• Reset memory is not visible to the Nios V processor, prompting errors during linker map validation. • Nios V processor does not support this hierarchical system. *Note:* The reset memory affects packaged subsystem and composed component only.	Implement reset memory in the following subsystems: • Lower-level subsystems • Within the same subsystems as the processor

Related Information

- Linker Map Validation on page 67
- BSP Drivers Tab, Quartus® Prime Pro Edition User Guide: Platform Designer
- Bridges, Quartus® Prime Pro Edition User Guide: Platform Designer

5.4. Tcl Scripts for BSP Settings

You can fully specify your Nios V processor BSP project with the BSP Editor settings and defaults in many cases. However, you might need to create some simple Tcl scripts to customize your BSP in some cases.

You control the characteristics of your BSP by manipulating BSP settings, using Tcl commands, or combining them into Tcl scripts.

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

5.4.1. Calling a Custom BSP Tcl Script

From the Nios V Command Shell, you can call a custom BSP Tcl script with the **niosv-bsp** utilities using the option `--script=<custom BSP Tcl script>`. In the BSP editor, you can execute a Tcl script when generating a BSP, through the **Create New BSP Settings File** dialog box.

An example of Tcl script to set `stdio` to `my_uart`:

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

Any settings you specify in your script override the BSP default values. When you update an existing BSP, you must include any scripts originally used to create it. Otherwise, your project's settings revert to the defaults.

When you use a custom Tcl script to create your BSP, you must include the script in the set of files archived in your version control system.

5.4.1.1. Tcl Script to Examine Hardware and Choose Settings

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

If a component has only one slave port connected to the Nios V processor, the slave descriptor is the same as the component's name (for example, onchip_mem_0).

Suppose a component has multiple slave ports connecting the Nios V processor to multiple resources in the component. In that case, the slave descriptor is the name of the component followed by an underscore and the slave port name (for example, onchip_mem_0_s1).

```
# Select a device connected to the processor 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 processor.
set slave_descs [get_slave_descs]
foreach slave_desc $slave_descs {

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

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

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

# Track last character device in case no JTAG UARTs found.
if { [is_char_device $slave_desc] } {
set last_stdio $slave_desc
}}
if {$first_jtag_uart != "none"} {
return $first_jtag_uart
}
return $last_stdio
}

# Call routine to determine stdio
set default_stdio [choose_default_stdio]

# Set stdio settings to use results of above call.
```

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

5.5. Revising Your BSP

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

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

You can revise a BSP at several levels. This section describes each level and guides when to use it.

Related Information

- The Nios V Processor Embedded Project Structure on page 85
- Nios V Processor Software Development and Implementation on page 18

5.5.1. Regenerating Your BSP

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

You can regenerate your BSP by using the **niosv-bsp --generate** command.

5.5.1.1. What Happens

Regenerating a BSP has the following effects:

- Reads the .**qsys** file for basic system parameters such as module base addresses and clock frequencies.
- Adjusts the default memory map to correspond to changes in memory sizes. If you are using a custom memory map, it is untouched.
- Retains all other existing settings in the BSP settings file.

  Note: Except for adjusting the default memory map, the BSP Editor does not ensure that the BSP settings are consistent with the hardware design in the .**qsys** file.

- Ensures that the correct set of BSP files is present, as follows:
  - Copy all required source files to the BSP directory tree.

    If a copied file (such as a HAL source file) already exists, the BSP Editor checks the file timestamp against the timestamp of the file in the Quartus Prime software installation directory. The tools do not replace the BSP file unless it differs from the distribution file.
  - Recreates all generated files.

5.5.1.2. When to Regenerate Your BSP

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

- You change your hardware design, but all BSP system-dependent settings remain consistent with the new .**qsys** file. The following are examples of system changes that do not affect BSP system-dependent settings:
  - Changing a component's base address
  - With the Internal Interrupt Controller (IIC), adding or removing hardware interrupts
  - With the IIC, changing a hardware interrupt number
  - Changing a clock frequency
  - Changing a simple processor option, such as cache size or core type
  - Changing a simple component option, other than memory size.
  - Adding a bridge
  - Adding a new component
  - Removing or renaming a component, other than a memory component, the stdio device, or the system timer device
  - Changing the size of a memory component when you are using the default memory map
- You want to eliminate any customized source files and revert to the distributed BSP code.

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

5.5.2. Updating Your BSP

When you update a BSP, you recreate the BSP settings file based on the current hardware definition and previous BSP settings. Using the new BSP settings file, the BSP files are regenerated.

You can update your BSP by using the **niosv-bsp --update** command with the --script option to define the BSP with a Tcl script. The Tcl script overrides the default BSP settings.

5.5.2.1. What Happens

Updating a BSP has the following effects:

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

5.5.2.2. When to Update Your BSP

Updating your BSP is necessary in the following circumstances:

- A change to your BSP settings is required.
- Changes to your .**qsys** file make it inconsistent with your BSP. The following are examples of system changes that affect BSP system-dependent settings:
  - Renaming the processor
  - Renaming or removing a memory, the stdio device, or the system timer device
  - Changing the size of a memory component when using a custom memory map
  - Changing the processor reset

5.5.3. Recreating Your BSP

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

You can recreate your BSP by using the **niosv-bsp --create** command with the --script option to define the BSP with a Tcl script. The Tcl script overrides the default BSP settings.

5.5.3.1. What Happens

Recreating a BSP has the following effects:

- System-dependent settings are created based on the current hardware system.
- Non-system-dependent settings can be selected by the default Tcl script, by values you specify, or both

5.5.3.2. When to Recreate Your BSP

If you modify the underlying hardware design, the best practice is to create a new BSP. Altera recommends modifying your BSP with the BSP Editor. Manually modifying a large number of interrelated settings can be difficult.

5.6. Specifying BSP Defaults

The BSP Editor sets BSP defaults using a set of Tcl scripts. These scripts specify default BSP settings. The scripts are located in the following directory:

```
<Intel Quartus Prime software installation directory>/niosv/scripts/bsp-defaults
```

Table 20. Default Tcl Script Components

Script	Level	Summary
`bsp-set-defaults.tcl`	Top-level	Sets system-dependent settings to default values.
`bsp-stdio-utils.tcl`	Helper	Specifies stdio device settings.
`bsp-timer-utils.tcl`	Helper	Specifies system timer device setting.

continued...

Script	Level	Summary
`bsp-linker-utils.tcl`	Helper	Specifies memory regions and section mappings for linker script.
`bsp-bootloader-utils.tcl`	Helper	Specifies bootloader-related settings.
`bsp-exception-utils.tcl`	Helper	Specifies instruction-related exception settings.

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

The BSP Editor executes the default Tcl script before any user-specified Tcl scripts. As a result, user input overrides settings made by the default Tcl script. You can pass command line option (`--script`) to the default Tcl script to override the choices it makes.

The default Tcl scripts use slave descriptors to assign devices

5.6.1. Top Level Tcl Script for BSP Defaults

The top level Tcl script for setting BSP defaults is `bsp-set-defaults.tcl`. This script specifies BSP system-dependent settings, which depend on the hardware system. Both the BSP Editor and **niosv-bsp** utilities call the default Tcl script.

The default Tcl script consists of a top-level Tcl script named `bsp-set-defaults.tcl` plus the helper Tcl scripts listed in the *Default Tcl Script Components* table. The helper Tcl scripts do the real work of examining the processor subsystem 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 bootloader settings (`bsp-bootloader-utils.tcl`)
- Default instruction-related exception settings (`bsp-exception-utils.tcl`)

5.6.1.1. 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.

The script searches the processor subsystem file for a slave descriptor with the string stdio in its name.

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

5.6.1.2. 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. By default, the Nios V processor internal timer is selected as `hal.sys_clk_timer`.

5.6.1.3. 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.

The script performs the following functions:

- 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.
- Assigns the `.text`, `.rodata`, `.rwdata`, `.bss`, `.heap`, and `.stack` section mappings to this default memory region.

  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).

5.6.1.4. Specifying the Default Bootloader Parameters

The `bsp-bootloader-utils.tcl` script provides procedures to specify the following BSP Boolean settings in hal.linker:

- `allow_code_at_reset`
- `enable_alt_load`
- `enable_alt_load_copy_rodata`
- `enable_alt_load_copy_rwdata`
- `enable_alt_load_copy_exceptions`

The script examines the `.text` section mapping and the Nios V processor reset agent port. If the `.text` section is mapped to the same memory as the Nios V processor reset agent port and the reset agent port is a flash memory device, the script assumes that the `alt_load()` function is being used.

The `alt_load()` function is an early part of the software initialization and handles the copying of sections on reset except the `.text` section. If it is necessary to copy the `.text` section to other memory region, the BSP needs to provide a custom bootloader code.

Table 21. Bootloader-Dependent Settings

Setting name	Value When Bootloader Is Used	Value When Bootloader Is Not Used
`allow_code_at_reset`	0	1
`enable_alt_load`	0	1
`enable_alt_load_copy_rodata`	0	1 if `.rodata` memory different than .text memory and `.rodata` memory is volatile; 0 otherwise
`enable_alt_load_copy_rwdata`	0	1
`enable_alt_load_copy_exceptions`	0	1 if `.exceptions` memory different than `.text` memory and `.exceptions` memory is volatile; 0 otherwise

5.6.1.5. Specifying the Default Exception Parameter

The `bsp-exception-utils.tcl` script provides procedures to configure the BSP setting, `hal.enable_instruction_related_exceptions_api` when the Nios V processor is handling instruction-related exceptions.

The script searches the processor subsystem to verify if the Nios V processor enabled the following features:

- Memory Protection Unit (MPU)
- Memory Management Unit (MMU)

The script enables the API for registering handlers to service the instruction-related exceptions.

5.7. Device Drivers and Software Packages

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

5.8. Boot Configurations for Altera FPGA Embedded Software

The HAL and RTOS BSPs support several boot configurations. The default Tcl script configures an appropriate boot configuration based on your hardware system and other settings.

5.8.1. Memory Types

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 `.qsys` file) indicates whether the `.qsys` file identifies the memory as a flash memory device.

The `IsNonVolatileStorage` property indicates whether the `.qsys` file identifies the memory as a non-volatile storage device. The contents of a non-volatile memory device are fixed and always present.

Note: You can initialize some FPGA memories when you configure the FPGA. 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 22. Memory Types Recognized when Making Decisions about Your Boot Configuration

Memory Type	Examples	IsFlash	IsNonVolatileStorage
Flash	Quad Serial Peripheral Interface (QSPI) flash devices	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.

5.8.2. Boot from Flash Configuration

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

This boot configuration has the following characteristics:

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

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

Altera provides example bootloaders for QSPI flash memory in the `<Intel Quartus Prime software installation directory>/niosv/components/bootloader` folder, precompiled to Motorola S-record Files (`.srec`). You can use the Altera-provided bootloader or provide your own.

Table 23. Altera-provided Bootloader

Altera-provided Bootloader	Description	Target Processor Core	File Location
GSFI bootloader	Nios V processor boot copier that supports QSPI flash memory connected through Generic Serial Flash Interface Intel FPGA IP.	Nios V/m processor	`<Intel Quartus Installation Directory>/niosv/components/bootloader/niosv_m_bootloader.srec`
		Nios V/g processor	`<Intel Quartus Installation Directory>/niosv/components/bootloader/niosv_g_bootloader.srec`

5.8.3. Run from Initialized Memory Configuration

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

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

This boot configuration has the following characteristics:

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

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

In this boot configuration, when the processor core resets, by default the `.rwdata` section is not reinitialized. Reinitialization would normally be done by a bootloader. However, this configuration has no bootloader because the software is running out of memory that is assumed to be preinitialized before start-up. If your software has a `.rwdata` section that must be reinitialized at processor reset, turn on the `hal.linker.enable_alt_load_copy_rwdata` setting in the BSP.

5.8.4. Run-time Configurable Reset Configuration

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

This boot configuration has the following characteristics:

- `alt_load()` might be called (depends on boot configuration)
- No code at reset in executable file

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

6. Overview of the Hardware Abstraction Layer

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

The HAL serves as a device driver package for Nios V processor systems, providing a consistent interface to the peripherals in your system. The Nios V processor software development tools extract system information from the Platform Designer System File (`.qsys`). The Board Support Package (BSP) Editor generates a custom HAL BSP specific to your hardware configuration. Changes in the hardware configuration automatically propagate to the HAL device driver configuration, eliminating bugs that emerge as a result of driver or IP configuration mismatch.

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

Related Information

- Driver Integration in the HAL API on page 127
- Developing with the Hardware Abstraction Layer on page 41
- HAL Peripheral Services on page 45
  For information about configuring and using HAL services.
- Reducing Code Footprint in Embedded Systems on page 108
  For information about increasing your application's performance, or minimizing the code footprint.
- Creating Embedded Drivers for HAL Device Classes on page 131

6.1. Getting Started with the Hardware Abstraction Layer

The easiest way to start using the HAL is to create a software project. In creating a new project, you also create a HAL BSP. You do not need to create or copy HAL files, or edit any HAL source code. The BSP Editor generates the HAL BSP for you.

In the Nios V processor software tools command line, you can create an example BSP based on the HAL using the **niosv-bsp** utility.

You must base the HAL on a specific Platform Designer system. A Nios V processor system consists of a Nios V processor core integrated with peripherals and memory. Platform Designer generates Nios V processor systems.

© Altera Corporation. Altera, the Altera logo, the 'a' logo, and other Altera marks are trademarks of Altera Corporation. Altera and Intel warrant performance of its FPGA and semiconductor products to current specifications in accordance with Altera's or Intel's standard warranty as applicable, but reserves the right to make changes to any products and services at any time without notice. Altera and Intel assume no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to inwriting by Altera or Intel. Altera and Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

You can use an Intel provided example project for Nios V processor to develop your Nios V processor system. You can re-target it to a different board later, or develop your own custom Nios V processor.

Related Information

- Getting Started from the Command Line on page 13
  For more information about creating a new project with the command line.
- Intel FPGA Design Store on page 378
  For more information about how to base your project on an Altera-provided example hardware system.

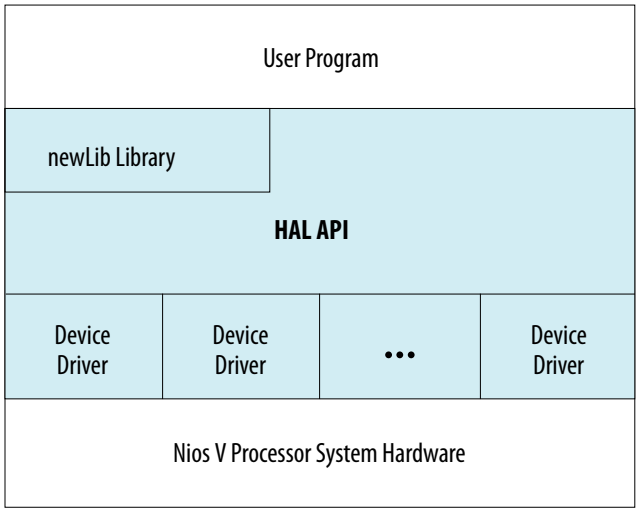
6.2. HAL Architecture for Embedded Software Systems

6.2.1. Services

The HAL provides the following services:

- Integration with the newlib ANSI C standard library—Provides the familiar C standard library functions
- Device drivers—Provides access to each device in the system
- The HAL API—Provides a consistent, standard interface to HAL services, such as device access, interrupt handling, and alarm facilities
- System initialization—Performs initialization tasks for the processor and the runtime environment before `main()`
- Device initialization—Instantiates and initializes each device in the system before `main()` runs

Figure 11. The Layers of a HAL-Based System



6.2.2. Applications versus Drivers

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

Related Information

- Developing Programs Using the Hardware Abstraction Layer on page 84
- Developing Device Drivers for the Hardware Abstraction Layer on page 127
  For more information about the Hardware Abstraction Layer.

6.2.3. Generic Device Models

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

6.2.3.1. Device Model Classes

The HAL provides models for the following classes of devices:

- Character-mode devices—Hardware peripherals that either or both send or receive characters serially, such as a UART.
- Timer devices—Hardware peripherals that count clock ticks and can generate periodic interrupt requests.
- File subsystems—A mechanism for accessing files stored in physical device(s).
  - Depending on the internal implementation, the file subsystem driver might access the underlying device(s) directly or use a separate device driver. For example, you can write a flash file subsystem driver that accesses flash using the HAL API for flash memory devices.
- Ethernet devices—Devices that provide access to an Ethernet connection for a networking stack. You need a networking stack to use an ethernet device.
- Direct memory access (DMA) devices—Peripherals that perform bulk data transactions from a data source to a destination. Sources and destinations can be memory or another device, such as an Ethernet connection.
- Flash memory devices—Non-volatile memory devices that use a special programming protocol to store data.

6.2.3.2. Benefits to Application Developers

The HAL defines a set of functions that you use to initialize and access each class of device. The API is consistent, regardless of the underlying implementation of the device hardware.

For example, to access character-mode devices and file subsystems, you can use the C standard library functions, such as `printf()` and `fopen()`. For application developers, you need not write low-level routines just to establish basic communication with the hardware for these classes of peripherals.

6.2.3.3. Benefits to Device Driver Developers

Each device model defines a set of driver functions necessary to manipulate the particular class of device. If you are writing drivers for a new peripheral, you need only provide this set of driver functions. As a result, your driver development task is predefined and well documented.

In addition, you can use existing HAL functions and applications to access the device, which saves software development effort. The HAL calls driver functions to access hardware. Application programmers call the ANSI C or HAL API to access hardware, rather than calling your driver routines directly.

Therefore, the usage of your driver is already documented as part of the HAL API.

6.2.4. C Standard Library—Newlib, Picolibc

The HAL integrates the ANSI C standard library in its runtime environment. The HAL uses Newlib by default as an open-source implementation of the C standard library. Newlib is a C library for use on embedded systems, making it a perfect match for the HAL and the Nios V processor. Alternatively, Picolibc, a smaller and slower C library option can be used in place of Newlib for code-size critical designs. Newlib and Picolibc licensing does not require you to release your source code or pay royalties for projects based on Newlib or Picolibc.

Related Information

Redhat Website
Online documentation for newlib by Red Hat

6.3. Embedded Hardware Supported by the HAL

Intel provides many peripherals for use in Nios V processor systems. Most Intel FPGA peripherals provide HAL device drivers that allow Nios V processor to access the hardware via the HAL API calls.

6.3.1. Nios V Processor Core Support

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

6.3.2. Supported Peripherals

6.3.2.1. Full HAL Support

The following Intel FPGA peripherals provide full HAL support:

- Character mode devices
  - UART core
  - JTAG UART core
- Flash memory devices
  - Quad Serial Peripheral Interface (QSPI) flash devices
- Timer devices
  - Timer core
- DMA devices
  - DMA controller core
  - Modular Scatter-gather DMA controller core

Note: This list does not include peripherals supported by third-party vendors that is compatible with Nios V processor.

6.3.2.2. Partial HAL Support

As all peripherals vendors provide a header file that defines the peripheral's low level hardware interface. You can use the definitions provided in the header to access the hardware inside HAL.

Inevitably, some peripherals have features that does not map well to our general-purpose HAL API. For these device, HAL API provides the UNIX-style `ioctl()` function to access them.

For example, Intel provides a general-purpose parallel I/O core that can be use with Nios V processor. However, this device does not fit into any generic device model class provided by HAL. As such, it provides a header file and a few dedicated accessors functions. `ioctl()` can be used to integrate its functionality into Nios V processor via HAL.

For complete details regarding software support for a peripheral, refer to the peripheral's documentation.

Related Information

Embedded Peripherals IP User Guide

7. Developing Programs Using the Hardware Abstraction Layer

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

Related Information

Applications versus Drivers on page 81

7.1. HAL BSP Settings

Every Nios V processor project has a Board Support Package (BSP) that defines the hardware or IP available in the project. HAL relies on the settings and hardware mapping in BSP. For example, HAL relies on BSP to define the hardware component associated with standard devices such as stdout.

The BSP produces the `system.h` file, which contains the BSP settings for your project and other HAL configuration settings you need to develop your software.

Note: Do not edit `system.h`. Use the Nios V processor tools to manipulate the system settings.

For details about how to control BSP settings, refer the following documents in Related Information.

Related Information

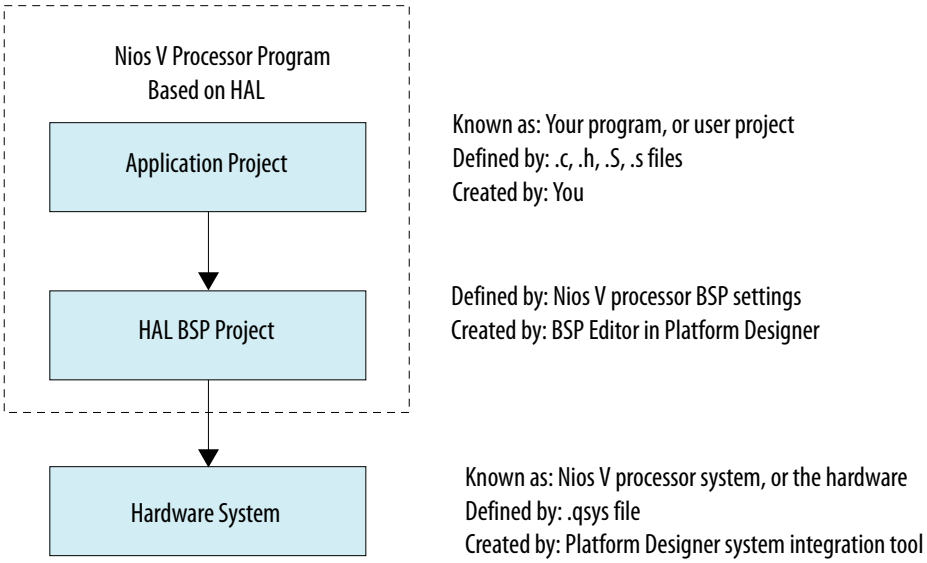
- Nios V Processor Board Support Package Editor on page 53
- Settings Managed by Nios V Processor Board Support Package Editor on page 296
- System Clock Driver on page 96
- Placement of the Heap and Stack on page 123

© Altera Corporation. Altera, the Altera logo, the 'a' logo, and other Altera marks are trademarks of Altera Corporation. Altera and Intel warrant performance of its FPGA and semiconductor products to current specifications in accordance with Altera's or Intel's standard warranty as applicable, but reserves the right to make changes to any products and services at any time without notice. Altera and Intel assume no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to inwriting by Altera or Intel. Altera and Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

7.2. The Nios V Processor Embedded Project Structure

Figure 12. The Nios V HAL Project Structure Emphasizing How the HAL BSP Fits In



Every HAL-based Nios V processor program consists of two parts of the project. Your application-specific code is contained in one project (the user application project), and it depends on a separate BSP project (the HAL BSP).

Related Information

- Revising Your BSP on page 70
  For more information about how the Nios® V processor tools keep your BSP up-to-date with your hardware system.
- Nios V Processor Software Development and Implementation on page 18

7.3. The system.h System Description File

The `system.h` file provides a complete software description of the Nios V system hardware. Not all information in `system.h` is useful to you as a programmer, and it is rarely necessary to include it explicitly in your C source files. Nonetheless, `system.h` holds the answer to the question, "What hardware is present in this system?"

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

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

The Nios V processor tools generates the `system.h` file for HAL BSP projects. The contents of `system.h` depend on both the hardware configuration and the HAL BSP properties.

Note: Do not edit `system.h`. Use Nios V processor tools facilities to manipulate system settings.

Example 7. Excerpts from a system.h File Detailing Hardware Configuration Options

```
/*
* sys_clk_timer configuration
*
*/
#define SYS_CLK_TIMER_NAME "/dev/sys_clk_timer"
#define SYS_CLK_TIMER_TYPE "altera_avalon_timer"
#define SYS_CLK_TIMER_BASE 0x00920800
#define SYS_CLK_TIMER_IRQ 0
#define SYS_CLK_TIMER_ALWAYS_RUN 0
#define SYS_CLK_TIMER_FIXED_PERIOD 0
/*
* jtag_uart configuration
*
*/
#define JTAG_UART_NAME "/dev/jtag_uart"
#define JTAG_UART_TYPE "altera_avalon_jtag_uart"
#define JTAG_UART_BASE 0x00920820
#define JTAG_UART_IRQ 1
```

7.4. Data Widths and the HAL Type Definitions

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

The header file `alt_types.h` defines the HAL type definitions.

Table 24. The HAL Type Definitions

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

Table 25. GNU Toolchain Data Widths

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

7.5. UNIX-Style Interface

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

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

- `_exit()`
- `close()`
- `fstat()`
- `getpid()`
- `gettimeofday()`
- `ioctl()`
- `isatty()`
- `kill()`
- `lseek()`
- `open()`
- `read()`
- `sbrk()`
- `settimeofday()`
- `stat()`
- `usleep()`
- `wait()`
- `write()`

The most commonly used functions are those that relate to file I/O.

Related Information

HAL API Reference on page 224
For more information about the API functions.

7.6. File System

The HAL provides infrastructure for UNIX-style file access. You can use this infrastructure to build a file system on any storage devices available in your hardware.For more information about the infrastructure, refer to an example in the Read-Only Zip File System chapter.

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

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

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

For more information about the UNIX-style functions, refer to the HAL API Reference chapter.

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

You must aways access files using absolute paths; there is no concept of a current directory.

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

Note: The standard header files `stdio.h`, `stddef.h`, and `stdlib.h` are installed with the HAL.

```
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#define BUF_SIZE (10)
int main(void)
{
   FILE* fp;
   char buffer[BUF_SIZE];
   fp = fopen ("/mount/rozipfs/test", "r"); if (fp == NULL)
```

```
   {
      printf ("Cannot open file.\n");
      exit (1);
   }
   fread (buffer, BUF_SIZE, 1, fp);
   fclose (fp);
   return 0;
}
```

7.7. Using Character-Mode Devices

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

7.7.1. Standard Input, Standard Output and Standard Error

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

Note: This program sends characters to whatever device is associated with `stdout` when the program is compiled.

Example 8. Hello World

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

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

7.7.2. General Access to Character Mode Devices

Accessing a character-mode device other than `stdin`, `stdout`, or `stderr` is as easy as opening and writing to a file.

Example 9. Writing Characters to a UART Called `uart1`

```
#include <stdio.h>
#include <string.h>
int main (void)
{
```

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

7.7.3. C++ Streams

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

7.7.4. /dev/null

All systems include the device **/dev/null**. Writing to **/dev/null** has no effect, and all data is discarded. **/dev/null** is used for safe I/O redirection during system startup and useful for applications that wish to sink unwanted data. The device is purely a software construct. It does not relate to any physical hardware device in the system.

7.7.5. Lightweight Character-Mode I/O

The HAL offers several methods of reducing the code footprint of character-mode device drivers.

Related Information

Reducing Code Footprint in Embedded Systems on page 108

7.7.6. Intel FPGA Logging Functions

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

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

Note: Avoid setting the Intel FPGA logging device to the device used for `stdout` or `stderr`. If Intel FPGA logging output is sent to `stdout` or `stderr`, the logging output might appear interleaved with the `stdout` or `stderr` output

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

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

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

Note: The Intel FPGA reduced device drivers do not support Intel FPGA logging.

7.7.6.1. Enabling Logging

Logging is enabled by setting **hal.log_port** to a JTAG UART or a UART device. You can find the **hal.log_port** in the BSP Editor. The setting allows the HAL to send log messages to the specified device.

Once **hal.log_port** is set, ALT_LOG_ENABLE is defined in `toolchain.cmake` and the ALT_LOG_FLAGS flag is set to 0, the default value. If **hal.log_port** is not set, then ALT_LOG_ENABLE and ALT_LOG_FLAGS do not appear in `toolchain.cmake`.

The build tools also set the ALT_LOG_PORT_TYPE and ALT_LOG_PORT_BASE values in `system.h` to point to the specified device.

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

Example 10. Default Boot Logging Output

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

Note: A write operation to the logging device stalls in ALT_LOG_PRINTF( ) until the characters are read from the logging device's output buffer. To ensure that the Nios V processor application completes initialization, run the **juart-terminal** command from the Nios V Command Shell to accept the logging output.

7.7.6.2. Extra Logging Options

In addition to the default boot messages, logging options are incorporated in Intel FPGA logging. Each option is controlled by a C preprocessor symbol.

Table 26. Intel FPGA Logging Options and Option Modifiers

Name	Description	
System clock log	Purpose	Prints out a message from the system clock interrupt handler at a specified interval. This indicates that the system is still running. The default interval is every 1 second.
	Preprocessor symbol	ALT_LOG_SYS_CLK_ON_FLAG_SETTING
	Modifiers	The system clock log has two modifiers, providing two different ways to specify the logging interval.

continued...

Name	Description	
		• ALT_LOG_SYS_CLK_INTERVAL—Specifies the logging interval in system clock ticks. The default is <clock ticks per second>, that is, one second. • ALT_LOG_SYS_CLK_INTERVAL_MULTIPLIER—Specifies the logging interval in seconds. The default is 1. When you modify ALT_LOG_SYS_CLK_INTERVAL_MULTIPLIER, ALT_LOG_SYS_CLK_INTERVAL is recalculated.
	Sample Output	`System Clock On 0` `System Clock On 1`
Write echo	Purpose	Every time `alt_write()` is called (normally, whenever characters are sent to `stdout`), the first <n> characters are echoed to a logging message. The message starts with the string `"Write Echo:"`. <n> is specified with ALT_LOG_WRITE_ECHO_LEN. The default is 15 characters.
	Preprocessor symbol	ALT_LOG_WRITE_ON_FLAG_SETTING
	Modifiers	ALT_LOG_WRITE_ECHO_LEN—Number of characters to echo. Default is 15.
	Sample Output	`Write Echo: Hello from Nios V`
JTAG startup log	Purpose	At JTAG UART driver initialization, print out a line with the number of characters in the software transmit buffer followed by the JTAG UART control register contents. The number of characters, prefaced by the string `"SW CirBuf"`, might be negative, because it is computed as (<tail_pointer> – <head_pointer>) on a circular buffer. For more information about the JTAG UART control register fields, refer to the *Embedded Peripherals IP User Guide*.
	Preprocessor symbol	ALT_LOG_JTAG_UART_STARTUP_INFO_ON_FLAG_SETTING
	Modifiers	None
	Sample Output	`JTAG Startup Info: SW CirBuf = 0, HW FIFO wspace=64 AC=0 WI=0 RI=0 WE=0 RE=1`
JTAG interval log	Purpose	Creates an alarm object to print out the same JTAG UART information as the JTAG startup log, but at a repeated interval. Default interval is 0.1 second, or 10 messages a second.
	Preprocessor symbol	ALT_LOG_JTAG_UART_ALARM_ON_FLAG_SETTING
	Modifiers	The JTAG interval log has two modifiers, providing two different ways to specify the logging interval. • ALT_LOG_JTAG_UART_TICKS—Logging interval in ticks. Default is <ticks_per_second> / 10. • ALT_LOG_JTAG_UART_TICKS_DIVISOR—Specifies the number of logs per second. The default is 10. When you modify ALT_LOG_JTAG_UART_TICKS_DIVISOR, ALT_LOG_JTAG_UART_TICKS is recalculated.
	Sample Output	`JTAG Alarm: SW CirBuf = 0, HW FIFO wspace=45 AC=0 WI=0 RI=0 WE=0 RE=1`
JTAG interrupt service routine (ISR) log	Purpose	Prints out a message every time the JTAG UART near-empty interrupt triggers. Message contains the same JTAG UART information as in the JTAG startup log.
	Preprocessor symbol	ALT_LOG_JTAG_UART_ISR_ON_FLAG_SETTING
	Modifiers	None
	Sample Output	`JTAG IRQ: SW CirBuf = -20, HW FIFO wspace=64 AC=0 WI=1 RI=0 WE=1 RE=1`

continued...

Name	Description	
Boot log	Purpose	Prints out messages tracing the software boot process. The boot log is turned on by default when Intel FPGA logging is enabled.
	Preprocessor symbol	`ALT_LOG_BOOT_ON_FLAG_SETTING`
	Modifiers	None
	Sample Output	For more information, refer to Enabling Logging.

Note: An option's modifiers are meaningful only when the option is enabled.

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

Several options have modifiers, which are additional preprocessor symbols controlling details of how the options work. For example, the system clock log's modifiers control the logging interval.

Related Information

Embedded Peripherals IP User Guide

7.7.6.3. Logging Levels

An additional preprocessor symbol, ALT_LOG_FLAGS, can be set to provide some grouping for the extra logging options. ALT_LOG_FLAGS implements logging levels based on performance impact. With higher logging levels, the Intel FPGA logging options take more processor time.

Table 27. ALT_LOG_FLAGS Logging Levels

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

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

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

7.7.6.4. Example: Creating a BSP with Logging

- System clock log
- JTAG startup log
- JTAG interval log, logging twice a second
- No write echo

Example 11. Creating BSP With Logging and Options

```
niosv-bsp -c -s=<path to .qsys file> -t=<bsp type> settings.bsp
--cmd="set_setting hal.log_port {uart1}"
--cmd="set_setting hal.make.cflags_user_flags {-DALT_LOG_FLAGS=2
-DALT_LOG_WRITE_ON_FLAG_SETTING=0 -DALT_LOG_JTAG_UART_TICKS_DIVISOR=2r}"
```

This `niosv-bsp` command modify `toolchain.cmake` to use uart1 for log output, and to add `-DALT_LOG_FLAGS=2` when calling gcc to compile the program.

7.7.6.5. Custom Logging Messages

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

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

Then use the following macro function:

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

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

If you want your custom logging message to be controlled by Intel FPGA logging preprocessor options, use the appropriate Intel FPGA logging option preprocessor flags from the ALT_LOG_FLAGS Logging Levels table or the Intel FPGA Logging Options and Option Modifiers table.

Example 12. Implementing Logging Options with Custom Logging Messages

```
/* The following example prints "Level 2 logging message" if
logging is set to level 2 or higher */
#if ( ALT_LOG_FLAGS >= 2 )
ALT_LOG_PRINTF ( "Level 2 logging message" );
#endif
/* The following example prints "Boot logging message" if boot logging
is turned on */
#if ( ALT_LOG_BOOT_ON_FLAG_SETTING == 1)
ALT_LOG_PRINTF ( "Boot logging message" );
#endif
```

7.7.6.6. Intel FPGA Logging Files

Table 28. HAL Implementation Files for Intel FPGA Logging

Location	File Name
`soft_processor/altera_hal2/HAL/inc/sys/`	`alt_log_printf.h`
`soft_processor/altera_hal2/HAL/src/`	`alt_log_printf.c`
`soft_processor/intel_niosv_m/HAL/src/`	`alt_log_macro.S`

Note: All file locations are relative to `<Intel Quartus Prime installation directory>/ip/altera folder`.

These files implement the logging options listed in the Intel FPGA Logging Options and Option Modifiers table. They also serve as examples of logging usage.

Table 29. HAL Example Files for Intel FPGA Logging

Location	File Name
`sopc_builder_ip/altera_avalon_jtag_uart/HAL/src/`	`altera_avalon_jtag_uart.c`
`sopc_builder_ip/altera_avalon_timer/HAL/src/`	`altera_avalon_timer_sc.c`
`soft_processor/altera_hal2/HAL/src/`	`alt_exit.c`
`soft_processor/altera_hal2/HAL/src/`	`alt_main.c`
`soft_processor/altera_hal2/HAL/src/`	`alt_write.c`
`soft_processor/intel_niosv_m/HAL/src/`	crt0.S

Note: All file locations are relative to `<Intel Quartus Prime installation directory>/ip/altera` folder.

7.8. Using File System

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

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

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

7.9. Using Timer Devices

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

The HAL API provides two types of timer device drivers:

- System clock driver—Supports alarms, such as you would use in a scheduler.
- Timestamp driver—Supports high-resolution time measurement.

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

The Nios V processor core provides an internal timer which can serve as both an interval timer and timestamp. If an external timer such as the Interval Timer core is used, it can only implement a timestamp or a system clock but not both.

For more information about where the HAL-specific API functions for accessing timer devices are defined, refer to the `sys/alt_alarm.h` and `sys/alt_timestamp.h` files.

7.9.1. System Clock Driver

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

For more information about how to control BSP settings, refer to the HAL BSP Settings chapter.

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

The system clock measures time in clock ticks. If you are working with both hardware and software, do not confuse the HAL system clock with the clock signal driving the Nios V processor hardware. The period of a HAL system clock tick is much longer than the hardware system clock. `system.h` defines the clock tick frequency.

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

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

Related Information

- HAL API Reference on page 224
  For more information about the API functions.
- HAL BSP Settings on page 84

7.9.2. Alarms

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

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

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

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

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

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

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

Example 13. Using a Periodic Alarm Callback Function

```
#include <stddef.h>
#include <stdio.h>
#include "sys/alt_alarm.h"
#include "alt_types.h"
/*
* The callback function.
*/
alt_u32 my_alarm_callback (void* context)
{
/* This function is called once per second */
return alt_ticks_per_second();
}
...
/* The alt_alarm must persist for the duration of the alarm. */
static alt_alarm alarm;
...
if (alt_alarm_start (&alarm,
alt_ticks_per_second(),
my_alarm_callback,
NULL) < 0)
```

```
{
printf ("No system clock available\n");
}
```

Related Information

Trap Handling on page 154
For more information about these functions.

7.9.3. Timestamp Driver

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

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

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

- `alt_timestamp_start()`
- `alt_timestamp()`

Calling `alt_timestamp_start()` starts the counter running. Subsequent calls to `alt_timestamp()` return the current value of the timestamp counter. Calling `alt_timestamp_start()` again resets the counter to zero. The behavior of the timestamp driver is undefined when the counter reaches (2^{32} - 1 or 2^{64} - 1, depending on the timer parametrization used).

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

Example 14. Using the Timestamp to Measure Code Execution Time

```
#include <stdio.h>

#include "sys/alt_timestamp.h"

#include "alt_types.h"

int main(void) {
  alt_u32 time1;
  alt_u32 time2;
  alt_u32 time3;
  if (alt_timestamp_start() < 0) {
    printf("No timestamp device available\n");
  } else {
    time1 = alt_timestamp();
    func1(); /* first function to monitor */
    time2 = alt_timestamp();
    func2(); /* second function to monitor */
    time3 = alt_timestamp();
    printf("time in func1 = %u ticks\n",
      (unsigned int)(time2 - time1));
    printf("time in func2 = %u ticks\n",
      (unsigned int)(time3 - time2));
    printf("Number of ticks per second = %u\n",
      (unsigned int) alt_timestamp_freq());
```

```
  }
  return 0;
}
```

Related Information

HAL API Reference on page 224

7.10. Using Flash Devices

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

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

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

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

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

The supportable flash IP for Nios V processor is the Generic Serial Flash Interface Intel FPGA IP.

Related Information

- Embedded Peripherals IP User Guide
- HAL API Reference on page 224
  For more information about the API functions.
- JEDEC Website
- Generic Serial Flash Interface Intel® FPGA IP User Guide

7.10.1. Simple Flash Access

This interface consists of the functions `alt_flash_open_dev()`, `alt_write_flash()`, `alt_read_flash()`, and `alt_flash_close_dev()`.

You open a flash device by calling `alt_flash_open_dev()`, which returns a file handle to a flash device. This function takes a single argument that is the name of the flash device, as defined in `system.h`.

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

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

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

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

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

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

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

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

Related Information

Fine-Grained Flash Access on page 101
For more information about the use of all of the functions in one code example.

7.10.2. Block Erasure or Corruption

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

The *Example of Writing Flash and Causing Unexpected Data Corruption* table shows the example of an 8-kilobyte (KB) flash memory comprising two 4-KB blocks.

1. First write 5 KB of all `0xAA` to flash memory at address `0x0000`, and then write 2 KB of all `0xBB` to address `0x1400`.
2. After the first write succeeds (at time t(2)), the flash memory contains 5 KB of `0xAA`, and the rest is empty (that is, `0xFF`).
3. Then the second write begins, but before writing to the second block, the block is erased. At this point, t(3), the flash contains 4 KB of `0xAA` and 4 KB of `0xFF`.
4. After the second write finishes, at time t(4), the 2 KB of `0xFF` at address `0x1000` is corrupted.

7.10.3. Fine-Grained Flash Access

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

- `alt_get_flash_info()`
- `alt_erase_flash_block()`
- `alt_write_flash_block()`

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

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

```
#include <stdio.h>
#include <string.h>
#include "sys/alt_flash.h"
#define BUF_SIZE 1024
int main ()
{
   alt_flash_fd* fd;
   int ret_code;
   char source[BUF_SIZE];
   char dest[BUF_SIZE];
   /* Initialize the source buffer to all 0xAA */
   memset(source, 0xAA, BUF_SIZE);
   fd = alt_flash_open_dev("/dev/ext_flash");
   if (fd!=NULL)
      {
         ret_code = alt_write_flash(fd, 0, source, BUF_SIZE);
         if (ret_code==0)
            {
               ret_code = alt_read_flash(fd, 0, dest, BUF_SIZE);
               if (ret_code==0)
                  {
                     /*
                      * Success.
                      * At this point, the flash is all 0xAA and we
                      * have read that all back to dest
                      */
                  }
```

```
            }
         alt_flash_close_dev(fd);
         }
         else
            {
               printf("Cannot open flash device\n");
            }
return 0;
}
```

Related Information

Simple Flash Access on page 99

7.10.3.1. alt_get_flash_info()

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

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

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

Table 30. Example of Writing Flash and Causing Unexpected Data Corruption

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

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

```
typedef struct flash_region {
  int offset; /* Offset of this region from start of the flash */
  int region_size; /* Size of this erase region */
```

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

```
  int number_of_blocks; /* Number of blocks in this region */
  int block_size; /* Size of each block in this erase region */
}
flash_region;
```

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

7.10.3.2. alt_erase_flash()

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

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

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

7.10.3.3. alt_write_flash_block()

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

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

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

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

Example 15. Using the Fine-Grained Flash Access API Functions

```
#include <string.h>
#include "sys/alt_flash.h"
#include "stdtypes.h"
#include "system.h"
#define BUF_SIZE 100
int main (void)
{
   flash_region* regions;
   alt_flash_fd* fd;
   int number_of_regions;
   int ret_code;
   char write_data[BUF_SIZE];
/* Set write_data to all 0xa */
   memset(write_data, 0xA, BUF_SIZE);
   fd = alt_flash_open_dev(EXT_FLASH_NAME);
   if (fd)
```

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

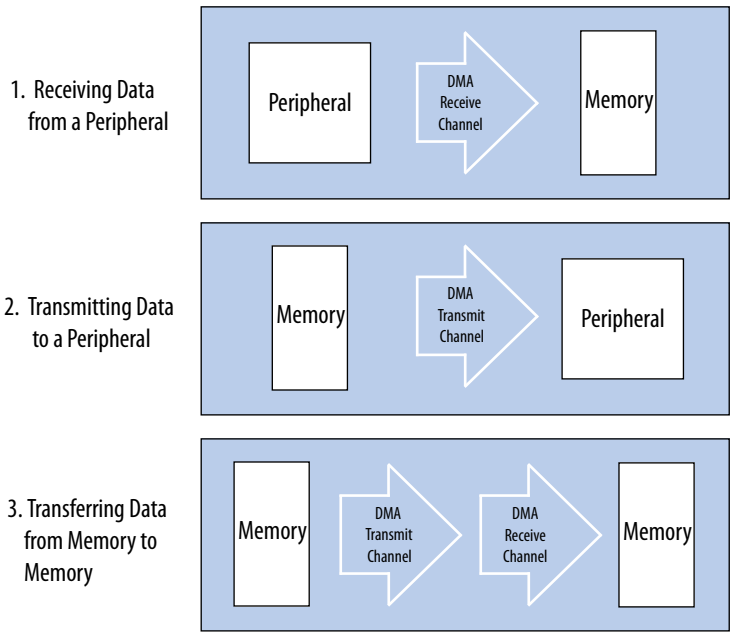
7.11. Using DMA Devices

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

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

Copying data from memory to memory involves both receive and transmit DMA channels simultaneously.

Figure 13. Three Basic Types of DMA Transactions



The API for access to DMA devices is defined in `sys/alt_dma.h`.

7.11.1. DMA Transmit Channels

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

Example 16. Obtaining a File Handle for a DMA Transmit Device dma_0

```
#include <stddef.h>

#include "sys/alt_dma.h"

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

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

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

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

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

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

Related Information

HAL API Reference on page 224
For more information about the API functions.

7.11.2. DMA Receive Channels

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

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

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

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

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

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

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

```
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include "sys/alt_dma.h"
#include "alt_types.h"
/* flag used to indicate the transaction is complete */
volatile int dma_complete = 0;
/* function that is called when the transaction completes */
void dma_done (void* handle, void* data)
   {
   dma_complete = 1;
   }
   int main (void)
      {
      alt_u8 buffer[1024];
      alt_dma_rxchan rx;
      /* Obtain a handle for the device */
      if ((rx = alt_dma_rxchan_open ("/dev/dma_0")) == NULL)
         {
            printf ("Error: failed to open device\n");
            exit (1);
         }
      else
         {
         /* Post the receive request */
         if (alt_dma_rxchan_prepare (rx, buffer, 1024, dma_done, NULL) <
0)
            {
               printf ("Error: failed to post receive request\n");
               exit (1);
            }
         /* Wait for the transaction to complete */
         while (!dma_complete);
            printf ("Transaction complete\n");
            alt_dma_rxchan_close (rx);
      }
      return 0;
}
```

7.11.2.1. Memory-to-Memory DMA Transactions

```
#include <stdio.h>

#include <stdlib.h>

#include "sys/alt_dma.h"

#include "system.h"

static volatile int rx_done = 0;
/*
 * Callback function that obtains notification that the data
 * is received.*/
```

```
static void done(void * handle, void * data) {
  rx_done++;
}
int main(int argc, char * argv[], char * envp[]) {
  int rc;
  alt_dma_txchan txchan;
  alt_dma_rxchan rxchan;
  void * tx_data = (void * ) 0x901000; /* pointer to data to send */
  void * rx_buffer = (void * ) 0x902000; /* pointer to rx buffer */
  /* Create the transmit channel */
  if ((txchan = alt_dma_txchan_open("/dev/dma_0")) == NULL) {
    printf("Failed to open transmit channel\n");
    exit(1);
  }
  /* Create the receive channel */
  if ((rxchan = alt_dma_rxchan_open("/dev/dma_0")) == NULL) {
    printf("Failed to open receive channel\n");
    exit(1);
  }

  /* Post the transmit request */
  if ((rc = alt_dma_txchan_send(txchan,
      tx_data,
      128,
      NULL,
      Null)) < 0) {
    printf("Failed to post transmit request, reason = %i\n", rc);
    exit(1);
  }
  /* Post the receive request */
  if ((rc = alt_dma_rxchan_prepare(rxchan,
        rx_buffer,
        128,
        done,
        NULL
      }
    } < 0) {
    printf("Failed to post read request, reason = %i\n", rc);
    exit(1);
  }
  /* wait for transfer to complete */
  while (!rx_done);
  printf("Transfer successful!\n");
  return 0;
}
```

7.12. Reducing Code Footprint in Embedded Systems

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

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

The following sections describe options to consider when you need to further reduce code size. Implementing the options in the following sections entails making changes to BSP settings.

For more information about manipulating BSP settings, refer to the HAL BSP Settings.

Related Information

Lightweight Character-Mode I/O on page 90

7.12.1. Apply Compiler Flags

riscv32-unknown-elf-gcc is the provided RISC-V toolchain in the Ashling RiscFree IDE for Intel FPGAs. Compiler switch may be of use when optimizing code for small RAM footprints. These switches can be set in the BSP Editor GUI settings for small code footprints.

Table 31. Compiler Switch

Compiler Switch	Value	Description
`hal.make.cflags_optimization`	• `-Os` • `-O0` • `-O1` • `-O2` • `-O3` • `-Os` • `-Ofast` • `-Og` • `-Oz`	Instructs the toolchain to improve performance or code size, or both. *Note:* You must enable `CMAKE_BUILD_TYPE=Release` when running cmake command.
`hal.make.cflags_user_flags`	`-ffunction-sections -fdata-sections`	Instructs the toolchain to enable dead code optimization and remove idle sections.
`hal.make.link_flags`	`-Wl,--gc-sections`	Instructs the toolchain to enable dead code optimization and remove idle sections.

Related Information

GNU Options That Control Optimization

7.12.1.1. Apply Different Compiler Flags in BSP and APP

The `toolchain.cmake` file lists the compiler switches set in the **BSP Editor**. Both BSP and APP `CMakeLists.txt` includes the `toolchain.cmake` for their respective software compilation. Thus, Ashling RiscFree IDE for Intel FPGAs implements the same compiler switches for both BSP and APP projects.

To apply different compiler flags for APP project, overwrite the compiler switches in the APP `CMakeLists.txt`. The following example shows the overwriting lines:

Example 17. Overwrite Compiler Switches in APP `CMakeLists.txt`

```
cmake_minimum_required(VERSION 3.14)
add_subdirectory(../bsp bsp)
include(../bsp/toolchain.cmake)

# Add/Overwrite compile options in toolchain.cmake
add_compile_option(
    -O0
)

# Add/Overwrite link options in toolchain.cmake
add_link_options(
)

# Add/Overwrite compile definitions in toolchain.cmake
```

```
add_compile_definitions(
)

# Remove compile definitions set in toolchain.cmake
remove_definitions(
)

…
```

7.12.1.2. Apply Different Compiler Flags within a Source File

Compilers provide ways to control the optimizer. riscv32-unknown-elf-gcc provides `pragma GCC` to control the compiler behavior temporarily . By using `pragma GCC optimize("O0")`, the optimization level is set to zero (no optimization) on a specified code section.

Example 18. Using pragma GCC in Source Code

```
#pragma GCC push_options
#pragma GCC optimize("O0")
void example() {
    int x;
    x = 0;
    x += 1;
}
#pragma GCC pop_options
```

Assume `hal.make.cflags_optimization` is `-O3`, this specific function is excluded from any optimizations.

7.12.2. Use Small Variant Device Drivers

Some devices provide two driver variants, a fast variant and a small variant. The feature sets provided by these two variants are device specific:

- Fast variant is full-featured with standard code footprint
- Small variant have smaller code footprint and less features

By default the HAL always uses the fast variant. You can select the reduced device driver for all hardware components, or for an individual component, through HAL BSP settings (`hal.enable_reduced_device_drivers`).

The small footprint option might also affect other peripherals. Refer to each peripheral's data sheet for complete details of its driver's small footprint behavior.

Table 32. Intel FPGA Peripherals Offering Small Footprint Drivers

Peripheral	Small Footprint Behavior
UART	Polled operation, rather than IRQ-driven
JTAG UART	Polled operation, rather than IRQ-driven

7.12.3. Reduce the File Descriptor Pool

The file descriptors that access character mode devices and files are allocated from a file descriptor pool. You can reduce the size of the file descriptor pool through a BSP setting. Altera recommends that you reduce the default size of 32 to the number of file descriptors your design is using.

7.12.4. Use /dev/null

At boot time, standard input, standard output, and standard error are all directed towards the null device, that is, **/dev/null**. This direction ensures that calls to `printf()` during driver initialization do nothing and therefore are harmless. After all drivers are installed, these streams are redirected to the channels configured in the HAL. The footprint of the code that performs this redirection is small, but you can eliminate it entirely by selecting `none` for `stdin`, `stdout`, and `stderr`. This selection assumes that you want to discard all data transmitted on standard out or standard error, and your program never receives input through `stdin`. You can control the assignment of `stdin`, `stdout`, and `stderr` channels by manipulating BSP settings.

7.12.5. Use a Smaller File I/O Library

7.12.5.1. Definition of "asnprintf()"

`newlib` implements a maze of 22 `printf()` variants, all based on the same core engine, and all with slightly different interfaces. Some are specified by the C standard, some by Posix, some are GNU extensions, and some are `newlib` extensions. The `newlib libc` web page documents the functions that conform to their respective standards.

For more information about `libc`, refer to the *The Red Hat `newlib` C Libray* web page.

The 22 variants fill a four-dimensional space of design alternatives:

- Plain vs varargs
- File vs buffer output
- Output buffer management alternatives:
  - Fixed buffer, unknown length
  - Fixed buffer, known length
  - `malloc()`ed-and-returned buffer
  - `realloc()`-able buffer
- Plain vs no-floating-point formatting support

The map to the maze is the set of prefix chars.

v*printf (mnemonic: v-for-varargs)

The canonical `printf()` functions take a variable number of arguments including one for each percent-specifier in the format string. This is convenient for you, but inconvenient for wrapper functions taking a variable number of arguments and then doing some sort of value-added before invoking the regular `printf()` library, because C provides no convenient, portable way for a function to read a variable number of arguments and then pass them to another function taking a variable number of arguments. Consequently, for each plain `printf()` function, the C library provides a v*printf variant which accepts the extra arguments in varargs vector format.

The "plain" (variable number of arguments) printf functions are:

- `fprintf()`
- `printf()`
- `asprintf()`
- `asnprintf()`
- `snprintf()`
- `sprintf()`
- `fiprintf()`
- `iprintf()`
- `asiprintf()`
- `asniprintf()`
- `sniprintf()`
- `siprintf()`

The fixed-number-of-argument (vararg-based) variants are:

- `vprintf()`
- `vasprintf()`
- `vasnprintf()`
- `vsprintf()`
- `vsnprintf()`
- `viprintf()`
- `vasiprintf()`
- `vasniprintf()`
- `vsiprintf()`
- `vsniprintf()`

Note: Despite the mnemonic, these functions are not really integer-only: They also support (for example) unsigned, char and string values.

Lumping all the above variants together, the full list of `printf()` functions supported by `newlib` is:

- `fprintf()`
- `vprintf()`
- `printf()`
- `asprintf()`
- `asnprintf()`
- `snprintf()`
- `sprintf()`
- `vasprintf()`
- `vasnprintf()`

- `vsprintf()`
- `vsnprintf()`
- `fiprintf()`
- `iprintf()`
- `viprintf()`
- `asiprintf()`
- `asniprintf()`
- `sniprintf()`
- `siprintf()`
- `vasiprintf()`
- `vasniprintf()`
- `vsiprintf()`
- `vsniprintf()`
- `dprintf`, `vdprintf`: Print to a file descriptor (vs FILE*).
- `dprintf`, `vdprintf`: No-floating-point-formatting versions of above.
- `fwprintf`, `swprintf`, `vfwprintf`, `vswprintf`, `vwprintf`, `wprintf`: Versions with wide-char support.

For more information about these `printf()` variants, refer to the `The Red Hat newlib C Libray` web page.

Related Information

Redhat Website
Online documentation for newlib by Red Hat

7.12.5.2. Use UNIX-Style File I/O

If you need to reduce the code footprint further, you can omit the newlib C library, and use the UNIX-style API.

For more information, refer to the UNIX-Style Interface chapter.

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

7.12.5.3. Emulate ANSI C Functions

If you choose to omit the full implementation of newlib, but you need a limited number of ANSI-style functions, you can implement them easily using UNIX-style functions.

Example 19. Unbuffered getchar()

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

7.12.6. Use the Minimal Character-Mode API

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

- `alt_printf()`: This function is similar to `printf()`, but supports only the %c %s, %x, and %% substitution strings. `alt_printf()` takes up substantially less code space than `printf()`, regardless whether you select the lightweight device driver API. `alt_printf()` occupies less than 1 KB with compiler optimization level `-O2`.
- `alt_putchar()`: Equivalent to `putchar()`.
- `alt_putstr()`: Similar to `puts()`, except that it does not append a newline character to the string.
- `alt_getchar()`: Equivalent to `getchar()`.

These functions are appropriate if your program only needs to accept command strings and send simple text messages.

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

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

For more information about the minimal character-mode functions, refer to the *HAL API Reference* chapter.

Related Information

HAL API Reference on page 224
For more information about the API functions.

7.12.7. Eliminate Unused Device Drivers

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

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

However, there are cases when a device must be present, but runtime software does not require a driver. The most common example is flash memory. The user program might boot from flash, but not use it at runtime; thus, it does not need a flash driver. You can selectively omit any individual driver, select a specific driver version, or substitute your own driver.

For more information about controlling driver configurations, refer to the Nios V Processor Board Support Package Editor chapter.

Another way to control the device driver initialization process is to use the free-standing environment. For more information, refer to the Boot Sequence and Entry Point chapter.

Unlike common device drivers, timer drivers have dedicated HAL settings. If you are not using HAL system clock or timestamp functionality, you can disable the settings to eliminate the unused timer driver. Configure as such:

```
set_setting hal.sys_clk_timer {none}
```

```
set_setting hal.timestamp_timer {none}
```

7.12.8. Use the Picolibc Library

The full `newlib` library functionality is often unnecessary for embedded systems, and undesirably large for systems needing a minimal RAM footprint. Intel FPGA supports a reduced-functionality reduced-size version of `newlib` which allows smaller RAM footprints to be achieved.

When using the Board Support Package, you can select the Picolibc library though the BSP setting.

For the latest update about the Picolibc Library, refer to the Picolibc library homepage developed by Keith Packard.

7.12.9. Eliminate Unused `alt_load()`

When booting Nios V processor using execute-in-place, the crt0 code block calls the `alt_load()` macros. The `alt_load()` macros handles the copying of sections (except `.text`) on reset from a memory into the system RAM. There can be two possible elimination methods to reduce the code footprint.

1. Partial Elimination

   If you are booting from On-Chip Memory (OCRAM), you need to enable `enable_alt_load_copy_rwdata` to preserve a copy of the .rwdata section of the application code. This ensures that the `.rwdata` is restored from modification during run-time, thus the application is initialized with the same .rwdata sections after every system reset.

   To enable the alt_load() function partially,

   ```
   set_setting hal.linker.enable_alt_load {true}
   set_setting hal.linker.enable_alt_load_copy_exceptions {false}
   set_setting hal.linker.enable_alt_load_copy_rodata {false}
   set_setting hal.linker.enable_alt_load_copy_rwdata {true}
   ```

```
set_setting hal.linker.enable_exception_stack {false}
```

2. Complete Elimination

   If you are re-programming a fresh hardware SOF file with initialized OCRAM, or re-downloading software ELF file after each system startup, you may disable the entire `alt_load()` functions. Since there is always a fresh `.rwdata` section before each system startup, there is no obligation to preserve it using the `enable_alt_load_copy_rwdata`.

   To disable the `alt_load()` function entirely.

```
set_setting hal.linker.enable_alt_load {false}
set_setting hal.linker.enable_alt_load_copy_exceptions {false}
set_setting hal.linker.enable_alt_load_copy_rodata {false}
set_setting hal.linker.enable_alt_load_copy_rwdata {false}
set_setting hal.linker.enable_exception_stack {false}
```

Related Information

- System Startup in HAL-based Application on page 42
  Describe the three-stage process of system startup.
- Nios® V Embedded Processor Design Handbook: Nios® V Processor Configuration and Booting Solutions
  Describe the booting methods available for Nios V processor.

7.12.10. Eliminate Unneeded Exit Code

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

You can eliminate unneeded exit code, if your application fulfils the requirements below:

- The application `main()` runs in an infinite loop
- No plan to return from `main()`

Two HAL options allow you to minimize or eliminate this exit code.

7.12.10.1. Eliminate Clean Exit

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

7.12.10.2. Eliminate All Exit Code

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

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

7.12.10.3. Turn off C++ Support

By default, the HAL provides support for C++ programs, including default constructors and destructors. You can disable C++ support by setting false to BSP setting hal.enable_c_plus_plus.

7.13. Interrupt Controllers

For more information about working with interrupt controllers, refer to the Exception Handling chapter.

7.14. Boot Sequence and Entry Point

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

7.14.1. Hosted Versus Free-Standing Applications

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

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

Note: Using the free-standing environment increases the complexity of writing Nios V programs, because you assume responsibility for initializing the system.

For more information about reducing code footprint, refer to and use the suggestions described in the Reducing Code Footprint in Embedded Systems chapter.

Note: It is easier to reduce the HAL BSP footprint by using BSP settings, than to use the free-standing mode.

7.14.2. Boot Sequence for HAL-Based Programs

7.14.2.1. System Initialization Code Boot Sequence

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

- Flushes the instruction and data cache (if present).
- Configures the stack pointer.
- Configures the global pointer register.
- Initializes the block started by symbol (BSS) region to zeroes using the linker-supplied symbols `__bss_start` and `__bss_end`. These are pointers to the beginning and the end of the BSS region.
- If there is no boot loader present in the system, copies to RAM any linker section whose run address is in RAM, such as `.rwdata`, `.rodata`, and `.exceptions`.
- Calls `alt_main()`.

Related Information

Global Pointer Register on page 123

7.14.2.2. Default Implementation Steps

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

- Calls the `alt_irq_init()` function, located in **alt_sys_init.c.** `alt_irq_init()` initializes the hardware interrupt controller. The Nios V processor tools creates the file `alt_sys_init.c` for each HAL BSP.
- Calls `ALT_OS_INIT()` to perform any necessary operating system specific initialization. For a system that does not include an operating system (OS) scheduler, this macro has no effect.
- If you are using the HAL with an operating system, initializes the `alt_fd_list_lock` semaphore, which controls access to the HAL file systems.
- Enables interrupts.
- Calls the `alt_sys_init()` function, also located in **alt_sys_init.c.** `alt_sys_init()` initializes all device drivers and software packages in the system.
- Redirects the C standard I/O channels (`stdin`, `stdout`, and `stderr`) to use the appropriate devices.
- Calls the C++ global constructors, using the `_do_ctors()` function.
- Registers the C++ global destructors to be called at system shutdown.
- Calls `main()`.
- Calls `exit()`, passing the return code of `main()` as the input argument for `exit()`.

`alt_main.c` provides this default implementation. The Nios V processor tools copies `alt_main.c` to your BSP directory.

7.14.3. Customizing the Boot Sequence

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

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

The following line of code is the prototype for `alt_main()`:

```
void alt_main (void)
```

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

`alt_sys_init.c` is a generated file, which you must not modify. However, the Nios V processor tools enables you to control the generated contents of `alt_sys_init.c`. To specify the initialization sequence in `alt_sys_init.c`, you manipulate the `auto_initialize` and `alt_sys_init_priority` properties of each driver, using the `set_sw_property` Tcl command.

Related Information

- Nios V Processor Board Support Package Editor on page 53
  For more information about generated files and how to control the contents of alt_sys_init.c and set_sw_property.
- Developing Device Drivers for the Hardware Abstraction Layer on page 127
  For more information about alt_sys_init.c.

7.15. Memory Usage

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

7.15.1. Memory Sections

By default, HAL-based systems are linked using a generated linker script that is created by the Nios V processor tools. This linker script controls the mapping of code and data to the available memory sections. The autogenerated linker script creates standard code and data sections (`.text`, `.rodata`, `.rwdata`, and `.bss`), plus a section for each physical memory device in the system. For example, if a memory component named `sdram` is defined in the `system.h` file, there is a memory section named `.sdram`.

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

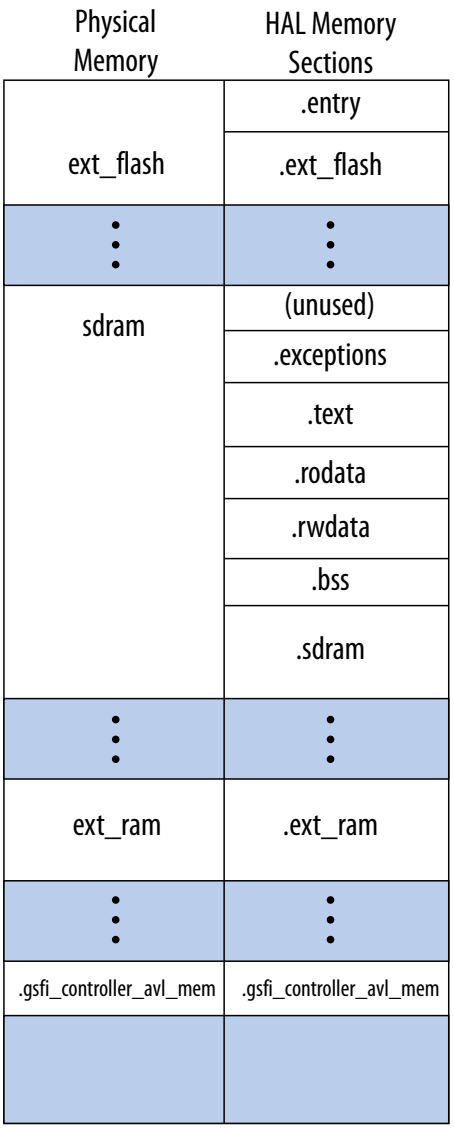
In a memory device containing the reset or exception address, the linker creates a normal (nonreserved) memory section above the `.entry` or `.exceptions` section. If there is a region of memory below the `.entry` or `.exceptions` section, it is unavailable to the Nios V processor software.

7.15.2. Assigning Code and Data to Memory Partitions

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

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

Figure 14. HAL Link Map - Unavailable Memory Region Below the .exceptions Section



7.15.2.1. Simple Placement Options

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

- `.text`—All remaining code
- `.rodata`—The read-only data
- `.rwdata`—Read-write data
- `.bss`—Zero-initialized data

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

You can launch the BSP Editor in Platform Designer, or at the command line of the Nios V processor tools. The BSP Editor allows you to perform the following tasks:

- Manipulates the linker's memory map.
- Displays memory section and region assignments graphically.
- See overlapping or unused sections of memory.

Related Information

- Nios V Processor Board Support Package Editor on page 53
  For more information about the BSP Package Editor.
- Getting Started from the Command Line on page 13

7.15.2.2. Advanced Placement Options

In your program source code, you can specify a target memory section for each piece of data or code. In C or C++, you can use the `section` attribute. This attribute must be placed in a function prototype; you cannot place it in the function declaration itself.

Example 20. Manually Assigning C Code to a Specific Memory Section

```
/* data should be initialized when using the section attribute */
int foo __attribute__ ((section (".ext_ram.rwdata"))) = 0;
void bar (void) __attribute__ ((section (".sdram.text")));
void bar (void)
{
foo++;
}
```

Note: A variable `foo` is placed in the memory named `ext_ram`, and the function `bar()` is placed in the memory named `sdram`.

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

```
.section .ext_ram.text
```

The section names `ext_ram` and `sdram` are examples. You need to use section names corresponding to your hardware.

When creating section names, use the following extensions:

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

  For more information about the use of these features, refer to the GNU compiler and assembler documentation.

Note: A powerful way to manipulate the linker memory map is by using the BSP Editor. With the BSP Editor, you can assign linker sections to specific physical regions, and then review a graphical representation of memory showing unused or overlapping regions.

7.15.3. Placement of the Heap and Stack

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

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

To specify the heap size limit, set the preprocessor symbol ALT_MAX_HEAP_BYTES to the maximum heap size in decimal. For example, the preprocessor argument `-DALT_MAX_HEAP_BYTES=1048576` sets the heap size limit to 0x100000. You can specify this command line option through a BSP setting.

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

Related Information

- Nios V Processor Board Support Package Editor on page 53
  For more information about the BSP Package Editor.
- HAL BSP Settings on page 84
- Nios V Processor Software Development and Implementation on page 18

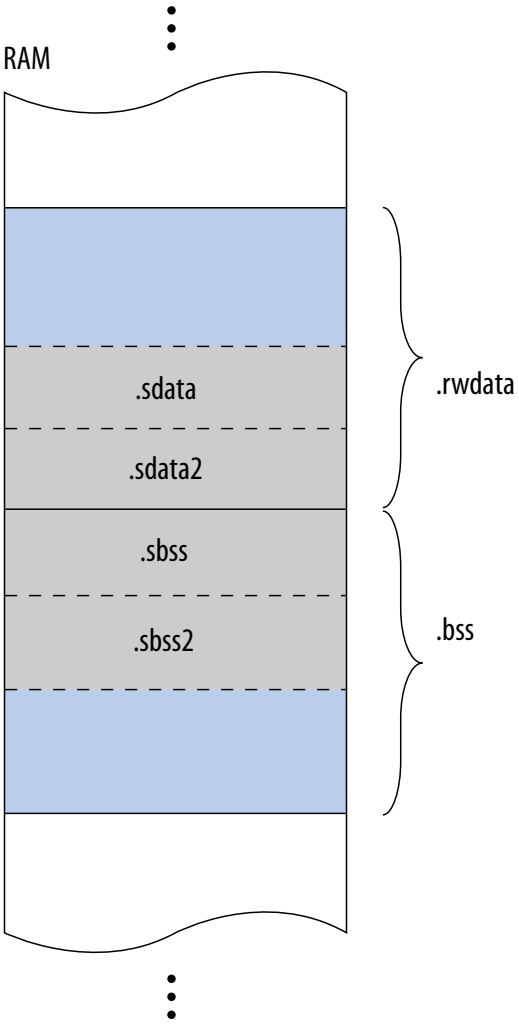
7.15.4. Global Pointer Register

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

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

The small data structures are allocated to the small global data sections, `.sdata`, `.sdata2`, `.sbss`, and `.sbss2`. The small global data sections are subsections of the `.rwdata` and `.bss` sections.

Figure 15. Small Global Data Sections



If the total size of the small global data structures is more than 8 KB, these data structures overflow the global pointer region. The linker produces an error message saying "`Unable to reach` <variable name> `... from the global pointer ... because the offset ... is out of the allowed range, -4096 to 4095.`"

The best solution is to use fewer global variables of size 8 bytes or less. Another solution is to place only the frequently used global variables inside the **.sdata + .sbss**.

When the size of the global variable is greater than 8 bytes, by default, it gets placed outside of the **.sdata + .sbss**, and takes longer to access it. You can improve space-time performance by placing only the frequently used global variables inside the **.sdata + .sbss** leaving space available to add global variables greater than 8 bytes; and by annotating the code to place these global variables inside the **.sdata + .sbss**.

Annotating the code is show in Advanced Placement Options.

For information about manipulating project settings, refer to HAL BSP Settings.

Related Information

System Initialization Code Boot Sequence on page 117

7.15.5. Boot Modes

The processor's boot memory is the memory that contains the reset vector. This device might be an external flash, or it might be an on-chip RAM. Regardless of the nature of the boot memory, HAL-based systems are constructed so that all program and data sections are initially stored in it.

The HAL provides a small boot loader program that copies these sections to their run time locations at boot time. You can specify run time locations for program and data memory by manipulating BSP settings.

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

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

When running in this mode, the HAL executable program must take responsibility for loading any sections that require loading to RAM. The `.rwdata`, `.rodata`, and `.exceptions` sections are loaded before the call to `alt_main()`, as required.

This loading is performed by the function `alt_load()`. To load these functions and data into memory manually; and to load any additional sections, use the `alt_load_section()` function.

For more information about `alt_load_section()`, refer to the HAL API Reference chapter.

Related Information

- HAL API Reference on page 224
- Nios® V Processor Configuration and Booting Solutions
  For more details about Nios® V processor booting method.

7.16. Working with HAL Source Files

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

7.16.1. Finding HAL Files

Determine the location of HAL source files when you create the BSP. HAL source files (and other BSP files) are copied to the BSP directory.

Related Information

HAL API Reference on page 224
For more information about the API functions.

7.16.2. Overriding HAL Functions

HAL source files are copied to your BSP directory when you create your BSP. If you regenerate a BSP, any HAL source files that differ from the installation files are copied. Avoid modifying BSP files. To override default HAL code, use BSP settings, or custom device drivers or software packages.

For more information about what happens when you regenerate a BSP, refer to Revising your BSP.

Note: Avoid modifying HAL source files. If you modify a HAL source file, you cannot regenerate the BSP without losing your changes. This makes it difficult to keep the BSP coordinated with changes to the underlying hardware system.

8. Developing Device Drivers for the Hardware Abstraction Layer

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

The process of integrating a software package with the HAL is nearly identical with the process for integrating a device driver.

Confine direct interaction with the hardware to device driver code. In general, the best practice is to keep most of your program code free of low-level access to the hardware. Wherever possible, use the high-level HAL application program interface (API) functions to access hardware. This makes your code more consistent and more portable to other Nios V systems that might have different hardware configurations.

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

- Integration in the HAL API
- Peripheral-specific API

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

Related Information

Applications versus Drivers on page 81

8.1. Driver Integration in the HAL API

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

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

Related Information

- Overview of the Hardware Abstraction Layer on page 79
- HAL Peripheral Services on page 45
  For information about configuring and using HAL services.

© Altera Corporation. Altera, the Altera logo, the 'a' logo, and other Altera marks are trademarks of Altera Corporation. Altera and Intel warrant performance of its FPGA and semiconductor products to current specifications in accordance with Altera's or Intel's standard warranty as applicable, but reserves the right to make changes to any products and services at any time without notice. Altera and Intel assume no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to inwriting by Altera or Intel. Altera and Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

- Reducing Code Footprint in Embedded Systems on page 108
  For information about increasing your application's performance, or minimizing the code footprint.

8.2. The HAL Peripheral-Specific API

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

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

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

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

Related Information

Integrating a Device Driver in the HAL on page 137
For details about integration in the HAL API.

8.3. Preparing for HAL Driver Development

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

Note: This chapter uses the variable `<Intel Quartus Prime installation>` to represent the location where the Quartus Prime software is installed. On a Windows system, by default, that location is `c:/<intelFPGA or intelFPGA_pro>/<version number>`.

For more information, refer to the Developing Programs Using the Hardware Abstraction Layer section.

8.4. Development Flow for Creating Device Drivers

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

1. Create the device header file that describes the registers.
2. Implement the driver functionality.
3. Test the driver
4. Integrate the device driver in the HAL framework.

8.5. Nios V Processor Hardware Design Concepts

This section discusses some basic concepts behind the Intel Platform Designer system integration tools. These concepts can enhance your understanding of the driver development process. You do not normally need to use a system integration tool when developing Nios V processor device drivers.

8.5.1. The Relationship Between the `.qsys` File and `system.h`

The system generation tool, Platform Designer, generates the Nios V processor system hardware. Hardware designers use the system generation tool to specify the architecture of the Nios V processor system and integrate the necessary peripherals and memory. Therefore, the definitions in `system.h`, such as the name and configuration of each peripheral, are a direct reflection of design choices made in the system generation tool. These design choices are encapsulated in the `.qsys` file. `system.h` is derived from the `.qsys` file.

For more information about the `system.h` header file, refer to the Developing Programs Using the Hardware Abstraction Layer section.

8.5.2. Using the System Generation Tool to Optimize Hardware

If you find unexpected definitions in `system.h`, remember that you can modify the contents of `system.h` by changing the underlying hardware with the system generation tool, Platform Designer. Before you modify a device driver to accommodate the hardware, it is worth considering whether the hardware can be improved easily with Platform Designer.

8.5.3. Components, Devices, and Peripherals

The Platform Designer system generation tools use the term "component" to describe hardware modules included in the system. In the context of Nios V software development, components are devices, such as peripherals or memories. In the following sections, "component" is used interchangeably with "device" and "peripheral" when the context is closely related to the system generation tool.

8.6. Accessing Hardware

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

All components provide a directory that defines the device hardware and software. For example, each component provided in the Quartus Prime software has its own directory in the `<Intel Quartus Prime installation>/ip/altera/` directory. Many components provide a header file that defines their hardware interface. The header file is named `<component name>_regs.h`, included in the `inc` subdirectory for the specific component. For example, the Intel-provided JTAG UART component defines its hardware interface in the file `<Intel Quartus Primeinstallation>/ip/altera/sopc_builder_ip/altera_avalon_jtag_uart/inc/altera_avalon_jtag_uart_regs.h`.

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

- Register access macros that provide either or both a read or write macro for each register in the component that supports the operation. The macros are:
  - `IORD_<component name>_<register name> (<component base address>)`
  - `IOWR_<component name>_<register name> (<component base address>, <data>)`

    For example, `altera_avalon_jtag_uart_regs.h` defines the following macros:
  - `IORD_ALTERA_AVALON_JTAG_UART_DATA()`
  - `IOWR_ALTERA_AVALON_JTAG_UART_DATA()`
  - `IORD_ALTERA_AVALON_JTAG_UART_CONTROL()`
  - `IOWR_ALTERA_AVALON_JTAG_UART_CONTROL()`
- Register address macros that return the physical address for each register in a component. The address register returned is the component's base address + the specified register offset value. These macros are named **IOADDR_**<component name>_<register name> (<component base address>).

  For example, `altera_avalon_jtag_uart_regs.h` defines the following macros:
  - `IOADDR_ALTERA_AVALON_JTAG_UART_DATA()`
  - `IOADDR_ALTERA_AVALON_JTAG_UART_CONTROL()`

    Use these macros only as parameters to a function that requires the specific address of a data source or destination. For example, a routine that reads a stream of data from a particular source register in a component might require the physical address of the register as a parameter.
- Bit-field masks and offsets that provide access to individual bit-fields in a register. These macros have the following names:
  - <component name>_<register name>_<name of field>_MSK—A bit-mask of the field
  - <component name>_<register name>_<name of field>_OFST—The bit offset of the start of the field

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

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

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

For more information about a complete example of the `_regs.h` file, refer to the component directory for any of the Intel-supplied components, such as `<Intel Quartus Prime installation>/ip/sopc_builder_ip/altera_avalon_jtag_uart/inc`.

8.7. Creating Embedded Drivers for HAL Device Classes

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

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

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

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

Related Information

Overview of the Hardware Abstraction Layer on page 79

8.7.1. Character-Mode Device Drivers

8.7.1.1. Create a Device Instance

For a device to be made available as a character mode device, it must provide an instance of the `alt_dev` structure. The `alt_dev` structure, defined in `<Intel Quartus Prime installation directory>/ip/altera/soft_processor/altera_hal2/HAL/inc/sys/alt_dev.h` , is essentially a collection of function pointers. These functions are called in response to application accesses to the HAL file system. For example, if you call the function `open()` with a file name that corresponds to this device, the result is a call to the `open()` function provided in this structure.

Example 21. alt_dev Structure

```
typedef struct
{
   alt_llist llist; /* for internal use */
   const char* name;
   int (*open) (alt_fd* fd, const char* name, int flags, int mode);
   int (*close) (alt_fd* fd);
   int (*read) (alt_fd* fd, char* ptr, int len);
   int (*write) (alt_fd* fd, const char* ptr, int len);
   int (*lseek) (alt_fd* fd, int ptr, int dir);
   int (*fstat) (alt_fd* fd, struct stat* buf);
```

```
   int (*ioctl) (alt_fd* fd, int req, void* arg);
}
alt_dev;
```

For more information about `open()`, `close()`, `read()`, `write()`, `lseek()`, `fstat()`, and `ioctl()`, refer to the HAL API Reference section.

8.7.1.1.1. Modifying the Global Error Status, errno

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

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

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

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

The following code defines the `alt_fd` structure:

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

where:

- `dev` is a pointer to the device structure for the device being used.
- `fd_flags` is the value of `flags` passed to `open()`.
- `priv` is a reserved, implementation-dependent argument, defined by the driver. If the driver requires any special, non-HAL-defined values to be maintained for each file or stream, you can store them in a data structure, and use `priv` maintains a pointer to the structure. The HAL ignores `priv`.

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

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

8.7.1.1.2. Default Behavior for Functions Defined in alt_dev

A driver is not required to provide all of the functions in the `alt_dev` structure. If a given function pointer is set to `NULL`, a default action is used instead.

Table 33. Default Behavior for Functions Defined in alt_dev

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

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

8.7.1.2. Register a Character Device

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

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

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

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

For more information, refer to the Developing Programs Using the Hardware Abstraction Layer section.

8.7.2. File Subsystem Drivers

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

8.7.2.1. Create a Device Instance

Creating and registering a file system is very similar to creating and registering a character-mode device. To make a file system available, create an instance of the `alt_dev` structure.

For more information, refer to the Character-Mode Device Drivers chapter.

The only distinction is that the `name` field of the device represents the mount point for the file subsystem. Of course, you must also provide any necessary functions to access the file subsystem, such as `read()` and `write()`, similar to the case of the character-mode device.

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

8.7.2.2. Register a File Subsystem Device

You can register a file subsystem using the following function:

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

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

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

For more information, refer to the Developing Programs Using the Hardware Abstraction Layer section.

8.7.3. Timer Device Drivers

8.7.3.1. System Clock Driver

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

```
void alt_tick (void)
```

The expectation is that `alt_tick()` is called in exception context.

To register the presence of a system clock driver, call the following function:

```
int alt_sysclk_init (alt_u32 nticks)
```

The input argument `nticks` is the number of system clock ticks per second, which is determined by your system clock driver. The return value of this function is zero on success, and nonzero otherwise.

For more information about writing interrupt service routines, refer to the Exception Handling section.

8.7.3.2. Timestamp Driver

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

For more information about using these functions, refer to the Developing Programs Using the Hardware Abstraction Layer section.

Related Information

HAL API Reference on page 224
For more information about the API functions.

8.7.4. Flash Device Drivers

8.7.4.1. Create a Flash Driver

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

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

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

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

- `alt_flash_open_dev()`
- `alt_flash_close_dev()`
- `alt_write_flash()`
- `alt_read_flash()`
- `alt_get_flash_info()`
- `alt_erase_flash_block()`
- `alt_write_flash_block()`
- `alt_flash_lock()`

where:

- the `base_addr` parameter is the base address of the flash memory
- `length` is the size of the flash in bytes
- `number_of_regions` is the number of erase regions in the flash
- `region_info` contains information about the location and size of the blocks in the flash device

For more information about the format of the `flash_region` structure, refer to Using Flash Devices in Developing Programs Using the Hardware Abstraction Layer.

Some flash devices allow you to read out the number of regions and their configuration at run time. For all other flash devices, these two fields must be defined at compile time.

8.7.4.2. Register a Flash Device

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

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

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

8.7.5. DMA Device Drivers

The HAL models a DMA transaction as being controlled by two endpoint devices: a receive channel and a transmit channel.

For more information about a complete description of the HAL DMA device model, refer to Using DMA Devices in the Developing Programs Using the Hardware Abstraction Layer section.

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

8.7.5.1. DMA Transmit Channel

Example 22. alt_dma_txchan Structure

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

8.7.5.2. DMA Receive Channel

Example 23. alt_dma_rxchan Structure

```
typedef struct alt_dma_txchan_dev_s alt_dma_txchan_dev;
struct alt_dma_txchan_dev_s {
  alt_llist llist;
  const char * name;
  int( * space)(alt_dma_txchan dma);
  int( * send)(alt_dma_txchan dma,
    const void * from,
      alt_u32 len,
```

```
      alt_txchan_done * done,
      void * handle);
  int( * ioctl)(alt_dma_txchan dma, int req, void * arg);
};
```

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

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

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

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

Table 34. Fields in the alt_dma_rxchan Structure

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

Related Information

HAL API Reference on page 224
For more information about the API functions.

8.8. Integrating a Device Driver in the HAL

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

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

Note: The process required to integrate a device driver is nearly identical to that required to develop a software package. The following sections describe the process for both.

Related Information

- The HAL Peripheral-Specific API on page 128
- Overriding the HAL Default Device Drivers on page 153

8.8.1. Overview

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

- A header file defining the package or driver interface
- A Tcl script specifying how to add the package or driver to a BSP

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

8.8.2. Assumptions and Requirements

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

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

- It must have a defining Tcl script. The Tcl script for each driver or software package provides the Nios V processor tools with a complete description of the driver or software. This description includes the following information:
  - Name—A unique name identifying the driver or software package
  - Source files—The location, name, and type of each C/C++ or assembly language source or header file
  - Associated hardware class (device drivers only)—The name of the hardware peripheral class the driver supports
  - Version and compatibility information—The driver or package version, and (for drivers) information about what device core versions it supports.
  - BSP type(s)—The supported operating system(s)
  - Settings—The visible parameters controlling software build and runtime configuration
- The Tcl script resides in the driver or software package root directory.
- The Tcl script's file name ends with **_sw.tcl.** Example: **custom_ip_block_sw.tcl.**
- The root directory of the driver or software package is located in a directory named **ip**, one level beneath the Quartus Prime project directory containing the design your BSP targets. This approach is recommended if your driver or software package is used only once, in a specific hardware project.

If your driver or software package uses the HAL auto initialization mechanism (`alt_sys_init()`), certain macros must be defined in a header file.

Related Information

- File Names and Locations on page 140
  For more information about how file names and directory structures conform to certain conventions
- Header Files and alt_sys_init.c on page 149
  For more information about header file.

- Board Support Package Tcl Commands on page 333
  For more information about the commands you can use in a driver Tcl script.
- AN 459: Guidelines for Developing a Nios II HAL Device Driver
  For more information about integrating a HAL device driver.

8.8.3. Nios V Processor BSP Generation Flow

This section describes the process by which the BSP Editor adds device drivers and software packages to your BSP. The Nios V processor tools provides a combination of command utilities, TCL commands and scripts that enable you to create and manage BSP settings. You can also configure Nios V Processor Tool in the Ashling RiscFree IDE to generate the BSP.

8.8.3.1. Component Discovery

When you run the BSP Editor to generate the BSP, a library of available drivers and software packages is populated.

The BSP generator locates software packages and drivers by inspecting a list of known locations determined by the Nios V processor tools, Quartus Prime software, and IP core® IP Library installers, as well as searching locations specified in certain system environment variables.

The BSP Editor identifies drivers and software packages by locating and sourcing Tcl scripts with file names ending in `_sw.tcl` in these locations.

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

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

8.8.3.2. Device Driver Versions

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

8.8.3.3. Device Driver and Software Package Inclusion

8.8.3.3.1. Specific Requests

The BSP generator adds software packages to the BSP if they are specifically requested during BSP generation, with the `enable_sw_package` command. If a software package or specific driver is requested and cannot be located, an error is generated and BSP generation or settings update halts.

For more information, refer to Board Support Package Tcl Commands.

8.8.3.3.2. No Specific Requests

If no specific device driver is requested, and no compatible device driver is located for a particular hardware module, the BSP Editor issues an informative message visible in either the `debug` or `verbose` generation output. This behavior is normal for many types of hardware, such as memory devices, that do not have device drivers.

Creating a Tcl script allows you to add extra definitions in the `system.h` file, enable automatic driver initialization through the `alt_sys_init.c` structure, and enable the Nios V Processor Tools to control any extra parameters that might exist.

With the Tcl software definition files in place, the Software Build Tools read in the Tcl file and populate the `CMakeLists.txt` and other support files accordingly.

When the BSP Editor adds each driver or software package to the system, it uses the data in the Tcl script defining the driver or software package to control each file copied in to the BSP. This rule also affects generated BSP files such as the `CMakeLists.txt`, `toolchain.cmake`, `system.h`, and the BSP settings and summary HTML files.

8.8.4. File Names and Locations

The Nios V processor software build tools find a device driver or software package by locating a Tcl script with the file name ending in `_sw.tcl`, and sourcing it.

Each peripheral in a Nios V processor system is associated with a specific component directory. This directory contains a file defining the software interface to the peripheral. For more information, refer to the Accessing Hardware chapter.

The file hierarchy that is suitable for the Nios V processor tools is located in the `<Intel Quartus Prime installation>/ip/altera/sopc_builder_ip` directory. This example assumes a device driver supporting a hardware component named `custom_component`.

Related Information

- Assumptions and Requirements on page 138
- AN 459: Guidelines for Developing a Nios II HAL Device Driver
  For more information about integrating a HAL device driver.

8.8.4.1. Source Code Discovery

Use Tcl scripts to specify the location of driver source files.

For more information, refer to the Nios V Processor BSP Generation Flow.

8.8.5. Driver and Software Package Tcl Script Creation

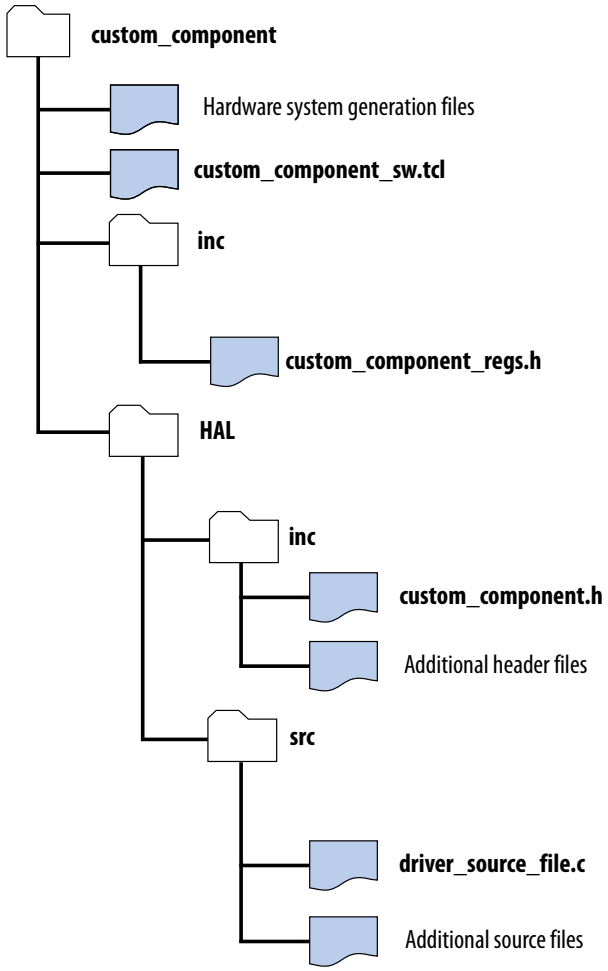
This section discusses writing a Tcl script to describe your software package or driver. The exact contents of the Tcl script depends on the structure and complexity of your driver or software. For many simple device drivers, you need only include a few commands. For more complex software, the BSP Editor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

The Tcl command and argument descriptions in this section are not exhaustive. For a detailed explanation of each command and all arguments, refer to the Board Support Package Tcl Commands section.

8.8.5.1. Example Device Driver File Hierarchy and Naming

For a reference in creating your own driver or software Tcl files, you can also view the driver and software package Tcl scripts included with the Nios V processor tools and the IP core IP library. These scripts are in the `<Intel Quartus Prime installation directory>/ip/altera/soft_processor` and `<Intel Quartus Prime installation directory>/ip/altera/sopc_builder_ip` folders, respectively.

Figure 16. Example Device Driver File Hierarchy and Naming



Note: **inc** - Contains header file(s) that define the device hardware interfaces. Contents in this directory are not HAL-specific and apply to a driver, regardless of whether it is based on the HAL, MicroC/OS-II, or any other RTOS environment.

Note: **HAL** - Contains software files required to integrate the device with the Nios V processor hardware abstraction layer. Files in this directory pertain specifically to the HAL.

8.8.5.2. Tcl Command Walkthrough for a Typical Driver or Software Package

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

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

8.8.5.2.1. Creating and Naming the Driver or Package

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

```
create_driver my_custom_component_driver
```

8.8.5.2.2. Identifying the Hardware Component Class

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

```
set_sw_property hw_class_name my_custom_component
```

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

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

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

For more information, refer to the Board Support Package Tcl Commands chapter.

Choose a name for your driver or software package that does not conflict with other Altera-supplied software or IP, or any third-party software or IP installed on your host system. The BSP generator uses the name you specify to look up the software package or driver during BSP creation. If the Nios V processor finds multiple compatible drivers or software packages with the same name, the behavior is undefined.

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

8.8.5.2.3. Setting the BSP Type

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

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

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

8.8.5.2.4. Specifying an Operating System

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

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

8.8.5.2.5. Specifying Source Files

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

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

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

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

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

8.8.5.2.6. Specifying a Subdirectory

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

```
set_sw_property bsp_subdirectory my_driver
```

Note: If the path begins with the BSP type (e.g `HAL`, `UCOSII`, `FREERTOS`, or `FreeRTOS-Kernel`), the BSP type is removed and replaced with the value of the `bsp_subdirectory` property.

8.8.5.2.7. Enabling Software Initialization

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

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

```
set_sw_property auto_initialize true
```

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

8.8.5.2.8. Adding Include Paths

By default, the generated BSP `CMakeLists.txt` include paths to find header files in `/inc` or <BSP `type`>`/inc` folders.

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

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

Note: If the path begins with the BSP type (e.g `HAL`, `UCOSII`, `FREERTOS`, or `FreeRTOS-Kernel`), the BSP type is removed and replaced with the value of the `bsp_subdirectory` property.

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

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

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

8.8.5.2.9. Version Compatibility

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

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

The <version> argument in each of the following versioning-related commands can be a string containing numbers and characters. Examples of version strings are `18.0`, `19.1.1`, `20.1`, and `20.1sp1`. The "." character is a separator. The BSP editor compares versions against each other to determine if one is more recent than the other, or if two are equal, by successively comparing the strings between each separator. Thus, `20.1` is greater than `19.4`, and `20.1sp1` is greater than `20.1`. Two versions are equal if their version assignment strings are identical.

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

```
set_sw_property version 21.1
```

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

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

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

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

8.8.5.3. Creating Settings for Device Drivers and Software Packages

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

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

8.8.5.3.1. How Settings Affect the Generated BSP

Settings affect the generated BSP in several ways:

- Settings are added either to the BSP `system.h` or `toolchain.cmake` as a preprocessor macro.
- Settings are stored in the BSP settings file, named with hierarchy information to prevent namespace collision.
- A default value of your choice is assigned to the setting so that the end user of the driver or package does not need to explicitly specify the setting when creating or updating a BSP.
- Settings are displayed in the BSP `summary.html` document, along with description text of your choice.

Arguments for add_sw_setting

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

- `type` - The data type, which controls formatting of the setting's value assignment in the appropriate generated file.
- `destination` - The destination file in the BSP.
- `displayName` - The name that is used to identify the setting when changing BSP settings or viewing the BSP summary.html document.
- `identifier` - Conceptually, this argument is the macro defined in a C language definition (the text immediately following #define), or the name of a variable in a `toolchain.cmake`.
- `value` - A default value assigned to the setting if the BSP user does not manually change it.
- `description` - Descriptive text, shown in the BSP summary.html document

8.8.5.3.2. Data Types

Several setting data types are available, controlled by the `type` argument to `add_sw_setting`. They correspond to the data types you can express as `#define` statements or values concatenated to `toolchain.cmake` variables. The specific setting type depends on your software's structure or BSP build needs.

Table 35. Data Type Settings

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

8.8.5.3.3. Setting Destination Files

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

Table 36. Destination File Settings

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

Note: Certain setting types are not compatible with the `toolchain.cmake` destination file types.

For more information, refer to the Tcl Commands for Drivers and Packages chapter.

8.8.5.3.4. Setting Display Name

A setting's `displayName` defines how you refers to the setting in the BSP tools. The BSP Tools append the `displayName` text after a . (dot) separator to your driver or software package's name (as defined in the `create_driver` or `create_sw_package` command). For example, if your driver is named

`my_peripheral_driver` and your setting's `displayName` is `small_driver`, BSPs with your driver have a setting `my_peripheral_driver.small_driver`. Thus each driver and software package has its own settings namespace.

8.8.5.3.5. Setting Identifier

The identifier argument of `add_sw_setting` controls the physical name of the setting in the generated BSP files. The physical name corresponds to the definition being created in `toolchain.cmake` and `system.h` . The identifier is commonly the text that your software uses in conditionally-compiled code. For example, suppose your software creates a buffer as follows:

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

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

8.8.5.3.6. Setting Default Value

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

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

For more information, refer to the Tcl Commands for Drivers and Packages chapter.

8.8.5.3.7. Setting Description

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

8.8.5.3.8. Setting Creation Example

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

Note: This example implements a setting for a driver that has two variants of a function, one implementing a small driver (minimal code footprint) and the other a fast driver (efficient execution).

A simple Boolean definition setting is added to your driver Tcl file. This feature allows BSP users to control your driver through the BSP settings interface.

- When you set the setting to `true` or `1`, the BSP defines `MY_CUSTOM_DRIVER_SMALL` in either `system.h` or the BSP `toolchain.cmake` file.
- When you compile the BSP, your driver is compiled with the appropriate routine incorporated in the object file.
- When you disable the setting, `MY_CUSTOM_DRIVER_SMALL` is not defined.

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

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

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

For more information about detailed usage and restrictions, refer to the Board Support Package Tcl Commands chapter.

8.9. Creating a Custom Device Driver for the HAL

This section describes how to provide appropriate files to integrate your device driver in the HAL. For more information about the correct locations for the files, refer to the Integrating a Device Driver in the HAL chapter

8.9.1. Header Files and alt_sys_init.c

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

When you create the driver or software package, you specify in a Tcl script whether you want the `alt_sys_init()` function to invoke your `INSTANCE` and `INIT` macros.

For more information, refer to the Enabling Software Initialization chapter.

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

Related Information

- Assumptions and Requirements on page 138
- AN 459: Guidelines for Developing a Nios II HAL Device Driver
  For more information about integrating a HAL device driver.

8.9.1.1. Creating alt_sys_init.c Based on Associated Header Files

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

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

Example 24. Excerpt from an alt_sys_init.c File Performing Driver Initialization

```
#include "system.h"
#include "sys/alt_sys_init.h"
/*
* device headers
*/
#include "altera_avalon_timer.h"
#include "altera_avalon_uart.h"
/*
* Allocate the device storage
*/
ALTERA_AVALON_UART_INSTANCE( UART1, uart1 );
ALTERA_AVALON_TIMER_INSTANCE( SYSCLK, sysclk );
/*
* Initialize the devices
*/
void alt_sys_init( void )
{
    ALTERA_AVALON_UART_INIT( UART1, uart1 );
    ALTERA_AVALON_TIMER_INIT( SYSCLK, sysclk );
}
```

8.9.2. Device Driver Source Code

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

When writing a device driver, apply the HAL I/O macros in the source code. The HAL provides the C-language macros `IORD` and `IOWR` that expand to the load and store assembly instructions. These macros are provided to enable HAL device drivers to access device registers.

Note: These instructions cannot bypass the data cache. You are required to place device registers within a peripheral region.

Generally, your program passes values defined in `system.h` as the BASE and `REGNUM` parameters. These macros are defined in the file `<Intel Quartus installation directory>/ip/altera/soft_processor/intel_niosv_common/HAL/inc/io.h`.

Table 37. HAL I/O Macro

Macro	Use
`IORD(BASE, REGNUM)`	Read the value of the register at offset `REGNUM` in a device with base address `BASE`. Registers are assumed to be offset by the address width of the bus.
`IOWR(BASE, REGNUM, DATA)`	Write the value `DATA` to the register at offset `REGNUM` in a device with base address `BASE`. Registers are assumed to be offset by the address width of the bus.
`IORD_32DIRECT(BASE, OFFSET)`	Make a 32-bit read access at the location with address `BASE+OFFSET`.
`IORD_16DIRECT(BASE, OFFSET)`	Make a 16-bit read access at the location with address `BASE+OFFSET`.
`IORD_8DIRECT(BASE, OFFSET)`	Make an 8-bit read access at the location with address `BASE+OFFSET`.
`IOWR_32DIRECT(BASE, OFFSET, DATA)`	Make a 32-bit write access to write the value `DATA` at the location with address `BASE+OFFSET`.
`IOWR_16DIRECT(BASE, OFFSET, DATA)`	Make a 16-bit write access to write the value `DATA` at the location with address `BASE+OFFSET`.
`IOWR_8DIRECT(BASE, OFFSET, DATA)`	Make an 8-bit write access to write the value `DATA` at the location with address `BASE+OFFSET`.

8.9.2.1. altera_avalon_jtag_uart.h Defining Macros

For example, `altera_avalon_jtag_uart_regs.h` defines specific I/O macros based on HAL I/O macros. It is written to access the DATA and CONTROL registers of the JTAG UART IP. Refer to Embedded Peripheral IP User Guide – JTAG UART Core Software Programming Model for more information on the soft IP's register map.

Using predefined `REGNUM` (`DATA = 0` and `CONTROL = 1`), the new I/O macros simplify the HAL I/O macros to two basic parameters – `BASE` and `DATA`.

- `IORD_ALTERA_AVALON_JTAG_UART_DATA(BASE)`- Simplify `IORD(BASE, REGNUM)` to `IORD(BASE, 0)`
- `IOWR_ALTERA_AVALON_JTAG_UART_DATA(BASE, DATA)`- Simplify `IOWR(BASE, REGNUM, DATA)` to `IOWR(BASE, 0, DATA)`
- `IORD_ALTERA_AVALON_JTAG_UART_CONTROL(BASE)`- Simplify `IORD(BASE, REGNUM)` to `IORD(BASE, 1)`
- `IOWR_ALTERA_AVALON_JTAG_UART_CONTROL(BASE, DATA)`- Simplify `IOWR(BASE, REGNUM, DATA)` to `IOWR(BASE, 1, DATA)`

These specific I/O macros are later applied intensively throughout the soft IP device driver source codes,

- `altera_avalon_jtag_uart_fd.c`
- `altera_avalon_jtag_uart_init.c`
- `altera_avalon_jtag_uart_ioctl.c`
- `altera_avalon_jtag_uart_read.c`
- `altera_avalon_jtag_uart_write.c`

Besides defining new I/O macros, you can adopt similar practice with defining the mask and offset macros in `altera_avalon_jtag_uart_regs.h`. They are commonly used in bitwise operations with register bit fields.

For more information about a complete example, refer to any of the Intel supplied device drivers, such as the JTAG UART driver in `<Intel Quartus Prime installation>/ip/sopc_builder_ip/altera_avalon_jtag_uart`.

Related Information

Software Programming Model, Embedded Peripherals IP User Guide
For more information about JTAG UART Core Software Programming Model

8.10. Reducing Code Footprint in HAL Embedded Drivers

The HAL provides several options for reducing the size, or footprint, of the BSP code. Some of these options require explicit support from device drivers. If you need to minimize the size of your software, consider using reduced footprint drivers. This technique usually reduces driver functionality.

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

When writing a device driver, if you choose to ignore the value of `ALT_USE_SMALL_DRIVERS`, the same version of the driver is used regardless of the definition of this macro. You can enable `ALT_USE_SMALL_DRIVERS` in a BSP with the `hal.enable_reduced_device_drivers` BSP setting.

For more information, refer to the Settings Managed by Nios V Boards Support Package Editor.

8.11. HAL Namespace Allocation

To avoid conflicting names for symbols defined by devices in the hardware system, all global symbols need a defined prefix. Global symbols include global variable and function names. For device drivers, the prefix is the name of the component followed by an underscore. Because this naming can result in long strings, an alternate short form is also permitted. This short form is based on the vendor name, for example `alt_` is the prefix for components published by Intel. It is expected that vendors test the interoperability of all components they supply.

For example, for the `altera_avalon_jtag_uart` component, the following function names are valid:

- `altera_avalon_jtag_uart_init()`
- `alt_jtag_uart_init()`

The following names are invalid:

- `avalon_jtag_uart_init()`
- `jtag_uart_init()`

As source files are located using search paths, these namespace restrictions also apply to file names for device driver source and header files.

8.12. Overriding the HAL Default Device Drivers

All components can elect to provide a HAL device driver.

However, if the driver supplied with a component is inappropriate for your application, you can override the default driver by supplying a different driver.

You can use the BSP Editor to specify a custom driver in the BSP Editor GUI or on the command line.

On the command line, you specify a custom driver with the following BSP Tcl command:

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

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

```
niosv-bsp -c -p=top.qpf -s=sys.qsys -T=hal settings.bsp --cmd=”set_driver
custom_driver {uart0}”
```

Related Information

Integrating a Device Driver in the HAL on page 137

9. Trap Handling

9.1. Nios V Processor Trap Handling Overview

The Nios V processor provides the following trap types:

- Platform interrupts
- Software interrupts
- Timer interrupts
- Software exceptions, which fall into the following categories:
  - Instruction-related exceptions
  - Miscellaneous exceptions

The Nios V processor offers the CLINT or CLIC architecture to handle traps. The interrupt controllers are discussed in detail in the Hardware Interrupt Controllers chapter.

9.1.1. Trap Handling Terminology

The following list of HAL terms outlines basic trap handling concepts:

- Application context—The status of the Nios V processor and the HAL during normal program execution, outside of CLINT architecture.
- Context switch—The process of saving the Nios V processor's registers on an exception or interrupt, and restoring them on return from the exception handling routine or ISR.
- Trap—A transfer of control away from a program's normal flow of execution, caused by an event, either internal or external to the processor, which requires immediate attention. Traps include exceptions and interrupts.
- Trap context—The status of the Nios V processor and the HAL after a trap(executing trap handling code, exception handler or ISR).
- Traps handling system—The complete system of routines that service all traps, and pass control to software exception handlers or ISRs.
- Traps latency—The time elapsed between the event that caused the trap (such as an unimplemented instruction or interrupt request) and the execution of the first instruction at the exception or interrupt handler address.
- Traps response time—The time elapsed between the event that caused the trap and the execution of the exception handler or ISR.
- Traps overhead—Additional processing required to service a trap, including HAL-specific processing and RTOS-specific processing if applicable.
- Trap Handling code—HAL-provided code that sets up the correct processor environment before passing control to the exception handler or ISR.

© Altera Corporation. Altera, the Altera logo, the 'a' logo, and other Altera marks are trademarks of Altera Corporation. Altera and Intel warrant performance of its FPGA and semiconductor products to current specifications in accordance with Altera's or Intel's standard warranty as applicable, but reserves the right to make changes to any products and services at any time without notice. Altera and Intel assume no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to inwriting by Altera or Intel. Altera and Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

- Exception Handler—Code specific to the exception type. The exception handler is distinct from the trap handling code, which takes care of general trap overhead tasks.
- Implementation-dependent instruction—A Nios V processor instruction that is not supported on all implementations of the Nios V core. For example, the `mul` instructions are implementation-dependent, because they are not supported on the Nios V/m core.
- Interrupt—Platform, software and timer interrupt.
- Interrupt controller—Known as CLINT. It enables Nios V processor to respond to an interrupt by transferring control to an ISR.
- Interrupt request (IRQ)—A status signal from an external peripheral that notifies the CPU that it (the peripheral) requires servicing.
- Interrupt service routine (ISR)—A software routine that handles an individual interrupt.
- Invalid instruction—An instruction that is not defined for any implementation of the Nios V processor.
- Maskable traps—Interrupts that can be disabled with the global `mtatus.mie` flag, or the dedicated trap enable bits in the `mie` register.
- Maximum disabled time—The maximum amount of continuous time that the system spends with maskable interrupts disabled.
- Maximum masked time—The maximum amount of continuous time that the system spends with a single interrupt masked.
- Miscellaneous exception—A software exception which is neither an unimplemented instruction nor an `ebreak` or `ecall` instruction.
- Nested interrupts—The process of a high-level interrupt taking control when a lower-level ISR is already running. Not supported in CLINT.
- Exception—An exception caused by a fatal hardware events, breakpoint or environment calls; that is, any traps other than an interrupt. Fatal hardware events includes address misaligned, access fault and illegal instruction.
- Worst-case trap latency—The value of the exception (or interrupt) latency, including the maximum disabled time or maximum masked time. Including the maximum disabled or masked time accounts for the case when the trap occurs at the beginning of the masked or disabled time.

9.2. Hardware Interrupt Controllers

Altera implements Nios V processor trap handling based on the RISC-V specification. The configuration of Nios V trap processing logic depends on either the Core-Local Interruptor (CLINT) or the Core-Local Interrupt Controller (CLIC). When you instantiate the Nios V processor in the system integration tool Platform Designer, you can configure the trap processing logic (**Interrupt Mode**) as either Direct Mode, Vectored Mode or CLIC mode.

Once the processor exits reset, its Machine Trap-Vector Base Address (mtvec) register always starts with the CLINT-Direct mode encoding. During processor runtime, you must reconfigure the interrupt controller based on the selected **Interrupt Mode** in Platform Designer. This is done by writing the mtvec register with the appropriate **Interrupt Mode** encoding.

Note: This mtvec reconfiguration is included in the Altera HAL firmware.

9.2.1. CLINT-Direct Mode

All traps (both exceptions and interrupts) are dispatched through a single top-level trap handling code. In other words, all traps are handled by code residing at a single location - the exception vector.

The CLINT-Direct is a simple, non-vectored interrupt controller. Upon receipt of a trap request, the CLINT-Direct :

1. Transfers control to exception vector.
2. Identifies which IRQ is currently asserted.
3. Allows trap handling code to mask other traps.

To complement CLINT-Direct, the trap handling code identifies the trap cause and dispatches the registered ISR. CLINT-Direct has 16 independent platform interrupt signals. These interrupt signals allow software to prioritize interrupts, although the interrupt signals themselves have no inherent priority from a hardware perspective

Table 38. Supported Traps

Type of Traps	List of Traps
Interrupt	• Machine software interrupt • Machine timer interrupt • 16 Platform interrupt
Exception	• Instruction address misaligned • Instruction access fault • Illegal Instruction • Breakpoint • Load address misaligned • Load access fault • Store address misaligned • Store access fault • Hardware Error

9.2.1.1. How the Hardware Works

The Nios V processor can respond to traps including platform interrupts, timer interrupt, software interrupts and software exceptions. When the Nios V processor responds to a trap, it performs the following tasks:

- Disables interrupts by clearing `mstatus.mie` and saves the previous value to `mstatus.mpie`.
- Saves the next execution address in Machine Exception Program Counter (mepc).
- Saves the cause of the trap in Machine Cause register (`mcause`).
- Switch to shadow register set through Shadow Register File Status register (`msrfstatus`) if enabled.
- Transfers control to the trap handling system (vector base address), held in the Machine Trap-Vector Base-Address (mtvec) register.
- The same transfer of control is applied to all types of traps – interrupts and exceptions.

Nios V traps are not vectored. Therefore, the same vector base address receives control for all types of traps. At the vector base address, the trap handling code must determine the type of trap (i.e. software exception, platform interrupt, timer interrupt or software interrupt), and assign the relevant handler.

All Nios V processor trap types are precise. This means that after a trap is handled, the Nios V processor can re-execute the instruction that caused the trap.

Related Information

Nios® V Processor Reference Manual
For more information about Machine Trap Handling CSRs behaviour, refer to the Nios® V Processor Reference Manual.

9.2.1.2. How the Software Works

The trap handling system implementation is one of many possible implementations of a trap handling system for the Nios V processor. Some features of the HAL trap handling system are constrained by the Nios V hardware, while others provide generally useful services.

You can take advantage of the HAL trap handling system without a complete understanding of the HAL implementation.

For more information about how to install ISRs using the HAL API, refer to *Interrupt Service Routines and Exception Handler*.

Related Information

Interrupt Service Routines and Exception Handler on page 170

9.2.1.2.1. Trap Handling System Structure

The trap handling system consists of the following components:

- The trap handling code
- The interrupt dispatcher
  - ISR for timer interrupts
  - ISR for software interrupts
  - ISR for each peripheral that generates platform interrupts
- The exception dispatcher
  - Handler for instruction-related exception

With the CLINT-Direct, all traps start from the same trap handling code. The trap handling code manages the processor context switch and RTOS overhead (if any). Next, it determines the source of the trap, and executes the correct dispatcher:

- If the trap is an interrupt, the interrupt dispatcher is executed.
- If the trap is an exception, the exception dispatcher is executed.

The interrupt dispatcher passes control to the relevant ISRs, while the exception dispatcher passes control to the correct exception handler.

The following files provides the HAL API for the trap handling system:

- `<Project directory>\software\bsp\HAL\src\intel_niosv_irq.c`
- `<Project directory>\software\bsp\HAL\src\machine_trap.S`
- `<Project directory>\software\bsp\HAL\src\alt_irq_handler.c`

Figure 17. HAL Trap Handling System in CLINT-Direct

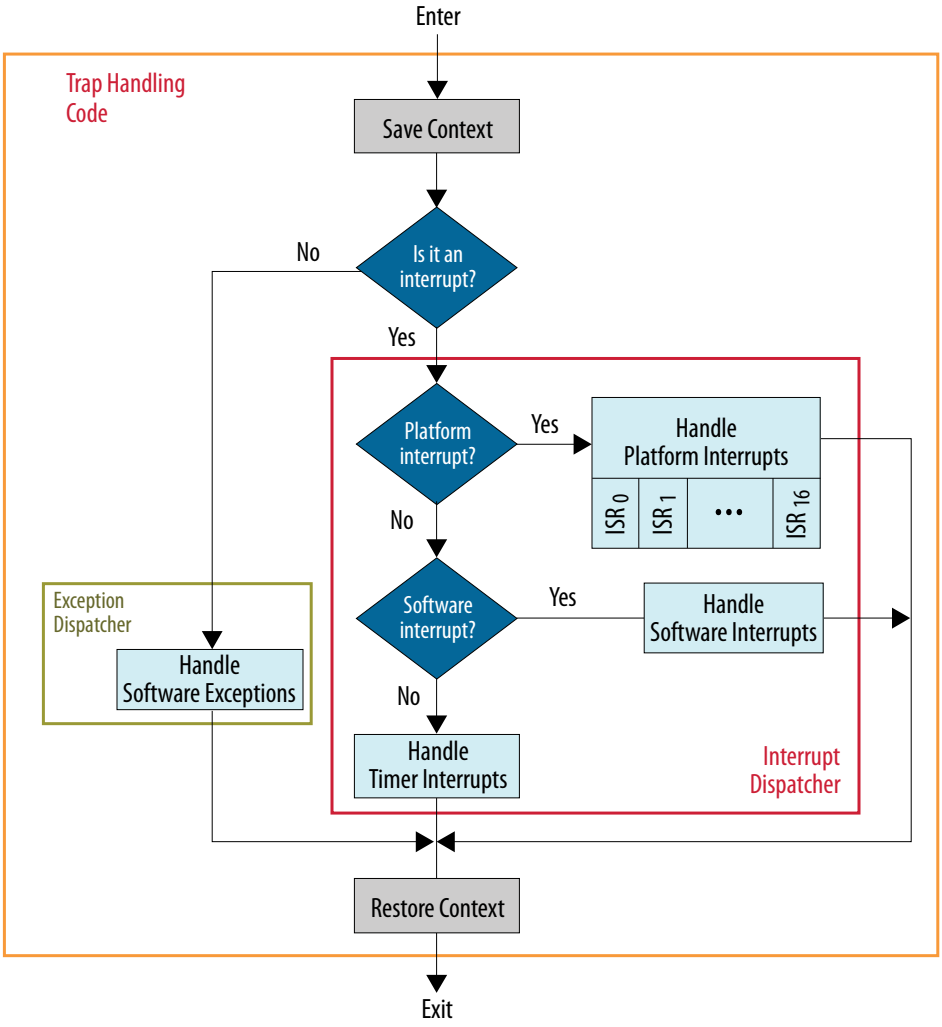


Table 39. CLINT-Direct Trap Handling

Traps	Address Computation	Action Taken
All Exceptions	pc <= Exception Vector	Enters trap handling code
Machine software interrupt		
Machine timer interrupt		

continued...

Traps	Address Computation	Action Taken
Platform interrupt 0		
...		
Platform interrupt 16		

9.2.1.2.2. Trap Handling Code

The trap handling code provided with the HAL is located at the Nios V processor's exception address (vector base address), held in the Machine Trap-Vector Base-Address (mtvec) register.

Traps Occurs

When a trap occurs, and the processor control transfers to the trap handling code, it does the following:

- Switches to the separate exception stack (if enabled)
- Stores register values onto the stack
  - Stores all register values if shadow register is turned off.
  - Store the lower 16 general-purpose register values only if shadow register is enabled. (This is not implemented in Altera HAL firmware.)
- Determines the type of traps (either exceptions or interrupts)
- Passes control to the correct dispatchers

The trap handling code looks at the Machine Cause (mcause) register to determine the trap type.

- If the Interrupt bit is set, the trap handling code transfer control to the interrupt dispatcher.
- Otherwise (Interrupt bit is not set), the trap handling code calls the exception dispatcher.

Return from Traps

Once a trap is serviced, and the ISR or exception handler returns to the trap handling code, it performs the following tasks:

- Restores the registers from the stack.
  - Restores all the registers if the shadow register is disabled.
  - Restores the lower 16 general-purpose registers only if the shadow register is enabled. (This is not implemented in Altera HAL firmware.)
- Restores the stack pointer (if a separate exception stack is used).
- Exits by issuing an MRET (exception return in machine mode) instruction.

9.2.1.2.3. Interrupt Dispatcher

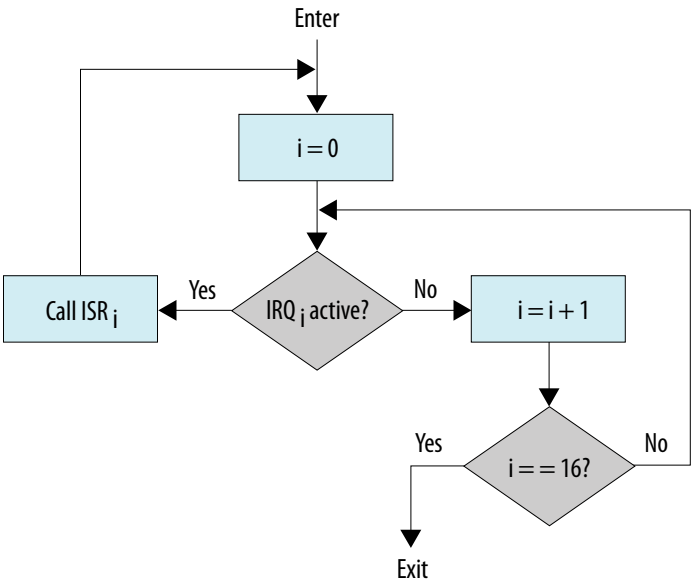
The interrupt dispatcher checks the interrupt bits in mip register. If any interrupts are pending, it calls the appropriate ISR.

Interrupt prioritization is as follows:

- All platform interrupts have higher priority than timer and software interrupts.
- Among the 16 platform interrupts, platform interrupt 0 has the highest priority, and 15 the lowest.
- This prioritization is inherent in the CLINT.

The interrupts dispatcher goes through the platform IRQs in mip register starting at platform IRQ_0, and finds the first (highest priority) active platform IRQ. Then, the interrupts calls the corresponding registered ISR. After this ISR executes, the dispatcher begins scanning the platform IRQs again, starting at platform IRQ_0. In this way, higher-priority interrupts are always processed before lower-priority interrupts. With all interrupts request fulfilled, the interrupt dispatcher returns to the trap handling code.

Figure 18. How Interrupt Dispatcher Handles Platform Interrupt



For more information about the Nios V processor mip registers, refer to the *Nios V Processor Reference Manual.*

For more information on registering an ISR, refer to the *Interrupt Service Routines and Exception Handler* chapter.

Related Information

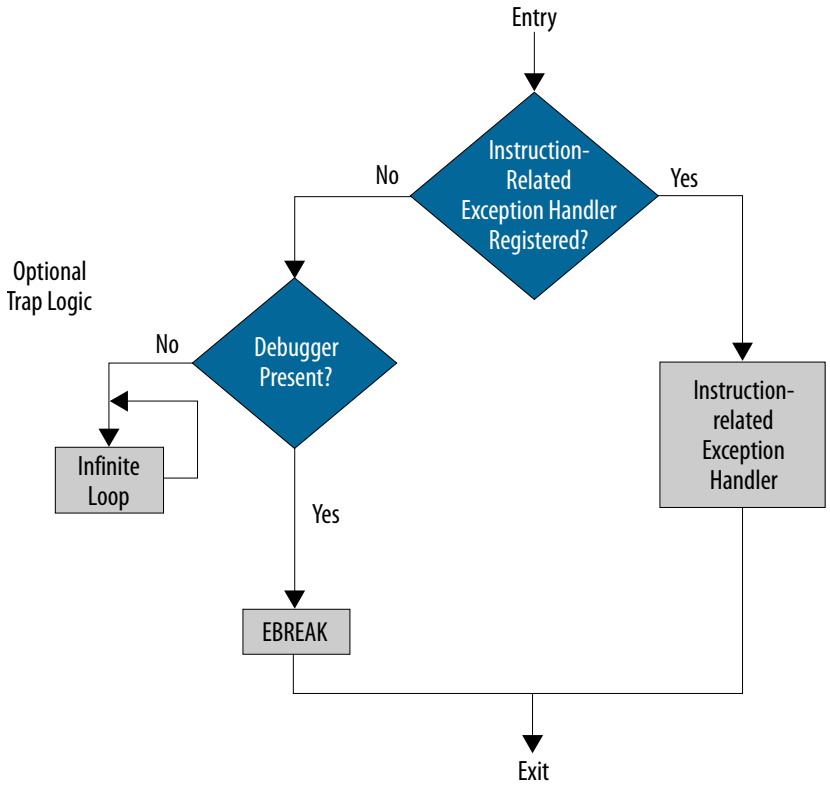
- Nios® V Processor Reference Manual
  For more information about Machine Trap Handling CSRs behaviour, refer to the Nios® V Processor Reference Manual.
- Interrupt Service Routines and Exception Handler on page 170

9.2.1.2.4. Exception Dispatcher

The exceptions dispatcher checks for a registered instruction-related exception handler.

- If an instruction-related exception handler is registered, the HAL software exception funnel calls it, then restores context and returns.
- If no instruction-related exception handler is registered,
  - If processor debug module is enabled, the processor enters Debug Mode using the ebreak instruction. Allowing the debugger to take control.
  - If there is no debug module, the processor enters an infinite loop.

Figure 19. Exception Dispatcher



For more information on registering an instruction-related exception handler, refer to the *Instruction-Related Exception Handler* chapter.

For more information about the RISCV-based processor EBREAK instruction, refer to the *The RISC-V Instruction Set Manual in RISC-V International* website.

Related Information

- The RISC-V Instruction Set Manual
- Interrupt Service Routines and Exception Handler on page 170
- Instruction-Related Exception Handler on page 178

9.2.2. CLINT-Vectored Mode

Different from CLINT-Direct, CLINT-Vectored enables the processor to work with a dedicated interrupt vector for every interrupt.

- Interrupts are handled separately from software exceptions. Each interrupt has its own dedicated vector, tied to its ISRs. Interrupt vectors are calculated from the base address of the vector table and the interrupt cause number.
- Software exception handling is the same as with the CLINT-Direct. All exceptions are handled by an exception dispatcher residing at a single location - the base address of the vector table.

The CLINT-Vectored is a simple, vectored interrupt controller. Upon receipt of a trap request, the CLINT-Vectored:

1. Transfers control to the exception vector (for all exceptions) or the individual interrupt vector (specific for each interrupt).
2. Allows exception (or interrupt) handling code to mask other traps.

The CLINT-Vectored carries out the role of dispatching handler (identifies the trap cause and dispatches the registered ISR). Thus, a simpler trap handling system is implemented under CLINT-Vectored. A total of 16 independent platform interrupt signals are available in CLINT-Vectored.

Table 40. Supported Traps

Type of Traps	List of Traps
Interrupt	• Machine software interrupt • Machine timer interrupt • 16 Platform interrupt
Exception	• Instruction address misaligned • Instruction access fault • Illegal Instruction • Breakpoint • Load address misaligned • Load access fault • Store address misaligned • Store access fault • Hardware Error

9.2.2.1. How the Hardware Works

The Nios V processor can respond to traps including hardware interrupts, timer interrupt, software interrupts and software exceptions. When the Nios V processor responds to a trap, it performs the following tasks:

- Disables interrupts by clearing mstatus.mie and saves the previous value to mstatus.mpie.
- Saves the next execution address in Machine Exception Program Counter (mepc).
- Saves the cause of the trap in the Machine Cause register (`mcause`).

- Switch to shadow register set through Shadow Register File Status register (`msrfstatus`) if enabled.
- For exception – Transfers control to the exception dispatcher. The vector table base address (held in the mtvec register) contains the jump instruction to the exception dispatcher.
- For interrupt – Transfer control to the interrupt ISR. Each interrupt vector contains the jump instruction to the individual ISRs. The processor calculates the interrupt vector, by adding vector table base address to four times the interrupt cause number.

Nios V traps are vectored. It is implemented using a vector table, that holds jump instruction to the associated ISR. The interrupt cause code determines which jump instruction is executed.

All Nios V processor trap types are precise. This means that after a trap is handled, the Nios Vprocessor can re-execute the instruction that caused the trap.

For more information about Machine Trap Handling CSRs behaviour, refer to the *Processor Reference Manual*.

Related Information

Nios® V Processor Reference Manual
For more information about Machine Trap Handling CSRs behaviour, refer to the Nios® V Processor Reference Manual.

9.2.2.2. How the Software Works

The trap handling system implementation is one of many possible implementations of a trap handling system for the Nios V processor. Some features of the HAL trap handling system are constrained by the Nios V processor hardware, while others provide generally useful services.

You can take advantage of the HAL trap handling system without a complete understanding of the HAL implementation.

For more information about how to install ISRs using the HAL API, refer to the *Interrupt Service Routines and Exception Handler*.

Related Information

Interrupt Service Routines and Exception Handler on page 170

9.2.2.2.1. Trap Handling System Structure

The CLINT-Vectored processor logic takes up the role of dispatching handler (identifies the trap cause and dispatches the registered ISR). Thus, trap handling code and interrupt dispatcher is not required for this implementation, simplifying the trap handling system.

The simplified trap handling system consists of the following components:

- ISR for timer interrupts
- ISR for software interrupts
- ISR for each peripheral that generates hardware interrupts
- The exception dispatcher
  - Handler for instruction-related exception

With the CLINT-Vectored, each interrupt is directly dispatched to its respective ISR. Only exceptions start from the same exception dispatcher. The exception dispatcher and each ISR need to manage the processor context switch and RTOS overhead (if any), respectively.

The following files provides the HAL API for the trap handling system:

- `<Project directory>\software\bsp\HAL\src\intel_niosv_irq.c`
- `<Project directory>\software\bsp\HAL\src\alt_vic_vector_table.S`
- `<Project directory>\software\bsp\HAL\src\alt_vic_isr_wrapper.c`

Figure 20. HAL Trap Handling System in CLINT-Vectored

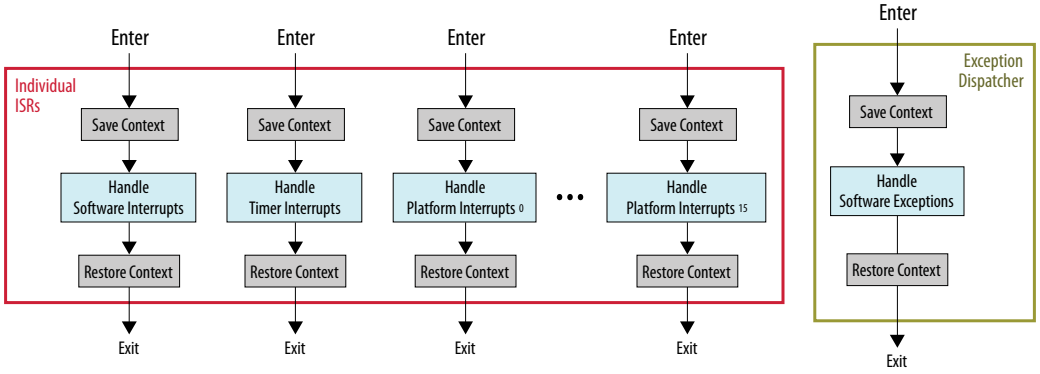


Table 41. CLINT-Vectored Trap Handling

Traps	Address Computation	Instruction	Action Taken
All Exceptions	pc <= Vector table	jal zero, alt_vic_exception_handler	Enters exception dispatcher
Machine software interrupt	pc <= Vector table + 12	jal zero, alt_vic_msw_isr	Enters machine software ISR
Machine timer interrupt	pc <= Vector table + 28	jal zero, alt_vic_mtimer_isr	Enters machine timer ISR
Platform interrupt 0	pc <= Vector table + 64	jal zero, alt_vic_platform_irq0	Enters platform ISR 0
…	…	…	…
Platform interrupt 16	pc <= Vector table + 128	jal zero, alt_vic_platform_irq15	Enters platform ISR 15

9.2.2.2.2. Interrupt Service Routine

Upon an interrupt, the CLINT-Vectored passes control to the ISR according to the interrupt vector (in vector table). The interrupt vector is calculated from the addition of the vector table base address with four times the interrupt cause number.

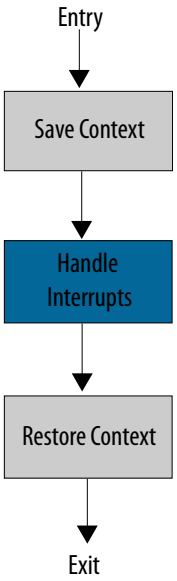
The interrupt performs the following tasks:

- Stores register values onto the stack
  - Stores all register values if the shadow register is turned off
  - Stores the lower 16 general-purpose registers only if the shadow register is enabled. (This is not implemented in Altera HAL firmware.)
- Execute the servicing tasks related to the interrupt

Once the ISR completes, it performs the following tasks:

- Restores the registers from the stack
  - Restores all the registers if shadow register is disabled
  - Restores the lower 16 general-purpose registers only if shadow register is enabled. (This is not implemented in Altera HAL firmware.)
- Exits by issuing an MRET (trap return in machine mode) instruction

Figure 21. ISR in CLINT-Vectored



For more information about registering an ISR, refer to the topic *Interrupt Service Routines and Exception Handler*.

Related Information

Interrupt Service Routines and Exception Handler on page 170

9.2.2.2.3. Exception Dispatcher

Upon an exception, the CLINT-Vectored passes control to the exception dispatcher in vector table. As stated in the RISC-V specification, the exception dispatcher is at the vector table base address.

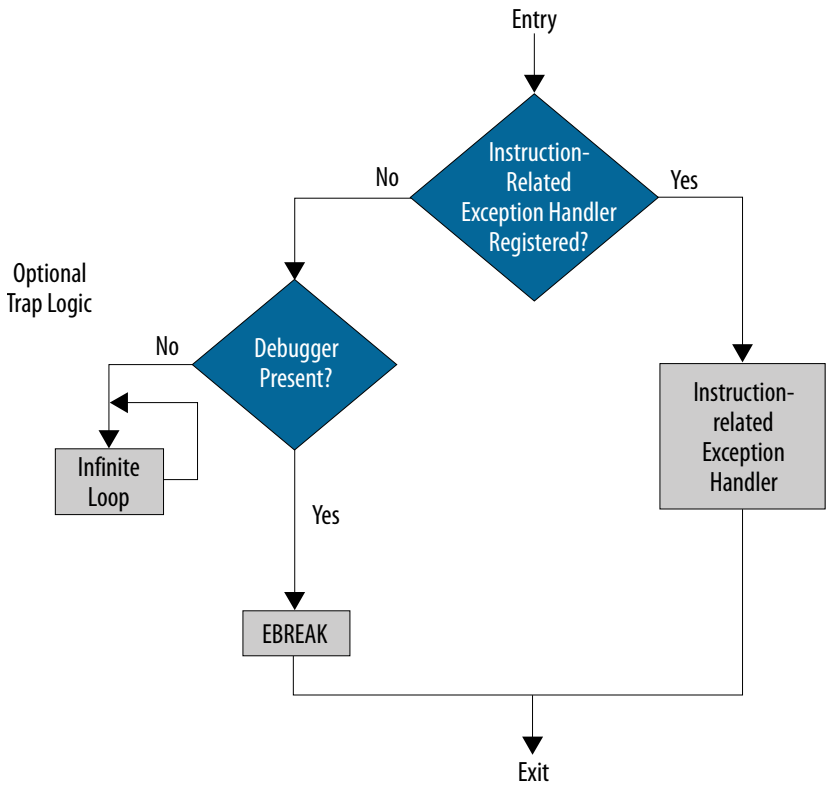
The exception dispatcher performs the following tasks:

- Switches to the separate exception stack (if enabled)
- Stores register values onto the stack
  - Stores all register values if the shadow register is disabled
  - Stores the lower 16 general-purpose registers only if the shadow register is enabled. (This is not implemented in Altera HAL firmware.)
- Checks for a registered instruction-related exception handler.
- If an instruction-related handler is registered, the exception dispatcher calls it, then restores context and returns.
- If no instruction-related exception handler is registered,
  - If processor debug module is enabled, the processor enters Debug Mode using the ebreak instruction. Allowing the debugger to take control.
  - If there is no debug module, the processor enters an infinite loop.

Once the exception is serviced, and the exception handler returns to the exception dispatcher, it performs the following tasks:

- Restores the registers from the stack
  - Restores all the registers if the shadow register is disabled
  - Restores the lower 16 general-purpose registers only if the shadow register is enabled. (This is not implemented in Altera HAL firmware.)
- Restores the stack pointer (if a separate exception stack is used)
- Exits by issuing an MRET (exception return in machine mode) instruction

Figure 22. Exception Dispatcher



For more information on registering an instruction-related exception handler, refer to the *Instruction-Related Exception Handle*r chapter.

For more information about the RISCV-based processor EBREAK instruction, refer to the *The RISC-V Instruction Set Manual* in RISC-V International website.

Related Information

Interrupt Service Routines and Exception Handler on page 170

9.2.3. CLIC Mode

Different from CLINT, a full-fledged CLIC trap processing logic provides the capabilities to handle interrupts as either non-vectored interrupts or vectored interrupts. It allows users to define individual interrupts as either non-vectored interrupts or vectored interrupts by encoding them as software-vectored or hardware-vectored, respectively.

- Software-vectored interrupts and exceptions are handled by the trap handling code residing at a single location – the vector base address in mtvec register.
- Hardware-vectored interrupts are handled differently. Each interrupt has its own dedicated vector, tied to its ISRs. Interrupt vectors are calculated from the base address of the vector table in mtvt register and the interrupt cause number.

The Nios V CLIC trap processing logic supports software-vectored interrupt only. Hardware-vectored interrupts are not supported.

The CLIC is a configurable interrupt controller on the following aspects :

- Pre-emption – Interrupt Level and Priority
- Polarity – Positive or Negative
- Trigger Type – Level or Edge
- Selective Vectoring – Not supported in Nios V processor. All interrupts are software-vectored.

By default, every interrupt starts at the same interrupt level (no pre-emption), positive level-triggered, and software-vectored after the processor is out of reset. You have to configure the configurable aspects thru the `clicintctl` and `clicintattr` CLIC registers.

Upon receipt of a trap request, the CLIC :

1. Select the interrupt with the highest interrupt level.
2. Transfer control to vector base address in mtvec register.
3. Allow the trap handling code to handle the traps.

To complement CLIC, the trap handling code identifies the trap cause and dispatches the registered ISR. CLIC can support up to 2048 independent platform interrupt signals.

Table 42. Supported Traps

Type of Traps	List of Traps
Interrupts (Only as software-vectored)	• Machine software interrupt • Machine timer interrupt • 16 - 2048 Platform interrupts
Exceptions	• Instruction address misaligned • Instruction access fault • Illegal Instruction • Breakpoint • Load address misaligned • Load access fault • Store address misaligned • Store access fault • Hardware Error

9.2.3.1. How the Hardware Works

When the Nios V processor responds to a software-vectored interrupt or exception, it performs the following tasks:

- Disables interrupts by clearing `mstatus.mie` and saves the previous value to `mstatus.mpie`.
- Saves the next execution address in Machine Exception Program Counter (`mepc`).
- Saves the cause of the trap in Machine Cause register (`mcause`).
- Saves the active interrupt level in Machine Interrupt Status Register (`mintstatus`).
- Switch to shadow register set through Shadow Register File Status register (`msrfstatus`) if enabled.

- Transfers control to the trap handling system (vector base address), held in the Machine Trap-Vector Base-Address (`mtvec`) register.
- The same transfer of control is applied to all types of traps – interrupts and exceptions.

Software-vectored interrupts are not vectored. Therefore, the same vector base address receives control for all software-vectored interrupts and exception.

At the vector base address, the trap handling code must support pre-emptive interrupt by saving CSRs – `msrfstatus`, `mepc` and `mcause` before reenabling interrupts (setting `mstatus.mie`). If there is no pre-emptive interrupt, the trap handling code proceeds to determine the type of trap (i.e. exception, platform interrupt, timer interrupt or software interrupt), and assign the relevant handler.

All Nios V processor trap types are precise. This means that after a trap is handled, the Nios V processor can re-execute the instruction that caused the trap.

9.2.3.2. How the Software Works

Before implementing the CLIC, it is required to limit the alignment of mtvec to 64-byte boundaries for processor system. Alternatively, assign the .exception linker section to a memory region that fulfils the following conditions:

- Start address is more than 0x40, and
- Start address is a multiple of 0x10.

For example, 0x40, 0x50, 0x60 and etc are suitable starting addresses.

Figure 23. Exception Linker Section for CLIC

Linker Section Mappings

Linker Section Name	Linker Region Name	Memory Device Name
.bss	intel_onchip_memory	intel_onchip_memory_0
.entry	reset	intel_onchip_memory_0
.exceptions	intel_onchip_memory	intel_onchip_memory_0
.heap	intel_onchip_memory	intel_onchip_memory_0
.rodata	intel_onchip_memory	intel_onchip_memory_0
.rwdata	intel_onchip_memory	intel_onchip_memory_0
.stack	intel_onchip_memory	intel_onchip_memory_0
.text	intel_onchip_memory	intel_onchip_memory_0

Linker Memory Regions

Linker Region Name	Address Range	Memory Device Name	Size (bytes)	Offset (bytes)
intel_onchip_memory	0x00000040 - 0x0007FFFF	intel_onchip_memory_0	524224	64
clic_unaccessible	0x00000020 - 0x0000003F	intel_onchip_memory_0	32	32
reset	0x00000000 - 0x0000001F	intel_onchip_memory_0	32	0

The Altera HAL firmware does not support CLIC mode, even though Platform Designer has selected CLIC as the preferred **Interrupt Mode**. For more information, refer to the *Core-Local Interrupt Controller (CLIC) RISC-V Privileged Architecture Extensions*.

Related Information

Core-Local Interrupt Controller (CLIC) RISC-V Privileged Architecture Extensions
For more information about the software implementation, refer to the Appendix.

9.3. Interrupt Service Routines and Exception Handler

9.3.1. Platform Interrupt Service Routines

Software often communicates with peripheral devices using platform interrupts. When a peripheral asserts its IRQ, it diverts the processor's normal execution flow. When such an interrupt occurs, an appropriate ISR must handle this interrupt and return the processor to its pre-interrupt state on completion.

When you create a board support package (BSP) project, the build tools include all needed device drivers. You do not need to write HAL ISRs unless you are interfacing to a custom peripheral. For reference purposes, this section describes the framework provided by HAL BSPs for handling platform interrupts.

For examples of HAL ISRs, refer to existing handlers for Intel components.Developing Programs Using the Hardware Abstraction Layer section.

Related Information

alt_irq_enabled() on page 250

9.3.1.1. HAL APIs for Platform Interrupts

Altera provides the Enhanced HAL Hardware Interrupt API for writing, registering, and managing platform ISRs. The Nios V processor supports up to 16 platform interrupts.

The interrupt APIs include the following types of routines:

- Routines to be called by a device driver to register an ISR
- Routines to be called by an ISR to manage its environment
- Routines to be called by BSP or application code to control ISR behavior

The interrupt APIs support the following types of BSPs:

- Bare-metal HAL BSP
- Micrium MicroC/OS-II (uC/OS-II) BSP
- FreeRTOS BSP

Note: The legacy platform interrupt API is deprecated and not supported in Nios V processor-based system.

9.3.1.1.1. Enhanced HAL Hardware Interrupt API

The BSP implements the Enhanced HAL Hardware Interrupt API, and each driver that supports the hardware interrupt API publishes this capability to the BSP Editor through its `<driver name>_sw.tcl` file.

A driver can publish its interrupt API support by way of a software property. The driver's `<driver name>_sw.tcl` file uses the `set_sw_property` command to set `supported_interrupt_apis` to `enhanced_interrupt_api`.

Using the Enhanced HAL Hardware Interrupt API to Implement ISRs

You need to perform the following steps to implement platform ISR with HAL hardware interrupt API:

1. Create your ISR functions prototype that handles platform interrupts.
2. Ensure your program registers the ISR with the HAL driver by calling `alt_ic_isr_register()`.
3. `alt_ic_isr_register()` enables platform interrupts for you.

Table 43. Enhanced HAL Hardware Interrupt API Functions

Function Name	Description
`alt_ic_isr_register()`	Register an interrupt handler to handle the platform interrupts.
`alt_ic_irq_enable()`	Enables a single interrupt.
`alt_ic_irq_disable()`	Disables a single interrupt.
`alt_ic_irq_enabled()`	Determine if corresponding interrupt is enabled.
`alt_irq_disable_all()`	Disable all interrupts and return context value correspond to the previous state.
`alt_irq_enable_all()`	Enable all interrupts to the previous state based on the returned context value from prior `alt_irq_disable_all()`.
`alt_irq_enabled()`	Determine if the processor's global interrupt enable is asserted.

Note: In the IIC, the functions are implemented by the HAL. For more information about each function, refer to the HAL API Reference section.

9.3.1.2. Writing a Platform ISR

The platform ISR you write must match the function prototype that `alt_ic_isr_register()` expects.

```
void (*alt_isr_func) (void* isr_context)
```

The parameter definition of `isr_context` is the same as for the `alt_ic_isr_register()` function.

From the point of view of the HAL exception handling system, the most important function of an ISR is to clear the associated peripheral's interrupt condition. The procedure for clearing a platform interrupt condition is specific to the peripheral.

When the ISR has finished servicing the platform interrupt, it returns to the HAL interrupt funnel that called it.

Note: If you write your ISR in assembly language, use `ret` to return. The HAL general exception funnel issues an MRET after restoring the application context.

Related Information

Embedded Peripherals IP User Guide

9.3.1.2.1. Using General Exception Funnels

The funnel code receives control from the general exception vector. The funnel performs tasks such as saving the stack pointer, saving registers and calling RTOS context-switch routines, and transfers control to the handler. When the handler returns, the funnel code performs tasks such as calling RTOS process-dispatch routines and restoring registers, and transfers control to the appropriate foreground task.

9.3.1.2.2. Running in a Restricted Environment

ISRs run in a restricted environment. Most of the HAL API calls are not available from ISRs. For example, accesses to the HAL file system are not permitted. As a general rule, when writing your own ISR, never include function calls that can block for any reason (such as waiting for a platform interrupt).

For more information about identifying these API functions that are not available to ISRs, refer to the HAL API Reference chapter.

Be careful when calling ANSI C standard library functions inside of an ISR. Avoid using the C standard library I/O API, because calling these functions can result in deadlock within the system, that is, the system can become permanently blocked in the ISR.

In particular, do not call `printf()` from within an ISR unless you are certain that `stdout` is mapped to a non-interrupt-based device driver. Otherwise, `printf()` can deadlock the system, waiting for a platform interrupt that never occurs because interrupts are disabled.

9.3.1.3. Registering a Platform ISR

Before the software can use an ISR, you must register it by calling `alt_ic_isr_register()`. The prototype for `alt_ic_isr_register()` is:

```
int alt_ic_isr_register(alt_u32 ic_id, alt_u32 irq, alt_isr_func isr, void
*isr_context, void* flags)
```

The function has the following parameters:

- `ic_id` is the interrupt controller identifier (ID) as defined in `system.h`.
  - With the IIC, `ic_id` is not significant.
- `irq` is the platform interrupt number for the device, as defined in `system.h`.
  - For the IIC, `irq` is the IRQ number. Interrupt priority corresponds inversely to the IRQ number. Therefore, IRQ_0 represents the highest priority interrupt and IRQ_{16} is the lowest.
- `isr` is a pointer to the ISR function that is called in response to IRQ number `irq`.
  - The ISR function prototype is:

    ```
    void (*alt_isr_func) (void* isr_context);
    ```

    The input argument provided to this function is the `isr_context`.

    Note: Registering a null pointer for `isr` results in the interrupt being disabled.
- `isr_context`:
  - Points to a data structure associated with the device driver instance.
  - Passed as the input argument to the ISR function. It is used to pass context-specific information to the ISR, and can point to any ISR-specific information. The context value is opaque to the HAL; it is provided entirely for the benefit of the user-defined ISR.
- `flags` is reserved.

9.3.1.3.1. Methods the HAL Uses to Register the ISR

The HAL registers the ISR by storing the function pointer (`isr`), in a lookup table.

The return code from `alt_ic_isr_register()` is zero if the function succeeded, and nonzero if it failed. If the HAL registers your ISR successfully, the associated Nios V processor platform interrupt (as defined by `irq`) is enabled on return from `alt_ic_isr_register()`. Each Nios V processor platform interrupt can register a single ISR only, and changing ISR on run-time is not supported.

For more information about `alt_ic_isr_register()`, refer to the HAL API Reference chapter.

Note: Registering the ISR can require hardware-specific initialization. The HAL code ensures that the registered ISR is correctly dispatched when a specific interrupt occurs. For more information and details about platform interrupt initialization specific to your peripheral, refer to the relevant chapter of the *Embedded Peripherals IP User Guide*.

Related Information

Embedded Peripherals IP User Guide

9.3.1.4. Enabling and Disabling Interrupts

The HAL enhanced interrupt API provides the functions:

- `alt_ic_irq_disable()`, `alt_ic_irq_enable()` allow you to to disable and enable individual interrupts.
- `alt_irq_disable_all()` disables all interrupts, and returns a context value.
- Call `alt_irq_enable_all()` pass in the context parameter to re-enable platform interrupts.. The interrupts are returned to their state prior to the call to `alt_irq_disable_all()`.
- `alt_irq_enabled()` returns nonzero if maskable exceptions are enabled. `alt_ic_irq_enabled()` determines whether a specified interrupt is enabled.

Note: Disable interrupts for as short a time as possible. Maximum interrupt latency increases with the longest amount of time interrupts are disabled. For more information about disabled interrupts, refer to the Keep Interrupts Enabled chapter.

Related Information

HAL API Reference on page 224
For more information about the API functions.

9.3.1.5. C Example

9.3.1.5.1. Writing a Platform ISR to Service a Button PIO Interrupt

This example is based on a Nios V processor system with a 4-bit PIO peripheral connected to push buttons. An IRQ is generated any time a button is pushed. The ISR code reads the PIO peripheral's edge capture register and stores the value to a global variable. The address of the global variable is passed to the ISR in the context pointer.

Example 25. Writing a Platform ISR to Service a Button PIO Interrupt

```
#include "system.h"

#include "altera_avalon_pio_regs.h"

#include "alt_types.h"

static void handle_button_interrupts(void * context) {
  /* Cast context to edge_capture's type. It is important that this
  be declared volatile to avoid unwanted compiler optimization. */
  volatile int * edge_capture_ptr = (volatile int * ) context;

  /* Read the edge capture register on the button PIO. Store value.*/
  * edge_capture_ptr = IORD_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE);

  /* Write to the edge capture register to reset it. */
  IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE, 0);

  /* Read the PIO to delay ISR exit. This is done to prevent a
  spurious interrupt in systems with high processor -> pio
  latency and fast interrupts. */
  IORD_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE);
}
```

9.3.1.5.2. Registering the Button PIO ISR with the HAL

Based on the code in the example, the following execution flow is possible:

- Button is pressed, generating an IRQ.
- The ISR gains control.
  - With the IIC, the HAL general exception funnel gains control of the processor, and dispatches the `handle_button_interrupts()` ISR.
- `handle_button_interrupts()` services the hardware interrupt and returns.
- Normal program operation continues with an updated value of `edge_capture`.

Example 26. Registering the Button PIO ISR with the HAL

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

#include "system.h"

...
/* Declare a global variable to hold the edge capture value. */
volatile int edge_capture;
...
/* Initialize the button_pio. */
static void init_button_pio()
{
    /* Recast the edge_capture pointer to match the
    alt_irq_register() function prototype. */
    void * edge_capture_ptr = (void * ) & edge_capture;
    /* Enable all 4 button interrupts. */
    IOWR_ALTERA_AVALON_PIO_IRQ_MASK(BUTTON_PIO_BASE, 0xf);
    /* Reset the edge capture register. */
    IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE, 0x0);
    /* Register the ISR. */
    alt_ic_isr_register(BUTTON_PIO_IRQ_INTERRUPT_CONTROLLER_ID,
      BUTTON_PIO_IRQ,
      handle_button_interrupts,
      edge_capture_ptr, 0x0);
}
```

9.3.2. Software Interrupt Service Routines

When software asserts its IRQ, it diverts the processor's normal execution flow. When such an interrupt occurs, an appropriate ISR must handle this interrupt and return the processor to its pre-interrupt state on completion.

9.3.2.1. HAL APIs for Software Interrupt

Intel provides the HAL Software Interrupt API for writing, registering, and managing ISRs. The Nios V processor supports one software interrupt only.

The interrupt API includes the following types of routines:

- Routines to register an ISR
- Routines to manage ISR environment
- Routines to control ISR behavior

The interrupt API supports the following types of BSPs:

- Bare-metal HAL BSP
- Micrium MicroC/OS-II (uC/OS-II) BSP
- FreeRTOS BSP

You need to perform the following steps to implement software ISR with HAL software interrupt API:

1. Create your ISR functions prototype that handles software interrupt.
2. Ensure your program registers the ISR with the HAL driver by calling `alt_niosv_register_msw_interrupt_handler()`.
3. Enable software interrupt with `alt_niosv_enable_msw_interrupt()`.

Table 44. HAL Software Interrupt API Functions

Function Name	Description
`alt_niosv_register_msw_interrupt handler()`	Register an interrupt handler to handle software interrupt.
`alt_niosv_enable_msw_interrupt()`	Enables software interrupt.
`alt_niosv_disable_msw_interrupt()`	Disables software interrupt.
`alt_niosv_is_msw_interrupt_enabled()`	Determine if the software interrupt is enabled.
`alt_niosv_trigger_msw_interrupt()`	Generate a software IRQ.
`alt_niosv_clear_msw_interrupt()`	Clear the current software IRQ

9.3.2.2. Writing a Software ISR

The ISR create must match the function prototype that `alt_niosv_register_msw_interrupt handler()` expected.

```
void (*sw_isr_func) (alt_u32 cause, alt_u32 epc, alt_u32 tval)
{
    /* You may declare the ISR code here. */
    alt_niosv_clear_msw_interrupt();
}
```

After the software interrupt is handled, you are required to clear the software interrupt by using the `alt_niosv_clear_msw_interrupt()` function.

Note: If you write your ISR in assembly language, use ret to return. The HAL general exception funnel issues an MRET after restoring the application context.

Related Information

Embedded Peripherals IP User Guide

9.3.2.3. Registering a Software ISR

Before software can use an ISR, you must register it by calling `alt_niosv_register_msw_interrupt_handler()`. The prototype for `alt_niosv_register_msw_interrupt_handler()` is:

```
void alt_niosv_register_msw_interrupt_handler(alt_niosv_sw_isr_t handle)
```

The function has the following parameters:

- `handle` is a pointer to the software ISR function.

  The ISR function prototype is:

  ```
  void (*sw_isr_func) (alt_u32 cause, alt_u32 epc, alt_u32 tval)
  ```

  For more information about `alt_niosv_register_msw_interrupt_handler()`, refer to the HAL API Reference chapter.

9.3.2.4. Enabling and Disabling Interrupts

The HAL software interrupt API provides the functions:

- `alt_niosv_disable_msw_interrupt()` and `alt_niosv_enable_msw_interrupt()` allow you to disable and enable software interrupt.
- `alt_niosv_is_msw_interrupt_enabled()` determines whether the software interrupt is enabled.
- `alt_niosv_trigger_msw_interrupt()` generates the software interrupt after enabling the software interrupt.

Note: Disable interrupts for as short a time as possible. Maximum interrupt latency increases with the longest amount of time interrupts are disabled. For more information about disabled interrupts, refer to the "Keep Interrupts Enabled" chapter.

For more information about these functions, refer to the "HAL API Reference" chapter.

9.3.2.5. C Example

9.3.2.5.1. Writing a Software ISR

This example is based on a Nios V processor-based system. A software IRQ is generated when running the `alt_niosv_trigger_msw_interrupt()` function. The ISR code increments a global variable and clears the software IRQ using the `alt_niosv_clear_msw_interrupt()`.

Example 27. Writing a Software ISR

```
#include "system.h"

#include "sys/msw_interrupt.h"

#include "alt_types.h"

void sw_isr(alt_u32 cause, alt_u32 epc, alt_u32 tval) {
  /* Increments the global variable. */
  ++sw_isr_cnt;
  /* Clears the software IRQ */
  alt_niosv_clear_msw_interrupt();
}
```

9.3.2.5.2. Registering the Software ISR with the HAL

Based on the code in the example, the following execution flow is possible:

- The software IRQ is generated.
- The software ISR gains control.
  - The HAL general exception funnel gains control of the processor, and dispatches the `sw_isr()` ISR.
- `sw_isr()` services the software interrupt, and returns to the normal program operation.
- Normal program operation continues with an updated value of `sw_isr_cnt`.

Example 28. Registering the Software ISR with the HAL

```
#include "system.h"

#include "sys/msw_interrupt.h"

#include "alt_types.h"

...
/* Declare a global variable to hold the incrementing value. */
volatile alt_u32 sw_isr_cnt = 0;
...
int main() {
  /* Register the ISR. */
  alt_niosv_register_msw_interrupt_handler(sw_isr);
  /* Enable software interrupt. */
  alt_niosv_enable_msw_interrupt();
  /* Generate the software interrupt. */
  alt_niosv_trigger_msw_interrupt();
}
```

9.3.3. Timer Interrupt Service Routine

The Nios V processor timer interrupt service routine manages the internal timer based on the RISC-V specification. The following registers represent the main components of the internal timer.

- mtime – Real-time counter which increments at a constant frequency.
- mtimecmp – Timer compares register for interrupt generation.

The processor timer ISR is essential for the critical timing routine within the processor operating system. Thus, Altera does not recommends for user-level applications. Alternatively, refer to *Developing Programs Using the Hardware Abstraction Layer – Using Timer Devices* for more information on developing user-level application using system clock, alarm, and timestamp.

Related Information

Using Timer Devices on page 96

9.3.4. Instruction-Related Exception Handler

The software exception funnel lets you handle instruction-related exceptions, such as the load address misaligned exceptions. The instruction-related exception handler is a custom handler. Your software registers the instruction-related exception handler with the HAL at startup time.

For more information about enabling instruction-related exception handlers, refer to Settings Managed by Nios V Processor Board Support Package Editor section.

Note: The `hal.enable_instruction_related_exceptions_api setting` must be enabled in the BSP in order for you to register an instruction-related exception handler.

9.3.4.1. Writing an Instruction-Related Exception Handler

The prototype for an instruction-related exception handler is as follows:

```
alt_exception_result handler(alt_exception_cause cause,
alt_u32 exception_pc, alt_u32 bad_addr);
```

The instruction-related exception handler's return value is a flag requesting that the HAL either re-execute the instruction or skip it.

The HAL exception funnel calls the instruction-related exception handler with the following arguments:

- `cause` — A value representing the exception type, as shown in the Nios V Processor Exception Cause Codes table
- `exception_pc` — Instruction address at which exception occurred
- `bad_addr` — Bad address associated with exception (if applicable)

Include the following header file in your instruction-related exception handler code:

```
#include “sys/alt_exceptions.h”
```

`alt_exceptions.h` provides type macro definitions required to interface your instruction-related exception handler to the HAL, including the cause codes shown in the Nios V Processor Exception Cause Codes table.

For more information about RISCV-based processor exception causes, refer to the *The RISC-V Instruction Set Manual Volume II* in *RISC-V International* website.

Related Information

The RISC-V Instruction Set Manual

9.3.4.1.1. Exception Cause Codes

Table 45. Nios V Processor Exception Cause Codes

Exception	Cause Code	Cause Symbol[2]	Possible Cause
Instruction address misaligned	0	`NIOSV_INSTRUCTION_ADDRESS_MISALIGNED`	• Incorrect jump or branch • Corrupted instruction stream • Faulty loader or linker • Hardware issues • Pointer errors • Compiler error
Instruction access fault	1	`NIOSV_INSTRUCTION_ACCESS_FAULT`	Unlikely to happen because Nios V processor does not support Physical Memory Protection (PMP).

continued...

(2) Cause symbols are defined in `sys/alt_exceptions.h`.

Exception	Cause Code	Cause Symbol[2]	Possible Cause
Illegal instruction	2	NIOSV_ILLEGAL_INSTRUCTION	• Corrupted code • Unsupported instruction • Jump to incorrect address • Linker error • Memory issues • Assembler/Compiler error • Programming error
Breakpoint	3	NIOSV_BREAKPOINT	• Debugging • Breakpoint instruction • Watchpoint or Conditional breakpoint • Software-Trigged breakpoint
Load address misaligned	4	NIOSV_LOAD_ADDRESS_MISALIGNED	• Pointer misuse • Data structure misalignment • Programming error • Compiler/Assembler error
Load access fault	5	NIOSV_LOAD_ACCESS_FAULT	Unlikely to happen because Nios V processor does not support Physical Memory Protection (PMP).
Store/AMO address misaligned	6	NIOSV_STORE_AMO_ADDRESS_MISALIGNED	• Pointer misuse • Data structure misalignment • Incorrect memory mapping • Programming error • Compiler/Assembler error
Store/AMO access fault	7	NIOSV_STORE_AMO_ACCESS_FAULT	Unlikely to happen because Nios V processor does not support Physical Memory Protection (PMP).
Environment call from U-mode	8	NIOSV_ENVIRONMENT_CALL_FROM_U_MODE	Unlikely to happen because Nios V processor does not support User mode.
Environment call from S-mode	9	NIOSV_ENVIRONMENT_CALL_FROM_S_MODE	Unlikely to happen because Nios V processor does not support Supervisor mode.
Reserved	10	NIOSV_RESERVED_BIT_10	—
Environment call from M-mode	11	NIOSV_ENVIRONMENT_CALL_FROM_M_MODE	• Custom system calls • Signaling an event
Instruction page fault	12	NIOSV_INSTRUCTION_PAGE_FAULT	Unlikely to happen because Nios V processor does not support virtual memory.
Load page fault	13	NIOSV_LOAD_PAGE_FAULT	Unlikely to happen because Nios V processor does not support virtual memory.
Reserved	14	NIOSV_RESERVED_BIT_14	—

continued...

(2) Cause symbols are defined in `sys/alt_exceptions.h`.

Exception	Cause Code	Cause Symbol[2]	Possible Cause
Store/AMO page fault	15	NIOSV_STORE_AMO_PAGE_FAULT	Unlikely to happen because Nios V processor does not support virtual memory.
Hardware error	19	NIOSV_ECC_EXCEPTION	ECC error in processor's memory blocks.
Cause unknown[3]	-1	NIOSV_UNDEFINED_CAUSE	—

Note: It is possible for an instruction-related exception to occur during execution of an ISR.

9.3.4.2. Registering an Instruction-Related Exception Handler

The HAL API function `alt_instruction_exception_register()` registers a single instruction-related exception handler. When you register an instruction-related exception handler, it replaces the default EBREAK.

The function prototype is as follows:

```
alt_instruction_exception_register
(
  alt_exception_result( * handler)
  (alt_exception_cause cause, alt_u32 exception_ pc,
    alt_u32 bad_addr)
);
```

The `handler` argument is a pointer to the instruction-related exception handler.

To use `alt_instruction_exception_register()`, include the following header file:

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

Note: The `hal.enable_instruction_related_exceptions_api` setting must be enabled in the BSP in order for you to register an instruction-related exception handler.

For more information, refer to Settings Managed by the Nios V Processor Board Support Package Editor section.

Note: Register the instruction-related exception handler as early as possible in function `main()`. This allows you to handle abnormal condition during startup. You can register an exception handler from the `alt_main()` function.

For more information about `alt_main()`, refer to Boot Sequence and Entry Point section.

(2) Cause symbols are defined in `sys/alt_exceptions.h`.

(3) This value is passed to the instruction-related exception handler if the cause argument if the cause is not known.

9.3.4.3. Removing an Instruction-Related Exception Handler

To remove a registered instruction-related exception handler, your C code must call the `alt_instruction_exception_register()` function, as follows:

```
alt_instruction_exception_register(null, null);
```

When the HAL removes the instruction-related exception handler, it restores back the default EBREAK.

9.3.4.4. Debugging for Instruction-Related Exception Handler

The JTAG UART functions in an interrupt-based scheme by default, which is more efficient in terms of CPU utilization. Due to this scheme, if the CPU core has entered a bad state soon after a call to `printf()`, the output from `printf()` may not be witnessed via the juart-terminal. For example, the code below may result in *Hello World* print not being seen in juart-terminal.

```
printf("Hello world.\n");
function_which_causes_exception();
```

9.3.4.4.1. Enable JTAG UART polling operation

To disable the interrupt mode, enable the polling mode to avoid the interrupt mode's exception. By enabling the polling operation, you can enable the instruction exception handler function as shown in Example of Instruction-Related Exception Handler. You can enable the polling driver can via niosv-bsp by executing the following command below.

```
niosv - bsp - u - e = "set_setting
  altera_avalon_jtag_uart_driver.enable_small_driver {1}"
bsp / settings.bsp
```

Note: Polling mode has reduced performance compared to the interrupt mode. Therefore, polling mode is only suitable for debugging or applications where size efficiency is more important than performance.

9.3.4.4.2. Example of Instruction-Related Exception Handler

Once the polling mode is enabled in the JTAG UART following the instruction from the previous section, the following steps can help identify the cause of non-interrupt related exceptions like load, store misaligned, or others.

1. In the `main.c` file, add the following lines to include the library and the following `instr_exception_handler()` function.

```
#include <inttypes.h>
#include <sys/alt_exceptions.h>
// Provides a way to register a custom instruction exception handler
```

```
alt_exception_result instr_exception_handler(alt_exception_cause cause,
    alt_u32 epc, alt_u32 tval)
{
    printf("Instruction exception!\n");
    printf(" * cause: %d\n", cause);
    printf(" * epc:   0x%" PRIx32 "\n", epc);
    printf(" * tval:  0x%" PRIx32 "\n", tval);
 while (1) {};
    return NIOSV_EXCEPTION_RETURN_REISSUE_INST; // Should not be reached.
}
```

2. Add the following section to register this new exception handling function in the `main()` function, right above the code which causes the exception.

```
nt main(void)
{
    alt_instruction_exception_register (instr_exception_handler);
    // Register custom instruction exception handler.

    printf("Hello world!\n");
    function_which_causes_the_exception();
   return 0;
}
```

9.3.4.4.3. Adding Custom Exception Handler

Once the polling mode is enabled in the JTAG UART following the instruction from the previous section, the following steps can help with seeing the cause of non-interrupt related exceptions like load or store misaligned, etc. In the `main.c` file, add the following lines to include the library and following `instr_exception_handler()` function.

```
#include <inttypes.h>

#include <sys/alt_exceptions.h> // Provides a way to register a custom
instruction exception handler

alt_exception_result instr_exception_handler(alt_exception_cause cause,
  alt_u32 epc, alt_u32 tval) {
  printf("Instruction exception!\n");
  printf(" * cause: %d\n", cause);
  printf(" * epc:   0x%"
    PRIx32 "\n", epc);
  printf(" * tval:  0x%"
    PRIx32 "\n", tval);
  while (1) {};
  return NIOSV_EXCEPTION_RETURN_REISSUE_INST; // Should not be reached.
}
```

Add the following section to register this new exception handling function in the `main()` function, right above the code which causes the exception.

```
int main(void) {
    alt_instruction_exception_register (instr_exception_handler);  // Register
custom instruction exception handler.

    printf("Hello world!\n");
```

```
    function_which_causes_the_exception();
   return 0;
}
```

9.4. Latency and Response Time

Trap latency, as defined in the previous section, is the time required for the hardware to respond to a trap. Response time, in contrast, is the time required to begin executing code specific to the exception cause, such as a particular ISR. Response time is the sum of trap latency and the time required for the HAL to carry out some or all of the following overhead tasks:

- Context save—Saving registers on the stack
- RTOS context switch—Calling context-switch function(s) if an RTOS is implemented
- Dispatch handler—Determining the cause of the exception, and transferring control to a specific handler or ISR

If you are concerned with system performance, response time is the more important than latency, because it reflects the time elapsed between the physical event and the system's specific response to that event.

9.4.1. Direct or Vectored Mode

Both implementations serve the same purpose in servicing traps, but with different interrupt response time:

- CLINT-Direct is non-vectored, requiring the processor to dispatch ISRs with a software routine.
- CLINT-Vectored, by contrast, can be vectored. ISR dispatch is managed by hardware, eliminating the processing time required for ISR dispatch, and substantially reducing interrupt response time. This is achieved through a minor increase in logic utilization.

The CLINT-Vectored has no impact on software exception latency or response time.

9.5. Improving Nios V Processor ISR Performance

If your software uses interrupts extensively, the performance of ISRs has high impact to overall software performance.

9.5.1. Software Performance Improvements

When seeking to improve the efficiency and performance of Nios V processor applications, consider software changes first. In some cases it might require less effort to implement software design changes that increase system efficiency. The following sections describe changes you can make in the software design to improve ISR performance

9.5.1.1. Execute Time-Intensive Algorithms in the Application Context

ISRs provide rapid, low latency response to changes in the state of hardware. They do the minimum necessary work to clear the interrupt condition and then return. If your ISR performs lengthy, noncritical processing, it can interfere with more critical tasks in the system.

If your ISR requires lengthy processing, design your software to perform this processing outside of the exception context. The ISR can use a message-passing mechanism to notify the application code to perform the lengthy processing tasks.

Deferring a task is simple in systems based on an RTOS such as MicroC/OS-II. In this case, you can create a thread to handle the processor-intensive operation, and the ISR can communicate with this thread using any of the RTOS communication mechanisms, such as event flags or message queues.

You can emulate this approach in a single-threaded HAL-based system. The main program polls a global variable managed by the ISR to determine whether it needs to perform the processor-intensive operation.

9.5.1.2. Implement Time-Intensive Algorithms in Hardware

Processor-intensive tasks must often transfer large amounts of data to and from peripherals. A general-purpose processor such as the Nios V processor is not the most efficient way to do this. Use direct memory access (DMA) hardware if it is available.

For more information about programming with DMA hardware, refer to Using DMA Devices.

9.5.1.3. Increase Buffer Size

If you are using DMA to transfer large data buffers, the buffer size can affect performance. Small buffers imply frequent interrupts, which lead to high overhead.

Increase the size of the transaction data buffer(s).

9.5.1.4. Use Double Buffering

Using DMA to transfer large data buffers might not provide a large performance increase if the Nios V processor must wait for DMA transactions to complete before it can perform the next task.

Double buffering allows the Nios V processor to process one data buffer while the hardware is transferring data to or from another.

9.5.1.5. Keep Interrupts Enabled

When interrupts are disabled, the Nios V processor cannot respond to interrupt events. Buffers and queues can fill or overflow. Even in the absence of overflow, maximum interrupt processing time can increase after interrupts are re-enabled, because the ISRs must process data backlogs.

Disable interrupts as infrequently as possible, and for the briefest time possible.

Instead of disabling all interrupts, call `alt_ic_irq_disable()` and `alt_ic_irq_enable()` to enable and disable individual interrupts.

To protect shared data structures, use RTOS structures such as semaphores.

Disable all interrupts only during critical system operations. In the code where interrupts are disabled, perform only the bare minimum of critical operations, and reenable interrupts immediately.

9.5.1.6. Use Fast Memory

ISR performance depends on memory speed.

For best performance, place the ISRs and the stack in the fastest available memory: preferably tightly-coupled memory (if available), or on-chip memory.

If it is not possible to place the main stack in fast memory, consider using a separate exception stack, mapped to a fast memory section, as described in the next section.

For more information about mapping memory, refer to Memory Usage chapter.

9.5.1.7. Use a Separate Exception Stack

The HAL implements two types of separate exception stack. Using a separate exception stack entails a slight additional overhead.

When processing a exception or interrupt, the processor must execute additional instructions on entry and exit, to change the stack pointer. Take this additional processing time into account if your interrupt response requirements are extremely strict.

9.5.1.7.1. Separate General Exception Stack

Use the `hal.linker.enable_interrupt_stack` BSP setting to enable a separate general exception stack.

The HAL general exception funnel code takes care of correctly changing the stack pointer on entry to and exit from an exception handler.

Table 46. Separate Exception Stack Usage

Interrupt Controller	Separate General Exception Stack	Application Stack	General Exception Stack
Internal Interrupt Controller (IIC)	Not enabled	• Application • Exceptions • Platform interrupts • Timer interrupts • Software interrupt	—
	Enabled	• Application	• Exceptions • Platform interrupts • Timer interrupts • Software interrupt

9.5.1.8. Use Nested Interrupts

By default, the HAL disables interrupts when it dispatches an ISR. This means that only one ISR can execute at any time, and ISRs are executed on a first-come first-served basis. This reduces the system overhead associated with interrupt processing,

and simplifies ISR development. The ISR does not need to be reentrant. ISRs can use and modify any global or static data structures or hardware registers that are not shared with application code.

However, first-come first-served execution means that the HAL interrupt priorities only have an effect if two IRQs are active at the same time. A low-priority interrupt occurring before a higher-priority interrupt can prevent the higher-priority ISR from executing. This is a form of priority inversion, and it can have a significant impact on ISR performance in systems that generate frequent interrupts.

A software system can achieve full interrupt prioritization by using nested ISRs. With nested ISRs, higher-priority interrupts are allowed to interrupt lower-priority ISRs.

This technique can improve the response time for higher-priority interrupts. Nested interrupt technique is not supported in CLINT.

Note: Nested ISRs increase the processing time for lower-priority interrupts.

If your ISR is very short, it might not be worth the overhead to enable nested interrupts. Enabling nested interrupts for a short ISR can actually increase the response time for higher-priority interrupts.

Note: If you use a separate exception stack with the CLINT, you cannot nest interrupts.

9.5.1.9. Use Compiler Optimization

For the best performance both in exception context and application context, use compiler optimization level -O3. Level -O2 also produces good results. Removing optimization altogether significantly increases exception response time.

9.5.2. Hardware Performance Improvements

Several simple hardware changes can provide a substantial improvement in ISR performance. These changes involve editing and regenerating the hardware component, and recompiling the Intel FPGA design.

In some cases, these hardware changes also require changes in the software architecture or implementation.

The following sections describe changes you can make in the hardware design to improve ISR performance.

9.5.2.1. Add Fast Memory

Increase the amount of fast on-chip memory available for data buffers.

9.5.2.2. Add a DMA Controller

A DMA controller performs bulk data transfers, reading data from a source address range and writing the data to a different address range. Add DMA controllers to move large data buffers. This allows the Nios V processor to carry out other tasks while data buffers are being transferred.

Related Information

Embedded Peripherals IP User Guide

9.5.2.3. Place the Handler in Fast Memory

For the fastest execution of exception handler code, place the handler in a fast memory device. For example, an on-chip RAM with zero wait states is preferable to a slow SDRAM.

9.5.2.4. Select Hardware Interrupt Priorities

Hardware interrupt priority levels can have a significant impact on system performance. If two interrupts can be asserted at the same time, it is important to assign a higher priority level to the more critical interrupt, so that it runs in preference to the less critical interrupt.

9.5.2.4.1. Hardware Interrupt Priorities with the CLINT

When selecting the IRQ for each peripheral, remember that the HAL hardware interrupt funnel treats IRQ_0 as the highest priority. Assign each peripheral's interrupt priority based on its need for fast servicing in the overall system. Avoid assigning multiple peripherals to the same IRQ.

9.6. Debugging Nios V Processor ISRs

You can debug an ISR by setting breakpoints in the ISR. The debugger completely halts the processor on reaching a breakpoint. In the meantime, however, the other hardware in your system continues to operate. Therefore, it is inevitable that other interrupts are ignored while the processor is halted. You can use the debugger to step through the ISR code, but the status of other interrupt-driven device drivers is generally invalid by the time you return the processor to normal execution. You must reset the processor to return the system to a valid state.

10. Cache and Tightly-Coupled Memory

Nios V/g processor core supports tightly-coupled memory, instruction, and data caches. This chapter discusses cache-related issues that you need to consider to ensure the program executes correctly on the Nios V processor. Most software based on the Nios V processor hardware abstraction layer (HAL) works correctly without special accommodations for caches. However, some software must manage the cache directly.

For code that needs direct control over the cache, the Nios V processor architecture provides facilities to perform the following actions:

- Initialize lines in the instruction and data caches.
- Flush lines in the instruction and data caches.
- Bypass the data cache using peripheral regions.

10.1. Nios V Processor Cache Implementation

Depending on the Nios V processor core, the processor might have data or instruction caches. You can write programs generically to function correctly on any Nios V processor, regardless of whether it has cache memory. For a Nios V processor without caches, cache management operations are benign and have no effect.

The details for a Nios V processor cache are defined in the `system.h` file.

For example, the following processor has 4 KB instruction and data caches with 32-byte cache lines.

```
#define ALT_CPU_DCACHE_LINE_SIZE 32

#define ALT_CPU_DCACHE_SIZE 4096

#define ALT_CPU_ICACHE_LINE_SIZE 32

#define ALT_CPU_ICACHE_SIZE 4096
```

The HAL application program interface (API) provides the following functions for managing cache memory:

- `alt_dcache_flush()`
- `alt_dcache_flush_no_writeback()`
- `alt_dcache_flush_all()`
- `alt_icache_flush()`
- `alt_icache_flush_all()`

© Altera Corporation. Altera, the Altera logo, the 'a' logo, and other Altera marks are trademarks of Altera Corporation. Altera and Intel warrant performance of its FPGA and semiconductor products to current specifications in accordance with Altera's or Intel's standard warranty as applicable, but reserves the right to make changes to any products and services at any time without notice. Altera and Intel assume no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to inwriting by Altera or Intel. Altera and Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

10.2. Initializing Cache after Reset

After reset, the contents of the instruction cache and data cache are unknown. They must be initialized for correct operation. The data cache is initialized early in the boot sequence before the rest of the software uses it. The Nios V processor caches cannot be turned off by software; they are always enabled. Upon reset, the processor hardware invalidates the entire instruction cache.

Programs based on the HAL need not manage the initialization of cache memory. The HAL C run-time initialization code (`crt0.S`) provides a default reset handler that performs cache initialization before `alt_main()` or `main()` is called.

10.3. Device Driver Cache Considerations

Device drivers typically access control registers associated with their device. These registers are mapped into the Nios V processor address space. Apply the following considerations for the device driver cache:

- The data cache must be bypassed when accessing device registers to ensure that accesses are not lost or deferred due to the data cache.
- The device register must be located within a peripheral region to bypass the data cache.
- HAL I/O macros cannot bypass the data cache.
- Declaring a C pointer `volatile` does not make pointer accesses bypass the data cache. The `volatile` keyword merely prevents the compiler from optimizing out accesses using the pointer.

Related Information

- Nios® V Processor Reference Manual - Bypassing Cache (Peripheral Region)
- Device Driver Cache Considerations on page 190

10.4. Managing Cache in Multi-Master and Multi-Processor Systems

The Nios V processor does not provide hardware cache coherency. Instead, software cache coherency must be provided when communicating through shared memory. The data cache contents of all processors accessing the shared memory must be managed by software to ensure that all masters read the most recent values and not overwrite new data with stale data. This management is done by using the data cache flushing and bypassing facilities to move data between the shared memory and the data cache(s) as needed.

10.5. Tightly-Coupled Memory

If you want cache performance all the time, place your code or data in a tightly-coupled memory. Tightly-coupled memory is fast on-chip memory that bypasses the cache and has guaranteed low latency. Tightly-coupled memory gives the best memory access performance. You assign code and data to tightly-coupled memory partitions in the same way as other memory sections.

Cache instructions do not affect tightly-coupled memory. However, cache-management instructions become NOPs, which might result in unnecessary overhead.

11. MicroC/OS-II Real-Time Operating System

11.1. Overview of the MicroC/OS-II RTOS

Note: In the Quartus Prime software, only the Nios V Processor IP HAL API drivers are verified within the MicroC/OS-II RTOS. If the Nios V processor system contains other soft IP, validate the respective soft IP drivers during development.

MicroC/OS-II is a real-time kernel produced by Micrium® Inc. in 1992. MicroC/OS-II is a portable, ROMable, scalable, pre-emptive, real-time, multitasking kernel. MicroC/OS-II is used in hundreds of commercial applications. It is implemented on more than 40 different processor architectures in addition to the Nios V processor.

MicroC/OS-II provides the following services:

- Tasks (threads)
- Event flags
- Message passing
- Memory management
- Semaphores
- Time management

The MicroC/OS-II RTOS kernel is ported to the Nios V processor. It is built on top of the Hardware Abstraction Layer (HAL) and board support package (BSP). It provides RTOS facilities that makes it easier to write programs to run on Nios V processor. Programs written on top of uCOS-II has the following advantages:

- Programs are portable to other Nios V processor hardware systems of the same variant.
- Programs are more resistant to changes in the underlying hardware.
- Programs can access all HAL services by calling the UNIX-like HAL application program interface (API).
- Interrupt Service Routines (ISR) are easy to implement.

For more information about MicroC/OS-II features and usage, refer to *MicroC/OS-II - The Real-Time Kernel* by Jean J. Labrosse (CMP Books).

Related Information

Micrium Software and Documentation

© Altera Corporation. Altera, the Altera logo, the 'a' logo, and other Altera marks are trademarks of Altera Corporation. Altera and Intel warrant performance of its FPGA and semiconductor products to current specifications in accordance with Altera's or Intel's standard warranty as applicable, but reserves the right to make changes to any products and services at any time without notice. Altera and Intel assume no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to inwriting by Altera or Intel. Altera and Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

11.1.1. Support and Licensing

Intel distributes MicroC/OS-II in the Quartus Prime for evaluation purposes only. Commercial use of MicroC/OS-II is possible under the Apache 2.0 license. For more information and support for uCOS-II please refer to Micrium Licensing webpage.

Related Information

Micrium Licensing

11.2. Other RTOS Providers

Intel distributes MicroC/OS-II to provide you with immediate access to an easy-to-use Real-time Operating System (RTOS). Other RTOSs are available from third-party vendors.

11.3. The Nios V Processor Implementation of MicroC/OS-II

Intel enables and supports the implementation of MicroC/OS-II kernel for Nios V processor. Intel provides ready-made, working MicroC/OS-II design example which can be generated from Platform Designer.

The Intel FPGA implementation of MicroC/OS-II is designed to be easy to use. Using the BSP project settings, you can control the configuration for all the RTOS's modules.

You need not modify source files directly to enable or disable kernel features. Nonetheless, Intel provides the Nios V processor-specific source code in case you want to examine it. The MicroC/OS-II source code is located in the following directories:

- Processor-specific code: `<Intel Quartus installation directory>/ip/altera/soft_processor/intel_niosv_m/UCOSII`
- Processor-independent code: `<Intel Quartus installation directory>/nios2eds>/components/micrium_uc_osii`

The MicroC/OS-II software package behaves like the drivers for hardware components: When MicroC/OS-II is included in a Nios V project, the header and source files from `components/micrium_uc_osii` are included in the project path, causing the MicroC/OS-II kernel to compile and link as part of the project.

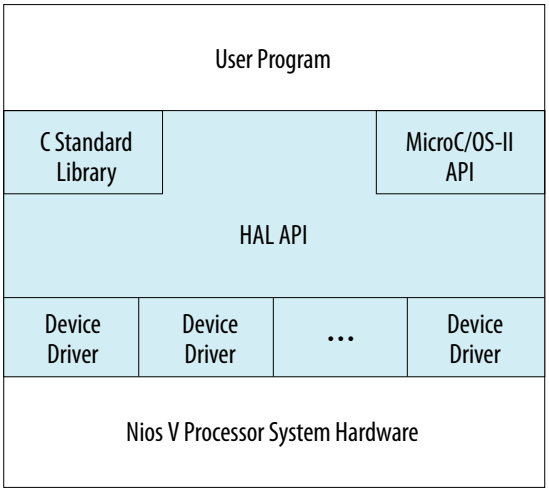
Related Information

Nios® V Embedded Processor Design Handbook

11.3.1. MicroC/OS-II Architecture

The Intel FPGA implementation of MicroC/OS-II for the Nios V processor extends the single-threaded HAL environment to include the MicroC/OS-II scheduler and the associated MicroC/OS-II API. The complete HAL API is available to all MicroC/OS-II projects.

Figure 24. Architecture of MicroC/OS-II Programs in Relation to the HAL API



The multi-threaded environment affects certain HAL functions.

11.3.2. MicroC/OS-II Device Drivers

Each peripheral (that is, each hardware component) can provide include files and source files in the `inc` and `src` subdirectories of the component's HAL directory.

In addition to the HAL directory, a component can optionally provide a `UCOSII` directory that contains code specific to the MicroC/OS-II environment. Similar to the HAL directory, the `UCOSII` directory contains `inc` and `src` subdirectories. When you create a MicroC/OS-II project, the BSP Editor copies the include and source directories for a given peripheral to your BSP subdirectory. These include and source directories are added to the search path for the project.

You can use the `UCOSII` directory to provide code that is used only in a multi-threaded environment. Other than these additional search directories, the mechanism for providing MicroC/OS-II device drivers is identical to the process for any other device driver.

The HAL system initialization process calls the MicroC/OS-II function `OSInit()` before `alt_sys_init()`, which instantiates and initializes each device in the system. Therefore, the complete MicroC/OS-II API is available to device drivers, although the system is still running in single-threaded mode until the program calls `OSStart()` from within `main()`.

Related Information

Nios V Processor Board Support Package Editor on page 53
For more information about the BSP Package Editor.

11.3.3. Thread-Safe Drivers

To enable a driver to be ported between the HAL and MicroC/OS-II environments, Altera defines a set of operating system-independent macros that provide access to operating system facilities. These macros implement functionality that is only relevant to a multi-threaded environment. When compiled for a multithreaded uCOS/II project,

the macros expand to MicroC/OS-II API calls. When compiled for a single-threaded HAL project, the macros expand to benign empty implementations. These macros are used in Intel-provided device driver code, and you can use them if you need to write a device driver with similar portability.

The path listed for the header file is relative to the `<Intel Quartus installation directory>/nios2eds/components/micrium_uc_osii/UCOSII/inc` directory.

Table 47. OS-Independent Macros for Thread-Safe HAL Drivers

Macros	MicroC/OS-II Implementation	Single-Threaded HAL Implementation
Event Flag Macros Defined in `os/alt_flag.h` and `priv/alt_flag_ucosii.h`.		
`ALT_FLAG_GRP(group)`	Create a flag group with the name `group`.	Empty Statement
`ALT_EXTERN_FLAG_GRP(group)`	Create an external reference to a flag group with name `group`.	
`ALT_STATIC_FLAG_GRP(group)`	Create a static flag group with the name `group`.	
`ALT_FLAG_CREATE(group, flags)`	Call OSFlagCreate() to initialize the flag group pointer, group, with the flag value flags. The error code is the return value of the macro.	Return 0 (success)
`ALT_FLAG_PEND(group, flags, wait_type, timeout)`	Call `OSFlagPend()` with the first four input arguments set to `group`, `flags`, `wait_type`, and `timeout` respectively. The error code is the return value of the macro.	
`ALT_FLAG_POST(group, flags, opt)`	Call `OSFlagPost()` with the first three input arguments set to `group`, `flags`, and `opt` respectively. The error code is the return value of the macro.	
Semaphore Macros Defined in `os/alt_sem.h` and `priv/alt_sem_ucosii.h`.		
`ALT_SEM(sem)`	Create an `OS_Event` pointer with the name `sem`.	Empty Statement
`ALT_EXTERN_SEM(sem)`	Create an external reference to an `OS_Event` pointer with the name `sem`.	
`ALT_STATIC_SEM(sem)`	Create a static OS_Eventpointer with the name `sem`.	
`ALT_STATIC_CREATE(sem, value)`	Call `OSSemCreate()` with the argument value to initialize `OS_Event` pointer, `sem`. The return value is zero on success, or negative otherwise.	Return 0 (success)
`ALT_SEM_PEND(sem, timeout)`	Call `OSSemPend()` with the first two arguments set to `sem` and `timeout` respectively. The error code is the return value of the macro.	
`ALT_SEM_POST(sem)`	Call `OSSemPost()` with the input argument `sem`.	

Table 48. Arguments for Event Flags Macros

Argument	Valid Inputs	Defined in Header
`wait_type`	• `ALT_FLAG_WAIT_SET_ALL_WITH_CONSUME` • `ALT_FLAG_WAIT_SET_ALL_WO_CONSUME` • `ALT_FLAG_WAIT_SET_ANY_WITH_CONSUME` • `ALT_FLAG_WAIT_SET_ANY_WO_CONSUME`	• `os/alt_flag.h` • `ucos_ii.h`
`timeout`	• `ALT_FLAG_WAIT_MAX_TIMEOUT`	
`opt`	• `ALT_FLAG_SET` • `ALT_FLAG_CLEAR`	

11.3.4. The newlib ANSI C Standard Library

Programs built on top of MicroC/OS-II can also call the ANSI C standard library functions. Some consideration is necessary in a multi-threaded environment to ensure that the C standard library functions are thread-safe. The newlib C library stores all global variables in a single structure referenced through the pointer `_impure_ptr`. However, the Intel FPGA MicroC/OS-II implementation creates a new instance of the structure for each task. During a context switch, the value of `_impure_ptr` is updated to point to the current task's version of this structure. In this way, the contents of the structure pointed to by `_impure_ptr` are treated as thread local. For example, through this mechanism each task has its own version of `errno`.

This thread-local data is allocated at the top of the task's stack. You must make allowance for thread-local data storage when allocating memory for stacks. In general, the `_reent` structure consumes approximately 900 bytes of data.

In addition, the MicroC/OS-II implementation provides appropriate task locking to ensure that heap accesses (calls to `malloc()` and `free()`) are also thread-safe.

11.3.5. Interrupt Service Routines for MicroC/OS-II

Implementing ISRs for MicroC/OS-II normally involves some housekeeping details, as described in *MicroC/OS-II: The Real-Time Kernel*. However, because the Nios V implementation of MicroC/OS-II is based on the HAL, several of these details are taken care of for you. The HAL performs the following housekeeping tasks for your interrupt service routine (ISR):

- Saves and restores processor registers
- Calls `OSIntEnter()` and `OSIntExit()`

The HAL also allows you to write your ISR in C, rather than assembly language.

Related Information

Trap Handling on page 154
For information about the exception handler software routines, HAL-provided services, and API.

11.4. Implementing MicroC/OS-II Projects for the Nios V Processor

To create a program based on MicroC/OS-II, start by setting the BSP properties so that it is a MicroC/OS-II project. You can control the configuration of the MicroC/OS-II kernel using BSP settings with the BSP Editor or on the Nios V command line.

- You must not edit header files (such as `OS_CFG.h`) or source code to configure the MicroC/OS-II features. The project settings are reflected in the BSP's `settings.bsp` file.
- MicroC/OS-II settings are identified by the prefix `ucosii.`. For example `ucosii.os_max_tasks`.

Related Information

- Nios V Processor Board Support Package Editor on page 53
  For more information about the BSP Package Editor.
- Nios V Processor Appendix on page 224
- µC/OS-II User Manual

12. MicroC/TCP-IP Protocol Stack

12.1. Overview of the MicroC/TCP-IP Protocol Stack

The µC/TCP-IP protocol stack is a small footprint open-source implementation of a high-performance TCP/IP stack, which Intel uses to implement a reliable, full-featured TCP/IP stack for a Nios V processor. The µC/TCP-IP protocol stack is designed for use in embedded systems with small memory footprint.

In the Quartus Prime Design Suite, the µC/TCP-IP protocol is a software package that you can add to your board support package (BSP), available through the Nios V processor software build tools. The µC/TCP-IP protocol includes these features:

- Internet Protocol (IP) including packet forwarding over multiple network interfaces
- Internet control message protocol (ICMP) for network maintenance and debugging
- User datagram protocol (UDP)
- Transmission Control Protocol (TCP) with congestion control, round trip time (RTT) estimation, and fast recovery and retransmit
- Dynamic host configuration protocol (DHCP)
- Address resolution protocol (ARP) for Ethernet
- Standard sockets API

Related Information

µC/TCP-IP

12.2. Support and Licensing

Intel distributes µC/OS-II and µC/TCP-IP with the Quartus Prime Design Suite under the Apache 2.0 license. For more information and support for µC/OS-II please refer to Micrium Licensing webpage.

Related Information

Micrium Licensing

12.3. Prerequisites for Understanding the MicroC/TCP-IP Protocol Stack

To make the best use of information in this chapter, you should be familiar with these topics:

- Sockets
- Nios V Processor Quick Start Guide
- MicroC/OS-II RTOS

© Altera Corporation. Altera, the Altera logo, the 'a' logo, and other Altera marks are trademarks of Altera Corporation. Altera and Intel warrant performance of its FPGA and semiconductor products to current specifications in accordance with Altera's or Intel's standard warranty as applicable, but reserves the right to make changes to any products and services at any time without notice. Altera and Intel assume no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to inwriting by Altera or Intel. Altera and Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

Related Information

- Network sockets
- Nios V Embedded Processor Design Handbook
- Overview of the MicroC/OS-II RTOS on page 191

12.4. Introduction to the MicroC/TCP-IP Protocol Stack - Nios V Processor Edition

Intel provides Nios V processor implementation of the µC/TCP-IP protocol stack, including source code, as part of a Nios V processor software package. The µC/TCP-IP protocol stack provides you with immediate access to a network stack for the Nios V processor.

Intel FPGA's implementation of the µC/TCP-IP protocol stack includes an API wrapper for the socket API.

The µC/TCP-IP protocol stack uses the MicroC/OS-II RTOS multithreaded environment. Therefore, to use the µC/TCP-IP protocol stack with the Nios V processor software build tools, you must choose the MicroC/OS-II BSP type. The Nios V processor system must also contain an Ethernet interface, or media access control (MAC). The Intel-provided µC/TCP-IP protocol stack includes MAC driver support for Triple Speed Ethernet Intel FPGA IP core.

There are two µC/TCP-IP Example Designs, using Triple Speed Ethernet Intel FPGA IP core, which act as reference to demonstrate µC/TCP-IP protocol stack:

- µC/TCP-IP IPerf Example Design
- µC/TCP-IP Simple Socket Server Example Design

Related Information

- Triple-Speed Ethernet Intel® FPGA IP
- Nios® V Processor - Using the MicroC/TCP-IP Stack
- Developing Programs Using the Hardware Abstraction Layer on page 84

12.4.1. Nios V Processor System Requirements

To use the MicroC/TCP-IP Stack, your Nios V processor system must meet the following requirements:

- The system hardware includes an Ethernet interface
- The BSP type based on MicroC/OS-II
- Enable BSP setting `ucosii.os_tmr_en`
- The system clock timer set to an appropriate timer device

12.5. The MicroC/TCP-IP Protocol Stack Files and Directories

Intel provides µC/TCP-IP source code for your reference. By default, the files are installed in `<Quartus Prime install path>\niosv\components\uc_tcp_ip` directory. The directory contains the original µC/TCP-IP protocol stack source code and documentation from vendor.

There are multiple libraries in the directory. The main library is µC-TCP-IP. The main library depends upon µC-Common, µC-CPU, µC-LIB, µC-DHCPc, and µC-Conf.

Table 49. MicroC/TCP-IP Library and Description

Library	Description
µC-TCP-IP	µC/TCP-IP protocol stack libraries
µC-Common	Includes a Kernel Abstraction Layer (KAL) to easily switch between different OS's. Nios V processor uses microC/OS-II KAL to implement µC/TCP-IP protocol stack
µC-CPU	Implements CPU specific features such as timestamp, IRQ disable/enable
µC-LIB	Includes library functions such as math functions and string manipulation
µC-DHCPc	Addition features to µC/TCP-IP to acquire the IP address via DHCP server instead of setting a static address
µC-Conf	Header configuration files for all the above libraries.

Intel FPGA's implementation of the µC/TCP-IP Stack is based on latest available version the protocol stack, with wrappers around the code to integrate it with the HAL. Quartus Prime updates the µC/TCP-IP Stack with each release. You may find the specific version info in the header of each source code.

Related Information

MicroC/TCP-IP Github Release

12.6. Enabling MicroC/TCP-IP Protocol Stack

You can use Nios V Command Shell CLI to enable µC/TCP-IP software package with the following command:

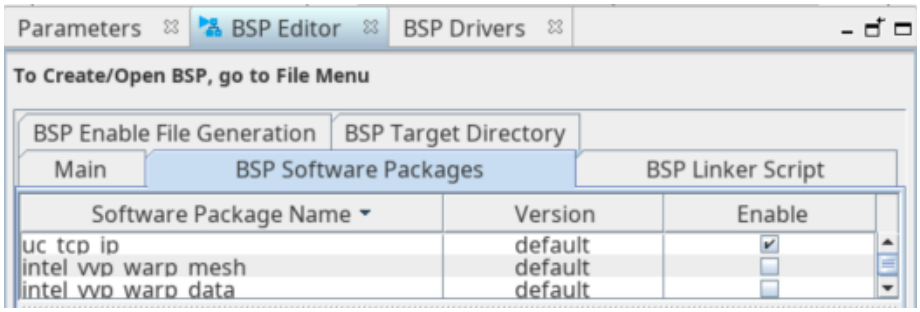
Note: Disable the vectorization in GCC because µC/TCP-IP stack is not compatible with GCC compiler.

```
niosv-bsp -c -t=ucosii -e="enable_sw_package uc_tcp_ip" \
-e=”set_setting hal.make.cflags_user_flags \
{-fno-tree-vectorize}” \
-s=<.qsys file> -p=<.qpf file>
```

Alternately, you can also enable it through Quartus Prime using the following steps:

1. Open the Platform Designer.
2. Click the **BSP Editor** tab.
3. Click the **BSP Software Packages** tab.
4. Tick the **Enable** checkbox.

Figure 25. BSP Editor



5. Define `-fno-tree-vectorize` in `hal.make.cflags_user_flags`.

Figure 26. Custom GCC Flags



12.7. Using the MicroC/TCP-IP Protocol Stack

Intel provides an example on how to use µC/TCP-IP protocol stack in Intel FPGA Development Kit. The primary application file is `uc_tcp_ip_init.c` which consist of the implementation source code for your reference. You can modify the code to meet your actual product requirements.

`uc_tcp_ip_init.c` performs below operations:

1. Calls `network_init()` function to:
   a. Initialize µC/TCP-IP tasks
   b. Add and start the TSE mSGDMA driver into system
   c. Configure IP address through static IP or using a DHCP server
2. Uses `network_stop()` function to stop a particular interface such as TSE instance. This function is at the end of `uc_tcp_ip_init.c`

Example 29. `network_init()` network stack initialization implementation code

```
CPU_BOOLEAN network_init(struct network_conf *conf, NET_IF_NBR *p_if_nbr) {
    NET_ERR    err_net;

    // µC/TCP-IP generic stack init.
    err_net = Net_Init(
            &NetRxTaskCfg,
            &NetTxDeallocTaskCfg,
            &NetTmrTaskCfg);
    if (err_net != NET_ERR_NONE) {
        logf("Failed to Net_Init(): (%d).\n", err_net);
        return DEF_FALSE;
    }

    // Update TSE sys info struct and MAC address from configuration.
    NetDev_Cfg_Ether_TSE.BaseAddr = (CPU_ADDR) conf->tse_sys_info;

    Mem_Copy(NetDev_Cfg_Ether_TSE.HW_AddrStr, conf->mac_addr,
```

```
        sizeof(CPU_CHAR) * NET_IF_802x_ADDR_SIZE_STR);

    // Add TSE.
    *p_if_nbr = NetIF_Add(
            (void *)&NetIF_API_Ether, // const for all eth interfaces
            (void *)&NetDev_API_TSE_mSGDMA_Intel_HAL,
            NULL, // (void *)&Net_DrvBSP_Nios_II,
            (void *)&NetDev_Cfg_Ether_TSE,
            NULL, // (void *)&NetPhy_API_Generic,
            NULL, // (void *)&NetPhy_Cfg_Ether_TSE,
                    &err_net);
    if (err_net != NET_IF_ERR_NONE) {
        logf("Failed to NetIF_Add(): (%d).\n", err_net);
        return DEF_FALSE;
    }

    // Start TSE.
    NetIF_Start(*p_if_nbr, &err_net);
    if (err_net != NET_IF_ERR_NONE){
        logf("Failed to NetIF_Start(): (%d).\n", err_net);
        return DEF_FALSE;
    }

    CPU_BOOLEAN ret;
    if (conf->use_dhcp) {
        ret = conf_dhcp(*p_if_nbr);
    } else {
        ret = conf_static(*p_if_nbr, conf);
    }

    return ret;
}
```

12.7.1. Initializing the Stack

Before you initialize the uc-tcp-ip stack, start the MicroC/OS-II scheduler by calling `OSStart()` from `main()`. Ensure RTOS and device I/O drivers are available, then only perform network stack initialization in a high priority task. You should perform all initialization call within an initialization task.

The `network_init()` function is called from the `uc_main_task()` function to initialize the network stack. The `uc_main_task()` then calls different µC/OS-II tasks or functions to implement the µC/TCP-IP Simple Socket Server or iPerf application.

You can use the following µC/OS-II tasks to create a thread, and performs actions specific task for simple socket server:

- `SSSCreateOSDataStructs()` — Creates an instance of all the µC/OS-II resources.
- `SSSCreateTasks()` — Initializes tasks that do not use the networking services.
- `SSSSimpleSocketServerTask()`— Manages the socket server connection, and calls relevant subroutines to manage the socket connection

You can use `App_IPerf_TaskTerminal()`— Initialize and run iPerf Terminal function prototype for iPerf application:

The `uc_main_task()` is then added as an RTOS task right before `OSStart()` is called.

Related Information

Nios® V Embedded Processor Design Handbook: MicroC/TCP-IP Simple Socket Server Concepts

12.7.2. Network IP configuration

You can configure the MAC and IP addresses of the µC/TCP-IP module by editing the `struct network_conf` conf in `main()`.

`struct network_conf` call out below function prototype in `uc_tcp_ip_init.c` to configure IP address through static IP or DHCP server

- `CPU_BOOLEAN conf_static()` – this function is used to configure static IP address
- `CPU_BOOLEAN conf_dhcp()` - this function calls the `AppInit_DHCPc()` to initialize µC/DHCPc and start DHCPc negotiation for the interface to acquire IP address dynamically. This function returns only after DHCP negotiation is completed

For more information on other µC/TCP-IP configuration, refer to *Nios V Embedded Processor Design Handbook* which contains below information.

- Configuring Hardware Name
- Configuring MAC and IP Addresses
- Configuring µC/TCP-IP Initialization
- Configuring iPerf Server Auto-Initialization

Related Information

Nios® V Embedded Processor Design Handbook: Optional Configuration

12.7.3. Example applications

Intel provides two example designs leveraging the µC/TCP-IP network stack:

- iPerf
- Simple Socket Server

You can generate both example designs using Command Line Interface or Platform Designer.

12.7.3.1. simple_socket_server.c

Example application code to demonstrate the use of MicroC/OS-II and µC/TCP-IP running on Nios V processor. The application code runs socket server task that listens for commands on a TCP/IP port and dispatches those commands to a set of LED management tasks.

12.7.3.2. app_iperf.c

Example application code to run Iperf tools with µC/TCP-IP protocol stack.

Related Information

Nios® V Embedded Processor Design Handbook: MicroC/TCP-IP Example Designs

13. FreeRTOS Real-Time Operating System

13.1. Overview of the FreeRTOS

Note: In the Quartus Prime software, only the Nios V Processor IP, JTAG UART IP, and Interval Timer IP are supported with FreeRTOS drivers. If Nios V processor system contains other soft IP, manually create their FreeRTOS drivers for use in the FreeRTOS BSP by leveraging the HAL driver, or develop a new FreeRTOS driver.

FreeRTOS is a real-time kernel solely owned, developed, and maintained by Amazon Web Services. It is a kernel with a scalable size that includes a tick-less power saving mode for small embedded systems. FreeRTOS is implemented on more than 40 different processor architectures and more than 15 toolchains, including the Nios V processor.

FreeRTOS provides the following services:

- Pre-emptive operation
- Task priority assignment
- Task notification mechanism
- Queues
- Binary and counting semaphores
- Mutexes
- Recursive Mutexes
- Software timers
- Event groups
- Tick hook and idle hook functions
- Stack overflow checking
- Trace recording
- Task run-time statistics gathering
- Full interrupt nesting model
- Software managed interrupt stack

The Nios V processor supports the FreeRTOS kernel. FreeRTOS is built on top of the Hardware Abstraction Layer (HAL) and board support package (BSP). FreeRTOS provides RTOS facilities that ease you to write programs to run on the Nios V processor. Programs written on top of FreeRTOS has the following advantages:

© Altera Corporation. Altera, the Altera logo, the 'a' logo, and other Altera marks are trademarks of Altera Corporation. Altera and Intel warrant performance of its FPGA and semiconductor products to current specifications in accordance with Altera's or Intel's standard warranty as applicable, but reserves the right to make changes to any products and services at any time without notice. Altera and Intel assume no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to inwriting by Altera or Intel. Altera and Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

- Programs are portable to other Nios V processor hardware systems of the same variant.
- Programs are more resistant to changes in the underlying hardware.
- Programs can access all HAL services by calling the UNIX-like HAL application program interface (API).
- Interrupt Service Routines (ISR) are easy to implement.

Related Information

- FreeRTOS Homepage
  Refer to the homepage for more information about FreeRTOS.
- KDB: Why is Mailbox Client HAL API unavailable for Agilex™ 5 FPGA devices?
  For more information about the example on how to intergrate Altera HAL Drivers into FreeRTOS. This example is based on the Mailbox Client HAL API.

13.1.1. Support and Licensing

Intel distributes FreeRTOS in the Quartus Prime for evaluation purposes only. The FreeRTOS kernel and other FreeRTOS libraries are distributed for free under the MIT open source license.

Related Information

FreeRTOS Licensing
For more information and support for FreeRTOS.

13.2. The Nios V Processor Implementation of FreeRTOS

Intel enables and supports the implementation of the FreeRTOS kernel for the Nios V processor. The Intel FPGA implementation of FreeRTOS is easy to use. You can control the configuration for all the RTOS modules using the BSP project settings.

You do not need to modify source files directly to enable or disable kernel features. Nonetheless, Intel provides the Nios V processor-specific source code. The FreeRTOS source code is located in the following directories:

- Processor-specific code: `<Intel Quartus installation directory>/ip/altera/soft_processor/intel_niosv_common/FREERTOS`
- Processor-independent code: `<Intel Quartus installation directory>/ip/altera/soft_processor/aws_freertos`

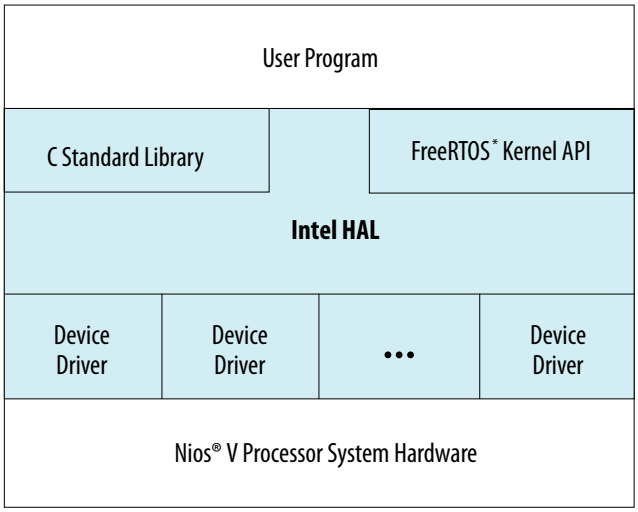
The FreeRTOS software package functions similarly to the drivers for hardware components. When you include FreeRTOS in a Nios V processor project, the header and source files from `ip/altera/soft_processor/aws_freertos` are included in the project path, causing the FreeRTOS kernel to compile and link as part of the project.

13.2.1. FreeRTOS Architecture

The Intel FPGA implementation of FreeRTOS for the Nios V processor processor extends the single-threaded HAL environment to include the FreeRTOS scheduler and the associated FreeRTOS API. The complete HAL API is available to all FreeRTOS projects.

The multi-threaded environment affects certain HAL functions.

Figure 27. Architecture of FreeRTOS Programs in Relation to the HAL API



13.2.2. FreeRTOS Device Drivers

Each peripheral or each hardware component can provide `include` files and `source` files in the `inc` and `src` subdirectories of the component's HAL directory.

In addition to the HAL directory, a component can optionally provide a FREERTOS directory that contains code specific to the FreeRTOS environment. Similar to the HAL directory, the FREERTOS directory contains `inc` and `src` subdirectories. When you create a FreeRTOS project, the BSP Editor copies the `include` and `source` directories for a given peripheral to your BSP subdirectory. These `include` and `source` directories are added to the search path for the project.

You can use the FREERTOS directory to provide code that is used only in a multithreaded environment. Other than these additional search directories, the mechanism for providing FreeRTOS device drivers is identical to the process for any other device driver.

The HAL system initialization process calls the function `ALT_OS_Init()` before `alt_sys_init()`, which instantiates and initializes each device in the system. Therefore, the complete FreeRTOS API is cavailable to device drivers, although the system is still running in single-threaded mode until the program calls `vTaskStartScheduler()` from within `main()`.

Related Information

Nios V Processor Board Support Package Editor on page 53
For more information about the BSP Package Editor.

13.2.3. Thread-Safe HAL Drivers

To enable a driver to be ported between the HAL and FreeRTOS environments, Intel defines a set of operating system-independent macros that provide access to operating system facilities. These macros implement functionality that is only relevant to a multi-threaded environment. You can use the macro in Intel-provided device driver code, and if you need to write a device driver with similar portability.

- When you compile for a multithreaded FreeRTOS project, the macros expand to FreeRTOS API calls.
- When you compile for a single-threaded HAL project, the macros expand to benign empty implementations.

The path listed for the header file is relative to the `/ip/altera/soft_processor / aws_freertos/FREERTOS/inc` directory.

Table 50. OS-Independent Macros for Thread-Safe HAL Drivers

Macros	FreeRTOS Implementation	Single-Threaded HAL Implementation
Event Flag Macros Defined in `os/alt_flag.h and priv/alt_flag_freertos.h`.		
`ALT_FLAG_GRP(group)`	Creates a flag group with the name `group`.	Empty Statement
`ALT_EXTERN_FLAG_GRP(group)`	Creates an external reference to a flag group with name `group`.	
`ALT_STATIC_FLAG_GRP(group)`	Creates a static flag group with the name `group`.	
`ALT_FLAG_CREATE(group, flags)`	• Calls `xEventGroupCreate()` to initialize the flag group pointer, `group`, with the flag value `flags`. • The error code is the return value of the macro.	Return 0 (success)
`ALT_FLAG_PEND(group, flags, wait_type, timeout)`	• Calls `xEventGroupWaitBits()` with the first four input arguments set to `group`, `flags`, `wait_type`, and `timeout` respectively. • The error code is the return value of the macro.	
`ALT_FLAG_POST(group, flags, opt)`	• Calls `xEventGroup[Set|Clear]Bits()` or `xEventGroup[Set|Clear]BitsFromISR()` with the first three input arguments set to `group`, `flags`, and `opt` respectively. • The error code is the return value of the macro.	
Semaphore Macros Defined in `os/alt_sem.h and priv/alt_sem_freertos.h`.		
`ALT_SEM(sem)`	Creates an `SemaphoreHandle_t` with the name `sem`.	Empty Statement
`ALT_EXTERN_SEM(sem)`	Creates an external reference to an `SemaphoreHandle_t` with the name `sem`.	
`ALT_STATIC_SEM(sem)`	Creates a static `SemaphoreHandle_t` with the name `sem`.	

continued...

Macros	FreeRTOS Implementation	Single-Threaded HAL Implementation
`ALT_STATIC_CREATE(sem, value)`	• Calls `xSemaphoreCreateCounting()` with the argument value to initialize initialize `SemaphoreHandle_t, sem`. • The return value is zero on success, or negative otherwise.	Return 0 (success)
`ALT_SEM_PEND(sem, timeout)`	• Calls `xSemaphoreTake()` with the first two arguments set to `sem` and `timeout` respectively. • The error code is the return value of the macro.	
`ALT_SEM_POST(sem)`	Calls `xSemaphoreGive()` with the input argument `sem`.	

Table 51. Arguments for Event Flags Macros

Argument	Valid Inputs	Defined in Header
`wait_type`	• `ALT_FLAG_WAIT_SET_ALL_WITH_CONSUME` • `ALT_FLAG_WAIT_SET_ALL_WO_CONSUME` • `ALT_FLAG_WAIT_SET_ANY_WITH_CONSUME` • `ALT_FLAG_WAIT_SET_ANY_WO_CONSUME`	`os/alt_flag.h`
`timeout`	• `ALT_FLAG_WAIT_MAX_TIMEOUT`	
`opt`	• `ALT_FLAG_SET` • `ALT_FLAG_CLEAR`	

13.2.4. The newlib ANSI C Standard Library

Programs built on top of FreeRTOS can also call the ANSI C standard library functions. Consider the following scenarios in a multi-threaded environment to ensure that the C standard library functions are thread-safe:

- The newlib C library stores all global variables in a single structure referenced through the pointer `_impure_ptr`. The Intel FPGA FreeRTOS implementation creates a new instance of the structure for each task. During a context switch, Nios V processor updates the value of `_impure_ptr` to point to the current task's version of this structure. In this way, Nios V processor treats the contents of the structure pointed to by `_impure_ptr` as thread local. For example, through this mechanism each task has its own version of `errno`. Nios V processor allocates this thread-local data at the top of the task's stack. You must make allowance for thread-local data storage when allocating memory for stacks. In general, the `_reent` structure consumes approximately 900 bytes of data.
- The FreeRTOS implementation provides appropriate task locking (`_malloc_lock()/unlock()` and `_env_lock()/unlock()`) to ensure that heap accesses (`calls to malloc()`, `free()`, `etc()`, and `setenv()`) are also thread-safe. By default, the `freertos.os_thread_safe_newlib` flag is set to 1 in the Board Support Package Editor to enable the thread-safe newlib library.

13.3. Implementing FreeRTOS Project for the Nios V Processor

To create a program based on FreeRTOS, create a Nios V processor Platform Designer system. Based on the system `.qsys` file, generate the FreeRTOS BSP project using BSP Editor, followed by the application project.

You can import and build the software applications in Ashling RiscFree IDE for Intel FPGA or using the command line. Once the FreeRTOS-based `.elf` file is built, you can download the software into the processor system.

Related Information

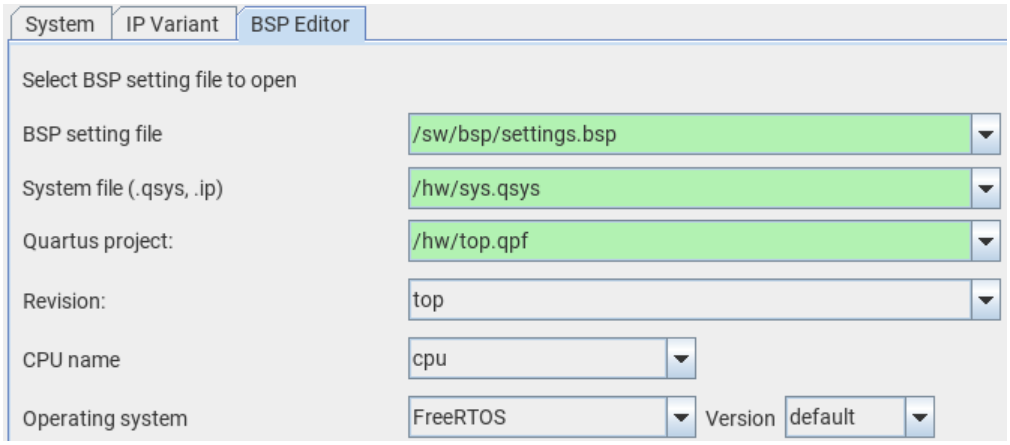
- Nios V Processor Board Support Package Editor on page 53
  For more information about the BSP Package Editor.
- Nios V Processor Board Support Package Editor on page 53
  For more information about the BSP Package Editor.
- Nios V Processor Appendix on page 224
- Ashling* RiscFree* Integrated Development Environment (IDE) for Altera® FPGAs User Guide
- Nios V Processor Software Development and Implementation on page 18

13.3.1. Create FreeRTOS BSP Project

After developing a Nios V processor system in Quartus Prime software, proceed with the following steps:

1. In the Quartus Prime software, go to **Tools ➤ Platform Designer**.
2. In the Platform Designer window, go to **File ➤ New BSP**. The **Create New BSP** window appears.
3. For **BSP setting file**, create a BSP file (`settings.bsp`).
4. For **System file (.qsys, .ip)**, select the Nios V processor Platform Designer system.
5. For **Quartus project**, select the example design Quartus Project File.
6. For **Revision**, select the target revision.
7. For **CPU name**, select the target processor.
8. Select the **Operating system** as **FreeRTOS**.
9. Click **Create** to create the BSP file.

Figure 28. Create New BSP window for FreeRTOS



13.3.2. Configure FreeRTOS Features

The default BSP settings enable all FreeRTOS features to facilitate most FreeRTOS-based applications, such as a simple Hello World. For further optimizations, you may configure the FreeRTOS features through the Intel HAL BSP and FreeRTOS BSP settings in the BSP Editor.

- You must not edit header files (such as `FreeRTOSConfig.h`) or kernel source code to configure the FreeRTOS features. Perform this action using the BSP Editor, and the new FreeRTOS settings are reflected in the BSP's `settings.bsp` file.
- The prefix freertos identify FreeRTOS settings. For example, `freertos.os_max_priorities`.

Related Information

- Intel HAL BSP on page 298
- FreeRTOS BSP on page 324

13.3.3. Write FreeRTOS-based Source Codes

The example below shows a simple Hello World application from any "Hello World on Nios V Processor Design Example" in FPGA Design Store.

This simple Hello World creates three tasks that print individual Hello World messages, followed by an idle period using `xTaskDelay()`. Each task has a defined priority that FreeRTOS refers to. If two or more tasks are out of idle simultaneously, the task with the highest priority is executed first.

A mutex (`g_mutex`) provides exclusive access to the whole processor system for a single task. The running task claims the mutex using `xSemaphorTake()` and releases it using `xSemaphoreGive()` after it completes its operation. Without claiming the mutex, a task waits till a maximum period is reached.

```
#include <stdio.h>
#include "FreeRTOS.h"
#include "task.h"
#include "timers.h"
#include "semphr.h"
```

```
#define TASK_STACKSIZE  2048

/* Task priority setting : Task1 > Task2 > Task3 */
#define TASK1_PRIORITY  (tskIDLE_PRIORITY + 3)
#define TASK2_PRIORITY  (tskIDLE_PRIORITY + 2)
#define TASK3_PRIORITY  (tskIDLE_PRIORITY + 1)

SemaphoreHandle_t g_mutex;

#define NUM_PRINTS_PER_TASK 5

int t1_ms_slept = 750 / portTICK_PERIOD_MS;
int t2_ms_slept = 450 / portTICK_PERIOD_MS;
int t3_ms_slept = 250 / portTICK_PERIOD_MS;

void prvTask1(void* pvParameters)
{
    int i = 0;
    int ms_slept = (*((int *)pvParameters));
    while (1)
    {
        xSemaphoreTake(g_mutex,portMAX_DELAY);
        if (i < NUM_PRINTS_PER_TASK) {
            printf("Hello from task1: %d\n\n", i++);
        }
        xSemaphoreGive(g_mutex);
        vTaskDelay(ms_slept);
    }
}

void prvTask2(void* pvParameters)
{
    int i = 0;
    int ms_slept = (*((int *)pvParameters));
    while (1)
    {
        xSemaphoreTake(g_mutex,portMAX_DELAY);
        if (i < NUM_PRINTS_PER_TASK) {
            printf("Hello from task2: %d\n\n", i++);
        }
        xSemaphoreGive(g_mutex);
        vTaskDelay(ms_slept);
    }
}

void prvTask3(void* pvParameters)
{
    int i = 0;
    int ms_slept = (*((int *)pvParameters));
    while (1)
    {
        xSemaphoreTake(g_mutex,portMAX_DELAY);
        if (i < NUM_PRINTS_PER_TASK) {
            printf("Hello from task3: %d\n\n", i++);
        }
        xSemaphoreGive(g_mutex);
        vTaskDelay(ms_slept);
    }
}

int main(void)
{
    printf("Hello FreeRTOS from main...\n");

    g_mutex = xSemaphoreCreateMutex();

    if (pdFAIL == xTaskCreate( prvTask1, "Task1", TASK_STACKSIZE,
                               &t1_ms_slept, TASK1_PRIORITY, NULL )){
        printf("Task1 creation fail!!!!\n");
    }
```

```
    if (pdFAIL == xTaskCreate( prvTask2, "Task2", TASK_STACKSIZE,
                               &t2_ms_slept, TASK2_PRIORITY, NULL )){
        printf("Task2 creation fail!!!!\n");
    }
    if (pdFAIL == xTaskCreate( prvTask3, "Task3", TASK_STACKSIZE,
                               &t3_ms_slept, TASK3_PRIORITY, NULL )){
        printf("Task3 creation fail!!!!\n");
    }

    vTaskStartScheduler();

    for( ;; );
}
```

Related Information

FPGA Design Store – Hello World on the Nios® V Processor

13.3.4. Generate Application CMake File

Execute the command below to generate an application CMakeLists.txt.

Note: You must open the Nios V Command Shell to run `niosv-app`.

```
$ niosv-shell
$ niosv-app --bsp-dir=<BSP folder> --app-dir=<APP folder> \
--srcs=<Source codes>
```

13.3.5. Build FreeRTOS BSP and Application Projects

To build the projects in Ashling RiscFreeRiscFree IDE for Intel FPGAs, refer to *Ashling RiscFreeRiscFree IDE for Intel FPGA* for Intel FPGAs User Guide on importing, building, and debugging Nios V processor application.

To build the projects in the command line, proceed using the `cmake` and `make` commands.

```
$ cmake -G "Unix Makefiles" -DCMAKE_BUILD_TYPE=Debug -B \
<APP folder>/debug -S <APP folder>
$ make -C <APP folder>/debug
```

Once the FreeRTOS-based .elf file is generated, you can continue programming the Nios V processor into the target FPGA device.

Related Information

- Building and Running Nios V Processor Software on page 38
- Nios® V Embedded Processor Design Handbook - Programming Nios® V into the FPGA Device
- Ashling* RiscFree* Integrated Development Environment (IDE) for Altera® FPGAs User Guide - Debug Setup for Nios® V Processor System

13.3.6. Integrating Device Drivers

If you want to integrate a device driver into your FreeRTOS build, you can follow one of the following methods:

- Manual integration without initialization.
- Manual integration with initialization.
- Integration using header files and TCL scripts.

Related Information

Integrating a Device Driver in the HAL on page 137
For more information about integrating using header files and TCL scripts.

13.3.6.1. Integrating Device Drivers without Initialization Manually

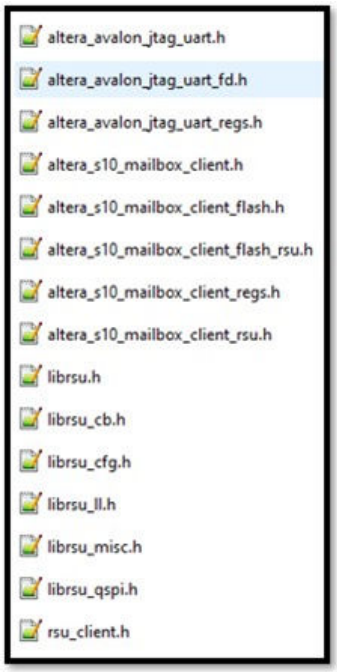
You need to copy the driver files into the APP directory. Modify the APP `CMakeLists.txt` to include the driver files in the APP build. Alternatively, you can include the source files of the device drivers using the `--srcs` argument when running niosv-app command.

13.3.6.2. Integrating Device Drivers with Initialization Manually

You must copy the driver into the BSP directory. Modify the BSP `CMakeLists.txt` to include the driver files in the BSP build. For instance, to copy the Mailbox Client IP driver into your BSP build, you can use the following instructions after you build your BSP:

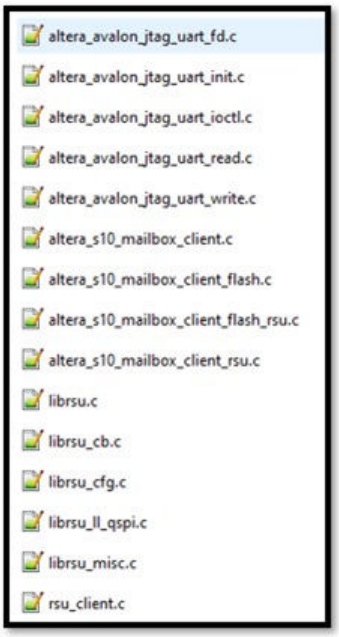
1. Navigate to the <BSP `project folder>/drivers`.
2. Separately navigate to <`Quartus Prime directory>/<Version>/ip/altera/pgm/altera_s10_mailbox_client/HAL`.
3. Observe that both folder directories are having the same – `inc` and `src` files.
4. Copy the contents of Quartus Prime HAL folders into the BSP drivers folders respectively.
5. Navigate to <`Quartus Prime directory>/<version>/ip/altera/pgm/altera_s10_mailbox_client/inc`.
6. Copy `altera_s10_mailbox_client_regs.h` into the BSP `drivers/inc` folder.
7. Verify the content of the BSP `inc` folder contains the files in the following figure.

Figure 29. BSP `inc` folder



8. Verify the content of the BSP `src` folder contains the files in the following figure.

Figure 30. BSP `src` folder



9. Modify the target_sources in BSP `CMakeLists.txt`.
   a. Add the following C files in the PRIVATE column:

```
drivers/src/altera_s10_mailbox_client.c
drivers/src/altera_s10_mailbox_client_flash.c
drivers/src/altera_s10_mailbox_client_flash_rsu.c
drivers/src/altera_s10_mailbox_client_rsu.c
drivers/src/librsu.c
drivers/src/librsu_cb.c
drivers/src/librsu_cfg.c
drivers/src/librsu_ll_qspi.c
drivers/src/librsu_misc.c
drivers/src/rsu_client.c
…
```

b. Add the header files in the PUBLIC column.

```
drivers/inc/altera_s10_mailbox_client.h
drivers/inc/altera_s10_mailbox_client_flash.h
drivers/inc/altera_s10_mailbox_client_flash_rsu.h
drivers/inc/altera_s10_mailbox_client_regs.h
drivers/inc/altera_s10_mailbox_client_rsu.h
drivers/inc/librsu.h
drivers/inc/librsu_cb.h
drivers/inc/librsu_cfg.h
drivers/inc/librsu_ll.h
drivers/inc/librsu_misc.h
drivers/inc/rsu_client.h
…
```

10. Modify the `alt_sys_init.c` in the following sections:
    - Device headers.
      - `#include "altera_s10_mailbox_client.h"`
    - Allocate the device storage.
      - ALTERA_S10_MAILBOX_CLIENT_INSTANCE (MAILBOX_CLIENT, mailbox_client);
    - Initialize non-interrupt controller device.
      - ALTERA_S10_MAILBOX_CLIENT_INIT (MAILBOX_CLIENT, mailbox_client);
11. Proceed to compile the software project.

14. Read-Only Zip File System

Intel FPGA provides a read-only zip file system for use with the hardware abstraction layer (HAL) as a software package. The read-only zip file system provides access to a simple file system stored in flash memory. This file system is suitable for embedded software use.

The drivers use the HAL generic device driver framework for file subsystems. Therefore, you can access the zip file subsystem using the ANSI C standard library I/O functions, such as `fopen()` and `fread()`.

The supported memories are General Purpose QSPI Flash, Configuration Flash in Control-block Based Devices, Active Serial (AS) Configuration Flash in SDM Based Devices, on-chip memory, and external memory.

14.1. Read-Only Zip File System in BSP Editor

The read-only zip file system is supported by Nios V processor software development flows. To use the zip file system, use the Board Support Package (BSP) Editor to include the software package

You must specify the following parameters in the BSP Editor to configure the file system:

Table 52. Read-Only Zip File System Parameters

Parameters	Memory	
	SDM AS Configuration Flash	Other
Name of the mount point for the file system (e.g., `/mnt/rozipfs`)	Yes	Yes
Offset from the memory base address	Yes	Yes
Base address of the memory in Platform Designer	No	Yes
Enable Mailbox Client IP driver	Yes	No
Name of the Mailbox Client IP from `system.h` (e.g., `/dev/mailbox_ip`)	Yes	No

© Altera Corporation. Altera, the Altera logo, the 'a' logo, and other Altera marks are trademarks of Altera Corporation. Altera and Intel warrant performance of its FPGA and semiconductor products to current specifications in accordance with Altera's or Intel's standard warranty as applicable, but reserves the right to make changes to any products and services at any time without notice. Altera and Intel assume no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to inwriting by Altera or Intel. Altera and Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

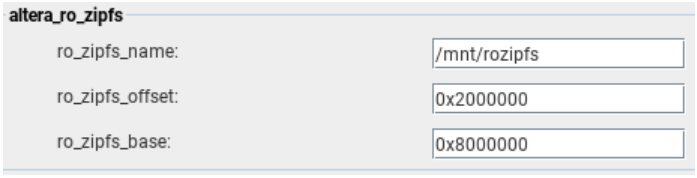
Figure 31. Configuring Read-Only Zip File System with SDM AS Configuration Flash



Note: Assumption:

- Using SDM AS configuration flash.
- Read-only zip file system starts from 0x2000000 (Offset of 0x2000000 from flash base address).

Figure 32. Configuring Read-Only Zip File System with Other Memories



Note: Assumption:

- Using other memories.
- Memory starts from 0x8000000 in Platform Designer.
- Read-only zip file system starts from 0xA000000 (Offset of 0x2000000 from memory base address).

After you make these settings, the BSP Editor includes and links the file subsystem in the project. After you rebuild the project, the `system.h` file reflects the presence of this software package in the system.

To read the `example.txt`, use the command as below.

```
fopen("/mnt/rozipfs/example.txt", "r");
```

14.2. Preparing the Zip File

Ensure that the zip file is uncompressed. The read-only zip file system uses the zip format only for bundling files; it does not provide the file decompression features for which zip utilities are known.

Create a zip file without compression using the WinZip GUI or the `-e0` option to turn off compression using either `winzip` or `pkzip` from a command line. When using the ZIP command in Linux, use the `-0` option to turn off compression.

```
$ zip -r example.zip -0 example.txt
```

14.3. Programming the Zip File to Flash

To access files in the zip file subsystem, you must load program the zip data into flash. The RISC-V GNU objcopy utility (`riscv32-unknown-elf-objcopy.exe`) creates an Intel HEX file (`.hex`) that includes the data for the zip file system.

```
$ riscv32-unknown-elf-objcopy --input-target binary \
--output-target ihex example.zip example.hex
```

You can load the HEX file in general-purpose QSPI flash or configuration flash.

14.3.1. General Purpose QSPI Flash

For general purpose QSPI flash, you need to instantiate the **Generic Serial Flash Interface IP** in the processor system to read the file system. Use a third-party flash programmer to program the HEX file system into the generic purpose QSPI flash.

14.3.2. Active Serial Configuration Flash in Control-block Based Devices

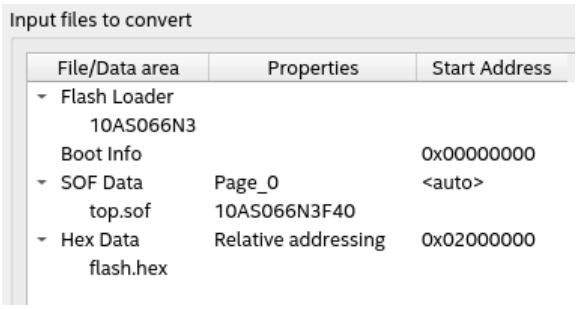
For Active Serial configuration flash in control-block based devices, you need to instantiate the Generic Serial Flash Interface Intel FPGA IP in the processor system to read the file system.

During JIC generation in **Convert Programming File**, you can attach the HEX file system as **Hex Data** with the following settings:

- **Relative Addressing** with specified start address (Offset from memory base address)
- **Big Endian**

You can program the JIC file using the Quartus Prime Programmer.

Figure 33. Attaching HEX File System as Convert Programming File



Note: Assume the *offset from the memory base address* in the BSP Editor is 0x2000000.

Related Information

Generic Serial Flash Interface Intel® FPGA IP User Guide

14.3.3. Active Serial Configuration Flash in SDM Based Devices

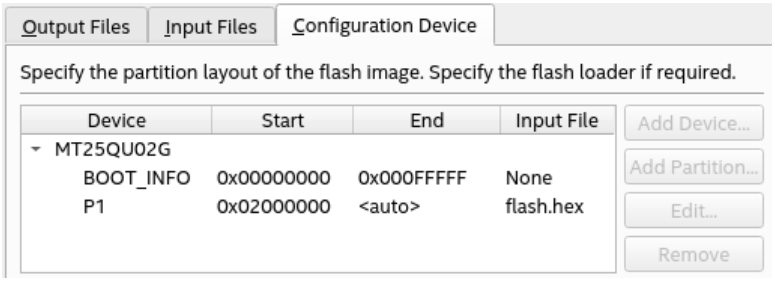
For SDM AS configuration flash, you need to instantiate the Mailbox Client Intel FPGA IP in the processor system to read the file system.

During JIC generation in **Programming File Generator**, you can attach the HEX file system as **Hex Data** with the following settings,

- **Start Addressing** with specified start address (Offset from memory base address)
- **Bit Swap**

And program the JIC file using the Quartus Prime Programmer.

Figure 34. Attaching HEX File System in Programming File Generator



Assumption: The offset from the memory base address is 0x2000000.

Related Information

- Mailbox Client Intel® FPGA IP User Guide
- Quartus® Prime Pro Edition User Guide: Programmer - Generating Secondary Programming Files (Programming File Generator)

14.4. Programming the Zip File to On-Chip ROM

To access files in the zip file subsystem, you must first program the zip data into the On-Chip ROM Intel FPGA IP (OCROM). To memory initialize the OCROM, you need an Altera-style HEX file (`.hex`), which includes the data for the zip file system.

You need the RISC-V GNU objcopy, RISC-V GNU linker, and elf2hex utility to generate an Altera-style HEX file.

```
$ riscv32-unknown-elf-objcopy --input-target binary \
--output-target elf32-littleriscv example.zip example.o

$ riscv32-unknown-elf-ld -e <OCROM Base Address> \
--Ttext <OCROM Base Address> -o example.elf example.o

$ elf2hex example.elf -b <OCROM Base Address> \
-e <OCROM End Address> -r <OCROM Data Width in bytes> \
-w <OCROM Data Width in bits> -o example.hex
```

Related Information

- On-Chip Memory II (RAM or ROM) Intel FPGA IP
- GNU RISC-V Embedded GCC Toolchain Reference on page 295

15. Publishing Component Information to Embedded Software

You can publish component information for use by software, such as a C compiler and a Board Support Package (BSP) generator. Information used by a C compiler might be a set of `#define` statements that describe some aspect of a component. Information used by a BSP Editor might be the identification of memory components, so that the BSP generator can create a linker script.

15.1. Embedded Component Information Flow

A Platform Designer IP component publishes information by including Tcl assignment statements in its component description file, `<component_name>_hw.tcl`. Each assignment is a name-value pair that can be associated with the entire component, or with a single interface. When the assignment statement applies to the entire component, it is set using the `set_module_assignment` command. Assignment statements that apply to an interface are set using the `set_interface_assignment` command.

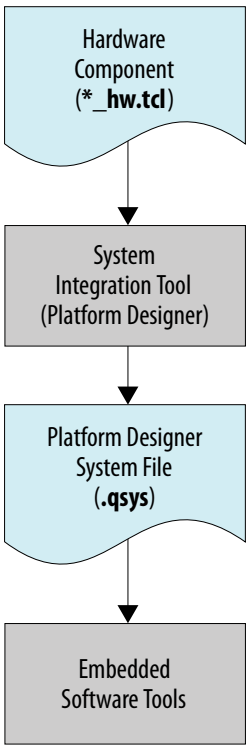
```
# These assignments apply to the entire component
# This is the syntax for the set_module_assignment command:
# set_module_assignment <assignment_name> <value>
# Here are 3 examples
set_module_assignment embeddedsw.CMacro.colorSpace "CMYK"
set_module_assignment embeddedsw.configuration.cpuArchitecture "My processor"
set_module_assignment embeddedsw.memoryInfo.IS_FLASH 1
```

```
# This is the syntax of the set_interface_assignment command:
# set_interface_assignment <interface_name> <assignment_name> <value>
# Here is an example
set_interface_assignment lcd0 embeddedsw.configuration.isPrintableDevice 1
```

© Altera Corporation. Altera, the Altera logo, the 'a' logo, and other Altera marks are trademarks of Altera Corporation. Altera and Intel warrant performance of its FPGA and semiconductor products to current specifications in accordance with Altera's or Intel's standard warranty as applicable, but reserves the right to make changes to any products and services at any time without notice. Altera and Intel assume no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to inwriting by Altera or Intel. Altera and Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

Figure 35. Embedded Component Information Flow



15.2. Embedded Software Assignments

Embedded software assignments are organized in a period-separated namespace. All of the assignments for embedded software tools have the prefix `embeddedsw.` The `embeddedsw` namespace is further divided into the following three sub-namespaces:

- C Macro—Assignment name prefix `embeddedsw.CMacro`
- Configuration—Assignment name prefix `embeddedsw.configuration`
- Memory Initialization—Assignment name prefix `embeddedsw.memoryInfo`

15.2.1. C Macro Namespace

You can use the C macro namespace to publish information about your component that is converted to `#define`'s in a C or C++ `system.h` file. C macro assignments are associated with the entire hardware component, not with individual interfaces.

The name of an assignment in the C macro namespace is `embeddedsw.CMacro.`<*assignmentName*>. You must format the value as a legal C or C++ expression.

Example 30. Assignment Statement for the BAUD_RATE of uart_0 in a Hardware System

```
# Tcl assignment statement included in the _hw.tcl file
add_parameter BAUD_RATE_PARAM integer 9600 "This is the default baud rate."
# Dynamically reassign the baud rate based on the parameter value
set_module_assignment embeddedsw.CMacro.BAUD_RATE \
[get_parameter_value BAUD_RATE_PARAM]
```

15.2.1.1. Generated Macro in system.h

Example 31. Generated Macro in system.h After Dynamic Reassignment

```
/* Generated macro in the system.h file after dynamic reassignment */
#define UART_0_BAUD_RATE 15200
```

Related Information

GCC, the GNU Compiler Collection

15.2.1.2. GCC C/C++ 32-bit Processor Constants

Table 53. Examples of How to Format GCC C/C++ 32-bit Processor Constants

C Data Type	Examples
boolean (`char`, `short`, `int`)	`1, 0`
32-bit signed integer (`int`, `long`)	`123, -50`
32-bit unsigned integer (`unsigned int`, `unsigned long`)	`123u, 0xef8472a0`
64-bit signed integer (`long long int`)	`4294967296LL, -4294967296LL`
64-bit unsigned integer (`unsigned long long int`)	`4294967296ULL, 0xac458701fd64ULL`
32-bit floating-point (`float`)	`3.14f`
64-bit floating-point (`double`)	`2.78, 314e-2`
character (`char`)	`'x'`
string (`const char*`)	`"Hello World!"`

15.2.2. Configuration Namespace

You can use the configuration namespace to pass configuration information to embedded software tools. You can associate configuration namespace assignments with the entire component or with a single interface.

The assignment name for the configuration namespace is `embeddedsw.configuration.<name>`. Intel FPGA's embedded software tools already have definitions for the data types of the configuration names listed in this section.

15.2.2.1. Configuration Data Types

Table 54. Configuration Data Types and Corresponding Format

Configuration Data Type	Format
boolean	1, 0
32-bit integer	123, -50
64-bit integer	4294967296, -4294967296
32-bit floating-point	3.14
64-bit floating-point	2.78, 314e-2
string	ABC

15.2.2.2. Component Configuration Information

Table 55. Component Configuration Information - Assign with set_module_assignment

Configuration Name	Type	Default	Meaning	Example
`cpuArchitecture`	string	—	Processor instruction set architecture. Provide this assignment if you want your component to be considered a processor.	My 8051
`requiredDriver`	boolean	0	If this configuration is 1 (true), the component requires a software driver. Software tools are expected to generate a warning if no driver is found.	1

15.2.2.3. Memory-Mapped Slave Information

Table 56. Avalon Memory-Mapped Slave Information - Assign with set_interface_assignment

Configuration Name	Type	Default	Meaning	Examples
`isMemoryDevice`	boolean	0	The slave port provides access to a memory device.	Intel FPGA® On-Chip Memory Component, DDR Controller,Quad Serial Peripheral Interface (QSPI) flash device controller
`isPrintableDevice`	boolean	0	The slave port provides access to a character-based device.	Intel FPGA UART, Intel FPGA JTAG UART
`isTimerDevice`	boolean	0	The slave port provides access to a timer device.	Intel FPGA Timer
`isEthernetMacDevice`	boolean	0	The slave port provides access to an Ethernet media access control (MAC).	Intel FPGA Triple-Speed Ethernet
`isNonVolatileStorage`	boolean	0	The memory device is a non-volatile memory device. The contents of a non-volatile memory device are fixed and always present. In normal operation, you can only read from	QSPI flash devices, on-chip FPGA memory configured as a ROM

continued...

Configuration Name	Type	Default	Meaning	Examples
			this memory. If this property is true, you must also set `isMemoryDevice` to true.	
`isFlash`	boolean	0	The memory device is a flash memory device. If `isFlash` is true, you must also set `isMemoryDevice` and `isNonVolatileStorage` to true.	QSPI flash devices
`affectsTransactionsOnMasters`	string	empty string	A list of master names delimited by spaces, for example `m1 m2`. Used when the slave port provides access to Avalon-MM control registers in the component. The control registers control transfers on the specified master ports. The slave port can configure the control registers for master ports on the listed components. The address space for this slave port is composed of the address spaces of the named master ports. Nios V embedded software tools use this information to generate `#define` directives describing the address space of these master ports.	Intel FPGA direct memory access (DMA), Modular Scatter-Gather DMA

16. Nios V Processor Appendix

16.1. HAL API

16.1.1. HAL API Reference

This chapter provides an alphabetically ordered list of all the functions in the hardware abstraction layer (HAL) application program interface (API). Each function is listed with its C prototype and a short description.

Related Information

- Timestamp Driver on page 98
- Boot Modes on page 125
- Nios® V Processor Configuration and Booting Solutions
  For more details about Nios® V processor booting method.
- Timestamp Timer on page 47

16.1.1.1. _exit()

Prototype

```
void _exit (int exit_code)
```

Commonly Called By

newlib C library

Thread-safe

Yes.

Available from ISR

No.

Include

```
<unistd.h>
```

Description

The newlib `exit()` function calls the `_exit()` function to terminate the current process. Typically, `exit()` calls this function when `main()` completes. Because there is only a single process in HAL systems, the HAL implementation blocks forever.

Interrupts are not disabled, so ISRs continue to execute.

The input argument, `exit_code`, is ignored.

© Altera Corporation. Altera, the Altera logo, the 'a' logo, and other Altera marks are trademarks of Altera Corporation. Altera and Intel warrant performance of its FPGA and semiconductor products to current specifications in accordance with Altera's or Intel's standard warranty as applicable, but reserves the right to make changes to any products and services at any time without notice. Altera and Intel assume no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to inwriting by Altera or Intel. Altera and Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

Return

--

16.1.1.2. _rename()

Prototype

```
int _rename(char *existing, char* new)
```

Commonly Called By

newlib C library

Thread-safe

Yes.

Available from ISR

Yes.

Include

```
<stdio.h>
```

Description

`_rename()` is unsupported in the HAL environment. This function has no effect and is implemented purely for `newlib` compatibility.

Return

It always returns -1, with errno set to ENOSYS.

16.1.1.3. alt_dcache_flush()

Prototype

```
void alt_dcache_flush (void* start, alt_u32 len)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

Yes.

Available from ISR

Yes.

Include

```
<sys/alt_cache.h>
```

Description

The `alt_dcache_flush()` function flushes the data cache for a memory region of length `len` bytes, starting at address start. Flushing the cache consists of writing back dirty data and then invalidating the cache.

In processors without data caches, it has no effect.

Return

--

16.1.1.4. alt_dcache_flush_all()

Prototype

```
void alt_dcache_flush_all (void)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

Yes.

Available from ISR

Yes.

Include

```
<sys/alt_cache.h>
```

Description

The `alt_dcache_flush_all()` function flushes, that is, writes back dirty data and then invalidates, the entire contents of the data cache.

In processors without data caches, it has no effect.

Return

--

16.1.1.5. alt_icache_flush_all()

Prototype

```
void alt_icache_flush_all (void)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

Yes.

Available from ISR

Yes.

Include

```
<sys/alt_cache.h>
```

Description

The `alt_icache_flush_all()` function invalidates the entire contents of the instruction cache.

In processors without instruction caches, it has no effect.

Return

--

16.1.1.6. alt_dcache_flush_no_writeback()

Prototype

```
void alt_dcache_flush_no_writeback (void* start, alt_u32 len)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

Yes.

Available from ISR

Yes.

Include

`<sys/alt_cache.h>`

Description

The `alt_dcache_flush_no_writeback()` is called to flush the data cache for a memory region of length `len` bytes, starting at address `start`. Any dirty lines in the data cache are NOT written back to memory. The cache becomes invalidated. In processors without data caches, it has no effect.

Return

--

16.1.1.7. alt_icache_flush_all()

Prototype

```
void alt_icache_flush_all (void)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

Yes.

Available from ISR

Yes.

Include

`<sys/alt_cache.h>`

Description

The `alt_icache_flush_all()` function invalidates the entire contents of the instruction cache.

In processors without instruction caches, it has no effect.

Return

--

16.1.1.8. alt_icache_flush()

Prototype

```
void alt_icache_flush (void* start, alt_u32 len)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

Yes.

Available from ISR

Yes.

Include

`<sys/alt_cache.h>`

Description

The `alt_icache_flush()` function invalidates the instruction cache for a memory region of length `len` bytes, starting at address `start`.

In processors without instruction caches, it has no effect.

Return

--

16.1.1.9. alt_alarm_start()

Prototype

```
int alt_alarm_start

( alt_alarm* alarm,

alt_u32 nticks,
```

```
alt_u32 (*callback) (void* context),
void* context )
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

Yes.

Available from ISR

Yes.

Include

`<sys/alt_alarm.h>`

Description

The `alt_alarm_start()` function schedules an alarm callback.

The HAL waits `nticks` system clock ticks before calling the `callback()` function. When the HAL calls `callback()`, it passes it the input argument `context`. The HAL does not use the `context` parameter. It only passes it as a parameter to the `callback()` function.

The `alarm` argument is a pointer to a structure that represents this alarm. You must create it, and it must have a lifetime that is at least as long as that of the alarm. However, you are not responsible for initializing the contents of the structure pointed to by `alarm`. This action is done by the call to `alt_alarm_start()`.

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

Return

The return value for `alt_alarm_start()` is zero on success, and negative otherwise. This function fails if there is no system clock available.

16.1.1.10. alt_alarm_stop()

Prototype

```
void alt_alarm_stop (alt_alarm* alarm)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

Yes.

Available from ISR

Yes.

Include

`<sys/alt_alarm.h>`

Description

You can call the `alt_alarm_stop()` function to cancel an alarm previously registered by a call to `alt_alarm_start()`. The input argument is a pointer to the alarm structure in the previous call to `alt_alarm_start()`.

On return the alarm is canceled, if it is still active.

Return

--

16.1.1.11. alt_dma_rxchan_depth()

Prototype

```
alt_u32 alt_dma_rxchan_depth(alt_dma_rxchan dma)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

Yes.

Available from ISR

No.

Include

`<sys/alt_dma.h>`

Description

The `alt_dma_rxchan_depth()` function returns the maximum number of receive requests that can be posted to the specified DMA transmit channel, `dma`.

Whether this function is thread-safe, or can be called from an ISR, depends on the underlying device driver. In general it safest to assume that it is not thread-safe.

Return

Returns the maximum number of receive requests that can be posted.

16.1.1.12. alt_dma_rxchan_close()

Prototype

```
int alt_dma_rxchan_close (alt_dma_rxchan rxchan)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

Yes.

Available from ISR

No.

Include

`<sys/alt_dma.h>`

Description

The `alt_dma_rxchan_close()` function notifies the system that the application has finished using the direct memory access (DMA) receive channel, `rxchan`. The current implementation always succeeds.

Return

The return value is always zero.

16.1.1.13. alt_dev_reg()

Prototype

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

Commonly Called By

Device drivers

Thread-safe

No.

Available from ISR

No.

Include

`<sys/alt_dev.h>`

Description

The `alt_dev_reg()` function registers a device with the system. After it is registered, you can access a device using the standard I/O functions.

For more information, refer to the "Developing Programs Using the Hardware Abstraction Layer" section.

The system behavior is undefined in the event that a device is registered with a name that conflicts with an existing device or file system.

The `alt_dev_reg()` function is not thread-safe in the sense that no other thread can use the device list at the time that `alt_dev_reg()` is called. Call `alt_dev_reg()` only in the following circumstances:

- When running in single-threaded mode.
- From a device initialization function called by `alt_sys_init()`. `alt_sys_init()` may only be called by the single-threaded C startup code.

Return

The return value is zero upon success. A negative return value indicates failure.

16.1.1.14. alt_dma_rxchan_open()

Prototype

```
alt_dma_rxchan alt_dma_rxchan_open (const char* name)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

Yes.

Available from ISR

No.

Include

```
<sys/alt_dma.h>
```

Description

The `alt_dma_rxchan_open()` function obtains an `alt_dma_rxchan` descriptor for a DMA receive channel. The input argument, name, is the name of the associated physical device, for example, `/dev/dma_0`.

Return

The return value is the requested `alt_dma_rxchan` descriptor on success and null otherwise. If an error occurs, `errno` is set to ENODEV.

16.1.1.15. alt_dma_rxchan_prepare()

Prototype

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

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

See description.

Available from ISR

See description.

Include

`<sys/alt_dma.h>`

Description

The `alt_dma_rxchan_prepare()` posts a receive request to a DMA receive channel. The input arguments are: `dma`, the channel to use; `data`, a pointer to the location that data is to be received to; `length`, the maximum length of the data to receive in bytes; `done`, callback function that is called after the data is received; `handle`, an opaque value passed to `done`.

Whether this function is thread-safe, or can be called from an ISR, depends on the underlying device driver. In general it safest to assume that it is not thread-safe.

Return

The return value is zero upon success. A negative return value indicates that the request cannot be posted.

16.1.1.16. alt_dma_rxchan_reg()

Prototype

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

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

No.

Available from ISR

No.

Include

`<sys/alt_dma_dev.h>`

Description

The `alt_dma_rxchan_reg()` function registers a DMA receive channel with the system.

After it is registered, a device can be accessed using the functions described in “Using DMA Devices” in the "Developing Programs Using the Hardware Abstraction Layer" section.

System behavior is undefined in the event that a channel is registered with a name that conflicts with an existing channel.

The `alt_dma_rxchan_reg()` function is not thread-safe if other threads are using the channel list at the time that `alt_dma_rxchan_reg()` is called. Call `alt_dma_rxchan_reg()` only in the following circumstances:

- When running in single-threaded mode.
- From a device initialization function called by `alt_sys_init()`. `alt_sys_init()` may only be called by the single-threaded C startup code.

Return

The return value is zero upon success. A negative return value indicates failure.

16.1.1.17. alt_dma_txchan_close()

Prototype

```
int alt_dma_txchan_close (alt_dma_txchan txchan)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

Yes.

Available from ISR

No.

Include

```
<sys/alt_dma.h>
```

Description

The `alt_dma_txchan_close` function notifies the system that the application has finished using the DMA transmit channel, `txchan`. The current implementation always succeeds.

Return

The return value is zero.

16.1.1.18. alt_dma_txchan_ioctl()

Prototype

```
int alt_dma_txchan_ioctl (alt_dma_txchan dma,
int req,
void* arg)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

See description.

Available from ISR

See description.

Include

`<sys/alt_dma.h>`

Description

The `alt_dma_txchan_ioctl()` function performs device specific I/O operations on the DMA transmit channel, `dma`. For example, some drivers support options to control the width of the transfer operations. The input argument, `req`, is an enumeration of the requested operation; `arg` is an additional argument for the request. The interpretation of `arg` is request dependent.

For more information, refer to the *Generic Requests DMA Might Support* table for the generic requests a device might support.

Whether a call to `alt_dma_txchan_ioctl()` is thread-safe, or can be called from an ISR, is device dependent. In general it safest to assume that it is not thread-safe.

Do not call the `alt_dma_txchan_ioctl()` function while DMA transfers are pending or can result in unpredictable behavior.

Return

A negative return value indicates failure; otherwise the interpretation of the return value is request specific.

16.1.1.19. alt_dma_txchan_open()

Prototype

```
alt_dma_txchan alt_dma_txchan_open (const char* name)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

Yes.

Available from ISR

No.

Include

`<sys/alt_dma.h>`

Description

The `alt_dma_txchan_open()` function obtains an `alt_dma_txchan()` descriptor for a DMA transmit channel. The input argument, `name`, is the name of the associated physical device, for example, `/dev/dma_0`.

Return

The return value is null on failure and the request `alt_dma_txchan` otherwise. If an error occurs, `errno` is set to ENODEV.

16.1.1.20. alt_dma_txchan_reg()

Prototype

```
int alt_dma_txchan_reg (alt_dma_txchan_dev* dev)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

No.

Available from ISR

No.

Include

`<sys/alt_dma_dev.h>`

Description

The `alt_dma_txchan_reg()` function registers a DMA transmit channel with the system.

After it is registered, a device can be accessed using the functions described in “Using DMA Devices” in the "Developing Programs Using the Hardware Abstraction Layer" section.

System behavior is undefined in the event that a channel is registered with a name that conflicts with an existing channel.

The `alt_dma_txchan_reg()` function is not thread-safe if other threads are using the channel list at the time that `alt_dma_txchan_reg()` is called. Call `alt_dma_txchan_reg()` only in the following circumstances:

- When running in single-threaded mode.
- From a device initialization function called by `alt_sys_init()`. `alt_sys_init()` may only be called by the single-threaded C startup code.

Return

The return value is zero upon success. A negative return value indicates failure.

16.1.1.21. alt_flash_close_dev()

Prototype

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

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

No.

Available from ISR

No.

Include

```
<sys/alt_flash.h>
```

Description

The `alt_flash_close_dev()` function closes a flash device. All subsequent calls to `alt_write_flash()`, `alt_read_flash()`, `alt_get_flash_info()`, `alt_erase_flash_block()`, or `alt_write_flash_block()` for this flash device fail.

Call the `alt_flash_close_dev()` function only when operating in single-threaded mode.

The only valid values for the `fd` parameter are those returned from the `alt_flash_open_dev` function. If any other value is passed, the behavior of this function is undefined.

Return

--

16.1.1.22. alt_exception_cause_generated_bad_addr()

Prototype

```
int alt_exception_cause_generated_bad_addr ( alt_exception_cause cause)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

--

Available from ISR

--

Include

<sys/alt_exceptions.h>

Description

This function validates the bad_addr argument to an instruction-related exception handler. The function parses the handler's cause argument to determine whether the bad_addr register contains the exception-causing address.

If the exception is of a type that generates a valid address in bad_addr, this function returns a nonzero value. Otherwise, it returns zero.

If the cause register is unimplemented in the Nios V processor core, this function always returns zero.

Return

A nonzero value means bad_addr contains the exception-causing address.

Zero means the value of bad_addr is to be ignored.

16.1.1.23. alt_erase_flash_block()

Prototype

```
int alt_erase_flash_block(alt_flash_fd* fd,

int offset,

int length)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

No.

Available from ISR

No.

Include

<sys/alt_flash.h>

Description

The alt_erase_flash_block() function erases an individual flash erase block. The parameter fd specifies the flash device; offset is the offset within the flash of the block to erase; length is the size of the block to erase. No error checking is performed to check that this is a valid block, or that the length is correct.

For more information, refer to "Using Flash Devices" in the "Developing Programs Using the Hardware Abstraction Layer" chapter.

Call the `alt_erase_flash_block()` function only when operating in single-threaded mode.

The only valid values for the `fd` parameter are those returned from the `alt_flash_open_dev` function. If any other value is passed, the behavior of this function is undefined.

Return

The return value is zero upon success. A negative return value indicates failure.

16.1.1.24. alt_dma_rxchan_ioctl()

Prototype

```
int alt_dma_rxchan_ioctl (alt_dma_rxchan dma,

int req,

void* arg)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

See description.

Available from ISR

See description.

Include

`<sys/alt_dma.h>`

Description

The `alt_dma_rxchan_ioctl()` function performs DMA I/O operations on the DMA receive channel, dma. The I/O operations are device specific. For example, some DMA drivers support options to control the width of the transfer operations. The input argument, req, is an enumeration of the requested operation; `arg` is an additional argument for the request. The interpretation of `arg` is request dependent.

Whether a call to `alt_dma_rxchan_ioctl()` is thread-safe, or can be called from an ISR, is device dependent. In general it safest to assume that it is not thread-safe.

Do not call the `alt_dma_rxchan_ioctl()` function while DMA transfers are pending or can result in unpredictable behavior.

For device-specific information about the Intel FPGA® DMA controller core, refer to the "DMA Controller Core" chapter in the *Embedded Peripherals IP User Guide*.

Return

A negative return value indicates failure. The interpretation of nonnegative return values is request specific.

Table 57. Generic Requests DMA Might Support

Request	Meaning
ALT_DMA_SET_MODE_8	Transfer data in units of 8 bits. The value of arg is ignored.
ALT_DMA_SET_MODE_16	Transfer data in units of 16 bits. The value of arg is ignored.
ALT_DMA_SET_MODE_32	Transfer data in units of 32 bits. The value of arg is ignored.
ALT_DMA_SET_MODE_64	Transfer data in units of 64 bits. The value of arg is ignored.
ALT_DMA_SET_MODE_128	Transfer data in units of 128 bits. The value of arg is ignored.
ALT_DMA_GET_MODE	Return the transfer width. The value of arg is ignored.
ALT_DMA_TX_ONLY_ON	The ALT_DMA_TX_ONLY_ON request causes a DMA channel to operate in a mode in which only the transmitter is under software control. The other side writes continuously from a single location. The address to which to write is the argument to this request.
ALT_DMA_TX_ONLY_OFF	Return to the default mode, in which both the receive and transmit sides of the DMA can be under software control.
ALT_DMA_RX_ONLY_ON	The ALT_DMA_RX_ONLY_ON request causes a DMA channel to operate in a mode in which only the receiver is under software control. The other side reads continuously from a single location. The address to read is the argument to this request.
ALT_DMA_RX_ONLY_OFF	Return to the default mode, in which both the receive and transmit sides of the DMA can be under software control.

16.1.1.25. alt_dma_txchan_space()

Prototype

```
int alt_dma_txchan_space (alt_dma_txchan dma)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

See description.

Available from ISR

See description/

Include

<sys/alt_dma.h>

Description

The alt_dma_txchan_space() function returns the number of transmit requests that can be posted to the specified DMA transmit channel, dma. A negative value indicates that the value cannot be determined.

Whether this function is thread-safe, or can be called from an ISR, depends on the underlying device driver. In general it safest to assume that it is not thread-safe.

Return

Returns the number of transmit requests that can be posted.

16.1.1.26. alt_dma_txchan_send()

Prototype

```
int alt_dma_txchan_send (alt_dma_txchan dma,

const void* from,

alt_u32 length,

alt_txchan_done* done,

void* handle)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

See description.

Available from ISR

See description.

Include

```
<sys/alt_dma.h>
```

Description

The `alt_dma_txchan_send()` function posts a transmit request to a DMA transmit channel. The input arguments are: `dma`, the channel to use; from, a pointer to the start of the data to send; `length`, the length of the data to send in bytes; `done`, a callback function that is called after the data is sent; and `handle`, an opaque value passed to done.

Whether this function is thread-safe, or can be called from an ISR, depends on the underlying device driver. In general it safest to assume that it is not thread-safe.

Return

The return value is negative if the request cannot be posted, and zero otherwise.

16.1.1.27. alt_flash_open_dev()

Prototype

```
alt_flash_fd* alt_flash_open_dev(const char* name)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

No.

Available from ISR

No.

Include

`<sys/alt_flash.h>`

Description

The `alt_flash_open_dev()` function opens a flash device. After it is opened, you can perform the following operations:

- Write to a flash device using `alt_write_flash()`
- Read from a flash device using `alt_read_flash()`
- Control individual flash blocks using `alt_get_flash_info()`, `alt_erase_flash_block()`, or `alt_write_flash_block()`.

  Call the `alt_flash_open_dev` function only when operating in single-threaded mode.

Return

The return value is zero upon failure. Any other value indicates success.

16.1.1.28. alt_fs_reg()

Prototype

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

Commonly Called By

Device drivers

Thread-safe

No.

Available from ISR

No.

Include

`<sys/alt_dev.h>`

Description

The `alt_fs_reg()` function registers a file system with the HAL. After it is registered, a file system can be accessed using the standard I/O functions.

For more information, refer to the "Developing Programs Using the Hardware Abstraction Layer" section.

System behavior is undefined in the event that a file system is registered with a name that conflicts with an existing device or file system.

`alt_fs_reg()` is not thread-safe if other threads are using the device list at the time that `alt_fs_reg()` is called. Call `alt_fs_reg()` only in the following circumstances:

- When running in single-threaded mode.
- From a device initialization function called by `alt_sys_init()`. `alt_sys_init()` may only be called by the single-threaded C startup code.

Return

The return value is zero upon success. A negative return value indicates failure.

16.1.1.29. alt_get_flash_info()

Prototype

```
int alt_get_flash_info(alt_flash_fd* fd,

flash_region** info,

int* number_of_regions)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

No.

Available from ISR

No.

Include

`<sys/alt_flash.h>`

Description

The `alt_get_flash_info()` function gets the details of the erase region of a flash part. The flash part is specified by the descriptor `fd`, a pointer to the start of the `flash_region` structures is returned in the `info` parameter, and the number of flash regions are returned in number of regions.

Call this function only when operating in single-threaded mode.

The only valid values for the `fd` parameter are those returned from the `alt_flash_open_dev` function. If any other value is passed, the behavior of this function is undefined.

Return

The return value is zero upon success. A negative return value indicates failure.

16.1.1.30. alt_ic_irq_disable()

Prototype

```
int alt_ic_irq_disable (alt_u32 ic_id, alt_u32 irq)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

Yes.

Available from ISR

No.

Include

```
<sys/alt_irq.h>
```

Description

The `alt_ic_irq_disable()` function disables a single interrupt.

The function arguments are as follows:

- `ic_id` is the interrupt controller identifier (ID) as defined in `system.h`, identifying the external interrupt controller in the daisy chain. This argument is ignored if the external interrupt controller interface is not implemented.
- `irq` is the interrupt request (IRQ) number, as defined in `system.h`, identifying the interrupt to enable.

Return

This function returns zero if successful, or nonzero otherwise. The function fails if the `irq` parameter is greater than the maximum interrupt port number supported by the interrupt controller.

16.1.1.31. alt_ic_irq_enabled()

Prototype

```
int alt_ic_irq_enabled (alt_u32 ic_id, alt_u32 irq)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

Yes.

Available from ISR

Yes.

Include

```
<sys/alt_irq.h>
```

Description

This function determines whether a specified interrupt is enabled.

The function arguments are as follows:

- `ic_id` is the interrupt controller ID as defined in `system.h`, identifying the external interrupt controller in the daisy chain. This argument is ignored if the external interrupt controller interface is not implemented.
- `irq` is the IRQ number, as defined in `system.h`, identifying the interrupt to enable.
- A driver for an external interrupt controller must implement this function.

Return

Returns zero if the specified interrupt is disabled, and nonzero otherwise.

16.1.1.32. alt_ic_isr_register()

Prototype

```
int alt_ic_isr_register (alt_u32 ic_id,

alt_u32 irq,

alt_isr_func isr,

void* isr_context,

void* flags)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

Yes.

Available from ISR

No.

Include

```
<sys/alt_irq.h>
```

Description

The `alt_ic_isr_register()` function registers an ISR. If the function is successful, the requested interrupt is enabled on return, and `isr` and `isr_context` are inserted in the vector table.

The function arguments are as follows:

- `ic_id` is the interrupt controller ID as defined in `system.h`, identifying the external interrupt controller in the daisy chain. This argument is ignored if the external interrupt controller interface is not implemented.
- `irq` is the IRQ number, as defined in `system.h`, identifying the interrupt to register.
- `isr` is the function that is called when the interrupt is accepted.
- `isr_context` is the input argument to `isr`. `isr_context` points to a data structure associated with the device driver instance.
- `flags` is reserved.

  The ISR function prototype is defined as follows:

  ```
  typedef void (*alt_isr_func) (void* isr_context);
  ```

  Calls to `alt_ic_isr_register()` replace previously registered handlers for interrupt `irq`.

  If `isr` is set to null, the interrupt is disabled.

Return

This function returns zero if successful, or nonzero otherwise. The function fails if the `irq` parameter is greater than the maximum interrupt port number supported by the interrupt controller.

16.1.1.33. alt_ic_irq_enable()

Prototype

```
int alt_ic_irq_enable (alt_u32 ic_id, alt_u32 irq)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

Yes.

Available from ISR

No.

Include

```
<sys/alt_irq.h>
```

Description

The `alt_ic_irq_enable()` function enables a single interrupt.

The function arguments are as follows:

- `ic_id` is the interrupt controller ID as defined in `system.h`, identifying the external interrupt controller in the daisy chain. This argument is ignored if the external interrupt controller interface is not implemented.
- `irq` is the IRQ number, as defined in `system.h`, identifying the interrupt to enable.

Return

This function returns zero if successful, or nonzero otherwise. The function fails if the `irq` parameter is greater than the maximum interrupt port number supported by the interrupt controller.

16.1.1.34. alt_instruction_exception_register()

Prototype

```
void alt_instruction_exception_register (

alt_exception_result (*handler)

( alt_exception_cause cause,

alt_u32 exception_pc,

alt_u32 bad_addr ))
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

No.

Available from ISR

Yes.

Include

`<sys/alt_exceptions.h>`

Description

The HAL API function `alt_instruction_exception_register()` registers an instruction-related exception handler. The `handler` argument is a pointer to the instruction-related exception handler.

You can only use this API function if you have enabled the `hal.enable_instruction_related_exceptions_api` setting in the board support package (BSP).

Register the instruction-related exception handler as early as possible in function `main()`. This allows you to handle abnormal conditions during startup.

You can register an exception handler from the `alt_main()` function.

A call to `alt_instruction_exception_register()` replaces the previously registered exception handler, if any. If `handler` is set to null, the instruction-related exception handler is removed.

Return

--

Related Information

Settings Managed by Nios V Processor Board Support Package Editor on page 296
For more comprehensive list of BSP settings for HAL.

16.1.1.35. alt_irq_cpu_enable_interrupts ()

Prototype

```
void alt_irq_cpu_enable_interrupts ()
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

Yes.

Available from ISR

Yes.

Include

```
<sys/alt_irq.h>
```

Description

The `alt_irq_cpu_enable_interrupts ()` function enables the CPU to start taking interrupts.

Return

--

Related Information

Trap Handling on page 154

16.1.1.36. alt_irq_disable_all()

Prototype

```
alt_irq_context alt_irq_disable_all (void)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

Yes.

Available from ISR

No.

Include

`<sys/alt_irq.h>`

Description

The `alt_irq_disable_all()` function disables the processor's interrupt feature by clearing the Machine-mode Interrupt-Enable bit (`mstatus.mie`).

Return

Pass the return value as the input argument to a subsequent call to `alt_irq_enable_all()`.

16.1.1.37. alt_irq_enable_all()

Prototype

```
void alt_irq_enable_all (alt_irq_context context)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

Yes.

Available from ISR

Yes.

Include

`<sys/alt_irq.h>`

Description

The `alt_irq_enable_all()` function enables the processor's interrupt feature that were previously disabled by `alt_irq_disable_all()`. The input argument, `context`, is the value returned by a previous call to `alt_irq_disable_all()`. Using `context` allows nested calls to `alt_irq_disable_all()` and `alt_irq_enable_all()`. As a result, `alt_irq_enable_all()` does not necessarily enable all interrupts, such as interrupts explicitly disabled by `alt_ic_irq_disable()`.

Return

--

16.1.1.38. alt_irq_enabled()

Prototype

```
int alt_irq_enabled (void)
```

Commonly Called By

Device drivers

Thread-safe

Yes.

Available from ISR

Yes.

Include

```
<sys/alt_irq.h>
```

Description

Determines whether interrupts are globally enabled by checking if `mstatus.mie` bit is set.

Return

Returns zero if interrupt-enable bit is disabled, and non-zero otherwise.

Related Information

Platform Interrupt Service Routines on page 170

16.1.1.39. alt_irq_init()

Prototype

```
void alt_irq_init (void* base)
```

Commonly Called By

C/C++ programs

Thread-safe

Yes.

Available from ISR

Yes.

Include

```
<sys/alt_irq.h>
```

Description

The `alt_irq_init()` function calls the initialization macros for all interrupt controllers in the system at config time, before any other non-interrupt controller driver is initialized. The `base` parameter is ignored and only present for backwards-compatibility. Altera recommended that `NULL` is passed in for the `base` parameter.

Return

--

16.1.1.40. alt_irq_pending ()

Prototype

```
void alt_irq_pending (void)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

Yes.

Available from ISR

Yes.

Include

```
<sys/alt_irq.h>
```

Description

The `alt_irq_pending ()` function returns a bit list of the current pending interrupts. This is used by `alt_irq_handler()` to determine which registered interrupt handlers should be called. This routine is only available for the Nios V internal interrupt controller.

Return

--

16.1.1.41. alt_llist_insert()

Prototype

```
void alt_llist_insert(alt_llist* list,

alt_llist* entry)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

No.

Available from ISR

Yes.

Include

`<sys/alt_llist.h>`

Description

The `alt_llist_insert()` function inserts the doubly linked list entry `entry` in the list `list`. This operation is not reentrant. For example, if a list can be manipulated from different threads, or from within both application code and an ISR, some mechanism is required to protect access to the list. Interrupts can be disabled, or in MicroC/OS-II, a `mutex` can be used.

Return

--

16.1.1.42. alt_llist_remove()

Prototype

```
void alt_llist_remove(alt_llist* entry)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

No.

Available from ISR

Yes.

Include

`<sys/alt_llist.h>`

Description

The `alt_llist_remove()` function removes the doubly linked list entry `entry` from the list it is currently a member of. This operation is not reentrant. For example if a list can be manipulated from different threads, or from within both application code and an ISR, some mechanism is required to protect access to the list. Interrupts can be locked, or in MicroC/OS-II, a `mutex` can be used.

Return

--

16.1.1.43. alt_load_section()

Prototype

```
void alt_load_section(alt_u32* from,

alt_u32* to,

alt_u32* end)
```

Commonly Called By

C/C++ programs

Thread-safe

No.

Available from ISR

No.

Include

`<sys/alt_load.h>`

Description

When operating in run-from-flash mode, the sections `.exceptions`, `.rodata`, and `.rwdata` are automatically loaded from the boot device to RAM at boot time. However, if there are any additional sections that require loading, the `alt_load_section()` function loads them manually before these sections are used.

Altera recommends to call below the macro `ALT_LOAD_SECTION_BY_NAME` and before `alt_dcache_flush_all()` and `alt_icache_flush_all()`.

The input argument `from` is the start address in the boot device of the section; `to` is the start address in RAM of the section, and `end` is the end address in RAM of the section.

To load one of the additional memory sections provided by the default linker script, use the macro `ALT_LOAD_SECTION_BY_NAME` rather than calling `alt_load_section()` directly. For example, to load the section `.onchip_ram`, use the following code:

```
ALT_LOAD_SECTION_BY_NAME(onchip_ram);
```

The leading '.' is omitted in the section name. This macro is defined in the header `sys/alt_load.h`.

Return

--

16.1.1.44. alt_nticks()

Prototype

```
alt_u32 alt_nticks (void)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

Yes.

Available from ISR

Yes.

Include

`<sys/alt_alarm.h>`

Description

The `alt_nticks()` function.

Return

Returns the number of elapsed system clock tick since reset. It returns zero if there is no system clock available.

16.1.1.45. alt_read_flash()

Prototype

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

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

No.

Available from ISR

No.

Include

`<sys/alt_flash.h>`

Description

The `alt_read_flash()` function reads data from flash. `length` bytes are read from the flash `fd`, starting `offset` bytes from the beginning of the flash and are written to the location `dest_addr`.

Call this function only when operating in single-threaded mode.

The only valid values for the `fd` parameter are those returned from the `alt_flash_open_dev` function. If any other value is passed, the behavior of this function is undefined.

Return

The return value is zero on success and nonzero otherwise.

16.1.1.46. alt_tick()

Prototype

```
void alt_tick (void)
```

Commonly Called By

Device drivers

Thread-safe

No.

Available from ISR

Yes.

Include

```
<sys/alt_alarm.h>
```

Description

Only the system clock driver may call the `alt_tick()` function. The driver is responsible for making periodic calls to this function at the rate specified in the call to `alt_sysclk_init()`. This function provides notification to the system that a system clock tick has occurred. This function runs as a part of the ISR for the system clock driver.

Return

--

16.1.1.47. alt_ticks_per_second()

Prototype

```
alt_u32 alt_ticks_per_second (void)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

Yes.

Available from ISR

Yes.

Include

```
<sys/alt_alarm.h>
```

Description

The `alt_ticks_per_second()` function returns the number of system clock ticks that elapse per second. If there is no system clock available, the return value is zero.

Return

Returns the number of system clock ticks that elapse per second.

16.1.1.48. alt_timestamp()

Prototype

```
alt_u32 alt_timestamp (void)
```

Commonly Called By

C/C++ programs

Thread-safe

See description.

Available from ISR

See description.

Include

```
<sys/alt_timestamp.h>
```

Description

The `alt_timestamp()` function returns the current value of the timestamp counter.

For more information, refer to “Using Timer Devices” in the "Developing Programs Using the Hardware Abstraction Layer" chapter.

The implementation of this function is provided by the timestamp driver. Therefore, whether this function is thread-safe and or available at interrupt level depends on the underlying driver.

Always call the `alt_timestamp_start()` function before any calls to `alt_timestamp()`. Otherwise the behavior of `alt_timestamp()` is undefined.

Return

Returns the current value of the timestamp counter.

16.1.1.49. alt_timestamp_freq()

Prototype

```
alt_u32 alt_timestamp_freq (void)
```

Commonly Called By

C/C++ programs

Thread-safe

See description.

Available from ISR

See description.

Include

```
<sys/alt_timestamp.h>
```

Description

The `alt_timestamp_freq()` function returns the rate at which the timestamp counter increments.

For more information, refer to “Using Timer Devices” in the "Developing Programs Using the Hardware Abstraction Layer" chapter.

The implementation of this function is provided by the timestamp driver. Therefore, whether this function is thread-safe and or available at interrupt level depends on the underlying driver.

Return

The returned value is the number of counter ticks per second.

16.1.1.50. alt_timestamp_start()

Prototype

```
int alt_timestamp_start (void)
```

Commonly Called By

C/C++ programs

Thread-safe

See description.

Available from ISR

See description.

Include

```
<sys/alt_timestamp.h>
```

Description

The `alt_timestamp_start()` function starts the system timestamp counter.

For more information, refer to “Using Timer Devices” in the "Developing Programs Using the Hardware Abstraction Layer" chapter.

The implementation of this function is provided by the timestamp driver. Therefore, whether this function is thread-safe and or available at interrupt level depends on the underlying driver.

This function resets the counter to zero, and starts the counter running.

Return

The return value is zero on success and nonzero otherwise.

16.1.1.51. alt_write_flash()

Prototype

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

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

No.

Available from ISR

No.

Include

```
<sys/alt_flash.h>
```

Description

The `alt_write_flash()` function writes data to flash. The data to be written is at address `src_addr`. `length` bytes are written to the flash `fd`, `offset` bytes from the beginning of the flash device address space.

Call this function only when operating in single-threaded mode. This function does not preserve any unwritten areas of any flash sectors affected by this write.

The only valid values for the `fd` parameter are those returned from the `alt_flash_open_dev` function. If any other value is passed, the behavior of this function is undefined.

Return

The return value is zero on success and nonzero otherwise.

16.1.1.52. alt_write_flash_block()

Prototype

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

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

No.

Available from ISR

No.

Include

`<sys/alt_flash.h>`

Description

The `alt_write_flash_block()` function writes one block of data to flash. The data to be written is at address `data`. `length` bytes are written to the flash `fd`, into the block starting at offset `block_offset` from the beginning of the flash address space. The data starts at offset `data_offset` from the beginning of the flash address space.

No check is performed on any of the parameters.

Call this function only when operating in single-threaded mode.

The only valid values for the `fd` parameter are those returned from the `alt_flash_open_dev` function. If any other value is passed, the behavior of this function is undefined.

Return

The return value is zero on success and nonzero otherwise.

16.1.1.53. close()

Prototype

```
int close (int fd)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

See description.

Available from ISR

No.

Include

`<unistd.h>`

Description

The `close()` function is the standard UNIX-style `close()` function, which closes the file descriptor `fd`.

Calls to `close()` are thread-safe only if the implementation of `close()` provided by the driver that is manipulated is thread-safe.

Valid values for the `fd` parameter are: `stdout`, `stdin`, and `stderr`, or any value returned from a call to `open()`.

Return

The return value is zero on success, and –1 otherwise. If an error occurs, `errno` is set to indicate the cause.

16.1.1.54. fstat()

Prototype

```
int fstat (int fd, struct stat *st)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

See description.

Available from ISR

No.

Include

```
<sys/stat.h>
```

Description

The `fstat()` function obtains information about the capabilities of an open file descriptor. The underlying device driver fills in the input `st` structure with a description of its functionality. Refer to the header file `sys/stat.h` provided with the compiler for the available options.

By default, file descriptors are marked as character devices, unless the underlying driver provides its own implementation of the `fstat()` function.

Calls to `fstat()` are thread-safe only if the implementation of `fstat()` provided by the driver that is manipulated is thread-safe.

Valid values for the `fd` parameter are: `stdout`, `stdin`, and `stderr`, or any value returned from a call to `open()`.

Return

The return value is zero on success, or –1 otherwise. If the call fails, `errno` is set to indicate the cause of the error.

16.1.1.55. fork()

Prototype

```
pid_t fork (void)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

Yes.

Available from ISR

No.

Include

```
<unistd.h>
```

Description

The `fork()` function is only provided for compatibility with newlib.

Return

Calls to `fork()` always fails with the return code –1 and `errno` set to ENOSYS.

16.1.1.56. fcntl()

Prototype

```
int fcntl(int fd, int cmd)
```

Commonly Called By

C/C++ programs

Thread-safe

No.

Available from ISR

No.

Include

```
<unistd.h>
<fcntl.h>
```

Description

The `fcntl()` function is a limited implementation of the standard `fcntl()` system call, which can change the state of the flags associated with an open file descriptor. Normally these flags are set during the call to `open()`. The main use of this function is to change the state of a device from blocking to nonblocking (for device drivers that support this feature).

The input argument `fd` is the file descriptor to be manipulated. `cmd` is the command to execute, which can be either `F_GETFL` (return the current value of the flags) or `F_SETFL` (set the value of the flags).

Return

If `cmd` is `F_SETFL`, the argument `arg` is the new value of flags, otherwise `arg` is ignored. Only the flags `O_APPEND` and `O_NONBLOCK` can be updated by a call to `fcntl()`. All other flags remain unchanged. The return value is zero on success, or –1 otherwise.

If `cmd` is `F_GETFL`, the return value is the current value of the flags. If an error occurs, –1 is returned.

In the event of an error, `errno` is set to indicate the cause.

16.1.1.57. execve()

Prototype

```
int execve(const char *path,
char *const argv[],
char *const envp[])
```

Commonly Called By

C/C++ programs

Thread-safe

Yes.

Available from ISR

Yes.

Include

`<unistd.h>`

Description

The `execve()` function is only provided for compatibility with newlib.

Return

Calls to `execve()` always fail with the return code –1 and `errno` set to ENOSYS.

16.1.1.58. getpid()

Prototype

```
pid_t getpid (void)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

Yes.

Available from ISR

No.

Include

`<unistd.h>`

Description

The `getpid()` function is provided for newlib compatibility and obtains the current process ID.

Return

Because HAL systems cannot contain multiple processes, `getpid()` always returns the same ID number.

16.1.1.59. kill()

Prototype

```
int kill(int pid, int sig)
```

Commonly Called By

newlib C library

Thread-safe

Yes.

Available from ISR

Yes.

Include

`<signal.h>`

Description

The `kill()` function is used by newlib to send signals to processes. The input argument `pid` is the ID of the process to signal, and `sig` is the signal to send. As there is only a single process in the HAL, the only valid values for `pid` are either the current process ID, as returned by `getpid()`, or the broadcast values, that is, `pid` must be less than or equal to zero.

The following signals result in an immediate shutdown of the system, without call to `exit()`: `SIGABRT`, `SIGALRM`, `SIGFPE`, `SIGILL`, `SIGKILL`, `SIGPIPE`, `SIGQUIT`, `SIGSEGV`, `SIGTERM`, `SIGUSR1`, `SIGUSR2`, `SIGBUS`, `SIGPOLL`, `SIGPROF`, `SIGSYS`, `SIGTRAP`, `SIGVTALRM`, `SIGXCPU`, and `SIGXFSZ`.

The following signals are ignored: `SIGCHLD` and `SIGURG`.

All the remaining signals are treated as errors.

Return

The return value is zero on success, or –1 otherwise. If the call fails, `errno` is set to indicate the cause of the error.

16.1.1.60. stat()

Prototype

```
int stat(const char *file_name,

struct stat *buf);
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

See description.

Available from ISR

No.

Include

`<sys/stat.h>`

Description

The `stat()` function is similar to the `fstat()` function—It obtains status information about a file. Instead of using an open file descriptor, like `fstat()`, `stat()` takes the name of a file as an input argument.

Calls to `stat()` are thread-safe only if the implementation of `stat()` provided by the driver that is manipulated is thread-safe.

Internally, the `stat()` function is implemented as a call to `fstat()`.

Return

--

16.1.1.61. settimeofday()

Prototype

```
int settimeofday (const struct timeval *t,

const struct timezone *tz)
```

Commonly Called By

C/C++ programs

Thread-safe

No.

Available from ISR

Yes.

Include

`<sys/time.h>`

Description

Sets the current time and timezone. If the `settimeofday()` function is called concurrently with a call to `gettimeofday()`, the value returned by `gettimeofday()` is unreliable.

Return

The return value is zero on success. If no system clock is available, the return value is `-1`, and `errno` is set to `ENOSYS`.

16.1.1.62. wait()

Prototype

```
int wait(int *status)
```

Commonly Called By

newlib C library

Thread-safe

Yes.

Available from ISR

Yes.

Include

`<sys/wait.h>`

Description

`newlib` uses the `wait()` function to wait for all child processes to exit. Because the HAL does not support spawning child processes, this function returns immediately.

Return

On return, the content of `status` is set to zero, which indicates there is no child processes.

The return value is always `–1` and `errno` is set to `ECHILD`, which indicates that there are no child processes to wait for.

16.1.1.63. unlink()

Prototype

```
int unlink(char *name)
```

Commonly Called By

newlib C library

Thread-safe

Yes.

Available from ISR

Yes.

Include

`<unistd.h>`

Description

The `unlink()` function is only provided for compatibility with newlib.

Return

Calls to `unlink()` always fails with the return code `–1` and `errno` set to `ENOSYS`.

16.1.1.64. sbrk()

Prototype

```
caddr_t sbrk(int incr)
```

Commonly Called By

newlib C library

Thread-safe

No.

Available from ISR

No.

Include

`<unistd.h>`

Description

The `sbrk()` function dynamically extends the data segment for the application. The input argument `incr` is the size of the block to allocate. Do not call `sbrk()` directly. If you want to dynamically allocate memory, use the newlib `malloc()` function.

Return

--

16.1.1.65. link()

Prototype

```
int link(const char *_path1,
const char *_path2)
```

Commonly Called By

newlib C library

Thread-safe

Yes.

Available from ISR

Yes.

Include

`<unistd.h>`

Description

The `link()` function is only provided for compatibility with newlib.

Return

Calls to `link()` always fails with the return code `–1` and `errno` set to `ENOSYS`.

16.1.1.66. lseek()

Prototype

```
off_t lseek(int fd, off_t ptr, int whence)
```

Commonly Called By

C/C++ programs

newlib C library

Thread-safe

See description.

Available from ISR

No.

Include

`<unistd.h>`

Description

The `lseek()` function moves the read/write pointer associated with the file descriptor `fd`. `lseek()` is wrapper function that passes control directly to the `lseek()` function registered for the driver associated with the file descriptor. If the driver does not provide an implementation of `lseek()`, an error is reported.

`lseek()` corresponds to the standard UNIX `lseek()` function.

You can use the following values for the input parameter, `whence`:

- SEEK_SET—The offset is set to `ptr` bytes.
- SEEK_CUR—The offset is incremented by `ptr` bytes.
- SEEK_END—The offset is set to the end of the file plus `ptr` bytes.

  Calls to `lseek()` are thread-safe only if the implementation of `lseek()` provided by the driver that is manipulated is thread-safe.

  Valid values for the `fd` parameter are: `stdout`, `stdin`, and `stderr`, or any value returned from a call to `open()`.

Return

On success, the return value is a nonnegative file pointer. The return value is –1 in the event of an error. If the call fails, `errno` is set to indicate the cause of the error.

16.1.1.67. open()

Prototype

```
int open (const char* pathname, int flags, mode_t mode)
```

Commonly Called By

C/C++ programs

Thread-safe

See description.

Available from ISR

No.

Include

```
<unistd.h>
<fcntl.h>
```

Description

The `open()` function opens a file or device and returns a file descriptor (a small, nonnegative integer for use in read, write, etc.)

`flags` is one of: `O_RDONLY`, `O_WRONLY`, or `O_RDWR`, which request opening the file in read-only, write-only, or read/write mode, respectively.

You can also bitwise-`OR` `flags` with `O_NONBLOCK`, which causes the file to be opened in nonblocking mode. Neither `open()` nor any subsequent operation on the returned file descriptor causes the calling function to wait.

Not all file systems/devices recognize this option.

`mode` specifies the permissions to use, if a new file is created. It is unused by current file systems, but is maintained for compatibility.

Calls to `open()` are thread-safe only if the implementation of `open()` provided by the driver that is manipulated is thread-safe.

Return

The return value is the new file descriptor, and –1 otherwise. If an error occurs, `errno` is set to indicate the cause.

16.1.1.68. alt_sysclk_init()

Prototype

```
int alt_sysclk_init (alt_u32 nticks)
```

Commonly Called By

Device drivers

Thread-safe

No.

Available from ISR

No.

Include

```
<sys/alt_alarm.h>
```

Description

The `alt_sysclk_init()` function registers the presence of a system clock driver. The input argument is the number of ticks per second at which the system clock is run.

The expectation is that this function is only called from within `alt_sys_init()`, that is, while the system is running in single-threaded mode. Concurrent calls to this function might lead to unpredictable results.

Return

This function returns zero on success; otherwise it returns a negative value. The call can fail if a system clock driver is already registered, or if no system clock device is available.

16.1.1.69. times()

Prototype

```
clock_t times (struct tms *buf)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

Yes.

Available from ISR

Yes.

Include

```
<sys/times.h>
```

Description

This `times()` function is provided for compatibility with newlib. It returns the number of clock ticks since reset. It also fills in the structure pointed to by the input parameter `buf` with time accounting information. The definition of the `tms` structure is:

```
typedef struct
{
clock_t tms_utime;
clock_t tms_stime;
clock_t tms_cutime;
clock_t tms_cstime;
};
```

The structure has the following elements:

- `tms_utime`: the processor time charged for the execution of user instructions
- `tms_stime`: the processor time charged for execution by the system on behalf of the process
- `tms_cutime`: the sum of the values of `tms_utime` and `tms_cutime` for all child processes
- `tms_cstime`: the sum of the values of `tms_stime` and `tms_cstime` for all child processes

  In practice, all elapsed time is accounted as system time. No time is ever attributed as user time. In addition, no time is allocated to child processes, as child processes cannot be spawned by the HAL.

Return

If there is no system clock available, the return value is zero, and `errno` is set to ENOSYS.

16.1.1.70. read()

Prototype

```
int read(int fd, void *ptr, size_t len)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

See description.

Available from ISR

No.

Include

`<unistd.h>`

Description

The `read()` function reads a block of data from a file or device. `read()` is wrapper function that passes control directly to the `read()` function registered for the device driver associated with the open file descriptor `fd`. The input argument, `ptr`, is the location to place the data read and `len` is the length of the data to read in bytes.

Calls to `read()` are thread-safe only if the implementation of `read()` provided by the driver that is manipulated is thread-safe.

Valid values for the `fd` parameter are: `stdout`, `stdin`, and `stderr`, or any value returned from a call to `open()`.

Return

The return argument is the number of bytes read, which might be less than the requested length

The return value is –1 upon an error. In the event of an error, `errno` is set to indicate the cause.

16.1.1.71. write()

Prototype

```
int write(int fd, const void *ptr, size_t len)
```

Commonly Called By

C/C++ programs

newlib C library

Thread-safe

See description.

Available from ISR

No.

Include

`<unistd.h>`

Description

The `write()` function writes a block of data to a file or device. `write()` is wrapper function that passes control directly to the `write()` function registered for the device driver associated with the file descriptor `fd`. The input argument `ptr` is the data to write and `len` is the length of the data in bytes.

Calls to `write()` are thread-safe only if the implementation of `write()` provided by the driver that is manipulated is thread-safe.

Valid values for the `fd` parameter are: `stdout`, `stdin`, and `stderr`, or any value returned from a call to `open()`.

Return

The return argument is the number of bytes written, which might be less than the requested length.

The return value is –1 upon an error. In the event of an error, `errno` is set to indicate the cause.

16.1.1.72. usleep()

Prototype

```
int usleep (unsigned int us)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

Yes.

Available from ISR

No.

Include

`<unistd.h>`

Description

The `usleep()` function blocks until at least `us` microseconds have elapsed.

Return

The `usleep()` function returns zero on success, or –1 otherwise. If an error occurs, `errno` is set to indicate the cause. The current implementation always succeeds.

16.1.1.73. alt_lock_flash()

Prototype

int alt_lock_flash(alt_flash_dev * flash_info,

alt_u32 sectors_to_lock)

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

No.

Available from ISR

No.

Include

<sys/alt_flash.h>

Description

Locking to range of the flash memory sectors, which protected from writing and erasing by passing the uninteger 32 bits value to the sectors_to_lock argument, where this argument depends on the specific flash device being used,and this argument value can be found in the flash device datasheet. The flash devices can be supported are shown as below:

EPCQ16, EPCQ32, EPCQ64, EPCQ128, EPCQ256, N25Q512, EPCQ512, EPCQL512, EPCQL1024

More Micron flash devices are supported in future, and being updated into this document.

Arguments

- *flash_info: Pointer to general flash device structure.
- sectors_to_lock: Block protection bits, including the top/bottom (TB) bit in the EPCQ or QSPI, according to the device. For example, in the EPCQ128 device, the bits are Bit4=TB Bit3=BP3 Bit2=BP2 Bit1=BP1 Bit0=BP0.

Return

- ***0 ➤ Success**
- **-EINVL ➤ Invalid arguments**
- **-ETIME ➤ Time out and skipping the looping after 0.7 sec**
- **-ENOLCK ➤ Sectors lock failed**

16.1.1.74. gettimeofday()

Prototype

```
int gettimeofday(struct timeval *ptimeval,

struct timezone *ptimezone)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

See description.

Available from ISR

Yes.

Include

<sys/time.h>

Description

The `gettimeofday()` function obtains a time structure that indicates the current time. This time is calculated using the elapsed number of system clock ticks, and the current time value set by the most recent call to `settimeofday()`.

If this function is called concurrently with a call to `settimeofday()`, the value returned by `gettimeofday()` is unreliable; however, concurrent calls to `gettimeofday()` are legal.

Return

The return value is zero on success. If no system clock is available, the return value is `-ENOTSUP`.

16.1.1.75. ioctl()

Prototype

```
int ioctl (int fd, int req, void* arg)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

See description.

Available from ISR

No.

Include

`<sys/ioctl.h>`

Description

The `ioctl()` function allows application code to manipulate the I/O capabilities of a device driver in driver-specific ways. This function is equivalent to the standard UNIX `ioctl()` function. The input argument `fd` is an open file descriptor for the device to manipulate, `req` is an enumeration defining the operation request, and the interpretation of `arg` is request specific.

For file subsystems, `ioctl()` is wrapper function that passes control directly to the appropriate device driver's `ioctl()` function (as registered in the driver's `alt_dev` structure).

For devices, `ioctl()` handles `TIOCEXCL` and `TIOCNXCL` requests internally, without calling the device driver. These requests lock and release a device for exclusive access. For requests other than `TIOCEXCL` and `TIOCNXCL`, `ioctl()` passes control to the device driver's `ioctl()` function.

Calls to `ioctl()` are thread-safe only if the implementation of `ioctl()` provided by the driver that is manipulated is thread-safe.

Valid values for the `fd` parameter are: `stdout`, `stdin`, and `stderr`, or any value returned from a call to `open()`.

Return

The interpretation of the return value is request specific. If the call fails, `errno` is set to indicate the cause of the error.

16.1.1.76. isatty()

Prototype

```
int isatty(int fd)
```

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

See description.

Available from ISR

No.

Include

`<unistd.h>`

Description

The `isatty()` function determines whether the device associated with the open file descriptor `fd` is a terminal device. This implementation uses the driver's `fstat()` function to determine its reply.

Calls to `isatty()` are thread-safe only if the implementation of `fstat()` provided by the driver that is manipulated is thread-safe.

Return

The return value is 1 if the device is a character device, and zero otherwise. If an error occurs, `errno` is set to indicate the cause.

16.1.1.77. alt_niosv_enable_msw_interrupt()

Prototype

```
void alt_niosv_enable_msw_interrupt()
```

Commonly Called By

C/C++ programs

Thread-safe

Yes.

Available from ISR

No.

Include

`<sys/msw_interrupt.h>`

Description

The `alt_niosv_enable_msw_interrupt()` function enables the machine software interrupt.

Return

--

16.1.1.78. alt_niosv_disable_msw_interrupt()

Prototype

```
void alt_niosv_disable_msw_interrupt()
```

Commonly Called By

C/C++ programs

Thread-safe

Yes.

Available from ISR

No.

Include

`<sys/msw_interrupt.h>`

Description

The `alt_niosv_disable_msw_interrupt()` function disables the machine software interrupt.

Return

--

16.1.1.79. alt_niosv_is_msw_interrupt_enabled()

Prototype

```
int alt_niosv_is_msw_interrupt_enabled()
```

Commonly Called By

C/C++ programs

Thread-safe

Yes.

Available from ISR

Yes.

Include

`<sys/msw_interrupt.h>`

Description

The `int alt_niosv_is_msw_interrupt_enabled()` function determines whether the machine software interrupt is enabled.

Return

Returns zero if the machine software interrupt is disabled, and nonzero otherwise.

16.1.1.80. alt_niosv_trigger_msw_interrupt()

Prototype

```
void alt_niosv_trigger_msw_interrupt()
```

Commonly Called By

C/C++ programs

Thread-safe

Yes.

Available from ISR

No.

Include

```
<sys/msw_interrupt.h>
```

Description

The `void alt_niosv_trigger_msw_interrupt()` function generates a software IRQ after the software interrupt is enabled using `alt_niosv_enable_msw_interrupt()`.

Return

--

16.1.1.81. alt_niosv_clear_msw_interrupt()

Prototype

```
alt_niosv_clear_msw_interrupt()
```

Commonly Called By

C/C++ programs

Thread-safe

Yes.

Available from ISR

Yes.

Include

```
<sys/msw_interrupt.h>
```

Description

The `alt_niosv_trigger_msw_interrupt()` function clears the current software IRQ after the software interrupt is handled successfully.

Return

--

16.1.1.82. alt_niosv_register_msw_interrupt_handler()

Prototype

```
void alt_niosv_register_msw_interrupt_handler(alt_niosv_sw_isr_t handle)
```

Commonly Called By

C/C++ programs

Thread-safe

Yes.

Available from ISR

No.

Include

```
<sys/msw_interrupt.h>
```

Description

The `alt_niosv_register_msw_interrupt_handler()` function registers a software ISR.

The function argument `handle` is a pointer to the software ISR function.

The ISR function prototype is `void (*sw_isr_func) (alt_u32 cause, alt_u32 epc, alt_u32 tval)`.

Calls to `alt_niosv_register_msw_interrupt_handler()` replace previously registered handlers for machine software interrupt.

Return

--.

16.1.2. HAL Standard Types

In the interest of portability, the HAL uses a set of standard type definitions in place of the ANSI C built-in types. The Table below describes these types, which are defined in the header file `alt_types.h`.

Table 58. HAL Standard Types

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

16.1.2.1. alt_getchar()

Prototype

`alt_getchar()`

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

See description.

Available from ISR

No.

Include

```
<sys/alt_driver.h>
<sys/alt_stdio.h>
<priv/alt_file.h>
<unistd.h>
```

Description

The `alt_getchar()` function uses the `ALT_DRIVER_READ()` macro to call directly to the driver, if available; otherwise, it uses the `newlib` provided `getchar()` routine.

Return

--

16.1.2.2. alt_putstr()

Prototype

`alt_putstr(const char* str)`

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

See description.

Available from ISR

No.

Include

```
<sys/alt_driver.h>
<sys/alt_stdio.h>
```

Description

The `alt_putstr()` function ses the `ALT_DRIVER_WRITE()` macro to call directly to the driver, if available; otherwise, it uses the `newlib` provided `fputs()` routine.

Return

The return value is zero on success and nonzero otherwise.

16.1.2.3. alt_putchar()

Prototype

`alt_putchar(int c)`

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

See description.

Available from ISR

No.

Include

```
<sys/alt_driver.h>
<sys/alt_stdio.h>
```

Description

The `alt_putchar()` function uses the ALT_DRIVER_WRITE() macro to call directly to the driver, if available; otherwise, it uses the `newlib` provided `putchar()` routine.

Return

The return value is zero on success and nonzero otherwise.

16.1.2.4. alt_printf()

Prototype

`alt_printf(const char* fmt, ... )`

Commonly Called By

C/C++ programs

Device drivers

Thread-safe

No.

Available from ISR

No.

Include

`<sys/alt_stdio.h>`

Description

The `alt_printf()` function provides a very minimal `printf` implementation for use with very small applications. Only the following format strings are supported: %x, %s, %c, and %%.

Return

--

16.2. Nios V Processor Tools Reference

This section provides a complete reference of all available commands, options, and settings for the Nios V processor tools which are useful for developing your own embedded software project.

The Nios V processor tools are command line utilities that are available when you use the Nios V Command Shell. It enables you to create applications, board support package (BSP), and library software for a particular Nios V processor hardware system. Nios V processor tools create a portable, cross-platform CMake-based project that can be easily modified later to suit your build flow. Proficient use of these tools for customization requires some expertise with CMake. The following sections summarize the details of the build tools utilities command and settings and TCL scripting available for generating a BSP and application software for your hardware design. There are many additional options that you can customize using Nios V processor tools.

16.2.1. Nios V Processor Command Line Utilities

The Nios V processor command line utilities tools are the entry point to configure Nios V processor settings. Everything you can do with the tools, such as specifying the settings, creating `CMakeLists.txt`, and building projects, is made available by the utilities.

All Nios V processor command line utilities tools share the following behavior:

- Send error messages and warning messages to stderr.
- Send messages (other than errors and warnings) to stdout.
- Display one error message for each error.
- Return an exit value of 1 if the tool detects any errors.

- Return an exit value of 0 if the tool does not detect any errors. (Warnings are not errors).
- If the help or version command line option is specified,
  - Return an exit value of 0, and take no other action.
  - Send the output (help or version number) to stdout.
- When an error is detected, suppresses all subsequent operations (such as writing files).

The following section provides a complete reference of all available commands, options, and settings for Nios V processor command line utilities tools. The reference is useful for developing your own embedded software projects, packages, or device drivers.

The Nios V processor command line tools described in this section are in the `<Intel Quartus Prime software installation directory>/niosv/bin` directory.

Related Information

- niosv-bsp on page 14
- niosv-app on page 15
- niosv-download on page 16

16.2.1.1. niosv-shell

Description

Launch the Nios V Command Shell to run Nios V processor command line utilities tools.

Usage

```
# Launch Nios V Command Shell
niosv-shell

# Run CMake inside Nios V Command Shell
niosv-shell -r “<Absolute path>/cmake -S software/app -G "Unix Makefiles"
-DCMAKE_BUILD_TYPE="Debug" -B software/app/build”

# Run Make inside Nios V Command Shell
niosv-shell -r “<Absolute path>/make -C software/app/build”

# Run a TCL script inside Nios V Command Shell
niosv-shell < example.tcl
```

Related Information

Nios V Command Shell on page 20

16.2.1.2. niosv-bsp

Synopsis

```
niosv-bsp <FUNCTIONS> [OPTIONS] <BSP FILE>
```

Description

Create, update, generate, or query a BSP. You must specify one of the following commands.

- create (-c)
- update (-u)
- generate (-g)
- query (-q)

A `.bsp` settings file is always required. When creating a BSP, specify a `.qsys` file (-s) and type (-t) . Altera recommends specifying a `.qpf` file (-p) so that settings used are from the Quartus Prime project.

Outputs the files `CMakeLists.txt` and `toolchain.cmake`.

A BSP `CMakeLists.txt` is created to compile the BSP library file. A `toolchain.cmake` file is also created which contained toolchain options like C compiler, C++ compiler, and link flags required for Nios V processor. The app `CMakeLists.txt` uses both files. The default BSP TCL scripts run unless specified for create.

By default, the settings file used is `<bsp directory>/settings.bsp`. You can change the setting by using `-s` flag.

Usage

```
# create settings
niosv-bsp -c -p=<.qpf> -s=<.qsys> -t=<bsp-type> [OPTIONS] settings_file
# update settings
niosv-bsp -u [OPTIONS] <BSP FILE>
# generate settings
niosv-bsp -g [OPTIONS] <BSP FILE>
# query settings
niosv-bsp -q [OPTIONS] <BSP FILE>

Example: niosv-bsp -c --quartus-project=top.qpf --qsys=sys.qsys --type=<hal or
ucosii> software/bsp/settings.bsp
```

Functions

Table 59. Functions and Descriptions

Functions	Description
`-c, --create`	• Optional. • Creates the BSP settings. • Generates BSP files.
`-u, --update`	• Optional. • Updates the BSP settings. • Regenerates BSP files.
`-g, --generate`	• Optional. • Generates the BSP files.
`-q, --query`	• Optional. • Queries the BSP files. • Allows exporting of BSP as TCL script by specifying option `-E=bsp.tcl`.

Options

Options	Description
`<1st arg value>`	• Required. • The `.bsp` settings file.
`-s, --qsys=<value>`	• Optional. • The `.qsys` file. • Supported in Quartus Prime Pro Edition only.
`-s, --sopcinfo=<value>`	• Optional. • The `.sopcinfo` file. • Supported in Quartus Prime Standard Edition only.
`-p, --quartus-project=<value>`	• Optional. • The Quartus Prime project file.
`-r, --rev=<value>`	• Optional. • The Quartus Prime project revision.
`-t, --type=<value>`	• Optional. • The BSP type. • For use with `-c/--create`. • Use ? as the value to list available types.
`-T, --type-version=<value>`	• Optional. • The BSP software element version. • For use with `-c/--create`. By default the latest version is used. • Use ? as the value to list available types and versions.
`-i, --cpu-instance=<value>`	• Optional. • The CPU IP instance name to use. • Required if there are multiple CPUs in the system.
`-n, --no-default`	• Optional. • Disable the BSP set defaults TCL script which automatically chooses sane defaults for stdout, code section mappings, and etc.
`-E, --export-as-tcl=<value>`	• Optional. • The TCL script file name that imports the BSP. • For use with `q/--query`.
`-x, --script=<value>`	• Optional. • Repeatable. • A BSP TCL script to evaluate the software packages. • May be specified multiple times.
`-e, --cmd=<value>`	• Optional. • Repeatable. • The BSP TCL command to evaluate the software packages. • May be specified multiple times. • Each `--cmd` holds a single Board Support Package Tcl Commands only. — Valid example: `--cmd="add_section_mapping .text ram" --cmd="add_section_mapping .exceptions ram"` — Invalid example: `--cmd="add_section_mapping .text ram add_section_mapping .exceptions ram"`
`-b, --bsp-dir=<value>`	• Optional. • The BSP directory. If omitted, the parent directory of the settings file is used by default.

continued...

Options	Description
`-l, --search-path=<value>`	• Optional. • If omitted, a standard default path is used. • If provided, a comma-separated list of paths is searched. • To include the standard path in your replacement, use "$", like `/extra/dir/**/*,$`.
`--jvm-max-heap-size=<value>`	• Optional. • The maximum memory size to be used for allocations when running this tool. • This value is specified as <size><unit> where unit can be m (or M) for multiples of megabytes or g (or G) for multiples of gigabytes. • The default value is 512m.
`--help`	• Optional. • Display help for this tool.

16.2.1.3. niosv-app

Synopsis

```
niosv-app [OPTIONS] --srcs=<value>
```

Description

Generate an application or library CMakeLists.txt. Either an application (-a) or library (-l) must be created. The BSP directory (-b) is required when creating an application. Libraries are independent of BSP; they do not link directly with a BSP.

The CMakeLists.txt provides details to cmake and make executables on building executables (`.elf` and `.hex`) for your application. The generated HEX file(s) is located in the same directory as the ELF file.

Usage

```
#Create an app project which links with a specified BSP
niosv-app -a=app_dir -b=bsp_dir -s=sources[OPTIONS]
#Create a library
niosv-app -l=lib_dir -s=sources -p=public_includes [OPTIONS]

Example: niosv-app --bsp-dir=software/bsp --app-dir=software/app --srcs=software/
app --elf-name=hello.elf
```

Options

Options	Description
`-a, --app-dir=<value>`	• Optional. • The application directory
`-l, --lib-dir=<value>`	• Optional. • The library directory.
`-b, --bsp-dir=<value>`	• Optional. • The Board Support Package (BSP) directory
`-s, --srcs=<value>`	• Required. • Comma-separated list of sources files (.c, .h, etc.) or source directories. • Source directories includes child source files(not recursive).

continued...

Options	Description
`-S, --srcs-recursive=<directory>`	• Same as `--srcs` option but recursively search for source files in or under *<directory>*. • Allows multiple `--srcs-recursive` options and you can freely mix the option with `--srcs` options.
`-i, --incs=<value>`	• Optional. • Comma-separated list of include directories.
`-I, --incs-recursive=<directory>`	• Same as `--incs` option but recursively search for source include files in or under `<directory>`. • Allows multiple `--incs-recursive`.
`-p, --public-incs=<value>`	• Optional. • Comma-separated list of public include directories. • These public include directories are made available to dependent applications or libraries.
`-e, --elf-name=<value>`	• Optional. • Choose a name for the .elf file that results from compiling the application.
`-L, --link-lib=<value>`	• Optional. • Repeatable. • The specified library directory links with this application or library. • May be specified multiple times.
`--jvm-max-heap-size=<value>`	• Optional. • The maximum memory size to be used for allocations when running this tool. • This value is specified as <size><unit> where unit can be m (or M) for multiples of megabytes or g (or G) for multiples of gigabytes. • The default value is 512m.
`-h, --help`	• Optional. • Display help for this tool

16.2.1.4. niosv-download

Synopsis

```
niosv-download [OPTION] <ELF FILE>
```

Description

Download an ELF file to a Nios V processor CPU. Optionally reset or resume a Nios V processor CPU.

Usage

```
#Download app.elf to the only Nios V instance connected.
niosv-download app.elf

#Download app.elf to the only Nios V instance and run from the entry point.
niosv-download -g app.elf

#Download app.elf to the only Nios V instance, issue a debug reset, then run
from the entry point.
niosv-download -g -r app.elf

#Download app.elf to the Nios V at cable 1, device 2, instance 2.
niosv-download app.elf -c 1 -d 2 -i 2
```

Options

Options	Description
`--help, -h`	Display the help message.
`--reset, -r`	Reset the targeted Nios V processor.
`--go, -g`	Resume execution of the targeted Nios V processor.
`--cable CABLE,-c CABLE`	Specify the cable of the targeted Nios V processor.
`--device DEVICE, -d DEVICE`	Specify the device of the targeted Nios V processor.
`--device DEVICE, -d DEVICE`	Specify the instance of the targeted Nios V processor.

16.2.1.5. niosv-stack-report

Synopsis

```
niosv-stack-report [OPTION] <ELF FILE>
```

Description

Reports memory available for stack and heap from an `.elf` file.

Usage

```
#Report stack and heap memory from generated ELF file
niosv-stack-report [OPTIONS] app.elf

Example: niosv-stack-report app.elf
Result:
app.elf
* 438.85 KB - Program size (code + initialized data).
* 1510.36 KB - Free for stack + heap
```

Options

Options	Description
`-p PREFIX, --prefix PREFIX`	• Optional. • Uses a different compiler prefix for running `<prefix>-nm` and `prefix>-size`. • `<prefix>` defaults to riscv-none-embed if not specified.
`-d, --debug`	• Optional. • Increases output verbosity.
`-h, --help`	• Optional. • Displays help for this tool

16.2.2. File Format Conversion Tools Reference

The file format conversion tools are in the `<Intel Quartus Prime software installation directory>/niosv/bin` directory.

Related Information

File Format Conversion Tools on page 22

16.2.2.1. elf2hex

Synopsis

```
elf2hex [OPTIONS] <ELF FILE> <HEX FILE>
```

Description

The elf2hex utility converts the software and data within an `.elf` file in the address range [base, end] into a `.hex` file that can be used as the initialization data for a memory component. The width option is to ensure that the resulting `.hex` file is formatted properly for its corresponding memory component.

This utility can also create lane files which are used to initialize multi-laned memory components. The maximum address span between base and end is 1 GByte (0x4000_0000).

Usage

```
#Generate hex file for OCM Memory initialization
elf2hex <elf input file> -b <On-chip memory start address> -w <On-chip memory
data width in bits> -e <On-chip memory end address> -r 4 -o <hex output file>

Example: elf2hex software/app/build/hello.elf  -b 0x0 -w 32 -e 0x4FFFF -r 4 -o
ram.hex
```

Table 60. Options

Options	Description
`-b,--base <addr>`	Hex data base address
`--big-endian-mem`	Force big-endian memory layout
`--debug`	Debug mode
`-e,--end <addr>`	Hex data end address
`-h,--help`	Print this message
`-i,--input <file>`	Input ELF file to process
`-l,--create-lanes <value>`	1 if lane files should be created
`--little-endian-mem`	Force little-endian memory layout
`--log <file>`	File for logging progress
`--lower`	If you prefer your hex in lower case
`--no-zero-fill`	No zero fill of empty sections
`-o,--output <file>`	Output Hex file
`-q,--quiet`	Only print errors
`-r,--record <length>`	Output record length (--width in bytes)
`-s,--silent`	Silent mode - same as quiet
`-v,--verbose`	Lots of interesting information
`-w,--width <value>`	Data width of the memory [8 | 16 | 32 | 39 | 64 | 128]

16.2.2.2. elf2flash

Synopsis

```
elf2flash [OPTIONS] <ELF FILE> <SREC FILE>
```

Description

The elf2flash utility converts the software and data within an ELF file in the address range [base, end] into a SREC file that can be programmed into the flash memory connected to an Intel FPGA.

This utility can also optionally insert a boot copier into the SREC file to copy the software from flash memory to RAM before running it.

Usage

```
#Generate SREC file
elf2flash --input <user application>.elf --output flash.srec --
program_record_format –-offset 0x0

#Insert boot copier into the SREC file
elf2flash --boot <boot_copier> --input <user application>.elf
--output flash.srec --reset <reset offset> --base <base address of flash memory>
--end <end address of flash memory
```

Options

Options	Description
`--after <file>`	Start output at address immediately following data in specified SREC file
`-b,--base <addr>`	ELF base address
`--boot <file>`	Boot copier SREC file
`--debug`	Debug mode
`-e,--end <addr>`	ELF end address
`--epcs`	Quad SPI flash mode
`-h,--help`	Print this message
`-i,--input <file>`	Input ELF file to process
`--log <file>`	File for logging progress
`-o,--output <file>`	Output SREC file
`--offset <offset>`	SREC offset
`--program_record_format`	Use program record format, synonymous to --epcs
`-q,--quiet`	Only print errors
`-r,--reset <addr>`	CPU reset address
`-s,--silent`	Silent mode - same as quiet
`--save`	Save intermediate files
`--sim_optimize <value>`	Optimize for simulation
`-v,--verbose`	Lots of interesting information

16.2.3. Other Command Line Utilities Tools Reference

Intel provides the command line tools in <Intel Quartus Prime installation directory>/quartus/bin or you can acquire the tools from open-source.

Related Information

Other Utilities Tools on page 22

16.2.3.1. juart-terminal

Synopsis

```
juart-terminal [TERMINAL OPTIONS] [CONNECTION OPTIONS] [TARGET OPTIONS]
```

Description

Performs terminal I/O with a JTAG UART in a Nios V processor-based system.

Usage

```
#connect to JTAG UART at cable 1
juart-terminal -c 1
```

Terminal Options

Terminal Options	Description
V, --version	Display version number
-h, --help	Show this message and exit
-v, --verbose	Show extra information during run (default)
-q, --quiet	Show minimal information
--write-pid	Write process ID to filename specified
-w, --wait	Wait for a signal before starting
--signal-pid	Send signal to process ID when ready
--flush	Empty buffers before displaying output
-o, --quit-after=SECS	Quit after SECS seconds. Default is never.
--no-quit-on-ctrl-d	Do not quit if Ctrl-D received from target
-u, --dump=OPTS	Dump extra debug information for OPTS

Connection Options

Connection Option	Description
-H, --hardware	Connect to a hardware target (default)
--persistent	Try to reconnect after an I/O error
--no-persistent	Opposite of persistent

Target Option

Target Option	Description
-c, --cable=CABLE	Use <CABLE JTAG> cable (default auto-detect)
-d, --device=DEVICE	Connect to <DEVICE> device (default auto-detect)
-i, --instance=INSTANCE	Connect to <INSTANCE> instance (default auto-detect)

Example 32. Specifying Cable, Device, and Instance

```
1) AGF FPGA Development Kit on Intel® FPGA Download Cable [USB-0]
   (JTAG Server Version 22.1.0 Build 174 03/30/2022 SC Pro Edition)
  C341A0DD   AGFB014R24A(.|R1|R2)/.. (IR=10)
    Design hash
    + Node 00486E00  Source/Probe #0
    + Node 08986E00  Nios V #0
    + Node 08986E01  Nios V #1
    + Node 0C006E00  JTAG UART #0
    + Node 0C006E01  JTAG UART #1
  031830DD   10M16S(A|C|L) (IR=10)
    Design hash
    + Node 08986E00  Nios V #0
    + Node 0C006E00  JTAG UART #0

2) Stratix 10H SoC Dev Kit on Intel® FPGA Download Cable [USB-1]
   (JTAG Server Version 22.1.0 Build 174 03/30/2022 SC Pro Edition)
  C322D0DD   1SX280HH1(.|S3)/1SX280HH2/.. (IR=10)
    Design hash
    + Node 00486E00  Source/Probe #0
    + Node 08986E00  Nios V #0
    + Node 08986E01  Nios V #1
    + Node 0C006E00  JTAG UART #0
    + Node 0C006E01  JTAG UART #1
```

Table 61. Selecting Target JTAG UART

Arguments	Target JTAG UART
--cable=1 --device=1 --instance=0	**Cable 1** 1) AGF FPGA Development Kit on Intel® FPGA Download Cable [USB-0] **Device 1** C341A0DD AGFB014R24A **Instance 0** Node 0C006E00 JTAG UART #0
--cable=1 --device=2 --instance=0	**Cable 1** 1) AGF FPGA Development Kit on Intel® FPGA Download Cable [USB-0] **Device 2** 031830DD 10M16S **Instance 0** Node 0C006E00 JTAG UART #0
--cable=2 --device=1 --instance=1	**Cable 2** 2) Stratix 10H SoC Dev Kit on Intel® FPGA Download Cable [USB-1] **Device 1** C322D0DD 1SX280HH1 **Instance 1** Node 0C006E01 JTAG UART #1

16.2.3.2. cmake

CMake is the open-source utility tool that manages the build process using `CMakeLists.txt`, generated from from the niosv-app and niosv-bsp utilities.

CMake uses the `CMakeLists.txt` file to generate the build file, used to build `main.elf` and `libbsp.a`. The setting can be changed via setting the CMake argument -G, --generator. Nios V processor software build tools are based on Unix Makefile generator. The default build type on single-configuration generator (CMAKE_BUILD_TYPE) is `Debug`. You can change the build type to `Release` for production.

```
Example: cmake -S software/app -G "Unix Makefiles" -DCMAKE_BUILD_TYPE="Debug" -B
software/app/build
```

CMake supports out-of-place/out-of-source builds in which build files need not be in the same place as source files. Build directory can be specified to be any arbitrary directory and it does not require to be in the same folder as `CMakeLists.txt`.

Related Information

- CMake
- CMake commands
  For more various actions the tool can perform.

16.2.3.3. make

The `make` utility automatically determines which pieces of a large program need to be recompiled, and issues commands to recompile them. Make gets its knowledge of how to build your program from a file called the Makefile, which lists each of the non-source files and how to compute it from other files.

```
Example: make -C software/app/build
```

16.2.3.4. OpenOCD

Nios V processor supports the Open On-Chip Debugger for debugging. Open On-Chip Debugger(OpenOCD), is an open source GNU Project Debugger(GDB) server implementation that allows a GDB Client to debug embedded target device. In a standard setup, OpenOCD translates debugging Remote Serial Protocol instructions from the GDB client into signals that is understood by the target device, then requests the device to perform the required action and communicate the result back to the GDB client.

Utility path:`../<QPDS version>/quartus/bin64/openocd.exe`

Synopsis

```
openocd [OPTIONS] <CONFIGURATION FILE>
```

Description

To start a OpenOCD client session using desired device configuration file .

Once you have connected to the OpenOCD client (thru gdb, tcl or telnet), run the OpenOCD commands to start debugging. For more information about the list of OpenOCD commands, refer to *OpenOCD: General Commands or RISC-V Architecture and Core Commands*.

Usage

```
#start an openocd client session using configuration nfile niosv-m.cfg
openocd -f niosv-m.cfg
```

Options

Options	Description
`-h, --help`	Display help menu
`-f, --file`	Use configuration file <name>
`-c, --command`	Run <command>
`-d or -d<n>, --debug or --debug<n>`	Set debug level to <level>, default value = 3 without <n> argument
`-v, --version`	Display OpenOCD version
`-s, --search`	Search for config files and scripts
`l, --log_output-`	Redirect log output to file <name>

Related Information

- OpenOCD Developer's Guide
- OpenOCD: General Commands
- OpenOCD: RISC-V Architecture and Core Commands

16.2.3.5. openocd-cfg-gen

Synopsis

```
openocd-cfg-gen [OPTIONS] <CONFIGURATION FILE>
```

Description

Generate the OpenOCD configuration file, that contain information of the active Test Access Ports (TAPs) on the device for Nios V processor debugging. You can refer to the JTAG chain device index when selecting the cable, device and instance.

Usage

```
#Display the JTAG chain device index
openocd-cfg-gen --print

#Generate OpenOCD configuration file (niosv-m.cfg)
openocd-cfg-gen niosv-m.cfg
```

Options

Option	Description
`--cable=CABLE, -c CABLE`	Specify the cable of the targeted Nios V processor.
`--device=DEVICE, -d DEVICE`	Specify the device of the targeted Nios V processor.
`--instance=INSTANCE,--i INSTANCE`	Specify the instance of the targeted Nios V processor.
`--bridge=BRIDGE,-b BRIDGE`	Specify the bridge of the targeted Nios V processor.
`--help, -h`	Display the help message.
`--print, -p`	Display the JTAG chain device index.

16.2.3.6. elf-insert

Description

A simple ELF 'fattening' tool used to inject additional sections into an ELF intended for the Nios V processor. The tool's primary purpose is to insert extra information into ELF files so that other downstream tools can extract it. The main goal is to make the ELF file the single handoff file that all downstream embedded tools can use.

For example, you can use `elf-insert` to insert CPU name, System ID and other information into the ELF file during development. Later, implement `elf-query` to extract the information for design identification or process connectivity.

The first argument is the ELF filename: (`elf-insert <filename>`).

Valid Additional Arguments

```
--qsys <qsysFabricModeEnabled>
--cpu_name <cpuName>
--stderr_dev <stderrDev>
--stdin_dev <stdinDev>
--stdout_dev <stdoutDev>
--sidp <sysidBase>
--id <sysidHash>
--timestamp <sysidTime>
--sof <sofFile>
--sopcinfo <sopcinfoFile>
--jar <jarFile>
--jdi <jdiFile>
--quartus_project_dir <quartusProjectDir>
--sopc_system_name <sopcSystemName>
--profiling_enabled <profilingEnabled>
--simulation_enabled <simulationEnabled>
--thread_model <threadModel>
```

Advanced Options

```
--jvm-max-heap-size=<value>
```

Optional. You can use the maximum memory size for allocations when running this tool. Specify the value as `<size><unit>` where the unit can be m (or M) for multiples of megabytes or g (or G) for multiples of gigabytes. The default value is 512 m.

16.2.3.7. elf-query

Description

A simple ELF 'querying' tool used to extract metadata inserted into an ELF intended for the Nios V processor. The tool's main purpose is to allow downstream tools to extract information from an ELF that was inserted using `elf-insert`. The main goal is to make the ELF file the single handoff file that all downstream embedded tools can use.

For example, you can use `elf-insert` to insert CPU name, System ID and other information into the ELF file during development. Later, implement `elf-query` to extract the information for design identification or process connectivity.

The first argument is the ELF filename: `(elf-query <filename>)`.

Valid Additional Arguments

```
--qsys
--cpu_name
--stderr_dev
--stdin_dev
--stdout_dev
--sidp
--id
--timestamp
--sof
--sopcinfo
--jar
--jdi
--quartus_project_dir
--sopc_system_name
--profiling_enabled
--simulation_enabled
--thread_model
```

Advanced Options

```
--jvm-max-heap-size=<value>
```

Optional. You can use the maximum memory size for allocations when running this tool. Specify the value as `<size><unit>` where the unit can be m (or M) for multiples of megabytes or g (or G) for multiples of gigabytes. The default value is 512 m.

16.3. GNU RISC-V Embedded GCC Toolchain Reference

Nios V processor uses the GNU RISC-V GCC toolchain to compile program. Install the Ashling RiscFree IDE for Intel FPGAs containing the following programs prefixed by riscv32-unknown-elf-*.

- riscv32-unknown-elf-addr2line
- riscv32-unknown-elf-g++
- riscv32-unknown-elf-gcov-dump
- riscv32-unknown-elf-ld
- riscv32-unknown-elf-readelf
- riscv32-unknown-elf-ar
- riscv32-unknown-elf-gcc

- riscv32-unknown-elf-gcov-tool
- riscv32-unknown-elf-ld.bfd
- riscv32-unknown-elf-size
- riscv32-unknown-elf-as
- riscv32-unknown-elf-gcc-11.2.0
- riscv32-unknown-elf-gdb
- riscv32-unknown-elf-lto-dump
- riscv32-unknown-elf-strings
- riscv32-unknown-elf-c++
- riscv32-unknown-elf-gcc-ar
- riscv32-unknown-elf-gdb-add-index
- riscv32-unknown-elf-nm
- riscv32-unknown-elf-strip
- riscv32-unknown-elf-c++filt
- riscv32-unknown-elf-gcc-nm
- riscv32-unknown-elf-gdb-add-index-py3
- riscv32-unknown-elf-objcopy
- riscv32-unknown-elf-cpp
- riscv32-unknown-elf-gcc-ranlib
- riscv32-unknown-elf-gcc-11.2.0
- riscv32-unknown-elf-objdump
- riscv32-unknown-elf-elfedit
- riscv32-unknown-elf-gcov
- riscv32-unknown-elf-gprof
- riscv32-unknown-elf-ranlib

Refer to Ashling RiscFree IDE for Intel FPGAs User Guide for the installation guide. For more information about GNU command options, please refer to GCC Command Options.

Related Information

- Other Utilities Tools on page 22
- Ashling* RiscFree* Integrated Development Environment (IDE) for Altera FPGAs User Guide

16.4. Settings Managed by Nios V Processor Board Support Package Editor

Settings are central to how you create and work with BSPs, software packages, and device drivers. You control the characteristics of your project by controlling the settings. The settings determine things like whether or not an operating system is supported, and the device drivers and other packages that are included.

Sometimes these settings are specified automatically, by scripts, and sometimes explicitly, with Tcl commands. Either way, settings are always involved.

This section contains a complete list of available settings for BSPs and for Intel FPGA-supported device drivers and software packages. It does not include settings for device drivers or software packages furnished by Intel partners or other third parties. If you are using a third-party driver or component, refer to the supplier's documentation.

Settings used in the Nios V processor Board Support Package Editor are organized hierarchically, for logical grouping and to avoid name conflicts. Each setting's position in the hierarchy is indicated by one or more prefixes. A prefix is an identifier followed by a dot (`.`). For example, `hal.enable_c_plus_plus` is a hardware abstraction layer (HAL) setting, while `ucosii.event_flag.os_flag_accept_en` is a member of the event flag subgroup of MicroC/OS-II settings.

Setting names are case-insensitive.

Related Information

HAL BSP Settings on page 84
For more information about how to control BSP settings.

16.4.1. Overview of BSP Settings

A BSP setting consists of a name/value pair.

The format in which you specify the setting value depends on the setting type. Several settings types are supported.

Table 62. Allowed Formats for each Setting Type

Setting Type	Format When Setting	Format When Getting
boolean	0/1 or false/true	0/1
number	0x prefix for hexadecimal or no prefix for a decimal number	decimal
string	Quoted string Use `"none"` to set empty string.	Quoted string

Table 63. BSP Setting Contexts

Setting Context	Description
Intel HAL	The setting is available with the Intel HAL BSP or any BSP based on it (for example, Micrium MicroC/OS-II).
Micrium MicroC/OS-II	The settings is available if using the Micrium MicroC/OS-II BSP. All Intel HAL BSP settings are also available because MicroC/OS-II is based on the Intel HAL BSP.
FreeRTOS	The settings is available if you are using the FreeRTOS BSP. All Intel HAL BSP settings are also available because FreeRTOS is based on the Intel HAL BSP.

Do not confuse BSP settings with BSP Tcl commands. This section covers BSP settings, including their types, meanings, and legal values.

16.4.2. Overview of Component and Driver Settings

The Nios V processor tools includes a number of standard software packages and device drivers. All of the software packages, and several drivers, have settings that you can manipulate when creating a BSP. This section lists the packages and drivers that have settings.

You can enable a software package or driver in the BSP Editor.

16.4.2.1. Intel FPGA Avalon-MM JTAG UART Driver Settings

The Intel FPGA Avalon Memory-Mapped® (Avalon-MM) JTAG UART driver settings are available if the `altera_avalon_jtag_uart` driver is present. By default, this driver is used if your hardware system has an `altera_avalon_jtag_uart` module connected to it.

16.4.2.2. Intel FPGA Avalon-MM UART Driver Settings

The Intel FPGA Avalon-MM UART driver settings are available if the `altera_avalon_uart` driver is present. By default, this driver is used if your hardware system has an `altera_avalon_uart` module connected to it.

16.4.3. BSP Settings Reference

This section lists all settings for BSPs, software packages, and device drivers.

16.4.3.1. Intel HAL BSP

hal.enable_instruction_related_exceptions_api

- **Identifier:**ALT_INCLUDE_INSTRUCTION_RELATED_EXCEPTION_API
- **Type:** Boolean definition
- **Default Value:** false
- **Destination File:** `system.h`
- **Description:** Enables application program interface (API) for registering handlers to service instruction-related exceptions. These exception types can be generated if various processor options are enabled, such as the memory management unit (MMU), memory protection unit (MPU), or other advanced exception types. Enabling this setting increases the size of the exception entry code.
- **Restrictions:** none

hal.max_file_descriptors

- **Identifier:** ALT_MAX_FD
- **Type:** Decimal number
- **Default Value:** 32

- **Destination File**: `system.h`
- **Description:** Determines the number of file descriptors statically allocated.
- **Restriction:** If hal.enable_lightweight_device_driver_api is true, there are no file descriptors so this setting is ignored. If hal.enable_lightweight_device_driver_api is false, this setting must be at least 4 because HAL needs a file descriptor for /dev/null, /dev/stdin, /dev/stdout, and /dev/stderr. This setting defines the value of ALT_MAX_FD in `system.h`.

hal.sys_clk_timer

- **Identifier:** ALT_SYS_CLK
- **Type:** Unquoted string
- **Default Value:** none
- **Destination File**: `system.h`
- **Description:** Slave descriptor of the system clock timer device. This device provides a periodic interrupt ("tick") and is typically required for RTOS use. This setting defines the value of ALT_SYS_CLK in `system.h`.
- **Restriction:** none

hal.timestamp_timer

- **Identifier**: ALT_TIMESTAMP_CLK
- **Type:** Unquoted string
- **Default Value:** none
- **Destination File**: `system.h`
- **Description:** Slave descriptor of timestamp timer device. This device is used by Intel HAL timestamp drivers for high-resolution time measurement. This setting defines the value of ALT_TIMESTAMP_CLK in `system.h`.
- **Restriction:** none

hal.linker.allow_code_at_reset

- **Identifier:** none
- **Type:** Boolean assignment
- **Default Value:** 0
- **Destination File:** none
- **Description:** Indicates if initialization code is allowed at the reset address. If true, defines the macro ALT_ALLOW_CODE_AT_RESET in `linker.h`.
- **Restriction:** This setting is typically false if an external bootloader (e.g. flash bootloader) is present.

hal.linker.enable_alt_load

- **Identifier:** none
- **Type:** Boolean assignment
- **Default Value:** 1

- **Destination File:** none
- **Description:** Enables the alt_load() facility. The alt_load() facility copies sections from the `.text` memory into RAM. If true, this setting sets up the VMA/LMA (virtual memory address/low memory address) of sections in linker.x to allow them to be loaded into the `.text` memory.
- **Restriction:** This setting is typically false if an external bootloader (e.g. flash bootloader) is present.

hal.linker.enable_alt_load_copy_exceptions

- **Identifier:** none
- **Type:** Boolean assignment
- **Default Value:** 0
- **Destination File:** none
- **Description:** Causes the alt_load() facility to copy the `.exceptions` section. If true, this setting defines the macro ALT_LOAD_COPY_EXCEPTIONS in `linker.h`.
- **Restriction:** none

hal.linker.enable_alt_load_copy_rodata

- **Identifier:** none
- **Type:** Boolean assignment
- **Default Value:** 0
- **Destination File:** none
- **Description:** Causes the alt_load() facility to copy the `.rodata` section. If true, this setting defines the macro ALT_LOAD_COPY_RODATA in `linker.h`.
- **Restriction:** none

hal.linker.enable_alt_load_copy_rwdata

- **Identifier:** none
- **Type:** Boolean assignment
- **Default Value:** 0
- **Destination File:** none
- **Description:** Causes the initialization code to copy the `.rwdata` section. If true, this setting defines the macro ALT_LOAD_COPY_RWDATA in `linker.h`.
- **Restriction:** none

hal.linker.enable_exception_stack

- **Identifier:** none
- **Type:** Boolean assignment
- **Default Value:** 0

- **Destination File:** none
- **Description:** Enables use of a separate exception stack. If true, defines the macro ALT_EXCEPTION_STACK in `linker.h`, adds a memory region called exception_stack to linker.x, and provides the symbols __alt_exception_stack_pointer and __alt_exception_stack_limit in linker.x.
- **Restriction:** The hal.linker.exception_stack_size and hal.linker.exception_stack_memory_region_name settings must also be valid. This setting must be false for MicroC/OS-II BSPs. The exception stack can be used to improve interrupt and other exception performance if an EIC is not implemented.

hal.linker.exception_stack_memory_region_name

- **Identifier:** none
- **Type:** Unquoted string
- **Default Value:** none
- **Destination File:** none
- **Description:** Name of the existing memory region to be divided up to create the `exception_stack` memory region. The selected region name is adjusted automatically when the BSP is generated to create the `exception_stack` memory region.
- **Restriction:** Only used if hal.linker.enable_exception_stack is true.

hal.linker.use_picolibc

- **Identifier:** USE_PICOLIBC
- **Type:** Boolean assignment
- **Default Value:** false
- **Destination File:** toolchain.cmake
- **Description:** Use Picolibc in place of Newlib to provide C library support.
- **Restriction:** none

hal.linker.exception_stack_size

- **Identifier:** none
- **Type:** Decimal number
- **Default Value:** 1024
- **Destination File:** none
- **Description:** Size of the exception stack in bytes.
- **Restriction:** Only used if hal.linker.enable_exception_stack is true. none

hal.toolchain.ar

- **Identifier:** CMAKE_AR
- **Type:** Unquoted string
- **Default Value:** riscv32-unknown-elf-ar

- **Destination File:** `toolchain.cmake`
- **Description:** Archiver command. Creates library files.
- **Restriction:** none

hal.toolchain.as

- **Identifier:** CMAKE_ASM_COMPILER
- **Type:** Unquoted string
- **Default Value:** riscv32-unknown-elf-gcc
- **Destination File:** `toolchain.cmake`
- **Description:** Assembler command. Note that CC is used for Nios V processor assembly language source files (**.S**).
- **Restriction:** none

hal.make.asflags

- **Identifier:** COMPILE_LANGUAGE:ASM
- **Type:** Unquoted string
- **Default Value:** -Wa,-gdwarf2
- **Destination File:** `toolchain.cmake`
- **Description:** Custom flags only passed to the assembler. This setting defines the value of COMPILE_LANGUAGE:ASM in `toolchain.cmake`.
- **Restriction:** none

hal.make.cflags_debug

- **Identifier:** CMAKE_C_FLAGS_DEBUG and CMAKE_CXX_FLAGS_DEBUG
- **Type:** Unquoted string
- **Default Value:** -g
- **Destination File:** `toolchain.cmake`
- **Description:** C/C++ compiler debug level when CMAKE_BUILD_TYPE=Debug. `-g` provides the default set of debug symbols required to debug a typical application. Omitting `-g` removes debug symbols from the `.elf` file. This setting defines the value of CMAKE_C_FLAGS_DEBUG and CMAKE_CXX_FLAGS_DEBUG in `toolchain.cmake`.
- **Restriction:** Run cmake command with the flag, CMAKE_BUILD_TYPE=Debug

hal.make.cflags_defined_symbols

- **Identifier:** none
- **Type:** Unquoted string
- **Default Value:** none

- **Destination File:** `toolchain.cmake`
- **Description:** Preprocessor macros to define. A macro definition in this setting has the same effect as a `#define` in source code. Adding `-DALT_DEBUG` to this setting has the same effect as `#define ALT_DEBUG` in a source file. Adding `-DFOO=1` to this setting is equivalent to the macro `#define FOO 1` in a source file. Macros defined with this setting are applied to all **.S**, C source (`.c`), and C++ files in the BSP. This setting defines the value of preprocessor macros in the `toolchain.cmake`.
- **Restriction:** none

hal.make.cflags_optimization

- **Identifier:** CMAKE_C_FLAGS_RELEASE and CMAKE_CXX_FLAGS_RELEASE
- **Type:** Unquoted string
- **Default Value:** -O2
- **Destination File:** `toolchain.cmake`
- **Description:** C/C++ compiler optimization level when CMAKE__BUILD_TYPE=`Release`. `-O0` = no optimization, `-O2` = normal optimization, etc. `-O0` is recommended for code that you want to debug since compiler optimization can remove variables and produce non-sequential execution of code while debugging. This setting defines the value of CMAKE_C_FLAGS_RELEASE and CMAKE_CXX_FLAGS_RELEASE in `toolchain.cmake`.
- **Restriction:** Run cmake command with the flag, CMAKE_BUILD_TYPE=`Release`

hal.make.cflags_undefined_symbols

- **Identifier:** none
- **Type:** Unquoted string
- **Default Value:** none
- **Destination File:** `toolchain.cmake`
- **Description:** Preprocessor macros to undefine. Undefined macros are similar to defined macros, but replicate the `#undef` directive in source code. To undefine the macro FOO use the syntax `-u FOO` in this setting. This is equivalent to `#undef FOO` in a source file. Note: the syntax differs from macro definition (there is a space, i.e. `-u FOO` versus `-DFOO`). Macros defined with this setting are applied to all **.S**, **.c**, and C++ files in the BSP. This setting undefines preprocessor macros in the `toolchain.cmake`.
- **Restriction:** none

hal.make.cflags_user_flags

- **Identifier:** none
- **Type:** Unquoted string
- **Default Value:** none

- **Destination File:** `toolchain.cmake`
- **Description:** Custom flags passed to the compiler when compiling C, C++, and .**S** files. This setting defines the value of custom flags in `toolchain.cmake`.
- **Restriction:** none

hal.make.cflags_warnings

- **Identifier:** none
- **Type:** Unquoted string
- **Default Value:** -Wall-Wformat-security
- **Destination File:** `toolchain.cmake`
- **Description:** C/C++ compiler warning level. `-Wall` is commonly used. -Wformat-security warns about calls to printf and scanf functions, where the format string is not a string and there are no format arguments. This setting defines the warning levels in `toolchain.cmake`.
- **Restriction:** none

hal.make.cxx_flags

- **Identifier:** COMPILE_LANGUAGE:CXX
- **Type:** Unquoted string
- **Default Value:** none
- **Destination File:** `toolchain.cmake`
- **Description:** Custom flags only passed to the C++ compiler. This setting defines the value of COMPILE_LANGUAGE:CXX in `toolchain.cmake`.
- **Restriction:** none

hal.make.link_flags

- **Identifier:** none
- **Type:** Unquoted string
- **Default Value:** none
- **Destination File:** toolchain.cmake
- **Description:** Linker flags. This setting defines the linker flags in `toolchain.cmake`.
- **Restriction:** none

hal.make.objdump_flags

- **Identifier:** ToolchainObjdumpFlags
- **Type:** Unquoted string
- **Default Value:** -Sdtx
- **Destination File:** toolchain.cmake
- **Description:** Objdump flags. This setting defines ToolchainObjdumpFlags in `toolchain.cmake`.
- **Restriction:** none

hal.toolchain.cc

- **Identifier:** CMAKE_C_COMPILER
- **Type:** Unquoted string
- **Default Value:** riscv32-unknown-elf-gcc
- **Destination File:** toolchain.cmake
- **Description:** C compiler command
- **Restriction:** none

hal.toolchain.cxx

- **Identifier:** CMAKE_CXX_COMPILER
- **Type:** Unquoted string
- **Default Value:** riscv32-unknown-elf-g++
- **Destination File:** `toolchain.cmake`
- **Description:** C++ compiler command
- **Restriction:** none

hal.toolchain.objdump

- **Identifier:** ToolchainObjdump
- **Type:** Unqouted string
- **Default Value:** riscv32-unknown-elf-objdump
- **Destination File:** `toolchain.cmake`
- **Description:** Objdump command
- **Restriction:** none

hal.make.enable_cflag_fstack_protector_strong

- **Identifier:** none
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `toolchain.cmake`
- **Description:** Enable the fstack-protector-strong compiler flag in `toolchain.cmake`. If true, add `-fstack-protector-strong`.
- **Restriction:** none

hal.make.enable_cflag_wformat_security

- **Identifier:** none
- **Type:** Boolean assignment
- **Default Value:** 1

- **Destination File:** `toolchain.cmake`
- **Description:** Enable the Wformat and Wformat-security compiler flags in `toolchain.cmake` to warn against security problems when using format functions. If true, add `-Wformat -Wformat-security`.
- **Restriction:** none

hal.toolchain.prefix

- **Identifier**: ToolchainPrefix
- **Type**: Unquoted string
- **Default Value**: riscv32-unknown-elf-
- **Description**: Prefix to be used for toolchain executables. The value affects the individual values of the archiver, assembler, compiler, and linker. Specifically, it affects the settings.
  - `hal.toolchain.ar`
  - `hal.toolchain.as`
  - `hal.toolchain.cc`
  - `hal.toolchain.cxx`
  - `hal.toolchain.objdump`

  Example of the value: `${hal.toolchain.prefix}gcc`, `${hal.toolchain.prefix}objdump`, and others.

  However, if the override setting **hal.toolchain.enable_executable_overrides** is enabled, the individual settings `hal.toolchain.ar`, `hal.toolchain.as`, `hal.toolchain.cc`, `hal.toolchain.cxx`, and `hal.toolchain.objdump` may be set to custom values and are not derived from this setting's value.

  The riscv32-unknown-elf toolchain is available with the Ashling RiscFree IDE for Intel FPGAs. You must install the Ashling RiscFree IDE for Intel FPGAs to use this toolchain.
- **Restriction**: none

hal.toolchain.enable_executable_overrides

- **Identifier**: none
- **Type**: Boolean assignment
- **Default value**: false
- **Description**: Enable custom values to be set for the toolchain executables. You can set the settings for `hal.toolchain.ar`, `hal.toolchain.as`, `hal.toolchain.cc`, `hal.toolchain.cxx`, and `hal.toolchain.objdump` to custom values.
- **Restriction**: none

hal.enable_c_plus_plus

- **Identifier:** ALT_NO_C_PLUS_PLUS
- **Type:** Boolean assignment
- **Default Value:** 1

- **Destination File:** `toolchain.cmake`
- **Description:** Enable support for a subset of the C++ language. This option increases code footprint by adding support for C++ constructors. Certain features, such as multiple inheritance and exceptions are not supported. If false, add `ALT_NO_C_PLUS_PLUS` in `toolchain.cmake`, and reduce code footprint.
- **Restriction:** none

hal.enable_clean_exit

- **Identifier:** ALT_NO_CLEAN_EXIT
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `toolchain.cmake`
- **Description:** When your application exits, close file descriptors, call C++ destructors, etc. Code footprint can be reduced by disabling clean exit. If disabled, adds `ALT_NO_CLEAN_EXIT` and `-Wl`, `--defsym`, `exit=_exit` in `toolchain.cmake`.
- **Restriction:** none

hal.enable_exit

- **Identifier:** ALT_NO_EXIT
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `toolchain.cmake`
- **Description:** Add exit() support. This option increases code footprint if your `main()` routine returns or calls `exit()`. If false, adds `ALT_NO_EXIT` in `toolchain.cmake`, and reduces footprint.
- **Restriction:** none

hal.enable_reduced_device_drivers

- **Identifier:** ALT_USE_SMALL_DRIVERS
- **Type:** Boolean assignment
- **Default Value:** 0
- **Destination File:** `toolchain.cmake`
- **Description:** Certain drivers are compiled with reduced functionality to reduce code footprint. Not all drivers observe this setting. If true, adds `ALT_USE_SMALL_DRIVERS` in `toolchain.cmake`. Typically, drivers support this setting with a polled mode. For example, the `altera_avalon_uart` and `altera_avalon_jtag_uart` reduced drivers operate in polled mode.
- **Restriction:** none

hal.enable_lightweight_device_driver_api

- **Identifier:** ALT_USE_DIRECT_DRIVERS
- **Type:** Boolean assignment
- **Default Value:** 0

- **Destination File:** toolchain.cmake
- **Description:** Enable lightweight device driver API to reduces code and data footprint. If true, add `ALT_USE_DIRECT_DRIVERS` in `toolchain.cmake`.
- **Restriction:** none

hal.enable_runtime_stack_checking

- **Identifier:** ALT_STACK_CHECK
- **Type:** Boolean assignment
- **Default Value:** 0
- **Destination File:** `toolchain.cmake`
- **Description:** Turns on HAL runtime stack checking feature. Enabling this setting enables the HAL to check if the heap collide with the stack when allocating heap memory, such as with `malloc()`. If true, adds `ALT_STACK_CHECK` in `toolchain.cmake`.
- **Restriction:** none

hal.enable_sim_optimize

- **Identifier:** ALT_SIM_OPTIMIZE
- **Type:** Boolean assignment
- **Default Value:** 0
- **Destination File:** `toolchain.cmake`
- **Description:** The BSP is compiled with optimizations to speedup HDL simulation such as initializing the cache, clearing the `.bss` section, and skipping long delay loops. If true, adds `ALT_SIM_OPTIMIZE` in `toolchain.cmake`.
- **Restriction:** When this setting is true, the BSP cannot run on hardware.

hal.log_port

- **Identifier:** ALT_LOG_PORT
- **Type:** Unquoted string
- **Default Value:** none
- **Destination File:** `system.h`
- **Description:** Slave descriptor of debug logging character-mode device. If defined, it enables extra debug messages in the HAL source. This setting is used by the Intel FPGA logging functions.

hal.log_flags

- **Identifier:** ALT_LOG_FLAGS
- **Type:** Decimal Number
- **Default Value:** 0
- **Destination File:** `toolchain.cmake`
- **Description:** The value is assigned to ALT_LOG_FLAGS in the generated `toolchain.cmake`. Refer to **hal.log_port** for further details. The valid range of this setting is -1 through 3.

hal.stderr

- **Identifier:** ALT_STDERR
- **Type:** Unquoted string
- **Default Value:** none
- **Destination File:** `system.h`
- **Description:** Slave descriptor of STDERR character-mode device. This setting is used by the ALT_STDERR family of defines in `system.h`.

hal.stdin

- **Identifier:**ALT_STDIN
- **Type:** Unquoted string
- **Default Value:** none
- **Destination File:** `system.h`
- **Description:** Slave descriptor of STDIN character-mode device. This setting is used by the ALT_STDIN family of defines in `system.h`.

hal.stdout

- **Identifier:** ALT_STDOUT
- **Type:** Unquoted string
- **Default Value:** none
- **Destination File:** `system.h`
- **Description:** Slave descriptor of STDOUT character-mode device. This setting is used by the ALT_STDOUT family of defines in `system.h`.

16.4.3.2. Micrium MicroC/OS-II BSP

ucosii.os_max_tasks

- **Identifier** : OS_MAX_TASKS
- **Type:** Decimal number
- **Default Value:** 10
- **Destination File:** `system.h`
- **Description:** Maximum number of tasks
- **Restriction:** none

ucosii.os_lowest_prio

- **Identifier:** OS_LOWEST_PRIO
- **Type:** Decimal number
- **Default Value:** 20
- **Destination File:** `system.h`
- **Description:** Lowest assignable priority
- **Restriction:** none

ucosii.os_thread_safe_newlib

- **Identifier:** OS_THREAD_SAFE_NEWLIB
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Thread safe C library
- **Restriction:** none

ucosii.miscellaneous.os_arg_chk_en

- **Identifier:** OS_ARG_CHK_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Enable argument checking
- **Restriction:** none

ucosii.miscellaneous.os_cpu_hooks_en

- **Identifier:** OS_CPU_HOOKS_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Enable MicroC/OS-II hooks
- **Restriction:** none

ucosii.miscellaneous.os_debug_en

- **Identifier:** OS_DEBUG_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Enable debug variables
- **Restriction:** none

ucosii.miscellaneous.os_sched_lock_en

- **Identifier:** OS_SCHED_LOCK_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Include code for OSSchedLock() and OSSchedUnlock()
- **Restriction:** none

ucosii.miscellaneous.os_task_stat_en

- **Identifier:** OS_TASK_STAT_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Enable statistics task
- **Restriction:** none

ucosii.miscellaneous.os_task_reg_tbl_size

- **Identifier:** OS_TASK_REG_TBL_SIZE
- **Type:** Decimal number
- **Default Value:** 32
- **Destination File:** `system.h`
- **Description:** Add 32-bit unsigned task specific registers.
- **Restriction:** none

ucosii.miscellaneous.os_task_stat_stk_chk_en

- **Identifier:** OS_TASK_STAT_STK_CHK_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Check task stacks from statistics task
- **Restriction:** none

ucosii.miscellaneous.os_tick_step_en

- **Identifier:** OS_TICK_STEP_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Enable tick stepping feature for uCOS-View
- **Restriction:** none

ucosii.miscellaneous.os_event_name_size

- **Identifier:** OS_EVENT_NAME_EN
- **Type:** Decimal number
- **Default Value:** 1

- **Destination File:** `system.h`
- **Description:** This constant determines whether names can be assigned to either a semaphore, a mutex, a mailbox or a message queue. If OS_EVENT_NAME_EN is set to 0, this feature is disabled. You should note that need to use `OSEventNameSet()` to set the name of either a semaphores, a mutex, a mailbox or a message queue. You need to use `OSEventNameGet()` to obtain the name of either a semaphores, a mutex, a mailbox or a message queue.
- **Restriction:** none

ucosii.miscellaneous.os_max_events

- **Identifier:** OS_MAX_EVENTS
- **Type:** Decimal number
- **Default Value:** 60
- **Destination File:** `system.h`
- **Description:** Maximum number of event control blocks
- **Restriction:** none

ucosii.miscellaneous.os_task_idle_stk_size

- **Identifier:** OS_TASK_IDLE_STK_SIZE
- **Type:** Decimal number
- **Default Value:** 512
- **Destination File:** `system.h`
- **Description:** Idle task stack size
- **Restriction:** none

ucosii.miscellaneous.os_task_stat_stk_size

- **Identifier:** OS_TASK_STAT_STK_SIZE
- **Type:** Decimal number
- **Default Value:** 512
- **Destination File:** `system.h`
- **Description:** Statistics task stack size
- **Restriction:** none

ucosii.task.os_task_change_prio_en

- **Identifier:** OS_TASK_CHANGE_PRIO_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Include code for OSTaskChangePrio()
- **Restriction:** none

ucosii.task.os_task_create_en

- **Identifier:** OS_TASK_CREATE_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Include code for OSTaskCreate()
- **Restriction:** none

ucosii.task.os_task_create_ext_en

- **Identifier:** OS_TASK_CREATE_EXT_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Include code for OSTaskCreateExt()
- **Restriction:** none

ucosii.task.os_task_del_en

- **Identifier:** OS_TASK_DEL_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Include code for OSTaskDel()
- **Restriction:** none

ucosii.task.os_task_name_size

- **Identifier:** OS_TASK_NAME_EN
- **Type:** Decimal number
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** This constant determines whether you can assign names to tasks. If OS_TASK_NAME_EN is set to 0, this feature is disabled and no RAM is used in the OS_TCB for the task name..
- **Restriction:** none

ucosii.task.os_task_profile_en

- **Identifier:** OS_TASK_PROFILE_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Include data structure for run-time task profiling
- **Restriction:** none

ucosii.task.os_task_query_en

- **Identifier:** OS_TASK_QUERY_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- Destination File: `system.h`
- **Description:** Include code for OSTaskQuery
- **Restriction:** none

ucosii.task.os_task_suspend_en

- **Identifier:** OS_TASK_SUSPEND_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Include code for OSTaskSuspend() and OSTaskResume()
- **Restriction:** none

ucosii.task.os_task_sw_hook_en

- **Identifier:** OS_TASK_SW_HOOK_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Include code for OSTaskSwHook()
- **Restriction:** none

ucosii.time.os_time_tick_hook_en

- **Identifier:** OS_TIME_TICK_HOOK_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Include code for OSTimeTickHook()
- **Restriction:** none

ucosii.time.os_time_dly_resume_en

- **Identifier:** OS_TIME_DLY_RESUME_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Include code for OSTimeDlyResume()
- **Restriction:** none

ucosii.time.os_time_dly_hmsm_en

- **Identifier:** OS_TIME_DLY_HMSM_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Include code for OSTimeDlyHMSM()
- **Restriction:** none

ucosii.time.os_time_get_set_en

- **Identifier:** OS_TIME_GET_SET_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Include code for OSTimeGet and OSTimeSet()
- **Restriction:** none

ucosii.os_flag_en

- **Identifier:** OS_FLAG_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- Destination File: `system.h`
- **Description:** Enable code for Event Flags. This setting is enabled by default in MicroC-OS/II BSPs, because it is required for correct functioning of Intel FPGA device drivers and the HAL in a multithreaded environment. Avoid disabling it.
- **Restriction:** none

ucosii.event_flag.os_flag_wait_clr_en

- **Identifier:** OS_FLAG_WAIT_CLR_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Include code for Wait on Clear Event Flags. This setting is enabled by default in MicroC-OS/II BSPs, because it is required for correct functioning of Intel FPGA device drivers and the HAL in a multithreaded environment. Avoid disabling it.
- **Restriction:** none

ucosii.event_flag.os_flag_accept_en

- **Identifier:** OS_FLAG_ACCEPT_EN
- **Type:** Boolean assignment
- **Default Value:** 1

- **Destination File:** `system.h`
- **Description:** Include code for OSFlagAccept(). This setting is enabled by default in MicroC-OS/II BSPs, because it is required for correct functioning of Intel FPGA device drivers and the HAL in a multithreaded environment. Avoid disabling it.
- **Restriction:** none

ucosii.event_flag.os_flag_del_en

- **Identifier:** OS_FLAG_DEL_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Include code for OSFlagDel(). This setting is enabled by default in MicroC-OS/II BSPs, because it is required for correct functioning of Intel FPGA device drivers and the HAL in a multithreaded environment. Avoid disabling it.
- **Restriction:** none

ucosii.event_flag.os_flag_query_en

- **Identifier:** OS_FLAG_QUERY_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Include code for OSFlagQuery(). This setting is enabled by default in MicroC-OS/II BSPs, because it is required for correct functioning of Intel FPGA device drivers and the HAL in a multithreaded environment. Avoid disabling it.
- **Restriction:** none

ucosii.event_flag.os_flag_name_en

- **Identifier:** OS_FLAG_NAME_EN
- **Type:** Decimal number
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** This constant determines whether names can be assigned to event flag groups. If OS_FLAG_NAME_EN is set to 0, this feature is disabled.
- **Restriction:** none

ucosii.event_flag.os_flags_nbits

- **Identifier:** OS_FLAGS_NBITS
- **Type:** Decimal number
- **Default Value:** 16
- **Destination File:** `system.h`
- **Description:** Event Flag bits (8,16,32). CAUTION: This is required by the HAL and many Intel FPGA device drivers; use caution in changing this value.
- **Restriction:** none

ucosii.event_flag.os_max_flags

- **Identifier:** OS_MAX_FLAGS
- **Type:** Decimal number
- **Default Value:** 20
- **Destination File:** `system.h`
- **Description:** Maximum number of Event Flags groups. CAUTION: This is required by the HAL and many Intel FPGA device drivers; use caution in reducing this value.
- **Restriction:** none

ucosii.os_mutex_en

- **Identifier:** OS_MUTEX_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Enable code for Mutex Semaphores
- **Restriction:** none

ucosii.mutex.os_mutex_accept_en

- **Identifier:** OS_MUTEX_ACCEPT_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Include code for OSMutexAccept()
- **Restriction:** none

ucosii.mutex.os_mutex_del_en

- **Identifier:** OS_MUTEX_DEL_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Include code for OSMutexDel()
- **Restriction:** none

ucosii.mutex.os_mutex_query_en

- **Identifier:** OS_MUTEX_QUERY_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Include code for OSMutexQuery
- **Restriction:** none

ucosii.os_sem_en

- **Identifier:** OS_SEM_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Enable code for semaphores. This setting is enabled by default in MicroC-OS/II BSPs, because it is required for correct functioning of Intel FPGA device drivers and the HAL in a multithreaded environment. Avoid disabling it.
- **Restriction:** none

ucosii.semaphore.os_sem_accept_en

- **Identifier:** OS_SEM_ACCEPT_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Include code for OSSemAccept(). This setting is enabled by default in MicroC-OS/II BSPs, because it is required for correct functioning of Intel FPGA device drivers and the HAL in a multithreaded environment. Avoid disabling it.
- **Restriction:** none

ucosii.semaphore.os_sem_set_en

- **Identifier:** OS_SEM_SET_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Include code for OSSemSet(). This setting is enabled by default in MicroC-OS/II BSPs, because it is required for correct functioning of Intel FPGA device drivers and the HAL in a multithreaded environment. Avoid disabling it.
- **Restriction:** none

ucosii.semaphore.os_sem_del_en

- **Identifier:** OS_SEM_DEL_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Include code for OSSemDel(). This setting is enabled by default in MicroC-OS/II BSPs, because it is required for correct functioning of Intel FPGA device drivers and the HAL in a multithreaded environment. Avoid disabling it.
- **Restriction:** none

ucosii.semaphore.os_sem_query_en

- **Identifier:** OS_SEM_QUERY_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Include code for OSSemQuery(). This setting is enabled by default in MicroC-OS/II BSPs, because it is required for correct functioning of Intel FPGA device drivers and the HAL in a multithreaded environment. Avoid disabling it.
- **Restriction:** none

ucosii.os_mbox_en

- **Identifier:** OS_MBOX_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Enable code for mailboxes
- **Restriction:** none

ucosii.mailbox.os_mbox_accept_en

- **Identifier:** OS_MBOX_ACCEPT_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- Destination File: `system.h`
- **Description:** Include code for OSMboxAccept()
- **Restriction:** none

ucosii.mailbox.os_mbox_del_en

- **Identifier:** OS_MBOX_DEL_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Include code for OSMboxDel()
- **Restriction:** none

ucosii.mailbox.os_mbox_post_en

- **Identifier:** OS_MBOX_POST_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Include code for OSMboxPost()
- **Restriction:** none

ucosii.mailbox.os_mbox_post_opt_en

- **Identifier:** OS_MBOX_POST_OPT_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Include code for OSMboxPostOpt()
- **Restriction:** none

ucosii.mailbox.os_mbox_query_en

- **Identifier:** OS_MBOX_QUERY_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Include code for OSMboxQuery()
- **Restriction:** none

ucosii.os_q_en

- **Identifier:** OS_Q_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Enable code for Queues
- **Restriction:** none

ucosii.queue.os_q_accept_en

- **Identifier:** OS_Q_ACCEPT_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Include code for OSQAccept()
- **Restriction:** none

ucosii.queue.os_q_del_en

- **Identifier:** OS_Q_DEL_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Include code for OSQDel()
- **Restriction:** none

ucosii.queue.os_q_flush_en

- **Identifier:** OS_Q_FLUSH_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Include code for OSQFlush()
- **Restriction:** none

ucosii.queue.os_q_post_en

- **Identifier:** OS_Q_POST_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Include code of OSQFlush()
- **Restriction:** none

ucosii.queue.os_q_post_front_en

- **Identifier:** OS_Q_POST_FRONT_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Include code for OSQPostFront()
- **Restriction:** none

ucosii.queue.os_q_post_opt_en

- **Identifier:** OS_Q_POST_OPT_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- Destination File: `system.h`
- **Description:** Include code for OSQPostOpt()
- **Restriction:** none

ucosii.queue.os_q_query_en

- **Identifier:** OS_Q_QUERY_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Include code for OSQQuery()
- **Restriction:** none

ucosii.queue.os_max_qs

- **Identifier:** OS_MAX_QS
- **Type:** Decimal number
- **Default Value:** 20
- **Destination File:** `system.h`
- **Description:** Maximum number of Queue Control Blocks
- **Restriction:** none

ucosii.os_mem_en

- **Identifier:** OS_MEM_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Enable code for memory management
- **Restriction:** none

ucosii.memory.os_mem_query_en

- **Identifier:** OS_MEM_QUERY_EN
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** Include code for OSMemQuery()
- **Restriction:** none

ucosii.memory.os_mem_name_en

- **Identifier:** OS_MEM_NAME_EN
- **Type:** Decimal number
- **Default Value:** 1
- **Destination File:** `system.h`
- **Description:** This constant determines whether names can be assigned to memory partitions. If OS_MEM_NAME_EN is set to 0, this feature is disabled and no RAM is used in the OS_MEM for the memory partition for storage of names.
- **Restriction:** none

ucosii.memory.os_max_mem_part

- **Identifier:** OS_MAX_MEM_PART
- **Type:** Decimal number
- **Default Value:** 60
- **Destination File:** `system.h`
- **Description:** Maximum number of memory partitions
- **Restriction:** none

ucosii.os_tmr_en

- **Identifier:** OS_TMR_EN
- **Type:** Boolean assignment
- **Default Value:** 0
- **Destination File:** `system.h`
- **Description:** Enable code for timers
- **Restriction:** none

ucosii.timer.os_task_tmr_stk_size

- **Identifier:** OS_TASK_TMR_STK_SIZE
- **Type:** Decimal number
- **Default Value:** 512
- **Destination File:** `system.h`
- **Description:** Stack size for timer task
- **Restriction:** none

ucosii.timer.os_task_tmr_prio

- **Identifier:** OS_TASK_TMR_PRIO
- **Type:** Decimal number
- **Default Value:** 2
- **Destination File:** `system.h`
- **Description:** Priority of timer task (0=highest)
- **Restriction:** none

ucosii.timer.os_tmr_cfg_max

- **Identifier:** OS_TMR_CFG_MAX
- **Type:** Decimal number
- **Default Value:** 16
- **Destination File:** `system.h`
- **Description:** Maximum number of timers
- **Restriction:** none

ucosii.timer.os_tmr_cfg_name_size

- **Identifier:** OS_TMR_CFG_NAME_SIZE
- **Type:** Decimal number
- **Default Value:** 16
- **Destination File:** `system.h`
- **Description:** Size of timer name
- **Restriction:** none

ucosii.timer.os_tmr_cfg_ticks_per_sec

- **Identifier:** OS_TMR_CFG_TICKS_PER_SEC
- **Type:** Decimal number
- **Default Value:** 10
- **Destination File:** `system.h`
- **Description:** Rate at which timer management task runs (Hz)
- **Restriction:** none

ucosii.timer.os_tmr_cfg_wheel_size

- **Identifier:** OS_TMR_CFG_WHEEL_SIZE
- **Type:** Decimal number
- **Default Value:** 2
- **Destination File:** `system.h`
- **Description:** Size of timer wheel (number of spokes)
- **Restriction:** none

16.4.3.3. FreeRTOS BSP

freertos.heap_method

- **Identifier:** HEAP_METHOD
- **Type:** Decimal number
- **Default Value:** 4
- **Destination File:** `system.h`
- **Description:** Choose and implement one of the following `heap_x.c` source files:
  - `heap_1.c` – Simplest and does not permit memory to be freed.
  - `heap_2.c` – Permits memory to be freed, but does not coalescence adjacent free blocks.
  - `heap_3.c` – Wraps the standard `malloc()` and `free()` for thread safety.
  - `heap_4.c` – Coalescences adjacent free blocks to avoid fragmentation, and includes absolute address placement option.
  - `heap_5.c` – As per `heap_4.c`, in addition to span heap across multiple non-adjacent memory areas.
- **Restriction:** Valid inputs as 1,2,3,4, and 5 only

freertos.os_port_has_mtime

- **Identifier:** OS_PORT_HAS_MTIME
- **Type:** Boolean definition
- **Default Value:** true
- **Destination File:** `system.h`
- **Description:** OS port has Machine Timer driver.
- **Restriction:** none

freertos.os_thread_safe_newlib

- **Identifier:** OS_THREAD_SAFE_C_LIBRARY
- **Type:** Boolean definition
- **Default Value:** true
- **Destination File:** `system.h`
- **Description:** Enable thread-safe C library.
- **Restriction:** none

freertos.os_use_preemption

- **Identifier:** OS_USE_PREEMPTION
- **Type:** Boolean definition
- **Default Value:** true
- **Destination File:** `system.h`
- **Description:** Enable OS preemption.
- **Restriction:** none

freertos.os_use_16_bit_ticks

- **Identifier:** OS_USE_16_BIT_TICKS
- **Type:** Boolean definition
- **Default Value:** false
- **Destination File:** `system.h`
- **Description:** Configure tick type. Select true for 16-bits, or false for 32-bits.
- **Restriction:** none

freertos.os_max_priorities

- **Identifier:** OS_MAX_PRIORITIES
- **Type:** Decimal number
- **Default Value:** 7
- **Destination File:** `system.h`
- **Description:** Maximum number of priorities supported.
- **Restriction:** none

freertos.os_min_stack_size

- **Identifier:** OS_MIN_STACK_SIZE
- **Type:** Decimal number
- **Default Value:** 300
- **Destination File:** `system.h`
- **Description:** OS minimum stack size.
- **Restriction:** none

freertos.task.os_task_priority_set_en

- **Identifier:** OS_TASK_PRIORITY_SET_EN
- **Type:** Boolean definition
- **Default Value:** true
- **Destination File:** `system.h`
- **Description:** Enable `vTaskPrioritySet()`.
- **Restriction:** none

freertos.task.os_task_priority_get_en

- **Identifier:** OS_TASK_PRIORITY_GET_EN
- **Type:** Boolean definition
- **Default Value:** true
- **Destination File:** `system.h`
- **Description:** Enable `uxTaskPriorityGet()`.
- **Restriction:** none

freertos.task.os_task_delete_en

- **Identifier:** OS_TASK_DELETE_EN
- **Type:** Boolean definition
- **Default Value:** true
- **Destination File:** `system.h`
- **Description:** Enable `vTaskDelete()`.
- **Restriction:** none

freertos.task.os_task_clean_up_resources_en

- **Identifier:** OS_TASK_CLEAN_UP_RESOURCES_EN
- **Type:** Boolean definition
- **Default Value:** true
- **Destination File:** `system.h`
- **Description:** Enable `vTaskCleanUpResources()`.
- **Restriction:** none

freertos.task.os_task_suspend_en

- **Identifier:** OS_TASK_SUSPEND_EN
- **Type:** Boolean definition
- **Default Value:** true
- **Destination File:** `system.h`
- **Description:** Enable `vTaskSuspend()`.
- **Restriction:** none

freertos.task.os_task_delay_until_en

- **Identifier:** OS_TASK_DELAY_UNTIL_EN
- **Type:** Boolean definition
- **Default Value:** true
- **Destination File:** `system.h`
- **Description:** Enable `vTaskDelayUntil()`.
- **Restriction:** none

freertos.task.os_task_delay_en

- **Identifier:** OS_TASK_DELAY_EN
- **Type:** Boolean definition
- **Default Value:** true
- **Destination File:** `system.h`
- **Description:** Enable `vTaskDelay()`.
- **Restriction:** none

freertos.task.os_task_get_state_en

- **Identifier:**OS_TASK_GET_STATE_EN
- **Type:** Boolean definition
- **Default Value:** true
- **Destination File:** `system.h`
- **Description:** Enable `eTaskGetState()`.
- **Restriction:** none

freertos.task.os_task_abort_delay_en

- **Identifier:** OS_TASK_ABORT_DELAY_EN
- **Type:** Boolean definition
- **Default Value:** true
- **Destination File:** `system.h`
- **Description:** Enable `xTaskAbortDelay()`.
- **Restriction:** none

freertos.task.os_task_get_handle_en

- **Identifier:** OS_TASK_GET_HANDLE_EN
- **Type:** Boolean definition
- **Default Value:** true
- **Destination File:** `system.h`
- **Description:** Enable `xTaskGetHandle()`.
- **Restriction:** none

freertos.task.os_use_idle_hook

- **Identifier:** OS_USE_IDLE_HOOK
- **Type:** Boolean definition
- **Default Value:** false
- **Destination File:** `system.h`
- **Description:** Include OS idle hook – `vApplicationIdleHook()`.
- **Restriction:** none

freertos.task.os_max_task_name_len

- **Identifier:** OS_MAX_TASK_NAME_LEN
- **Type:** Decimal number
- **Default Value:** 16
- **Destination File:** `system.h`
- **Description:** Maximum task name length.
- **Restriction:** none

freertos.task.os_tls_bytes

- **Identifier:** OS_TLS_BYTES_PER_TASK
- **Type:** Decimal number
- **Default Value:** 256
- **Destination File:** `system.h`
- **Description:** Number of bytes for thread local support variables which map to the `.tdata` and `.tbss` program sections. The value can be determined by referring to the `_tls_size` linker symbol.
- **Restriction:** none

freertos.task.os_use_timers

- **Identifier:** OS_USE_TIMERS
- **Type:** Boolean definition
- **Default Value:** true
- **Destination File:** `system.h`
- **Description:** Use OS timers.
- **Restriction:** none

freertos.task.os_timer_pend_function_call_en

- **Identifier:** OS_TIMER_PEND_FUNCTION_CALL_EN
- **Type:** Boolean definition
- **Default Value:** true
- **Destination File:** `system.h`
- **Description:** Enable `xTimerPendFunctionCall()`.
- **Restriction:** none

freertos.task.os_use_tick_hook

- **Identifier:** OS_USE_TICK_HOOK
- **Type:** Boolean definition
- **Default Value:** true
- **Destination File:** `system.h`
- **Description:** Include OS tick hook – `vApplicationTickHook()`.
- **Restriction:** none

freertos.task.os_timer_queue_len

- **Identifier:** OS_TIMER_QUEUE_LEN
- **Type:** Decimal number
- **Default Value:** 8
- **Destination File:** `system.h`
- **Description:** Timer queue length.
- **Restriction:** none

freertos.task.os_timer_task_depth

- **Identifier:** OS_TIMER_TASK_DEPTH
- **Type:** Decimal number
- **Default Value:** 160
- **Destination File:** `system.h`
- **Description:** OS timer task depth.
- **Restriction:** none

freertos.mutex.os_mutex_en

- **Identifier:** OS_MUTEX_EN
- **Type:** Boolean definition
- **Default Value:** true
- **Destination File:** `system.h`
- **Description:** Enable mutex semaphores.
- **Restriction:** none

freertos.mutex.os_recursive_mutex_en

- **Identifier:**
- **Type:** Boolean definition
- **Default Value:** true
- **Destination File:** `system.h`
- **Description:** Enable recursive mutex.
- **Restriction:** none

freertos.semaphore.os_counting_sem_en

- **Identifier:** OS_COUNTING_SEM_EN
- **Type:** Boolean definition
- **Default Value:** true
- **Destination File:** `system.h`
- **Description:** Enable counting semaphores.
- **Restriction:** none

freertos.semaphore.os_semaphore_get_mutex_holder_en

- **Identifier:** OS_SEMAPHORE_GET_MUTEX_HOLDER_EN
- **Type:** Boolean definition
- **Default Value:** true
- **Destination File:** `system.h`
- **Description:** Enable `xSemaphoreGetMutexHolder()`.
- **Restriction:** none

freertos.queue.os_use_q_sets

- **Identifier:** OS_USE_QUEUE_SETS
- **Type:** Boolean definition
- **Default Value:** true
- **Destination File:** `system.h`
- **Description:** Use OS queue sets.
- **Restriction:** none

freertos.queue.os_q_reg_size

- **Identifier:** OS_Q_REG_SIZE
- **Type:** Decimal number
- **Default Value:** 8
- **Destination File:** `system.h`
- **Description:** OS queue registry size.
- **Restriction:** none

freertos.memory.os_memory_dynamic_allocation_en

- **Identifier:** OS_MEMORY_DYNAMIC_ALLOCATION_EN
- **Type:** Boolean definition
- **Default Value:** true
- **Destination File:** `system.h`
- **Description:** Enable memory dynamic allocation.
- **Restriction:** none

freertos.memory.os_malloc_failed_hook_en

- **Identifier:** OS_MALLOC_FAILED_HOOK_EN
- **Type:** Boolean definition
- **Default Value:** true
- **Destination File:** `system.h`
- **Description:** Enable malloc failed hook.
- **Restriction:** none

freertos.memory.os_total_heap_size

- **Identifier:** OS_TOTAL_HEAP_SIZE
- **Type:** Decimal number
- **Default Value:** 80000
- **Destination File:** `system.h`
- **Description:** OS total heap size.
- **Restriction:** none

freertos.miscellaneous.os_use_trace_facility

- **Identifier:** OS_USE_TRACE_FACILITY
- **Type:** Boolean definition
- **Default Value:** true
- **Destination File:** `system.h`
- **Description:** Use OS trace facility.
- **Restriction:** none

freertos.miscellaneous.os_check_stack_overflow

- **Identifier:** OS_CHECK_STACK_OVERFLOW
- **Type:** Boolean definition
- **Default Value:** true
- **Destination File:** `system.h`
- **Description:** Include OS stack overflow checking – `vApplicationStackOverflowHook()`.
- **Restriction:** none

freertos.miscellaneous.os_idle_should_yield

- **Identifier:** OS_IDLE_SHOULD_YIELD
- **Type:** Boolean definition
- **Default Value:** false
- **Destination File:** `system.h`
- **Description:** Control the behavior of tasks at the idle priority.
- **Restriction:** none

16.4.3.4. Device Drivers BSP

altera_avalon_uart_driver.enable_small_driver

- **Identifier:** ALTERA_AVALON_UART_SMALL
- **Type:** Boolean definition
- **Default Value:** false
- **Destination File:** `toolchain.cmake`
- **Description:** Small-footprint (polled mode) driver
- **Restriction:** none

altera_avalon_uart_driver.enable_ioctl

- **Identifier:** ALTERA_AVALON_UART_USE_IOCTL
- **Type:** Boolean definition
- **Default Value:** false
- **Destination File:** `toolchain.cmake`
- **Description:** Enable driver ioctl() support. This feature is not compatible with the small driver; ioctl() support is not compiled if either the UART `enable_small_driver` or the HAL `enable_reduced_device_drivers` setting is enabled.
- **Restriction:** none

altera_avalon_jtag_uart_driver.enable_small_driver

- **Identifier:** ALTERA_AVALON_JTAG_UART_SMALL
- **Type:** Boolean definition
- **Default Value:** false
- **Destination File:** `toolchain.cmake`
- **Description:** Small-footprint (polled mode) driver
- **Restriction:** none

altera_avalon_uart_driver.enable_jtag_uart_ignore_fifo_full_error

- **Identifier:** ALTERA_AVALON_JTAG_UART_IGNORE_FIFO_FULL_ERROR
- **Type:** Boolean definition
- **Default Value:** false
- **Destination File:** `toolchain.cmake`
- **Description:** JTAG UART FIFO full error after timeout
- **Restriction:** none

intel_niosv_m_hal_driver.internal_timer_ticks_per_sec

- **Identifier:** NIOSV_INTERNAL_TIMER_TICKS_PER_SECOND
- **Type:** Decimal Number
- **Default Value:** 1000

- **Destination File:** `system.h`
- **Description:**
  - Replace CLOCKS_PER_SEC
  - Implement as configTICK_RATE_HZ in FreeRTOS to configure the period of each time slice.
- **Restriction:** none

16.5. Board Support Package Tcl Commands

BSP Tcl commands are a crucial component of the Nios V processor tools. Tcl commands allow you to exercise detailed control over BSP generation, as well as to define drivers and software packages. This section describes the Tcl commands, the environments in which they run, and how the commands work together.

Related Information

- Assumptions and Requirements on page 138
- AN 459: Guidelines for Developing a Nios II HAL Device Driver
  For more information about integrating a HAL device driver.

16.5.1. Tcl Command Environments

The Nios V processor tools supports Tcl commands in the following environments:

- BSP setting specification—In this environment, you manipulate BSP settings to control the static characteristics of the BSP. BSP setting commands are executed before the BSP is generated.
- BSP generation callbacks—In this environment, you exercise further control over BSP details, managing settings that interact with one another and with the hardware design. BSP callbacks run at BSP generation time.
- Device driver and software package definition—In this environment, you bundle source files into a custom driver or package. This process prepares the driver or package so that a BSP developer can include it in a BSP using the BSP Editor.

The following sections describe each Tcl environment in detail, listing the available commands.

16.5.2. Tcl Commands for BSP Settings

There are two ways to use BSP Tcl commands to manipulate project settings:

- Calling the Tcl commands by using the `--cmd` option of the **niosv-bsp** utility
- Adding the Tcl commands in a Tcl script specified with the `--script` option of the **niosv-bsp** utility

16.5.2.1. add_memory_device

Usage

`add_memory_device` *<device name> <base address> *

Options

- *<device name>*: String with the name of the memory device.
- *<base address>*: The base address of the memory device. Hexadecimal or decimal string.
- **: The size (span) of the memory device. Hexadecimal or decimal string.

Description

This command is provided to define a user-defined external memory device, outside the hardware system. Such a device would typically be mapped through a bridge component. This command adds an external memory device to the BSP's memory map, allowing the BSP to define memory regions and section mappings for the memory as if it were part of the system. The external memory device parameters are stored in the BSP settings file.

16.5.2.2. add_memory_region

Usage

`add_memory_region <name> <slave_desc> <offset> <span>`

Options

- *<name>*: String with the name of the memory region to create.
- *<slave_desc>*: String with the slave descriptor of the memory device for this region.
- *<offset>*: String with the byte offset of the memory region from the memory device base address.
- **: String with the span of the memory region in bytes.

Description

Creates a new memory region for the linker script. This memory region must not overlap with any other memory region and must be within the memory range of the associated slave descriptor. The offset and span are decimal numbers unless prefixed with 0x.

Example

```
add_memory_region onchip_ram0 onchip_ram0 0 0x100000
```

16.5.2.3. add_section_mapping

Usage

`add_section_mapping <section_name> <memory_region_name>`

Options

- *<section_name>*: String with the name of the linker section.
- *<memory_region_name>*: String with the name of the memory region to map.

Description

Maps the specified linker section to the specified linker memory region. If the section does not already exist, `add_section_mapping` creates it. If it already exists, `add_section_mapping` overrides the existing mapping with the new one. The linker creates the section mappings in the order in which they appear in the linker script.

Example

```
add_section_mapping .text onchip_ram0
```

16.5.2.4. are_same_resource

Usage

`are_same_resource <slave_desc1> <slave_desc2>`

Options

- *<slave_desc1>*: String with the first slave descriptor to compare.
- *<slave_desc2>*: String with the second slave descriptor to compare.

Description

Returns a boolean value that indicates whether the two slave descriptors are connected to the same resource. To connect to the same resource, the two slave descriptors must be associated with the same module. The module specifies whether two slaves access the same resource or different resources within that module. For example, a dual-port memory has two slaves that access the same resource (the memory). However, you could create a module that has two slaves that access two different resources such as a memory and a control port.

16.5.2.5. delete_memory_region

Usage

`delete_memory_region <region_name>`

Options

- *<region_name>*: String with the name of the memory region to delete.

Description

Deletes the specified memory region. The region must exist to avoid an error condition.

16.5.2.6. delete_section_mapping

Usage

`delete_section_mapping <section_name>`

Options

- <section_name>: String with the name of the section.

Description

Deletes the specified section mapping.

Example

```
delete_section_mapping .text
```

16.5.2.7. disable_sw_package

Usage

```
disable_sw_package <software_package_name>
```

Options

- <*software_package_name*>: String with the name of the software package.

Description

Disables the specified software package. Settings belonging to the package are no longer available in the BSP, and associated source files are not included in the CMakeLists.txt. It is an error to disable a software package that is not enabled.

16.5.2.8. enable_sw_package

Usage

```
enable_sw_package <software_package_name>
```

Options

- <*software_package_name*>: String with the name of the software package, with the version number optionally appended with a ':'.

Description

Enables a software package. Adds its associated source files and settings to the BSP. Specify the desired version in the form <*software_package_name*>:<*version*>. If you do not specify the version, `enable_sw_package` selects the latest available version.

Examples

- Example 1:`enable_sw_package altera_safeclib:19.1`
- Example 2:`enable_sw_package my_sw_package`

16.5.2.9. get_addr_span

Usage

```
get_addr_span <slave_desc>
```

Options

- <*slave_desc*>: String with the slave descriptor to query.

Description

Returns the address span (length in bytes) of the slave descriptor as an integer decimal number.

Example

`puts [get_addr_span onchip_ram_64_kbytes]` Returns: `65536`

16.5.2.10. get_assignment

Usage

`get_assignment <module_name> <assignment_name>`

Options

- *<module_name>*: Module instance name to query for assignment
- *<assignment_name>*: Module instance assignment name to query for

Description

Returns the name of the value of the assignment for a specified module instance name.

Example

```
puts [get_assignment "cpu0"
"embeddedsw.configuration.breakSlave"]
```

Returns: `memory_0.s0`

16.5.2.11. get_available_drivers

Usage

`get_available_drivers <module_name>`

Options

- *<module_name>*: String with the name of the module to query.

Description

Returns a list of available device driver names that are compatible with the specified module instance. The list is empty if there are no drivers available for the specified slave descriptor. The format of each entry in the list is the driver name followed by a colon and the version number (if provided).

Example

`puts [get_available_drivers jtag_uart]`

Returns: `altera_avalon_jtag_uart_driver:7.2 altera_avalon_jtag_uart_driver:6.1`

16.5.2.12. get_available_sw_packages

Usage

```
get_available_sw_packages
```

Options

None

Description

Returns a list of software package names that are available for the current BSP. The format of each entry in the list is a string containing the package name followed by a colon and the version number (if provided).

Example

```
puts [get_available_sw_packages]
```

Returns:

```
altera_quad_seven_seg:19.1

altera_safeclib:19.1
```

16.5.2.13. get_base_addr

Usage

```
get_base_addr <slave_desc>
```

Options

- *<slave_desc>*: String with the slave descriptor to query.

Description

Returns the base byte address of the slave descriptor as an integer decimal number.

Example

```
puts [get_base_addr jtag_uart]
```

Returns: 67616

16.5.2.14. get_break_offset

Usage

```
get_break_offset
```

Options

None

Description

Returns the byte offset of the processor break address.

Example

```
puts [get_break_offset] Returns: 32
```

16.5.2.15. get_break_slave_desc

Usage

```
get_break_slave_desc
```

Options

None

Description

Returns the slave descriptor associated with the processor break address. If null, then the break device is internal to the processor (debug module).

Example

```
puts [get_break_slave_desc]
```

Returns: `onchip_ram_64_kbytes`

16.5.2.16. get_cpu_name

Usage

```
get_cpu_name
```

Options

None

Description

Returns the name of the BSP specific processor.

Example

```
puts [get_cpu_name] Returns: cpu_0
```

16.5.2.17. get_current_memory_regions

Usage

get_current_memory_regions

Options

None

Description

Returns a sorted list of records representing the existing linker script memory regions. Each record in the list represents a memory region. Each record is a list containing the region name, associated memory device slave descriptor, offset, and span, in that order.

Example

```
puts [get_current_memory_regions]
```

Returns: `{reset onchip_ram0 0 32} {onchip_ram0 onchip_ram0 32 1048544}`

16.5.2.18. get_current_section_mappings

Usage

```
get_current_section_mappings
```

Options

None

Description

Returns a list of lists for all the current section mappings. Each list represents a section mapping with the format {section_name memory_region}. The order of the section mappings matches their order in the linker script.

Example

```
puts [get_current_section_mappings]
```

Returns: `{.text onchip_ram0} {.rodata onchip_ram0} {.rwdata onchip_ram0} {.bss onchip_ram0} {.heap onchip_ram0} {.stack onchip_ram0}`

16.5.2.19. get_default_memory_regions

Usage

```
get_default_memory_regions
```

Options

None

Description

Returns a sorted list of records representing the default linker script memory regions. The default linker script memory regions are the best guess for memory regions based on the reset address and exception address of the processor associated with the BSP, and all other processors in the system that share memories with the processor associated with the BSP. Each record in the list represents a memory region. Each record is a list containing the region name, associated memory device slave descriptor, offset, and span, in that order.

Example

```
puts [get_default_memory_regions]
```

Returns:
`{reset onchip_ram0 0 32} {onchip_ram0 onchip_ram0 32 1048544}`

16.5.2.20. get_driver

Usage

```
get_driver <module_name>
```

Options

- *<module_name>*: String with the name of the module instance to query.

Description

Returns the driver name associated with the specified module instance. The format is *<driver name>* followed by a colon and the version (if provided). Returns the string "none" if there is no driver associated with the specified module instance name.

Examples

- Example 1:`puts [get_driver jtag_uart]`

  Returns:
  `altera_avalon_jtag_uart_driver:7.2`
- Example 2:`puts [get_driver onchip_ram_64_kbytes]`

  Returns:
  none

16.5.2.21. get_enabled_sw_packages

Usage

```
get_enabled_sw_packages
```

Options

None

Description

Returns a list of currently enabled software packages. The format of each entry in the list is the software package name followed by a colon and the version number (if provided).

Example

```
puts [get_enabled_sw_packages]
```

Returns:
`altera_safeclib:19.1`

16.5.2.22. get_exception_offset

Usage

```
get_exception_offset
```

Options

None

Description

Returns the byte offset of the processor exception address.

Example

```
puts [get_exception_offset]
```

Returns:
32

16.5.2.23. get_exception_slave_desc

Usage

```
get_exception_slave_desc
```

Options

None

Description

Returns the slave descriptor associated with the processor exception address.

Example

```
puts [get_exception_slave_desc]
```

Returns:
`onchip_ram_64_kbytes`

16.5.2.24. get_fast_tlb_miss_exception_offset

Usage

```
get_fast_tlb_miss_exception_offset
```

Options

None

Description

Returns the byte offset of the processor fast translation lookaside buffer (TLB) miss exception address. Only a processor with an MMU has such an exception address.

Example

```
puts [get_fast_tlb_miss_exception_offset]
```

Returns:
32

16.5.2.25. get_fast_tlb_miss_exception_slave_desc

Usage

```
get_fast_tlb_miss_exception_slave_desc
```

Options

None

Description

Returns the slave descriptor associated with the processor fast TLB miss exception address. Only a processor with an MMU has such an exception address.

Example

```
puts [get_fast_tlb_miss_exception_slave_desc]
```

Returns:
`onchip_ram_64_kbytes`

16.5.2.26. get_interrupt_controller_id

Usage

`get_interrupt_controller_id <slave_desc>`

Options

- <*slave_desc*>: String with the slave descriptor to query.

Description

Returns the interrupt controller ID of the slave descriptor (-1 if not a connected interrupt controller).

16.5.2.27. get_irq_interrupt_controller_id

Usage

`get_irq_interrupt_controller_id <slave_desc>`

Options

- <*slave_desc*>: String with the slave descriptor to query.

Description

Returns the interrupt controller ID connected to the IRQ associated with the slave descriptor (-1 if none).

16.5.2.28. get_irq_number

Usage

`get_irq_number <slave_desc>`

Options

- <*slave_desc*>: String with the slave descriptor to query.

Description

Returns the interrupt request number of the slave descriptor, or -1 if no interrupt request number is found.

16.5.2.29. get_memory_region

Usage

```
get_memory_region <name>
```

Options

- <*name*>: String with the name of the memory region.

Description

Returns the linker script region information for the specified region. The format of the region is a list containing the region name, associated memory device slave descriptor, offset, and span in that order.

Example

```
puts [get_memory_region reset]
```

Returns:

```
reset onchip_ram0 0 32
```

16.5.2.30. get_module_class_name

Usage

```
get_module_class_name <module_name>
```

Options

- <*module_name*>: String with the module instance name to query.

Description

Returns the name of the module class associated with the module instance.

Example

```
puts [get_module_class_name jtag_uart0]
```

Returns:

```
altera_avalon_jtag_uart
```

16.5.2.31. get_module_name

Usage

```
get_module_name <slave_desc>
```

Options

- <*slave_desc*>: String with the slave descriptor to query.

Description

Returns the name of the module instance associated with the slave descriptor. If a module with one slave, or if it has multiple slaves connected to the same resource, the slave descriptor is the same as the module name. If a module has multiple slaves that do not connect to the same resource, the slave descriptor consists of the module name followed by an underscore and the slave name.

Example

```
puts [get_module_name multi_jtag_uart0_s1]
```

Returns:
```
multi_jtag_uart0
```

16.5.2.32. get_reset_offset

Usage

```
get_reset_offset
```

Options

None

Description

Returns the byte offset of the processor reset address.

Example

```
puts [get_reset_offset]
```

Returns:
```
0
```

16.5.2.33. get_reset_slave_desc

Usage

```
get_reset_slave_desc
```

Options

None

Description

Returns the slave descriptor associated with the processor reset address.

Example

```
puts [get_reset_slave_desc]
```

Returns:
```
onchip_ram_64_kbytes
```

16.5.2.34. get_section_mapping

Usage

```
get_section_mapping <section_name>
```

Options

- <*section_name*>: String with the section name to query.

Description

Returns the name of the memory region for the specified linker section. Returns null if the linker section does not exist.

Example

```
puts [get_section_mapping .text]
```

Returns:
`onchip_ram0`

16.5.2.35. get_setting

Usage

```
get_setting <name>
```

Options

- <*name*>: String with the name of the setting to get.

Description

Returns the value of the specified BSP setting. `get_setting` returns boolean settings with the value 1 or 0. If the value of the setting is an empty string, `get_setting` returns `"none"`.

Example

```
puts [get_setting hal.enable_c_plus_plus]
```

Returns:
`0`

16.5.2.36. get_setting_desc

Usage

```
get_setting_desc <name>
```

Options

- <*name*>: String with the name of the setting to get the description for.

Description

Returns a string describing the BSP setting.

Example

```
puts [get_setting_desc hal.enable_c_plus_plus]
```

Returns:

```
Enable support for a subset of the C++ language.
This option increases code footprint by adding support for C++ constructors.
Certain features, such as multiple inheritance and exceptions are not supported.
If false, adds -DALT_NO_C_PLUS_PLUS to ALT_CPPFLAGS in toolchain.cmake, and
reduces code footprint.
```

16.5.2.37. get_slave_descs

Usage

```
get_slave_descs
```

Options

None

Description

Returns a sorted list of all the slave descriptors connected to the Nios V processor.

Example

```
puts [get_slave_descs]
```

Returns:

```
jtag_uart0

onchip_ram0
```

16.5.2.38. is_char_device

Usage

```
is_char_device <slave_desc>
```

Options

- *<slave_desc>*: String with the slave descriptor to query.

Description

Returns a boolean value that indicates whether the slave descriptor is a character device.

Examples

- Example 1:`puts [is_char_device jtag_uart]`

  Returns:
  `1`
- Example 2:`puts [is_char_device onchip_ram_64_kbytes]`

  Returns:
  `0`

16.5.2.39. is_connected_interrupt_controller_device

Usage

```
is_connected_interrupt_controller_device <slave_desc>
```

Options

- <*slave_desc*>: String with the slave descriptor to query.

Description

Returns a boolean value that indicates whether the slave descriptor is an interrupt controller device that is connected to the processor so that the interrupt controller sends interrupts to the processor.

16.5.2.40. is_connected_to_data_master

Usage

```
is_connected_to_data_master <slave_desc>
```

Options

- <*slave_desc*>: String with the slave descriptor to query.

Description

Returns a boolean value that indicates whether the slave descriptor is connected to a data master.

16.5.2.41. is_connected_to_instruction_master

Usage

```
is_connected_to_instruction_master <slave_desc>
```

Options

- <*slave_desc*>: String with the slave descriptor to query.

Description

Returns a boolean value that indicates whether the slave descriptor is connected to an instruction master.

16.5.2.42. is_ethernet_mac_device

Usage

```
is_ethernet_mac_device <slave_desc>
```

Options

- <*slave_desc*>: String with the slave descriptor to query.

Description

Returns a boolean value that indicates whether the slave descriptor is an Ethernet MAC device.

16.5.2.43. is_flash

Usage

```
is_flash <slave_desc>
```

Options

- <*slave_desc*>: String with the slave descriptor to query.

Description

Returns a boolean value that indicates whether the slave descriptor is a flash memory device.

16.5.2.44. is_memory_device

Usage

```
is_memory_device <slave_desc>
```

Options

- <*slave_desc*>: String with the slave descriptor to query.

Description

Returns a boolean value that indicates whether the slave descriptor is a memory device.

Examples

- Example 1:`puts [is_memory_device jtag_uart]`

  Returns:`0`
- Example 2:`puts [is_memory_device onchip_ram_64_kbytes]`

  Returns:`1`

16.5.2.45. is_non_volatile_storage

Usage

```
is_non_volatile_storage <slave_desc>
```

Options

<*slave_desc*>: String with the slave descriptor to query.

Description

Returns a boolean value that indicates whether the slave descriptor is a non-volatile storage device.

16.5.2.46. is_timer_device

Usage

```
is_timer_device <slave_desc>
```

Options

<*slave_desc*>: String with the slave descriptor to query.

Description

Returns a boolean value that indicates whether the slave descriptor is a timer device.

16.5.2.47. log_debug

Usage

`log_debug <message>`

Options

<*message*>: String with message to log.

Description

Displays a message to the host's `stdout` when the logging level is `debug`.

16.5.2.48. log_default

Usage

log_default <*message*>

Options

- <*message*>: String with message to log.

Description

Displays a message to the host's `stdout` when the logging level is `default` or higher.

Example

```
log_default "This is a default message."
```

Displays:

```
INFO: Tcl message: "This is a default message."
```

16.5.2.49. log_error

Usage

log_error <*message*>

Options

- <*message*>: String with message to log.

Description

Displays a message to the host's `stderr`, regardless of logging level.

16.5.2.50. log_verbose

Usage

log_verbose <*message*>

Options

- <*message*>: String with message to log.

Description

Displays a message to the host's `stdout` when the logging level is `verbose` or higher.

16.5.2.51. set_driver

Usage

`set_driver <driver_name> <module_name>`

Options

- <*driver_name*>: String with the name of the device driver to use.
- <*module_name*>: String with the name of the module instance to set.

Description

Selects the specified device driver for the specified module instance. The <driver_name> argument includes a version number, delimited by a colon (:). If you omit the version number, set_driver uses the latest available version of the driver that is compatible with the component specified by the <module_name> argument.

If <*driver_name*> is none, the specified module instance does not use a driver. If <*driver_name*> is not none, it must be the name of the associated component class.

Examples

- Example 1:`set_driver altera_avalon_jtag_uart_driver:7.2 jtag_uart`
- Example 2:`set_driver none jtag_uart`

16.5.2.52. set_ignore_file

Usage

`set_ignore_file <software_component_name> <file_name> <ignore>`

Options

- <`software_component_name`>: Name of the driver, software package, or operating system to which the file belongs.
- <*file_name*>: Name of the file.
- <*ignore*>: Set to true to ignore (not generate or copy) the file, false to generate or copy the file normally.

Description

You can use this command to have a specific BSP file ignored (not generated or copied) during BSP generation. This command allows you to take ownership of a specific file, modify it, and prevent the Nios V processor BSP Tools from overwriting your modifications.

<software_component_name> can have one of the following values:

- *<driver_name>*—The name of a driver, as specified with the `create_driver` command in the `*_sw.tcl` file that defines the driver. Specifies that *<file_name>* is a copied file associated with a device driver.
- *<software_package_name>*—The name of a software package, specified with the `create_sw_package` command in the `*_sw.tcl` file that defines the package. Specifies that *<file_name>* is a copied file associated with a software package.
- *<OS_name>*—The name of an OS, specified with the `create_os` command in the `*_sw.tcl` file that defines the OS, and is used in the **niosv-bsp --create** to specify the BSP type. Specifies that *<file_name>* is a copied file associated with an OS.
- `generated`—Specifies that *<file_name>* is a generated top-level BSP file. The list of generated BSP files depends on the BSP type.

Note: For a list of generated files associated with a third-party OS, refer to the OS supplier's documentation.

Related Information

Nios V Processor Board Support Package Editor on page 53
For more information about the BSP Package Editor.

16.5.2.53. set_setting

Usage

```
set_setting <name> <value>
```

Options

- *<name>*: String with the name of the setting.
- *<value>*: String with the value of the setting.

Description

Sets the value for the specified BSP setting. Legal values for boolean settings are true, false, 1, and 0. Use the keyword none instead of an empty string to set a string to an empty value.

Example

```
set_setting hal.enable_reduced_device_drivers true
```

16.5.2.54. update_memory_region

Usage

```
update_memory_region <name> <slave_desc> <offset> <span>
```

Options

- <*name*>: String with the name of the memory region to update.
- <*slave_desc*>: String with the slave descriptor of the memory device for this region.
- <*offset*>: String with the byte offset of the memory region from the memory device base address.
- <*span*>: String with the span of the memory region in bytes.

Description

Updates an existing memory region for the linker script. This memory region must not overlap with any other memory region and must be within the memory range of the associated slave descriptor. The offset and span are decimal numbers unless prefixed with 0x.

Example

```
update_memory_region onchip_ram0 onchip_ram0 0 0x100000
```

16.5.2.55. update_section_mapping

Usage

```
update_section_mapping <section_name> <memory_region_name>
```

Options

- <*section_name*>: String with the name of the linker section.
- <*memory_region_name*>: String with the name of the memory region to map.

Description

Updates the specified linker section. The linker creates the section mappings in the order in which they appear in the linker script.

Example

```
update_section_mapping .text onchip_ram0
```

16.5.2.56. add_default_memory_regions

Usage

```
add_default_memory_regions
```

Description

Defaults the BSP to use default linker script memory regions. The default linker script memory regions are the best guess for memory regions based on the reset address and exception address of the processor associated with the BSP, and all other processors in the system that share memories with the processor associated with the BSP.

16.5.2.57. create_bsp

Usage

```
create_bsp <bspType> <bsp version> <processor name> <sopcinfo>
```

Options

- <*bspType*>: Type of BSP to create.
- <*bsp version*>: Version of BSP software element to utilize.
- <*processor name*>: Name of processor instance for BSP
- <*sopcinfo*>: .sopcinfo generated file that describes the system the BSP is for.

Description

Creates a new BSP.

16.5.2.58. generate_bsp

Usage

generate_bsp <*bspDir*>

Options

- <*bspDir*>: BSP directory to generate files to.

Description

Generates a new BSP.

16.5.2.59. get_available_bsp_type_versions

Usage

```
get_available_bsp_type_versions <bsp_type> <sopcinfo_path>
```

Options

- <*bsp_type*>: BSP type identifier (e.g. hal, ucosii, and freertos).
- <*sopcinfo_path*>: SOPC Information File path. Its parent folder might include custom BSP IP software components (`*_sw.tcl`).

Description

Gets the available BSP type versions.

16.5.2.60. get_available_bsp_types

Usage

```
get_available_bsp_types <sopcinfo_path>
```

Options

- <*sopcinfo_path*>: SOPC Information File path. Its parent folder might include custom BSP IP software components (`*_sw.tcl`).

Description

Gets the available BSP type identifiers.

16.5.2.61. get_available_cpu_architectures

Usage

```
get_available_cpu_architectures
```

Description

Gets the available processor architectures.

16.5.2.62. get_available_cpu_names

Usage

```
get_available_cpu_names <sopcinfo_path>
```

Options

- <*sopcinfo_path*>: SOPC Information File path that contains processor instances

Description

Gets the processor names given a SOPC system.

16.5.2.63. get_available_software

Usage

```
get_available_software <bsp_type> <filter> <sopcinfo_path>
```

Options

- <*bsp_type*>: BSP type identifier (e.g. hal, ucosii, and freertos).
- <*sopcinfo_path*>: SOPC Information File path. Its parent folder might include custom BSP IP software components (`*_sw.tcl`).
- <*filter*>: A filter can be applied to restrict results. The following filters are available:
  - `all`
  - `drivers`
  - `sw_packages`
  - `os_elements`

    Note: Comma-separated tokens are acceptable.

Description

Gets the available software (drivers, software packages, and bsp components) for a given BSP type.

16.5.2.64. get_available_software_setting_properties

Usage

```
get_available_software_setting_properties <setting_name>
<software_name> <software_version> <sopcinfo_path>
```

Options

- <*software_name*>: Name of a software component (for example, "altera_avalon_uart_driver", or "hal").
- <*software_version*>: Enter "default" for latest version, or a specific version number.
- <*setting_name*>: Name of a selected software component setting to get properties for(e.g. hal.linker.allow_code_at_reset).
- <*sopcinfo_path*>: SOPC Information File path. Its parent folder might include custom BSP IP software components (*_sw.tcl).

Description

Gets the available setting names for a software component.

16.5.2.65. get_available_software_settings

Usage

get_available_software_settings <*software_name*> <*software_version*> <*sopcinfo_path*>

Options

- <*software_name*>: Name of a software component (e.g. altera_avalon_uart_driver).
- <*software_version*>: Enter "default" for latest version, or a specific version number.
- <*sopcinfo_path*>: SOPC Information File path. Its parent folder can include custom BSP IP software components (*_sw.tcl).

Description

Gets the available setting names for a software component.

16.5.2.66. get_bsp_version

Usage

```
get_bsp_version
```

Description

Gets the version of the BSP operating system software element.

16.5.2.67. get_cpu_architecture

Usage

```
get_cpu_architecture <processor_name> <sopcinfo_path>
```

Options

- <*processor_name*>: processor instance name
- <*sopcinfo_path*>: SOPC Information File path that contains processor_name instance

Description

Gets the processor architecture (e.g. niosv) of a specified processor instance given a SOPC system.

16.5.2.68. get_sopcinfo_file

Usage

```
get_sopcinfo_file
```

Description

Returns the path of the BSP specific SOPC Information File.

16.5.2.69. get_supported_bsp_types

Usage

```
get_supported_bsp_types <processor_name> <sopcinfo_path>
```

Options

- <*processor_name*>: processor instance name
- <*sopcinfo_path*>: SOPC Information File path. Its parent folder can include custom BSP IP software components (`*_sw.tcl`).

Description

Gets the BSP types supported for a given processor and SOPC system.

16.5.2.70. is_bsp_hal_extension

Usage

```
is_bsp_hal_extension
```

Description

Returns a boolean value that indicates whether the BSP instantiated is of a type based on Intel HAL.

16.5.2.71. open_bsp

Usage

```
open_bsp <settingsFile>
```

Options

- <*settingsFile*>: .bsp settings file to open.

Description

Opens an existing BSP.

16.5.2.72. save_bsp

Usage

```
save_bsp <settingsFile>
```

Options

- <*settingsFile*>: .bsp settings file to save BSP to.

Description

Saves a new BSP.

16.5.2.73. set_bsp_version

Usage

```
set_bsp_version <version>
```

Options

- <*version*>: Version of BSP type software element to use.

Description

Sets the version of the BSP operating system software element to a specific value. The value `"default"` uses the latest version available. If this call is not used, the BSP is created using the 'default' BSP software element version.

16.5.2.74. set_logging_mode

Usage

```
set_logging_mode <mode>
```

Options

- <*mode*>: Logging Mode: 'silent', 'default', 'verbose', 'debug'

Description

Sets the verbosity level of the logger. Useful to eliminate tool informative messages

16.5.3. Tcl Commands for BSP Generation Callbacks

If you are defining a device driver or a software package, you can define Tcl callback functions to run whenever a BSP is generated containing your driver or package. Tcl callback functions enable you to create settings dynamically for the driver or package. This capability is essential when the driver or package settings must be customized to the hardware configuration, or to other BSP settings.

Tcl callback scripts are defined and controlled from the `*_sw.tcl` file associated with the driver or package. In `*_sw.tcl`, you can specify where the Tcl functions come from, when function runs, and the scope of each Tcl function's operation.

When the BSP is generated with your driver or software package, the settings you define in the callback scripts are inserted in `settings.bsp`.

You specify the source of the callback functions with the `set_sw_property` command, using the `callback_source_file` property.

A Tcl callback function can run at one of the following times:

- BSP initialization
- BSP generation
- BSP validation

Note: Although you can specify a new setting's value when you create the setting at BSP initialization, the setting's value can change between initialization and generation. For example, the BSP developer might edit the setting in the BSP Editor.

A Tcl callback can function in either of the following scopes:

- Component class
- Component instance

You specify each callback function's runtime environment by using the appropriate property in the `set_sw_property` command, as shown in the table below.

Table 64. Callback Properties

Property as specified in set_sw_property	Run time	Scope	Callback Arguments
`initialization_callback`	Initialization	Component instance	Component instance name
`validation_callback`	Validation	Component instance	Component instance name
`generation_callback`	Generation	Component instance	• Component instance name • BSP generate target directory • Driver BSP subdirectory(4)

continued...

(4) The BSP subdirectory into which the driver or package files are copied

Property as specified in set_sw_property	Run time	Scope	Callback Arguments
`class_initialization_callback`	Initialization	Component class	Driver class name
`class_validation_callback`	Validation	Component class	Driver class name
`class_generation_callback`	Generation	Component class	• Driver class name • BSP generate target directory • Driver BSP subdirectory(1)

Tcl callbacks have access to a specialized set of commands, described in this section. In addition, Tcl callbacks can use any read-only BSP setting Tcl command.

Note: For more information about BSP setting Tcl commands, refer to the Tcl Commands for BSP Settings chapter. When a Tcl callback creates a setting, it can specify the value. However, callbacks cannot change the value of a pre-existing setting.

16.5.3.1. add_class_sw_setting

Usage

`add_class_sw_setting <setting-name> <setting-type>`

Options

- <*setting-name*>: Name of the setting to persist in the BSP settings file. This is prepended with the driver class name associated with this callback script
- <*setting-type*>: Type of the setting to persist in the BSP settings file.

Description

Creates a BSP setting that is associated with a particular software driver element class. The `set_class_sw_setting_property` command is required to set the values for fields pertaining to a BSP software setting definition. This command is only valid for a callback script. A callback script is set in the driver's `*_sw.tcl` file, using the command `set_sw_property callback_source_file <filename>`.

Example

```
add_class_sw_setting MY_FAVORITE_SETTING String
```

16.5.3.2. add_class_systemh_line

Usage

`add_class_systemh_line <macro-name> <macro-value>`

Options

- <*macro-name*>: Macro to be added to the system.h file for the generated BSP
- <*macro-value*>: Value associated with the macro-name to be added to the system.h file for the generated BSP

Description

This adds a system.h assignment or macro during a driver callback execution. The BSP typically uses this during the generate phase depending on the generator. This command is only valid for a callback script. A callback script is set in the driver's `*_sw.tcl` file, using the command `set_sw_property callback_source_file <filename>`.

Example

```
add_class_systemh_line MY_MACRO "Macro_Value";
```

16.5.3.3. add_module_sw_property

Usage

`add_module_sw_property <property-name> <property-value>`

Options

- *<property-name>*: Name of the property to add to the BSP for a module instance
- *<property-value>*: Value of the property to add to the BSP for a module instance

Description

This adds a software property to the BSP driver of this module instance. The BSP typically uses this during the generate phase depending on the generator. This command is only valid for a callback script. A callback script is set in the driver's `*_sw.tcl` file, using the command `set_sw_property callback_source_file <filename>`.

Example

add_module_sw_setting MY_FAVORITE_SETTING String

16.5.3.4. add_module_sw_setting

Usage

`add_module_sw_setting <setting-name> <setting-type>`

Options

- *<setting-name>*: Name of the setting to persist in the BSP settings file. This is prepended with the module name associated with this callback script
- *<setting-type>*: Type of the setting to persist in the BSP settings file.

Description

Creates a BSP setting that is associated with a particular instance of hardware module in a SOPC system. The `set_module_sw_setting_property` command is required to set the values for fields pertaining to a BSP software setting definition. This command is only valid for a callback script. A callback script is set in the driver's `*_sw.tcl` file, using the command `set_sw_property callback_source_file <filename>`.

Example

```
add_module_sw_setting MY_FAVORITE_SETTING String
```

16.5.3.5. add_module_systemh_line

Usage

```
add_module_systemh_line <macro-name> <macro-value>
```

Options

- <*macro-name*>: Macro to be added to the `system.h` file for the generated BSP
- <*macro-value*>: Value associated with the macro-name to be added to the `system.h` file for the generated BSP

Description

This adds a system.h assignment or macro during a driver callback execution. The BSP typically uses this during the generate phase depending on the generator. This command is only valid for a callback script. A callback script is set in the driver's `*_sw.tcl` file, using the command `set_sw_property callback_source_file <filename>`.

Example

```
add_module_systemh_line MY_MACRO "Macro_Value";
```

16.5.3.6. add_systemh_line

Usage

```
add_systemh_line <sw> <name> <value
```

- <*sw*>: The software (OS) that the `system.h` text is associated with
- <*name*>: Name of macro to write into `system.h` (left-hand side of `#define`)
- <*value*>: Name of value to assign to macro in `system.h` (right-hand side of `#define`)

Description

Adds a line of text to the `system.h` file. The <sw> argument is the name of the software type (typically an operating system name) that the `system.h` text applies to. In the context of an operating system Tcl script, the name in the `create_os` <*name*> command must be used. The text is a name-value pair that creates a macro (`#define` statement) in the `system.h` file.

Note: This command can only be used by Tcl scripts that are registered to run at BSP generation time by an operating system.

Example

```
add_systemh_line UCOSII OS_TICKS_PER_SEC 100
```

16.5.3.7. get_class_peripheral

Usage

```
get_class_peripheral <instance-name> <irq-number>
```

Options

- <*instance-name*>: Name of EIC module instance to find connected peripheral for.
- <*irq-number*>: IRQ number to locate connected peripheral device

Description

This command is used on an EIC instance callback to obtain a peripheral slave descriptor connected to a specific IRQ port number. This command is only valid for a callback script.

Example

```
get_class_peripheral eic_1 $irq_2;
```

16.5.3.8. get_module_assignment

Usage

```
get_module_assignment <assignment-name>
```

Options

- <*assignment-name*>: Name of the module assignment to retrieve the value for, as defined for the module instance in the `.sopcinfo` file

Description

Given a module assignment key, return the assignment value of a module associated with the callback script using this command. The callback script must be set in the `*_sw.tcl` file using the following command:
`set_sw_property callback_source_file <filename>`

Example

```
puts [get_module_assignment
embeddedsw.configuration.isMemoryDevice]
```

Returns:
`true`

16.5.3.9. get_module_name

Usage

```
get_module_name
```

Options

None

Description

Returns the name of the module associated with the callback script using this command. The callback script must be set in the `*_sw.tcl` file using the following command:
`set_sw_property callback_source_file <filename>`

Example

```
puts [get_module_name]
```

Returns:
`jtag_uart`

16.5.3.10. get_module_peripheral

Usage

get_module_peripheral *<irq-number>*

Options

- *<irq-number>*: IRQ number to locate connected peripheral device

Description

This command is used on an EIC instance callback to obtain a peripheral slave descriptor connected to a specific IRQ port number. This command is only valid for a callback script.

Example

```
get_module_peripheral 2;
```

16.5.3.11. get_module_sw_setting_value

Usage

`get_module_sw_setting_value <setting-name>`

Options

- *<setting-name>*: Name of the module software setting to retrieve the value for, as defined by the `add_module_sw_setting` command.

Description

Given a module software setting name, return the setting value. The callback script using this command must be set in the `*_sw.tcl` file using the following command:
`set_sw_property callback_source_file <filename>`

You can use this command in a generation or validation callback to retrieve the current value of a setting created in an initialization callback.

Example

```
puts [get_module_sw_setting_value MY_SETTING]
```

Returns:
"My setting value"

16.5.3.12. get_peripheral_property

Usage

`get_peripheral_property <slave-descriptor> <property-name>`

Options

- <*slave-descriptor*>: Slave descriptor of a connected peripheral device
- <*property-name*>: Property name to query from the connected peripheral device

Description

This command is used on an EIC instance callback to obtain a connected peripheral property value. This command is only valid for a callback script. A callback script is set in the driver's `*_sw.tcl` file, using the command `set_sw_property callback_source_file <filename>`.

Example

```
get_peripheral_property jtag_uart supports_preemption;
```

16.5.3.13. remove_class_systemh_line

Usage

`remove_class_systemh_line <macro-name>`

Options

- <*macro-name*>: Macro to be removed to the system.h file for the generated BSP

Description

This removes a system.h assignment or macro during a driver callback execution. The BSP typically uses this during the generate phase depending on the generator. This command is only valid for a callback script. A callback script is set in the driver's `*_sw.tcl` file, using the command `set_sw_property callback_source_file <filename>`.

Example

```
remove_class_systemh_line MY_MACRO;
```

16.5.3.14. remove_module_systemh_line

Usage

`remove_module_systemh_line <macro-name>`

Options

- <*macro-name*>: Macro to be removed to the system.h file for the generated BSP

Description

This removes a system.h assignment or macro during a driver callback execution. The BSP typically uses this during the generate phase depending on the generator. This command is only valid for a callback script. A callback script is set in the driver's `*_sw.tcl` file, using the command `set_sw_property callback_source_file <filename>`.

Example

```
remove_module_systemh_line MY_MACRO;
```

16.5.3.15. set_class_sw_setting_property

Usage

`set_class_sw_setting_property <setting-name> <property> <value>`

Options

- `<setting-name>`: Name of the setting to persist in the BSP settings file associated with the driver class of this callback script
- `<property>`: Name of the setting property to update
- `<value>`: Value of the setting property to update

Description

Update a driver class software setting property. The setting must be added using the `add_class_sw_setting` command before calling this method. This command is only valid for a callback script. A callback script is set in the driver's `*_sw.tcl` file, using the command `set_sw_property callback_source_file <filename>`.

You can set the following setting properties:

- `destination`
- `identifier`
- `value`
- `default_value`
- `description`
- `restrictions`
- `group`

Example

```
set_class_sw_setting_property MY_FAVORITE_SETTING default-value
'42'
```

16.5.3.16. set_module_sw_setting_property

Usage

`set_module_sw_setting_property <setting-name> <property> <value>`

Options

- <*setting-name*>: Name of the setting to persist in the BSP settings file associated with the SOPC module of this callback script
- <property>: Name of the setting property to update
- <value>: Value of the setting property to update

Description

Update a module's software setting property. The setting must be added using the `add_module_sw_setting` command before calling this method. This command is only valid for a callback script. A callback script is set in the driver's `*_sw.tcl` file, using the command `set_sw_property callback_source_file <filename>`.

You can set the following setting properties:

- `destination`
- `identifier`
- `value`
- `default_value`
- `description`
- `restrictions`
- `group`

Example

```
set_module_sw_setting_property MY_FAVORITE_SETTING default-value
'42'
```

16.5.4. Tcl Commands for Drivers and Packages

This section describes the tools that you use to specify and manipulate the settings and characteristics of a custom software package or driver. Typically, when creating a custom software package or device driver, or importing a package or driver from another development environment, you need these more powerful tools.

A device driver and a software package are both collections of source files added to the BSP. A device driver is associated with a particular component class (for example, `altera_avalon_jtag_uart`). A software package is not associated with any particular component class, but implements a functionality such as Safe C Library.

To define a device driver or software package, you create a Tcl script defining its characteristics. This section describes the Tcl commands available to define device drivers and software packages.

The following commands are available for device driver and software package creation:

- add_sw_property
- add_sw_setting
- add_sw_setting2
- create_driver

- create_os
- create_sw_package
- set_sw_property
- set_sw_setting_property

Related Information

- Tcl Scripts for BSP Settings on page 68
- Settings Managed by Nios V Processor Board Support Package Editor on page 296
- Tcl Commands for BSP Settings on page 333

16.5.4.1. add_sw_property

Usage

`add_sw_property <property> <value>`

Options

- <*property*>: Name of property.
- <*value*>: Value assigned, or appended to the current value.

Description

This command defines a property for a device driver or software package. A property is a list of values (for example, a list of file names). The `add_sw_property` command defines a property if it is not already defined. The command appends a new value to the list of values if the property is already defined.

In the case of a property consisting of a file name or directory name, use a relative path. Specify the path relative to the directory containing the Tcl script.

This command supports the following properties:

- `asm_source`—Adds a RISC-V assembly language source file (`.s` or .S) to BSPs containing your package. **niosv-bsp** copies assembly source files into a BSP and adds them to the source build list in the BSP `CMakeLists.txt`. This property is optional.
- `c_source`—Adds a C source file (`.c`) to BSPs containing your package. **niosv-bsp** copies C source files into a BSP and adds them to the source build list in the BSP `CMakeLists.txt`. This property is optional.
- `cpp_source`—Adds a C++ source file (`.cpp`, `.cc`, or `.cxx`) to BSPs containing your package. **niosv-bsp** copies the C++ source files into a BSP and adds them to the source build list in the BSP `CMakeLists.txt`. This property is optional.
- `include_source`—Adds an include file (typically .h) to BSPs containing your package. **niosv-bsp** copies include files into a BSP, but does not add them to the generated `CMakeLists.txt`. This property is optional.
- `include_directory`—`add_sw_property` adds the path to the generated public `CMakeLists.txt` shared by the BSP and applications or libraries referencing it. `add_sw_property` compiles all files with the include directory listed in the compiler arguments. This property is optional.

- `lib_source`—Adds a precompiled library file (typically `.a`) to each BSP containing the driver or package. **niosv-bsp** copies the precompiled library file into the BSP directory and adds both the library file name and the path (required to locate the library file) into to the BSP's `CMakeLists.txt` file. Applications using the BSP link with the library file. The library file name must conform to the following pattern: `lib<`*name*`>.a` where <*name*> is a nonempty string. Example: `add_sw_property lib_source HAL/lib/libcomponent.a` This property is optional.
- `specific_compatible_hw_version`—Specifies that the device driver only supports the specified component hardware version. See the `version` property of the `set_sw_property` command for information about version strings. This property applies only to device drivers (see the `create_driver` command), not to software packages. If your driver supports all versions of a peripheral after a specific release, use the `set_property min_compatible_hw_version` command instead. This property is optional. This property is only available for device drivers.
- `supported_bsp_type`—Adds a specific BSP type (operating system) to the list of supported operating systems that the driver or software package supports. Specify HAL if the software supports the Intel HAL, or operating systems that extend it. If your software is operating system-neutral and works on multiple HAL-based operating systems, state HAL only. If your software or driver contains code that depends on a particular operating system, state compatibility with that operating system only, but not HAL.

  The name of another operating system to support must match the name of the operating system exactly. This operating system name string is the same as that used to create a BSP with the `niosv-bsp` command, as well as in the `.tcl` script that describes the operating system, in its `create_os` command.
  When you create a BSP with an operating system that extends HAL, such as `UCOSII`, and the BSP tools select a driver for a particular hardware module, precedence is given to drivers which state compatibility with a that specific operating system (OS) before a more generic driver stating HAL compatibility. This property is only available for device drivers and software packages. This property must be set to at least one operating system.
- `alt_cppflags_addition`—Adds a line of arbitrary text to the `ALT_CPPFLAGS` variable in the BSP `toolchain.cmake` file. This technique can be useful if you wish to have a static compilation flag or definition that all BSP, application, and library files receive during software build. This property is optional.
- `excluded_hal_source`—Specifies a file to exclude from the a BSP generated with an operating system that extends HAL. The value is the path to a BSP file to exclude, with respect to the BSP root. This property is optional.
- `systemh_generation_script`—Specifies a .**tcl** script to execute during generation of the BSP `system.h` file. This script runs with the tcl commands available to other BSP settings tcl scripts, and allow you to influence the contents of the `system.h` file. This property is available only to operating systems, created with the `create_os` command. This property is optional.
- `txt_source`--Adds a text file (.txt) to BSPs. Example: `add_sw_property txt_source files.txt`.

16.5.4.2. add_sw_setting

Usage

```
add_sw_setting <type> <destination> <displayName> <identifier>
<value> <description>
```

Options

- <*type*>: Setting type - Boolean, QuotedString, UnquotedString.
- <*destination*>: The destination BSP file associated with the setting, or the module generator that processes this setting.
- <*displayName*>: Setting name.
- <*identifier*>: Name of the macro created for a generated destination file.
- <*value*>: Default value of the setting.
- <*description*>: Setting description.

Description

This command creates a BSP setting associated with a software package or device driver. The setting is available whenever the software package or device driver is present in the BSP. **niosv-bsp** converts the setting and its value into either a C preprocessor macro or BSP makefile variable. `add_sw_setting` passes macro definitions to the compiler using the `-D` command line option, or adds them to the `system.h` file as `#define` statements.

The setting only exists once even if there are multiple instances of a software package. You can use the BSP Tcl commands `set_setting` and `get_setting` to set or get the setting. The value of the setting persists in the BSP settings file.

To create a setting, you must define each of the following parameters.

- `type`—This parameter formats the setting value during BSP generation. The following supported types and usage restrictions apply:
  - `boolean_define_only`—Defines a macro if the setting's value is 1 or true. Example: `#define LCD_PRESENT`. No macro is defined if the setting's value is 0 or false. This setting type supports the `system_h_define` and `public_mk_define` destinations, defined below.
  - `boolean`—Defines a macro or makefile variable and sets it to 1 (if the value is 1 or true) or 0 (if the value is 0 or false). Example: `#define LCD_PRESENT 1`. This type supports all destinations.
  - `character`—Defines a macro as a single character with single quotes around the character. Example: `#define DELIMITER ':'`. This type supports the `system_h_define` destination, defined below.
  - `decimal_number`—Defines a macro or makefile variable and sets it with an unquoted decimal (integer) number. Example: `#define NUM_COPROCESSORS 3`. This type supports all destinations.
  - `double`—Defines a macro name and sets it to a value with a decimal point. Example: `#define PI 3.1416`. This type supports the `system_h_define` destination, defined below.
  - `float`—Defines a macro name and sets it to a value with a decimal point and `f` character. Example: `#define PI 3.1416f`. This type supports the `system_h_define` destination, defined below.
  - `hex_number`—Defines a macro name and sets it to a value with a `0x` prepended to the value. Example: `#define LCD_SIZE 0x1000`. This type supports the `system_h_define` destination, defined below.
  - `quoted_string`—Quoted strings always have the macro name and setting value added to the destination files. In the destination, the setting value is enclosed in quotation marks. Example: `#define DFLT_ERR "General error"`
    If the setting value contains white space, you must place quotation marks around the value string in the Tcl script. This type supports the `system_h_define` destination, defined below.
  - `unquoted_string`—Unquoted strings define a macro or makefile variable with setting name and value in the destination file. In the destination file, the setting value is not enclosed in quotation marks. Example: `#define DFLT_ERROR Error` This type supports all destinations.

- `destination`—The destination parameter specifies where `add_sw_setting` puts the setting in the generated BSP. `add_sw_settings` supports the following destinations:
  - `system_h_define`—With this destination, `add_sw_settings` formats settings as `#define <setting name> [<setting value>]` macros in the `system.h` file
  - `public_mk_define`—With this destination, `add_sw_settings` formats settings as `-D<setting name>[=<setting value>]` additions to the ALT_CPPFLAGS variable in the BSP `toolchain.cmake` file. `toolchain.cmake` passes the flag to the C preprocessor for each source file in the BSP, and in applications and libraries using the BSP.
  - `makefile_variable`—With this destination, `add_sw_settings` formats settings as makefile variable additions to the BSP `toolchain.cmake` file. The variable name must be unique in the `toolchain.cmake` file.
- `displayName`—The name of the setting. Settings exist in a hierarchical namespace. A period separates levels of the hierarchy. Settings created in your Tcl script are located in the hierarchy under the driver or software package name you specified in the `create_driver` or `create_sw_package` command. Example: `my_driver.my_setting`. The Nios V processor BSP Editor adds the hierarchical prefix to the setting name.
- `identifier`—The name of the macro or makefile variable being defined. In a setting added to the `system.h` file at generation time, this parameter corresponds to the text immediately following the `#define` statement.
- `value`—The default value associated with the setting. If you do not assign a value to the option, its value is this default value. Valid initial values are TRUE, `1`, FALSE, and `0` for `boolean` and `boolean_define_only` setting types, a single character for the `character` type, integer numbers for the `decimal_number` setting type, integer numbers with or without a `0x` prefix for the `hex_number` type, numbers with decimals for `float_number` and `double_number` types, or an arbitrary string of text for quoted and unquoted string setting types. For string types, if the value contains any white space, you must enclose it in quotation marks.
- `description`—Descriptive text that is inserted along with the setting value and name in the `summary.html` file. You must enclose the description in quotation marks if it contains any spaces. If the description includes any special characters (such as quotation marks), you must escape them with the backslash (\) character. The description field is mandatory, but can be an empty string (`""`).

16.5.4.3. add_sw_setting2

Usage

`add_sw_setting2 <setting-name> <setting-type>`

Options

- *<setting-name>*: Name of the setting to persist in the BSP settings file. This is prepended with BSP Type when persisted.
- *<setting-type>*: Type of the setting to persist in the BSP settings file.

Description

This command creates a BSP setting that is associated with a particular software component. The 'set_sw_setting_property' command is required to set the values for fields pertaining to a BSP software setting definition.

Example

```
add_sw_setting2 MY_FAVORITE_SETTING unquoted_string
```

16.5.4.4. create_driver

Usage

```
create_driver <name>
```

Options

- <*name*>: Name of device driver.

Description

This command creates a new device driver instance available for the Nios V processor BSP Editor. This command must precede all others that describe the device driver in its Tcl script. You can only have one `create_driver` command in each Tcl script. If the `create_driver` command appears in the Tcl script, the `create_sw_package` and `create_os` commands cannot appear.

The name argument is usually distinct from all other device drivers and software packages that the Nios V processor tools might locate. You can specify driver name identical to another driver if the driver you are describing has a unique version number assignment.

If your driver differs for different operating systems, you need to provide a unique name for each BSP type.

This command is required, unless you use the `create_sw_package` or `create_os` commands, as appropriate.

16.5.4.5. create_os

Usage

```
create_os <name>
```

Options

- <*name*>: Name of operating system (BSP type).

Description

This command creates a new operating system (OS) instance (also known as a BSP type) available for the Nios V processor BSP tools. This command must precede all others that describe the OS in its Tcl script. You can only have one `create_os` command in each Tcl script. If the `create_os` command appears in the Tcl script, the `create_driver` or `create_sw_package` commands cannot appear.

The name argument is usually distinct from all other operating systems that the Nios V processor tools might locate. You can specify an OS name identical to OS if the OS you are describing has a unique version number assignment.

This command is required, unless you use the `create_driver` or `create_sw_package` commands, as appropriate.

16.5.4.6. create_sw_package

Usage

`create_sw_package <name>`

Options

- *<name>*: Name of the software package.

Description

This command creates a new software package instance available for the Nios V processor BSP Editor. This command must precede all others that describe the software package in its Tcl script. You can only have one `create_sw_package` command in each Tcl script. If the `create_sw_package` command appears in the Tcl script, the `create_driver` or `create_os` commands cannot appear.

The name argument is usually distinct from all other device drivers and software packages that the Nios V processor tools might locate. You can specify a name identical to another software package if the software package you are describing has a unique version number assignment.

If your software package differs for different operating systems, you need to provide a unique name for each BSP type.

This command is required, unless you use the `create_driver` or `create_os` commands, as appropriate.

16.5.4.7. set_sw_property

Usage

`set_sw_property <property> <value>`

Options

- *<property>*: Type of software property being set.
- *<value>*: Value assigned to the property.

Description

Sets the specified value to the specified property. The properties this command supports can only hold a single value. This command overwrites the existing (or default) contents of a particular property with the specified value. This command applies to device drivers and software packages.

This command supports the following properties:

- `hw_class_name`—The name of the hardware class which your device driver supports. The hardware class name is also the **Component Name** shown in the Component Editor. Example: `altera_avalon_uart`. This property is only available for device drivers. This property is required for all drivers.
- `version`—The version number of this package. `set_sw_property` uses version numbers to determine compatibility between hardware (peripherals) and their software (drivers), as well as to choose the most recent software or driver if multiple compatible versions are available. A version can be any alphanumeric string, but is usually a major and one or more minor revision integers. The dot (.) character separates major and minor revision numbers. Examples: `9.0`, `5.0sp1`, `3.2.11`. This property is optional, but recommended. If you do not specify a version, the newest version of the package is used.
- `min_compatible_hw_version`—Specifies that the device driver supports the specified hardware version, or all greater versions. This property is only available for device drivers. If your device driver supports only one or more specific versions of a hardware class, use the `add_sw_property specific_compatible_hw_version` command instead. See the `version` property documentation for information about version strings. This property is optional. This property is only available for device drivers.
- `auto_initialize`—Boolean value that specifies `alt_sys_init.c` needs to initialize your package. If enabled, you must provide a header file containing `INSTANCE` and `INIT` macros.

  This property is optional; if unspecified, `alt_sys_init.c` does not contain references to your driver or software. This property is only available for device drivers and software packages.
- `bsp_subdirectory`—Specifies the top-level directory where **niosv-bsp** copies all source files for this package. This property is a path relative to the top-level BSP directory. This property is optional; if unspecified, **niosv-bsp** copies the driver or software package into the **drivers** subdirectory of any BSP including this software.
- `alt_sys_init_priority`—This property assigns a priority to the software package or device driver. The value of this property must be a positive integer. Use this property to customize the order of macro calls in the BSP `alt_sys_init.c` file. Specifying the priority is useful if your software or driver must be initialized before or after other software in the system. For example, your driver might depend on another driver already being initialized.
  This property is optional. The default priority is `1000`. This property is only available for device drivers and software packages.
- `display_name`—This property is used for user interfaces and other tools that wish to show a human-readable name to identify the software being described in the .**tcl** script. `display_name` is set to a few words of text (in quotes) that name your software. For example: `Intel FPGA Nios V driver`.
  This property is optional. If not set, tools that attempt to use the display name use the package name created with the appropriate `create_` command.

- `extends_bsp_type`—This property specifies which BSP type that an operating system (created with the `create_os` command) extends (if any). Currently, only the Intel FPGA HAL (HAL) is supported. This command is required for all operating systems that wish to use HAL-compatible generators in the Nios V processor BSP tools. It is also required for operating systems that require the Intel HAL, device driver, or software package source files that are HAL compatible in BSPs created with that operating system. An operating system that extends HAL is presumed to be compatible with device drivers that support HAL. This command is only available for operating systems.
- `callback_source_file`—This property specifies a Tcl source file containing callback functions.
- `initialization_callback`—This property specifies the name of a Tcl callback function which is intended to run in the following environment:
  - Run time: initialization
  - Scope: component instance
  - Function argument(s): component instance name
- `validation_callback`—This property specifies the name of a Tcl callback function which is intended to run in the following environment:
  - Run time: validation
  - Scope: component instance
  - Function argument(s): component instance name
- `generation_callback`—This property specifies the name of a callback function which is intended to run in the following environment:
  - Run time: generation
  - Scope: component instance
  - Function argument(s): component instance name, BSP generate target directory, driver BSP subdirectory
- `class_initialization_callback`—This property specifies the name of a callback function which is intended to run in the following environment:
  - Run time: initialization
  - Scope: component instance
  - Function argument(s): driver class name

- `class_validation_callback`—This property specifies the name of a callback function which is intended to run in the following environment:
  - Run time: validation
  - Scope: component instance
  - Function argument(s): driver class name
- `class_generation_callback`—This property specifies the name of a callback function which is intended to run in the following environment:
  - Run time: generation
  - Scope: component instance
  - Function argument(s): driver class name, BSP generate target directory, driver BSP subdirectory
- `supported_interrupt_apis`—Specifies the interrupt API that the device driver supports. Specify `enhanced_interrupt_api` only because the Nios V processor supports the Enhanced HAL Interrupt API only.

Note: This property is only available for device drivers.

- `isr_preemption_supported`—Specify true if your device driver ISR can be preempted by a higher priority ISR. If you do not specify whether ISR preemption is supported, the Nios V processor tools assumes that your device driver does not support preemption. If your driver does not have an ISR, but the associated device has an interrupt port, you can set this property to true.

Note: This property is valid for operating systems and device drivers.

16.5.4.8. set_sw_setting_property

Usage

`set_sw_setting_property <setting-name> <property-name> <property-value>`

Options

- *<setting-name>*: Name of the setting to persist in the BSP settings file.
- *<property-name>*: Name of the setting property to update.
- *<property-value>*: Value of the setting property to update.

Description

Update a software components setting property. The setting must already be added using `add_sw_setting2` command before calling this method.

Example

```
set_sw_setting_property MY_FAVORITE_SETTING default-value '42'
```

16.6. Nios V Processor Design Example

Note: If you are using Quartus Prime Standard Edition software, you can acquire the design examples through the Nios V Processor IP. To generate the example design, refer to the topic *Intel Quartus® Prime Software Support* in *Nios® V Embedded Processor Design Handbook*.

Related Information

Nios® V Embedded Processor Design Handbook : Nios® V Processor Device Support
For more information about invoking the BSP Editor, generating design example, and Nios® V processor device support in Quartus® Prime Standard.

16.6.1. Intel FPGA Design Store

You can download a Nios V processor design example from the Intel FPGA Design Store.

The design examples provide scripts that allow you to automate the design example generation, and build both a BSP project and a simple application project. The design examples are preset to an Intel FPGA device. If you are targeting a different Intel FPGA device, migrate the design example project to the specific device.

Follow these steps to acquire the Nios V processor example design:

1. Search for the Nios V processor-related example designs in the Intel FPGA Design Store.
2. Select any example design.
3. Download the package according to the Quartus Prime software version of your host machine.
4. Proceed to open the project PAR file.
5. Refer to the `readme.txt` for the how-to guide.

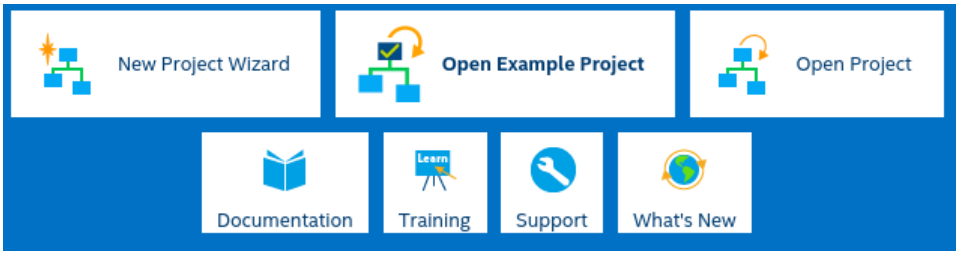
Related Information

- Nios V Processor Tools on page 18
- Altera FPGA Technical Training
  Online training demonstrations located on the Embedded Design Curriculum page of the Altera FPGA Technical Training website.
- Nios V Processor Embedded Design Handbook
- Getting Started with the Hardware Abstraction Layer on page 79
- FPGA Design Store

16.6.2. Quartus Prime Software

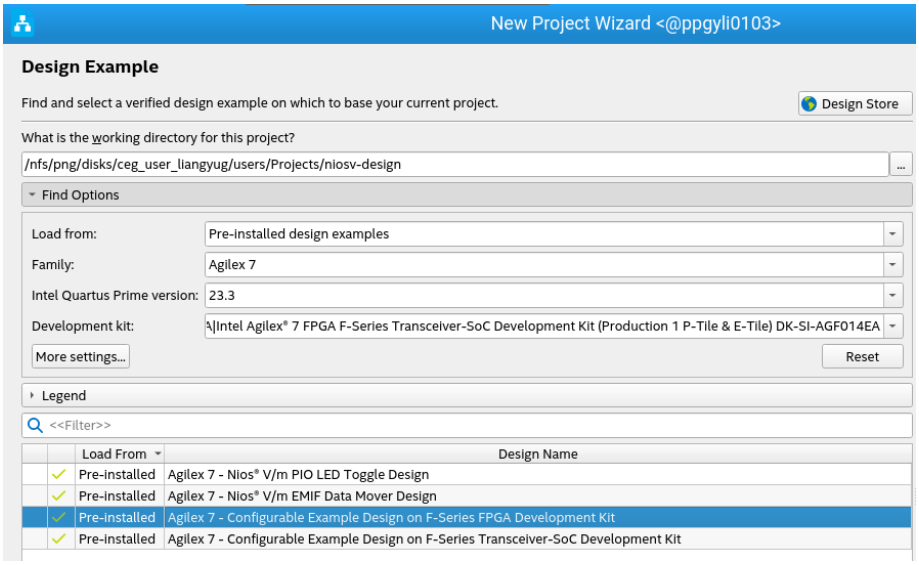
1. Open the **Quartus Prime** software.
2. Select **Open Example Project** from the homepage.

Figure 36. Open Example Project



3. In the **New Project Wizard (Design Example)**, fill up the following options:
   a. **What is the working directory for this project?**
   b. **Load from** – Online design examples.
   c. **Family** – Select the target Intel FPGA device.
   d. **Intel Quartus Prime version** – Select the correct software version.
   e. **Development kit** – Select the Intel FPGA development kit.
4. Select the design example. The info box at the right displays the basic details of the example design.
5. Click **Next** and accept the Software License Agreement. The example design is ready for customization.
6. Refer to the `readme.txt` in the project directory for the how-to guide.

Figure 37. New Project Wizard



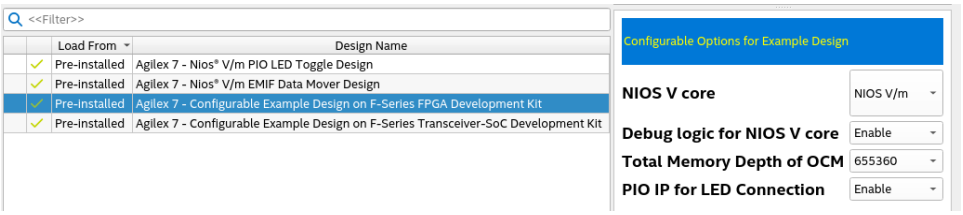
16.6.2.1. Configurable Example Design

The Configurable Example Design is an example design that supports pre-built system configurations and is pre-installed in the Quartus Prime software. Each Configurable Example Design supports different sets of configurable options.

The figure below shows the **Agilex 7 – Configurable Example Design on F-series FPGA Development Kit**. You can select any following preferred configurations:

- **NIOS V core**
- **Debug logic for NIOS V core**
- **Total Memory Depth of On-Chip Memory (OCM)**
- **PIO IP for LED Connection**

Figure 38. Configurable Options in Configurable Example Designs



16.6.3. Migrating to Other Intel FPGA Device

The following are the basic device migration steps:

1. New device assignment
   a. Go to **Assignments ➤ Device...**
   b. Select the appropriate Device family and Name.
      Note: If the SmartVID feature is supported, configure the correct SmartVID settings.
2. New pin assignment
   a. Go to **Assignments ➤ Assignments Editor**.
   b. Reassign the appropriate pin assignment to the board.
3. Recompile the design example.

16.7. Manual System PATH Variable Setup

The Nios V Command Shell provides a similar setup upon execution and reverts after exiting the Nios V Command Shell. Nios V Command Shell offers a temporary approach to setting up a host system for Nios V processor development.

Table 65. Recommended System PATH

PATH Variables	Related Executables
`$QUARTUS_ROOTDIR/../niosv/bin`	• niosv-download • niosv-shell • niosv-stack-report • niosv-app

continued...

PATH Variables	Related Executables
	• niosv-bsp • elf2hex • elf2flash
$QUARTUS_ROOTDIR/../riscfree/build_tools/bin	• make • sh (Windows-only) • rm (Windows-only) • mkdir (Windows-only) • echo (Windows-only) • busybox (Windows-only)
$QUARTUS_ROOTDIR/../riscfree/debugger/gdbserver-riscv	• ash-riscv-gdb-server
$QUARTUS_ROOTDIR/../riscfree/build_tools/cmake/bin	• cmake
$QUARTUS_ROOTDIR/../riscfree/toolchain/riscv32-unknown-elf/bin	• riscv32-unkown-elf GCC toolchain
In Windows: $QUARTUS_ROODIR/bin64 In Linux: $QUARTUS_ROODIR/linux64	• openocd • openocd-cfg-gen • juart-terminal • quartus_pgm
$QUARTUS_ROOTDIR/../qsys/bin	qsys-edit

17. Nios V Processor Software Developer Handbook Archives

For the latest and previous versions of this user guide, refer to Nios® V Processor Software Developer Handbook. If an IP or software version is not listed, the user guide for the previous IP or software version applies.

IP versions are the same as the Quartus Prime Design Suite software versions up to v19.1. From Quartus Prime Design Suite software version 19.2 or later, IP cores have a new IP versioning scheme.

© Altera Corporation. Altera, the Altera logo, the 'a' logo, and other Altera marks are trademarks of Altera Corporation. Altera and Intel warrant performance of its FPGA and semiconductor products to current specifications in accordance with Altera's or Intel's standard warranty as applicable, but reserves the right to make changes to any products and services at any time without notice. Altera and Intel assume no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to inwriting by Altera or Intel. Altera and Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

18. Revision History for the Nios V Processor Software Developer Handbook

Document Version	Quartus Prime Version	Changes
2025.04.07	25.1	• Added a new figure to the topic *Nios® V Processor Software Development Environment*. • Added the following topics to the chapter *Developing Programs Using the Hardware Abstraction Layer*: — *File System* — *Using File System* • Updated the following topics: — *Hardware Interrupt Controller*. — *CLINT-Direct Mode* — *CLINT-Vectored Mode* — *CLIC Mode* — *niosv-bsp* — *niosv-app*
2025.01.27	24.3.1	• Renamed section *Exception Handling* to *Trap Handling*. — Updated the mention of *trap* to *exception* in the following topics: • *Nios V Processor Trap Handling Overview* • *Trap Handling Terminology* — Updated the mention of *hardware interrupt* to *platform interrupt* throughout the document. — Added new sub-topics to the *Hardware Interrupt Controller*: • *CLINT-Direct Mode* • *CLINT-Vectored Mode* — Updated table *Nios V Processor Exception Cause Codes* with possible cause column. — Added new topic: *Direct or Vectored Mode*. — Updated the description for *OpenOCD*. • Updated the topic *Micrium MicroC/OS-II BSP* — Removed the following settings: • ucosii.miscellaneous.os_task_stat_en • ucosii.miscellaneous.os_event_name_size • ucosii.task.os_task_name_size • ucosii.event_flag.os_flag_name_size • ucosii.memory.os_mem_name_size • ucosii.timer.os_tmr_cfg_name_size — Added the following settings: • ucosii.miscellaneous.os_task_reg_tbl_size • ucosii.miscellaneous.os_event_name_en • ucosii.task.os_task_name_en • ucosii.event_flag.os_flag_name_en • ucosii.memory.os_mem_name_en • ucosii.timer.os_tmr_cfg_name_en

continued...

© Altera Corporation. Altera, the Altera logo, the 'a' logo, and other Altera marks are trademarks of Altera Corporation. Altera and Intel warrant performance of its FPGA and semiconductor products to current specifications in accordance with Altera's or Intel's standard warranty as applicable, but reserves the right to make changes to any products and services at any time without notice. Altera and Intel assume no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to inwriting by Altera or Intel. Altera and Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

Document Version	Quartus Prime Version	Changes
2024.10.12	24.3	• Updated table *Basic Options* to add *OpenOCD* option. • Updated topic *Nios V Command Shell* to add niosv-shell command prompt on Windows. • Updated table *Other Command Line Tools* to add *elf-insert* and *elf-query* tools. • Added new sub-topics to the topic *Implementing FreeRTOS Project for the Processor*: — *Integrating Device Drivers* — *Integrating Device Drivers without Initialization Manually* — *Integrating Device Drivers with Initialization Manually* • Added new sub-topics to the topic: *Other Command Line Utilities Tools Reference* — *elf-insert* — *elf-query*
2024.07.08	24.2	• Added the following topics: — *Using Ashling RiscFree IDE for Intel FPGA* — *Using BSP Editor* — *Using Quartus Programmer* — *Apply Different Compiler Flags within a Source File* — *Nios V Processor Timer Interrupt Service Routine*. • Updated the following topics: — *niosv-bsp* — *Nios V Command Shell* — *niosv-shell* — *niosv-bsp* — *juart-terminal* — *Device Drivers BSP*
2024.04.01	24.1	• Added the following topics: — *Hierarchical Platform Designer Systems*. — *Handling Hierarchical Systems*. — *Manual System PATH Variable Setup* • Updated the following topics: — *Device Driver Source Code*. — *altera_avalon_jtag_uart_regs.h Defining Macros*. • Added new chapter *Cache and Tightly-Coupled Memory*. • Updated *FreeRTOS Real-Time Operating System*. Added the following topics: — *Create FreeRTOS BSP Project* — *Configure FreeRTOS Features* — *Write FreeRTOS-based Source Codes* — *Generate Application CMake File* — *Build FreeRTOS BSP and Application Projects* • Updated *Read-Only Zip File System*. Updated the following topics: — *Read-Only Zip File System in BSP Editor* — *Active Serial Configuration Flash in SDM Based Devices* • Removed the following functions from *HAL API Reference*: — alt_uncached_malloc() — alt_uncached_free() — alt_remap_uncached() — alt_remap_cached()
2023.12.04	23.4	• Edited topic *elf2flash* to replace FLASH with SREC file. • Updated *Enabling MicroC/TCP-IP Protocol Stack*. • Added new section *Read-Only Zip File System*.

continued...

Document Version	Quartus Prime Version	Changes
2023.10.02	23.3	• Added topic *Apply Different Compiler Flags in BSP and APP*. • Edited description in the table *Options* in the topic *elf2hex*. • Added subtopics to *Processor Design Example*: — *Intel FPGA Design Store* — *Quartus Prime Software* — *Configurable Example Design* — *Migrating to Other Intel FPGA Device*
2023.07.20	23.2	• Edited the following topics: — *niosv-app* — *Apply Compiler Flags* — *Use Small Variant Device Drivers* — *Reduce File Descriptor Pool* — *Eliminate Unused Device Drivers* — *Eliminate Unneeded Exit Code* — *Intel HAL BSP* • Added the following topics: — *Use the Picolibc library* — *Eliminate Unused alt_load()*
2023.05.26	23.1	• Added links to *AN 980: Nios V Processor Quartus Prime Software Support*. • Updated the code for elf2hex in the topic: *File Format Conversion Tools Reference*.
2023.04.10	23.1	• Edited the following topics to add the mention of FreeRTOS: — *niosv-bsp* — *Selecting the Operating System* — *Nios V Processor Board Support Package Editor* — *Board Support Packages* — *BSP Files* — *Boot Configurations for Intel FPGA Embedded Software* — *Tcl Command Walkthrough for a Typical Driver or Software Package* — *Setting the BSP Type* — *Specifying a Subdirectory* — *Adding Include Paths* — *HAL APIs for Hardware Interrupts* — *Thread-Safe Drivers* — *Thread-Safe HAL Drivers* — *Importing Tcl Script to Create a New BSP* — *HAL Initialization* — *Configuring FreeRTOS* — *Optional Real-Time Operating System* • Edited the table *Intel-provided Bootloader* in the topic *Boot from Flash Configuration*. • Edited the topic *Using Timer Devices* and *Processor Appendix*. • Edited the topic *C Standard Library—newlib* to add Picolib. • Added a new section *FreeRTOS Real-Time Operating System*. • Rearranged the content in *BSP Settings Reference* to the following new topics: — *Intel HAL BSP* — *Micrium MicroC/OS-II BSP* — *FreeRTOS BSP* — *Device Drivers BSP*

continued...

Document Version	Quartus Prime Version	Changes
2023.02.14	22.4	• Removed the Table *Nios® V Processor Device Support* from the topic *Quartus Prime Software Support*. Added a link to refer the Table in *Nios® V Embedded Processor Design Handbook*. • Updated the steps in *Nios V® Processor Design Example* to acquire the Nios V processor example design in the Intel FPGA Design Store.
2022.10.31	22.1std	Added topic *Quartus Prime Software Support* for Quartus Prime Standard Edition software release.
2022.09.26	22.3	Initial release.