Nios II Classic Software Developer’s Handbook
Contents

Overview of Embedded Development................................................................. 1-1
  Installing Windows® Subsystem for Linux® (WSL) on Windows®...................... 1-1
  Prerequisites for Understanding the Embedded Design Suite.......................... 1-1
  Finding Nios II EDS Files................................................................................ 1-2
  Software Development Environment.............................................................. 1-2
  EDS Development Flows.................................................................................. 1-2
    SBT Development Flow................................................................................. 1-2
  Programs.......................................................................................................... 1-3
    Makefiles and the SBT.................................................................................... 1-3
    Software Project Types.................................................................................. 1-4
  Intel FPGA Software Packages for Embedded Systems................................... 1-5
  Embedded Design Examples............................................................................. 1-5
    Hardware Examples....................................................................................... 1-5
    Software Examples....................................................................................... 1-6
  Third-Party Embedded Tools Support............................................................. 1-6
  Additional Information.................................................................................... 1-6

Getting Started with the Graphical User Interface.............................................. 2-1
  Installing Eclipse IDE into EDS........................................................................ 2-1
  Getting Started with Nios II Software in Eclipse............................................ 2-2
    The Nios II SBT for Eclipse Workbench....................................................... 2-2
    Creating a Project........................................................................................ 2-3
    Navigating the Project.................................................................................. 2-4
    Building the Project.................................................................................... 2-5
    Configuring the FPGA................................................................................. 2-5
    Running the Project on Nios II Hardware.................................................... 2-5
    Debugging the Project on Nios II Hardware................................................ 2-6
    Creating a Simple BSP................................................................................. 2-11
  Makefiles and the SBT for Eclipse................................................................. 2-12
    Eclipse Source Management........................................................................ 2-12
    User Source Management............................................................................ 2-13
    BSP Source Management............................................................................ 2-14
  Using the BSP Editor.................................................................................... 2-14
    Tcl Scripting and the BSP Editor.................................................................. 2-15
    Starting the BSP Editor.............................................................................. 2-15
    The BSP Editor Screen Layout.................................................................... 2-15
    The Command Area.................................................................................... 2-15
    The Console Area....................................................................................... 2-19
    Exporting a Tcl Script.................................................................................. 2-19
    Creating a New BSP.................................................................................... 2-20
    BSP Validation Errors.................................................................................. 2-21
  Run Configurations in the SBT for Eclipse.................................................... 2-21
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tc1 Commands</td>
<td>3-4</td>
</tr>
<tr>
<td>Tcl Scripts</td>
<td>3-4</td>
</tr>
<tr>
<td>The Command Shell</td>
<td>3-4</td>
</tr>
<tr>
<td>Getting Started in the SBT Command Line</td>
<td>3-5</td>
</tr>
<tr>
<td>Prerequisites</td>
<td>3-5</td>
</tr>
<tr>
<td>Creating Hello_World for an Intel FPGA Development Board</td>
<td>3-6</td>
</tr>
<tr>
<td>Running Hello_World on an Intel FPGA Development Board</td>
<td>3-6</td>
</tr>
<tr>
<td>Debugging hello_world</td>
<td>3-7</td>
</tr>
<tr>
<td>Software Build Tools Scripting Basics</td>
<td>3-8</td>
</tr>
<tr>
<td>Creating a BSP with a Script</td>
<td>3-9</td>
</tr>
<tr>
<td>Creating an Application Project with a Script</td>
<td>3-11</td>
</tr>
<tr>
<td>Running make</td>
<td>3-12</td>
</tr>
<tr>
<td>Creating Memory Initialization Files</td>
<td>3-12</td>
</tr>
</tbody>
</table>

**Software Build Tools** ...................................................................... 4-1

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Road Map for the SBT</td>
<td>4-2</td>
</tr>
<tr>
<td>What the Build Tools Create</td>
<td>4-2</td>
</tr>
<tr>
<td>Comparing the Command Line with Eclipse</td>
<td>4-2</td>
</tr>
<tr>
<td>Makefiles</td>
<td>4-2</td>
</tr>
<tr>
<td>Modifying Makefiles</td>
<td>4-3</td>
</tr>
<tr>
<td>Makefile Targets</td>
<td>4-3</td>
</tr>
<tr>
<td>Embedded Software Projects</td>
<td>4-4</td>
</tr>
<tr>
<td>Applications and Libraries</td>
<td>4-4</td>
</tr>
<tr>
<td>Board Support Packages</td>
<td>4-5</td>
</tr>
<tr>
<td>Software Build Process</td>
<td>4-7</td>
</tr>
<tr>
<td>Common BSP Tasks</td>
<td>4-8</td>
</tr>
<tr>
<td>Adding the SBT to Your Tool Flow</td>
<td>4-8</td>
</tr>
<tr>
<td>Linking and Locating</td>
<td>4-10</td>
</tr>
<tr>
<td>Other BSP Tasks</td>
<td>4-15</td>
</tr>
<tr>
<td>Details of BSP Creation</td>
<td>4-19</td>
</tr>
<tr>
<td>BSP Settings File Creation</td>
<td>4-21</td>
</tr>
<tr>
<td>Generated and Copied Files</td>
<td>4-21</td>
</tr>
<tr>
<td>HAL BSP Files and Folders</td>
<td>4-21</td>
</tr>
<tr>
<td>Linker Map Validation</td>
<td>4-26</td>
</tr>
<tr>
<td>Tcl Scripts for BSP Settings</td>
<td>4-26</td>
</tr>
<tr>
<td>Calling a Custom BSP Tcl Script</td>
<td>4-26</td>
</tr>
<tr>
<td>Revising Your BSP</td>
<td>4-29</td>
</tr>
<tr>
<td>Rebuilding Your BSP</td>
<td>4-29</td>
</tr>
<tr>
<td>Regenerating Your BSP</td>
<td>4-30</td>
</tr>
<tr>
<td>Updating Your BSP</td>
<td>4-32</td>
</tr>
<tr>
<td>Recreating Your BSP</td>
<td>4-33</td>
</tr>
<tr>
<td>Specifying BSP Defaults</td>
<td>4-34</td>
</tr>
<tr>
<td>Top Level Tcl Script for BSP Defaults</td>
<td>4-35</td>
</tr>
<tr>
<td>Specifying the Default stdio Device</td>
<td>4-36</td>
</tr>
<tr>
<td>Specifying the Default System Timer</td>
<td>4-37</td>
</tr>
<tr>
<td>Specifying the Default Memory Map</td>
<td>4-37</td>
</tr>
<tr>
<td>Specifying Default Bootloader Parameters</td>
<td>4-37</td>
</tr>
<tr>
<td>Using Individual Default Tcl Procedures</td>
<td>4-38</td>
</tr>
</tbody>
</table>
Overview of the Hardware Abstraction Layer................................. 5-1
Getting Started with the Hardware Abstraction Layer........................ 5-1
HAL Architecture for Embedded Software Systems............................ 5-2
Services...................................................................................... 5-2
Layers of a HAL-Based System...................................................... 5-2
Applications versus Drivers.......................................................... 5-2
Generic Device Models................................................................ 5-3
C Standard Library—newlib.......................................................... 5-4
Embedded Hardware Supported by the HAL.................................. 5-4
Processor Core Support................................................................. 5-4
Supported Peripherals................................................................. 5-4
MPU Support.............................................................................. 5-5
MMU Support.............................................................................. 5-6
Developing Programs Using the Hardware Abstraction Layer............. 6-1
HAL BSP Settings......................................................................... 6-1
The Embedded Project Structure.................................................... 6-2
The system.h System Description File........................................... 6-3
Data Widths and the HAL Type Definitions....................................... 6-3
UNIX-Style Interface.................................................................... 6-4
File System.................................................................................. 6-5
Using Character-Mode Devices....................................................... 6-6
Standard Input, Standard Output and Standard Error....................... 6-6
General Access to Character Mode Devices.................................... 6-7
C++ Streams.............................................................................. 6-7
/dev/null.................................................................................... 6-7
Lightweight Character-Mode I/O.................................................... 6-7
Intel FPGA Logging Functions..................................................... 6-7
Using File Subsystems................................................................... 6-13
Host-Based File System............................................................... 6-13
Using Timer Devices.................................................................... 6-14
System Clock Driver................................................................. 6-14
Alarms...................................................................................... 6-15
Timestamp Driver....................................................................... 6-16
Developing Device Drivers for the Hardware Abstraction Layer .............. 7-1

Driver Integration in the HAL API .......................................................... 7-1
The HAL Peripheral-Specific API ............................................................. 7-2
Preparing for HAL Driver Development ................................................. 7-2
Development Flow for Creating Device Drivers ...................................... 7-2
Hardware Design Concepts ................................................................. 7-3
  The Relationship Between the .sopcinfo File and system.h .................. 7-3
  Using the System Generation Tool to Optimize Hardware ................... 7-3
  Components, Devices, and Peripherals ............................................. 7-3
Accessing Hardware ......................................................................... 7-3
Creating Embedded Drivers for HAL Device Classes ............................ 7-5
  Character-Mode Device Drivers ....................................................... 7-5
  File Subsystem Drivers ................................................................. 7-5
  Timer Device Drivers ................................................................. 7-8
  Flash Device Drivers ................................................................ 7-9

Using Flash Devices ................................................................................. 6-17
  Simple Flash Access ........................................................................ 6-17
  Block Erasure or Corruption ............................................................. 6-18
  Fine-Grained Flash Access .............................................................. 6-19
Using DMA Devices ................................................................................. 6-23
  DMA Transmit Channels ................................................................. 6-24
  DMA Receive Channels ................................................................. 6-24
Using Interrupt Controllers ................................................................. 6-27
Reducing Code Footprint in Embedded Systems ....................................... 6-27
  Enable Compiler Optimizations .................................................... 6-28
  Use Reduced Device Drivers .......................................................... 6-28
  Reduce the File Descriptor Pool ..................................................... 6-29
  Use /dev/null .................................................................................. 6-29
  Use a Smaller File I/O Library ......................................................... 6-29
  Use the Lightweight Device Driver API ............................................ 6-37
  Use the Minimal Character-Mode API ............................................. 6-37
  Eliminate Unused Device Drivers .................................................... 6-38
  Eliminate Unneeded Exit Code ....................................................... 6-39
  Turn off C++ Support ................................................................... 6-39
Boot Sequence and Entry Point ............................................................ 6-39
  Hosted Versus Free-Standing Applications ....................................... 6-39
  Boot Sequence for HAL-Based Programs ......................................... 6-40
  Customizing the Boot Sequence ..................................................... 6-41
Memory Usage ..................................................................................... 6-41
  Memory Sections ........................................................................... 6-41
  Assigning Code and Data to Memory Partitions ................................. 6-42
  Placement of the Heap and Stack ................................................... 6-44
  Global Pointer Register ................................................................ 6-44
  Boot Modes ..................................................................................... 6-46
Working with HAL Source Files ............................................................. 6-47
  Finding HAL Files ......................................................................... 6-47
  Overriding HAL Functions ............................................................. 6-47

Using DMA Devices................................................................................. 6-23
  DMA Transmit Channels ................................................................. 6-24
  DMA Receive Channels ................................................................. 6-24
Using Interrupt Controllers ................................................................. 6-27
Reducing Code Footprint in Embedded Systems ....................................... 6-27
  Enable Compiler Optimizations .................................................... 6-28
  Use Reduced Device Drivers .......................................................... 6-28
  Reduce the File Descriptor Pool ..................................................... 6-29
  Use /dev/null .................................................................................. 6-29
  Use a Smaller File I/O Library ......................................................... 6-29
  Use the Lightweight Device Driver API ............................................ 6-37
  Use the Minimal Character-Mode API ............................................. 6-37
  Eliminate Unused Device Drivers .................................................... 6-38
  Eliminate Unneeded Exit Code ....................................................... 6-39
  Turn off C++ Support ................................................................... 6-39
Boot Sequence and Entry Point ............................................................ 6-39
  Hosted Versus Free-Standing Applications ....................................... 6-39
  Boot Sequence for HAL-Based Programs ......................................... 6-40
  Customizing the Boot Sequence ..................................................... 6-41
Memory Usage ..................................................................................... 6-41
  Memory Sections ........................................................................... 6-41
  Assigning Code and Data to Memory Partitions ................................. 6-42
  Placement of the Heap and Stack ................................................... 6-44
  Global Pointer Register ................................................................ 6-44
  Boot Modes ..................................................................................... 6-46
Working with HAL Source Files ............................................................. 6-47
  Finding HAL Files ......................................................................... 6-47
  Overriding HAL Functions ............................................................. 6-47

Developing Device Drivers for the Hardware Abstraction Layer .............. 7-1

Driver Integration in the HAL API .......................................................... 7-1
The HAL Peripheral-Specific API ............................................................. 7-2
Preparing for HAL Driver Development ................................................. 7-2
Development Flow for Creating Device Drivers ...................................... 7-2
Hardware Design Concepts ................................................................. 7-3
  The Relationship Between the .sopcinfo File and system.h .................. 7-3
  Using the System Generation Tool to Optimize Hardware ................... 7-3
  Components, Devices, and Peripherals ............................................. 7-3
Accessing Hardware ......................................................................... 7-3
Creating Embedded Drivers for HAL Device Classes ............................ 7-5
  Character-Mode Device Drivers ....................................................... 7-5
  File Subsystem Drivers ................................................................. 7-5
  Timer Device Drivers ................................................................. 7-8
  Flash Device Drivers ................................................................ 7-9

Using DMA Devices................................................................................. 6-23
  DMA Transmit Channels ................................................................. 6-24
  DMA Receive Channels ................................................................. 6-24
Exception Handling......................................................................................................................... 8-1

Exception Handling Overview........................................................................................................... 8-1
  Exception Handling Terminology........................................................................................................ 8-1
  Interrupt Controllers............................................................................................................................ 8-3
  Latency and Response Time.................................................................................................................. 8-5

Interrupt Service Routines.................................................................................................................... 8-6
  HAL APIs for Hardware Interrupts........................................................................................................ 8-7
  HAL ISR Restrictions............................................................................................................................. 8-11
  Writing an ISR.................................................................................................................................... 8-11
  Registering an ISR with the Enhanced Interrupt API........................................................................ 8-13
  Enabling and Disabling Interrupts........................................................................................................ 8-14
  Configuring an External Interrupt Controller.................................................................................... 8-14
  C Example.......................................................................................................................................... 8-15
  Upgrading to the Enhanced HAL Interrupt API................................................................................ 8-16

Improving ISR Performance................................................................................................................ 8-17
  Software Performance Improvements................................................................................................. 8-17
  Hardware Performance Improvements................................................................................................. 8-22

Debugging ISRs...................................................................................................................................... 8-24

HAL Exception Handling System Implementation.................................................................................. 8-25
  Exception Handling System Structure................................................................................................. 8-25
  General Exception Funnel..................................................................................................................... 8-26
  Hardware Interrupt Funnel................................................................................................................... 8-26
  Software Exception Funnel.................................................................................................................... 8-28
  Invalid Instructions............................................................................................................................... 8-32

The Instruction-Related Exception Handler......................................................................................... 8-32
  Writing an Instruction-Related Exception Handler............................................................................ 8-32
  Registering an Instruction-Related Exception Handler....................................................................... 8-34
  Removing an Instruction-Related Exception Handler.......................................................................... 8-35

Cache and Tightly-Coupled Memory.................................................................................................... 9-1
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache Implementation</td>
<td>9-1</td>
</tr>
<tr>
<td>Defining Cache Properties</td>
<td>9-2</td>
</tr>
<tr>
<td>Initializing the Cache after Reset</td>
<td>9-2</td>
</tr>
<tr>
<td>Assembly Code to Initialize the Instruction Cache</td>
<td>9-3</td>
</tr>
<tr>
<td>Assembly Code to Initialize the Data Cache</td>
<td>9-3</td>
</tr>
<tr>
<td>HAL Behavior for Initializing the Cache after Reset</td>
<td>9-3</td>
</tr>
<tr>
<td>Device Driver Cache Considerations</td>
<td>9-3</td>
</tr>
<tr>
<td>HAL Behavior for Device Driver Cache Considerations</td>
<td>9-3</td>
</tr>
<tr>
<td>Cache Considerations for Writing Program Loaders</td>
<td>9-4</td>
</tr>
<tr>
<td>HAL Behavior for Cache Considerations for Writing Program Loaders</td>
<td>9-5</td>
</tr>
<tr>
<td>Managing Cache in Multi-Master and Multi-Processor Systems</td>
<td>9-5</td>
</tr>
<tr>
<td>Cache Implementation</td>
<td>9-5</td>
</tr>
<tr>
<td>Bit-31 Cache Bypass</td>
<td>9-5</td>
</tr>
<tr>
<td>HAL Behavior for Managing Cache in Multi-Master and Multi-Processor Systems</td>
<td>9-6</td>
</tr>
<tr>
<td>Tightly-Coupled Memory</td>
<td>9-6</td>
</tr>
<tr>
<td>Tightly-Coupled Memory</td>
<td>9-6</td>
</tr>
<tr>
<td>MicroC/OS-II Real-Time Operating System</td>
<td>10-1</td>
</tr>
<tr>
<td>Overview of the MicroC/OS-II RTOS</td>
<td>10-1</td>
</tr>
<tr>
<td>Licensing</td>
<td>10-1</td>
</tr>
<tr>
<td>Other RTOS Providers</td>
<td>10-2</td>
</tr>
<tr>
<td>The Nios II Implementation of MicroC/OS-II</td>
<td>10-2</td>
</tr>
<tr>
<td>MicroC/OS-II Architecture</td>
<td>10-2</td>
</tr>
<tr>
<td>MicroC/OS-II Device Drivers</td>
<td>10-3</td>
</tr>
<tr>
<td>Thread-Safe HAL Drivers</td>
<td>10-4</td>
</tr>
<tr>
<td>The newlib ANSI C Standard Library</td>
<td>10-5</td>
</tr>
<tr>
<td>Interrupt Service Routines for MicroC/OS-II</td>
<td>10-5</td>
</tr>
<tr>
<td>Implementing MicroC/OS-II Projects for the Processor</td>
<td>10-6</td>
</tr>
<tr>
<td>Ethernet and the NicheStack TCP/IP Stack</td>
<td>11-1</td>
</tr>
<tr>
<td>Prerequisites for Understanding the NicheStack TCP/IP Stack</td>
<td>11-1</td>
</tr>
<tr>
<td>Introduction to the NicheStack TCP/IP Stack - Edition</td>
<td>11-2</td>
</tr>
<tr>
<td>The NicheStack TCP/IP Stack Files and Directories</td>
<td>11-2</td>
</tr>
<tr>
<td>Support and Licensing</td>
<td>11-2</td>
</tr>
<tr>
<td>Other TCP/IP Stack Providers for the Processor</td>
<td>11-3</td>
</tr>
<tr>
<td>Using the NicheStack TCP/IP Stack - Edition</td>
<td>11-3</td>
</tr>
<tr>
<td>System Requirements</td>
<td>11-3</td>
</tr>
<tr>
<td>The NicheStack TCP/IP Stack Tasks</td>
<td>11-3</td>
</tr>
<tr>
<td>Initializing the Stack</td>
<td>11-4</td>
</tr>
<tr>
<td>Calling the Sockets Interface</td>
<td>11-6</td>
</tr>
<tr>
<td>Configuring the NicheStack TCP/IP Stack in a Program</td>
<td>11-8</td>
</tr>
<tr>
<td>NicheStack TCP/IP Stack General Settings</td>
<td>11-8</td>
</tr>
<tr>
<td>IP Options</td>
<td>11-9</td>
</tr>
<tr>
<td>TCP Options</td>
<td>11-9</td>
</tr>
<tr>
<td>Further Information</td>
<td>11-9</td>
</tr>
<tr>
<td>Known Limitations</td>
<td>11-9</td>
</tr>
</tbody>
</table>
Read-Only Zip File System ................................................................. 12-1
  Using the Read-Only Zip File System in a Project ................................................. 12-1
  Preparing the Zip File ......................................................................................... 12-1
  Programming the Zip File to Flash ...................................................................... 12-2

Publishing Component Information to Embedded Software ...................... 13-1
  Embedded Component Information Flow .......................................................... 13-1
  Embedded Component Information Flow Diagram ........................................... 13-1
  Tcl Assignment Statements ............................................................................ 13-2
  Embedded Software Assignments ..................................................................... 13-2
  C Macro Namespace ....................................................................................... 13-2
  Configuration Namespace ............................................................................... 13-3
  Memory Initialization Namespace .................................................................. 13-7

HAL API Reference ...................................................................................... 14-1
  HAL API Functions ......................................................................................... 14-1
    _exit() ........................................................................................................... 14-1
    _rename() ..................................................................................................... 14-2
    alt_dcache_flush() ....................................................................................... 14-3
    alt_dcache_flush_all() ............................................................................... 14-3
    alt_dcache_flush_no_writeback() ............................................................... 14-4
    alt_uncached_malloc() ............................................................................... 14-5
    alt_uncached_free() .................................................................................... 14-6
    alt_remap_uncached() ................................................................................ 14-7
    alt_remap_cached() ..................................................................................... 14-7
    alt_icache_flush_all() ................................................................................ 14-8
    alt_icache_flush() ........................................................................................ 14-9
    alt_alarm_start() ........................................................................................ 14-10
    alt_alarm_stop() .......................................................................................... 14-11
    alt_dma_rxchan_depth() .............................................................................. 14-12
    alt_dma_rxchan_close() ............................................................................... 14-13
    alt_dev_reg() ................................................................................................ 14-14
    alt_dma_rxchan_open() ................................................................................ 14-15
    alt_dma_rxchan_prepare() ......................................................................... 14-16
    alt_dma_rxchan_reg() ................................................................................... 14-17
    alt_dma_txchan_close() ............................................................................... 14-18
    alt_dma_txchan_ioctl() ................................................................................ 14-19
    alt_dma_txchan_open() ............................................................................... 14-19
    alt_dma_txchan_reg() ................................................................................... 14-20
    alt_dma_txchan_close() ............................................................................... 14-21
    alt_exception_cause_generated_bad_addr() ............................................... 14-23
    alt_erase_flash_block() .............................................................................. 14-23
    alt_dma_rxchan_ioctl() ................................................................................ 14-24
    alt_dma_txchan_space() ............................................................................... 14-26
    alt_dma_txchan_send() ............................................................................... 14-27
alt_flash_open_dev()................................................................. 14-28
alt_fs_reg()............................................................................. 14-29
alt_get_flash_info()................................................................. 14-29
alt_ic_irq_disable()................................................................. 14-30
alt_ic_irq_enabled()................................................................. 14-31
alt_ic_isr_register()............................................................... 14-32
alt_ic_irq_enable()................................................................. 14-34
alt_instruction_exception_register()...................................... 14-35
alt_irq_disable().................................................................. 14-36
alt_irq_cpu_enable_interrupts()............................................. 14-37
alt_irq_disable_all().............................................................. 14-38
alt_irq_enable()................................................................... 14-39
alt_irq_enable_all()............................................................... 14-39
alt_irq_enabled().................................................................. 14-40
alt_irq_init()........................................................................ 14-41
alt_irq_pending()................................................................. 14-42
alt_irq_register()................................................................. 14-43
alt_llist_insert().................................................................. 14-44
alt_llist_remove()................................................................. 14-45
alt_load_section()................................................................ 14-46
alt_nticks().......................................................................... 14-47
alt_read_flash()..................................................................... 14-47
alt_tick().............................................................................. 14-48
alt_ticks_per_second()........................................................... 14-49
alt_timestamp()................................................................... 14-50
alt_timestamp_freq()............................................................. 14-51
alt_timestamp_start()............................................................ 14-51
alt_write_flash().................................................................. 14-52
alt_write_flash_block()......................................................... 14-53
close()................................................................................. 14-54
fstat().................................................................................. 14-55
fork()................................................................................... 14-56
fcntl()................................................................................. 14-57
execve().............................................................................. 14-58
getpid()............................................................................... 14-58
kill().................................................................................. 14-59
stat()................................................................................... 14-60
settimeofday().................................................................... 14-61
wait().................................................................................. 14-62
unlink().............................................................................. 14-62
sbrk().................................................................................. 14-63
link().................................................................................. 14-63
lseek()................................................................................. 14-64
alt_sysclk_init().................................................................. 14-65
open().................................................................................. 14-65
times()................................................................................. 14-66
read().................................................................................. 14-67
write()................................................................................ 14-68
usleep()............................................................................... 14-70
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>alt_lock_flash()</td>
<td>14-71</td>
</tr>
<tr>
<td>gettimeofday()</td>
<td>14-72</td>
</tr>
<tr>
<td>ioctl()</td>
<td>14-73</td>
</tr>
<tr>
<td>isatty()</td>
<td>14-74</td>
</tr>
<tr>
<td><strong>HAL Standard Types</strong></td>
<td>14-75</td>
</tr>
<tr>
<td>alt_getchar()</td>
<td>14-75</td>
</tr>
<tr>
<td>alt_putchar()</td>
<td>14-75</td>
</tr>
<tr>
<td>alt_putchar()</td>
<td>14-75</td>
</tr>
<tr>
<td>alt_putstr()</td>
<td>14-75</td>
</tr>
<tr>
<td>alt_getchar()</td>
<td>14-76</td>
</tr>
<tr>
<td>alt_putstr()</td>
<td>14-76</td>
</tr>
<tr>
<td>alt_putchar()</td>
<td>14-76</td>
</tr>
<tr>
<td>alt_putstr()</td>
<td>14-76</td>
</tr>
<tr>
<td><strong>ADC HAL Device Driver</strong></td>
<td>14-78</td>
</tr>
<tr>
<td>adc_stop()</td>
<td>14-79</td>
</tr>
<tr>
<td>adc_start()</td>
<td>14-79</td>
</tr>
<tr>
<td>adc_set_mode_run_once()</td>
<td>14-79</td>
</tr>
<tr>
<td>adc_set_mode_run_continuously</td>
<td>14-79</td>
</tr>
<tr>
<td>adc_recalibrate()</td>
<td>14-80</td>
</tr>
<tr>
<td>adc_interrupt_enable()</td>
<td>14-80</td>
</tr>
<tr>
<td>adc_interrupt_disable()</td>
<td>14-80</td>
</tr>
<tr>
<td>adc_clear_interrupt_status</td>
<td>14-80</td>
</tr>
<tr>
<td>adc_wait_for_interrupt - ADC Sample Storage Status Register</td>
<td>14-80</td>
</tr>
<tr>
<td>adc_interrupt_asserted</td>
<td>14-80</td>
</tr>
<tr>
<td>adc_wait_for_interrupt - IRQ Status Register</td>
<td>14-80</td>
</tr>
<tr>
<td>alt_adc_word_read()</td>
<td>14-80</td>
</tr>
<tr>
<td>adc_interrupt_disable()</td>
<td>14-80</td>
</tr>
<tr>
<td>adc_interrupt_enable()</td>
<td>14-80</td>
</tr>
<tr>
<td>adc_recalibrate()</td>
<td>14-80</td>
</tr>
<tr>
<td>adc_set_mode_run_once()</td>
<td>14-80</td>
</tr>
<tr>
<td>adc_set_mode_run_continuously</td>
<td>14-80</td>
</tr>
<tr>
<td><strong>Software Build Tools Reference</strong></td>
<td>15-1</td>
</tr>
<tr>
<td>Software Build Tools Utilities</td>
<td>15-1</td>
</tr>
<tr>
<td>Logging Levels</td>
<td>15-2</td>
</tr>
<tr>
<td>Setting Values</td>
<td>15-2</td>
</tr>
<tr>
<td>Utility and Script Summary</td>
<td>15-2</td>
</tr>
<tr>
<td>nios2-app-generate-makefile</td>
<td>15-3</td>
</tr>
<tr>
<td>nios2-bsp-create-settings</td>
<td>15-4</td>
</tr>
<tr>
<td>nios2-bsp-generate-files</td>
<td>15-6</td>
</tr>
<tr>
<td>nios2-bsp-query-settings</td>
<td>15-7</td>
</tr>
<tr>
<td>nios2-bsp-update-settings</td>
<td>15-7</td>
</tr>
<tr>
<td>nios2-lib-generate-makefile</td>
<td>15-8</td>
</tr>
<tr>
<td>nios2-bsp-editor</td>
<td>15-9</td>
</tr>
<tr>
<td>nios2-app-update-makefile</td>
<td>15-10</td>
</tr>
<tr>
<td>nios2-lib-update-makefile</td>
<td>15-11</td>
</tr>
<tr>
<td>nios2-swexample-create</td>
<td>15-13</td>
</tr>
<tr>
<td>nios2-elf-insert</td>
<td>15-15</td>
</tr>
<tr>
<td>nios2-elf-query</td>
<td>15-15</td>
</tr>
<tr>
<td>nios2-flash-programmer-generate</td>
<td>15-16</td>
</tr>
<tr>
<td>nios2-bsp-console</td>
<td>15-16</td>
</tr>
<tr>
<td>alt-file-convert (BETA)</td>
<td>15-17</td>
</tr>
<tr>
<td><strong>Design Example Scripts</strong></td>
<td>15-17</td>
</tr>
<tr>
<td>create-this-bsp</td>
<td>15-19</td>
</tr>
<tr>
<td>create-this-app</td>
<td>15-19</td>
</tr>
<tr>
<td>Finding create-this-app and create-this-bsp</td>
<td>15-24</td>
</tr>
</tbody>
</table>
There are two variations of Software Developer’s Handbook:

- The Classic Software Developer’s Handbook describes embedded software development tools for the Classic cores; and there are no future updates planned.
- The Software Developer’s Handbook describes embedded software development tools for the Nios II processor. It does not describe IP cores.

This handbook describes the basic information needed to develop embedded software for the Intel FPGA Nios II processor. The Intel FPGA processor contains new features added after the 14.0 version. The chapters in this handbook describe the software development environment, the Embedded Design Suite (EDS) tools that are available to you, and the process for developing software.

Installing Windows* Subsystem for Linux* (WSL) on Windows*

Starting with EDS in version 19.2 and version 19.1, the Cygwin component in the Windows’ version of EDS has been removed and replaced with WSL.

The procedure for installing WSL:

1. Go to https://docs.microsoft.com/en-us/windows/wsl/install-win10 and follow Microsoft’s instructions to install Ubuntu 18.04 LTS for WSL.
   
   Note: Windows 10 build version 16215.0 or higher is the recommended operating system version.

2. After installation has successfully completed, launch Ubuntu 18.04.

3. Install additional distro packages required for EDS using the following commands:

   ```
sudo apt install wsl
sudo apt install dos2unix sudo apt install make
```

   Note: • For the Command Shell, use all command line tools, as before, but you need to add .exe to launch a Windows executable, like eclipse-nios2.exe or jtagconfig.exe.
   • BSP and application projects from previous releases are not compatible with this WSL solution. You are required to regenerate your projects.

Prerequisites for Understanding the Embedded Design Suite

The Software Developer’s Handbook assumes you have a basic familiarity with embedded processor concepts. You do not need to be familiar with any specific Intel FPGA technology or with Intel FPGA
development tools. Familiarity with Intel FPGA hardware development tools can give you a deeper understanding of the reasoning behind the software development environment. However, software developers can create and debug applications without further knowledge of Intel FPGA technology.

**Finding Nios II EDS Files**

When you install the Nios II EDS, you specify a root directory for the EDS file structure. This root directory must be adjacent to the Quartus® II installation. When you install the latest release of the Nios II EDS on the Windows operating system, choose a local root folder that identifies the content, for example: `c:\altera\<latest release number>\nios2eds`.

**Note:** For simplicity, this handbook refers to this directory as `<Nios II EDS install path>`.

**Software Development Environment**

The EDS provides a consistent software development environment that works for all processor systems. With the EDS running on a host computer, an Intel FPGA FPGA, and a JTAG download cable (such as an Intel FPGA USB-Blaster™ download cable), you can write programs for and communicate with any processor system. The processor’s JTAG debug module provides a single, consistent method to connect to the processor using a JTAG download cable. Accessing the processor is the same, regardless of whether a device implements only a processor system, or whether the processor is embedded deeply in a complex multiprocessor system. Therefore, you do not need to spend time manually creating interface mechanisms for the embedded processor.

The EDS includes proprietary and open-source tools (such as the GNU C/C++ tool chain) for creating programs. The EDS automates board support package (BSP) creation for processor-based systems, eliminating the need to spend time manually creating BSPs. The BSP provides a C/C++ runtime environment, insulating you from the hardware in your embedded system. Intel FPGA BSPs contain the Intel FPGA hardware abstraction layer (HAL), an optional RTOS, and device drivers.

**EDS Development Flows**

A development flow is a way of using a set of development tools together to create a software project. The EDS provides the following development flows for creating programs:

- The Software Build Tools (SBT), which provides two user interfaces:
  - The SBT command line
  - The SBT for Eclipse™

**SBT Development Flow**

The SBT allows you to create software projects, with detailed control over the software build process. The same SBT utilities, scripts and Tcl commands are available from both the command line and the SBT for Eclipse graphical user interface (GUI).

The SBT allows you to create and manage single-threaded programs as well as complex applications based on an RTOS and middleware libraries available from Intel FPGA and third-party vendors.
The SBT provides powerful Tcl scripting capabilities. In a Tcl script, you can query project settings, specify project settings conditionally, and incorporate the software project creation process in a scripted software development flow. Tcl scripting is supported both in Eclipse and at the command line.

**Related Information**

Software Build Tools on page 4-1
For more information about Tcl scripting.

**SBT for Eclipse**

The SBT for Eclipse is a thin GUI layer that runs the SBT utilities and scripts behind the scenes, presenting a unified development environment. The SBT for Eclipse provides a consistent development platform that works for all processor systems. You can accomplish all software development tasks within Eclipse, including creating, editing, building, running, debugging, and profiling programs.

The SBT for Eclipse is based on the popular Eclipse framework and the Eclipse C/C++ development toolkit (CDT) plugins. The SBT creates your project makefiles for you, and Eclipse provides extensive capabilities for interactive debugging and management of source files.

The SBT for Eclipse also allows you to import and debug projects you created in the Command Shell.

**Related Information**

- Getting Started with the Graphical User Interface on page 2-1
  For more information about the Nios II SBT for Eclipse.
- Eclipse Foundation
  For more information about Eclipse, visit the Eclipse Foundation website.

**SBT Command Line**

In the SBT command line development flow, you create, modify, build, and run programs with SBT commands typed at a command line or embedded in a script. You run the SBT commands from the Command Shell.

**Note:** To debug your command-line program, import your SBT projects to Eclipse. You can further edit, rebuild, run, and debug your imported project in Eclipse.

**Related Information**

Getting Started from the Command Line on page 3-1
For more information about the Nios II SBT in command-line mode

**Programs**

Each program you develop consists of an application project, optional user library projects, and a BSP project. You build your program to create an Executable and Linking Format File (.elf) which runs on a processor.

The SBT creates software projects for you. Each project is based on a makefile.

**Makefiles and the SBT**

The makefile is the central component of a software project, whether the project is created with the SBT for Eclipse, or on the command line. The makefile describes all the components of a software project and how
they are compiled and linked. With a makefile and a complete set of C/C++ source files, your software project is fully defined.

As a key part of creating a software project, the SBT creates a makefile for you. Projects are sometimes called "user-managed," because you, the user, are responsible for the content of the project makefile. You use the SBT to control what goes in the makefile.

**Related Information**
Software Build Tools Reference on page 15-1
For more information about creating makefiles.

**Software Project Types**

**Application Project**
A C/C++ application project consists of a collection of source code, plus a makefile. A typical characteristic of an application is that one of the source files contains function `main()`. An application includes code that calls functions in libraries and BSPs. The makefile compiles the source code and links it with a BSP and one or more optional libraries, to create one .elf file.

**User Library Project**
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. A collection of common arithmetical functions is one example. A user library does not contain a `main()` function.

**BSP Project**
A BSP project is a specialized library containing system-specific support code. A BSP provides a software runtime environment customized for one processor in a hardware system. The EDS provides tools to modify settings that control the behavior of the BSP.

A BSP contains the following elements:

- Hardware abstraction layer
- Optional custom newlib C standard library
- Device drivers
- Optional software packages
- Optional real-time operating system

**Related Information**
- Intel FPGA Software Packages for Embedded Systems on page 1-5
- Overview of the Hardware Abstraction Layer on page 5-1
- Software Build Tools Reference on page 15-1
  For more information, refer to the "Nios II Embedded Software Projects" chapter.
- Intel FPGA Software Packages for Embedded Systems on page 1-5
- MicroC/OS-II Real-Time Operating System on page 10-1

---

(1) The complete HTML documentation for newlib resides in the EDS directory.
Intel FPGA Software Packages for Embedded Systems

The EDS includes software packages to extend the capabilities of your software. You can include these software packages in your BSP.

Table 1-1: Intel FPGA Software Packages Distributed with the EDS

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read-only zip file system</td>
<td>Refer to the &quot;Read-Only Zip File System” chapter.</td>
</tr>
<tr>
<td>Host file system</td>
<td>Refer to the &quot;Developing Programs Using the Hardware Abstraction Layer” chapter.</td>
</tr>
</tbody>
</table>

Related Information

- **Read-Only Zip File System** on page 12-1
- **Developing Programs Using the Hardware Abstraction Layer** on page 6-1
- **Ethernet and the NicheStack TCP/IP Stack** on page 11-1
- **Embedded Software**
  For more information about a complete list of the additional software packages available from Intel FPGA’s partners

Embedded Design Examples

The EDS includes documented software examples to demonstrate all prominent features of the processor and the development environment. The examples can help you start the development of your custom design. They provide a stable starting point for exploring design options. Also, they demonstrate many commonly used features of the EDS.

Note: The hardware design examples are available on the Embedded Processor Design Examples web page.

Related Information

Embedded Processor Design Examples

Hardware Examples

You can run hardware designs on many development boards. The hardware examples for each development board are available on the Design Examples web page.

Note: The with MMU design is intended to demonstrate Linux. This design does not work with the SBT, because the SBT does not support the MMU.

Related Information

- **Nios II Ethernet Standard Design Example**
- **Nios II Processor with Memory Management Unit Design Example**
- **Intel FPGA Development Kits**
Software Examples

You can run software examples that run on many of the hardware design examples described in the previous section.

The software examples directory contains the following files:
- Source file (.c)
- Header file (.h)
- readme.txt
- template.xml

Figure 1-1: Software Design Example Directory Structure

Third-Party Embedded Tools Support

Several third-party vendors support the processor, providing products such as design services, operating systems, stacks, other software libraries, and development tools.

Related Information
Processor
For more information about the most up-to-date information about third-party support for the processor.

Additional Information

This handbook is one part of the complete processor documentation suite. Consult the following references for further information:
The Processor Reference Guide defines the processor hardware architecture and features, including the instruction set architecture.

The Embedded Peripherals IP User Guide provides a reference for the peripherals distributed with the processor. This handbook describes the hardware structure and software drivers for each peripheral.

The Embedded Design Handbook describes how to use the software development tools effectively, and recommends design styles and practices for developing, debugging, and optimizing embedded systems.

The Intel FPGA Knowledge Database is an Internet resource that offers solutions to frequently asked questions with an easy-to-use search engine.

Application notes and tutorials offer step-by-step instructions on using the processor for a specific application or purpose. These documents are available on the Intel FPGA Processor Documentation web page.

The EDS documentation launchpad. The launchpad is an HTML page installed with the EDS, which provides links to documentation, examples, and other resources. The way you open the launchpad depends on your software platform.

- In the Windows operating system, on the Start menu, point to Programs > Intel FPGA > Nios II EDS, and click <version> Documentation.
- In the Linux operating system, open < EDS install path>/documents/index.html in a web browser.

Related Information

- **Nios II Processor Reference Guide**
  For more information on hardware architecture and features, including the instruction set architecture

- **Embedded Peripherals IP User Guide**
  For more information on hardware structure and Nios II software drivers for each peripheral

- **Embedded Design Handbook**
  For more information on design styles and practices for developing, debugging, and optimizing embedded systems

- **Intel FPGA Knowledge Database**
  For more information, refer to the Knowledge Database page of the Intel FPGA website.

- **Intel FPGA Nios II Processor Documentation**
The Nios™ II Software Build Tools (SBT) for Eclipse is a set of plugins based on the Eclipse framework and the Eclipse C/C++ development toolkit (CDT) plugins. The Nios II SBT for Eclipse provides a consistent development platform that works for all Nios II embedded processor systems. You can accomplish all Nios II software development tasks within Eclipse, including creating, editing, building, running, debugging, and profiling programs.

Installing Eclipse IDE into EDS

Starting with EDS v19.1, the EDS requires the Eclipse IDE component to be manually installed.

The procedure for installing Eclipse IDE:

1. Download CDT 8.8.1 which is Eclipse C/C++ IDE for Mars.
2. Extract the downloaded file into this directory: `<installation directory>/nios2eds/bin`. You should see the `<installation directory>/nios2eds/bin/eclipse` folder after extraction is done.
3. Rename the `<installation directory>/nios2eds/bin/eclipse` folder to `<installation directory>/nios2eds/bin/eclipse_nios2`.
4. Extract `<installation directory>/nios2eds/bin/eclipse_nios2_plugins.zip` for Windows or `<installation directory>/nios2eds/bin/eclipse_nios2_plugins.tar.gz` for Linux to `<installation directory>/nios2eds/bin`. The extraction overrides files in `<installation directory>/nios2eds/bin/eclipse_nios2`.
5. Verify the extraction is done correctly by making sure you see the `<installation directory>/nios2eds/bin/eclipse_nios2/plugin_customization.ini` file.
6. You can now launch SBT for Eclipse using `eclipse-nios2.exe`.

Note: The instructions are also included in the `<installation directory>/nios2eds/bin/README` file.
Getting Started with Nios II Software in Eclipse

Writing software for the Nios II processor is similar to writing software for any other microcontroller family. The easiest way to start designing effectively is to purchase a development kit from Intel FPGA that includes documentation, a ready-made evaluation board, a getting-started reference design, and all the development tools necessary to write Nios II programs.

Modifying existing code is a common, easy way to learn to start writing software in a new environment. The Nios II Embedded Design Suite (EDS) provides many example software designs that you can examine, modify, and use in your own programs. The provided examples range from a simple "Hello world" program, to a working RTOS example, to a full TCP/IP stack running a web server. Each example is documented and ready to compile.

This section guides you through the most fundamental operations in the Nios II SBT for Eclipse in a tutorial-like fashion. It shows how to create an application project for the Nios II processor, along with the board support package (BSP) project required to interface with your hardware. It also shows how to build the application and BSP projects in Eclipse, and how to run the software on an Intel FPGA development board.

The Nios II SBT for Eclipse Workbench

The term 'workbench’ refers to the Nios II SBT for Eclipse desktop development environment. The workbench is where you edit, compile and debug your programs in Eclipse.

Perspectives, Editors, and Views

Each workbench window contains one or more perspectives. Each perspective provides a set of capabilities for accomplishing a specific type of task.

Most perspectives in the workbench comprise an editor area and one or more views. An editor allows you to open and edit a project resource (i.e., a file, folder, or project). Views support editors, and provide alternative presentations and ways to navigate the information in your workbench.

Any number of editors can be open at once, but only one can be active at a time. The main menu bar and toolbar for the workbench window contain operations that are applicable to the active editor. Tabs in the editor area indicate the names of resources that are currently open for editing. An asterisk (*) indicates that an editor has unsaved changes. Views can also provide their own menus and toolbars, which, if present, appear along the top edge of the view. To open the menu for a view, click the drop-down arrow icon at the right of the view’s toolbar or right-click in the view. A view might appear on its own, or stacked with other views in a tabbed notebook.

For detailed information about the Eclipse workbench, perspectives, and views, refer to the Eclipse help system.

Before you create a Nios II project, you must ensure that the Nios II perspective is visible. To open the Nios II perspective, on the Window menu, point to Open Perspective, then Other, and click Nios II.

The Intel FPGA Bytestream Console

The workbench in Eclipse for Nios II includes a bytestream console, available through the Eclipse Console view. The Intel FPGA bytestream console enables you to see output from the processor’s stdout and stderr devices, and send input to its stdin device.
Creating a Project

In the Nios II perspective, on the File menu, point to Nios II Application and BSP from Template. The Nios II Application and BSP from Template wizard appears. This wizard provides a quick way to create an application and BSP at the same time.

Alternatively, you can create separate application, BSP and user library projects.

Specifying the Application

In the first page of the Nios II Application and BSP from Template wizard, you specify a hardware platform, a project name, and a project template. You optionally override the default location for the application project, and specify a processor name if you are targeting a multiprocessor hardware platform.

You specify a BSP in the second page of the wizard.

Specifying the Hardware Platform

You specify the target hardware design by selecting a SOPC Information File (.sopcinfo) in the SOPC Information File name box.

Specifying the Project Name

Select a descriptive name for your project. The SBT creates a folder with this name to contain the application project files.

Letters, numbers, and the underscore (_) symbol are the only valid project name characters. Project names cannot contain spaces or special characters. The first character in the project name must be a letter or underscore. The maximum filename length is 250 characters.

Related Information

Specifying the BSP on page 2-4

For more information about how the SBT also creates a folder to contain BSP project files.

Specifying the Project Template

Project templates are ready-made, working software projects that serve as examples to show you how to structure your own Nios II projects. It is often easier to start with a working project than to start a blank project from scratch.

You select the project template from the Templates list.

The hello_world template provides an easy way to create your first Nios II project and verify that it builds and runs correctly.

Specifying the Project Location

The project location is the parent directory in which the SBT creates the project folder. By default, the project location is under the directory containing the .sopcinfo file, in a folder named software.

To place your application project in a different folder, turn off Use default location, and specify the path in the Project location box.
Specifying the Processor

If your target hardware contains multiple Nios II processors, CPU name contains a list of all available processors in your design. Select the processor on which your software is intended to run.

Specifying the BSP

When you have finished specifying the application project in the first page of the Nios II Application and BSP from Template wizard, you proceed to the second page by clicking Next.

On the second page, you specify the BSP to link with your application. You can create a new BSP for your application, or select an existing BSP. Creating a new BSP is often the simplest way to get a project running the first time.

You optionally specify the name and location of the BSP.

Specifying the BSP Project Name

By default, if your application project name is <project>, the BSP is named <project>_bsp. You can type in a different name if you prefer. The SBT creates a directory with this name, to contain the BSP project files.

Related Information

Specifying the Project Name on page 2-3
For more information about how the BSP project names are subject to the same restrictions as application project names.

Specifying the BSP Project Location

The BSP project location is the parent directory in which the SBT creates the folder. The default project location is the same as the default location for an application project. To place your BSP in a different folder, turn off Use default location, and specify the BSP location in the Project location box.

Selecting an Existing BSP

As an alternative to creating a BSP automatically from a template, you can associate your application project with a pre-existing BSP. Select Select an existing BSP project from your workspace, and select a BSP in the list. The Create and Import buttons to the right of the existing BSP list provide convenient ways to add BSPs to the list.

Creating the Projects

When you have specified your BSP, you click Finish to create the projects.

The SBT copies required source files to your project directories, and creates makefiles and other generated files. Finally, the SBT executes a make clean command on your BSP.

Related Information

Software Build Tools on page 4-1
For more information about the folders and files in a Nios II BSP.

Navigating the Project

When you have created a Nios II project, it appears in the Project Explorer view, which is typically displayed at the left side of the Nios II perspective. You can expand each project to examine its folders and files.
Building the Project

To build a Nios II project in the Nios II SBT for Eclipse, right-click the project name and click **Build Project**. A progress bar shows you the build status. The build process can take a minute or two for a simple project, depending on the speed of the host machine. Building a complex project takes longer.

During the build process, you view the build commands and command-line output in the Eclipse **Console** view.

When the build process is complete, the following message appears in the **Console** view, under the **C-Build [<project name>]** title:

```
[<project name> build complete]
```

If the project has a dependency on another project, such as a BSP or a user library, the SBT builds the dependency project first. This feature allows you to build an application and its BSP with a single command.

**Related Information**

Software Build Tools Reference on page 15-1

Configuring the FPGA

Before you can run your software, you must ensure that the correct hardware design is running on the FPGA. To configure the FPGA, you use the Programmer.

In the Windows operating system, you start the Programmer from the Nios II SBT for Eclipse, through the Nios II menu. In the Linux operating system, you start Programmer from the software.

The project directory for your hardware design contains an SRAM Object File (.sof) along with the .sopcinfo file. The .sof file contains the hardware design to be programmed in the FPGA.

**Related Information**

Programmer

For more information about programming an FPGA with Programmer.

Running the Project on Nios II Hardware

This section describes how to run a Nios II program using the Nios II SBT for Eclipse on Nios II hardware, such as an Intel FPGA development board.

**Note:** If your project was created with version 10.1 or earlier of the Nios II SBT, you must re-import it to create the Nios II launch configuration correctly.

To run a software project, right-click the application project name, point to **Run As**, and click **Nios II Hardware**. To run a software project as a ModelSim simulation, right-click the application project name, point to **Run As**, and click **Nios II ModelSim**.
This command carries out the following actions:

- Creates a Nios II run configuration.
- Builds the project executable. If all target files are up to date, nothing is built.
- Establishes communications with the target, and verifies that the FPGA is configured with the correct hardware design.
- Downloads the Executable and Linking Format File (.elf) to the target memory
- Starts execution at the .elf entry point.

Program output appears in the Nios II Console view. The Nios II Console view maintains a terminal I/O connection with a communication device connected to the Nios II processor in the hardware system, such as a JTAG UART. When the Nios II program writes to stdout or stderr, the Nios II Console view displays the text. The Nios II Console view can also accept character input from the host keyboard, which is sent to the processor and read as stdin.

To disconnect the terminal from the target, click the **Terminate** icon in the Nios II Console view. Terminating only disconnects the host from the target. The target processor continues executing the program.

**Related Information**

- **Run Configurations in the SBT for Eclipse** on page 2-21
  For more information about about run configurations.
- **Lauterbach GmbH Website**
  For more information about the Nios II instruction set.

### Debugging the Project on Nios II Hardware

This section describes how to debug a Nios II program using the Nios II SBT for Eclipse on Nios II hardware, such as an Intel FPGA development board.

To debug a software project, right-click the application project name, point to **Debug As**, and click **Nios II Hardware**. This command carries out the following actions:

- Creates a Nios II run configuration.
- Builds the project executable. If all target files are up to date, nothing is built.
- If debugging on hardware, establishes communications with the target, and verifies that the FPGA is configured with the correct hardware design.
- Downloads the .elf to the target memory.
- Sets a breakpoint at the top of main().
- Starts execution at the .elf entry point.

The Eclipse debugger with the Nios II plugins provides a Nios II perspective, allowing you to perform many common debugging tasks. Debugging a Nios II program with the Nios II plugins is generally the same as debugging any other C/C++ program with Eclipse and the CDT plugins.

For information about debugging with Eclipse and the CDT plugins, refer to the Eclipse help system.

**Related Information**

**Run Configurations in the SBT for Eclipse** on page 2-21
For more information about about run configurations.

### List of Debugging Tasks with the Nios II SBT for Eclipse

The debugging tasks you can perform with the Nios II SBT for Eclipse include the following tasks:
• Controlling program execution with commands such as:
  • Suspend (pause)
  • Resume
  • Terminate
  • Step Into
  • Step Over
  • Step Return
• Setting breakpoints and watchpoints
• Viewing disassembly
• Instruction stepping mode
• Displaying and changing the values of local and global variables in the following formats:
  • Binary
  • Decimal
  • Hexadecimal
• Displaying watch expressions
• Viewing and editing registers in the following formats:
  • Binary
  • Decimal
  • Hexadecimal
• Viewing and editing memory in the following formats:
  • Hexadecimal
  • ASCII
  • Signed integer
  • Unsigned integer
• Viewing stack frames in the Debug view

Console View
Just as when running a program, Eclipse displays program output in the Console view of Eclipse. The Console view maintains a terminal I/O connection with a communication device connected to the Nios II processor in the hardware system, such as a JTAG UART. When the Nios II program writes to `stdout` or `stderr`, the Console view displays the text. The Console view can also accept character input from the host keyboard, which is sent to the processor and read as `stdin`.

Disconnecting the Terminal from the Target
To disconnect the terminal from the target, click the Terminate icon in the Console view. Terminating only disconnects the host from the target. The target processor continues executing the program.

Note: If your project was created with version 10.1 or earlier of the Nios II SBT, you must re-import it to create the Nios II launch configuration correctly.

Using the Intel FPGA Bytestream Console
The Intel FPGA bytestream console enables you to see output from the processor’s `stdout` and `stderr` devices, and send input to its `stdin` device. The function of the Intel FPGA bytestream console is similar to the `nios2-terminal` command-line utility.

Open the Intel FPGA bytestream console in the Eclipse Console view the same way as any other Eclipse console, by clicking the Open Console button.
When you open the Intel FPGA bytestream console, the **Bytestream Console Selection** dialog box shows you a list of available bytestreams. This is the same set of bytestreams recognized by System Console. Select the bytestream connected to the processor you are debugging.

You can send characters to the processor’s `stdin` device by typing in the bytestream console. Be aware that console input is buffered on a line-by-line basis. Therefore, the processor does not receive any characters until you press the Enter key.

**Note:** A bytestream device can support only one connection at a time. You must close the Intel FPGA bytestream console before attempting to connect to the processor with the `nios2-terminal` utility, and vice versa.

**Related Information**

*Analyzing and Debugging Designs with the System Console*
For more information about how System Console recognizes bytestreams.

### Run Time Stack Checking And Exception Debugging

#### Before you begin

To enable extra exception information, navigate to **MegaWizard > Advanced Features Exception Checking > Extra Information Register**, and recompile the HW project and regenerate the BSP in the SBT for Eclipse.

1. Enable the **Run Time Stack Checking** in the BSP project from NIOS II SBT for Eclipse BSP Editor. From the BSP project, right-click and navigate to **BSP editor Settings > Advanced > hal > enable_run_time_stack_checking**.
2. Rebuild BSP and software.
3. Ensure that the FPGA is configured.
4. Start the Debug Session by navigating to **Debug As > Hardware**.
5. Run the Software.

#### Exception Debugging

To allow easier debugging of exceptions, first enable the extra exception information in the .

**Note:** This is already enabled if you have an MMU.

Also you can navigate to **MegaWizard > Advanced Features Exception Checking > Extra Information Register**.

**Note:** There are other options you can choose, like unimplemented instructions.

When an exception is hit, the cause value in the **Exception Register** can be decoded using the Exceptions (In Decreasing Priority Order) table from the **Processor Reference Handbook**.

**Note:** This table only provides the general cause.

**Related Information**

- **Nios II Classic Processor Reference Handbook**
  For more information about the Exception Register Decode Table, refer to the "Exception Overview" chapter in the "Programming Model" section.
Stack Overflow

To enable Stack Checking, go to the BSP Editor and click on the Settings tab, click on Advanced, hal, and then click enable_runtime_stack_checking. When the Stack Checking is enabled, extra code is added at the start of each function call to:

- Check the current value of the stack pointer
- Compare this to the max stack size, which is stored in the Exception Temp (ET) Register

If the stack pointed to is outside of the valid range, the software branches and calls a “break 3” instruction. This is seen by the Debug Control module.

**Note:** With stack checking on, malloc() and new() can detect heap exhaustion, as well.

**Example 2-1: Example of function with stack checking code**

```c
___vfprintf_internal_r:
000002ec:   addi sp,sp,-1308
000002f0:   bgeu sp,et,0x2f8 <___vfprintf_internal_r+12>
000002f4:   break 3
```

The bgeu and break 3 lines are what is added for the stack overflow checking. If the stack pointer has grown beyond its limits the break is called.

**Related Information**

**Embedded Design Handbook**
For more information, refer to the "Stack Overflow" chapter of the Embedded Design Handbook.

**Recognizing and Debugging a Stack Overflow**

When a stack overflow occurs, having registered an instruction-related exception handler helps you identify it by its behavior.

**Default Instruction-Related Exception Handler**

The default value for an instruction-related exception handler is when it is not registered.

If you don't register an instruction-related exception handler, the “break 3” instruction is picked up by the software trap logic and a break is passed to the debugger. You must roll back through the history in the debugger to find the memory operation that triggered the stack checking break.

**Note:** With stack checking on, malloc() and new() can detect heap exhaustion.
How to Isolate the Cause of a Sigtrap

How to isolate the cause of a **sigtrap** seen in the debugger with no instruction-related exception handle?

The Debugger breaks with **sigtrap**:

1. Use the thread view in the debug window and select the last state.
   
   This is the highest number. The last thread is the actual call than overflowed.

2. Switch to instruction stepping mode in the debugger by pressing the i-> button in the debug window, which opens the memory disassembly view.
   
   If there has been a stack overflow the disassembly view should show execution pointing to a **break3** after the stack check:

   ```
   ___vfprintf_internal_r:
   000002ec:   addi sp,sp,-1308
   000002f0:   bgeu sp,et,0x2f8 <___vfprintf_internal_r+12>
   000002f4:   break 3
   ```

3. Check the value of **sp** and **et** which holds the max stack side in the register view.

4. Move to the prior state in the debug window and re-check **sp** vs **et**.

**Figure 2-1: Debug window**
Custom Instruction-Related Exception Handler

For use outside the debugger, you can register your own instruction-related exception handler which is called when the break (or any exception) is seen.

On an exception, including overflow, the HAL calls the instruction-related exception handler, passing in the cause field from the exception register, and the address which caused the exception. At this point, it is up to you to decide what to do.

For more information about how to register an instruction-related exception, refer to

Related Information

- **Exception Handling** on page 8-1
  This chapter provides more information about the details on how to register an instruction-related exception.
- **Writing an Instruction-Related Exception Handler** on page 8-32
  This chapter provides more information about the details on how to register an instruction-related exception.
- **Registering an Instruction-Related Exception Handler** on page 8-34
  This chapter provides more information about the details on how to register an instruction-related exception.
- **Programming Model**
  For more information, refer to the "Programming Model" chapter of the *Nios II Processor Reference Handbook*.

Creating a Simple BSP

You create a BSP with default settings using the **Nios II Board Support Package** wizard. To start the wizard, on the **File** menu, point to **New** and click **Board Support Package**. The **Board Support Package** wizard enables you to specify the following BSP parameters:

- The name
- The underlying hardware design
- The location
- The operating system and version

You can select the operating system only at the time you create the BSP. To change operating systems, you must create a new BSP by using the Additional arguments to the **nios2-bsp** script.

If you intend to run the project in the ModelSim™ simulation environment, use the Additional arguments parameter to specify the location of the test bench simulation package descriptor file (**.spd**). The .spd file is located in the project directory. Specify the path as follows: ```-set QUARTUS_PROJECT_DIR=<relative path>```.

**Note:** Intel FPGA recommends that you use a relative path name, to ensure that the location of your project is independent of the installation directory.

After you have created the BSP, you have the following options for GUI-based BSP editing:

- To access and modify basic BSP properties, right-click the BSP project, click **Properties > BSP Properties**.
- To modify parameters and settings in detail using the Nios II BSP Editor, refer to *Using the BSP Editor*. 
Makefiles and the SBT for Eclipse

The SBT for Eclipse creates and manages the makefiles for software projects. When you create a project, the SBT creates a makefile based on the source content you specify and the parameters and settings you select. When you modify the project in Eclipse, the SBT updates the makefile to match.

Details of how each makefile is created and maintained vary depending on the project type, and on project options that you control. The authoritative specification of project contents is always the makefile, regardless how it is created or updated.

By default, the SBT manages the list of source files in your makefile, based on actions you take in Eclipse. However, in the case of applications and libraries, you have the option to manage sources manually. Both styles of source management are discussed in the following sections.

Eclipse Source Management

Application and user library makefiles are based on source files and properties that you specify directly. Eclipse source management allows you to add and remove source files with standard Eclipse actions, such as dragging a source file into and out of the Project Explorer view and adding a new source file through the File menu.

You can examine and modify many makefile properties in the Application Properties or Library Properties dialog box. To open the dialog box, right-click the project, click Properties > Application Properties or Properties > Library Properties.

Modifying a Makefile with Eclipse Source Management

Table 2-1: GUI Actions that Modify an Application or Makefile with Eclipse Source Management

<table>
<thead>
<tr>
<th>Modification</th>
<th>Where Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specifying the application or user library name</td>
<td>Application Properties or Library Properties dialog box.</td>
</tr>
<tr>
<td>Adding or removing source files</td>
<td>For more information, refer to the Eclipse help system.</td>
</tr>
<tr>
<td>Specifying a path to an associated BSP</td>
<td>Project References dialog box.</td>
</tr>
<tr>
<td>Specifying a path to an associated user library</td>
<td>Project References dialog box.</td>
</tr>
<tr>
<td>Enabling, disabling or modifying compiler options</td>
<td>Application Properties or Library Properties dialog box.</td>
</tr>
</tbody>
</table>
After the SBT has created a makefile, you can modify the makefile in the following ways:

- With the SBT for Eclipse.
- With SBT commands from the Command Shell.

When modifying a makefile, the SBT preserves any previous nonconflicting modifications, regardless how those modifications were made.

After you modify a makefile with the Command Shell, in Eclipse you must right-click the project and click **Update linked resource** to keep the Eclipse project view in step with the makefile.

When the SBT for Eclipse modifies a makefile, it locks the makefile to prevent corruption by other processes. You cannot edit the makefile from the command line until the SBT has removed the lock.

If you want to exclude a resource (a file or a folder) from the makefile temporarily, without deleting it from the project, you can use the **Remove from Build** command. Right-click the resource and click **Remove from Build**. When a resource is excluded from the build, it does not appear in the makefile, and Eclipse ignores it. However, it is still visible in the Project Explorer, with a modified icon. To add the resource back into the build, right-click the resource and click **Add to Build**.

**Note:** Do not use the Eclipse **Exclude from build** command. With software projects, you must use the **Remove from Build** and **Add to Build** commands instead.

### Absolute Source Paths and Linked Resources

By default, the source files for an Eclipse project are stored under the project directory. If your project must incorporate source files outside the project directory, you can add them as linked resources.

An Eclipse linked resource can be either a file or a folder. With a linked folder, all source files in the folder and its subfolders are included in the build.

When you add a linked resource (file or folder) to your project, the SBT for Eclipse adds the file or folder to your makefile with an absolute path name. You might use a linked resource to refer to common source files in a fixed location. In this situation, you can move the project to a different directory without disturbing the common source file references.

A linked resource appears with a modified icon (green dot) in the Project Explorer, to distinguish it from source files and folders that are part of the project. You can use the Eclipse debugger to step into a linked source file, exactly as if it were part of the project.

You can reconfigure your project to refer to any linked resource either as an individual file, or through its parent folder. Right-click the linked resource and click **Update Linked Resource**.

You can use the **Remove from Build** and **Add to Build** commands with linked resources. When a linked resource is excluded from the build, its icon is modified with a white dot.

You can use Eclipse to create a path variable, defining the location of a linked resource. A path variable makes it easy to modify the location of one or more files in your project.

For information about working with path variables and creating linked resources, refer to the Eclipse help system.

### User Source Management

You can remove a makefile from source management control through the **Application Properties** or **Library Properties** dialog box.

Simply turn off **Enable source management** to convert the makefile to user source management. When **Enable source management** is off, you must update your makefile manually to add or remove source files to or from the project. The SBT for Eclipse makes no changes to the list of source files, but continues to manage all other project parameters and settings in the makefile.
Modifying a Makefile with User Source Management

Editing a makefile manually is an advanced technique. Intel FPGA recommends that you avoid manual editing. The SBT provides extensive capabilities for manipulating makefiles while ensuring makefile correctness.

In a makefile with user-managed sources, you can refer to source files with an absolute path. You might use an absolute path to refer to common source files in a fixed location. In this situation, you can move the project to a different directory without disturbing the common source file references.

Projects with user-managed sources do not support the following features:

- Linked resources
- The Add to Build command
- The Remove from Build command

Table 2-2: GUI Actions that Modify an Application or a Makefile with User Source Management

<table>
<thead>
<tr>
<th>Modification</th>
<th>Where Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specifying the application or user library name</td>
<td>Application Properties or Library Properties dialog box</td>
</tr>
<tr>
<td>Specifying a path to an associated BSP</td>
<td>Project References dialog box</td>
</tr>
<tr>
<td>Specifying a path to an associated user library</td>
<td>Project References dialog box</td>
</tr>
<tr>
<td>Enabling, disabling or modifying compiler options</td>
<td>Application Properties or Library Properties dialog box</td>
</tr>
</tbody>
</table>

Note: With user source management, the source files shown in the Eclipse Project Explorer view do not necessarily reflect the sources built by the makefile. To update the Project Explorer view to match the makefile, right-click the project and click Sync from Build.

BSP Source Management

BSP makefiles are handled differently from application and user library makefiles. BSP makefiles are based on the operating system, BSP settings, selected software packages, and selected drivers. You do not specify BSP source files directly.

BSP makefiles must be managed by the SBT, either through the BSP Editor or through the SBT command-line utilities.

Related Information
Using the BSP Editor on page 2-14
For more information about specifying BSPs

Using the BSP Editor

Typically, you create a BSP with the SBT for Eclipse. The plugins provide the basic tools and settings for defining your BSP. For more advanced BSP editing, use the BSP Editor. The BSP Editor provides all the tools you need to create even the most complex BSPs.
Tcl Scripting and the BSP Editor

The BSP Editor provides support for Tcl scripting. When you create a BSP in the BSP Editor, the editor can run a Tcl script that you specify to supply BSP settings.

You can also export a Tcl script from the BSP Editor, containing all the settings in an existing BSP. By studying such a script, you can learn about how BSP Tcl scripts are constructed.

Starting the BSP Editor

You start the BSP Editor in one of the following ways:

- Right-click an existing project, point to , and click BSP Editor. The editor loads the BSP Settings File (.bsp) associated with your project, and is ready to update it.
- On the menu, click BSP Editor. The editor starts without loading a .bsp file.
- Right-click an existing BSP project and click Properties. In the Properties dialog box, select BSP Properties > BSP Editor. The editor loads your .bsp file for update.

The BSP Editor Screen Layout

The BSP Editor screen is divided into two areas. The top area is the command area, and the bottom is the console area. The details of the BSP Editor screen areas are described in this section.

Below the console area is the Generate button. This button is enabled when the BSP settings are valid. It generates the BSP target files, as shown in the Target BSP Directory tab.

The Command Area

In the command area, you specify settings and other parameters defining the BSP. The command area contains several tabs:

- The Main tab
- The Software Packages tab
- The Drivers tab
- The Linker Script tab
- The Enable File Generation tab
- The Target BSP Directory tab

Each tab allows you to view and edit a particular aspect of the .bsp, along with relevant command line parameters and Tcl scripts.

The settings that appear on the Main, Software Packages and Drivers tabs are the same as the settings you manipulate on the command line.

Related Information

Software Build Tools Reference on page 15-1

The Main Tab

The Main tab presents general settings and parameters, and operating system settings, for the BSP. The BSP includes the following settings and parameters:

- The path to the .sopcinfo file specifying the target hardware
- The processor name
- The operating system and version

Note: You cannot change the operating system in an existing BSP. You must create a new BSP based on the desired operating system.
• The BSP target directory—the destination for files that the SBT copies and creates for your BSP.
• BSP settings

BSP settings appear in a tree structure. Settings are organized into Common and Advanced categories. Settings are further organized into functional groups. The available settings depend on the operating system.

When you select a group of settings, the controls for those settings appear in the pane to the right of the tree. When you select a single setting, the pane shows the setting control, the full setting name, and the setting description.

Related Information
• The Software Packages Tab on page 2-16
• The Drivers Tab on page 2-16
  For more information about how the software package and driver settings are presented separately.

The Software Packages Tab
The Software Packages tab allows you to insert and remove software packages in your BSP, and control software package settings.

At the top of the Software Packages tab is the software package table, listing each available software package. The table allows you to select the software package version, and enable or disable the software package.

The operating system determines which software packages are available.

Many software packages define settings that you can control in your BSP. When you enable a software package, the available settings appear in a tree structure, organized into Common and Advanced settings.

When you select a group of settings, the controls for those settings appear in the pane to the right of the tree. When you select a single setting, the pane shows the setting control, the full setting name, and the setting description.

Enabling and disabling software packages and editing software package settings can have a profound impact on BSP behavior. Refer to the documentation for the specific software package for details.

Related Information
• The Drivers Tab on page 2-16
  For more information about how the software package and driver settings are presented separately.
• The Main Tab on page 2-15
• Read-Only Zip File System on page 12-1
  For more information about the read-only zip file system.
• Ethernet and the NicheStack TCP/IP Stack on page 11-1

The Drivers Tab
The Drivers tab allows you to select, enable, and disable drivers for devices in your system, and control driver settings.

At the top of the Drivers tab is the driver table, mapping components in the hardware system to drivers. The driver table shows components with driver support. Each component has a module name, module version, module class name, driver name, and driver version, determined by the contents of the hardware system. The table allows you to select the driver by name and version, as well as to enable or disable each driver.
When you select a driver version, all instances of that driver in the BSP are set to the version you select. Only one version of a given driver can be used in an individual BSP.

Many drivers define settings that you can control in your BSP. Available driver settings appear in a tree structure below the driver table, organized into **Common** and **Advanced** settings.

When you select a group of settings, the controls for those settings appear in the pane to the right of the tree. When you select a single setting, the pane shows the setting control, the full setting name, and the setting description.

Enabling and disabling device drivers, changing drivers and driver versions, and editing driver settings, can have a profound impact on BSP behavior. Refer to the relevant component documentation and driver information for details.

**Related Information**
- The Software Packages Tab on page 2-16
- The Main Tab on page 2-15
- Embedded Peripherals IP User Guide
  For more information about Intel FPGA components.

**The Linker Script Tab**

The Linker Script tab allows you to view available memory in your hardware system, and examine and modify the arrangement and usage of linker regions in memory.

When you make a change to the memory configuration, the SBT validates your change.

**Note:** Rearranging linker regions and linker section mappings can have a very significant impact on BSP behavior.

**Related Information**
- The Problems Tab on page 2-19
  If there is a problem, a message appears in the Problems tab in the console area.

**Linker Section Mappings**

At the top of the Linker Script tab, the Linker Section Mappings table shows the mapping from linker sections to linker regions. You can edit the BSP linker section mappings using the following buttons located next to the linker section table:

- **Add**—Adds a linker section mapping to an existing linker region. The Add button opens the Add Section Mapping dialog box, where you specify a new section name and an existing linker region.
- **Remove**—Removes a mapping from a linker section to a linker region.
- **Restore Defaults**—Restores the section mappings to the default configuration set up at the time of BSP creation.

**Linker Regions**

At the bottom of the Linker Script tab, the Linker Memory Regions table shows all defined linker regions. Each row of the table shows one linker region, with its address range, memory device name, size, and offset into the selected memory device.
You reassign a defined linker region to a different memory device by selecting a different device name in the Memory Device Name column. The Size and Offset columns are editable. You can also edit the list of linker regions using the following buttons located next to the linker region table:

- **Add**—Adds a linker region in unused space on any existing device. The Add button opens the Add Memory Region dialog box, where you specify the memory device, the new memory region name, the region size, and the region’s offset from the device base address.
- **Remove**—Removes a linker region definition. Removing a region frees the region’s memory space to be used for other regions.
- **Add Memory Device**—Creates a linker region representing a memory device that is outside the hardware system. The button launches the Add Memory Device dialog box, where you can specify the device name, memory size and base address. After you add the device, it appears in the linker region table, the Memory Device Usage Table dialog box, and the Memory Map dialog box. This functionality is equivalent to the add_memory_device Tcl command.

**Note:** Ensure that you specify the correct base address and memory size. If the base address or size of an external memory changes, you must edit the BSP manually to match. The SBT does not automatically detect changes in external memory devices, even if you update the BSP by creating a new settings file.

- **Restore Defaults**—restores the memory regions to the default configuration set up at the time of BSP creation.
- **Memory Usage**—Opens the Memory Device Usage Table. The Memory Device Usage Table allows you to view memory device usage by defined memory region. As memory regions are added, removed, and adjusted, each device’s free memory, used memory, and percentage of available memory are updated. The rightmost column is a graphical representation of the device’s usage, according to the memory regions assigned to it.
- **Memory Map**—Opens the Memory Map dialog box. The memory map allows you to view a map of system memory in the processor address space. The Device table is a read-only reference showing memories in the hardware system that are mastered by the selected processor. Devices are listed in memory address order.

To the right of the Device table is a graphical representation of the processor’s memory space, showing the locations of devices in the table. Gaps indicate unmapped address space.

**Note:** This representation is not to scale.

**Related Information**

Software Build Tools Reference on page 15-1

**Enable File Generation Tab**

The Enable File Generation tab allows you to take ownership of specific BSP files that are normally generated by the SBT. When you take ownership of a BSP file, you can modify it, and prevent the SBT from overwriting your modifications. The Enable File Generation tab shows a tree view of all target files to be generated or copied when the BSP is generated. To disable generation of a specific file, expand the software component containing the file, expand any internal directory folders, select the file, and right-click. Each disabled file appears in a list at the bottom of the tab. This functionality is equivalent to the set_ignore_file Tcl command.

**Note:** If you take ownership of a BSP file, the SBT can no longer update it to reflect future changes in the underlying hardware. If you change the hardware, be sure to update the file manually.

**Related Information**

Software Build Tools Reference on page 15-1
Target BSP Directory Tab

The **Target BSP Directory** tab is a read-only reference showing you what output to expect when the BSP is generated.

It does not depict the actual file system, but rather the files and directories to be created or copied when the BSP is generated. Each software component, including the operating system, drivers, and software packages, specifies source code to be copied into the BSP target directory. The files are generated in the directory specified on the **Main** tab.

When you generate the BSP, existing BSP files are overwritten, unless you disable generation of the file in the **Enable File Generation** tab.

The Console Area

The console area shows results of settings and commands that you select in the command area. The console area consists of the following tabs:

- The **Information** tab
- The **Problems** tab
- The **Processing** tab

The Information Tab

The **Information** tab shows a running list of high-level changes you make to your BSP, such as adding a software package or changing a setting value.

The Problems Tab

The **Problems** tab shows warnings and errors that impact or prohibit BSP creation. For example, if you inadvertently specify an invalid linker section mapping, a message appears in the **Problems** tab.

The Processing Tab

When you generate your BSP, the **Processing** tab shows files and folders created and copied in the BSP target directory.

Exporting a Tcl Script

When you have configured your BSP to your satisfaction, you can export the BSP settings as a Tcl script. This feature allows you to perform the following tasks:

- Regenerate the BSP from the command line
- Recreate the BSP as a starting point for a new BSP
- Recreate the BSP on a different hardware platform
- Examine the Tcl script to improve your understanding of Tcl command usage

The exported Tcl script captures all BSP settings that you have changed since the previous time the BSP settings file was saved. If you export a Tcl script after creating a new BSP, the script captures all nondefault settings in the BSP. If you export a Tcl script after editing a pre-existing BSP, the script captures your changes from the current editing session.

To export a Tcl script, in the Tools menu, click **Export Tcl Script**, and specify a filename and destination path. The file extension is `.tcl`.

You can later run your exported script as a part of creating a new BSP.
Creating a New BSP

To create a BSP in the Nios II BSP Editor, use the New BSP command in the File menu to open the New BSP dialog box. This dialog box controls the creation of a new BSP settings file. The BSP Editor loads this new BSP after the file is created.

In this dialog box, you specify the following parameters:

- The .sopcinfo file defining the hardware platform.
- The CPU name of the targeted processor.
- The BSP type and version.

**Note:** You can select the operating system only at the time you create the BSP. To change operating systems, you must create a new BSP.

- The operating system version.
- The name of the BSP settings file. It is created with file extension .bsp.
- Absolute or relative path names in the BSP settings file. By default, relative paths are enabled for filenames in the BSP settings file.
- An optional Tcl script that you can run to supply additional settings.

Normally, you specify the path to your .sopcinfo file relative to the BSP directory. This enables you to move, copy and archive the hardware and software files together. If you browse to the .sopcinfo file, or specify an absolute path, the Nios II BSP Editor offers to convert your path to the relative form.

Using a Tcl Script in BSP Creation

When you create a BSP, the New BSP Settings File dialog box allows you to specify the path and filename of a Tcl script. The BSP Editor runs this script after all other BSP creation steps are done, to modify BSP settings.

This feature allows you to perform the following tasks:

- Recreate an existing BSP as a starting point for a new BSP
- Recreate a BSP on a different hardware platform
- Include custom settings common to a group of BSPs

The Tcl script can be created by hand or exported from another BSP.

**Related Information**

- **Exporting a Tcl Script** on page 2-19
  For more information about how to create a Tcl script from an existing BSP.
- **Software Build Tools** on page 4-1
  For more information about Tcl scripts and BSP settings, refer to "Tcl Scripts for BSP Settings".
BSP Validation Errors

If you modify a hardware system after basing a BSP on it, some BSP settings might no longer be valid. This is a very common cause of BSP validation errors. Eliminating these errors usually requires correcting a large number of interrelated settings.

If your modifications to the underlying hardware design result in BSP validation errors, the best practice is to update or recreate the BSP. Updating and recreating BSPs is very easy with the BSP Editor.

If you recreate your BSP, you might find it helpful to capture your old BSP settings by exporting them to a Tcl script. You can edit the Tcl script to remove any settings that are incompatible with the new hardware design.

Related Information

- **Using a Tcl Script in BSP Creation** on page 2-20
  For more information about how to run a Tcl script during BSP creation.
- **Exporting a Tcl Script** on page 2-19
  For more information about how to create a Tcl script from an existing BSP.
- **Software Build Tools** on page 4-1
  For more information about Tcl scripts and BSP settings, refer to "Tcl Scripts for BSP Settings".

Run Configurations in the SBT for Eclipse

Eclipse uses run configurations to control how it runs and debugs programs. Run configurations in the Nios II SBT for Eclipse have several features that help you debug Nios II software running on FPGA platforms.

Opening the Run Configuration Dialog Box

You can open the run configuration dialog box two ways:

- You can right-click an application, point to Run As, and click Run Configurations.
- You can right-click an application, point to Debug As, and click Debug Configurations.

Depending on which way you opened the run configuration dialog box, the title is either Run Configuration or Debug Configuration. However, both views show the same run configurations.

**Note:** If your project was created with version 10.1 or earlier of the Nios II SBT, you must re-import it to create the Nios II launch configuration correctly.

Each run configuration is presented on several tabs. This section describes each tab.

The Project Tab

On this tab, you specify the application project to run. The Advanced button opens the Nios II ELF Section Properties dialog box. In this dialog box, you can control the runtime parameters in the following ways:

- Specify the processor on which to execute the program (if the hardware design provides multiple processors)
- Specify the device to use for standard I/O
- Specify the expected location, timestamp and value of the system ID
- Specify the path to the JTAG Debugging Information File (.jdi)
- Enable or disable profiling
The Target Connection Tab

This tab allows you to control the connection between the host machine and the target hardware in the following ways:

- Select the cable, if more than one cable is available
- Allow software to run despite a system ID value or timestamp that differs from the hardware
- Reset the processor when the software is downloaded

The System ID Properties button allows you to examine the system ID and timestamp in both the .elf file and the hardware. This can be helpful when you need to analyze the cause of a system ID or timestamp mismatch.

The Debugger Tab

In this tab, you optionally enable the debugger to halt at a specified entry point.

Optimizing Project Build Time

When you build a project, the project makefile builds any components that are unbuilt or out of date. For this reason, the first time you build a project is normally the slowest. Subsequent builds are fast, only rebuilding sources that have changed.

With Gen 2, the Windows host compile and build time performance has improved. For example, it is now three times faster to build the webserver example.

Importing a Command-Line Project

If you have software projects that were created with the SBT command line, you can import the projects into the SBT for Eclipse for debugging and further development. This section discusses the import process.

Your command-line C/C++ application, and its associated BSP, is created on the command line. Any SBT command-line project is ready to import into the SBT for Eclipse. No additional preparation is necessary.

Nios II Command-Line Projects

The Nios II SBT for Eclipse imports the following kinds of Nios II command-line projects:

- Command-line C/C++ application project
- Command-line BSP project
- Command-line user library project

You can edit, build, debug, and manage the settings of an imported project exactly the same way you edit, build, debug, and manage the settings of a project created in Nios II SBT for Eclipse.

Importing through the Import Wizard

The SBT for Eclipse imports each type of project through the Import wizard. The Import wizard determines the kind of project you are importing, and configures it appropriately.
You can continue to develop project code in your SBT project after importing the project into Eclipse. You can edit source files and rebuild the project, using the SBT either in Eclipse or on the command line.

Related Information

Getting Started from the Command Line on page 3-1
For more information about creating projects with the command line.

Road Map

Importing and debugging a project typically involves several of the following tasks. You do not need to perform these tasks in this order, and you can repeat or omit some tasks, depending on your needs.

- Import a command-line C/C++ application
- Import a supporting project
- Debug a command-line C/C++ application
- Edit command-line C/C++ application code

When importing a project, the SBT for Eclipse might make some minor changes to your makefile. If the makefile refers to a source file located outside the project directory tree, the SBT for Eclipse treats that file as a linked resource. However, it does not add or remove any source files to or from your makefile.

When you import an application or user library project, the SBT for Eclipse allows you to choose Eclipse source management or user source management. Unless your project has an unusual directory structure, choose Eclipse source management, to allow the SBT for Eclipse to automatically maintain your list of source files.

You debug and edit an imported project exactly the same way you debug and edit a project created in Eclipse.

Import a Command-Line C/C++ Application

To import a command-line C/C++ application, perform the following steps:

1. Start the SBT for Eclipse.
2. On the File menu, click Import. The Import dialog box appears.
3. Expand the Software Build Tools Project folder, and select Import Software Build Tools Project.
4. Click Next. The File Import wizard appears.
5. Click Browse and locate the directory containing the C/C++ application project to import.
6. Click OK. The wizard fills in the project path.
7. Specify the project name in the Project name box.
   
   Note: You might see a warning saying "There is already a .project file at: <path>". This warning indicates that the directory already contains an Eclipse project. Either it is an Eclipse project, or it is a command-line project that is already imported into Eclipse. If the project is already in your workspace, do not re-import it.
8. Click Finish. The wizard imports the application project.

After you complete these steps, the SBT for Eclipse can build, debug, and run the complete program, including the BSP and any libraries. The SBT for Eclipse builds the project using the SBT makefiles in your imported C/C++ application project. Eclipse displays and steps through application source code exactly as if the project were created in the SBT for Eclipse. However, Eclipse does not have direct information about where BSP or user library code resides. If you need to view, debug or step through BSP or user library source code, you need to import the BSP or user library.
Importing a Project with Absolute Source Paths

If your project uses an absolute path to refer to a source file, the SBT for Eclipse imports that source file as a linked resource. In this case, the import wizard provides a page where you can manage how Eclipse refers to the source: as a file, or through a parent directory.

User-Managed Source Files

When you import a application or user library project, the SBT for Eclipse offers the option of user source management. User source management is helpful if you prefer to update your makefile manually to reflect source files added to or removed from the project.

With user source management, Eclipse never makes any changes to the list of source files in your makefile. However, the SBT for Eclipse manages all other project parameters and settings, just as with any other software project.

If your makefile refers to a source file with an absolute path, when you import with user source management, the absolute path is untouched, like any other source path. You might use an absolute path to refer to common source files in a fixed location. In this situation, you can move the project to a different directory without disturbing the common source file references.

User source management is not available with BSP projects. BSP makefiles are based on the operating system, BSP settings, selected software packages, and selected drivers. You do not specify BSP source files directly.

Importing a Supporting Project

While debugging a C/C++ application, you might need to view, debug or step through source code in a supporting project, such as a BSP or user library. To make supporting project source code visible in the Eclipse debug perspective, you need to import the supporting project.

If you do not need BSP or user library source code visible in the debugger, you can skip this task, and proceed to debug your project exactly as if you had created it in Eclipse.

If you have several C/C++ applications based on one BSP or user library, import the BSP or user library once, and then import each application that is based on the BSP or user library. Each application’s makefile contains the information needed to find and build any associated BSP or libraries.

User source management is not available with BSP projects. BSP makefiles are based on the operating system, BSP settings, selected software packages, and selected drivers. You do not specify BSP source files directly.

User-Managed Source Files

When you import a application or user library project, the SBT for Eclipse offers the option of user source management. User source management is helpful if you prefer to update your makefile manually to reflect source files added to or removed from the project.

With user source management, Eclipse never makes any changes to the list of source files in your makefile. However, the SBT for Eclipse manages all other project parameters and settings, just as with any other software project.

If your makefile refers to a source file with an absolute path, when you import with user source management, the absolute path is untouched, like any other source path. You might use an absolute path to refer to common source files in a fixed location. In this situation, you can move the project to a different directory without disturbing the common source file references.

User source management is not available with BSP projects. BSP makefiles are based on the operating system, BSP settings, selected software packages, and selected drivers. You do not specify BSP source files directly.
Packaging a Library for Reuse

This section shows how to create and use a library archive file (`.a`) in the Software Build Tools for Eclipse. This technique enables you to provide a library to another engineer or organization without providing the C source files. This process entails two tasks:

1. Create a user library
2. Create a application project based on the user library

Creating the User Library

To create a user library, perform the following steps:

1. In the File menu, point to New and click Library.
2. Type a project name, for example `test_lib`.
3. For Location, browse to the directory containing your library source files (`.c` and `.h`).
4. Click Finish.
5. Build the project to create the `.a` file (in this case `libtest_lib.a`).

Using the Library

To use the library in an application project, perform the following steps:

1. Create your application project.
2. To set the library path in the application project, right-click the project, and click Properties.
3. Expand Application Properties. In Application Paths, next to Application include directories, click Add and browse to the directory containing your library header files.
4. Next to Application library directories, click Add and browse to the directory containing your `.a` file.
5. Next to Library name, click Add and type the library project name you selected when you created your user library.
6. Click OK.
7. Build your application.

As this example shows, the `.c` source files are not required to build the application project. To hand off the library to another engineer or organization for reuse, you provide the following files:

- library archive file (`.a`)
- Software header files (`.h`)

Related Information

Creating a Project on page 2-3

Creating a Software Package

This section shows how you can build a custom library into a BSP as a software package. The software package can be linked to any BSP through the BSP Editor.

This section contains an example illustrating the steps necessary to include any software package into a Nios II BSP.
To create and exercise the example software package, perform the following steps:

1. Locate the ip directory in your Intel FPGA Complete Design Suite installation. For example, if the Intel FPGA Complete Design Suite version 14.1 is installed on the Windows operating system, the directory might be c:\altera\14.1\ip. Under the ip directory, create a directory for the software package. For simplicity, this section refers to this directory as <example package>.

2. In <example package>, create a subdirectory named EXAMPLE_SW_PACKAGE. In <example package>/EXAMPLE_SW_PACKAGE, create two subdirectories named inc and lib.

3. In <example package>/EXAMPLE_SW_PACKAGE/inc, create a new header file named example_sw_package.h containing the following code:

   /* Example Software Package */
   void example_sw_package(void);

4. In <example package>/EXAMPLE_SW_PACKAGE/lib, create a new C source file named example_sw_package.c containing the following code:

   /* Example Software Package */
   #include <stdio.h>
   #include "..\inc\example_sw_package.h"

   void example_sw_package(void)
   {
       printf ("Example Software Package. \n");
   }

5. In <example package>, create a new Tcl script file named example_sw_package_sw.tcl containing the following code:

   #
   # example_sw_package_sw.tcl
   #

   # Create a software package known as "example_sw_package"
   create_sw_package example_sw_package

   # The version of this software
   set_sw_property version 14.1

   # Location in generated BSP that sources should be copied into
   set_sw_property bsp_subdirectory Example_SW_Package

   #
   # Source file listings...
   #

   # C/C++ source files
   #add_sw_property c_source EXAMPLE_SW_PACKAGE/src/my_source.c

   # Include files
   add_sw_property include_source
6. In the SBT for Eclipse, create a Nios II application and BSP project based on the Hello World template. Set the application project name to `hello_example_sw_package`.

7. Create a new C file named `hello_example_sw_package.c` in the new application project containing the following code:

```c
/*
 * "Hello World" example.
 *
 * This example prints 'Hello from Nios II' to the STDOUT stream. It also
 * tests inclusion of a user software package.
 */

#include <stdio.h>
#include "example_sw_package.h"

int main()
{
    printf("Hello from Nios II\n");
    example_sw_package();
    return 0;
}
```

8. Delete `hello_world.c` from the `hello_example_sw_package` application project.

9. In the File menu, point to New and click Nios II Library.

10. Set the project name to `example_sw_package_library`.

11. For Location, browse to `<example package>\EXAMPLE_SW_PACKAGE\lib`.

   **Note:** Building the library here is required, because the resulting `.a` is referenced here by `example_sw_package_sw.tcl`.

12. Click Finish.
13. Build the example_sw_package_library project to create the libexample_sw_package_library.a library archive.

14. Right-click the BSP project, point to Nios II, and click BSP Editor to open the BSP Editor.

15. In the Software Packages tab, find example_sw_package in the software package table, and enable it.
   If there are any errors in a software package's *_sw.tcl file, such as an incorrect path that causes a file to not be found, the software package does not appear in the BSP Editor.

16. Click the Generate button to regenerate the BSP. On the File menu, click Save to save your changes to settings.bsp.

17. In the File menu, click Exit to exit the BSP Editor.

18. Build the hello_example_sw_package_bsp BSP project.

19. Build the hello_example_sw_package application project.

   hello_example_sw_package.elf is ready to download and execute.

Programming Flash in Intel FPGA Embedded Systems

Many Nios II processor systems use external flash memory to store one or more of the following items:

- Program code
- Program data
- FPGA configuration data
- File systems

Flash programmer tools allow you to program software, FPGA configuration, and application-specific binary data into flash memory devices. The tools support combining these different types of data so that they can be stored in a single flash device.

In , the Nios II SBT for Eclipse provides the Nios II flash programmer GUI-based tool and associated utilities, to help you manage and program the contents of flash memory. The sections below describe how to use these tools.

In , the Nios II Flash programmer has been deprecated and the Professional programmer should be used instead. You can access the Professional Programmer by performing the following steps:

1. Start the software. The Pro Edition window appears.

Starting the Flash Programmer

You start the flash programmer by clicking Flash Programmer in the Nios II menu.

When you first open the flash programmer, no controls are available until you open or create a Flash Programmer Settings File (.flash-settings).

Creating a Flash Programmer Settings File

The .flash-settings file describes how you set up the flash programmer GUI to program flash. This information includes the files to be programmed to flash, a .sopcinfo file describing the hardware configuration, and the file programming locations. You must create or open a flash programmer settings file before you can program flash.
You create a flash programmer settings file through the File menu. When you click New, the New Flash Programmer Settings File dialog box appears.

**Specifying the Hardware Configuration**

You specify the hardware configuration by opening a .sopcinfo file. You can locate the .sopcinfo file in either of two ways:

- Browse to a BSP settings file. The flash programmer finds the .sopcinfo file associated with the BSP.
- Browse directly to a .sopcinfo file.

Once you have identified a hardware configuration, details about the target hardware appear at the top of the Nios II flash programmer screen.

Also at the top of the Nios II flash programmer screen is the Hardware Connections button, which opens the Hardware Connections dialog box. This dialog box allows you to select a download cable, and control system ID behavior.

**Related Information**

The Target Connection Tab on page 2-22

**The Flash Programmer Screen Layout**

The flash programmer screen is divided into two areas. The top area is the command area, and the bottom is the console area. The details of the flash programmer screen areas are described in this section.

Below the console area is the Start button. This button is enabled when the flash programmer parameters are valid. It starts the process of programming flash.

**The Command Area**

In the command area, you specify settings and other parameters defining the flash programmer settings file. The command area contains one or more tabs. Each tab represents a flash memory component available in the target hardware. Each tab allows you to view the parameters of the memory component, and view and edit the list of files to be programmed in the component.

The Add and Remove buttons allow you to create and edit the list of files to be programmed in the flash memory component.

The File generation command box shows the commands used to generate the Motorola S-record Files (.flash) used to program flash memory.

The File programming command box shows the commands used to program the .flash files to flash memory.

The Properties button opens the Properties dialog box, which allows you to view and modify information about an individual file. In the case of a .elf, the Properties button provides access to the project reset address, the flash base and end addresses, and the boot loader file (if any).

The flash programmer determines whether a boot loader is required based on the load and run locations of the .text section. You can use the Properties dialog box to override the default boot loader configuration.
The Console Area

The console area shows results of settings and commands that you select in the command area. The console area consists of the following tabs:

- The Information tab
- The Problems tab
- The Processing tab

The Information Tab

The Information tab shows the high-level changes you make to your flash programmer settings file.

The Problems Tab

The Problems tab shows warnings and error messages about the process of flash programmer settings file creation.

The Processing Tab

When you program flash, the Processing tab shows the individual programming actions as they take place.

Saving a Flash Programmer Settings File

When you have finished configuring the input files, locations, and other settings for programming your project to flash, you can save the settings in a .flash-settings file. With a .flash-settings file, you can program the project again without reconfiguring the settings. You save a .flash-settings file through the File menu.

Flash Programmer Options

Through the Options menu, you can control several global aspects of flash programmer behavior, as described in this section.

Related Information

Nios II Flash Programmer User’s Guide
For more information about these features.

Staging Directories

Through the Staging Directories dialog box, you control where the flash programmer creates its script and .flash-settings files.

Generate Files

If you disable this option, the flash programmer does not generate programming files, but programs files already present in the directory. You might use this feature to reprogram a set of files that you have previously created.

Program Files

If you disable this option, the flash programmer generates the programming files and the script, but does not program flash. You can use the files later to program flash by turning off the Generate Files option.

Erase Flash Before Programming

When enabled, this option erases flash memory before programming.
Run From Reset After Programming

When enabled, this option resets and starts the Nios II processor after programming flash.

Creating Memory Initialization Files

Sometimes it is useful to generate memory initialization files. For example, to program your FPGA with a complete, running system, you must include the memory contents in your .sof file. In this configuration, the processor can boot directly from internal memory without downloading.

Creating a Hexadecimal (Intel-Format) File (.hex) is a necessary intermediate step in creating such a .sof file. The SBT for Eclipse can create .hex files and other memory initialization formats.

To generate correct memory initialization files, the SBT needs details about the physical memory configuration and the types of files required. Typically, this information is specified when the hardware system is generated.

Note: If your system contains a user-defined memory, you must specify these details manually.

Related Information
Generate Memory Initialization Files by the Legacy Method on page 2-31

Generate Memory Initialization Files

To generate memory initialization files, perform the following steps:

1. Right-click the application project.
2. Point to Make targets and click Build to open the Make Targets dialog box.
3. Select mem_init_generate.
4. Click Build. The makefile generates a separate file (or files) for each memory device. It also generates a IP File (.qip). The .qip file tells the software where to find the initialization files.
5. Add the .qip file to your project.
6. Recompile your project.

Generate Memory Initialization Files by the Legacy Method

To generate memory initialization files by the legacy method, perform the following steps:

1. Right-click the application project.
2. Point to Make targets and click Build to open the Make Targets dialog box.
3. Select mem_init_install.
4. Click Build. The makefile generates a separate file (or files) for each memory device. The makefile inserts the memory initialization files directly in the project directory for you.
5. Recompile your project.

Related Information
Hardware Reference
For information about working in the stand-alone flow.
Memory Initialization Files for User-Defined Memories

Generating memory initialization files requires detailed information about the physical memory devices, such as device names and data widths. Normally, the SBT extracts this information from the .sopcinfo file. However, in the case of a user-defined memory, the .sopcinfo file does not contain information about the data memory, which is outside the system. Therefore, you must provide this information manually.

You specify memory device information when you add the user-defined memory device to your BSP. The device information persists in the BSP settings file, allowing you to regenerate memory initialization files at any time, exactly as if the memory device were part of the hardware system.

Specify the memory device information in the Advanced tab of the Add Memory Device dialog box. Settings in this tab control makefile variables in mem_init.mk. On the Advanced tab, you can control the following memory characteristics:

- The physical memory width. The device's name in the hardware system.
- The memory initialization file parameter name. Every memory device can have an HDL parameter specifying the name of the initialization file. The ModelSim launch configuration overrides the HDL parameter to specify the memory initialization filename. When available, this method is preferred for setting the memory initialization filename.
- The Mem init filename parameter can be used in systems as an alternative method of specifying the memory initialization filename. The Mem init filename parameter directly overrides any filename specified in the HDL.
- Connectivity to processor master ports. These parameters are used when creating the linker script.
- The memory type: volatile, CFI flash or EPCS flash.
- Byte lanes.
- You can also enable and disable generation of the following memory initialization file types:
  - .hex file
  - .dat and .sym files
  - .flash file

Related Information

Publishing Component Information to Embedded Software on page 13-1
For more information about this parameter, refer to "Embedded Software Assignments".

Specifying the Memory Device Information in the Advanced Tab

Specify the memory device information in the Advanced tab of the Add Memory Device dialog box. Settings in this tab control makefile variables in mem_init.mk.

On the Advanced tab, you can control the following memory characteristics:

- The physical memory width.
- The device's name in the hardware system.
- The memory initialization file parameter name. Every memory device can have an HDL parameter specifying the name of the initialization file. The ModelSim launch configuration overrides the HDL parameter to specify the memory initialization filename. When available, this method is preferred for setting the memory initialization filename.
- The Mem init filename parameter can be used in systems as an alternative method of specifying the memory initialization filename. The Mem init filename parameter directly overrides any filename specified in the HDL.
- Connectivity to processor master ports. These parameters are used when creating the linker script.
• The memory type: volatile, CFI flash or EPCS flash.
• Byte lanes.
• You can also enable and disable generation of the following memory initialization file types:
  • .hex file
  • .dat and .sym files
  • .flash file

Related Information
Publishing Component Information to Embedded Software on page 13-1
For more information about this parameter, refer to "Embedded Software Assignments".

Running a System with ModelSim
You can run a program on hardware, such as an Intel FPGA development board, or you can run it in the ModelSim simulation environment.

Note: If your project was created with version 10.1 or earlier of the SBT, you must re-import it to create the launch configuration correctly.

Using ModelSim with an SOPC Builder-Generated System
If your hardware system was generated by SOPC Builder, running a software project in ModelSim is very similar to running it on hardware.

To run a software project in ModelSim, right-click on the application project name, point to Run As, and click ModelSim.

To debug a software project in ModelSim, right-click on the application project name, point to Debug As, and click ModelSim.

Related Information
Running the Project on Nios II Hardware on page 2-5

Using ModelSim with a System-Generated System
To run a system-generated system with ModelSim, you must first create a simulation model and test bench, and specify memory initialization files. You create your simulation model and test bench using the steps that apply to any design.


Related Information
Quartus Prime Standard Edition Handbook Volume 1: Design and Synthesis

Preparing your Software for ModelSim
Creating the software projects is nearly the same as when you run the project on hardware. To prepare your software for ModelSim simulation, perform the following steps:

1. Create your software project.
If you need to initialize a user-defined memory, you must take special steps to create memory initialization files correctly.

2. Build your software project.

3. Create a ModelSim launch configuration with the following steps:
   a. Right-click the application project name, point to Run As, and click Run Configurations. In the Run Configurations dialog box, select Nios II ModelSim, and click the New button.
   b. In the Main tab, ensure that the correct software project name and .elf file are selected.
   c. Click Apply to save the launch configuration.
   d. Click Close to close the dialog box.

   If you are simulating multiple processors, create a launch configuration for each processor, and create a launch group.

4. Open the run configuration you previously created. Click Run. The Nios II SBT for Eclipse performs a make mem_init_generate command to create memory initialization files, and launches ModelSim.

5. At the ModelSim command prompt, type ldr.

Related Information

- Creating a Project on page 2-3
- Building the Project on page 2-5
- Generate Memory Initialization Files by the Legacy Method on page 2-31
- Creating a Simple BSP on page 2-11

Potential Error Message

When you create the launch configuration, you might see the following error message:

SEVERE: The project location has not been set in the ELF section. You can manually override this setting in the launch configuration's ELF file 'Advanced' properties page.

Related Information

Creating a Simple BSP on page 2-11

GCC Tool Chain

The EDS version 14.0 is the first version of the cores. After 14.0, cores are used. The Nios II GNU toolchain has been upgraded, as well. The following GCC versions are available in the following EDS versions:

- GCC 4.8.3 in 14.1
- GCC 4.9.1 in 15.0

Note: When upgrading to a new tool chain, there are -specific changes and GCC changes and enhancements.

Related Information

- Porting to GCC 4.8
  For more information about how GNU also provides a porting guide to GCC 4.8 to document common issues.
- Porting to GCC 4.9
  For more information about how GNU also provides a porting guide to GCC 4.9 to document common issues.
 GCC Releases
For more information about full GCC release notes.

Specific Changes
- Use __buildin_custom_* instead of -mcustom-* or #pragma to reliably generate Floating Point Custom Instructions (FPCI), independent of compiler optimization level and command line flags.
- To use -mcustom-* or #pragma for Floating Point Custom Instructions (FPCI):
  - The -ffinite-math-only flag must be used to generate fmins and fmax FPCI
  - The optimization (non -O0 flag) must be used to generate fsqrts FPCI
  - Users implementing transcendental functions in hardware must use the -funsafe-math-optimizations flag to generate the FPCI for the transcendental functions fsins(), fcoss(), ftans(), fatans(), fexps(), flogs() and corresponding double-precision functions.
  - The Pragma format has changed from eg. #pragma custom_fadds 253 to #pragma GCC target("custom-fadds=253") and function attributes provide an alternative format __attribute__((target("custom-fadds=253"))).
  - Use the -mel/-meb flags instead of -EL/-EB for endian settings. Software Build Tool for Eclipse (SBTE) users must regenerate the BSP for this setting to take effect.
  - The -mreverse-bitfields flag and reverse_bitfields pragma are no longer supported.
  - The -fstack-check flag must be used instead of -mstack-check to enable stack checking.

GCC Changes and Enhancements
- The -Wa,-relax-all flag in nios2-elf-gcc GCC 4.7.3 supports function calls and programs exceeding the 256 MB limit.
- When used with optimization, inline assembly code with the asm operator needs to declare values imported from C and exported back to C, using the mechanisms described on the "Extended Asm - Assembler Instructions with C Expression Operands" page.
- Pre-standard C++ headers are not supported in GCC 4.7.3. Replace pre-standard C++ with standard C++ eg. #include <iostream.h>, cout, endl with #include <iostream>, std::cout and std::endl, respectively.
- The compile flag -Wl,--defsym foo=bar where bar is an undefined symbol, generates error at the linker level in GCC 4.7.3. GCC 4.1.2 does not include this check.

Related Information
- Software Build Tools on page 4-1
  For more information about the GCC toolchains.
- Getting Started from the Command Line on page 3-1
- Extended Asm - Assembler Instructions with C Expression Operands
- Intel FPGA Software Installation and Licensing Manual
  For more information about installing the Quartus Prime software.

Eclipse Usage Notes
The behavior of certain Eclipse and CDT features is modified by the SBT for Eclipse. If you attempt to use these features the same way you would with a non-project, you might have problems configuring or building your project. This section discusses such features.

If you launch the Software Build Tools for Eclipse from the command shell, you cannot pause execution of the application when debugging the application. Running the program in this way closes the GDB.
connection to the target, leaving the processor running. This is caused by running Eclipse from the Cygwin environment.

To ensure that the pause button works, launch the Nios II Software Build Tool for Eclipse either from or directly from your operating system's Start menu.

**Configuring Application and Library Properties**

To configure project properties specific to SBT application and library projects, use the **Application Properties** and **Library Properties** tabs of the **Properties** dialog box.

To open the appropriate properties tab, right-click the application or library project and click **Properties**. Depending on the project type, **Application Properties** or **Library Properties** tab appears in the list of tabs. Click the appropriate Properties tab to open it.

**Comparing the Application Properties and Library Properties tabs**

The **Application Properties** and **Library Properties** tabs are nearly identical. These tabs allow you to control the following project properties:

- The name of the target .elf file (application project only)
- The library name (library project only)
- A list of symbols to be defined in the makefile
- A list of symbols to be undefined in the makefile
- A list of assembler flags
- Warning level flags
- A list of user flags
- Generation of debug symbols
- Compiler optimization level
- Generation of object dump file (application project only)
- Source file management
- Path to associated BSP (required for application, optional for library)

**Configuring BSP Properties**

To configure BSP settings and properties, use the BSP Editor.

**Related Information**

- **Using the BSP Editor** on page 2-14
  For more information about the BSP Editor.
- **Using the BSP Editor** on page 2-14
  For more information about the BSP Editor.

**Exclude from Build Not Supported**

The **Exclude from Build** command is not supported. You must use the **Remove from Nios II Build** and **Add to Nios II Build** commands instead.

This behavior differs from the behavior of the Nios II SBT for Eclipse in version 9.1.

**Selecting the Correct Launch Configuration Type**

If you try to debug a software project as a CDT Local C/C++ Application launch configuration type, you see an error message, and the Debug perspective fails to open. This is expected CDT behavior in the
Eclipse platform. Local C/C++ Application is the launch configuration type for a standard CDT project. To invoke the plugins, you must use a launch configuration type.

**Note:** If your project was created with version 10.1 or earlier of the SBT, you must re-import it to create the launch configuration correctly.

**Target Connection Options**

The Nios II launch configurations offer the following Nios II-specific options in the **Target Connection** tab:

- Disable 'Nios II Console' view
- Ignore mismatched system ID
- Ignore mismatched system timestamp
- Download ELF to selected target system
- Start processor
- Reset the selected target system

**Renaming Projects**

To rename a project in the SBT for Eclipse, perform the following steps:

1. Right-click the project and click **Rename**.
2. Type the new project name.
3. Right-click the project and click **Refresh**.

If you neglect to refresh the project, you might see the following error message when you attempt to build it:

```
Resource <original_project_name> is out of sync with the system
```

**Running Shell Scripts from the SBT for Eclipse**

Many SBT utilities are implemented as shell scripts. You can use Eclipse external tools configurations to run shell scripts. However, you must ensure that the shell environment is set up correctly.

To run shell scripts from the SBT for Eclipse, execute the following steps:

1. Start the Command Shell.
2. Start the SBT for Eclipse by typing the following command:
   ```
eclipse-nios2
   ```
   You must start the SBT for Eclipse from the command line in both the Linux and Windows operating systems, to set up the correct shell environment.
3. From the Eclipse Run menu, select to **External Tools > External Tools Configurations**.
4. Create a new tools configuration, or open an existing tools configuration.
5. On the **Main** tab, set **Location** and **Argument**.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Location</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td><code>$(env_var:QUARTUS_ROOTDIR)\bin\cygwin\bin\sh.exe</code></td>
<td><code>-c &quot;&lt;script name&gt; &lt;script args&gt;&quot;</code></td>
</tr>
</tbody>
</table>
Table 2-4: Location and Argument Values Used to Run elf2hex --help from Eclipse

<table>
<thead>
<tr>
<th>Platform</th>
<th>Location</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>${env_var:QUARTUS_ROOTDIR}\bin\cygwin\bin\sh.exe</td>
<td>--c &quot;elf2hex --help&quot;</td>
</tr>
<tr>
<td>Linux</td>
<td>${env_var:SOPC_KIT_NIOS2}/bin/elf2hex</td>
<td>--help</td>
</tr>
</tbody>
</table>

6. On the Build tab, ensure that Build before launch and its related options are set appropriately.

By default, a new tools configuration builds all projects in your workspace before executing the command. This might not be the desired behavior.

7. Click Run. The command executes in the Command Shell, and the command output appears in the Eclipse Console tab.

Related Information
Getting Started from the Command Line on page 3-1

Must Use Build Configuration

Although Eclipse can support multiple build configurations, you must use the build configuration for projects.

Note: If your project was created with version 10.1 or earlier of the SBT, you must re-import it to create the launch configuration correctly.

CDT Limitations

The following tables describe the Eclipse CDT features not supported by the plugins. The features listed in the left column are supported by the Eclipse CDT plugins, but are not supported by plugins; and the right column lists alternative features supported by the plugins.

Table 2-5: New Project Wizard

<table>
<thead>
<tr>
<th>Unsupported CDT Feature</th>
<th>Alternative Nios II Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++</td>
<td>To create a new project, use one of the following wizards:</td>
</tr>
<tr>
<td>• C Project</td>
<td>• Application</td>
</tr>
<tr>
<td>• C++ Project</td>
<td>• Application and BSP from Template</td>
</tr>
<tr>
<td>• Convert to a C/C++ Project</td>
<td>• Board Support Package</td>
</tr>
<tr>
<td>• Source Folder</td>
<td>• Library</td>
</tr>
</tbody>
</table>
### Table 2-6: Build configurations

<table>
<thead>
<tr>
<th>Unsupported CDT Feature</th>
<th>Alternative Nios II Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Right-click project and point to <strong>Build Configurations</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Debugger</strong> tab</td>
<td></td>
</tr>
<tr>
<td><strong>Stop on startup</strong></td>
<td></td>
</tr>
</tbody>
</table>
| The plugins only support a single build configuration. This feature is supported only at the top of `main()`.

### Table 2-7: Exclude from Build (from version 10.0 onwards)

<table>
<thead>
<tr>
<th>Unsupported CDT Feature</th>
<th>Alternative Nios II Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Right-click source files</td>
<td></td>
</tr>
<tr>
<td>Use <strong>Remove from Build</strong> and <strong>Add to Build</strong>.</td>
<td></td>
</tr>
</tbody>
</table>

### Table 2-8: Project Properties

<table>
<thead>
<tr>
<th>Unsupported CDT Feature</th>
<th>Alternative Nios II Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>C/C++ Build</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Builder Settings</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Makefile generation</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Build location</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Behavior</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Build on resource save (Auto build)</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Build Variables</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Discovery Options</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Environment</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Settings</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Tool Chain Editor</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Current builder</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Used tools</strong></td>
<td></td>
</tr>
<tr>
<td>By default, the SBT generates makefiles automatically.</td>
<td></td>
</tr>
<tr>
<td>The build location is determined with the <strong>Application Properties</strong> or <strong>BSP Properties</strong> dialog box.</td>
<td></td>
</tr>
<tr>
<td>To change the toolchain, use the <strong>Current tool chain</strong> option.</td>
<td></td>
</tr>
</tbody>
</table>

| **C/C++ General**  |
| **Enable project specific settings**  |
| **Documentation tool comments**  |
| **Documentation**  |
| **File Types**  |
| **Indexer**  |
| **Build configuration for the indexer**  |
| **Language Mappings**  |
| **Paths and Symbols**  |
| The plugins only support a single build configuration.  |
| Use **Application Properties** and **Application Paths**.  |
Table 2-9: Window Preferences

<table>
<thead>
<tr>
<th>Unsupported CDT Feature</th>
<th>Alternative Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++</td>
<td></td>
</tr>
<tr>
<td>• Build scope</td>
<td>The plugins only support a single build configuration.</td>
</tr>
<tr>
<td>• Build project</td>
<td>The plugins only support a single build configuration.</td>
</tr>
<tr>
<td>configurations</td>
<td></td>
</tr>
<tr>
<td>• Build Variables</td>
<td></td>
</tr>
<tr>
<td>• Environment</td>
<td></td>
</tr>
<tr>
<td>• File Types</td>
<td></td>
</tr>
<tr>
<td>• Indexer</td>
<td></td>
</tr>
<tr>
<td>• Build configuration</td>
<td></td>
</tr>
<tr>
<td>for the indexer</td>
<td></td>
</tr>
<tr>
<td>• Language Mappings</td>
<td></td>
</tr>
<tr>
<td>• New CDT project wizard</td>
<td></td>
</tr>
</tbody>
</table>

Enhancements for Build Configurations in SBT and SBT for Eclipse

The SBT command line tools `nios2-app-update-make` and `nios2-lib-update-makefile` now support six new options specifically for handling build configurations, which are fully backwards compatible even if it is unused and omitted.

For SBT for Eclipse, a few GUI options are added:

- Dropdown combo box showing selected build config
- Button for managing build configs (add/remove/activate)

Build Configurations in SBT

Application and library `makefile` are enhanced to support multiple build configurations. There are new command line options in `nios2-app-update-makefile` and `nios2-lib-update-makefile` for creating, deleting and updating build configurations.

These command line options are:

Table 2-10: New Command Line Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--add-build-config &lt;config&gt;</code></td>
<td>Adds a new build configuration with the name <code>&lt;config&gt;</code>, initializes the new build configuration using an existing build configuration named <code>&lt;base&gt;</code>.</td>
</tr>
<tr>
<td><code>&lt;base&gt;</code> is optional and defaults to the active configuration. <code>&lt;base&gt;</code> is always ignored if only one build configuration is available.</td>
<td></td>
</tr>
<tr>
<td><code>--remove-build-config &lt;config&gt;</code></td>
<td>Removes an existing build configuration. No effect if only one build configuration is available.</td>
</tr>
<tr>
<td><code>--list-build-config &lt;config&gt;</code></td>
<td>Returns name of all build configurations. Returns empty string if only one build configuration is available.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>--get-active-build-config</code></td>
<td>Returns the name of active build configuration.</td>
</tr>
<tr>
<td></td>
<td>Returns empty string if only one build configuration is available.</td>
</tr>
<tr>
<td><code>--set-active-build-config</code></td>
<td>Set the build configuration named <code>&lt;config&gt;</code> active.</td>
</tr>
<tr>
<td><code>&lt;config&gt;</code></td>
<td>No effect if only one build configuration is available.</td>
</tr>
<tr>
<td><code>--build-config</code></td>
<td>Only use (read or modify) the build configuration named <code>&lt;config&gt;</code> but do not set it as the active build configuration.</td>
</tr>
<tr>
<td><code>&lt;config&gt;</code></td>
<td>No effect if only one build configuration is available.</td>
</tr>
</tbody>
</table>

**Note:** These new options are optional and can be used together with all existing `nios2-app-update-makefile` and `nios2-lib-update-makefile` command line options.

**Note:** The BSP makefile does not support multiple build configurations.

**Build Configurations in SBT for Eclipse**

Application and library projects are enhanced to support multiple build configurations. There are new GUI options available for creating, deleting, and updating build configurations.
The following shows how the application properties page looks with the new build configuration options highlighted in red:

**Figure 2-2: Application Properties**

![Application Properties Dialog box](image)

Clicking on the **Managed Configurations** button shows the following dialog for adding, removing, and activating build configurations:

**Figure 2-3: Managed Configurations**

![Managed Configurations Dialog box](image)
Note: The BSP project does not support multiple build configurations.
The Nios II Software Build Tools (SBT) allows you to construct a wide variety of complex embedded software systems using a command-line interface. From this interface, you can execute Software Built Tools command utilities, and use scripts (or other tools) to combine the command utilities in many useful ways.

Advantages of Command-Line Software Development

The SBT command line offers the following advantages over the SBT for Eclipse:

- 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 several Tcl scripts to control the creation of a board support package (BSP).
- You can use command line tools in a bash script to build several projects at once.

The SBT command-line interface is designed to work in the Command Shell.

Related Information

The Command Shell on page 3-4

Outline of the SBT Command-Line Interface

The SBT command-line interface consists of:

- Command-line utilities
- Command-line scripts
- Tcl commands
- Tcl scripts

These elements work together in the Command Shell to create software projects.

Utilities

The SBT command-line utilities enable you to create software projects. You can call these utilities from the command line or from a scripting language of your choice (such as perl or bash). On Windows, these utilities have a .exe extension. The SBT resides in the < EDS install path>/sdk2/bin directory.
For more information about the command-line utilities provided by the SBT, refer to “Intel FPGA-Provided Development Tools” in the Software Build Tools section.

Related Information
Software Build Tools on page 4-45

Scripts

Nios II SBT scripts implement complex behavior that extends the capabilities provided by the utilities.

<table>
<thead>
<tr>
<th>Command</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>nios2-bsp</td>
<td>Creates or updates a BSP</td>
</tr>
<tr>
<td>create-this-app</td>
<td>Creates a software example and builds it</td>
</tr>
<tr>
<td>create-this-bsp</td>
<td>Creates a BSP for a specific hardware design example and builds it</td>
</tr>
</tbody>
</table>

Note: There are create-this-app scripts for each software example and several create-this-bsp scripts for each hardware design example. For more information, refer to “Nios II Design Example Scripts” in the “Nios II Software Build Tools Reference” section.

Related Information
Design Example Scripts on page 15-23

nios2-bsp

Usage

nios2-bsp <bsp-type> <bsp-dir> [sopc] [override]...

Options

- <bsp-type>: hal or ucosii.
- <bsp-dir>: Path to the BSP directory.
- <sopc>: The path to the .sopcinfo file or its directory.
- <override>: Options to override defaults.

Description

The nios2-bsp script calls nios2-bsp-create-settings or nios2-bsp-update-settings to create or update a BSP settings file, and the nios2-bsp-generate-files command to create the BSP files. The Nios II Embedded Design Suite (EDS) supports the following BSP types:

- hal
- ucosii

BSP type names are case-insensitive.

This utility produces a BSP of <bsp-type> in <bsp-dir>. If the BSP does not exist, it is created. If the BSP already exists, it is updated to be consistent with the associated hardware system.
The default Tcl script is used to set the following system-dependent settings:

- stdio character device
- System timer device
- Default linker memory
- Boot loader status (enabled or disabled)

If the BSP already exists, `nios2-bsp` overwrites these system-dependent settings.

The default Tcl script is installed at `<Nios II EDS install path>/sdk2/bin/bsp-set-defaults.tcl`

When creating a new BSP, this utility runs `nios2-bsp-create-settings`, which creates `settings.bsp` in `<bsp-dir>`.

When updating an existing BSP, this utility runs `nios2-bsp-update-settings`, which updates `settings.bsp` in `<bsp-dir>`.

After creating or updating the `settings.bsp` file, this utility runs `nios2-bsp-generate-files`, which generates files in `<bsp-dir>`.

**Required arguments:**

- `<bsp-type>`: Specifies the type of BSP. This argument is ignored when updating a BSP. This argument is case-insensitive. The `nios2-bsp` script supports the following values of `<bsp-type>`:
  - `hal`
  - `ucosii`
- `<bsp-dir>`: Path to the BSP directory. Use "." to specify the current directory.

**Optional arguments:**

- `<sopc>`: The path name of the `.sopcinfo` file. Alternatively, specify a directory containing a `.sopcinfo` file. In the latter case, the tool finds a file with the extension `.sopcinfo`. This argument is ignored when updating a BSP. If you omit this argument, it defaults to the current directory.
- `<override>`: Options to override defaults. The `nios2-bsp` script passes most overrides to `nios2-bsp-create-settings` or `nios2-bsp-update-settings`. It also passes the `--silent`, `--verbose`, `--debug`, and `--log` options to `nios2-bsp-generate-files`. `nios2-bsp` passes the following overrides to the default Tcl script:
  - `--default_stdio <device>`|`none`|`DONT_CHANGE` Specifies stdio device.
  - `--default_sys_timer <device>`|`none`|`DONT_CHANGE` Specifies system timer device.
  - `--default_memory_regions` `DONT_CHANGE` Suppresses creation of new default memory regions when updating a BSP. Do not use this option when creating a new BSP.
  - `--default_sections_mapping <region>`|`DONT_CHANGE` Specifies the memory region for the default sections.
  - `--use_bootloader` `0`|`1`|`DONT_CHANGE` Specifies whether a boot loader is required.

On a preexisting BSP, the value `DONT_CHANGE` prevents associated settings from changing their current value.
create-this-app

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

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

create-this-bsp

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

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

Tcl Commands

Tcl commands are a crucial component of the SBT. Tcl commands allow you to exercise detailed control over BSP generation, as well as to define drivers and software packages.

Tcl Scripts

The SBT provides powerful Tcl scripting capabilities. In a Tcl script, you can query project settings, specify project settings conditionally, and incorporate the software project creation process in a scripted software development flow. The SBT uses Tcl scripting to customize your BSP according to your hardware and the settings you select. You can also write custom Tcl scripts for detailed control over the BSP.

The Command Shell

The Command Shell is a bash command-line environment initialized with the correct settings to run command-line tools. The Command Shell supports the GCC toolchain.

For more information about GCC toolchains, refer to "Intel FPGA-Provided Development Tools" in the "Software Build Tools" section.

Related Information

- Software Build Tools on page 4-45
- GNU Compiler Tool Chain on page 4-44
- Overview of Embedded Development on page 1-1

Starting the Nios II Command Shell

To open the Nios II Command Shell, perform the following steps, depending on your environment:
In the Windows operating system, on the Start menu, point to **Programs > Intel FPGA > Nios II EDS <version>**, and click **Nios II <version> Command Shell**.

In the Linux operating system, in a command shell, change directories to `<Nios II EDS install path>`, and type the command `nios2_command_shell.sh`.

**Auto-Executing a Command in the Nios II Command Shell**

In certain situations, you might need to run a command or a script automatically after the Nios II Command Shell is initialized. When you start the Nios II Command Shell environment, to automatically execute a command perform one of the following steps, depending on your environment:

- In the Windows operating system, execute the following command:
  
  `<Nios II EDS install path>/Nios II Command Shell.bat <command>`

- In the Linux operating system, execute the following command:
  
  `<Nios II EDS install path>/nios2_command_shell.sh <command>`

For example, in Windows, to run an automated build, you might execute the following command:

  `"<Nios II EDS install path>/Nios II Command Shell.bat" custom_build.sh`

The Nios II Command Shell startup script (Nios II Command Shell.bat or nios2_command_shell.sh) makes no special assumptions about its initial environment. You can use the Nios II Command Shell with auto-execution from any environment that accepts commands native to your host operating system. For example, in Linux you can use `crontab` to schedule a job to run in the Nios II Command Shell at a later time.

**Getting Started in the SBT Command Line**

Using the Nios II SBT on the command line is the best way to learn about it. The following tutorial guides you through the process of creating, building, running, and debugging a “Hello World” program with a minimal number of steps. Later chapters provide more of the underlying details, allowing you to take more control of the process. The goal of this chapter is to show you that the basic process is simple and straightforward.

The Nios II SBT includes a number of scripts that demonstrate how to combine command utilities to obtain the results you need. This tutorial uses a **create-this-app** script as an example.

**Prerequisites**

To complete this tutorial, you must have the following:

- Intel FPGA Quartus Prime development software, version 8.0 or later. The software must be installed on a Windows or Linux computer that meets the Quartus Prime minimum requirements.
- The Intel FPGA Nios II Embedded Design Suite (EDS), version 8.0 or later.
- An Intel FPGA development board.
- A download cable such as the Intel FPGA USB-Blaster™ cable.

You run the Nios II SBT commands from the Nios II Command Shell.

**Related Information**

*The Command Shell* on page 3-4
Creating Hello_World for an Intel FPGA Development Board

In this section you create a simple “Hello World” project. To create and build the hello_world example for an Intel FPGA development board, perform the following steps:

1. Start the Command Shell.\(^{(2)}\)
2. Create a working directory for your hardware and software projects. The following steps refer to this directory as <projects>.
3. Change to the <projects> directory by typing the following command:
   \[cd \texttt{<projects>}\]
4. Locate a hardware example for your Intel FPGA development board. For example, if you have a Stratix® IV GX FPGA Development Kit, you might select < EDS install path>/examples/verilog/niosII_stratixIV_4sgx230/triple_speed_ethernet_design.
5. Copy the hardware example to your <projects> working directory, using a command such as the following:
   \[cp -R /altera/100/nios2eds/examples/verilog/niosII_stratixIV_4sgx230/triple_speed_ethernet_design .\]
6. Ensure that the working directory and all subdirectories are writable by typing the following command:
   \[chmod -R +w\]
7. The <projects> directory contains a subdirectory named software_examples/app/hello_world. The following steps refer to this directory as <application>.
8. Change to the <application> directory by typing the following command:
   \[cd \texttt{<application>}\]
9. Type the following command to create and build the application:
   \[./create-this-app\]

The create-this-app script copies the application source code to the <application> directory, runs nios2-app-generate-makefile to create a makefile (named Makefile), and then runs make to create an Executable and Linking Format File (.elf). The create-this-app script finds a compatible BSP by looking in <projects>/software_examples/bsp. In the case of hello_world, it selects the hal_default BSP.

To create the example BSP, create-this-app calls the create-this-bsp script in the BSP directory.

Related Information
The Command Shell on page 3-4

Running Hello_World on an Intel FPGA Development Board

To run the hello_world example on an Intel FPGA development board, perform the following steps:

1. Start the Command Shell.
2. Download the SRAM Object File (.sof) for the Quartus Prime project to the Intel FPGA development board.
   \[./create-this-app\]

This step configures the FPGA on the Intel FPGA development board.

\(^{(2)}\) For more information, refer to the "The Command Shell" chapter.
Note: The .sof file resides in <projects>, along with your Project File (.qpf). You download it by typing the following commands:

- cd <projects>
- nios2-configure-sof

The board is configured and ready to run the project's executable code.

The nios2-configure-sof utility runs the Programmer to download the .sof file. You can also run the quartus_pgm command directly.

For more information about programming the hardware, refer to the Hardware Development Tutorial.

3. Start another command shell displaying both command shells on your desktop.
4. In the second command shell, run the terminal application to connect to the Intel FPGA development board through the JTAG UART port by typing the nios2-terminal command.
5. Return to the original command shell and ensure that <projects>/software_examples/app/hello_world is the current working directory.
6. Download and run the hello_world executable program using the nios2-download -g hello_world.elf command.
   Hello from Nios II! appears in the second command shell.

Related Information
- Nios II Hardware Development Tutorial
- Nios II Hardware Development Tutorial

Debugging hello_world
An integrated development environment is the most powerful environment for debugging a software project. You debug a command-line project by importing it to the Nios II SBT for Eclipse. After you import the project, Eclipse uses your makefiles to build the project. This two-step process combines the advantages of the SBT command line development flow with the convenience of a GUI debugger.

This section discusses the process of importing and debugging the hello_world application.

Import the hello_world Application
To import the hello_world application, perform the following steps:

1. Launch the Nios II SBT for Eclipse.
2. On the File menu, click Import. The Import dialog box appears.
3. Expand the Nios II Project folder, and select Import Nios II project.
4. Click Next. The File Import wizard appears.
5. Click Browse and navigate to the <application> directory, containing the hello_world application project.
6. Click OK. The wizard fills in the project path.
7. Type the project name hello_world in the Project name box.
8. Click Finish. The wizard imports the application project.

Note: If you want to view the BSP source files while debugging, you also need to import the BSP project into the Nios II SBT for Eclipse.
Download Executable Code and Start the Debugger

1. Right-click the hello_world project, point to Debug As, and click Hardware.
2. If the Confirm Perspective Switch dialog box appears, click Yes.

   After a moment, the main() function appears in the editor. There is a blue arrow next to the first line of code, indicating that execution has stopped on this line.

**Note:** When targeting hardware, the Debug As command does the following tasks:

- Creates a default debug configuration for the target board
- Establishes communication with the target board
- Downloads the .elf file to memory on the target board
- Sets a breakpoint at main()
- Instructs the processor to begin executing the code

3. In the Run menu, click Resume to resume execution. You can also resume execution by pressing F8.

   **Note:** When debugging a project in Eclipse, you can also pause, stop, and single-step the program, set breakpoints, examine variables, and perform many other common debugging tasks.

Related Information

- [Importing a Command-Line Project](#) on page 2-22
  For more information about debugging projects in the Nios II SBT for Eclipse.
- [Getting Started with the Graphical User Interface](#) on page 2-1
  For more information about debugging projects in the Nios II SBT for Eclipse, refer to “Getting Started with Eclipse”.

Software Build Tools Scripting Basics

This section provides an example to teach you how you can create a software application using a command line script.

In this section, assume that you want to build a software application for a Nios II system that features the LAN91C111 10/100 Non-PCI Ethernet Single Chip MAC + PHY component and supports the NicheStack® TCP/IP stack. Furthermore, assume that you have organized the hardware design files and the software source files.
Creating a BSP with a Script

A simple method for creating a BSP is to use the `nios2-bsp` script as in the following example:

```bash
nios2-bsp ucosii ../SOPC/ --cmd enable_sw_package altera_iniche \
    --set altera_iniche.iniche_default_if lan91c111
nios2-bsp lwhal ../user/data/FastNetProject/FastNetHW/
make
```

Table 3-1: Description of nios2-bsp Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Purpose</th>
<th>Further Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>ucosii</td>
<td>Sets the operating system to MicroC/OS-II</td>
<td>For more information, refer to “Settings Managed by the Software Build Tools”.</td>
</tr>
<tr>
<td>.</td>
<td>Specifies the directory in which the BSP is to be created</td>
<td>—</td>
</tr>
<tr>
<td>../SOPC/</td>
<td>Points to the location of the hardware project</td>
<td>—</td>
</tr>
<tr>
<td>Argument</td>
<td>Purpose</td>
<td>Further Information</td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
<td>---------------------</td>
</tr>
<tr>
<td>--cmd enable_sw_package altera_iniche</td>
<td>Adds the NicheStack TCP/IP stack software package to the BSP</td>
<td>For more information, refer to “Settings Managed by the Software Build Tools”. For more information, refer to &quot;Software Build Tools Tcl Commands&quot;.</td>
</tr>
<tr>
<td>--set altera_iniche.iniche_default_if lan91c111</td>
<td>Specifies the default hardware interface for the NicheStack TCP/IP Stack - Nios II Edition</td>
<td>For more information, refer to “Settings Managed by the Software Build Tools”.</td>
</tr>
</tbody>
</table>

The **nios2-bsp** script uses the .sopcinfo file to create the BSP files. You can override default settings chosen by **nios2-bsp** by supplying command-line arguments, Tcl scripts, or both.

**Figure 3-2: nios2-bsp Command Flow**
Creating an Application Project with a Script

To create application projects, use **nios2-app-generate-makefile** as in the following example:

```
nios2-app-generate-makefile --bsp-dir ../BSP \ 
--elf-name telnet-test.elf \ 
--src-dir source/ make
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>--bsp-dir ../BSP</td>
<td>Specifies the location of the BSP on which this application is based</td>
</tr>
<tr>
<td>--elf-name telnet-test.elf</td>
<td>Specifies the name of the executable file</td>
</tr>
<tr>
<td>--src-dir source/</td>
<td>Tells <strong>nios2-app-generate-makefile</strong> where to find the C source files</td>
</tr>
</tbody>
</table>
Running make

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

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

```
make clean download-elf
```

You can modify an application or user library makefile with the *nios2-lib-update-makefile* and *nios2-app-update-makefile* utilities. With these utilities, you can execute the following tasks:

- Add source files to a project
- Remove source files from a project
- Add compiler options to a project's make rules
- Modify or remove compiler options in a project's make rules

Creating Memory Initialization Files

To create memory initialization files for a Nios II system, you can use the Command Shell. Change to the software application folder, and type:

```
make mem_init_generate
```

This command creates the memory initialization and simulation files for all memory devices. It also generates a Quartus Prime IP File (*.qip*). The *.qip* file tells the Quartus Prime software where to find the initialization files. Add the *.qip* file to your Quartus Prime project.
This chapter describes the Nios II Software Build Tools (SBT), a set of utilities and scripts that creates and builds embedded C/C++ application projects, user library projects, and board support packages (BSPs). The SBT supports a repeatable, scriptable, and archivable process for creating your software product.

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

• The EclipseGUI
• The Command Shell

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

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

• "Getting Started with the Graphical User Interface"
• "Getting Started from the Command Line"

This chapter assumes you are familiar with the following topics:

• The GNU make utility. Intel FPGA recommends you use version 3.80 or later. On the Windows platform, GNU make version 3.80 is provided with the EDS.

You can obtain general information about GNU make from the Free Software Foundation, Inc. website.

• Board support packages.

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

• Micrium MicroC/OS-II.

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

• Tcl scripting language.

Related Information

• Overview of Embedded Development on page 1-1
• Getting Started with the Graphical User Interface on page 2-1
• Getting Started from the Command Line on page 3-1
Road Map for the SBT

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

What the Build Tools Create

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

The SBT creates the following types of projects:

- application—A program implementing some desired functionality, such as control or signal processing.
- BSP—A library providing access to hardware in the system, such as UARTs and other I/O devices. A BSP provides a software runtime environment customized for one processor in a hardware system. A BSP optionally also includes the operating system, and other basic system software packages such as communications protocol stacks.
- User library—A library implementing a collection of reusable functions, such as graphics algorithms.

Comparing the Command Line with Eclipse

Aside from the Eclipse GUI, there are very few differences between the SBT command line and the SBT for Eclipse.

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

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

Makefiles

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

- Application or user library makefile—A simple makefile that builds the application or user library with user-provided source files
- BSP makefile—A more complex makefile, generated to conform to user-specified settings and the requirements of the target hardware system

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

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

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

### Modifying Makefiles

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

For more information, refer to the *Getting Started with the Graphical User Interface* section.

### Table 4-2: Command-Line Utilities for Updating Makefiles

<table>
<thead>
<tr>
<th>Project Type</th>
<th>Utilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>nios2-app-update-makefile</td>
</tr>
<tr>
<td>Library</td>
<td>nios2-lib-update-makefile</td>
</tr>
<tr>
<td>BSP</td>
<td>nios2-bsp-update-settings</td>
</tr>
<tr>
<td></td>
<td>nios2-bsp-generate-files</td>
</tr>
</tbody>
</table>

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

**Related Information**

- *Getting Started with the Graphical User Interface* on page 2-1
- *Updating Your BSP* on page 4-32

### Makefile Targets

Intel FPGA recommends that you study the generated makefiles for further details about the application makefile targets.

---

(3) For more information about updating BSP makefiles, refer to “Updating Your BSP” on page 4–30.
Table 4-3: Application Makefile Targets

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

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

**Embedded Software Projects**

The SBT supports the following kinds of software projects:

- C/C++ application projects
- C/C++ user library projects
- BSP projects

This section discusses each type of project in detail.

**Applications and Libraries**

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

The private makefile builds one of two types of files:

- A .elf file—For an application
- A library archive file (.a)—For a user library
For a user library, the SBT also generates a public makefile, called `public.mk`. The public makefile is included in the private makefile for any application (or other user library) that uses the user library.

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

**Supported Source File Types**

The SBT examines the extension of each source file to determine the programming language.

**Table 4-4: Supported Programming Languages with the Corresponding File Extensions**

<table>
<thead>
<tr>
<th>Programming Language</th>
<th>File Extensions(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>.c</td>
</tr>
<tr>
<td>C++</td>
<td>.cpp, .cx</td>
</tr>
<tr>
<td>assembly language; sources are built directly by the assembler without preprocessing</td>
<td>.s</td>
</tr>
<tr>
<td>assembly language; sources are preprocessed by the C preprocessor, allowing you to include header files</td>
<td>.S</td>
</tr>
</tbody>
</table>

**Board Support Packages**

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

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

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

**Overview of BSP Creation**

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

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

---

(4) All file extensions are case-sensitive.
Parts of a BSP

Hardware Abstraction Layer

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

For more information about the HAL, refer to the "HAL API Reference" section.

Related Information
HAL API Reference on page 14-1

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

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

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

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

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

- A software package is not associated with specific hardware.
- A software package might have settings that impact its compilation. These settings become part of the BSP settings.

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

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

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

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

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

**Software Build Process**

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

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

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

   For more information about available BSP settings, refer to the "Software Build Tools Reference" chapter.

3. Define a BSP. Use the SBT to specify the components in the BSP, and the values of any relevant settings. The result of this step is a BSP settings file, called **settings.bsp**.

   For more information about creating BSPs, refer to the "Board Support Packages" chapter.

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

5. Optionally create a user library. If you need to include a custom software user library, you collect the user library source files in a single directory, and create a user library makefile. The Nios II build tools can create a makefile for you. You can also create a makefile by hand, or you can autogenerate a makefile and then customize it by hand.

   For more information about creating user library projects, refer to the "Applications and Libraries" chapter.

6. Collect your application source code. When you are learning, this might be a software example. When you are developing a product, it is probably a collection of C/C++ source files developed by someone in your organization.

   For more information about creating application projects, refer to the "Applications and Libraries" chapter.

7. Create an application makefile. The easiest approach is to let the build tools create the makefile for you. You can also create a makefile by hand, or you can autogenerate a makefile and then customize it by hand.

   For more information about creating makefiles, refer to the "Makefiles" chapter.

**Note**: You must use the correct directory path for fetching the respective files from your project folder.

**Related Information**

- [Applications and Libraries](#) on page 4-4
- [Makefiles](#) on page 4-2
- [Board Support Packages](#) on page 4-5
- [Software Build Tools Reference](#) on page 15-1
Common BSP Tasks

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

Although this section describes tasks in terms of the SBT command line flow, you can also carry out most of these tasks with the SBT for Eclipse.

Related Information

- Using Version Control on page 4-8
- Copying, Moving, or Renaming a BSP on page 4-9
- Handing Off a BSP on page 4-10
- Changing the Default Linker Memory Region on page 4-14
- Changing a Linker Section Mapping on page 4-15
- Managing Device Drivers on page 4-16
- Creating a Custom Version of newlib on page 4-17
- Creating a BSP for an Intel FPGA Development Board on page 4-15
- Creating Memory Initialization Files on page 4-10
- Modifying Linker Memory Regions on page 4-11
- Creating a Custom Linker Section on page 4-12
- Querying Settings on page 4-16
- Controlling the stdio Device on page 4-17
- Configuring Optimization and Debugger Options on page 4-18
- Getting Started with the Graphical User Interface on page 2-1

For more information about carrying out the tasks with the Nios II SBT for Eclipse, refer to the "Getting Started with the Graphical User Interface" chapter.

Adding the SBT to Your Tool Flow

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

Using Version Control

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

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

Creating BSP by Running a User Defined Script to Call nios2-bsp

If you create a BSP by running your own script that calls nios2-bsp, you can put your script under version control. If your script provides any Tcl scripts to nios2-bsp (using the --script option), you must also put these Tcl scripts under version control. If you install a new release of EDS and run your script to create a new BSP or to update an existing BSP, the internal implementation of your BSP might change slightly due to improvements in EDS.
For more information, refer to “Revising Your BSP” on page 4–28 for a discussion of BSP regeneration with EDS updates.

Related Information
Revising Your BSP on page 4-29

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

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

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

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

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

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

For more information about .d files, refer to the GNU make documentation, available from the Free Software Foundation, Inc. website.

Another way to copy a BSP is to run the nios2-bsp-generate-files command to populate a BSP directory and pass it the path to the BSP settings file of the BSP that you wish to copy.
If you rename or move a BSP, you must manually revise any references to the BSP name or location in application or user library makefiles.

Related Information

- GNU Website
  For more information about .d files.
- Nios II Embedded Design Suite Support

Handing Off a BSP

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

Linking and Locating

When auto-generating a HAL BSP, the SBT makes some reasonable assumptions about how you want to use memory.

For more information, refer to “Specifying the Default Memory Map” on page 4–35.

However, in some cases these assumptions might not work for you. For example, you might implement a custom boot configuration that requires a bootloader in a specific location; or you might want to specify which memory device contains your interrupt service routines (ISRs).

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

Related Information

Specifying the Default Memory Map on page 4-37

Creating Memory Initialization Files

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

Although the application makefile provides the mem_init.mk targets, it does not build any of them by default. The SBT creates the memory initialization files in the application directory (under a directory named mem_init). The SBT optionally copies them to your Quartus Prime project directory and HDL simulation directory.

Note: The SBT does not generate a definition of QUARTUS_PROJECT_DIR in your application makefile.

If you have an on-chip RAM, and require that a compiled software image be inserted in your SRAM Object File (.sof) at Quartus Prime compilation, you must manually specify the value of
QUARTUS_PROJECT_DIR in your application makefile. You must define QUARTUS_PROJECT_DIR before the mem_init.mk file is included in the application makefile, as in the following example:

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

<table>
<thead>
<tr>
<th>Table 4-5: mem_init.mk Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Target</strong></td>
</tr>
<tr>
<td>mem_init_install</td>
</tr>
<tr>
<td>mem_init_generate</td>
</tr>
<tr>
<td>mem_init_clean</td>
</tr>
<tr>
<td>.hex</td>
</tr>
<tr>
<td>.dat</td>
</tr>
<tr>
<td>.sym</td>
</tr>
<tr>
<td>.flash</td>
</tr>
<tr>
<td>&lt;memory name&gt;</td>
</tr>
</tbody>
</table>

### Modifying Linker Memory Regions

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

Suppose you have a memory region named onchip_ram. The Tcl script named reserve_1024_onchip_ram.tcl 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 "Software Build Tools Reference" chapter.

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

# Create two memory regions out of the original region.
add_memory_region onchip_ram $slave_desc $offset $new_span
add_memory_region onchip_special $slave_desc $split_offset $split_span

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

nios2-bsp hal my_bsp --script reserve_1024_onchip_ram.tcl

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

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

nios2-bsp hal my_bsp --default_memory_regions DONT_CHANGE

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

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

Related Information

- Calling a Custom BSP Tcl Script on page 4-26
- Software Build Tools Reference on page 15-1
- Software Build Tools Reference on page 15-1
  For an explanation of each Tcl command used in this example, refer to the "Nios II Software Build Tools Reference" chapter.

Creating a Custom Linker Section

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

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

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

Creating a Linker Section for an Existing Region

To create your own section named special_section that is mapped to the linker region named onchip_special, use the following command to run nios2-bsp:
When the `nios2-bsp-generate-files` utility (called by `nios2-bsp`) generates the linker script `linker.x`, the linker script has a new section mapping. The order of section mappings in the linker script is determined by the order in which the `add_section_mapping` command creates the sections. If you use `nios2-bsp`, the default Tcl script runs before the `--cmd` option that creates the `special_section` section.

If you run `nios2-bsp` again to update your BSP, you do not need to provide the `add_section_mapping` command again because the default Tcl script only modifies section mappings for the default sections listed in the Default Section Names table.

### Dividing a Linker Region to Create a New Region and Section

This example works with any hardware design containing an on-chip memory named `tightly_coupled_instruction_memory` connected to a instruction master.

**Example 4–2. To Create a Section named .isrs in the tightly_coupled_instruction_memory on-chip memory**

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

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

### Using the Create a New Region and Section Tcl Script

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

1. Create the Tcl script as shown in the example, above.
2. Edit your `create-this-bsp` script, and add the following argument to the `nios2-bsp` command line:
   ```bash
tool --script <script name>.tcl
   ``
3. In the BSP project, edit `timer_interrupt_latency.h`. In the `timer_interrupt_latency_irq()` function, change the `.section` directive from `.exceptions` to `.isrs`.
4. Rebuild the application by running `make`.

### Excerpts from Object Dump Files

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

Example 4–3. Excerpts from Object Dump File

```
Sections:
Idx Name Size VMA LMA File off Algn
.
.
6 .isrs 000000c0 04000c00 04000c00 000000b4 2**2
CONTENTS, ALLOC, LOAD, READONLY, CODE
.
.
9 .tightly_coupled_instruction_memory 00000000 04000000 04000000 00013778 2**0
CONTENTS
.
.
SYMBOL TABLE:
00000000 l d .entry 00000000
30000020 l d .exceptions 00000000
30000150 l d .text 00000000
30010e14 l d .rodata 00000000
30011788 l d .rwdata 00000000
30013624 l d .bss 00000000
04000c00 l d .isrs 00000000
00000020 l d .ext_flash 00000000
03200000 l d .epcs_controller 00000000
04000000 l d .tightly_coupled_instruction_memory 00000000
04004000 l d .tightly_coupled_data_memory 00000000
.
.
```

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 tightly-coupled instruction memory, adjacent to the tightly_coupled_instruction_memory region.

Example 4–4. Excerpt From linker.x

```
MEMORY
{
    reset : ORIGIN = 0x0, LENGTH = 32
    tightly_coupled_instruction_memory : ORIGIN = 0x40000000, LENGTH = 3072
    isrs_region : ORIGIN = 0x4000c00, LENGTH = 1024
    .
    .
}
```

Changing the Default Linker Memory Region

The default Tcl script chooses the largest memory region connected to your processor as the default region.
For more information about all default memory sections mapped to this default region, refer to the previous chapter, "Creating a Custom Linker Section".

You can pass in a command-line option to the default Tcl script to override this default mapping. To map all default sections to onchip_ram, type the following command:

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

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

**Related Information**

Creating a Custom Linker Section on page 4-12

**Changing a Linker Section Mapping**

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

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

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

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

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

**Other BSP Tasks**

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

**Creating a BSP for an Intel FPGA Development Board**

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

1. Start the Command Shell.
   
   For details about the Command Shell, refer to the "Getting Started from the Command Line" chapter.
2. Create a working directory for your hardware and software projects. The following steps refer to this directory as <projects>.
3. Make <projects> the current working directory.
4. Find a hardware example corresponding to your Intel FPGA development board. For example, if you have a Stratix® IV development board, you might select < EDS install path>/examples/verilog/niosII_stratixIV_4sgx230/triple_speed_ethernet_design.
5. Copy the hardware example to your working directory, using a command such as the following:

```bash
cp -R /altera/100/nios2eds/examples/verilog/
    /niosII_stratixIV_4sgx230/triple_speed_ethernet_design
```

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

```bash
chmod -R +w
```

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

For a description of the example BSPs, refer to “Design Example Scripts” in the Software Build Tools Reference section.

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

```bash
./create-this-bsp
```

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

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

**Related Information**

- [Overview of Embedded Development](#) on page 1-1
- [Software Build Tools Reference](#) on page 15-1
  
  For a description of the example BSPs, refer to “Nios II Design Example Scripts” chapter.
- [Getting Started from the Command Line](#) on page 3-1
  
  For details about the Nios II Command Shell.

**Querying Settings**

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

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

For more information about the `nios2-bsp-query-settings` command-line options, refer to the Software Build Tools Reference section.

**Related Information**

[Software Build Tools Reference](#) on page 15-1

For more information about the `nios2-bsp-query-settings` command-line options.

**Managing Device Drivers**

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

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

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

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

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

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

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

### Creating a Custom Version of newlib

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

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

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

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

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

For the most up-to-date list of precompiled libraries and the corresponding switches, enter the following command:

```
nios2-elf-gcc --print-multi-lib
```

For more information about specific flags, refer to the "Options" section in the GCC online documentation.

**Related Information**

**Nios II Options**

### Controlling the stdio Device

The build tools offer several ways to control the details of your stdio device configuration, such as the following:
To prevent a default stdio device from being chosen, use the following command:

```
nios2-bsp hal my_bsp --default_stdio none
```

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

```
nios2-bsp hal my_bsp --default_stdio uart1
```

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

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

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

**Configuring Optimization and Debugger Options**

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

You can control the optimization and debug level through the project makefile, which determines the compiler options.

**Example 4–5. Default Application Makefile Settings**

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

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

**Example 4–6. Application Makefile Settings with Optimization**

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

**Note:** When you change compiler options in a makefile, before building the project, run `make clean` to ensure that all sources are recompiled with the correct flags.

For more information about makefile editing and `make clean`, refer to the “Applications and Libraries” chapter.

**Related Information**

*Applications and Libraries* on page 4-4

**Configuring a BSP for Debugging**

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

```
nios2-bsp hal my_bsp --set hal.make.bsp_cflags_debug -g \
      --set hal.make.bsp_cflags_optimization -O0r
```

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

You can easily copy an existing BSP and modify it to create a different build configuration.

For more information, refer to the “Copying, Moving, or Renaming a BSP” chapter.

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

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

Related Information

**Copying, Moving, or Renaming a BSP** on page 4-9

---

**Details of BSP Creation**

BSP creation is the same in the Nios II SBT for Eclipse as at the command line. The `nios2-bsp-create-settings` utility creates a new BSP settings file.

For more information about BSP settings files, refer to the “BSP Settings File Creation” chapter.

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

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

For more information about `set_ignore_file` and other SBT Tcl commands, refer to Software Build Tools Tcl Commands in the "Software Build Tools Reference" chapter.

**Note:** Avoid modifying BSP files. Use BSP settings, or custom device drivers or software packages, to customize your BSP.
Figure 4-1: Default Tcl Script and nios2-bsp-generate-files Both Using the .sopcinfo file

Note: Nothing prevents you from modifying a BSP generated file. However, after you do so, it becomes difficult to update your BSP to match changes in your hardware system. If you regenerate your BSP, your previous changes to the generated file are destroyed.

For more information about regenerating your BSP, refer to the “Revising Your BSP” chapter.

Related Information
- **Revising Your BSP** on page 4-29
- **BSP Settings File Creation** on page 4-21
- **Software Build Tools Reference** on page 15-1
**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 SBT 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 `nios2-bsp-generate-files` utility has access to the .sopcinfo file.

The `nios2-bsp-create-settings` utility creates a new BSP settings file. The `nios2-bsp-update-settings` utility updates an existing BSP settings file. The `nios2-bsp-query-settings` utility reports the setting values in an existing BSP settings file. The `nios2-bsp-generate-files` utility generates a BSP from the BSP settings file.

**Figure 4-2: Interaction between the SBT and the BSP Settings File**

---

**Generated and Copied Files**

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

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

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

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

**HAL BSP Files and Folders**

The SBT creates the HAL BSP directory in the location you specify.

**HAL BSP After Generating Files**

The SBT places generated files in the top-level BSP directory, and copied files in the HAL and drivers directories.
### Table 4-6: Generated BSP Files

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

**Note:** For more information about the alt_sys_init.c and system.h files, refer to the Developing Programs Using the Hardware Abstraction Layer section.

**Related Information**

- Creating Memory Initialization Files on page 4-10
- Developing Programs Using the Hardware Abstraction Layer on page 6-1
### Copied BSP Files

#### Table 4-7: Copied BSP Files

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

HAL BSP After Build

my_hal_bsp

- settings.bsp
- summary.html
- Makefile
- public.mk
- mem.init.mk
- system.h
- alt_sys_init.c
- linker.h
- linker.x
- memory.gdb

HAL

- src (*.c, *.S files)
- inc (*.h files)

drivers

- src (*.c, *.S files)
- inc (*.h files)

obj

HAL

- src (.o files)

drivers

- src (.o files)

libhal_bsp.a
Linker Map Validation

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

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

Tcl Scripts for BSP Settings

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

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

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

For more information about the Tcl commands used to manipulate BSPs, refer to “Software Build Tools Tcl Commands” in the Software Build Tools Reference section.

Related Information
Software Build Tools Reference on page 15-1

Calling a Custom BSP Tcl Script

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

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

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

For more information about using Tcl scripts in the SBT for Eclipse, refer to Using the BSP Editor in the "Getting Started with the Graphical User Interface" chapter.

For more information, refer to an example of custom Tcl script usage in the “Creating Memory Initialization Files” chapter.
Note: Any settings you specify in your script override the BSP default values. For more information about BSP defaults, refer to the “Specifying BSP Defaults” chapter.

Note: 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.

Note: When you use a custom Tcl script to create your BSP, you must include the script in the set of files archived in your version control system. For more information, refer to the “Using Version Control” chapter.

Related Information

- Specifying BSP Defaults on page 4-34
- Using Version Control on page 4-8
- Creating Memory Initialization Files on page 4-10
- Getting Started with the Graphical User Interface on page 2-1

**Simple Tcl Script**

Example 4–8. To Set stdio to a Device with the name 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
```

**Tcl Script to Examine Hardware and Choose Settings**

Note: The SBT uses slave descriptors to refer to components connected to the 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 processor, the slave descriptor is the same as the name of the component (for example, onchip_mem_0). If a component has multiple slave ports connecting the to multiple resources in the component, the slave descriptor is the name of the component followed by an underscore and the slave port name (for example, onchip_mem_0_s1).

```
# 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
        } else {
            # If no JTAG UARTs are found, use last character device.
            if { $first_jtag_uart eq "none" } {
                set first_jtag_uart $slave_desc
            } else {
                # If no character devices are found, use first JTAG UART.
                if { $last_stdio eq "none" } {
                    set last_stdio $slave_desc
                } else {
                    # If both character and JTAG UARTs are found, use last character device.
                    if { [regexp "\d$" $first_jtag_uart] } {
                        set first_jtag_uart $slave_desc
                    } else {
                        set last_stdio $slave_desc
                    }
                }
            }
        }
    }
    return "none"
}
```

Send Feedback
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 
# 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

Related Information
- Developing Device Drivers for the Hardware Abstraction Layer on page 7-1
- Specifying BSP Defaults on page 4-34

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

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

Rebuilding Your BSP

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

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

Related Information
Regenerating Your BSP on page 4-30

How to Rebuild Your BSP
In the SBT for Eclipse, right-click the BSP project and click Build.
On the command line, change to the BSP directory and type `make`.

## Regenerating Your BSP

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

### What Happens

Regenerating a BSP has the following effects:

- Reads the `.sopcinfo` file for basic system parameters such as module base addresses and clock frequencies.
- Retrieves the current system identification (ID) from the `.sopcinfo` file. Ensures that the correct system ID is inserted in the `.elf` file the next time the BSP is built.
- Adjusts the default memory map to correspond to changes in memory sizes. If you are using a custom memory map, it is untouched.
- Retains all other existing settings in the BSP settings file.

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

- Ensures that the correct set of BSP files is present, as follows:
  - Copies all required source files to the BSP directory tree. Copied BSP files are listed in the "Copied BSP Files" ([Table 4–8 on page 4–23](#)).
    
    If a copied file (such as a HAL source file) already exists, the SBT checks the file timestamp against the timestamp of the file in the EDS installation. The tools do not replace the BSP file unless it differs from the distribution file.
  - Recreates all generated files. Generated BSP files are listed in the "Generated BSP Files" table ([Table 4–7 on page 4–23](#)).

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

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

### When to Regenerate Your BSP

Regenerating your BSP is required (and sufficient) in the following circumstances:
You change your hardware design, but all BSP system-dependent settings remain consistent with the new .sopcinfo file. The following are examples of system changes that do not affect BSP system-dependent settings:

- Changing a component’s base address
- With the internal interrupt controller (IIC), adding or removing hardware interrupts
- With the IIC, changing a hardware interrupt number
- Changing a clock frequency
- Changing a simple processor option, such as cache size or core type
- Changing a simple component option, other than memory size.
- Adding a bridge
- Adding a new component
- Removing or renaming a component, other than a memory component, the stdio device, or the system timer device
- Changing the size of a memory component when you are using the default memory map

Note: Unless you are sure that your modified hardware design remains consistent with your BSP settings, update your BSP. For more information, refer to the “Updating Your BSP” chapter.

- 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.

- You have installed a new version of the EDS, and you want the updated BSP software implementations.
- When you attempt to rebuild your project, an error message indicates that the BSP must be updated.
- You have updated or recreated the BSP settings file.

Related Information
Updating Your BSP on page 4-32

How to Regenerate Your BSP
You can regenerate your BSP in the SBT for Eclipse, or with SBT commands at the command line.

Regenerating Your BSP in Eclipse
In the SBT for Eclipse, right-click the BSP project, point to , and click Generate BSP.

For more information about generating a BSP with the SBT for Eclipse, refer to the Getting Started with the Graphical User Interface section.

Related Information
Getting Started with the Graphical User Interface on page 2-1

Regenerating Your BSP from the Command Line
From the command line, use the nios2-bsp-generate-files command.

For more information about the nios2-bsp-generate-files command, refer to the Software Build Tools Reference section.

Related Information
Software Build Tools Reference on page 15-1
Updating Your BSP

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

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

What Happens

Updating a BSP has the following effects:

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

For more information about actions taken when you regenerate the BSP after updating it, refer to the “Regenerating Your BSP” chapter.

Related Information

Regenerating Your BSP on page 4-30

When to Update Your BSP

Updating your BSP is necessary in the following circumstances:

• A change to your BSP settings is required.
• Changes to your .sopcinfo file make it inconsistent with your BSP. The following are examples of system changes that affect BSP system-dependent settings:
  • Renaming the processor
  • Renaming or removing a memory, the stdio device, or the system timer device
  • Changing the size of a memory component when using a custom memory map
  • Changing the processor reset or exception slave port or offset
  • Adding or removing an external interrupt controller (EIC)
  • Changing the parameters of an EIC
• When you attempt to rebuild your project, an error message indicates that you must update the BSP.

How to Update Your BSP

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

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

For more information about the nios2-bsp-update-settings command, refer to the "Software Build Tools Reference" chapter.

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

For more information about using the default Tcl script, refer to the “Specifying BSP Defaults” chapter.

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

The nios2-bsp script executes the default Tcl script every time it runs, overwriting previous default settings.
For more information about preserving all settings, including the default settings, use the DONT_CHANGE keyword, described in the “Top Level Tcl Script for BSP Defaults” chapter.

Alternatively, you can provide nios2-bsp with command-line options or Tcl scripts to override the default settings.

For more information about using the nios2-bsp script, refer to the "Software Build Tools Reference" chapter.

Related Information
• Specifying BSP Defaults on page 4-34
• Top Level Tcl Script for BSP Defaults on page 4-35
• Software Build Tools Reference on page 15-1
  For more information about the nios2-bsp-update-settings command and using the nios2-bsp script.

Recreating Your BSP

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

Note: After you recreate your BSP, you must always regenerate it.

What Happens
Recreating a BSP has the following effects:
• System-dependent settings are created based on the current hardware system.
• Non-system-dependent settings can be selected by the default Tcl script, by values you specify, or both.

For more information about actions taken when you generate the BSP after recreating it, refer to the “Regenerating Your BSP” chapter.

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

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

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

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

For more information about exporting Tcl scripts, refer to Using the BSP Editor in the "Getting Started with the Graphical User Interface" chapter.

By recreating the BSP settings file with a Tcl script that specifies all BSP settings, you can reproduce the original BSP while ensuring that system-dependent settings are adjusted correctly based on any changes in the hardware system.
For more information about Tcl scripting with the SBT, refer to the “Tcl Scripts for BSP Settings”.

**Related Information**
- [Tcl Scripts for BSP Settings](#) on page 4-26
- [Getting Started with the Graphical User Interface](#) on page 2-1

### Recreating Your BSP in Eclipse

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

For more information, refer to “Getting Started with Software in Eclipse” and “Using the BSP Editor” in the Getting Started with the Graphical User Interface section.

**Related Information**
- [Getting Started with the Graphical User Interface](#) on page 2-1

### Recreating Your BSP at the Command Line

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

The `nios2-bsp-create-settings` command does not apply default settings to your BSP. However, you can use the `--script` command-line option to run the default Tcl script.

For more information about the default Tcl script, refer to the “Specifying BSP Defaults” chapter.

For more information about using the `nios2-bsp-create-settings` command, refer to the "Software Build Tools Reference" chapter.

**Related Information**
- [Specifying BSP Defaults](#) on page 4-34
- [Software Build Tools Reference](#) on page 15-1

### Specifying BSP Defaults

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

< EDS install path>/sdk2/bin

**Table 4-8: Default Tcl Script Components**

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

For more information about Tcl scripting with the SBT, refer to the “Tcl Scripts for BSP Settings” chapter. The SBT uses the default Tcl scripts to specify default values for system-dependent settings. System-dependent settings are BSP settings that reference system information in the .sopcinfo file.

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

You can pass command-line options to the default Tcl script to override the choices it makes or to prevent it from making changes to settings.

For more information, refer to the “Top Level Tcl Script for BSP Defaults” chapter.

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

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

The default Tcl scripts use slave descriptors to assign devices.

Related Information

- [Tcl Scripts for BSP Settings](#) on page 4-26
- [Top Level Tcl Script for BSP Defaults](#) on page 4-35
- [Tcl Scripts for BSP Settings](#) on page 4-26
- [Top Level Tcl Script for BSP Defaults](#) on page 4-35

### Top Level Tcl Script for BSP Defaults

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

The default Tcl script consists of a top-level Tcl script named `bsp-set-defaults.tcl` plus the helper Tcl scripts listed in the "Default Tcl Script Components" table (Table 4–9). The helper Tcl scripts do the real work of examining the .sopcinfo file and choosing appropriate defaults.
The bsp-set-defaults.tcl script sets the following defaults:
- stdio character device (bsp-stdio-utils.tcl)
- System timer device (bsp-timer-utils.tcl)
- Default linker memory regions (bsp-linker-utils.tcl)
- Default linker sections mapping (bsp-linker-utils.tcl)
- Default boot loader settings (bsp-bootloader-utils.tcl)

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

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

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

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

**Specifying the Default stdio Device**

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

For more information about these settings, refer to the Software Build Tools Reference section.

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

Related Information
Software Build Tools Reference on page 15-1

Specifying the Default System Timer

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

For more information about this setting, refer to the Software Build Tools Reference section.

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

- It must be a timer, that is, is_timer_device must return true.
- It must have a slave port connected to the processor.

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

Related Information
Software Build Tools Reference on page 15-1

Specifying the Default Memory Map

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

For more information about these commands, refer to the Software Build Tools Reference section.

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

For more information about this setting, refer to the Software Build Tools Reference section.

Related Information
Software Build Tools Reference on page 15-1

Specifying Default Bootloader Parameters

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

- hal.linker.allow_code_at_reset
- hal.linker.enable_alt_load_copy_rodata
- hal.linker.enable_alt_load_copy_rdata
- hal.linker.enable_alt_load_copy_exceptions

For more information about these settings, refer to the Software Build Tools Reference section.
Related Information
Software Build Tools Reference on page 15-1

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

If a boot loader is enabled, the assumption is that the boot loader is located at the reset address and handles the copying of sections on reset. If there is no boot loader, the BSP might need to provide code to handle these functions. You can use the alt_load() function to implement a boot loader.

Table 4-10: Boot Loader-Dependent Settings

<table>
<thead>
<tr>
<th>Setting name</th>
<th>Value When Boot Loader Enabled</th>
<th>Value When Boot Loader Disabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>hal.linker.allow_code_at_reset</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>hal.linker.enable_alt_load_copy_rodata</td>
<td>0</td>
<td>if .rodata memory different than .text memory and .rodata memory is volatile; 0 otherwise</td>
</tr>
<tr>
<td>hal.linker.enable_alt_load_copy_rwdata</td>
<td>0</td>
<td>if .rwdata memory different than .text memory; 0 otherwise</td>
</tr>
<tr>
<td>hal.linker.enable_alt_load_copy_exceptions</td>
<td>0</td>
<td>if .exceptions memory different than .text memory and .exceptions memory is volatile; 0 otherwise</td>
</tr>
</tbody>
</table>

Related Information
Software Build Tools Reference on page 15-1

Using Individual Default Tcl Procedures
The default Tcl script consists of the top-level bsp-call-proc.tcl script plus the helper scripts listed in the "Default Tcl Script Components" table (Table 4–9 on page 4–33). The procedure call Tcl script allows you to call a specific procedure in the helper scripts, if you want to invoke some of the default Tcl functionality without running the entire default Tcl script.

The procedure call Tcl script has the following usage:

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

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

(5) For more information about the settings in this table, refer to the Software Build Tools Reference section.
Device Drivers and Software Packages

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

For more information about integrating device drivers and software packages with the SBT, refer to the Developing Device Drivers for the Hardware Abstraction Layer section.

Related Information
Developing Device Drivers for the Hardware Abstraction Layer on page 7-1

Boot Configurations for Intel FPGA Embedded Software

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

For more information about the HAL boot loader process, refer to the Developing Programs Using the Hardware Abstraction Layer section.

Related Information
Developing Programs Using the Hardware Abstraction Layer on page 6-1

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

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

<table>
<thead>
<tr>
<th>Memory Type</th>
<th>Examples</th>
<th>IsFlash</th>
<th>IsNonVolatileStorage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flash</td>
<td>Common flash interface (CFI), erasable programmable configurable serial (EPCS) device</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>ROM</td>
<td>On-chip memory configured as ROM, HardCopy ROM</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>RAM</td>
<td>On-chip memory configured as RAM, HardCopy RAM, SDRAM, synchronous static RAM (SSRAM)</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

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

## Boot from Flash Configuration

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

This boot configuration has the following characteristics:

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

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

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

## Boot from Monitor Configuration

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

This boot configuration has the following characteristics:

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

The default Tcl script assumes no boot loader is in use, so it chooses this configuration only if you enable it. To enable this configuration, pass the following argument to the default Tcl script: `enable_bootloader 1`

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

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

## Run from Initialized Memory Configuration

The reset address points to the beginning of the application in memory (no boot loader). The reset memory must have its contents initialized before the processor comes out of reset. The initialization might be implemented by using a non-volatile reset memory (for example, flash, ROM, initialized FPGA RAM) or by an external master (for example, another processor) that writes the reset memory. The HAL C run-time startup code (crt0) initializes the instruction cache, uses `alt_load()` to copy select sections to their VMAs, and then jumps to `__start`. For each associated section (`.rwdata`, `.rodata`, `.exceptions`), Boolean settings control this behavior. The default Tcl scripts set these to default values as described in the "Boot Loader-Dependent Settings" table (Table 4–11 on page 4–36).

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

This boot configuration has the following characteristics:

- `alt_load()` called
- Code at reset in executable file
The default Tcl script chooses this configuration when the reset and .text memory are the same. In this boot configuration, when the processor core resets, by default the .rwdata section is not reinitialized. Reinitialization would normally be done by a boot loader. However, this configuration has no boot loader, because the software is running out of memory that is assumed to be preinitialized before startup.

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

**Run-time Configurable Reset Configuration**

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

This boot configuration has the following characteristics:

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

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

**Intel FPGA-Provided Embedded Development Tools**

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

**Software Build Tool GUls**

The EDS provides the following SBT GUls for software development:

- The SBT for Eclipse
- The BSP Editor
- The Flash Programmer

For more information about how each GUI is primarily a thin layer providing graphical control of the command-line tools, refer to “The Nios II Command-Line Commands” chapter.

**Table 4-12: Summary of the Correlation Between GUI Features and the SBT Command Line**

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

**Related Information**

The **Command-Line Commands** on page 4-43

**The SBT for Eclipse**

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

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

- In the Windows operating system, on the Start menu, point to **Programs > EDS <version>**, and click **Software Build Tools for Eclipse**.
- From the Command Shell, by typing `eclipse-nios2`.

For more information about the SBT for Eclipse, refer to the Getting Started with the Graphical User Interface section.

**Related Information**

Getting Started with the Graphical User Interface on page 2-1
The BSP Editor

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

- From the menu in the SBT for Eclipse
- From the Command Shell, by typing `nios2-bsp-editor`.

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

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

For more information about the BSP Editor, refer to “Using the BSP Editor” in the Getting Started with the Graphical User Interface section.

Related Information

- Getting Started with the Graphical User Interface on page 2-1

The Flash Programmer

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

- Executable code and data
- Bootstrap code to copy code from flash to RAM, and then run from RAM
- HAL file subsystems
- FPGA hardware configuration data

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

- From the menu in the SBT for Eclipse
- From the Command Shell, by typing:
  
  `nios2-flash-programmer-generate`

The Command Shell

The Command Shell is a `bash` command-line environment initialized with the correct settings to run command-line tools. The EDS includes two versions of the Command Shell, for the two supported GCC toolchain versions.

For more information, refer to the “GNU Compiler Tool Chain” chapter.

For more information about launching the Command Shell, refer to the "Getting Started from the Command Line" chapter.

The Command-Line Commands

This section describes the Intel FPGA command-line tools. You can run these tools from the Command Shell.

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

```
<name of tool> --help
```
GNU Compiler Tool Chain

The compiler tool chain is based on the standard GNU GCC compiler, assembler, linker, and make facilities. Intel FPGA provides and supports the standard GNU compiler tool chain for the processor.

The EDS includes version GCC 4.8.3 of the GCC toolchain.

For more information about installing the Quartus Prime, refer to the Intel FPGA Software Installation and Licensing Manual.

The following options are new for the GCC toolchain:

**Code density and performance**

Use of the -mgpopt=global setting is recommended as it generally deliver results with better code density *and* performance. (Note: this requires that everything is compiled with the same -Gn switch setting; use of -Gn is not generally recommended).

**C++ code size reduction: -fno-exceptions**

When a developer is using C++ but trying not to link in the (big) C++ exception-handling machinery, nios2-elf-g++ does, in some cases, link in the exception-handling machinery where it is not actually required. This switch suppresses that behaviour, which results in a smaller code footprint for those situations.

**Response to address 0x00 access: -fdelete-null-pointer-checks**

Developers often have RAM at address 0x00 (== NULL pointer in gcc and most C compilers). From gcc 4.9 onwards, by default gcc detects attempts to read or write to address 0x00 and converts them to traps; typically in embedded/-based systems, these traps are not handled so the code fails silently. This means that code that works when compiled with earlier versions of gcc could silently fail when compiled with gcc-4.9 (onwards).

In order to avoid this for this behavior has been modified so that accesses to address 0x00 works as expected for systems but there may be a slight negative effect on code performance. In systems that are known not to read/write RAM at address 0x00, use of the switch --fdelete-null-pointer-checks restores the original gcc behavior and may provide a small performance boost.

**Expansion of __builtin_trap**

The latest versions of GCC now produce trap 3 instead of break 3 in the expansion of the __builtin_trap function and in other situations where a trap is emitted to indicate undefined runtime behaviour. This is for compliance with the ABI for Linux targets, which does not permit the use of the break instruction in user code.

**GDB breakpoint instruction**

GDB now uniformly uses trap 31 instead of a break instruction for software breakpoints. This is for consistency with the ABI for Linux targets.

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

- nios2-elf-gcc
- nios2-elf-as
- nios2-elf-ld
- nios2-elf-objdump
- nios2-elf-size

The exception is the make utility, which is simply named make.
The GNU tools reside in the following location:

- `< EDS install path>/bin-gnu` directory

Refer to the following additional sources of information:

- For more information about managing GCC toolchains in the SBT for Eclipse—“Managing Toolchains in Eclipse” in the “Getting Started with the Graphical User Interface” chapter.
- For more information about selecting the toolchain on the command line—the "Getting Started from the Command Line" chapter.
- For more information about a comprehensive list of GNU tools—the GNU HTML documentation, refer to the `Embedded Design Suite Support` page on the Intel FPGA website.
- For more information about GNU, refer to the Free Software Foundation website.

**Related Information**

- Overview of Embedded Development on page 1-1
- Getting Started with the Graphical User Interface on page 2-1
- Getting Started from the Command Line on page 3-1
- GNU website

**Software Build Tools**

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

You can call these utilities and scripts on the command line or from the scripting language of your choice (such as `perl` or `bash`).

**Note:** For usage information, enter "--help" after the command and a list of required and optional arguments for the command appears.

**Table 4-13: Utilities and Scripts Included in the SBT**

<table>
<thead>
<tr>
<th>Command</th>
<th>Summary</th>
<th>Utility</th>
<th>Script</th>
</tr>
</thead>
<tbody>
<tr>
<td>nios2-app-generate-makefile</td>
<td>Creates an application makefile</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>nios2-lib-generate-makefile</td>
<td>Creates a user library makefile</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>nios2-app-update-makefile</td>
<td>Modifies an existing application makefile</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>nios2-lib-update-makefile</td>
<td>Modifies an existing user library makefile</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>nios2-bsp-create-settings</td>
<td>Creates a BSP settings file</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>nios2-bsp-update-settings</td>
<td>Updates the contents of a BSP settings file</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>nios2-bsp-query-settings</td>
<td>Queries the contents of a BSP settings file</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>nios2-bsp-generate-files</td>
<td>Generates all files for a given BSP settings file</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>nios2-bsp</td>
<td>Creates or updates a BSP</td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

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

For more information about the SBT, refer to the "Getting Started from the Command Line" chapter.
Related Information

- **Overview of Embedded Development** on page 1-1
- **Getting Started from the Command Line** on page 3-1

**File Format Conversion Tools**

File format conversion is sometimes necessary when passing data from one utility to another.

**Note:** For usage information, enter "-h" after the command and a list of options for the command appears.

The file conversion utilities are:

- `alt-file-convert`
- `bin2flash`
- `elf2dat`
- `elf2flash`
- `elf2hex`
- `elf2mif`
- `flash2dat`

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

**alt-file-convert (BETA)**

**Description**

`alt-file-convert` (BETA): General file conversion tool. For , it is primarily used for generating a Nios II application image for Max Onchip Flash and EPCQ.

**Usage**

```
alt-file-convert -I <input_type> -O <output_type> [option(s)] --input=<input_file>
--output=<output_file>
```

For , the BETA version is limited to the following uses:

- Convert between Intel HEX (byte addressed) and Quartus HEX (word addressed)
- Convert between Quartus HEX files of various widths
- Covert an `.elf` file to a HEX file and append a bootcopier (used for application flash image for Max On-chip Flash and EPCQ)

**Options**

- `-h, --help` - prints usage
- `-I` - input type
- `-O` - output type
- `--base` - base address (in hex) of target memory (default 0x0)
- `--end` - end address (in hex) of target memory (default 0xFFFFFFFF)
- `--reset` - reset address (in hex) of target memory (default None)
- `--input` - path to input file
- `--output` - path to output file
- `--in-data-width` - data width of inputfile [8, 16, 32, 64, 128, 256] (default 8)
- `--out-data-width` - data width of target memory [8, 16, 32, 64, 128, 256] (default None)
- `--boot` - location of boot file to be appended (srec format)
Examples

Converting from Intel Hex (IHEX) to a Quartus Hex (HEX) for a memory with a 32-bit data width:

```
alt-file-convert -I ihex -O
  hex out-data-width 32 in.ihex out.hex
```

Converting from an .elf file to a flash and appending a bootcopier given as a srec file:

```
alt-file-convert -I elf32-littlenios2 -O flash in.elf out.flash --have-boot-copier
  --boot boot.elf
```

bin2flash

Description

The bin2flash utility converts any binary data file to a FLASH file that can be used by the flash programmer.

Usage

```
```

Options

```
--debug             debug mode
--help              print this message
--input=<file>      input Binary file to process
--location=<addr>   design location within the flash
--log=<file>        file for logging progress
--output=<file>     output flash file
--quiet             only print errors
--silent            silent mode - same as quiet
--verbose           lots of interesting information
```

elf2dat

Description

Converts a .elf file to a .dat file format appropriate for Verilog HDL hardware simulators.

Usage

```
elf2dat --infile=file --outfile=file --width=width --base=address
  --end=address [--pad=number] [--create-lanes=number]
  [--little-endian-mem] [--big-endian-mem]
```

Options

```
--infile=<elf-input-filename>
--outfile=<dat-output-filename>
--base=<base address>
--end=<end address>
--pad=[0 | 1] (default 1)
--create-lanes=[0 | 1] (default 0)
```
elf2dat

Transforms the data within an ELF file in the address range [base, end] into the corresponding DAT file. Lane files are optionally created (--create-lanes=1, default is 0). Lane file names are generated based on the output file by inserting "_lane0", "_lane1", etc. before the ".dat" extension of the output filename. If "--pad=1" is specified, any unspecified locations in memory is filled with zeros. If "--little-endian-mem" is specified, the memory is assumed to be little-endian. If "--big-endian-mem" is specified, the memory is assumed to be big-endian. If neither --little-endian-mem or --big-endian-mem is specified, the memory is assumed to be little-endian. Note that the endianness of the ELF file never effects the result.

Example

```
elf2dat --infile=foo.elf --outfile=bar.dat --width=32 --create-lanes=1
    --base=0 --end=0x1000
```

creates DAT files bar.dat, bar_lane0.dat, bar_lane1.dat, bar_lane2.dat, bar_lane3.dat

elf2flash

Description

The elf2flash utility takes a software file in ELF format and translates it to a FLASH file that can be programmed into the flash memory connected to an Intel FPGA.

Usage

```
    [--offset=offset] [--sim_optimize=<value>] [--output=file]
```

Options

```
--after=[<file>]  Start output at address immediately
    following data in specified .flash file
--base=<addr>    flash base address
--boot=<file>    boot copier SREC file
--debug          debug mode
--end=<addr>     flash end address
--ecps           epcs flash mode
--help           print this message
--input=<file>   input ELF file to process
--log=<file>     file for logging progress
--offset=<offset>  EPCS offset
--output=<file>  output flash file
--quiet          only print errors
--reset=<addr>   CPU reset address
--save           save intermediate files
--silent         silent mode - same as quiet
--sim_optimize=[<value>]  Optimize for simulation
--verbose        lots of interesting information
```

The elf2flash utility converts the software and data within an ELF file in the address range [base, end] into a FLASH file. This utility can also optionally insert a boot copier into the FLASH file to copy the software from flash memory to RAM before running it.
elf2hex

Description

The elf2hex utility takes a software file in ELF format and translates it to a HEX file that can be used as the initialization data for a memory component.

Usage

```
elf2hex [--debug] [--record=length] [--base=addr] [--quiet]
[--width=value] [--big-endian-mem] [--no-zero-fill]
[--create-lanes=[value]] [--lower] [--end=addr] [--verbose] [--silent]
[--output=file]
```

Options

```
--base=<addr>              Hex data base address
--big-endian-mem           Force big-endian memory layout
--create-lanes=[value]     1 if lane files should be created
--debug                    debug mode
--end=<addr>               Hex data end address
--help                     print this message
--input=<file>             input ELF file to process
--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
--output=<file>            output Hex file
--quiet                    only print errors
--record=<length>          Output record length, in bytes.
--silent                   silent mode - same as quiet
--verbose                  lots of interesting information
--width=<value>            [ 8 | 16 | 32 | 39 | 64 | 128]
```

The elf2hex utility converts the software and data within an ELF file in the address range [base, end] into a HEX file. The width option is used to insure 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.

elf2mif

Description

elf2-h transforms the data within an elf file in the address range [low, high] into the corresponding -H file. Lane files are optionally created (--create-lanes=1, default is 0). Lane file names are generated based on the output file by inserting "_lane0", "_lane1", etc. before the ".-h" extension of the output filename.

Usage

```
elf2-h elf_file [low_address high_address] [--width=width]
[--create-lanes=lanes] [-h_file]
```
### Options

- **elf_file**
  - The elf input file
- **low address**
  - Beginning of the range to transform
- **high address**
  - End of the range to transform
- **--width=width**
  - [8 | 16 | 32 | 64 | 128]
- **--create-lanes=lanes**
  - [0 | 1] (default 0)
- **-h_file**
  - The -H file to create

### Example

```
elf2-h foo.elf 0 0x0FFF --width=32 --create_lanes=1 bar.-h
```

creates -H files—bar.-h, bar_lane0.-h, bar_lane1.-h, bar_lane2.-h, and bar_lane3.-h

### flash2dat

**Description**

Converts an SRAM Object File (.sof) to a .flash file.

**Usage**

```
flash2dat --infile=<flash-input-filename> --outfile=<dat-output-filename> --width=[8 | 16 | 32 | 64 | 128] --base=<base byte-address of range of interest> --end=<last byte-address of range of interest> --pad=<1 if all locations should be represented in output> --create-lanes=<1 if lane files should be created> --relocate-input=<input-data relocation offset (usually=base)> [--little-endian-mem] [--big-endian-mem]
```

**Example**

```
flash2dat --infile=foo.flash --outfile=foo.dat --width=32 --base=0x1000000 --end=0x1000FFF --pad=1 --create-lanes=1 --relocate=0x1000000
```

Creates files foo.dat, foo_lane0.dat, foo_lane1.dat, foo_lane2.dat and foo_lane3.dat.
foo.dat contains 4K bytes of data, padded with 0 where data was not present in the .flash file. foo_lane[0123].dat each contain 1K bytes of data from their respective byte lanes. The input .flash file data should be relocated by 0x1000000 to correspond to the specified --base address.

### sof2flash

**Description**

The sof2flash utility takes an FPGA configuration file in SOF format and translates it to a FLASH file that can be programmed into the flash memory connected to an FPGA.

**Usage**

```
```
Options

--activeparallel create parallel flash contents for active parallel programming
--compress compress the SOF
--debug debug mode
--epcq EPCQ flash mode of size 256Mbits
--epcq128 EPCQ flash mode of size 128Mbits or less
--epcs EPICS flash mode
--help print this message
--input=<file> input SOF file to process
--log=<file> file for logging progress
--offset=<addr> design location within the flash
--optionbit=<optionbitaddress> Option Bit Address
--options=<file> intermediate Options file (for EPSCS only)
--output=<file> output flash file
--pfl Pfl mode
--programmingmode=<programmingmode> Programming mode
--quiet only print errors
--save save intermediate files
--timestamp produce a s0 record with timestamp of input file
--verbose lots of interesting information

Other Command-Line Tools

Note: For usage information, enter "--help" after the command and a list of options for the command appears.

The command-line tools for developing programs are:

- nios2-download
- nios2-flash-programmer-generate
- nios2-flash-programmer (64-bit support)
- nios2-gdb-server (64-bit support)
- nios2-terminal (64-bit support)
- validate_zip
- nios2-configure-sof
- jtagconfig

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

nios2-download

Description

This tool prepares a target system for debugging, it checks that the hardware and software are compatible, downloads the software and optionally makes the target processor run the downloaded code.
Usage


Options

-h|--help                  Prints this message.
-C|--directory <dir name>  Change to this directory before running
-c|--cable <cable name>    Specifies which JTAG cable to use (not needed if you only have one cable).
-d|--device <device index> Specifies in which device you want to look for the CPU (1 = device nearest TDI etc.)
-i|--instance <instance>   Specifies the INSTANCE value of the CPU JTAG debug module (auto-detected if you specify an ELF file or use --directory to point to one)
-r|--reset-target         Reset the target SOPC system before use.
-s|--sidp <address>        Base-address of System ID peripheral on target
                         This base-address must be specified in hex (0x...)
-I|--id <system-id-value>  Unique ID code for target system
-t|--timestamp <time-stamp> Timestamp for target-system (when last generated)
                        --accept-bad-sysid       Continue even if the system ID comparison fails
                        <filename.elf>              An ELF file to download to the target
-w|--wait <seconds>        Wait for time specified before starting processor
-g|--go                    Run processor from entry point after downloading.
                        --stop                   Stop processor (leave it in a paused state).
                        --tcpport <port> | auto  Listen on specified TCP port for a connection from
                         GDB (with "auto" a free port is used).
                        --write-gmon <file>     Write profiling data to the file specified when the
                         target exits.
                        --jdi <file>            Specify which .jdi file to read the INSTANCE
                         value of the CPU JTAG debug module from. (if not specified, looks in the same folder
                         as the .ptf file, as specified in generated.sh)
                        --cpu_name <name>       CPU module name to use when trying to scan for
                         INSTANCE value from the .jdi file. Uses the
                         name specified in generated.sh if it is not passed in.
                        NOTE: nios2-download needs a .jdi file and a cpu_name
                         to search for the INSTANCE value. This can be
                         supplied through the command line, or searched
                         for with information from generated.sh

If you specify the --go option then the target processor starts before this tool exits. If --go is not specified but a file is downloaded or the target processor needs to be reset then it is left in the paused state. Otherwise the target processor state is left unchanged unless --stop is specified (in which case it is paused). Return codes are: 0 for success; 1 if the system ID did not match; 2 if no CPUs are found; a return code of 4 or more indicates an error which needs manual intervention to solve.
nios2-flash-programmer-generate

Description

The nios2-flash-programmer-generate command converts multiple files to the .flash format and then program them to the designated target flash devices (--program-flash). This is a convenience utility that manages calls to the following command line utilities: bin2flash, elf2flash, sof2flash, and nios2-flash-programmer. This utility also generates a convenience shell script that captures the sequence of conversion and flash programmer command line tool expressions.

Usage

nios2-flash-programmer-generate --sopcinfo <filename> --flash-dir <filepath> [options]

Options

--flash-dir <filepath> Path to the directory where the flash files are generated. Use . for the current directory. This command overwrites pre-existing files in <filepath> without warning.

--sopcinfo <filename> The SOPC Builder system file (.sopcinfo).

Optional Arguments:

--accept-bad-sysid Continue even if the system ID comparison fails

--add-bin <fname> <flash-slave-desc> <offset> Specify an bin file to convert and program. The filename, target flash slave descriptor, and target offset amount are required. This option can be used multiple times for .sof files.

--add-elf <fname> <flash-slave-desc> <extra-elf2flash-arguments> Specify an elf file to convert and program. The filename, and target flash slave descriptor are required. This option can be used multiple times for .elf files. The [extra-elf2flash-arguments] can be any of these options supported by 'elf2flash': 'save', 'sim_optimize'. The 'elf2flash' options: 'base', 'boot', 'end', 'reset' have default values computed, but are also supported as [extra-elf2flash-arguments] for manual override of those defaults. Do not prepend extra arguments with '--'

--add-sof <fname> <flash-slave-desc> <offset> <extra-sof2flash-arguments> Specify an sof file to convert and program. The filename, target flash slave descriptor, and target offset amount are required. This option can be used multiple times for .sof files. The [extra-sof2flash-arguments] can be any of these options supported by 'sof2flash': 'activeparallel', 'compress', 'save', 'timestamp'. Do not prepend extra arguments with '--'

--cable <cable name> Specifies which JTAG cable to use (not needed if you only have one cable). Not used without --program-flash option

--cpu <cpu name> The SOPC Builder system file CPU name. Not
required if only one CPU in the system.

--debug
Sends debug information, exception traces, verbose output, and default information about the command's operation, to stdout.

--device <device name>
Specifies in which device you want to look for the Nios II debug core ('1' is device nearest TDI etc.). Not used without --program-flash option

--erase-first
Erase entire flash targets before programming them. Not used without --program-flash option

--extended-help
Displays full information about this command and its options.

--go
Run processor from reset vector after program.

--help
Displays basic information about this command and its options.

--id <address>
Unique ID code for target system. Not used without --program-flash option

--instance <instance value>
Specifies the INSTANCE value of the debug core (not needed if there is exactly one on the chain). Not used without --program-flash option

--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.

--log <filename>
Creates a debug log and write to specified file. Also logs debug information to stdout.

--mmu
Specifies if the CPU with the corresponding INSTANCE value has an MMU (not needed if there is exactly one CPU in the system). Not used without --program-flash option

--program-flash
Providing this flag causes calls to nios2-flash-programmer to be generated and executed. This results in flash targets being programmed.

--script-dir <filepath>
Path to the directory where a shell script of this tools executed command lines is generated. This convenient script can be used in place of this 'nios2-flash-programmer-generate' command. Use . for the current directory. This command overwrites pre-existing files in <filepath> without warning.

--sidp <address>
Base-address of System ID peripheral on target. Not used without --program-flash option

--silent
Suppresses information about the command's operation normally sent to stdout.

--timestamp <time-stamp>
Timestamp for target-system (when last generated).

--verbose
Sends verbose output, and default information about the command's operation, to stdout.

--version
Displays the version of this command and exits with a zero exit status.
nios2-flash-programmer

Description

Programs data to flash memory on the target board.

Usage

nios2-flash-programmer [-h|--help] [-c|--cable <cable name>]
[-d|--device <device index>] [-i|--instance <instance>]
[-s|--sidp <address>] [-I|--id <id>] [-t|--timestamp <timestamp>]
-b|--base <address> [-e|--epcs] <action> [-g|--go] [--dualdie]

Options

-h|--help       Print this message
-Q|--quiet      Don't print anything if everything works
--debug        Print debug information
-c|--cable <cable name> Specifies which JTAG cable to use (not needed if
                you only have one cable)
-d|--device <device index> Specifies in which device you want to look for the
                debug core (1 = device nearest TDI etc.)
--dualdie       Force override the die size to be dual die.
-i|--instance <instance> Specifies the INSTANCE value of the debug core
                (not needed if there is exactly one on the chain)
-s|--sidp <address> Base-address of System ID peripheral on target
-I|--id <system-id-value> Unique ID code for target system
-t|--timestamp <time-stamp> Timestamp for target-system (when last generated)
--accept-bad-sysid Continue even if the system ID comparison fails
-b|--base <address> Base address of FLASH/EPCS to operate on
-e|--epcs This operation is on an EPSCS flash
-M|--mmu MMU present in the cpu.
-E|--erase <start>+<size> Erase a range of bytes in the flash, or the entire
                flash before/instead of programming it.
-P|--program Program flash from the input files (the default)
                --no-keep-nearby Don't preserve bytes which need to be erased but
                which aren't specified in the input file
-Y|--verify <filename>* Verify that contents of flash match input files
                The names of the file(s) to program or verify
-R|--read <file> Read flash contents into file
-B|--read-bytes <start>+<size> Specify which bytes to read
-g|--go Run processor from reset vector after program.

Input files should be in Motorola S-Record format. Addresses within the files are interpreted as offsets
from the base address of the flash. Output files written by the tool are in the same format. The flash
programmer supports all CFI flashes which use the AMD programming algorithm (CFI algorithm 2) or
the Intel algorithm (1 or 3).

nios2-flash-programmer (64-bit support)

Description

Programs data to flash memory on the target board.
Usage

nios2-flash-programmer [-h/--help] [-c/--cable <cable name>] [-d/--device <device index>] [-i/--instance <instance>] [-s/--sidp <address>] [-I/--id <id>] [-t/--timestamp <timestamp>] -b/--base <address> [-e/--epcs] <action> [-g/--go] [--dualdie]

Options

-h/--help          Print this message
-Q/--quiet         Don't print anything if everything works
-0/--debug         Print debug information
-c/--cable <cable name> Specifies which JTAG cable to use (not needed if you only have one cable)
-d/--device <device index> Specifies in which device you want to look for the debug core (1 = device nearest TDI etc.)
--dualdie          Force override the die size to be dual die.
-i/--instance <instance> Specifies the INSTANCE value of the debug core (not needed if there is exactly one on the chain)
-s/--sidp <address> Base-address of System ID peripheral on target
-I/--id <system-id-value> Unique ID code for target system
-t/--timestamp <time-stamp> Timestamp for target-system (when last generated)
--accept-bad-sysid Continue even if the system ID comparison fails
-b/--base <address> Base address of FLASH/EPCS to operate on
-e/--epcs          This operation is on an EPCS flash
-M/--mmu           MMU present in the cpu.
-E/--erase <start>+<size> Erase a range of bytes in the flash, or the entire flash before/instead of programming it.
-P/--program       Program flash from the input files (the default)
-0/--keep-nearby    Don't preserve bytes which need to be erased but which aren't specified in the input file
-Y/--verify        Verify that contents of flash match input files
<filename>*        The names of the file(s) to program or verify
-R/--read <file>   Read flash contents into file
-B/--read-bytes <start>+<size> Specify which bytes to read
-g/--go           Run processor from reset vector after program.

Input files should be in Motorola S-Record format. Addresses within the files are interpreted as offsets from the base address of the flash. Output files written by the tool are in the same format. The flash programmer supports all CFI flashes which use the AMD programming algorithm (CFI algorithm 2) or the Intel algorithm (1 or 3).

nios2-gdb-server

Description

Translates GNU debugger (GDB) remote serial protocol packets over Transmission Control Protocol (TCP) to JTAG transactions with a target processor.

Usage

Options

- Options

-h/--help  Print this message
--write-pid <filename>  Write the process ID to the file specified
--signal-pid <filename>  Send a SIGUSR1 to the other process when ready
-c/--cable <cable name>  Specifies which JTAG cable to use (not needed if
                              you only have one cable)
-d/--device <device index>  Specifies in which device you want to look for the
                              OCI core (1 = device nearest TDI etc.)
-i/--instance <instance>  Specifies the INSTANCE value of the JTAG debug core
                              (not needed if there is exactly one on the chain)
-r/--reset-target  Always reset the target system before use
-C/--init-cache  Always initialize target processor cache before use
-s/--sidp <address>  Base-address of System ID peripheral on target
-I/--id <system-id-value>  Unique ID code for target system
-t/--timestamp <time-stamp>  Timestamp for target-system (when last generated)
--accept-bad-sysid  Continue even if the system ID comparison fails
<filename.srec>*  S-Record file(s) to download to the target
--no-verify  Don't verify after downloading
-w/--wait <seconds>  Wait for time specified before starting processor
                        (signal USR1 terminates the wait early)
-g/--go  Run processor from entry point after downloading
--stop  Stop processor (leave it in a paused state).
--tcpport <port> | auto  Listen on specified TCP port for a connection from
                           GDB (with "auto" a free port is used).
--tcpdebug  Display the GDB remote protocol running over TCP
--tcptimeout <seconds>  Timeout if GDB has not connected in time specified
--tcppersist  Don't stop when GDB disconnects
--thread-vars <vars>  Decode target thread list using specified string
--write-gmon <gmonfile>  Wait for target to complete and write GMON data
                        (if available) to the file specified
--wait-target-exit  Wait for target to complete and return target exit
                        code as our return code if no other errors

The processor state after execution is chosen as follows:

• If the GDB remote protocol is enabled then this chooses the state.
• If the --go option is specified then the processor is started.
• If the --stop option is specified then the processor is left paused.
• If a file is downloaded or the processor is reset then it is paused.
• Otherwise, the processor state is restored.

nios2-gdb-server (64-bit support)

Description

Translates GNU debugger (GDB) remote serial protocol packets over Transmission Control Protocol
(TCP) to JTAG transactions with a target processor.

Software Build Tools

Altera Corporation

Send Feedback
Usage

nios2-gdb-server [-h/--help] [-c/--cable <cable name>]
                [-d/--device <device index>] [-i/--instance <instance>]
                [-s/--sidp <address>] [-I/--id <id>] [-t/--timestamp <timestamp>]
                [-w/--wait <seconds>] [--no-verify] [-g/--go] [-s/--stop]
                [--tcpport=<port> | auto] [--tcpdebug] [--tcptimeout <seconds>]
                [--tcppersist] [--thread-vars <vars>]
                [--write-gmon <gmonfile>] [--wait-target-exit]
                <filename.srec>*

Options

-h/--help                  Print this message
--write-pid <filename>   Write the process ID to the file specified
--signal-pid <filename>  Send a SIGUSR1 to the other process when ready
-c/--cable <cable name>   Specifies which JTAG cable to use (not needed if
                          you only have one cable)
-d/--device <device index> Specifies in which device you want to look for the
                             OCI core (1 = device nearest TDI etc.)
-i/--instance <instance>  Specifies the INSTANCE value of the JTAG debug core
                          (not needed if there is exactly one on the chain)
-r/--reset-target         Always reset the target system before use
-C/--init-cache           Always initialize target processor cache before use
-s/--sidp <address>       Base-address of System ID peripheral on target
-I/--id <system-id-value> Unique ID code for target system
-t/--timestamp <time-stamp> Timestamp for target-system (when last generated)
--accept-bad-sysid        Continue even if the system ID comparison fails
<filename.srec>*          S-Record file(s) to download to the target
--no-verify               Don't verify after downloading
-w/--wait <seconds>       Wait for time specified before starting processor
                          (signal USR1 terminates the wait early)
-g/--go                   Run processor from entry point after downloading
                          --stop         Stop processor (leave it in a paused state).
                          --tcpport <port> | auto Listen on specified TCP port for a connection from
                          GDB (with "auto" a free port is used).
                          --tcpdebug     Display the GDB remote protocol running over TCP
                          --tcptimeout <seconds> Timeout if GDB has not connected in time specified
                          --tcppersist   Don't stop when GDB disconnects
                          --thread-vars <vars> Decode target thread list using specified string
                          --write-gmon <gmonfile> Wait for target to complete and write GMON data
                          (if available) to the file specified
                          --wait-target-exit Wait for target to complete and return target exit
                          code as our return code if no other errors

The processor state after execution is chosen as follows:
- If the GDB remote protocol is enabled then this chooses the state.
- If the --go option is specified then the processor is started.
- If the --stop option is specified then the processor is left paused.
- If a file is downloaded or the processor is reset then it is paused.
- Otherwise, the processor state is restored.
nios2-terminal

Description
Performs terminal I/O with a JTAG UART in a system.

Usage
nios2-terminal [TERMINAL OPTIONS] [CONNECTION OPTIONS] [TARGET OPTIONS]

Options

Terminal Options

- `-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
- `--flush` empty buffers before displaying output
- `--signal-pid` send signal to process ID when ready
- `--quit-after=SECS` quit after SECS second(s) (default is never)
- `--no-quit-on-ctrl-d` don't quit if Ctrl-D received from target
- `--dump=OPTS` dump extra debug information for OPTS
- `--gmon` write gmon.out file if seen from target
- `--no-gmon` do not write gmon.out

Connection Options

- `--hardware` connect to a hardware target (default)
- `--iss` launch and connect to iss target
- `--modelsim` connect to a modelsim target
- `--uart` connect to hardware using a UART
- `--persistent` try to reconnect after an I/O error
- `--no-persistent` opposite of persistent

Connect to Hardware Options

- `--cable=CABLE` use CABLE JTAG cable (default auto-detect)
- `--device=DEVICE` connect to DEVICE device (default auto-detect)
- `--instance=INSTANCE` connect to INSTANCE instance (default auto-detect)

Launch and Connect to ISS Options

- `--elf=FILE` launch ISS with FILE in simulation memory
- `--ptf=FILE` launch ISS with FILE as system description
- `--debug` launch ISS in debug mode waiting for gdb connection at reset or entry point
- `--no-debug` launch ISS in run mode
- `--iss-arg=ARG` pass additional argument ARG to nios2-iss, may be specified repeatedly

Connect to ModelSim Options

- `--instance_name=NAME` connect to jtag_uart instance name
Connect to Hardware using UART Options

```bash
--port=PORT            full name of port to connect to (e.g. /dev/com1)
--baud-rate=RATE       baud rate to connect to UART
```

**Note:** If a development board has more than one JTAG port, the `nios2-terminal` command cannot work unless the correct JTAG cable is identified. In order to identify the correct JTAG cable, you must first run the `jtagconfig` command. For example, if the output states that USB-Blaster is port 2, then you can run the command using the correct JTAG port `nios2-terminal -c 2`.

### nios2-terminal (64-bit support)

**Description**

Performs terminal I/O with a JTAG UART in a system.

**Usage**

```
nios2-terminal [TERMINAL OPTIONS] [CONNECTION OPTIONS] [TARGET OPTIONS]
```

**Options**

#### Terminal Options

- `-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
- `--flush` empty buffers before displaying output
- `--signal-pid` send signal to process ID when ready
- `-o`, `--quit-after=SECS` quit after SECS second(s) (default is never)
- `--no-quit-on-ctrl-d` don't quit if Ctrl-D received from target
- `--dump=OPTS` dump extra debug information for OPTS
- `--gmon` write gmon.out file if seen from target
- `--no-gmon` do not write gmon.out

#### Connection Options

- `-H`, `--hardware` connect to a hardware target (default)
- `--persistent` try to reconnect after an I/O error
- `--no-persistent` opposite of persistent

#### Connect to Hardware Options

- `-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)

**Note:** If a development board has more than one JTAG port, the `nios2-terminal` command cannot work unless the correct JTAG cable is identified. In order to identify the correct JTAG cable, you must first run the `jtagconfig` command. For example, if the output states that USB-Blaster is port 2, then you can run the command using the correct JTAG port `nios2-terminal -c 2`.
validate_zip

Description
This utility checks that the zip file you have chosen is compatible with the Read Only Zip Filing system. It returns 0 if the zip file is compatible, -1 if not.

Usage
validate_zip name_of_zip_file

nios2-configure-sof

Description
This tool configures an configurable part. If no explicit SOF file is provided then it tries and work out the correct file to use.

Usage
nios2-configure-sof [-h/--help] [-C/--directory <dir name>] [-c/--cable <cable name>] [-d/--device <device index>] [ <filename.sof> ]

Options
-h/--help Prints this message.
-C/--directory <dir name> Change to this directory before running
-c/--cable <cable name> Specifies which JTAG cable to use (not needed if you only have one cable).
-d/--device <device index> Specifies in which device you want to look for the CPU (1 = device nearest TDI etc.)
<filename.sof> The SOF file you want to download

If you have exactly one cable attached then nios2-configure-sof uses this cable by default. If you have more than one cable attached then you must specify one. Use `jtagconfig` to find out which cables you have available. If a SOF file is specified, then that file is configured into the part. If no file is specified, or if the specified filename does not have a path associated with it then the tool searches for matching SOF files, first in the current directory and then in the directory which contains the PTF file for the project in the current directory.

jtagconfig

Description
Allows you configure the JTAG server on the host machine. It can also detect a JTAG chain and set up the download hardware configuration.

Usage
jtagconfig [--enum]
jtagconfig --add <type> <port> [ <name> ]

Software Build Tools

Altera Corporation

Send Feedback
Options

```bash
jtagconfig --remove <cable>
jtagconfig --getparam <cable> <param>
jtagconfig --setparam <cable> <param> <value>
jtagconfig --define <jtagid> <name> <irlength>
jtagconfig --undefine <jtagid> <name> <irlength>
jtagconfig --defined
jtagconfig --addserver <server> <password>
jtagconfig --enableremote <password>
jtagconfig --disableremote
jtagconfig --version
jtagconfig --help
jtagconfig --extrahelp
```

---

**--version | -v**  
Displays the version of the jtagconfig utility you are using.

**--help | -h**  
Displays the options for the jtagconfig utility.

**--extrahelp**  
Displays this help information.

**--enum**  
Displays the JTAG ID code for the devices in each JTAG chain attached to programming hardware. If the JTAG Server recognizes a device, it displays the device name. If the JTAG Server cannot use a device, it marks the device with an exclamation mark (!).

The following code is an example of output from the 'jtagconfig --enum' command:

```
1) ByteBlasterMV on LPT1
   090010DD   EPXA10
   049220DD   EPXA_ARM922
   04000101   !
   090000DD   ! EP20K1000E
   04056101   !
   010020DD   EPC2
   010020DD   EPC2
   010020DD   EPC2
```

In this example, the JTAG Server does not recognize two of the eight devices, and cannot use three devices. If you need to use one of the unused devices then use the --define option to define unused devices with JTAG IDs.

You can also use this option by typing the command 'jtagconfig' without an argument.

**--add <hardware> <port> [<name>]**  
Specifies the port to which you attached new programming hardware. For example, 'jtagconfig --add byteblastermv lpt1' specifies that you attached a ByteBlasterMV cable to port LPT1. You can also use '![<name>]' to add a string that describes the hardware to the output from the 'jtagconfig --enum' command.

The JTAG Server automatically detects cables that you attach to a USB port, therefore you do not need to use this command for these cables.

**--remove <cable>**  
Removes the hardware setup that is indicated with the number listed for the hardware in the output of the '--enum' command. In the example for the '--enum' command, above, the number '1') is listed for the ByteBlasterMV cable. Therefore, the command-line 'jtagconfig --remove 1' removes the hardware setup for the ByteBlasterMV cable.
If the hardware specified is attached to a remote JTAG server then the connection to the remote JTAG server is removed instead.

```
--getparam <cable> <param>
Get a hardware specific parameter from the cable specified. The parameters supported vary by cable type, see the documentation for your cable for details.

--setparam <cable> <param> <value>
Set a hardware specific parameter on the cable specified. The parameters supported vary by cable type, see the documentation for your cable for details. This command can only set numeric parameters (the suffixes k and M are recognized when parsing the value).

--define <jtagid> <name> <irlength>
Tells the JTAG server about the name and IR length of a new device. This is stored and used in future enumerations to reduce the number of devices which cannot be used.

--undefine <jtagid> <name>
Tells the JTAG server to remove information about the name and IR length of a device.

--defined
Displays the JTAG IDs, names and IR lengths of all non-Intel FPGA devices known about by the JTAG server.

--addserver <servername> <password>
Tells Quartus to connect to the remote JTAG server and make all cables on that server available to local applications. An IP address or DNS name may be used to specify the server to connect to. The password given here must match the password used on the remote server.

--enableremote <password>
Tells the JTAG server to allow connections from remote machines. These machines must specify the same password when connecting.

--disableremote
Tells the JTAG server not to accept any more connections from remote machines. Remote connections currently in use are not terminated.
```

**Note:** If a development board has more than one JTAG port, you must run this command first to identify the correct JTAG cable. For example, you have to run this command to identify the correct JTAG cable and then run the `nios2-terminal` command using the correct JTAG port: `nios2-terminal -c 2`.

### Restrictions

The SBT supports BSPs incorporating the Intel FPGA HAL and Micrium MicroC/OS-II only.
The HAL is a lightweight embedded runtime environment that provides a simple device driver interface for programs to connect 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()`, `fwrite()`, etc.

The HAL serves as a device driver package for processor systems, providing a consistent interface to the peripherals in your system. The software development tools extract system information from your SOPC Information File (`.sopcinfo`). The Software Build Tools (SBT) generate a custom HAL board support package (BSP) specific to your hardware configuration. Changes in the hardware configuration automatically propagate to the HAL device driver configuration. As a result, changes in the underlying hardware are prevented from creating bugs.

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.

**Getting Started with the Hardware Abstraction Layer**

The easiest way to get started using the HAL is to create a software project. In the process of creating a new project, you also create a HAL BSP. You do not need to create or copy HAL files, nor edit any of the HAL source code. The SBT generates the HAL BSP for you.

For an example on creating a simple HAL software project, refer to the “Getting Started with Software in Eclipse” chapter.

In the SBT command line, you can create an example BSP based on the HAL using one of the `create-this-bsp` scripts supplied with the Nios II Embedded Design Suite.

You must base the HAL on a specific hardware system. A system consists of a processor core integrated with peripherals and memory. Systems are generated by .

If you do not have a custom system, you can base your project on an Intel FPGA-provided example hardware system. In fact, you can first start developing projects targeting an Intel FPGA development board, and later re-target the project to a custom board. You can easily change the target hardware system later.

For more information about creating a new project with the SBT command line, refer to either:
HAL Architecture for Embedded Software Systems

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

Layers of a HAL-Based System

The Layers of a HAL-Based System

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 through 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.

For further details about the HAL, refer to the "Developing Programs Using the Hardware Abstraction Layer" and "Developing Device Drivers for the Hardware Abstraction Layer" chapters.

Related Information
- Developing Programs Using the Hardware Abstraction Layer on page 6-1
- Developing Device Drivers for the Hardware Abstraction Layer on page 7-1

Generic Device Models

The HAL provides generic device models for classes of peripherals found in embedded systems, such as timers, Ethernet MACPHY 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.

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 such as the Intel FPGA-provided NicheStack TCP/IP Stack - Edition. 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—Nonvolatile memory devices that use a special programming protocol to store data.

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.

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.

C Standard Library—newlib

The HAL integrates the ANSI C standard library in its runtime environment. The HAL uses newlib, 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 processor. newlib licensing does not require you to release your source code or pay royalties for projects based on newlib.

Note: For newlib, newlib is upgraded to version 1.18.

The ANSI C standard library is well documented.

For more information about the most well-known reference of the ANSI C standard library, refer to the book: The C Programming Language by B. Kernighan and D. Ritchie, published by Prentice Hall. It is also available in over 20 languages.

Related Information
Redhat Website
For more information, refer to the online documentation for newlib by Red Hat.

Embedded Hardware Supported by the HAL

Processor Core Support

The HAL supports all available processor core implementations.

Supported Peripherals

Provide Full HAL Support

Intel FPGA provides many peripherals for use in Nios II processor systems. Most Intel FPGA peripherals provide HAL device drivers that allow you to access the hardware with the HAL API. The following Intel FPGA peripherals provide full HAL support:

- Character mode devices
  - UART core
  - JTAG UART core
  - LCD 16207 display controller
- Flash memory devices
  - Common flash interface compliant flash chips
  - Intel FPGA’s erasable programmable configurable serial (EPCS) serial configuration device controller
- File subsystems
  - Intel FPGA host based file system
  - Intel FPGA read-only zip file system
- Timer devices
  - Timer core
- DMA devices
  - DMA controller core
  - Scatter-gather DMA controller core
- Ethernet devices
  - Triple Speed Ethernet IP core® function
  - Standard Microchip Solutions (SMSC) LAN91C111 10/100 Non-PCI Ethernet Single Chip MAC + PHY
  - EPCQ soft IP peripheral
    - Upgraded to add support for x4 mode and L devices, giving faster access to the EPCQ device from Nios or other FPGA based masters

The LAN91C111 and Triple Speed Ethernet components require the MicroC/OS-II runtime environment. For more information, refer to the "Ethernet and the NicheStack TCP/IP Stack - Edition" chapter.

**Note:** Third-party vendors offer additional peripherals not listed here.

**Related Information**
- [Ethernet and the NicheStack TCP/IP Stack](#) on page 11-1

**Provide Partial HAL Support**

All peripherals (both from Intel FPGA and third party vendors) must provide a header file that defines the peripheral’s low-level interface to hardware. Therefore, all peripherals support the HAL to some extent. However, some peripherals might not provide device drivers. If drivers are not available, use only the definitions provided in the header files to access the hardware. Do not use unnamed constants, such as hard-coded addresses, to access a peripheral.

Inevitably, certain peripherals have hardware-specific features with usage requirements that do not map well to a general-purpose API. The HAL handles hardware-specific requirements by providing the UNIX-style `ioctl()` function. Because the hardware features depend on the peripheral, the `ioctl()` options are documented in the description for each peripheral.

Some peripherals provide dedicated accessor functions that are not based on the HAL generic device models. For example, Intel FPGA provides a general-purpose parallel I/O (PIO) core for use with the Nios II processor system. The PIO peripheral does not fit in any class of generic device models provided by the HAL, and so it provides a header file and a few dedicated accessor functions only.

For complete details regarding software support for a peripheral, refer to the peripheral’s description.

For more information about Intel FPGA-provided peripherals, refer to the [Embedded Peripherals IP User Guide](#).

**Related Information**
- [Embedded Peripherals IP User Guide](#)

**MPU Support**

The HAL does not include explicit support for the optional memory protection unit (MPU) hardware. However, it does support an advanced exception handling system that can handle MPU exceptions.

For more information about handling MPU and other advanced exceptions, refer to the [Exception Handling section](#).
For more information about the MPU hardware implementation, refer to the Programming Model chapter of the Processor Reference Handbook.

Related Information

- Exception Handling on page 8-1
  For more information about handling MPU and other advanced exceptions.
- Programming Model
  For more information about the MPU hardware implementation.

**MMU Support**

The HAL does not support the optional memory management unit (MMU) hardware. To use the MMU, you need to implement a full-featured operating system.

For more information about the MMU, refer to the "Programming Model" chapter of the Processor Reference Handbook.

Related Information

- Programming Model
  For more information about the Nios II MMU.
This chapter discusses how to develop embedded programs for the Nios® II embedded processor based on the Intel FPGA® hardware abstraction layer (HAL). The application program interface (API) for HAL-based systems is readily accessible to software developers who are new to the 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()`.


**HAL BSP Settings**

Every board support package (BSP) has settings that determine the BSP’s characteristics. For example, HAL BSPs have settings to identify the hardware components associated with standard devices such as `stdout`. Defining and manipulating BSP settings is an important part of project creation. You manipulate BSP settings with the BSP Editor, with command-line options, or with Tcl scripts.

**Note:** For details about how to control BSP settings, refer to one or more of the following documents:

- For more information about the SBT for Eclipse, refer to the ”Getting Started with the Graphical User Interface” chapter.
- For more information about the SBT command line, refer to the ”Software Build Tools” chapter.

For more information about detailed descriptions of available BSP settings, refer to the ”Software Build Tools Reference” chapter.

Many HAL settings are reflected in the `system.h` file, which provides a helpful reference for details about your BSP.

For more information about `system.h`, refer to the “The system.h System Description File” chapter.

**Note:** Do not edit `system.h`. The EDS provides tools to manipulate system settings.

**Related Information**

- [The system.h System Description File](#) on page 6-3
The Embedded Project Structure

The creation and management of software projects based on the HAL is integrated tightly with the SBT. This section discusses the projects as a basis for understanding the HAL.

**Note:** The label for each block describes what or who generated that block, and an arrow points to each block’s dependency.

**Figure 6-1: The HAL Project Structure Emphasizing How the HAL BSP Fits In**

Every HAL-based program consists of two projects.

For more information, refer to the figure above. Your application-specific code is contained in one project (the user application project), and it depends on a separate BSP project (the HAL BSP).

The application project contains all the code you develop. The executable image for your program ultimately results from building both projects.

With the SBT for Eclipse, the tools create the HAL BSP project when you create your application project. In the SBT command line flow, you create the BSP using `nios2-bsp` or a related tool.

The HAL BSP project contains all information needed to interface your program to the hardware. The HAL drivers relevant to your hardware system are incorporated in the BSP project.

The BSP project depends on the hardware system, defined by a SOPC Information File (.sopcinfo). The SBT can keep your BSP up-to-date with the hardware system. This project dependency structure isolates your program from changes to the underlying hardware, and you can develop and debug code without concern about whether your program matches the target hardware.

You can use the SBT to update your BSP to match updated hardware. You control whether and when these updates occur.

For more information about how the SBT keeps your BSP up-to-date with your hardware system, refer to “Revising Your BSP” in the "Software Build Tools" chapter.
In summary, when your program is based on a HAL BSP, you can always keep it synchronized with the target hardware with a few simple SBT commands.

Related Information

- Software Build Tools Reference on page 15-1
- Embedded Software Projects on page 4-4

The system.h System Description File

The system.h file provides a complete software description of the 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 SBT 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. The SBT provides facilities to manipulate system settings.

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

Example 6–1. Excerpts from a system.h File Detailing Hardware Configuration Options

```c
/*
 * 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
```

Related Information

HAL BSP Settings on page 6-1

Data Widths and the HAL Type Definitions

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

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

Table 6-1: The HAL Type Definitions

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>alt_8</td>
<td>Signed 8-bit integer.</td>
</tr>
<tr>
<td>alt_u8</td>
<td>Unsigned 8-bit integer.</td>
</tr>
<tr>
<td>alt_16</td>
<td>Signed 16-bit integer.</td>
</tr>
<tr>
<td>alt_u16</td>
<td>Unsigned 16-bit integer.</td>
</tr>
<tr>
<td>alt_32</td>
<td>Signed 32-bit integer.</td>
</tr>
<tr>
<td>alt_u32</td>
<td>Unsigned 32-bit integer.</td>
</tr>
<tr>
<td>alt_64</td>
<td>Signed 64-bit integer.</td>
</tr>
<tr>
<td>alt_u64</td>
<td>Unsigned 64-bit integer.</td>
</tr>
</tbody>
</table>

Table 6-2: GNU Toolchain Data Widths

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8 bits.</td>
</tr>
<tr>
<td>short</td>
<td>16 bits.</td>
</tr>
<tr>
<td>long</td>
<td>32 bits.</td>
</tr>
<tr>
<td>int</td>
<td>32 bits.</td>
</tr>
</tbody>
</table>

UNIX-Style Interface

The HAL API provides a number of UNIX-style functions. The UNIX-style functions provide a familiar development environment for new programmers, 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()`
The most commonly used functions are those that relate to file I/O.

For more information, refer to the “File System” chapter.

For more information about the use of these functions, refer to the "HAL API Reference" chapter.

Related Information

- **File System** on page 6-5
- **HAL API Reference** on page 14-1

**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, 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 these 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.

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

The HAL file infrastructure also allows you to manipulate character mode devices 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 \texttt{uart1} in is named \texttt{/dev/uart1} in \texttt{system.h}.

\textbf{Note:} The standard header files \texttt{stdio.h}, \texttt{stddef.h}, and \texttt{stdlib.h} are installed with the HAL.

```c
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#define BUF_SIZE (10)
int main(void)
{
    FILE* fp;
    char buffer[BUF_SIZE];
    fp = fopen ("/mount/rozipfs/test", "r"); if (fp == NULL)
    {
        printf ("Cannot open file.\n");
        exit (1);
    }
    fread (buffer, BUF_SIZE, 1, fp);
    fclose (fp);
    return 0;
}
```

For more information about the use of these functions, refer to the newlib C library documentation installed with the EDS. To access the documentation, go to the Windows Start menu, click \texttt{Programs > Intel FPGA > > Documentation}.

\textbf{Related Information}

- \texttt{HAL API Reference} on page 14-1
- \texttt{Read-Only Zip File System} on page 12-1

\section*{Using Character-Mode Devices}

A character-mode device is a hardware peripheral that either or both 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 \texttt{file.h}. The HAL also supports the concept of standard input, standard output, and standard error, allowing programs to call the \texttt{stdio.h} I/O functions.

\section*{Standard Input, Standard Output and Standard Error}

Using standard input (\texttt{stdin}), standard output (\texttt{stdout}), and standard error (\texttt{stderr}) is the easiest way to implement simple console I/O. The HAL manages \texttt{stdin}, \texttt{stdout}, and \texttt{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 \texttt{printf()} to standard out, and \texttt{perror()} to standard error. You associate each channel to a specific hardware device by manipulating BSP settings.

\textbf{Note:} This program sends characters to whatever device is associated with \texttt{stdout} when the program is compiled.
Example 6–3. Hello World

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

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

## 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 6–4. Writing Characters to a UART Called uart1

```c
#include <stdio.h>
#include <string.h>
int main (void)
{
    char* msg = "hello world";
    FILE* fp;
    fp = fopen ("/dev/uart1", "w");
    if (fp!=NULL)
    {
        fprintf(fp, "%s",msg);
        fclose (fp);
    }
    return 0;
}
```

### C++ Streams

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

/dev/null

All systems include the device /dev/null. Writing to /dev/null has no effect, and all data is discarded. /dev/null is used for safe I/O redirection during system startup. This device can also be useful for applications that wish to sink unwanted data.

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

### Lightweight Character-Mode I/O

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

For more information, refer to the “Reducing Code Footprint in Embedded Systems” chapter.

Related Information

Reducing Code Footprint in Embedded Systems on page 6-27

### 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 stdout 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.

**Note:** 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.

### Enabling Logging

Logging is enabled by setting `hal.log_port` to a JTAG UART or a UART device. 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 `public.mk` and the `ALT_LOG_FLAGS` flag is set to 0, the default value.

**Note:** You can set `ALT_LOG_FLAGS` to any value from 1 to 4, to determine the level of output.

**Note:** If `hal.log_port` is not set, then `ALT_LOG_ENABLE` and `ALT_LOG_FLAGS` do not appear in `public.mk`.

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 6–5. Default Boot Logging Output

```plaintext
[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 application completes initialization, run the `nios2-terminal` command from the Command Shell to accept the logging output.

**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 6-3: Intel FPGA Logging Options and Option Modifiers**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>System clock log</strong></td>
<td>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.</td>
</tr>
<tr>
<td>Preprocessor symbol</td>
<td><code>ALT_LOG_SYS_CLK_ON_FLAG_SETTING</code></td>
</tr>
<tr>
<td>Modifiers</td>
<td>The system clock log has two modifiers, providing two different ways to specify the logging interval.</td>
</tr>
<tr>
<td></td>
<td>• <code>ALT_LOG_SYS_CLK_INTERVAL</code>—Specifies the logging interval in system clock ticks. The default is <code>&lt;clock ticks per second&gt;</code>, that is, one second.</td>
</tr>
<tr>
<td></td>
<td>• <code>ALT_LOG_SYS_CLK_INTERVAL_MULTIPLIER</code>—Specifies the logging interval in seconds. The default is 1. When you modify <code>ALT_LOG_SYS_CLK_INTERVAL_MULTIPLIER</code>, <code>ALT_LOG_SYS_CLK_INTERVAL</code> is recalculated.</td>
</tr>
</tbody>
</table>
| Sample Output         | System Clock On 0  
                       | System Clock On 1                                                                                                                              |

<p>| <strong>Write echo</strong>        | Every time <code>alt_write()</code> is called (normally, whenever characters are sent to stdout), the first <code>&lt;n&gt;</code> characters are echoed to a logging message. The message starts with the string &quot;Write Echo: &quot;. <code>&lt;n&gt;</code> is specified with <code>ALT_LOG_WRITE_ECHO_LEN</code>. The default is 15 characters. |
| Preprocessor symbol   | <code>ALT_LOG_WRITE_ON_FLAG_SETTING</code>                                                                                                             |
| Modifiers             | <code>ALT_LOG_WRITE_ECHO_LEN</code>—Number of characters to echo. Default is 15.                                                                       |
| Sample Output         | Write Echo: Hello from Nio                                                                                                                  |</p>
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Preprocessor symbol</th>
<th>Modifiers</th>
<th>Sample Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>JTAG startup log</strong></td>
<td>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 &quot;SW CirBuf&quot;, might be negative, because it is computed as ((\text{&lt;tail_pointer&gt;} - \text{&lt;head_pointer&gt;})) on a circular buffer. For more information about the JTAG UART control register fields, refer to the Embedded Peripherals IP User Guide.</td>
<td>ALT_LOG_JTAG_UART_STARTUP_INFO_ON_FLAG_SETTING</td>
<td>None</td>
<td>JTAG Startup Info: SW CirBuf = 0, HW FIFO wspace=64 AC=0 WI=0 RI=0 WE=0 RE=1</td>
</tr>
<tr>
<td><strong>JTAG interval log</strong></td>
<td>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.</td>
<td>ALT_LOG_JTAG_UART_ALARM_ON_FLAG_SETTING</td>
<td>None</td>
<td>JTAG Alarm: SW CirBuf = 0, HW FIFO wspace=45 AC=0 WI=0 RI=0 WE=0 RE=1</td>
</tr>
<tr>
<td><strong>JTAG interrupt service routine (ISR) log</strong></td>
<td>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.</td>
<td>ALT_LOG_JTAG_UART_ISR_ON_FLAG_SETTING</td>
<td>None</td>
<td>JTAG IRQ: SW CirBuf = -20, HW FIFO wspace=64 AC=0 WI=1 RI=0 WE=1 RE=1</td>
</tr>
</tbody>
</table>
### 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.

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preprocessor symbol</td>
</tr>
<tr>
<td>ALT_LOG_BOOT_ON_FLAG_SETTING</td>
</tr>
<tr>
<td>Modifiers</td>
</tr>
<tr>
<td>None</td>
</tr>
<tr>
<td>Sample Output</td>
</tr>
<tr>
<td>For more information, refer to the “Enabling Intel FPGA Logging” chapter.</td>
</tr>
</tbody>
</table>

**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
- Enabling Logging on page 6-8

### 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 6-4: ALT_LOG_FLAGS Logging Levels

<table>
<thead>
<tr>
<th>Logging Level</th>
<th>Logging</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Boot log (default)</td>
</tr>
<tr>
<td>1</td>
<td>Level 0 plus system clock log and JTAG startup log</td>
</tr>
<tr>
<td>2</td>
<td>Level 1 plus JTAG interval log and write echo</td>
</tr>
<tr>
<td>3</td>
<td>Level 2 plus JTAG ISR log</td>
</tr>
<tr>
<td>-1</td>
<td>Silent mode—No Intel FPGA logging</td>
</tr>
</tbody>
</table>

**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.
Example: Creating a BSP with Logging

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

Example 6–6. Creating BSP With Logging and Options

```
nios2-bsp hal my_bsp ../my_hardware.sopcinfo \
   --set hal.log_port uart1 \
   --set hal.make.bsp_cflags_user_flags \n   -DAL_T_LOG_FLAGS=2 \n   -DAL_T_LOG_WRITE_ON_FLAG_SETTING=0 \n   -DAL_T_LOG_JTAG_UART_TICKS_DIVISOR=2r
```

The -DAL_T_LOG_FLAGS=2 argument adds -DAL_T_LOG_FLAGS=2 to the ALT_CPP_FLAGS make variable in public.mk.

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 is 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 (Table 6–4), or the "Intel FPGA Logging Options and Option Modifiers" table (Table 6–3 on page 6–10).

Example 6–7. 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
```

Intel FPGA Logging Files

Table 6-5: HAL Implementation Files for Intel FPGA Logging

<table>
<thead>
<tr>
<th>Location</th>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>components/altera_hal/HAL/inc/sys/</td>
<td>alt_log_printf.h</td>
</tr>
</tbody>
</table>
### Table 6-6: HAL Example Files for Intel FPGA Logging

<table>
<thead>
<tr>
<th>Location</th>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>components/altera_hal/HAL/src/</td>
<td>alt_log_printf.c</td>
</tr>
<tr>
<td>components/altera_nios2/HAL/src/</td>
<td>alt_log_macro.S</td>
</tr>
<tr>
<td>components/altera_avalon_jtag_uart/HAL/src/</td>
<td>altera_avalon_jtag_uart.c</td>
</tr>
<tr>
<td>components/altera_avalon_timer/HAL/src/</td>
<td>altera_avalon_timer_sc.c</td>
</tr>
<tr>
<td>components/altera_hal/HAL/src/</td>
<td>alt_exit.c</td>
</tr>
<tr>
<td>components/altera_hal/HAL/src/</td>
<td>alt_main.c</td>
</tr>
<tr>
<td>components/altera_hal/HAL/src/</td>
<td>alt_write.c</td>
</tr>
<tr>
<td>components/altera_nios2/HAL/src/</td>
<td>crt0.S</td>
</tr>
</tbody>
</table>

**Note:** All file locations are relative to `<Nios II EDS install path>`.

These files implement the logging options listed in the "Intel FPGA Logging Options and Option Modifiers" table (Table 6–3 on page 6–10). They also serve as examples of logging usage.

---

**Using File Subsystems**

The HAL generic device model for file subsystems allows access to data stored in an associated storage device using the C standard library file I/O functions. For example, the Intel FPGA 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()`.

For more information about the use of file I/O functions, refer to the newlib C library documentation installed with the Nios II EDS. On the Windows Start menu, click **Programs > Intel FPGA > <version> > EDS <version> Documentation**.

**Host-Based File System**

The host-based file system enables programs executing on a target board to read and write files stored on the host computer. The SBT for Eclipse transmits file data over the Intel FPGA download cable. Your program accesses the host based file system using the ANSI C standard library I/O functions, such as `fopen()` and `fread()`. The host-based file system is a software package which you add to your BSP.
The following features and restrictions apply to the host based file system:

- The host-based file system makes the C/C++ application project directory and its subdirectories available to the HAL file system on the target hardware.
- The target processor can access any file in the project directory. Be careful not to corrupt project source files.
- The host-based file system only operates while debugging a project. It cannot be used for run sessions.
- Host file data travels between host and target serially through the Intel FPGA download cable, and therefore file access time is relatively slow. Depending on your host and target system configurations, it can take several milliseconds per call to the host. For higher performance, use buffered I/O function such as fread() and fwrite(), and increase the buffer size for large files.

You configure the host-based file system using the BSP Editor. The host-based file system has one setting: the mount point, which specifies the mount point within the HAL file system. For example, if you name the mount point /mnt/host and the project directory on your host computer is /software/project1, in a HAL-based program, the following code opens the file /software/project1/datafile.dat:

```c
fopen("/mnt/host/datafile.dat", "r");
```

### Using Timer Devices

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

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.

An individual timer peripheral can behave as either a system clock or a timestamp, 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.

### 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. For embedded engineers who deal with both hardware and software, do not confuse the HAL system clock with the clock signal driving the processor hardware. The period of a HAL system clock tick is generally much longer than the hardware system clock. `system.h` defines the clock tick frequency.

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

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

For more information about the use of these functions, refer to the "HAL API Reference" chapter.

Related Information

- **HAL BSP Settings** on page 6-1
- **HAL API Reference** on page 14-1

## 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()`:

```c
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.

For more information about the use of these functions, refer to the "Exception Handling" chapter.

### Example 6–8. Using a Periodic Alarm Callback Function

```c
#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
Exception Handling on page 8-1

Timestamp Driver

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

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

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

- alt_timestamp_start()
- alt_timestamp()

Calling alt_timestamp_start() starts the counter running. Subsequent calls to alt_timestamp() return 
the current value of the timestamp counter. Calling alt_timestamp_start() again resets the counter to 
zero. The behavior of the timestamp driver is undefined when the counter reaches $(2^{32} - 1)$.

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

For more information about the use of these functions, refer to the HAL API Reference section.

Example 6–9. Using the Timestamp to Measure Code Execution Time

```c
#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));
```
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, such as the Intel FPGA erasable programmable configurable serial (EPCS) configuration device, you must use the HAL API to both read and write data.

This section describes the HAL API for the flash device model. The following two APIs provide 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`. For more information about the use of these functions, refer to the HAL API Reference section.

For more information about the Common Flash Interface, including the organization of common flash interface (CFI) erase regions and blocks, refer to the JEDEC website.

For more information about the CFI standard, refer to the JEDEC website and search for document JESD68.

Related Information

- HAL API Reference on page 14-1
- JEDEC Website

For more information about the Common Flash Interface standard, including the organization of common flash interface (CFI) erase regions and blocks, refer to the JEDEC website and search for document JESD68.

Simple Flash Access

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

printf ("Number of ticks per second = %u\n",
        (unsigned int)alt_timestamp_freq());
    return 0;
}
For more information about the use of all of these functions in one code example, refer to the code in the “Using the Simple Flash API Functions to Access a Flash Device Named /dev/ext/flash” example in the "Fine-Grained Flash Access" section.

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:

```c
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:

```c
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:

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

Related Information
Fine-Grained Flash Access on page 6-19

**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 (Table 6–7) shows the example of an 8-kilobyte (KB) flash memory comprising two 4-KB blocks. First write 5 KB of all 0xAA to flash memory at address 0x0000, and then write 2 KB of all 0xBB to address 0x1400. 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). 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. After the second write finishes, at time t(4), the 2 KB of 0xFF at address 0x1000 is corrupted.
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.

```c
#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;
    }
}
```
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:

```c
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 6-7: Example of Writing Flash and Causing Unexpected Data Corruption

<table>
<thead>
<tr>
<th>Address</th>
<th>Block</th>
<th>Before First Write</th>
<th>Time t(0)</th>
<th>Time t(1)</th>
<th>Time t(2)</th>
<th>Time t(3)</th>
<th>Time t(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>1</td>
<td>Unknown</td>
<td>FF</td>
<td>AA</td>
<td>AA</td>
<td>AA</td>
<td></td>
</tr>
<tr>
<td>0x0400</td>
<td>1</td>
<td>Unknown</td>
<td>FF</td>
<td>AA</td>
<td>AA</td>
<td>AA</td>
<td></td>
</tr>
<tr>
<td>0x0800</td>
<td>1</td>
<td>Unknown</td>
<td>FF</td>
<td>AA</td>
<td>AA</td>
<td>AA</td>
<td></td>
</tr>
<tr>
<td>0x0C00</td>
<td>1</td>
<td>Unknown</td>
<td>FF</td>
<td>AA</td>
<td>AA</td>
<td>AA</td>
<td></td>
</tr>
<tr>
<td>0x1000</td>
<td>2</td>
<td>Unknown</td>
<td>FF</td>
<td>AA</td>
<td>FF</td>
<td>FF(6)</td>
<td></td>
</tr>
<tr>
<td>0x1400</td>
<td>2</td>
<td>Unknown</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>BB</td>
<td></td>
</tr>
<tr>
<td>0x1800</td>
<td>2</td>
<td>Unknown</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>BB</td>
<td></td>
</tr>
<tr>
<td>0x1C00</td>
<td>2</td>
<td>Unknown</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td></td>
</tr>
</tbody>
</table>

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

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

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

alt_erase_flash()

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

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

Unintentionally cleared to FF during erasure for second write.
The flash memory is identified by the handle `fd`. The block is identified as being `offset` bytes from the beginning of the flash memory, and the block size is passed in `length`.

**alt_write_flash_block()**

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

```c
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 6–11. Using the Fine-Grained Flash Access API Functions

```c
#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;
}
```
**alt_lock_flash()**

**Prototype**

```c
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**

```c
<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
- `-EINVAL` > Invalid arguments
- `-ETIME` > Time out and skipping the looping after 0.7 sec
- `-ENOLCK` > Sectors lock failed
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 6-2: Three Basic Types of DMA Transactions](image)

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

For more information about the use of these functions, refer to the *HAL API Reference* section.

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

For more information about cache memory, refer to the *Cache and Tightly-Coupled Memory* section.

Related Information
- **HAL API Reference** on page 14-1
- **Cache and Tightly-Coupled Memory** on page 9-1
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 6–12. Obtaining a File Handle for a DMA Transmit Device dma_0

```c
#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:

```c
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`.

For more information, refer to the HAL API Reference section.

Related Information
HAL API Reference on page 14-1

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
The prototype for `alt_dma_rxchan_prepare()` is:

```c
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.

```c
#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);
    }
}
```
Related Information

HAL API Reference on page 14-1

Memory-to-Memory DMA Transactions

```c
#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++;
}
/*
 * Post the transmit request */
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 = altdma_txchan_send (txchan,
        tx_data,
        128,
        NULL,
```
Using Interrupt Controllers

The HAL supports two types of interrupt controllers:

- The internal interrupt controller
- An external interrupt controller component

For more information about working with interrupt controllers, refer to the "Exception Handling" chapter.

Related Information
Exception Handling on page 8-1

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 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. The hello_world_small example project demonstrates the use of some of these options to reduce code size to the absolute minimum.

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
HAL BSP Settings on page 6-1
Enable Compiler Optimizations

A number of nios2-elf-gcc compiler switches may be of use when optimizing code for small RAM footprints. These switches may be set directly on the compiler command line (typically in a Makefile, in the CFLAGS definition) or by the BSP GUI editor settings.

- **-fno-delete-null-pointer-checks:** (IMPORTANT!) You should always set this switch if your design includes valid data at address zero. If you do not, nios2-elf-gcc may silently produce incorrect code, since the C standard specifies that a NULL pointer must never be dereferenced. GCC v. 4.9 is particularly prone to this.
- **-Os:** This is the most important compiler switch when optimizing for space. This instructs nios2-elf-gcc to pervasively optimize for space rather than speed. (But you may wish to use -Og when debugging. This generates code easier to understand in the debugger.)
- **-fno-exceptions:** If you are coding a small RAM footprint program in C++ (risky, because C++ tends to add unwanted overhead) and attempting to avoid linking in C++ exception handling code to save space, this switch can dissuade nios2-elf-g++ from linking in the exception handling code “just to be safe” in cases where it is not sure whether exception handling is needed.
- **-mgpopt=global:** This switch improves code in both space and speed relative to the -mgpopt=local default. You should always use -mgpopt=global unless you using different -Gn compile switch settings for different compilation units, which should seldom if ever be the case.
- **-mgpopt=data:** This improves code in both space and speed relative to -mgpopt=global if all of your data fits in 64 KB of address space, which is often the case for small embedded projects. You may need to edit your linker script to ensure that _gp is defined to be 32 KB beyond the start of your 64 KB of data. (You may use the "-Wl,-verbose" compiler switch to display your current linker script and the "-T" compiler switch to specify a different linker script.)
- **-gpopt=all:** This improves code in both space and speed relative to "-mgpopt=data" if all of your data AND code fit in 64 KB of address space. As above, you may need to modify your linker script to set _gp to be 32 KB beyond the start of your 64 KB of combined code and data.
- **-ffunction-sections:** This places each function in a separate code section in the .o file, making it possible for ld to produce smaller executables by not linking in unused functions. This results in larger .o files (due to the increased number of code section headers) and in slower links (since ld must process the increased number of code section headers) but that is rarely an issue on contemporary machines.
- **-ffunction-sections:** As above, but placing each global data item in its own .o file section.

Related Information

**Specifying BSP Defaults** on page 4-34

Use Reduced 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. The fast variant is full-featured, and the small variant provides a reduced code footprint.

By default the HAL always uses the fast driver variants. You can select the reduced device driver for all hardware components, or for an individual component, through HAL BSP settings.

The small footprint option might also affect other peripherals. Refer to each peripheral's data sheet for complete details of its driver's small footprint behavior.
Table 6-8: Intel FPGA Peripherals Offering Small Footprint Drivers

<table>
<thead>
<tr>
<th>Peripheral</th>
<th>Small Footprint Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>UART</td>
<td>Polled operation, rather than IRQ-driven</td>
</tr>
<tr>
<td>JTAG UART</td>
<td>Polled operation, rather than IRQ-driven</td>
</tr>
<tr>
<td>Common flash interface controller</td>
<td>Driver excluded in small footprint mode</td>
</tr>
<tr>
<td>LCD module controller</td>
<td>Driver excluded in small footprint mode</td>
</tr>
<tr>
<td>EPCS serial configuration device</td>
<td>Driver excluded in small footprint mode</td>
</tr>
</tbody>
</table>

Reduce the File Descriptor Pool

The file descriptors that access character mode devices and files are allocated from a file descriptor pool. You can change the size of the file descriptor pool through a BSP setting. The default is 32.

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 null for stdin, stdout, and stderr. This selection assumes that you want to discard all data transmitted on standard out or standard error, and your program never receives input through stdin. You can control the assignment of stdin, stdout, and stderr channels by manipulating BSP settings.

Use a Smaller File I/O Library

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, unkown 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()
fprintf()
iprintf()
asiprintf()
asnprintf()
sniprintf()
siprintf()

The fixed-number-of-argument (vararg-based) variants are:

vprintf()
vasprintf()
vASNPRINTF()
vSNPRINTF()
vprintf()
vSNPRINTF()
viprintf()
vVIPRINTF()
vasiprintf()
vSNIPRINTF()
vSNPRINTF()

*s*printf (mnemonic: prints to string buffer)

The canonical printf() function prints to an output stream; the *s*printf functions instead print into a RAM buffer, allowing the buffered results to be further processed. The functions in this family are:

asprintf()
asnprintf()
snprintf()
sprintf()
vasprintf()
vASNPRINTF()
vSNPRINTF()
vprintf()
vSNPRINTF()
viprintf()
vVIPRINTF()
vasiprintf()
vSNIPRINTF()
vSNPRINTF()
These functions differ in part in how the destination buffer is managed:

- The original functions in this family, like sprintf took a buffer pointer as input and wrote into it. Since the length of the buffer was not given, these functions would frequently overrun the given buffer, resulting in reliability and security issues. The functions in this family are:
  
  ```c
  sprintf()
  vsprintf()
  snprintf()
  vsnprintf()
  ```

- *sn*printf (mnemonic: 'n' for number of bytes in buffer): variants attempted to deal with this problem by passing a buffer length as well as a buffer, allowing the function to avoid overrunning the buffer. This mitigates the security issues but results in incorrect output when the buffer is insufficient. The functions in this family are:
  
  ```c
  snprintf()
  vsnprintf()
  snprintf()
  vsnprintf()
  ```

- *as*printf (mnemonic: 'a' for allocate): variants deal with this problem by malloc()ing and returning the buffer holding the result string. It must then free() the buffer. This avoids the security and reliability issues at the cost of significant added de/allocation overhead. The functions in this family are:
  
  ```c
  asprintf()
  vasprintf()
  asisprintf()
  vasisprintf()
  ```

- *asn*printf: variants try to reduce this overhead by taking a buffer and a limit and realloc()ing it only if the output won’t fit in the provided buffer. The functions in this family are:
  
  ```c
  asnprintf()
  vsnprintf()
  asnprintf()
  vsnprintf()
  ```

*i*printf (mnemonic: 'i' for 'integer-only')

The usual printf() functions support formatting of floating point numbers. Consequently they not only contain a fair amount of logic to do the actual formatting of floating point numbers, but can also link into the executable image such things as a floating point emulation library. If the application does not actually use floating point numbers, and if a small RAM footprint is desired, this can be very counterproductive. Consequently, newlib defines a parallel set of printf functions which lack floating-point formatting support.

The usual set of printf() functions is:

```c
fprintf()
vprintf()
```
printf()
asprintf()
asnprintf()
sprintf()
vasprintf()
vasnprintf()
vsprintf()
vsnprintf()

The parallel set of no-floating-point formatting support printf functions is:

fiprintf()
iprintf()
viprintf()
asiprintf()
asniprintf()
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()
sprintf()
vvasprintf()
vvasnprintf()
vsprintf()
vsnprintf()
fiprintf()
iprintf()
viprintf()
asiprintf()
asniprintf()
vasiprintf()
vasniprintf()
vsiprintf()
vsniprintf()
Additional printf variants supported by newlib libc (but not libsmallc!) are:

- dprintf, vdprintf: Print to a file descriptor (vs FILE*).
- diprintf, vdiprintf: 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**

**The Red Hat newlib C Library**

**Use the Small newlib C Library**

The full newlib library functionality is often unnecessary for embedded systems, and undesirably large for systems needing a minimal RAM footprint. Intel FPGA provides a reduced-functionality reduced-size "Small C" version of newlib which allows smaller RAM footprints to be achieved.

The Intel FPGA "Small C" newlib implementation is selected via the -msmallc command line option to nios2-elf-gcc.

When using the GUI interface, you may select the small newlib library though the "Small C" BSP setting. Table 6-9 summarizes the differences between the "Normal C" and "Small C" newlib library versions. You can select the small newlib library through BSP settings.

**Table 6-9: Comparison of "Small C" and "Normal C" newlib Libraries**

<table>
<thead>
<tr>
<th>Limitation</th>
<th>Functions Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Both Small C and Normal C libraries implement the traditional printf()</td>
<td>asprintf()</td>
</tr>
<tr>
<td>family of routines. However, in Small C, floating point formatting is</td>
<td>fprintf()</td>
</tr>
<tr>
<td>not implemented because the %f and %g options are not supported.</td>
<td>snprintf()</td>
</tr>
<tr>
<td></td>
<td>sprintf()</td>
</tr>
<tr>
<td></td>
<td>vfprintf()</td>
</tr>
<tr>
<td></td>
<td>vsnprintf()</td>
</tr>
<tr>
<td></td>
<td>vsprintf()</td>
</tr>
<tr>
<td>Both Small C and Normal C libraries implement alternate integer-only</td>
<td>asiprintf()</td>
</tr>
<tr>
<td>printf() functions with no %f and %g support.</td>
<td>asniprintf()</td>
</tr>
<tr>
<td></td>
<td>fiprintf()</td>
</tr>
<tr>
<td></td>
<td>iprintf()</td>
</tr>
<tr>
<td></td>
<td>siprintf()</td>
</tr>
<tr>
<td></td>
<td>sniprintf()</td>
</tr>
<tr>
<td></td>
<td>vasiprintf()</td>
</tr>
<tr>
<td></td>
<td>vsiprintf()</td>
</tr>
<tr>
<td></td>
<td>vsnprintf()</td>
</tr>
<tr>
<td>Limitation</td>
<td>Functions Affected</td>
</tr>
<tr>
<td>------------</td>
<td>--------------------</td>
</tr>
</tbody>
</table>
| Wide-character functions are implemented in the Normal C but NOT in the Small C library. | These functions are provided ONLY in the Normal C library:  
  fgetwc()  
  fgetws()  
  fputs()  
  fputs()  
  fwrite()  
  fprintf()  
  fwrite()  
  fscanf()  
  getw()  
  getwc()  
  getwchar()  
  putw()  
  putwc()  
  putwchar()  
  swprintf()  
  ungetwc()  
  vfprintf()  
  vfwprintf()  
  vsprintf()  
  vswprintf()  
  wbuf()  
  wprintf()  
  wcsprintf()  
  vwprintf()  
  wscanf()  
  wsetup() |
| The `scanf()` family of routines is supported in the Normal C but NOT in the Small C library. | These functions are provided ONLY in the Normal C library:  
  fscanf()  
  sscanf()  
  siscanf()  
  scanf()  
  scanf()  
  scanf()  
  sscanf()  
  sscanf()  
  swscanf()  
  vfscanf()  
  vfwscanf()  
  viscanf()  
  vscanf()  
  vswscanf()  
  wscanf() |
| Seeking is supported in the Normal C but NOT in the Small C library. | These functions are provided ONLY in the Normal C library:  
  fseek()  
  ftell() |
<table>
<thead>
<tr>
<th>Limitation</th>
<th>Functions Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Small C library has no support for opening or closing FILE *. Only pre-opened stdout, stderr, and stdin are available.</td>
<td>These functions are provided ONLY in the Normal C Library:</td>
</tr>
<tr>
<td></td>
<td>fopen()</td>
</tr>
<tr>
<td></td>
<td>fclose()</td>
</tr>
<tr>
<td></td>
<td>fdopen()</td>
</tr>
<tr>
<td></td>
<td>fcloseall()</td>
</tr>
<tr>
<td></td>
<td>fileno()</td>
</tr>
<tr>
<td>The Small C library implements NO buffering of stdio.h() output routines.</td>
<td>These functions provide no buffering in the Small C library:</td>
</tr>
<tr>
<td></td>
<td>fprintf()</td>
</tr>
<tr>
<td></td>
<td>fputc()</td>
</tr>
<tr>
<td></td>
<td>fputs()</td>
</tr>
<tr>
<td></td>
<td>perror()</td>
</tr>
<tr>
<td></td>
<td>putc()</td>
</tr>
<tr>
<td></td>
<td>putchar()</td>
</tr>
<tr>
<td></td>
<td>puts()</td>
</tr>
<tr>
<td></td>
<td>printf()</td>
</tr>
<tr>
<td>The Small C library provides NO stdio.h() input routines.</td>
<td>These functions are provided ONLY in the Normal C library:</td>
</tr>
<tr>
<td></td>
<td>fgetc()</td>
</tr>
<tr>
<td></td>
<td>fgets()</td>
</tr>
<tr>
<td></td>
<td>fgetwc()</td>
</tr>
<tr>
<td></td>
<td>fgetws()</td>
</tr>
<tr>
<td></td>
<td>fscanf()</td>
</tr>
<tr>
<td></td>
<td>gets()</td>
</tr>
<tr>
<td></td>
<td>fread()</td>
</tr>
<tr>
<td></td>
<td>fscanf()</td>
</tr>
<tr>
<td></td>
<td>fwscanf()</td>
</tr>
<tr>
<td></td>
<td>getc()</td>
</tr>
<tr>
<td></td>
<td>getchar()</td>
</tr>
<tr>
<td></td>
<td>getline()</td>
</tr>
<tr>
<td></td>
<td>gets()</td>
</tr>
<tr>
<td></td>
<td>getw()</td>
</tr>
<tr>
<td></td>
<td>getwchar()</td>
</tr>
<tr>
<td></td>
<td>iscanf()</td>
</tr>
<tr>
<td></td>
<td>scanf()</td>
</tr>
<tr>
<td></td>
<td>sscanf()</td>
</tr>
<tr>
<td></td>
<td>swscanf()</td>
</tr>
<tr>
<td></td>
<td>vfscanf()</td>
</tr>
<tr>
<td></td>
<td>vfwscanf()</td>
</tr>
<tr>
<td></td>
<td>viscanf()</td>
</tr>
<tr>
<td></td>
<td>vscanf()</td>
</tr>
<tr>
<td></td>
<td>vsscanf()</td>
</tr>
<tr>
<td></td>
<td>vwscanf()</td>
</tr>
<tr>
<td></td>
<td>wscanf()</td>
</tr>
</tbody>
</table>
The Small C library provides NO support for locale.

These functions are provided ONLY in the Normal C library:

- setlocale()
- localeconv()

**Note:** These functions are a Nios II extension. GCC does not implement them in the small newlib C library.

**Note:** The small newlib C library does not support MicroC/OS-II.

For more information about the Intel FPGA "Small C" version of the newlib C library, refer to the newlib documentation installed with the EDS. You can get to this location by clicking on the Windows Start menu, and then navigating to Programs > Intel FPGA >> Documentation. You can also find this information by referring to the "Red Hat newlib C Library Documentation" web page.

**Note:** The Intel FPGA "Small C" version of the newlib C library differs considerably from the normal C version. **Table 6-9** summarizes the differences.

**Related Information**

*Red Hat newlib C Library Documentation web page*

---

**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 EDS provides ANSI C file I/O, in the newlib C library, because there is a per-access performance overhead associated with accessing devices and files using the UNIX-style file I/O functions. The ANSI C file I/O provides buffered access, thereby reducing the total number of hardware I/O accesses performed. Also the ANSI C API is more flexible and therefore easier to use. However, these benefits are gained at the expense of code footprint.

**Related Information**

*UNIX-Style Interface* on page 6-4

---

**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 6–15. Unbuffered getchar()**

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

This example is from the book: *The C Programming Language, Second Edition*, by Brian W. Kernighan and Dennis M. Ritchie. This standard textbook contains many other useful functions.
Use the Lightweight Device Driver API

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

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

- JTAG UART
- UART
- Optrex 16207 LCD

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

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

This option is appropriate if your program has a limited need for file I/O. The Intel FPGA host-based file system and the Intel FPGA read-only zip file system are not available with the reduced device driver API. You can select the reduced device drivers through BSP settings.

By default, the lightweight device driver API is disabled.

For more information about the lightweight device driver API, refer to the Developing Device Drivers for the Hardware Abstraction Layer section.

Related Information

Developing Programs Using the Hardware Abstraction Layer on page 6-1

Use the Minimal Character-Mode API

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

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

These functions are appropriate if your program only needs to accept command strings and send simple text messages. Some of them are helpful only in conjunction with the lightweight device driver API.

For more information, refer to the “Use the Lightweight Device Driver API” chapter.

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.

Related Information

Use the Lightweight Device Driver API on page 6-37
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(). In conjunction with the lightweight device driver API, this function further reduces code footprint. In the absence of the lightweight API, it calls putchar().

alt_putstr()

Similar to puts(), except that it does not append a newline character to the string. In conjunction with the lightweight device driver API, this function further reduces code footprint. In the absence of the lightweight API, it calls puts().

alt_getchar()

Equivalent to getchar(). In conjunction with the lightweight device driver API, this function further reduces code footprint. In the absence of the lightweight API, it calls getchar().

For more information about the minimal character-mode functions, refer to the "HAL API Reference" chapter.

Related Information
HAL API Reference on page 14-1

Eliminate Unused Device Drivers

If a hardware device is present in the system, by default the development flows assume the device needs drivers, and configure the HAL BSP accordingly. If the HAL can find an appropriate driver, it creates an instance of this driver. If your program never actually accesses the device, resources are being used unnecessarily to initialize the device driver.

If the hardware includes a device that your program never uses, consider removing the device from the hardware. This reduces both code footprint and FPGA resource usage.

However, there are cases when a device must be present, but runtime software does not require a driver. The most common example is flash memory. The user program might boot from flash, but not use it at runtime; thus, it does not need a flash driver.

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 "Software Build Tools" 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.

Related Information
- Boot Sequence and Entry Point on page 6-39
- Software Build Tools on page 4-1
- Boot Sequence and Entry Point on page 6-39
Eliminate Unneeded Exit Code

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

Eliminate Clean Exit

To avoid the overhead associated with providing a clean exit, your program can use the function `_exit()` in place of `exit()`. This function does not require you to change source code. You can select the `_exit()` function through a BSP setting.

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.

Turn off C++ Support

By default, the HAL provides support for C++ programs, including default constructors and destructors. You can disable C++ support through a BSP setting.

Boot Sequence and Entry Point

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

Hosted 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 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 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 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.

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

Related Information
Reducing Code Footprint in Embedded Systems on page 6-27

Boot Sequence for HAL-Based Programs

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.
- 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 .rdata, .rodata, and .exceptions.

For more information, refer to “Global Pointer Register” on page 6-41.
- Calls alt_main().

Related Information
Global Pointer Register on page 6-44

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 development flow 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++ constructors, using the _doctors() function.
- Registers the C++ destructors to be called at system shutdown.
- Calls main().
- Calls exit(), passing the return code of main() as the input argument for exit().

alt_main.c, installed with the EDS, provides this default implementation. The SBT copies alt_main.c to your BSP directory.
Customizing the Boot Sequence

You can provide your own implementation of the start-up sequence by simply defining `alt_main()` in your 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()`:

```c
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 SBT 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_init` and `alt_sys_init_priority` properties of each driver, using the `set_sw_property` Tcl command.

For more information about generated files and how to control the contents of `alt_sys_init.c`, refer to the "Software Build Tools" chapter.

For more information about `alt_sys_init.c`, refer to the "Developing Device Drivers for the Hardware Abstraction Layer" chapter.

For more information about the `set_sw_property` Tcl command, refer to the "Software Build Tools" chapter.

Related Information

- **Software Build Tools** on page 4-1
  For more information about generated files and how to control the contents of `alt_sys_init.c`; and for more information about the `set_sw_property` Tcl command.

- **Developing Device Drivers for the Hardware Abstraction Layer** on page 7-1
  For more information about `alt_sys_init.c`.

Memory Usage

This section describes how the HAL uses memory and arranges code, data, stack, and other logical memory sections, in physical memory.

Memory Sections

By default, HAL-based systems are linked using a generated linker script that is created by the SBT. 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 processor’s reset and exception addresses are a special case. The 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 software.

**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 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 6-3: HAL Link Map - Unavailable Memory Region Below the .exceptions Section**
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.

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

The BSP Editor is a very convenient way to manipulate the linker's memory map. The BSP Editor displays memory section and region assignments graphically, allowing you to see overlapping or unused sections of memory. The BSP Editor is available either through the Nios II SBT for Eclipse, or at the command line of the SBT.

For more information, refer to the "Getting Started from the Command Line" chapter.

Related Information

- HAL BSP Settings on page 6-1
- Getting Started from the Command Line on page 3-1

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 6–16. Manually Assigning C Code to a Specific Memory Section

```c
/* data should be initialized when using the section attribute */
int foo __attribute__ ((section (".ext_ram.rwdata"))) = 0;
void bar (void) __attribute__ ((section (".sdram.text")));
    {
        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. This documentation is installed with the EDS. To find it, open the EDS documentation launchpad, scroll down to Software Development, and click Using the GNU Compiler Collection (GCC).

Note: A powerful way to manipulate the linker memory map is by using the Nios II 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. You start the BSP Editor from the Command Shell. For details about using the BSP Editor, refer to the editor's tool tips.

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.

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

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.

For more information about selecting stack and heap placement, and setting up stack checking, refer to the "Software Build Tools" chapter.

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

Related Information

• HAL BSP Settings on page 6-1
• Software Build Tools Reference on page 15-1
• Embedded Software Projects on page 4-4

Global Pointer Register

The global pointer register enables fast access to global data structures in programs. The 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 6-4: Small Global Data Sections**

If the total size of the small global data structures is more than 64 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, -32678 to 32767."

The best solution is to use fewer global variables of size 8 bytes or less. This is the default setting for the -Gn switch, perhaps by combining some into records. Another solution is to place only the frequently used global variables inside the .sdata + .sbss.

If the previous solution fails, a quick and easy solution is to use the -mgpopt=none switch. This removes the 64 KB limit on the size of the .sdata + .sbss sections addressed by the GP register. This is accomplished at the cost of using bigger, slower (typically 3-instruction) code sequences to address such values, in place of the typically single-instruction code sequences otherwise used.
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 `#unique_406/unique_406_Connect_42_example_cnm_wbq_yy`.

For information about manipulating project settings, refer to "HAL BSP Settings".

Related Information

- **HAL BSP Settings** on page 6-1
- **GCC Nios II Options**

**Boot Modes**

The processor's boot memory is the memory that contains the reset vector. This device might be an external flash or an Intel FPGA EPCS serial configuration device, or it might be an on-chip RAM. Regardless of the nature of the boot memory, HAL-based systems are constructed so that all program and data sections are initially stored 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 Intel FPGA flash programmer places a boot loader at the reset address. This boot loader is responsible for loading all program and data sections before the call to `_start`. When booting from an EPCS device, this loader function is provided by the hardware.

You can boot a processor from Intel FPGA EPCQ flash memory (EPCQx1, EPCQx4) using an Intel FPGA serial flash controller. The Intel FPGA Serial Flash Controller with Avalon interface allows processor systems to access an Intel FPGA EPCQ flash memory, which supports standard, quad and single- or dual-I/O mode. The processor Software Build Tools (SBT) supports the booting from the Intel FPGA Serial Flash Controller. In addition, a hardware abstraction layer (HAL) driver is available for the Intel FPGA Serial Flash Controller that allows an application to read, write, or erase flash.

For more information about how to build a bootable system for a processor application executing in place (XIP) from EPCQ flash, refer to AN736: Processor Booting From Intel FPGA Serial Flash (EPCQ).

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 `.rdata, .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 14-1
Working with HAL Source Files

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

Finding HAL Files

You 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.

For more information, refer to "Software Build Tools Reference" of the Software Developer’s Handbook.

Related Information

- Software Build Tools Reference on page 15-1
- Embedded Software Projects on page 4-4

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” in the "Software Build Tools” section.

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.

For more information, refer to “Embedded Software Projects” in the "Software Build Tools” section.

Related Information

- Software Build Tools Reference on page 15-1
- Embedded Software Projects on page 4-4
Embedded systems typically have application-specific hardware features that require custom device drivers. This chapter describes how to develop device drivers and integrate them with the hardware abstraction layer (HAL).

This chapter also describes how to develop software packages for use with HAL board support packages (BSPs). The process of integrating a software package with the HAL is nearly identical with the process for integrating a device driver.

Confine direct interaction with the hardware to device driver code. In general, the best practice is to keep most of your program code free of low-level access to the hardware. Wherever possible, use the high-level HAL application program interface (API) functions to access hardware. This makes your code more consistent and more portable to other Nios® II systems that might have different hardware configurations.

When you create a new driver, you can integrate the driver with the HAL framework at one of the following two levels:

- Integration in the HAL API
- Peripheral-specific API

**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.

### 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 5-1
For more information about the descriptions of the HAL generic device model classes.
The HAL Peripheral-Specific API

If the peripheral does not belong to one of the HAL generic device model classes, you need to provide a device driver with an interface that is specific to the hardware implementation. In this case, the API to the device is separate from the HAL API. Programs access the hardware by calling the functions you provide, not the HAL API.

The up-front effort to implement integration in the HAL API is higher, but you gain the benefit of the HAL and C standard library API to manipulate devices.

For details about integration in the HAL API, refer to the “Integrating a Device Driver in the HAL” chapter.

All the 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 7-15

Preparing for HAL Driver Development

This chapter assumes that you are familiar with C programming for the HAL.

For more information, refer to the "Developing Programs Using the Hardware Abstraction Layer" section.

**Note:** This chapter uses the variable `<Intel FPGA installation>` to represent the location where the Intel FPGA Complete Design Suite is installed. On a Windows system, by default, that location is c:/intelfpga/<version number>.

Related Information

- Developing Programs Using the Hardware Abstraction Layer on page 6-1

Development Flow for Creating Device Drivers

The steps to develop a new driver for the HAL depend on your device details. However, the following generic steps apply to all device classes.

1. Create the device header file that describes the registers. This header file might be the only interface required.
2. Implement the driver functionality.
3. Test from `main()`.
4. Proceed to the final integration of the driver in the HAL environment.
5. Integrate the device driver in the HAL framework.
Hardware Design Concepts

This section discusses some basic concepts behind the Intel 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 device drivers.

The Relationship Between the .sopcinfo File and system.h

The system generation tool, , generates the processor system hardware. Hardware designers use the system generation tool to specify the architecture of the processor system and integrate the necessary peripherals and memory. Therefore, the definitions in system.h, such as the name and configuration of each peripheral, are a direct reflection of design choices made in the system generation tool. These design choices are encapsulated in the .sopcinfo file. system.h is derived from the .sopcinfo file.

For more information about the system.h header file, refer to the "Developing Programs Using the Hardware Abstraction Layer" section.

Related Information

Developing Programs Using the Hardware Abstraction Layer on page 6-1

Using the System Generation Tool to Optimize Hardware

If you find less-than-optimal definitions in system.h, remember that you can modify the contents of system.h by changing the underlying hardware with the system generation tool. Before you write a device driver to accommodate imperfect hardware, it is worth considering whether the hardware can be improved easily with the system generation tool.

Components, Devices, and Peripherals

The system generation tools use the term "component" to describe hardware modules included in the system. In the context of software development, components are devices, such as peripherals or memories. In the following sections, “component” is used interchangeably with “device” and "peripheral” when the context is closely related to the system generation tool.

Accessing Hardware

Software accesses the hardware with macros that abstract the memory-mapped interface to the device. This section describes the macros that define the hardware interface for each device.

All components provide a directory that defines the device hardware and software. For example, each component provided in the software has its own directory in the <Intel FPGA installation>/ip/altera/sopc_builder_ip directory. Many components provide a header file that defines their hardware interface. The header file is named <component name>_regs.h, included in the inc subdirectory for the specific component. For example, the Intel FPGA-provided JTAG UART component defines its hardware interface in the file <Intel FPGA installation>/ip/altera/sopc_builder_ip/altera_avalon_jtag_uart/inc/altera_avalon_jtag_uart_regs.h.
The _regs.h header file defines the following access macros for the component:

- Register access macros that provide 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 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 FPGA-supplied components, such as <Intel FPGA installation>/ip/sopc_builder_ip/altera_avalon_jtag_uart/inc.

Related Information

- AN 459: Guidelines for Developing a Nios II HAL Device Driver
  For more information about developing device drivers for HAL BSPs.
- Cache and Tightly-Coupled Memory on page 9-1
  For more information about the effects of cache management and device access.
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
- Ethernet devices

The following sections describe how to implement device drivers for each class of device, and how to register them for use in HAL-based systems.

Related Information
Overview of the Hardware Abstraction Layer on page 5-1

Character-Mode Device Drivers

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 `<EDS install path>/components/altera_hal/HAL/inc/sys/alt_dev.h`, is essentially a collection of function pointers. These functions are called in response to application accesses to the HAL file system. For example, if you call the function `open()` with a file name that corresponds to this device, the result is a call to the `open()` function provided in this structure.

Example 7–1. `alt_dev` Structure

```c
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;
```

Developing Device Drivers for the Hardware Abstraction Layer

Altera Corporation
For more information about `open()`, `close()`, `read()`, `write()`, `lseek()`, `fstat()`, and `ioctl()`, refer to the "HAL API Reference" section.

Related Information
HAL API Reference on page 14-1

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:

```c
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.

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 7-1: Default Behavior for Functions Defined in `alt_dev`

<table>
<thead>
<tr>
<th>Function</th>
<th>Default Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>open</td>
<td>Calls to <code>open()</code> for this device succeed, unless the device was previously locked by a call to <code>ioctl()</code> with <code>req = TIOCSEXCL</code>.</td>
</tr>
<tr>
<td>close</td>
<td>Calls to <code>close()</code> for a valid file descriptor for this device always succeed.</td>
</tr>
<tr>
<td>read</td>
<td>Calls to <code>read()</code> for this device always fail.</td>
</tr>
<tr>
<td>Function</td>
<td>Default Behavior</td>
</tr>
<tr>
<td>----------</td>
<td>------------------</td>
</tr>
<tr>
<td>write</td>
<td>Calls to write() for this device always fail.</td>
</tr>
<tr>
<td>lseek</td>
<td>Calls to lseek() for this device always fail.</td>
</tr>
<tr>
<td>fstat</td>
<td>The device identifies itself as a character mode device.</td>
</tr>
<tr>
<td>ioctl</td>
<td>ioctl() requests that cannot be handled without reference to the device fail.</td>
</tr>
</tbody>
</table>

In addition to the function pointers, the `alt_dev` structure contains two other fields: `llist` and `name`. `llist` is for internal use, and must always be set to the value `ALT_LLIST_ENTRY`. `name` is the location of the device in the HAL file system and is the name of the device as defined in `system.h`.

**Register a Character Device**

After you create an instance of the `alt_dev` structure, the device must be made available to the system by registering it with the HAL and by calling the following function:

```c
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.

**Related Information**

*Developing Device Drivers for the Hardware Abstraction Layer* on page 7-1

**File Subsystem Drivers**

A file subsystem device driver is responsible for handling file accesses beneath a specified mount point in the global HAL file system.

**Create a Device Instance**

Creating and registering a file system is very similar to creating and registering a character-mode device. To make a file system available, create an instance of the `alt_dev` structure.

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.

**Related Information**

*Character-Mode Device Drivers* on page 7-5

**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.

Related Information
Developing Device Drivers for the Hardware Abstraction Layer on page 7-1

Timer Device Drivers

System Clock Driver
A system clock device model requires a driver to generate the periodic clock tick. There can be only one system clock driver in a system. You implement a system clock driver as an interrupt service routine (ISR) for a timer peripheral that generates a periodic interrupt. The driver must provide periodic calls to the following function:

void alt_tick (void)

The expectation is that alt_tick() is called in exception context.

To register the presence of a system clock driver, call the following function:

int alt_sysclk_init (alt_u32 nticks)

The input argument nticks is the number of system clock ticks per second, which is determined by your system clock driver. The return value of this function is zero on success, and nonzero otherwise.

For more information about writing interrupt service routines, refer to the "Exception Handling" section.

Related Information
Exception Handling on page 8-1

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.

For more information about using these functions, refer to the "HAL API Reference" section.

Related Information
• HAL API Reference on page 14-1
• Developing Device Drivers for the Hardware Abstraction Layer on page 7-1
Flash Device Drivers

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:

```c
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, such as common flash interface (CFI)-compliant devices, allow you to read out the number of regions and their configuration at run time. For all other flash devices, these two fields must be defined at compile time.

Related Information

**Developing Device Drivers for the Hardware Abstraction Layer** on page 7-1
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:

```c
int alt_flash_device_register( alt_flash_fd* fd)
```

This function takes a single input argument, which is the device structure to register. The return value is zero upon success. A negative return value indicates that the device cannot be registered.

DMA Device Drivers

The HAL models a DMA transaction as being controlled by two endpoint devices: a receive channel and a transmit channel.

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`.

Related Information

Developing Device Drivers for the Hardware Abstraction Layer on page 7-1

DMA Transmit Channel

Example 7-2. `alt_dma_txchan` Structure

```c
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);
};
```

DMA Receive Channel

Example 7-3. `alt_dma_rxchan` Structure

```c
typedef alt_dma_rxchan_dev_s alt_dma_rxchan;
struct alt_dma_rxchan_dev_s
{  
  alt_llist list;
  const char* name;
  alt_u32 depth;
  int (*prepare)(alt_dma_rxchan dma,
    void* data,
    alt_u32 len,
    alt_rxchan_done* done,
    void* handle);
  int (*ioctl)(alt_dma_rxchan dma, int req, void* arg);
};
```

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:

```c
int alt_dma_rxchan_reg (alt_dma_rxchan_dev* dev)
```

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

### Table 7-2: Fields in the `alt_dma_rxchan` Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>llist</td>
<td>This function is for internal use and must always be set to the value <code>ALT_LLLIST_ENTRY</code>.</td>
</tr>
<tr>
<td>name</td>
<td>The name that refers to this channel in calls to <code>alt_dma_rxchan_open()</code>. <code>name</code> is the name of the device as defined in <code>system.h</code>.</td>
</tr>
<tr>
<td>depth</td>
<td>The total number of receive requests that can be outstanding at any given time.</td>
</tr>
<tr>
<td>prepare</td>
<td>A pointer to a function that is called as a result of a call to the application API function <code>alt_dma_rxchan_prepare()</code>. This function posts a receive request to the DMA device. The parameters passed to <code>alt_dma_rxchan_prepare()</code> are passed directly to <code>prepare()</code>. For a description of parameters and return values, refer to the &quot;HAL API Reference&quot; section.</td>
</tr>
<tr>
<td>ioctl</td>
<td>This is a function that provides device specific I/O control. Refer to <code>sys/alt_dma_dev.h</code> for a list of the generic options that a device might wish to support.</td>
</tr>
</tbody>
</table>

### Related Information

- **HAL API Reference** on page 14-1

### Ethernet Device Drivers

The HAL generic device model for Ethernet devices provides access to the NicheStack® TCP/IP Stack - Edition running on the MicroC/OS-II operating system. You can provide support for a new Ethernet device by supplying the driver functions that this section defines.

Before you consider writing a device driver for a new Ethernet device, you need a basic understanding of the Intel FPGA implementation of the NicheStack TCP/IP Stack and its usages.
Architectural Layers of a MicroC/OS-II software application

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

For more information, refer to the "Ethernet and the NicheStack TCP/IP Stack - Edition" section.

Related Information

Ethernet and the NicheStack TCP/IP Stack on page 11-1

Writing a New Ethernet Device Driver

The easiest way to write a new Ethernet device driver is to start with Intel FPGA's implementation for the Standard Microsystems Corporation (SMSC) lan91c111 device, and modify it to suit your Ethernet media access controller (MAC). This section assumes you take this approach. Starting from a known working example makes it easier for you to learn the most important details of the NicheStack TCP/IP Stack implementation.

The source code for the Lan91c111 10/100 Non-PCI Ethernet Single Chip MAC + PHY driver is provided with the Quartus Prime software. For the sake of brevity, this section refers to this directory as <SMSC path>. The source files are in the <SMSC path>/src/iniche and <SMSC path>/inc/iniche directories.

A number of useful NicheStack TCP/IP Stack files are installed with the Embedded Design Suite (EDS), under the <Nios II EDS install path>/components/altera_iniche/UCOSII directory. For the sake of brevity, this chapter refers to this directory as <iniche path>.


You need not edit the NicheStack TCP/IP Stack source code to implement a NicheStack-compatible driver because the source code is provided for your reference. The files are installed with the EDS in the <iniche...
The Ethernet device driver interface is defined in `<niche path>/inc/alt_iniche_dev.h`.

**Related Information**

- **NicheStack Technical Reference Manual**

- **NicheStackRef.zip**
  For access to the NicheStackRef.zip file.

**Provide the NicheStack Hardware Interface Routines**

The NicheStack TCP/IP Stack architecture requires several network hardware interface routines:

- Initialize hardware
- Send packet
- Receive packet
- Close
- Dump statistics

For more information about these routines, refer to "Porting Engineer Provided Functions" in the *NicheStack Technical Reference Manual*.

The NicheStack TCP/IP Stack system code uses the

<table>
<thead>
<tr>
<th>Table 7-3: SMSC LAN91C111 Hardware Interface Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Prototype Function</strong></td>
</tr>
<tr>
<td>-------------------------</td>
</tr>
<tr>
<td>n_init()</td>
</tr>
<tr>
<td>pkt_send()</td>
</tr>
<tr>
<td>pkt_send()</td>
</tr>
<tr>
<td>pkt_send()</td>
</tr>
<tr>
<td>Packet receive</td>
</tr>
<tr>
<td>mechanism</td>
</tr>
<tr>
<td>n_close()</td>
</tr>
<tr>
<td>n_stats()</td>
</tr>
<tr>
<td>s91_isr()</td>
</tr>
<tr>
<td>s91_rcv()</td>
</tr>
<tr>
<td>s91_dma_rx_done()</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>s91_dma_rx_done()</td>
</tr>
</tbody>
</table>
net structure internally to define its interface to device drivers. The net structure is defined in net.h, in\
iniche path/31src. Among other things, the net structure contains the following things:

- A field for the IP address of the interface
- A function pointer to a low-level function to initialize the MAC device
- Function pointers to low-level functions to send packets

Typical NicheStack code refers to type NET, which is defined as *net.

Related Information

  For more information about the NicheStack TCP/IP Stack implementation, refer to the NicheStack
- NicheStackRef.zip
  For access to the NicheStackRef.zip file.

Provide *INSTANCE and *INIT Macros

To enable the HAL to use your driver, you must provide two HAL macros. The names of these macros are
based on the name of your network interface component, according to the following templates:

- <component name>_INSTANCE
- <component name>_INIT

For examples, refer to ALTERA_AVALON_LAN91C111_INSTANCE and ALTERA_AVALON_LAN91C111_INIT in
<SMSC path>/inc/iniche/altera_avalon_lan91c111_iniche.h, which is included in <iniche path>/inc/altera_avalon_lan91c111.h.

You can copy altera_avalon_lan91c111_iniche.h and modify it for your own driver. The HAL
expects to find the *INIT and *INSTANCE macros in <component name>.h.

For more information, refer to the “Header Files and alt_sys_init.c” chapter. You can accomplish this with
a #include directive as in altera_avalon_lan91c111.h, or you can define the macros directly in
<component name>.h.

Your *INSTANCE macro declares data structures required by an instance of the MAC. These data structures
must include an alt_iniche_dev structure. The *INSTANCE macro must initialize the first three fields of
the alt_iniche_dev structure, as follows:

- The first field, llist, is for internal use, and must always be set to the value ALT_LLIST_ENTRY.
- The second field, name, must be set to the device name as defined in system.h. For example, altera_avalon_lan91c111_iniche.h uses the C preprocessor's ## (concatenation) operator to reference the LAN91C111_NAME symbol defined in system.h.
- The third field, init_func, must point to your software initialization function.

For more information, refer to the “Provide a Software Initialization Function” chapter.

For example, altera_avalon_lan91c111_iniche.h inserts a pointer to
altera_avalon_lan91c111_init().

Your *INIT macro initializes the driver software. Initialization must include a call to the
alt_iniche_dev_reg() macro, defined in alt_iniche_dev.h. This macro registers the device with the
HAL by adding the driver instance to alt_iniche_dev_list.

When your driver is included in a BSP project, the HAL automatically initializes your driver by invoking the
*INSTANCE and *INIT macros from its alt_sys_init() function. For more information about the
*INSTANCE and *INIT macros, refer to the "Header Files and alt_sys_init.c" chapter.
Provide a Software Initialization Function

The *INSTANCE() macro inserts a pointer to your initialization function in the alt_iniche_dev structure. For more information, refer to the “Provide *INSTANCE and *INIT Macros” chapter.

Your software initialization function must perform at least the following three tasks:

- Initialize the hardware and verify its readiness
- Finish initializing the alt_iniche_dev structure
- Call get_mac_addr()

The initialization function must perform any other initialization your driver needs, such as creation and initialization of custom data structures and ISRs.

For more information about the get_mac_addr() function, refer to the "Ethernet and the NicheStack TCP/IP Stack - Edition" chapter.

For more information about an example of a software initialization function, refer to alt_avalon_lan91c111_init() in <SMSC path>/src/iniche/smsc91x.c.

Related Information
- Provide *INSTANCE and *INIT Macros on page 7-14
- Ethernet and the NicheStack TCP/IP Stack on page 11-1

Integrating a Device Driver in the HAL

The SBT can incorporate device drivers and software packages supplied by Intel FPGA, supplied by other third-party developers, or created by you. This section describes how to prepare device drivers and software packages so the BSP generator recognizes and adds them to a generated BSP.

You can take advantage of this service, whether you created a device driver for one of the HAL generic device models, or you created a peripheral-specific device driver.

**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. Certain steps are not needed for software packages, as noted in the text.

Overview

To publish a device driver or a software package, you provide the following items:

- A header file defining the package or driver interface
- A Tcl script specifying how to add the package or driver to a BSP

The header file and Tcl script are described in the following sections.

Assumptions and Requirements

Typically, you are developing a device driver or software package for eventual incorporation in a BSP. The driver or package is to be incorporated in the BSP by an end user who has limited knowledge of the driver
or package internal implementation. To add your driver or package to a BSP, the end user must rely on the
driver or package settings that you create with the tools described in this section.

For a device driver or software package to work with the SBT, it must meet the following criteria:

- It must have a defining Tcl script. The Tcl script for each driver or software package provides the SBT
  with a complete description of the driver or software. This description includes the following informa-
  tion:
    - 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) informa-
tion 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 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.

For more information on how file names and directory structures conform to certain conventions, refer to
the "File Names and Locations" chapter.

If your driver or software package uses the HAL auto initialization mechanism (alt_sys_init()), certain
macros must be defined in a header file.

For more information about this header file, refer to the "Header Files and alt_sys_init.c" chapter.

For more information about integrating a HAL device driver, refer to AN 459: Guidelines for Developing a
Nios II HAL Device Driver.

For more information about the commands you can use in a driver Tcl script, refer to the "Software Build
Tools Reference" chapter.

Related Information

- File Names and Locations on page 7-18
- Header Files and alt_sys_init.c on page 7-27
- AN 459: Guidelines for Developing a Nios II HAL Device Driver
- Software Build Tools Reference on page 15-1

The BSP Generator

This section describes the process by which the BSP generator adds device drivers and software packages
to your BSP. The BSP generator, a subset of the SBT, is a combination of command utilities and scripts that
enable you to create and manage BSPs and their settings.

For more information about the SBT, refer to the "Overview of Embedded Development" chapter.

For more information about the SBT, refer to the "Getting Started from the Command Line" chapter.
Related Information

- **Overview of Embedded Development** on page 1-1
- **Getting Started from the Command Line** on page 3-1

**Component Discovery**

When you run any BSP generator utility, a library of available drivers and software packages is populated. The BSP generator locates software packages and drivers by inspecting a list of known locations determined by the Intel FPGA EDS, Quartus Prime software, and IP core® IP Library installers, as well as searching locations specified in certain system environment variables.

The BSP tools identify 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 generator only looks at driver files that conform to the criteria listed in this section.

After locating each driver and software package, the SBT searches for a suitable driver for each hardware module in the hardware system (mastered by the processor that the BSP is generated for), as well as software packages that the BSP creator requested.

**Device Driver Versions**

In the case of device drivers, the highest version of driver that is compatible with the associated hardware peripheral is added to the BSP, unless specified otherwise by the device driver management commands.

For more information, refer to the "Software Build Tools Reference" section.

**Related Information**

**Software Build Tools Reference** on page 15-1

**Device Driver and Software Package Inclusion**

**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.

For more information, refer to “Software Build Tools Tcl Commands” in the "Software Build Tools Reference" chapter.

**Related Information**

**Software Build Tools Reference** on page 15-1

**No Specific Requests**

If no specific device driver is requested, and no compatible device driver is located for a particular hardware module, the BSP generator issues an informative message visible in either the `debug` or `verbose` generation output. This behavior is normal for many types of hardware, such as memory devices, that do not have device drivers. If a software package or specific driver is requested and cannot be located, an error is generated and BSP generation or settings update halts.

Creating a Tcl script allows you to add extra definitions in the `system.h` file, enable automatic driver initialization through the `alt_sys_init.c` structure, and enable the SBT to control any extra parameters that might exist.
With the Tcl software definition files in place, the SBT reads in the Tcl file and populate the makefiles and other support files accordingly.

When the SBT adds each driver or software package to the system, it uses the data in the Tcl script defining the driver or software package to control each file copied in to the BSP. This rule also affects generated BSP files such as the BSP Makefile, public.mk, system.h, and the BSP settings and summary HTML files.

When you create a new software project, the SBT generates the contents of alt_sys_init.c to match the specific hardware contents of the system.

File Names and Locations
The 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.

For more information, refer to the “The Nios II BSP Generator” chapter.

Each peripheral in a 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.

To enable the SBT to find your component device driver, place the Tcl script in a directory named ip under your hardware project directory.

The file hierarchy that is suitable for the Nios II SBT is located in the <Intel FPGA installation>/ip/altera/sopc_builder_ip directory. This example assumes a device driver supporting a hardware component named custom_component.

Related Information
- The BSP Generator on page 7-16
- Accessing Hardware on page 7-3

Source Code Discovery
You use Tcl scripts to specify the location of driver source files.

For more information, refer to the “The Nios II BSP Generator” chapter.

Related Information
The BSP Generator on page 7-16

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 SBT provides powerful features that give the BSP end user control of your software or driver’s operation.

The Tcl command and argument descriptions in this section are not exhaustive. For a detailed explanation of each command and all arguments, refer to the “Software Build Tools Reference” section.

Related Information
Software Build Tools Reference on page 15-1
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 EDS and the IP core IP library. These scripts are in the `<EDS install path>/components` and `<IP core IP library install path>/sopc_builder_ip` folders, respectively.

Figure 7-1: Example Device Driver File Hierarchy and Naming

![](Image)

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 hardware abstraction layer. Files in this directory pertain specifically to the HAL.

Tcl Command Walkthrough for a Typical Driver or Software Package

The Tcl script excerpts in this section describe a typical device driver or software package.
The example in this section creates a device driver for a hardware peripheral whose component class name is `my_custom_component`. The driver supports both HAL and MicroC/OS-II BSP types. It has a single C source file (.c) and two C header files (.h).

**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:

```tcl
create_driver my_custom_component_driver
```

**Identifying the Hardware Component Class**

Each driver must identify the hardware component class the driver is associated with in the `set_sw_property` command's `hw_class_name` argument. The following example associates the driver with a hardware class called `my_custom_component`:

```tcl
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.

For more information about the `set_sw_property` command, refer to the "Software Build Tools Reference" chapter.

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 SBT uses your driver only if you specify it explicitly.

For more information, refer to the "Software Build Tools Reference" chapter.

Choose a name for your driver or software package that does not conflict with other Intel FPGA-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 SBT finds multiple compatible drivers or software packages with the same name, it might pick any of them.

If you intend to distribute your driver or software package, Intel FPGA recommends prefixing all names with your organization's name.

**Related Information**

[Software Build Tools Reference](#) on page 15-1

**Setting the BSP Type**

You must specify each operating system (or BSP type) that your driver or software package supports. Use the `add_sw_property` command's `supported_bsp_type` argument to specify each compatible operating system. In most cases, a driver or software package supports both Intel FPGA HAL (hal) and Micrium MicroC/OS-II (ucosii) BSP types, as in the following example:

```tcl
add_sw_property supported_bsp_type hal
add_sw_property supported_bsp_type ucosii
```

*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.
Specifying an Operating System

Many drivers and software packages do not require any particular operating system. However, you can structure your software to provide different source files depending on the operating system used.

If your driver or software has different source files, paths, or settings that depend on the operating system used, write a Tcl script for each variant of the driver or software package. Each script must specify the same software package or driver name in the `create_driver` or `create_sw_package` command, and same `hw_class_name` in the case of device drivers. Each script must specify only the files, paths, and other settings that pertain to that operating system. During BSP generation, only drivers or software packages that specify compatibility with the selected operating system (OS) type are eligible to add to the BSP.

Specifying Source Files

Using the Tcl command interface, you must specify each source file in your driver or software package that you want in the generated BSP. The commands discussed in this section add driver source files and specify their location in the file system and generated BSP.

The `add_sw_property` command's `c_source` and `asm_source` arguments add a single `.c` or assembly language source file (.s or .S) to your driver or software package. You must express path information to the source relative to the driver root (the location of the Tcl file). `add_sw_property` copies source files to BSPs that incorporate the driver, using the path information specified, and adds them to source file list in the generated BSP makefile. When you build the BSP using `make`, the driver source files are compiled as follows:

```
add_sw_property c_source HAL/src/my_driver.c
```

The `add_sw_property` command's `include_source` argument adds a single header file in the path specified to the driver. The paths are relative to the driver root. `add_sw_property` copies header files to the BSP during generation, using the path information specified at generation time. It does not include header files in the makefile.

```
add_sw_property include_source inc/my_custom_component_regs.h
add_sw_property include_source HAL/inc/my_custom_component.h
```

Specifying a Subdirectory

You can optionally specify a subdirectory in the generated BSP for your driver or software package files using the `bsp_subdirectory` argument to `set_sw_property`. All driver source and header files are copied to this directory, along with any path or hierarchy information specified with each source or header file. If no `bsp_subdirectory` is specified, your driver or software package is placed under the `drivers` folder of the generated BSP. Set the subdirectory as follows:

```
set_sw_property bsp_subdirectory my_driver
```

Note: If the path begins with the BSP type (e.g. `HAL` or `UCOSII`), the BSP type is removed and replaced with the value of the `bsp_subdirectory` property.

Enabling Software Initialization

If your driver or software package uses the HAL 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`.

For more information, refer to the “Provide *INSTANCE and *INIT Macros” chapter.
To support this functionality in BSPs, you must set the `set_sw_property` command's `auto_initialize` argument to `true` using the following Tcl command:

\[
\text{set_sw_property \{auto\}_{initialize} \{true\}}
\]

If you do not turn on this attribute, `alt_sys_init.c` does not invoke the `INIT` and `INSTANCE` macros.

**Related Information**

*Provide *INSTANCE and *INIT Macros on page 7-14*

## Adding Include Paths

By default, the generated BSP `Makefile` and `public.mk` add include paths to find header files in `/inc` or `<BSP type>/inc` folders.

You might need to set up a header file directory hierarchy to logically organize your code. You can add additional include paths to your driver or software package using the `add_sw_property` command's `include_directory` argument as follows:

\[
\text{add_sw_property \{include\}_{directory} UCOSII/inc/protocol/h}
\]

**Note:** If the path begins with the BSP type (e.g. `HAL` or `UCOSII`), the BSP type is removed and replaced with the value of the `bsp_subdirectory` property.

Additional include paths are added to the preprocessor flags in the BSP `public.mk` file. These preprocessor flags allow BSP source files, as well as application and user library source files that reference the BSP, to find the include path while each source file is compiled.

**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:

\[
\text{#include "protocol/h/\{filename\}}"
\]

## Version Compatibility

Your device driver or software package can optionally specify versioning information through the Tcl command interface. The driver and software package Tcl commands specifying versioning information allow the following functionality:

- You can request a specific version of your driver or software package with BSP settings.
- You can make updates to your device driver and specify that the driver is still compatible with a minimum hardware class version, or specific hardware class versions. This facility is especially useful in situations in which a hardware design is stable and you foresee making software updates over time.

The `<version>` argument in each of the following versioning-related commands can be a string containing numbers and characters. Examples of version strings are `8.0`, `5.1.1`, `6.1`, and `6.1sp1`. The `"."` character is a separator. The BSP generator compares versions against each other to determine if one is more recent than the other, or if two are equal, by successively comparing the strings between each separator. Thus, `2.1` is greater than `2.0`, and `2.1sp1` is greater than `2.1`. Two versions are equal if their version assignment strings are identical.

Use the `version` argument of `set_sw_property` to assign a version to your driver or software package. If you do not assign a version to your software or device driver, the version of the EDS installation (containing the BSP commands being executed) is set for your driver or software package:

\[
\text{set_sw_property \{version\} 7.1}
\]
Device drivers (but not software packages) can use the \texttt{min\_compatible\_hw\_version} and \texttt{specific\_compatible\_hw\_version} arguments to establish compatibility with their associated hardware class, as follows:

\begin{verbatim}
set_sw_property min_compatible_hw_version 5.0.1
add_sw_property specific_compatible_hw_version 6.1sp1
\end{verbatim}

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, Intel FPGA recommends setting the \texttt{min\_compatible\_hw\_version} of your driver to the lowest version of the associated hardware class your driver is compatible with.

### Creating Settings for Device Drivers and Software Packages

The BSP generator allows you to publish settings for individual device drivers and software packages. These settings are visible and can be modified by the BSP user, if the BSP includes your driver or software package. Use the Tcl command interface to create settings.

The Tcl command that publishes settings is especially useful if your driver or software package has build or runtime options that are normally specified with \texttt{#define} statements or makefile definitions at software build time. Settings can also add custom variable declarations to the BSP \texttt{Makefile}.

### How Settings Affect the Generated BSP

Settings affect the generated BSP in several ways:

- Settings are added either to the BSP \texttt{system.h} or \texttt{public.mk}, or to the BSP makefile as a variable.
- Settings are stored in the BSP settings file, named with hierarchy information to prevent namespace collision.
- A default value of your choice is assigned to the setting so that the end user of the driver or package does not need to explicitly specify the setting when creating or updating a BSP.
- Settings are displayed in the BSP \texttt{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:

- \texttt{type} - The data type, which controls formatting of the setting's value assignment in the appropriate generated file.
- \texttt{destination} - The destination file in the BSP.
- \texttt{displayName} - The name that is used to identify the setting when changing BSP settings or viewing the BSP \texttt{summary.html} document.
- \texttt{identifier} - Conceptually, this argument is the macro defined in a C language definition (the text immediately following \texttt{#define}), or the name of a variable in a makefile.
- \texttt{value} - A default value assigned to the setting if the BSP user does not manually change it.
- \texttt{description} - Descriptive text, shown in the BSP \texttt{summary.html} document.

### Data Types

Several setting data types are available, controlled by the \texttt{type} argument to add\_sw\_setting. They correspond to the data types you can express as \texttt{#define} statements or values concatenated to makefile variables. The specific setting type depends on your software's structure or BSP build needs.
### Table 7-4: Data Type Settings

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Setting Value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean definition</td>
<td>boolean_define_only</td>
<td>A definition that is generated when true, and absent when false. Use a boolean definition in your C source files with the #ifdef &lt;setting&gt; ... #endif construct.</td>
</tr>
<tr>
<td>Boolean assignment</td>
<td>boolean</td>
<td>A definition assigned to 1 when true, 0 when false. Use a boolean assignment in your C source files with the #if &lt;setting&gt; ... #else ... construct.</td>
</tr>
<tr>
<td>Character</td>
<td>character</td>
<td>A definition with one character surrounded by single quotation marks ('')</td>
</tr>
<tr>
<td>Decimal number</td>
<td>decimal_number</td>
<td>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];</td>
</tr>
<tr>
<td>Double precision number</td>
<td>double</td>
<td>A definition with a double-precision floating point number such as 123.4</td>
</tr>
<tr>
<td>Floating point number</td>
<td>float</td>
<td>A definition with a single-precision floating point number such as 234.5</td>
</tr>
<tr>
<td>Hexadecimal number</td>
<td>hex_number</td>
<td>A definition with a number prefixed with 0x, such as 0x1000. Useful for specifying memory addresses or bit masks</td>
</tr>
<tr>
<td>Quoted string</td>
<td>quoted_string</td>
<td>A definition with a string in quotes, such as &quot;Buffer&quot;</td>
</tr>
<tr>
<td>Unquoted string</td>
<td>unquoted_string</td>
<td>A definition with a string not in quotes, such as BUFFER</td>
</tr>
</tbody>
</table>

### 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>
<thead>
<tr>
<th>Destination File</th>
<th>Setting Value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>system.h</code></td>
<td><code>system_h_</code> define</td>
<td>This destination file is recommended in most cases. Your source code must use a <code>#include &lt;system.h&gt;</code> statement to make the setting definitions available. Settings appear as <code>#define</code> statements in <code>system.h</code>.</td>
</tr>
<tr>
<td><code>public.mk</code></td>
<td><code>public_mk_</code> define</td>
<td>Definitions appear as <code>-D</code> statements in <code>public.mk</code>, 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.</td>
</tr>
<tr>
<td><code>BSP makefile</code></td>
<td><code>makefile_</code> variable</td>
<td>Settings appear as <code>makefile</code> variable assignments in the BSP makefile.</td>
</tr>
</tbody>
</table>

**Note:** Certain setting types are not compatible with the `public.mk` or **Makefile** destination file types. For more information, refer to the "Software Build Tools Reference" chapter.

**Related Information**

Software Build Tools Reference on page 15-1

**Setting Display Name**

The setting `displayName` controls what the end user of the driver or package (the BSP developer) types to control the setting in their BSP. BSPs append the `displayName` text after a . (dot) separator to your driver or software package's name (as defined in the `create_driver` or `create_sw_package` command). For example, if your driver is named `my_peripheral_driver` and your setting's `displayName` is `small_driver`, BSPs with your driver have a setting `my_peripheral_driver.small_driver`. Thus each driver and software package has its own settings namespace.

**Setting Generation Name**

The setting `generationName` of `add_sw_setting` controls the physical name of the setting in the generated BSP files. The physical name corresponds to the definition being created in `public.mk` and `system.h`, or the `make` variable created in the BSP **Makefile**. The `generationName` is commonly the text that your software uses in conditionally-compiled code. For example, suppose your software creates a buffer as follows:

```c
unsigned int driver_buffer[MY_DRIVER_BUFFER_SIZE];
```

You can enter the exact text, `MY_DRIVER_BUFFER_SIZE`, in the `generationName` argument.

**Setting Default Value**

The `value` argument of `add_sw_setting` holds the default value of your setting. This value propagates to the generated BSP unless the end user of the driver or package (the BSP developer) changes the setting's assignment before BSP generation.

**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 "Software Build Tools Reference" chapter.
Setting Description

The description argument of add_sw_setting contains a brief description of the setting. The description argument is required. Place quotation marks (""") around the text of the description. The description text appears in the generated BSP summary.html document.

Setting Creation Example

```
#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 users set the setting to true or 1, the BSP defines MY_CUSTOM_DRIVER_SMALL in either system.h or the BSP public.mk file. When the user compiles the BSP, your driver is compiled with the appropriate routine incorporated in the object file. When a user disables the setting, MY_CUSTOM_DRIVER_SMALL is not defined.

You add the MY_CUSTOM_DRIVER_SMALL setting to your driver as follows using the add_sw_setting Tcl command:

```
add_sw_setting boolean_define_only system_h_define small_driver MY_CUSTOM_DRIVER_SMALL false "Enable the small implementation of the driver for my_peripheral"
```

**Note:** Each Tcl command must reside on a single line of the Tcl file. This example is wrapped due to space constraints.

Each argument has several variants.

For more information about detailed usage and restrictions, refer to the "Software Build Tools Reference" chapter.

Related Information

**Software Build Tools Reference** on page 15-1
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.

Related Information
Integrating a Device Driver in the HAL on page 7-15

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
Enabling Software Initialization on page 7-21

Creating alt_sys_init.c Based on Associated Header Files

The Software Build Tools (SBT) creates alt_sys_init.c based on the header files associated with each device driver and software package. For a device driver, the header file must define the macros <component name>_INSTANCE and <component name>_INIT.

Like a device driver, a software package provides an INSTANCE macro, which alt_sys_init() invokes once. A software package header file can optionally provide an INIT macro.

Example 7–4. Excerpt from an alt_sys_init.c File Performing Driver Initialization

```c
#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 );
}
**Defining Macros**

For example, `altera_avalon_jtag_uart.h` must define the macros `ALTERA_AVALON_JTAG_UART_INSTANCE` and `ALTERA_AVALON_JTAG_UART_INIT`. The purpose of these macros is as follows:

- The `*_INSTANCE` macro performs any required static memory allocation. For drivers, `*_INSTANCE` is invoked once per device instance, so that memory can be initialized on a per-device basis. For software packages, `*_INSTANCE` is invoked once.
- The `*_INIT` macro performs runtime initialization of the device driver or software package.

In the case of a device driver, both macros take two input arguments:

- The first argument, `name`, is the capitalized name of the device instance.
- The second argument, `dev`, is the lower case version of the device name. `dev` is the name given to the component at system generation time.

You can use these input parameters to extract device-specific configuration information from the `system.h` file.

The name of the header file must be as follows:

- Device driver: `<hardware component class>.h`. For example, if your driver targets the `altera_avalon_uart` component, the file name is `altera_avalon_uart.h`.
- Software packages `<package name>.h`. For example, if you create the software package with the following command:

  ```
  create_sw_package my_sw_package
  ```

  the header file is called `my_sw_package.h`.

  For more information about a complete example, refer to any of the Intel FPGA-supplied device drivers, such as the JTAG UART driver in `<Intel FPGA installation>/ip/sopc_builder_ip/altera_avalon_jtag_uart`.

**Note:** For optimal project rebuild time, do not include the peripheral header in `system.h`. It is included in `alt_sys_init.c`.

**Device Driver Source Code**

In addition to the header file, the component driver might need to provide compilable source code, to be incorporated in the BSP. This source code is specific to the hardware component, and resides in one or more C files (or assembly language files).

**Reducing Code Footprint in HAL Embedded Drivers**

The HAL provides several options for reducing the size, or footprint, of the BSP code. Some of these options require explicit support from device drivers. If you need to minimize the size of your software, consider using one or both of the following techniques in your custom device driver:

- Provide reduced footprint drivers. This technique usually reduces driver functionality.
- Support the lightweight device driver API. This technique reduces driver overhead. It need not reduce functionality, but it might restrict your flexibility in using the driver.

These techniques are discussed in the following sections.
Provide Reduced Footprint Drivers

The HAL defines a C preprocessor macro named ALT_USE_SMALL_DRIVERS that you can use in driver source code to provide alternate behavior for systems that require a minimal code footprint. If ALT_USE_SMALL_DRIVERS is not defined, driver source code implements a fully featured version of the driver. If the macro is defined, the source code might provide a driver with restricted functionality. For example a driver might implement interrupt-driven operation by default, but polled (and presumable smaller) operation if ALT_USE_SMALL_DRIVERS is defined.

When writing a device driver, if you choose to ignore the value of ALT_USE_SMALL_DRIVERS, the same version of the driver is used regardless of the definition of this macro.

You can enable ALT_USE_SMALL_DRIVERS in a BSP with the hal.enable_reduced_device_drivers BSP setting.

For more information, refer to the "Software Build Tools Reference" chapter.

Related Information
Software Build Tools Reference on page 15-1

Support the Lightweight Device Driver API

Using Character-Mode Functions
The lightweight device driver API allows you to minimize the overhead of character-mode device drivers. It does this by removing the need for the alt_fd file descriptor table, and the alt_dev data structure required by each driver instance.

If you want to support the lightweight device driver API on a character-mode device, you need to write at least one of the lightweight character-mode functions listed in the "Driver Functions for Lightweight Device Driver API" table (Table 7–7). Implement the functions needed by your software. For example, if you only use the device for stdout, you only need to implement the <component class>_write() function.

Table 7-6: Driver Functions for Lightweight Device Driver API

| Function          | Purpose                        | Example
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;component class&gt;_read()</td>
<td>Implements character-mode read functions</td>
<td>altera_avalon_jtag_uart_read()</td>
</tr>
<tr>
<td>&lt;component class&gt;_write()</td>
<td>Implements character-mode write functions</td>
<td>altera_avalon_jtag_uart_write()</td>
</tr>
<tr>
<td>&lt;component class&gt;_ioctl()</td>
<td>Implements device-dependent functions</td>
<td>altera_avalon_jtag_uart_ioctl()</td>
</tr>
</tbody>
</table>

Using Macros
When you build your BSP with ALT_USE_DIRECT_DRIVERS enabled, instead of using file descriptors, the HAL accesses your drivers with the following macros:

(7) Based on component altera_avalon_jtag_uart.
Invoking Macros in Your Application Software

- ALT_DRIVER_READ(instance, buffer, len, flags)
- ALT_DRIVER_WRITE(instance, buffer, len, flags)
- ALT_DRIVER_IOCTL(instance, req, arg)

These macros are defined in `<EDS install path>/components/altera_hal/HAL/inc/sys/alt_driver.h`.

These macros, together with the system-specific macros that the SBT creates in `system.h`, generate calls to your driver functions. For example, with lightweight drivers turned on, `printf()` calls the HAL write() function, which directly calls your driver's `<component class>_write()` function, bypassing file descriptors.

You can enable ALT_USE_DIRECT_DRIVERS in a BSP with the `hal.enable_lightweight_device_driver_api` BSP setting.

For more information, refer to the "Software Build Tools Reference" chapter.

Related Information
Software Build Tools Reference on page 15-1

Invoking Macros in Your Application Software

You can also take advantage of the lightweight device driver API by invoking ALT_DRIVER_READ(), ALT_DRIVER_WRITE() and ALT_DRIVER_IOCTL() in your application software. To use these macros, include the header file `sys/alt_driver.h`. Replace the instance argument with the device instance name macro from `system.h`; or if you are confident that the device instance name never changes, you can use a literal string, for example custom_uart_0.

Calling Direct Without Macros

Another way to use your driver functions is to call them directly, without macros. If your driver includes functions other than `<component class>_read()`, `<component class>_write()` and `<component class>_ioctl()`, you must call those functions directly from your application.

HAL Namespace Allocation

To avoid conflicting names for symbols defined by devices in the hardware system, all global symbols need a defined prefix. Global symbols include global variable and function names. For device drivers, the prefix is the name of the component followed by an underscore. Because this naming can result in long strings, an alternate short form is also permitted. This short form is based on the vendor name, for example alt_ is the prefix for components published by Intel FPGA. It is expected that vendors test the interoperability of all components they supply.

For example, for the altera_avalon_jtag_uart component, the following function names are valid:

- altera_avalon_jtag_uart_init()
- alt_jtag_uart_init()

The following names are invalid:

- avalon_jtag_uart_init()
- jtag_uart_init()

As source files are located using search paths, these namespace restrictions also apply to file names for device driver source and header files.
Overriding the HAL Default Device Drivers

All components can elect to provide a HAL device driver.

For more information, refer to “Integrating a Device Driver in the HAL” on page 7–17.

However, if the driver supplied with a component is inappropriate for your application, you can override the default driver by supplying a different driver.

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

For more information about selecting device drivers, refer to “Using the BSP Editor” in the “Getting Started with the Graphical User Interface” chapter.

On the command line, you specify a custom driver with the following BSP Tcl command:

```
set_driver <driver name> <component name>
```

For example, if you are using the nios2-bsp command, you replace the default driver for uart0 with a driver called custom_driver as follows:

```
nios2-bsp hal my_bsp --cmd set_driver custom_driver uart0
```

Related Information

- Integrating a Device Driver in the HAL on page 7-15
- Getting Started with the Graphical User Interface on page 2-1
This chapter discusses how to write programs to handle exceptions in the Nios® II processor architecture. Emphasis is placed on how to process hardware interrupt requests by registering a user-defined interrupt service routine (ISR) with the hardware abstraction layer (HAL). This information applies to embedded software projects created with the Software Build Tools (SBT), either in Eclipse or on the command line.

For more information and low-level details about handling exceptions and hardware interrupts on the architecture, refer to the "Programming Model" chapter.

Related Information
Programming Model

Exception Handling Overview

The processor provides the following exception types:

- Hardware interrupts
- Software exceptions, which fall into the following categories:
  - Unimplemented instructions
  - Software traps
  - Miscellaneous exceptions

The processor offers two distinct approaches to handling hardware interrupts:

- The internal interrupt controller (IIC)
- The external interrupt controller (EIC) interface

The interrupt controllers are discussed in detail in the “Interrupt Controllers” chapter.

Related Information
Interrupt Controllers on page 8-3

Exception Handling Terminology

The following list of HAL terms outlines basic exception handling concepts:
- Application context—The status of the processor and the HAL during normal program execution, outside of exception funnels and handlers.

- Context switch—The process of saving the processor’s registers on a software exception or hardware interrupt, and restoring them on return from the exception handling routine or ISR.

- Exception—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. Exceptions include software exceptions and hardware interrupts.

- Exception context—The status of the processor and the HAL after a software exception or hardware interrupt, when funnel code, a software exception handler, or an ISR is executing.

- Exception handling system—The complete system of software routines that service all exceptions, including hardware interrupts, and pass control to software exception handlers and ISRs as necessary.

- Exception (or interrupt) latency—The time elapsed between the event that causes the exception (such as an unimplemented instruction or interrupt request) and the execution of the first instruction at the exception (or interrupt vector) address.

- Exception (or interrupt) response time—The time elapsed between the event that causes the exception and the execution of the handler.

- Exception overhead—Additional processing required to service a software exception or hardware interrupt, including HAL-specific processing and RTOS-specific processing if applicable.

- Funnel code—HAL-provided code that sets up the correct processor environment for an exception-specific handler, such as an ISR.

- Handler—Code specific to the exception type. The handler code is distinct from the funnel code, which takes care of general exception overhead tasks.

- Hardware interrupt—An exception caused by an explicit hardware request signal from an external device. A hardware interrupt diverts the processor’s execution flow to a ISR, to ensure that a hardware condition is handled in a timely manner.

- Implementation-dependent instruction—A processor instruction that is not supported on all implementations of the core. For example, the mul and div instructions are implementation-dependent, because they are not supported on the /e core.

- Interrupt—Hardware interrupt.

- Interrupt controller—Hardware enabling the processor to respond to an interrupt by transferring control to an ISR.

- Interrupt request (IRQ)—Hardware interrupt.

- Interrupt service routine (ISR)—A software routine that handles an individual hardware interrupt.

- Invalid instruction—An instruction that is not defined for any implementation of the processor.

- Maskable exceptions—Exceptions that can be disabled with the status.PIE flag, including internal hardware interrupts, maskable external hardware interrupts, and software exceptions, but not including nonmaskable external interrupts.

- Maximum disabled time—The maximum amount of continuous time that the system spends with maskable exceptions 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 a trap instruction.

For more information, refer to the “Miscellaneous Exceptions” chapter.

- Nested interrupts—See pre-emption.

- Pre-emption—The process of a high-priority interrupt taking control when a lower-priority ISR is already running. Also: nested interrupts.
• Software exception—An exception caused by a software condition; that is, any exception other than a hardware interrupt. This includes unimplemented instructions and trap instructions.
• Unimplemented instruction—An implementation-dependent instruction that is not supported on the particular core implementation that is in your system. For example, in the /e core, mul and div are unimplemented.
• Worst-case exception (or interrupt) 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 exception (or interrupt) occurs at the beginning of the masked or disabled time.

Related Information
Miscellaneous Exceptions on page 8-31

Interrupt Controllers

The configuration of exception processing depends on the type of hardware interrupt controller. You select the hardware interrupt controller when you instantiate the processor in the system integration tool.

For more information and details about selecting a hardware interrupt controller, refer to the "Instantiating the Processor" chapter of the Processor Reference Handbook.

Related Information
Instantiating the Nios II Processor
For more information and details about selecting a hardware interrupt controller.

Internal Interrupt Concepts

With the IIC, exception handling is implemented in classic RISC fashion. All exception types, including hardware interrupts, are dispatched through a single top-level exception funnel. This means that all exceptions (hardware and software) are handled by code residing at a single location, the exception address.

The IIC is a simple, nonvectored hardware interrupt controller. Upon receipt of an interrupt request, the IIC transfers control to the general exception address. The hardware indicates which IRQ is currently asserted, and allows software to mask individual interrupts.

With the IIC, the HAL interrupt funnel identifies the hardware interrupt cause in software, and dispatches the registered ISR.

The IIC is available in all revisions of the processor.

External Interrupt Concepts

The EIC interface enables the processor to work with a separate external interrupt controller component. An EIC can be a custom component that you provide. Intel FPGA provides an example of an EIC, the vectored interrupt controller (VIC).

For more information about the VIC, refer to the "Vectored Interrupt Controller Core" chapter in the Embedded Peripherals IP User Guide.

With an EIC, hardware interrupts are handled separately from software exceptions. Hardware interrupts have separate vectors and funnels. Each interrupt can have its own handler, or handlers can be shared. Software exception handling is the same as with the IIC.

The EIC interface provides extensive capabilities for customizing your interrupt hardware. You can design, connect and configure an interrupt controller that is optimal for your application.
When an external hardware interrupt occurs, the processor transfers control to an individual vector address, which can be unique for each interrupt. The HAL provides the following services:

- Registering ISRs
- Setting up the vector table
- Transferring control from the vector table to your ISR

An EIC can be used with shadow register sets. A shadow register set is a complete alternate set of general-purpose registers, which can be used to maintain a separate runtime context for an ISR.

An EIC provides the following information about each hardware interrupt:

**Related Information**

**Vectored Interrupt Controller Core**

**Requested Handler Address**

The requested handler address (RHA) specifies the address of the funnel associated with the hardware interrupt. The availability of an RHA for each interrupt allows the processor to jump directly to the interrupt funnel specific to the interrupting device, reducing interrupt latency.

**Requested Interrupt Level**

The processor uses the requested interrupt level (RIL) to prioritize the hardware interrupt request versus any interrupt it is currently processing. While handling an interrupt, the processor normally only takes higher-level interrupts.

**Requested Register Set**

If shadow register sets are implemented on the core, an EIC specifies a requested register set (RRS) when it asserts an interrupt request. When the processor takes the hardware interrupt, the processor switches to the requested register set. When an interrupt has a dedicated register set, the ISR avoids the overhead of saving registers for a context switch.

Multiple hardware interrupts can be configured to share a register set. However, at run time, the processor does not allow pre-emption between interrupts assigned to the same register set unless this feature is specifically enabled. In this case, the ISRs must be written so as to avoid register corruption.

For more information, refer to an example of a driver that manages pre-emption within a register set in the "Vectored Interrupt Controller Core" chapter in the *Embedded Peripherals IP User Guide*.

**Related Information**

**Vectored Interrupt Controller Core**

**Requested NMI Mode**

If the interrupt is configured as a nonmaskable interrupt (NMI), the EIC asserts requested NMI (RNMI). Any hardware interrupt can be nonmaskable, depending on the configuration of the EIC. An NMI typically signals a critical system event requiring immediate handling, to ensure either system stability or deterministic real-time performance.

**Shadow Register Sets**

Although shadow register sets can be implemented independently of the EIC interface, typically the two features are used together. Combining shadow register sets with an appropriate EIC, you can minimize or eliminate the context switch overhead for critical hardware interrupts.
Latency and Response Time

Exception (interrupt) latency, as defined in the previous section, is the time required for the hardware to respond to an exception. Response time, in contrast, is the time required to begin executing code specific to the exception cause, such as a particular ISR. Response time includes latency plus 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.

Internal or External Interrupt Controller

The IIC is nonvectored, requiring the processor to dispatch ISRs with a software routine. An EIC, by contrast, can be vectored. With a vectored EIC, such as the Intel FPGA® VIC, ISR dispatch is managed by hardware, eliminating the processing time required for ISR dispatch, and substantially reducing hardware interrupt response time.

An EIC has no impact on software exception latency or response time.

Shadow Register Sets

In conjunction with an EIC, shadow register sets speed up hardware interrupt response by making it unnecessary to save registers on the stack. This feature has no impact on interrupt latency, but significantly reduces interrupt response time.

Shadow register sets have no impact on software exception response time.

How the Hardware Works

The processor can respond to exceptions including software exceptions and hardware interrupts. When the processor responds to an exception, it performs the following tasks:

- Saves the status register in estatus. This means that if hardware interrupts are enabled, the PIE bit of estatus is set.
- Disables hardware interrupts.
- Saves the next execution address in ea (r29).
- Transfers control to the appropriate exception address, as follows:
  - Software exception or internal hardware interrupt—processor general exception address
  - External hardware interrupt—Device-specific interrupt address

All exception types are precise. This means that after an exception is handled, the processor can re-execute the instruction that caused the exception.

The processor always re-executes the instruction after the software exception handler or ISR has completed, when the exception processing system returns to the application context.

Several exception types, such as the advanced exceptions, are optional in the processor core. The presence of these exception types depends on how the hardware designer configures the core at the time of hardware generation.
The processor’s response to hardware interrupts depends on which interrupt controller is implemented. The following sections describe the hardware behavior with each interrupt controller.

For more information about the processor exception controller and hardware interrupt controllers, including a list of optional exception types, refer to the "Processor Architecture" chapter of the Processor Reference Handbook.

Related Information

- Invalid Instructions on page 8-32
- Processor Architecture

How the Internal Interrupt Controller Works

With the IIC, 32 independent hardware interrupt signals are available. These interrupt signals allow software to prioritize interrupts, although the interrupt signals themselves have no inherent priority.

Note: With the IIC, exceptions are not vectored. Therefore, the same exception address receives control for all types of exceptions. The general exception funnel at that address must determine the type of software exception or hardware interrupt.

How an External Interrupt Controller Works

With an EIC, the processor supports an arbitrary number of independent hardware interrupt signals. Interrupts are typically vectored, with interrupt priority levels associated in hardware. Vectoring allows the processor to transfer control directly to each ISR. Hardware interrupt levels allow the most critical interrupts to pre-empt lower-priority interrupts. Because both of these features are implemented in hardware, the system can handle an interrupt without executing general exception funnel code.

Note: The details of hardware interrupt vectoring and prioritization are specific to the EIC implementation.

For more information, refer to an example of an EIC implementation in the "Vectored Interrupt Controller Core" chapter in the Embedded Peripherals IP User Guide.

Note: The HAL supports external interrupt controllers only if they are connected in one of the following ways:

- Directly to the EIC interface
- Through the daisy-chain port on another EIC

Related Information

Vectored Interrupt Controller Core

Interrupt Service Routines

Software often communicates with peripheral devices using hardware 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 hardware interrupts.

For examples of HAL ISRs, refer to existing handlers for Intel FPGA components.

For more information about the Intel FPGA-provided HAL handlers, refer to the "Developing Programs Using the Hardware Abstraction Layer" section.

**Related Information**

- Developing Programs Using the Hardware Abstraction Layer on page 6-1

---

**HAL APIs for Hardware Interrupts**

The HAL provides an enhanced application program interface (API) for writing, registering and managing ISRs. This API is compatible with both internal and external hardware interrupt controllers.

Intel FPGA also supports a legacy hardware interrupt API. This API supports only the IIC. If you have a custom driver written prior to version 9.1, it uses the legacy API.

Both 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

Both interrupt APIs support the following types of BSPs:

- HAL BSP without an RTOS
- HAL-based RTOS BSP, such as a MicroC/OS-II BSP

**Note:** The legacy API is deprecated. Write new drivers using the enhanced API, even if they are only intended to support the IIC. Drivers for devices supporting an EIC must use the enhanced API. Existing legacy drivers continue to be supported until further notice. Make plans to port them to the enhanced API.

When an EIC is present, the controller’s driver provides driver settings for the BSP, which can be used to configure the driver. The number and types of the settings depends on the EIC implementation and the number of EICs present.

For more information, refer to an example of EIC driver settings in the "Vectored Interrupt Controller Core" chapter in the *Embedded Peripherals IP User Guide*.

**Related Information**

- Vectored Interrupt Controller Core
- The Enhanced HAL Interrupt API on page 8-8

**Selecting an Interrupt API**

When the SBT creates a BSP, it determines whether the BSP must implement the legacy interrupt API. Each driver that supports the enhanced API publishes this capability to the SBT through its `<driver name>_sw.tcl` file. The BSP implements the enhanced API if all drivers support it. It implements the legacy API only if required by the drivers.
In determining the interrupt API to use, the SBT ignores any devices whose interrupts are not connected to the processor associated with the BSP.

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 either `legacy_interrupt_api`, `enhanced_interrupt_api`, or both.

Drivers supporting the enhanced API always publish that support. If `supported_interrupt_apis` is undefined, the SBT assumes that the driver only supports the legacy API.

Starting in 9.1, all Intel FPGA device drivers support both APIs. These drivers can be used in a BSP along with legacy drivers. The SBT determines whether the legacy API is required, and implements it only if it is required. If there are no drivers requiring the legacy API, the BSP implements the enhanced API.

A driver can be written to support only the enhanced API. However, you cannot combine such a driver with legacy drivers.

For more information and details about writing a driver to support both APIs, refer to the "Supporting Multiple Interrupt APIs" chapter.

### The Enhanced HAL Interrupt API

#### Table 8-1: Enhanced HAL Interrupt API Functions that Manage Hardware Interrupt Processing

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Implemented By</th>
</tr>
</thead>
<tbody>
<tr>
<td>alt_ic_isr_register()</td>
<td>Interrupt controller driver ()</td>
</tr>
<tr>
<td>alt_ic_irq_enable()</td>
<td>Interrupt controller driver ()</td>
</tr>
<tr>
<td>alt_ic_irq_disable()</td>
<td>Interrupt controller driver ()</td>
</tr>
<tr>
<td>alt_ic_irq_enabled()</td>
<td>Interrupt controller driver ()</td>
</tr>
<tr>
<td>alt_irq_disable_all()</td>
<td>HAL</td>
</tr>
<tr>
<td>alt_irq_enable_all()</td>
<td>HAL</td>
</tr>
<tr>
<td>alt_irq_enabled()</td>
<td>HAL</td>
</tr>
</tbody>
</table>

**Note:** If the system is based on an EIC, these functions must be implemented by the EIC driver. If the system is based in the IIC, the functions are implemented by the HAL. For more information about each function, refer to the "HAL API Reference" section.

### Using the Enhanced HAL Interrupt API to Implement ISRs

Using the enhanced HAL API to implement ISRs requires that you perform the following steps:

1. Write your ISR that handles hardware interrupts for a specific device.
2. Ensure that your program registers the ISR with the HAL by calling the `alt_ic_isr_register()` function. `alt_ic_isr_register()` enables hardware interrupts for you.
The SBT inserts the following symbol definitions in `system.h`, indicating the configuration of the processor’s interrupt-related hardware options:

- **NIOS2_EIC_PRESENT** — If defined, indicates that one or more EICs are present
- **NIOS2_NUM_OF_SHADOW_REG_SETS** — Indicates how many shadow register sets are present. The maximum value is 63. If there are no shadow register sets, the value is 0.

### The External Interrupt Controller Driver

To be compliant with the HAL enhanced interrupt API, the driver for an EIC must support the functions listed under “The Enhanced HAL Interrupt API” chapter.

For more information, refer to the “The Enhanced HAL Interrupt API” chapter.

In addition, it can provide functions to support any special hardware features.

For more information, refer to the “Using the HAL Interrupt API with the VIC” chapter.

**Related Information**

- [The Enhanced HAL Interrupt API](#) on page 8-8
- [Using the HAL Interrupt API with the VIC](#) on page 8-9

### Using the HAL Interrupt API with the VIC

The Intel FPGA driver for the VIC component supports the HAL enhanced interrupt API.

The VIC driver provides support for multiple, daisychained VIC devices. It also includes support for shadow register sets. A BSP driver setting allows you to enable automatic pre-emption (fast nested interrupts). Automatic pre-emption means that the processor leaves maskable exceptions enabled when accepting a hardware interrupt.

For more information about fast nested interrupts, refer to “Exception Processing” in the "Programming Model" chapter of the [Nios II Processor Reference Handbook](#).

The VIC device driver also provides the following device-specific functions:

- `int alt_vic_sw_interrupt_set(alt_u32 ic_id, alt_u32 irq);`
- `int alt_vic_sw_interrupt_clear(alt_u32 ic_id, alt_u32 irq);`
- `alt_u32 alt_vic_sw_interrupt_status(alt_u32 ic_id, alt_u32 irq);`
- `int alt_vic_irq_set_level(alt_u32 ic_id, alt_u32 irq, alt_u32 level);`

For more information, refer to a detailed discussion of the VIC device-specific driver routines in the "Vectored Interrupt Controller Core" chapter in the [Embedded Peripherals IP User Guide](#).

The EIC driver controls where hardware interrupt vector tables are located. For example, the Intel FPGA VIC driver locates the vector table in the `.text` section by default, but allows you to position the vector table in a different section with a driver setting.

**Note:** The memory in which you place the vector table must be connected to both instruction and data master ports on the processor.

**Related Information**

- [Programming Model](#)
- [Vectored Interrupt Controller Core](#)

### The Legacy HAL Interrupt API

The legacy HAL interrupt API defines the following functions to manage hardware interrupt processing:
Using the Legacy HAL API to Implement ISRs

- `alt_irq_register()`
- `alt_irq_disable()`
- `alt_irq_enable()`
- `alt_irq_disable_all()`
- `alt_irq_enable_all()`
- `alt_irq_interruptible()`
- `alt_irq_non_interruptible()`
- `alt_irq_enabled()``

For more information about these functions, refer to the "HAL API Reference" chapter.

Legacy drivers do not define the `supported_interrupt_apis` property. The absence of this property indicates to the SBT that they require the legacy interrupt API.

Using the Legacy HAL API to Implement ISRs

Using the legacy HAL API to implement ISRs requires that you perform the following steps:

1. Write your ISR that handles hardware interrupts for a specific device.
2. Ensure that your program registers the ISR with the HAL by calling the `alt_irq_register()` function. `alt_irq_register()` enables hardware interrupts for you, by calling `alt_irq_enable_all()`.

Supporting Multiple Interrupt APIs

When you write or update a custom device driver, Intel FPGA recommends that you write it in one of two ways:

- Write it to support the enhanced HAL interrupt API—Write the driver this way if you intend to use it only in combination with other drivers supporting the enhanced API.
- Write it to support both the enhanced and the legacy API—Write the driver this way if you need to use it in combination with legacy drivers supporting only the legacy API.

**Note:** Intel FPGA recommends using the enhanced API even if your processor implements the IIC. The enhanced API supports both types of interrupt controller, and the legacy API is deprecated.

When the SBT selects the interrupt API, it defines one of the following symbols in `system.h`, to identify which interrupt API is available:

- `ALT_ENHANCED_INTERRUPT_API_PRESENT`—Defined if the enhanced API is implemented
- `ALT_LEGACY_INTERRUPT_API_PRESENT`—Defined if the legacy API is implemented

In your driver code, use these symbols to determine which API calls to make.

To support both APIs, your driver must publish its interrupt API support by way of a software property. In your driver's `<driver name>_sw.tcl` file, use the `set_sw_property` command to set `supported_interrupt_apis` to both `legacy_interrupt_api` and `enhanced_interrupt_api`.

For more information about the `set_sw_property` command, refer to the “Software Build Tools Tcl Commands” section of the "Software Build Tools Reference" section.

**Related Information**

- [Software Build Tools Reference](#) on page 15-1
HAL ISR Restrictions

When your system has an EIC, the HAL interrupt support imposes the following restrictions:

- Nonmaskable hardware interrupts must use a shadow register set.
- Nonmaskable hardware interrupts cannot share a register set with a maskable hardware interrupt.

Writing an ISR

The ISR you write must match the prototype that alt_ic_isr_register() expects. The prototype for your ISR function must match the following prototype:

```c
void (*alt_isr_func) (void* isr_context)
```

The parameter definition of `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 an hardware interrupt condition is specific to the peripheral.

For more information, refer to the relevant chapter in the "Embedded Peripherals IP User Guide".

When the ISR has finished servicing the hardware interrupt, it must return to the HAL interrupt funnel that called it.

**Note:** If you write your ISR in assembly language, use `ret` to return. The HAL interrupt funnel issues an `eret` after restoring the application context.

Related Information

Embedded Peripherals IP User Guide

Using Interrupt Funnels

The HAL creates a vector table for each EIC connected to the processor. In the vector table, the HAL inserts a branch to the correct funnel for each interrupt-driven device supported by the BSP, depending on the device driver characteristics and pre-emption settings. Funnels can be shared by multiple hardware interrupts, if the drivers have compatible characteristics.

The funnel code receives control from the general exception or interrupt vector, depending on which interrupt controller is implemented. The funnel performs tasks such as switching 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.

The HAL includes the following interrupt funnels:

- Shadow register set, pre-emption disabled—Hardware interrupt assigned to a shadow register set, with pre-emption within the register set disabled. This funnel does not preserve register context. Hardware guarantees that only one ISR runs with the shadow register set at any time.
- Shadow register set, pre-emption enabled—Hardware interrupt assigned to a shadow register set. An interrupt can pre-empt another interrupt using the same register set. This funnel preserves register context, so that handlers is assigned to the same register set do not corrupt one another’s context.
- Nonmaskable interrupt—Nonmaskable hardware interrupt assigned to a shadow register set, with pre-emption within the register set disabled. This funnel does not preserve register context. Hardware guarantees that only one ISR runs in the shadow register set at any time.
The HAL funnel code is called from the vector table.

Running in a Restricted Environment
ISRs run in a restricted environment. A large number 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 hardware 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 hardware interrupt that never occurs because interrupts are disabled.

Related Information
HAL API Reference on page 14-1

Managing Pre-emption
The HAL enhanced interrupt API supports interrupt pre-emption. When pre-emption is enabled, a higher-level interrupt can take control even if an ISR is already running. A device driver must be specifically written to function correctly under pre-emption. When a device driver supports pre-emption, it publishes this capability through the isr_preemption_supported driver setting. When constructing the BSP, the SBT checks each device driver to determine whether it supports pre-emption. If all drivers in the BSP support pre-emption, it is enabled.

Legacy device drivers do not publish the isr_preemption_supported property. Therefore the SBT assumes that they do not support pre-emption. If your legacy custom driver supports pre-emption, and you want to allow pre-emption in the BSP, you must update the driver to use the enhanced interrupt API.

Note: To enable the enhanced interrupt API, ensure that all drivers in the system are updated to use the enhanced interrupt API.

For more information and details about the isr_preemption_supported driver setting, refer to the set_sw_property command in the “Software Build Tools Tcl Commands” section of the "Software Build Tools Reference" chapter.

Operating systems can also publish the isr_preemption_supported property.

The HAL enhanced interrupt API supports automatic pre-emption. Automatic pre-emption means that maskable exceptions remain enabled when the processor accepts the hardware interrupt. This means that your ISR can immediately be pre-empted by a higher-level ISR, without any need to execute the eret instruction.

Automatic pre-emption can only take place when the pre-empting hardware interrupt uses a different register set from the interrupt being pre-empted.

Automatic pre-emption is only available if you enable it in the BSP settings.

Related Information
Software Build Tools Reference on page 15-1
Registering an ISR with the Enhanced Interrupt API

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:

```c
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 daisychained EICs, `ic_id` identifies the EIC in the daisy chain. With the IIC, `ic_id` is not significant.
- `irq` is the hardware 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, IRQ0 represents the highest priority interrupt and IRQ31 is the lowest.
  - For an EIC, `irq` is the interrupt port ID.
- `isr_context` points to a data structure associated with the device driver instance. `isr_context` is 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.
- `isr` is a pointer to the ISR function that is called in response to IRQ number `irq`. The ISR function prototype is:
  ```c
  void (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.

- `flags` is reserved.

**Related Information**

The Enhanced HAL Interrupt API on page 8-8

**Methods the HAL Uses to Register the ISR**

The HAL registers the ISR by one of the following methods:

- For the IIC, by storing the function pointer, `isr`, in a lookup table.
- For an EIC, by configuring the vector table with the appropriate funnel code

For more information, refer to the “Using Interrupt Funnels” chapter.

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 hardware interrupt (as defined by `irq`) is enabled on return from `alt_ic_isr_register()`.

**Note:** Hardware-specific initialization might also be required.

When a specific interrupt occurs, the HAL code ensures that the registered ISR is correctly dispatched.

For more information and details about hardware interrupt initialization specific to your peripheral, refer to the relevant chapter of the Embedded Peripherals IP User Guide.
For more information about `alt_ic_isr_register()`, refer to the "HAL API Reference" chapter.

**Note:** The HAL legacy interrupt API provides a different function for registering hardware interrupts. For all new and updated drivers, Intel FPGA recommends using the enhanced API described in this section.

For more information about the legacy API function, `alt_irq_register()`, refer to the "HAL API Reference" chapter.

**Related Information**
- [Using Interrupt Funnels](#) on page 8-11
- [HAL API Reference](#) on page 14-1
- [Embedded Peripherals IP User Guide](#)

### Enabling and Disabling Interrupts

The HAL enhanced interrupt API provides the functions `alt_ic_irq_disable()`, `alt_ic_irq_enable()`, `alt_ic_irq_enabled()`, `alt_irq_disable_all()`, `alt_irq_enable_all()`, and `alt_irq_enabled()` to allow a program to disable hardware interrupts for certain sections of code, and reenable them later. `alt_ic_irq_disable()` and `alt_ic_irq_enable()` allow you to disable and enable individual interrupts. `alt_irq_disable_all()` disables all interrupts, and returns a context value. To reenable hardware interrupts, you call `alt_irq_enable_all()` and pass in the context parameter. In this way, 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 hardware 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.

**Note:** The HAL legacy interrupt API provides different functions for enabling and disabling individual interrupts. For all new and updated drivers, Intel FPGA recommends using the enhanced API described in this section.

For more information about the legacy API functions, `alt_irq_disable()` and `alt_irq_enable()`, refer to the "HAL API Reference" chapter.

**Related Information**
- [Keep Interrupts Enabled](#) on page 8-18
- [HAL API Reference](#) on page 14-1

### Configuring an External Interrupt Controller

The driver for an EIC provides specialized driver settings that are created at the time you generate the BSP. These settings customize the driver to the EIC configuration found in the system. The number and type of settings depends on the EIC implementation, as well as on the number and configuration of EICs in the hardware system. The SBT creates the BSP with default values, selected to ensure useful system performance. You can optimize these settings at the time you create the BSP. For details of how to manipulate the EIC driver settings, refer to the documentation for your specific EIC.
The driver for an EIC can provide specialized functions to manage any implementation-specific features of the EIC. An example would be modifying interrupt priority levels at runtime.

For more information, refer to the examples in the "Vectored Interrupt Controller Core" chapter in the *Embedded Peripherals IP User Guide*.

**Related Information**

**Vectored Interrupt Controller Core**

### C Example

**An ISR to Service a Button PIO Interrupt**

This example is based on a 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 8-1: Example 8–1. An ISR to Service a Button PIO Interrupt**

```c
#include "system.h"
#include "altera_avalon_pio_regs.h"
#include "alt_types.h"
#ifdef ALT_ENHANCED_INTERRUPT_API_PRESENT
static void handle_button_interrupts(void* context)
#else
static void handle_button_interrupts(void* context, alt_u32 id)
#endif
{
    /* 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);
}
```

### 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.
  • With an EIC, the processor branches to the address in the vector table, which transfers control to
    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 8-2: Example 8–2. Registering the Button PIO ISR with the HAL

```c
#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. */
    #ifdef ALT_ENHANCED_INTERRUPT_API_PRESENT
    alt_ic_isr_register(BUTTON_PIO_IRQ_INTERRUPT_CONTROLLER_ID,
                         BUTTON_PIO_IRQ,
                         handle_button_interrupts,
                         edge_capture_ptr, 0x0);
    #else
    alt_irq_register( BUTTON_PIO_IRQ,
                        edge_capture_ptr,
                        handle_button_interrupts );
    #endif
}
```

Note: Additional software examples that demonstrate implementing ISRs, such as the
count_binary example project template, are installed with the Embedded Design
Suite (EDS).

Upgrading to the Enhanced HAL Interrupt API

If you have custom device drivers, Intel FPGA recommends that you upgrade them to use the enhanced
HAL interrupt API. The enhanced API maintains compatibility with the IIC, while supporting external
interrupt controllers. The legacy HAL interrupt API is deprecated.

If you plan to use an EIC, you must upgrade your custom driver to the enhanced HAL interrupt API.
Upgrading your device driver is very simple, requiring only minor changes to some function calls.
For more information and details of the API functions, refer to the "HAL API Reference" chapter.
If your upgraded driver might need to function in a BSP with legacy drivers, code it to support both APIs. For more information, refer to the “Supporting Multiple Interrupt APIs” chapter.

Related Information
• Supporting Multiple Interrupt APIs on page 8-10
• HAL API Reference on page 14-1

Improving ISR Performance

If your software uses hardware interrupts extensively, the performance of ISRs is probably the most critical determinant of your overall software performance.

Software Performance Improvements

In improving your ISR performance, you probably consider software changes first. However, in some cases it might require less effort to implement hardware design changes that increase system efficiency.

For more information about hardware optimizations, refer to the “Hardware Performance Improvements” chapter.

The following sections describe changes you can make in the software design to improve ISR performance.

Related Information
Hardware Performance Improvements on page 8-22

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 hardware 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.
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 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” in the "Developing Programs Using the Hardware Abstraction Layer chapter”.

Related Information
Developing Programs Using the Hardware Abstraction Layer on page 6-1

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).

Use Double Buffering
Using DMA to transfer large data buffers might not provide a large performance increase if the processor must wait for DMA transactions to complete before it can perform the next task.

Double buffering allows the processor to process one data buffer while the hardware is transferring data to or from another.

Keep Interrupts Enabled
When interrupts are disabled, the processor cannot respond quickly to hardware 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.

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” in the "Developing Programs Using the Hardware Abstraction Layer” chapter.

For more information about tightly-coupled memory, refer to the "Cache and Tightly-Coupled Memory” chapter.

Related Information
• Developing Programs Using the Hardware Abstraction Layer on page 6-1
Use a Separate Exception Stack

The HAL implements two types of separate exception stack. Their availability depends on the interrupt controller, as described in this section. The "Separate Exception Stack Usage" table (Table 8–3) outlines the availability of separate exception stacks, and how they can be used with each type of interrupt controller.

**Note:** Using a separate exception stack entails a slight additional overhead. When processing a software exception or hardware interrupt, the processor must execute an additional instruction on entry and exit, to change the stack pointer. Take this additional processing time into account if your interrupt response requirements are extremely strict.

**Separate General Exception Stack**

The separate general exception stack is available with either the internal or the external interrupt controller.

Use the `hal.linker.enable_exception_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.

**Separate Hardware Interrupt Stack**

The separate hardware interrupt stack is available with the EIC interface. The separate hardware interrupt stack is not applicable to the IIC. With the IIC, hardware interrupts and software exceptions use the same stack. The following BSP settings enable you to control the separate hardware interrupt stack:

- `hal.linker.enable_interrupt_stack` enables a separate hardware interrupt stack.
- `hal.linker.interrupt_stack_size` controls the size of the hardware interrupt stack.
- `hal.linker.interrupt_stack_memory_region_name` enables you to control where the hardware interrupt stack is positioned in memory.

The HAL funnel code takes care of correctly changing the stack pointer on entry to and exit from an ISR.

Table 8-3: Separate Exception Stack Usage

<table>
<thead>
<tr>
<th>Interrupt Controller</th>
<th>BSP Settings</th>
<th>Application Stack</th>
<th>General Exception Stack</th>
<th>Hardware Interrupt Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal</td>
<td>No</td>
<td>—</td>
<td>Application</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>—</td>
<td>Application</td>
<td>—</td>
</tr>
</tbody>
</table>

*Note: The table is incomplete and requires filling in the missing values.*
Use Nested Hardware 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 hardware 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 hardware 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.

Note: Nested ISRs increase the processing time for lower-priority hardware interrupts.

Note: If your ISR is located in a vector table, the HAL does not provide funnel code. In this case, your code must manage the stack pointer, as well as all other funnel code functions.

For more information about implementing a separate hardware interrupt stack, refer to AN595: Vectored Interrupt Controller Applications and Usage.

Related Information
AN595: Vectored Interrupt Controller Usage and Applications

Note: If your ISR is located in a vector table, the HAL does not provide funnel code. In this case, your code must manage the stack pointer, as well as all other funnel code functions.

For more information about implementing a separate hardware interrupt stack, refer to AN595: Vectored Interrupt Controller Applications and Usage.

Use Nested Hardware 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 hardware 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 hardware 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.

Note: Nested ISRs increase the processing time for lower-priority hardware interrupts.
If your ISR is very short, it might not be worth the overhead to enable nested hardware 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 IIC, you cannot nest hardware interrupts.

For more information about separate exception stacks, refer to “Use a Separate Exception Stack”.

**Related Information**

Use a Separate Exception Stack on page 8-19

### Nested Hardware Interrupts with the Internal Interrupt Controller

To implement nested hardware interrupts with the IIC, use the `alt_irq_interruptible()` and `alt_irq_non_interruptible()` legacy functions to bracket code in a processor-intensive ISR.

**Note:** The legacy API is deprecated. Write new drivers using the enhanced API, even if they are only intended to support the IIC. Drivers for devices supporting an EIC must use the enhanced API. Existing legacy drivers continue to be supported until further notice. Make plans to port them to the enhanced API.

If you choose to continue using the legacy APIs, you must first comment out `#define ALT_ENHANCED_INTERRUPT_API_PRESENT` in `system.h`.

The call to `alt_irq_interruptible()` adjusts the interrupt mask so that higher-priority interrupts can take control from the running ISR. When your ISR calls `alt_irq_non_interruptible()`, the interrupt mask is returned to its previous state.

**Note:** If your ISR calls `alt_irq_interruptible()`, it must call `alt_irq_non_interruptible()` before returning. Otherwise, the HAL exception handling system might lock up.

### Nested Hardware Interrupts with an External Interrupt Controller

The HAL enhanced interrupt API supports nested hardware interrupts, also known as interrupt pre-emption. A device driver must be specifically written to function correctly under pre-emption.

Legacy device drivers do not publish the `isr_preemption_supported` property. Therefore the SBT assumes that they do not support pre-emption. If your legacy custom driver supports pre-emption, and you want to allow pre-emption in the BSP, you must update the driver to use the enhanced HAL interrupt API.

The HAL enhanced interrupt API also supports automatic pre-emption. Automatic pre-emption means that maskable exceptions remain enabled when the processor accepts the hardware interrupt.

For more information about pre-emption with an EIC, refer to the “Managing Pre-Emption” chapter.

In the vector table, the HAL inserts a branch to the correct funnel for each hardware interrupt, depending on the pre-emption settings.

**Related Information**

Managing Pre-Emption on page 8-12

### Locate ISR Body in Vector Table

If you are using a vectored EIC, and you have a critical ISR of small size, you might achieve a performance improvement by positioning the ISR code directly in the vector table. In this way, you eliminate the overhead of branching from the vector table through the HAL funnel to your ISR.
The EIC’s driver provides a default vector table entry size. For example, with the Intel FPGA VIC, the default size is 16 bytes. To accommodate your ISR, adjust the entry size with a driver setting when you create the BSP.

**Note:** Positioning an ISR in a vector table is an advanced and error-prone technique, not directly supported by the HAL. You must exercise great caution to ensure that the ISR code fits in the vector table entry. If your ISR overflows the vector table entry, it corrupts other entries in the vector table, and your entire interrupt handling system. When your ISR is located in the vector table, it does not need to be registered. Do not call `alt_ic_isr_register()`, because it overwrites the contents of the vector table. The HAL does not provide funnel code. Therefore, your code must manage all funnel code functions.

For more information about locating an ISR in a vector table, refer to *AN595: Vectored Interrupt Controller Usage and Applications*.

**Related Information**

*AN595: Vectored Interrupt Controller Usage and Applications*

**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.

For more information about compiler optimizations, refer to “Reducing Code Footprint in Embedded Systems” in the “Developing Programs Using the Hardware Abstraction Layer chapter”.

**Related Information**

*Developing Programs Using the Hardware Abstraction Layer* on page 6-1

**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 Quartus II design.

In some cases, these changes also require changes in the software architecture or implementation.

For more information about these and other software optimizations, refer to the “Software Performance Improvements” chapter.

The following sections describe changes you can make in the hardware design to improve ISR performance.

**Related Information**

*Software Performance Improvements* on page 8-17

**Use Vectored Hardware Interrupts**

By default, the processor has a nonvectored IIC. The HAL provides software to dispatch each hardware interrupt to its specific ISR. By contrast, vectoring allows the processor to transfer control directly to the ISR with minimal software intervention.

The options available for hardware interrupt vectoring depend on the interrupt controller configured in the hardware, as described in this section.
Using the Interrupt Vector Custom Instruction

The processor core offers an interrupt vector custom instruction that accelerates hardware interrupt vector dispatch in the HAL. You can include this custom instruction to improve your program's interrupt response time.

When the interrupt vector custom instruction is present in the processor, the HAL source detects it at compile time and generates code using the custom instruction.

When using an interrupt vector custom instruction, you cannot use a separate exception stack.

**Note:** The interrupt vector custom instruction is only available in hardware systems generated by SOPC Builder.

For more information about the interrupt vector custom instruction, refer to “Interrupt Vector Custom Instruction” in the "Instantiating the Processor” chapter of the *Processor Reference Handbook*.

**Related Information**
- Using Tightly Coupled Memory with the Nios II Processor Tutorial
- Instantiating the Nios II Processor

Using an External Interrupt Controller

The EIC port allows you to connect a customizable external interrupt controller component. An EIC can be vectored. An example is the Intel FPGA VIC.

For more information about the VIC, refer to the "Vectored Interrupt Controller Core" chapter in the *Embedded Peripherals IP User Guide*.

**Related Information**
- Vectored Interrupt Controller Core

Add Fast Memory

Increase the amount of fast on-chip memory available for data buffers. Ideally, implement tightly-coupled memory that the software can use for buffers.

For more information about tightly-coupled memory, refer to the "Cache and Tightly-Coupled Memory" chapter.

For more information about tightly-coupled memory, refer to the "Using Tightly Coupled Memory with the Processor Tutorial.

**Related Information**
- Cache and Tightly-Coupled Memory on page 9-1
- Using Tightly Coupled Memory with the Nios II Processor Tutorial
- Instantiating the Nios II Processor

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 processor to carry out other tasks while data buffers are being transferred.

For more information about DMA controllers, refer to the "DMA Controller Core" and "Scatter-Gather DMA Controller Core” chapters in the *Embedded Peripherals IP User Guide*. 
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. For best performance, store exception handling code and data in tightly-coupled memory.

Use a Fast Core

For processing in both the exception context and the application context, the /f core is the fastest, and the /e core (designed for small size) is the slowest.

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.

Hardware Interrupt Priorities with the Internal Interrupt Controller

When selecting the IRQ for each peripheral, remember that the HAL hardware interrupt funnel treats IRQ0 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.

Hardware Interrupt Priorities with an External Interrupt Controller

With an EIC, the hardware interrupt priority level can be more flexible than with the IIC. The method of assigning priority levels to IRQs depends on the specific EIC implementation.

For example, with the Intel FPGA VIC, you can adjust hardware interrupt priority levels at runtime, with the alt_vic_irq_set_level() function.

For more information about the VIC, refer to the "Vectored Interrupt Controller Core" chapter in the Embedded Peripherals IP User Guide.

Related Information

Vectored Interrupt Controller Core

Debugging 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.

With the IIC, the ipending register (ctl4) is masked to all zeros during single-stepping. This masking prevents the processor from servicing interrupts that are asserted while you single-step through code. As a result, if you try to single-step through a part of the exception handling system that reads the ipending register, such as alt_irq_entry() or alt_irq_handler(), the code does not detect any pending interrupts. This issue does not affect debugging software exceptions. You can set breakpoints in your ISR
code (and single-step through it), because the interrupt funnel has already used ipending to determine which device caused the hardware interrupt.

**HAL Exception Handling System Implementation**

The exception handling system implementation is one of many possible implementations of an exception handling system for the processor. Some features of the HAL exception handling system are constrained by the hardware, while others provide generally useful services.

You can take advantage of the HAL exception 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 “Nios II Interrupt Service Routines” chapter.

**Related Information**

[Interrupt Service Routines](#) on page 8-6

**Exception Handling System Structure**

The exception handling system consists of the following components:

- The general exception funnel
- The software exception funnel
- The hardware interrupt funnel(s)
- An ISR for each peripheral that generates hardware interrupts

With the IIC, there is a single hardware interrupt funnel. This funnel manages processor context switch and RTOS overhead (if any). It determines the source of the IRQ, and dispatches the correct ISR.

With an EIC, hardware interrupt funnels are configured by the EIC driver. With a vectored EIC, such as the Intel FPGA VIC, there are multiple hardware interrupt funnels. Each funnel manages processor context switch if necessary, and RTOS overhead if any. ISR dispatch is managed by hardware.

With the IIC, when the processor generates an exception, the general exception funnel receives control. The general exception funnel passes control to either the hardware interrupt funnel or the software exception funnel. The hardware interrupt funnel passes control to one or more ISRs.

Each time an exception occurs, the exception handling system services either a software exception or hardware interrupts, with hardware interrupts having a higher priority. The HAL IIC support does not include nested exceptions, but can handle multiple hardware interrupts per context switch.

For more information, refer to the “Hardware Interrupt Funnel” chapter.

With an EIC, the general exception funnel handles only software exceptions. An IRQ causes the processor to transfer control to one of the interrupt funnels, which branches directly to the ISR.

**Related Information**

[Hardware Interrupt Funnel](#) on page 8-26
General Exception Funnel
The general exception funnel provided with the HAL is located at the processor's exception address. When a software exception or internal hardware interrupt occurs, and control transfers to the general exception funnel, it does the following:

- Switches to the separate exception stack (if enabled)
- Stores register values onto the stack
- Determines the type of exception, and passes control to the software exception funnel or the hardware interrupt funnel

Hardware Interrupt Dispatch with the Internal Interrupt Controller
With the IIC, the general exception funnel dispatches hardware interrupts as well as software exceptions.

The general exception funnel looks at the estatus register to determine the interrupt enable status. If the PIE bit is set, hardware interrupts were enabled at the time the exception happened. If so, the general exception funnel transfers control to the hardware interrupt funnel. The hardware interrupt funnel looks at the IRQ bits in ipending. If any IRQs are asserted, the interrupt funnel calls the appropriate hardware interrupt handler.

If hardware interrupts are not enabled at the time of the exception, it is not necessary to look at ipending. If no IRQs are active, there is no hardware interrupt, and the exception is a software exception. In this case, the general exception funnel calls the software exception funnel.

All hardware interrupts are higher priority than software exceptions.

Note: With an EIC, IRQs are dispatched by hardware. The HAL general exception funnel only handles software exceptions.

For more information about the processor estatus and ipending registers, refer to the "Programming Model" chapter of the Processor Reference Handbook.

Related Information
Programming Model

Returning from Exceptions
After returning from the ISR or software exception handler, the general exception funnel performs the following tasks:

- Restores the stack pointer, if a separate exception stack is used
- Restores the registers from the stack
- Exits by issuing an eret (exception return) instruction

Hardware Interrupt Funnel
The configuration of the HAL hardware interrupt funnel depends on the interrupt controller implemented in the processor core.
Note: This figure shows the algorithm that the HAL general exception funnel uses to distinguish between hardware interrupts and software exceptions.

Interrupt Funnel for the Internal Interrupt Controller

With the IIC, the processor supports 32 hardware interrupts. In the HAL funnel, hardware interrupt 0 has the highest priority, and 31 the lowest. This prioritization is a feature of the HAL funnel, and is not inherent in the interrupt controller.

The hardware interrupt funnel calls the user-registered ISRs. It goes through the IRQs in ipending starting at 0, and finds the first (highest priority) active IRQ. Then it calls the corresponding registered ISR. After this ISR executes, the funnel begins scanning the IRQs again, starting at IRQ0. In this way, higher-priority interrupts are always processed before lower-priority interrupts. When all IRQs are clear, the hardware interrupt funnel returns to the top level.

When the interrupt vector custom instruction is present in the processor, the HAL source detects it at compile time and generates code using the custom instruction.

For more information, refer to the “Using the Interrupt Vector Custom Instruction” chapter.

Related Information
Using the Interrupt Vector Custom Instruction on page 8-23
Interrupt Funnels for External Interrupt Controllers

With the EIC interface, the processor supports a potentially unlimited number of hardware interrupts on daisychained EICs. The interrupt priority level can be software-configurable. Details of setting interrupt priorities depend on the particular EIC implementation. The hardware ensures that the highest-priority interrupt is always serviced first.

You register ISRs at system initialization time. Interrupt dispatch is handled by hardware.

For more information, refer to the “Exception Handling System Structure” chapter.

Related Information
Exception Handling System Structure on page 8-25

Interrupt Funnels for Internal Interrupt Controllers

HAL Hardware Interrupt Funnel for the Internal Interrupt Controller

The HAL provides the following interrupt funnels:

- Shadow register set, pre-emption disabled
- Shadow register set, pre-emption enabled
- Nonmaskable interrupt

For more information, refer to the “Using Interrupt Funnels” chapter.

Related Information
Using Interrupt Funnels on page 8-11

Software Exception Funnel

Software exceptions can include unimplemented instructions, traps, and miscellaneous exceptions.
Software exception handling depends on options selected in the BSP. If you have enabled unimplemented instruction emulation, the software exception funnel first checks whether an unimplemented instruction caused the exception. If so, it emulates the instruction. Otherwise, it handles traps and miscellaneous exceptions.

**Unimplemented Instructions**

You can include a handler to emulate unimplemented instructions. The processor architecture defines the following implementation-dependent instructions:

- `mul`
- `muli`
- `mulxss`
- `mulxsu`
- `mulxuu`
- `div`
- `divu`

For more information about unimplemented instructions, refer to “Unimplemented Instructions” in the "Processor Architecture" chapter of the *Processor Reference Handbook*.

For more information about how unimplemented instructions are different from invalid instructions, refer to the “Invalid Instructions” chapter.

**Related Information**

- **Invalid Instructions** on page 8-32
- **Processor Architecture**

**When to Use the Unimplemented Instruction Handler**

You do not normally need the unimplemented instruction handler, because the HAL includes software emulation for unimplemented instructions from its run-time libraries if you are compiling for a processor that does not support the instructions.

You might need the unimplemented instruction handler under the following circumstances:

- You are running a program on an implementation of the processor other than the one you compiled for. The best solution is to build your program for the correct processor implementation. Resort to the unimplemented instruction handler only if it is not possible to determine the processor implementation at compile time.
- You have assembly language code that uses an implementation-dependent instruction.

If unimplemented instruction emulation is disabled, but the processor encounters an unimplemented instruction, the software exception funnel treats the exception as a miscellaneous exception.

For more information about miscellaneous exceptions, refer to the “Miscellaneous Exceptions” chapter. Otherwise, if instruction emulation is not enabled, this logic is omitted.

**Related Information**

**Miscellaneous Exceptions** on page 8-31

**Using the Unimplemented Instruction Handler**

To include the unimplemented instruction handler, turn on the `hal.enable_mul_div_emulation` BSP property. The emulation routines occupy less than \( \frac{3}{4} \) kilobyte(KB) of memory.
Note: An exception handler must never execute an unimplemented instruction. The HAL exception handling system does not support nested software exceptions.

Figure 8-2: HAL Software Exception Funnel Including the Optional Instruction Emulation Logic

Instruction-Related Exceptions

If the cause of the software exception is not an unimplemented instruction, the HAL software exception funnel checks for a registered instruction-related exception handler. If no instruction-related exception handler is registered, the exception is handled.

For more information, refer to the “Software Trap Handling” chapter. If a handler is registered, the HAL software exception funnel calls it, then restores context and returns.
For more information, refer to the “The Nios II Instruction-Related Exception Handler” chapter for a description of the instruction-related exception handler and how to register it.

Related Information

- The Instruction-Related Exception Handler on page 8-32
- Software Trap Handling on page 8-31

Software Trap Handling

If no instruction-related exception handler is registered, the HAL software exception funnel checks for a trap instruction. If the exception is caused by a trap instruction, the trap exception handler executes a break instruction. The break instruction transfers control to a hardware debug core, if one is available. If the exception is not caused by a trap instruction, it is treated as a miscellaneous exception.

Miscellaneous Exceptions

If the software exception is not caused by an unimplemented instruction or a trap, it is a miscellaneous exception.

If a debug core is present in the processor, traps and miscellaneous exceptions are handled identically, by executing a break instruction.

For more information about the flowchart of the HAL software exception funnel, including the optional trap logic, refer to the "HAL Software Exception Funnel Including the Optional Instruction Emulation Logic" figure (Figure 8–3).

If a debug core is present in the processor, the trap logic is omitted.

In a debugging environment, the processor executes a break, allowing the debugger to take control. In a nondebugging environment, the processor enters an infinite loop.

For more information about the processor break instruction, refer to the "Programming Model" chapter.

For more information about the processor break instruction, refer to the "Instruction Set Reference" chapter of the Processor Reference Handbook.

Miscellaneous exceptions can occur for these reasons:

- Advanced exceptions, the memory protection unit (MPU), or the memory management unit (MMU) are implemented in the processor core.
- You need to include the unimplemented instruction handler.
- A peripheral is generating spurious hardware interrupts. This is a symptom of a serious hardware problem. A peripheral might generate spurious hardware interrupts if it deasserts its interrupt output before an ISR has explicitly serviced it.

For more information about how to handle advanced and MPU exceptions, refer to the “The Nios II Instruction-Related Exception Handler” chapter.

For more information about how you need to implement a full-featured operating system to handle MMU exceptions, refer to the "Programming Model" chapter.

For more information about the unimplemented instruction handler, refer to the “Unimplemented Instructions” chapter.

Related Information

- The Instruction-Related Exception Handler on page 8-32
- Unimplemented Instructions on page 8-29
- Instruction Set Reference
Invalid Instructions

An invalid instruction word contains invalid codes in the OP or OPX field. For normal core implementations, the result of executing an invalid instruction is undefined; processor behavior is dependent on the core.

Therefore, the software exception funnel cannot detect or respond to an invalid instruction.

For more information about how invalid instructions are different from unimplemented instructions, refer to the "Unimplemented Instructions" chapter.

For more information, refer to the "Core Implementation Details" chapter of the Processor Reference Handbook.

Related Information
- Unimplemented Instructions on page 8-29
- Nios II Core Implementation Details

The Instruction-Related Exception Handler

The software exception funnel lets you handle instruction-related exceptions, such as the advanced exceptions. The instruction-related exception handler is a custom handler. Your software registers the instruction-related exception handler with the HAL at startup time.

**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 about the instruction-related exceptions, refer to the "Programming Model" chapter of the Processor Reference Handbook.

For more information about enabling instruction-related exception handlers, refer to “Settings Managed by the Software Build Tools” in the "Software Build Tools Reference" section.

When you register an instruction-related exception handler, it takes the place of the break/optional trap logic.

When you remove the instruction-related exception handler, the HAL restores the default break/optional trap logic.

Related Information
- Programming Model
- Software Build Tools Reference on page 15-1

Writing an Instruction-Related Exception Handler

The prototype for an instruction-related exception handler is as follows:

```c
alt_exception_result handler (  
    alt_exception_cause cause,  
    alt_u32 addr,  
    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 "Exception Cause Codes" table (Table 8–4)
- **addr**—Instruction address at which exception occurred
- **bad_addr**—Bad address register (if implemented)

Include the following header file in your instruction-related exception handler code:

```c
#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 "Exception Cause Codes" table (Table 8–4).

The API function `alt_exception_cause_generated_bad_addr()` is provided by the HAL, for the use of the instruction-related exception handler. This function parses the `cause` argument and determines if `bad_addr` contains the exception-causing address.

For more information about processor exception causes, refer to “Exception Processing” in the "Programming Model" chapter of the *Processor Reference Handbook*.

**Related Information**

**Programming Model**

**Exception Cause Codes**

**Table 8-4: Exception Cause Codes**

<table>
<thead>
<tr>
<th>Exception</th>
<th>Cause Code</th>
<th>Cause Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>0</td>
<td>NIOS2_EXCEPTION_RESET</td>
</tr>
<tr>
<td>Processor-only Reset Request</td>
<td>1</td>
<td>NIOS2_EXCEPTION_CPU_ONLY_RESET_REQUEST</td>
</tr>
<tr>
<td>Hardware Interrupt</td>
<td>2</td>
<td>NIOS2_EXCEPTION_INTERRUPT</td>
</tr>
<tr>
<td>Trap Instruction</td>
<td>3</td>
<td>NIOS2_EXCEPTION_TRAP_INST</td>
</tr>
<tr>
<td>Unimplemented Instruction</td>
<td>4</td>
<td>NIOS2_EXCEPTION_UNIMPLEMENTED_INST</td>
</tr>
<tr>
<td>Illegal Instruction</td>
<td>5</td>
<td>NIOS2_EXCEPTION_ILLEGAL_INST</td>
</tr>
<tr>
<td>Misaligned Data Address</td>
<td>6</td>
<td>NIOS2_EXCEPTION_MISALIGNED_DATA_ADDR</td>
</tr>
<tr>
<td>Misaligned Destination Address</td>
<td>7</td>
<td>NIOS2_EXCEPTION_MISALIGNED_TARGET_PC</td>
</tr>
<tr>
<td>Division Error</td>
<td>8</td>
<td>NIOS2_EXCEPTION_DIVISION_ERROR</td>
</tr>
<tr>
<td>Supervisor-only Instruction Addr</td>
<td>9</td>
<td>NIOS2_EXCEPTION_SUPERVISORONLY_INST_ADDR</td>
</tr>
<tr>
<td>Supervisor-only Instruction</td>
<td>10</td>
<td>NIOS2_EXCEPTION_SUPERVISORONLY_INST</td>
</tr>
</tbody>
</table>

(8) Cause symbols are defined in *sys/alt_exceptions.h*. 
<table>
<thead>
<tr>
<th>Exception</th>
<th>Cause Code</th>
<th>Cause Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supervisor-only Data Address</td>
<td>11</td>
<td>NIOS2_EXCEPTION_SUPERVISORONLY_DATA_ADDR</td>
</tr>
<tr>
<td>Translation lookaside buffer (TLB) Miss</td>
<td>12</td>
<td>NIOS2_EXCEPTION_TLB_MISS</td>
</tr>
<tr>
<td>TLB Permission Violation (execute)</td>
<td>13</td>
<td>NIOS2_EXCEPTION_TLB_EXECUTE_PERM_VIOLATION</td>
</tr>
<tr>
<td>TLB Permission Violation (read)</td>
<td>14</td>
<td>NIOS2_EXCEPTION_TLB_READ_PERM_VIOLATION</td>
</tr>
<tr>
<td>TLB Permission Violation (write)</td>
<td>15</td>
<td>NIOS2_EXCEPTION_TLB_WRITE_PERM_VIOLATION</td>
</tr>
<tr>
<td>MPU Region Violation (instruction)</td>
<td>16</td>
<td>NIOS2_EXCEPTION_MPU_INST_REGION_VIOLATION</td>
</tr>
<tr>
<td>MPU Region Violation (data)</td>
<td>17</td>
<td>NIOS2_EXCEPTION_MPU_DATA_REGION_VIOLATION</td>
</tr>
<tr>
<td>Cause unknown (9)</td>
<td>-1</td>
<td>NIOS2_EXCEPTION_CAUSE_NOT_PRESENT</td>
</tr>
</tbody>
</table>

If there is an instruction-related exception handler, it is called at the end of the software exception funnel (if the funnel has not recognized a hardware interrupt, unimplemented instruction or trap). It takes the place of the break or infinite loop. Therefore, to support debugging, execute a break on a trap instruction.

**Note:** It is possible for an instruction-related exception to occur during execution of an ISR.

### Registering an Instruction-Related Exception Handler

The HAL API function `alt_instruction_exception_register()` registers a single instruction-related exception handler.

The function prototype is as follows:

```c
alt_instruction_exception_register (alt_exception_result (*handler) (alt_exception_cause, alt_u32, alt_u32 ));
```

The `handler` argument is a pointer to the instruction-related exception handler.

To use `alt_instruction_exception_register()`, include the following header file:

```c
#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 Software Build Tools” in the "Software Build Tools Reference" section.

---

(8) Cause symbols are defined in `sys/alt_exceptions.h`.

(9) This value is passed to the instruction-related exception handler if the cause argument if the cause is not known; for example, if the cause register not implemented in the processor core.
**Note:** Register the instruction-related exception handler as early as possible in function `main()`. This allows you to handle abnormal condition during startup. You register an exception handler from the `alt_main()` function.

For more information about `alt_main()`, refer to “Boot Sequence and Entry Point” in the "Developing Programs Using the Hardware Abstraction Layer" section.

**Related Information**
- Developing Programs Using the Hardware Abstraction Layer on page 6-1
- Software Build Tools Reference on page 15-1

**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:

```c
alt_instruction_exception_register ( null, null );
```

When the HAL removes the instruction-related exception handler, it restores the default break/optional trap logic.
Nios II embedded processor cores can contain instruction and data caches. This chapter discusses cache-related issues that you need to consider to guarantee that your program executes correctly on the processor. Fortunately, most software based on the hardware abstraction layer (HAL) works correctly without any special accommodations for caches. However, some software must manage the cache directly.

For code that needs direct control over the cache, the 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 during load and store instructions

This chapter discusses the following common cases in which you must manage the cache:

- Initializing cache after reset
- Writing device drivers
- Writing program loaders
- Managing cache in multi-master or multi-processor systems

This chapter covers cache management issues that affect programmers. It does not discuss the fundamental operation of caches. Refer to *The Cache Memory Book* by Jim Handy for a discussion of general cache management issues.

**Related Information**

*The Cache Memory Book* by Jim Handy

**Cache Implementation**

Depending on the core implementation, a processor system might or might not have data or instruction caches. You can write programs generically so that they function correctly on any processor, regardless of whether it has cache memory. For a core without one or both caches, cache management operations are benign and have no effect.

The current cores have no hardware cache coherency mechanism. Therefore, if multiple masters can access shared memory, software must explicitly maintain coherency across all masters.

For more information about the features of each core implementation, refer to the "Core Implementation Details” chapter of the Processor Reference Handbook.
Defining Cache Properties
The details for a particular processor system are defined in the system.h file.

Example 9–1. An Excerpt from system.h that Defines the Cache Structure

```
#define NIOS2_ICACHE_SIZE 4096
#define NIOS2_DCACHE_SIZE 0
#define NIOS2_ICACHE_LINE_SIZE 32
#define NIOS2_DCACHE_LINE_SIZE 0
```

This system has a 4 KB instruction cache with 32 byte lines, and no data cache.

HAL API Functions for Managing Cache
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()`
- `alt_uncached_malloc()`
- `alt_uncached_free()`
- `alt_remap_uncached()`
- `alt_remap_cached()`

For more information about API functions, refer to the "HAL API Reference" section.

Related Information
HAL API Reference on page 14-1

Initializing the Cache after Reset
After reset, the contents of the instruction cache and data cache are unknown. They must be initialized at the start of the software reset handler for correct operation.

The caches cannot be disabled by software; they are always enabled. To allow proper operation, a processor reset causes the instruction cache to invalidate the one instruction cache line that corresponds to the reset handler address. This forces the instruction cache to fetch instructions corresponding to this cache line from memory. The reset handler address must be aligned to the size of the instruction cache line.

It is the responsibility of the first eight instructions of the reset handler to initialize the remainder of the instruction cache. The `initi` instruction initializes a single instruction cache line. Do not use the `flushi` instruction because it might cause undesired effects when used to initialize the instruction cache in future implementations.
Assembly Code to Initialize the Instruction Cache

Place the \texttt{initi} instruction in a loop that executes \texttt{initi} for each instruction cache line address.

Example 9–2. Assembly Code to Initialize the Instruction Cache

\begin{verbatim}
mov r4, r0
movhi r5, %hi(NIOS2_ICACHE_SIZE)
ori r5, r5, %lo(NIOS2_ICACHE_SIZE)
icache_init_loop:
    initi r4
    addi r4, r4, NIOS2_ICACHE_LINE_SIZE
    bltu r4, r5, icache_init_loop
\end{verbatim}

After the instruction cache is initialized, the data cache must also be initialized. The \texttt{initd} instruction initializes a single data cache line. Do not use the \texttt{flushd} instruction for this purpose, because it writes dirty lines back to memory. The data cache is undefined after reset, including the cache line tags. Using \texttt{flushd} can cause unexpected writes of random data to random addresses. The \texttt{initd} instruction does not write back dirty data.

Assembly Code to Initialize the Data Cache

Example 9–3. Assembly Code to Initialize the Data Cache

\begin{verbatim}
mov r4, r0
movhi r5, %hi(NIOS2_DCACHE_SIZE)
ori r5, r5, %lo(NIOS2_DCACHE_SIZE)
dcache_init_loop:
    initd 0(r4)
    addi r4, r4, NIOS2_DCACHE_LINE_SIZE
    bltu r4, r5, dcache_init_loop
\end{verbatim}

\textbf{Note:} Place the \texttt{initd} instruction in a loop that executes \texttt{initd} for each data cache line address.

It is legal to execute instruction and data cache initialization code on cores that do not implement one or both of the caches. The \texttt{initi} and \texttt{initd} instructions are simply treated as \texttt{nop} instructions if there is no cache of the corresponding type present.

HAL Behavior for Initializing the Cache after Reset

Programs based on the HAL need not manage the initialization of cache memory. The HAL C run-time code (\texttt{crt0.S}) provides a default reset handler that performs cache initialization before \texttt{alt_main()} or \texttt{main()} is called.

Device Driver Cache Considerations

Device drivers typically access control registers associated with their device. These registers are mapped into the address space. When accessing device registers, the data cache must be bypassed to ensure that accesses are not lost or deferred due to the data cache.

When writing a device driver, bypass the data cache with the \texttt{ldio/stio} family of instructions. On cores without a data cache, these instructions behave just like their corresponding \texttt{ld/st} instructions, and therefore are benign.

\textbf{Note:} Declaring a C pointer \texttt{volatile} does not make pointer accesses bypass the data cache. The \texttt{volatile} keyword merely prevents the compiler from optimizing out accesses using the pointer. This \texttt{volatile} behavior is different from the methodology for the first-generation Nios processor.
HAL Behavior for Device Driver Cache Considerations

The HAL provides the C-language macros `IORD` and `IOWR` that expand to the appropriate assembly instructions to bypass the data cache. The `IORD` macro expands to the `ldwio` instruction, and the `IOWR` macro expands to the `stwio` instruction. These macros are provided to enable HAL device drivers to access device registers.

All of these macros bypass the data cache when they perform their operation. In general, your program passes values defined in `system.h` as the `BASE` and `REGNUM` parameters. These macros are defined in the file `<EDS install path>/components/altera_nios2/HAL/inc/io.h`.

### Table 9-1: HAL I/O Macros to Bypass the Data Cache

<table>
<thead>
<tr>
<th>Macro</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>IORD(BASE, REGNUM)</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>IOWR(BASE, REGNUM, DATA)</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>IORD_32DIRECT(BASE, OFFSET)</code></td>
<td>Make a 32-bit read access at the location with address BASE+OFFSET.</td>
</tr>
<tr>
<td><code>IORD_16DIRECT(BASE, OFFSET)</code></td>
<td>Make a 16-bit read access at the location with address BASE+OFFSET.</td>
</tr>
<tr>
<td><code>IORD_8DIRECT(BASE, OFFSET)</code></td>
<td>Make an 8-bit read access at the location with address BASE+OFFSET.</td>
</tr>
<tr>
<td><code>IOWR_32DIRECT(BASE, OFFSET, DATA)</code></td>
<td>Make a 32-bit write access to write the value DATA at the location with address BASE+OFFSET.</td>
</tr>
<tr>
<td><code>IOWR_16DIRECT(BASE, OFFSET, DATA)</code></td>
<td>Make a 16-bit write access to write the value DATA at the location with address BASE+OFFSET.</td>
</tr>
<tr>
<td><code>IOWR_8DIRECT(BASE, OFFSET, DATA)</code></td>
<td>Make an 8-bit write access to write the value DATA at the location with address BASE+OFFSET.</td>
</tr>
</tbody>
</table>

Cache Considerations for Writing Program Loaders

Software that writes instructions to memory, such as program loaders, needs to ensure that old instructions are flushed from the instruction cache and processor pipeline. This flushing is accomplished with the `flushi` and `flushp` instructions, respectively. Additionally, if new instruction(s) are written to memory using store instructions that do not bypass the data cache, you must use the `flushd` instruction to flush the new instruction(s) from the data cache to memory.

Example 9–4. Assembly Code That Writes a New Instruction to Memory

```assembly
/*
 * Assume new instruction in r4 and
 * instruction address already in r5.
```


```c
/*
stw r4, 0(r5)
flushd 0(r5)
flushi r5
flushp
```

**Note:** Notice that this example uses the stw/flushd pair instead of the stwio instruction. The stwio instruction does not flush the data cache, and therefore might leave stale data in the data cache.

The `stw` instruction writes the new instruction in `r4` to the instruction address specified by `r5`. If a data cache is present, the instruction is written just to the data cache and the associated line is marked dirty. The `flushd` instruction writes the data cache line associated with the address in `r5` to memory and invalidates the corresponding data cache line. The `flushi` instruction invalidates the instruction cache line associated with the address in `r5`. Finally, the `flushp` instruction ensures that the processor pipeline has not prefetched the old instruction at the address specified by `r5`.

This code sequence is correct for all implementations. If a core does not have a particular kind of cache, the corresponding flush instruction (`flushd` or `flushi`) is executed as a `nop`.

**HAL Behavior for Cache Considerations for Writing Program Loaders**

The HAL API does not provide functions for this cache management case.

**Managing Cache in Multi-Master and Multi-Processor Systems**

The architecture 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 do 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.

Uncached data and cached data can no longer be allocated on the same line in the data cache because the core does not update the cache in an uncached line. This is the behavior for Classic. If you have existing code and use a `/f` with a data cache, then you have to check your software to ensure that it does not mix cacheable and uncacheable data on the same cache line.

provides two options for data cache bypass:

- Bit-31 cache bypass is set by default for compatibility
- If 32-bit addressing is selected, then any code/drivers that rely on bit-31 cache bypass needs modification to use cache bypass macros/instructions or the peripheral memory region.

**Cache Implementation**

When using the core, the peripheral region is introduced, where there is a 32-bit address option. With , for an uncached write, where bit 31 is set or in the peripheral memory region, the cache is bypassed. This behavior is the industry standard.

**Bit-31 Cache Bypass**

The `ldio/stio` family of instructions explicitly bypass the data cache. Bit-31 provides an alternate method to bypass the data cache. Using the bit-31 cache bypass, the normal `ld/st` family of instructions can be
used to bypass the data cache if the most significant bit of the address (bit 31) is set to one. The value of bit 31 is only used internally to the processor; bit 31 is forced to zero in the actual address accessed. This limits the maximum byte address space to 31 bits.

Using bit 31 to bypass the data cache is a convenient mechanism for software because the cacheability of the associated address is contained in the address. This usage allows the address to be passed to code that uses the normal `ld/st` family of instructions, while still guaranteeing that all accesses to that address consistently bypass the data cache.

Bit-31 cache bypass is only provided in the `/f` core, and must not be used with other cores. The other cores limit their maximum byte address space to 31 bits to ease migration of code from one implementation to another. They effectively ignore the value of bit 31, which allows code written for a `/f` core using bit 31 cache bypass to run correctly on other current implementations. In general, this feature depends on the core implementation.

For more information, refer to the "Core Implementation Details" chapter of the Processor Reference Handbook.

Related Information

Nios II Core Implementation Details

HAL Behavior for Managing Cache in Multi-Master and Multi-Processor Systems

The HAL provides the C-language `IORD_*DIRECT` macros that expand to the `ldio` family of instructions and the `IOWR_*DIRECT` macros that expand to the `stio` family of instructions.

For more information, refer to the "HAL I/O Macros to Bypass the Data Cache" table (Table 9–1 on page 9–4).

These macros are provided to access noncacheable memory regions.

The HAL provides the `alt_uncached_malloc()`, `alt_uncached_free()`, `alt_remap_uncached()`, and `alt_remap_cached()` routines to allocate and manipulate regions of uncached memory. These routines are available on cores with or without a data cache—code written for a core with a data cache is completely compatible with a core without a data cache.

The `alt_uncached_malloc()` routine guarantees that the allocated memory region is not in the data cache and that all subsequent accesses to the allocated memory regions bypass the data cache. In the case there is no data cache implemented the `alt_uncached_malloc()` routine simply calls `malloc()`. If data cache is implemented, but bit 31 is not set to be used as a cache bypass then the following link error results when building the code:

```
ALT_LINK_ERROR("alt_uncached_malloc() is not available because CPU is not configured to use bit 31 of address to bypass data cache");
```

The `alt_remap_uncached()` routine is not available with cores with data caches because mixing cacheable and uncacheable data on the same line is not supported. This function results in a link error when used with cores.

In the case there is no data cache implemented the `alt_uncached_free()` routine simply calls `free()`. If data cache is implemented, but bit 31 is not set to be used as a cache bypass then a link error results when building the code.

Tightly-Coupled Memory

If you want the performance of cache 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.

For more information, refer to “Memory Usage” in the "Developing Programs Using the Hardware Abstraction Layer" section.

Related Information

Developing Programs Using the Hardware Abstraction Layer on page 6-1
Overview of the MicroC/OS-II RTOS

MicroC/OS-II is a popular 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 processor.

MicroC/OS-II provides the following services:

- Tasks (threads)
- Event flags
- Message passing
- Memory management
- Semaphores
- Time management

The MicroC/OS-II kernel operates on top of the hardware abstraction layer (HAL) board support package (BSP) for the processor. Because of this architecture, MicroC/OS-II development for the processor has the following advantages:

- Programs are portable to other hardware systems.
- Programs are resistant to changes in the underlying hardware.
- Programs can access all HAL services, calling the UNIX-like HAL application program interface (API).
- ISRs are easy to implement.

Note: MicroC/OS-II is not compatible with external interrupt controllers on the External Interrupt Controller (EIC) interface; and can only run on systems using the internal interrupt controller.

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 website

Licensing

Intel distributes MicroC/OS-II in the Embedded Design Suite (EDS) for evaluation purposes only.
If you plan on using MicroC/OS-II in a commercial product, you must obtain a license from the Micrium Licensing Website.

Note: Micrium offers free licensing for universities and students. Contact Micrium for details.

Related Information
Micrium Licensing Website

Other RTOS Providers

Intel FPGA distributes MicroC/OS-II to provide you with immediate access to an easy-to-use RTOS. In addition to MicroC/OS-II, many other RTOSs are available from third-party vendors.

Related Information
Embedded Software
For more information, refer to the complete list of RTOSs that support the processor.

The Nios II Implementation of MicroC/OS-II

Intel FPGA has ported MicroC/OS-II to the Nios II processor. Intel FPGA distributes MicroC/OS-II in the Nios II EDS, and supports the Nios II implementation of the MicroC/OS-II kernel. Ready-made, working examples of MicroC/OS-II programs are installed with the Nios II EDS. In addition, Intel FPGA development boards are preprogrammed with a web server reference design based on MicroC/OS-II and the NicheStack TCP/IP Stack - Nios II Edition.

The Intel FPGA implementation of MicroC/OS-II is designed to be easy to use. Using the Nios II 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 FPGA provides the Nios II processor-specific source code in case you wish to examine it. The MicroC/OS-II source code is located in the following directories:

- Processor-specific code: <Nios II EDS install path>/components/altera_nios2/UCOSII
- Processor-independent code: <Nios II EDS install path>/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 II 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.

MicroC/OS-II Architecture

The Intel FPGA implementation of MicroC/OS-II for the 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.
The multi-threaded environment affects certain HAL functions.

Related Information
HAL API Reference
For more information about the consequences of calling a particular HAL function in a multi-threaded environment.

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, these directories are added to the search paths for source and include files.

The Software Build Tools (SBT) copies the files to your BSP obj subdirectory.

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
- Developing Device Drivers for the Hardware Abstraction Layer on page 7-1
  For more information about developing device drivers.
Thread-Safe HAL Drivers

To enable a driver to be ported between the HAL and MicroC/OS-II environments, Intel FPGA 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 MicroC/OS-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 FPGA-provided device driver code, and you can use them if you need to write a device driver with similar portability.

For more information about the functionality in the MicroC/OS-II environment, refer to *MicroC/OS-II: The Real-Time Kernel*.

The path listed for the header file is relative to the `<Nios II EDS install path>/components/micrium_uc_osii/UCOSII/inc` directory.

### Table 10-1: OS-Independent Macros for Thread-Safe HAL Drivers

<table>
<thead>
<tr>
<th>Macro</th>
<th>Defined in Header</th>
<th>MicroC/OS-II Implementation</th>
<th>Single-Threaded HAL Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALT_FLAG_GRP(group)</td>
<td>os/alt_flag.h</td>
<td>Create a pointer to a flag group with the name <code>group</code>.</td>
<td>Empty statement</td>
</tr>
<tr>
<td>ALT_EXTERN_FLAG_GRP(group)</td>
<td>os/alt_flag.h</td>
<td>Create an external reference to a pointer to a flag group with name <code>group</code>.</td>
<td>Empty statement</td>
</tr>
<tr>
<td>ALT_STATIC_FLAG_GRP(group)</td>
<td>os/alt_flag.h</td>
<td>Create a static pointer to a flag group with the name <code>group</code>.</td>
<td>Empty statement</td>
</tr>
<tr>
<td>ALT_FLAG_CREATE(group, flags)</td>
<td>os/alt_flag.h</td>
<td>Call OSFlagCreate() to initialize the flag group pointer, <code>group</code>, with the flags value <code>flags</code>. Error code is the return value of the macro.</td>
<td>Return 0 (success)</td>
</tr>
<tr>
<td>ALT_FLAG_PEND(group, flags, wait_type, timeout)</td>
<td>os/alt_flag.h</td>
<td>Call OSFlagPend() with the first four input arguments set to group, flags, wait_type, and timeout respectively. Error code is the return value of the macro.</td>
<td>Return 0 (success)</td>
</tr>
<tr>
<td>ALT_FLAG_POST(group, flags, opt)</td>
<td>os/alt_flag.h</td>
<td>Call OSFlagPost() with the first three input arguments set to group, flags, and opt respectively. Error code is the return value of the macro.</td>
<td>Return 0 (success)</td>
</tr>
<tr>
<td>Macro</td>
<td>Defined in Header</td>
<td>MicroC/OS-II Implementation</td>
<td>Single-Threaded HAL Implementation</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------------------------</td>
<td>-------------------------------------</td>
</tr>
<tr>
<td>ALT_SEM(sem)</td>
<td>os/alt_sem.h</td>
<td>Create an OS_EVENT pointer with the name sem.</td>
<td>Empty statement</td>
</tr>
<tr>
<td>ALT_EXTERN_SEM(sem)</td>
<td>os/alt_sem.h</td>
<td>Create an external reference to an OS_EVENT pointer with the name sem.</td>
<td>Empty statement</td>
</tr>
<tr>
<td>ALT_STATIC_SEM(sem)</td>
<td>os/alt_sem.h</td>
<td>Create a static OS_EVENT pointer with the name sem.</td>
<td>Empty statement</td>
</tr>
<tr>
<td>ALT_SEM_CREATE(sem, value)</td>
<td>os/alt_sem.h</td>
<td>Call OSSemCreate() with the argument value to initialize the OS_EVENT pointer sem. The return value is zero on success, or negative otherwise.</td>
<td>Return 0 (success)</td>
</tr>
<tr>
<td>ALT_SEM_PEND(sem, timeout)</td>
<td>os/alt_sem.h</td>
<td>Call OSSemPend() with the first two argument set to sem and timeout respectively. The error code is the return value of the macro.</td>
<td>Return 0 (success)</td>
</tr>
<tr>
<td>ALT_SEM_POST(sem)</td>
<td>os/alt_sem.h</td>
<td>Call OSSemPost() with the input argument sem.</td>
<td>Return 0 (success)</td>
</tr>
</tbody>
</table>

The newlib ANSI C Standard Library

Programs based on 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 for the normal C library, or 90 bytes for the reduced-footprint C library.

For more information about the contents of the _reent structure, refer to the newlib documentation. On the Windows Start menu, click Programs > Intel FPGA > Documentation.

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.

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 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**

*Exception Handling* on page 8-1
For more information about writing ISRs with the HAL.

---

**Implementing MicroC/OS-II Projects for the 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 SBT for Eclipse, or on the command line.

- You do not need to 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 `system.h` file; `OS_CFG.h` simply includes `system.h`.
- MicroC/OS-II settings are identified by the prefix `ucosii`.
- The meaning of each setting is defined fully in *MicroC/OS-II: The Real-Time Kernel*.

**Related Information**

- *Getting Started with the Graphical User Interface* on page 2-1
- *Software Build Tools* on page 4-1
  For more information about how to configure MicroC/OS-II with BSP settings
- *Software Build Tools Reference* on page 15-1
  For more information, refer to a list of available MicroC/OS-II BSP settings in "Settings Managed by the Software Build Tools".
For the processor, the NicheStack TCP/IP Stack is a small footprint implementation of the TCP/IP suite. The focus of the NicheStack TCP/IP Stack implementation is to reduce resource usage while providing a full-featured TCP/IP stack. The NicheStack TCP/IP Stack is designed for use in embedded systems with small memory footprints, making it suitable for processor systems.

The NicheStack TCP/IP Stack is a software package that you can add to your board support package (BSP), available through the Software Build Tools (SBT). The NicheStack TCP/IP Stack 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 application program interface (API)

**Prerequisites for Understanding the NicheStack TCP/IP Stack**

To make the best use of information in this chapter, you should be familiar with these topics:

- Sockets
- Embedded Design Suite (EDS)
- MicroC/OS-II RTOS

For more information about the several books available on the topic of programming with sockets, refer to *Unix Network Programming* by Richard Stevens.

For more information about the several books available on the topic of programming with sockets, refer to *Internetworking with TCP/IP Volume 3* by Douglas Comer.

**Related Information**

- **Overview of Embedded Development**
  For more information about the Embedded Design Suite (EDS).
- **MicroC/OS-II Real-Time Operating System**
  For more information about MicroC/OS-II.
Introduction to the NicheStack TCP/IP Stack - Edition

Intel FPGA provides the implementation of the NicheStack TCP/IP Stack, including source code, in the EDS. The NicheStack TCP/IP Stack provides you with immediate access to a stack for Ethernet connectivity for the processor. Intel FPGA’s implementation of the NicheStack TCP/IP Stack includes an API wrapper, providing the standard, well documented socket API.

The NicheStack TCP/IP Stack uses the MicroC/OS-II RTOS multithreaded environment. Therefore, to use the NicheStack TCP/IP Stack with the Nios II EDS, you must base your C/C++ project on the MicroC/OS-II RTOS. The processor system must also contain an Ethernet interface, or media access control (MAC).

The Intel FPGA-provided NicheStack TCP/IP Stack includes driver support for the following two MACs:

- The SMSC LAN91C111 device
- The Intel FPGA Triple Speed Ethernet IP core

The Embedded Design Suite includes hardware for both MACs. The NicheStack TCP/IP Stack driver is interrupt-based, so you must ensure that interrupts for the Ethernet component are connected.

Intel FPGA’s implementation of the NicheStack TCP/IP Stack is based on the hardware abstraction layer (HAL) generic Ethernet device model. In the generic device model, you can write a new driver to support any target Ethernet MAC, and maintain the consistent HAL and sockets API to access the hardware.

Related Information

Developing Device Drivers for the Hardware Abstraction Layer on page 7-1
For more information about writing an Ethernet device driver.

The NicheStack TCP/IP Stack Files and Directories

Intel provides the source code for your reference. By default the files are installed with the EDS in the <EDS install path>/components/altera_iniche/UCOSII directory. For the sake of brevity, this chapter refers to this directory as <iniche path>. You need not edit the NicheStack TCP/IP Stack source code to use the stack in a C/C++ program.

Under <iniche path>, the original code is maintained; as much as possible; under the <iniche path> directory. This organization facilitates upgrading to more recent versions of the NicheStack TCP/IP Stack. The directory contains the original NicheStack TCP/IP Stack source code and documentation specific to the implementation of the NicheStack TCP/IP Stack, including source code supporting MicroC/OS-II.

Intel FPGA’s implementation of the NicheStack TCP/IP Stack is based on version 3.1 of the protocol stack, with wrappers around the code to integrate it with the HAL.

Related Information

Nios II Processor Documentation Page

Support and Licensing

The NicheStack TCP/IP Stack is a TCP/IP protocol stack created by InterNiche Technologies, Inc. The version provided by Intel is supplied as example code only and is supplied without product support. If you require a supported TCP/IP stack you should license a product from a third-party software vendor.
You can license a newer version of the NicheStack TCP/IP Stack and other protocol stacks directly from InterNiche Technologies, Inc.

Related Information
InterNiche Technologies, Inc.

Other TCP/IP Stack Providers for the Processor

Other third party vendors also provide Ethernet support for the processor. Notably, third party RTOS vendors often offer Ethernet modules for their particular RTOS frameworks.

Related Information
Embedded Software
For more information about products available from third party providers.

Using the NicheStack TCP/IP Stack - Edition

The primary interface to the NicheStack TCP/IP Stack is the standard sockets interface. In addition, you call the following functions to initialize the stack and drivers:

- `alt_iniche_init()`
- `netmain()`

You also use the global variable `iniche_net_ready` in the initialization process.

You must provide the following simple functions, which the HAL system code calls to obtain the MAC address and IP address:

- `get_mac_addr()`
- `get_ip_addr()`

System Requirements

To use the NicheStack TCP/IP Stack, your system must meet the following requirements:

- The system hardware must include an Ethernet interface with interrupts enabled.
- The BSP must be based on MicroC/OS-II.
- The MicroC/OS-II RTOS must be configured to have the following settings:
  - `TimeManagement / OSTimeTickHook` must be enabled.
  - Maximum Number of Tasks must be four or less.
  - The system clock timer must be set to point to an appropriate timer device.

The NicheStack TCP/IP Stack Tasks

The NicheStack TCP/IP Stack, in its standard configuration, consists of two fundamental tasks. Each of these tasks consumes a MicroC/OS-II thread resource, along with some memory for the thread's stack. In addition to the tasks your program creates, the following tasks run continuously:
The NicheStack main task, `tk_netmain()` — After initialization, this task sleeps until a new packet is available for processing. Packets are received by an interrupt service routine (ISR). When the ISR receives a packet, it places it in the receive queue, and wakes up the main task.

The NicheStack tick task, `tk_nettick()` — This task wakes up periodically to monitor for time-out conditions.

These tasks are started when the initialization process succeeds in the `netmain()` function.

**Note:** You can modify the task priority and stack sizes using `#define` statements in the configuration file `ipport.h`. You can create additional system tasks by enabling other options in the NicheStack TCP/IP Stack by editing `ipport.h`.

Related Information
`netmain()` on page 11-4

Initializing the Stack

Before you initialize the stack, start the MicroC/OS-II scheduler by calling `OSStart()` from `main()`. Perform stack initialization in a high priority task, to ensure that your code does not attempt further initialization until the RTOS is running and I/O drivers are available.

To initialize the stack, call the functions `alt_iniche_init()` and `netmain()`. Global variable `iniche_net_ready` is set true when stack initialization is complete.

**Note:** Ensure that your code does not use the sockets interface before `iniche_net_ready` is set to true.

Related Information
`iniche_net_ready` on page 11-4

For more information and example, call `alt_iniche_init()` and `netmain()` from the highest priority task, and wait for `iniche_net_ready` before allowing other tasks to run.

`alt_iniche_init()`

`alt_iniche_init()` initializes the stack for use with the MicroC/OS-II operating system.

The prototype for `alt_iniche_init()` is:

```c
void alt_iniche_init(void)
```

When used this way, `alt_iniche_init()` returns nothing and has no parameters.

`netmain()`

`netmain()` is responsible for initializing and launching the NicheStack tasks. The prototype for `netmain()` is:

```c
void netmain(void)
```

`netmain()` returns nothing and has no parameters.

`iniche_net_ready`

When the NicheStack stack has completed initialization, it sets the global variable `iniche_net_ready` to a non-zero value.

**Note:** Do not call any NicheStack API functions (other than for initialization) until `iniche_net_ready` is true.
Example 11-1: Instantiating the NicheStack TCP/IP Stack Using iniche_net_ready

```c
void SSSInitialTask(void *task_data)
{
    INT8U error_code;
    alt_iniche_init();
    netmain();

    while (!iniche_net_ready)
        TK_SLEEP(1);
    /* Now that the stack is running, perform the application
     * initial configuration steps */
    .
    .
}
```

Macro `TK_SLEEP()` is part of the NicheStack TCP/IP Stack operating system (OS) porting layer.

**get_mac_addr() and get_ip_addr()**

The NicheStack TCP/IP Stack system code calls `get_mac_addr()` and `get_ip_addr()` during the device initialization process. These functions are necessary for the system code to set the MAC and IP addresses for the network interface, which you select with the `altera_iniche.iniche_default_if` BSP setting.

Because you write these functions yourself, your system has the flexibility to store the MAC address and IP address in an arbitrary location, rather than a fixed location hard-coded in the device driver. For example, some systems might store the MAC address in flash memory, while others might have the MAC address in on-chip embedded memory.

Both functions take as parameters device structures used internally by the NicheStack TCP/IP Stack. However, you do not need to know the details of the structures. You only need to know enough to fill in the MAC and IP addresses.

**Prototype for get_mac_addr()**

The prototype for `get_mac_addr()` is:

```c
int get_mac_addr(NET net, unsigned char mac_addr[6]);
```

You must implement the `get_mac_addr()` function to assign the MAC address to the `mac_addr` argument. Leave the `net` argument untouched.

The prototype for `get_mac_addr()` is in the header file `<iniche path>/inc/alt_iniche_dev.h`. The `NET` structure is defined in the `<iniche path>/src/h/net.h` file.

For demonstration purposes only, the MAC address is stored at address `CUSTOM_MAC_ADDR` in this example. There is no error checking in this example. In a real application, if there is an error, `get_mac_addr()` must return -1.

**Example 11-2: An Implementation of get_mac_addr()**

```c
#include <alt_iniche_dev.h>
#include "includes.h"
#include "ipport.h"
#include "tcpport.h"
#include <io.h>
int get_mac_addr(NET net, unsigned char mac_addr[6])
{
```
Prototype for get_ip_addr()

You must write the function `get_ip_addr()` to assign the IP address of the protocol stack. Your program can either assign a static address, or request the DHCP to find an IP address. The function prototype for `get_ip_addr()` is:

```c
int get_ip_addr(alt_iniche_dev* p_dev,
                ip_addr*   ipaddr,
                ip_addr*   netmask,
                ip_addr*   gw,
                int*       use_dhcp);
```

get_ip_addr() sets the return parameters as follows:

```c
IP4_ADDR(&ipaddr, IPADDR0,IPADDR1,IPADDR2,IPADDR3);
IP4_ADDR(&gw, GWADDR0,GWADDR1,GWADDR2,GWADDR3);
IP4_ADDR(&netmask, MSKADDR0,MSKADDR1,MSKADDR2,MSKADDR3);
```

For the dummy variables `IP_ADDR0-3`, substitute expressions for bytes 0-3 of the IP address. For `GWADDR0-3`, substitute the bytes of the gateway address. For `MSKADDR0-3`, substitute the bytes of the network mask. For example, the following statement sets `ip_addr` to IP address 137.57.136.2:

```c
IP4_ADDR ( ip_addr, 137, 57, 136, 2 );
```

To enable DHCP, include the line:

```c
*use_dhcp = 1;
```

The NicheStack TCP/IP stack attempts to get an IP address from the server. If the server does not provide an IP address within 30 seconds, the stack times out and uses the default settings specified in the `IP4_ADDR()` function calls.

To assign a static IP address, include the lines:

```c
*use_dhcp = 0;
```

The prototype for `get_ip_addr()` is in the header file `<iniche path>/inc/alt_iniche_dev.h`

`INICHE_DEFAULT_IF`, defined in `system.h`, identifies the network interface that you defined at system generation time. You can control `INICHE_DEFAULT_IF` through the `iniche_default_if` BSP setting.

`DHCP_CLIENT`, also defined in `system.h`, specifies whether to use the DHCP client application to obtain an IP address. You can set or clear this property with the `altera_iniche.dhcp_client` setting.

Calling the Sockets Interface

After you initialize your Ethernet device, use the sockets API in the remainder of your program to access the IP stack.
To create a new task that talks to the IP stack using the sockets API, you must use the function 
TK_NEWTASK(). The TK_NEWTASK() function is part of the NicheStack TCP/IP Stack operating system (OS) 
porting layer. TK_NEWTASK() calls the MicroC/OS-II OSTaskCreate() function to create a thread, and 
performs some other actions specific to the NicheStack TCP/IP Stack.

The prototype for TK_NEWTASK() is:

```
int TK_NEWTASK(struct inet_task_info* nettask);
```

Example 11-3: An Implementation of get_ip_addr()

```
#include <alt_iniche_dev.h>
#include "includes.h"
#include "ipport.h"
#include "tcpport.h"
int get_ip_addr(alt_iniche_dev* p_dev,
    ip_addr* ipaddr,
    ip_addr* netmask,
    ip_addr* gw,
    int* use_dhcp)
{
    int ret_code = -1;
    /*
    * The name here is the device name defined in system.h
    */
    if (!strcmp(p_dev->name, "/dev/INICHE_DEFAULT_IF))
    {
        /* The following is the default IP address if DHCP
        fails, or the static IP address if DHCP_CLIENT is
        undefined. */
        IP4_ADDR(&ipaddr, 10, 1, 1, 3);
        /* Assign the Default Gateway Address */
        IP4_ADDR(&gw, 10, 1, 1, 254);
        /* Assign the Netmask */
        IP4_ADDR(&netmask, 255, 255, 255, 0);
        #ifdef DHCP_CLIENT
        *use_dhcp = 1;
        #else
        *use_dhcp = 0;
        #endif
        /* DHCP_CLIENT */
        ret_code = ERR_OK;
    }
    return ret_code;
}
```

**Note:** There is no error checking in this example. In a real application, you might need to return -1 on 
error.

The prototype is defined in `<iniche path>/src/nios2/osport.h`. You can include this header file as 
follows:

```
#include "osport.h"
```

You can find other details of the OS porting layer in the osport.c file in the NicheStack TCP/IP Stack 
component directory, `<iniche path>/src/nios2/`.

Related Information

**Using the NicheStack TCP/IP Stack - Edition Tutorial**

For more information about how to use TK_NEWTASK() in an application.
Configuring the NicheStack TCP/IP Stack in a Program

The NicheStack TCP/IP Stack has many options that you can configure using `#define` directives in the file `ipport.h`. The EDS allows you to configure certain options (that is, modify the `#defines` in `system.h`) without editing source code. The most commonly accessed options are available through a set of BSP options, identifiable by the prefix `altera_iniche`.

Some less-frequently-used options are not accessible through the BSP settings. If you need to modify these options, you must edit the `ipport.h` file manually.

You can find `ipport.h` in the `debug/system_description` directory for your BSP project.

The following sections describe the features that you can configure using the SBT. Both development flows provide a default value for each feature. In general, these values provide a good starting point, and you can later fine-tune the values to meet the needs of your system.

NicheStack TCP/IP Stack General Settings

The ARP, UDP, and IP protocols are always enabled.

**Table 11-1: Protocol Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>Enables and disables the TCP.</td>
</tr>
</tbody>
</table>

**Table 11-2: Global Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use DHCP to automatically assign IP address</td>
<td>If this option is turned on, the component uses DHCP to acquire an IP address. If this option is turned off, you must assign a static IP address.</td>
</tr>
<tr>
<td>Enable statistics</td>
<td>If this option is turned on, the stack keeps counters of packets received, errors, etc. The counters are defined in <code>mib</code> structures defined in various header files in directory <code>&lt;iniche path&gt;/31src/h</code> For details about <code>mib</code> structures, refer to the NicheStack documentation.</td>
</tr>
<tr>
<td>MAC interface</td>
<td>If the IP stack has more than one network interface, this parameter indicates which interface to use.</td>
</tr>
</tbody>
</table>

For more information about BSP settings for the NicheStack, refer to the "Software Build Tools Reference" section.

**Related Information**

**Known Limitations** on page 11-9
IP Options

Table 11-3: IP Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward IP packets</td>
<td>If there is more than one network interface, this option is turned on, and the IP stack for one interface receives packets that are not addressed to it, the stack forwards the packet out of the other interface.</td>
</tr>
<tr>
<td>Reassemble IP packet fragments</td>
<td>If this option is turned on, the NicheStack TCP/IP Stack reassembles IP packet fragments as full IP packets. Otherwise, it discards IP packet fragments. This topic is explained in <em>Unix Network Programming</em> by Richard Stevens.</td>
</tr>
</tbody>
</table>

Related Information

Known Limitations on page 11-9

TCP Options

Table 11-4: TCP Zero Copy Options Available When Enabled

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use TCP zero copy</td>
<td>This option enables the NicheStack zero copy TCP API. This option allows you to eliminate buffer-to-buffer copies when using the NicheStack TCP/IP Stack. For details, refer to the NicheStack reference manual. You must modify your application code to take advantage of the zero copy API.</td>
</tr>
</tbody>
</table>

Further Information

The tutorial provides in-depth information about the NicheStack TCP/IP Stack, and illustrates how to use it in a networking application.

Related Information

Using the NicheStack TCP/IP Stack - Edition Tutorial
For more information about the NicheStack implementation

Known Limitations

Although the NicheStack code contains features intended to support multiple network interfaces, these features are not tested in the edition. Refer to the NicheStack TCP/IP Stack reference manual and source code for information about multiple network interface support.
Intel FPGA provides a read-only zip file system for use with the hardware abstraction layer (HAL). 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 take advantage of 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 Intel FPGA read-only zip file system is provided as a software package. All source and header files for the HAL drivers are located in the directory `<Nios II EDS install path>/components/altera_ro_zipfs/HAL/`.

Using the Read-Only Zip File System in a Project

The read-only zip file system is supported by both Nios II software development flows. You need not edit any source code to include and configure the file system. To use the zip file system, you use the development tools to include it as a software package for the board support package (BSP) project.

You must specify the following four parameters to configure the file system:

- The name of the flash device where you wish to program the file system.
- The offset in the address space of this flash device.
- The name of the mount point for this file subsystem in the HAL file system. For example, if you name the mount point `/mnt/zipfs`, the following code opens a file in the zip file:

  ```c
  fopen("/mnt/zipfs/hello", "r");
  ```

  This code, called from within a HAL-based program, opens the file `hello` for reading.
- The name of the zip file you wish to use.

The next time you build your project after you make these settings, the development tools include and link 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.

Preparing the Zip File

The zip file must be uncompressed. The read-only zip file system uses the zip format only for bundling files together; it does not provide the file decompression features for which zip utilities are known.

Creating a zip file with no compression is straightforward using the WinZip GUI. Alternately, use the `-e0` option to disable compression when using either `winzip` or `pkzip` from a command line.
Programming the Zip File to Flash

For your program to access files in the zip file subsystem, you must first program the zip data to flash. As part of the project build process, the development tools create a Motorola S-record file (.flash) that includes the data for the zip file system.

You then use the Flash Programmer to program the zip file system data to flash memory on the board.

Related Information

Flash Programmer User Guide
For more information about programming flash.
This document describes how to publish hardware component information for embedded software tools. 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 generator might be the identification of memory components, so that the BSP generator can create a linker script.

Embedded Component Information Flow

Embedded Component Information Flow Diagram
Tcl Assignment Statements

A 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.

```tcl
# 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
```

When you generate a hardware system, the system integration tool, _, creates an `<sopc_builder_system>.sopcinfo` file that includes all of the assignments for your component. The embedded software tools use these assignments for further processing. The system integration tool does not require any of the information included in these assignments to build the hardware representation of the component. The tool simply passes the assignments from the `_hw.tcl` file to the `.sopcinfo`.

Related Information

Quartus Prime Standard Edition Handbook Volume 1: Design and Synthesis
For more information about the `_hw.tcl` file and using Tcl to define hardware components, refer to the

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`

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 13–2. Assignment Statement for the BAUD_RATE of uart_0 in a Hardware System

```tcl
# 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]
```
**Generated Macro in system.h**

Example 13–3. Generated Macro in system.h After Dynamic Reassignment

```c
/* Generated macro in the system.h file after dynamic reassignment */
#define UART_0_BAUD_RATE 15200
```

For more information about formatting constants, refer to the GNU web page.

**Related Information**

- **GCC, the GNU Compiler Collection**
  For more information about formatting constants, refer to the GNU webpage.
- **GNU web page**
  For more information about formatting constants

**GCC C/C++ 32-bit Processor Constants**

Table 13-1: Examples of How to Format GCC C/C++ 32-bit Processor Constants

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean (char, short, int)</td>
<td>1, 0</td>
</tr>
<tr>
<td>32-bit signed integer (int, long)</td>
<td>123, -50</td>
</tr>
<tr>
<td>32-bit unsigned integer (unsigned int, unsigned long)</td>
<td>123u, 0xef8472a0</td>
</tr>
<tr>
<td>64-bit signed integer (long long int)</td>
<td>4294967296LL, -4294967296LL</td>
</tr>
<tr>
<td>64-bit unsigned integer (unsigned long long int)</td>
<td>4294967296ULL, 0xac458701fd64ULL</td>
</tr>
<tr>
<td>32-bit floating-point (float)</td>
<td>3.14f</td>
</tr>
<tr>
<td>64-bit floating-point (double)</td>
<td>2.78, 314e-2</td>
</tr>
<tr>
<td>character (char)</td>
<td>'x'</td>
</tr>
<tr>
<td>string (const char*)</td>
<td>&quot;Hello World!&quot;</td>
</tr>
</tbody>
</table>

**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 FPGAs embedded software tools already have definitions for the data types of the configuration names listed in this section.
Configuration Data Types

Table 13-2: Configuration Data Types and Corresponding Format

<table>
<thead>
<tr>
<th>Configuration Data Type</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>1, 0</td>
</tr>
<tr>
<td>32-bit integer</td>
<td>123, -50</td>
</tr>
<tr>
<td>64-bit integer</td>
<td>4294967296, -4294967296</td>
</tr>
<tr>
<td>32-bit floating-point</td>
<td>3.14</td>
</tr>
<tr>
<td>64-bit floating-point</td>
<td>2.78, 314e-2</td>
</tr>
<tr>
<td>string</td>
<td>ABC</td>
</tr>
</tbody>
</table>

Component Configuration Information

Table 13-3: Component Configuration Information - Assign with set_module_assignment

<table>
<thead>
<tr>
<th>Configuration Name</th>
<th>Type</th>
<th>Default</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpuArchitecture</td>
<td>string</td>
<td>—</td>
<td>Processor instruction set architecture. Provide this assignment if you</td>
<td>My 8051</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>want your component to be considered a processor.</td>
<td></td>
</tr>
<tr>
<td>required-Driver</td>
<td>boolean</td>
<td>0</td>
<td>If this configuration is 1 (true), the component requires a software</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>driver. Software tools are expected to generate a warning if no driver</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>is found.</td>
<td></td>
</tr>
</tbody>
</table>
### Memory-Mapped Slave Information

**Table 13-4: Avalon Memory-Mapped Slave Information - Assign with set_interface_assignment**

<table>
<thead>
<tr>
<th>Configuration Name</th>
<th>Type</th>
<th>Default</th>
<th>Meaning</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>isMemoryDevice</td>
<td>boolean</td>
<td>0</td>
<td>The slave port provides access to a memory device.</td>
<td>Intel FPGA® On-Chip Memory Component, DDR Controller, erasable programmable configurable serial (EPCS) Controller</td>
</tr>
<tr>
<td>isPrintableDevice</td>
<td>boolean</td>
<td>0</td>
<td>The slave port provides access to a character-based device.</td>
<td>Intel FPGA UART, Intel FPGA JTAG UART, Intel FPGA LCD</td>
</tr>
<tr>
<td>isTimerDevice</td>
<td>boolean</td>
<td>0</td>
<td>The slave port provides access to a timer device.</td>
<td>Intel FPGA Timer</td>
</tr>
<tr>
<td>isEthernetMacDevice</td>
<td>boolean</td>
<td>0</td>
<td>The slave port provides access to an Ethernet media access control (MAC).</td>
<td>Intel FPGA Triple-Speed Ethernet</td>
</tr>
<tr>
<td>isNonVolatileStorage(10)</td>
<td>boolean</td>
<td>0</td>
<td>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 this memory. If this property is true, you must also set isMemoryDevice to true.</td>
<td>Common flash interface (CFI) Flash, EPCS Flash, on-chip FPGA memory configured as a ROM</td>
</tr>
</tbody>
</table>

---

(10) Some FPGA RAMs support initialization at power-up from the SRAM Object File (.sof) or programmer object file (.pof), but are not considered non-volatile because this ability might not be used.
<table>
<thead>
<tr>
<th>Configuration Name</th>
<th>Type</th>
<th>Default</th>
<th>Meaning</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>isFlash</td>
<td>boolean</td>
<td>0</td>
<td>The memory device is a flash memory device. If isFlash is true, you must also set isMemoryDevice and isNonVolatileStorage to true.</td>
<td>CFI Flash, EPCS Flash</td>
</tr>
<tr>
<td>hideDevice</td>
<td>boolean</td>
<td>0</td>
<td>Do not make this slave port visible to the embedded software tools.</td>
<td>Nios II debug slave port</td>
</tr>
<tr>
<td>affectsTransactionsOnMasters</td>
<td>string</td>
<td>empty string</td>
<td>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. Embedded software tools use this information to generate #define directives describing the address space of these master ports.</td>
<td>Intel FPGA direct memory access (DMA), Intel FPGA Scatter/Gather DMA</td>
</tr>
</tbody>
</table>
Note:

Streaming Source Information

Table 13-5: Avalon Streaming Slave Interface Source Information - Assign with set_interface_assignment

<table>
<thead>
<tr>
<th>Configuration Name</th>
<th>Type</th>
<th>Default</th>
<th>Meaning</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>isInterruptController-Sender (11)</td>
<td>boolean</td>
<td>0</td>
<td>The interface sends interrupts to an interrupt controller receiver interface.</td>
<td>Intel FPGA Vectored Interrupt Controller</td>
</tr>
<tr>
<td>transportsInterrupts-FromReceivers (12)</td>
<td>string</td>
<td>empty string</td>
<td>A list of interrupt receiver interface names delimited by spaces. Used when the interrupt controller sender interface can transport daisy-chained interrupts from one or more interrupt controller receiver ports on the same module.</td>
<td>Intel FPGA Vectored Interrupt Controller daisy-chain input</td>
</tr>
</tbody>
</table>

Streaming Sink Information

Table 13-6: Streaming Sink Information - Assign with set_interface_assignment

<table>
<thead>
<tr>
<th>Configuration Name</th>
<th>Type</th>
<th>Default</th>
<th>Meaning</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>isInterruptController-Receiver (13)</td>
<td>boolean</td>
<td>0</td>
<td>The interface receives interrupts (optionally daisy-chained) from an interrupt controller sender interface.</td>
<td>Intel FPGA Vectored Interrupt Controller, Intel FPGA</td>
</tr>
</tbody>
</table>

Memory Initialization Namespace

You use the memory initialization namespace to pass memory initialization information to embedded software tools. Use this namespace to create memory initialization files, including .flash, .hex, .dat, and .sym files. You use memory initialization files for the following tasks:

- Flash programming
- RTL simulation
- Creating initialized FPGA RAMs for Quartus Prime compilation

You only need to provide these assignments if your component is a memory device that you want to initialize.

---

(11) An interrupt sender interface is an Avalon-ST source providing interrupt information according to the external interrupt controller (EIC) protocol.

(12) An interrupt receiver interface is an Avalon-ST sink receiving interrupt information from an EIC.

(13) An interrupt controller receiver interface is an Avalon-ST sink receiving interrupt information from an EIC.
The assignment name for the memory initialization namespace is `embeddedsw.memoryInfo.<name>`. The embedded software tools already have definitions for the data types of the possible values.

Table 13-7: Memory Initialization Data Types and Corresponding Format

<table>
<thead>
<tr>
<th>Memory Initialization Data Type</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>1, 0</td>
</tr>
<tr>
<td>32-bit integer</td>
<td>123, -50</td>
</tr>
<tr>
<td>string</td>
<td>ABC</td>
</tr>
</tbody>
</table>

Note: Quotation marks are not required.

Table 13-8: Memory Initialization Information - Assign with set_module_assignment Command

<table>
<thead>
<tr>
<th>Memory Initialization Name</th>
<th>Type</th>
<th>Default</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>HAS_BYTE_LANE</td>
<td>boolean</td>
<td>0</td>
<td>Create a memory initialization file for each byte.</td>
</tr>
<tr>
<td>IS_FLASH</td>
<td>boolean</td>
<td>0</td>
<td>Component is a flash device.</td>
</tr>
<tr>
<td>IS_EPCS</td>
<td>boolean</td>
<td>0</td>
<td>If <code>IS_FLASH</code> and <code>IS_EPCS</code> are both 1, component is an EPCS flash device. If <code>IS_FLASH</code> is 1 and <code>IS_EPCS</code> is 0, the component is a CFI flash device. If <code>IS_EPCS</code> is 1, <code>IS_FLASH</code> must also be 1.</td>
</tr>
<tr>
<td>GENERATE_HEX</td>
<td>boolean</td>
<td>0</td>
<td>Create an Intel hexadecimal file (.hex).</td>
</tr>
<tr>
<td>GENERATE_DAT_SYM</td>
<td>boolean</td>
<td>0</td>
<td>Create a .dat and a .sym file.</td>
</tr>
<tr>
<td>GENERATE_FLASH</td>
<td>boolean</td>
<td>0</td>
<td>Create a Motorola S-record File (.flash).</td>
</tr>
<tr>
<td>INCLUDE_WARNING_MSG</td>
<td>string</td>
<td>empty string</td>
<td>Display a warning message when creating memory initialization files.</td>
</tr>
<tr>
<td>MEM_INIT_FILENAME</td>
<td>string</td>
<td>Module instance name</td>
<td>Name of the memory initialization file, without any file type suffix.</td>
</tr>
<tr>
<td>MEM_INIT_DATA_WIDTH</td>
<td>32-bit integer</td>
<td>none (mandatory)</td>
<td>Width of memory initialization file in bits. May be different than the slave port data width.</td>
</tr>
</tbody>
</table>
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. Each listing provides information about whether the function is thread-safe when running in a multi-threaded environment, and whether it can be called from an interrupt service routine (ISR).

This chapter only lists the functionality provided by the HAL. The complete newlib API is also available from within HAL systems. For example, newlib provides printf(), and other standard I/O functions, which are not described here.

**Note:** Each function description lists the C header file that your code must include to access the function. Because header files include other header files, the function prototype might not be defined in the listed header file. However, you must include the listed header file in order to include all definitions on which the function depends.

For more information about the newlib API, refer to the newlib documentation. On the Windows Start menu, click Programs > Intel FPGA > [version] > [version] > Documentation.

**HAL API Functions**

_Exit()

**Prototype**

void _exit (int exit_code)

**Commonly Called By**

newlib C library

**Thread-safe**

Yes.

**Available from ISR**

No.
### _rename()

#### 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.

#### Return

```
-1
```

#### Related Information

newlib Library Documentation

### _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

The `_rename()` function is provided for `newlib` compatibility.

#### Return

It always fails with return code `-1`, and with `errno` set to `ENOSYS`.

#### Related Information

newlib Library Documentation
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

--

Related Information

- alt_dcache_flush_all() on page 14-3
- alt_icache_flush() on page 14-9
- alt_icache_flush_all() on page 14-8
- alt_remap_cached() on page 14-7
- alt_remap_uncached() on page 14-7
- alt_uncached_free() on page 14-6
- alt_uncached_malloc() on page 14-5

alt_dcache_flush_all()

Prototype

void alt_dcache_flush_all (void)

Commonly Called By

C/C++ programs
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.

Related Information

- `alt_dcache_flush()` on page 14-3
- `alt_icache_flush()` on page 14-9
- `alt_icache_flush_all()` on page 14-8
- `alt_remap_cached()` on page 14-7
- `alt_remap_uncached()` on page 14-7
- `alt_uncached_free()` on page 14-6
- `alt_uncached_malloc()` on page 14-5
Include

```c
#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.

Return

--

`alt_uncached_malloc()`

Prototype

```c
volatile void* alt_uncached_malloc (size_t size)
```

Commonly Called By

- C/C++ programs
- Device drivers

Thread-safe

Yes.

Available from ISR

No.

Include

```c
#include <sys/alt_cache.h>
```

Description

The `alt_uncached_malloc()` routine guarantees that the allocated memory region is not in the data cache and that all subsequent accesses to the allocated memory regions bypass the data cache. In the case there is no data cache implemented the `alt_uncached_malloc()` routine simply calls `malloc()`. If data cache is implemented, but bit 31 is not set to be used as a cache bypass then the following link error results when building the code:

```
ALT_LINK_ERROR("alt_uncached_malloc() is not available because CPU is not configured to use bit 31 of address to bypass data cache");
```

Return

If sufficient memory cannot be allocated, this function returns null, otherwise a pointer to the allocated space is returned.

Related Information

- `alt_dcache_flush()` on page 14-3
- `alt_dcache_flush_all()` on page 14-3

HAL API Reference

Altera Corporation

Send Feedback
alt_uncached_free()

Prototype

```c
void alt_uncached_free (volatile void* ptr)
```

Commonly Called By

- C/C++ programs
- Device drivers

Thread-safe

Yes.

Available from ISR

No.

Include

```c
#include <sys/alt_cache.h>
```

Description

In the case there is no data cache implemented the `alt_uncached_free()` routine simply calls `free()`. If data cache is implemented, but bit 31 is not set to be used as a cache bypass then a link error results when building the code.

Return

--

Related Information

- `alt_dcache_flush()` on page 14-3
- `alt_dcache_flush_all()` on page 14-3
- `alt_icache_flush()` on page 14-9
- `alt_icache_flush_all()` on page 14-8
- `alt_remap_cached()` on page 14-7
- `alt_remap_uncached()` on page 14-7
- `alt_uncached_malloc()` on page 14-5
alt_remap_uncached()

Prototype

```c
volatile void* alt_remap_uncached (void* ptr,
   alt_u32 len);
```

Commonly Called By

- C/C++ programs
- Device drivers

Thread-safe

Yes.

Available from ISR

No.

Include

```c
#include <sys/alt_cache.h>
```

Description

The `alt_remap_uncached()` routine is not available with cores with data caches because mixing cacheable and uncacheable data on the same line is not supported. This function results in a link error when used with Nios II cores.

Return

The return value for this function is the remapped memory region.

Related Information

- `alt_dcache_flush()` on page 14-3
- `alt_dcache_flush_all()` on page 14-3
- `alt_icache_flush()` on page 14-9
- `alt_icache_flush_all()` on page 14-8
- `alt_remap_cached()` on page 14-7
- `alt_uncached_free()` on page 14-6
- `alt_uncached_malloc()` on page 14-5

alt_remap_cached()

Prototype

```c
void* alt_remap_cached (volatile void* ptr,
   alt_u32 len);
```
alt_remap_cached()

<table>
<thead>
<tr>
<th>Commonly Called By</th>
<th>C/C++ programs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Device drivers</td>
</tr>
</tbody>
</table>

| Thread-safe | Yes. |

| Available from ISR | No. |

| Include | <sys/alt_cache.h> |

| Description | The `alt_remap_cached()` function remaps a region of memory for cached access. The memory to map is `len` bytes, starting at address `ptr`. Processors that do not have a data cache return uncached memory. |

| Return | The return value for this function is the remapped memory region. |

| Related Information | • `alt_dcache_flush()` on page 14-3  
• `alt_dcache_flush_all()` on page 14-3  
• `alt_icache_flush()` on page 14-9  
• `alt_icache_flush_all()` on page 14-8  
• `alt_remap_uncached()` on page 14-7  
• `alt_uncached_free()` on page 14-6  
• `alt_uncached_malloc()` on page 14-5 |

alt_icache_flush_all()

| Prototype | `void alt_icache_flush_all (void)` |

<table>
<thead>
<tr>
<th>Commonly Called By</th>
<th>C/C++ programs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Device drivers</td>
</tr>
</tbody>
</table>

| Thread-safe | Yes. |

alt_icache_flush_all()
<table>
<thead>
<tr>
<th>Available from ISR</th>
<th>Yes.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Include</td>
<td><code>&lt;sys/alt_cache.h&gt;</code></td>
</tr>
<tr>
<td>Description</td>
<td>The <code>alt_icache_flush_all()</code> function invalidates the entire contents of the instruction cache. In processors without instruction caches, it has no effect.</td>
</tr>
<tr>
<td>Return</td>
<td>--</td>
</tr>
<tr>
<td>Related Information</td>
<td>• <code>alt_dcache_flush()</code> on page 14-3</td>
</tr>
<tr>
<td></td>
<td>• <code>alt_dcache_flush_all()</code> on page 14-3</td>
</tr>
<tr>
<td></td>
<td>• <code>alt_icache_flush()</code> on page 14-9</td>
</tr>
<tr>
<td></td>
<td>• <code>alt_remap_cached()</code> on page 14-7</td>
</tr>
<tr>
<td></td>
<td>• <code>alt_remap_uncached()</code> on page 14-7</td>
</tr>
<tr>
<td></td>
<td>• <code>alt_uncached_free()</code> on page 14-6</td>
</tr>
<tr>
<td></td>
<td>• <code>alt_uncached_malloc()</code> on page 14-5</td>
</tr>
</tbody>
</table>

### `alt_icache_flush()`

**Prototype**

```c
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**

```c
#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

---

Related Information

- `alt_dcache_flush()` on page 14-3
- `alt_dcache_flush_all()` on page 14-3
- `alt_icache_flush_all()` on page 14-8
- `alt_remap_cached()` on page 14-7
- `alt_remap_uncached()` on page 14-7
- `alt_uncached_free()` on page 14-6
- `alt_uncached_malloc()` on page 14-5

**alt_alarm_start()**

Prototype

```c
int alt_alarm_start
(alternate_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

```c
#include <sys/alt_alarm.h>
```

Description

The `alt_alarm_start()` function schedules an alarm callback.

For more information, refer to “Using Timer Devices” in the "Developing Programs Using the Hardware Abstraction Layer" chapter.
The HAL waits \texttt{nticks} system clock ticks before calling the \texttt{callback()} function. When the HAL calls \texttt{callback()}, it passes it the input argument \texttt{context}. The HAL does not use the \texttt{context} parameter. It only passes it as a parameter to the \texttt{callback()} function.

The \texttt{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 \texttt{alarm}. This action is done by the call to \texttt{alt_alarm_start()}. One alarm is created for each call to \texttt{alt_alarm_start()}. Multiple alarms can run simultaneously.

\textbf{Return}

The return value for \texttt{alt_alarm_start()} is zero on success, and negative otherwise. This function fails if there is no system clock available.

\textbf{Related Information}

- \textbf{Developing Programs Using the Hardware Abstraction Layer} on page 6-1
- \texttt{alt_alarm_stop()} on page 14-11
- \texttt{alt_nticks()} on page 14-47
- \texttt{alt_sysclk_init()} on page 14-65
- \texttt{alt_tick()} on page 14-48
- \texttt{alt_ticks_per_second()} on page 14-49
- \texttt{gettimeofday()} on page 14-72
- \texttt{settimeofday()} on page 14-61
- \texttt{times()} on page 14-67
- \texttt{usleep()} on page 14-70

\textbf{alt_alarm_stop()}

\textbf{Prototype}

\begin{verbatim}
void alt_alarm_stop (alt_alarm* alarm)
\end{verbatim}

\textbf{Commonly Called By}

- C/C++ programs
- Device drivers

\textbf{Thread-safe}

Yes.

\textbf{Available from ISR}

Yes.

\textbf{Include}

\begin{verbatim}
<sys/alt_alarm.h>
\end{verbatim}
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.

Related Information
- `alt_alarm_start()` on page 14-10
- `alt_nticks()` on page 14-47
- `alt_sysclk_init()` on page 14-65
- `alt_tick()` on page 14-48
- `alt_ticks_per_second()` on page 14-49
- `gettimeofday()` on page 14-72
- `settimeofday()` on page 14-61
- `times()` on page 14-67
- `usleep()` on page 14-70

**alt_dma_rxchan_depth()**

**Prototype**

```c
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**

```c
<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.
Returns the maximum number of receive requests that can be posted.

Related Information

- `alt_dma_rxchan_close()` on page 14-13
- `alt_dma_rxchan_open()` on page 14-15
- `alt_dma_rxchan_prepare()` on page 14-16
- `alt_dma_rxchan_reg()` on page 14-17
- `alt_dma_txchan_close()` on page 14-18
- `alt_dma_txchan_ioctl()` on page 14-19
- `alt_dma_txchan_open()` on page 14-20
- `alt_dma_txchan_reg()` on page 14-21
- `alt_dma_txchan_send()` on page 14-27
- `alt_dma_txchan_space()` on page 14-26

### `alt_dma_rxchan_close()`

**Prototype**

```c
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**

```c
<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 zero on success and negative otherwise.

**Related Information**

- `alt_dma_rxchan_depth()` on page 14-12
- `alt_dma_rxchan_open()` on page 14-15
alt_dev_reg()

Prototype

```c
int alt_dev_reg(alt_dev* dev)
```

Commonly Called By

Device drivers

Thread-safe

No.

Available from ISR

No.

Include

```c
<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.

Related Information

- Developing Programs Using the Hardware Abstraction Layer on page 6-1
**alt_dma_rxchan_open()**

*Prototype*

```c
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*

```c
#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 null on failure and non-null otherwise. If an error occurs, `errno` is set to `ENODEV`.

*Related Information*

- `alt_dma_rxchan_close()` on page 14-13
- `alt_dma_rxchan_depth()` on page 14-12
- `alt_dma_rxchan_prepare()` on page 14-16
- `alt_dma_rxchan_reg()` on page 14-17
- `alt_dma_txchan_close()` on page 14-18
- `alt_dma_txchan_ioctl()` on page 14-19
- `alt_dma_txchan_open()` on page 14-20
- `alt_dma_txchan_reg()` on page 14-21
- `alt_dma_txchan_send()` on page 14-27
- `alt_dma_txchan_space()` on page 14-26
alt_dma_rxchan_prepare()

Prototype

```c
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

```c
#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.

Related Information

- `alt_dma_rxchan_close()` on page 14-13
- `alt_dma_rxchan_depth()` on page 14-12
- `alt_dma_rxchan_open()` on page 14-15
- `alt_dma_rxchan_reg()` on page 14-17
- `alt_dma_rxchan_close()` on page 14-18
- `alt_dma_txchan_ioctl()` on page 14-19
- `alt_dma_txchan_close()` on page 14-20
- `alt_dma_txchan_reg()` on page 14-21
- `alt_dma_txchan_send()` on page 14-27
- `alt_dma_txchan_space()` on page 14-26
alt_dma_rxchan_reg()

Prototype

```c
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

```c
#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.

Related Information

- [Developing Programs Using the Hardware Abstraction Layer](#) on page 6-1
- [alt_dma_rxchan_close()](#) on page 14-13
- [alt_dma_rxchan_depth()](#) on page 14-12
- [alt_dma_rxchan_open()](#) on page 14-15
- [alt_dma_rxchan_prepare()](#) on page 14-16
- [alt_dma_txchan_close()](#) on page 14-18
- [alt_dma_txchan_ioctl()](#) on page 14-19
- [alt_dma_txchan_open()](#) on page 14-20
alt_dma_txchan_close()

Prototype

```c
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

```c
#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 on success and negative otherwise.

Related Information

- `alt_dma_rxchan_close()` on page 14-13
- `alt_dma_rxchan_depth()` on page 14-12
- `alt_dma_rxchan_open()` on page 14-15
- `alt_dma_rxchan_prepare()` on page 14-16
- `alt_dma_rxchan_reg()` on page 14-17
- `alt_dma_txchan_ioctl()` on page 14-19
- `alt_dma_txchan_open()` on page 14-20
- `alt_dma_txchan_reg()` on page 14-21
- `alt_dma_txchan_send()` on page 14-27
- `alt_dma_txchan_space()` on page 14-26
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

```
#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 (Table 14–1 on page 14–11) 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 unpredictable behavior could result.

Return

A negative return value indicates failure; otherwise the interpretation of the return value is request specific.

Related Information

- `alt_dma_rxchan_close()` on page 14-13
- `alt_dma_rxchan_depth()` on page 14-12
- `alt_dma_rxchan_open()` on page 14-15
- `alt_dma_rxchan_prepare()` on page 14-16
- `alt_dma_rxchan_reg()` on page 14-17
- `alt_dma_txchan_close()` on page 14-18
- `alt_dma_txchan_open()` on page 14-20
alt_dma_txchan_open()

Prototype

```c
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

```c
#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 non-null otherwise. If an error occurs, `errno` is set to `ENODEV`.

Related Information

- `alt_dma_rxchan_close()` on page 14-13
- `alt_dma_rxchan_depth()` on page 14-12
- `alt_dma_rxchan_open()` on page 14-15
- `alt_dma_rxchan_prepare()` on page 14-16
- `alt_dma_rxchan_reg()` on page 14-17
- `alt_dma_txchan_close()` on page 14-18
- `alt_dma_txchan_ioctl()` on page 14-19
- `alt_dma_txchan_reg()` on page 14-21
- `alt_dma_txchan_send()` on page 14-27
- `alt_dma_txchan_space()` on page 14-26
alt_dma_txchan_reg()

Prototype

```c
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

```c
#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.

Related Information

- [Developing Programs Using the Hardware Abstraction Layer](#) on page 6-1
- `alt_dma_rxchan_close()` on page 14-13
- `alt_dma_rxchan_depth()` on page 14-12
- `alt_dma_rxchan_open()` on page 14-15
- `alt_dma_rxchan_prepare()` on page 14-16
- `alt_dma_rxchan_reg()` on page 14-17
- `alt_dma_txchan_close()` on page 14-18
- `alt_dma_txchan_ioctl()` on page 14-19
**alt_flash_close_dev()**

**Prototype**

```c
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**

```c
#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()`, `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**

```
--
```

**Related Information**

- `alt_flash_open_dev()` on page 14-28
- `alt_get_flash_info()` on page 14-29
- `alt_read_flash()` on page 14-47
- `alt_write_flash()` on page 14-52
- `alt_write_flash_block()` on page 14-53
**alt_exception_cause_generated_bad_addr()**

**Prototype**

```c
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**

```c
#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 II 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.

**Related Information**

`alt_instruction_exception_register()` on page 14-35

**alt_erase_flash_block()**

**Prototype**

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

**Commonly Called By**

- C/C++ programs
alt_dma_rxchan_ioctl()

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.

Related Information
Developing Programs Using the Hardware Abstraction Layer on page 6-1

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 unpredictable behavior could result.

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 14-1: Generic Requests DMA Might Support

<table>
<thead>
<tr>
<th>Request</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALT_DMA_SET_MODE_8</td>
<td>Transfer data in units of 8 bits. The value of <code>arg</code> is ignored.</td>
</tr>
<tr>
<td>ALT_DMA_SET_MODE_16</td>
<td>Transfer data in units of 16 bits. The value of <code>arg</code> is ignored.</td>
</tr>
<tr>
<td>ALT_DMA_SET_MODE_32</td>
<td>Transfer data in units of 32 bits. The value of <code>arg</code> is ignored.</td>
</tr>
<tr>
<td>ALT_DMA_SET_MODE_64</td>
<td>Transfer data in units of 64 bits. The value of <code>arg</code> is ignored.</td>
</tr>
<tr>
<td>ALT_DMA_SET_MODE_128</td>
<td>Transfer data in units of 128 bits. The value of <code>arg</code> is ignored.</td>
</tr>
<tr>
<td>ALT_DMA_GET_MODE</td>
<td>Return the transfer width. The value of <code>arg</code> is ignored.</td>
</tr>
<tr>
<td>ALT_DMA_TX_ONLY_ON</td>
<td>The <code>ALT_DMA_TX_ONLY_ON</code> 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.</td>
</tr>
<tr>
<td>ALT_DMA_TX_ONLY_OFF</td>
<td>Return to the default mode, in which both the receive and transmit sides of the DMA can be under software control.</td>
</tr>
<tr>
<td>ALT_DMA_RX_ONLY_ON</td>
<td>The <code>ALT_DMA_RX_ONLY_ON</code> 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.</td>
</tr>
</tbody>
</table>
### Request

<table>
<thead>
<tr>
<th>Request</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALT_DMA_RXONLY_OFF</td>
<td>Return to the default mode, in which both the receive and transmit sides of the DMA can be under software control.</td>
</tr>
</tbody>
</table>

### Related Information

**DMA Controller Core**

For more information, refer to the "DMA Controller Core" section of the *Embedded Peripherals IP User Guide*.

### alt_dma_txchan_space()

**Prototype**

```c
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**

```c
#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.

**Related Information**

- `alt_dma_rxchan_close()` on page 14-13
- `alt_dma_rxchan_depth()` on page 14-12
- `alt_dma_rxchan_open()` on page 14-15
- `alt_dma_rxchan_prepare()` on page 14-16
- `alt_dma_rxchan_reg()` on page 14-17
- `alt_dma_txchan_close()` on page 14-18
alt_dma_txchan_send()

Prototype

```c
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

```c
#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;
- `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.

Related Information

- `alt_dma_rxchan_close()` on page 14-13
- `alt_dma_rxchan_depth()` on page 14-12
- `alt_dma_rxchan_open()` on page 14-15
- `alt_dma_rxchan_prepare()` on page 14-16
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.

Related Information

- `alt_flash_close_dev()` on page 14-22
- `alt_get_flash_info()` on page 14-29
- `alt_read_flash()` on page 14-47
- `alt_write_flash()` on page 14-52
- `alt_write_flash_block()` on page 14-53
**alt_fs_reg()**

**Prototype**

```c
int alt_fs_reg (alt_dev* dev)
```

**Commonly Called By**

Device drivers

**Thread-safe**

No.

**Available from ISR**

No.

**Include**

```c
#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.

**Related Information**

- [Developing Programs Using the Hardware Abstraction Layer](#)
- `alt_dev_reg()` on page 14-14

**alt_get_flash_info()**

**Prototype**

```c
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.

**Related Information**
- `alt_flash_close_dev()` on page 14-22
- `alt_flash_open_dev()` on page 14-28
- `alt_read_flash()` on page 14-47
- `alt_write_flash()` on page 14-52
- `alt_write_flash_block()` on page 14-53

### `alt_ic_irq_disable()`

**Prototype**

```c
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.
- A driver for an external interrupt controller (EIC) must implement this function.

### 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 external interrupt controller.

### Related Information

- `alt_ic_irq_enable()` on page 14-34
- `alt_ic_irq_enabled()` on page 14-31
- `alt_ic_isr_register()` on page 14-32
- `alt_irq_cpu_enable_interrupts()` on page 14-37
- `alt_irq_disable()` on page 14-36
- `alt_irq_disable_all()` on page 14-38
- `alt_irq_enable()` on page 14-39
- `alt_irq_enable_all()` on page 14-39
- `alt_irq_enabled()` on page 14-40
- `alt_irq_init()` on page 14-41
- `alt_irq_pending()` on page 14-42
- `alt_irq_register()` on page 14-43

### `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 EIC must implement this function.

Return

Returns zero if the specified interrupt is disabled, and nonzero otherwise.

Related Information

- `alt_ic_irq_enable()` on page 14-34
- `alt_ic_isr_register()` on page 14-32
- `alt_irq_cpu_enable_interrupts()` on page 14-37
- `alt_irq_disable()` on page 14-36
- `alt_irq_disable_all()` on page 14-38
- `alt_irq_enable()` on page 14-39
- `alt_irq_enable_all()` on page 14-39
- `alt_irq_enabled()` on page 14-40
- `alt_irq_init()` on page 14-41
- `alt_irq_pending()` on page 14-42
- `alt_irq_register()` on page 14-43

`alt_ic_isr_register()`

Prototype

```c
int alt_ic_isr_register (alt_u32 ic_id,
                         alt_u32 irq,
                         alt_isr_func isr,
                         void* isr_context,
```


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:

```c
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. A driver for an EIC must implement this function.

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 external interrupt controller.

Related Information

- `alt_ic_irq_enable()` on page 14-34
- `alt_ic_irq_enabled()` on page 14-31
- `alt_irq_cpu_enable_interrupts()` on page 14-37
alt_ic_irq_enable()

Prototype

```c
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.
- A driver for an EIC must implement this function.

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 external interrupt controller.

Related Information

- `alt_ic_irq_enabled()` on page 14-31
- `alt_ic_isr_register()` on page 14-32
alt_instruction_exception_register()

Prototype

```c
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).

- For more information, refer to “Settings Managed by the Software Build Tools” in the "Software Build Tools Reference" chapter.

- 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.

For more information about usage, refer to the "Exception Handling" section.

### Return

--

### Related Information

- `alt_exception_cause_generated_bad_addr()` on page 14-23
- `alt_irq_register()` on page 14-43

## `alt_irq_disable()`

### Prototype

```c
int alt_irq_disable (alt_u32 id)
```

### Commonly Called By

- C/C++ programs
- Device drivers

### Thread-safe

Yes.

### Available from ISR

No.

### Include

```c
<priv/alt_legacy_irq.h>
```

### Description

The `alt_irq_disable()` function disables a single interrupt.

This function is part of the legacy HAL interrupt API, which is deprecated. Intel FPGA recommends using the enhanced HAL interrupt API.

For more information about using the enhanced HAL interrupt API, refer to “Interrupt Service Routines” in the "Exception Handling" chapter.

### Return

The return value is zero.

### Related Information

- **Interrupt Service Routines** on page 8-6
- `alt_ic_irq_enable()` on page 14-34
- `alt_ic_irq_enabled()` on page 14-31
alt_irq_cpu_enable_interrupts ()

Prototype

```c
void alt_irq_cpu_enable_interrupts ()
```

Commonly Called By

- C/C++ programs
- Device drivers

Thread-safe

Yes.

Available from ISR

Yes.

Include

```c
#include <sys/alt_irq.h>
```

Description

The `alt_irq_cpu_enable_interrupts ()` function enables the CPU to start taking interrupts.

Return

--

Related Information

- `alt_ic_irq_enable()` on page 14-34
- `alt_ic_irq_enabled()` on page 14-31
- `alt_ic_isr_register()` on page 14-32
- `alt_irq_disable()` on page 14-36
- `alt_irq_disable_all()` on page 14-38
- `alt_irq_enable()` on page 14-39
- `alt_irq_enable_all()` on page 14-39
- `alt_irq_enabled()` on page 14-40
alt_irq_disable_all()

Prototype

```c
alt_irq_context alt_irq_disable_all (void)
```

Commonly Called By

C/C++ programs
Device drivers

Thread-safe

Yes.

Available from ISR

No.

Include

```c
<sys/alt_irq.h>
```

Description

The `alt_irq_disable_all()` function disables all maskable interrupts. Nonmaskable interrupts (NMIs) are unaffected.

Return

Pass the return value as the input argument to a subsequent call to `alt_irq_enable_all()`.

Related Information

- `alt_ic_irq_enable()` on page 14-34
- `alt_ic_irq_enabled()` on page 14-31
- `alt_ic_isr_register()` on page 14-32
- `alt_irq_cpu_enable_interrupts()` on page 14-37
- `alt_irq_disable()` on page 14-36
- `alt_irq_enable()` on page 14-39
- `alt_irq_enable_all()` on page 14-39
- `alt_irq_enabled()` on page 14-40
- `alt_irq_init()` on page 14-41
- `alt_irq_pending()` on page 14-42
- `alt_irq_register()` on page 14-43
**alt_irq_enable()**

**Prototype**

```
int alt_irq_enable (alt_u32 id)
```

**Commonly Called By**

- C/C++ programs
- Device drivers

**Thread-safe**

Yes.

**Available from ISR**

No.

**Include**

```
<priv/alt_legacy_irq.h>
```

**Description**

The `alt_irq_enable()` function enables a single interrupt.

This function is part of the legacy HAL interrupt API, which is deprecated. Intel FPGA recommends using the enhanced HAL interrupt API.

**Return**

The return value is zero.

**Related Information**

- `alt_ic_irq_enable()` on page 14-34
- `alt_ic_irq_enabled()` on page 14-31
- `alt_ic_isr_register()` on page 14-32
- `alt_irq_cpu_enable_interruptions()` on page 14-37
- `alt_irq_disable()` on page 14-36
- `alt_irq_disable_all()` on page 14-38
- `alt_irq_enable_all()` on page 14-39
- `alt_irq_enabled()` on page 14-40
- `alt_irq_init()` on page 14-41
- `alt_irq_pending()` on page 14-42
- `alt_irq_register()` on page 14-43

**alt_irq_enable_all()**

**Prototype**

```
void alt_irq_enable_all (alt_irq_context context)
```
alt_irq_enable_all()

**Description**

The `alt_irq_enable_all()` function enables all interrupts 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_irq_disable()`.

**Return**

---

**Related Information**

- `alt_ic_irq_enable()` on page 14-34
- `alt_ic_irq_enabled()` on page 14-31
- `alt_ic_isr_register()` on page 14-32
- `alt_irq_cpu_enable_interruption()` on page 14-37
- `alt_irq_disable()` on page 14-36
- `alt_irq_disable_all()` on page 14-38
- `alt_irq_enable()` on page 14-39
- `alt_irq_enabled()` on page 14-40
- `alt_irq_init()` on page 14-41
- `alt_irq_pending()` on page 14-42
- `alt_irq_register()` on page 14-43

**alt_irq_enabled()**

**Prototype**

```c
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 maskable exceptions (status.PIE) are enabled.

For more information about using the enhanced HAL interrupt API, refer to “Interrupt Service Routines” in the "Exception Handling" chapter.

Return

Returns zero if interrupts are disabled, and non-zero otherwise.

Related Information

- alt_ic_irq_enable() on page 14-34
- alt_ic_irq_enabled() on page 14-31
- alt_ic_isr_register() on page 14-32
- alt_irq_cpu_enable_interruption() on page 14-37
- alt_irq_disable() on page 14-36
- alt_irq_disable_all() on page 14-38
- alt_irq_enable() on page 14-39
- alt_irq_enable_all() on page 14-39
- alt_irq_init() on page 14-41
- alt_irq_pending() on page 14-42
- alt_irq_register() on page 14-43

alt_irq_init()

Prototype

void alt_irq_init (void* base)

Commonly Called By

C/C++ programs
Device drivers

Thread-safe

Yes.
alt_irq_pending ()

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. It is recommended that NULL is passed in for the "base" parameter.

Return

--

Related Information

- alt_ic_irq_enable() on page 14-34
- alt_ic_irq_enabled() on page 14-31
- alt_ic_isr_register() on page 14-32
- alt_irq_cpu_enable_interrupts () on page 14-37
- alt_irq_disable() on page 14-36
- alt_irq_disable_all() on page 14-38
- alt_irq_enable() on page 14-39
- alt_irq_enable_all() on page 14-39
- alt_irq_enabled() on page 14-40
- alt_irq_pending () on page 14-42
- alt_irq_register() on page 14-43

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 internal interrupt controller.

Return

--

Related Information

- `alt_ic_irq_enable()` on page 14-34
- `alt_ic_irq_enabled()` on page 14-31
- `alt_ic_isr_register()` on page 14-32
- `alt_irq_cpu_enable_interrupts()` on page 14-37
- `alt_irq_disable()` on page 14-36
- `alt_irq_disable_all()` on page 14-38
- `alt_irq_enable()` on page 14-39
- `alt_irq_enable_all()` on page 14-39
- `alt_irq_enabled()` on page 14-40
- `alt_irq_init()` on page 14-41
- `alt_irq_register()` on page 14-43

`alt_irq_register()`

Prototype

```c
int alt_irq_register (alt_u32 id,

    void* context,
    void (*isr)(void*, alt_u32))
```

Commonly Called By

Device drivers

Thread-safe

Yes.

Available from ISR

No.

Include

<priv/alt_legacy_irq.h>
**Description**

The `alt_irq_register()` function registers an ISR. If the function is successful, the requested interrupt is enabled on return.

The input argument `id` is the interrupt to enable. `isr` is the function that is called when the interrupt is active. `context` and `id` are the two input arguments to `isr`.

Calls to `alt_irq_register()` replace previously registered handlers for interrupt `id`.

If `irq_handler` is set to null, the interrupt is disabled.

1. This function is part of the legacy HAL interrupt API, which is deprecated. Intel recommends using the enhanced HAL interrupt API.

For more information about using the enhanced HAL interrupt API, refer to “Interrupt Service Routines” in the “Exception Handling” section.

**Return**

The `alt_irq_register()` function returns zero if successful, or non-zero otherwise.

**Related Information**

- [Interrupt Service Routines](#) on page 8-6
- `alt_ic_irq_enable()` on page 14-34
- `alt_ic_irq_enabled()` on page 14-31
- `alt_ic_isr_register()` on page 14-32
- `alt_irq_cpu_enable_interruents()` on page 14-37
- `alt_irq_disable()` on page 14-36
- `alt_irq_disable_all()` on page 14-38
- `alt_irq_enable()` on page 14-39
- `alt_irq_enable_all()` on page 14-39
- `alt_irq_enabled()` on page 14-40
- `alt_irq_init()` on page 14-41
- `alt_irq_pending()` on page 14-42

**alt_llist_insert()**

**Prototype**

```c
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 locked, or in MicroC/OS-II, a `mutex` can be used.

Return

--

Related Information

`alt_llist_remove()` on page 14-45

alt_llist_remove()

Prototype

```c
void alt_llist_remove(alt_llist* entry)
```

Commonly Called By

- C/C++ programs
- Device drivers

Thread-safe

No.

Available from ISR

Yes.

Include

```c
<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

--
alt_load_section()

Prototype

```c
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

```c
#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.

It is recommended 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:

```c
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

--
alt_nticks()  

Prototype  

```c
alt_u32 alt_nticks (void)
```

Commonly Called By  

- C/C++ programs  
- Device drivers

Thread-safe  

Yes.

Available from ISR  

Yes.

Include  

```c
#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.

Related Information  

- `alt_alarm_start()` on page 14-10  
- `alt_alarm_stop()` on page 14-11  
- `alt_sysclk_init()` on page 14-65  
- `alt_tick()` on page 14-48  
- `alt_ticks_per_second()` on page 14-49  
- `gettimeofday()` on page 14-72  
- `settimeofday()` on page 14-61  
- `times()` on page 14-67  
- `usleep()` on page 14-70

alt_read_flash()  

Prototype  

```c
int alt_read_flash(alt_flash_fd* fd,
                  int offset,
                  void* dest_addr,
```
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.

The return value is zero on success and nonzero otherwise.

### Related Information
- `alt_flash_close_dev()` on page 14-22
- `alt_flash_open_dev()` on page 14-28
- `alt_get_flash_info()` on page 14-29
- `alt_write_flash()` on page 14-52
- `alt_write_flash_block()` on page 14-53

## alt_tick()

### Prototype

```c
void alt_tick (void)
```

### Commonly Called By

Device drivers

### Thread-safe

No.
**Available from ISR**

Yes.

**Include**

```c
#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**

--

**Related Information**

- `alt_alarm_start()` on page 14-10
- `alt_alarm_stop()` on page 14-11
- `alt_nticks()` on page 14-47
- `alt_sysclk_init()` on page 14-65
- `alt_ticks_per_second()` on page 14-49
- `_gettimeofday()` on page 14-72
- `settimeofday()` on page 14-61
- `times()` on page 14-67
- `usleep()` on page 14-70

**alt_ticks_per_second()**

**Prototype**

```c
alt_u32 alt_ticks_per_second (void)
```

**Commonly Called By**

- C/C++ programs
- Device drivers

**Thread-safe**

Yes.

**Available from ISR**

Yes.

**Include**

```c
#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.

**Related Information**

- `alt_alarm_start()` on page 14-10
- `alt_alarm_stop()` on page 14-11
- `alt_nticks()` on page 14-47
- `alt_sysclk_init()` on page 14-65
- `alt_tick()` on page 14-48
- `gettimeofday()` on page 14-72
- `settimeofday()` on page 14-61
- `times()` on page 14-67
- `usleep()` on page 14-70

---

**alt_timestamp()**

**Prototype**

```c
alt_u32 alt_timestamp (void)
```

**Commonly Called By**

C/C++ programs

**Thread-safe**

See description.

**Available from ISR**

See description.

**Include**

```c
#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.

Related Information

- `alt_timestamp_freq()` on page 14-51
- `alt_timestamp_start()` on page 14-51

`alt_timestamp_freq()`

Prototype

```c
alt_u32 alt_timestamp_freq (void)
```

Commonly Called By

C/C++ programs

Thread-safe

See description.

Available from ISR

See description.

Include

```c
#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.

Related Information

- `alt_timestamp()` on page 14-50
- `alt_timestamp_start()` on page 14-51

`alt_timestamp_start()`

Prototype

```c
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.

Related Information

- `alt_timestamp()` on page 14-50
- `alt_timestamp_freq()` on page 14-51

**alt_write_flash()**

Prototype

```c
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.

For more information, refer to “Using Flash Devices” in the "Developing Programs Using the Hardware Abstraction Layer" chapter.

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.

Related Information

- Developing Programs Using the Hardware Abstraction Layer on page 6-1
- alt_flash_close_dev() on page 14-22
- alt_flash_open_dev() on page 14-28
- alt_get_flash_info() on page 14-29
- alt_read_flash() on page 14-47
- alt_write_flash_block() on page 14-53

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**

```c
#include <sys/alt_flash.h>
```

**Description**
The `alt_write_flash_block()` function writes one block of data of 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.

For more information, refer to “Using Flash Devices” in the "Developing Programs Using the Hardware Abstraction Layer" chapter.

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.

**Related Information**
- Developing Programs Using the Hardware Abstraction Layer on page 6-1
- `alt_flash_close_dev()` on page 14-22
- `alt_flash_open_dev()` on page 14-28
- `alt_get_flash_info()` on page 14-29
- `alt_read_flash()` on page 14-47
- `alt_write_flash()` on page 14-52

---

### close()

**Prototype**

```c
int close (int fd)
```

**Commonly Called By**

- C/C++ programs
- Device drivers

**Thread-safe**
See 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()`.

The return value is zero on success, and –1 otherwise. If an error occurs, `errno` is set to indicate the cause.

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.

**Related Information**

- `fcntl()` on page 14-57
- `ioctl()` on page 14-73
- `isatty()` on page 14-74
- `lseek()` on page 14-64
- `open()` on page 14-66
- `read()` on page 14-68
- `stat()` on page 14-60
- `write()` on page 14-69
- newlib Library Documentation

---

**fork()**

**Prototype**

```c
pid_t fork (void)
```

**Commonly Called By**

- C/C++ programs
- Device drivers

**Thread-safe**

Yes.

**Available from ISR**

No.

**Include**

```c
<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`. 

---

Alterra Corporation

HAL API Reference

Send Feedback
### fcntl()

#### Prototype

```c
int fcntl(int fd, int cmd)
```

#### Commonly Called By

- C/C++ programs

#### Thread-safe

- No.

#### Available from ISR

- No.

#### Include

```c
<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.

#### Related Information

- `fstat()` on page 14-55
- `ioctl()` on page 14-73
- `isatty()` on page 14-74
- `lseek()` on page 14-64
- `open()` on page 14-66
- `read()` on page 14-68
- `stat()` on page 14-60
### execve()

**Prototype**

```c
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`.

**Related Information**

- [newlib Library Documentation](#)

### getpid()

**Prototype**

```c
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.

Related Information

newlib Library Documentation

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, SIGTALRM, 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, \texttt{errno} is set to indicate the cause of the error.

Related Information

\href{newlib_library_documentation}{newlib Library Documentation}

\textbf{stat()}

\textbf{Prototype}

\begin{verbatim}
int stat(const char *file_name,
struct stat *buf);
\end{verbatim}

\textbf{Commonly Called By}

- C/C++ programs
- Device drivers

\textbf{Thread-safe}

See description.

\textbf{Available from ISR}

No.

\textbf{Include}

\begin{verbatim}
<sys/stat.h>
\end{verbatim}

\textbf{Description}

The \texttt{stat()} function is similar to the \texttt{fstat()} function—it obtains status information about a file. Instead of using an open file descriptor, like \texttt{fstat()}, \texttt{stat()} takes the name of a file as an input argument.

Calls to \texttt{stat()} are thread-safe only if the implementation of \texttt{stat()} provided by the driver that is manipulated is thread-safe.

Internally, the \texttt{stat()} function is implemented as a call to \texttt{fstat()}.

\textbf{Return}

\begin{verbatim}
--
\end{verbatim}

Related Information

- \texttt{fcntl()} on page 14-57
- \texttt{fstat()} on page 14-55
- \texttt{ioctl()} on page 14-73
- \texttt{isatty()} on page 14-74
- \texttt{lseek()} on page 14-64
- \texttt{open()} on page 14-66
settimeofday()

Prototype

```c
int settimeofday (const struct timeval *t,
                 const struct timezone *tz)
```

Commonly Called By

C/C++ programs

Thread-safe

No.

Available from ISR

Yes.

Include

```c
#include <sys/time.h>
```

Description

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`.

Related Information

- `alt_alarm_start()` on page 14-10
- `alt_alarm_stop()` on page 14-11
- `alt_nticks()` on page 14-47
- `alt_sysclk_init()` on page 14-65
- `alt_tick()` on page 14-48
- `alt_ticks_per_second()` on page 14-49
- `gettimeofday()` on page 14-72
- `times()` on page 14-67
- `usleep()` on page 14-70
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.

Related Information

newlib Library Documentation

unlink()

Prototype

int unlink(char *name)

Commonly Called By

newlib C library

Thread-safe

Yes.

Available from ISR

Yes.
**Include**

```c
<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`.

**Related Information**

newlib Library Documentation

---

**sbrk()**

**Prototype**

```c
caddr_t sbrk(int incr)
```

**Commonly Called By**

newlib C library

**Thread-safe**

No.

**Available from ISR**

No.

**Include**

```c
<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 wish to dynamically allocate memory, use the newlib `malloc()` function.

**Return**

--

**Related Information**

newlib Library Documentation

---

**link()**

**Prototype**

```c
int link(const char *path1, const char *path2)
```
link()

Prototype

```c
off_t link(int fd, off_t ptr, int whence)
```

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`.

Related Information

newlib Library Documentation

lseek()

Prototype

```c
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 a 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.

**Related Information**

- fcntl() on page 14-57
- fstat() on page 14-55
- ioctl() on page 14-73
- isatty() on page 14-74
- open() on page 14-66
- read() on page 14-68
- stat() on page 14-60
- write() on page 14-69
- newlib Library Documentation

**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.

Related Information

- `alt_alarm_start()` on page 14-10
- `alt_alarm_stop()` on page 14-11
- `alt_nticks()` on page 14-47
- `alt_tick()` on page 14-48
- `alt_ticks_per_second()` on page 14-49
- `gettimeofday()` on page 14-72
- `settimeofday()` on page 14-61
- `times()` on page 14-67
- `usleep()` on page 14-70

open()

Prototype

```c
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

```c
#include <unistd.h>
#include <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 process 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.

**Related Information**

- fcntl() on page 14-57
- fstat() on page 14-55
- ioctl() on page 14-73
- isatty() on page 14-74
- lseek() on page 14-64
- read() on page 14-68
- stat() on page 14-60
- write() on page 14-69

newlib Library Documentation

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:

```c
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.

Related Information

- alt_alarm_start() on page 14-10
- alt_alarm_stop() on page 14-11
- alt_nticks() on page 14-47
- alt_sysclk_init() on page 14-65
- alt_tick() on page 14-48
- alt_ticks_per_second() on page 14-49
- gettimeofday() on page 14-72
- settimeofday() on page 14-61
- usleep() on page 14-70
- newlib Library Documentation

read()

Prototype

```c
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 a 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.

Related Information

- `fcntl()` on page 14-57
- `fstat()` on page 14-55
- `ioctl()` on page 14-73
- `isatty()` on page 14-74
- `lseek()` on page 14-64
- `open()` on page 14-66
- `stat()` on page 14-60
- `write()` on page 14-69
- newlib Library Documentation

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.

**Related Information**

- `fcntl()` on page 14-57
- `fstat()` on page 14-55
- `ioctl()` on page 14-73
- `isatty()` on page 14-74
- `lseek()` on page 14-64
- `open()` on page 14-66
- `read()` on page 14-68
- `stat()` on page 14-60
- newlib Library Documentation

**usleep()**

**Prototype**

```c
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.

Related Information
- `alt_alarm_start()` on page 14-10
- `alt_alarm_stop()` on page 14-11
- `alt_nticks()` on page 14-47
- `alt_sysclk_init()` on page 14-65
- `alt_tick()` on page 14-48
- `alt_ticks_per_second()` on page 14-49
- `gettimeofday()` on page 14-72
- `settimeofday()` on page 14-61
- `times()` on page 14-67

### 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
- -EINVAL > Invalid arguments
- -ETIME > Time out and skipping the looping after 0.7 sec
- -ENOLCK > Sectors lock failed

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`.

Related Information

- `alt_alarm_start()` on page 14-10
- `alt_alarm_stop()` on page 14-11
- `alt_nticks()` on page 14-47
- `alt_systicks_init()` on page 14-65
- `alt_tick()` on page 14-48
- `alt_ticks_per_second()` on page 14-49
- `settimeofday()` on page 14-61
- `times()` on page 14-67
- `usleep()` on page 14-70
- `newlib Library Documentation`

### ioctl()

**Prototype**

```c
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**

```c
#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.

**Related Information**

- `fcntl()` on page 14-57
- `fstat()` on page 14-55
- `isatty()` on page 14-74
- `lseek()` on page 14-64
- `open()` on page 14-66
- `read()` on page 14-68
- `stat()` on page 14-60
- `write()` on page 14-69
- newlib Library Documentation

**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.

**Related Information**

- `fcntl()` on page 14-57
- `fstat()` on page 14-55
- `ioctl()` on page 14-73
- `lseek()` on page 14-64
- `open()` on page 14-66
- `read()` on page 14-68
- `stat()` on page 14-60
- `write()` on page 14-69
- newlib Library Documentation

**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>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alt_8</td>
<td>Signed 8-bit integer.</td>
</tr>
<tr>
<td>alt_u8</td>
<td>Unsigned 8-bit integer.</td>
</tr>
<tr>
<td>alt_16</td>
<td>Signed 16-bit integer.</td>
</tr>
<tr>
<td>alt_u16</td>
<td>Unsigned 16-bit integer.</td>
</tr>
<tr>
<td>alt_32</td>
<td>Signed 32-bit integer.</td>
</tr>
<tr>
<td>alt_u32</td>
<td>Unsigned 32-bit integer.</td>
</tr>
<tr>
<td>alt_64</td>
<td>Signed 64-bit integer.</td>
</tr>
<tr>
<td>alt_u64</td>
<td>Unsigned 64-bit integer.</td>
</tr>
</tbody>
</table>

**alt_getchar()**

**Prototype**

```
alt_getchar()
```
alt_putchar()

Prototype

```c
alt_putchar(const char* str)
```

Commonly Called By

- C/C++ programs
- Device drivers

Thread-safe

See description.

Available from ISR

No.

alt_putstr()

Prototype

```c
alt_putstr(const char* str)
```

Commonly Called By

- C/C++ programs
- Device drivers

Thread-safe

See description.

Available from ISR

No.

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.

Related Information

- `alt_putchar()` on page 14-77
- `alt_putstr()` on page 14-76
- `alt_printf()` on page 14-78
- `newlibf()` on page 14-78
- `newlib Library Documentation`
Include

<sys/alt_driver.h>
<sys/alt_stdio.h>

Description

The `alt_putstr()` function sets 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.

Related Information

- `alt_putchar()` on page 14-77
- `alt_getchar()` on page 14-75
- `alt_printf()` on page 14-78
- newlib Library Documentation

alt_putchar()

Prototype

```c
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.
alt_printf()  

Prototype  

```c
alt_printf(const char* fmt, ...)
```

Commonly Called By  
C/C++ programs  
Device drivers

Thread-safe  
No.

Available from ISR  
No.

Include  
```c
#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  
--

Related Information  
- `alt_putchar()` on page 14-77  
- `alt_putchar()` on page 14-76  
- `alt_putchar()` on page 14-75

ADC HAL Device Driver  
The Intel FPGA Modular ADC IP core provides a HAL device driver. You can integrate the device driver into the HAL system library for systems.
adc_stop

Prototype
void adc_stop(int sequencer_base)

Arguments
sequencer_base: Sequencer base value.

Description
This function writes 0 to the sequencer CMD register RUN bit; and then it polls the RUN bit until it is 0.

Return

adc_start

Prototype
void adc_start(int sequencer_base)

Arguments
sequencer_base: Sequencer base value.

Description
This function sets the sequencer CMD register RUN bit.

Return

adc_set_mode_run_once

Prototype
void adc_set_mode_run_once(int sequencer_base)

Arguments
sequencer_base: Sequencer base value.

Description
This function sets the sequencer CMD register MODE bits to once.

Note: Stop the ADC before calling this function, changing ADC mode while RUN bit is set has no effect.

Return
### adc_set_mode_run_continuously

**Prototype**

```c
void adc_set_mode_run_continuously(int sequencer_base);
```

**Arguments**

- `sequencer_base`: Sequencer base value.

**Description**

This function sets the sequencer CMD register MODE bits to continuous.

**Note**: Stop the ADC before calling this function, changing ADC mode while RUN bit is set has no effect.

**Return**

---

### adc_recalibrate

**Prototype**

```c
void adc_recalibrate(int sequencer_base);
```

**Arguments**

- `sequencer_base`: Sequencer base value.

**Description**

- Backup CMD register, because some values can be overwritten.
- Stop the ADC Sequencer Core.
- Set the recalibration request bits.
- Start the ADC Sequencer Core.
- Poll for RUN bit to be clear.
- Restore CMD register

**Return**

---

### adc_interrupt_enable

**Prototype**

```c
void adc_interrupt_enable(int);
```

**Arguments**

- `sample_store_base`: Base address of sample store core.
**Description**

This function sets the IRQ enable bit in the ADC Sample Storage IRQ register.

**Return**

—

**adc_interrupt_disable**

**Prototype**

```c
void adc_interrupt_disable(int sample_store_base)
```

**Arguments**

- `sample_store_base`: Base address of sample store core.

**Description**

This function clears the IRQ enable bit in the ADC Sample Storage IRQ register.

**Return**

—

**adc_clear_interrupt_status**

**Prototype**

```c
void adc_clear_interrupt_status(int sample_store_base)
```

**Arguments**

- `sample_store_base`: Base address of sample store core.

**Description**

This function clears the W1C bits in the ADC sample Storage Status register.

**Return**

—

**adc_wait_for_interrupt - ADC Sample Storage Status Register**

**Prototype**

```c
void adc_wait_for_interrupt(int sample_store_base)
```

**Arguments**

- `sample_store_base`: Base address of sample store core.
Description
This function waits while the ADC sample storage status register is 0.

Return
—

adc_interrupt_asserted

Prototype
int adc_interrupt_asserted(int sample_store_base)

Arguments
sample_store_base: Base address of sample store core.

Description
This function reads the sample storage irq status register.

Return
-1 if 1, else returns 0

adc_wait_for interrupt - IRQ Status Register

Prototype
void adc_wait_for_interrupt(int sample_store_base)

Arguments
sample_store_base: Base address of sample store core.

Description
This function waits while the sample storage IRQ status register is 0.

Return
—

alt_adc_word_read

Prototype
int alt_adc_word_read (alt_u32 sample_store_base, alt_u32* dest_ptr, alt_u32 len)

Arguments
• sample_store_base: Base address of sample store core.
• *dest_ptr: destination buffer
• len: number of 32-bit reads
<table>
<thead>
<tr>
<th><strong>Description</strong></th>
<th>Reads words from the sample store</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Return</strong></td>
<td>_</td>
</tr>
</tbody>
</table>
This chapter provides a complete reference of all available commands, options, and settings for the Nios II Software Build Tools (SBT). This reference is useful for developing your own embedded software projects, packages, or device drivers.

Related Information
- Getting Started from the Command Line on page 3-1
  For more information about what you should read before using this chapter.
- Software Build Tools on page 4-1
  For more information about familiarizing yourself with its parts.

Software Build Tools Utilities

The build tools utilities are an entry point to the SBT. Everything you can do with the tools, such as specifying settings, creating makefiles, and building projects, is made available by the utilities.

All SBT utilities share the following behavior:
- Sends error messages and warning messages to stderr.
- Sends normal messages (other than errors and warnings) to stdout.
- Displays one error message for each error.
- Returns an exit value of 1 if it detects any errors.
- Returns an exit value of 0 if it does not detect any errors. (Warnings are not errors.)
- If the help or version command-line option is specified, returns an exit value of 0, and takes no other action. Sends the output (help or version number) to stdout.
- When an error is detected, suppresses all subsequent operations (such as writing files).

Logging Levels

All the utilities support multiple status-logging levels. You specify the logging level on the command line. At each level, the utilities report the status as listed under Description. Each level includes the messages from all lower levels.
Table 15-1: SBT Logging Levels

<table>
<thead>
<tr>
<th>Logging Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>silent (lowest)</td>
<td>No information is provided except for errors and warnings (sent to stderr).</td>
</tr>
<tr>
<td>default</td>
<td>Minimal information is provided (for example, start and stop operation of SBT phases).</td>
</tr>
<tr>
<td>verbose</td>
<td>Detailed information is provided (for example, lists of files written).</td>
</tr>
<tr>
<td>debug (highest)</td>
<td>Debug information is provided (for example, stack backtraces on errors). This level is for reporting problems.</td>
</tr>
</tbody>
</table>

Only one logging level is possible at a time, so these options are all mutually exclusive.

Table 15-2: Command-Line Options to Select Each Logging Level

<table>
<thead>
<tr>
<th>Command-Line Option</th>
<th>Logging Level</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>default</td>
<td>If there is no command-line option, the default level is selected.</td>
</tr>
<tr>
<td>--silent</td>
<td>silent</td>
<td>Selects silent level of logging.</td>
</tr>
<tr>
<td>--verbose</td>
<td>verbose</td>
<td>Selects verbose level of logging.</td>
</tr>
<tr>
<td>--debug</td>
<td>debug</td>
<td>Selects debug level of logging.</td>
</tr>
<tr>
<td>--log &lt;fname&gt;</td>
<td>debug</td>
<td>All information is written to &lt;fname&gt; in addition to being sent to the stdout and stderr devices.</td>
</tr>
</tbody>
</table>

For , there is full error correction code (ECC) support:

- Register file
- Instruction cache
- Data cache
- Tightly-Coupled Memory (TCM)

Setting Values

The value of a setting is specified with the --set command-line option to nios2-bsp-create-settings or nios2-bsp-update-settings, or with the set_setting Tcl command. The value of a setting is obtained with the --get command-line option to nios2-bsp-query-settings or with the get_setting Tcl command.

Related Information

Settings Managed by the Software Build Tools on page 15-25

Utility and Script Summary

The following command-line utilities and scripts are available:
Related Information

- nios2-app-generate-makefile on page 15-3
- nios2-bsp-create-settings on page 15-4
- nios2-bsp-generate-files on page 15-6
- nios2-bsp-query-settings on page 15-7
- nios2-bsp-update-settings on page 15-8
- nios2-lib-generate-makefile on page 15-9
- nios2-bsp-editor on page 15-10
- nios2-app-update-makefile on page 15-11
- nios2-lib-update-makefile on page 15-13
- nios2-swexample-create on page 15-15
- nios2-elf-insert on page 15-16
- nios2-elf-query on page 15-16
- nios2-bsp on page 3-2
- nios2-bsp-console on page 15-21

nios2-app-generate-makefile

Usage: nios2-app-generate-makefile --bsp-dir <directory> [ <options> ]

Description: The nios2-app-generate-makefile command generates an application makefile (called Makefile). The path to a BSP created by nios2-bsp-generate-files is a mandatory command line option.

Required Arguments:

--bsp-dir <directory>: Specifies the path to the BSP generated files directory (populated using the nios2-bsp-generate-files command).

Optional Arguments:

- --app-dir <directory>: Directory to place the application makefile and executable and linking format file (.elf). If omitted, it defaults to the current directory.
- --debug: Output debug, exception traces, verbose, and default information about the command's operation to stdout.
- --elf-name <filename>: Name of the .elf file to create. If omitted, it defaults to the first source file specified with the file name extension replaced with .elf and placed in the application directory.
- --extended-help: Displays full information about this command and its options.
- --help: Displays basic information about this command and its options.
- --inc-dir <directory>: Searches for source include files in <filepath>. Use "." to look in the current directory. Multiple --inc-dir options are allowed.
- --inc-rdir <directory>: Same as --inc-dir option but recursively search for source include files in or under <filepath>. Multiple --inc-rdir options are allowed.
- --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.
- --log <filename>: Create a debug log and write to specified file. Also logs debug information to stdout.
• --no-src: Allows no sources files to be set in the Makefile. You must add source files in manually before compiling.
• --set <name> <value>: Set the makefile variable called <name> to <value>. If the variable exists in the managed section of the makefile, <value> replaces the default settings. If the variable does not already exist, it is added. Multiple --set options are allowed.
• --silent: Suppress information about the command’s operation normally sent to stdout.
• --src-dir <directory>: Searches for source files in <directory>. Use . to look in the current directory. Multiple --src-dir options are allowed.
• --src-files <filenames>: Adds a list of space-separated source file names to the makefile. The list of file names is terminated by the next option or the end of the command line. Multiple --src-files options are allowed.
• --src-rdir <directory>: Same as --src-dir option but recursively search for source files in or under <directory>. Multiple --src-rdir options are allowed and can be freely mixed with --src-dir options.
• --use-lib-dir <directory>: Specifies the path to a dependent user library directory. The user library directory must contain a makefile fragment called public.mk. Multiple --use-lib-dir options are allowed.
• --verbose: Output verbose, and default information about the command’s operation to stdout.
• --version: Displays the version of this command and exits with a zero exit status.

Examples:
Example:

>'nios2-app-generate-makefile --bsp-dir ../my_bsp --src-files hello_world.c'

Generates a Makefile setup to compile one source file in the current directory.

>'make'

Recursively makes the BSP and then compiles and links the application to create hello_world.elf.

Example:

>'nios2-app-generate-makefile --bsp-dir ../my_bsp --app-dir /tmp/my_dir/release --elf foo --src-rdir /data/jball/src/foo --src-files extra1.c extra2.c'

Generates a Makefile in /tmp/my_dir/release that builds an ELF called /tmp/mydir/release/foo.elf. The source files are all those recursively found under /data/jball/src/foo and extra files called extra1.c and extra2.c in the current directory.

nios2-bsp-create-settings

Usage

nios2-bsp-create-settings [--bsp-dir <directory>]
[--cmd <tcl command>] [--cpu-name <cpu name>]
[--debug] [--extended-help] [--get-cpu-arch]
[--help] [--jdi <filename>]
[--librarian-factory-path <directory>]

Altera Corporation
Options

- **--bsp-dir <directory>**: Path to the directory where the BSP files are generated. Use . for the current directory. The directory <directory> must exist. This command overwrites preexisting files in <directory> without warning.
- **--cmd <tcl command>**: Runs the specified Tcl command. Multiple --cmd options are allowed.
- **--cpu-name <cpu name>**: The name of the processor that the BSP supports. Optional for a single-processor system. Use ? to list available processor names.
- **--debug**: Sends debug information, exception traces, verbose output, and default information about the command's operation, to stdout.
- **--extended-help**: Displays full information about this command and its options. Also displays Tcl command help for the --cmd and --script options.
- **--get-cpu-arch**: Queries for processor architecture from the processor specified. Does not create a BSP.
- **--help**: Displays basic information about this command and its options.
- **--jdi <filename>**: The location of the JTAG Debugging Information File (.jdi) generated by the Quartus Prime software. The .jdi file specifies the name-to-node mappings for the JTAG chain elements. The tool inserts the JTAG Debugging Information File (.jdi) path in public.mk. If no .jdi path is specified, the command searches the directory containing the SOPC Information File (.sopcinfo), and uses the first .jdi file found.
- **--librarian-factory-path<directory>**: Comma-separated librarian search path. Use $ for default factory search path.
- **--librarian-path<directory>**: Comma-separated librarian search path. Use $ for default search path.
- **--log <filename>**: Creates a debug log and write to specified file. Also logs debug information to stdout.
- **--script <filename>**: Run the specified Tcl script with optional arguments. Multiple --script options are allowed.
- **--set <name> <value>**: Sets the setting called <name> to <value>. Multiple --set options are allowed.
- **--settings <filename>**: File name of the BSP settings file to create. This file is created with a .bsp file extension. It overwrites any existing settings file.
- **--silent**: Suppresses information about the command's operation normally sent to stdout.
- **--sopc <filename>**: The .sopcinfo file used to create the BSP.
- **--type <OS name>**: BSP type. Use ? or types to list available BSP types for this option. Use names to list the display names of available BSP types. For a DPX system, always set this argument to lwhal.
- **--type-version <version>**: BSP software component version. By default the latest version is used. default value can be used to reset to this default behavior. Use ? to list available BSP types and versions.
- **--verbose**: Sends verbose output, and default information about the command's operation, to stdout.
- **--version**: Displays the version of this command and exits with a zero exit status.
**Description**

If you use `nios2-bsp-create-settings` to create a settings file without any command-line options, Tcl commands, or Tcl scripts to modify the default settings, it creates a settings file that fails when running `nios2-bsp-generate-files`. Failure occurs because the `nios2-bsp-create-settings` command is able to create reasonable defaults for most settings, but the command requires additional information for system-dependent settings. The default Tcl scripts set the required system-dependent settings. Therefore it is better to use default Tcl scripts when calling `nios2-bsp-create-settings` directly. For an example of how to use the default Tcl scripts, refer to the `nios2-bsp` script.

If no command-line arguments are specified, this command returns an exit value of 1 and sends a help message to stderr.

**Note:** For more information about this command, use the `--extended-help` option to display comprehensive usage information.

**Example**

```
nios2-bsp-create-settings --settings my_settings.bsp --sopc \\n../my_sopc.sopcinfo --type hal --script default_settings.tcl
```

**Related Information**

*Tcl Commands for BSP Settings* on page 15-64

**nios2-bsp-generate-files**

**Usage**

```
nios2-bsp-generate-files --bsp-dir <directory> 
[----debug] [--extended-help] [--help] 
[--librarian-factory-path <directory>] 
[--librarian-path <directory>] [--log <filename>] 
--settings <filename> [--silent] [--verbose] 
[--version]
```

**Options**

- `--bsp-dir <directory>`: Path to the directory where the BSP files are generated. Use . for the current directory. The directory `<directory>` must exist. This command overwrites preexisting files in `<directory>` without warning.
- `--debug`: Sends debug, exception trace, verbose, and default information about the command’s operation to stdout.
- `--extended-help`: Displays full information about this command and its options.
- `--help`: Displays basic information about this command and its options.
- `--librarian-factory-path <directory>`: Comma-separated librarian search path. Use $ for default factory search path.
- `--librarian-path <directory>`: Comma-separated librarian search path. Use $ for default search path.
- `--log <filename>`: Creates a debug log and writes to specified file. Also logs debug information to stdout.
- `--settings <filename>`: File name of an existing BSP Settings File (.bsp) to generate files from.
- **--silent**: Suppresses information about the command’s operation normally sent to `stdout`.
- **--verbose**: Sends verbose and default information about the command’s operation to `stdout`.
- **--version**: Displays the version of this command and exits with a zero exit status.

### Description

The `nios2-bsp-generate-files` command populates the files in a BSP directory. The path to an existing `.bsp` file and the path to the BSP directory are mandatory command-line options. Files are written to the specified BSP directory. Generated files are created unconditionally. Copied files are copied from the EDS installation folder only if they are not present in the BSP directory, or if the existing files differ from the installation files.

If no command-line arguments are specified, this command returns an exit value of 1 and sends a help message to `stderr`.

**Note:** For more information about this command, use the `--extended-help` option to display comprehensive usage information.

### nios2-bsp-query-settings

#### Usage

```
nios2-bsp-query-settings [--cmd <tcl command>]  
  [--debug]  [--extended-help]  [--get <name>]  
  [--get-all]  [--help]  
  [--librarian-factory-path <directory>]  
  [--librarian-path <directory>]  [--log <filename>]  
  [--script <filename>]  --settings <filename>  
  [--show-descriptions]  [--show-names]  [--silent]  
  [--verbose]  [--version]  
```

#### Options

- **--cmd <tcl command>**: Run the specified Tcl command. Multiple `--cmd` options are allowed.
- **--debug**: Output debug, exception traces, verbose, and default information about the command’s operation to `stdout`.
- **--extended-help**: Displays full information about this command and its options.
- **--get <name>**: Display the value of the setting called `<name>`. Multiple `--get` options are allowed. Each value appears on its own line in the order the `--get` options are specified. Mutually exclusive with the `--get-all` option.
- **--get-all**: Display the value of all BSP settings in order sorted by option name. Each option appears on its own line. Mutually exclusive with the `--get` option.
- **--help**: Displays basic information about this command and its options.
- **--librarian-factory-path <directory>**: Comma-separated librarian search path. Use `$` for default factory search path.
- **--librarian-path <directory>**: Comma-separated librarian search path. Use `$` for default search path.
- **--log <filename>**: Create a debug log and write to specified file. Also logs debug information to `stdout`.
- **--script <filename>**: Run the specified Tcl script with optional arguments. Multiple `--script` options are allowed.
- **--settings <filename>**: File name of an existing BSP settings file to query settings from.
- **--show-descriptions**: Displays the description of each option after the value.
- **--show-names**: Displays the name of each option before the value.
- **--silent**: Suppress information about the command's operation normally sent to stdout.
- **--verbose**: Output verbose, and default information about the command's operation to stdout.
- **--version**: Displays the version of this command and exits with a zero exit status.

**Description**

The **nios2-bsp-query-settings** command provides information from a .bsp file. The path to an existing .bsp file is a mandatory command-line option. The command does not modify the settings file. Only requested information is displayed on stdout; no informational messages are displayed.

If no command-line arguments are specified, this command returns an exit value of 1 and sends a help message to stderr.

**Note:** For more information about this command, use the **--extended-help** option to display comprehensive usage information.

**nios2-bsp-update-settings**

**Usage**

```bash
```

**Options**

- **--bsp-dir <directory>**: Path to the directory where the BSP files are generated. Use . for the current directory. The directory <directory> must exist.
- **--cmd <tcl command>**: Run the specified Tcl command. Multiple --cmd options are allowed.
- **--cpu-name <cpu name>**: The name of the Nios II processor that the BSP supports. This argument is useful if the hardware design contains multiple processors. Optional for a single-processor design.
- **--debug**: Output debug, exception traces, verbose, and default information about the command's operation to stdout.
- **--extended-help**: Displays full information about this command and its options.
- **--help**: Displays basic information about this command and its options.
- **--jdi <filename>**: The location of the .jdi file generated by the Quartus Prime software. The .jdi file specifies the name-to-node mappings for the JTAG chain elements. The tool inserts the .jdi path in public.mk. If no .jdi path is specified, the command searches the directory containing the .sopcinfo file, and uses the first .jdi file found.
- **--librarian-factory-path <directory>**: Comma-separated librarian search path. Use $ for default factory search path.
- **--librarian-path <directory>**: Comma-separated librarian search path. Use $ for default search path.
- **--log <filename>**: Create a debug log and write to specified file. Also logs debug information to stdout.
- **--script <filename>**: Run the specified Tcl script with optional arguments. Multiple --script options are allowed.
- **--set <name> <value>**: Set the setting called <name> to <value>. Multiple --set options are allowed.
- **--settings <filename>**: File name of an existing BSP settings file to update.
• **--silent**: Suppress information about the command's operation normally sent to stdout.
• **--sopc <filename>**: The .sopcinfo file to update the BSP with. It is recommended to create a new BSP if the design has changed significantly. This argument is useful if the path to the original .sopcinfo file has changed.
• **--verbose**: Output verbose, and default information about the command's operation to stdout.
• **--version**: Displays the version of this command and exits with a zero exit status.

**Description**

The `nios2-bsp-update-settings` command updates an existing Nios II .bsp file. The path to an existing .bsp file is a mandatory command-line option. The command modifies the settings file so the file must have write permissions. You might want to use the **--script** option to pass the default Tcl script to the `nios2-bsp-update-settings` command to make sure that your BSP is consistent with your system (this is what the **nios2-bsp** command does).

If no command-line arguments are specified, this command returns an exit value of 1 and sends a help message to stderr.

**Note:** For more information about this command, use the **--extended-help** option to display comprehensive usage information.

**nios2-lib-generate-makefile**

**Usage**

```
nios2-lib-generate-makefile [--bsp-dir <directory>]  
[--debug] [--extended-help] [--help]  
[--lib-dir <directory>] [--lib-name <filename>]  
[--log <filename>] [--no-src]  
[--public-inc-dir <directory>] [--set <name> <value>]  
[--silent] [--src-dir <directory>]  
[--src-files <filenames>] [--src-rdir <directory>]  
[--use-lib-dir <directory>] [--verbose]  
[--version]
```

**Options**

• **--bsp-dir <directory>**: Path to the BSP generated files directory (populated using the **nios2-bsp-generate-files** command).
• **--debug**: Output debug, exception traces, verbose, and default information about the command's operation to stdout.
• **--extended-help**: Displays full information about this command and its options.
• **--help**: Displays basic information about this command and its options.
• **--lib-dir <directory>**: Destination directory for the user library archive file (.a), the user library makefile, and public.mk. If omitted, it defaults to the current directory.
• **--lib-name <filename>**: Name of the user library being created. The user library file name is the user library name with a lib prefix and .a suffix added. Do not include these in the user library name itself. If the user library name option is omitted, the user library name defaults to the name of the first source file with its extension removed.
• **--log <filename>**: Create a debug log and write to specified file. Also logs debug information to stdout.
• **--no-src**: Allows no sources files to be set in the Makefile. You must add source files manually before compiling.
• --public-inc-dir <directory>: Path to a directory that contains C header files (.h) that are to be made available (that is, public) to users of the user library. This directory is added to the appropriate variable in public.mk. Multiple --public-inc-dir options are allowed.

• --set <name> <value>: Set the makefile variable called <name> to <value>. If the variable exists in the managed section of the makefile, <value> replaces the default settings. It adds the makefile variable if it does not already exist. Multiple --set options are allowed.

• --silent: Suppress information about the command’s operation normally sent to stdout.

• --src-dir <directory>: Search for source files in <directory>. Use . to look in the current directory. Multiple --src-dir options are allowed.

• --src-files <filenames>: A list of space-separated source file names added to the makefile. The list of file names is terminated by the next option or the end of the command line. Multiple --src-files options are allowed.

• --src-rdir <directory>: Same as --src-dir option but recursively search for source files in or under <directory>. Multiple --src-rdir options are allowed and can be freely mixed with --src-dir options.

• --use-lib-dir <directory>: Path to a dependent user library directory. The user library directory must contain a makefile fragment called public.mk. Multiple --use-lib-dir options are allowed.

• --verbose: Output verbose, and default information about the command’s operation to stdout.

• --version: Displays the version of this command and exits with a zero exit status.

Description

The nios2-lib-generate-makefile command generates a user library makefile (called Makefile). The path to a BSP created by nios2-bsp-generate-files is an optional command-line option.

If no command-line arguments are specified, this command returns an exit value of 1 and sends a help message to stderr.

Note: For more information about this command, use the --extended-help option to display comprehensive usage information.

nios2-bsp-editor

Usage


Options

• --extended-help: Displays full information about this command and its options.

• --fontsize <point size>: The default point size for GUI fonts is 11. Use this option to adjust the point size.

• --help: Displays basic information about this command and its options.

• --librarian-factory-path <directory>: Comma-separated librarian search path. Use $ for default factory search path.

• --librarian-path <directory>: Comma-separated librarian search path. Use $ for default search path.
- **--log <filename>**: Create a debug log and write to specified file.
- **--settings <filename>**: File name of an existing BSP settings file to update.
- **--version**: Displays the version of this command and exits with a zero exit status.

**Description**

The **nios2-bsp-editor** command is a GUI application for creating and editing board support packages for designs.

**Note:** For more information about this command, use the **--extended-help** option to display comprehensive usage information.

**nios2-app-update-makefile**

**Usage**

nios2-app-update-makefile --app-dir <directory>
[--add-lib-dir <directory>] [--add-src-dir <directory>]
[--add-src-files <filenames>] [--add-src-rdir <directory>] [--debug]
[--get-asflags] [--get-bsp-dir] [--get-debug-level]
[--get-defined-symbols] [--get-elf-name] [--get-optimization]
[--get-undefined-symbols] [--get-user-flags] [--get-warnings]
[--help] [--list-lib-dir] [--list-src-files] [--lock]
[--log <filename>] [--no-src] [--remove-lib-dir <directory>]
[--remove-src-dir <directory>] [--remove-src-files <filenames>]
[--remove-src-rdir <directory>] [--set <name>]
[--set-asflags <value>] [--set-bsp-dir <directory>]
[--set-debug-level <value>] [--set-defined-symbols <value>]
[--set-elf-name <name>] [--set-optimization <value>]
[--set-undefined-symbols <value>] [--set-user-flags <value>]
[--set-warnings <value>] [--show-managed-section] [--show-names]
[--silent] [--unlock] [--verbose] [--version]

**Options**

- **--app-dir <directory>**: Path to the Application Directory with the generated makefile.
- **--add-lib-dir <directory>**: Add a path to dependent user library directory
- **--add-src-dir <directory>**: Add source files in <directory>. Use . to look in the current directory. Multiple --add-src-dir options are allowed.
- **--add-src-files <filenames>**: A list of space-separated source file names to be added to the makefile. The list of file names is terminated by the next option or the end of the command line. Multiple --src-files options are allowed.
- **--add-src-rdir <directory>**: Same as --add-src-dir option but recursively search for source files in or under <directory>. Multiple --add-src-rdir options are allowed and can be freely mixed with --src-dir options.
- **--debug**: Output debug, exception traces, verbose, and default information about the command's operation to stdout.
- **--extended-help**: Displays full information about this command and its options.
- **--force**: Update the Makefile even if it's locked
- **--get <name>**: Get the values of Makefile variables
- **--get-all**: Get all variables in the managed section of the Makefile
- **--get-asflags**: Get user assembler flags
- **--get-bsp-dir**: Get the BSP generated files directory
- **--get-debug-level**: Get debug level flag
nios2-app-update-makefile

- `--get-defined-symbols`: Get defined symbols flag
- `--get-elf-name`: Get the name of .elf file
- `--get-optimization`: Get optimization flag
- `--get-undefined-symbols`: Get undefined symbols flag
- `--get-user-flags`: Get user flags
- `--get-warnings`: Get warnings flag
- `--help`: Displays basic information about this command and its options.
- `--list-lib-dir`: List all paths to dependent user library directories
- `--list-src-files`: List all source files in the makefile.
- `--lock`: Lock the Makefile to prevent it from being updated
- `--log <filename>`: Create a debug log and write to specified file. Also logs debug information to stdout.
- `--no-src`: Remove all source files in the makefile
- `--remove-lib-dir <directory>`: Remove a path to dependent user library directory
- `--remove-src-dir <directory>`: Remove source files in <directory>. Use . to look in the current directory. Multiple `--remove-src-dir` options are allowed.
- `--remove-src-files <filenames>`: A list of space-separated source file names to be removed from the makefile. The list of file names is terminated by the next option or the end of the command line. Multiple `--src-files` options are allowed.
- `--remove-src-rdir <directory>`: Same as `--remove-src-dir` option but recursively search for source files in or under <directory>. Multiple `--remove-src-rdir` options are allowed and can be freely mixed with `--src-dir` options.
- `--set <name> <value>`: Set the value of a Makefile variable called <name>
- `--set-asflags <value>`: Set user assembler flags
- `--set-bsp-dir <directory>`: Set the BSP generated files directory
- `--set-debug-level <value>`: Set debug level flag
- `--set-defined-symbols <value>`: Set defined symbols flag
- `--set-elf-name <name>`: Set the name of .elf file
- `--set-optimization <value>`: Set optimization flag
- `--set-undefined-symbols <value>`: Set undefined symbols flag
- `--set-user-flags <value>`: Set user flags
- `--set-warnings <value>`: Set warnings flag
- `--show-managed-section`: Show the managed section in the Makefile
- `--show-names`: Show name of the variables
- `--silent`: Suppress information about the command’s operation normally sent to stdout.
- `--unlock`: Unlock the Makefile
- `--verbose`: Output verbose, and default information about the command’s operation to stdout.
- `--version`: Displays the version of this command and exits with a zero exit status.

Description

The `nios2-app-update-makefile` command updates an application makefile to add or remove source files. If no command-line arguments are specified, this command returns an exit value of 1 and sends a help message to stderr.

Note: The `--add-src-dir, --add-src-rdir, --remove-src-dir, and --remove-src-rdir` options add and remove files found in `<directory>` at the time the command is executed. Files subsequently added to or removed from the directory are not reflected in the makefile.
Note: For more information about this command, use the --extended-help option to display comprehensive usage information.

nios2-lib-update-makefile

Usage

nios2-lib-update-makefile --lib-dir <directory>

|--add-lib-dir <directory> |--add-pubic-inc-dir <directory>
|--add-src-dir <directory> |--add-src-files <filenames>
|--add-src-rdir <directory> |--debug|--extended-help|--force
|--get <name> |--get-all |--get-asflags |--get-bsp-dir
|--get-debug-level |--get-defined-symbols |--get-lib-name
|--get-optimization |--get-undefined-symbols |--get-user-flags
|--get-warnings |--help |--list-lib-dir |--list-public-inc-dir
|--list-src-files |--lock |--log <filename> |--no-src
|--remove-lib-dir <directory> |--remove-public-inc-dir <directory>
|--remove-sr-dir <directory> |--remove-src-files <filenames>
|--remove-src-rdir <directory> |--set <name> <value>
|--set-asflags <value> |--set-bsp-dir <directory>
|--set-debug-level <value> |--set-defined-symbols <value>
|--set-lib-name <name> |--set-optimization <value>
|--set-undefined-symbols <value> |--set-user-flags <value>
|--set-warnings <value> |--show-managed-section |--show-names
|--silent |--unlock |--verbose |--version

Options

- --add-lib-dir <directory>: Add a path to dependent user library directory
- --add-public-inc-dir <directory>: Add a directory that contains C-language header files
- --add-src-dir <directory>: Add source files in <directory>. Use . to look in the current directory. Multiple --add-src-dir options are allowed.
- --add-src-files <filenames>: A list of space-separated source file names to be added to the makefile. The list of file names is terminated by the next option or the end of the command line. Multiple --src-files options are allowed.
- --add-src-rdir <directory>: Same as --add-src-dir option but recursively search for source files in or under <directory>. Multiple --add-src-rdir options are allowed and can be freely mixed with --src-dir options.
- --debug: Output debug, exception traces, verbose, and default information about the command's operation to stdout.
- --extended-help: Displays full information about this command and its options.
- --force: Update the Makefile even if it is locked
--get <name>: Get the values of Makefile variables
--get-all: Get all variables in the managed section of the Makefile
--get-asflags: Get user assembler flags
--get-bsp-dir: Get the BSP generated files directory
--get-debug-level: Get debug level flag
--get-defined-symbols: Get defined symbols flag
--get-lib-name: Get the name of user library
--get-optimization: Get optimization flag
--get-undefined-symbols: Get undefined symbols flag
--get-user-flags: Get user flags
--get-warnings: Get warnings flag
--help: Displays basic information about this command and its options.
--list-lib-dir: List all paths to dependent user library directories
--list-public-inc-dir: List all public include directories
--list-src-files: List all source files in the makefile.
--lock: Lock the Makefile to prevent it from being updated
--log <filename>: Create a debug log and write to specified file. Also logs debug information to stdout.
--no-src: Remove all source files
--remove-lib-dir <directory>: Remove a path to dependent user library directory
--remove-public-inc-dir <directory>: Remove a include directory
--remove-src-dir <directory>: Remove source files in <directory>. Use . to look in the current directory. Multiple --remove-src-dir options are allowed.
--remove-src-files <filenames>: A list of space-separated source file names to be removed from the makefile. The list of file names is terminated by the next option or the end of the command line. Multiple --src-files options are allowed.
--remove-src-rdir <directory>: Same as --remove-src-dir option but recursively search for source files in or under <directory>. Multiple --remove-src-rdir options are allowed and can be freely mixed with --src-dir options.
--set <name> <value>: Set the value of a Makefile variable called <name>
--set-asflags <value>: Set user assembler flags
--set-bsp-dir <directory>: Set the BSP generated files directory
--set-debug-level <value>: Set debug level flag
--set-defined-symbols <value>: Set defined symbols flag
--set-lib-name <name>: Set the name of user library
--set-optimization <value>: Set optimization flag
--set-undefined-symbols <value>: Set undefined symbols flag
--set-user-flags <value>: Set user flags
--set-warnings <value>: Set warnings flag
--show-managed-section: Show the managed section in the Makefile
--show-names: Show name of the variables
--silent: Suppress information about the command’s operation normally sent to stdout.
--unlock: Unlock the Makefile
--verbose: Output verbose, and default information about the command’s operation to stdout.
--version: Displays the version of this command and exits with a zero exit status.

Description

The **nios2-lib-update-makefile** command updates a user library makefile to add or remove source files.
If no command-line arguments are specified, this command returns an exit value of 1 and sends a help message to stderr.

**Note:** The --add-src-dir, --add-src-rdir, --remove-src-dir, and --remove-src-rdir options add and remove files found in <directory> at the time the command is executed. Files subsequently added to or removed from the directory are not reflected in the makefile.

**Note:** For more information about this command, use the --extended-help option to display comprehensive usage information.

### nios2-swexample-create

#### Usage

```
nios2-create-swexample [Options]
```

#### Options

- **--name:** Specify the name of the software project to create.
- **--sopc-dir:** Specify the SOPC builder directory [deprecated].
- **--sopc-file:** Specify the SOPC builder file [required].
- **--type:** Specify the software design example template type. Required.
- **--list:** List all valid software design example template types.
- **--app-dir:** Specify the application directory to create. Default: `<sopc-dir>/software_examples/app/<name>`
- **--bsp-dir:** Specify the bsp directory to create. Default: `<sopc-dir>/software_examples/bsp/<bsp-type>`
- **--no-app:** Do not generate the create-this-app script
- **--no-bsp:** Do not generate the create-this-bsp script
- **--elf-name:** Name of the .elf file to create.
- **--describe:** Describe the software example type specified and exit.
- **--describeX:** Verbosely describe the software example type specified and exit.
- **--describeAll:** Describe all the software example types and exit.
- **--search:** Search for software example templates in the specified directory. Default: `$SOPC_KIT_NIOS2/examples/software`.
- **--help:** Print this message and exit.
- **--silent:** Do not echo commands.
- **--version:** Print the version number of nios2-create-swexample and exit.
- **--cpu-name:** Create the software example using the cpuname specified.
- **--jvm-max-heap-size:** 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. [Optional]

#### Description

This utility creates a template software example for a given SOPC system.
nios2-elf-insert

Usage
Simple ELF 'fattening' tool used to inject additional sections into an ELF intended for the soft-core processor. The main purpose of this tool is to insert extra information into ELF files so that downstream tools can extract that information. The main motivation for this tool is to make the ELF file the single handoff file that all downstream embedded tools can agree on.

For example, the SBT build flow uses nios-elf-insert to insert the cpu name and the sysid information into the ELF file at build time. Later on downstream, the eclipse debugger can use nios2-elf-query to extract this information to auto-populate all the various GUI settings with the correct default values.

The first argument is the ELF filename: (nios2-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. 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.
```

Related Information
nios2-elf-query on page 15-16

nios2-elf-query

Usage
Simple ELF 'querying' tool used to extract metadata inserted into an ELF intended for the soft-core processor. The main purpose of this tool is so that downstream tools can extract information from an ELF that was inserted using nios2-elf-insert. The main motivation for this tool is to make the ELF file the single handoff file that all downstream embedded tools can agree on.

For example, the SBT build flow uses nios-elf-insert to insert the cpu name and the sysid information into the ELF file at build time. Later on downstream, the eclipse debugger can use nios2-elf-
query to extract this information to auto-populate all the various GUI settings with the correct default values.

The first argument is the ELF filename: (nios2-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. 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.
```

Related Information

nios2-elf-insert on page 15-16

nios2-flash-programmer-generate

Usage

```
nios2-flash-programmer-generate [--accept-bad-sysid]
[---add-bin <fname> <flash-slave-desc> <offset>]
[---add-elf <fname> <flash-slave-desc> <extra-elf2flash-arguments>]
[---add-sof <fname> <flash-slave-desc> <offset>
 <extra-sof2flash-arguments>]
[---cable <cable name>] [---cpu <processor_name>] [---debug]
[---device <device name>] [---erase-first] [---extended-help]
[---flash-dir <directory> [---go] [---help] [---id <address>]
[---instance <instance value>] [---log <filename>] [---mmu]
[---program-flash] [---script-dir <directory>] [---sidp <address>]
[---silent] --sopcinfo <filename> [---verbose] [---version]
```
Options

- **--accept-bad-sysid**: Continue even if the system identifier (ID) comparison fails.
- **--add-bin <fname> <flash-slave-desc> <offset>**: Specify a binary file to convert and program. The filename, target flash slave descriptor, and target offset amount are required. This option can be used multiple times for SRAM Object Files (.sof).
- **--add-elf <fname> <flash-slave-desc> <extra-elf2flash-arguments>**: Specify a .elf file to convert and program. The filename and target flash slave descriptor are required. This option can be used multiple times for .elf files. `<extra-elf2flash-arguments>` can be any of the following options supported by `elf2flash`:
  - save
  - sim_optimize

  The following `elf2flash` options have default values computed, but are also supported as `<extra-elf2flash-arguments>` for manual override of those defaults:
  - base
  - boot
  - end
  - reset
- **--add-sof <fname> <flash-slave-desc> <offset> <extra-sof2flash-arguments>**: Specify a .sof file to convert and program. The filename, target flash slave descriptor, and target offset arguments are required. This option can be used multiple times for .sof files. `<extra-sof2flash-arguments>` can be any of the following options supported by `sof2flash`:
  - activeparallel
  - compress
  - save
  - timestamp
  - options
- **--cable <cable name>**: Specifies which JTAG cable to use (not needed if you only have one cable). Not used without `--program-flash` option.
- **--cpu <processor_name>**: The Nios II processor name from the .sopcinfo file. Not required if only one Nios II processor in the system.
- **--debug**: Sends debug information, exception traces, verbose output, and default information about the command's operation, to stdout.
- **--device <device name>**: Specifies in which device you want to look for the Nios II debug core. Device 1 is the device nearest TDI. Not used without `--program-flash` option.
- **--erase-first**: Erase entire flash targets before programming them. Not used without `--program-flash` option.
- **--extended-help**: Displays full information about this command and its options.
- **--flash-dir <directory>**: Path to the directory where the flash files are generated. Use . for the current directory. This command overwrites pre-existing files in `<directory>` without warning.
- **--go**: Run processor from reset vector after program.
- **--help**: Displays basic information about this command and its options.
- **--id <address>**: Unique ID code for target system. Not used without `--program-flash` option.
- **--instance <instance value>**: Specifies the INSTANCE value of the debug core (not needed if there is exactly one on the chain). Not used without `--program-flash` option.
- **--log <filename>**: Creates a debug log and write to specified file. Also logs debug information to stdout.
- **--mmu**: Specifies if the processor with the corresponding INSTANCE value has an MMU (not needed if there is exactly one processor in the system). Not used without `--program-flash` option.
• **--program-flash**: Providing this flag causes calls to **nios2-flash-programmer** to be generated and executed. This results in flash targets being programmed.

• **--script-dir <directory>**: Path to the directory where a shell script of this tool's executed command lines is generated. This script can be used in place of this **nios2-flash-programmer-generate** command. Use . for the current directory. This command overwrites pre-existing files in <directory> without warning.

• **--sidp <address>**: Base address of system ID peripheral on the target. Not used without **--program-flash** option.

• **--silent**: Suppresses information about the command's operation normally sent to stdout.

• **--sopcinfo <filename>**: The .sopcinfo file.

• **--verbose**: Sends verbose output, and default information about the command's operation, to stdout.

• **--version**: Displays the version of this command and exits with a zero exit status.

**Description**

The **nios2-flash-programmer-generate** command converts multiple files to a .flash in Motorola S-record format, and programs them to the designated target flash devices (**--program-flash**). This is a convenience utility that manages calls to the following command line utilities

- bin2flash
- elf2flash
- sof2flash
- nios2-flash-programmer

This utility also generates a script that captures the sequence of conversion and flash programmer commands.

If no command-line arguments are specified, this command returns an exit value of 1 and sends a help message to stderr.

**Example**

```bash
nios2-flash-programmer-generate --sopcinfo=C:\my_design.sopcinfo \
--flash-dir=. \
--add-sof C:\my_design\test.sof 0x0C000000 memory_0 compress save \
--add-elf C:\my_app\my_app.elf 0x08000000 memory_0 \
--program-flash
```

**nios2-bsp**

**Usage**

```
nios2-bsp <bsp-type> <bsp-dir> [<sopc>] [<override>]...
```

**Options**

- **<bsp-type>**: hal or ucosii.
- **<bsp-dir>**: Path to the BSP directory.
- **<sopc>**: The path to the .sopcinfo file or its directory.
- **<override>**: Options to override defaults.
Description

The nios2-bsp script calls nios2-bsp-create-settings or nios2-bsp-update-settings to create or update a BSP settings file, and the nios2-bsp-generate-files command to create the BSP files. The Nios II Embedded Design Suite (EDS) supports the following BSP types:

- hal
- ucosii

BSP type names are case-insensitive.

This utility produces a BSP of `<bsp-type>` in `<bsp-dir>`. If the BSP does not exist, it is created. If the BSP already exists, it is updated to be consistent with the associated hardware system.

The default Tcl script is used to set the following system-dependent settings:

- `stdio` character device
- System timer device
- Default linker memory
- Boot loader status (enabled or disabled)

If the BSP already exists, nios2-bsp overwrites these system-dependent settings.

The default Tcl script is installed at `<Nios II EDS install path>/sdk2/bin/bsp-set-defaults.tcl`

When creating a new BSP, this utility runs nios2-bsp-create-settings, which creates `settings.bsp` in `<bsp-dir>`.

When updating an existing BSP, this utility runs nios2-bsp-update-settings, which updates `settings.bsp` in `<bsp-dir>`.

After creating or updating the `settings.bsp` file, this utility runs nios2-bsp-generate-files, which generates files in `<bsp-dir>`

Required arguments:

- `<bsp-type>`: Specifies the type of BSP. This argument is ignored when updating a BSP. This argument is case-insensitive. The nios2-bsp script supports the following values of `<bsp-type>`:
  - hal
  - ucosii
  - `<bsp-dir>`: Path to the BSP directory. Use "." to specify the current directory.
Optional arguments:

- `<sopc>`: The path name of the .sopcinfo file. Alternatively, specify a directory containing a .sopcinfo file. In the latter case, the tool finds a file with the extension .sopcinfo. This argument is ignored when updating a BSP. If you omit this argument, it defaults to the current directory.

- `<override>`: Options to override defaults. The `nios2-bsp` script passes most overrides to `nios2-bsp-create-settings` or `nios2-bsp-update-settings`. It also passes the --silent, --verbose, --debug, and --log options to `nios2-bsp-generate-files`

`nios2-bsp` passes the following overrides to the default Tcl script:

- `--default_stdio <device>` | none|DONT_CHANGE
  Specifies stdio device.

- `--default_sys_timer <device>` | none|DONT_CHANGE
  Specifies system timer device.

- `--default_memory_regions DONT_CHANGE`
  Suppresses creation of new default memory regions when updating a BSP. Do not use this option when creating a new BSP.

- `--default_sections_mapping <region>` | DONT_CHANGE
  Specifies the memory region for the default sections.

- `--use_bootloader 0|1|DONT_CHANGE`
  Specifies whether a boot loader is required.
  On a preexisting BSP, the value DONT_CHANGE prevents associated settings from changing their current value.

**Note:** The "--" prefix is stripped when the option is passed to the underlying utility.

If no command-line arguments are specified, this command returns an exit value of 1 and sends a help message to stderr.

**nios2-bsp-console**

**Usage**

```
nios2-bsp-console [--cmd <tcl> <command>] [--extended-help] [--gui]
    [--help] [--script <filename>] [--version]
```

**Options**

- `--cmd <tcl> <command>`: Runs the specified Tcl command. Multiple --cmd options are allowed.

- `--extended-help`: Displays full information about this command and its options. Lists BSP Tcl command help for the --cmd and --script options

- `--gui`: This option opens a Tcl console for creating, editing, and generating BSPs.

- `--help`: Displays basic information about this command and its options.

- `--script <filename>`: Run the specified Tcl script with optional arguments. Multiple --script options are allowed.

- `--version`: Displays the version of this command and exits with a zero exit status.
When invoked with no arguments, **nios2-bsp-console** starts an interactive command-line Tcl interpreter for creating, editing, and generating BSPs.

**Related Information**

*Tcl Commands for BSP Settings* on page 15-64

### alt-file-convert (BETA)

**Description**

alt-file-convert (BETA): General file conversion tool. For , it is primarily used for generating a Nios II application image for Max Onchip Flash and EPCQ.

**Usage**

alt-file-convert -I <input_type> -O <output_type> [option(s)] --input=<input_file> --output=<output_file>

For , the BETA version is limited to the following uses:

- Convert between Intel HEX (byte addressed) and Quartus HEX (word addressed)
- Convert between Quartus HEX files of various widths
- Convert an .elf file to a HEX file and append a bootcopier (used for application flash image for Max On-chip Flash and EPCQ)

**Options**

- `-h, --help` - prints usage
- `-I - input type`
- `-O - output type`
- `--base - base address (in hex) of target memory (default 0x0)`
- `--end - end address (in hex) of target memory (default 0xFFFFFFFF)`
- `--reset - reset address (in hex) of target memory (default None)`
- `--input - path to input file`
- `--output - path to output file`
- `--in-data-width - data width of inputfile [8, 16, 32, 64, 128, 256] (default 8)`
- `--out-data-width - data width of target memory [8, 16, 32, 64, 128, 256] (default None)`
- `--boot - location of boot file to be appended (srec format)`

**Examples**

Converting from Intel Hex (IHEX) to a Quartus Hex (HEX) for a memory with a 32-bit data width:

```
alt-file-convert -I ihex -O hex out-data-width 32 in.ihex out.hex
```

Converting from an .elf file to a flash and appending a bootcopier given as a srec file:

```
alt-file-convert -I elf32-littlenios2 -O flash in.elf out.flash --have-boot-copier --boot boot.elf
```
Design Example Scripts

The SBT includes commands that allow you to create sample BSPs and applications. In this example, `nios2-swexample-create` is used to create the `create_helloworld.sh` script.

Example 15-1: create_helloworld.sh

```bash
# Define user definitions
PROJECT_NAME=test
SAMPLE_TYPE=hello_world_small
SOPCINFO_DIR=./Bemicro_nios_v2_sample
SOPCINFO_FILE=qsys.sopcinfo
CPU_NAME=CPU

# Define internal symbols
BSP_STR=_bsp
APP_NAME=$PROJECT_NAME
BSP_NAME=$PROJECT_NAME$BSP_STR
APP_DIR=$SOPCINFO_DIR/software/$APP_NAME
BSP_DIR=$SOPCINFO_DIR/software/$BSP_NAME

# Create create-this-app and create-this-bsp script
nios2-swexample-create --name=$PROJECT_NAME \
    --type=$SAMPLE_TYPE \
    --sopc-file=$SOPCINFO_DIR/$SOPCINFO_FILE \
    --app-dir=$APP_DIR \
    --bsp-dir=$BSP_DIR

# Build BSP and Application

cd $APP_DIR
./create-this-app
```

Related Information

`nios2-swexample-create` on page 15-15

create-this-bsp

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

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

create-this-app

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

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

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

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

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

The create-this-app script for each software design example is in <Nios II EDS install path>/examples/<HDL>/niosII_<board type>/<design name>/software_examples/app/<example name>. For example, the create-this-app script for the Hello World software example running on the triple-speed ethernet design example for the Stratix IV GX FPGA Development Kit might be located in c:/altera/100/nios2eds/examples/verilog/niosII_stratixIV_4sgx230/triple_speed_ethernet_design/software_examples/app/hello_world.

Similarly, the create-this-bsp script for each software design example is in <Nios II EDS install path>/examples/<HDL>/niosII_<board type>/<design name>/software_examples/bsp/<BSP_type>. For example, the create-this-bsp script for the basic HAL BSP to run on the triple-speed ethernet design example for the Stratix IV GX FPGA Development Kit might be located in c:/altera/100/nios2eds/examples/verilog/niosII_stratixIV_4sgx230/triple_speed_ethernet_design/software_examples/bsp/hal_default.
Figure 15-1: Software Design Example Directory Structure

```
<design> (e.g. standard)
    Quartus II files (e.g. standard.qpf)
    Hardware system files (e.g. standard.sopcinfo)
    software_examples
        app
            software examples (e.g. hello_world)
        bsp
            BSP examples (e.g. hal_standard)
```

Settings Managed by the Software Build Tools

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 such as `create-this-bsp`, 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 FPGA 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 SBT 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

- [Getting Started from the Command Line](#) on page 3-1
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 15-5: Allowed Formats for each Setting Type

<table>
<thead>
<tr>
<th>Setting Type</th>
<th>Format When Setting</th>
<th>Format When Getting</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>0/1 or false/true</td>
<td>0/1</td>
</tr>
<tr>
<td>number</td>
<td>0x prefix for hexadecimal or no prefix for a decimal number</td>
<td>decimal</td>
</tr>
<tr>
<td>string</td>
<td>Quoted string</td>
<td>Quoted string</td>
</tr>
<tr>
<td></td>
<td>Use &quot;none&quot; to set empty string.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>In the SBT command line, to specify a string value with embedded spaces, surround the string with a backslash-double-quote sequence (*). For example:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>--set APP_INCLUDE_DIRS &quot;lcd board&quot;</td>
<td></td>
</tr>
</tbody>
</table>

Table 15-6: BSP Setting Contexts

<table>
<thead>
<tr>
<th>Setting Context</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel FPGA LWHAL</td>
<td>Settings available with the Intel FPGA Lightweight HAL BSP or any BSP based on it.</td>
</tr>
<tr>
<td>Intel FPGA HAL</td>
<td>Settings available with the Intel FPGA HAL BSP or any BSP based on it (for example, Micrium MicroC/OS-II).</td>
</tr>
<tr>
<td>Micrium MicroC/OS-II</td>
<td>Settings available if using the Micrium MicroC/OS-II BSP. All Intel FPGA HAL BSP settings are also available because MicroC/OS-II is based on the Intel FPGA HAL BSP.</td>
</tr>
<tr>
<td>Intel FPGA BSP Makefile Generator</td>
<td>Settings available if using the Intel FPGA BSP makefile generator (generates the Makefile and public.mk files). These settings control the contents of makefile variables. This generator is always present in Intel FPGA HAL BSPs or any BSPs based on the Intel FPGA HAL.</td>
</tr>
<tr>
<td>Intel FPGA BSP Linker Script Generator</td>
<td>Settings available if using the Intel FPGA BSP linker script generator (generates the linker.x and linker.h files). This generator is always present in Intel FPGA HAL BSPs or any BSPs based on the Intel FPGA HAL.</td>
</tr>
</tbody>
</table>
Do not confuse BSP settings with BSP Tcl commands. This section covers BSP settings, including their types, meanings, and legal values.

Related Information
Tcl Commands for BSP Settings on page 15-64

Overview of Component and Driver Settings

The EDS 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.

Related Information
Tcl Commands for BSP Settings on page 15-64

Intel FPGA Host-Based File System Settings

The Intel FPGA host-based file system has one setting. If the Intel FPGA host-based file system is enabled, you must debug (not run) applications based on the BSP for the host-based file system to function. The host-based file system relies on the GNU debugger running on the host to provide host-based file operations.

The host-based file system enables debugging information in your project, by setting APP_CFLAGS_OPTIMIZATION to -g in the makefile.

To include the host-based file system in your BSP, enable the altera_hostfs software package.

Intel FPGA Read-Only Zip File System Settings

The Intel FPGA read-only zip file system has several settings. If the read-only zip file system is enabled, it adds -DUSE_RO_ZIPFS to ALT_CPPFLAGS in public.mk.

To include the read-only zip file system in your BSP, enable the altera_ro_zipfs software package.

Intel FPGA NicheStack TCP/IP - Nios II Edition Stack Settings

The Intel FPGA NicheStack TCP/IP Network Stack - Nios II Edition has several settings. The stack is only available in MicroC/OS-II BSPs. If the NicheStack TCP/IP stack is enabled, it adds -DALT_INICHE to ALT_CPPFLAGS in public.mk.

To include the NicheStack TCP/IP networking stack in your BSP, enable the altera_iniche software package.

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.

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.
Settings Reference

This section lists all settings for BSPs, software packages, and device drivers.

hal.enable_instruction_related_exceptions_api

- **Identifier:** none
- **Type:** Boolean definition
- **Default Value:** false
- **Destination File:** none
- **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:** none
- **Type:** Decimal number
- **Default Value:** 32
- **Destination File:** none
- **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.disable_startup_thread_sync

- **Identifier:** ALT_DISABLE_STARTUP_THREAD_SYNC
- **Type:** Boolean definition
- **Default Value:** false
- **Destination File:** `system.h`
- **Description:** Disables thread synchronization checking on startup. By default, startup code in `crt0.S` assumes that the .rdata section must be reloaded every time the system is reset. Thread 0 waits until the .rdata section is reloaded before executing initialization code.
- **Restriction:** The `hal.disable_startup_thread_sync` setting allows you to disable this restriction in your BSP if your software is written without initialized global or static variables. This setting might be useful if you develop assembly language, and want to take advantage of initialization code in `crt0.S`.
- **Restrictions:** Do not disable startup thread synchronization under the following circumstances:
  - Your code uses initialized global or static variables
  - Your application uses memory management functions such as `alt_malloc()`, `alt_free()` and `alt_calloc()`
hal.enable_small_stack
- **Identifier:** none
- **Type:** Boolean assignment
- **Default Value:** 0
- **Destination File:** public.mk
- **Description:** lwhal.enable_small_stack turns off a build warning that indicates the setting 'lwhal.thread_stack_size' might be too small (< 384 for printf) for your application.
- **Restriction:** none

hal.exclude_default_exception
- **Identifier:** ALT_EXCLUDE_DEFAULT_EXCEPTION
- **Type:** Boolean definition
- **Default Value:** false
- **Destination File:** system.h
- **Description:** Excludes default exception vector. If true, this setting defines the macro ALT_EXCLUDE_DEFAULT_EXCEPTION in system.h.
- **Restriction:** none

hal.sys_clk_timer
- **Identifier:** none
- **Type:** Unquoted string
- **Default Value:** none
- **Destination File:** none
- **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:** none
- **Type:** Unquoted string
- **Default Value:** none
- **Destination File:** none
- **Description:** Slave descriptor of timestamp timer device. This device is used by Intel FPGA HAL timestamp drivers for high-resolution time measurement. This setting defines the value of ALT_TIMESTAMP_CLK in system.h.
- **Restriction:** none

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_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_SIZE
- **Type:** Decimal number
- **Default Value:** 32
- **Destination File:** system.h
- **Description:** Size of name of Event Control Block groups
- **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_SIZE
- **Type:** Decimal number
- **Default Value:** 32
- **Destination File:** system.h
- **Description:** Size of 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()
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_size

- **Identifier:** OS_FLAG_NAME_SIZE
- **Type:** Decimal number
- **Default Value:** 32
- **Destination File:** system.h
- **Description:** Size of name of Event Flags group. CAUTION: This is required by the HAL and many Intel FPGA device drivers; use caution in reducing this value.
- **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_size

- **Identifier:** OS_MEM_NAME_SIZE
- **Type:** Decimal number
- **Default Value:** 32
- **Destination File:** system.h
- **Description:** Size of memory partition name
- **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

altera_avalon_uart_driver.enable_small_driver
- Identifier: ALTERA_AVALON_UART_SMALL
- Type: Boolean definition
- Default Value: false
- Destination File: public.mk
- 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**: public.mk
- **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**: public.mk
- **Description**: Small-footprint (polled mode) driver
- **Restriction**: none

**altera_hostfs.hostfs_name**

- **Identifier**: ALTERA_HOSTFS_NAME
- **Type**: Quoted string
- **Default Value**: /mnt/host
- **Destination File**: system.h
- **Description**: Mount point
- **Restriction**: none

**altera_iniche.iniche_default_if**

- **Identifier**: INICHE_DEFAULT_IF
- **Type**: Quoted string
- **Default Value**: NOT_USED
- **Destination File**: system.h
- **Description**: Deprecated setting: Default media access control (MAC) interface. This setting is used in some legacy Intel FPGA networking examples. It is not needed in new projects. If this setting appears in an existing project, Intel FPGA recommends that you make any necessary changes to remove it. This setting might be removed in a future release.
- **Restriction**: none

**altera_iniche.enable_dhcp_client**

- **Identifier**: DHCP_CLIENT
- **Type**: Boolean definition
- **Default Value**: true
- **Destination File**: system.h
- **Description**: Use dynamic host configuration protocol (DHCP) to automatically assign Internet protocol (IP) address
- **Restriction**: none
altera_iniche.enable_ip_fragments
- **Identifier:** IP_FRAGMENTS
- **Type:** Boolean definition
- **Default Value:** true
- **Destination File:** system.h
- **Description:** Reassemble IP packet fragments
- **Restriction:** none

altera_iniche.enable_include_tcp
- **Identifier:** INCLUDE_TCP
- **Type:** Boolean definition
- **Default Value:** true
- **Destination File:** system.h
- **Description:** Enable Transmission Control Protocol (TCP)
- **Restriction:** none

altera_iniche.enable_tcp_zerocopy
- **Identifier:** TCP_ZEROCOPY
- **Type:** Boolean definition
- **Default Value:** false
- **Destination File:** system.h
- **Description:** Use TCP zero-copy
- **Restriction:** none

altera_iniche.enable_net_stats
- **Identifier:** NET_STATS
- **Type:** Boolean definition
- **Default Value:** false
- **Destination File:** system.h
- **Description:** Enable statistics
- **Restriction:** none

altera_ro_zipfs.ro_zipfs_name
- **Identifier:** ALTERA_RO_ZIPFS_NAME
- **Type:** Quoted string
- **Default Value:** /mnt/rozipfs
- **Destination File:** system.h
- **Description:** Mount point
- **Restriction:** none
altera_ro_zipfs.ro_zipfs_offset
- **Identifier:** ALTERA_RO_ZIPFSOFFSET
- **Type:** Hexadecimal number
- **Default Value:** 0x100000
- **Destination File:** system.h
- **Description:** Offset of file system from base of flash
- **Restriction:** none

altera_ro_zipfs.ro_zipfs_base
- **Identifier:** ALTERA_RO_ZIPFSBASE
- **Type:** Hexadecimal number
- **Default Value:** 0x0
- **Destination File:** system.h
- **Description:** Base address of flash memory device
- **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.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.
hal.linker.enable_interrupt_stack

- **Identifier:** none
- **Type:** Boolean assignment
- **Default Value:** 0
- **Destination File:** none
- **Description:** Enables use of a separate interrupt stack. If true, defines the macro
  `ALT_INTERRUPT_STACK` in `linker.h`, adds a memory region called `interrupt_stack` to `linker.x`, and
  provides the symbols `__alt_interrupt_stack_pointer` and `__alt_interrupt_stack_limit` in
  `linker.x`.
- **Restriction:** The `hal.linker.interrupt_stack_size` and
  `hal.linker.interrupt_stack_memory_region_name` settings must also be valid. This setting must be
  false for MicroC/OS-II BSPs. Only enable this setting for systems with an EIC. If an EIC is not
  implemented, use the separate exception stack to improve interrupt and other exception performance.

.linker.interrupt_stack_memory_region_name

- **Identifier:** none
- **Type:** Unquoted String
- **Default Value:** none
- **Destination File:** none
- **Description:** Name of the existing memory region that is divided up to create the `interrupt_stack`
  memory region. The selected region name is adjusted automatically when the BSP is generated to
  create the `interrupt_stack` memory region.
- **Restriction:** Only used if `hal.linker.enable_interrupt_stack` is true.

hal.linker.interrupt_stack_size

- **Identifier:** none
- **Type:** Decimal Number
- **Default Value:** 1024
- **Destination File:** none
- **Description:** Size of the interrupt stack in bytes.
- **Restriction:** Only used if `hal.linker.enable_interrupt_stack` is true.

hal.make.ar

- **Identifier:** AR
- **Type:** Unquoted string
- **Default Value:** nios2-elf-ar
- **Destination File:** BSP `makefile`
- **Description:** Archiver command. Creates library files.
- **Restriction:** none

hal.make.ar_post_process

- **Identifier:** AR_POST_PROCESS
- **Type:** Unquoted string
- **Default Value:** none
- **Destination File:** BSP `makefile`
- **Description:** Command executed after archiver execution.
- **Restriction:** none
hal.make.ar_pre_process

- Identifier: AR_PRE_PROCESS
- Type: Unquoted string
- Default Value: none
- Destination File: BSP makefile
- Description: Command executed before archiver execution.
- Restriction: none

hal.make.as

- Identifier: AS
- Type: Unquoted string
- Default Value: nios2-elf-gcc
- Destination File: BSP makefile
- Description: Assembler command. Note that CC is used for Nios II assembly language source files (.S).
- Restriction: none

hal.make.as_post_process

- Identifier: AS_POST_PROCESS
- Type: Unquoted string
- Default Value: none
- Destination File: BSP makefile
- Description: Command executed after each assembly file is compiled.
- Restriction: none

hal.make.as_pre_process

- Identifier: AS_PRE_PROCESS
- Type: Unquoted string
- Default Value: none
- Destination File: BSP makefile
- Description: Command executed before each assembly file is compiled.
- Restriction: none

hal.make.bsp_arflags

- Identifier: BSP_ARFLAGS
- Type: Unquoted string
- Default Value: -src
- Destination File: BSP makefile
- Description: Custom flags only passed to the archiver. This content of this variable is directly passed to the archiver rather than the more standard ARFLAGS. The reason for this is that GNU Make assumes some default content in ARFLAGS. This setting defines the value of BSP_ARFLAGS in Makefile.
- Restriction: none
hal.make.bsp_asflags

- **Identifier**: BSP_ASFLAGS
- **Type**: Unquoted string
- **Default Value**: -Wa,-gdwarf2
- **Destination File**: BSP makefile
- **Description**: Custom flags only passed to the assembler. This setting defines the value of BSP_ASFLAGS in Makefile.
- **Restriction**: none

hal.make.bsp_cflags_debug

- **Identifier**: BSP_CFLAGS_DEBUG
- **Type**: Unquoted string
- **Default Value**: -g
- **Destination File**: BSP makefile
- **Description**: C/C++ compiler debug level. -g provides the default set of debug symbols typically required to debug a typical application. Omitting -g removes debug symbols from the .elf file. This setting defines the value of BSP_CFLAGS_DEBUG in Makefile.
- **Restriction**: none

hal.make.bsp_cflags_defined_symbols

- **Identifier**: BSP_CFLAGS_DEFINED_SYMBOLS
- **Type**: Unquoted string
- **Default Value**: none
- **Destination File**: BSP makefile
- **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 BSP_CFLAGS_DEFINED_SYMBOLS in the BSP makefile.
- **Restriction**: none

hal.make.bsp_cflags_optimization

- **Identifier**: BSP_CFLAGS_OPTIMIZATION
- **Type**: Unquoted string
- **Default Value**: -O0
- **Destination File**: BSP makefile
- **Description**: C/C++ compiler optimization level. -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 BSP_CFLAGS_OPTIMIZATION in Makefile.
- **Restriction**: none
**hal.make.bsp_cflags_undefined_symbols**

- **Identifier:** BSP_CFLAGS_UNDEFINED_SYMBOLS
- **Type:** Unquoted string
- **Default Value:** none
- **Destination File:** BSP makefile
- **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 defines the value of BSP_CFLAGS_UNDEFINED_SYMBOLS in the BSP Makefile.
- **Restriction:** none

**hal.make.bsp_cflags_user_flags**

- **Identifier:** BSP_CFLAGS_USER_FLAGS
- **Type:** Unquoted string
- **Default Value:** none
- **Destination File:** BSP makefile
- **Description:** Custom flags passed to the compiler when compiling C, C++, and `.S` files. This setting defines the value of BSP_CFLAGS_USER_FLAGS in Makefile.
- **Restriction:** none

**hal.make.bsp_cflags_warnings**

- **Identifier:** BSP_CFLAGS_WARNINGS
- **Type:** Unquoted string
- **Default Value:** `-Wall`
- **Destination File:** BSP makefile
- **Description:** C/C++ compiler warning level. `-Wall` is commonly used. This setting defines the value of BSP_CFLAGS_WARNINGS in Makefile.
- **Restriction:** none

**hal.make.bsp_cxx_flags**

- **Identifier:** BSP_CXXFLAGS
- **Type:** Unquoted string
- **Default Value:** none
- **Destination File:** BSP makefile
- **Description:** Custom flags only passed to the C++ compiler. This setting defines the value of BSP_CXXFLAGS in Makefile.
- **Restriction:** none
hal.make.bsp_inc_dirs
- **Identifier:** BSP_INC_DIRS
- **Type:** Unquoted string
- **Default Value:** none
- **Destination File:** BSP makefile
- **Description:** Space separated list of extra include directories to scan for header files. Directories are relative to the top-level BSP directory. The -I prefix is added by the makefile, therefore you must not include it in the setting value. This setting defines the value of BSP_INC_DIRS in the makefile.
- **Restriction:** none

hal.make.build_post_process
- **Identifier:** BUILD_POST_PROCESS
- **Type:** Unquoted string
- **Default Value:** none
- **Destination File:** BSP makefile
- **Description:** Command executed after BSP built.
- **Restriction:** none

hal.make.build_pre_process
- **Identifier:** BUILD_PRE_PROCESS
- **Type:** Unquoted string
- **Default Value:** none
- **Destination File:** BSP makefile
- **Description:** Command executed before BSP built.
- **Restriction:** none

hal.make.cc
- **Identifier:** CC
- **Type:** Unquoted string
- **Default Value:** nios2-elf-gcc -xc
- **Destination File:** BSP makefile
- **Description:** C compiler command
- **Restriction:** none

hal.make.cc_post_process
- **Identifier:** CC_POST_PROCESS
- **Type:** Unquoted string
- **Default Value:** none
- **Destination File:** BSP makefile
- **Description:** Command executed after each .c or .S file is compiled.
- **Restriction:** none
hal.make.cc_pre_process
- **Identifier**: CC_PRE_PROCESS
- **Type**: Unquoted string
- **Default Value**: none
- **Destination File**: BSP makefile
- **Description**: Command executed before each `.c` or `.S` file is compiled.
- **Restriction**: none

hal.make.cxx
- **Identifier**: CXX
- **Type**: Unquoted string
- **Default Value**: nios2-elf-gcc -xc++
- **Destination File**: BSP makefile
- **Description**: C++ compiler command
- **Restriction**: none

hal.make.cxx_post_process
- **Identifier**: CXX_POST_PROCESS
- **Type**: Unquoted string
- **Default Value**: none
- **Destination File**: BSP makefile
- **Description**: Command executed before each C++ file is compiled.
- **Restriction**: none

hal.make.cxx_pre_process
- **Identifier**: CXX_PRE_PROCESS
- **Type**: Unquoted string
- **Default Value**: none
- **Destination File**: BSP makefile
- **Description**: Command executed before each C++ file is compiled.
- **Restriction**: none

hal.make.ignore_system Derived.big_endian
- **Identifier**: none
- **Type**: Boolean assignment
- **Default Value**: 0
- **Destination File**: public.mk
- **Description**: Enable BSP generation to query if SOPC system is big endian. If true ignores export of `ALT_CFLAGS += -meb` to public.mk if big endian system. If true ignores export of `ALT_CFLAGS += -mel` if little endian system. This setting is intended for big endian processor and has no effect to default little endian processor.
- **Restriction**: none

*Note*: The `hal.make.ignore_system Derived.big_endian` setting is only available in EDS Standard version.
hal.make.ignore_system_derived.fpu_present

- Identifier: none
- Type: Boolean assignment
- Default Value: 0
- Destination File: public.mk
- Description: Enable BSP generation to query if SOPC system has FPU present. If true ignores export of 'ALT_CFLAGS += -mhard-float' to public.mk if FPU is found in the system. If true ignores export of 'ALT_CFLAGS += -mhard-soft' if FPU is not found in the system.
- Restriction: none

hal.make.ignore_system_derived.hardware_divide_present

- Identifier: none
- Type: Boolean assignment
- Default Value: 0
- Destination File: public.mk
- Description: Enable BSP generation to query if SOPC system has hardware divide present. If true ignores export of 'ALT_CFLAGS += -mno-hw-div' to public.mk if no division is found in system. If true ignores export of 'ALT_CFLAGS += -mhw-div' if division is found in the system.
- Restriction: none

hal.make.ignore_system_derived.hardware_fp_cust_inst_divider_present

- Identifier: none
- Type: Boolean assignment
- Default Value: 0
- Destination File: public.mk
- Description: Enable BSP generation to query if SOPC system floating point custom instruction with a divider is present. If true ignores export of 'ALT_CFLAGS += -mcustom-fpu-cfg=60-2' and 'ALT_LDFLAGS += -mcustom-fpu-cfg=60-2' to public.mk if the custom instruction is found in the system.
- Restriction: none

hal.make.ignore_system_derived.hardware_fp_cust_inst_no_divider_present

- Identifier: none
- Type: Boolean assignment
- Default Value: 0
- Destination File: public.mk
- Description: Enable BSP generation to query if SOPC system floating point custom instruction without a divider is present. If true ignores export of 'ALT_CFLAGS += -mcustom-fpu-cfg=60-1' and 'ALT_LDFLAGS += -mcustom-fpu-cfg=60-1' to public.mk if the custom instruction is found in the system.
- Restriction: none
hal.make.ignore_system_derived.sopc_simulation_enabled

- **Identifier:** none
- **Type:** Boolean assignment
- **Default Value:** 0
- **Destination File:** public.mk
- **Description:** Enable BSP generation to query if SOPC system has simulation enabled. If true ignores export of 'ELF_PATCH_FLAG += --simulation_enabled' to public.mk.
- **Restriction:** none

hal.make.ignore_system_derived.debug_core_present

- **Identifier:** none
- **Type:** Boolean assignment
- **Default Value:** 0
- **Destination File:** public.mk
- **Description:** Enable BSP generation to query if SOPC system has a debug core present. If true ignores export of 'CPU_HAS_DEBUG_CORE = 1' to public.mk if a debug core is found in the system. If true ignores export of 'CPU_HAS_DEBUG_CORE = 0' if no debug core is found in the system.
- **Restriction:** none

hal.make.ignore_system_derived.hardware_multiplier_present

- **Identifier:** none
- **Type:** Boolean assignment
- **Default Value:** 0
- **Destination File:** public.mk
- **Description:** Enable BSP generation to query if SOPC system has multiplier present. If true ignores export of 'ALT_CFLAGS += -mno-hw-mul' to public.mk if no multiplier is found in the system. If true ignores export of 'ALT_CFLAGS += -mhw-mul' if multiplier is found in the system.
- **Restriction:** none

hal.make.ignore_system_derived.hardware_mulx_present

- **Identifier:** none
- **Type:** Boolean assignment
- **Default Value:** 0
- **Destination File:** public.mk
- **Description:** Enable BSP generation to query if SOPC system has hardware mulx present. If true ignores export of 'ALT_CFLAGS += -mno-hw-mulx' to public.mk if no mulx is found in the system. If true ignores export of 'ALT_CFLAGS += -mhw-mulx' if mulx is found in the system.
- **Restriction:** none
hal.make.ignore_system_derived.sopc_system_base_address

- **Identifier:** none
- **Type:** Boolean assignment
- **Default Value:** 0
- **Destination File:** public.mk
- **Description:** Enable BSP generation to query SOPC system for system ID base address. If true ignores export of `SOPC_SYSID_FLAG += --sidp=<address>` and `ELF_PATCH_FLAG += --sidp=<address>` to public.mk.
- **Restriction:** none

hal.make.ignore_system_derived.sopc_system_id

- **Identifier:** none
- **Type:** Boolean assignment
- **Default Value:** 0
- **Destination File:** public.mk
- **Description:** Enable BSP generation to query SOPC system for system ID. If true ignores export of `SOPC_SYSID_FLAG += --id=<sysid>` and `ELF_PATCH_FLAG += --id=<sysid>` to public.mk.
- **Restriction:** none

hal.make.ignore_system_derived.sopc_system_timestamp

- **Identifier:** none
- **Type:** Boolean assignment
- **Default Value:** 0
- **Destination File:** public.mk
- **Description:** Enable BSP generation to query SOPC system for system timestamp. If true ignores export of `SOPC_SYSID_FLAG += --timestamp=<timestamp>` and `ELF_PATCH_FLAG += --timestamp=<timestamp>` to public.mk.
- **Restriction:** none

hal.make.rm

- **Identifier:** RM
- **Type:** Unquoted string
- **Default Value:** rm -f
- **Destination File:** BSP makefile
- **Description:** Command used to remove files when building the clean target.
- **Restriction:** none

hal.custom_newlib_flags

- **Identifier:** CUSTOM_NEWLIB_FLAGS
- **Type:** Unquoted string
- **Default Value:** none
- **Destination File:** public.mk
- **Description:** Build a custom version of newlib with the specified space-separated compiler flags.
- **Restriction:** The custom newlib build is placed in the <bsp root>/newlib directory, and is used only for applications that utilize this BSP.
hal.enable_c_plus_plus

- **Identifier:** ALT_NO_C_PLUS_PLUS
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** public.mk
- **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, adds -DALT_NO_C_PLUS_PLUS to ALT_CPPFLAGS in public.mk, and reduces code footprint.
- **Restriction:** none

hal.enable_clean_exit

- **Identifier:** ALT_NO_CLEAN_EXIT
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** public.mk
- **Description:** When your application exits, close file descriptors, call C++ destructors, etc. Code footprint can be reduced by disabling clean exit. If disabled, adds -DALT_NO_CLEAN_EXIT to ALT_CPPFLAGS and -Wl,--defsym,exit=_exit to ALT_LDFLAGS in public.mk.
- **Restriction:** none

hal.enable_exit

- **Identifier:** ALT_NO_EXIT
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** public.mk
- **Description:** Add exit() support. This option increases code footprint if your main() routine returns or calls exit(). If false, adds -DALT_NO_EXIT to ALT_CPPFLAGS in public.mk, and reduces footprint.
- **Restriction:** none

hal.enable_gprof

- **Identifier:** ALT_PROVIDE_GMON
- **Type:** Boolean assignment
- **Default Value:** 0
- **Destination File:** public.mk
- **Description:** Causes code to be compiled with gprof profiling enabled and the application .elf file to be linked with the GPROF library. If true, adds -DALT_PROVIDE_GMON to ALT_CPPFLAGS and -pg to ALT_CFLAGS in public.mk.
- **Restriction:** none
hal.enable_lightweight_device_driver_api

- **Identifier:** ALT_USE_DIRECT_DRIVERS
- **Type:** Boolean assignment
- **Default Value:** 0
- **Destination File:** public.mk
- **Description:** Enables lightweight device driver API. This reduces code and data footprint by removing the HAL layer that maps device names (e.g., /dev/uart0) to file descriptors. Instead, driver routines are called directly. The open(), close(), and lseek() routines always fail if called. The read(), write(), fstat(), ioctl(), and isatty() routines only work for the stdio devices. If true, adds `-DALT_USE_DIRECT_DRIVERS` to `ALT_CPPFLAGS` in `public.mk`.
- **Restriction:** The Intel FPGA Host and read-only ZIP file systems cannot be used if `hal.enable_lightweight_device_driver_api` is true.

hal.enable_mul_div_emulation

- **Identifier:** ALT_NO_INSTRUCTION_EMULATION
- **Type:** Boolean assignment
- **Default Value:** 0
- **Destination File:** public.mk
- **Description:** Adds code to the BSP to emulate multiply and divide instructions. This code is independent of any emulation code added by the C/C++ compiler. If false, adds `-DALT_NO_INSTRUCTION_EMULATION` to `ALT_CPPFLAGS` in `public.mk`. You do not normally need to enable this option, because the C/C++ compiler detects whether the target processor core supports the multiply and divide instructions directly. If you compile for a core that lacks support for the instructions, the HAL includes the required software emulation in its run-time libraries. However, you might need to enable `hal.enable_mul_div_emulation` under the following circumstances:
  - You expect to run the software on an implementation of the processor other than the one you compiled for. The best solution is to build your program for the correct processor implementation. Resort to the `hal.enable_mul_div_emulation` if it is not possible to determine the processor implementation at compile time.
  - You have assembly language code that uses an implementation-dependent instruction.
- **Restriction:** none

hal.enable_reduced_device_drivers

- **Identifier:** ALT_USE_SMALL_DRIVERS
- **Type:** Boolean assignment
- **Default Value:** 0
- **Destination File:** public.mk
- **Description:** Certain drivers are compiled with reduced functionality to reduce code footprint. Not all drivers observe this setting. If true, adds `-DALT_USE_SMALL_DRIVERS` to `ALT_CPPFLAGS` in `public.mk`. 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. Several device drivers are disabled entirely in reduced drivers mode. These include the altera_avalon_cfi_flash, altera_avalon_epcs_flash_controller, and altera_avalon_lcd_16207 drivers. As a result, certain API routines fail (HAL flash access routines). You can define a symbol provided by each driver to prevent it from being removed.
- **Restriction:** none
hal.enable_runtime_stack_checking

- **Identifier:** ALT_STACK_CHECK
- **Type:** Boolean assignment
- **Default Value:** 0
- **Destination File:** public.mk
- **Description:** Turns on HAL runtime stack checking feature. Enabling this setting causes additional code to be placed into each subroutine call to generate an exception if a stack collision occurs with the heap or statically allocated data. If true, adds `-DALT_STACK_CHECK` and `-fstack-check` to `ALT_CPPFLAGS` in `public.mk`.
- **Restriction:** none

hal.enable_sim_optimize

- **Identifier:** ALT_SIM_OPTIMIZE
- **Type:** Boolean assignment
- **Default Value:** 0
- **Destination File:** public.mk
- **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 `-DALT_SIM_OPTIMIZE` to `ALT_CPPFLAGS` in `public.mk`.
- **Restriction:** When this setting is true, the BSP cannot run on hardware.

hal.enable_small_c_library

- **Identifier:** none
- **Type:** Boolean assignment
- **Default Value:** 0
- **Destination File:** public.mk
- **Description:** Causes the small newlib (C library) to be used. This reduces code and data footprint at the expense of reduced functionality. Several newlib features are removed such as floating-point support in `printf()`, `stdin` input routines, and buffered I/O. The small C library is not compatible with Micrium MicroC/OS-II. If true, adds `-msmallc` to `ALT_LDFLAGS` and adds `-DSMALL_C_LIB` to `ALT_CPPFLAGS` in `public.mk`.
- **Restriction:** none
hal.enable_sopc_sysid_check

- **Identifier:** none
- **Type:** Boolean assignment
- **Default Value:** 1
- **Destination File:** public.mk
- **Description:** Enables system ID check. If a System ID component is connected to the processor associated with this BSP, the system ID check is enabled in the creation of command-line arguments to download an .elf file to the target. Otherwise, system ID and timestamp values are left out of public.mk for the application makefile download-elf target definition. With the system ID check disabled, the EDS tools do not automatically ensure that the application .elf file (and BSP it is linked against) corresponds to the hardware design on the target. If false, adds --accept-bad-sysid to SOPC_SYSID_FLAG in public.mk. Intel FPGA strongly recommends leaving hal.enable_sopc_sysid_check enabled. This setting is exposed to support rare cases in which FPGA logic resources are in extremely short supply. When the system ID check is disabled, the software is unable to detect whether the software is running on the correct hardware version. This situation can lead to subtle errors that are difficult to diagnose.
- **Restriction:** none

hal.log_port

- **Identifier:** 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:** public.mk
- **Description:** The value is assigned to ALT_LOG_FLAGS in the generated public.mk. Refer to hal.log_port for further details. The valid range of this setting is -1 through 3.

hal.stderr

- **Identifier:** STDERR
- **Type:** Unquoted string
- **Default Value:** none
- **Destination File:** public.mk
- **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:** 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:** 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.

hal.thread_stack_size

- **Identifier:** ALT_THREAD_STACK_SIZE
- **Type:** Decimal number
- **Default Value:** The default value of lwhal.thread_stack_size is selected by the default Tcl script launched when a LWHAL BSP is created. lwhal.thread_stack_size is set to 3/4 of the size of the memory region to which the .stack section is assigned, if the region is shared with other sections (the default case).
- **Destination File:** system.h
- **Description:** Defines stack size for a thread (in bytes). This setting defines the value of ALT_THREAD_STACK_SIZE in system.h.
- **Restriction:** none

## Application and User Library Makefile Variables

The SBT constructs application and makefile libraries for you, inserting makefile variables appropriate to your project configuration. You can control project build characteristics by manipulating makefile variables at the time of project generation. You control a variable with the `--set` command line option, as in the following example:

```
nios2-bsp hal my_bsp --set APP_CFLAGS_WARNINGS "-Wall"
```

The following utilities and scripts support modifying makefile variables with the `--set` option:

- nios2-app-generate-makefile
- nios2-lib-generate-makefile
- nios2-app-update-makefile
- nios2-lib-update-makefile
- nios2-bsp

## Application Makefile Variables

You can modify the following application makefile variables on the command line:
**CREATE_OBJDUMP**—Assign 1 to this variable to enable creation of an object dump file (.objdump) after linking the application. The `nios2-elf-objdump` utility is called to create this file. An object dump contains information about all object files linked into the .elf file. It provides a complete view of all code linked into your application. An object dump contains a disassembly view showing each instruction and its address.

**OBJDUMP_INCLUDE_SOURCE**—Assign 1 to this variable to include source code inline with disassembled instructions in the object dump. When enabled, this includes the `--source` switch when calling the object dump executable. This is useful for debugging and examination of how the preprocessor and compiler generate instructions from higher level source code (such as C) or from macros.

**OBJDUMP_FULL_CONTENTS**—Assign 1 to this variable to include a raw display of the contents of the .text linker section. When enabled, this variable includes the `--full-contents` switch when calling the object dump executable.

**CREATE_.elf_DERIVED_FILES**—Setting this variable to 1 creates the HDL simulation and onchip memory initialization files when you invoke the makefile with the `all` target. When this variable is 0 (the default), these files are only created when you make the `mem_init_generate` or `mem_init_install` target.

**CREATE_LINKER_MAP**—Assign 1 to this variable to enable creation of a link map file (.map) after linking the application. A link map file provides information including which object files are included in the executable, the path to each object file, where objects and symbols are located in memory, and how the common symbols are allocated.

**APP_CFLAGS_DEFINED_SYMBOLS**—This variable allows you to define macros using the `-D` argument, for example `-D <macro name>`. The contents of this variable are passed to the compiler and linker without modification.

**APP_CFLAGS_UNDEFINED_SYMBOLS**—This variable allows you to remove macro definitions using the `-U` argument, for example `-U <macro name>`. The contents of this variable are passed to the compiler and linker without modification.

**APP_CFLAGS_OPTIMIZATION**—The C/C++ compiler optimization level. For example, `-O0` provides no optimization and `-O2` provides standard optimization. `-O0` is recommended for debugging code, because compiler optimization can remove variables and produce non-sequential execution of code while debugging.

**APP_CFLAGS_DEBUG_LEVEL**—The C/C++ compiler debug level. `-g` provides the default set of debug symbols typically required to debug an application. Omitting `-g` omits debug symbols from the .elf.

**APP_CFLAGS_WARNINGS**—The C/C++ compiler warning level. `-Wall` is commonly used, enabling all warning messages.

**APP_CFLAGS_USER_FLAGS**

**APP_INCLUDE_DIRS**—Use this variable to specify paths for the preprocessor to search. These paths commonly contain C header files (.h) that application code requires. Each path name is formatted and passed to the preprocessor with the `-I` option.

You can add multiple directories by enclosing them in double quotes, for example `--set APP_INCLUDE_DIRS "../my_includes ../../other_includes"`.

**APP_LIBRARY_DIRS**—Use this variable to specify paths for additional libraries that your application links with.

**Note:** When you specify a user library path with `APP_LIBRARY_DIRS`, you also need to specify the user library names with the `APP_LIBRARY_NAMES` variable.

`APP_LIBRARY_DIRS` specifies only the directory where the user library file(s) are located, not the library archive file (.a) name.

**Note:** Do not use this variable to specify the path to a BSP or user library created with the SBT. The paths to these libraries are specified in `public.mk` files included in the application makefile.
You can add multiple directories by enclosing them in double quotes, for example

```
--set APP_LIBRARY_DIRS "../my_libs ../../other_libs"
```

- **APP_LIBRARY_NAMES**—Use this variable to specify the names of additional libraries that your application must link with. Library files are `.a` files.

**Note:** You do not specify the full name of the `.a` file. Instead, you specify the user library name `<name>`, and the SBT constructs the filename `lib<name>.a`. For example, if you add the string "math" to **APP_LIBRARY_NAMES**, the SBT assumes that your library file is named `libmath.a`.

Each specified user library name is passed to the linker with the `-l` option. The paths to locate these libraries must be specified in the **APP_LIBRARY_DIRS** variable.

**Note:** You cannot use this variable to specify a BSP or user library created with the SBT. The paths to these libraries are specified in `public.mk` file included in the application makefile.

- **BUILD_PRE_PROCESS**—This variable allows you to specify a command to be executed prior to building the application, for example,

```
cp *.elf ../lastbuild
```

- **BUILD_POST_PROCESS**—This variable allows you to specify a command to be executed after building the application, for example,

```
cp *.elf //production/test/nios2executables
```

### User Library Makefile Variables

You can modify the following user library makefile variables on the command line:

- **LIB_CFLAGS_DEFINED_SYMBOLS**—This variable allows you to define macros using the `-D` argument, for example `-D <macro name>`. The contents of this variable are passed to the compiler and linker without modification.

- **LIB_CFLAGS_UNDEFINED_SYMBOLS**—This variable allows you to remove macro definitions using the `-U` argument, for example `-U <macro name>`. The contents of this variable are passed to the compiler and linker without modification.

- **LIB_CFLAGS_OPTIMIZATION**—The C/C++ compiler optimization level. For example, `-O0` provides no optimization and `-O2` provides standard optimization. `-O0` is recommended for debugging code, because compiler optimization can remove variables and produce non-sequential execution of code while debugging.

- **LIB_CFLAGS_DEBUG_LEVEL**—The C/C++ compiler debug level. `-g` provides the default set of debug symbols typically required to debug an application. Omitting `-g` omits debug symbols from the `.elf`.

- **LIB_CFLAGS_WARNINGS**—The C/C++ compiler warning level. `-Wall` is commonly used, enabling all warning messages.

- **LIB_CFLAGS_USER_FLAGS**
- **LIB_INCLUDE_DIRS**—You can add multiple directories by enclosing them in double quotes, for example

```
--set LIB_INCLUDE_DIRS "../my_includes ../../other_includes"
```

### Standard Build Flag Variables

The SBT creates makefiles supporting the following standard makefile command-line variables:

- **CFLAGS**
- **CPPFLAGS**
- **ASFLAGS**
- **CXXFLAGS**

You can define flags in these variables on the makefile command line, or in a script that invokes the makefile. The makefile passes these flags on to the corresponding GCC tool.
Software Build Tools Tcl Commands

Tcl commands are a crucial component of the SBT. 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.

Tcl Command Environments

The SBT 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 SBT.

The following sections describe each Tcl environment in detail, listing the available commands.

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 with the utilities **nios2-bsp-create-settings**, **nios2-bsp-update-settings**, and **nios2-bsp-query-settings**
- Adding the Tcl commands in a Tcl script specified with the **--script** option

Related Information

- **Settings Managed by the Software Build Tools** on page 15-25
  For more information about the settings that are available in a Nios II project and the tools that you use to specify and manipulate these settings.
- **Software Build Tools Utilities** on page 15-1
  For more information about how to call Tcl commands from utilities.
- **nios2-bsp-create-settings** on page 15-4
- **nios2-bsp-update-settings** on page 15-8
- **nios2-bsp-query-settings** on page 15-7

add_memory_device

**Usage**

```
add_memory_device <device name> <base address> <span>
```

**Options**

- **<device name>**: String with the name of the memory device.
- **<base address>**: The base address of the memory device. Hexadecimal or decimal string.
- **<span>**: 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.

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.
- <span>: 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

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
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.

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.

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
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 BSP makefile. It is an error to disable a software package that is not enabled.

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_hostfs:7.2
- Example 2: enable_sw_package my_sw_package

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
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

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

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_hostfs:7.2 altera_ro_zipfs:7.2

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

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

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).
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

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}

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

```ruby
puts [get_current_section_mappings]
```

Returns:

```
{.text onchip_ram0} {.rodata onchip_ram0} {.rwdata onchip_ram0} {.bss onchip_ram0} {.heap onchip_ram0} {.stack onchip_ram0}
```

get_default_memory_regions

Usage

```ruby
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

```ruby
puts [get_default_memory_regions]
```

Returns:

```
{reset onchip_ram0 0 32} {onchip_ram0 onchip_ram0 32 1048544}
```

get_driver

Usage

```ruby
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.
get_enabled_sw_packages

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

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_hostfs:7.2

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
**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

**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

**get_fast_tlb_miss_exception_slave_desc**

**Usage**
get_fast_tlb_miss_exception_slave_desc

**Options**
None
get_interrupt_controller_id

Description
Returns the interrupt controller ID of the slave descriptor (-1 if not a connected interrupt controller).

Usage
get_interrupt_controller_id <slave_desc>

Options
• <slave_desc>: String with the slave descriptor to query.

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).

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.
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

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

get_module_name

Usage

get_module_name <slave_desc>

Options

• <slave_desc>: String with the slave descriptor to query.
get_reset_offset

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

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

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
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

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".

The get_setting command is equivalent to the --get command-line option.

Example

puts [get_setting hal.enable_gprof]

Returns:

0

get_setting_desc

Usage

get_setting_desc <name>

Options

• <name>: String with the name of the setting to get the description for.
get_slave_descs

Description
Returns a string describing the BSP setting.

Example
puts [get_setting_desc hal.enable_gprof]
Returns:
"This example compiles the code with gprof profiling enabled and links the application .elf with the GPROF library. If true, adds `-DALT_PROVIDE_GMON` to `ALT_CPPFLAGS` and `-pg` to `ALT_CFLAGS` in public.mk."

get_slave_descs

Usage
get_slave_descs

Options
None

Description
Returns a sorted list of all the slave descriptors connected to the processor.

Example
puts [get_slave_descs]
Returns:
jtag_uart0 onchip_ram0

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

---

**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.

---

**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.

---

**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.
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.

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.

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
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.

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.

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.

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."

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.

logVerbose

Usage
logVerbose <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.

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

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 SBT 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 `nios2-bsp-create-settings` 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

Software Build Tools on page 4-1
For more information about a list of generated files associated with HAL and MicroC/OS-II BSP

set_setting

Usage

set_setting <name> <value>

Options

- `<name>`: String with the name of the setting.
- `<value>`: String with the value of the setting.
update_memory_region

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. The set_setting command is equivalent to the --set command-line option.

Example
set_setting hal.enable_gprof true

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

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
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.

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.

generate_bsp

Usage

generate_bsp <bspDir>

Options

• <bspDir>: BSP directory to generate files to.

Description

Generates a new BSP.

get_available_bsp_type_versions

Usage

get_available_bsp_type_versions <bsp_type> <sopcinfo_path>
get_available_bsp_types

Options

- `<bsp_type>`: BSP type identifier (e.g. hal, ucosii).
- `<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.

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.

get_available_cpu_architectures

Usage

get_available_cpu_architectures

Description

Gets the available processor architectures.

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.

get_available_software

Usage

get_available_software <bsp_type> <filter> <sopcinfo_path>
Options

- `<bsp_type>`: BSP type identifier (e.g. hal, ucosii).
- `<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.

`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.

`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.
get_bsp_version

Usage

get_bsp_version

Description

Gets the version of the BSP operating system software element.

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. nios2) of a specified processor instance given a SOPC system.

get_nios2_dpx_thread_num

Usage

get_nios2_dpx_thread_num

Description

If the BSP is mastered by a DPX processor, then this function returns the number of threads the processor supports. Otherwise it returns null.

get_sopcinfo_file

Usage

get_sopcinfo_file

Description

Returns the path of the BSP specific SOPC Information File.

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.

**is_bsp_hal_extension**

Usage

```bash
is_bsp_hal_extension
```

Description

Returns a boolean value that indicates whether the BSP instantiated is of a type based on Intel FPGA HAL.

**is_bsp_lwhal_extension**

Usage

```bash
is_bsp_lwhal_extension
```

Description

Returns a boolean value that indicates whether the BSP instantiated is of a type based on Intel FPGA Lightweight HAL.

**open_bsp**

Usage

```bash
open_bsp <settingsFile>
```

Options

- `<settingsFile>`: .bsp settings file to open.

Description

Opens an existing BSP.

**save_bsp**

Usage

```bash
save_bsp <settingsFile>
```

Options

- `<settingsFile>`: .bsp settings file to save BSP to.
**set_bsp_version**

**Description**
Saves a new BSP.

**set_bsp_version**

**Usage**

```bash
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.

**set_logging_mode**

**Usage**

```bash
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.

**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 15-7: Callback Properties

<table>
<thead>
<tr>
<th>Property as specified in set_sw_property</th>
<th>Run time</th>
<th>Scope</th>
<th>Callback Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialization_callback</td>
<td>Initialization</td>
<td>Component instance</td>
<td>Component instance name</td>
</tr>
<tr>
<td>validation_callback</td>
<td>Validation</td>
<td>Component instance</td>
<td>Component instance name</td>
</tr>
<tr>
<td>generation_callback</td>
<td>Generation</td>
<td>Component instance</td>
<td>Component instance name, BSP generate target directory, Driver BSP subdirectory</td>
</tr>
<tr>
<td>class_initialization_callback</td>
<td>Initialization</td>
<td>Component class</td>
<td>Driver class name</td>
</tr>
<tr>
<td>class_validation_callback</td>
<td>Validation</td>
<td>Component class</td>
<td>Driver class name</td>
</tr>
<tr>
<td>class_generation_callback</td>
<td>Generation</td>
<td>Component class</td>
<td>Driver class name, BSP generate target directory, Driver BSP subdirectory</td>
</tr>
</tbody>
</table>

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.

### Related Information

**Tcl Commands for BSP Settings** on page 15-64

---

(15) The BSP subdirectory into which the driver or package files are copied
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

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";

add_module_sw_property

Usage

add_module_sw_property <property-name> <property-value>
add_module_sw_setting

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
```

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
```

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
**add_systemh_line**

**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";
```

**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
```

**get_classPeripheral**

**Usage**

```
get_classPeripheral <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_classPeripheral eic_1 $irq_2;
```
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

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

get_module_peripheral

Usage

get_module_peripheral <irq-number>
get_module_sw_setting_value

Options
• `<setting-name>`: Name of the module software setting to retrieve the value for, as defined by the `add_module_sw_setting` command.

Usage
get_module_sw_setting_value `<setting-name>`

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:

```tcl
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
```tcl
puts [get_module_sw_setting_value MY_SETTING]
```

Returns:
"My setting value"

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;

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;

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;
**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'
```

**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'
```

**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.

For more information about how to manipulate settings on existing software packages and device drivers, refer to “Settings Managed by the Software Build Tools” on page 15–34.

For more information about how to manipulate settings on existing software packages and device drivers, refer to “Tcl Commands for BSP Settings” on page 15–76.

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 TCP/IP.

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.

For more information about creating Tcl scripts, refer to the “Tcl Scripts for BSP Settings” in the Software Build Tools section.

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**

- Settings Managed by the Software Build Tools on page 15-25
- Tcl Commands for BSP Settings on page 15-64
- add_sw_property on page 15-100
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 Nios II assembly language source file (.s or .S) to BSPs containing your package. nios2-bsp-generate-files copies assembly source files into a BSP and adds them to the source build list in the BSP makefile. This property is optional.

• c_source—Adds a C source file (.c) to BSPs containing your package. nios2-bsp-generate-files copies C source files into a BSP and adds them to the source build list in the BSP makefile. This property is optional.

• cpp_source—Adds a C++ source file (.cpp, .cc, or .cxx) to BSPs containing your package. nios2-bsp-generate-files copies the C++ source files into a BSP and adds them to the source build list in the BSP makefile. This property is optional.

• include_source—Adds an include file (typically .h) to BSPs containing your package. nios2-bsp-generate-files copies include files into a BSP, but does not add them to the generated makefile. This property is optional.

• include_directory—Adds a directory to the ALT_INCLUDE_DIRS variable in the BSP’s public.mk file. Adding a directory to ALT_INCLUDE_DIRS allows all source files to find include files in this directory. add_sw_property adds the path to the generated public makefile 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. nios2-bsp-generate-files 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 public.mk 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 FPGA 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 nios2-bsp-* commands, 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 public.mk 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.

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. nios2-bsp-generate-files

Software Build Tools Reference Altera Corporation

Send Feedback
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. Set or get the setting with the --set and --get command-line options of the nios2-bsp, nios2-bsp-create-settings, nios2-bsp-query-settings, and nios2-bsp-update-settings commands. You can also 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 file. 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 public.mk file. public.mk 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 makefile. The variable name must be unique in the makefile.

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 II SBT 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 ("").

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

create_driver

Usage

create_driver <name>

Options

• <name>: Name of device driver.

Description

This command creates a new device driver instance available for the SBT. 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 SBT 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.

**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 II 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 SBT 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.

**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 SBT. 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 SBT 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.
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 nios2-bsp-generate-files 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, nios2-bsp-generate-files 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 II 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 II BSP tools. It is also required for operating systems that require the Intel FPGA 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 `legacy_interrupt_api` if the device driver supports the legacy API only or `enhanced_interrupt_api` if the device driver supports the enhanced API only. Specify both using a quoted list if the device driver supports both APIs.

If you do not specify which API your device driver supports, the Nios II SBT assumes that only the legacy interrupt API is supported. The Nios II SBT analyzes this property for each driver in the system to determine the appropriate API to be used in the system.

**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 II SBT 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.

**Related Information**

**Software Build Tools** on page 4-1
For more information about the legacy and enhanced APIs, refer to "Exception Handling".

**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'
```
Software Build Tools Path Names

There are some restrictions on how you can specify file paths when working with the SBT. The tools are designed for the maximum possible compatibility with a variety of computing environments. By following the restrictions in this section, you can ensure that the build tools work smoothly with other tools in your tool chain.

Command Arguments

Many software build tool commands take file name and directory path arguments. You can provide these arguments in any of several supported cross-platform formats. The SBT supports the following path name formats:

- **Quoted Windows**—A drive letter followed by a colon, followed by directory names delimited with backslashes, surrounded by double quotes. Example of a quoted Windows absolute path: "c:\altera\72\nios2eds\examples\verilog\niosII_cyclone_1c20\standard"
  
  Quoted Windows relative paths omit the drive letter, and begin with two periods followed by a backslash. Example:
  
  ".\niosII_cyclone_1c20\standard"

- **Escaped Windows**—The same as quoted Windows, except that each backslash is replaced by a double backslash, and the double quotes are omitted. Examples:
  
  c:\\altera\\72\\nios2eds\\examples\\verilog\\niosII_cyclone_1c20\\standard
  
  ..\\niosII_cyclone_1c20\standard

- **Linux**—An optional forward slash, followed by directory names delimited with forward slashes. Examples:
  
  /altera/72/nios2eds/examples/verilog/niosII_cyclone_1c20/standard
  
  verilog/niosII_cyclone_1c20/standard
  
  Linux relative paths begin with two periods followed by a forward slash. Example:
  
  ../niosII_cyclone_1c20/standard

- **Mixed**—The same as quoted Windows, except that each backslash is replaced by a forward slash, and the double quotes are omitted. Examples:
  
  c:/altera/72/nios2eds/examples/verilog/niosII_cyclone_1c20/standard
  
  ../niosII_cyclone_1c20/standard

- **Cygwin**—An absolute Cygwin path consists of the pseudo-directory name "/cygdrive/", followed by the lower case Windows drive name, followed by directory names delimited with forward slashes. Example:
  
  /cygdrive/c/altera/72/nios2eds/examples/verilog/niosII_cyclone_1c20/standard
  
  Cygwin relative paths are the same as Linux relative paths. Example:
  
  ../niosII_cyclone_1c20/standard

The SBT accepts both relative and absolute path names.
Table 15-8: Path Name Format Support for SBT utilities and makefiles

<table>
<thead>
<tr>
<th>Context</th>
<th>Formats supported on Linux</th>
<th>Formats supported on Windows with Cygwin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Utilities and scripts</td>
<td>Linux</td>
<td>• Quoted Windows (17)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Mixed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Escaped Windows</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Cygwin</td>
</tr>
<tr>
<td>Makefiles</td>
<td>Linux</td>
<td>• Mixed (18)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Cygwin</td>
</tr>
</tbody>
</table>

Object File Directory Tree

The makefile created by the SBT creates a new directory tree for generated object files. To the extent possible, the object file directory tree retains the structure of the corresponding source directory.

For example, if you specify the path to a source file as

```
src/util/special/tools.c
```

the makefile places the corresponding object code in

```
obj/util/special/tools.o
```

The makefile does not create object directories outside the project directory root. If the source file path you specify is a relative path beginning with "..", the SBT flattens the path name prior to creating the object directory structure.

For example, if you specify the path to a source file as

```
../special/tools.c
```

the makefile places the corresponding object code in

```
obj/tools.o
```

If you specify an absolute path to source files under Cygwin, the SBT creates the `obj` directory structure as if you had used the Cygwin form of the path name. For example, if you specify the path to a source file as

```
c:/dev/app/special/tools.c
```

the SBT places the corresponding object code in

```
obj/cygdrive/c/dev/app/special/tools.o
```

Related Information

**Embedded Software Projects** on page 4-4
For more information about the object file directory structure.

---

(16) These rules apply to any Unix-like platform.
(17) These rules apply to other Unix-like shells running on Windows. The Command Shell, provided with the EDS, is based on Cygwin. Examples in this chapter are designed for the Command Shell.
(18) The build tools automatically convert path names to Cygwin format.