



Nios II DPX Datapath Processor

Handbook



101 Innovation Drive
San Jose, CA 95134
www.altera.com

NIIDPXHB-1.0

Document last updated for Altera Complete Design Suite version:
Document publication date:

11.0
May 2011



[Subscribe](#)

© 2011 Altera Corporation. All rights reserved. ALTERA, ARRIA, CYCLONE, HARDCOPY, MAX, MEGACORE, NIOS, QUARTUS and STRATIX are Reg. U.S. Pat. & Tm. Off. and/or trademarks of Altera Corporation in the U.S. and other countries. All other trademarks and service marks are the property of their respective holders as described at www.altera.com/common/legal.html. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.



Section I. Nios II DPX Hardware Reference

Chapter 1. Nios II DPX Architecture

Reading Prerequisites	1-1
Installation Directory Structure	1-2
Functional Blocks	1-3
Multithreaded Processor	1-4
Message Interface Unit	1-10
Reset Controller	1-14
Debug Unit	1-15
Debug Statistics Collector	1-16
Nios II DPX Clock Domains	1-18
Nios II DPX Processor Reset Signals	1-19
Reset Sequence with Debugger	1-20
Reset Considerations when Designing Outside of Qsys	1-21
Nios II DPX Processor Interfaces	1-21
Interfaces	1-21
Memory Addressing and Byte Order	1-37
Nios II DPX Datapath Processor Dual-Core Configuration	1-41
Loading Nios II DPX Software in a Deployed System	1-42

Chapter 2. Instantiating the Nios II DPX Datapath Processor

Instantiating for a Qsys System	2-1
Parameter Settings	2-1
Instantiating for a Stand-Alone System	2-10
Nios II DPX Context Address Adapter	2-10

Chapter 3. System Verification

RTL Simulation	3-1
Simulation Model, Testbench and Initialization Files	3-1
Create a Simulation Script for ModelSim	3-2
Record Suitable Waveforms	3-2
Performance Monitoring	3-2
Packet Debug	3-3
Debug Flag Bit	3-3
PEs and the Debug Flag	3-4
Using Debug Flag Breakpoint Capability	3-4
Hardware PE Debug	3-6

Additional Information

Document Revision History	Info-1
How to Contact Altera	Info-1
Typographic Conventions	Info-1

Section II. Nios II DPX Software Development

Chapter 4. Overview of the Nios II DPX MTP

The MTP in the Context of the Nios II DPX Datapath Processor	4-1
Event-Driven Processing	4-1
Nios II DPX Multithreading	4-2
Dual-Processor Configurations	4-2
Nios II DPX Programming Considerations	4-3
Memory and I/O	4-3
The Nios II DPX Debug Interface	4-3
Exception Controller	4-4
The Nios II DPX Software Development Environment	4-4
The Nios II SBT Development Flow	4-4
Nios II DPX Programs	4-5
Finding Nios II EDS Files	4-7

Chapter 5. Software Programming Model

Overview of the Nios II DPX MTP	5-1
The Event-Driven Programming Model	5-1
Tasks, PE Messages and Events	5-1
Context Data	5-4
Nios II DPX Registers	5-4
General-purpose Registers	5-4
Extension Registers	5-5
Control Registers	5-6
Extended Control Registers	5-7
Developing Software Tasks for the Datapath Processor	5-10
The Nios II DPX Task ID	5-10
Sending PE Messages Between Tasks	5-11
Writing Task Code	5-11
The Null Task ID	5-14
Resource Sharing	5-14
Task-Related Instructions	5-14
Context Management	5-16
Creating a Context	5-16
Maintaining the CID Free List	5-17
Data Ordering with the DPX Datapath Processor	5-17
Sequence Number Reordering	5-18
CID Ordering	5-18
Using the Nios II DPX Extension Registers	5-19
Accessing Extension Registers	5-20
Nios II DPX Memory Model	5-20
Physical Memory Access	5-20
Memory Organization	5-21
Advanced Topics	5-23
Sending Multiple PE Messages	5-23
Spawning a New Task	5-23
Avoiding System Deadlock	5-24
Exception Processing	5-26
Reset Exceptions	5-27
Break Exceptions	5-28
Instruction-Related Exceptions	5-28
Instruction Set Categories	5-30
Data Transfer Instructions	5-30
Bit Manipulation Instructions	5-31
Arithmetic Instructions	5-31
Move Instructions	5-32

Comparison Instructions	5-32
Shift and Rotate Instructions	5-33
Message Passing Instructions	5-33
Program Control Instructions	5-34
Thread Control Instructions	5-35
Other Control Instructions	5-35
No-operation Instruction	5-35
Potential Unimplemented Instructions	5-35

Chapter 6. Getting Started with the Graphical User Interface

Introduction to the Nios II DPX Debugging Environment	6-1
Getting Started	6-2
The Nios II SBT for Eclipse Workbench	6-2
Creating a Project	6-3
Navigating the Project	6-5
Building the Project	6-6
Configuring the FPGA	6-6
Debug Setup	6-6
Debugging the Project	6-9
Working with Stand-Alone Systems	6-14
Running a Nios II DPX System with ModelSim	6-16
Makefiles and the Nios II SBT for Eclipse	6-17
Eclipse Source Management	6-17
User Source Management	6-19
BSP Source Management	6-20
Using the BSP Editor	6-20
Tcl Scripting and the Nios II BSP Editor	6-20
Starting the Nios II BSP Editor	6-20
The Nios II BSP Editor Screen Layout	6-21
The Command Area	6-21
The Console Area	6-25
Exporting a Tcl Script	6-26
Creating a New BSP	6-26
BSP Validation Errors	6-27
Configuring Component Search Paths	6-27
Importing a Command-Line Project	6-28
Road Map	6-28
Import a Command-Line C Application	6-29
Import a Supporting Project	6-29
User-Managed Source Files	6-30
Packaging a Library for Reuse	6-30
Creating the User Library	6-30
Using the Library	6-31
Memory Initialization Files	6-31
Managing Toolchains in Eclipse	6-32
Eclipse Usage Notes	6-32
Thread-Specific Breakpoints	6-32
DSF Disassembly View Required	6-33
Configuring Application and Library Properties	6-33
Configuring BSP Properties	6-33
Exclude from Build Not Supported	6-33
Selecting the Correct Launch Configuration Type	6-34
Renaming Nios II DPX MTP Projects	6-34
Running Shell Scripts from the SBT for Eclipse	6-34

Must Use Nios II Build Configuration	6–35
CDT Limitations	6–35

Chapter 7. Getting Started from the Command Line

Advantages of the Command Line	7–1
Outline of the Nios II SBT Command-Line Interface	7–1
Utilities	7–2
The nios2-bsp Script	7–2
Tcl Commands	7–2
Tcl Scripts	7–2
The Nios II Command Shell	7–2
Scripting Basics	7–2
Creating a BSP with a Script	7–3
Creating an Application Project with a Script	7–4
Running make	7–5
Creating Memory Initialization Files	7–5

Chapter 8. Understanding the Nios II DPX Board Support Package

Nios II DPX Software Development Tools	8–1
The Nios II DPX GNU Toolchain	8–1
newlib for the Nios II DPX MTP	8–2
Using the Nios II Software Build Tools	8–2
The Lightweight Hardware Abstraction Layer (LWHAL)	8–2
Startup Code	8–3
Stack	8–3
Device Drivers	8–3
Differences from newlib	8–4
Software Tasks	8–5
Minimal Character-Mode API	8–6
Managing Memory Usage with the LWHAL	8–7
Custom Device Drivers for the LWHAL	8–7
Exception Handling	8–8
Break Handler	8–9
Nios II DPX BSP Creation	8–9
LWHAL BSP Files and Folders	8–9
Linker Map Validation	8–14
Specifying BSP Defaults for the Nios II DPX MTP	8–14
Top Level Tcl Script for BSP Defaults	8–15
Specifying the Default stdio Device	8–16
Specifying the Default Memory Map	8–16
Using Individual Default Tcl Procedures	8–17
Hardware Requirements	8–17
Lightweight HAL Function Reference	8–17
LWHAL Function Macros	8–18
LWHAL Functions	8–19
LWHAL Extended Instruction Macros	8–20
LWHAL Driver Functions	8–22
Lightweight HAL Standard Types	8–24
Creating a BSP for a Stand-Alone System	8–25
Creating a BSP from the Command Line	8–25
Creating a BSP with the BSP Editor	8–25

Chapter 9. Nios II DPX MTP Instruction Set and Application Binary Interface

The Nios II DPX MTP Instruction Set	9-1
Instruction Formats	9-1
Instruction Encodings	9-5
Assembler Pseudo-Instructions	9-9
Assembler Macros	9-10
Nios II DPX MTP Instruction Set Reference	9-11
Nios II DPX Extended Instruction Set Reference	9-104
The Nios II DPX MTP Application Binary Interface	9-112
Data Types	9-112
Memory Alignment	9-112
Register Usage	9-112
Stacks	9-114
Arguments and Return Values	9-119
DWARF-2 Definition	9-121
Object Files	9-121
Relocation	9-121
Development Environment	9-123

Chapter 10. SBT Reference for the Nios II DPX MTP

Nios II Software Build Tools Utilities	10-1
Logging Levels	10-2
Setting Values	10-2
Utility and Script Summary	10-3
nios2-app-generate-makefile	10-4
nios2-bsp-create-settings	10-6
nios2-bsp-generate-files	10-8
nios2-bsp-query-settings	10-9
nios2-bsp-update-settings	10-11
nios2-lib-generate-makefile	10-13
nios2-bsp-editor	10-15
nios2-app-update-makefile	10-16
nios2-lib-update-makefile	10-19
nios2-swexample-create	10-22
nios2-elf-insert	10-23
nios2-elf-query	10-24
nios2-bsp	10-25
nios2-bsp-console	10-27
Settings	10-28
Overview of BSP Settings	10-28
Overview of Component and Driver Settings	10-29
Settings Reference	10-30
Application and User Library Makefile Variables	10-39
Application Makefile Variables	10-39
User Library Makefile Variables	10-41
Standard Build Flag Variables	10-42
Tcl Commands	10-42
Tcl Command Environments	10-42
Tcl Commands for BSP Settings	10-42
Tcl Commands for BSP Generation Callbacks	10-69
Tcl Commands for Drivers and Packages	10-77
Path Names	10-84
Command Arguments	10-85
Object File Directory Tree	10-86

Additional Information

Document Revision History	Info-1
How to Contact Altera	Info-1
Typographic Conventions	Info-1

The *Nios® II DPX Hardware Reference* section of the *Nios II DPX Datapath Processor Handbook* provides information about Nios II DPX hardware topics.

This section includes the following chapters:

- Chapter 1, Nios II DPX Architecture
- Chapter 2, Instantiating the Nios II DPX Datapath Processor
- Chapter 3, System Verification



101 Innovation Drive
San Jose, CA 95134
www.altera.com

© 2011 Altera Corporation. All rights reserved. ALTERA, ARRIA, CYCLONE, HARDCOPY, MAX, MEGACORE, NIOS, QUARTUS and STRATIX are Reg. U.S. Pat. & Tm. Off. and/or trademarks of Altera Corporation in the U.S. and other countries. All other trademarks and service marks are the property of their respective holders as described at www.altera.com/common/legal.html. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.



The Nios II DPX datapath processor is a multithreaded, programmable, soft processor core optimized for control, scheduling, and processing of datapaths. The processor allows parallel processing of multiple blocks of data while maintaining the context of each data block.

The Nios II DPX datapath processor is optimized for datapath applications, such as packet processing, by conserving resources and eliminating sources of nondeterministic behavior that impair real-time performance. The processor is a processing element (PE) within an Altera® event-driven datapath processing system in Altera FPGAs.

 For information about Altera event-driven datapath processing, refer to the *Altera Event-Driven Datapath Processing Design Handbook*.

The Nios II DPX datapath processor offers the following features:

- Highly-predictable pipeline suitable for real-time applications
- Event-driven architecture with hardware support for efficient use of hardware acceleration
- Extended Nios II instruction set architecture (ISA), allowing use of the Nios II embedded design suite
- Optimized hardware with high f_{MAX} and low resource usage

Reading Prerequisites

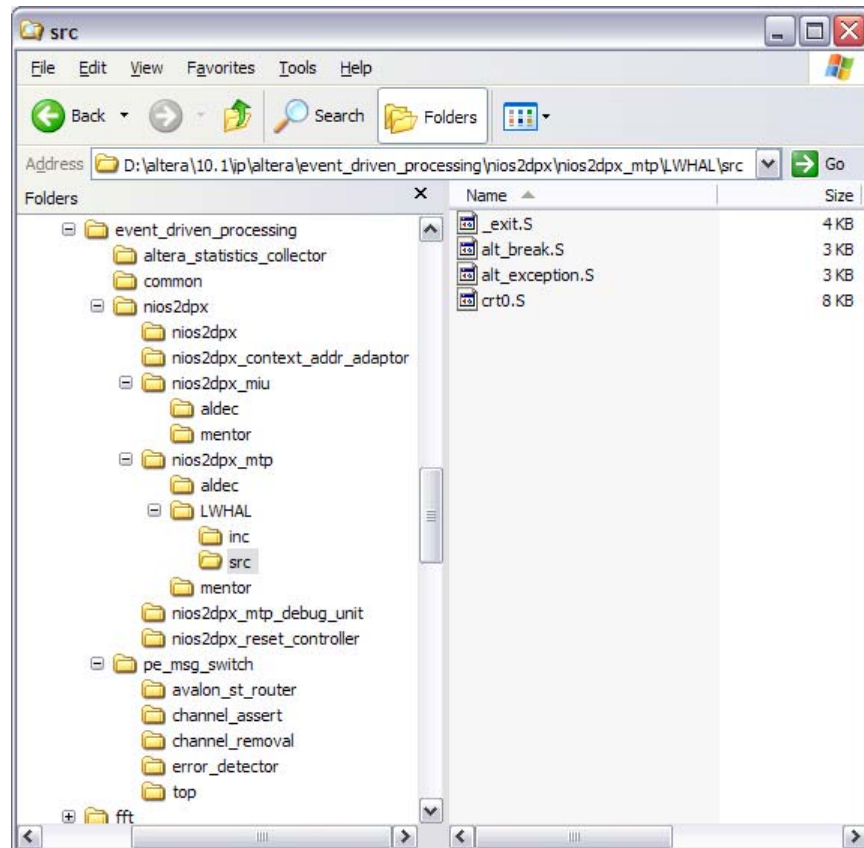
This handbook assumes you are familiar with the terminology and concepts (such as event-driven processing, contexts, tasks, and messages) presented in the *Altera Event-Driven Datapath Processing Design Handbook*.

An understanding of the Avalon Streaming (Avalon-ST) and Avalon Memory-Mapped (Avalon-MM) interfaces, presented in the *Avalon Interface Specifications*, is also helpful.

Installation Directory Structure

Figure 1–1 shows the directory structure for the Nios II DPX datapath processor cores and microcores. The path to the folders shown is *<Quartus II installation directory>/ip/altera/event_driven_processing*.

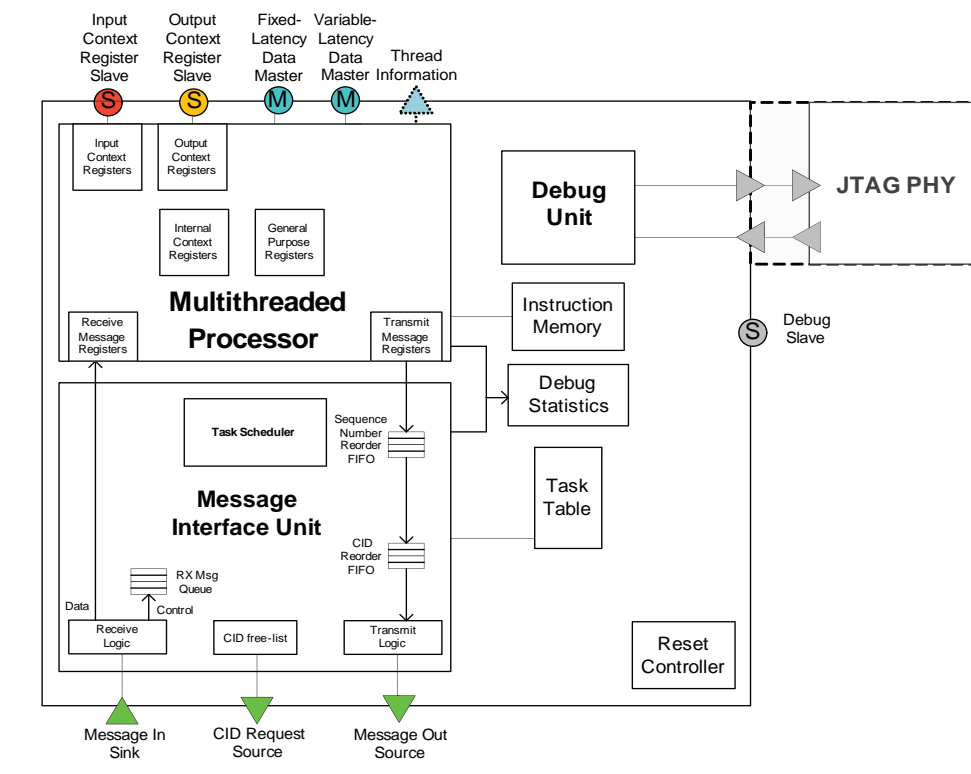
Figure 1–1. Nios II DPX Datapath Processor Installation Directories



Functional Blocks


Figure 1-2 shows a simplified Nios II DPX datapath processor block diagram.

Figure 1-2. Simplified Nios II DPX Datapath Processor Block Diagram



The Nios II DPX datapath processor has the following key functional blocks:

- **Multithreaded processor (MTP)**—A hardware-multithreaded processor based on the Nios II instruction set architecture. The MTP instructions are stored in an instruction memory that is contained within the Nios II DPX datapath processor.
- **Message interface unit (MIU)**—The MIU sends, receives, and manages messages using Avalon-ST PE message interfaces. The MIU translates received messages into work scheduled onto the MTP threads in the form of software tasks. The MIU also allows the MTP to delegate work to other PEs by managing and sending Avalon-ST PE messages.
- **Debug unit**—The debug unit manages standard processor-level debug features, such as loading executable software, setting breakpoints, single stepping, and accessing registers for software tasks executing on the MTP.
- **Statistics collector**—The statistics collector provides access to various statistics gathered by the MIU and MTP at run time. The monitoring done to obtain these statistics is performed non-intrusively and is constantly updated at run time, facilitating run-time system-monitoring and system-debug capability.
- **Reset controller**—The reset controller controls system reset requests and internal processor core soft resets.

 The Nios II DPX datapath processor is highly configurable. For configuration options, refer to [Chapter 2, Instantiating the Nios II DPX Datapath Processor](#).

The following sections describe the functional blocks of the Nios II DPX datapath processor.

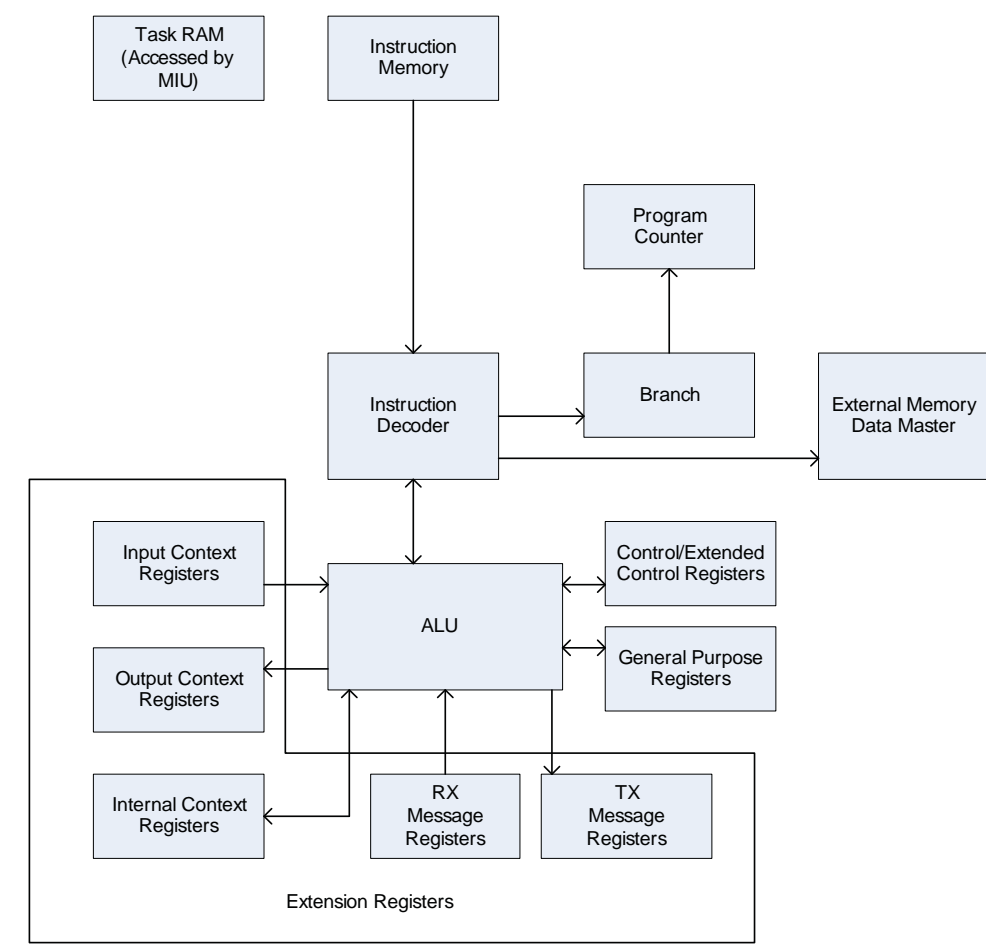
Multithreaded Processor

The MTP is the processing core within the Nios II DPX processor. The MTP includes the following features:

- Hardware multithreading, with eight hardware threads
- Big-endian processor
- Highly-predictable real-time operation, without instruction cache, data cache, or external interrupts
- Single and dual MTP core configurations

[Figure 1-3](#) shows the MTP functional blocks.

Figure 1-3. Nios II DPX MTP Block Diagram



The following sections describe the functional blocks of the MTP.

ALU

The arithmetic logic unit (ALU) executes the following types of instructions:

- Arithmetic
- Logic
- Shift
- Control operation



For a complete list and description of instructions, refer to the *Nios II DPX MTP Instruction Set and Application Binary Interface* chapter in the *Nios II DPX Software Development* section of the *Nios II DPX Datapath Processor Handbook*.

Register File

The Nios II DPX processor register file includes the following types of registers:

- General-purpose registers—Serve as temporary registers for the software task program.
- Extension registers—Hold message data arguments and context-specific data for software running on the MTP.
- Control registers—Provide thread-specific information to the software tasks.
- Extended control registers—Hold message control information for software tasks.

The following sections give a general description of each type of register.



For additional information about these registers, refer to the *Programming Model* chapter in the *Nios II DPX Software Development* section of the *Nios II DPX Datapath Processor Handbook*.

General Purpose Registers

The MTP contains eight general-purpose register banks, each containing 32 general-purpose registers named r0 through r31. Each bank is associated with a thread. The processor switches general-purpose register banks automatically, transparent to the software executed by a thread. When a software task executing on a thread completes and another software task starts on the same thread, the contents of the general-purpose registers are undefined.

Extension Registers

The MTP contains a user-defined number of extension register banks. Each bank consists of a configurable number of registers assigned to different purposes. The maximum number of extension registers in each bank is 32, namely r32 through r63. The processor switches extension register banks automatically, transparent to the software executed by a thread. Contents of extension register banks remain intact throughout the life of the context or message.

The MTP threads have access to the following extension registers:

- RX message registers
- TX message registers
- Internal context registers

- **Input context registers**
- **Output context registers**

The number of each kind of extension register is configurable. The Nios II DPX processor supports the extension register configurations shown in [Table 2-2 on page 2-2](#).

The input and output context registers share the same register numbers. This sharing is possible because the Nios II DPX toolchain uses the fact that the input context registers are read-only registers and the output context registers are write-only registers to point read and write instructions to the correct register bank. The same scenario is also true for the RX and TX message registers.



Keep [Table 2-2 on page 2-2](#) in mind when configuring your message format. The maximum number of data arguments that the Nios II DPX datapath processor can send or receive in an Avalon-ST PE message is determined by the number of RX and TX message registers. For more information about the message format, refer to the *Message Format* chapter of the [Altera Event-Driven Datapath Processing Design Handbook](#).

The following sections describe the types of extension registers.

RX Message Registers

RX message registers are banks of 32-bit registers used to pass message data arguments from the MIU to software tasks running in the MTP. You specify the number of banks of RX message registers in your processor with the **Number of receive IDs** parameter when instantiating the Nios II DPX processor. For more information, refer to [“Message Interface Unit Tab” on page 2-4](#).

Each bank of RX message registers corresponds to an RXID. When a thread accesses an RX message register, the correct RX message register bank is implicitly indexed by the RXID currently associated with the thread. RX message registers are read-only. Reading them from a given task provides access to the arguments associated with the message that started that task. The registers remain valid for the duration of the software task unless an `rxfree` instruction releases the associated RXID. For more information, refer to [“RX Message Flow” on page 1-12](#).

TX Message Registers

TX message registers are banks of 32-bit registers used to pass message data arguments from software tasks running in the MTP to the MIU. You specify the number of banks of TX message registers in your processor with the **Number of transmit IDs** parameter when instantiating the Nios II DPX processor. For more information, refer to [“Message Interface Unit Tab” on page 2-4](#).

Each bank of TX message registers corresponds to an TXID. When a thread accesses an TX message register, the correct TX message register bank is implicitly indexed by the TXID currently associated with the thread. TX message registers are write-only. Writing them from a given task stores the arguments associated with the next message sent by that task. For more information, refer to [“TX Message Flow” on page 1-12](#).

Internal Context Registers

Internal context registers are banks of 32-bit registers local to the MTP and serve as context-specific persistent storage. You specify the number of banks of internal context registers in your processor with the **Number of context IDs** parameter when instantiating the Nios II DPX processor. For more information, refer to [“Message Interface Unit Tab” on page 2-4](#).

Each bank of context registers corresponds to a context ID (CID). When a thread accesses a context register, the correct context register bank is implicitly indexed by the CID currently associated with the thread. This applies to internal, input, and output context registers.

Software tasks can access only the internal context register bank that corresponds to the CID associated with the running task. All tasks processing messages with the same CID have access to the same bank of internal context registers.



There is also an advanced option that uses the number of threads to determine the number of banks of internal context registers. For more information, refer to the **Context register indexing mode** parameter in [“Advanced Options Tab” on page 2-9](#). When thread ID is used, all software tasks running on the same thread have access to the same bank of internal context registers.

Input Context Registers

Input context registers are banks of 32-bit registers used to pass context data into the Nios II DPX processor. You specify the number of banks of input context registers in your processor with the **Number of context IDs** parameter when instantiating the Nios II DPX processor. For more information, refer to [“Message Interface Unit Tab” on page 2-4](#).

Software tasks can access only the input context register bank that corresponds to the CID associated with the running task, and the access is read-only. The MTP uses an Avalon-MM slave interface to receive input context register data. For more information, refer to [“Input Context Register Interface” on page 1-25](#).

Output Context Registers

Output context registers are banks of 32-bit registers used to pass context data out of the Nios II DPX processor. You specify the number of banks of output context registers in your processor with the **Number of context IDs** parameter when instantiating the Nios II DPX processor. For more information, refer to [“Message Interface Unit Tab” on page 2-4](#).

Software tasks can access only the output context register bank that corresponds to the CID associated with the running task, and the access is write-only. The MTP uses an Avalon-MM slave interface to send output context register data. For more information, refer to [“Output Context Register Interface” on page 1-29](#).

Control Registers

The MTP has two control registers available to software tasks, `cpuid` and `threadnum`.



For more information, refer to the *Programming Model* chapter in the [Nios II DPX Software Development](#) section of the *Nios II DPX Datapath Processor Handbook*.

Extended Control Registers

The Nios II DPX uses extended control registers to inspect control information in received messages and modify control information in transmitted messages. Extended control registers can be per-processor or per-thread registers.



For more information, refer to the *Programming Model* chapter in the *Nios II DPX Software Development* section of the *Nios II DPX Datapath Processor Handbook*.

Memory

This section describes the memory within the Nios II DPX processor and the external memory dependencies.

Instruction Memory

Software instructions run from internal on-chip instruction memory. This instruction memory has a fixed access time of two cycles. The application code can be preloaded into the instruction memory, or loaded into the processor from the debugger or using the debug access slave. The instruction memory is read by the MTP and read and written by external modules such as the Nios II DPX debugger.

Data Memory

Each Nios II DPX MTP exports an Avalon-MM fixed-latency data master and an optional Avalon-MM variable-latency data master interface, for attaching data memory used by load and store instructions.



For a dual-core Nios II DPX datapath processor, there are two fixed-latency and optionally two variable-latency data masters, one per MTP. For more information, refer to “*Nios II DPX Datapath Processor Dual-Core Configuration*” on page 1-41.

Memory attached to the Avalon-MM fixed-latency data master must have read latency of two cycles and zero wait-state reads and writes. Memory connected to the Avalon-MM variable-latency data master must assert appropriate Avalon-MM control signals for data transfers.



Loads and stores performed to variable-latency memory can result in thread stalls. For more information, refer to “*Threading Model*” on page 1-9.

The fixed-latency master must have exclusive access to any connected slaves. Because memory mapped systems where multiple masters connect to a single slave require arbitration, and arbitration might apply backpressure to one of the masters, the variable-latency data master must be used if a slave is shared between masters.

The fixed-latency data master has an associated address range that you configure when instantiating the Nios II DPX datapath processor. Loads and stores within range are initiated over the fixed-latency data master; loads and stores not in this range are initiated over the variable-latency data master if enabled.

You can partition memory with the optional thread information interface described in “*Thread Information Interface*” on page 1-35. For example, you can partition memory to automatically index into the data memory based on the current thread number.

Task Address Table

The task address table is internal on-chip memory in the Nios II DPX processor that contains the starting addresses of each software task in instruction memory. The MIU uses the task address table similar to an interrupt vector table to look up the software task starting address associated with each received message. The table can be loaded when the device is programmed as part of the SRAM Object File (.sof), or loaded by the host or debugger before releasing the Nios II MTP from reset. External modules can write to, and read from, the task address table.

Interrupts and Exceptions

The Nios II DPX processor does not support external interrupts, and only excepts a limited set of exceptions. The processor can generate the following exceptions:

- Illegal instruction exception
- Unsupported instruction exception
- Debug exception

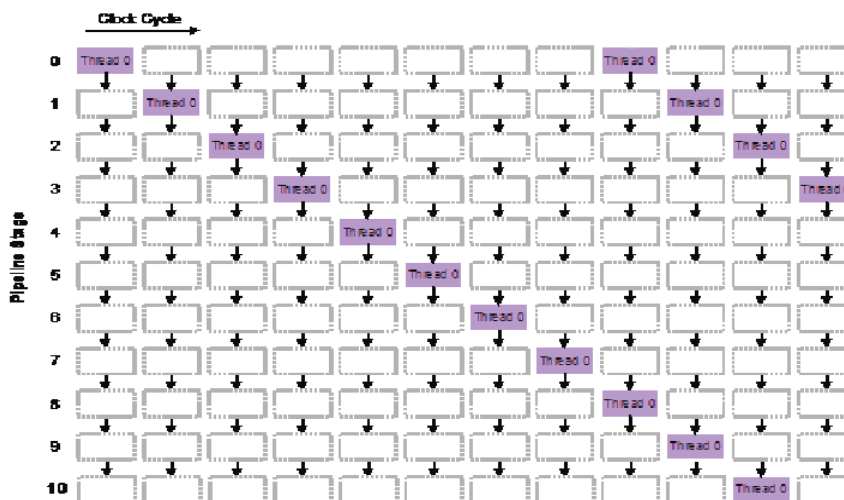
For a complete list and description of possible exceptions, refer to “Exception Processing” in the *Software Programming Model* chapter of the *Nios II DPX Software Development* section of the *Nios II DPX Datapath Processor Handbook*.

Threading Model

The MTP implements hardware threads that essentially appear to the software as multiple independent processors. The type of threading model is sometimes called barrel multithreading or interleaved multithreading. The MTP has eight hardware threads. Each thread has its own bank of general-purpose registers and program counter (PC). From a software perspective, each thread is its own processor that operates independently.

Figure 1-4 illustrates a single thread running on the MTP as it progresses through the MTP pipeline.

Figure 1-4. Single Thread Running on Eight-Thread MTP

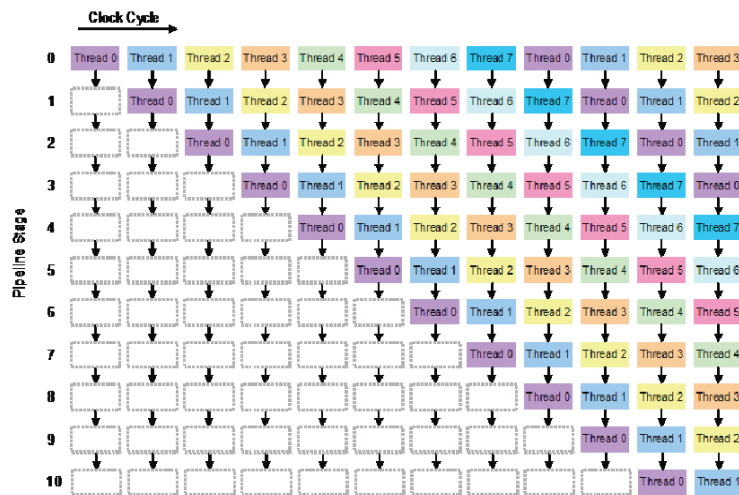


An instruction moves through the processor's pipeline advancing by one pipeline stage each clock cycle. However, unlike a traditional processor, the next instruction in the program does not follow directly afterward. Instead, the next instruction enters the pipeline eight clock cycles later. The eight-cycle gap eliminates the need for pipeline-stalling logic, eliminates the need for forwarding-control logic, and allows the processor to run at a higher frequency than would otherwise be possible.

By providing each thread its own independent context of execution, including general-purpose registers and program counter, the eight-cycle gap per thread allows each pipeline stage to process instructions from multiple threads.

Figure 1-5 illustrates eight threads running concurrently on an MTP core. The eight threads (essentially separate software programs), share the processor's pipeline and appear to run concurrently.

Figure 1-5. Eight Threads Running on Eight-Thread MTP



Instructions for a given thread are only issued every eight clock cycles, and eight threads share the pipeline. Thus, if the Nios II DPX processor clock is 400 MHz, the MTP can support eight threads, each thread effectively running at 50 MHz.

This threading technique means that the MTP has no stalls, other than load or store instructions accessing variable-latency data memory, and has no hazards or need for a branch predictor. All instructions are executed within a single thread cycle (eight clock cycles), except for load and store instructions that access variable-latency data memory. If a load or store instruction stalls, the stalled instruction is revisited eight clock cycles later and in the meantime the other seven threads operate normally. A stalled instruction in one thread does not impact execution of the other threads.

Message Interface Unit

The main functions of the MIU are:

- Receive and manage buffering of incoming messages
- Schedule threads on the MTP to execute tasks requested by the messages

- Transmit messages generated by the tasks executing on the MTP, applying flow control as requested by the MTP tasks
- Manage and allocate CIDs requested by external entities

The following sections describe the functional aspects of the MIU from a message-flow perspective. You might find [Figure 1-2 on page 1-3](#) helpful to understand the MIU functionality.

Register Usage

When the MIU receives a message, the MIU assigns an available bank of RX message registers, an available bank of TX message registers, and the associated RXID and TXID identifiers to the message. Software tasks read data arguments for the incoming message from the RX message registers, and store data arguments for the task's outgoing message in the TX message registers. Tasks that send more than one message can also request a bank of TX message registers for each message sent.

The RX message registers and TX message registers appear as extension registers to the software, as described in ["Extension Registers" on page 1-5](#). The RXID associated with a message is freed when a task processing the received message executes an `exit` or `rxfree` instruction. The TXID becomes unavailable to the task creating the sent message when the task executes a `snd` or `sndi` instruction.

Once freed by the MIU, the RXID, TXID, and the associated message register banks can be used for storing new message data.



Each task must send a message (execute a `snd` or `sndi` instruction). Failure to do so results in the TXID assigned to that task never being freed.

When CIDs are used, the task has access to three additional banks of extension registers, namely, input context registers, output context registers, and internal context registers. The CID, contained within a message's control information, is used to select these banks of registers. External PEs load context specific information into the input context registers for software tasks to read using the ["Input Context Register Interface" on page 1-25](#). The software task loads context specific information in the output context registers, which can be read by external PEs using the ["Output Context Register Interface" on page 1-29](#).

CID Usage

The MIU maintains a list of available CIDs. You configure the number of contexts using the **Number of context IDs** parameter when instantiating the Nios II DPX datapath processor.



For information about contexts, refer to the *Introduction to Altera Event-Driven Datapath Processing* chapter of the [Altera Event-Driven Datapath Processing Design Handbook](#).

Typically when new input data enters the system, the input PE requests a CID from the Nios II DPX processor using the “[CID Request Interface](#)” on page 1-24, which sets up a context for the data with the CID granted by the processor. The CID is associated with the input data and stays associated through message passing for the life of the data's processing. The MIU uses the CID contained within a message to schedule processing of the context data by software tasks on the MTP.

Once all messages pertaining to a context are processed, a software task must free the CID for reuse.

If a software task needs to send more than one message, the task can request additional CIDs using the `cidalloc` instruction. You enable additional CID allocation capability with the **Context ID allocation support** parameter when instantiating the Nios II DPX datapath processor. For more information, refer to “[Message Interface Unit Tab](#)” on page 2-4.

RX Message Flow

When the MIU receives a message, the MIU copies the data arguments from the message into the RX message registers so the data is available to the software task. The MIU then copies the control information from the message to the RX message queue for scheduling. This control information contains a task ID which is used to schedule the appropriate software task to process the message.

The MIU uses an Avalon-ST PE message sink interface to receive Avalon-ST PE messages. For more information, refer to “[RX Message Interface](#)” on page 1-22.

Task Address Table and Thread Scheduling

The MIU uses a block of memory that contains a task address table. The table maps the task ID values carried in incoming messages to starting addresses of software tasks that execute on the MTP threads. The task address table can be pre-initialized when the device is programmed, or uploaded through the debugger or the debug access slave.

On arrival of a message in the RX message queue, the MIU looks up the software task's starting address in the task address table using the task ID from the message, and looks for an idle thread. On detecting an idle thread, the MIU passes the starting address of the software task routine to the PC of the idle thread, and the MTP thread executes the software task requested in the incoming message. On completion, the hardware thread goes to an idle state and is again available to be scheduled to execute a new software task.

TX Message Flow

Software tasks executing on the MTP initiate message sending. The software task supplies the following message control information to the MIU:

- The ID of the destination PE
- The ID of the task to be performed by the destination PE
- The number of data arguments loaded in the TX message registers
- Options controlling the flow of the message, used internally by the Nios II DPX processor

On receiving instruction from the MTP, the MIU places the message's control information for transmission in a transmit queue. From the transmit queue, control flows in one or more of the following ways:

- To the CID message reorder buffer in the Nios II DPX processor, described in [“Context ID Ordering” on page 1-13](#).
- To the sequence number message reorder buffer in the Nios II DPX processor, described in [“Sequence Number Ordering” on page 1-14](#).
- Out of the Nios II DPX processor using the TX message interface, described in [“TX Message Interface” on page 1-23](#).
- To a null message, which does nothing.

Which route the flow takes is determined by parameters you select when you instantiate the processor, and the options the software task specifies when initiating the message with the `snd` or `sndi` instruction. For more information, refer to [“TX Message Ordering”](#).

The MIU uses an Avalon-ST PE message source interface to send Avalon-ST PE messages from the Nios II DPX processor. For more information, refer to [“TX Message Interface” on page 1-23](#).

TX Message Ordering

The Nios II DPX processor offers the two optional methods for maintaining message order. When ordering is needed, the Nios II DPX can order messages using either or both of the following methods:

- Context ID ordering—Typically controls data flowing out of the entire system
- Sequence number ordering—Controls data flowing out of the processor

Context ID Ordering

Because Altera event-driven datapath processing allows for parallel processing of contexts to improve efficiency, it is possible for a context initiated at a later time to finish prior to a context initiated earlier. The optional CID ordering enforcement ensures that the output of the system is maintained in strict order, based on the CID. You enable CID ordering with the **Context ID ordering enforcement** parameter when instantiating the Nios II DPX processor. For more information, refer to [“Message Interface Unit Tab” on page 2-4](#).

When CID ordering enforcement is enabled, software tasks specify which messages are CID ordered. Normally, CID ordering is applied to only one message per context, typically the message going to the output PE, and thus maintains the order of the contexts going in and out of the entire system.



For more information about software task sending options, refer to the `snd` instruction in the *MTP Instruction Set and Application Binary Interface* chapter in the [Nios II DPX Software Development](#) section of the *Nios II DPX Datapath Processor Handbook*.

The MIU moves a message flagged for CID ordering from the transmit queue to the CID message reorder buffer. The message remains in the system and is not passed to the output PE until all transmit messages with earlier-issued CIDs are processed.

Sequence Number Ordering

Because the Nios II datapath processor allows for parallel execution of software tasks to improve efficiency, it is possible for messages coming into the processor at a later time to finish processing prior to messages that entered earlier. The optional sequence number ordering ensures that all messages generated by the processor leave in the same order that their corresponding messages arrived. You enable sequence number ordering with the **Sequence number ordering enforcement** parameter when instantiating the Nios II DPX processor. For more information, refer to “[Message Interface Unit Tab](#)” on page 2-4.

When sequence number ordering is enabled, the MIU assigns a sequence number to each message received. When the MTP initiates an outgoing message, the MIU moves the message into the sequence number message reorder buffer. The message remains in the buffer until all transmit messages with earlier-issued sequence numbers are processed, at which time the MIU sends the message out of the Nios II DPX processor using the TX message interface, described in “[TX Message Interface](#)” on page 1-23.



Software tasks can send more than a single message. For such software tasks, you must ensure that only one message is sequence number ordered. All other messages sent by the task must bypass sequence number ordering. For more information about software task sending options, refer to the `snd` instruction in the *MTP Instruction Set and Application Binary Interface* chapter in the *Nios II DPX Software Development* section of the *Nios II DPX Datapath Processor Handbook*.

Reset Controller

The Nios II DPX processor contains a reset controller that coordinates system reset requests and an internal processor soft reset. These resets are accessible through the debug access slave.

[Table 1-1](#) lists the memory map for the reset controller.

Table 1-1. Reset Controller Memory Map

Byte Offset	Access	Description
0x0000	Read/Write	Bit 0 controls the <code>system_reset_request_n</code> reset signal output. Set the bit to 1 to assert a system reset. Set the bit to 0 to release the system reset.
0x0004	Read/Write	Bit 0 controls the processor soft reset. Set the bit to 1 to trigger processor soft reset. Set the bit to 0 to release the processor soft reset.
0x0008	Read	Bit 0 contains the system reset state. Bit 1 contains the processor soft reset state.

The system reset request signal and processor soft reset are de-asserted at power-up. The processor soft reset must be asserted only while the system reset request signal is asserted, and remains asserted until explicitly de-asserted by the debugger or other external circuitry. For more information, refer to “[Nios II DPX Processor Reset Signals](#)” on page 1-19.

Debug Unit

The Nios II DPX datapath processor contains a debug unit that provides common debug capability such as hardware breakpoints and single stepping. It also provides high-level operations such as reading and writing to registers, instruction memory and data memory.



For more information about debug capability, refer to the *Getting Started with the Graphical User Interface* chapter in the *Nios II DPX Software Development* section of the *Nios II DPX Datapath Processor Handbook*.

Debug Unit Configuration Options

You enable or disable the debug unit, the internal JTAG PHY, and the Avalon-MM slave debug interface when instantiating the Nios II DPX processor. For more information, refer to “*Nios II DPX Datapath Processor Tab*” on page 2-1.

For designs with the debug unit omitted, the task address table, data memory, and instruction memory must be initialized as part of the .sof file or by an external entity through the Avalon-MM slave debug interface.

When instantiating multiple Nios II DPX datapath processors, it might be desirable to omit the internal JTAG PHY. As an alternative, connect an external JTAG PHY that is shared by the multiple Nios II DPX datapath processor instances.

Debug Interface Memory Map

Enabling the Avalon-MM slave debug interface, allows an external Avalon-MM master to access the instruction RAM, task address table, statistics information, and reset controller register. The debug access slave presents 23 bits of address with a span of 0x00500000 bits. You expose the interface with the **Enable debug access slave interface** parameter when instantiating the Nios II DPX processor. For more information, refer to “*Nios II DPX Datapath Processor Tab*” on page 2-1.

Table 1-2 lists the memory map for the debug slave access.

Table 1-2. Debug Access Slave Memory Map

Byte Offset	End	Span	Limit	Description
0x00000000	$2^{INST_ADDR_BITS} - 1$	$2^{INST_ADDR_BITS}$	4 MB	Instruction RAM
0x00400000	$0x00400000 + 2^{TASK_ADDR_WIDTH} - 1$	$2^{TASK_ADDR_WIDTH}$	16,384 entries	Task address table
0x00402000	0x00402007	8	8 bytes	Reset controller
0x00404000	$2^{STATS_ADDR_BITS} - 1$	$2^{STATS_ADDR_BITS}$	16 KB	Statistics
0x00408000	0x004fffff	0x00018000	—	Reserved



The debug unit (or other external configuration circuitry) has access to instruction memory and the task address table. Update these memory areas only when the MTP is held in soft reset. Updating the instructions or task table while threads are running can produce unpredictable results.

Debug Statistics Collector

Nios II DPX datapath processor implements non-intrusive statistics counters to aid in system debugging and monitoring. These counters are accessible over the Avalon-MM slave debug interface.

Table 1–3 shows the memory map for the statistics counters, starting at offset 0x00401000 in the debug interface memory map (shown in Table 1–2).

Table 1–3. Statistics Counters Memory Map (Part 1 of 3)

Byte Offset	Statistic Type	Statistic	Description
0x0000	Counter	MTP instruction execution counters	32-bit counters, categorized by instruction type per thread, count the number of instructions executed. Each instruction type in the following list contains eight 32-bit counters, one for each thread: <ul style="list-style-type: none"> ■ 0x0000—Load instruction counters ■ 0x0020—Store instruction counters ■ 0x0040—Arithmetic and logic instruction counters ■ 0x0060—Compare and branch instruction counters ■ 0x0080—Debug instruction counters ■ 0x00A0—Exit instruction counters ■ 0x00C0—Send instruction counters ■ 0x00E0—Other instruction counters
0x0100	Counter	CID usage counters or MTP instruction execution counters	For single-core Nios II DPX datapath processor configurations, the 32-bit counters count the number of times tasks run for each CID up to 64 CIDs. For dual-core Nios II DPX datapath processor configurations, the 32-bit counters count the number of instructions executed on the eight threads of the second MTP core. (Refer to previous row's description.)
0x0200	Counter	Input task ID usage counter	32-bit counters count the number of times tasks are executed for each input task ID up to 64 task IDs.
0x0300	Counter	Output task ID usage counter	32-bit counters count the number of messages sent for each output task ID up to 64 task IDs.
0x0400	Counter	Task ticks monitor	32-bit counters count thread cycles (that is, clock cycles / 8) taken for running a task from start to completion for each task ID up to 64 task IDs. These counters are not cumulative; they count only the last run.

Table 1-3. Statistics Counters Memory Map (Part 2 of 3)

Byte Offset	Statistic Type	Statistic	Description
0x0500	Level, reset-on-read	CID ticks monitor	<p>32-bit indicators report the number of thread cycles (that is, clock cycles / 8) taken for executing the task for the message with CID 0, the last time a task executed. Monitoring the duration of processing for CID 0 is equivalent to monitoring the duration for any single CID. The collected data is representative of the time taken to process a single data item. Each item in the following list contains one 32-bit indicator:</p> <ul style="list-style-type: none"> ■ 0x0500—Current value ■ 0x0504—Latched level value ■ 0x0508—Latched minimum value ■ 0x050A—Latched maximum value <p>Reading offset 0x0500 resets the minimum and maximum values. To preserve the minimum and maximum values, use the indicator at offset 0x0D00.</p>
0x0510	Level, reset-on-read	Free CID FIFO level	<p>32-bit indicators monitor the CID free list level. Each item in the following list contains one 32-bit indicator:</p> <ul style="list-style-type: none"> ■ 0x0510—Current value ■ 0x0514—Latched level value ■ 0x0518—Latched minimum value ■ 0x051A—Latched maximum value <p>Reading offset 0x0510 resets the minimum and maximum values. To preserve the minimum and maximum values, use the indicator at offset 0x0D10.</p>
0x0520	Level, reset-on-read	RX message queue level	<p>32-bit indicators monitor the RX message queue level. Each item in the following list contains one 32-bit indicator:</p> <ul style="list-style-type: none"> ■ 0x0520—Current value ■ 0x0524—Latched level value ■ 0x0528—Latched minimum value ■ 0x052A—Latched maximum value <p>Reading offset 0x0520 resets the minimum and maximum values. To preserve the minimum and maximum values, use the indicator at offset 0x0D20.</p>
0x0540	Level, reset-on-read	CID reorder level	<p>32-bit indicators monitor the CID reorder queue fill level. Each item in the following list contains one 32-bit indicator:</p> <ul style="list-style-type: none"> ■ 0x0540—Current value ■ 0x0544—Latched level value ■ 0x0548—Latched minimum value ■ 0x054A—Latched maximum value <p>Reading offset 0x0540 resets the minimum and maximum values. To preserve the minimum and maximum values, use the indicator at offset 0x0D40.</p>
0x0D00	Level	CID ticks monitor	<p>Same as 0x0500 except reading the current value does not reset minimum and maximum values.</p>

Table 1–3. Statistics Counters Memory Map (Part 3 of 3)

Byte Offset	Statistic Type	Statistic	Description
0x0D10	Level	Free CID level	Same as 0x0510 except reading the current value does not reset minimum and maximum values.
0x0D20	Level	RX message queue level	Same as 0x0520 except reading the current value does not reset minimum and maximum values.
0x0D40	Level	CID reorder level	Same as 0x0540 except reading the current value does not reset minimum and maximum values.

Nios II DPX Clock Domains

Table 1–4 shows the clock domains for the Nios II DPX processor, along with the clock signal name and interfaces in each domain. Each domain operates independently with cross-clocking logic between domains. For information about each interface, refer to “Nios II DPX Processor Interfaces” on page 1–21.

Table 1–4. Nios II DPX Clock Domains

Domain	Clock Signal	Interfaces
Message	message_clk	RX message interface, TX message interface, CID request interface
Input context register	context_in_clk	Input context register interface
Output context register	context_out_clk	Output context register interface
MTP	cpu_clk	Fixed- and variable-latency data masters
Debug	debug_clk	Debug interface

Nios II DPX Processor Reset Signals

Figure 1-6 shows an example reset block diagram for a Nios II DPX datapath processor instantiated in Qsys.

Figure 1-6. Example Reset Block Diagram

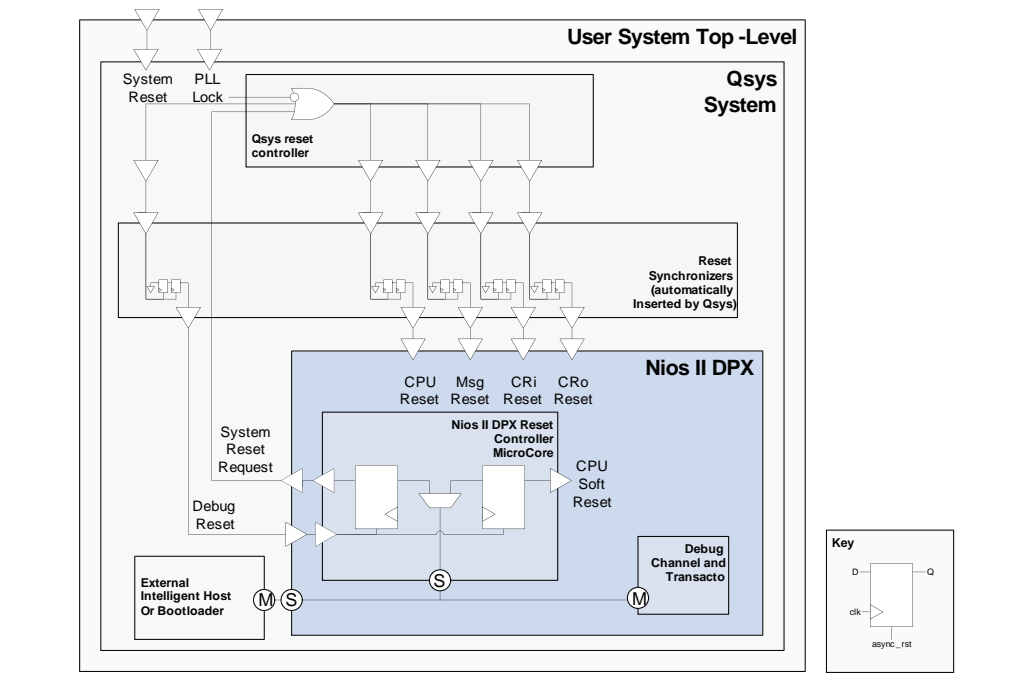


Table 1-5 shows the external resets associated with Nios II DPX datapath processor. All resets can be asserted asynchronously but must be de-asserted synchronous to their clock domains.

Table 1-5. Nios II DPX Datapath Processor Reset Signals

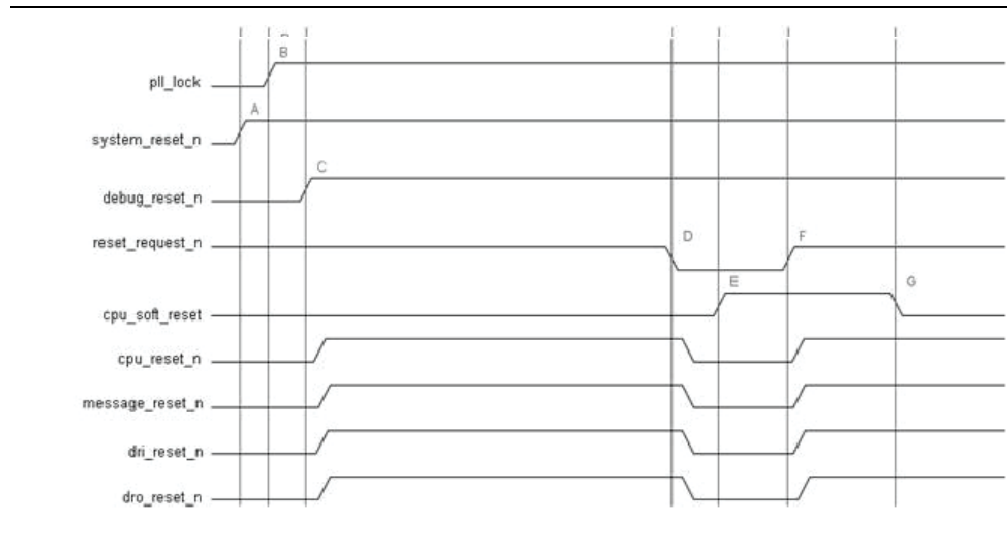
Clock Signal	Direction	Description
cpu_rst_n	input	This reset is associated with the MTP clock domain and must be de-asserted synchronous to <code>cpu_clk</code> signal.
message_rst_n	input	This reset is associated with the message clock domain and must be de-asserted synchronous to the <code>message_clk</code> signal.
context_in_rst_n	input	This reset is associated with the input context register clock domain and must be de-asserted synchronous to <code>context_in_clk</code> signal.
context_out_rst_n	input	This reset is associated with the output context register clock domain and must be de-asserted synchronous to <code>context_out_clk</code> signal.
debug_rst_n	Input	This reset is driven by the debug channel and is associated with the debug clock domain.
reset_request_n	Output	This reset is controlled through reset controller. The reset signal is synchronous to <code>debug_clk</code> signal.
cpu_soft_reset	internal	This reset is controlled through reset controller. The reset must be asserted while the <code>system_reset_request_n</code> signal is asserted and after asserting other Nios II DPX processor reset signals.

The processor soft reset prevents the MTP from fetching instructions, which allows the debugger (or an external host) to configure the Nios II DPX processor elements such as the task address table and instruction memory. The processor soft reset must be asserted only while the system reset request signal is asserted.

Reset Sequence with Debugger

Figure 1-7 shows a sample timing diagram for the example Qsys system shown in Figure 1-6.

Figure 1-7. Reset Sequence



The following list describes the initial state of the system:

- System reset is asserted.
- PLL lock is not asserted.
- All Nios II DPX processor reset input signals are asserted.
- `reset_request_n` and `cpu_soft_reset` signals are not asserted.

The following sequence of events occurs at the lettered points in Figure 1-7 when the system starts up:

- A. System reset is de-asserted.
- B. PLL lock is achieved.
- C. Once system reset is de-asserted and PLL lock is achieved, the debug channel releases the debug reset synchronous to the debug clock, allowing the host to configure the debugger, including reading debug core status and ID registers, and setting hardware breakpoints.

With the system reset de-asserted, the PLL lock achieved, and the `reset_request_n` signal de-asserted, releasing the debug reset results in de-asserting all other Nios II DPX processor reset signals. Because all reset inputs are de-asserted, the Nios II DPX MTP starts fetching instructions and the debugger can be configured.

- D. The debugger writes to the reset controller to assert the `reset_request_n` signal. Asserting the `reset_request_n` signal causes assertion of all Nios II DPX processor reset input signals through external logic, typically inserted by Qsys. The `debug_rst_n` signal remains de-asserted preserving the debug unit's state.
- E. The debugger asserts `cpu_soft_reset` through the reset controller.
- F. The debugger de-asserts the `reset_request_n` signal through the reset controller, while keeping the `cpu_soft_reset` signal asserted. This prevents the MTP from fetching and executing instructions, allowing the debugger to configure the system and the Nios II DPX processor instruction memory and task address table.
- G. Once configured, the debugger de-asserts the `cpu_soft_reset` signal allowing the MTP to fetch instructions and begin operating normally.

Reset Considerations when Designing Outside of Qsys

When designing a system outside of Qsys, you must manually create all the logic shown in [Figure 1-6](#) as part of your top-level design. Consider the following points when designing outside of Qsys:

- All resets must be de-asserted synchronous to their clock domains.
- External logic must ensure that all reset domains are carefully managed. For example, resetting the MTP domain (the `cpu_reset` signal) without resetting the MIU domain (the `message_reset` signal) can leave the system in an indeterminate state.
- The processor soft reset can only be asserted when asserting system reset request and processor reset.
- If the system requires configuration on power up, external logic must perform an appropriate startup sequence similar to the following example:
 - a. Assert all resets on power-up.
 - b. De-assert all reset input signals once PLL lock is achieved.
 - c. Assert system reset request and processor soft reset in the reset controller.
 - d. De-assert reset request in the reset controller, and configure the system and Nios II DPX processor.
 - e. De-assert processor soft reset.

Nios II DPX Processor Interfaces

This section describe the interfaces of the Nios II DPX datapath processor.

Interfaces

The Nios II DPX datapath processor has the following interfaces:

- RX message interface—Avalon-ST sink interface allows receiving Avalon-ST PE messages.
- TX message source interface—Avalon-ST source interface allows transmission of Avalon-ST PE messages.

- CID Request interface—Allows an external entity, typically the input PE, to request CIDs when generating a message for a new context.
- Input context register interface—Write-only Avalon-MM slave interface allows an external entity to write context-specific data into the input context registers.
- Output context register interface—Read-only Avalon-MM slave interface allows an external entity to retrieve context-specific data from the output context registers.
- Fixed- and variable-latency data masters—Allow the MTP to fetch data from external memory referenced by your software tasks.
- Thread information interface—Provides context specific information to partition data memory for context-specific accesses over the fixed- and variable-latency data masters.
- Avalon-MM slave debug interface—Allows an external Avalon-MM master to access the following modules within the Nios II DPX processor:
 - Instruction RAM for code download and configuration in systems without debug enabled
 - Task address table
 - Statistics
 - Reset controller registers
- Avalon-ST debug interfaces—Allow the Avalon-ST JTAG PHY to connect to the debug module.
 - Debug channel in—Avalon-ST sink interface
 - Debug channel out—Avalon-ST source interface

RX Message Interface

The RX message interface is an Avalon-ST sink with zero ready latency operating in the message clock domain. In your system, connect the interface to a message interconnect or directly to the message interface of another PE.

Table 1–6 lists the RX message interface signals.

Table 1–6. RX Message Interface Signals

Signal (1)	Width (Bits)	Direction	Description
ready	1	Output	De-asserted by sink to prohibit a transfer.
valid	1	Input	Asserted to activate a bus cycle.
startofpacket	1	Input	Asserted on the first cycle of a packet.
endofpacket	1	Input	Asserted on the last cycle of a packet.
data	<i>RX message control word width + 32</i>	Input	Message's data and control contents.

Notes to Table 1–6:

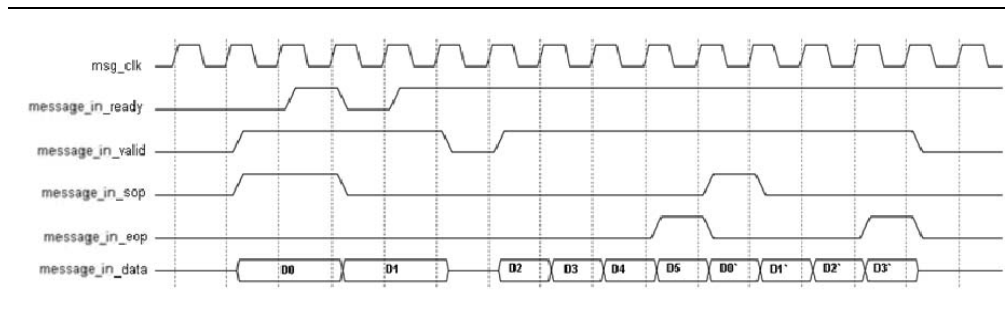
(1) In the RTL code, these signals have a `message_in_` prefix.

The upper bits of the data signal contain the control word. The control word is held constant for the duration of the transfer. The control word layout is configurable when instantiating the Nios II DPX datapath processor. For more information, refer to “External Interfaces (advanced) Tab” on page 2-6.

The lower 32 bits of the data signal contain a 32-bit data argument. Data arguments are loaded sequentially in the RX message registers starting from offset 0 and transferred one argument per beat. The extension registers configuration, described in “Extension Registers” on page 1-5, determines the maximum number of data arguments to transfer.

Figure 1-8 shows the RX message interface timing.

Figure 1-8. RX Message Interface Timing Diagram



TX Message Interface

The TX message interface is an Avalon-ST source with zero ready latency operating in the message clock domain. In your system, connect the interface to a message interconnect or directly to the message interface of another PE.

Table 1-7 lists the TX message interface signals.

Table 1-7. TX Message Interface Signals

Signal (1)	Width (Bits)	Direction	Description
ready	1	Input	De-asserted by sink to prohibit a transfer.
valid	1	Output	Asserted to activate a bus cycle.
startofpacket	1	Output	Asserted on the first cycle of a packet.
endofpacket	1	Output	Asserted on the last cycle of a packet.
data	<i>TX message control word width + 32</i>	Output	Message's data and control contents.

Notes to Table 1-7:

(1) In the RTL code, these signals have a message_out_ prefix.

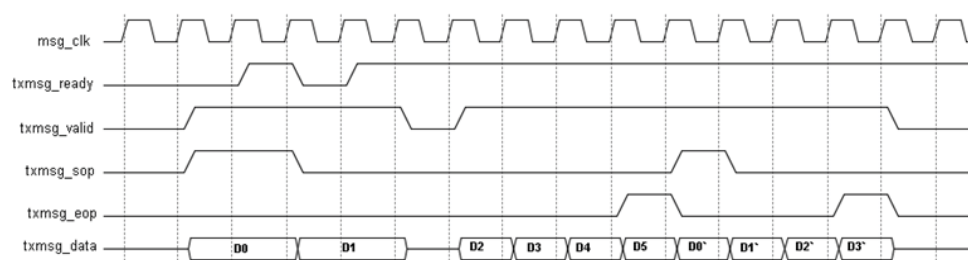
The upper bits of the data signal contain the control word. The control word is held constant for duration of the transfer. The control word layout is configurable when instantiating the Nios II DPX datapath processor. For more information, refer to “External Interfaces (advanced) Tab” on page 2-6.

The lower 32 bits of the data signal contain a 32-bit data argument. Data arguments are loaded sequentially in the TX message registers starting from offset 0 and transferred one argument per beat. The extension registers configuration, described in “[Extension Registers](#)” on page 1-5, determines the maximum number of data arguments to transfer. The actual number of data arguments to transfer is specified by software as an argument in the `snd` and `sndi` instructions.

For more information about software task sending options, refer to the `snd` instruction in the *MTP Instruction Set and Application Binary Interface* chapter in the *Nios II DPX Software Development* section of the *Nios II DPX Datapath Processor Handbook*.

Figure 1-9 shows the TX message interface timing.

Figure 1-9. TX Message Interface Timing Diagram



CID Request Interface

The CID request interface is an Avalon-ST source with zero ready latency operating in the message clock domain. The input PE uses this interface to request a CID to assign to incoming data. This interface only exists when **Number of context IDs** is greater than zero. For more information, refer to “[Message Interface Unit Tab](#)” on page 2-4.

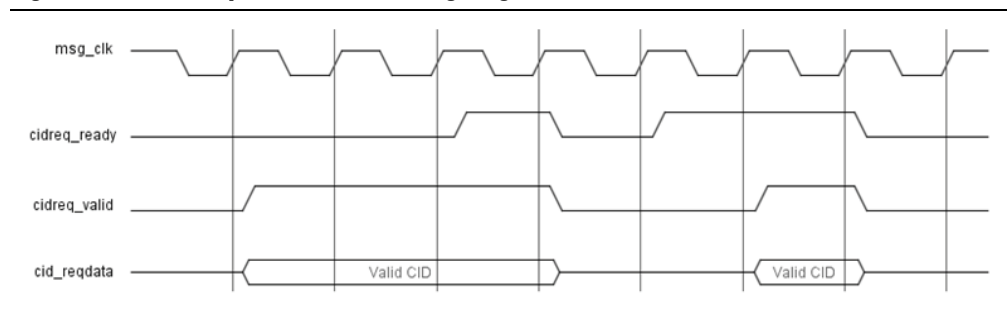
Table 1-8 lists the CID request interface signals.

Table 1-8. CID Request Interface Signals

Signal	Width (Bits)	Direction	Description
<code>cid_request_ready</code>	1	Input	Asserted to request a CID.
<code>cid_request_valid</code>	1	Output	Asserted when CIDs are available. When de-asserted, no CIDs are available.
<code>cid_request_data</code>	$\log_2(\text{number of CIDs})$	Output	This signal provides the CID value. This signal is only valid when <code>cid_request_valid</code> is asserted.

Figure 1-10 shows the CID request interface timing.

Figure 1-10. CID Request Interface Timing Diagram



Input Context Register Interface

The input context register interface is a write-only Avalon-MM slave with zero wait states operating in the input context register clock domain. In a typical system, connect the interface to an Avalon-MM master port on your input PE.

This interface only exists when your extension register configuration includes input context registers. For more information, refer to “Nios II DPX Datapath Processor Tab” on page 2-1.



The input context register interface is configured as a byte-invariant big-endian interface. For more information, refer to “Memory Addressing and Byte Order” on page 1-37.

There is one bank of input context registers per CID. The extension registers configuration, described in “Extension Registers” on page 1-5, determines the number of registers per bank, with a maximum of 16 registers per bank. A 7-bit wide byte address used to access the registers within each bank allows access to up to 128 bytes of input context register data. 128 bytes is more address space than is needed by any of the extension register configurations, thus, allows for possible expansion in future versions of the Nios II DPX datapath processor.

The input context register interface can support up to 128 bytes of input context register storage. The register address is word-oriented. Therefore, the possible register address widths are as shown in Table 1-9.

Table 1-9. Input Context Register Address Widths

<CIDataWidth> (1)	<CIRegAddrWidth> (2)
32	5
64	4
128	3

Notes to Table 1-9:

(1) <CIDataWidth> represents the number of bits in each input context register.

(2) <CIRegAddrWidth> represents the number of address bits required to address an input context register.

Table 1–10 lists the input context register interface signals.

Table 1–10. Input Context Register Interface Signals

Signal	Width (Bits)	Direction	Description
context_in_address	$\langle CIDW \rangle + \langle CRegAddrWidth \rangle$ (1) (2)	Input	This signal addresses the input context registers (CRi). The MSBs contain a CID and identify a bank of registers. The LSBs identify the offset to the registers within the bank.
context_in_write	1	Input	This signal indicates that the address and data phase of a write cycle are in progress, and that the context_in_address, context_in_byteenable, and context_in_writedata signals are all valid.
context_in_byteenable	$\langle CIDataWidth \rangle / 8$ (3)	Input	This signal is used with write byte masks during write cycles only. Writes to any particular byte lane are ignored if the byte enable is not asserted. The MTP treats the input context registers as 32-bit registers and only does 32-bit reads.
context_in_writedata	$\langle CIDataWidth \rangle = 32, 64, \text{ or } 128$ (3)	Input	This signal supplies the data for a write cycle. The data is valid only when the context_in_write signal is asserted. At other times, the data can change and is ignored.

Notes to Table 1–10:

- (1) $\langle CIDW \rangle$ represents the number of bits in a CID, calculated as $\log_2(\langle \text{number of CIDs} \rangle)$.
- (2) $\langle CRegAddrWidth \rangle$ represents the number of address bits required to address an input context register. $\langle CRegAddrWidth \rangle$ is defined in Table 1–9.
- (3) $\langle CIDataWidth \rangle$ is determined by the **Width of input context register data bus** parameter, described in “Message Interface Unit Tab” on page 2–4.

The width of the input context register data bus determines how many 32-bit registers are accessible in the context_in_writedata signal in any one bus cycle. The context_in_address signal determines which registers are accessed. Because the input context register interface is configured as a byte-invariant big-endian interface, the data on the context_in_writedata signal is byte swapped when the hardware writes the data into the CRi registers. The following sections describe the three data bus width options.

Input Context Register Data Width = 32

When the width of input context register data bus is 32 bits, registers are accessed one at a time. The least significant five bits of the `context_in_address` signal point to the register to access from the specified context. Table 1-11 shows how the 32-bit data lane maps to the CRi registers, where x is the value contained in `context_in_address[4:0]`. The byte-enable signal, `context_in_byteenable`, can be used to select individual bytes to update within the selected register.

Table 1-11. 32-Bit Data Lane Mapping

Byte Lane	Context Register
<code>context_in_writedata[7:0]</code>	$\text{CRi}<X>[31:24]$
<code>context_in_writedata[15:8]</code>	$\text{CRi}<X>[23:16]$
<code>context_in_writedata[23:16]</code>	$\text{CRi}<X>[15:8]$
<code>context_in_writedata[31:24]</code>	$\text{CRi}<X>[7:0]$

The bit arrangement of the `context_in_address` signal with a 32-bit data bus is shown in Table 1-12.

Table 1-12. context_in_address with 32-Bit Words

$<CIDW> + 5$...	5	4	...	0
CID					Register number

Input Context Register Data Width = 64

When the width of input context register data bus is 64 bits, registers are accessed two at a time. The least significant four bits of `context_in_address` form an index to a pair of registers for the specified context. Table 1-13 shows how the two 32-bit sections of the 64-bit data lanes map to the CRi registers, where x is the value contained in `context_in_address[3:0]`. For example, for `context_in_address[3:0] = 0`, the context registers selected are CRi0 and CRi1. For `context_in_address[3:0] = 1`, the context registers selected are CRi2 and CRi3. The byte-enable signal, `context_in_byteenable`, can be used to select individual bytes to update within the selected pair of registers.

Table 1-13. 64-Bit Data Lane Mapping

Byte Lane	Context Register
<code>context_in_writedata[7:0]</code>	$\text{CRi}<2x+1>[31:24]$
<code>context_in_writedata[15:8]</code>	$\text{CRi}<2x+1>[23:16]$
<code>context_in_writedata[23:16]</code>	$\text{CRi}<2x+1>[15:8]$
<code>context_in_writedata[31:24]</code>	$\text{CRi}<2x+1>[7:0]$
<code>context_in_writedata[39:32]</code>	$\text{CRi}<2x>[31:24]$
<code>context_in_writedata[47:40]</code>	$\text{CRi}<2x>[23:16]$
<code>context_in_writedata[55:48]</code>	$\text{CRi}<2x>[15:8]$
<code>context_in_writedata[63:56]</code>	$\text{CRi}<2x>[7:0]$

The bit arrangement of the context_in_address signal with a 64-bit data bus is shown in Table 1-14.

Table 1-14. context_in_address with 64-Bit Words

$\langle CIDW \rangle + 4$...	4	3	...	0
CID					(Register number) / 2

Input Context Register Data Width = 128

When the width of input context register data bus is 128 bits, registers are accessed four at a time. The least significant three bits of context_in_address form an index to a quartet of registers for the specified context. Table 1-15 shows how the four 32-bit lanes of the 128-bit data bus map to the input context registers, where x is the value contained in context_in_address[2:0]. For example, for context_in_address[2:0] = 0, the context registers selected are CRi0, CRi1, CRi2, and CRi3. For context_in_address [2:0] = 1, the context registers selected are CRi4, CRi5, CRi6, and CRi7. The byte-enable signal, context_in_byteenable, can be used to select individual bytes to update within the selected quartet of registers.

Table 1-15. 128-Bit Data Lane Mapping

Byte Lane	Context Register
context_in_writedata[7:0]	CRi<4x+3>[31:24]
context_in_writedata[15:8]	CRi<4x+3>[23:16]
context_in_writedata[23:16]	CRi<4x+3>[15:8]
context_in_writedata[31:24]	CRi<4x+3>[7:0]
context_in_writedata[39:32]	CRi<4x+2>[31:24]
context_in_writedata[47:40]	CRi<4x+2>[23:16]
context_in_writedata[55:48]	CRi<4x+2>[15:8]
context_in_writedata[63:56]	CRi<4x+2>[7:0]
context_in_writedata[71:64]	CRi<4x+1>[31:24]
context_in_writedata[79:72]	CRi<4x+1>[23:16]
context_in_writedata[87:80]	CRi<4x+1>[15:8]
context_in_writedata[95:88]	CRi<4x+1>[7:0]
context_in_writedata[103:96]	CRi<4x>[31:24]
context_in_writedata[111:104]	CRi<4x>[23:16]
context_in_writedata[119:112]	CRi<4x>[15:8]
context_in_writedata[127:120]	CRi<4x>[7:0]

The bit arrangement of the context_in_address signal with a 128-bit data bus is shown in [Table 1-16](#).

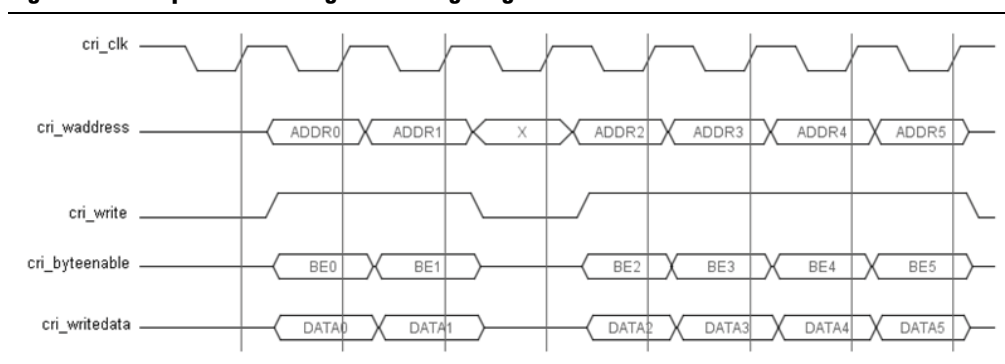
Table 1-16. context_in_address with 128-Bit Words

$\langle CIDW \rangle + 3$...	3	2	...	0
	CID	(Register number) / 4			

Input Context Register Interface Timing

[Figure 1-11](#) shows the input context register interface timing.

Figure 1-11. Input Context Register Timing Diagram



Output Context Register Interface

The output context register interface is a read-only Avalon-MM slave with a two-cycle read latency and zero wait states operating in the output context register clock domain. In a typical system, connect the interface to the Avalon-MM master port on your output PE.

This interface only exists when your extension register configuration includes output context registers. For more information, refer to [“Nios II DPX Datapath Processor Tab” on page 2-1](#).



The output context register interface is configured as a byte-invariant big-endian interface. For more information, refer to [“Memory Addressing and Byte Order” on page 1-37](#).

There is one bank of output context registers per CID. The extension registers configuration, described in [“Extension Registers” on page 1-5](#), determines the number of registers per bank, with a maximum of 16 registers per bank. A 7-bit wide byte address used to access the registers within each bank allows access to up to 128 bytes of output context register data. 128 bytes is more address space than is needed by any of the extension register configurations, thus, allows for possible expansion in future versions of the Nios II DPX datapath processor.

The output context register interface can support up to 128 bytes of output context register storage. The register address is word-oriented. Therefore, the possible register address widths are as shown in Table 1–17.

Table 1–17. Output Context Register Address Widths

<CODataWidth> (1)	<CORegAddrWidth> (2)
32	5
64	4
128	3

Notes to Table 1–9:

- (1) <CODataWidth> represents the number of bits in each output context register.
- (2) <CORegAddrWidth> represents the number of address bits required to address an output context register.

Table 1–18 lists the output context register interface signals.

Table 1–18. Output Context Register Interface Signals

Signal	Width (Bits)	Direction	Description
context_out_address	<CIDW> + <CORegAddrWidth> (1) (2)	Input	This signal addresses the output context registers (CRO). The MSBs contain a CID and select a bank of registers, and the LSBs select the register within the bank.
context_out_read	1	Input	This signal initiates a read request from an Avalon-MM master. This signal can be permanently asserted because the output context register interface has a two-cycle read latency and zero wait states.
context_out_readdata	<CODataWidth> = 32, 64, or 128 (3)	Output	This signal supplies the data from the addressed output context register.

Notes to Table 1–18:

- (1) <CIDW> represents the number of bits in a CID, calculated as $\log_2(\text{number of CIDs})$.
- (2) <CORegAddrWidth> represents the number of address bits required to address an output context register. <CORegAddrWidth> is defined in Table 1–17.
- (3) <CODataWidth> is determined by the **Width of output context register data bus** parameter, described in “Message Interface Unit Tab” on page 2–4.

The width of output context register data bus determines how many 32-bit registers are accessible in the context_out_readdata signal in any one bus cycle. The context_out_address signal determines which registers are accessed. Because the output context register interface is configured as a byte-invariant big-endian interface, the data on the context_out_readdata signal is byte swapped when the hardware writes the data into the CRO registers. The following sections describe the three data bus width options.

Output Context Register Data Width = 32

When the width of output context register data bus is 32 bits, registers are accessed one at a time. The least significant five bits of the `context_out_address` signal point to the register to access from the specified context. Table 1–19 shows how the 32-bit data lane maps to the CRO registers, where x is the value contained in `context_out_address[4:0]`.

Table 1–19. 32-Bit Data Lane Mapping

Byte Lane	Context Register
<code>context_out_readdata[7:0]</code>	$\text{CRO}_{\langle X \rangle}[31:24]$
<code>context_out_readdata[15:8]</code>	$\text{CRO}_{\langle X \rangle}[23:16]$
<code>context_out_readdata[23:16]</code>	$\text{CRO}_{\langle X \rangle}[15:8]$
<code>context_out_readdata[31:24]</code>	$\text{CRO}_{\langle X \rangle}[7:0]$

The bit arrangement of the `context_out_address` signal with a 32-bit data bus is shown in Table 1–20.

Table 1–20. context_out_address with 32-Bit Words

$\langle \text{CIDW} \rangle + 5$...	5	4	...	0
CID					Register number

Output Context Register Data Width = 64

When the width of output context register data bus is 64 bits, registers are accessed two at a time. The least significant four bits of `context_out_address` form an index to a pair of registers for the specified context. Table 1–21 shows how the two 32-bit sections of the 64-bit data lanes map to the CRO registers, where x is the value contained in `context_out_address[3:0]`. For example, for `context_out_address[3:0] = 0`, the context registers selected are CRO0 and CRO1. For `context_out_address[3:0] = 1`, the context registers selected are CRO2 and CRO3.

Table 1–21. 64-Bit Data Lane Mapping

Byte Lane	Context Register
<code>context_out_readdata[7:0]</code>	$\text{CRO}_{\langle 2x+1 \rangle}[31:24]$
<code>context_out_readdata[15:8]</code>	$\text{CRO}_{\langle 2x+1 \rangle}[23:16]$
<code>context_out_readdata[23:16]</code>	$\text{CRO}_{\langle 2x+1 \rangle}[15:8]$
<code>context_out_readdata[31:24]</code>	$\text{CRO}_{\langle 2x+1 \rangle}[7:0]$
<code>context_out_readdata[39:32]</code>	$\text{CRO}_{\langle 2x \rangle}[31:24]$
<code>context_out_readdata[47:40]</code>	$\text{CRO}_{\langle 2x \rangle}[23:16]$
<code>context_out_readdata[55:48]</code>	$\text{CRO}_{\langle 2x \rangle}[15:8]$
<code>context_out_readdata[63:56]</code>	$\text{CRO}_{\langle 2x \rangle}[7:0]$

The bit arrangement of the context_out_address signal with a 64-bit data bus is shown in Table 1–22.

Table 1–22. context_out_address with 64-Bit Words

$\langle CIDW \rangle + 4$...	4	3	...	0
CID					(Register number) / 2

Output Context Register Data Width = 128

When the width of output context register data bus is 128 bits, registers are accessed four at a time. The least significant three bits of context_out_address form an index to a quartet of registers for the specified context. Table 1–23 shows how the four 32-bit lanes of the 128-bit data bus map to the output context registers, where x is the value contained in context_out_address[2:0]. For example, for context_out_address[2:0] = 0, the context registers selected are CRO0, CRO1, CRO2, and CRO3. For context_out_address[2:0] = 1, the context registers selected are CRO4, CRO5, CRO6, and CRO7.

Table 1–23. 128-Bit Data Lane Mapping

Byte Lane	Context Register
context_out_readdata[7:0]	CRO<4x+3>[31:24]
context_out_readdata[15:8]	CRO<4x+3>[23:16]
context_out_readdata[23:16]	CRO<4x+3>[15:8]
context_out_readdata[31:24]	CRO<4x+3>[7:0]
context_out_readdata[39:32]	CRO<4x+2>[31:24]
context_out_readdata[47:40]	CRO<4x+2>[23:16]
context_out_readdata[55:48]	CRO<4x+2>[15:8]
context_out_readdata[63:56]	CRO<4x+2>[7:0]
context_out_readdata[71:64]	CRO<4x+1>[31:24]
context_out_readdata[79:72]	CRO<4x+1>[23:16]
context_out_readdata[87:80]	CRO<4x+1>[15:8]
context_out_readdata[95:88]	CRO<4x+1>[7:0]
context_out_readdata[103:96]	CRO<4x>[31:24]
context_out_readdata[111:104]	CRO<4x>[23:16]
context_out_readdata[119:112]	CRO<4x>[15:8]
context_out_readdata[127:120]	CRO<4x>[7:0]

The bit arrangement of the context_out_address signal with a 128-bit data bus is shown in Table 1–24.

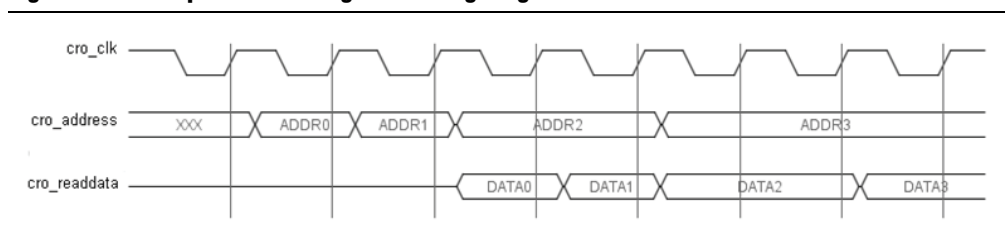
Table 1–24. context_out_address with 128-Bit Words

$\langle CIDW \rangle + 3$...			3	2	...	0
	CID						(Register number) / 4

Output Context Register Interface Timing

Figure 1–12 shows the output context register interface timing. The context_out_read signal is permanently asserted.

Figure 1–12. Output Context Register Timing Diagram



Fixed-Latency Data Master Interface

The fixed-latency data master interface is an Avalon-MM master with a two-cycle read latency giving the MTP access to data memory and operates in the MTP clock domain. In your system, connect this interface directly to each fixed-latency peripheral and the data memory for the Nios II DPX processor.



The fixed-latency data master interface is configured as a byte-invariant big-endian interface. For more information, refer to “Memory Addressing and Byte Order” on page 1–37.

Table 1–25 lists the fixed-latency data master interface signals. The fixed-latency data master does not have waitrequest and readdatavalid signals and instead expects the readdata signal to be returned two cycles after the request is initiated.

Table 1–25. Fixed-Latency Data Master Interface Signals (Part 1 of 2)

Signal (1)	Width (Bits)	Direction	Description
read	1	Output	Indicates that the address phase of a read cycle is in progress and that the address signal is valid. The data phase is exactly two cycles later.
write	1	Output	Indicates that the address phase of a write cycle is in progress and that the address, byteenable, and data signals are valid.
address	Variable	Output	Byte address for the read or write signal.
readdata	32	Input	Data supplied by the memory controller for read operations. The data phase is exactly two cycles after the read signal is asserted.

Table 1-25. Fixed-Latency Data Master Interface Signals (Part 2 of 2)

Signal (1)	Width (Bits)	Direction	Description
writedata	32	Output	Supplies the write data for a write cycle. The data is valid only when the <code>write</code> signal is asserted. At other times, the data can change and the signal must be ignored.
byteenable	4	Output	Byte masks for write cycles only. The memory controller must ignore writes for a lane that does not have the corresponding <code>byteenable</code> signal asserted.

Note to Table 1-25:

(1) In the RTL code, these signals have a `fixed_latency_data_master_` prefix.

This interface can be used in conjunction with the “[Thread Information Interface](#)” on [page 1-35](#) to partition memory for context-specific accesses.



Having multiple threads or processors accessing the same peripherals might create contention. To avoid contention, consider using a mutex core or partitioning memory based on thread or CID using the thread information interface.

Variable-Latency Data Master Interface

The variable-latency data master interface is an Avalon-MM master giving the MTP access to data memory and operates in the MTP clock domain.



The variable-latency data master interface is configured as a byte-invariant big-endian interface. For more information, refer to “[Memory Addressing and Byte Order](#)” on [page 1-37](#).

[Table 1-26](#) lists the variable-latency data master interface signals.

Table 1-26. Variable-Latency Data Master Interface Signals

Signal (1)	Width (Bits)	Direction	Description
read	1	Output	Indicates that the address phase of a read cycle is in progress and that the address signal is valid.
write	1	Output	Indicates that the address phase of a write cycle is in progress and that the address, <code>byteenable</code> , and data signals are valid.
address	Variable	Output	Byte address for the <code>read</code> or <code>write</code> signal.
waitrequest	1	Input	Asserted by connected slaves or message interconnect to stall a read or write operation.
readdatavalid	1	Input	Indicates valid data has arrived.
readdata	32	Input	Data supplied by the memory controller for read operations.
writedata	32	Output	Supplies the write data for a write cycle. The data is valid only when the <code>write</code> signal is asserted. At other times, the data can change and the signal must be ignored.
byteenable	4	Output	Byte masks for write cycles only. The memory controller must ignore writes for a lane that does not have the corresponding <code>byteenable</code> signal asserted.

Notes to Table 1-26:

(1) In the RTL code, these signals have a `variable_latency_data_master_` prefix.

This interface can be used in conjunction with the “[Thread Information Interface](#)” on [page 1-35](#) to partition memory for context-specific accesses.

Thread Information Interface

The thread information interface provides the thread number, CID, RXID, and TXID associated with the software task performing a data access over the fixed- or variable-latency data master. This information can be used to partition data memory into segments based on context-specific information.

Table 1–27 lists the thread information interface signals.

Table 1–27. Thread Information Interface Signals

Signal	Width	Direction	Description
threadinfo_data	Variable	Output	Provides the thread number, CID, RXID, and TXID associated with the software task performing a data access.

The data on the threadinfo_data signal is arranged from the highest data pins to the lowest as thread number, CID, RXID and TXID. The information is aligned on bit boundaries defined by their widths when instantiating the Nios II DPX processor. For more information, refer to “[Message Interface Unit Tab](#)” on page 2–4.

The variable-latency data master and fixed-latency data master have a corresponding thread information interface.

Avalon-MM Debug Access Slave Interface

This interface is an Avalon-MM slave interface operating in the debug clock domain that allows an external Avalon-MM master to access the debug addresses described in “[Debug Unit](#)” on page 1–15. This interface only exists when **Enable debug access slave interface** is on. For more information, refer to “[Nios II DPX Datapath Processor Tab](#)” on page 2–1.

Table 1–28 lists the debug access slave interface signals.

Table 1–28. Avalon-MM Debug Access Slave Interface Signals

Signal (1)	Width (Bits)	Direction	Description
read	1	Input	Indicates that the address phase of a read cycle is in progress and that the address signal is valid. The data phase is exactly two cycles later.
write	1	Input	Indicates that the address phase of a write cycle is in progress and that the address, byteenable, and data signals are valid.
address	21	Input	Byte address for the read or write signal.
waitrequest	1	Output	Asserted by connected slaves or message interconnect to stall a read or write operation.
readdatavalid	1	Output	Indicates valid data has arrived.
readdata	32	Output	Data supplied by the memory controller for read operations. The data phase is exactly two cycles after the read signal is asserted.
writedata	32	Input	Supplies the write data for a write cycle. The data is valid only when the write signal is asserted. At other times, the data can change and the signal must be ignored.
byteenable	4	Input	Byte masks for write cycles only. The memory controller must ignore writes for a lane that does not have the corresponding byteenable signal asserted.

Note to Table 1–28:

(1) In the RTL code, these signals have a debug_access_slave_ prefix.

Avalon-ST Debug Interfaces

These debug interfaces are Avalon-ST interfaces operating in the debug clock domain that allow multiple Nios II DPX datapath processors to connect to a single JTAG PHY external to the Nios II DPX processor. These interfaces only exist when **Enable internal JTAG PHY** is off and **Enable debug unit** is on. For more information, refer to “Nios II DPX Datapath Processor Tab” on page 2–1.

Table 1–29 lists the debug channel in Avalon-ST sink interface signals.

Table 1–29. Avalon-ST Debug Channel In Interface Signals

Signal (1)	Width (Bits)	Direction	Description
ready	1	Output	De-asserted by source to prohibit a transfer.
valid	1	Input	Asserted to activate a bus cycle.
channel	1 (2 in dual core)	Input	When 0, selects Avalon packets to transactions converter. When 1, selects processor debug unit 1. When 2, selects processor debug unit 2.
sop	1	Input	Asserted on the first cycle of a packet.
eop	1	Input	Asserted on the last cycle of a packet.
data	8	Input	Message's data and control contents.

Notes to Table 1–29:

(1) In the RTL code, these signals have a debug_in_ prefix.

Table 1–30 lists the debug channel out Avalon-ST source interface signals.

Table 1–30. Avalon-ST Debug Channel Out Interface Signals

Signal (1)	Width (Bits)	Direction	Description
ready	1	Input	De-asserted by sink to prohibit a transfer.
valid	1	Output	Asserted to activate a bus cycle.
channel	1 (2 in dual core)	Output	When 0, selects Avalon packets to transactions converter. When 1, selects processor debug unit 1. When 2, selects processor debug unit 2.
sop	1	Output	Asserted on the first cycle of a packet.
eop	1	Output	Asserted on the last cycle of a packet.
data	8	Output	Message's data and control contents.

Notes to Table 1–30:

(1) In the RTL code, these signals have a debug_out_ prefix.

Memory Addressing and Byte Order

Information in this section applies to the following Nios II DPX datapath processor interfaces:

- “Input Context Register Interface” on page 1–25
- “Output Context Register Interface” on page 1–29
- “Fixed-Latency Data Master Interface” on page 1–33
- “Variable-Latency Data Master Interface” on page 1–34

The memory-mapped interfaces on the Nios II DPX datapath processor follow the Avalon-MM interface specification. The specification requires that a descending bit order is used, with data bits 7 down to 0 representing byte offset 0 of any master or slave port. This type of interface is often called *little endian*. Table 1–31 shows the byte lane addresses for the Avalon-MM protocol.

Table 1–31. Byte Lane Addresses for the Avalon-MM Protocol

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Address 3								Address 2								Address 1								Address 0							



For more information about bit and byte ordering with Avalon-MM interfaces, refer to the *Avalon-MM Byte Ordering Considerations* chapter of the *Embedded Design Handbook*.

The Nios II DPX datapath processor is commonly used in networking applications, and network protocols are typically *big-endian*. For example, Table 1–32 shows an extract from an IPv4 header.

Table 1–32. Example IPv4 Header (Part 1 of 2)

Byte Address	Description	Sample Data
0	Version	0x45
1	DSCP	0x00

Table 1-32. Example IPv4 Header (Part 2 of 2)

Byte Address	Description	Sample Data
2	Total length (MS)	0x01
3	Total length (LS)	0x00
...		
12	Source address (MS)	0xC0
13	Source address	0xA8
14	Source address	0x00
15	Source address (LS)	0x01

The ordering of the IPv4 header bytes poses a potential problem. When the software view of memory from within the Nios II DPX matches the Avalon-MM interface view, word and halfword access to this IPv4 packet would provide byte-swapped results.

The code and comments in [Example 1-1](#) show the results of reading the example IPv4 header using a little endian processor, such as the Nios II processor.

Example 1-1. Reading IPv4 Header with Little Endian Processor

```
# Read version (byte read from address 0 to r1)
ldb r1, 0(r0)    # r1 = 0x45 CORRECT

# Read total length (halfword read from address 2 to r2)
ldh r2, 2(r0)    # r2 = 0x0001 BACKWARDS

# Read source address (word read from address 12 to r3)
ldw r3, 12(r0)   # r3 = 0x0100A8C0 BACKWARDS
```

To avoid this problem, the Nios II DPX datapath processor employs a byte-invariant big-endian implementation, often referred to as *BE-8*. The Nios II DPX processor provides an Avalon-MM compliant bus interface, but word and halfword accesses swap the bytes in hardware.

The following code and code comments show the results of reading the example IPv4 header using a big-endian processor, such as the Nios II DPX datapath processor:

```
# Read version (byte read from address 0 to r1)
ldb r1, 0(r0)    # r1 = 0x45 CORRECT

# Read total length (halfword read from address 2 to r2)
ldh r2, 2(r0)    # r2 = 0x0100 CORRECT

# Read source address (word read from address 12 to r3)
ldw r3, 12(r0)   # r3 = 0xC0A80001 CORRECT
```

[Table 1-33](#) shows the mapping between software accesses and Avalon-MM byte lanes using the Nios II DPX datapath processor.

Table 1-33. Big-Endian Mapping (Part 1 of 2)

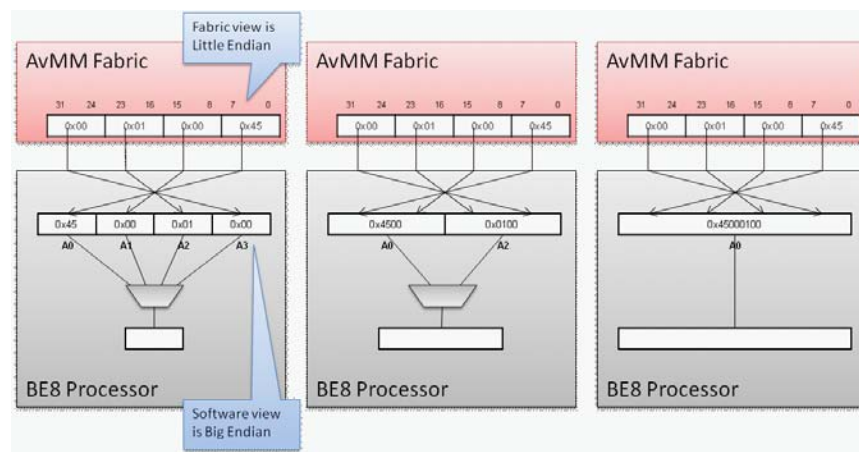
Instruction	31:24	23:16	15:8	7:0
//0xA0 -> 0 stb r1, 0(r0)				0xA0
//0xA1 -> 1 stb r2, 1(r0)			0xA1	

Table 1-33. Big-Endian Mapping (Part 2 of 2)

Instruction	31:24	23:16	15:8	7:0
//0xA2 -> 2 stb r3, 2(r0)		0xA2		
//0xA3 -> 3 stb r4, 3(r0)	0xA3			
//0xA0A1 -> 0 stb r5, 0(r0)			0xA1	0xA0
//A2A3 -> 2 stb r6, 2(r0)	0xA3	0xA2		
//0xA0A1A2A3 -> 0 stb r7, 0(r0)	0xA3	0xA2	0xA1	0xA0

Figure 1-13 shows a conceptual view of byte, halfword, and word memory accesses using the Nios II DPX processor. The byte swapping shown occurs in hardware between the Avalon-MM interface and the software representation.

Figure 1-13. Byte Swapping in Hardware



Accessing Peripheral Registers

Accessing peripheral control and status registers using the Nios II DPX datapath processor requires an extra step. As shown in Figure 1-14, word and halfword accesses to these registers, byte-swapped by the hardware, need to be unswapped by the software.

Figure 1-14. Byte Swapping in Hardware

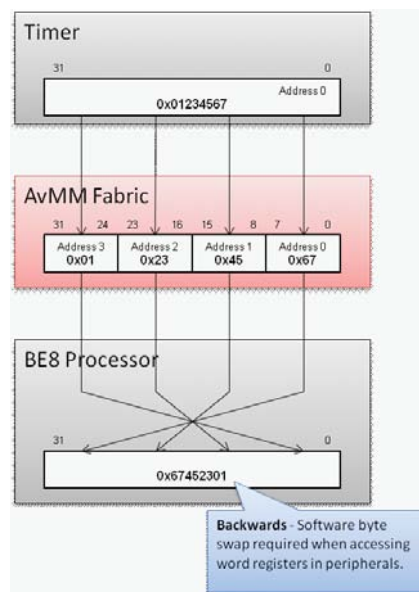


Table 1-34 lists macros, defined in `byte_order.h`, that provide one way to software byte swap when accessing peripheral registers from C code.

Table 1-34. Byte-Swapping Macros

Macro	Description	Behavior
<code>SWAP_BE8_32(x)</code>	Byte swaps a 32-bit value, yielding a 32-bit result.	$((x) \ll 24) \& 0xff000000) $ $((x) \ll 8) \& 0x00ff0000) $ $((x) \gg 8) \& 0x0000ff00) $ $((x) \gg 24) \& 0x000000ff)$
<code>SWAP_BE8_16(x)</code>	Byte swaps a 16-bit value, yielding a 16-bit result.	$((x) \ll 8) \& 0xff00) $ $((x) \gg 8) \& 0x00ff)$

However, when accessing peripheral registers, Altera recommends using the `IORD` and `IOWR` macros described in Table 1-35. These macros, defined in `io.h`, implement byte swapping when necessary.

Table 1-35. Direct-Access Macros (Part 1 of 2)

Macro	Description
<code>IORD_32DIRECT(BASE, OFFSET)</code>	Reads from the 32-bit peripheral register at byte address <code>BASE + OFFSET</code> and byte swaps the resulting data.
<code>IORD_16DIRECT(BASE, OFFSET)</code>	Reads from the 16-bit peripheral register at byte address <code>BASE + OFFSET</code> and byte swaps the resulting data.

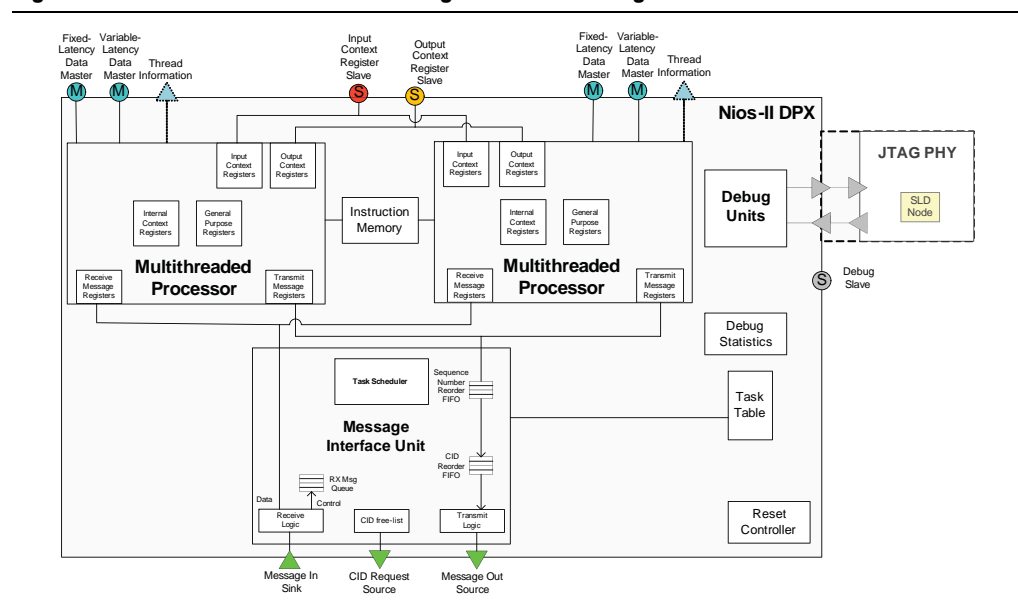
Table 1-35. Direct-Access Macros (Part 2 of 2)

Macro	Description
<code>IORD_8DIRECT(BASE, OFFSET)</code>	Reads from the 8-bit peripheral register at byte address <i>BASE</i> + <i>OFFSET</i> . No byte swap is required.
<code>IOWR_32DIRECT(BASE, OFFSET)</code>	Byte swaps the 32-bit data value and writes the result to the 32-bit peripheral register at byte address <i>BASE</i> + <i>OFFSET</i> .
<code>IOWR_16DIRECT(BASE, OFFSET)</code>	Byte swaps the 16-bit data value and writes the result to the 16-bit peripheral register at byte address <i>BASE</i> + <i>OFFSET</i> .
<code>IOWR_8DIRECT(BASE, OFFSET)</code>	Writes the 8-bit data value to the 8-bit peripheral register at byte address <i>BASE</i> + <i>OFFSET</i> . No byte swap is required.

Nios II DPX Datapath Processor Dual-Core Configuration

You can configure the Nios II DPX with a dual MTP core. Figure 1-15 shows a simplified Nios II DPX datapath processor block diagram configured with a dual core.

Figure 1-15. Nios II DPX Dual-Core Configuration Block Diagram



The dual-core configuration uses resources more efficiently over instantiating two single-core Nios II DPX datapath processors. In particular, the on-chip instruction memory and message buffers are shared.

The topology of the dual-core configuration differs from the single-core configuration in the following ways:

- The dual-core Nios II DPX datapath processor appears to software as a Nios II DPX datapath processor with sixteen threads instead of eight. As incoming messages initiate tasks, the tasks are allocated to one of the two processor cores. When CIDs are used, tasks are allocated in alternation, with even CIDs assigned to MTP 0 and odd CIDs to MTP 1.
- Both MTP cores share the same instruction memory.

- Data memory is a dual-port on-chip memory with a two-cycle read latency. On each Nios II DPX MTP core, the fixed-latency master is connected to one port of the dual-port memory.
- Each of the two MTP cores provides its own fixed- and variable-latency data masters. However, each MTP must have the same view of memory, typically by connecting both master ports to each port of the dual-port memory.

Loading Nios II DPX Software in a Deployed System

During development, you typically load software into the Nios II DPX datapath processor using the debugger. Deployed systems need a different approach. The following approaches load the software without the debugger:

- Include the software in the `.sof` file, including the instruction memory, task table, and initial data memory contents. With this approach, the software can only be changed by providing a new `.sof` file.
- Load the software from flash memory through external circuitry connected to the debug access slave interface.

The following procedure is an example of the steps required when using the debug access slave to load the software in a deployed system:

1. Assert the system reset request by writing to the reset controller using the debug access slave.
2. Assert the processor soft reset by writing to the reset controller using the debug access slave.
3. Release the system reset request by writing to the reset controller using the debug access slave.
4. Preload any data sections (such as `.rdata` or `.rodata`) into data memory. The debug access slave does not provide access to data memory, so you need to devise a suitable scheme. In a single core system, you might connect the second port of the data memory as another slave on the same bus as the debug access slave. In a dual core system, or in other systems where both ports of the data memory are used, you might create a two-stage bootloader for the Nios II DPX processor to program data memory before loading the deployed software.
5. Program the instruction memory using the debug access slave.
6. Program the task table using the debug access slave.
7. Release the processor soft reset by writing to the reset controller using the debug access slave.

This chapter provides information for instantiating the Nios II DPX processor as part of a Qsys system, or for export as a stand-alone component.

Instantiating for a Qsys System

The parameter editor GUI in Qsys allows you to specify parameters for each Nios II DPX datapath processor in your system. To instantiate the processor in your Qsys design, perform the following steps:

1. On the Tools menu in the Quartus® II software, click **Qsys**.
2. On the **Component Library** tab in Qsys, expand **Processors**.
3. Double-click **Nios II DPX Datapath Processor**.

Parameter Settings

The Nios II DPX Datapath Processor parameter editor has several tabs. The following sections describe the settings available on each tab.

Nios II DPX Datapath Processor Tab

The **Nios II DPX Datapath Processor** tab lists the main settings for configuring the Nios II DPX datapath processor. Table 2–1 lists the parameters, their possible values, and their descriptions.

Table 2–1. Nios II DPX Datapath Processor Tab Parameters (Part 1 of 2)

Name	Values	Description
Nios II DPX configuration		
Enable dual-core mode	On/Off	Enables dual MTPs in the Nios II DPX datapath processor.
PEID	0 - 63	Specifies the processing element ID (PEID) of the Nios II DPX datapath processor. This value is placed in the source field of the control word for outgoing messages. Make sure the PEID is within the range of the source and destination fields of message format and message interconnect.
Enable debug unit	On/Off	Enables the debug unit, which provides common debug capabilities such as hardware breakpoints and single-stepping. The debug unit provides a mechanism to read and write registers, instruction memory, and data memory.
Enable internal JTAG PHY	On/Off	Includes an internal JTAG PHY in the Nios II DPX datapath processor. When the internal JTAG PHY is not included, an external JTAG PHY interface is provided. Disabling this option allows multiple Nios II DPX datapath processors to use a shared JTAG PHY.
Enable debug access slave interface	On/Off	Enables an Avalon-MM Slave interface which provides access to the Instruction memory, the task address table, the statistics collector, and the reset controller.
Bytes of program memory	0 - 4 MB	Configures the size of the program memory for the MTP in bytes.

Table 2-1. Nios II DPX Datapath Processor Tab Parameters (Part 2 of 2)

Name	Values	Description
Nios II DPX extension register configuration		
Extension register configuration	Configuration 0 - Configuration 14	Specifies how to allocate the 32 extension registers. Refer to Table 2-2 for the available configurations.
Number of message registers	Refer to Table 2-2 .	Displays the number of extension registers allocated as message registers based on the specified Extension register configuration .
Number of internal context registers	Refer to Table 2-2 .	Displays the number of extension registers allocated as internal context registers based on the specified Extension register configuration .
Number of input/output context registers	Refer to Table 2-2 .	Displays the number of extension registers allocated as input/output context registers based on the specified Extension register configuration .

[Table 2-2](#) shows the available extension register configurations.

Table 2-2. Extension Register Configurations

Configuration	Registers per RX/TX Message Register Bank		Registers per Internal Context Register Bank		Registers per Input/Output Context Register Bank	
0 (default)	8	r32-r39	16	r40-r55	8	r56-r63
1	8	r32-r39	8	r40-r47	16	r48-r63
2	16	r32-r47	8	r48-r55	8	r56-r63
3	16	r32-r47	16	r48-r63	0	
4	16	r32-r47	0		16	r48-r63
5	32	r32-r63	0		0	
6	8	r32-r39	0		0	
7	8	r32-r39	0		8	r40-r47
8	8	r32-r39	0		16	r40-r55
9	8	r32-r39	8	r40-r47	0	
10	8	r32-r39	8	r40-r47	8	r48-r55
11	8	r32-r39	16	r48-r63	0	
12	16	r32-r47	0		0	
13	16	r32-r47	0		8	r40-r55
14	16	r32-r47	8	r48-r55	0	



When referring to extension registers, software normally uses mnemonic names such as rx0, rather than physical names such as r32. Physical extension register names are available only in assembly language.

Multithreaded Processor Tab

The **Multithreaded Processor** tab lists the settings for configuring the Nios II DPX MTP. [Table 2-3](#) lists the parameters, their possible values, and their descriptions.

Table 2-3. Multithreaded Processor Tab Parameters

Name	Values	Description
Vector address		
Custom reset/exception/debug vector offsets	On/Off	Allows you to specify your own reset, exception, and debug addresses.
Custom exception address offset	Variable	The customized offset address in the instruction memory for the exception vector for all threads.
Exception address	Variable	The instruction memory base address plus the exception address offset. This field is read-only.
Custom break address offset	Variable	The customized offset address in the instruction memory for the break address for all threads.
Break address	Variable	The instruction memory base address plus the break address offset. This field is read-only.
Custom reset address offset	Variable	The initial PC used for all threads on their first cycle when reset is de-asserted. Make sure the customized offset address resides in instruction memory space and is the first word address in the program memory.
Reset address	Variable	The instruction memory base address plus the reset address offset. This field is read-only.
Data master		
Data master byte address bit width	8 - 31	Specifies the bit width for the fixed- and variable-latency data master byte-addressable address bus.
Enable variable-latency data master	On/Off	Enables the variable-latency data master, allowing the MTP to access peripherals which do not have a guaranteed read latency of 2 or that are shared. The variable-latency data master accesses all addresses outside the range defined for the fixed-latency data master, defined by fixed-latency data master low address and fixed-latency data master high address.
Fixed-latency data master low address	Variable	Defines the lower boundary for fixed-latency data master address range.
Fixed-latency data master high address	Variable	Defines the upper boundary for fixed-latency data master address range.
Enable data master threadinfo interface	On/Off	Enables an external Avalon-ST signal to be used with the fixed- and variable-latency data masters that provide the thread number, CID, RXID, and TXID associated with the software task performing the data access.

Message Interface Unit Tab

The **Message Interface Unit** tab lists the main settings for configuring the Nios II DPX MIU. [Table 2-4](#) lists the parameters, their possible values, and their descriptions.

Table 2-4. Message Interface Unit Tab Parameters (Part 1 of 2)

Name	Values	Description
Message unit configuration		
Context ID ordering enforcement	On/Off	Includes logic for CID order enforcement. CID order enforcement is under software control.
Context ID allocation support	On/Off	Enables the CID allocation support, allowing the MTP to request additional CIDs using the <code>cidalloc</code> command. Turning this option off disables the <code>cidalloc</code> command.
Number of reserved context IDs for allocation (per processor core)	Variable	If CID allocation support is enabled, specifies how many of the available CIDs can be allocated using the <code>cidalloc</code> command. The specified number of CIDs is reserved for each processor core. The total number of reserved CIDs is reported in Total number of reserved context IDs for allocation .
Sequence number ordering enforcement	On/Off	Includes logic for sequence number order enforcement. Sequence number order enforcement is under software control.
Advanced receive ID/transmit ID Management	On/Off	Allows you to alter the default number of RX and TX message register banks available in the system.
Number of context IDs	0 - 255	Specifies the maximum number of context IDs available in the system. The width of the Context ID field in the message format is based on this parameter.
Number of receive IDs	0 - 255	Specifies the maximum number of RX message register banks available in the system. When set to zero, RXIDs are disabled and the RX message registers are indexed by the CID.
Number of transmit IDs	0 - 255	Specifies the maximum number of TX message register banks available in the system. When set to zero, TXIDs are disabled and the TX message registers are indexed by the CID.
Number of destination IDs	0 - 255	Specifies the maximum number of destination IDs available in the system. The width of the <code>destination</code> field in the message format is based on this parameter.
Number of output task IDs	0 - 255	Specifies the maximum number of output task IDs available in the system. The width of the <code>output task</code> field in the message format is based on this parameter.
Number of source IDs	0 - 255	Specifies the maximum number of source IDs available in the system. The width of the <code>source</code> field in the message format is based on this parameter.
Number of input task IDs	0 - 255	Specifies the maximum number of input task IDs available in the system. The width of the <code>input task</code> field in the message format and the number of entries in the task address table are based on this parameter.
Number of flag bits	0 - 255	This parameter indicates the total number of flag bits supported and available in the <code>message_flags</code> extended control register. The width of the flag field in the message format is based on this parameter.
Number of sequence numbers	0 - 255	When Sequence number ordering enforcement is on, specifies the maximum number of sequence numbers available in the system.

Table 2-4. Message Interface Unit Tab Parameters (Part 2 of 2)

Name	Values	Description
Bit offset of debug flag	0 to <code>flags</code> field width -1	Specifies the location of the debug flag in the <code>flags</code> field of the PE message format control word.
Number of user message bits	Variable	Specifies the number of bits in the <code>user_message</code> field of the PE message format control word.
Width of input context register data bus	32, 64, 128	Specifies the width of the input context register data bus.
Width of output context register data bus	32, 64, 128	Specifies the width of the output context register data bus.

Memory Options (advanced) Tab

The **Memory Options** tab lists the settings for configuring the Nios II DPX datapath processor memory. [Table 2-1](#) lists the parameters, their possible values, and their descriptions.

Table 2-5. Nios II DPX Datapath Processor Tab Parameters (Part 1 of 2)

Name	Values	Description
Nios II DPX RAM configuration		
Instruction memory initialization filename (1)	ASCII, no spaces	Specifies the name of the Hexadecimal (Intel-Format) File (.hex) for initializing the memories in your system during simulation and synthesis.
Expected instruction memory initialization file	Read-only	Displays the expected file specified by Instruction memory initialization filename .
Instruction memory type	Varies by device	Specifies the type of instruction memory to use.
Task memory initialization filename (1)	ASCII, no spaces	Specifies the name of the task memory initialization file to use for simulation and synthesis.
Expected task memory initialization file	Read-only	Displays the expected file specified by Task memory initialization filename .
Task memory type	Varies by device	Specifies the type of memory to use for the task address table.
Message unit RAM configuration		
Receive message register memory type	Varies by device	Specifies the type of memory to use for the RX message registers.
Transmit message register memory type	Varies by device	Specifies the type of memory to use for the TX message registers.
Input context register memory type	Varies by device	Specifies the type of memory to use for the input context registers.
Output context register memory type	Varies by device	Specifies the type of memory to use for the output context registers.
Context register memory type	Varies by device	Specifies the type of memory to use for the internal context registers.
Sequence number FIFO memory ram block type	Varies by device	Specifies the type of memory to use for the sequence number FIFO memory ram block.
Note to Table 2-5: (1) These parameters might be important in a multiprocessor system. If the processors are intended to run different software, you must select distinct instruction and task memory initialization filenames. Otherwise, when the software tools generate memory initialization files for one processor, they might overwrite the initialization files for another processor.		

Table 2-5. Nios II DPX Datapath Processor Tab Parameters (Part 2 of 2)

Name	Values	Description
Context ID FIFO memory ram block type	Varies by device	Specifies the type of memory to use for the context ID FIFO memory ram block.
Receive message register memory ram block type	Varies by device	Specifies the type of memory to use for the receive message register memory ram block.
Transmit message register memory ram block type	Varies by device	Specifies the type of memory to use for the transmit message register memory ram block.
Receive ID queue memory ram block type	Varies by device	Specifies the type of memory to use for the RXID queue memory ram block.
Transmit ID queue memory ram block type	Varies by device	Specifies the type of memory to use for the TXID queue memory ram block.
Transmit message controller FIFO memory ram block type	Varies by device	Specifies the type of memory to use for the transmit message controller FIFO memory ram block.
Note to Table 2-5: (1) These parameters might be important in a multiprocessor system. If the processors are intended to run different software, you must select distinct instruction and task memory initialization filenames. Otherwise, when the software tools generate memory initialization files for one processor, they might overwrite the initialization files for another processor.		

External Interfaces (advanced) Tab

The **External Interfaces** tab allows you to configure the control word in your message format by assigning the base location of each control word field. Each field's width is determined by the next field's base location, with the destination field's width determined by the control word width. To remove a field, set the next field's base location to the same value, making the width equal to zero.



The default values are sufficient for most systems without adjustment.

Table 2-6 shows the Nios II DPX default PE message format control word fields and field widths.

Table 2-6. Default PE Message Control Word

32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
destination								source								taskid								CID								flags



For more information about PE message formats, refer to the *Message Format* chapter of the *Altera Event-Driven Datapath Processing Design Handbook*.

The **External Interfaces** tab lists the settings for configuring the Nios II DPX datapath processor PE message format. [Table 2-7](#) lists the parameters, their possible values, and their descriptions.

Table 2-7. External Interfaces Tab Parameters (Part 1 of 2)

Name	Values	Description
Receive message interface		
Receive message control word width	Variable	Specifies the width in bits of the RX message control word.
Receive message control destination base	0 to RX message control word width -1	Specifies the bit offset of the <code>destination</code> field in the RX message control word. The width of the field is controlled by the Number of destination IDs parameter on the Message Interface Unit tab.
Receive message control source base	0 to RX message control word width -1	Specifies the bit offset of the <code>source</code> field in the RX message control word. The width of the field is controlled by the Number of destination IDs parameter on the Message Interface Unit tab.
Receive message control input task ID base	0 to RX message control word width -1	Specifies the bit offset of the <code>input task</code> field in the RX message control word. The width of the field is controlled by the Number of input task IDs parameter on the Message Interface Unit tab.
Receive message control context ID base	0 to RX message control word width -1	Specifies the bit offset of the <code>CID</code> field in the RX message control word. The width of the field is controlled by the Number of context IDs parameter on the Message Interface Unit tab.
Receive message control flag bits base	0 to RX message control word width -1	Specifies the bit offset of the <code>flags</code> field in the RX message control word. The width of the field is controlled by the Number of flag bits parameter on the Message Interface Unit tab.
Receive message control user message bits base	0 to RX message control word width -1	Specifies the bit offset of the <code>user</code> field in the RX message control word. The width of the field is controlled by Number of user message bits on the Advanced Options tab.
Transmit message interface		
Transmit message control has same settings as receive message control	On/Off	When on, the RX message control word parameter values are also used for the TX message control word.
Transmit message control word width	Variable	Specifies the width in bits of the TX message control word.
Transmit message control destination base	0 to TX message control word width -1	Specifies the bit offset of the <code>destination</code> field in the TX message control word. The width of the field is controlled by the Number of destination IDs parameter on the Message Interface Unit tab.
Transmit message control source base	0 to TX message control word width -1	Specifies the bit offset of the <code>source</code> field in the TX message control word. The width of the field is controlled by the Number of source IDs parameter on the Message Interface Unit tab.
Transmit message control input task ID base	0 to TX message control word width -1	Specifies the bit offset of the <code>input task</code> field in the TX message control word. The width of the field is controlled by the Number of input task IDs parameter on the Message Interface Unit tab.
Transmit message control context ID base	0 to TX message control word width -1	Specifies the bit offset of the <code>CID</code> field in the TX message control word. The width of the field is controlled by the Number of context IDs parameter on the Message Interface Unit tab.

Table 2-7. External Interfaces Tab Parameters (Part 2 of 2)

Name	Values	Description
Transmit message control flag bits base	0 to TX message control word width -1	Specifies the bit offset of the <code>flags</code> field in the TX message control word. The width of the field is controlled by the Number of flag bits parameter on the Message Interface Unit tab.
Transmit message control user message bits base	0 to TX message control word width -1	Specifies the bit offset of the <code>user</code> field in the TX message control word. The width of the field is controlled by Number of user message bits on the Advanced Options tab.

Debug Statistics Tab

The **Debug Statistics** tab lists the settings for configuring the Nios II DPX statistics collector. [Table 2-8](#) lists the parameters, their possible values, and their descriptions.

Table 2-8. Debug Statistics Tab Parameters (Part 1 of 2)

Name	Values	Description
Statistics and debug counters		
Enable debug statistics	On/Off	Enables the debug statistics collector, allowing monitoring of the MTP and MIU.
Enable processor instruction counters	On/Off	Enables counting of the number of instructions executed of the following types: <ul style="list-style-type: none"> ■ Load ■ Store ■ Arithmetic/Logic ■ Compare/Branch ■ Call ■ Exit ■ Send ■ Other
Enable processor stats for context ID	On/Off	Count Unavailable in dual-core mode
Enable processor stats for input task ID	On/Off	When on, causes the statistics collector to count the number of times tasks were executed for each Task ID up to 64 Task IDs
Enable processor stats for output task	On/Off	Enables counting of the number of PE messages sent for each of the first 64 task IDs
Enable processor stats for task ticks	On/Off	Enables counting of the number of clock cycles taken for running the previous task from start to completion for each of the first 64 task IDs. These counters are not cumulative; they reset each time a new task runs.
Enable message interface unit level count for processing cycles on CID 0	On/Off	Enables counting of the number of clock cycles taken for running the previous task from start to completion for the PE message with CID = 0. These counters are not cumulative; they reset each time a new task runs.
Enable message interface unit level count stats for free context ID FIFO	On/Off	Enables monitoring of the current, maximum, and minimum levels of the free CID FIFO.

Table 2-8. Debug Statistics Tab Parameters (Part 2 of 2)

Name	Values	Description
Enable message interface unit level count stats for input task queue	On/Off	Enables monitoring of the current, maximum, and minimum levels of the input task queue.
Enable message interface unit level count stats for output task queue	On/Off	Enables monitoring of the current, maximum, and minimum levels of the output task queue.
Enable message interface unit level count stats for CID reordering level	On/Off	Enables monitoring of the current, maximum, and minimum levels of the CID reorder queue.
Enable message interface unit level count stats for sequence number message reorder level	On/Off	When Sequence number ordering enforcement on the Message Interface Unit tab is on, enables monitoring of the current, maximum, and minimum levels of the sequence number reorder queue.

Advanced Options Tab

The **Advanced Options** tab lists the settings for configuring the Nios II DPX datapath processor advanced options. [Table 2-9](#) lists the parameters, their possible values, and their descriptions.

Table 2-9. Advanced Options Tab Parameters

Name	Values	Description
Advanced option configuration		
Assign CPUID control register value manually	On/Off	Enables manually assigning the read-only value of the <code>cpuid</code> control register. When this option is off, the <code>cpuid</code> control register is set to the PEID.
Custom CPUID control register value	32-bit number	Specifies the value for the <code>cpuid</code> control register.
Actual CPUID control register value	32-bit number	When Assign CPUID control register value manually is on, displays the value for the <code>cpuid</code> control register. When off, displays the default value for the <code>cpuid</code> control register. In dual-core Nios II DPX processor systems, each MTP has the same <code>cpuid</code> .
Enable insert instruction	On/Off	Enables the Nios II DPX processor <code>insert</code> instruction.
Enable message interface control register	On/Off	Enables the extended control registers used by the MIU, namely, <code>message_flags</code> , <code>message_user</code> , <code>message_id0</code> , and <code>message_id1</code> .
Context register indexing mode	per CID, per thread	Specifies the operating mode of the context registers.
Sequence number FIFO implementation	Logic Element, FIFO	Specifies the location of the sequence number FIFO.
Thread argument pipeline implementation	Logic Element, RAM	Specifies the location of the thread argument pipeline.

Instantiating for a Stand-Alone System

You can generate a Nios II DPX processor core in Qsys with all connections exported, and then connect the processor manually in HDL. To instantiate a Nios II DPX processor for a stand-alone system, perform the following steps:

1. On the Tools menu in the Quartus II software, click **Qsys**.
2. On the File menu in Qsys, click **New System**.
3. On the **System Contents** tab, perform the following steps:
 - a. Remove the default clock component.
 - b. Use the instantiation steps and parameter settings in “[Instantiating for a Qsys System](#)” on page 2-1 to add a Nios II DPX datapath processor from the component library.
 - c. Click **Filters** and select **All** from the **Filter** list to provide access to all ports.
 - d. In the **Export As** column, click **Click to export** in each row to export all interfaces.
 - e. Optionally rename any of the interfaces to match your system design.
4. On the File menu, click **Save As**, enter a name for your system and save.
5. On the **Generation** tab, perform the following steps:
 - a. Select the simulation and synthesis files you want to generate.
 - b. Specify the output directory.
 - c. Click **Generate**.

Qsys generates a system you can connect manually in HDL. For information about the Nios II DPX interfaces, refer to “[Nios II DPX Processor Interfaces](#)” on page 1-21.



For more information about stand-alone systems, refer to the *Getting Started with the Graphical User Interface* chapter in the [Nios II DPX Software Development](#) section of the *Nios II DPX Datapath Processor Handbook*.

Nios II DPX Context Address Adapter

Using the thread information interface, data memory can be optionally indexed by CID, thread number, RXID, or TXID. The Nios II DPX Context Address Adapter expands the address bus with additional thread information, allowing access to standard memory-mapped components in a context specific way.

In Qsys, the Nios II DPX Context Address Adapter parameter editor is available on the Component Library tab under the **Processor Additions** category. [Table 2-10](#) shows the **Nios II DPX Context Address Adapter** parameters available in the parameter editor.

Table 2-10. Nios II DPX Context Address Adapter Parameters

Name	Value	Description
General		
Address Mode	Varies	Displays the composition result of the output signal.
Address Indexing Using Thread Info		
Index Thread Info in address	On/Off	Index bits of the thread information signal as part of the address.
Thread Info subsection LSB bit	0 to 31	LSB of the thread information signal to extract and put into the address. This parameter is ignored when Index Thread Info in address is not enabled.
Thread Info subsection MSB bit	0 to 31	MSB of the thread information signal to extract and put into the address. This parameter is ignored when Index Thread Info in address is not enabled.
Thread Info subsection address bit position	0 to 31	Bit offset where address bits start being replaced with the specified thread information subsection. This bit position is relative to the <code>in_address</code> bit position.
Thread Info bits inserted		
Thread Info Avalon-ST Sink Interface		
Thread Info data width	1 to 32	Bus width of Avalon-ST sink port.
Thread Info Termination with static value		
Static Thread Info	On/Off	Enables terminate thread information Avalon-ST sink port with a static value.
Thread Info static value	0x00000000 -0xFFFFFFFF	Static value used to terminate thread information Avalon-ST sink port.
Static Thread Info value		
Avalon-MM Master and Slave Interfaces		
Use read	On/Off	Enable <code>read</code> signal for all Avalon-MM interfaces.
Use readdata	On/Off	Enable <code>readdata</code> signal for all Avalon-MM interfaces.
Use write	On/Off	Enable <code>write</code> signal for all Avalon-MM interfaces.
Use writedata	On/Off	Enable <code>writedata</code> signal for all Avalon-MM interfaces.
Use byteenable	On/Off	Enable <code>byteenable</code> signal for all Avalon-MM interfaces.
Use readdatavalid	On/Off	Enable <code>readdatavalid</code> signal for all Avalon-MM interfaces.
Use waitrequest	On/Off	Enable <code>waitrequest</code> signal for all Avalon-MM interfaces.
Slave read latency	0 to 63	Read latency for Avalon-MM slave.
Maximum pending read	0 to 64	Maximum pending read for the Avalon-MM In slave.
Slave address width	On/Off	Address extension Avalon-MM slave address width.
Slave data width	8, 16, 32, 64, 128, 256, 512, 1024	Avalon-MM data width.
Slave byteenable width	Varies	Displays the width of the byte enable portion of Slave data width .
Master address width	Varies	Displays the total width of the slave address bus and the inserted thread information bits.

The Nios II DPX datapath processor is an event-driven multithreaded processor. The DPX datapath processor processes external events, and can delegate processing tasks to other PEs in the system. Hence debugging or verifying a Nios II DPX datapath processor solution requires system-level debugging and verification strategies.

This chapter presents the following strategies for performing system-level debug and verification of Nios II DPX datapath processor systems:

- Register transfer level (RTL) simulation
- Packet debug for a complete input-to-output context processing debug and verification
- Tools for debugging and verifying custom PEs

RTL Simulation

RTL simulation is a powerful means of debugging the interaction between the Nios II DPX datapath processor and various PEs in a system. RTL simulation of a Nios II DPX processor system requires the following steps:

- Generate a Verilog HDL simulation model of your Qsys system.
- Create a suitable testbench which exercises your design.
- Build your Nios II DPX software project, and generate **.hex** files for initializing the memories in your system at the beginning of your simulation.
- Create a simulation script that builds and runs your simulation.
- Debug your simulation by examining generated waveforms.

The following sections describe how to carry out these steps using the ModelSim® simulator and the Altera Complete Design Suite.

Simulation Model, Testbench and Initialization Files

You create your Nios II DPX simulation model and testbench using the steps that apply to any Qsys design.

-  Refer to “Qsys Design Flow” in the *Creating a System with Qsys* chapter in Volume 1 of the *Quartus II Handbook*.

You create memory initialization files using the Nios II SBT for Eclipse.

-  Refer to “Memory Initialization Files” on page 6–46.

Create a Simulation Script for ModelSim

The following command can be added to a ModelSim do script to compile your Nios II DPX processor Qsys system:

```
vlog -sv <path to Qsys file>/my_system/sim_verilog/my_system.v \
+libext+.v +libext+.sv +libext+.vo \
+incdir+<path to Qsys file>/my_system/sim_verilog/submodules \
-y <path to Qsys file>/my_system/sim_verilog \
-y <path to Qsys file>/my_system/sim_verilog/submodules \
-y <path to Qsys file>/my_system/sim_verilog/submodules/mentor \
-y <path to Qsys file>/my_system/sim_verilog/submodules/common
```



You might also wish to enhance this script to automatically copy the .hex files from the **mem_init** directory.

Record Suitable Waveforms

In addition to the input and output signals of the Nios II DPX core, you can add additional signals that allow you to monitor the program counter of your Nios II DPX processor. Use this information in conjunction with disassembly of your program to monitor the flow of your program in an RTL simulation. A suitable disassembly is provided in the **.objdump** file in the directory where you built your software.

For single-core systems, add the following signals from the Nios II DPX processor core top level:

- message_unit_thread_dispatch_channel
- nios2dpx_mtp_processor_0_fixed_latency_instruction_master_address
- nios2dpx_mtp_processor_0_fixed_latency_instruction_master_readdata



The **message_unit_thread_dispatch_channel** signal contains the thread number associated with the instruction address and data in the same cycle.

For dual-core systems, add the following signals from the Nios II DPX processor core top level:

- message_unit_thread_dispatch0_channel
- nios2dpx_mtp_processor_0_fixed_latency_instruction_master_address
- nios2dpx_mtp_processor_0_fixed_latency_instruction_master_readdata
- message_unit_thread_dispatch1_channel
- nios2dpx_mtp_processor_1_fixed_latency_instruction_master_address
- nios2dpx_mtp_processor_1_fixed_latency_instruction_master_readdata

Performance Monitoring

You can use Altera's System Console debug facilities to collect and report runtime statistics from your system. These statistics provide important information about the functionality and performance of your system.

The Nios II DPX datapath processor contains circuitry for collecting suitable statistics, and software for collecting these statistics is provided with the Nios II DPX Packet Processing design example.



For information about processor statistics, refer to the *Getting Started with the Nios II DPX Datapath Processor Tutorial*.

You can also add statistics capture facilities to your own PEs. System Console's statistics capture facilities can periodically sample data from any circuitry that provides an Avalon-MM slave port. You can display these statistics within the System Console GUI by means of a simple Tcl script.

Packet Debug

The PE message format defines a debug flag bit within its control word. The Nios II DPX debugger allows entering debug mode on receiving PE messages with the debug flag set. This feature allows debugging all PE messages processed by the Nios II DPX datapath processor for a specific context, allowing context-specific system debug and verification. The following sections provide information on this packet debug capability.

Debug Flag Bit

The debug flag is a single bit in the `flags` field in the control word of your PE messages. You specify the position of this flag with the **Bit offset of debug flag** parameter when instantiating the Nios II DPX processor. For more information, refer to “*Message Interface Unit Tab*” on page 2-4 and “*External Interfaces (advanced) Tab*” on page 2-6.

Using the Nios II DPX debugger, you can instruct the Nios II DPX datapath processor to stop on the first instruction of any task that is executed for a PE message with an enabled debug flag. Only those threads that are scheduled to execute tasks for PE messages with enabled debug flags enter debug mode; other threads are not affected and continue to remain in the state they were.

The received PE message's flag bits are loaded in `message_flags` extended control register. The contents of this register are used as the flag bits of transmit messages. You can modify the contents of the `message_flags` extended control register through software or by using the Nios II DPX debugger while the thread is suspended.



For more information about extended control registers, refer to the *Software Programming Model* chapter in the *Nios II DPX Software Development* section of the *Nios II DPX Datapath Processor Handbook*.



For more information about the processor's debug flag breakpoint capability from the Nios II DPX debugger, refer to the *Getting Started with the Graphical User Interface* chapter in the *Nios II DPX Software Development* section of the *Nios II DPX Datapath Processor Handbook*.

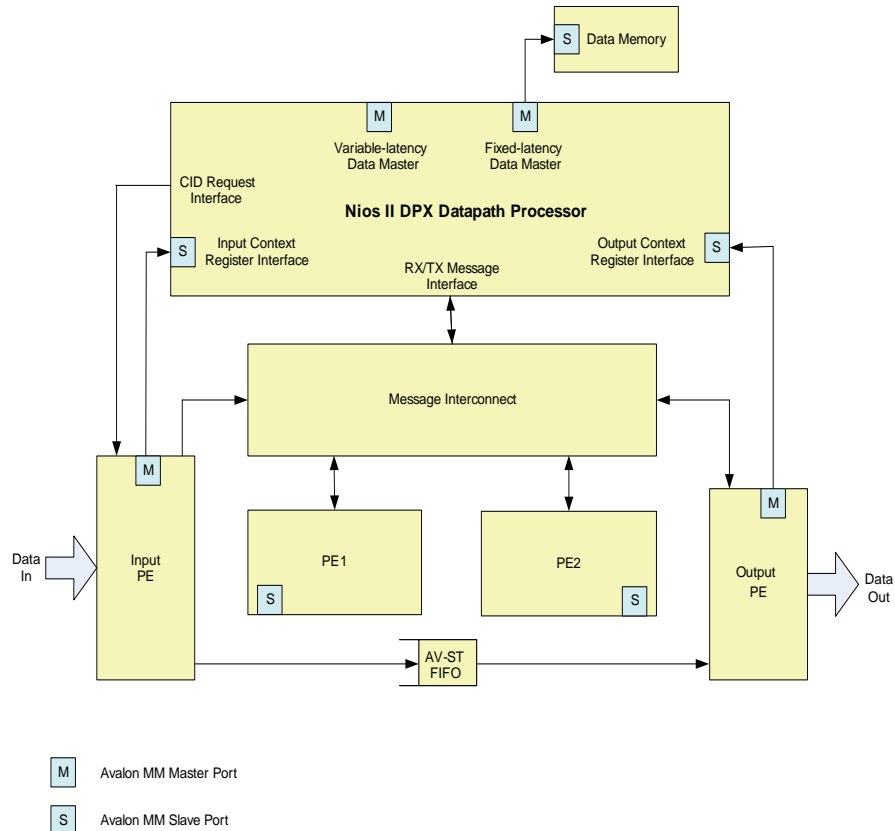
PEs and the Debug Flag

PEs have access to the debug flag value in the received messages and can perform custom debug actions if the debug flag is set. When transmitting messages, the PEs provide a value for the debug flag. Normally, PEs transfer the debug flag value from the received PE message to the transmitted PE message. However, some debugging scenarios might require modifying the debug flag value.

Using Debug Flag Breakpoint Capability

Figure 3–1 shows a sample Nios II DPX datapath processor system. The input PE receives an external stimulus and generates PE messages for the system. The output PE generates its output when it receives a PE message indicating context processing is complete. The Nios II DPX datapath processor executes application software to process messages in the system. It can delegate work to the two example hardware PEs.

Figure 3–1. Sample Nios II DPX Datapath Processor System



The following sections illustrate some of the possible message debugging scenarios. The scenarios described are not exhaustive and other methods that do not rely on the debug flag are possible.

Context Processing Debug

Nios II DPX datapath processor's debug flag breakpoint capability provides the ability to debug all tasks involved in processing a specific context.

By enabling the debug flag bit when processing a PE message received from the input PE, all PE messages generated by the Nios II DPX datapath processor using the same context as the input PE message have their debug flag enabled. As long as all PEs transfer the debug flag bit from the incoming PE message to the transmitted PE messages generated in context of the received PE message, all PE messages that are generated within the system in response to the input PE message have their debug flag enabled. For all PE messages with the debug flag bit set, namely, all PE messages in reference to a specific context, the Nios II DPX datapath processor threads scheduled for processing these context specific PE messages enter the debug mode. This method provides a complete context processing debug.

The following steps show a way to perform context processing debugging:

1. Enable the debug flag hardware breakpoint feature with the `monitor set_packet_debug` command in System Console.
2. From the Nios II DPX debugger, set a breakpoint at the task that the processor executes when the processor receives the PE message of interest from the input PE.
3. Stimulate the input PE to generate a suitable PE message. The thread that is scheduled to execute the task stops at the breakpoint.
4. Using the Nios II DPX debugger, enable the debug flag in the `message_flags` extended control register. As a result, any PE message generated by this task has its debug flag set.
5. Disable the task-specific breakpoint set in step 1.

As long as other PEs transfer the debug flag value to the messages generated for this context, the Nios II DPX processor threads that are scheduled to execute tasks to process PE messages in this context enter debug mode.

To perform context processing debugging for a specific context that can happen at an arbitrary time, design the input PE to identify the context of interest and set the debug flag bit for the PE message it generates in reference to the context. For example, in a packet processing system, you could set the debug flag for all packets of a particular type.

You can also use hardware breakpoints in a thread-specific way. In addition to the previous steps, you can note the thread number in use and set a thread-specific hardware breakpoint.



For more information about context-specific debugging and breakpoints, refer to the *Getting Started with the Graphical User Interface* chapter in the *Nios II DPX Software Development* section of the *Nios II DPX Datapath Processor Handbook*.

PE Message Debug

Some situations might require debugging a specific PE message, or debugging the task triggered by a specific message. In this case, enable the debug flag when sending the message of interest. The Nios II DPX processor enters debug mode when the corresponding message is received (assuming the PEs involved pass on the debug flag).

Hardware PE Debug

The following list suggests some strategies to provide debug access to hardware PEs:

- Implementing Avalon-MM slave access—Design your PEs to contain status and control registers for monitoring or controlling debug or verification features specific to the PE. Make these registers accessible over an Avalon-MM slave interface for access by an Avalon-MM master such as a Nios II processor for embedded monitoring and control, or by a PC host using the JTAG to Avalon Master Bridge IP core.
- Observing RTL signals—Use the Altera SignalTap® II Logic Analyzer to monitor signals that are vital for controllability and observability when generating the device bit stream. These signals are listed in [“Record Suitable Waveforms” on page 3-2](#).

This chapter provides additional information about the document and Altera.

Document Revision History

The following table shows the revision history for this document.

Date	Version	Changes
May 2011	2.0	Updated for ACDS v11.0
December 2010	1.0	Initial release

How to Contact Altera

To locate the most up-to-date information about Altera products, refer to the following table.

Contact (1)	Contact Method	Address
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	custrain@altera.com
Product literature	Website	www.altera.com/literature
Non-technical support (General) (Software Licensing)	Email	nacomp@altera.com
	Email	authorization@altera.com









Note to Table:

(1) You can also contact your local Altera sales office or sales representative.

Typographic Conventions

The following table shows the typographic conventions this document uses.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Indicate command names, dialog box titles, dialog box options, and other GUI labels. For example, Save As dialog box. For GUI elements, capitalization matches the GUI.
bold type	Indicates directory names, project names, disk drive names, file names, file name extensions, software utility names, and GUI labels. For example, \qdesigns directory, D: drive, and chiptrip.gdf file.
<i>Italic Type with Initial Capital Letters</i>	Indicate document titles. For example, <i>Stratix IV Design Guidelines</i> .
<i>italic type</i>	Indicates variables. For example, $n + 1$. Variable names are enclosed in angle brackets (< >). For example, <file name> and <project name>.pof file.

Visual Cue	Meaning
Initial Capital Letters	Indicate keyboard keys and menu names. For example, the Delete key and the Options menu.
“Subheading Title”	Quotation marks indicate references to sections within a document and titles of Quartus II Help topics. For example, “Typographic Conventions.”
Courier type	Indicates signal, port, register, bit, block, and primitive names. For example, <code>data1</code> , <code>tdi</code> , and <code>input</code> . The suffix <code>n</code> denotes an active-low signal. For example, <code>resetn</code> . Indicates command line commands and anything that must be typed exactly as it appears. For example, <code>c:\qdesigns\tutorial\chiptrip.gdf</code> . Also indicates sections of an actual file, such as a Report File, references to parts of files (for example, the AHDL keyword <code>SUBDESIGN</code>), and logic function names (for example, <code>TRI</code>).
	An angled arrow instructs you to press the Enter key.
1., 2., 3., and a., b., c., and so on	Numbered steps indicate a list of items when the sequence of the items is important, such as the steps listed in a procedure.
	Bullets indicate a list of items when the sequence of the items is not important.
	The hand points to information that requires special attention.
	A question mark directs you to a software help system with related information.
	The feet direct you to another document or website with related information.
	A caution calls attention to a condition or possible situation that can damage or destroy the product or your work.
	A warning calls attention to a condition or possible situation that can cause you injury.
	The envelope links to the Email Subscription Management Center page of the Altera website, where you can sign up to receive update notifications for Altera documents.

The *Nios® II DPX Software Development* section of the *Nios II DPX Datapath Processor Handbook* provides the basic information needed to develop software for the Altera® Nios II DPX MTP. This section describes the Nios II DPX MTP software development environment, the Altera Embedded Design Suite (EDS) tools available to you, and the process for developing software.

The *Nios II DPX Software Development* section assumes you have a basic familiarity with embedded processor concepts.

Familiarity with Altera hardware development tools can give you a deeper understanding of the reasoning behind the Nios II DPX MTP software development environment.

This section includes the following chapters:


- Chapter 4, Overview of the Nios II DPX MTP
- Chapter 5, Software Programming Model
- Chapter 6, Getting Started with the Graphical User Interface
- Chapter 7, Getting Started from the Command Line
- Chapter 8, Understanding the Nios II DPX Board Support Package
- Chapter 9, Nios II DPX MTP Instruction Set and Application Binary Interface
- Chapter 10, SBT Reference for the Nios II DPX MTP

This chapter provides a high-level overview of how the Nios II DPX MTP fits into the Nios II DPX datapath processor. It outlines the software development environment for the Nios II DPX MTP. This chapter contains the following sections:

- “The MTP in the Context of the Nios II DPX Datapath Processor”
- “Event-Driven Processing” on page 4-1
- “Nios II DPX Multithreading” on page 4-2
- “Dual-Processor Configurations” on page 4-2
- “Nios II DPX Programming Considerations” on page 4-3
- “The Nios II DPX Software Development Environment” on page 4-4

The MTP in the Context of the Nios II DPX Datapath Processor

The Nios II DPX MTP is a dedicated, special-purpose microprocessor embedded in the Nios II DPX datapath processor. The DPX MTP is a submodule in the Nios II DPX core, which includes instruction memory and hardware support for task and thread control, messaging, system analysis and debugging. The Nios II DPX MTP and its environment are optimized for datapath processing tasks, such as packet processing.


 For additional information about the Nios II DPX MTP and its hardware environment, refer to the *Nios II DPX Architecture* chapter, in the *Nios II DPX Hardware Reference* section of the *Nios II DPX Datapath Processor Handbook*.

Event-Driven Processing

Nios II DPX designs are based on a event-driven processing paradigm. The programming model for the Nios II DPX MTP is different from that of a conventional processor. Generally, to program the Nios II DPX MTP, you do not write a `main()` function that executes for the lifetime of the application. Instead, `main()` typically only executes some minimal initialization tasks and returns. The Nios II DPX software consists of short routines, called tasks, that are executed in response to the receipt of a PE message.

A task is analogous to an interrupt service routine (ISR). However, a task provides better performance than a conventional ISR, because all necessary context information is provided to the processor through the message interface hardware.

In a Nios II DPX system, processing elements (PEs) are connected to one another through a message interconnect. The Nios II DPX datapath processor is an example of a PE. PEs can also be specialized hardware accelerators or other processors with message interfaces. Each PE is capable of performing one or more tasks. In a typical system, the Nios II DPX datapath processor communicates with a heterogenous collection of several PEs.

 For information about PEs and Altera event-driven datapath processing, refer to “Event-Driven Methodology” in the *Altera Event-Driven Datapath Processing Design Handbook*.


Nios II DPX Multithreading


The Nios II DPX MTP is an interleaved multithreaded processor, capable of executing eight threads simultaneously. Each thread can execute a task.

Each thread has its own register context, enabling threads to execute independently. You can think of the threads as eight identical, separate processors.

Each thread is independent of the other threads. If a thread stalls, the remaining threads continue to execute as usual. Thread stalls, however, are rare, because of hardware features such as interleaved multithreading and fixed-latency memory masters.

MTP hardware threads are interchangeable. Software normally need not determine which hardware thread it is running on at any particular time. However, the current thread number can be read from the `threadnum` control register if necessary.


 Because the threads are independent, you must use care with shared resources, just as in any multithreaded programming environment. You must protect shared resources, such as device registers, with a mutual exclusion mechanism, such as semaphores.

 For detailed information about Nios II DPX multithreading, refer to “Functional Description” in the *Nios II DPX Architecture* chapter, in the *Nios II DPX Hardware Reference* section of the *Nios II DPX Datapath Processor Handbook*.

Dual-Processor Configurations

If your application requires more than eight threads, you can configure the Nios II DPX datapath processor with dual MTP cores. A dual-core Nios II DPX datapath processor functions the same as a single-core processor, except that it supports sixteen simultaneous threads. The two cores have identical memory maps, and run the same software. When a task needs to run, it can run on either MTP core.

A dual-processor configuration, which providing sixteen threads, uses less memory than two distinct Nios II DPX MTPs, because the two MTPs share instruction memory.

 For information about instantiating a dual-core Nios II DPX datapath processor, refer to the *Instantiating the Nios II DPX Datapath Processor* chapter, in the *Nios II DPX Hardware Reference* section of the *Nios II DPX Datapath Processor Handbook*.

Nios II DPX Programming Considerations

The event-driven paradigm, and the Nios II DPX MTP's unique architecture and hardware environment, make programming the MTP different from programming a general-purpose processor. Before you start, become familiar with the *Altera Event-Driven Datapath Processing Design Handbook*, as well as "Functional Description" in the *Nios II DPX Architecture* chapter, in the *Nios II DPX Hardware Reference* section of the *Nios II DPX Datapath Processor Handbook*. This topic provides a detailed description of the Nios II DPX architecture.

Memory and I/O

The Nios II DPX MTP has separate address spaces for instructions and data.

Instruction memory is an on-chip memory embedded in the Nios II DPX core, ensuring that threads never stall waiting for an instruction.

The Nios II DPX MTP possesses two kinds of data master interfaces: fixed-latency and variable-latency.

The fixed-latency data master interfaces provide access to data that requires time-critical access without stalling. The Nios II DPX MTP can use fixed-latency data master interfaces to connect to any memory or peripheral that has zero wait states and a read latency of two.

Variable-latency data master interfaces can access any Avalon-MM slave interface. Access through a variable-latency data master always causes a stall for at least one cycle.

The Nios II DPX MTP does not support caching. Whether or not a particular memory access stalls is strictly determined by the type of master interface used and the characteristics of the slave interface.

All Nios II DPX memory accesses use the byte-invariant big-endian convention (BE-8).



For details about how Nios II DPX memories are instantiated and configured, refer to the *Instantiating the Nios II DPX Datapath Processor* chapter, in the *Nios II DPX Hardware Reference* section of the *Nios II DPX Datapath Processor Handbook*.

The Nios II DPX Debug Interface

The Nios II DPX MTP interfaces to a debug unit, enabling the Nios II DPX debugger to provide typical debugging features such as breakpoints, single-stepping, and viewing registers and memory. The debugger interface enables nonstop debugging, in which the debugger can selectively debug one thread while the other threads continue to run normally.

For details about the Nios II DPX debugger, refer to [Chapter 6, Getting Started with the Graphical User Interface](#).

Exception Controller

The Nios II DPX MTP supports a small set of exception types. For information about working with Nios II DPX exceptions, refer to [Chapter 5, Software Programming Model](#).

Once a task is launched on a hardware thread, it must run to completion. There is no way to preempt or interrupt a running task.

The Nios II DPX Software Development Environment

The Nios II EDS provides a consistent software development environment that works for all Nios II DPX systems. With the Nios II EDS running on a host computer, an Altera FPGA, and a JTAG download cable (such as an Altera USB-Blaster™ download cable), you can write programs for and communicate with any Nios II DPX system. The Nios II DPX datapath processor debug module provides a single, consistent method to connect using a JTAG download cable. Therefore, you do not need to spend time manually creating interface mechanisms for the embedded processor.

The Nios II EDS includes proprietary and open-source tools (such as the GNU C/C++ tool chain) for creating Nios II DPX programs. The Nios II EDS automates board support package (BSP) creation for Nios II DPX MTP-based systems, eliminating the need to spend time manually creating BSPs. The BSP provides a C runtime environment, insulating you from the hardware in your embedded system. Nios II DPX BSPs contain the Altera lightweight hardware abstraction layer (LWHAL) and simple device drivers.

The Nios II SBT Development Flow

A development flow is a way of using a set of development tools together to create a software project. The Nios II EDS provides the Nios II Software Build Tools (SBT) development flow for creating Nios II programs. This development flow provides the following user interfaces:

- The Nios II SBT command line
- The Nios II SBT for Eclipse™ graphical user interface (GUI)

The Nios II SBT allows you to create Nios II DPX software projects, with detailed control over the software build process. The same Nios II SBT utilities, scripts and Tcl commands are available from both the command line and the Nios II SBT for Eclipse.

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.



For information about Tcl scripting, refer to the [Nios II Software Build Tools](#) chapter of the *Nios II Software Developer's Handbook*.

The Nios II SBT for Eclipse

The Nios II SBT for Eclipse is a GUI that runs the Nios II SBT utilities and scripts, presenting a unified development environment. You can accomplish all software development tasks within Eclipse, including creating, editing, building, running, debugging, and profiling programs.

The Nios II SBT for Eclipse is based on the Eclipse 3.5 framework and the Eclipse C/C++ development toolkit (CDT) 6.0 plugins. The Nios II 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 Nios II Command Shell.



For details about the Nios II SBT for Eclipse, refer to [Chapter 6, Getting Started with the Graphical User Interface](#). For details about Eclipse, visit the Eclipse Foundation website (www.eclipse.org).

The Nios II SBT Command Line

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

The Nios II SBT command line flow is useful if you have a large multiprocessor project maintained in a source control system. You can create and build software projects from the command line or a shell script, enabling you to integrate the software build process with your other tools.

For further information about the Nios II SBT in command-line mode, refer to [Chapter 7, Getting Started from the Command Line](#).

To debug a command-line program, you can import your SBT projects to Eclipse. You can then further edit, rebuild, run, and debug your imported project in Eclipse. Alternatively, you can use any of several other system debugging tools, including the following tools:

- GDB at the command line
- System Console
- SignalTap



For detailed information about debugging Nios II DPX systems, refer to the *System Verification* chapter in the [Nios II DPX Hardware Reference](#) section of the *Nios II DPX Datapath Processor Handbook*.

Nios II DPX Programs

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

The Nios II SBT creates software projects for you. Each project is based on a makefile. This section discusses makefiles and projects.



The C++ language is not supported in the Nios II DPX MTP software development environment.

Makefiles and the SBT

The makefile is the central component of a Nios II DPX software project, whether the project is created with the Nios II SBT for Eclipse, or at the command line. The makefile describes all the components of a software project and how they are compiled and linked.

As a key part of creating a software project, the SBT creates a makefile for you. Nios II DPX projects are sometimes called “user-managed,” because you, the user, are responsible for the content of the project makefile. You use the Nios II SBT to define the contents of the makefile.



The *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook* provides detailed information about creating makefiles.

Nios II DPX Software Project Types

The following sections describe the project types that constitute a Nios II DPX program.

Application Project

A Nios II DPX C application project consists of a collection of source code, plus a makefile. A typical characteristic of a Nios II DPX application is that one of the source files contains function `main()`, while other files contain tasks. An application can include code that calls functions in libraries and BSPs. The makefile compiles the source code and links it with a BSP and an optional library or 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 Nios II DPX BSP project is a specialized library containing system-specific support code. A BSP provides a software runtime environment customized for a processor instance in a hardware system. The Nios II EDS provides tools to modify settings that control the behavior of the BSP.

A BSP for the Nios II DPX MTP contains the following elements:

- Lightweight hardware abstraction layer—For information, refer to [Chapter 8, Understanding the Nios II DPX Board Support Package](#).
- Device drivers—For information, refer to “Software Projects” in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

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. For example, if the Nios II EDS 10.1 is installed on the Windows operating system, the root directory might be **c:\altera\10.1\nios2eds**.

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

This chapter describes the Nios II DPX software programming model, including event-driven programming on the Nios II DPX datapath processor, and how tasks, events, and messages function. The purpose of this chapter is to enable you to design and write Nios II DPX software correctly.

This chapter discusses processor features at the assembly language level, as well as the mechanics and syntax of creating tasks and sending PE messages. It describes how the Nios II DPX datapath processor interacts with other PEs (components of the Nios II DPX system lying outside the Nios II DPX datapath processor).

Fully understanding the contents of this chapter requires prior knowledge of computer architecture, software processes and process management, exception handling, and instruction sets.



For a general introduction to datapath processing concepts, refer to the *Altera Event-Driven Datapath Processing Design Handbook*.

Overview of the Nios II DPX MTP

The main component in the Nios II DPX datapath processor is the Nios II DPX multithreaded processor (MTP). To software, the MTP appears as multiple processors called threads. Each thread contains its own register bank and executes independently of the other threads.

Some programming elements, such as instruction memory and a control register that contains the processor ID, are shared by all threads. Some elements, such as program counters (PC), general-purpose registers, control registers, and stack, exist separately for each thread. And some elements, such as extension registers, exist separately based on other criteria.

The MTP is streamlined for efficient, deterministic processing. There are no operating mode options, no interrupt support, no memory protection or memory management units. The processor supports a minimal set of exception types; software is not expected to make significant use of exceptions.

The Event-Driven Programming Model

This section introduces key concepts of event-driven programming on the Nios II DPX datapath processor.

Tasks, PE Messages and Events

In event-driven programming, work is carried out by tasks. A task is a series of steps performed on data, triggered by an event. A task can be implemented either in hardware or software. This chapter describes software tasks and how software tasks interact with hardware tasks.

Each task has a clear-cut start and finish. There are no infinite loops or wait loops in tasks. The task starts, executes the necessary steps, and then terminates.

Work flows between tasks by means of PE messages. PE messages carry status, and indicate what is to be done next. The next task is specified by the unique task ID. Tasks also possess data, referred to as *context*. Typically, a PE message indicates the context data by reference. For more information about context, see “Context Data” on page 5-4.

A set of tasks, interoperating by means of PE messages, constitute a complete application.

An event is defined as a significant occurrence at a particular point in time. The most common events are the receipt or transmission of a PE message. However, custom hardware events, particularly the arrival of valid data on system inputs, also occur and must be handled.

The order of events in an event-driven system determines the order in which tasks are carried out. The start of a task is triggered by an event.

Thus, the simplest possible event-driven flow resembles the following:

1. Task executes its steps.
2. Task sends a PE message.
3. Task terminates.
4. A messaging network delivers the message to its intended recipient, initiating the next task.
5. Sequence repeats from step 1.

In an actual system, many tasks execute in parallel, and their relationships can be more complex than the sequence shown here.

Task-Based Software

Nios II DPX datapath processor software consists of short task routines that are executed in response to the receipt of a PE message.

Software Tasks

You write a task as a function in C, or as a subroutine-like block of code in assembly language. Special syntax identifies a C function as a task. For details about task syntax, refer to “Writing Task Code” on page 5-11.

Your task can call ordinary, non-task C functions or assembly language subroutines. Write such functions or subroutines just as you would for any ordinary processor architecture.


A software task executes on a thread in the Nios II DPX MTP.



For more information about threads, refer to “Multithreaded Processor” in the *Nios II DPX Architecture* chapter, in the *Nios II DPX Hardware Reference* section of the *Nios II DPX Datapath Processor Handbook*.



Any task can execute on any thread. If the Nios II DPX datapath processor is configured with a dual core, any task can execute on either of the two Nios II DPX MTP cores. Threads are interchangeable. Which task is running on which thread at any given time is of no significance to the behavior of the software.

 Dual-core Nios II DPX systems are discussed in “Nios II DPX Datapath Processor Dual-Core Configuration” in the *Nios II DPX Architecture* chapter, in the *Nios II DPX Hardware Reference* section of the *Nios II DPX Datapath Processor Handbook*.

Each task is dispatched by the Nios II DPX hardware when a PE message identifies that task as the next to be executed. Any required arguments are preloaded into special extended hardware registers. See “Using the Nios II DPX Extension Registers” on page 5–19 for more information about the extended registers.

If no thread is available when an event occurs, the Nios II DPX datapath processor stores the message and waits until one of the tasks terminates and frees its thread.

Once a task is started on a hardware thread, it runs to completion. There is no way to preempt or interrupt a running task.

The Role of the main() Function

An event-driven C application has a function `main()`, like any other C program. However, the role of `main()` is significantly simpler.

`main()` runs on each thread at startup time. In a Nios II DPX application, `main()` performs any global initialization required by all software tasks, and then exits. All subsequent software processing is performed by the tasks.

PE Messages

A PE message carries control information, arguments and potentially other data between components in the system. PE messages can be transmitted in response to events. The receipt of a PE message is an event.

The Nios II DPX datapath processor includes hardware optimized to accelerate PE message passing. Special registers and instructions provide direct access to message arguments. For more information about these registers and instructions, see “Using the Nios II DPX Extension Registers” on page 5–19 and “Task-Related Instructions” on page 5–14.

Receiving PE Messages

Each PE message identifies the next task to be executed. This task might be a software task implemented on a Nios II DPX MTP, or a hardware task implemented on a PE external to the Nios II DPX datapath processor.

Each PE message also identifies context data, if it is needed. See “Context Data” on page 5–4 for more information about context data in messages.

The information in each PE message is copied by the hardware directly into processor registers, eliminating the need to load message arguments from data memory.

Sending PE Messages

When the task finishes, it normally designates the next task to execute. The next task is designated in a PE message.

To send a PE message, you move any required message data into the transmit message registers, and execute the `snd` or `sndi` instruction.

Every task must send at least one PE message, typically when it terminates.

Context Data

Context is information that is shared among a defined set of task instances.

The Nios II DPX datapath processor can be configured to use the CID to manage context data, enabling tasks to share context with minimal overhead. The CID is a token held by one task at a time. The CID enables the task that holds it to read and modify the context data, while ensuring that no other task attempts to read or modify it concurrently.

Typically, the CID indexes into a shared memory containing application-specific data such as packet data (header and payload). Context can also include cached data such as a packet header if a PE is expected to perform many tasks on the same packet. The significance of the CID is application-specific.

The number of available CIDs is specified when the Nios II DPX hardware system is generated. Since the number of CIDs is finite, you must take several CID management issues into consideration, especially when using CID ordering. For information about CID management, see [“Context Management” on page 5-16](#).

Nios II DPX Registers

The MTP register set includes general-purpose registers, extension registers, control registers, and extended control registers. Some registers are part of the MTP itself, while others are extensions of the MTP. This section discusses each register type.

See [“Developing Software Tasks for the Datapath Processor” on page 5-10](#) for descriptions of how to use the registers to send and receive messages.

General-purpose Registers

For each thread, the base MTP architecture provides thirty-two 32-bit general-purpose registers, r0 through r31, as shown in [Table 5-1](#). Some registers have names recognized by the assembler. For example, the zero register (r0) must be configured at startup time to return the value zero. The ra register (r31) holds the return address used by procedure calls and is implicitly accessed by the call, callr and ret instructions. C compilers use a common procedure-call convention, assigning specific meaning to registers r1 through r23 and r26 through r28.

Table 5-1. MTP General-purpose Registers (Part 1 of 2)

Register	Name	Function	Register	Name	Function
r0	zero	0x00000000 (1)	r16		Callee-saved register
r1	at	Assembler temporary	r17		Callee-saved register
r2		Return value	r18		Callee-saved register
r3		Return value	r19		Callee-saved register
r4		Register arguments	r20		Callee-saved register
r5		Register arguments	r21		Callee-saved register
r6		Register arguments	r22		Callee-saved register
r7		Register arguments	r23		Callee-saved register
r8		Caller-saved register	r24		Callee-saved register
r9		Caller-saved register	r25	bt	Breakpoint temporary (2)

Table 5–1. MTP General-purpose Registers (Part 2 of 2)

Register	Name	Function	Register	Name	Function
r10		Caller-saved register	r26	gp	Global pointer
r11		Caller-saved register	r27	sp	Stack pointer
r12		Caller-saved register	r28	fp	Frame pointer
r13		Caller-saved register	r29	ea	Exception return address
r14		Caller-saved register	r30	ba	Breakpoint return address
r15		Caller-saved register	r31	ra	Return address

Notes to Table 5–1:

- (1) This value must be configured by software at startup. Software must not write any nonzero value to this register.
- (2) r25 is used exclusively by the debug module.

For more information, refer to “The Nios II DPX MTP Application Binary Interface” on page 9–112.

Extension Registers

The MTP architecture provides an interface allowing the Nios II DPX datapath processor to add additional banks of thirty-two 32-bit registers as extension registers. These registers are normally referenced with the mnemonic register names listed in Table 5–14 on page 5–19. In assembly language, they can also be referenced with the physical register names r32 through r63.

Accessing Extension Registers

Accessing the extension registers from software is very similar to accessing the general-purpose registers, with the following exceptions:

- Write-only extension registers (TX message and output context) can only be the destination register in assembly language instructions.
- Read-only extension registers (RX message and input context) can only be the first source register in assembly language instructions.
- No extension register can be the second source register in assembly language instructions.
- Some assembly language instructions have further restrictions. For more information, refer to the instruction descriptions in “The Nios II DPX MTP Instruction Set” on page 9–1.



For information about the use of extension registers by the Nios II DPX datapath processor, refer to “Functional Blocks” in the *Nios II DPX Architecture* chapter, in the *Nios II DPX Hardware Reference* section of the *Nios II DPX Datapath Processor Handbook*. For information about programming extension registers in your design, refer to “Developing Software Tasks for the Datapath Processor” on page 5–10.

Control Registers

Control registers report the status and control the behavior of the processor. Control registers are accessed differently than the general-purpose registers. The special instructions `rdctl` and `wrctl` provide the only means to read and write to the control registers.



When writing to control registers, all undefined bits must be written as zero.

The base MTP architecture provides two control registers. Table 5–2 shows details of the defined control registers. All defined control registers have names recognized by the assembler.

Table 5–2. Control Register Names and Descriptions

Register	Name	Register Contents
5	<code>cpuid</code>	Unique processor identifier
16	<code>threadnum</code>	Thread number

The following sections describe the defined control registers of the base MTP architecture.

The `cpuid` Register

The `cpuid` register holds a constant value that uniquely identifies each processor in a multiprocessor system. The `cpuid` value is determined at system generation time. The `cpuid` register is read-only; writing to the register has no effect.



Each MTP in a dual-core Nios II DPX processor must have the same `cpuid`.

The `threadnum` Register

The `threadnum` register holds the thread number of the associated thread. The `threadnum` register is read-only; writing to the register has no effect.



The `threadnum` register is a per-thread control register. There is one `threadnum` register for each thread in the processor.

Table 5–3 shows the layout of the `threadnum` register.

Table 5–3. `threadnum` Control Register Fields (Note 1)

31	...	N	N-1	1	0
0			THREADNUM		

Notes to Table 5–3:

(1) N represents the number of bits needed to represent the total number of threads in the MTP, namely, $\log_2(\text{number of threads})$.

Table 5-4 shows details of the fields defined in the `threadnum` register.

Table 5-4. threadnum Control Register Field Descriptions

Field	Description	Access	Reset	Available
THREADNUM	THREADNUM is the thread number field. The width of this field is variable and needs to contain enough bits to encode the number of threads. For example, for eight threads, this field must be at least three bits wide or wider.	Read	Thread number	Per thread

Extended Control Registers

The MTP architecture provides an interface that allows the Nios II DPX datapath processor to add additional control registers as extended control registers. Some extended control registers are global to the Nios II DPX MTP. Others are local to each thread. As with standard control registers, you can use the `rdctl` and `wrctl` instructions to access the contents of the extended control registers.

Table 5-5 shows details of the extended control registers defined in the Nios II DPX datapath processor.

Table 5-5. Extended Control Register Names and Descriptions

Physical Register Name	Mnemonic	Register Contents
ctl17	reserved	reserved
ctl24	message_flags (1)	Message interface unit (MIU) flags register
ctl25	message_user (1)	MIU user field register
ctl26	message_id0 (1)	MIU task register
ctl27	message_id1 (1)	MIU ID register

Notes to Table 5-5:

- (1) The MIU control registers are per-context control registers. One bank of MIU control registers exists for every context in the Nios II DPX datapath processor.

The following sections describe the extended control registers defined in the Nios II DPX datapath processor.

The message_flags Register

The `message_flags` register is the message interface unit flags register. It holds the message flags for a given task. The contents of this register are set based on the `flags` field of the incoming message that started the task. Software tasks can read and write this register.

Table 5-6 shows the layout of the `message_flags` register.

Table 5-6. `message_flags` Extended Control Register Fields (Note 1)

31	...	N	N-1	1	0
0			FLAGS		

Notes to Table 5-6:

- (1) N represents the total number of flag bits you specify when instantiate the Nios II DPX processor. You specify this value with the **Number of flag bits** parameter when instantiating your Nios II DPX processor. For more information, refer to “Instantiation for the Qsys Flow” in the *Instantiating the Nios II DPX Datapath Processor* chapter, in the *Nios II DPX Hardware Reference* section of the *Nios II DPX Datapath Processor Handbook*.

Table 5-7 shows details of the fields defined in the `message_flags` register.

Table 5-7. `message_flags` Extended Control Register Field Descriptions

Field	Description	Access	Reset	Available
FLAGS	FLAGS is the message flags field. The width of this field is variable and defined when instantiate the Nios II DPX processor. Bit 0 of the <code>flags</code> register contains the debug flag. For more information about the debug flag, refer to the <i>System Verification</i> chapter in the <i>Nios II DPX Hardware Reference</i> section of the <i>Nios II DPX Datapath Processor Handbook</i> .	Read/Write	0	Per thread

The `message_user` Register

The `message_user` register is the message user field register. It holds user-defined message information for a given task. The contents of this register are set based on the user field of the incoming message that started the task. Software tasks can read and write this register.

Table 5-8 shows the layout of the `message_user` register.

Table 5-8. `message_user` Extended Control Register Fields (Note 1)

31	...	N	N-1	1	0
0			USER		

Note to Table 5-8:

- (1) N represents the total number of user-defined bits you specify when instantiate the Nios II DPX processor. You specify this value with the **Number of user message bits** parameter when instantiating your Nios II DPX processor. For more information, refer to “Instantiation for the Qsys Flow” in the *Instantiating the Nios II DPX Datapath Processor* chapter, in the *Nios II DPX Hardware Reference* section of the *Nios II DPX Datapath Processor Handbook*.

Table 5-9 shows details of the fields defined in the `message_user` register.

Table 5-9. `message_user` Extended Control Register Field Descriptions

Field	Description	Access	Reset	Available
USER	USER is the message user-defined field. The width of this field is variable and defined when instantiate the Nios II DPX processor.	Read/Write	0	Per thread

The `message_id0` Register

The `message_id0` register is the message interface unit task register. It holds the node IDs and task ID for a given context. The `message_id0` register is read-only; writing to the register has no effect.

Table 5-12 shows the layout of the message_id0 register.

Table 5-10. message_id0 Extended Control Register Fields (Note 1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MIU_VERSION								PEID								SOURCE								RX_TASKID							

Notes to Table 5-10:

- (1) High-order bits in each field are set to zero when the defined width for a field in the message format is less than eight bits.

Table 5-11 shows details of the fields defined in the message_id0 register.

Table 5-11. message_id0 Extended Control Register Field Descriptions

Field	Description	Access	Reset	Available
RX_TASKID	RX_TASKID is the input task ID field. The width of this field is variable and defined at the time of DPX processor instantiation.	Read	0	Per thread
SOURCE	SOURCE is the source ID field. The width of this field is variable and defined at the time of DPX processor instantiation.	Read	0	Per thread
PEID	PEID contains a unique identifier for the Nios II DPX PE.	Read	0	Per thread
MIU_VERSION	MIU_VERSION contains the version number of the message interface unit hardware.	Read	0	Per thread

The message_id1 Register

The message_id1 register is the message interface unit ID register. It holds the sequence number, CID, RXID and TXIDs for a given context. The message_id1 register is read-only; writing to the register has no effect.

Table 5-12 shows the layout of the message_id1 register.

Table 5-12. message_id1 Extended Control Register Fields (Note 1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SEQNUM								CID								RXID								TXID							

Notes to Table 5-12:

- (1) High-order bits in each field are set to zero when the defined width for a field in the message format is less than eight bits.

Table 5-13 shows details of the fields defined in the `message_id1` register.

Table 5-13. `message_id1` Extended Control Register Field Descriptions

Field	Description	Access	Reset	Available
TXID	TXID is the transmit ID field. The width of this field is variable and contains enough bits to encode the number of TXIDs specified in the Number of transmit IDs processor configuration parameter.	Read	0	Per thread
RXID	RXID is the receive ID field. The width of this field is variable and contains enough bits to encode the number of RXIDs specified in the Number of receive IDs processor configuration parameter.	Read	0	Per thread
CID	CID is the context ID field. The width of this field is variable and contains enough bits to encode the number of CIDs specified in the Number of context IDs processor configuration parameter.	Read	0	Per thread
SEQNUM	SEQNUM is the sequence number field. The width of this field is variable and contains enough bits to encode the number of sequence numbers specified in the <code>NUM_SEQNUMS</code> processor configuration parameter.	Read	0	Per thread

Developing Software Tasks for the Datapath Processor

This section describes how to write software tasks for the Nios II DPX datapath processor.

The Nios II DPX Task ID

The task ID is a number that identifies a specific task in a given PE (such as the Nios II DPX datapath processor). A task ID's scope is its PE, so every task ID in a PE must be unique.

Every task must have a task ID. “Writing Task Code” on page 5-11 describes how you assign a task ID to a software task. Hardware task IDs are generally configured in the PE hardware.

The PEID is the address of a PE. All PE messages contain the following fields:

- Source—the PEID of the sending PE
- Destination—the PEID meant to receive the PE message
- Task ID—the specific task to run on the destination

When a Nios II DPX task sends a message to another PE, it uses the destination and task ID fields to specify the next task to run. The PEID and task ID together uniquely identify one task in a Nios II DPX system. Altera recommends defining PEIDs and task IDs in a common file accessible both to hardware definition files and software source files.

When the Nios II DPX datapath processor receives a message from another PE, it uses the task ID to select the software task to run on the Nios II DPX MTP.

Sending PE Messages Between Tasks

The Nios II DPX datapath processor has two PE message buffers, one for transmitted and one for received messages. Each buffer consists of a bank of message registers, indexed by TXID or RXID, discussed in the following sections. Buffer sizes are configurable.



A task function cannot be called directly by another task, or by any other function. Tasks are entry points that receive their arguments in the extension registers. If you want to “call” one task from another task, you must send a message. For information, see [“Spawning a New Task” on page 5-23](#).

Working With the RXID

Each message received by the Nios II DPX datapath processor is assigned an RXID. The RXID is an identifier used to select a bank of receive message registers (RX registers) to receive the message arguments. When a task starts running on a specific thread in response to the message, the thread uses the RXID to access the arguments in the RX registers directly.

The Nios II DPX datapath processor frees the RXID when the task terminates. Optionally, when the task has used the data in the receive message registers, it can free the RXID before it exits, with the `rxfree` instruction. This technique is useful if there is a shortage of RXIDs.

Working With the TXID

The TXID controls the resources for transmitting a PE message. Each TXID corresponds to a dedicated bank of registers, called *transmit message registers*. A TXID gives the thread access to the necessary registers for the message.

When a thread owns a TXID, the processor automatically indexes the corresponding transmit message registers, without needing to use any special addressing.

To send a message, a task must have a TXID. Every task must send at least one PE message. Therefore, each task is allocated a default TXID when it is assigned to a thread and starts running. If no TXID is available, no thread is assigned to the task. The task waits until a TXID becomes available.

Every task must ensure that its default TXID is released before terminating, by sending a message. To send a PE message, you move any required message data into the transmit message registers, and execute the `snd` or `sndi` instruction.

To send more than one PE message from a task, you must execute the `txalloc` instruction, to allocate a new TXID. Every task that successfully allocates a TXID must send an additional message before it terminates, to free the allocated TXID. See [“Sending Multiple PE Messages” on page 5-23](#).

Writing Task Code

You can write a task in C or assembly language. The underlying task mechanics are the same. The contents of a task are similar to a C function or an assembly language subroutine.

Because tasks are dispatched by hardware, and not called by software, the Nios II DPX toolchain treats a task as a program entry point, like `main()`. Tasks do not return to a caller. They terminate with an `exit` instruction, which simply puts the processor thread into an idle state.

Writing a Task in C

In C, use the `task` attribute to indicate to the Nios II DPX toolchain that the function defines a task. The syntax is as follows:

```
void __attribute__(( task ( <task id> ) ) <task function name> () {}
```

This attribute causes the function to terminate by restoring the stack pointer and executing an `exit` instruction.



When a task executes the `exit` instruction, the current RXID is released (unless it was previously released with the `rxfree` instruction).

The return statement is optional.

Before it terminates, each task must send a message to free the default TXID. See [“Working With the TXID” on page 5-11](#) for details.

[Example 5-1](#) shows how the task attribute is used.

Example 5-1. Task Declaration

```
#include "nios2dpx.h"

// Define the task ID used to call "my_task"
#define MY_TASK_ID 1

// Define the destination ID and task ID used by the message sent by "my_task".
// This defines the next operation to be performed.
#define OUTPUT_DESTINATION 0x3f
#define OUTPUT_TASK_ID 0x3f

// Definition of "my_task"
// Accepts a message with a single argument, adds the value of the argument
// to an internal context register, and sends a message
void __attribute__((task(MY_TASK_ID))) my_task()
{
    // Read the argument from the incoming message from RX0, and add it to
    // an internal context register (CR0)
    CR0 = CR0 + RX0;

    // Create an output message with one argument, the value of the
    // incremented context register. The message has a destination and
    // task ID set from constants, no special options, and a length of
    // one argument. This argument is copied from the internal context
    // register (CR0), to the first transmit argument (TX0).
    TX0 = CR0;
    NIOS2DPX_SNDI(OUTPUT_DESTINATION, OUTPUT_TASK_ID, OPT_NONE, 1);
}
```

Because a task is treated as a program entry point, the C runtime environment does not support any callee-saved registers when the task attribute is in effect.

When writing tasks for the Nios II DPX MTP, you can disregard many of the reentrancy issues that you might associate with low-overhead multithreaded environments. For example, each thread has a dedicated register bank, so each task can freely use registers as if it were the only thread executing.

Threads might also be furnished with dedicated thread-addressed memories. This option is selectable at system generation time. For more details, see [“Nios II DPX Memory Model” on page 5-20](#).

For information about the C prologue and epilogue in the Nios II DPX runtime environment, refer to [Chapter 8, Understanding the Nios II DPX Board Support Package](#).

Writing a Task in Assembly Language

The LWHAL BSP includes a predefined task table (in `task_table.S`) containing the entry point for each task. In assembly language, to install your task in the task table, you must declare its entry point with a global symbol whose name is of the form `__task_<n>`, where `<n>` is the task ID.

In C, the creation of a `__task_<n>` symbol is handled automatically.

If you develop your software in assembly language, you must insert the `exit` instruction at the end of each task.

When a task executes the `exit` instruction, the current RXID is released (unless it was previously released with the `rxfree` instruction).



Before it terminates, each task must send a message to free the default TXID. See [“Working With the TXID” on page 5-11](#) for details.

[Example 5-2](#) illustrates how to declare a task in assembly language.

Example 5-2. Declaring a Task in Assembly Language

```
/* Declare a task with task name mytask, task Id = 4. */
.align 2
.global mytask
.type mytask, @function
.global __task_4
.set __task_4, mytask          /* The LWHAL defines the __task_0 to __task_N */
                                /* symbols in task_table.S. */
                                /* __task_0 is task number 0, __task_1 is */
                                /* task number 1 and so on. */

mytask:
/* Reads RX registers and stores in CR registers */
mov cr0, rx0
mov cr1, rx1
mov cr2, rx2
mov cr3, rx3

mov tx0, rx0
movi r2, 2                    /* dstid = 0, taskid = 2 */
sandi r3, r2, ((OPT_SNDEXIT << 5) | 1) /* length = 1 */
exit
```

The Null Task ID

A TXID is returned to the TXID free list when the task sends a message. There is no mechanism for a thread to explicitly release a TXID.

If your task needs to release a TXID without triggering another task, it can send a message with the null task ID, 0xFF. A message with the null task ID is captured and discarded by the Nios II DPX datapath processor's message interface unit (MIU).

Resource Sharing

Resource sharing in a multithreaded environment introduces several challenges. For example, when multiple threads share one resource, a thread can corrupt another thread's data. This section discusses techniques you can use to share resources effectively in a Nios II DPX system.

The most common shared resource is memory. Nios II DPX tasks can share a resource through the context mechanism, using CIDs to control access to data and prevent corruption. However, some resources cannot be treated as task context. For example, a hardware peripheral might be needed by multiple tasks.

See [“Nios II DPX Memory Model” on page 5–20](#).

If the system is designed so that two tasks need concurrent access to the same resource, the system must include an external hardware mutex to prevent collisions. All tasks that access the resource must use the mutual exclusion hardware to protect it. This applies to both hardware and software tasks.

For information about using an Altera mutex in a Nios II DPX design, refer to [Chapter 8, Understanding the Nios II DPX Board Support Package](#).

Task-Related Instructions

The Nios II DPX MTP provides several special-purpose extended instructions to support event-driven programming. This section describes those instructions and their use.

Your C program can execute each extended instruction by invoking a specific macro provided by the LWHAL BSP. All program macros are defined in the BSP include file `nios2dpx.h`, which must be included in each source file.

For details about C extended instruction macros, refer to [“LWHAL Extended Instruction Macros” on page 8–20](#). For details about the extended instructions, refer to [Chapter 9, Nios II DPX MTP Instruction Set and Application Binary Interface](#).

RXID Free (rxfree)

The `rxfree` instruction returns the task's RXID to the RXID free list. The RXID is always freed when the task exits, but a task can use this instruction before the task is complete, so that the Nios II DPX datapath processor can accept a new PE message. This optimization is useful if the total number of RXIDs is smaller than the total number of CIDs.



The number of CIDs and RXIDs is determined when the system hardware is configured.

C code can execute the `rxfree` instruction with the following macro:

```
NIOS2DPX_RXFREE ( )
```

TXID Allocate (`txalloc`)

The `txalloc` instruction allocates a TXID, and associates that TXID and its transmit message registers to the running thread. If the thread already has a TXID, it does nothing. A thread can have only one TXID at a time.

Tasks can use this instruction to send multiple PE messages. A task must always have a TXID to send a message. Each time a task executes, the first message is sent with the default TXID, which is allocated by the system when it starts the task. If an additional message is to be sent, the task must allocate a new TXID after sending the first message.

C code can execute the `txalloc` instruction with the following macro:

```
NIOS2DPX_TXALLOC ( dest )
```

After executing the instruction, this macro places the status in `dest`. If no TXIDs are available, `txalloc` reports a failure.

If your system uses CID ordering or sequence number reordering, only enable ordering on one PE message per task. See [“Avoiding System Deadlock” on page 5-24](#).

If a task sends multiple messages, it must avoid duplicating the context. For further information, refer to [“Context Management” on page 5-16](#) and [“Avoiding System Deadlock” on page 5-24](#).

CID Allocate (`cidalloc`)

The `cidalloc` instruction requests a new CID, in order to create a new context. It is used to spawn a new task.

The Nios II DPX datapath processor can keep a CID in reserve. If the thread does not already have a reserve CID, `cidalloc` allocates a new CID. The instruction places the old CID in reserve, and the thread switches to the new CID. If no CIDs are available, this instruction returns a failure code, and takes no other action.

C code can execute the `cidalloc` instruction with the following macro:

```
NIOS2DPX_CIDALLOC ( dest )
```

After executing the instruction, this macro places the status in `dest`.

If a task sends multiple messages, it must avoid duplicating the context. For further information, refer to [“Context Management” on page 5-16](#) and [“Avoiding System Deadlock” on page 5-24](#).

You must ensure that all CIDs are freed once they are no longer needed. Freeing the CID adds it back into the CID free list so that it can be reused.

Send (`snd`)

The `snd` instruction transmits a PE message. The instruction contains the following information:

- The PE to receive the message
- The operation (task ID) to run on that PE

- Operational flags, specifying options such as how to manage the CID
- The number of transmit message registers in the message

If the task has a CID in reserve, it switches back to the reserve CID. If there is no reserve CID, it continues with the current CID.

This instruction marks the reserve CID as invalid. Until the reserve CID is invalidated, your code cannot successfully execute the `cidalloc` instruction.

C code can execute the `snd` instruction with the following macro:

```
NIOS2DPX_SND ( destID, taskID, options, length )
```

The macro arguments are as follows:

- `destID`—Unique identifier of destination PE
- `taskID`—Unique identifier of destination task
- `options`—Message control options
- `length`—Number of message arguments. To transmit the maximum number of TX registers (`NUM_TX`), set `length` to zero.

Send Immediate (sndi)

The `sndi` instruction is the same as the `snd` instruction, except that the `options` and `length` arguments are represented by a 16-bit immediate value. Therefore the `sndi` instruction supports no more than 11 option bits.

C code can execute the `sndi` instruction with the following macro:

```
NIOS2DPX_SNDI ( destID, taskID, options, length )
```

[Example 5-1 on page 5-12](#) shows how to use `NIOS2DPX_SNDI()` to send a message.

Exit

The `exit` instruction terminates a thread's processing in the current task.

When you write a task in C with LWHAL support, an `exit` instruction is automatically included at the end of each task function. For details, see [“The Lightweight Hardware Abstraction Layer \(LWHAL\)” on page 8-2](#).

Context Management

This section discusses how to use the Nios II DPX context management features.



For general information about maintaining context, refer to “Maintaining Context” in the *Introduction to Altera Event-Driven Datapath Processing* chapter of the [Altera Event-Driven Datapath Processing Design Handbook](#).

Creating a Context

Typically, a context is created in an input PE, flows through the system, and is disposed of by the Nios II DPX datapath processor in response to a message from the output PE. However, there are exceptions to this flow. For more information, see [“Spawning a New Task” on page 5-23](#).



If your system uses either sequence number reordering or CID ordering, and a task sends multiple messages, only one message from each task can invoke the reordering mechanism. For more information, see [“Avoiding System Deadlock” on page 5-24](#). It is preferable to avoid designing tasks that send multiple PE messages in a system that uses sequence number ordering.

Maintaining the CID Free List

The Nios II DPX datapath processor typically acts as the context manager, used for allocating CIDs and maintaining the CID free list. The CID free list contains all CIDs not currently in use. This section describes rules that software must follow to correctly maintain the CID free list.

Software must ensure that all CIDs are freed once they are no longer needed. Freeing the CID adds it back into the CID free list so that it can be reused. A CID can only be freed by a task on the Nios II DPX datapath processor.

Failure to free a CID results in a CID leak. For details about CID leaks, refer to [“Avoiding System Deadlock” on page 5-24](#).

A software task can free a CID when it sends a message. When the PE message is generated, use the `OPT_FREECID` option flag. When `OPT_FREECID` is set, the CID is freed by the Nios II DPX datapath processor after the PE message is sent to the destination PE. If `OPT_FREECID` is not set, the CID is passed to the destination PE.

Software must not free a CID that is already freed. If this happens, a second copy of the CID is added to the CID free list. The result is a duplicated CID, potentially resulting in undefined system behavior.



The hardware does not protect against CID leaks or duplicated CIDs.

Typically, the CID is not freed until the packet leaves the system. When the CID is freed, it is available for reuse, and the buffer space that it controls can be overwritten.

If a task sends multiple messages, only one message should normally carry the original CID. To avoid duplicating the context, each additional message must have a new CID, created with the `cidalloc` instruction. If you duplicate context (by sending PE messages with same CID to two different PEs) the result could be two threads working on the same, unprotected data. This situation can also cause system deadlock. For this reason, CID duplication is not recommended. For more information, see [“Avoiding System Deadlock” on page 5-24](#).

If your task needs to dispatch two tasks, use the spawning technique, described in the next section.

Data Ordering with the DPX Datapath Processor

In many systems, data ordering must be enforced, to ensure that data leaves the datapath processing system in the order in which it arrived. This technique is useful, for instance, if your system is receiving and processing data packets, and must ensure that it retransmits them without changing the order.



Data ordering increases the risk of system deadlock. For information about avoiding deadlock, see [“Avoiding System Deadlock” on page 5-24](#).

There are many techniques that can be used to maintain data order. This section discusses the following two mechanisms offered by the Nios II DPX datapath processor:

- Sequence number reordering
- CID ordering

Sequence Number Reordering

Sequence number reordering ensures that all PE messages leave the Nios II DPX datapath processor in the same order that the corresponding messages arrive. Any message that is to be sent by the datapath processor is held in a reorder queue in the datapath processor until all preceding messages are sent. This reorder mechanism is an option that can be selected when the Nios II DPX hardware is implemented.

With the option, the Nios II DPX datapath processor tags each incoming context with a sequence number. The sequence number is attached to the context until completion. When an outgoing message is sent with that context, the sequence number is attached to it. The messages are stored in the sequence number reordering queue of the Nios II DPX datapath processor. Each outgoing message is stored here until all messages with previous sequence numbers are sent. This ensures that the order of messages exiting the Nios II DPX datapath processor is the same as when the corresponding message entered.

In order to allow multiple messages to be sent by a task when sequence number reordering is being used, the Nios II DPX software is able to mark outgoing messages such that they bypass the reordering queue. In this case, tasks are responsible for disabling ordering when they send a PE message.



If your system uses sequence number reordering, only enable ordering on one PE message per task. See [“Avoiding System Deadlock” on page 5-24](#).

CID Ordering

CID ordering ensures that packets leave the system in the same order that they arrive, but allows them to get out of order during intermediate stages of processing. Packets use the CID as a reordering tag, and so they are effectively tagged in the first processing stage, and only reordered in the final processing stage.

CID reordering must be applied to exactly one message in the life of a CID. For this reason the Nios II DPX software is able to control whether or not to pass messages through the CID reorder queue. For example, you might use the CID reordering queue on all messages going to the output PE. This allows all messages from the Nios II DPX datapath processor to the output PE to be sent in the order of the CID, ensuring the output PE sends the data out in order. All messages to other PEs are allowed and bypass the queue.



If your system uses CID ordering, only enable ordering on one PE message per context. See [“Avoiding System Deadlock” on page 5-24](#).

Tasks are responsible for enabling ordering when they send a PE message.

Using the Nios II DPX Extension Registers

This section discusses some special considerations for using the Nios II DPX extension registers.

The number of extension registers is configurable at the time of system generation. When parameterizing the system, you select the number of each type of extension register. Receive and transmit message registers are always present.

Certain extension registers are either read-only or write-only with respect to the Nios II DPX MTP. Furthermore, there are restrictions on extension register usage in the instruction fields.


 For details about Nios II DPX extension registers and how to access them, refer to “Multithreaded Processor” in the *Nios II DPX Architecture* chapter, in the *Nios II DPX Hardware Reference* section of the *Nios II DPX Datapath Processor Handbook*. For general information about the Nios II DPX registers, see “Nios II DPX Registers” on page 5-4.

Table 5-14 lists the assembly language names for the extension registers.


 For information about how extension registers are accessed, refer to “Functional Blocks” in the *Nios II DPX Architecture* chapter, in the *Nios II DPX Hardware Reference* section of the *Nios II DPX Datapath Processor Handbook*.

Table 5-14. Assembly Language Names for Extension Registers

Register Group	Register Names	Notes
Receive message registers (1)	RX0 to RX<n> (0<n≤31)	When a task starts running, the receive message registers contain the arguments carried by the PE message that triggered the task, eliminating the need to load message arguments from memory.
Input context registers (2)	CRi0 to CRi<n> (0<n≤15)	These registers, which are read by the processor and written by external hardware, contain values specific to the CID that the current thread is using. They can be used to share information between hardware PEs and software tasks processing the same context.
Output context registers (2)	CRo0 to CRo<n> (0<n≤15)	These registers, which written by the processor and read by external hardware, contain values specific to the CID that the current thread is using. They can be used to share information between hardware PEs and software tasks processing the same context.
Internal context registers (2)	CR0 to CR<n> (0<n≤15)	These registers, which are both read or written by the processor, contain values specific to the CID that the current thread is using. They can be used to share information between different task processing the same context.
Transmit message registers (3)	TX0 to TX<n> (0<n≤31)	Prior to transmission of a PE message, a software task places message arguments in the transmit message registers.

Notes to Table 5-14:

- (1) Each RXID provides access to a unique bank of receive message registers.
- (2) Each CID provides access to a unique bank of context registers, including input, output, and internal context registers.
- (3) Each TXID provides access to a unique bank of transmit message registers.

Accessing Extension Registers

The Nios II DPX assembler strictly enforces register usage based on the configuration selected when the hardware is generated. The assembler only allows you to refer to implemented registers. It enforces read-only and write-only registers, and restrictions on which operands registers can appear in.

The Nios II DPX C compiler and assembler both take a command-line switch specifying the register configuration. The compiler passes the command-line switch to the assembler. The assembler reports an error if a prohibited operation appears in the code.

When it generates the BSP, the SBT determines the register configuration by examining the system **.sopcinfo** file, and places the correct register configuration flag in **public.mk**.

Accessing Extension Registers in C

You can access extension registers by using macro names defined by the Nios II DPX toolchain. The toolchain defines names only for those extension registers that are actually implemented in the hardware system. To use the register names, include the header file **nios2dpx.h** in each source file. Extension registers are declared as type **unsigned int**. **nios2dpx.h** declares the following register names, depending on the number of registers present in the hardware:

- Receive message registers—RX0 to RX31
- Transmit message registers—TX0 to TX31
- Internal context registers—CR0 to CR15
- Input context registers—CRi0 to CRi15
- Output context registers—CRo0 to CRo15

Example 5-1 on page 5-12 shows the use of the extension registers in C.

Accessing Extension Registers in Assembly Language

Assembly language code can refer to extension register with the same syntax as for general registers. See Table 5-14 on page 5-19 for a list of register mnemonics.

Example 5-2 on page 5-13 shows the use of the extension registers in assembly language.

Nios II DPX Memory Model

This section describes how the Nios II DPX MTP accesses memory.

Physical Memory Access


MTP addresses are 32 bits, allowing access up to a 4-gigabyte address space. The locations of memory within the address space are specified in the parameter editor. Reading from or writing to an address that does not map to a memory produces an undefined result.

The processor's data bus is 32 bits wide. It is possible to read and write data memory as bytes, half-words (16 bits), or words (32 bits).

Instruction and data memory are accessed by separate Avalon masters. Therefore the Nios II DPX MTP cannot read and write instructions from the data master.


In the Nios II DPX MTP address map, data is at addresses 0x00000000—0x7ffffff. The most significant bit of the address is disregarded in data accesses. Instructions reside physically at 0x80000000 and higher. However, in the Nios II DPX MTP memory map, instructions appear starting at address 0, since the most significant bit of the address is asserted by the hardware

The Nios II DPX architecture is big-endian. The Nios II DPX instruction and data masters use the byte-invariant big-endian convention (BE-8).

 For more information about the Nios II DPX BE-8 memory organization, refer to “Nios II DPX Processor Interfaces” in the *Nios II DPX Architecture* chapter, in the *Nios II DPX Hardware Reference* section of the *Nios II DPX Datapath Processor Handbook*.

The MTP architecture supports register+immediate addressing.

Instruction memory and data memory are shared by all threads.

 For more information, refer to “Functional Blocks” in the *Nios II DPX Architecture* chapter, in the *Nios II DPX Hardware Reference* section of the *Nios II DPX Datapath Processor Handbook*.


The Nios II DPX MTP can access data memory either of two ways:

- Through the fixed-latency data master interface—a direct connection to memory with exactly two cycles of read latency
- Through the variable-latency data master interface—any Avalon-MM slave

Fixed-latency accesses are faster than variable-latency accesses. Typically, a system uses the fixed-latency data master to access critical data structures, and the variable-latency data master to access noncritical data and hardware peripherals.

You configure the data master interfaces as described in the *Nios II DPX Architecture* chapter, in the *Nios II DPX Hardware Reference* section of the *Nios II DPX Datapath Processor Handbook*. When accessing data memory, the Nios II DPX MTP automatically determines which data master interface to use based on the memory address.

If the Nios II DPX datapath processor is instantiated in dual-core configuration, the SBT and the GNU toolchain assume that the memory map for the two Nios II DPX MTPs is identical. You must observe this restriction when connecting the data masters.

 For detailed information about the Nios II DPX variable-latency data master, refer to “Nios II DPX Processor Interfaces” in the *Nios II DPX Architecture* chapter, in the *Nios II DPX Hardware Reference* section of the *Nios II DPX Datapath Processor Handbook*.


Memory Organization

Memory in a Nios II DPX system falls into one of the following categories:

- Thread storage—dedicated to a particular thread on a particular Nios II DPX datapath processor
- Context storage—associated with a particular CID

- Datapath processor storage—available to all threads on a particular Nios II DPX datapath processor
- Global storage—available to all threads on all Nios II DPX datapath processors

Thread- and context-based memory partitioning are implemented using the `threadinfo` interface and a context memory adaptor.

 For details about using a context memory adaptor, refer to “Nios II DPX Context Address Adapter” in the *Instantiating the Nios II DPX Datapath Processor* chapter, in the *Nios II DPX Hardware Reference* section of the *Nios II DPX Datapath Processor Handbook*.

Thread Storage

In a parallel-processing environment, if you have the same task code running on different threads, it accesses the same data structures at the same memory addresses. Without some sort of memory management, this arrangement is untenable because one thread corrupts another thread’s data.

You can configure the system so that the memory is physically addressed by the data address from the processor with an offset based on the thread number. This is like having a very simple memory management unit (MMU), giving each thread a unique memory region. A thread is physically incapable of accessing physical addresses outside its region.

In this configuration, thread numbers are used to address unique segments of memory, transparent to the software model. To each thread, it appears as if it has a dedicated memory at a fixed address.

Memory partitioned by thread is useful for non-packet-oriented applications, such as video processing, where all threads are operating on the same data structures.

You can also use the thread number to manage separate stacks for each thread. Using a physical address with an offset based on the thread number makes each thread physically incapable of corrupting other threads’ stacks. To manage the stack in this way, you must modify `crt0.S` to initialize the stack correctly.

These options must be configured in hardware. No software toolchain support is required.

Context Storage

Memory partitioned by context is useful for applications such as packet processing, where each task has exclusive access to its data while it is running, and passes it to the next task when it terminates.

Typically, memory is partitioned by context to ease the storage of information about every packet currently in the datapath controlled by the Nios II DPX datapath processor. In order for this buffer to persist after a task is complete and the packet is sent for processing by another task, the memory is indexed by the CID.

Datapath Processor Storage

Typically, a Nios II DPX system is implemented with some memory that is accessible to all threads in a Nios II DPX datapath processor. In this case, the threads must use an effective form of mutual exclusion to prevent data corruption. This topic is discussed in “Resource Sharing” on page 5-14.

Global Storage

Normally, multiple Nios II DPX processors in a single system do not share memory. Most data required by a task is specific either to a particular thread, or a particular context.

However, it is possible to share memory between multiple DPX processors. In this case, arbitration is required in the memory mapped interconnect, and so the variable latency data master must be used.

Advanced Topics

This section discusses advanced programming topics that are not relevant to all systems.

Sending Multiple PE Messages

Typically, each task sends one message, when its processing is complete. However, your code can send additional messages if necessary, with certain precautions.

The Nios II DPX datapath processor uses the TXID to allocate space in the Tx message queue. Every task is assigned a TXID when it starts running. When it transmits a PE message, it relinquishes its TXID.

Each task must allocate a new TXID before sending an additional message. Your code can request a new TXID using the `txalloc` instruction.

Generally, an additional message spawns another task. If this task requires context data, your code must allocate an additional CID. For details, see [“Spawning a New Task” on page 5-23](#).



If the system enforces sequence number reordering, and a task sends multiple PE messages, only one message can invoke sequence number reordering. All other messages must be sent with the sequence number ordering bypass option set.

In a system that uses sequence number ordering, it is risky for tasks to send multiple PE messages. Consider partitioning tasks so that each task sends only one message. For further information, see [“Data Ordering with the DPX Datapath Processor” on page 5-17](#).

Spawning a New Task

Sometimes a software task running on a Nios II DPX datapath processor needs to spawn another task running on the same processor. To do this, the task must send a PE message with the destination set to the same processor that it is running on. This technique requires that the message interconnect be configured so that the Nios II DPX datapath processor can send messages to itself.

Example 5-3 shows how to spawn a new task by sending a message with a new CID, whilst the current thread continues on the existing CID. Similarly you can send a message on the current CID, whilst the current thread continues on a new CID.

Example 5-3. Spawning a Task

```
#include "nios2dpx.h"
#include "mytaskids.h"

void __attribute__((task(MYTASKID))) mytask(void)
/* MYTASKID is a task ID value defined in mytaskids.h */
{
    int result;
    int my_cri0, my_cri1;
    int my_cr0;
    int my_tx0;

    /* Reads extension registers. */
    my_cri0 = CRi0;
    my_cri1 = CRi1;

    /* Save context register. */
    my_cr0 = CR0;

    NIOS2DPX_CIDALLOC(result); /* Get a new CID, swap to new CID context */
    if(result == ERR_OK)
    {
        CR0 = my_cr0; /* Copy data to new context */
        my_tx0 = (my_cri0 << 16) | (my_cri1 >> 16);
        /* Writes extension registers. */
        TX0 = my_tx0;
        /* Send a message with new CID, swapping back to original CID. */
        NIOS2DPX_SNDI(0 /*dstId*/, 0 /* taskId*/, OPT_FREECID, 1);
    }

    NIOS2DPX_TXALLOC(result); /* Get a new TXID. */
    if(result == ERR_OK)
    {
        TX0 = RX0;
        TX1 = RX1;
        /* Send a message with original CID. */
        NIOS2DPX_SNDI(0 /*dstId*/, 2 /* taskId*/, 0 /* opts */, 2 /* length */);
    }

    return;
}
```

Avoiding System Deadlock

There are several ways that software can inadvertently create a deadlock condition in a Nios II DPX system. This section discusses several common deadlock situations, and how you can avoid them.

CID Leak

Every CID that is allocated must be freed when it is no longer needed: for example, when the corresponding data packet leaves the Nios II DPX system. Your code must ensure that every CID allocation is balanced by a message with the `OPT_FREECID` flag set. This applies to all CID allocations, whether initiated by hardware through the CID request interface or by software with the `cidalloc` instruction. For more information on this topic, refer to [“Maintaining the CID Free List” on page 5-17](#).

CID Ordering Violations

If CID reordering is being used, exactly one message for each CID must be routed through the CID reorder buffer, by using the `OPT_CIDORDER` option with the `snd` or `sndi` instruction. If this restriction is violated, undefined behavior, including CID queue stalls and potential deadlock, might result.

TXID Leak

Every task must ensure that its default TXID is released before terminating, by sending a message. Every task that successfully allocates a TXID must send an additional PE message before it terminates, to free the allocated TXID.



Failure to send a message in a task is also likely to cause a CID leak. For details, see [“CID Leak”](#).

TXID Free List Empty

The `txalloc` instruction can fail if no TXIDs are available. This situation is more likely if some tasks issue multiple events per task, or if reordering is enabled, as TXIDs cannot be freed until the PE message is sent to the destination PE. If all TXIDs are in use, and no running task is able to send a message and free a TXID, the system deadlocks.

You can avoid this problem by increasing the number of TXIDs.

Software can avoid this problem by sending a message promptly, ensuring that every TXID is freed.



The total number of available TXIDs is determined when the hardware is generated.

Ordering Queue Full

If your system uses data ordering, and a task sends multiple messages, only one message from each task can invoke the reordering mechanism. Typically this is either the first or the last message sent by the task. All other messages must disable reordering. If a task sends a second message with ordering enabled, one of the messages stalls in the ordering queue. If this happens repeatedly, the ordering queue fills, and the system locks up.

This deadlock hazard applies equally to sequence number reordering and CID ordering.

Sequence Number Violations

When a system uses sequence number reordering, you must ensure that every sequence number is returned to the sequence reorder queue exactly once. Otherwise, one of two problems arise:

- Sequence number leak—If a sequence number fails to return to the queue, the system locks up waiting for the missing sequence number.
- Sequence number duplication—If the reordering queue receives multiple messages with the same sequence number, only one of the messages is reordered correctly. The remaining messages stall for an indeterminate period of time. As a result, some arbitrary future messages stall. The system continues to display message stalling behavior indefinitely.

To ensure that all sequence numbers return to the queue exactly once, design your system to conform to the following rules:

- Each task sends at least one message.
- If a task sends more than one message, sequence number reordering is enabled on only one message. Sequence number ordering is enforced by default when you send a message. On all other messages, you must set the `OPT_KEEPSEQNUM` flag to one in the `snd` or `sndi` instruction to disable sequence number ordering.

Exception Processing

Each of the MTP exceptions falls into one of the following categories:

- Reset exception—Occurs when the Nios II DPX processor is reset. Control is transferred to the reset address you specify in the IP core setup parameters.
- Break exception—Occurs when the debug module requests control. Control is transferred to the break address you specify in the IP core setup parameters.
- Instruction-related exception—Occurs when any of several internal conditions occurs, as detailed in [Table 5-15 on page 5-27](#). Control is transferred to the exception address you specify in the IP core setup parameters.

There are no interrupt exceptions, because the Nios II DPX MTP core does not implement interrupts.

[Table 5-15](#) shows all possible MTP exceptions in order of highest to lowest priority.

Table 5-15. Nios II DPX Exceptions (In Decreasing Priority Order)

Exception	Exception Type	Saved Address Register (1)	Vector (2)
Reset	Reset	n/a	Reset
Hardware Break	Break	ba (3)	Break
Break Instruction	Instruction-related	ba (3)	Break
Trap Instruction	Instruction-related	ea (3)	Exception
Unimplemented Instruction	Instruction-related	ea (3)	Exception

Notes to Table 5-15:

- (1) Specifies the register in which the address is saved. The address saved is always `pc+4`.
- (2) Specifies which exception vector address the processor passes control to when the exception occurs.
- (3) Refer to [Table 5-1 on page 5-4](#) for descriptions of the `ea` and `ba` registers.

Reset Exceptions

The reset state is undefined for all system components, including general-purpose registers, extension registers, control registers, and instruction and data memory.

When a hard processor reset signal is asserted, all operation of the MTP resets and suspends, and a processor soft reset signal (`cpu_soft_reset`) is asserted for each thread in the processor. The soft reset causes the MTP to suspend instruction fetch and execution until the soft reset signal is de-asserted. The soft reset can only be asserted while the hard reset signal (`cpu_rst_n`) is asserted but can be kept asserted after hard reset is de-asserted.

While soft reset is asserted and hard reset is de-asserted, the MTP is fully functional but does not fetch instructions. This scenario allows debug operations and the instruction access slave to operate so that host debug operations function as normal and the debugger or external logic can access instruction memory.



The main use model of the soft reset is to disable the Nios II DPX MTP while instruction, task, and data memories are written by the debugger or external logic.

The MTP reset address is defined by the Nios II DPX hardware system, and is selectable at system generation time. External logic connected to the thread management interfaces controls the PCs of the threads.

When both the hard and soft resets de-assert and the MTP comes out of reset, the first instruction executed on each thread must be `andi r0, r0, 0` to set `r0` to zero.



If the debugger gains control before the first instruction executes for a thread, the debugger must execute the `andi r0, r0, 0` instruction before executing other instructions.



For information about configuring the reset address, refer to the *Instantiating the Nios II DPX Datapath Processor* chapter, in the *Nios II DPX Hardware Reference* section of the *Nios II DPX Datapath Processor Handbook*.

Break Exceptions

A break is a transfer of control away from a program's normal flow of execution for the purpose of debugging. Software debugging tools can take control of the processor core via the debug module.

Break processing is the means by which software debugging tools implement debug and diagnostic features, such as breakpoints. Break processing is a type of exception processing, but the break mechanism is independent from general exception processing. A break can occur during exception processing, enabling debug tools to debug exception handlers.

The processor enters the break processing state under either of the following conditions:

- The processor executes the break instruction. This is often referred to as a software break.
- The debug module asserts a hardware break.

Processing a Break

A break causes the processor to take the following steps:

1. Writes the address of the instruction following the instruction that caused the exception to the `ba` register (`r30`).
2. Transfers execution to the break vector you specified in the IP core setup parameters.

The contents of the break vector are under control of the debugger. The debugger executes code on the Nios II DPX MTP by repeatedly performing the following steps:

1. Writes an instruction to the break vector
2. Asserts the break request signal to cause the processor to execute the instruction at the break vector.

When the debugger is ready to return control to the Nios II DPX software, it inserts a `bret` instruction at the break vector.

Understanding Register Usage

The general-purpose registers `bt` (`r25`) and `ba` (`r30`) are reserved for debugging. Code is not prevented from writing to these registers, but debug code might overwrite the values. The break handler can use `bt` (`r25`) to help save additional registers.

Returning From a Break

The `bret` instruction returns program execution to the address in the `ba` register (`r30`). Aside from `bt` and `ba`, all registers are guaranteed to be returned to their pre-break state after returning from the break handler.

Instruction-Related Exceptions

Instruction-related exceptions occur during execution of MTP instructions. This section describes the possible exceptions and how they are processed.

Instruction-related exceptions on the Nios II DPX MTP are fatal. In a complex multithreaded datapath processing system, there is no realistic way for a task to recover from a fault on an individual thread, or to continue without disrupting the rest of the system.

The Nios II DPX MTP supports the following instruction-related exceptions:

- break instruction
- trap instruction
- Unimplemented instruction

Break Instruction

The break instruction is treated as a break exception. For more information, refer to [“Break Exceptions” on page 5-28](#).

Trap Instruction

When a program issues the trap instruction, the processor generates a software trap exception, transferring control to the exception handler. For information about the exception handler, see [“The Lightweight Hardware Abstraction Layer \(LWHAL\)” on page 8-2](#).

The trap instruction is not intended to support an RTOS.

Unimplemented Instruction

An unimplemented instruction is an instruction word value that is not defined in the Nios II DPX MTP implementation. Many word values have no meaning in the instruction set architecture. Others are available only if the target device possesses a DSP block.

When a program attempts to execute an unimplemented instruction, the processor generates an exception, transferring control to the exception handler. For information about the exception handler, see [“The Lightweight Hardware Abstraction Layer \(LWHAL\)” on page 8-2](#).



To determine the unused opcodes and opcode extensions, refer to the encodings tables in [“The Nios II DPX MTP Instruction Set” on page 9-1](#).


The following instructions are available only if the target device possesses a DSP block:

- mulxss
- mulxsu
- mulxuu

The Nios II DPX MTP provides no way to determine the cause of an exception.

Because instruction-related exceptions are fatal, the Nios II DPX lightweight hardware abstraction layer (LWHAL) implements an exception handler that consists of a break instruction. The break instruction transfers control to the Nios II DPX MTP debug core.

All threads share a single exception vector, defined at the time of core instantiation. The SBT creates the `.exceptions` linker region at the exception vector address.

 For details about the `trap`, `mulxss`, `mulxsu`, and `mulxuu` instructions, refer to [Chapter 9, Nios II DPX MTP Instruction Set and Application Binary Interface](#). For information about configuring the exception vector, refer to the *Instantiating the Nios II DPX Datapath Processor* chapter, in the *Nios II DPX Hardware Reference* section of the *Nios II DPX Datapath Processor Handbook*.

Processing an Instruction-Related Exception


An instruction-related exception causes the processor to take the following steps:

1. Writes the address of the instruction following the break to the `ea` register (`r29`).
2. Transfers execution to the exception handler, stored at the exception vector you specified in the IP core setup parameters.

Software cannot modify the exception vector. Programmers do not directly access exception vectors, and can write programs without awareness of the address.

The lightweight hardware abstraction layer (LWHAL) default exception handler is a simple routine that passes control to the debug unit.

You can create your own exception handler. The routine must save registers on entry and restore them on exit. Saving the register contents on the stack is a typical, re-entrant implementation.

 For more information about writing exception handlers, refer to [“Exception Handling” on page 8-8](#).

Instruction Set Categories

This section introduces the MTP instruction set, categorized by type of operation performed.

Data Transfer Instructions

The MTP architecture is a load-store architecture. Load and store instructions handle all data movement between registers, memory, and peripherals. Memories and peripherals share a common address space.

[Table 5-16](#) describes the wide (32-bit) load and store instructions.

Table 5-16. Wide Data Transfer Instructions

Instruction	Description
<code>ldw</code> <code>stw</code>	The <code>ldw</code> and <code>stw</code> instructions load and store 32-bit data words from/to memory. The effective address is the sum of a register's contents and a signed immediate value contained in the instruction.

The data transfer instructions in [Table 5-17](#) support byte and half-word transfers.

Table 5-17. Narrow Data Transfer Instructions

Instruction	Description
ldb ldbu stb ldh ldhu sth	ldb, ldbu, ldh and ldhu load a byte or half-word from memory to a register. ldb and ldh sign-extend the value to 32 bits, and ldbu and ldhu zero-extend the value to 32 bits. stb and sth store byte and half-word values, respectively.

Bit Manipulation Instructions

Bit manipulation instructions isolate and operate on specific bits in a register. Refer to [Table 5-18](#).

Table 5-18. Bit Manipulation Instructions

Instruction	Description
and or xor nor	These are the standard 32-bit logical operations. These operations take two register values and combine them bitwise to form a result for a third register.
andi andci ori xori	These operations are immediate versions of the and, or, and xor instructions. The 16-bit immediate value is zero-extended to 32 bits, and then combined with a register value to form the result.
andhi andchi orhi xorhi	In these versions of and, or, and xor, the 16-bit immediate value is shifted logically left by 16 bits to form a 32-bit operand. Zeroes are shifted in from the right.
extract insert merge	These operations isolate and change specific bits while leaving the remaining bits unchanged. extract gets bits from a register while insert and merge replace bits in a register. extract and insert shift bits to the low end of the register while merge specifies the bits in place. The insert instruction is an optional instruction. It can be configured at system generation time. If software attempts to execute an insert instruction on a processor that does not implement the instruction, the processor generates an exception.

Arithmetic Instructions

Arithmetic instructions support addition, subtraction, and multiplication operations. Refer to [Table 5-19](#).

Table 5-19. Arithmetic Instructions (Part 1 of 2)

Instruction	Description
add sub mul	These are the standard 32-bit arithmetic operations. These operations take two registers as input and store the result in a third register.
addi subi muli	These instructions are immediate versions of the add, sub, and mul instructions. The instruction word includes a 12-bit signed value for addi, subi, and muli.

Table 5-19. Arithmetic Instructions (Part 2 of 2)

Instruction	Description
<code>mulxss</code> <code>mulxuu</code>	These instructions provide access to the upper 32 bits of a 32x32 multiplication operation. Choose the appropriate instruction depending on whether the operands should be treated as signed or unsigned values. It is not necessary to precede these instructions with a <code>mul</code> .
<code>mulxsu</code>	This instruction is used in computing a 128-bit result of a 64x64 signed multiplication.

Move Instructions

These instructions provide move operations to copy the value of a register or an immediate value to another register. Refer to [Table 5-20](#).

Table 5-20. Move Instructions

Instruction	Description
<code>mov</code> <code>movhi</code> <code>movhi20</code> <code>movi</code> <code>movui</code> <code>movia</code>	<code>mov</code> copies the value of one register to another register. <code>movi</code> moves a 12-bit signed immediate value to a register, and sign-extends the value to 32 bits. <code>movhi</code> and <code>movhi20</code> move 16-bit and 20-bit immediate values, respectively, into the uppermost bits of a register, inserting zeros in the remaining bit positions. <code>movui</code> moves a 16-bit immediate value into the lower 16 bits of a register, inserting zeros in the upper 16 bits. Use <code>movia</code> to load a register with an address.

Comparison Instructions

The MTP architecture supports a number of comparison instructions. All of these instructions perform a comparison as follows:

1. Compare two registers or a register and an immediate value
2. Return the result of the comparison in a result register. The result can be true (one) or false (zero).

These instructions perform all the equality and relational operators of the C programming language. Comparison instructions are listed in [Table 5-21](#).

Table 5-21. Comparison Instructions (Part 1 of 2)

Instruction	Description
<code>cmpeq</code>	Return true if values equal (==)
<code>cmpne</code>	Return true if values not equal (!=)
<code>cmpge</code>	Compare signed values, and return true if first value greater than or equal to second value (≥)
<code>cmpgeu</code>	Compare unsigned values, and return true if first value greater than or equal to second value (unsigned ≥)
<code>cmpgt</code>	Compare signed values, and return true if first value greater than second value (signed >)
<code>cmpgtu</code>	Compare unsigned values, and return true if first value greater than second value (unsigned >)
<code>cmple</code>	Compare signed values, and return true if first value less than or equal to second value (signed ≤)
<code>cmpleu</code>	Compare signed values, and return true if first value less than or equal to second value (unsigned ≤)
<code>cmplt</code>	Compare signed values, and return true if first value less than second value (signed <)

Table 5-21. Comparison Instructions (Part 2 of 2)

Instruction	Description
cmpltu	Compare unsigned values, and return true if first value less than second value (unsigned <)
cmpeqi cmpnei cmpgei cmpgeui cmpgti cmpgtui cmplei cmpleui cmplti cmpltui	These instructions are immediate versions of the comparison operations. They compare the value of a register and a 12-bit immediate value. Signed operations sign-extend the immediate value to 32 bits. Unsigned operations fill the upper bits with zero.

Shift and Rotate Instructions

The following instructions provide shift and rotate operations. The number of bits to rotate or shift can be specified in a register or an immediate value. Refer to [Table 5-22](#).

Table 5-22. Shift and Rotate Instructions

Instruction	Description
rol ror roli	The <code>rol</code> and <code>roli</code> instructions provide left bit rotation. <code>roli</code> uses an immediate value to specify the number of bits to rotate. The <code>ror</code> instructions provides right bit rotation. There is no immediate version of <code>ror</code> , because <code>roli</code> can be used to implement the equivalent operation.
sll slli sra srl srai srli	These shift instructions implement the << and >> operators of the C programming language. The <code>sll</code> , <code>slli</code> , <code>srl</code> , <code>srli</code> instructions provide left and right logical bit-shifting operations, inserting zeros. The <code>sra</code> and <code>srai</code> instructions provide arithmetic right bit-shifting, duplicating the sign bit in the most significant bit. <code>slli</code> , <code>srli</code> and <code>srai</code> use an immediate value to specify the number of bits to shift.

Message Passing Instructions

Message passing instructions support hardware message passing. Refer to [Table 5-23](#).

Table 5-23. Message Passing Instructions (Note 1), (2)

Instruction	Description
snd sndi	The <code>snd</code> and <code>sndi</code> instructions send messages.
txalloc	The <code>txalloc</code> instruction allocates a new TXID and a new transmit buffer, for use by the current thread. (3)
rxfree	The <code>rxfree</code> releases the RXID held by the current thread, along with the associated receive message buffer. (3)
cidalloc	The <code>cidalloc</code> instruction obtains a new context ID for the current thread, facilitating spawning.

Notes to Table 5-23:

- (1) The meaning of the register arguments is described in [Chapter 9, Nios II DPX MTP Instruction Set and Application Binary Interface](#). Also see “[Sending PE Messages Between Tasks](#)” on page 5-11.
- (2) Set the destination register to `r0` when the destination value is not required.
- (3) For information about using RXID and TXID, see “[Sending PE Messages Between Tasks](#)” on page 5-11.

Program Control Instructions

The MTP architecture supports the unconditional jump, branch, and call instructions listed in [Table 5-24](#). These instructions do not have delay slots.

Table 5-24. Unconditional Jump and Call Instructions

Instruction	Description
<code>call</code>	This instruction calls a subroutine using an immediate value as the subroutine's absolute address, and stores the return address in register <code>ra</code> .
<code>callr</code>	This instruction calls a subroutine at the absolute address contained in a register, and stores the return address in register <code>ra</code> . This instruction serves the role of dereferencing a C function pointer.
<code>ret</code>	The <code>ret</code> instruction is used to return from subroutines called by <code>call</code> or <code>callr</code> . <code>ret</code> loads and executes the instruction specified by the address in register <code>ra</code> .
<code>jmp</code>	The <code>jmp</code> instruction jumps to an absolute address contained in a register. <code>jmp</code> is used to implement switch statements of the C programming language.
<code>jmp_i</code>	The <code>jmp_i</code> instruction jumps to an absolute address using an immediate value to determine the absolute address.
<code>jrel</code>	The <code>jrel</code> instruction jumps to an address relative to the current PC address. <code>jrel</code> is used to jump into a branch table to implement C-language switch statements efficiently. This instruction is only efficient when the case constants in the switch statement are grouped closely together.
<code>br</code>	This instruction branches relative to the current instruction. A 12-bit signed immediate value gives the word offset, which is equivalent to a signed 14-bit byte offset, of the next instruction to execute.

The conditional branch instructions compare register values directly, and branch if the expression is true. Refer to [Table 5-25](#). The conditional branches support the following equality and relational comparisons of the C programming language:

- `==` and `!=`
- `<` and `<=` (signed and unsigned)
- `>` and `>=` (signed and unsigned)

The conditional branch instructions do not have delay slots.

Table 5-25. Conditional Branch Instructions

Instruction	Description
<code>bge</code> <code>bgeu</code> <code>bgt</code> <code>bgtu</code> <code>ble</code> <code>bleu</code> <code>blt</code> <code>bltu</code> <code>beq</code> <code>bne</code>	These instructions provide relative branches that compare two register values and branch if the expression is true. Refer to "Comparison Instructions" on page 5-32 for a description of the relational operations implemented.

Thread Control Instructions

The MTP architecture supports hardware multithreading. Refer to [Table 5-26](#).

Table 5-26. Thread Control Instructions

Instruction	Description
<code>exit</code>	This instruction releases a thread, marking it as idle and therefore available to execute a new task. The current task is terminated.

Other Control Instructions

[Table 5-27](#) shows other control instructions.

Table 5-27. Other Control Instructions

Instruction	Description
<code>trap</code> <code>eret</code>	The <code>trap</code> and <code>eret</code> instructions generate and return from exceptions. These instructions are similar to the <code>call/ret</code> pair, but are used for exceptions. <code>trap</code> saves the return address in the <code>ea</code> register, and then transfers execution to the general exception handler. <code>eret</code> returns from exception processing by executing the instruction specified by the address in <code>ea</code> .
<code>break</code> <code>bret</code>	The <code>break</code> and <code>bret</code> instructions generate and return from breaks. <code>break</code> and <code>bret</code> are used exclusively by software debugging tools. Programmers never use these instructions in application code.
<code>rdctl</code> <code>wrcctl</code>	These instructions read and write control registers, such as the <code>cpuid</code> register. The value is read from or stored to a general-purpose register.

No-operation Instruction

The MTP assembler provides a no-operation instruction, `nop`.

Potential Unimplemented Instructions

A Nios II DPX processor core might not support all instructions if it is implemented on a device without a DSP block. In this case, the processor generates an exception after issuing an unimplemented instruction. The following instructions can generate an unimplemented instruction exception:

- `insert`
- `mulxss`
- `mulxsu`
- `mulxuu`

The Nios II EDS allows you to construct and debug a wide variety of complex software systems using a set of GUI interfaces. From these interfaces, you can create application, BSP, and library projects, and you can build and debug these projects.

This chapter introduces you to project creation and debugging with the EDS GUI tools.

The Nios II EDS provides two GUI tools for developing and debugging Nios II DPX software: the System Console and the Nios II SBT for Eclipse. Eclipse is an IDE providing common software development and debugging tools. The System Console allows you to download software and configure a debug server for each Nios II DPX datapath processor.

This chapter familiarizes you with the features of the Nios II DPX GUI tools. This chapter contains the following sections:

- “Getting Started”
- “Makefiles and the Nios II SBT for Eclipse” on page 6–17
- “Using the BSP Editor” on page 6–20
- “Importing a Command-Line Project” on page 6–28
- “Managing Toolchains in Eclipse” on page 6–32
- “Eclipse Usage Notes” on page 6–32

Introduction to the Nios II DPX Debugging Environment

The System Console performs low-level debugging of hardware systems. The System Console includes the GDB server control panel, which enables you to download a `.elf` file to each Nios II DPX datapath processor, and launch a GDB server for each processor. GDB servers enable the SBT for Eclipse to communicate with and control the Nios II DPX MTPs through a debugging connection, such as JTAG over a USB-Blaster.

The SBT for Eclipse is a set of plugins to 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 DPX Multi-Threaded Processor (MTP) systems. The debugger allows you to perform many common debugging tasks. You can accomplish all software development tasks within Eclipse, including creating, editing, building, and debugging programs.

Alternatively, you can use the GDB command line to debug Nios II DPX software without Eclipse. Third-party GDB server GUIs are also available. The use of these tools is outside the scope of this handbook, and not supported by Altera.



For detailed documentation of the GDB command line, refer to *Debugging with GDB: The GNU Source-Level Debugger*, available from the Free Software Foundation website (www.fsf.org).

Getting Started

Modifying existing code is a common, easy way to learn to start writing software in a new environment. To get started with the Nios II DPX MTP, download the *Getting Started with the Nios II DPX MTP Tutorial* and its accompanying example design.



The *Getting Started with the Nios II DPX MTP Tutorial* and its accompanying example design are available on the [Packet Processing Design Example Using Nios II DPX Datapath Processor](#) page of the Altera website.

This section guides you through the most fundamental operations in the Nios II DPX debugging environment. It shows how to create an application project for the Nios II DPX MTP, 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 Altera development board.

The debugging flow is divided in two stages: debug setup and debugging.



If your Nios II DPX hardware was generated using Qsys v10.1, you cannot debug it using the Nios II EDS v11.0 or later. You must regenerate your Nios II DPX hardware and the associated BSP.



For information about regenerating your hardware in Qsys, refer to “Qsys Design Flow” in the *Creating a System with Qsys* chapter in *Volume 1: Design and Synthesis of the Quartus II Handbook*. For information about regenerating your BSP, refer to “Revising your BSP” in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer’s Handbook*.

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 DPX 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 Nios II DPX Launch Configuration

You run Nios II DPX projects using the Nios II DPX launch configuration. The Nios II DPX launch configuration is implemented by the Nios II plugins. It enables a GDB client running in Eclipse to attach to a running GDB server. You cannot use the Nios II Hardware launch configuration for a Nios II DPX project.

The Altera Bytestream Console

The workbench in Eclipse for Nios II includes a bytestream console, available through the Eclipse **Console** view. The Altera bytestream console enables you to see output from the processor's stdout device. For information about the Altera bytestream console, see [“Using the Altera Bytestream Console” on page 6–12](#).

Creating a Project

A complete Nios II DPX software project consists of an application project and a BSP project. This section describes how to create both projects in the Nios II Software Build Tools for Eclipse.


The BSP and the application project must be created separately. You cannot use a project template to create a Nios II DPX project.

Creating the 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 **Nios II Board Support Package**.

The **Nios II Board Support Package** wizard enables you to specify the following BSP parameters:

- The BSP project name
- The underlying hardware design (**.sopcinfo** file)
- The BSP project location
- The BSP type (Lightweight HAL) and version
- The target Nios II DPX MTP, if the system contains more than one core
- Additional arguments to the **nios2-bsp** script

-  For details about **nios2-bsp** command arguments, refer to “Details of BSP Creation” in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

Specifying the BSP Project Name

The SBT creates a directory with this name, to contain the BSP 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.

Specifying the BSP Project Location


The BSP project location is the parent directory in which the SBT creates the folder.

By default, the project location is under the directory containing the **.sopcinfo** file, in a folder named **software**.

To place your BSP in a different folder, turn off **Use default location**, and specify the BSP location in the **Project location** box.

Creating the Project

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

 For details about what happens when Nios II DPX projects are created, refer to “Nios II DPX BSP Creation” on page 8–9. For details about the **make clean** command, refer to “Makefiles” in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer’s Handbook*.

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, point to **Properties** and click **Nios II BSP Properties**.
- To modify parameters and settings in detail, use the Nios II BSP Editor, described in “Using the BSP Editor”.

Creating the Application

You create an application project using the **Nios II Application** wizard. To start the wizard, on the File menu, point to **New** and click **Nios II Application**.

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.

Specifying the BSP

To select the BSP location, browse to the BSP you created in “Creating the BSP” on page 6–3.

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.

Additional Arguments

In the **Additional Arguments** text box, you can specify any additional command-line arguments for the **nios2-app-generate-makefile** command. The full command line is displayed in the **Command** text box.

If you intend to run the project in the Nios II DPX MTP ModelSim® simulation environment, use the **Additional arguments** parameter to specify the location of the testbench file (**.spd**). The **.spd** file is located in the Quartus II project directory. Specify the path as follows:

```
--set QUARTUS_PROJECT_DIR=<relative path>
```

Altera recommends that you use a relative path name, to ensure that your project is independent of the installation directory.

For information about available command-line arguments for **nios2-app-generate-makefile**, refer to [Chapter 10, SBT Reference for the Nios II DPX MTP](#).

Creating the Project

When you have specified your application, you click **Finish** to create the project. The SBT creates the makefile and executes a **make clean** command on your application.



For details about what happens when Nios II DPX projects are created, refer to [“Nios II DPX BSP Creation” on page 8–9](#). For details about the **make clean** command, refer to [“Makefiles” in the Nios II Software Build Tools chapter of the Nios II Software Developer’s Handbook](#).

Adding Source Files

After the project is created, you have an empty framework for a Nios II DPX application, with a makefile that specifies no source files. You can add or create source files in the project to implement your software design.



For information about creating and adding source files to an Eclipse project, see the Eclipse help system.

When you create or add a source file, the SBT adds the file to your makefile. For details, see [“Makefiles and the Nios II SBT for Eclipse” on page 6–17](#).

Navigating the Project

When you have created a 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.

For an explanation of the folders and files in a Nios II DPX BSP, refer to “[Nios II DPX BSP Creation](#)” on page 8–9.

Building the Project

To build a 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.



For details about Nios II SBT commands and output, refer to [Chapter 10, SBT Reference for the Nios II DPX MTP](#).

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.

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 Quartus® II Programmer.

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



For details about programming an FPGA with Quartus II Programmer, refer to the [Quartus II Programmer](#) chapter in *Volume 3: Verification of the Quartus II Handbook*.

Debug Setup

In this stage, you set up a System Console debug session configuration script, one or more Eclipse debug configurations, and an optional launch group. This configuration information then is reused throughout the lifetime of the project.

Downloading the Project and Launching GDB Server

You use the GDB server control panel in System Console to download your software project(s) and launch the GDB debug server(s). To launch the GDB server control panel, perform the following steps:

1. In the Nios II menu, click **System Console**.
2. Select the application project to download and click **OK**.
3. In the Tools menu in System Console, click **GDB Server Control Panel**.

System Console determines what Nios II DPX datapath processors are present in the system, through the JTAG debug interface.

The GDB server control panel enables you to specify the following parameters:

- What Nios II DPX datapath processor(s) to use.
- What **.elf** file(s) to download to them.



The GDB server control panel does not validate the **.elf** file you select. Ensure that you select the correct **.elf** file, with a BSP generated for the version of the hardware design that is loaded on the target device.

- The desired TCP port number(s). The default TCP port number used by GDB is 10000.

If your system uses a dual-core Nios II DPX datapath processor, both Nios II DPX MTPs must run the same program code, because the cores share one program memory. Therefore, there is one **.elf** file per Nios II DPX datapath processor. You must start one GDB server for each Nios II DPX datapath processor.

If your system has more than one Nios II DPX datapath processor, enter the desired TCP port number for the first processor. The GDB server control panel assigns sequential port numbers to the remaining processors. You can manually modify these port numbers if you wish.

Make a note of the port number(s) you select. You need them when you create the debug configuration(s) in Eclipse, in [“Creating Eclipse Debug Configurations”](#).


After you have selected the **.elf** files and port numbers, select each Nios II DPX MTP and click **Launch Server**. The **Launch Server** button downloads the **.elf** and starts the GDB server.

By default, each Nios II DPX MTP thread pauses at the entry point (reset address). You can start the threads running from Eclipse. Alternatively, if you want the Nios II DPX MTP(s) to start running immediately, select the desired MTP(s) and click **Start Program**.

If your Nios II DPX hardware design is out of date, you might see the following error message:

```
The DPX hardware and debug driver versions are mismatched. Please
upgrade your DPX hardware.
```

This message appears if your Nios II DPX hardware was generated using Qsys v10.1, and you attempt to debug it using the Nios II EDS v11.0 or later. You must regenerate your Nios II DPX hardware and the associated BSP.

 For information about regenerating your hardware in Qsys, refer to “Qsys Design Flow” in the *Creating a System with Qsys* chapter in *Volume 1: Design and Synthesis* of the *Quartus II Handbook*. For information about regenerating your BSP, refer to “Revising your BSP” in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer’s Handbook*.

Saving the GDB Server Configuration

When you have determined that your GDB server configuration is satisfactory, you can save it as a Tcl script, by clicking **Save Session**. Saving the configuration as a Tcl script enables you to start future debugging sessions much more easily.

By default, the configuration Tcl script is saved in the software project directory. You can optionally save the script elsewhere in the file system. You can load it using the source Tcl command.

With a debugging configuration Tcl script, you can start a debugging session from the System Console command line. You can also run the script from the GDB command line, if you choose.


All actions that are available through the GUI are available as discrete Tcl commands.


Creating Eclipse Debug Configurations

Your project must be built and downloaded, and GDB server must be running, before you begin these steps.

The first step in debugging with Eclipse is to create a Nios II DPX debug configuration for each Nios II DPX datapath processor. This type of debug configuration enables Eclipse to attach to a GDB server that is already running.

Although the Eclipse documentation refers to a debug configuration as a “launch configuration,” be aware that the Nios II DPX launch configuration does not launch the Nios II DPX software. It merely attaches to the GDB server that you set up in [“Downloading the Project and Launching GDB Server” on page 6-7](#).

 Do not use the Nios II Hardware launch configuration for a Nios II DPX project.

To create a debug configuration, right-click the application project name, point to **Debug As**, and click **Debug Configurations**. In the **Debug Configurations** dialog box, select **Nios II DPX Hardware**, and click the **New** button (). Then perform these steps:

1. In the **Main** tab, ensure that the correct software project name and **.elf** file are selected.
2. In the **Debugger** tab, under **Debugger Options**, select **Connection**.
3. At **Port number**, ensure that the port number matches what you selected in [“Downloading the Project and Launching GDB Server” on page 6-7](#).
4. Click **Apply** to save the debug configuration.
5. Click **Close** to close the dialog box.

Multiple Nios II DPX Datapath Processors

You must create one debug configuration for each Nios II DPX datapath processor in the system. If you need multiple debug configurations, create them by performing the following steps:

1. Create a single debug configuration as described in the previous section.
2. To create an additional debug configuration, select the first debug configuration and click **Duplicate** on the right-click menu.
3. Edit the new debug configuration and adjust the port number to match the port number you noted in “[Downloading the Project and Launching GDB Server](#)” on [page 6-7](#).
4. Repeat steps 1 through 3 to create any additional needed debug configurations.



Make sure that the GDB server port numbers and the downloaded **.elf** file for each Nios II DPX datapath processor, as specified in the GDB server configuration, match the port number and **.elf** file in each debug configuration.

Multi-Core Launches

If you have multiple debug configurations, create an Eclipse launch group. Launch groups are an Eclipse feature that allows multiple debug configurations to be started at the same time. You choose which debug configurations are added to the group. You can use the launch group in any place where you can use a debug configuration.



For details about Eclipse launch groups, refer to the Eclipse help system.

Debugging the Project

This section describes how to debug a Nios II DPX program using the System Console and Nios II SBT for Eclipse. This section assumes that you have already performed the debug configuration steps described in “[Debug Setup](#)” on [page 6-6](#).

Here are the steps to debug a Nios II DPX datapath processor application:

1. Open the System Console GDB server control panel, as described in “[Downloading the Project and Launching GDB Server](#)” on [page 6-7](#).
2. Load a debug session by double clicking on the Tcl script that you created in “[Saving the GDB Server Configuration](#)” on [page 6-8](#). This script executes the following steps:
 - a. Downloads the **.elf** file
 - b. Starts the bytestream console
 - c. Starts the GDB server

You can optionally interact with the debugger by typing GDB commands inside the Console window of the corresponding GDB process to run, single step, and take other common debugging actions.



The **tbreak** command is not supported by the Nios II DPX datapath processor.

In the Nios II SBT for Eclipse, open the debug configuration you created in “[Creating Eclipse Debug Configurations](#)” on page 6–8. Click **Debug** to attach the Eclipse debugging session to threads already running or paused on the Nios II DPX datapath processor(s).

This command carries out the following actions:

- Establishes communications with the running GDB server
- Sets a breakpoint at the `.elf` entry point
- Pauses execution at the `.elf` entry point

If a processor is paused (the default when you start the debugging session), you can start or resume the thread with the Eclipse **Resume** button.

You cannot use an Eclipse run configuration for a Nios II DPX project. If you wish to run the Nios II DPX software, you can use a debug configuration, and disable all breakpoints.



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

Thread Representation

Each Nios II DPX MTP is represented as a GDB process. Its threads are represented as GDB threads. Each thread is labeled with a thread ID.

A single-core Nios II DPX datapath processor has eight threads. A dual-core processor has 16 threads.

The SBT for Eclipse treats a dual-core processor like a single-core processor with 16 threads.

Hardware thread numbers range from 0 to $\langle n \rangle - 1$, where $\langle n \rangle$ is the number of threads. However, Eclipse thread IDs range from 1 to $\langle n \rangle$. Because threads are interchangeable, normally you need not be concerned with hardware thread numbers.


You can select a thread in the Debug view and start or stop it individually with the Eclipse Resume and Suspend commands. Alternatively, you can select the parent node and run all threads.

Debugging Actions

The Eclipse debugger with the Nios II plugins provides a Nios II perspective, allowing you to perform many common debugging tasks. The debugging actions you can perform with the Nios II SBT for Eclipse include the following actions:

- Controlling program execution with commands such as:
 - Suspend (pause)
 - Resume
 - Terminate
 - Step Into
 - Step Over
 - Step Return

- Setting breakpoints
- Viewing disassembly
- Instruction stepping mode


You can enable Eclipse to step through individual processor instructions by clicking . When you turn on instruction stepping mode, the **DSF Disassembly** view automatically opens.

- Displaying and changing the values of local and global variables in the following formats:
 - Binary
 - Decimal
 - Hexadecimal
- Viewing and editing registers in the following formats:
 - Binary
 - Decimal
 - Hexadecimal


The debugger can view and edit Nios II DPX extension registers. However, it cannot edit the read-only registers (such as `RX0` and `CRi0`), and it cannot read the write-only registers (such as `TX0` and `CRo0`). Because the read-only and write-only registers share register addresses, if you attempt to write to a read-only register (such as `RX0`), you write to the corresponding write-only register (such as `TX0`). Also, if you attempt to read from a write-only register, you read the value of the corresponding read-only register.

 For detailed information about Nios II DPX extension registers, refer to [“Using the Nios II DPX Extension Registers” on page 5–19](#).

- Viewing and editing memory in the following formats:
 - Hexadecimal
 - ASCII
 - Signed integer
 - Unsigned integer
- Viewing stack frames in the **Debug** view
- Typing GDB commands in the GDB console, in the Eclipse **Console** view

 The `tbreak` command is not supported by the Nios II DPX datapath processor.

Console output to `stdout` appears in the System Console.

 For information about debugging commands supported by Eclipse and the CDT plugins, refer to the Eclipse help system.

If you modify and rebuild your source code, you must use System Console to reload the `.elf` and restart the GDB server. For more information, see “[Stopping and Restarting](#)”.

Breakpoints

The Nios II DPX debugging environment offers both software and hardware breakpoints.

Software Breakpoints

Software breakpoints are set using Eclipse breakpoint infrastructure. Eclipse sets the breakpoint for all threads, in all running debug sessions.



You cannot filter software breakpoints by thread. Although Eclipse offers GDB thread-specific breakpoints, this feature is designed for software threads, and does not work correctly with Nios II DPX hardware threads. To set a breakpoint on a specific hardware thread, use a hardware breakpoint.

Hardware Breakpoints

Eclipse does not have infrastructure to set hardware breakpoints. You must use the `hbreak` GDB command to set thread-specific hardware breakpoints. This command must be issued inside the GDB server control panel, either in Eclipse or in System Console.

You can set no more than four hardware breakpoints at a time.

Stopping and Restarting

When you stop the debugger, for example to modify and rebuild the code, the GDB server stops listening. You must restart the GDB server from the GDB server control panel as follows:

1. In the GDB server control panel, stop the GDB server by clicking **Stop Server**.
2. Click **Launch Server** to restart the server.

Eclipse breakpoints are preserved from the previous debug session.

If you have modified the code, the breakpoint locations might no longer be relevant. You must clear and reset your breakpoints.

If you need to assert a hardware reset, close any open GDB servers and other System Console services connected to the Nios II DPX datapath processor, such as views of the processor or memory.

Using the Altera Bytestream Console

The Altera bytestream console enables you to see output from the processor's `stdout` device. The function of the Altera bytestream console is similar to the `nios2-terminal` command-line utility.

Open the Altera 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 Altera 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.



For information about how System Console recognizes bytestreams, refer to the *Analyzing and Debugging Designs with the System Console* chapter in *Volume 3: Verification of the Quartus II Handbook*.



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

Context-Specific Debugging for Nios II DPX Systems

When debugging a datapath processing system, you need to be able to follow a particular data packet, or context, through the system. A context usually follows a path that takes it from an input PE, through multiple PEs including the Nios II DPX datapath processor, to finally reach an output PE. As it traverses this path, the context and its data are transformed by the action of hardware and software tasks.

To effectively analyze and debug the system, you need to follow the execution of the many software tasks that act on selected contexts. The software debugging tools enable you to trace the context whenever it returns to the Nios II DPX datapath processor.

You cannot use the software debugging tools to trace a context through hardware PEs.

To achieve this context-specific debugging, you select a context that you are interested in following, and set a debug flag associated with that context.

The debug flag is passed from PE to PE in messages on the same context. When the debug flag returns to the Nios II DPX datapath processor, it activates a hardware breakpoint, causing the thread to break on entry to the task. This allows you to continue debugging with the same context.

Debug Flag Management

For any suspended thread, you can set or reset a debug flag to accompany the context being processed by the thread. The debug flag is physically associated with the message sent by the task when it terminates. When a task is started by a message that contains a debug flag, all messages sent by that task also have the debug flag set. Thus, the debug flag remains with the context until it is explicitly removed.

You control context-specific debug flags by typing commands into the GDB console. The debug flag management commands are as follows:

monitor set_debug_flag <thread-id> <value>

This command sets or resets the debug flag currently being processed by the thread specified in <thread-id>. <value> can be 0 or 1, where 0 means reset the flag and 1 means set the flag. The thread must already be suspended before you issue this command. Otherwise the command fails with an error message.

You specify a thread with the GDB thread ID. You can see the thread ID by examining the Eclipse thread list.

monitor get_debug_flag <thread-id>

This command queries the value of the debug flag for the message currently being processed by the specified thread. The return value is 0 if the flag is not set, and 1 if the flag is set. The thread must already be suspended before you issue this command. Otherwise the command fails with an error message.

monitor set_packet_debug <value>

This command enables or disables the hardware breakpoint which causes the Nios II DPX MTP to break on the first instruction of a task ready to process a message marked with a debug flag. <value> can be 0 to disable the breakpoint or 1 to enable the breakpoint. In the case of a dual-core Nios II DPX datapath processor, the setting applies to both cores.

monitor get_packet_debug

This command queries whether the processor is enabled to break on the arrival of a message marked with a debug flag.

Working with Stand-Alone Systems

In the Nios II DPX stand-alone hardware development flow, you create a Nios II DPX datapath processor in Qsys, and then export it to your system manually.



For information about working in the stand-alone flow, refer to the *Instantiating the Nios II DPX Datapath Processor* chapter, in the *Nios II DPX Hardware Reference* section of the *Nios II DPX Datapath Processor Handbook*.

In the stand-alone flow, special steps are required to create a BSP. Extra steps are required because the data memory, external to the Nios II DPX datapath processor, is not represented in the Nios II DPX .sopcinfo file. This requires the data memory to be specified during the BSP creation.

You must manually specify the external memory and the section mapping in the BSP editor. After you do this, you can create a Tcl script to automate the steps in case you need to recreate the BSP.

To create your BSP the first time, and create a Tcl script, perform the following steps.

1. Create a new BSP, as described in “[Creating a New BSP](#)” on page 6-26. Save the BSP.
2. Add the data memory device to the linker script as follows:
 - a. On the **Linker Script** tab, click **Add Memory Device**. On the **Main** tab, fill in **Device Name**, **Memory Size**, and **Base Address**.
 - b. If you need to generate memory initialization files for your user-defined memory, you must specify additional memory parameters. Use the **Advanced** tab to specify these settings, as described in “[Creating Memory Initialization Files](#)”.
 - c. Click **OK**.

3. Add the data memory region to the linker script as follows:
 - a. In the **Linker Memory Regions** table, add the linker memory region for the data memory by clicking **Add**. Enter the region name and region size (size of the memory). Select the memory device specified in the previous step, and enter the memory offset.
 - b. Click **Add**.

For more information about the **Linker Script** tab, see “[Linker Regions](#)” on [page 6–23](#).

4. Enter the thread stack size (optional). In the **Main** Tab, in the **Advanced** settings, select **lwhal**. Enter the stack size under **thread_stack_size**, ensuring that the memory is large enough to contain a stack for each thread.
5. On the **Linker Script** tab, click **Restore Defaults**. This command applies the default Tcl scripts to your BSP, to add any unassigned sections to the data RAM.
6. Save and generate the BSP.
7. If desired, export your memory settings to a Tcl script as described in “[Exporting a Tcl Script](#)” on [page 6–26](#).

To recreate the BSP with the same external memory, use the script as described in “[Using a Tcl Script in BSP Creation](#)” on [page 6–27](#).

Creating Memory Initialization Files

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

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 Qsys 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**.

In the **Advanced** tab, you can control the following memory characteristics:

- The physical memory width.
- The device’s path and name in the Qsys hierarchy.
- The memory initialization file parameter name. Every memory device can have an HDL parameter specifying the name of the initialization file. The Nios II DPX ModelSim launch configuration overrides the HDL parameter to specify the memory initialization filename.
- 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

The **Mem init filename** parameter is not used in Nios II DPX systems. You can disregard it.

Running a Nios II DPX System with ModelSim

To run a Nios II DPX system with ModelSim, you must first create a testbench and specify memory initialization files. Creating the software projects is nearly the same as when you run the project on hardware.




For information about creating a testbench and specifying memory initialization files, see “RTL Simulation” in the *System Verification* chapter in the *Nios II DPX Hardware Reference* section of the *Nios II DPX Datapath Processor Handbook*.

To prepare your software for ModelSim simulation, perform the following steps:

1. Create your software project, as described in “Creating a Project” on page 6-3.

Be sure to specify the Quartus II project path, as described in “Additional Arguments” on page 6-5.

If you are creating software for a stand-alone system, you must take special steps to create memory initialization files correctly. These steps are described in “Creating Memory Initialization Files” on page 6-15.

2. Build your software project, as described in “Building the Project” on page 6-6.
3. Create a ModelSim launch configuration as follows:
 - 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 Nios II DPX datapath processors, create a launch configuration for each processor, and create a launch group, as described in “Multiple Nios II DPX Datapath Processors” on page 6-9.

4. Open the debug 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 `ld`↵.



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

SEVERE: The Quartus II 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.

To correct this error, perform the following steps:

1. Click the **Advanced** button.
2. At **Quartus II project directory**, browse to the directory containing your Quartus II project **.spd** file.
3. Click **Close**.

To avoid this error condition, specify the Quartus II project directory when you create your application project, as described in ["Additional Arguments" on page 6-5](#).

Makefiles and the Nios II SBT for Eclipse

The Nios II SBT for Eclipse creates and manages the makefiles for Nios II DPX MTP software projects. When you create a project, the Nios II 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 Nios II 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 Nios II 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

Nios II DPX MTP 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 **Nios II Application Properties** or **Nios II Library Properties** dialog box. To open the dialog box, right-click the project, click **Properties**, and click **Nios II Application Properties** or **Nios II Library Properties**.

Table 6-1 lists GUI actions that make changes to an application or user library makefile under Eclipse source management.

Table 6-1. Modifying a Makefile with Eclipse Source Management

Modification	Where Modified
Specifying the application or user library name	Nios II Application Properties or Nios II Library Properties dialog box.
Adding or removing source files	 Refer to the Eclipse help system.
Specifying a path to an associated BSP	Project References dialog box.
Specifying a path to an associated user library	Project References dialog box.
Enabling, disabling or modifying compiler options	Nios II Application Properties or Nios II Library Properties dialog box.

After the SBT has created a makefile, you can modify the makefile in the following ways:

- With the Nios II SBT for Eclipse, as described in Table 6-1.
- With Nios II SBT commands from the Nios II 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 Nios II 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 Nios II 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 Nios II DPX MTP makefile temporarily, without deleting it from the project, you can use the **Remove from Nios II Build** command. Right-click the resource and click **Remove from Nios II 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 Nios II Build**.



Do not use the Eclipse **Exclude from build** command. With Nios II DPX MTP software projects, you must use the **Remove from Nios II Build** and **Add to Nios II 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 Nios II Build** and **Add to Nios II Build** commands with linked resources. When a linked resource is excluded from the build, its icon is modified with a white dot.



For information about working with linked resources, refer to the Eclipse help system.

User Source Management

You can remove a makefile from source management control through the **Nios II Application Properties** or **Nios II 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.

Editing a makefile manually is an advanced technique. Altera 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 Nios II Build** command
- The **Remove from Nios II Build** command

Table 6–2 lists GUI actions that make changes to an application or user library makefile under user source management.

Table 6–2. Modifying a Makefile with User Source Management


Modification	Where Modified
Specifying the application or user library name	Nios II Application Properties or Nios II Library Properties dialog box
Specifying a path to an associated BSP	Project References dialog box
Specifying a path to an associated user library	Project References dialog box
Enabling, disabling or modifying compiler options	Nios II Application Properties or Nios II Library Properties dialog box

 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 Nios II Build**.

BSP Source Management

Nios II BSP makefiles are handled differently from application and user library makefiles. BSP makefiles are based on the BSP type, 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.

 For further details about specifying BSPs, refer to “Using the BSP Editor”.

Using the BSP Editor

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

Tcl Scripting and the Nios II BSP Editor

The Nios II 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 Nios II BSP Editor

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

- Right-click an existing project, point to **Nios II**, and click **BSP Editor**. The editor loads the BSP Settings File (**.bsp**) associated with your project, and is ready to update it.

- On the Nios II menu, click **Nios II 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 **Nios II BSP Properties**, and click **BSP Editor**. The editor loads your **.bsp** file for update.

The Nios II BSP Editor Screen Layout

The Nios II 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 Nios II 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.



For detailed descriptions of settings defined for Altera-provided BSP types, software packages, and drivers, refer to [Chapter 10, SBT Reference for the Nios II DPX MTP](#).

The Main Tab

The **Main** tab presents general settings and parameters, and BSP type 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 BSP type and version



You cannot change the BSP type in an existing BSP. You must create a new BSP based on the desired BSP type.

- 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 BSP type.

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.

Software package and driver settings are presented separately, as described in “[The Software Packages Tab](#)” and “[The Drivers Tab](#)”.

The advanced thread stack size setting, `lwhal.thread_stack_size`, is usually an important setting for Nios II DPX BSPs.

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 BSP type 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.

General settings, BSP type settings, and driver settings are presented separately, as described in “[The Main Tab](#)” and “[The Drivers Tab](#)”.

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. For Altera components, refer to the *Embedded Peripherals IP User Guide*.

General settings, BSP type settings, and software package settings are presented separately, as described in “[The Main Tab](#)” and “[The Software Packages Tab](#)”.

The Linker Script Tab

The **Linker Script** tab allows you to view available memory in your 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. If there is a problem, a message appears in the **Problems** tab in the console area, as described in “[The Problems Tab](#)” on page 6-25.



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

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 Nios II DPX 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.

 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.

 For information about `add_memory_device` and other SBT Tcl commands, refer to “Tcl Commands” in [Chapter 10, SBT Reference for the Nios II DPX MTP](#).

- **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 Nios II DPX 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.

This representation is not to scale.

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.



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.



For information about `set_ignore_file` and other SBT Tcl commands, refer to “Tcl Commands” in [Chapter 10, SBT Reference for the Nios II DPX MTP](#).

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 BSP type, 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 following sections describe each 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.



To run a Tcl script during BSP creation, refer to [“Using a Tcl Script in BSP Creation”](#). For details about default BSP settings, refer to [“Specifying BSP Defaults for the Nios II DPX MTP” on page 8–14](#). For detailed information about Tcl command usage, refer to [“Tcl Commands” on page 10–42](#). For information about recreating and regenerating BSPs, refer to [“Revising Your BSP”](#) in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer’s Handbook*.

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 (Lightweight HAL) and version.
- 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 Nios II 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.

 “**Exporting a Tcl Script**” describes how to create a Tcl script from an existing BSP. Refer to “*Specifying BSP Defaults for the Nios II DPX MTP*” on page 8–14.


BSP Validation Errors

If you modify a Nios II DPX 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.

 For complete information about updating and recreating BSPs, refer to “**Revising Your BSP**” in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer’s Handbook*.

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.

 For details about exporting and using Tcl scripts, refer to “**Exporting a Tcl Script**” and “**Using a Tcl Script in BSP Creation**”. For a detailed discussion of updating BSPs for modified Nios II DPX systems, refer to “**Revising Your BSP**” in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer’s Handbook*.

Configuring Component Search Paths

By default, the SBT discovers system components using the same search algorithm as Qsys. You can define additional search paths to be used for locating components.

You define additional search paths through the **Edit Custom Search Paths** dialog box. In the Tools menu, click **Options**, select **BSP Component Search Paths**, and click **Custom Component Search Paths**. You can specify multiple search paths. Each path can be recursive.

Importing a Command-Line Project

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

Your command-line C application, and its associated BSP, might be created on the command line. Regardless of its origin, any Nios II SBT command-line project is ready to import into the Nios II SBT for Eclipse. No additional preparation is necessary.

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

- Command-line 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.

The Nios II 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.



For information about creating projects with the command line, refer to [Chapter 7, Getting Started from 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 application
- Import a supporting project
- Debug a command-line C application
- Edit command-line 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 Nios II 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 Application

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

1. Start the Nios II SBT for Eclipse.
2. On the File menu, click **Import**. The **Import** dialog box appears.
3. Expand the **Nios II Software Build Tools Project** folder, and select **Import Nios II Software Build Tools Project**.
4. Click **Next**. The **File Import** wizard appears.
5. Click **Browse** and locate the directory containing the 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.



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 Nios II SBT for Eclipse can build, debug, and run the complete program, including the BSP and any libraries. The Nios II SBT for Eclipse builds the project using the SBT makefiles in your imported C application project. Eclipse displays and steps through application source code exactly as if the project were created in the Nios II 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. The process of importing supporting projects, such as BSPs and libraries, is described in “[Import a Supporting Project](#)”.

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.



For information about managing linked resources, refer to “[Absolute Source Paths and Linked Resources](#)” on page 6-18.

Import a Supporting Project

While debugging a 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 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.

The steps for importing a supporting project are exactly the same as those shown in [“Import a Command-Line C Application”](#).

User-Managed Source Files

When you import a Nios II DPX MTP application or user library project, the Nios II 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 Nios II DPX MTP 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 BSP type, BSP settings, selected software packages, and selected drivers. You do not specify BSP source files directly.



For details about how the SBT for Eclipse handles makefiles with user-managed sources, refer to [“User Source Management”](#) on page 6-19.

Packaging a Library for Reuse

This section shows how to create and use a library archive file (.a) in the Nios II 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 Nios II user library
2. Create a Nios II 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 **Nios II 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 a Nios II application project, perform the following steps:

1. Create your Nios II application project as described in [“Creating a Project” on page 6-3](#).
2. To set the library path in the application project, right-click the project, and click **Properties**.
3. Expand **Nios II Application Properties**. In **Nios II 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 in [“Creating the 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:

- Nios II library archive file (**.a**)
- Software header files (**.h**)

Memory Initialization Files

Sometimes it is useful to generate memory initialization files. For example, to program your FPGA with a complete, running Nios II DPX 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 Nios II SBT for Eclipse can create **.hex** files and other memory initialization formats.

To generate correct memory initialization files, the Nios II 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.



If your system contains a user-defined memory, you must specify these details manually. For information, see [“Creating Memory Initialization Files” on page 6-15](#).

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 Quartus II IP File (**.qip**). The **.qip** file tells the Quartus II software where to find the initialization files.
5. Add the **.qip** file to your Quartus II project.
6. Recompile your Quartus II project.

Managing Toolchains in Eclipse

To change the GCC toolchain version in Eclipse, right-click the project and click **Properties**. In the **Properties** dialog box, expand the **C/C++ Build** tab and select **Tool Chain Editor**. Select the appropriate Nios II DPX MTP GCC toolchain, depending on your host operating system.

After you select the toolchain, the SBT for Eclipse continues to use that toolchain for your project unless you change it again.



If you move the project to a different host platform, you must manually change to the appropriate toolchain for the new host platform. For example, if you move a project from a Windows host to a Linux host, use the **Properties** dialog box to select **Linux Nios II DPX MTP GCC 4**.



For general information about the GCC toolchains, refer to *“Nios II DPX Software Development Tools” on page 8–1*. For information about selecting the toolchain on the command line, refer to *Chapter 7, Getting Started from the Command Line*.

Eclipse Usage Notes

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

Thread-Specific Breakpoints

You cannot filter software breakpoints by thread. Although Eclipse offers GDB thread-specific breakpoints, this feature is designed for software threads, and does not work correctly with Nios II DPX hardware threads. To set a breakpoint on a specific hardware thread, use a hardware breakpoint, as described in *“Breakpoints” on page 6–12*.

DSF Disassembly View Required

To view disassembled Nios II DPX code in Eclipse, you must use the **DSF Disassembly** view. You cannot use the standard Eclipse **Disassembly** view to view disassembled Nios II DPX code.

When you turn on instruction stepping mode, the **DSF Disassembly** view automatically opens.

Configuring Application and Library Properties

To configure project properties specific to Nios II SBT application and library projects, use the **Nios II Application Properties** and **Nios II 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, **Nios II Application Properties** or **Nios II Library Properties** tab appears in the list of tabs. Click the appropriate Properties tab to open it.

The **Nios II Application Properties** and **Nios II 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 Nios II BSP Editor.

For detailed information about the BSP Editor, refer to [“Using the BSP Editor” on page 6-20](#).

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.

Selecting the Correct Launch Configuration Type

If you try to debug a Nios II DPX MTP software project as a CDT Local C/C++ Application launch configuration type, you see an error message, and the Nios II 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 Nios II plugins, you must use a Nios II launch configuration type. For details, see [“Debugging the Project” on page 6–9](#).

Renaming Nios II DPX MTP Projects

To rename a project in the Nios II 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 Nios II Command Shell, as described in [Chapter 7, Getting Started from the Command Line](#).
2. Start the Nios II 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, point to **External Tools**, and click **External Tools Configurations**.
4. Create a new tools configuration, or open an existing tools configuration.

- On the **Main** tab, set **Location** and **Argument** as shown in Table 6-3.

Table 6-3. Location and Argument to Run Shell Script from Eclipse

Platform	Location	Argument
Windows	\${env_var:QUARTUS_ROOTDIR}\bin\cygwin\bin\sh.exe	-c "<script name> <script args>"
Linux	\${env_var:SOPC_KIT_NIOS2}/bin/<script name>	<script args>

For example, to run the command `elf2hex --help`, set **Location** and **Argument** as shown in Table 6-4.

Table 6-4. Location and Argument to Run elf2hex --help from Eclipse

Platform	Location	Argument
Windows	\${env_var:QUARTUS_ROOTDIR}\bin\cygwin\bin\sh.exe	-c "elf2hex --help"
Linux	\${env_var:SOPC_KIT_NIOS2}/bin/elf2hex	--help

- 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.

- Click **Run**. The command executes in the Nios II Command Shell, and the command output appears in the Eclipse **Console** tab.

Must Use Nios II Build Configuration

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

CDT Limitations

The features listed in the left column of Table 6-5 are supported by the Eclipse CDT plugins, but are not supported by Nios II plugins. The right column lists alternative features supported by the Nios II plugins.

Table 6-5. Eclipse CDT Features Not Supported by the Nios II Plugins (Part 1 of 2)

Unsupported CDT Feature	Alternative Nios II Feature
New Project Wizard C/C++ <ul style="list-style-type: none"> C Project C++ Project C/C++ <ul style="list-style-type: none"> Convert to a C/C++ Project Source Folder 	To create a new project, use one of the following Nios II wizards: <ul style="list-style-type: none"> Nios II Application Nios II Board Support Package Nios II Library

Table 6–5. Eclipse CDT Features Not Supported by the Nios II Plugins (Part 2 of 2)

Unsupported CDT Feature	Alternative Nios II Feature
Build configurations <ul style="list-style-type: none"> ■ Right-click project and point to Build Configurations 	The Nios II plugins only support a single build configuration.
Exclude from Build (from v10.0 onwards) Right-click source files	Use Remove from Nios II Build and Add to Nios II Build .
Project Properties C/C++ Build <ul style="list-style-type: none"> ■ Builder Settings <ul style="list-style-type: none"> ■ Makefile generation ■ Build location ■ Behaviour <ul style="list-style-type: none"> ■ Build on resource save (Auto build) ■ Build Variables ■ Discovery Options ■ Environment ■ Settings ■ Tool Chain Editor <ul style="list-style-type: none"> ■ Current builder ■ Used tool C/C++ General <ul style="list-style-type: none"> ■ Enable project specific settings ■ Documentation ■ File Types ■ Indexer <ul style="list-style-type: none"> ■ Build configuration for the indexer ■ Language Mappings ■ Paths and Symbols 	<p>By default, the Nios II SBT generates makefiles automatically. The build location is fixed.</p> <p>To change the toolchain, use the Current tool chain option</p> <p>The Nios II plugins only support a single build configuration.</p> <p>Use Nios II Application Properties and Nios II Application Paths</p>
Window Preferences C/C++ <ul style="list-style-type: none"> ■ Build scope ■ Build project configurations ■ Build Variables ■ Environment ■ File Types ■ Indexer <ul style="list-style-type: none"> ■ Build configuration for the indexer ■ Language Mappings ■ New CDT project wizard 	<p>The Nios II plugins only support a single build configuration.</p> <p>The Nios II plugins only support a single build configuration.</p>

The Nios II SBT allows you to construct a wide variety of complex 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.

This chapter introduces you to project creation with the SBT at the command line.

This chapter includes the following sections:

- “Advantages of the Command Line”
- “Outline of the Nios II SBT Command-Line Interface”
- “Scripting Basics” on page 7-2
- “Running make” on page 7-5

Advantages of the Command Line

The Nios II SBT command line offers the following advantages over the Nios II 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 Nios II SBT command-line interface is designed to work in the Nios II Command Shell.



For details about the Nios II Command Shell, refer to “The Nios II Command Shell” on page 7-2.

Outline of the Nios II SBT Command-Line Interface

The Nios II SBT command-line interface consists of:

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

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

Utilities

The Nios II 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 Nios II SBT resides in the `<Nios II EDS install path>/sdk2/bin` directory.



Refer to “Altera-Provided Development Tools” in the *Nios II Software Build Tools* chapter of the *Nios II Software Developer’s Handbook* for a summary of the command-line utilities provided by the Nios II SBT.

The nios2-bsp Script

nios2-bsp is a convenience script that extends the capabilities provided by the utilities. **nios2-bsp** either creates or updates a BSP, depending on whether the BSP you specify already exists.

Tcl Commands

Tcl commands are a crucial component of the Nios II 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 Nios II Command Shell

The Nios II Command Shell is a bash command-line environment initialized with the correct settings to run Nios II command-line tools.

To open the Nios II Command Shell, execute the following steps, depending on your environment:

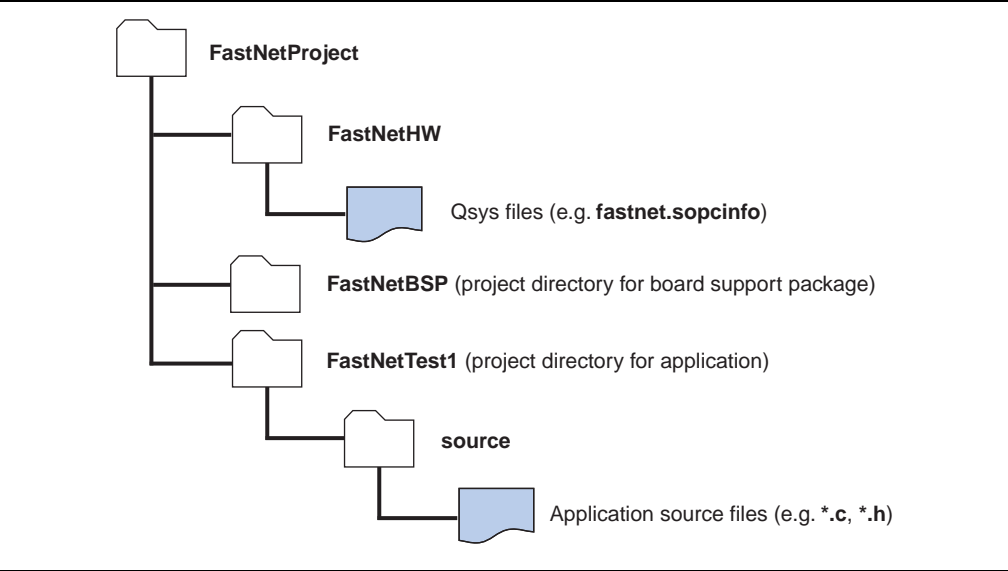
- In the Windows operating system, on the Start menu, point to **Programs > Altera > Nios II EDS <version>**. Select GCC 4 by clicking **Nios II <version> Command Shell**.
- In the Linux operating system, in a command shell, change directories to `<Nios II EDS install path>`, and select GCC 4 by typing `nios2_command_shell.sh`.

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 DPX system. Furthermore, assume that you have organized the hardware design files and the software source files as shown in [Figure 7-1](#).

Figure 7-1. Simple Software Project Directory Structure



Creating a BSP with a Script

One easy method for creating a BSP is to use the **nios2-bsp** script. The script in [Example 7-1](#) creates a BSP and then builds it.

Example 7-1. nios2-bsp

```
nios2-bsp lwhal . ../user/data/FastNetProject/FastNetHW/
make
```

[Table 7-1](#) shows the meaning of each argument to the **nios2-bsp** script in [Example 7-1](#).


 For additional information about the **nios2-bsp** command, refer to “Nios II Software Build Tools Utilities” in the “[Nios II Software Build Tools Utilities](#)” on [page 10-1](#).

Table 7-1. nios2-bsp Example Arguments


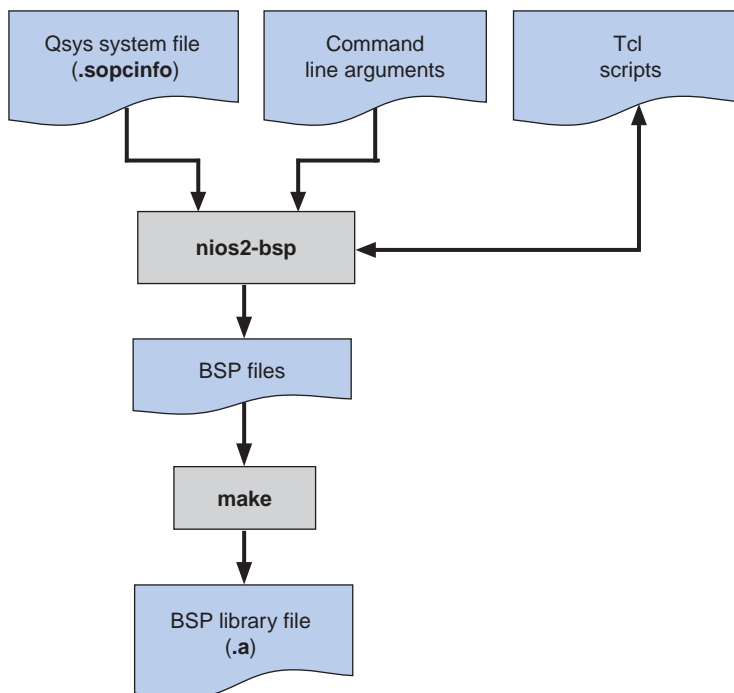
Argument	Purpose	 Further Information
lwhal	Sets the BSP type to Lightweight HAL	“Settings” on page 10-28
.	Specifies the directory in which the BSP is to be created	—
../SOPC/	Points to the location of the hardware project	—

Figure 7-2 shows the flow to create a BSP using the **nios2-bsp** script. 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 7-2. nios2-bsp Command Flow



Creating an Application Project with a Script

You use **nios2-app-generate-makefile** to create application projects. The script in Example 7-2 creates an application project and builds it.


Example 7-2. nios2-app-generate-makefile

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

Table 7-2 shows the meaning of each argument in Example 7-2.

Table 7-2. nios2-app-generate-makefile Example Arguments

Argument	Purpose
<code>--bsp-dir ../BSP</code>	Specifies the location of the BSP on which this application is based
<code>--elf-name test.elf</code>	Specifies the name of the executable file
<code>--src-dir source/</code>	Tells nios2-app-generate-makefile where to find the C source files

- 
- For further information about each command argument in [Table 7-2](#), refer to “Nios II Software Build Tools Utilities” in the “[Nios II Software Build Tools Utilities](#)” on [page 10-1](#).

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 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 memory initialization files for a Nios II DPX system, you can use the Nios II 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.

This chapter describes board support packages generated by the SBT for the Nios II DPX MTP. It enables you to understand the BSP creation process introduced in [Chapter 6, Getting Started with the Graphical User Interface](#) and [Chapter 7, Getting Started from the Command Line](#).

This chapter includes a description of the LWHAL and a comprehensive reference to the LWHAL API functions.

Nios II DPX Software Development Tools

The Nios II DPX GNU Toolchain

The GNU toolchain for the Nios II DPX MTP includes the following utilities:

- `nios2dpx-elf-addr2line`
- `nios2dpx-elf-ar`
- `nios2dpx-elf-as`
- `nios2dpx-elf-c++filt`
- `nios2dpx-elf-cpp`
- `nios2dpx-elf-gcc`
- `nios2dpx-elf-gcc-4.1.2`
- `nios2dpx-elf-gccbug`
- `nios2dpx-elf-gcov`
- `nios2dpx-elf-gdb`
- `nios2dpx-elf-gprof`
- `nios2dpx-elf-ld`
- `nios2dpx-elf-nm`
- `nios2dpx-elf-objcopy`
- `nios2dpx-elf-objdump`
- `nios2dpx-elf-ranlib`
- `nios2dpx-elf-readelf`
- `nios2dpx-elf-size`
- `nios2dpx-elf-strings`
- `nios2dpx-elf-strip`

The Nios II DPX toolchain port is based on the following GNU tool versions:

- binutils 2.20
- GCC 4.1.2
- newlib 1.16
- GDB 7.0



The Nios II DPX toolchain does not support the C++ language.

newlib for the Nios II DPX MTP

The Nios II DPX toolchain supports a single, precompiled variant of newlib. The supported variant of newlib is compiled with the following attributes:

- BE-8 endianness
- No profiling
- No `mulx` instruction support
- Support for a hardware multiply

newlib is configured with `-DMALLOC_PROVIDED`. However, the standard `malloc()`, `calloc()`, `free()`, and `realloc()` functions are not supported. Instead, newlib supports `alt_malloc()`, `alt_free()`, and `alt_calloc()`, as described in “[Managing Memory Usage with the LWHAL](#)” on page 8-7.

Using the Nios II Software Build Tools

To create software projects for the Nios II DPX MTP, you use the Nios II SBT. The SBT utilities work the same for the Nios II DPX MTP as for the Nios II processor. The Nios II DPX MTP entails the following differences in how you use the tools:

- The runtime library for the Nios II DPX MTP is based on the Altera Lightweight Hardware Abstraction Layer (LWHAL). The LWHAL, and the associated newlib implementation, have substantially fewer features, and a smaller footprint, than the HAL environment commonly used with the Nios II processor. The LWHAL is described in the next section.
- A different set of default Tcl scripts is provided to create Nios II DPX BSPs. The Nios II DPX default Tcl scripts are described in “[Specifying BSP Defaults for the Nios II DPX MTP](#)” on page 8-14.
- The Nios II boot configurations do not apply to a Nios II DPX system.



For detailed information about how to use the Nios II SBT, refer to the [Nios II Software Build Tools](#) chapter of the *Nios II Software Developer's Handbook*.

The Lightweight Hardware Abstraction Layer (LWHAL)

The LWHAL is a very lightweight runtime environment with some newlib-like features.

Startup Code

The LWHAL provides system initialization code in the C runtime library (**crt0.S**). This code performs the following startup sequence for each thread:

- Sets the `r0` register to 0
- Configures the `gp` register (all threads have the same value)
- Configures the `sp` register
- Waits for the `.rdata` section to be initialized (if the `lwhal.alt_load_copy_rdata` BSP setting is enabled)
- Calls `main()`

One thread (thread 0) performs the following additional steps:

- Initializes the `.bss` region to zeroes
- Copies `.rdata` to RAM (if the `lwhal.alt_load_copy_rdata` BSP setting is enabled)
- Initializes heap if used

You provide the `main()` function. The function prototype for `main()` is as follows:

```
int main (void)
```

The return value from `main()` is ignored.

crt0.S executes the `exit` instruction after the startup sequence is complete. The `exit` instruction tells the external logic that the thread is idle.

crt0.o gets linked first in the `.text.entry` section, at the reset vector specified in the hardware. When a reset occurs, each thread begins execution at the reset vector.

Stack

Each thread has its own stack pointer. All threads have the same stack size, defined by the `lwhal.thread_stack_size` BSP setting.

If you select a stack size less than 384, the minimum required for the LWHAL `printf()` function, you see a compiler warning at build time. You can disable this warning with the `lwhal.enable_small_stack` BSP setting.



For detailed information about the `lwhal.enable_small_stack` setting, refer to [“Settings Reference” on page 10–30](#).

Device Drivers

The LWHAL BSP includes the following device drivers:

- JTAG UART
- UART
- Mutex
- PIO

JTAG UART Driver

The LWHAL's simplified JTAG UART driver uses polled mode (blocking mode), and no interrupts.

UART Driver

The LWHAL's simplified Altera Avalon UART driver uses polled mode (blocking mode), and no interrupts.

PIO Driver

The PIO driver for Nios II DPX MTP consists of accessor macros to write to and read from the PIO registers.

Mutex Driver

The mutex core provides a protocol to ensure mutually exclusive ownership of a shared resource. Multiprocessor and multi-threaded environments can use the mutex core to determine which processor and thread owns the mutex.



In an LWHAL project, you must use the LWHAL driver for the Mutex core. The LWHAL mutex driver is described in “[Lightweight HAL Function Reference](#)” on page 8-17.



For detail about the mutex core, refer to the *Mutex Core* chapter of the [Embedded Peripherals IP User Guide](#).

Differences from newlib

The LWHAL environment makes it possible to link with applications written for the newlib C library. However, the majority of POSIX.1 (IEEE 1003.1) system services, as well as newlib library functions that depend on them, are not needed for DPX task development.

POSIX.1 Stubs

The unimplemented POSIX.1 functions are stubs, so any newlib library functions that depend on them fail at runtime.



For a complete description of POSIX.1 stub implementations, refer to the Sourceware website (www.sourceware.org).

The following POSIX.1 functions are stubs:

- `close()`
- `execve()`
- `fcntl()`
- `fork()`
- `fstat()`
- `getpid()`

- `gettimeofday()`
- `ioctl()`
- `isatty()`
- `kill()`
- `link()`
- `lseek()`
- `open()`
- `_rename()`
- `sbrk()`
- `settimeofday()`
- `stat()`
- `times()`
- `unlink()`
- `wait()`
- `usleep()`
- `environ()`
- `read()`

The `write()` Function

The `write()` function is implemented as a write to `stdout`. `write()` is called by the minimal character-mode API. You do not need to call `write()` directly.

The `_exit()` Function

The `_exit()` routine terminates the thread. A child function (called by a task function) can call `exit()` to terminate the thread. You do not need to call `_exit()` directly.

Software Tasks

The contents of a task are normally quite simple. You can code a task in C or assembly language. The underlying task mechanics are the same.

Because tasks are dispatched by hardware, and not called by software, the Nios II DPX toolchain treats a task as a program entry point, like `main()`. The runtime environment does not support any callee-saved registers in a task.



For more information about software tasks, refer to “[Developing Software Tasks for the Datapath Processor](#)” on page 5–10.

Writing a Task

C Syntax

In C, use the `task` attribute to indicate to the Nios II DPX toolchain that the function defines a task. The syntax is as follows:

```
void __attribute__ (( task ( <task id> ) ) <task function name> ( ) { }
```

This attribute causes the function to terminate with an `exit` instruction, rather than returning.

Assembly Language Syntax

In assembly language, you declare a task as a global entry point whose name is of the form `__task_<n>`, where `<n>` is the task ID.

You must insert the `exit` instruction at the end of each task.

Sending a Message

This section describes how to send a message using the LWHAL.



For detailed information about sending messages, refer to [Chapter 5, Software Programming Model](#).

C Syntax

C code can send a message with either of the following macros:

```
NIOS2DPX_SND ( destID, taskID, options, length )
```

```
NIOS2DPX_SNDI ( destID, taskID, options, length )
```

For information about these macros, see [“LWHAL Extended Instruction Macros” on page 8–20](#).

Assembly Language Syntax

In assembly language, to send a PE message, you move any required message data into the transmit message registers, and execute the `snd` or `sndi` instruction.

Minimal Character-Mode API

The LWHAL supports a minimal character-mode API. It provides very simple features and a small code footprint. This API includes the following functions:

- `alt_printf()`
- `alt_putchar()`
- `alt_putstr()`

The LWHAL maps the standard newlib functions to these functions. Include `alt_stdio.h` in your source file to use the newlib function names. See [“Lightweight HAL Function Reference” on page 8–17](#) for details.

Standard I/O BSP Settings

The stdout BSP settings determine the existence, base address, and device class of the stdout device. **system.h** defines one of the following macros depending on which device is designated as stdout:

- `ALT_STDOUT_IS_JTAG_UART`—JTAG UART
- `ALT_STDOUT_IS_UART`—UART

Managing Memory Usage with the LWHAL

The LWHAL supports a custom memory-management API with a small code footprint. This API includes the following functions:

- `alt_malloc()`
- `alt_free()`
- `alt_calloc()`

The LWHAL maps the standard newlib functions to these functions. Include **alt_malloc.h** in your source file to use the newlib function names. See “[Lightweight HAL Function Reference](#)” on page 8-17 for details.

All the threads share one heap. The linker symbol `__alt_heap_start` defines the start of heap. The heap grows upwards in memory.



The LWHAL memory management functions are not thread-safe. To use them safely, your system must include mutex hardware, such as the Altera Avalon Mutex. Software must use the mutex to regulate access to the memory management functions.

For more information about using the mutex core, see “[Mutex Driver API](#)” on page 8-23.

Custom Device Drivers for the LWHAL

In an event-driven programming environment, device drivers need to be extremely lightweight. The LWHAL provides very limited services for device drivers. If your driver requires initialization, such as memory allocation, you must manually insert the required function calls into `main()`.

No device handles are provided in the LWHAL, except for a very limited stdout. For maximum flexibility, Altera recommends that each driver function call accept a pointer to the device base address as an argument. Review Altera-provided device driver code for examples.

To enable the SBT to discover a custom device driver when it creates a BSP, you describe the driver with a Tcl file.



The process for integrating a custom device driver for a Nios II DPX BSP is identical to the process for integrating a custom device driver for a Nios II BSP. For a detailed description of this process, refer to the [Developing Device Drivers for the Hardware Abstraction Layer](#) chapter of the *Nios II Software Developer's Handbook*.

Exception Handling

Each exception, including unimplemented instructions and the trap instruction, causes the Nios II DPX MTP to transfer execution to an exception address. The address of the exception handler is specified in the hardware design at system generation time and the software cannot modify it. All threads share one exception handler. The exception code is linked in the `.exceptions` section.

The default exception handler triggers a software break. You can exclude the default exception code if you do not expect the exceptions to happen, or you want to use your own exception handler. The BSP setting `lwhal.exclude_default_exception` excludes the default exception vector. You can hook in your own exception vector by enabling `lwhal.exclude_default_exception` and linking your exception code in the `.exceptions` section.

Implementing an Exception Handler

You can override the default exception handler, and provide a custom exception handler, with the `lwhal.exclude_default_exception` BSP setting.

Ensure that you allocate enough instruction memory for your custom exception handler. Because the default exception handler is minimal, the SBT allocates very little space for the `.exceptions` section.

One simple approach to allocating exception handler space is to locate the body of the handler in the `.text` section. In this configuration, you would place a jump to the exception handler in the `.exceptions` section.

Alternatively, you can enlarge the `.exceptions` section by specifying custom exception and break addresses when you generate the Nios II DPX hardware. To change the size of the exception vector, perform the following steps:

1. Determine *<handler size>*, the number of bytes required by your exception handler.
2. Open the Nios II DPX datapath processor in Qsys, and open the Multithreaded Processor tab. By default, the reset, break and exception offsets are configured to leave 4 bytes for the exception handler.
3. Turn on **Custom reset/exception/debug vector offsets**.
4. Subtract (*<handler size>* – 4) from **Custom exception address offset**, **Custom break address offset**, and **Custom reset address offset**.



You must adjust the exception, break and reset addresses together, to ensure that the break and reset vectors have the correct sizes and positions for the LWHAL.



For information about changing the Nios II DPX datapath processor's parameters, refer to "Instantiating for a Qsys System" in the *Instantiating the Nios II DPX Datapath Processor* chapter, in the *Nios II DPX Hardware Reference* section of the *Nios II DPX Datapath Processor Handbook*. For information about using BSP settings such as `lwhal.exclude_default_exception`, refer to [Chapter 8, Understanding the Nios II DPX Board Support Package](#).

Break Handler

When a software or hardware break is asserted, the Nios II DPX MTP transfers control to the break handler. The address of the break handler is specified in the hardware design at system generation time. The `.break` section provides space for break handler code. The debug unit overwrites the break handler code whenever a break occurs. You do not need to provide a break handler.

Nios II DPX BSP Creation

This section describes how an LWHAL BSP is created.

LWHAL BSP Files and Folders

The Nios II SBT creates the LWHAL BSP directory in the location you specify. [Figure 8-1](#) shows a BSP directory after the SBT creates a BSP and generates BSP files. The SBT places generated files in the top-level BSP directory, and copied files in the **LWHAL** and **drivers** directories.

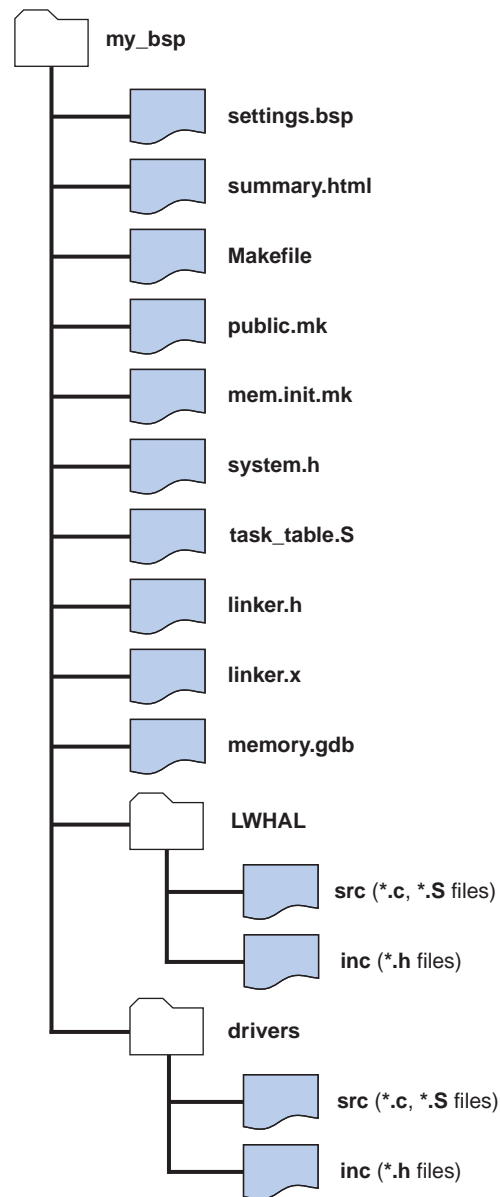
Figure 8-1. LWHAL BSP After Generating Files

Table 8-1 details all the generated BSP files shown in Figure 8-1.

Table 8-1. Generated BSP Files

File Name	Function
settings.bsp	<p>Contains the following information:</p> <ul style="list-style-type: none"> ■ BSP settings ■ Path to .sopcinfo file defining hardware ■ Linker memory regions ■ Linker section mappings <p>This file is coded in XML.</p> <p>On the command line, settings.bsp is created by the nios2-bsp-create-settings command, and optionally updated by the nios2-bsp-update-settings command. The nios2-bsp-query-settings command is available to parse information from the settings file for your scripts. The settings.bsp file is an input to nios2-bsp-generate-files.</p> <p>The Nios II SBT for Eclipse provides equivalent functionality.</p>
summary.html	Provides summary documentation of the BSP. You can view summary.html with a hypertext viewer or browser, such as Internet Explorer or Firefox . If you change the settings.bsp file, the SBT updates the summary.html file the next time you regenerate the BSP.
Makefile	Used to build the BSP. The targets you use most often are all and clean . The all target (the default) builds the liblwhal_bsp.a library file. The clean target removes all files created by a make of the all target.
public.mk	A makefile fragment that provides public information about the BSP. The file is designed to be included in other makefiles that use the BSP, such as application makefiles. The BSP Makefile also includes public.mk .
mem_init.mk	A makefile fragment that defines targets and rules to convert an application executable file to memory initialization files (.dat , .hex , and .flash) for HDL simulation, flash programming, and initializable FPGA memories. The mem_init.mk file is designed to be included by an application makefile. For usage, refer to any application makefile generated when you run the SBT.
task_table.S	An assembly language file that maps task IDs to software task functions. Unassigned task IDs are mapped to the exception vector.
system.h	Contains the C declarations describing the configuration of the Nios II DPX MTP core(s), the BSP memory map, and other system information needed by software applications.
linker.h	Contains information about the linker memory layout. system.h includes the linker.h file.
linker.x	Contains a linker script for the GNU linker.
memory.gdb	Contains memory region declarations for the GNU debugger.
obj Directory	Contains the object code files for all source files in the BSP. The hierarchy of the BSP source files is preserved in the obj directory.
liblwhal_bsp.a Library	<p>Contains the LWHAL BSP library. All object files are combined in the library file.</p> <p>The LWHAL BSP library file is always named liblwhal_bsp.a.</p>

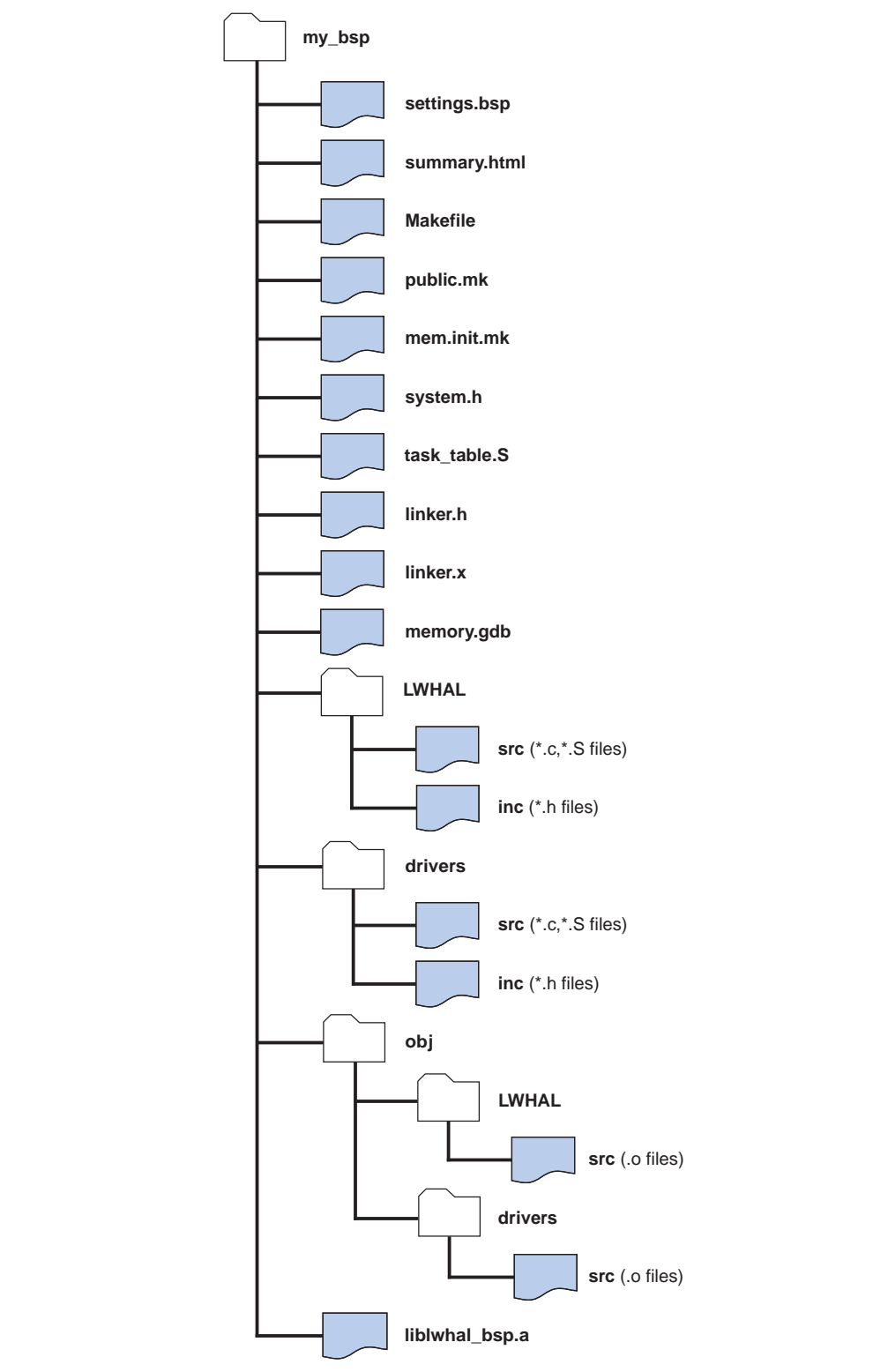
Table 8–2 details all the copied BSP files shown in Figure 8–1.

Table 8–2. Copied BSP Files

File Name	Function
LWHAL Directory	Contains LWHAL 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 LWHAL C run-time startup code, resides in the LWHAL/src directory. The nios2dpx_mtp.h and nios2dpx.h header files, containing macro definitions for access to special Nios II DPX MTP instructions and registers, resides in the LWHAL/inc 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 LWHAL directory.

Figure 8-2 shows a BSP directory after executing **make**.

Figure 8-2. LWHAL BSP After Build



Linker Map Validation

The LWHAL linker map is very simple. The SBT automatically selects reasonable memory regions and section mappings, based on the Nios II DPX configuration. After the BSP is created, you can edit the regions and section mappings, either from the command line, or in the BSP editor.

When a BSP is generated, the SBT validates the linker region and section mappings, to ensure that they are valid for a LWHAL project. If there are problems with the Nios II DPX configuration, for example if the SBT is unable to allocate a `.stack` section big enough for a stack for each thread, BSP generation fails.

See “[Specifying the Default Memory Map](#)” on page 8-16 for more information about how the memory map is created.



Linker map validation presents special issues for stand-alone systems. See “[Creating a BSP for a Stand-Alone System](#)” on page 8-25 for information.

Specifying BSP Defaults for the Nios II DPX MTP

The Nios II SBT sets BSP defaults using a set of Tcl scripts. [Table 8-3](#) lists the components of the LWHAL BSP default Tcl scripts included in the Nios II SBT. These scripts specify default BSP settings. The scripts are located in the `<Nios II EDS install path>/sdk2/bin` directory.

Table 8-3. Default Tcl Script Components

Script	Level	Summary
<code>lwhal-set-defaults.tcl</code>	Top-level	Sets system-dependent settings to default values.
<code>lwhal-call-proc.tcl</code>	Top-level	Calls a specified procedure in one of the helper scripts.
<code>lwhal-stdio-utils.tcl</code>	Helper	Specifies <code>stdio</code> device settings.
<code>lwhal-linker-utils.tcl</code>	Helper	Specifies memory regions and section mappings for linker script.



For more information about Tcl scripting with the SBT, refer to “Tcl Scripts for BSP Settings” in the [Nios II Software Build Tools](#) chapter of the *Nios II Software Developer's Handbook*.

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

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

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

The default Tcl script makes the following choices for you based on your Nios II DPX system:

- stdio character device
- Default linker memory regions
- Default linker sections mapping

The default Tcl scripts use slave descriptors to assign devices.

Top Level Tcl Script for BSP Defaults

The top level Tcl script for setting BSP defaults is **lwhal-set-defaults.tcl**. This script specifies BSP system-dependent settings, which depend on the Nios II DPX 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 **lwhal-set-defaults.tcl** explicitly. Both the Nios II SBT for Eclipse and the **nios2-bsp** script call the default Tcl script by invoking either **nios2-bsp-create-settings** or **nios2-bsp-update-settings** with the `--script lwhal-set-defaults.tcl` option.

The default Tcl script consists of a top-level Tcl script named **lwhal-set-defaults.tcl** plus the helper Tcl scripts listed in [Table 8-3](#). The helper Tcl scripts do the real work of examining the **.sopcinfo** file and choosing appropriate defaults.

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

- stdio character device (**lwhal-stdio-utils.tcl**)
- Default linker memory regions (**lwhal-linker-utils.tcl**)
- Default linker sections mapping (**lwhal-linker-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:

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

Table 8-4 lists default Tcl script arguments in detail. All arguments are optional. If present, each argument must be in the form of a name and argument value, separated by white space. All argument values are strings. For any argument not specified, the corresponding helper script chooses a suitable default value. In every case, if the argument value is `DONT_CHANGE`, the default Tcl scripts leave the setting unchanged. The `DONT_CHANGE` value allows fine-grained control of what settings the default Tcl script changes and is useful when updating an existing BSP.

Table 8-4. Default Tcl Script Command-Line Options

Argument Name	Argument Value
<code>default_stdio</code>	Slave descriptor of default <code>stdout</code> device. Set to <code>none</code> if no <code>stdout</code> device desired.
<code>default_memory_regions</code>	Controls generation of memory regions. By default, lwhal-linker-utils.tcl removes and regenerates all current memory regions. Use the <code>DONT_CHANGE</code> keyword to suppress this behavior.
<code>default_sections_mapping</code>	Slave descriptor of the memory device to which the default sections are mapped. This argument has no effect if <code>default_memory_regions == DONT_CHANGE</code> .

Specifying the Default stdio Device

The **lwhal-stdio-utils.tcl** script provides procedures to choose a default `stdio` slave descriptor and to set the `lwhal.stdout` BSP setting to that value.



For more information about these settings, refer to [Chapter 10, SBT Reference for the Nios II DPX MTP](#).

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

Specifying the Default Memory Map

The **lwhal-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 **lwhal-linker-utils.tcl** script uses the `add_memory_region` and `add_section_mapping` BSP Tcl commands.



For more information about these commands, refer to [Chapter 10, SBT Reference for the Nios II DPX MTP](#).

The script chooses the largest volatile memory region as the default memory region. If there is no volatile memory region, **lwhal-linker-utils.tcl** chooses the largest non-volatile memory region. The script assigns the .text, .rodata, .rwdata, .bss, .heap, and .stack section mappings to this default memory region.

The LWHAL linker map ensures that **crt0.o** is the first file in the .text section. .text maps to a memory region that starts at the reset address and extends to the end of that memory. Typically the reset address is the first address in the memory.

The linker defines the symbols `__bss_start` and `__bss_end`. These are pointers to the beginning and the end of the .bss region.

Using Individual Default Tcl Procedures

The default Tcl script consists of the top-level **lwhal-call-proc.tcl** script plus the helper scripts listed in [Table 8-3 on page 8-14](#). 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:

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

lwhal-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 **lwhal-call-proc.tcl** script includes the same files as the **lwhal-set-defaults.tcl** script, so any function in those included files is available.

Hardware Requirements

The target hardware system must meet the following requirements:

- It must contain one or more Nios II DPX datapath processors.
- The program memory must be large enough to accommodate the specified break and exception offsets, if any.
- There must be enough memory space for a .stack section of at least $\langle N \rangle \times \langle S \rangle$ bytes, where $\langle N \rangle$ is the number of threads in the Nios II DPX datapath processor, and $\langle S \rangle$ is the per-thread stack size specified with the `lwhal.thread_stack_size` BSP setting.

If your hardware system does not meet these requirements, the SBT cannot generate a valid LWHAL BSP.

Lightweight HAL Function Reference

This section provides a list of all the functions in the LWHAL API and standard drivers. Each function is listed with its C prototype and a short description. Each listing provides information about whether the function is thread-safe.



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.

The LWHAL API is different from newlib. See [“Differences from newlib” on page 8-4](#).

LWHAL Function Macros

The LWHAL function macros provide useful, lightweight, newlib-like functionality. This section describes the function macros.

calloc()

Prototype: `void *calloc(size_t nelem, size_t elsize);`
 Commonly called by: C programs
 Include: **alt_malloc.h**
 Description: `#define calloc alt_calloc`

free()

Prototype: `void free (void *ptr);`
 Commonly called by: C programs
 Include: **alt_malloc.h**
 Description: `#define free alt_free`

malloc()

Prototype: `void *malloc (unsigned int size);`
 Commonly called by: C programs
 Include: **alt_malloc.h**
 Description: `#define malloc alt_malloc`

printf()

Prototype: `void printf(const char* fmt, ...)`
 Commonly called by: C programs
 Include: **alt_stdio.h**
 Description: `#define printf alt_printf`

putchar()

Prototype: `void putchar(int c)`
 Commonly called by: C programs
 Include: **alt_stdio.h**
 Description: `#define putchar alt_putchar`

puts()

Prototype: `void puts(const char* str)`
Commonly called by: C programs
Include: **alt_stdio.h**
Description: `#define puts` `alt_puts`

LWHAL Functions

alt_calloc()

Prototype: `void *alt_calloc(size_t nelem, size_t elsize);`
Commonly called by: C programs
Thread-safe: No.
Include: **alt_malloc.h**
Description: This function is similar to newlib `calloc()`.
Return: The `alt_calloc()` function returns a pointer to the allocated memory. If `alt_calloc()` fails, it returns `null`.

alt_free()

Prototype: `void alt_free (void *ptr);`
Commonly called by: C programs
Thread-safe: No.
Include: **alt_malloc.h**
Description: This function is similar to newlib `free()`. `alt_free()` puts the memory block back in the free list. If the adjacent blocks are free, it coalesces the freed memory with the adjacent blocks.
Return: None

alt_malloc()

Prototype: `void *alt_malloc (unsigned int size);`
Commonly called by: C programs
Thread-safe: No.
Include: **alt_malloc.h**
Description: This function is similar to newlib `malloc()`.
`alt_malloc()` uses a first-fit algorithm. The LWHAL maintains a list of free blocks. `alt_malloc()` searches the list for a block to fit the requested memory size. When a block is found that is bigger than the requested memory, it splits the block into two parts. One is returned to the caller, and the other is put back into the free list.
Return: The `alt_malloc()` function returns a pointer to the allocated memory. If `alt_malloc()` fails, it returns `null`.

alt_printf()

Prototype:	<code>void alt_printf(const char* fmt, ...)</code>
Commonly called by:	C programs
Thread-safe:	No.
Include:	alt_stdio.h
Description:	This function is similar to newlib <code>printf()</code> . It supports the %c, %s, %x, %d, %u, %X, and %% substitution strings.
Return:	None

alt_putchar()

Prototype:	<code>void alt_putchar(int c)</code>
Commonly called by:	C programs
Thread-safe:	No.
Include:	alt_stdio.h
Description:	Similar to <code>putchar()</code>
Return:	None

alt_putstr()

Prototype:	<code>void alt_putstr(const char* str)</code>
Commonly called by:	C programs
Thread-safe:	No.
Include:	alt_stdio.h
Description:	Similar to <code>puts()</code>
Return:	None

LWHAL Extended Instruction Macros

C code can execute Nios II DPX MTP extended instructions by using the macros listed in this section.

For descriptions of the extended instructions, refer to “[Nios II DPX Extended Instruction Set Reference](#)” on page 9–104.

NIOS2DPX_CIDALLOC()

Prototype:	<code>NIOS2DPX_CIDALLOC (dest)</code>
Arguments:	<code>dest</code> —Set to the status returned by the instruction
Include:	nios2dpx.h

NIOS2DPX_RXFREE()

Prototype: NIOS2DPX_RXFREE ()
Arguments: None
Include: **nios2dpx.h**

NIOS2DPX_SND()

Prototype: NIOS2DPX_SND (destID, taskID, options, length)
 ■ destID—Unique identifier of destination PE
 ■ taskID—Unique identifier of destination task
Arguments: ■ options—Message control options
 ■ length—Number of message arguments
Include: **nios2dpx.h**

NIOS2DPX_SNDI()

Prototype: NIOS2DPX_SNDI (destID, taskID, options, length)
 ■ destID—Unique identifier of destination PE
 ■ taskID—Unique identifier of destination task
Arguments: ■ options—Message control options. For limitations on this argument, refer to the instruction
 definition in [“Nios II DPX Extended Instruction Set Reference” on page 9–104](#).
 ■ length—Number of message arguments
Include: **nios2dpx.h**

NIOS2DPX_TXALLOC()

Prototype: NIOS2DPX_TXALLOC (dest)
Arguments: dest—Set to the status returned by the instruction
Include: **nios2dpx.h**

LWHAL Driver Functions

The SBT includes the LWHAL driver for each supported device that it discovers at the time of BSP generation. Drivers can be disabled or overridden. This section lists the function calls supported by each driver.

JTAG UART Driver API

The LWHAL JTAG UART driver has one function call.

altera_avalon_jtag_uart_lwhal_putchar()

Prototype: `void altera_avalon_jtag_uart_lwhal_putchar(void* base, int character)`
Commonly called by: `alt_putchar()`
Thread-safe: No.
Include:
Description: Writes a single character to the JTAG UART
Function arguments:

- `base`—JTAG UART base address
- `character`—Character to be written

Return: None

UART Driver API

The LWHAL UART driver has one function call.

altera_avalon_uart_lwhal_putchar()

Prototype: `void altera_avalon_uart_lwhal_putchar(void* base, int character)`
Commonly called by: `alt_putchar()`
Thread-safe: No.
Include:
Description: Writes a single character to the UART
Function arguments:

- `base`—UART base address
- `character`—Character to be written

Return: None

Mutex Driver API

This section lists the function calls supported by the LWHAL mutex driver.

altera_avalon_mutex_lwhal_is_mine()

Prototype: `int altera_avalon_mutex_lwhal_is_mine(void* base, alt_u16 owner)`

Commonly called by: C programs

Thread-safe: Yes.

Include: **altera_avalon_mutex_lwhal.h**

Description: Determines if this owner owns the mutex.

Function arguments:

- `base`—Mutex base address
- `owner`—Unique identifier value for mutex owner

Return: The `altera_avalon_mutex_lwhal_is_mine()` function returns non-zero if the mutex is owned by this owner.

altera_avalon_mutex_lwhal_lock()

Prototype: `int altera_avalon_mutex_lwhal_lock(void* base, alt_u16 owner, alt_u16 value)`

Commonly called by: C programs

Thread-safe: Yes.

Include: **altera_avalon_mutex_lwhal.h**

Description: Locks the mutex. Does not return until it has successfully claimed the mutex (blocking).

Function arguments:

- `base`—Mutex base address
- `owner`—Unique identifier value for mutex owner
- `value`—The new value to write to the mutex

Return: The `altera_avalon_mutex_lwhal_lock()` function returns zero if the mutex is successfully locked, or non-zero otherwise.

altera_avalon_mutex_lwhal_trylock()

Prototype: `int altera_avalon_mutex_lwhal_trylock(void* base, alt_u16 owner, alt_u16 value)`
Commonly called by: C programs
Thread-safe: Yes.
Include: **altera_avalon_mutex_lwhal.h**
Description: Tries once to lock the mutex. Return immediately if it fails to lock the mutex (non-blocking).
Function arguments:

- **base**—Mutex base address
- **owner**—Unique identifier value for mutex owner
- **value**—The new value to write to the mutex

Return: The `altera_avalon_mutex_lwhal_trylock()` function returns zero if the mutex is successfully locked, or non-zero otherwise.

altera_avalon_mutex_lwhal_unlock()

Prototype: `void altera_avalon_mutex_lwhal_unlock(void* base, alt_u16 owner)`
Commonly called by: C programs
Thread-safe: Yes.
Include: **altera_avalon_mutex_lwhal.h**
Description: Releases mutex. Upon release, the value stored in the mutex is set to zero. If the caller does not hold the mutex, the behavior of this function is undefined.
Function arguments:

- **base**—Mutex base address
- **owner**—Unique identifier value for mutex owner

Return: None.

Lightweight HAL Standard Types

In the interest of portability, the LWHAL uses a set of standard type definitions in place of the ANSI C built-in types. Table 8-5 describes these types, which are defined in the header file **alt_types.h**.

Table 8-5. Standard Types

Type	Description
<code>alt_8</code>	Signed 8-bit integer.
<code>alt_u8</code>	Unsigned 8-bit integer.
<code>alt_16</code>	Signed 16-bit integer.
<code>alt_u16</code>	Unsigned 16-bit integer.
<code>alt_32</code>	Signed 32-bit integer.
<code>alt_u32</code>	Unsigned 32-bit integer.
<code>alt_64</code>	Signed 64-bit integer.
<code>alt_u64</code>	Unsigned 64-bit integer.

Creating a BSP for a Stand-Alone System

Stand-alone user flow support allows a user-defined memory device to be added to the memory map. A user-defined memory device is a device that is defined outside the Qsys or SOPC Builder hardware system. Because its parameters are not available in the `.sopcinfo` file, you must provide them manually. Defining a user-defined memory device allows linker memory regions and section mappings to be created with the BSP tools.

Creating a BSP from the Command Line

From the command line, you specify a user-defined memory device with the `add_memory_device` Tcl command. The command syntax is as follows:

```
add_memory_device <device name> <base address> <size>
```

You can use the following syntax to create a BSP for a stand-alone system on the command line:

```
nios2-bsp lwhal <bsp-dir> [<sopc>]  
--cmd=add_memory_device <device name> <base address> <span>  
--cmd=add_memory_region <name> <slave_desc> <offset> <span>  
--script=$SOPC_KIT_NIOS2/sdk2/bin/lwhal-call-proc.tcl set_default_sections_mapping
```

The last command in this command line reapplies the default section mappings by calling the `lwhal-call-proc.tcl` script. This command is executed last. It causes the SBT to recreate the memory map using the newly added memory device.



For detailed information about the `add_memory_device` Tcl command, refer [Chapter 10, SBT Reference for the Nios II DPX MTP](#).

Creating a BSP with the BSP Editor

You can start the BSP editor several ways. For the purpose of creating a new BSP, start the BSP editor one of the following ways:

- From the Nios II SBT for Eclipse. On the Nios II menu, click **Nios II BSP Editor**. The editor starts without loading a `.bsp` file.
- From the Nios II Command Shell. Type:

```
nios2-bsp
```

In the BSP Editor, the **Add Memory Device** button allows you to specify a user-defined device. In the **Main** tab, you can specify the following memory characteristics:

- Component name.
- Base address.
- Memory size.

In the **Advanced** tab, you can control the following memory characteristics:

- The memory initialization filename for SOPC Builder systems. This parameter is not used with Qsys systems.
- The physical memory width.
- The device's path and name in the Qsys hierarchy. This parameter is not used with SOPC Builder systems.
- The memory initialization file parameter name. Every memory device can have an HDL parameter specifying the name of the initialization file. The Nios II DPX ModelSim launch configuration overrides the HDL parameter to specify the memory initialization filename. This parameter is not used with SOPC Builder systems.
- 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.

In the **Advanced** tab, you can also enable and disable generation of the following memory initialization file types:

- **.hex** file.
- **.dat** and **.sym** files.
- **.flash** file.

Parameters in the **Advanced** tab are available only through the BSP Editor.

Once added, the device appears in the **Memory Map** dialog box and **Memory Device Usage Table**, and you can define linker regions and mappings as usual.

To use the BSP Editor to create a BSP for a stand-alone system, perform the following steps:

1. Create a new BSP. Generation error messages appear. You can safely ignore them for now.
2. In the **Linker Script** tab, add a user-defined memory device with the **Add Memory Device** button.



If you need to generate memory initialization files, specify them in this step, as described in [“Creating Memory Initialization Files” on page 6-15](#).

3. Define a memory region in the user-defined memory device, using the **Add** button under **Linker Regions**.
4. Click **Restore Defaults** to correctly define memory sections for the new memory configuration.


Alternatively, define the sections by hand.

5. Generate the BSP.



Any previous generation errors are resolved.

If you anticipate needing to repeat these steps more than once for the same stand-alone system, export a Tcl script to automate the steps, as described in [“Exporting a Tcl Script” on page 6-33](#).

-  For more information about creating and working with user-defined memory devices, refer to [“Using the BSP Editor” on page 6-28](#).

The Nios II DPX MTP Instruction Set

This section introduces the Nios II DPX MTP instruction format and provides a detailed reference of the MTP instruction set.

Instruction Formats

The Nios II DPX MTP implements the following instruction format types:

- IX type
- I-16 type
- I-12 type
- I-5 type
- R-3 type
- BMX type

IX (Immediate Extended) Type Instruction Format

The defining characteristic of the IX type instruction word format is that it contains a 20-bit immediate value embedded within the instruction word.

The IX type instruction format is:

Table 9–1. IX (Immediate eXtended) Instruction Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OP				A						OPIX		IMM20																			

IX type instruction words contain:

- A 4-bit opcode field OP
- A 2-bit extended opcode field OPIX
- A 6-bit register field A
- A 20-bit immediate data field IMM20

IX type instructions include `call`, `jmp`, and `movhi20`.

The `jmp` instruction is a variant on the IX type instruction format, using the A field for additional address bits.

Table 9–2. IX (Immediate eXtended) Instruction Format—`jmp` Instruction

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OP				IMM26[25:20]								1	IMM26[19:0]																		

I-16 (16-Bit Immediate) Type Instruction Format

The defining characteristic of the I-16 type instruction format is that it contains a 16-bit immediate value embedded within the instruction word.

The I-16 type instruction format is:

Table 9-3. I-16 (Immediate 16) Instruction Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OP				A						B						IMM16															

I-16 type instruction words contain:

- A 4-bit opcode field OP
- Two 6-bit register fields A and B
- A 16-bit immediate data field IMM16

In most cases, fields A and IMM16 specify the source operands, and field B specifies the destination register. IMM16 is considered signed except for logical operations and unsigned comparisons.

I-16 type instructions include arithmetic operations such as `addi`, `ori`, and `xori`.

I-12 (12-Bit Immediate) Type Instruction Format

The defining characteristic of the I-12 type instruction word format is that it contains a 12-bit immediate value embedded within the instruction word.

I-12 type instructions include arithmetic operations such as `addi` and `muli`, branch operations, and load and store operations.

The I-12 type instruction format is:

Table 9-4. I-12 (Immediate 12) Instruction Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OP				A						B						OPI-12				IMM12											

I-12 type instruction words contain:

- A 4-bit opcode field OP
- A 4-bit extended opcode field OPI-12
- Two 6-bit register fields A and B
- A 12-bit immediate data field IMM12

I-5 (5-Bit Immediate) Type Instruction Format

The defining characteristic of the I-5 type instruction word format is that it contains a 5-bit immediate value embedded within the instruction word.

The I-5 type instruction format is:

Table 9-5. I-5 (Immediate 5) Instruction Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OP				A						B						OPI-12				0			OPI-5			IMM5					

I-5 type instruction words contain:

- A 4-bit opcode field OP
- A 4-bit extended opcode field OPI-12
- A 3-bit extended opcode field OPI-5
- Two 6-bit register fields A and B
- A 5-bit immediate data field IMM5

I-5 type instructions include instructions such as `rol` and `slli`.

R-3 (Three Register) Type Instruction Format

The defining characteristic of the R-3 type instruction word format is that all arguments and results are specified as registers.

The R-3 type instruction format is:

Table 9-6. R-3 (3 Register) Instruction Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OP				A						B						C				0	0		OPR-3								

R-3 type instructions contain:

- A 4-bit opcode field OP
- A 6-bit extended opcode field OPR-3
- Three 6-bit register fields A, B, and C

R-3 type instructions include arithmetic and logical operations such as `add` and `nor`, comparison operations such as `cmpeq` and `cmplt`, and other operations that need only register operands.

BMX Type Instruction Format

The BMX (Bit Manipulation eXtension) instruction type is a variant on the I-12 type, using the IMM12 field to specify a range of bits within a data word.

BMX instructions perform operations at the bit level. All the BMX instructions use two 5-bit immediate fields in the instruction to encode all possible contiguous range of bits in a 32-bit value. The LSB immediate value encodes the position of the least-significant bit of the contiguous range of bits and the MSB immediate value encodes the position of the most-significant bit of the contiguous range.

The BMX instruction format is:

Table 9-7. BMX I-12 Instruction Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
OP				A						B						OPI-12				0	MSB						LSB					

BMX type instructions contain:

- A 4-bit opcode field OP
- A 4-bit extended opcode field OPI-12
- Two 6-bit register fields A and B
- An MSB field
- An LSB field

Instruction Encodings

Table 9–8 on page 9–5 through Table 9–12 on page 9–9 show the instruction encodings for the Nios II DPX MTP.

Table 9–8. OP Opcode Field Codes

OP	Instruction
0x0	See Table 9–9, “OPIX Extended Opcode Field Codes, for IX Type Instructions”
0x1	See Table 9–10, “OPI-12 Extended Opcode Field Codes for I-12 and BMX Type Instructions”
0x2	
0x3	See Table 9–11, “OPR-3 Extended Opcode Field Codes for R-3 Type Instructions”
0x4	sndi
0x5	
0x6	
0x7	
0x8	andci
0x9	andi
0xA	ori
0xB	xori
0xC	andchi
0xD	andhi
0xE	orhi
0xF	xorhi

Table 9–9. OPIX Extended Opcode Field Codes, for IX Type Instructions

OPIX (1)	Instruction
0x0	call
0x1	jmp
0x2	movhi20

Note to Table 9–9:

(1) OP = 0

Table 9–10. OPI-12 Extended Opcode Field Codes for I-12 and BMX Type Instructions

OP	OPI-12	Instruction
0x1	0x0	See Table 9–12, “OPI-5 Extended Opcode Field Codes for I-5 Type Instructions”
	0x1	cmpgei
	0x2	cmplti
	0x3	cmpnei
	0x4	cmpeqi
	0x5	cmpgeui
	0x6	cmpltui
	0x7	
	0x8	
	0x9	bge
	0xA	blt
	0xB	bne
	0xC	beq
	0xD	bgeu
	0xE	bltu
	0xF	
0x2	0x0	ldb
	0x1	ldh
	0x2	ldw
	0x3	addi
	0x4	ldbu
	0x5	ldhu
	0x6	
	0x7	
	0x8	stb
	0x9	sth
	0xA	stw
	0xB	muli
	0xC	insert (1)
	0xD	
	0xE	extract (1)
	0xF	merge (1)

Note to Table 9–10:

(1) BMX

Table 9–11. OPR-3 Extended Opcode Field Codes for R-3 Type Instructions (Part 1 of 2)

OPR-3 (1)	Instruction
0x00	
0x01	eret
0x02	
0x03	
0x04	rol
0x05	ret
0x06	nor
0x07	mulxuu
0x08	cmpge
0x09	bret
0x0A	exit
0x0B	
0x0C	ror
0x0D	jmp
0x0E	and
0x0F	
0x10	cmplt
0x11	
0x12	cidalloc
0x13	
0x14	sll
0x15	nextpc
0x16	or
0x17	mulxsu
0x18	cmpne
0x19	
0x1A	
0x1B	
0x1C	srl
0x1D	callr
0x1E	xor
0x1F	mulxss
0x20	cmpeq
0x21	
0x22	
0x23	
0x24	

Note to Table 9–11:

(1) OP = 3

Table 9–11. OPR-3 Extended Opcode Field Codes for R-3 Type Instructions (Part 2 of 2)

OPR-3 (1)	Instruction
0x25	
0x26	rdctl
0x27	mul
0x28	cmpgeu
0x29	
0x2A	
0x2B	
0x2C	
0x2D	
0x2E	wrctl
0x2F	
0x30	cmpltu
0x31	add
0x32	txalloc
0x33	
0x34	
0x35	jrel
0x36	
0x37	
0x38	
0x39	sub
0x3A	rxfree
0x3B	
0x3C	sra
0x3D	
0x3E	
0x3F	

Note to Table 9–11:

(1) OP = 3

Table 9-12. OPI-5 Extended Opcode Field Codes for I-5 Type Instructions

OPI-5 (1)	Instruction
0x0	rol <i>i</i>
0x1	
0x2	slli
0x3	srli
0x4	trap
0x5	break
0x6	
0x7	srai

Note to Table 9-12:

- (1) OP = 1
OPI-12 = 0

Assembler Pseudo-Instructions

Table 9-13 lists pseudo-instructions available in Nios II DPX MTP assembly language. Pseudo-instructions are used in assembly source code like regular assembly instructions. Each pseudo-instruction is implemented at the machine level using an equivalent instruction. The `movia` pseudo-instruction is the only exception, being implemented with two instructions. Most pseudo-instructions do not appear in disassembly views of machine code.

Table 9-13. Assembler Pseudo-Instructions (Part 1 of 2)

Pseudo-Instruction	Equivalent Instruction
<code>bgt rA, rB, label</code>	<code>blt rB, rA, label</code>
<code>bgtu rA, rB, label</code>	<code>bltu rB, rA, label</code>
<code>ble rA, rB, label</code>	<code>bge rB, rA, label</code>
<code>bleu rA, rB, label</code>	<code>bgeu rB, rA, label</code>
<code>br label</code>	<code>beq r0, r0, label</code>
<code>cmpgt rC, rA, rB</code>	<code>cmplt rC, rB, rA</code>
<code>cmpgti rB, rA, IMMED</code>	<code>cmpgei rB, rA, (IMMED+1)</code>
<code>cmpgtu rC, rA, rB</code>	<code>cmpltu rC, rB, rA</code>
<code>cmpgtui rB, rA, IMMED</code>	<code>cmpgeui rB, rA, (IMMED+1)</code>
<code>cmple rC, rA, rB</code>	<code>cmpge rC, rB, rA</code>
<code>cmplei rB, rA, IMMED</code>	<code>cmplti rB, rA, (IMMED+1)</code>
<code>cmpleu rC, rA, rB</code>	<code>cmpgeu rC, rB, rA</code>
<code>cmpleui rB, rA, IMMED</code>	<code>cmpltui rB, rA, (IMMED+1)</code>
<code>mov rC, rA</code>	<code>add rC, rA, r0</code>
<code>movhi rB, IMMED</code>	<code>orhi rB, r0, IMMED</code>
<code>movi rB, IMMED</code>	<code>addi, rB, r0, IMMED</code>
<code>movia rB, label</code>	<code>movhi20 rB, %hi20adj(label)</code> <code>addi rB, rB, %lo12(label)</code>

Table 9-13. Assembler Pseudo-Instructions (Part 2 of 2)

Pseudo-Instruction	Equivalent Instruction
movui rB, IMMED	ori rB, r0, IMMED
nop	add r0, r0, r0
subi rB, rA, IMMED	addi rB, rA, (-IMMED)

Assembler Macros

The Nios II DPX assembler provides macros to extract 12-bit, 16-bit, and 20-bit fields from labels and from 32-bit immediate values. Table 9-14 lists the available macros. These macros return 12-bit, 16-bit, or 20-bit signed or unsigned values depending on where they are used. For example, when used with an instruction that requires a 16-bit signed immediate value, these macros return a value ranging from -32768 to 32767. When used with an instruction that requires a 16-bit unsigned immediate value, these macros return a value ranging from 0 to 65535.

Table 9-14. Assembler Macros

Macro	Description	Operation
%lo(immed32)	Extract bits [15..0] of immed32	immed32 & 0xFFFF
%hi(immed32)	Extract bits [31..16] of immed32	(immed32 >> 16) & 0xFFFF
%hiadj(immed32)	Extract bits [31..16] and adds bit 15 of immed32	((immed32 >> 16) & 0xFFFF) + ((immed32 >> 15) & 0x1)
%gprel(immed32)	Replace the immed32 address with an offset from the global pointer	immed32 - _gp
%hi20	Extract bits [31..12] of immed32	(immed32 >> 12) & 0xFFFFF
%hi20adj	Extract bits [31..12] of immed32 and adds bit 11 of immed32	((immed32 >> 12) & 0xFFFFF) + ((immed32 >> 11) & 0x1)
%lo12	Extract bits [11..0] of immed32	immed32 & 0xFFF

Table 9-15 shows the notation conventions used to describe instruction operation.

Table 9-15. Notation Conventions (Part 1 of 2)

Notation	Meaning
$X \leftarrow Y$	X is written with Y
$PC \leftarrow X$	The program counter (PC) is written with address X; the instruction at X is the next instruction to execute
PC	The address of the assembly instruction in question
rA, rB, rC	One of the 32-bit general-purpose registers
IMM _n	An <i>n</i> -bit immediate value, embedded in the instruction word
IMMED	An immediate value
X_n	The <i>n</i> th bit of X, where <i>n</i> = 0 is the LSB
$X_{n..m}$	Consecutive bits <i>n</i> through <i>m</i> of X
0xNNMM	Hexadecimal notation
$X : Y$	Bitwise concatenation For example, (0x12 : 0x34) = 0x1234

Table 9–15. Notation Conventions (Part 2 of 2)

Notation	Meaning
$\sigma(X)$	The value of X after being sign-extended to a full register-sized signed integer
$X \gg n$	The value X after being right-shifted n bit positions
$X \ll n$	The value X after being left-shifted n bit positions
$X \& Y$	Bitwise logical AND
$X Y$	Bitwise logical OR
$X \wedge Y$	Bitwise logical XOR
$\sim X$	Bitwise logical NOT (one's complement)
Mem8[X]	The byte located in data memory at byte address X
Mem16[X]	The halfword located in data memory at byte address X
Mem32[X]	The word located in data memory at byte address X
label	An address label specified in the assembly file
(signed) rX	The value of rX treated as a signed number
(unsigned) rX	The value of rX treated as an unsigned number

Nios II DPX MTP Instruction Set Reference

This section describes general-purpose Nios II DPX MTP instructions. The following pages list the instructions in alphabetical order.

add**add****Operation:** $rC \leftarrow rA + rB$ **Assembler Syntax:** `add rC, rA, rB`**Example:** `add r6, r7, r8`**Description:** Calculates the sum of rA and rB. Stores the result in rC. Used for both signed and unsigned addition.**Usage:** **Carry Detection (unsigned operands):**

Following an add operation, a carry out of the MSB can be detected by checking whether the unsigned sum is less than one of the unsigned operands. The carry bit can be written to a register, or a conditional branch can be taken based on the carry condition. The following examples show both cases.

```
add rC, rA, rB           # The original add operation
cmpltu rD, rC, rA        # rD is written with the carry bit
```

```
add rC, rA, rB           # The original add operation
bltu rC, rA, label       # Branch if carry generated
```

Overflow Detection (signed operands):

An overflow is detected when two positives are added and the sum is negative, or when two negatives are added and the sum is positive. The overflow condition can control a conditional branch, as shown in the following example.

```
add rC, rA, rB           # The original add operation
xor rD, rC, rA           # Compare signs of sum and rA
xor rE, rC, rB           # Compare signs of sum and rB
and rD, rD, rE           # Combine comparisons
blt rD, zero, label      # Branch if overflow occurred
```

Extended Register Restrictions: rB cannot be an extended register.**Exceptions:** None**Instruction Type:** R-3

Instruction Fields: A = Register index of operand rA
 B = Register index of operand rB
 C = Register index of operand rC

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				A (src1)							B (src2)							C (dst)				0	0		0x31						

addi

add immediate

Operation: $rB \leftarrow rA + \sigma(\text{IMM12})$

Assembler Syntax: `addi rB, rA, IMM12`

Example: `addi r6, r7, -100`

Description: Sign-extends the 12-bit immediate value and adds it to the value of rA. Stores the sum in rB.

Usage: **Carry Detection (unsigned operands):**

Following an addi operation, a carry out of the MSB can be detected by checking whether the unsigned sum is less than one of the unsigned operands. The carry bit can be written to a register, or a conditional branch can be taken based on the carry condition. The following examples show both cases.

```
addi rB, rA, IMM12           # The original add operation
cmpltu rD, rB, rA            # rD is written with the carry bit
```

```
addi rB, rA, IMM12           # The original add operation
bltu rB, rA, label           # Branch if carry generated
```

Overflow Detection (signed operands):

An overflow is detected when two positives are added and the sum is negative, or when two negatives are added and the sum is positive. The overflow condition can control a conditional branch, as shown in the following example.

```
addi rB, rA, IMM12           # The original add operation
xor rC, rB, rA               # Compare signs of sum and rA
xorhi rD, rB, IMM12          # Compare signs of sum and IMM12
and rC, rC, rD               # Combine comparisons
blt rC, zero, label          # Branch if overflow occurred
```

Extended Register Restrictions: None

Exceptions: None

Instruction Type: I-12

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
IMM12 = 12-bit signed immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x2				A (src1)							B (dst)					0x3			IMM12 (src2)												

and**bitwise logical and**

Operation:	$rC \leftarrow rA \ \& \ rB$
Assembler Syntax:	and rC, rA, rB
Example:	and r6, r7, r8
Description:	Calculates the bitwise logical AND of rA and rB and stores the result in rC.
Extended Register Restrictions:	None
Exceptions:	rB cannot be an extension register.
Instruction Type:	R-3
Instruction Fields:	A = Register index of operand rA B = Register index of operand rB C = Register index of operand rC

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				A (src1)						B (src2)						C (dst)						0	0		0xE						

andci

bitwise logical and clear immediate

Operation: $rB \leftarrow rA \& (0xFFFF : IMM16)$

Assembler Syntax: `andci rB,rA,IMM16`

Example: `andci r6,r7,100`

Description: Calculates the bitwise logical AND of rA and (0xFFFF : IMM16) and stores the result in rB.

Extended Register Restrictions: None

Exceptions: None

Instruction Type: I-16

Instruction Fields:
A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit unsigned immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x8				A (src1)						B (dst)						IMM16 (src2)															

andchi **bitwise logical and clear immediate into high halfword****Operation:** $rB \leftarrow rA \& (\text{IMM16} : 0\text{xFFFF})$ **Assembler Syntax:** `andchi rB,rA,IMM16`**Example:** `andchi r6,r7,100`**Description:** Calculates the bitwise logical AND of rA and (IMM16 : 0xFFFF) and stores the result in rB.**Extended Register
Restrictions:****Exceptions:** None**Instruction Type:** I-16**Instruction Fields:**
A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit unsigned immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x8				A (src1)							B (dst)					IMM16 (src2)															

andhi bitwise logical and immediate into high halfword

Operation: $rB \leftarrow rA \& (IMM16 : 0x0000)$

Assembler Syntax: `andhi rB, rA, IMM16`

Example: `andhi r6, r7, 100`

Description: Calculates the bitwise logical AND of rA and (IMM16 : 0x0000) and stores the result in rB.

Extended Register Restrictions: None

Exceptions: None

Instruction Type: I-16

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit unsigned immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0xD				A (src1)						B (dst)						IMM16 (src2)															

andi**bitwise logical and immediate****Operation:** $rB \leftarrow rA \& (0x0000 : IMM16)$ **Assembler Syntax:** `andi rB, rA, IMM16`**Example:** `andi r6, r7, 100`**Description:** Calculates the bitwise logical AND of rA and (0x0000 : IMM16) and stores the result in rB.**Extended Register Restrictions:** None**Exceptions:** None**Instruction Type:** I-16**Instruction Fields:**
A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit unsigned immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x9				A (src1)						B (dst)						IMM16 (src2)															

beq

branch if equal

Operation: if (rA == rB)
then $PC \leftarrow PC + 4 + \sigma(\text{IMM12})$
else $PC \leftarrow PC + 4$

Assembler Syntax: beq rA, rB, label

Example: beq r6, r7, label

Description: If rA == rB, then beq transfers program control to the instruction at label. In the instruction encoding, the offset given by IMM12 is treated as a signed number of words relative to the instruction immediately following beq. Therefore, IMM12 is equivalent to a 14-bit offset in bytes.

Extended Register Restrictions: None

Exceptions:

Instruction Type: I-12

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
IMM12 = 12-bit signed immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x1				A (src1)								B (dst)				0xC				IMM12 (src2)											

bge**branch if greater than or equal signed**

Operation: if ((signed) rA >= (signed) rB)
then $PC \leftarrow PC + 4 + \sigma(\text{IMM12})$
else $PC \leftarrow PC + 4$

Assembler Syntax: bge rA, rB, label

Example: bge r6, r7, top_of_loop

Description: If (signed) rA >= (signed) rB, then bge transfers program control to the instruction at label. In the instruction encoding, the offset given by IMM12 is treated as a signed number of words relative to the instruction immediately following bge. Therefore, IMM12 is equivalent to a 14-bit offset in bytes.

Extended Register Restrictions: None

Exceptions:

Instruction Type: I-12

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
IMM12 = 12-bit signed immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x1				A (src1)						B (dst)						0x9				IMM12 (src2)											

bgeu

branch if greater than or equal unsigned

Operation: if ((unsigned) rA >= (unsigned) rB)
then PC ← PC + 4 + σ (IMM12)
else PC ← PC + 4

Assembler Syntax: bgeu rA, rB, label

Example: bgeu r6, r7, top_of_loop

Description: If (unsigned) rA >= (unsigned) rB, then bgeu transfers program control to the instruction at label. In the instruction encoding, the offset given by IMM12 is treated as a signed number of words relative to the instruction immediately following bgeu. Therefore, IMM12 is equivalent to a 14-bit offset in bytes.

Extended Register Restrictions: None

Exceptions:

Instruction Type: I-12

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
IMM12 = 12-bit signed immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x1				A (src1)						B (dst)						0xD		IMM12 (src2)													

bgt**branch if greater than signed**

Operation:	if ((signed) rA > (signed) rB) then PC ← label else PC ← PC + 4
Assembler Syntax:	bgt rA, rB, label
Example:	bgt r6, r7, top_of_loop
Description:	If (signed) rA > (signed) rB, then <code>bgt</code> transfers program control to the instruction at label.
Extended Register Restrictions:	None
Pseudo-instruction:	<code>bgt</code> is implemented with the <code>blt</code> instruction by swapping the register operands.

bgtu

branch if greater than unsigned

Operation:	if ((unsigned) rA > (unsigned) rB) then PC ← label else PC ← PC + 4
Assembler Syntax:	bgtu rA, rB, label
Example:	bgtu r6, r7, top_of_loop
Description:	If (unsigned) rA > (unsigned) rB, then bgtu transfers program control to the instruction at label.
Extended Register Restrictions:	None
Pseudo-instruction:	bgtu is implemented with the bltu instruction by swapping the register operands.

ble**branch if less than or equal signed**

Operation:	if ((signed) rA <= (signed) rB) then PC ← label else PC ← PC + 4
Assembler Syntax:	ble rA, rB, label
Example:	ble r6, r7, top_of_loop
Description:	If (signed) rA <= (signed) rB, then ble transfers program control to the instruction at label.
Extended Register Restrictions:	None
Pseudo-instruction:	ble is implemented with the bge instruction by swapping the register operands.

bleu

branch if less than or equal to unsigned

Operation:	if ((unsigned) rA <= (unsigned) rB) then PC ← label else PC ← PC + 4
Assembler Syntax:	bleu rA, rB, label
Example:	bleu r6, r7, top_of_loop
Description:	If (unsigned) rA <= (unsigned) rB, then bleu transfers program counter to the instruction at label.
Extended Register Restrictions:	None
Pseudo-instruction:	bleu is implemented with the bgeu instruction by swapping the register operands.

blt**branch if less than signed**

Operation: if ((signed) rA < (signed) rB)
 then $PC \leftarrow PC + 4 + \sigma(\text{IMM12})$
 else $PC \leftarrow PC + 4$

Assembler Syntax: `blt rA, rB, label`

Example: `blt r6, r7, top_of_loop`

Description: If (signed) rA < (signed) rB, then `blt` transfers program control to the instruction at label. In the instruction encoding, the offset given by IMM12 is treated as a signed number of words relative to the instruction immediately following `blt`. Therefore, IMM12 is equivalent to a 14-bit offset in bytes.

Extended Register Restrictions: None

Exceptions:

Instruction Type: I-12

Instruction Fields: A = Register index of operand rA
 B = Register index of operand rB
 IMM12 = 12-bit signed immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x1				A						B						0xA				IMM12											

bltu

branch if less than unsigned

Operation: if ((unsigned) rA < (unsigned) rB)
then $PC \leftarrow PC + 4 + \sigma(\text{IMM12})$
else $PC \leftarrow PC + 4$

Assembler Syntax: bltu rA, rB, label

Example: bltu r6, r7, top_of_loop

Description: If (unsigned) rA < (unsigned) rB, then bltu transfers program control to the instruction at label. In the instruction encoding, the offset given by IMM12 is treated as a signed number of words relative to the instruction immediately following bltu. Therefore, IMM12 is equivalent to a 14-bit offset in bytes.

Extended Register Restrictions: None

Exceptions:

Instruction Type: I-12

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
IMM12 = 12-bit signed immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x1				A						B						0xE				IMM12											

bne**branch if not equal**

Operation: if ($rA \neq rB$)
 then $PC \leftarrow PC + 4 + \sigma(\text{IMM12})$
 else $PC \leftarrow PC + 4$

Assembler Syntax: `bne rA, rB, label`

Example: `bne r6, r7, top_of_loop`

Description: If $rA \neq rB$, then `bne` transfers program control to the instruction at `label`. In the instruction encoding, the offset given by `IMM12` is treated as a signed number of words relative to the instruction immediately following `bne`. Therefore, `IMM12` is equivalent to a 14-bit offset in bytes.

Extended Register Restrictions: None

Exceptions:

Instruction Type: I-12

Instruction Fields: A = Register index of operand rA
 B = Register index of operand rB
 IMM12 = 12-bit signed immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x1				A				B				0xB				IMM12															

br

unconditional branch

Operation: $PC \leftarrow PC + 4 + \sigma(\text{IMM12})$

Assembler Syntax: `br label`

Example: `br top_of_loop`

Description: Transfers program control to the instruction at label. In the instruction encoding, the offset given by IMM12 is treated as a signed number of words relative to the instruction immediately following `br`. Therefore, IMM12 is equivalent to a 14-bit offset in bytes.

Extended Register Restrictions: None

Exceptions:

Pseudo-instruction: `br` is implemented as `beq zero, zero, IMM16`.

break**debugging breakpoint**

Operation:	$ba \leftarrow PC + 4$ $PC \leftarrow \text{break handler address}$
Assembler Syntax:	<code>break</code> <code>break imm5</code>
Example:	<code>break</code>
Description:	Breaks program execution and transfers control to the debugger break-processing routine. Saves the address of the next instruction in register <code>ba</code> . Transfers execution to the break handler.

The 5-bit immediate field `imm5` is ignored by the processor, but it can be used by the debugger.

`break` with no argument is the same as `break 0`.

Usage: `break` is used by debuggers exclusively. Only debuggers should place `break` in an application program, operating system, or exception handler. The address of the break handler is specified at system generation time.

Some debuggers support `break` and `break 0` instructions in source code. These debuggers treat the `break` instruction as a normal breakpoint.

Extended Register Restrictions: None

Exceptions: Break

Instruction Type: I-5

Instruction Fields: `IMM5` = Type of breakpoint

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x1				A (src1)							B (dst)					0x0				0			0x5			IMM5					

bret

breakpoint return

Operation: $PC \leftarrow ba$

Assembler Syntax: `bret`

Example: `bret`

Description: Transfers execution to the address in `ba`.

Usage: `bret` is used by debuggers exclusively and must not appear in application programs, operating systems, or exception handlers.

Extended Register Restrictions: None

Instruction Type: R-3

Instruction Fields: None

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				0x1e						0						0						0	0		0x9						

call**call subroutine**

Operation:	$ra \leftarrow PC + 4$ $PC \leftarrow (PC_{31..20} : IMM20 \times 4)$
Assembler Syntax:	<code>call label</code>
Example:	<code>call write_char</code>
Description:	Saves the address of the next instruction in register <code>ra</code> , and transfers execution to the instruction at address $(PC_{31..20} : IMM20 \times 4)$.
Usage:	<code>call</code> can transfer execution anywhere within the 4-megabyte (MB) range determined by $PC_{31..20}$. The Nios II DPX GNU linker does not automatically handle cases in which the address is out of this range.
Extended Register Restrictions:	None
Exceptions:	None
Instruction Type:	IX
Instruction Fields:	$IMM20 = 20\text{-bit unsigned immediate value}$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x0				0							0x0		IMM20																		

callr

call subroutine in register

Operation: $ra \leftarrow PC + 4$
 $PC \leftarrow rA$

Assembler Syntax: `callr rA`

Example: `callr r6`

Description: Saves the address of the next instruction in the return address register, and transfers execution to the address contained in register rA.

Usage: `callr` is used to dereference C-language function pointers.

Extended Register Restrictions: None

Instruction Type: R-3

Instruction Fields: A = Register index of operand rA

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				A				0				0x1f				0		0		0x1D											

cmpeq**compare equal**

Operation: if (rA == rB)
then rC ← 1
else rC ← 0

Assembler Syntax: cmpeq rC, rA, rB

Example: cmpeq r6, r7, r8

Description: If rA == rB, then stores 1 to rC; otherwise, stores 0 to rC.

Usage: cmpeq performs the == operation of the C programming language. Also, cmpeq can be used to implement the C logical negation operator "!".

cmpeq rC, rA, zero # Implements rC = !rA

Extended Register Restrictions: rB cannot be an extension register.

Exceptions: None

Instruction Type: R-3

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				A						B						C						0	0		0x20						

cmpeqi

compare equal immediate

Operation:	if ($rA \sigma (\text{IMM12})$) then $rB \leftarrow 1$ else $rB \leftarrow 0$
Assembler Syntax:	<code>cmpeqi rB, rA, IMM12</code>
Example:	<code>cmpeqi r6, r7, 100</code>
Description:	Sign-extends the 12-bit immediate value IMM12 to 32 bits and compares it to the value of rA. If $rA == \sigma (\text{IMM12})$, <code>cmpeqi</code> stores 1 to rB; otherwise stores 0 to rB.
Usage:	<code>cmpeqi</code> performs the <code>==</code> operation of the C programming language.
Extended Register Restrictions:	None
Exceptions:	None
Instruction Type:	I-12
Instruction Fields:	A = Register index of operand rA B = Register index of operand rB IMM12 = 12-bit signed immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x1				A				B				0x4				IMM12															

cmpge**compare greater than or equal signed**

Operation: if ((signed) rA >= (signed) rB)
then rC ← 1
else rC ← 0

Assembler Syntax: cmpge rC, rA, rB

Example: cmpge r6, r7, r8

Description: If rA >= rB, then stores 1 to rC; otherwise stores 0 to rC.

Usage: cmpge performs the signed >= operation of the C programming language.

Extended Register Restrictions: rB cannot be an extension register.

Exceptions: None

Instruction Type: R-3

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x3				A						B						C						0	0		0x8							

cmpgei compare greater than or equal signed immediate

Operation:	if ((signed) rA >= (signed) σ (IMM12)) then rB \leftarrow 1 else rB \leftarrow 0
Assembler Syntax:	cmpgei rB, rA, IMM12
Example:	cmpgei r6, r7, 100
Description:	Sign-extends the 12-bit immediate value IMM12 to 32 bits and compares it to the value of rA. If rA >= α (IMM12), then cmpgei stores 1 to rB; otherwise stores 0 to rB.
Usage:	cmpgei performs the signed >= operation of the C programming language.
Extended Register Restrictions:	None
Exceptions:	None
Instruction Type:	I-12
Instruction Fields:	A = Register index of operand rA B = Register index of operand rB IMM12 = 12-bit signed immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x1				A				B				0x1				IMM12															

cmpgeu**compare greater than or equal unsigned**

Operation: if ((unsigned) rA >= (unsigned) rB)
then rC ← 1
else rC ← 0

Assembler Syntax: cmpgeu rC, rA, rB

Example: cmpgeu r6, r7, r8

Description: If rA >= rB, then stores 1 to rC; otherwise stores 0 to rC.

Usage: cmpgeu performs the unsigned >= operation of the C programming language.

Extended Register Restrictions: rB cannot be an extension register.

Exceptions: None

Instruction Type: R-3

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				A						B						C						0	0		0x28						

cmpgeui compare greater than or equal unsigned immediate

Operation:	if ((unsigned) rA >= (unsigned) (0x0000 : IMM12)) then rB ← 1 else rB ← 0
Assembler Syntax:	cmpgeui rB, rA, IMM12
Example:	cmpgeui r6, r7, 100
Description:	Zero-extends the 12-bit immediate value IMM12 to 32 bits and compares it to the value of rA. If rA >= (0x0000 : IMM12), then cmpgeui stores 1 to rB; otherwise stores 0 to rB.
Usage:	cmpgeui performs the unsigned >= operation of the C programming language.
Extended Register Restrictions:	None
Exceptions:	None
Instruction Type:	I-12
Instruction Fields:	A = Register index of operand rA B = Register index of operand rB IMM12 = 12-bit unsigned immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x1				A				B				0x5				IMM12															

cmpgt**compare greater than signed**

Operation:	if ((signed) rA > (signed) rB) then rC ← 1 else rC ← 0
Assembler Syntax:	cmpgt rC, rA, rB
Example:	cmpgt r6, r7, r8
Description:	If rA > rB, then stores 1 to rC; otherwise stores 0 to rC.
Usage:	cmpgt performs the signed > operation of the C programming language.
Extended Register Restrictions:	rA cannot be an extension register.
Pseudo-instruction:	cmpgt is implemented with the cmplt instruction by swapping its rA and rB operands.

cmpgti

compare greater than signed immediate

Operation:	if ((signed) rA > (signed) IMMED) then rB ← 1 else rB ← 0
Assembler Syntax:	<code>cmpgti rB, rA, IMMED</code>
Example:	<code>cmpgti r6, r7, 100</code>
Description:	Sign-extends the immediate value IMMED to 32 bits and compares it to the value of rA. If rA > α (IMMED), then <code>cmpgti</code> stores 1 to rB; otherwise stores 0 to rB.
Usage:	<code>cmpgti</code> performs the signed > operation of the C programming language. The maximum allowed value of IMMED is 32766. The minimum allowed value is -32769.
Extended Register Restrictions:	None
Pseudo-instruction:	<code>cmpgti</code> is implemented using a <code>cmpgei</code> instruction with an IMM12 immediate value of IMMED + 1.

cmpgtu**compare greater than unsigned**

Operation:	if ((unsigned) rA > (unsigned) rB) then rC \leftarrow 1 else rC \leftarrow 0
Assembler Syntax:	cmpgtu rC, rA, rB
Example:	cmpgtu r6, r7, r8
Description:	If rA > rB, then stores 1 to rC; otherwise stores 0 to rC.
Usage:	cmpgtu performs the unsigned > operation of the C programming language.
Extended Register Restrictions:	rA cannot be an extension register.
Pseudo-instruction:	cmpgtu is implemented with the <code>cmpltu</code> instruction by swapping its rA and rB operands.

cmpgtui

compare greater than unsigned immediate

Operation:	if ((unsigned) rA > (unsigned) IMMED) then rB \leftarrow 1 else rB \leftarrow 0
Assembler Syntax:	cmpgtui rB, rA, IMMED
Example:	cmpgtui r6, r7, 100
Description:	Zero-extends the immediate value IMMED to 32 bits and compares it to the value of rA. If rA > IMMED, then cmpgtui stores 1 to rB; otherwise stores 0 to rB.
Usage:	cmpgtui performs the unsigned > operation of the C programming language. The maximum allowed value of IMMED is 65534. The minimum allowed value is 0.
Extended Register Restrictions:	None
Pseudo-instruction:	cmpgtui is implemented using a cmpgeui instruction with an IMM12 immediate value of IMMED + 1.

cmple**compare less than or equal signed**

Operation:	if ((signed) rA <= (signed) rB) then rC ← 1 else rC ← 0
Assembler Syntax:	cmple rC, rA, rB
Example:	cmple r6, r7, r8
Description:	If rA <= rB, then stores 1 to rC; otherwise stores 0 to rC.
Usage:	cmple performs the signed <= operation of the C programming language.
Extended Register Restrictions:	rA cannot be an extension register.
Pseudo-instruction:	cmple is implemented with the cmpge instruction by swapping its rA and rB operands.

cmplei **compare less than or equal signed immediate**

Operation:	if ((signed) rA < (signed) IMMED) then rB \leftarrow 1 else rB \leftarrow 0
Assembler Syntax:	cmplei rB, rA, IMMED
Example:	cmplei r6, r7, 100
Description:	Sign-extends the immediate value IMMED to 32 bits and compares it to the value of rA. If rA \leq α (IMMED), then cmplei stores 1 to rB; otherwise stores 0 to rB.
Usage:	cmplei performs the signed \leq operation of the C programming language. The maximum allowed value of IMMED is 32766. The minimum allowed value is -32769.
Extended Register Restrictions:	None
Pseudo-instruction:	cmplei is implemented using a cmplti instruction with an IMM12 immediate value of IMMED + 1.

cmpleu**compare less than or equal unsigned**

Operation:	if ((unsigned) rA < (unsigned) rB) then rC ← 1 else rC ← 0
Assembler Syntax:	cmpleu rC, rA, rB
Example:	cmpleu r6, r7, r8
Description:	If rA <= rB, then stores 1 to rC; otherwise stores 0 to rC.
Usage:	cmpleu performs the unsigned <= operation of the C programming language.
Extended Register Restrictions:	rA cannot be an extension register.
Pseudo-instruction:	cmpleu is implemented with the cmpgeu instruction by swapping its rA and rB operands.

cmpleui **compare less than or equal unsigned immediate**

Operation:	if ((unsigned) rA <= (unsigned) IMMED) then rB ← 1 else rB ← 0
Assembler Syntax:	cmpleui rB, rA, IMMED
Example:	cmpleui r6, r7, 100
Description:	Zero-extends the immediate value IMMED to 32 bits and compares it to the value of rA. If rA <= IMMED, then cmpleui stores 1 to rB; otherwise stores 0 to rB.
Usage:	cmpleui performs the unsigned <= operation of the C programming language. The maximum allowed value of IMMED is 65534. The minimum allowed value is 0.
Extended Register Restrictions:	None
Pseudo-instruction:	cmpleui is implemented using a cmpltui instruction with an IMM12 immediate value of IMMED + 1.

cmplt**compare less than signed**

Operation: if ((signed) rA < (signed) rB)
then rC ← 1
else rC ← 0

Assembler Syntax: cmplt rC, rA, rB

Example: cmplt r6, r7, r8

Description: If rA < rB, then stores 1 to rC; otherwise stores 0 to rC.

Usage: cmplt performs the signed < operation of the C programming language.

Extended Register Restrictions: rB cannot be an extension register.

Exceptions: None

Instruction Type: R-3

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				A						B						C						0	0		0x10						

cmplti

compare less than signed immediate

Operation:	if ((signed) $rA < (\text{signed}) \sigma(\text{IMM12})$) then $rB \leftarrow 1$ else $rB \leftarrow 0$
Assembler Syntax:	<code>cmplti rB, rA, IMM12</code>
Example:	<code>cmplti r6, r7, 100</code>
Description:	Sign-extends the 12-bit immediate value IMM12 to 32 bits and compares it to the value of rA. If $rA < \sigma(\text{IMM12})$, then <code>cmplti</code> stores 1 to rB; otherwise stores 0 to rB.
Usage:	<code>cmplti</code> performs the signed < operation of the C programming language.
Extended Register Restrictions:	None
Exceptions:	None
Instruction Type:	I-12
Instruction Fields:	A = Register index of operand rA B = Register index of operand rB IMM12 = 12-bit signed immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x1				A				B				0x2				IMM12															

cmpltu**compare less than unsigned**

Operation: if ((unsigned) rA < (unsigned) rB)
then rC ← 1
else rC ← 0

Assembler Syntax: cmpltu rC, rA, rB

Example: cmpltu r6, r7, r8

Description: If rA < rB, then stores 1 to rC; otherwise stores 0 to rC.

Usage: cmpltu performs the unsigned < operation of the C programming language.

Extended Register Restrictions: rB cannot be an extension register.

Exceptions: None

Instruction Type: R-3

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				A						B						C						0	0		0x30						

cmpltui

compare less than unsigned immediate

Operation: if ((unsigned) rA < (unsigned) (0x0000 : IMM12))
then rB ← 1
else rB ← 0

Assembler Syntax: cmpltui rB, rA, IMM12

Example: cmpltui r6, r7, 100

Description: Zero-extends the 12-bit immediate value IMM12 to 32 bits and compares it to the value of rA. If rA < (0x0000 : IMM12), then cmpltui stores 1 to rB; otherwise stores 0 to rB.

Usage: cmpltui performs the unsigned < operation of the C programming language.

Extended Register Restrictions: None

Exceptions: None

Instruction Type: I-12

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
IMM12 = 12-bit unsigned immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x1				A				B				0x6				IMM12															

cmpne**compare not equal**

Operation: if (rA != rB)
then rC ← 1
else rC ← 0

Assembler Syntax: cmpne rC, rA, rB

Example: cmpne r6, r7, r8

Description: If rA != rB, then stores 1 to rC; otherwise stores 0 to rC.

Usage: cmpne performs the != operation of the C programming language.

Extended Register Restrictions: rB cannot be an extension register.

Exceptions: None

Instruction Type: R-3

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				A						B						C						0	0		0x18						

cmpnei

compare not equal immediate

Operation: if ($rA \neq \sigma(\text{IMM12})$)
then $rB \leftarrow 1$
else $rB \leftarrow 0$

Assembler Syntax: `cmpnei rB, rA, IMM12`

Example: `cmpnei r6, r7, 100`

Description: Sign-extends the 12-bit immediate value IMM12 to 32 bits and compares it to the value of rA. If $rA \neq \sigma(\text{IMM12})$, then `cmpnei` stores 1 to rB; otherwise stores 0 to rB.

Usage: `cmpnei` performs the `!=` operation of the C programming language.

Extended Register Restrictions: None

Exceptions: None

Instruction Type: I-12

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
IMM12 = 12-bit signed immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x1				A				B				0x3				IMM12															

eret**exception return****Operation:** $PC \leftarrow ea$ **Assembler Syntax:** `eret`**Example:** `eret`**Description:** Transfers execution to the address in `ea`.**Usage:** Use `eret` to return from exception handling routines.**Extended Register Restrictions:** None**Instruction Type:** R-3**Instruction Fields:** None

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x3				A						B						C						0	0		0x1							

extract

extract bit field region

Operation: $rB \leftarrow \text{zero_extend} (rA [\text{msb} : \text{lsb}])$

Assembler Syntax: `extract rB, rA, msb, lsb`

Example: `extract r6, r5, 22, 10`

Description: Extracts the specified bit field from rA, zero extends the value, and stores result into rB. The extract instruction performs the equivalent of a logical right shift of rA by LSB bits followed by a mask to force bits above MSB to zero.

Instruction Type: BMX I-12

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
MSB = 5-bit unsigned immediate value
LSB = 5-bit unsigned immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x2				A (src1)						B (dst)						0xE				0		MSB					LSB				

insert**insert bit field region**

Operation: $rB \leftarrow \{ rB [31 : (msb+1)], rA [msb-lsb : 0], rB [(lsb-1) : 0] \}$

Assembler Syntax: `insert rB, rA, msb, lsb`

Example: `insert r6, r5, 22, 10`

Description: The `insert` instruction inserts bits from `rA` into `rB`. It performs the equivalent of a left shift of `rA` by `LSB` bits followed by a bit-by-bit muxing between bits in `rA` and `rB`. The result is both a source and a destination register number). If `MSB` is 31, there are no `rB` bits above the shifted `rA` field in the destination. If `LSB` is 0, there are no `rB` bits below the shifted `rA` field in the destination.

The `insert` instruction is an optional instruction. It can be configured at system generation time. If software attempts to execute an `insert` instruction on a processor that does not implement the instruction, the processor generates an exception.

Attempting to execute the `insert` instruction on an Nios II DPX MTP that does not implement it causes an unimplemented instruction exception.

Exceptions: Unimplemented instruction

Instruction Type: BMX I-12

Instruction Fields: `A` = Register index of operand `rA`
`B` = Register index of operand `rB`
`MSB` = 5-bit unsigned immediate value
`LSB` = 5-bit unsigned immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x2				A (src1)						B (dst)						0xE				0		MSB					LSB				

jmp

computed jump

Operation: $PC \leftarrow rA$

Assembler Syntax: `jmp rA`

Example: `jmp r12`

Description: Transfers execution to the address contained in register rA.

Usage: It is illegal to jump to the address contained in register r31. To return from subroutines called by `call` or `callr`, use `ret` instead of `jmp`.

Extended Register Restrictions: None

Instruction Type: R-3

Instruction Fields: A = Register index of operand rA

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				A						B						C						0	0		0xD						

jmp*i*

jump immediate

Operation:	$PC \leftarrow (PC_{31..28} : IMM26 \times 4)$
Assembler Syntax:	<code>jmp<i>i</i> label</code>
Example:	<code>jmp<i>i</i> write_char</code>
Description:	Transfers execution to the instruction at address $(PC_{31..28} : IMM26 \times 4)$.
Usage:	<code>jmp<i>i</i></code> is a low-overhead local jump. <code>jmp<i>i</i></code> can transfer execution anywhere within the 256-MB range determined by $PC_{31..28}$. The Nios II DPX GNU linker does not automatically handle cases in which the address is out of this range.
Extended Register Restrictions:	None
Exceptions:	None
Instruction Type:	IX
Instruction Fields:	$IMM26$ = 26-bit unsigned immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x0				IMM26[25:20]								0x1	IMM26[19:0]																		

jrel

jump relative

Operation: $pc \leftarrow pc + 4 + (rA \ll 2)$

Assembler Syntax: jrel rA

Example: jrel r12

Description: Transfers execution to the instruction at address $(pc + 4 + (rA \ll 2))$.

Usage: This instruction can be used to jump into a branch table to implement C-language switch statements efficiently. Note that this instruction is only efficient when the case constants in the switch statement are grouped closely together. pc is a byte address.

Extended Register Restrictions: None

Exceptions:

Instruction Type: R-3

Instruction Fields: A = Register index of operand rA

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				A				0x0				0x0				0		0		0x35											

ldb

load byte from memory or I/O peripheral

Operation:

$rB \leftarrow \sigma(\text{Mem8}[rA + \sigma(\text{IMM12})])$

Assembler Syntax:

ldb rB, byte_offset(rA)

Example:

ldb r6, 100(r5)

Description:

Computes the effective byte address specified by the sum of rA and the instruction's signed 12-bit immediate value. Loads register rB with the desired memory byte, sign extending the 8-bit value to 32 bits.

Usage:

Extended Register Restrictions:

rA cannot be an extension register.

Instruction Type:

I-12

Instruction Fields:

A = Register index of operand rA

B = Register index of operand rB

IMM12 = 12-bit signed immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x2				A (dst)						B (addr src)						0x0				IMM12											

ldbu

load unsigned byte from memory or I/O peripheral

Operation: $rB \leftarrow 0x000000 : \text{Mem8}[rA + \sigma(\text{IMM12})]$

Assembler Syntax: `ldbu rB, byte_offset(rA)`

Example: `ldbu r6, 100(r5)`

Description: Computes the effective byte address specified by the sum of rA and the instruction's signed 12-bit immediate value. Loads register rB with the desired memory byte, zero extending the 8-bit value to 32 bits.

Usage:

Extended Register Restrictions: rA cannot be an extended register.

Instruction Type: I-12

Instruction Fields:
A = Register index of operand rA
B = Register index of operand rB
IMM12 = 12-bit signed immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x2				A (dst)						B (addr src)						0x4				IMM12 (addr offset)											

ldh

load halfword from memory or I/O peripheral

Operation:

$rB \leftarrow \sigma(\text{Mem16}[rA + \sigma(\text{IMM12})])$

Assembler Syntax:

ldh rB, byte_offset(rA)

Example:

ldh r6, 100(r5)

Description:

Computes the effective byte address specified by the sum of rA and the instruction's signed 12-bit immediate value. Loads register rB with the memory halfword located at the effective byte address, sign extending the 12-bit value to 32 bits. The effective byte address must be halfword aligned. If the byte address is not a multiple of 2, the operation is undefined.

Usage:

Extended Register Restrictions:

rA cannot be an extended register.

Instruction Type:

I-12

Instruction Fields:

A = Register index of operand rA
B = Register index of operand rB
IMM12 = 12-bit signed immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x2				A (dst)						B (addr src)						0x1		IMM12 (addr offset)													

ldhu load unsigned halfword from memory or I/O peripheral

Operation: $rB \leftarrow 0x0000 : \text{Mem16}[rA + \sigma(\text{IMM12})]$

Assembler Syntax: `ldhu rB, byte_offset(rA)`

Example: `ldhu r6, 100(r5)`

Description: Computes the effective byte address specified by the sum of rA and the instruction's signed 12-bit immediate value. Loads register rB with the memory halfword located at the effective byte address, zero extending the 12-bit value to 32 bits. The effective byte address must be halfword aligned. If the byte address is not a multiple of 2, the operation is undefined.

Usage:

Extended Register Restrictions: rA cannot be an extended register.

Instruction Type: I-12

Instruction Fields:
A = Register index of operand rA
B = Register index of operand rB
IMM12 = 12-bit signed immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x2				A (dst)						B (addr src)						0x5			IMM12 (addr offset)												

ldw

load 32-bit word from memory or I/O peripheral

Operation:

$rB \leftarrow \text{Mem32}[rA + \sigma(\text{IMM12})]$

Assembler Syntax:

ldw rB, byte_offset(rA)

Example:

ldw r6, 100(r5)

Description:

Computes the effective byte address specified by the sum of rA and the instruction's signed 12-bit immediate value. Loads register rB with the memory word located at the effective byte address. The effective byte address must be word aligned. If the byte address is not a multiple of 4, the operation is undefined.

Usage:

Extended Register Restrictions:

rA cannot be an extended register.

Instruction Type:

I-12

Instruction Fields:

A = Register index of operand rA
B = Register index of operand rB
IMM12 = 12-bit signed immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x2				A (dst)						B (addr src)						0x2				IMM12 (addr offset)											

merge

merge bit field region

- Operation:** $rA \leftarrow \{ rA [31 : (msb+1)], rB [msb : lsb], rA [(lsb-1) : 0] \}$
- Assembler Syntax:** `merge rA, rB, msb, lsb`
- Example:** `merge r6, r5, 22, 10`
- Description:** The `merge` instruction merges bits from `rB` into `rA`. It performs the equivalent of an `insert` instruction except there is no shift operation and bits are written to `rA` instead of `rB`.
- Usage:** The `merge` instruction can be used to clear any contiguous range of bits in any register connected to `src1` by using zero (`r0`) for the `rB` source.
- Instruction Type:** BMX I-12
- Instruction Fields:**
- A = Register index of operand `rA`
 - B = Register index of operand `rB`
 - MSB = 5-bit unsigned immediate value
 - LSB = 5-bit unsigned immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0x2				A (src1 & dst)							B (src2)							0xF				0		MSB					LSB				

mov**move register to register**

Operation:	$rC \leftarrow rA$
Assembler Syntax:	<code>mov rC, rA</code>
Example:	<code>mov r6, r7</code>
Description:	Moves the contents of rA to rC.
Extended Register Restrictions:	None
Pseudo-instruction:	<code>mov</code> is implemented as <code>add rC, rA, zero</code> .

movhi

move immediate into high halfword

Operation: $rB \leftarrow (IMM16 : 0x0000)$

Assembler Syntax: `movhi rB, IMM16`

Example: `movhi r6, 0x8000`

Description: Writes the immediate value IMM16 into the high halfword of rB, and clears the lower halfword of rB to 0x0000.

Usage: The maximum allowed value of IMM16 is 65535. The minimum allowed value is 0. To load a 32-bit constant into a register, first load the upper 16 bits using a `movhi` pseudo-instruction. The `%hi()` macro can be used to extract the upper 16 bits of a constant or a label. Then, load the lower 16 bits with an `ori` instruction. The `%lo()` macro can be used to extract the lower 16 bits of a constant or label as shown in the following example.

```
movhi rB, %hi(value)
ori rB, rB, %lo(value)
```

An alternative method to load a 32-bit constant into a register uses the `%hiadj()` macro and the `addi` instruction as shown in the following example.

```
movhi rB, %hiadj(value)
addi rB, rB, %lo(value)
```

Extended Register Restrictions: None

Pseudo-instruction: `movhi` is implemented as `orhi rB, zero, IMM16`.

movhi20**move 20 bit immediate into high halfword**

Operation: $rA \leftarrow (IMM20 \ll 12)$

Assembler Syntax: `movhi rB, IMM20`

Example: `movhi r6, 0x8000`

Description: Writes the immediate value IMM20 into the high 20 bits of rB, and clears the lower 12 bits of rB to 0x000.

Usage: The maximum allowed value of IMM20 is 1048575. The minimum allowed value is 0. To load a 32-bit constant into a register, first load the upper 20 bits using a `movhi20` instruction. The `%hi20()` macro can be used to extract the upper 20 bits of a constant or a label. Then, load the lower 12 bits with an `ori` instruction. The `%lo12()` macro can be used to extract the lower 12 bits of a constant or label as shown in the following example.

```
movhi20 rB, %hi20(value)
ori rB, rB, %lo12(value)
```

Extended Register Restrictions: None

Exceptions:**Instruction Type:****Instruction Fields:**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x0				A (dst)								0x2		IMM20																	

movi

move signed immediate into word

Operation:	$rB \leftarrow \sigma(\text{IMM12})$
Assembler Syntax:	<code>movi rB, IMM12</code>
Example:	<code>movi r6, -30</code>
Description:	Sign-extends the immediate value IMM12 to 32 bits and writes it to rB.
Usage:	The maximum allowed value of IMM12 is 2047. The minimum allowed value is -2048. To load a 32-bit constant into a register, refer to the <code>movhi</code> instruction.
Extended Register Restrictions:	None
Pseudo-instruction:	<code>movi</code> is implemented as <code>addi rB, zero, IMM12</code> .

movia**move immediate address into word**

Operation:	$rB \leftarrow \text{label}$
Assembler Syntax:	<code>movia rB, label</code>
Example:	<code>movia r6, function_address</code>
Description:	Writes the address of label to rB.
Extended Register Restrictions:	None
Pseudo-instruction:	movia is implemented as: <code>movhi20 rB, %hi20adj(label)</code> <code>addi rB, rB, %lo12(label)</code>

movui

move unsigned immediate into word

Operation: $rB \leftarrow (0x0000 : IMM16)$

Assembler Syntax: `movui rB, IMM16`

Example: `movui r6, 100`

Description: Zero-extends the immediate value IMM16 to 32 bits and writes it to rB.

Usage: The maximum allowed value of IMM16 is 65535. The minimum allowed value is 0. To load a 32-bit constant into a register, refer to the `movhi` instruction.

Extended Register Restrictions: None

Pseudo-instruction: `movui` is implemented as `ori rB, zero, IMM16`.

mul**multiply**

Operation: $rC \leftarrow (rA \times rB)_{31..0}$

Assembler Syntax: `mul rC, rA, rB`

Example: `mul r6, r7, r8`

Description: Multiplies rA times rB and stores the 32 low-order bits of the product to rC. The result is the same whether the operands are treated as signed or unsigned integers.

Usage:**Carry Detection (unsigned operands):**

Before or after the multiply operation, the carry out of the MSB of rC can be detected using the following instruction sequence:

```
mul rC, rA, rB           # The mul operation (optional)
mulxuu rD, rA, rB        # rD is nonzero if carry occurred
cmpne rD, rD, zero       # rD is 1 if carry occurred, 0 if not
```

The `mulxuu` instruction writes a nonzero value into rD if the multiplication of unsigned numbers generates a carry (unsigned overflow). If a 0/1 result is desired, follow the `mulxuu` with the `cmpne` instruction.

Overflow Detection (signed operands):

After the multiply operation, overflow can be detected using the following instruction sequence:

```
mul rC, rA, rB           # The original mul operation
cmplt rD, rC, zero
mulxss rE, rA, rB
add rD, rD, rE            # rD is nonzero if overflow
cmpne rD, rD, zero       # rD is 1 if overflow, 0 if not
```

The `cmplt-mulxss-add` instruction sequence writes a nonzero value into rD if the product in rC cannot be represented in 32 bits (signed overflow). If a 0/1 result is desired, follow the instruction sequence with the `cmpne` instruction.

Extended Register Restrictions:

rB cannot be an extended register.

Instruction Type: R-3

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				A (src1)							B (src2)							C (dst)				0	0		0x27						

multi

multiply immediate

Operation:

$$rB \leftarrow (rA \times \alpha(\text{IMM12}))_{31..0}$$

Assembler Syntax:

multi rB, rA, IMM12

Example:

multi r6, r7, -100

Description:

Sign-extends the 12-bit immediate value IMM12 to 32 bits and multiplies it by the value of rA. Stores the 32 low-order bits of the product to rB. The result is independent of whether rA is treated as a signed or unsigned number.

Carry Detection and Overflow Detection:

For a discussion of carry and overflow detection, refer to the mul instruction.

Instruction Type:

I-12

Instruction Fields:

A = Register index of operand rA

B = Register index of operand rB

IMM12 = 12-bit signed immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x2				A (src1)						B (dst)						0xB				IMM12 (src2)											

mulxss**multiply extended signed/signed**

Operation: $rC \leftarrow ((\text{signed}) rA) \times ((\text{signed}) rB))_{63..32}$

Assembler Syntax: `mulxss rC, rA, rB`

Example: `mulxss r6, r7, r8`

Description: Treating `rA` and `rB` as signed integers, `mulxss` multiplies `rA` times `rB`, and stores the 32 high-order bits of the product to `rC`.

Nios II DPX MTPs that do not implement the `mulxss` instruction cause an unimplemented instruction exception.

Usage: Use `mulxss` and `mul` to compute the full 64-bit product of two 32-bit signed integers. Furthermore, `mulxss` can be used as part of the calculation of a 128-bit product of two 64-bit signed integers. Given two 64-bit integers, each contained in a pair of 32-bit registers, (`S1 : U1`) and (`S2 : U2`), their 128-bit product is $(U1 \times U2) + ((S1 \times U2) \ll 32) + ((U1 \times S2) \ll 32) + ((S1 \times S2) \ll 64)$. The `mulxss` and `mul` instructions are used to calculate the 64-bit product $S1 \times S2$.

Extended Register Restrictions: `rB` cannot be an extended register.

Exceptions: Unimplemented instruction

Instruction Type: R-3

Instruction Fields: `A` = Register index of operand `rA`
`B` = Register index of operand `rB`
`C` = Register index of operand `rC`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				A (src1)						B (src2)						C (dst)						0	0		0x1F						

mulxsu

multiply extended signed/unsigned

Operation: $rC \leftarrow ((\text{signed})\ rA) \times ((\text{unsigned})\ rB))_{63..32}$

Assembler Syntax: `mulxsu rC, rA, rB`

Example: `mulxsu r6, r7, r8`

Description: Treating *rA* as a signed integer and *rB* as an unsigned integer, `mulxsu` multiplies *rA* times *rB*, and stores the 32 high-order bits of the product to *rC*.

Nios II DPX MTPs that do not implement the `mulxsu` instruction cause an unimplemented instruction exception.

Usage: `mulxsu` can be used as part of the calculation of a 128-bit product of two 64-bit signed integers. Given two 64-bit integers, each contained in a pair of 32-bit registers, (*S1* : *U1*) and (*S2* : *U2*), their 128-bit product is: $(U1 \times U2) + ((S1 \times U2) \ll 32) + ((U1 \times S2) \ll 32) + ((S1 \times S2) \ll 64)$. The `mulxsu` and `mul` instructions are used to calculate the two 64-bit products $S1 \times U2$ and $U1 \times S2$.

Extended Register Restrictions: *rB* cannot be an extended register.*

Exceptions: Unimplemented instruction

Instruction Type: R-3

Instruction Fields: *A* = Register index of operand *rA*
B = Register index of operand *rB*
C = Register index of operand *rC*

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				A (src1)						B (src2)						C (dst)						0	0		0x17						

mulxuu**multiply extended unsigned/unsigned**

Operation: $rC \leftarrow ((\text{unsigned}) rA) \times ((\text{unsigned}) rB))_{63..32}$

Assembler Syntax: `mulxuu rC, rA, rB`

Example: `mulxuu r6, r7, r8`

Description: Treating rA and rB as unsigned integers, `mulxuu` multiplies rA times rB and stores the 32 high-order bits of the product to rC.

Nios II DPX MTPs that do not implement the `mulxuu` instruction cause an unimplemented instruction exception.

Usage: Use `mulxuu` and `mul` to compute the 64-bit product of two 32-bit unsigned integers. Furthermore, `mulxuu` can be used as part of the calculation of a 128-bit product of two 64-bit signed integers. Given two 64-bit signed integers, each contained in a pair of 32-bit registers, (S1 : U1) and (S2 : U2), their 128-bit product is $(U1 \times U2) + ((S1 \times U2) \ll 32) + ((U1 \times S2) \ll 32) + ((S1 \times S2) \ll 64)$. The `mulxuu` and `mul` instructions are used to calculate the 64-bit product $U1 \times U2$.

`mulxuu` also can be used as part of the calculation of a 128-bit product of two 64-bit unsigned integers. Given two 64-bit unsigned integers, each contained in a pair of 32-bit registers, (T1 : U1) and (T2 : U2), their 128-bit product is $(U1 \times U2) + ((U1 \times T2) \ll 32) + ((T1 \times U2) \ll 32) + ((T1 \times T2) \ll 64)$. The `mulxuu` and `mul` instructions are used to calculate the four 64-bit products $U1 \times U2$, $U1 \times T2$, $T1 \times U2$, and $T1 \times T2$.

Extended Register Restrictions: rB cannot be an extended register.

Exceptions: Unimplemented instruction

Instruction Type: R-3

Instruction Fields: A = Register index of operand rA

B = Register index of operand rB

C = Register index of operand rC

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				A (src1)						B (src2)						C (dst)						0	0		0x7						

nextpc

get address of following instruction

Operation: $rC \leftarrow PC + 4$

Assembler Syntax: `nextpc rC`

Example: `nextpc r6`

Description: Stores the address of the next instruction to register rC.

Usage: A relocatable code fragment can use `nextpc` to calculate the address of its data segment. `nextpc` is the only way to access the PC directly.

Extended Register Restrictions: None

Exceptions: None

Instruction Type: R-3

Instruction Fields: c = Register index of operand rC

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				A (src1)						B (src2)						C (dst)						0	0		0x15						

nop**no operation**

Operation:	None
Assembler Syntax:	<code>nop</code>
Example:	<code>nop</code>
Description:	<code>nop</code> does nothing.
Extended Register Restrictions:	None
Pseudo-instruction:	<code>nop</code> is implemented as <code>add zero, zero, zero</code> .

nor

bitwise logical nor

Operation: $rC \leftarrow \sim(rA \mid rB)$

Assembler Syntax: `nor rC, rA, rB`

Example: `nor r6, r7, r8`

Description: Calculates the bitwise logical NOR of rA and rB and stores the result in rC.

Extended Register Restrictions: rB cannot be an extended register.

Exceptions: None

Instruction Type: R-3

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				A (src1)						B (src2)						C (dst)						0	0		0x6						

or**bitwise logical or****Operation:** $rC \leftarrow rA \mid rB$ **Assembler Syntax:** `or rC, rA, rB`**Example:** `or r6, r7, r8`**Description:** Calculates the bitwise logical OR of rA and rB and stores the result in rC.**Extended Register Restrictions:** rB cannot be an extended register.**Exceptions:** None**Instruction Type:** R-3**Instruction Fields:**
A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				A (src1)						B (src2)						C (dst)						0	0		0x16						

orhi bitwise logical or immediate into high halfword

Operation: $rB \leftarrow rA \mid (\text{IMM16} : 0x0000)$

Assembler Syntax: `orhi rB, rA, IMM16`

Example: `orhi r6, r7, 100`

Description: Calculates the bitwise logical OR of rA and (IMM16 : 0x0000) and stores the result in rB.

Extended Register Restrictions: None

Exceptions: None

Instruction Type: I-16

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit signed immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0xE				A (src1)						B (dst)						IMM16 (src2)															

ori**bitwise logical or immediate****Operation:** $rB \leftarrow rA \mid (0x0000 : IMM16)$ **Assembler Syntax:** `ori rB, rA, IMM16`**Example:** `ori r6, r7, 100`**Description:** Calculates the bitwise logical OR of rA and (0x0000 : IMM16) and stores the result in rB.**Extended Register Restrictions:** None**Exceptions:** None**Instruction Type:** I-16**Instruction Fields:**
A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit unsigned immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0xA				A (src1)						B (dst)						IMM16 (src2)															

rdctl

read from control register

Operation: $rC \leftarrow \text{ctlN}$

Assembler Syntax: `rdctl rC, ctlN`

Example: `rdctl r3, ctl31`

Description: Reads the value contained in control register `ctlN` and writes it to register `rC`.
Applicable both to standard control registers and extended control registers.

Extended Register Restrictions: None

Exceptions:

Instruction Type: R-3

Instruction Fields: `C` = Register index of operand `rC`
`N` = Control register index of operand `ctlN`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x3				0x0												C (dst)						0	0		0x26							

ret**return from subroutine**

Operation: $PC \leftarrow ra$

Assembler Syntax: `ret`

Example: `ret`

Description: Transfers execution to the address in `ra`.

Usage: Any subroutine called by `call` or `callr` must use `ret` to return.

Extended Register Restrictions: None

Instruction Type: R-3

Instruction Fields: None

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				0				0				0				0	0		0x5												

rol

rotate left

Operation: $rC \leftarrow rA \text{ rotated left } rB_{4..0} \text{ bit positions}$

Assembler Syntax: `rol rC, rA, rB`

Example: `rol r6, r7, r8`

Description: Rotates rA left by the number of bits specified in $rB_{4..0}$ and stores the result in rC. The bits that shift out of the register rotate into the least-significant bit positions. Bits 31–5 of rB are ignored.

Extended Register Restrictions: rB cannot be an extended register.

Exceptions: None

Instruction Type: R-3

Instruction Fields:
A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				A (src1)						B (src2)						C (dst)						0	0		0x4						

rol**rotate left immediate****Operation:** $rC \leftarrow rA \text{ rotated left IMM5 bit positions}$ **Assembler Syntax:** `rol rC, rA, IMM5`**Example:** `rol r6, r7, 3`**Description:** Rotates rA left by the number of bits specified in IMM5 and stores the result in rC. The bits that shift out of the register rotate into the least-significant bit positions.**Usage:** In addition to the rotate-left operation, `rol` can be used to implement a rotate-right operation. Rotating left by $(32 - \text{IMM5})$ bits is the equivalent of rotating right by IMM5 bits.**Extended Register Restrictions:** None**Exceptions:** None**Instruction Type:** I-5**Instruction Fields:**
A = Register index of operand rA
C = Register index of operand rC
IMM5 = 5-bit unsigned immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x1				A (src1)						B (dst)						0x0				0			0x0			IMM5 (src2)					

ror

rotate right

Operation: $rC \leftarrow rA \text{ rotated right } rB_{4..0} \text{ bit positions}$

Assembler Syntax: `ror rC, rA, rB`

Example: `ror r6, r7, r8`

Description: Rotates rA right by the number of bits specified in $rB_{4..0}$ and stores the result in rC. The bits that shift out of the register rotate into the most-significant bit positions. Bits 31– 5 of rB are ignored.

Extended Register Restrictions: rB cannot be an extended register.

Exceptions: None

Instruction Type: R-3

Instruction Fields:
A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				A (src1)						B (src2)						C (dst)						0	0		0xC						

sll**shift left logical**

Operation: $rC \leftarrow rA \ll (rB_{4..0})$

Assembler Syntax: `sll rC, rA, rB`

Example: `sll r6, r7, r8`

Description: Shifts rA left by the number of bits specified in rB_{4..0} (inserting zeroes), and then stores the result in rC. `sll` performs the `<<` operation of the C programming language.

Extended Register Restrictions: rB cannot be an extended register.

Exceptions: None

Instruction Type: R-3

Instruction Fields:
 A = Register index of operand rA
 B = Register index of operand rB
 C = Register index of operand rC

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				A (src1)							B (src2)							C (dst)				0	0		0x14						

slli

shift left logical immediate

Operation: $rC \leftarrow rA \ll IMM5$

Assembler Syntax: `slli rC, rA, IMM5`

Example: `slli r6, r7, 3`

Description: Shifts rA left by the number of bits specified in IMM5 (inserting zeroes), and then stores the result in rC.

Usage: `slli` performs the `<<` operation of the C programming language.

Extended Register Restrictions: None

Exceptions: None

Instruction Type: I-5

Instruction Fields:
A = Register index of operand rA
C = Register index of operand rC
IMM5 = 5-bit unsigned immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x1				A (src1)						C (dst)						0x0				0				0x2			IMM5 (src2)				

sra**shift right arithmetic****Operation:** $rC \leftarrow (\text{signed})\ rA \gg ((\text{unsigned})\ rB_{4..0})$ **Assembler Syntax:** `sra rC, rA, rB`**Example:** `sra r6, r7, r8`**Description:** Shifts rA right by the number of bits specified in rB_{4..0} (duplicating the sign bit), and then stores the result in rC. Bits 31–5 are ignored.**Usage:** `sra` performs the signed `>>` operation of the C programming language.**Extended Register Restrictions:** rB cannot be an extended register.**Exceptions:** None**Instruction Type:** R-3**Instruction Fields:**
A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				A (src1)							B (src2)							C (dst)				0	0		0x3C						

srai

shift right arithmetic immediate

- Operation:

$rC \leftarrow (\text{signed})\ rA \gg ((\text{unsigned})\ IMM5)$
- Assembler Syntax:

srai rC, rA, IMM5
- Example:

srai r6, r7, 3
- Description:

Shifts rA right by the number of bits specified in IMM5 (duplicating the sign bit), and then stores the result in rC.
- Usage:

srai performs the signed >> operation of the C programming language.
- Extended Register Restrictions:

None
- Exceptions:

None
- Instruction Type:

I-5
- Instruction Fields:

A = Register index of operand rA
C = Register index of operand rC
IMM5 = 5-bit unsigned immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x1				A (src1)						C (dst)						0x0				0			0x7			IMM5 (src2)					

srl**shift right logical**

Operation: $rC \leftarrow (\text{unsigned})\ rA \gg ((\text{unsigned})\ rB_{4..0})$

Assembler Syntax: `srl rC, rA, rB`

Example: `srl r6, r7, r8`

Description: Shifts rA right by the number of bits specified in rB_{4..0} (inserting zeroes), and then stores the result in rC. Bits 31–5 are ignored.

Usage: `srl` performs the unsigned `>>` operation of the C programming language.

Extended Register Restrictions: rB cannot be an extended register.

Exceptions: None

Instruction Type: R-3

Instruction Fields:
 A = Register index of operand rA
 B = Register index of operand rB
 C = Register index of operand rC

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				A (src1)						B (src2)						C (dst)						0	0		0x1C						

srli

shift right logical immediate

Operation:	$rC \leftarrow (\text{unsigned})\ rA \gg ((\text{unsigned})\ IMM5)$
Assembler Syntax:	srli rC, rA, IMM5
Example:	srli r6, r7, 3
Description:	Shifts rA right by the number of bits specified in IMM5 (inserting zeroes), and then stores the result in rC.
Usage:	srli performs the unsigned >> operation of the C programming language.
Extended Register Restrictions:	None
Exceptions:	None
Instruction Type:	I-5
Instruction Fields:	A = Register index of operand rA C = Register index of operand rC IMM5 = 5-bit unsigned immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x1				A (src1)						C (dst)						0x0				0			0x3			IMM5 (src2)					

stb

store byte to memory or I/O peripheral

Operation:

$$\text{Mem8}[\text{rA} + \sigma(\text{IMM12})] \leftarrow \text{rB}_{7:0}$$

Assembler Syntax:

stb rB, byte_offset(rA)

Example:

stb r6, 100(r5)

Description:

Computes the effective byte address specified by the sum of rA and the instruction's signed 12-bit immediate value. Stores the low byte of rB to the memory byte specified by the effective address.

Usage:

Extended Register Restrictions:

rA cannot be an extended register.

Instruction Type:

I-12

Instruction Fields:

A = Register index of operand rA
B = Register index of operand rB
IMM12 = 12-bit signed immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x2				A (st data src)				B (addr src)				0x8				IMM12 (addr offset)															

sth store halfword to memory or I/O peripheral

Operation: $\text{Mem16}[\text{rA} + \sigma(\text{IMM12})] \leftarrow \text{rB}_{15:0}$

Assembler Syntax: `sth rB, byte_offset(rA)`

Example: `sth r6, 100(r5)`

Description: Computes the effective byte address specified by the sum of rA and the instruction's signed 12-bit immediate value. Stores the low halfword of rB to the memory location specified by the effective byte address. The effective byte address must be halfword aligned. If the byte address is not a multiple of 2, the operation is undefined.

Usage:

Extended Register Restrictions: rA cannot be an extended register.

Instruction Type: I-12

Instruction Fields:
A = Register index of operand rA
B = Register index of operand rB
IMM12 = 12-bit signed immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x2				A (st data src)				B (addr src)				0x9				IMM12 (addr offset)															

stw**store word to memory or I/O peripheral**

Operation: $\text{Mem32}[\text{rA} + \sigma(\text{IMM12})] \leftarrow \text{rB}$

Assembler Syntax: `stw rB, byte_offset(rA)`

Example: `stw r6, 100(r5)`

Description: Computes the effective byte address specified by the sum of rA and the instruction's signed 12-bit immediate value. Stores rB to the memory location specified by the effective byte address. The effective byte address must be word aligned. If the byte address is not a multiple of 4, the operation is undefined.

Usage:

Extended Register Restrictions: rA cannot be an extended register.

Instruction Type: I-12

Instruction Fields:
 A = Register index of operand rA
 B = Register index of operand rB
 IMM12 = 12-bit signed immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x2				A (st data src)				B (addr src)				0xA				IMM12 (addr offset)															

sub

subtract

Operation: $rC \leftarrow rA - rB$

Assembler Syntax: `sub rC, rA, rB`

Example: `sub r6, r7, r8`

Description: Subtract rB from rA and store the result in rC.

Usage: **Carry Detection (unsigned operands):**

The carry bit indicates an unsigned overflow. Before or after a `sub` operation, a carry out of the MSB can be detected by checking whether the first operand is less than the second operand. The carry bit can be written to a register, or a conditional branch can be taken based on the carry condition. The following examples show both cases.

```
sub rC, rA, rB           # The original sub operation (optional)
cmpltu rD, rA, rB        # rD is written with the carry bit
sub rC, rA, rB           # The original sub operation (optional)
bltu rA, rB, label       # Branch if carry generated
```

Overflow Detection (signed operands):

Detect overflow of signed subtraction by comparing the sign of the difference that is written to rC with the signs of the operands. If rA and rB have different signs, and the sign of rC is different than the sign of rA, an overflow occurred. The overflow condition can control a conditional branch, as shown in the following example.

```
sub rC, rA, rB           # The original sub operation
xor rD, rA, rB           # Compare signs of rA and rB
xor rE, rA, rC           # Compare signs of rA and rC
and rD, rD, rE           # Combine comparisons
blt rD, zero, label      # Branch if overflow occurred
```

Extended Register Restrictions: rB cannot be an extended register.

Exceptions: None

Instruction Type: R-3

Instruction Fields: A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				A (src1)							B (src2)							C (dst)				0	0		0x39						

subi**subtract immediate**

Operation:	$rB \leftarrow rA - \sigma(\text{IMMED})$
Assembler Syntax:	<code>subi rB, rA, IMMED</code>
Example:	<code>subi r8, r8, 4</code>
Description:	Sign-extends the immediate value IMMED to 32 bits, subtracts it from the value of rA and then stores the result in rB.
Usage:	The maximum allowed value of IMMED is 32768. The minimum allowed value is -32767.
Extended Register Restrictions:	None
Pseudo-instruction:	subi is implemented as <code>addi rB, rA, -IMMED</code>

trap

trap

Operation:	$ea \leftarrow PC + 4$ $PC \leftarrow \text{exception handler address}$
Assembler Syntax:	<code>trap</code> <code>trap imm5</code>
Example:	<code>trap</code>
Description:	Saves the address of the next instruction in register <code>ea</code> , and transfers execution to the exception handler. The address of the exception handler is specified at system generation time.

The 5-bit immediate field `imm5` is ignored by the processor, but it can be used by the debugger.

`trap` with no argument is the same as `trap 0`.

Usage:	To return from the exception handler, execute an <code>eret</code> instruction.
---------------	---

Extended Register Restrictions:	None
--	------

Exceptions:	Trap
--------------------	------

Instruction Type:	I-5
--------------------------	-----

Instruction Fields:	<code>IMM5</code> = Type of breakpoint
----------------------------	--

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x1				0				0x1d				0x0				0				0x4				IMM5 (src2)							

wrctl**write to control register**

Operation:	$ctlN \leftarrow rA$
Assembler Syntax:	wrctl ctlN, rA
Example:	wrctl ctl6, r3
Description:	Writes the value contained in register rA to the control register ctlN. Applicable both to standard control registers and extended control registers.
Extended Register Restrictions:	None
Exceptions:	
Instruction Type:	R-3
Instruction Fields:	A = Register index of operand rA N = Control register index of operand ctlN

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				A (src1)						N (dst)						0x0				0	0		0x2E								

XOR

bitwise logical exclusive or

Operation: $rC \leftarrow rA \wedge rB$

Assembler Syntax: `xor rC, rA, rB`

Example: `xor r6, r7, r8`

Description: Calculates the bitwise logical exclusive-or of rA and rB and stores the result in rC.

Extended Register Restrictions: rB cannot be an extended register.

Exceptions: None

Instruction Type: R-3

Instruction Fields:
A = Register index of operand rA
B = Register index of operand rB
C = Register index of operand rC

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				A (src1)						B (src2)						C (dst)						0	0		0x1E						

xorhi **bitwise logical exclusive or immediate into high halfword****Operation:** $rB \leftarrow rA \wedge (\text{IMM16} : 0x0000)$ **Assembler Syntax:** `xorhi rB, rA, IMM16`**Example:** `xorhi r6, r7, 100`**Description:** Calculates the bitwise logical exclusive XOR of rA and (IMM16 : 0x0000) and stores the result in rB.**Extended Register Restrictions:** None**Exceptions:** None**Instruction Type:** I-16**Instruction Fields:**
A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit unsigned immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0xF				A (src1)						B (dst)						IMM16 (src2)															

xori

bitwise logical exclusive or immediate

Operation: $rB \leftarrow rA \wedge (0x0000 : IMM16)$

Assembler Syntax: `xori rB, rA, IMM16`

Example: `xori r6, r7, 100`

Description: Calculates the bitwise logical exclusive OR of rA and (0x0000 : IMM16) and stores the result in rB.

Extended Register Restrictions: None

Exceptions: None

Instruction Type: I-16

Instruction Fields:
A = Register index of operand rA
B = Register index of operand rB
IMM16 = 16-bit unsigned immediate value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0xB				A (src1)						B (dst)						IMM16 (src2)															

Nios II DPX Extended Instruction Set Reference

This section describes Nios II DPX MTP instructions that are specific to event-driven, multi-threaded processing.

Table 9-16 lists option codes that you can use in the message control word argument to the `snd` and `sndi` instructions. These option codes are defined in `nios2dpx.h`.

Table 9-16. OPT bits

Instruction Bit	OPT Field Bit	Name	Description
31:9	26:4	—	Reserved
8	3	OPT_KEESEQNUM	<p>Bypass the sequence number reorder buffer. Turning this option on has the following effects:</p> <ul style="list-style-type: none"> ■ Prevents the message from being reordered ■ Prevents the message sequence number from being returned to the free sequence number list <p>For information about how to use this flag correctly, see “Advanced Topics” on page 5-23.</p> <p>This flag has no effect if sequence number ordering is disabled in the hardware.</p>
7	2	OPT_CIDORDER	Pass message through the CID reorder buffer. This option enables the message to be reordered by CID.
6	1	OPT_SNDEXIT	<p>Transfer control of thread to the scheduler. Tell the scheduler to release the thread (exit) after <code>snd</code> instruction.</p> <p>This option is used to save a thread cycle when a task exits. If another task is not ready to run, the thread might continue to execute until it reaches an <code>exit</code> instruction. This behavior is not predictable.</p> <p>You must restrict use of the <code>OPT_SNDEXIT</code> option to tasks implemented in assembly language. In C, it would terminate the thread before the epilogue runs, corrupting the stack.</p>
5	0	OPT_FREECID	Free the associated CID.

Table 9-17 and Table 9-18 specify the argument encodings used by the `snd` and `sndi` instructions.

Table 9-17. Message Destination Word

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DSTID																OTID															

Table 9-18. Message Control Word

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OPT (snd)																											LENGTH (1)				
																OPT (s _{ndi}) (2)											LENGTH (1)				

Note to Table 9-18:

- (1) To transmit all available TX registers, set `LENGTH` to zero.
- (2) The `sndi` instruction limits the message control word to a 16-bit immediate value. Therefore `sndi` only supports 11 option bits.

For the meanings of the option bits, see Table 9-16.

Table 9-19 lists the status codes supported by the extended instructions. These status codes are defined in `nios2dpx.h`.

Table 9-19. Extended Instruction Status Codes

Status Code	Meaning
ERR_OK	Instruction successful
ERR_NO_ID	Thread does not own a TXID or CID
ERR_BUF_FULL	Queue is full

cidalloc**CID allocate**

Operation: Obtains a new CID.

Assembler Syntax: `cidalloc rC`

Example: `cidalloc r1`

Description: Gets a new CID and swaps to it, keeping the old CID in reserve. Does nothing if there is a CID in reserve already. `cidalloc` also gets a TXID if needed

Set `rC` to the status as follows:

- `ERR_OK`—The new CID and TXID are valid.
- `ERR_NO_ID`—Failure to allocate either a valid CID or a valid TXID. The thread continues to use the old CID.

If CIDs are not implemented in the Nios II DPX datapath processor, the `cidalloc` instruction is treated as a `nop`.

Once allocated, the currently active CID is switched to the new CID. Following a `snd` instruction, the currently active CID reverts to the original CID.

Extended Register Restrictions: None

Exceptions:

Instruction Type: R-3

Instruction Fields: C = Register index of operand `rC`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				0				0				C (dst)				0		0		0x12											

exit

exit

Operation:	Exits the currently running thread.
Assembler Syntax:	exit
Example:	exit
Description:	If the thread owns an RXID, mark it invalid and release it to the RXID free list. Then put the processor thread into an idle state.
Extended Register Restrictions:	None
Exceptions:	None
Instruction Type:	R-3

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				0				0				0				0				0				0xA							

rxfree

RXID free

- Operation:

Frees the RXID of a thread
- Assembler Syntax:

rxfree rC
- Example:

rxfree r6
- Description:

Returns a thread's RXID to the RXID free list, and marks the thread as not owning an RXID. Does nothing if the thread did not own an RXID.

Sets rC to the status as follows:

■ ERR_OK—The RXID is valid and is freed successfully.

■ ERR_NO_ID—The RXID is not valid.

If the hardware is configured with zero RXIDs, this instruction is treated as nop.
- Extended Register Restrictions:

None
- Exceptions:

None
- Instruction Type:

R-3
- Instruction Fields:

C = Register index of operand rC

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				0				0				C (dst)				0	0		0x3A												

snd

send

Operation:	Sends a message with register message control word
Assembler Syntax:	snd rC, rA, rB
Example:	snd r1, r2, r3
Description:	<p>Sends a message using the current CID. If there is a CID in reserve, switch to it, else, continue with the current CID. Mark the CID in reserve as invalid to allow another <code>cidalloc</code> to work.</p> <p>The <code>snd</code> instruction specifies:</p> <ul style="list-style-type: none"> ■ The PE – by using a unique destination ID (<code>DSTID</code>) ■ The operation to run on that PE (<code>OTID</code>) ■ Some operational flags (<code>OPT</code>) ■ The number of TX registers to send in the message (<code>LENGTH</code>) <p>Adds the message specified by the message destination and control words to the Tx message queue</p> <p>If thread owns a TXID, mark it invalid.</p> <p><code>rA</code> = Message destination word. See Table 9-17 on page 9-104.</p> <p><code>rB</code> = Message control word. See Table 9-18 on page 9-104.</p> <p>Sets <code>rC</code> to the status as follows:</p> <ul style="list-style-type: none"> ■ <code>ERR_OK</code>—The message is sent correctly. ■ <code>ERR_NO_ID</code>—No TXID is owned. The message is not sent. ■ <code>ERR_BUF_FULL</code>—The TX queue is full. The message is not sent.
Extended Register Restrictions:	None
Exceptions:	None
Instruction Type:	R-3
Instruction Fields:	<p>A = Register index of operand rA</p> <p>B = Register index of operand rB</p> <p>C = Register index of operand rC</p>

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				A (src1)							B (src2)							C (dst)				0	0		0x22						

sndi**send immediate**

Operation: Sends a message with immediate message control word

Assembler Syntax: `sndi rB, rA, imm16`

Example: `sndi r1, r2, imm16`

Description: `rA` = Message destination word. See [Table 9-17 on page 9-104](#).
`imm16` = Message control word. See [Table 9-18 on page 9-104](#).

Because the message control word is limited to a 16-bit immediate value, this instruction only supports 11 option bits.

Aside from specifying the limitations on the message control word, the `sndi` instruction is identical to the `snd` instruction.

Extended Register Restrictions: None

Exceptions: None

Instruction Type: I-16

Instruction Fields: A = Register index of operand rA
 B = Register index of operand rB

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x4				A (src1)							B (dst)					IMM16 (src2)															

txalloc

TXID allocate

Operation: Requests a TXID.

Assembler Syntax: txalloc rC

Example: txalloc r6

Description: If the thread does not currently own a TXID, request one. The return value indicates whether the thread owns a valid TXID or not. If TXID is valid, set *<status>* = 0 [ERR_OK]. If TXID is not valid, set *<status>* = 1 [ERR_NO_ID]. Returns: rC = *<status>*.
If the hardware is configured with zero TXIDs, this instruction is treated as nop.

Extended Register Restrictions: None

Exceptions: None

Instruction Type: R-3

Instruction Fields: C = Register index of operand rC

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				0				0				C (dst)				0	0		0x32												

The Nios II DPX MTP Application Binary Interface

This section describes the Application Binary Interface (ABI) for the Nios II DPX MTP. The ABI describes:

- How data is arranged in memory
- Behavior and structure of the stack
- Function calling conventions

Data Types

Table 9–20 shows the size and representation of the C data types for the Nios II DPX MTP.

Table 9–20. Representation of Data Types

Type	Size (Bytes)	Representation
char, signed char	1	two's complement (ASCII)
unsigned char	1	binary (ASCII)
short, signed short	2	two's complement
unsigned short	2	binary
int, signed int	4	two's complement
unsigned int	4	binary
long, signed long	4	two's complement
unsigned long	4	binary
pointer	4	binary
long long	8	two's complement
unsigned long long	8	binary

Memory Alignment

Contents in memory are aligned as follows:

- A function must be aligned to a minimum of 32-bit boundary.
- The minimum alignment of a data element is its natural size. A data element larger than 32 bits need only be aligned to a 32-bit boundary.
- Structures, unions, and strings must be aligned to a minimum of 32 bits.
- Bit fields inside structures are always 32-bit aligned.

Register Usage

The ABI adds additional usage conventions to the Nios II DPX MTP register file defined in [Chapter 5, Software Programming Model](#). The ABI uses the registers as shown in [Table 9–21](#).



The general purpose registers r32 through r63 are reserved registers. The Nios II DPX C compiler ignores these registers when doing register allocation.

Table 9–21. Nios II DPX ABI Register Usage

Register	Name	Used by Compiler	Callee-Saved (1)	Normal Usage
r0	zero	✓		0x00000000
r1	at			Assembler temporary
r2		✓		Return value (least-significant 32 bits)
r3		✓		Return value (most-significant 32 bits)
r4		✓		Register arguments (first 32 bits)
r5		✓		Register arguments (second 32 bits)
r6		✓		Register arguments (third 32 bits)
r7		✓		Register arguments (fourth 32 bits)
r8		✓		Callee-saved general-purpose registers
r9		✓		
r10		✓		
r11		✓		
r12		✓		
r13		✓		
r14		✓		
r15		✓		
r16		✓	✓	Callee-saved general-purpose registers
r17		✓	✓	
r18		✓	✓	
r19		✓	✓	
r20		✓	✓	
r21		✓	✓	
r22		✓	✓	
r23		✓	✓	
r24		✓	✓	
r25	bt			Break temporary
r26	gp	✓		Global pointer
r27	sp	✓		Stack pointer
r28	fp	✓	(2)	Frame pointer
r29	ea			Exception return address
r30	ba			Break return address
r31	ra	✓		Return address

Notes to Table 9–21:

- (1) A function can use one of these registers if it saves it first. The function must restore the register's original value before exiting.
- (2) If the frame pointer is not used, the register is available as a callee-saved temporary register. Refer to "Frame Pointer Elimination" on page 9–116.

The endianness of values greater than 8 bits is BE-8.



For information about BE-8 data representation, refer to the *Nios II DPX Architecture* chapter, in the *Nios II DPX Hardware Reference* section of the *Nios II DPX Datapath Processor Handbook*.

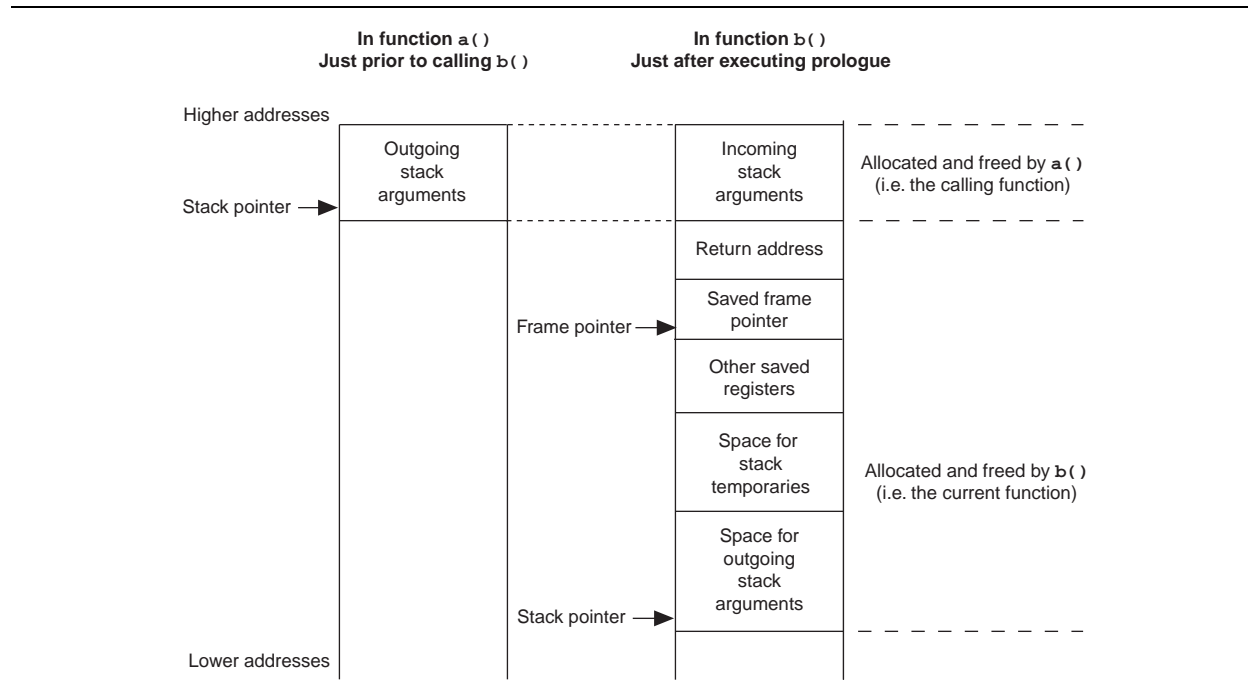
Stacks

The stack grows downward, towards lower addresses. The stack pointer points to the last used slot. The frame pointer points to the saved frame pointer near the top of the stack frame.

Function Stack Setup

Figure 9-1 shows an example of the structure of a current frame. In this case, function `a()` calls function `b()`, and the stack is shown before the call and after the prologue in the called function has completed.

Figure 9-1. Stack Pointer, Frame Pointer and the Current Frame



Each section of the current frame is aligned to a 32-bit boundary. The ABI requires the stack pointer be 32-bit aligned at all times.

Task Stack Setup

Use the task attribute to designate a function as a task entry point.

The task prologue sets up the stack and frame pointers much like the standard prologue. Unlike the normal prologue, the task prologue does not save the callee-saved registers. The task function assumes all callee saved registers are empty scratch registers upon entry into the task function. Another difference between a task and a standard function is that a task function has no arguments on the stack.

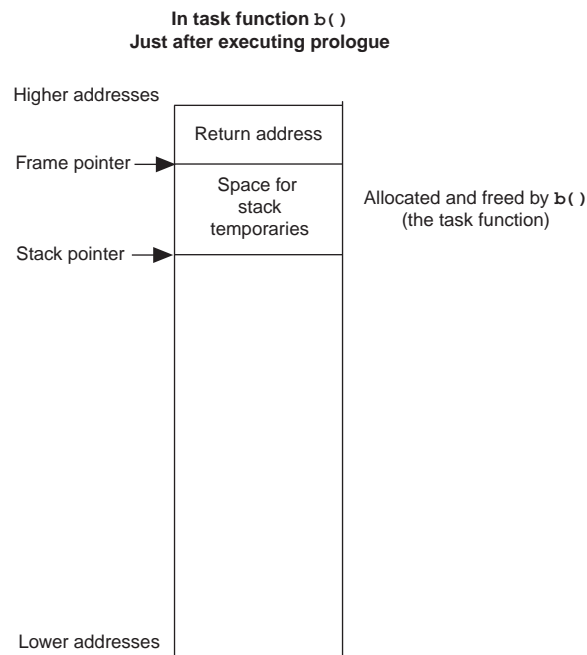
The task epilogue restores the stack and frame pointers just like the normal epilogue. Unlike the normal epilogue, the task epilogue does not restore the callee-saved registers. The task epilogue terminates with the `exit` instruction instead of the `ret` instruction.

A task function is declared as follows:

```
void __attribute__((task (<task id>))) <task function name> () {}
```

Figure 9-2 shows an example of the stack for a task function. The stack is shown after the prologue in the task has completed.

Figure 9-2. Stack Pointer in a Task Function



Naked Stack Setup

Use the `naked` attribute to indicate that the specified function does not need the prologue and epilogue sequences generated by the compiler. It is up to the programmer to provide these sequences.

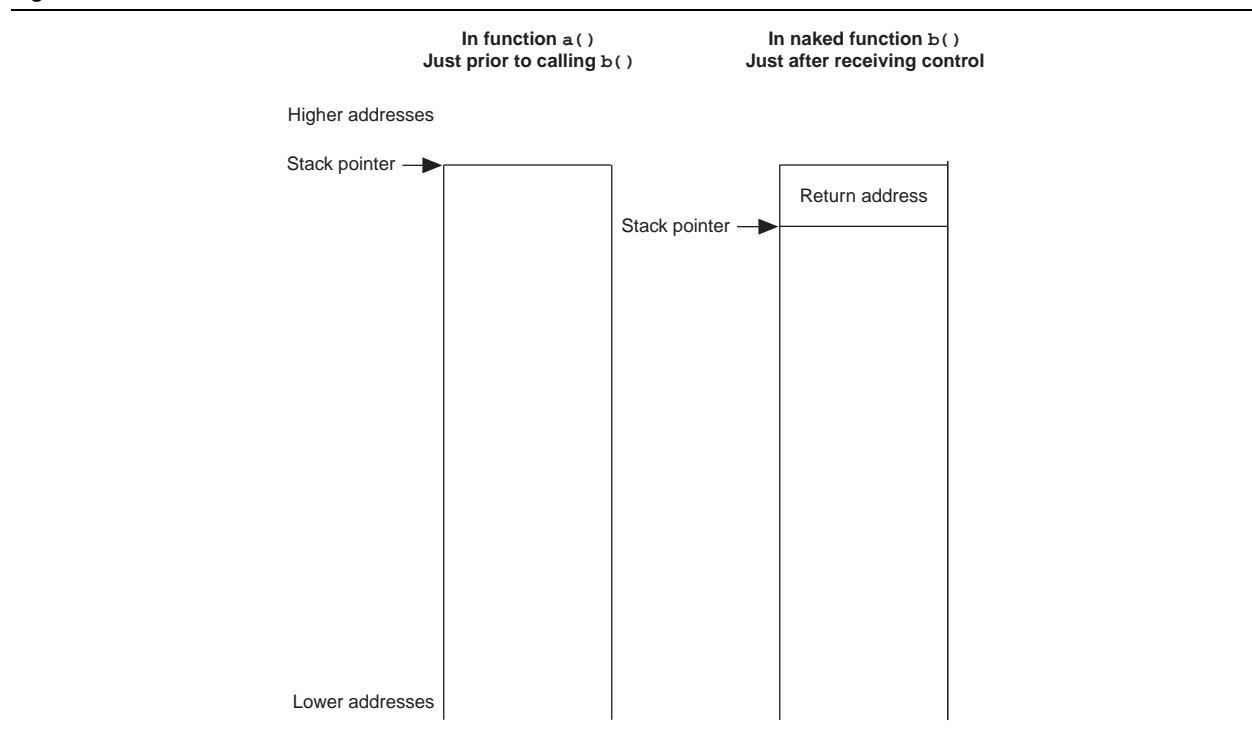
Naked functions are used to implement the body of an assembly function, while allowing the compiler to construct the requisite function declaration for the assembler. The only statements that can be safely included in naked functions are assembly language statements that do not have operands. Avoid using other statements, especially declarations of local variables and `if` statements.

A naked function is declared as follows:

```
void __attribute__((naked)) my_naked_function() {}
```

Figure 9-3 shows an example of the stack for a naked function. The stack is shown after control has passed to the first statement in the function.

Figure 9-3. Stack Pointer in a Naked Function



Frame Pointer Elimination

The frame pointer is provided for debugger support. If you are not using a debugger, you can optimize your code by eliminating the frame pointer, using the `-fomit-frame-pointer` compiler option. When the frame pointer is eliminated, register `fp` is available as a temporary register.

Call Saved Registers

The compiler is responsible for saving registers that need to be saved in a function. If there are any such registers, they are saved on the stack, from high to low addresses, in the following order: `ra`, `fp`, `r2`, `r3`, `r4`, `r5`, `r6`, `r7`, `r8`, `r9`, `r10`, `r11`, `r12`, `r13`, `r14`, `r15`, `r16`, `r17`, `r18`, `r19`, `r20`, `r21`, `r22`, `r23`, `r24`, `r25`, `gp`, and `sp`. Stack space is not allocated for registers that are not saved.

Further Examples of Stacks

There are a number of special cases for stack layout, which are described in this section.

Stack Frame for a Function With `alloca()`

The Nios II DPX stack frame implementation provides support for the `alloca()` function, defined in the Berkeley Software Distribution (BSD) extension to C, and implemented by the gcc compiler. Figure 9-4 depicts what the frame looks like after `alloca()` is called. The space allocated by `alloca()` replaces the outgoing arguments and the outgoing arguments get new space allocated at the bottom of the frame.


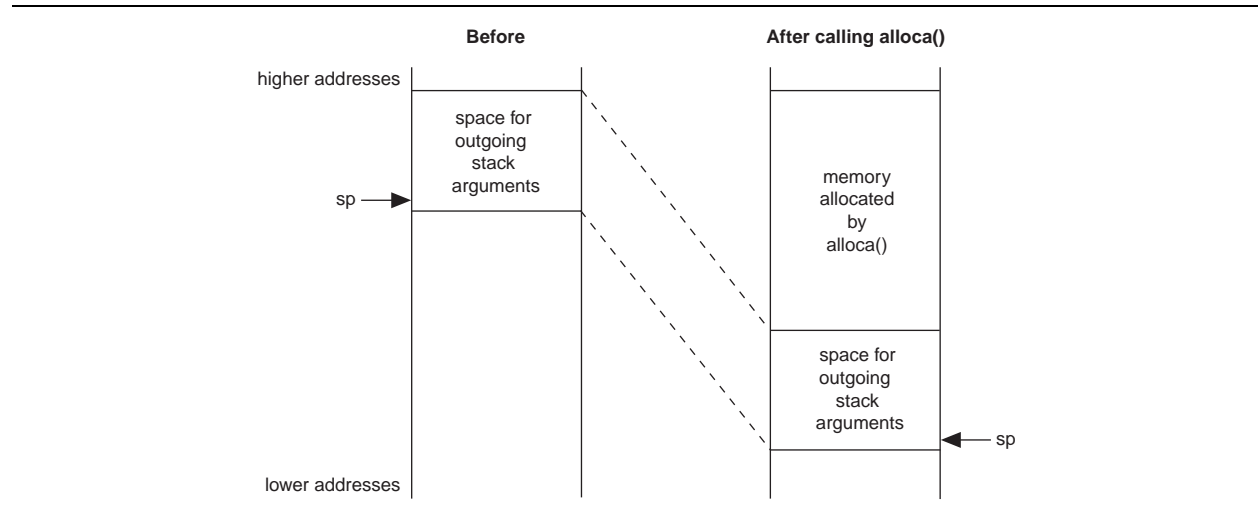
 The Nios II DPX C compiler maintains a frame pointer for any function that calls `alloca()`, even if `-fomit-frame-pointer` is specified.

Figure 9-4. Stack Frame after Calling `alloca()`

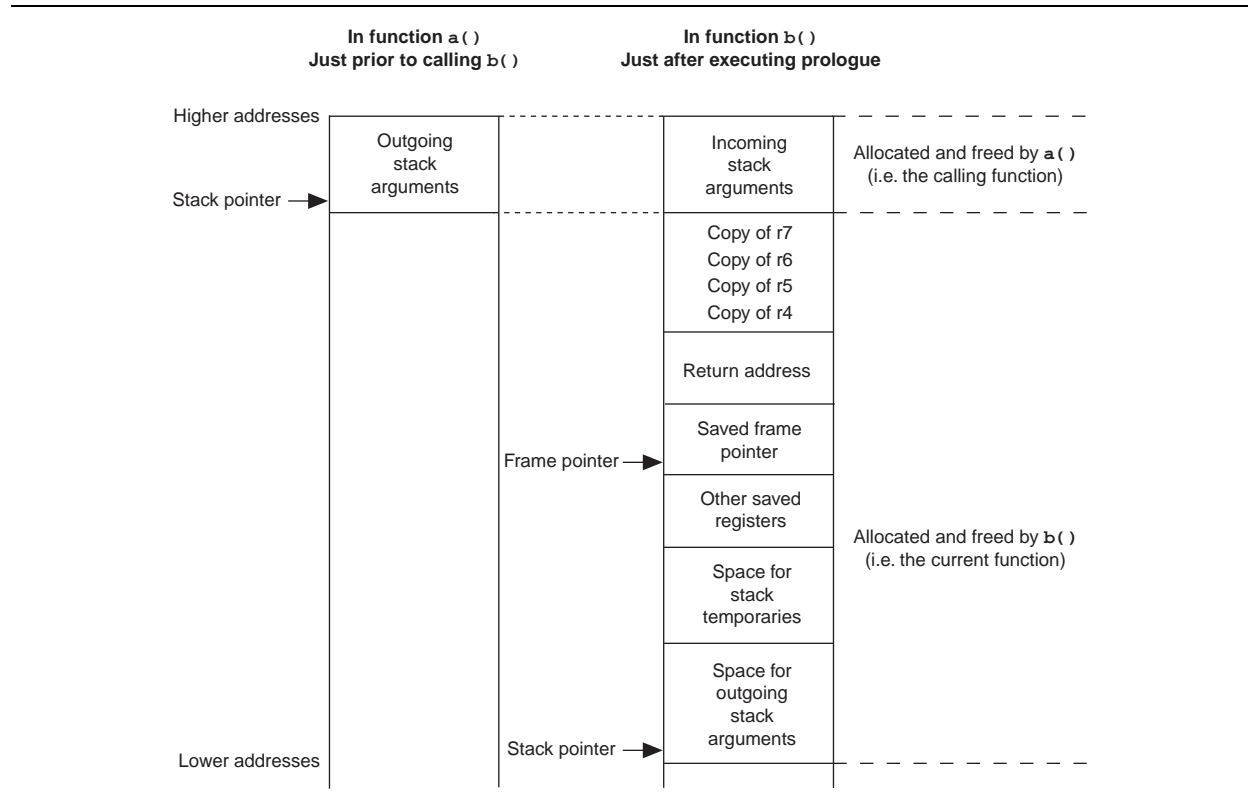


Stack Frame for a Function with Variable Arguments

Functions that take variable arguments (varargs) still have their first 16 bytes of arguments arriving in registers r4 through r7, just like other functions.

In order for varargs to work, functions that take variable arguments allocate 16 extra bytes of storage on the stack. They copy to the stack the first 16 bytes of their arguments from registers r4 through r7 as shown in [Figure 9-5](#).

Figure 9-5. Stack Frame Using Variable Arguments



Stack Frame for a Function with Structures Passed By Value

Functions that take struct value arguments still have their first 16 bytes of arguments arriving in registers r4 through r7, just like other functions.

If part of a structure is passed using registers, the function might need to copy the register contents back to the stack. This operation is similar to that required in the variable arguments case as shown in [Figure 9-5](#).

Function Prologues

The Nios II DPX C compiler generates function prologues that allocate the stack frame of a function for storage of stack temporaries and outgoing arguments. In addition, each prologue is responsible for saving the state of the calling function. This entails saving certain registers on the stack. These registers, the callee-saved registers, are listed in [Table 9-21 on page 9-113](#). A function prologue is required to save a callee-saved register only if the function uses the register.

Given the function prologue algorithm, when doing a back trace, a debugger can disassemble instructions and reconstruct the processor state of the calling function.



An even better way to find out what the prologue has done is to use information stored in the DWARF-2 debugging fields of the executable and linkable format (.elf) file.

The instructions found in a Nios II DPX function prologue perform the following tasks:

- Adjust the stack pointer (to allocate the frame)
- Store registers to the frame
- Set the frame pointer to the location of the saved frame pointer

Example 9-1 shows a function prologue.

Example 9-1. A function prologue

```
/* Adjust the stack pointer */
addi    sp, sp, -16    /* make a 16-byte frame */

/* Store registers to the frame */
stw     ra, 12(sp)     /* store the return address */
stw     fp, 8(sp)      /* store the frame pointer*/
stw     r16, 4(sp)     /* store callee-saved register */
stw     r17, 0(sp)     /* store callee-saved register */

/* Set the new frame pointer */
addi    fp, sp, 8
```

Prologue Variations

The following variations can occur in a prologue:

- In a task function, callee-saved registers are not saved.
- In a naked function, the prologue is absent.
- If the function's frame size is greater than 2047 bytes, extra temporary registers are used in the calculation of the new stack pointer as well as for the offsets of where to store callee-saved registers. The extra registers are needed because of the maximum size of immediate values allowed by the Nios II DPX MTP.
- If the frame pointer is not in use, the final instruction, recalculating the frame pointer, is not generated.
- If variable arguments are used, extra instructions store the argument registers on the stack.
- If the compiler designates the function as a leaf function, the return address is not saved.
- If optimizations are on, especially instruction scheduling, the order of the instructions might change and become interlaced with instructions located after the prologue.

Arguments and Return Values

This section discusses the details of passing arguments to functions and returning values from functions.

Arguments

The first 16 bytes to a function are passed in registers r4 through r7. The arguments are passed as if a structure containing the types of the arguments were constructed, and the first 16 bytes of the structure are located in r4 through r7.

A simple example:

```
int function (int a, int b);
```

The equivalent structure representing the arguments is:

```
struct { int a; int b; };
```

The first 16 bytes of the struct are assigned to r4 through r7. Therefore r4 is assigned the value of *a* and r5 the value of *b*.

The first 16 bytes to a function taking variable arguments are passed the same way as a function not taking variable arguments. The called function must clean up the stack as necessary to support the variable arguments. Refer to [“Stack Frame for a Function with Variable Arguments”](#) on page 9-118.

Return Values

Return values of types up to 8 bytes are returned in r2 and r3. For return values greater than 8 bytes, the caller must allocate memory for the result and must pass the address of the result memory as a hidden zero argument.

The hidden zero argument is best explained through an example.

Example 9-2. Returned struct

```
/* b() computes a structure-type result and returns it */
STRUCT b(int i, int j)
{
    ...
    return result;
}

void a(...)
{
    ...
    value = b(i, j);
}
```

In [Example 9-2](#), if the result type is no larger than 8 bytes, `b()` returns its result in r2 and r3.

If the return type is larger than 8 bytes, the Nios II DPX C compiler treats this program as if `a()` had passed a pointer to `b()`. [Example 9-3](#) shows how the Nios II DPX C compiler sees the code in [Example 9-2](#).

Example 9-3. Returned struct is Larger than 8 Bytes

```
void b(STRUCT *p_result, int i, int j)
{
    ...
    *p_result = result;
}

void a(...)
{
    STRUCT value;
    ...
    b(*value, i, j);
}
```

DWARF-2 Definition

Registers `r0` through `r31` are assigned numbers 0 through 31 in all DWARF-2 debugging sections.

Object Files

Nios II DPX MTP object file headers contain Nios II DPX MTP-specific values as shown in [Table 9-22](#).

Table 9-22. Nios II DPX MTP-Specific ELF Header Values

Member	Value
<code>e_ident[EI_CLASS]</code>	<code>ELFCLASS32</code>
<code>e_ident[EI_DATA]</code>	<code>ELFDATA2LSB</code>
<code>e_machine</code>	<code>EM_ALTERA_NIOS2 == 113</code>

Relocation

In a Nios II DPX MTP object file, each relocatable address reference possesses a relocation type. The relocation type specifies how to calculate the relocated address. [Table 9-23](#) lists the calculation for address relocation for each Nios II DPX relocation type. The bit mask specifies where the address is found in the instruction.

Table 9-23. Nios II DPX Relocation Calculation (Part 1 of 2)

Name	Value	Overflow check (1)	Relocated Address R (2)	Bit Mask M	Bit Shift B
<code>R_NIOS2_NONE</code>	0	n/a	None	n/a	n/a
<code>R_NIOS2_S16</code>	1	Yes	$S + A$	<code>0x0000FFFF</code>	0
<code>R_NIOS2_U16</code>	2	Yes	$S + A$	<code>0x0000FFFF</code>	0
<code>R_NIOS2_PCREL16</code>	3	Yes	$((S + A) - 4) - PC$	<code>0x0000FFFF</code>	0
<code>R_NIOS2_CALL26</code>	4	No	$(S + A) >> 2$	<code>0x0FCFFFFF</code>	0

Table 9-23. Nios II DPX Relocation Calculation (Part 2 of 2)

Name	Value	Overflow check (1)	Relocated Address R (2)	Bit Mask M	Bit Shift B
R_NIOS2_IMM5	5	Yes	$(S + A) \& 0x1F$	0x000007C0	0
R_NIOS2_CACHE_OPX	6	n/a	None	n/a	n/a
R_NIOS2_IMM6	7	n/a	None	n/a	n/a
R_NIOS2_IMM8	8	Yes	$(S + A) \& 0xFF$	0x000000FF	0
R_NIOS2_HI16	9	No	$((S + A) \gg 16) \& 0xFFFF$	0x0000FFFF	0
R_NIOS2_LO16	10	No	$(S + A) \& 0xFFFF$	0x0000FFFF	0
R_NIOS2_HIADJ16	11	No	$Adj(S+A)$	0x0000FFFF	0
R_NIOS2_BFD_RELOC_32	12	No	$S + A$	0xFFFFFFFF	0
R_NIOS2_BFD_RELOC_16	13	Yes	$(S + A) \& 0xFFFF$	0x0000FFFF	0
R_NIOS2_BFD_RELOC_8	14	Yes	$(S + A) \& 0xFF$	0x000000FF	0
R_NIOS2_GPREL	15	No	$(S + A - GP) \& 0xFFFF$	0x000000FF	0
R_NIOS2_GNU_VTINHERIT	16	n/a	None	n/a	n/a
R_NIOS2_GNU_VTENTRY	17	n/a	None	n/a	n/a
R_NIOS2_UJMP	18	No	$((S + A) \gg 16) \& 0xFFFF$, $(S + A + 4) \& 0xFFFF$	0x0000FFFF	0
R_NIOS2_CJMP	19	No	$((S + A) \gg 16) \& 0xFFFF$, $(S + A + 4) \& 0xFFFF$	0x0000FFFF	0
R_NIOS2_CALLR	20	No	$((S + A) \gg 16) \& 0xFFFF$, $(S + A + 4) \& 0xFFFF$	0x0000FFFF	0
R_NIOS2_ALIGN	21	n/a	None	n/a	n/a
R_NIOS2_S12	41	Yes	$S+A$	0x000000FF	0
R_NIOS2_U12	42	Yes	$S+A$	0x000000FF	0
R_NIOS2_BMX_LSB	43	Yes	$S+A$	0x0000001F	0
R_NIOS2_BMX_MSB	44	Yes	$S+A$	0x000003E0	5
R_NIOS2_PCREL14_S2	45	Yes	$((S + A) - 4) - PC \gg 2$	0x000000FF	0
R_NIOS2_U20	46	Yes	$S+A$	0x000FFFFF	0
R_NIOS2_HI20	47	No	$((S + A) \gg 12) \& 0xFFFFF$	0x000FFFFF	0
R_NIOS2_LO12	48	No	$S+A$	0x000000FF	0
R_NIOS2_HIADJ20	49	No	$Adj20(S+A)$	0x000FFFFF	0
R_NIOS2_ILLEGAL	22	n/a	None	None	None

Notes to Table 9-23:

(1) For relocation types where no overflow check is performed, the relocated address is truncated to fit the instruction.

(2) Expressions in this column use the following conventions:

- n S: Symbol address
- n A: Addend
- n PC: Program counter
- n GP: Global pointer
- n $Adj(X)$: $((X \gg 16) \& 0xFFFF) + ((X \gg 15) \& 0x1) \& 0xFFFF$
- n $Adj20(X)$: $((X \gg 12) \& 0xFFFFF) + ((X \gg 11) \& 0x1) \& 0xFFFFF$
- n BA: The base address at which a shared library is loaded

With the information in [Table 9-23](#), any Nios II DPX MTP instruction can be relocated by manipulating it as an unsigned 32-bit integer, as follows:

$$Xr = ((R \ll B) \& M \mid (X \& \sim M));$$

where:


- R is the relocated address, calculated as shown in [Table 9-23](#)
- B is the bit shift shown in [Table 9-23](#)
- M is the bit mask shown in [Table 9-23](#)
- X is the original instruction
- Xr is the relocated instruction

Development Environment

The following symbols are defined:

- `__nios2`
- `__nios2__`
- `__NIO2`
- `__NIO2__`
- `__nios2_big_endian`
- `__nios2_big_endian__`
- `__nios2_6b`
- `__nios2_6b__`

This chapter provides a complete reference of all available commands, options, and settings for the Nios II SBT, as it applies to Nios II DPX software development. This reference is useful for)developing your own software projects, packages, or device drivers.

 Before using this chapter, read [Chapter 7, Getting Started from the Command Line](#), and familiarize yourself with the parts of [Chapter 8, Understanding the Nios II DPX Board Support Package](#) that are relevant to your tasks.

This chapter includes the following sections:

- “Nios II Software Build Tools Utilities” on page 10–1
- “Settings” on page 10–28
- “Application and User Library Makefile Variables” on page 10–39
- “Tcl Commands” on page 10–42
- “Path Names” on page 10–84

Nios II Software Build Tools Utilities

The build tools utilities are an entry point to the Nios II SBT. Everything you can do with the tools, such as specifying settings, creating makefiles, and building projects, is made available by the utilities.

All Nios II 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. Table 10-1 shows the logging levels supported. At each level, the utilities report the status as listed under **Description**. Each level includes the messages from all lower levels.

Table 10-1. Nios II SBT Logging Levels

Logging Level	Description
silent (lowest)	No information is provided except for errors and warnings (sent to <code>stderr</code>).
default	Minimal information is provided (for example, start and stop operation of SBT phases).
verbose	Detailed information is provided (for example, lists of files written).
debug (highest)	Debug information is provided (for example, stack backtraces on errors). This level is for reporting problems to Altera.

Table 10-2 shows the command-line options used to select each logging level. Only one logging level is possible at a time, so these options are all mutually exclusive.

Table 10-2. Selecting Logging Level

Command-Line Option	Logging Level	Comments
none	default	If there is no command-line option, the default level is selected.
<code>--silent</code>	silent	Selects silent level of logging.
<code>--verbose</code>	verbose	Selects verbose level of logging.
<code>--debug</code>	debug	Selects debug level of logging.
<code>--log <fname></code>	debug	All information is written to <code><fname></code> in addition to being sent to the <code>stdout</code> and <code>stderr</code> devices.

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.

For more information about settings values and formats, refer to “Settings” on page 10-28.

Utility and Script Summary

The following command-line utilities and scripts are available:

- “nios2-app-generate-makefile”
- “nios2-bsp-create-settings” on page 10-6
- “nios2-bsp-generate-files” on page 10-8
- “nios2-bsp-query-settings” on page 10-9
- “nios2-bsp-update-settings” on page 10-11
- “nios2-lib-generate-makefile” on page 10-13
- “nios2-bsp-editor” on page 10-15
- “nios2-app-update-makefile” on page 10-16
- “nios2-lib-update-makefile” on page 10-19
- “nios2-swexample-create” on page 10-22
- “nios2-elf-insert” on page 10-23
- “nios2-elf-query” on page 10-24
- “nios2-bsp-console” on page 10-27

nios2-app-generate-makefile

Usage

```
nios2-app-generate-makefile [--app-dir <directory>]
  --bsp-dir <directory> [--debug]
  [--elf-name <filename>] [--extended-help] [--help]
  [--log <filename>] [--no-src] [--set <name> <value>]
  [--silent] [--src-dir <directory>]
  [--src-files <filenames>] [--src-rdir <directory>]
  [--use-lib-dir <directory>] [--verbose]
  [--version]
```

Options

- `--app-dir <directory>`: Directory to place the application makefile and executable and linking format file (.elf). If omitted, it defaults to the current directory.
- `--bsp-dir <directory>`: Specifies the 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.
- `--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.
- `--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.

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. If no command-line arguments are specified, this command returns an exit value of 1 and sends a help message to `stderr`.

For more details about this command, use the `--extended-help` option to display comprehensive usage information.

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>]
  [--librarian-path <directory>] [--log <filename>]
  [--script <filename>] [--set <name> <value>]
  --settings <filename> [--silent]
  --sopc <filename> --type <OS name> [--type-version <version>]
  [--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.
- **--cmd <tcl command>**: Runs the specified Tcl command. Multiple --cmd options are allowed.
- **--cpu-name <cpu name>**: The name of the Nios II DPX MTP that the BSP supports. Optional for a single-processor system.
- **--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.
- **--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® II 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>**: 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 SOPC Information File (**.sopcinfo**) used to create the BSP.
- `--type <OS name>`: BSP type. Set 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.

For more details about this command, use the `--extended-help` option to display comprehensive usage information.

Example

```
nios2-bsp-create-settings --settings my_settings.bsp --sopc \  
  ../my_sopc.sopcinfo --type lwhal --script default_settings.tcl
```

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 Nios II 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.

For more details 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.

For more details about this command, use the `--extended-help` option to display comprehensive usage information.

nios2-bsp-update-settings

Usage

```
nios2-bsp-update-settings [--bsp-dir <directory>]  
  [--cmd <tcl command>] [--cpu-name <cpu name>]  
  [--debug] [--extended-help] [--help] [--jdi <filename>]  
  [--librarian-factory-path <directory>]  
  [--librarian-path <directory>] [--log <filename>]  
  [--script <filename>] [--set <name> <value>]  
  --settings <filename> [--silent]  
  [--sopc <filename>] [--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.
- **--cmd <tcl command>**: Run the specified Tcl command. Multiple --cmd options are allowed.
- **--cpu-name <cpu name>**: The name of the Nios II DPX MTP that the BSP supports. This argument is useful if the hardware design contains multiple Nios II DPX MTPs. 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 II 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 DPX MTP **`.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`.

For more details 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.

For more details about this command, use the `--extended-help` option to display comprehensive usage information.

nios2-bsp-editor

Usage

```
nios2-bsp-editor [--extended-help]
  [--fontsize <point size>] [--help]
  [--librarian-factory-path <directory>]
  [--librarian-path <directory>] [--log <filename>]
  [--settings <filename>] [--version]
```

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 Nios II DPX MTP designs.

For more details 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]
  [--extended-help] [--force] [--get <name>] [--get-all]
  [--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
- **--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`.

For more details about this command, use the `--extended-help` option to display comprehensive usage information.



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.

nios2-lib-update-makefile

Usage

```
nios2-lib-update-makefile --lib-dir <directory>
  [--add-lib-dir <directory>] [--add-public-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-src-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.

For more details about this command, use the `--extended-help` option to display comprehensive usage information.



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.

nios2-swexample-create

Usage

```
nios2-create-swexample [--name] --sopc-dir --type [--list] [--app-dir]
  [--bsp-dir] [--no-app] [--no-bsp] [--elf-name] [--describe]
  [--describeX] [--describeAll] [--search] [--help] [--silent]
  [--version]
```

Options

- **--name**: Specify the name of the software project to create.
- **--sopc-dir** Specify the hardware design directory. Required.
- **--type**: Specify the software example design template type. Required.
- **--list**: List all valid software example design 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: `<Nios II EDS install path>/examples/software`
- **--help**: Displays basic information about this command and its options.
- **--silent**: Do not echo commands.
- **--version**: Print the version number of nios2-create-swexample and exit.

Description

This utility creates a template software example for a given SOPC system.

If no command-line arguments are specified, this command returns an exit value of 1 and sends a help message to stderr.

nios2-elf-insert

Usage

```
nios2-elf-insert <filename> [--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>]
```

Options

- <filename>: the ELF filename
- --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>

If no command-line arguments are specified, this command returns an exit value of 1 and sends a help message to stderr.

nios2-elf-query

Usage

```
nios2-elf-query <filename> [--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]
```

Options

- <filename>: the ELF filename
- --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

If no command-line arguments are specified, this command returns an exit value of 1 and sends a help message to stderr.

nios2-bsp

Usage

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

Options

- **<bsp-type>**: lwhal.
- **<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 LWHAL BSP type for the Nios II DPX MTP

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 LWHAL BSP type for the Nios II DPX MTP.
- **<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_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.

On a preexisting BSP, the value **DONT_CHANGE** prevents associated settings from changing their current value.

 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. Available Tcl commands are listed in “[Tcl Commands for BSP Settings](#)” on page 10-42.
- `--extended-help`: Displays full information about this command and its options. Lists Altera BSP Tcl command help for the `--cmd` and `--script` options
- `--gui`: This option opens a Tcl console for creating, editing, and generating Altera 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. Available Tcl commands are listed in “[Tcl Commands for BSP Settings](#)” on page 10-42.
- `--version`: Displays the version of this command and exits with a zero exit status.

Description

When invoked with no arguments, **nios2-bsp-console** starts an interactive command-line Tcl interpreter for creating, editing, and generating Altera BSPs. Available Tcl commands are listed in “[Tcl Commands for BSP Settings](#)” on page 10-42.

Settings

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 the device drivers and other packages that are included.

Every example in this handbook involves specifying or manipulating settings. 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 Altera-supported device drivers and software packages. It does not include settings for device drivers or software packages furnished by Altera partners or other third parties. If you are using a third-party driver or component, refer to the supplier's documentation.

Settings used in the Nios II 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 (.).

Setting names are case-insensitive.

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 10-3 shows the allowed formats for each setting type.

Table 10-3. Setting Formats

Setting Type	Format When Setting	Format When Getting
boolean	0/1 or false/true	0/1
number	0x prefix for hexadecimal or no prefix for a decimal number	decimal
string	Quoted string Use "none" to set empty string. In the SBT command line, to specify a string value with embedded spaces, surround the string with a backslash-double-quote sequence (\"). For example: <code>--set APP_INCLUDE_DIRS \"lcd board\"</code>	Quoted string

There are several contexts for BSP settings, as shown in [Table 10-4](#).

Table 10-4. BSP Setting Contexts

Setting Context	Description
Altera LWHAL	Settings available with the Altera Lightweight HAL BSP or any BSP based on it.
Altera BSP Makefile Generator	Settings available if using the Altera BSP makefile generator (generates the Makefile and public.mk files). These settings control the contents of makefile variables. This generator is always present in Altera LWHAL BSPs or any BSPs based on the Altera LWHAL.
Altera BSP Linker Script Generator	Settings available if using the Altera BSP linker script generator (generates the linker.x and linker.h files). This generator is always present in Altera LWHAL BSPs or any BSPs based on the Altera LWHAL.

Do not confuse BSP settings with BSP Tcl commands. This section covers BSP settings, including their types, meanings, and legal values. The Tcl commands, which include tools for manipulating the settings, are covered in [“Tcl Commands for BSP Settings” on page 10-42](#).

Overview of Component and Driver Settings

The Nios II 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 Nios II BSP Editor. In the SBT command line flow, use the `enable_sw_package` Tcl command, described in [“Tcl Commands for BSP Settings” on page 10-42](#).

Settings Reference

This section lists all settings for BSPs, software packages, and device drivers.

lwhal.disable_startup_thread_sync

Identifier	ALT_DISABLE_STARTUP_THREAD_SYNC
Type	Boolean definition
Default Value	false
Destination File	system.h
Description	<p>Disables thread synchronization checking on startup.</p> <p>By default, startup code in crt0.S assumes that the <code>.rdata</code> section must be reloaded every time the system is reset. Thread 0 waits until the <code>.rdata</code> section is reloaded before executing initialization code.</p> <p>The <code>lwhal.disable_startup_thread_sync</code> 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.</p>
Restrictions	<p>Do not disable startup thread synchronization under the following circumstances:</p> <ul style="list-style-type: none"> ■ Your code uses initialized global or static variables <p>Your application uses memory management functions such as <code>alt_malloc()</code>, <code>alt_free()</code> and <code>alt_calloc()</code></p>

lwhal.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.
Restrictions	none

lwhal.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 <code>ALT_EXCLUDE_DEFAULT_EXCEPTION</code> in <code>system.h</code> .
Restrictions	none

lwhal.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 <code>.rwdata</code> section. If true, this setting defines the macro <code>ALT_LOAD_COPY_RWDATA</code> in linker.h .
Restrictions	none

lwhal.make.ar

Identifier	AR
Type	Unquoted string
Default Value	nios2-elf-ar
Destination File	BSP makefile
Description	Archiver command. Creates library files.
Restrictions	none

lwhal.make.ar_post_process

Identifier	AR_POST_PROCESS
Type	Unquoted string
Default Value	none
Destination File	BSP makefile
Description	Command executed after archiver execution.
Restrictions	none

lwhal.make.ar_pre_process

Identifier	AR_PRE_PROCESS
Type	Unquoted string
Default Value	none
Destination File	BSP makefile
Description	Command executed before archiver execution.
Restrictions	none

lwhal.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 DPX MTP assembly language source files (.S).
Restrictions	none

lwhal.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.
Restrictions	none

lwhal.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.
Restrictions	none

lwhal.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.
Restrictions	none

lwhal.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.
Restrictions	none

lwhal.make.bsp_cflags_debug

Identifier	BSP_CFLAGS_DEBUG
Type	Unquoted string
Default Value	-g
Destination File	BSP makefile
Description	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. This setting defines the value of BSP_CFLAGS_DEBUG in Makefile.
Restrictions	none

lwhal.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 and C source (.c), files in the BSP. This setting defines the value of BSP_CFLAGS_DEFINED_SYMBOLS in the BSP makefile.
Restrictions	none

lwhal.make.bsp_cflags_optimization

Identifier	BSP_CFLAGS_OPTIMIZATION
Type	Unquoted string
Default Value	-O0
Destination File	BSP makefile
Description	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 nonsequential execution of code while debugging. This setting defines the value of BSP_CFLAGS_OPTIMIZATION in Makefile.
Restrictions	none

lwhal.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 <code>#undef</code> directive in source code. To undefine the macro <code>FOO</code> use the syntax <code>-u FOO</code> in this setting. This is equivalent to <code>#undef FOO</code> in a source file. Note: the syntax differs from macro definition (there is a space, i.e. <code>-u FOO</code> versus <code>-DFOO</code>). Macros defined with this setting are applied to all .S and .c files in the BSP. This setting defines the value of <code>BSP_CFLAGS_UNDEFINED_SYMBOLS</code> in the BSP Makefile.
Restrictions	none

lwhal.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 and .S files. This setting defines the value of <code>BSP_CFLAGS_USER_FLAGS</code> in Makefile.
Restrictions	none

lwhal.make.bsp_cflags_warnings

Identifier	BSP_CFLAGS_WARNINGS
Type	Unquoted string
Default Value	-Wall
Destination File	BSP makefile
Description	C compiler warning level. <code>-Wall</code> is commonly used. This setting defines the value of <code>BSP_CFLAGS_WARNINGS</code> in Makefile.
Restrictions	none

lwhal.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 <code>-I</code> prefix is added by the makefile, therefore you must not include it in the setting value. This setting defines the value of <code>BSP_INC_DIRS</code> in the makefile.
Restrictions	none

lwhal.make.build_post_process

Identifier	BUILD_POST_PROCESS
Type	Unquoted string
Default Value	none
Destination File	BSP makefile
Description	Command executed after BSP built.
Restrictions	none

lwhal.make.build_pre_process

Identifier	BUILD_PRE_PROCESS
Type	Unquoted string
Default Value	none
Destination File	BSP makefile
Description	Command executed before BSP built.
Restrictions	none

lwhal.make.cc

Identifier	CC
Type	Unquoted string
Default Value	nios2-elf-gcc -xc
Destination File	BSP makefile
Description	C compiler command
Restrictions	none

lwhal.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.
Restrictions	none

lwhal.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.
Restrictions	none

lwhal.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.
Restrictions	none

lwhal.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.
Restrictions	none

lwhal.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.
Restrictions	none

lwhal.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.
Restrictions	none

lwhal.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.
Restrictions	none

lwhal.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.
Restrictions	none

lwhal.make.rm

Identifier	RM
Type	Unquoted string
Default Value	rm -f
Destination File	BSP makefile
Description	Command used to remove files when building the <code>clean</code> target.
Restrictions	none

lwhal.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 .

lwhal.thread_stack_size

Identifier	ALT_THREAD_STACK_SIZE
Type	Decimal number
Default Value	The default value of <code>lwhal.thread_stack_size</code> is selected by the default Tcl script launched when a LWHAL BSP is created. <code>lwhal.thread_stack_size</code> is set to 3/4 of the size of the memory region to which the <code>.stack</code> 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 <code>ALT_THREAD_STACK_SIZE</code> in <code>system.h</code> .
Restrictions	none

Application and User Library Makefile Variables

The Nios II 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 lwhal 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** target.




Creating the HDL simulation and onchip memory initialization files increases project build time.

- **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 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 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 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.


 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.


 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.

 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.

 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 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 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 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
- ASFLAGS

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.

Tcl Commands

Tcl commands are a crucial component of the Nios II 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 Nios II 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

[“Settings” on page 10-28](#) describes settings that are available in a Nios II DPX MTP project. This section describes the tools that you use to specify and manipulate these settings.

You manipulate project settings with BSP Tcl commands. The commands in this section are used with the utilities **nios2-bsp-create-settings**, **nios2-bsp-update-settings**, and **nios2-bsp-query-settings**. You can call the Tcl commands directly on a utility command line using the `--cmd` option, or you can put them in a Tcl script, specified with the `--script` option. For details about how to call Tcl commands from utilities, refer to [“Nios II Software Build Tools Utilities” on page 10-1](#).

-  For more information about creating Tcl scripts, refer to “[Specifying BSP Defaults for the Nios II DPX MTP](#)” on page 8–14. This chapter includes a discussion of the default Tcl script, which provides excellent usage examples of many of the Tcl commands described in this section.

The following commands are available to manipulate BSP settings:

- “[add_memory_device](#)” on page 10–45
- “[add_memory_region](#)” on page 10–45
- “[add_section_mapping](#)” on page 10–46
- “[are_same_resource](#)” on page 10–46
- “[delete_memory_region](#)” on page 10–46
- “[delete_section_mapping](#)” on page 10–47
- “[disable_sw_package](#)” on page 10–47
- “[enable_sw_package](#)” on page 10–47
- “[get_addr_span](#)” on page 10–48
- “[get_assignment](#)” on page 10–48
- “[get_available_drivers](#)” on page 10–48
- “[get_available_sw_packages](#)” on page 10–49
- “[get_base_addr](#)” on page 10–49
- “[get_break_offset](#)” on page 10–50
- “[get_break_slave_desc](#)” on page 10–50
- “[get_cpu_name](#)” on page 10–50
- “[get_current_memory_regions](#)” on page 10–51
- “[get_current_section_mappings](#)” on page 10–51
- “[get_default_memory_regions](#)” on page 10–51
- “[get_driver](#)” on page 10–52
- “[get_enabled_sw_packages](#)” on page 10–52
- “[get_exception_offset](#)” on page 10–53
- “[get_exception_slave_desc](#)” on page 10–53
- “[get_fast_tlb_miss_exception_offset](#)” on page 10–53
- “[get_fast_tlb_miss_exception_slave_desc](#)” on page 10–54
- “[get_memory_region](#)” on page 10–54
- “[get_module_class_name](#)” on page 10–55
- “[get_module_name](#)” on page 10–55
- “[get_reset_offset](#)” on page 10–55
- “[get_reset_slave_desc](#)” on page 10–56
- “[get_section_mapping](#)” on page 10–56

- “get_setting” on page 10-56
- “get_setting_desc” on page 10-57
- “get_slave_descs” on page 10-57
- “is_char_device” on page 10-58
- “is_connected_to_data_master” on page 10-58
- “is_connected_to_instruction_master” on page 10-58
- “is_ethernet_mac_device” on page 10-59
- “is_flash” on page 10-59
- “is_memory_device” on page 10-59
- “is_non_volatile_storage” on page 10-60
- “is_timer_device” on page 10-60
- “log_debug” on page 10-60
- “log_default” on page 10-60
- “log_error” on page 10-61
- “log_verbose” on page 10-61
- “set_driver” on page 10-61
- “set_ignore_file” on page 10-62
- “set_setting” on page 10-62
- “update_memory_region” on page 10-63
- “update_section_mapping” on page 10-63
- “add_default_memory_regions” on page 10-63
- “create_bsp” on page 10-64
- “generate_bsp” on page 10-64
- “get_available_bsp_type_versions” on page 10-64
- “get_available_bsp_types” on page 10-65
- “get_available_cpu_architectures” on page 10-65
- “get_available_cpu_names” on page 10-65
- “get_available_software” on page 10-65
- “get_available_software_setting_properties” on page 10-66
- “get_available_software_settings” on page 10-66
- “get_bsp_version” on page 10-66
- “get_cpu_architecture” on page 10-67
- “get_nios2_dpx_thread_num” on page 10-67
- “get_sopcinfo_file” on page 10-67
- “get_supported_bsp_types” on page 10-67

- “is_bsp_hal_extension” on page 10-67
- “is_bsp_lwhal_extension” on page 10-68
- “open_bsp” on page 10-68
- “save_bsp” on page 10-68
- “set_bsp_version” on page 10-68
- “set_logging_mode” on page 10-69

add_memory_device

Usage

`add_memory_device <device name> <base address> `

Options

- `<device name>`: String with the name of the memory device.
- `<base address>`: The base address of the memory device. Hexadecimal or decimal string.
- ``: The size (span) of the memory device. Hexadecimal or decimal string.

Description

This command is provided to define a user-defined external memory device, outside the Nios II DPX 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> `

Options

- `<name>`: String with the name of the memory region to create.
- `<slave_desc>`: String with the slave descriptor of the memory device for this region.
- `<offset>`: String with the byte offset of the memory region from the memory device base address.
- ``: String with the span of the memory region in bytes.

Description

Creates a new memory region for the linker script. This memory region must not overlap with any other memory region and must be within the memory range of the associated slave descriptor. The offset and span are decimal numbers unless prefixed with 0x.

Example

```
add_memory_region onchip_ram0 onchip_ram0 0 0x100000
```

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 ':'.
For example, `altera_hostfs:7.2`.

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).

Example

```
puts [get_break_slave_desc]
```

Returns:

onchip_ram_64_kbytes

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

```
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

`get_default_memory_regions`

Options

None

Description

Returns a sorted list of records representing the default linker script memory regions. The default linker script memory regions are the best guess for memory regions based on the reset address and exception address of the processor associated with the BSP, and all other processors in the system that share memories with the processor associated with the BSP. Each record in the list represents a memory region. Each record is a list containing the region name, associated memory device slave descriptor, offset, and span, in that order.

Example

```
puts [get_default_memory_regions]
```

Returns:

```
{reset onchip_ram0 0 32} {onchip_ram0 onchip_ram0 32 1048544}
```

get_driver**Usage**

```
get_driver <module_name>
```

Options

- *<module_name>*: String with the name of the module instance to query.

Description

Returns the driver name associated with the specified module instance. The format is *<driver name>* followed by a colon and the version (if provided). Returns the string "none" if there is no driver associated with the specified module instance name.

Examples

- Example 1:

```
puts [get_driver jtag_uart]
```

Returns:

```
altera_avalon_jtag_uart_driver:7.2
```

- Example 2:

```
puts [get_driver onchip_ram_64_kbytes]
```

Returns:

```
none
```

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

Description

Returns the slave descriptor associated with the processor fast TLB miss exception address. Only a processor with an MMU has such an exception address.

Example

```
puts [get_fast_tlb_miss_exception_slave_desc]
```

Returns:

```
onchip_ram_64_kbytes
```

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.

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 lwhal.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.

Description

Returns a string describing the BSP setting.

Example

```
puts [get_setting_desc lwhal.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 Nios II DPX MTP.

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_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.

log_verbose

Usage

```
log_verbose <message>
```

Options

- <message>: String with message to log.

Description

Displays a message to the host's stdout when the logging level is verbose or higher.

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 which is compatible with the module 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 BSP type 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.
- `generated`—Specifies that *<file_name>* is a generated top-level BSP file. The list of generated BSP files depends on the BSP type. For a list of generated files associated with LWHAL BSPs, refer to [“Nios II DPX BSP Creation” on page 8–9](#).

set_setting**Usage**

```
set_setting <name> <value>
```

Options

- *<name>*: String with the name of the setting.
- *<value>*: String with the value of the setting.

Description

Sets the value for the specified BSP setting. Legal values for boolean settings are true, false, 1, and 0. Use the keyword `none` instead of an empty string to set a string to an empty value. The `set_setting` command is equivalent to the `--set` command-line option.

Example

```
set_setting lwhal.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.
- ****: 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>
```

Options

- <bsp_type>: BSP type identifier (e.g. hal, ucousii).
- <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 <sopcinfolpath>
```

Options

- <sopcinfolpath>: 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 <sopcinfolpath>
```

Options

- <sopcinfolpath>: 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> <sopcinfolpath>
```

Options

- <bsp_type>: BSP type identifier (e.g. hal, ucousii).
- <sopcinfolpath>: 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. Filters are "all", "drivers", "sw_packages", and "os_elements". 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> <sopcinfolpath>
```

Options

- <software_name>: Name of a software component (e.g. "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).
- <sopcinfolpath>: 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> \  
    <sopcinfolpath>
```

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.
- <sopcinfolpath>: 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> <sopcinfolpath>`

Options

- `<processor_name>`: processor instance name
- `<sopcinfolpath>`: 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 Nios II DPX processor, then return the number of threads it supports. Otherwise return null.

get_sopcinfol_file

Usage

`get_sopcinfol_file`

Description

Returns the path of the BSP specific SOPC Information File.

get_supported_bsp_types

Usage

`get_supported_bsp_types <processor_name> <sopcinfolpath>`

Options

- `<processor_name>`: processor instance name
- `<sopcinfolpath>`: 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

`is_bsp_hal_extension`

Description

Returns a boolean value that indicates whether the BSP instantiated is of a type based on Altera HAL.

is_bsp_lwhal_extension**Usage**

```
is_bsp_lwhal_extension
```

Description

Returns a boolean value that indicates whether the BSP instantiated is of a type based on Altera Lightweight HAL.

open_bsp**Usage**

```
open_bsp <settingsFile>
```

Options

- <settingsFile>: .bsp settings file to open.

Description

Opens an existing BSP.

save_bsp**Usage**

```
save_bsp <settingsFile>
```

Options

- <settingsFile>: .bsp settings file to save BSP to.

Description

Saves a new BSP.

set_bsp_version**Usage**

```
set_bsp_version <version>
```

Options

- <version>: Version of BSP type software element to use.

Description

Sets the version of the BSP operating system software element to a specific value. The value "default" uses the latest version available. If this call is not used, the BSP is created using the 'default' BSP software element version.

set_logging_mode

Usage

```
set_logging_mode <mode>
```

Options

- <mode>: Logging Mode: 'silent', 'default', 'verbose', 'debug'

Description

Sets the verbosity level of the logger. Useful to eliminate tool informative messages

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



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 [Table 10-5](#).

Table 10-5. Callback Properties

Property as specified in <code>set_sw_property</code>	Run time	Scope	Callback Arguments
<code>initialization_callback</code>	Initialization	Component instance	Component instance name
<code>validation_callback</code>	Validation	Component instance	Component instance name
<code>generation_callback</code>	Generation	Component instance	Component instance name, BSP generate target directory, driver BSP subdirectory (1)
<code>class_initialization_callback</code>	Initialization	Component class	Driver class name
<code>class_validation_callback</code>	Validation	Component class	Driver class name
<code>class_generation_callback</code>	Generation	Component class	Driver class name, BSP generate target directory, driver BSP subdirectory (1)

Note to Table 10-5:

(1) The BSP subdirectory into which the driver or package files are copied

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.



Refer to “[Tcl Commands for BSP Settings](#)” on page 10-42 for details about BSP setting Tcl commands.



When a Tcl callback creates a setting, it can specify the value. However, callbacks cannot change the value of a pre-existing setting.

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>
```

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
```

add_module_sw_setting

Usage

```
add_module_sw_setting <setting-name> <setting-type>
```

Options

- *<setting-name>*: Name of the setting to persist in the BSP settings file. This is prepended with the module name associated with this callback script
- *<setting-type>*: Type of the setting to persist in the BSP settings file.

Description

Creates a BSP setting that is associated with a particular instance of hardware module in a SOPC system. The `set_module_sw_setting_property` command is required to set the values for fields pertaining to a BSP software setting definition. This command is only valid for a callback script. A callback script is set in the driver's `*_sw.tcl` file, using the command `set_sw_property callback_source_file <filename>`.

Example

```
add_module_sw_setting MY_FAVORITE_SETTING String
```

add_module_systemh_line**Usage**

```
add_module_systemh_line <macro-name> <macro-value>
```

Options

- `<macro-name>`: Macro to be added to the system.h file for the generated BSP
- `<macro-value>`: Value associated with the macro-name to be added to the system.h file for the generated BSP

Description

This adds a system.h assignment or macro during a driver callback execution. The BSP typically uses this during the generate phase depending on the generator. This command is only valid for a callback script. A callback script is set in the driver's `*_sw.tcl` file, using the command `set_sw_property callback_source_file <filename>`.

Example

```
add_module_systemh_line MY_MACRO "Macro_Value";
```

get_class_peripheral**Usage**

```
get_class_peripheral <instance-name> <irq-number>
```

Options

- `<instance-name>`: Name of EIC module instance to find connected peripheral for.
- `<irq-number>`: IRQ number to locate connected peripheral device

Description

This command is used on an EIC instance callback to obtain a peripheral slave descriptor connected to a specific IRQ port number. This command is only valid for a callback script.

Example

```
get_class_peripheral eic_1 $irq_2;
```

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>
```

Options

- *<irq-number>*: IRQ number to locate connected peripheral device

Description

This command is used on an EIC instance callback to obtain a peripheral slave descriptor connected to a specific IRQ port number. This command is only valid for a callback script.

Example

```
get_module_peripheral 2;
```

get_module_sw_setting_value**Usage**

```
get_module_sw_setting_value <setting-name>
```

Options

- *<setting-name>*: Name of the module software setting to retrieve the value for, as defined by the `add_module_sw_setting` command.

Description

Given a module software setting name, return the setting value. The callback script using this command must be set in the `*_sw.tcl` file using the following command:

```
set_sw_property callback_source_file <filename>
```

You can use this command in a generation or validation callback to retrieve the current value of a setting created in an initialization callback.

Example

```
puts [get_module_sw_setting_value MY_SETTING]
```

Returns:

```
"My setting value"
```

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. To manipulate settings on existing software packages and device drivers, refer to [“Settings” on page 10-28](#) and [“Tcl Commands for BSP Settings” on page 10-42](#).

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 [“Specifying BSP Defaults for the Nios II DPX MTP” on page 8-14](#).

The following commands are available for device driver and software package creation:

- [“add_sw_property” on page 10-77](#)
- [“add_sw_setting” on page 10-79](#)
- [“create_driver” on page 10-81](#)
- [“create_sw_package” on page 10-82](#)
- [“set_sw_property” on page 10-82](#)

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 DPX MTP 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.
- **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 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 LWHAL/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 BSP types that the driver or software package supports. Specify **LWHAL** if the software supports the Altera LWHAL, or BSP types that extend it. If your software is BSP type-neutral and works on multiple LWHAL-based BSP types, state **LWHAL** only. If your software or driver contains code that depends on a particular BSP type, state compatibility with that BSP type only, but not **LWHAL**. The name of another BSP type to support must match the name of the BSP type exactly. This BSP type 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 BSP type, in its **create_os** command.
When you create a BSP with an BSP type that extends **LWHAL** and the BSP tools

select a driver for a particular hardware module, precedence is given to drivers which state compatibility with a that specific BSP type (OS) before a more generic driver stating LWHAL compatibility.

This property is only available for device drivers and software packages. This property must be set to at least one BSP type.

- `excluded_lwhal_source`—Specifies a file to exclude from the a BSP generated with an BSP type that extends LWHAL. The value is the path to a BSP file to exclude, with respect to the BSP root. This property is optional.

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** 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` generators.
 - **boolean**—Defines a macro or makefile variable 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 generators.
 - **character**—Defines a macro with a single character with single quotes around the character. Example: `#define DELIMITER ':'`. This type supports the `system_h_define` destination.
 - **decimal_number**—Decimal numbers define a macro or makefile variable with an unquoted decimal (integer) number. Example: `#define NUM_COPROCESSORS 3`. This type supports all destinations.
 - **double**—Double numbers have a macro name and setting value in the destination file including decimal point. Example: `#define PI 3.1416`. This type supports the `system_h_define` destination.
 - **float**—Float numbers have a macro name and setting value in the destination file including decimal point and `f` character. Example: `#define PI 3.1416f`. This type supports the `system_h_define` destination.
 - **hex_number**—Hex numbers have a macro name and setting value in the destination file with `0x` prepended to the value. Example: `#define LCD_SIZE 0x1000`. This type supports the `system_h_define` destination.
 - **quoted_string**—Quoted strings always have the macro name and setting value added to the destination files. In the destination, the setting value is enclosed in quotation marks. Example:
`#define DFLT_ERR "General error"`
 If the setting value contains white space, you must also place quotation marks around the value string in the Tcl script.
 This type supports the `system_h_define` destination.
 - **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 (`" "`).

create_driver

Usage

```
create_driver <name>
```

Options

- **<name>**: Name of device driver.

Description

This command creates a new device driver instance available for the Nios II 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 BSP types, 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_sw_package

Usage

```
create_sw_package <name>
```

Options

- `<name>`: Name of the software package.

Description

This command creates a new software package instance available for the Nios II 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 BSP types, 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 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.
- **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: Altera Nios II DPX MTP 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.
- **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

Path Names

There are some restrictions on how you can specify file paths when working with the Nios II 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 Nios II 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 Nios II 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 Nios II SBT accepts both relative and absolute path names.

Table 10-6 shows the supported path name formats for each platform, for Nios II SBT utilities and makefiles.

Table 10-6. Path Name Format Support

Context	Formats supported on Linux (1)	Formats supported on Windows with Cygwin
Utilities and scripts	Linux	<ul style="list-style-type: none"> ■ Quoted Windows (2) ■ Mixed (2) ■ Escaped Windows (2) ■ Cygwin
Makefiles	Linux	<ul style="list-style-type: none"> ■ Mixed (3) ■ Cygwin (3)
Notes to Table 10-6: (1) These rules apply to any Unix-like platform. (2) These rules apply to other Unix-like shells running on Windows. The Nios II Command Shell, provided with the Nios II EDS, is based on Cygwin. Examples in this chapter are designed for the Nios II Command Shell. (3) The build tools automatically convert path names to Cygwin format		

Object File Directory Tree

The makefile created by the Nios II 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 object file directory structure is illustrated in “Nios II DPX BSP Creation” on page 8-9.

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 Nios II 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 Nios II 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 Nios II SBT places the corresponding object code in

```
obj/cygdrive/c/dev/app/special/tools.o
```


This chapter provides additional information about the document and Altera.

Document Revision History

The following table shows the revision history for this document.

Date	Version	Changes
May 2011	2.0	Updated for ACDS v11.0
December 2010	1.0	Initial release

How to Contact Altera

To locate the most up-to-date information about Altera products, refer to the following table.

Contact (1)	Contact Method	Address
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	custrain@altera.com
Product literature	Website	www.altera.com/literature
Non-technical support (General) (Software Licensing)	Email	nacomp@altera.com
	Email	authorization@altera.com









Note to Table:

(1) You can also contact your local Altera sales office or sales representative.

Typographic Conventions

The following table shows the typographic conventions this document uses.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Indicate command names, dialog box titles, dialog box options, and other GUI labels. For example, Save As dialog box. For GUI elements, capitalization matches the GUI.
bold type	Indicates directory names, project names, disk drive names, file names, file name extensions, software utility names, and GUI labels. For example, \qdesigns directory, D: drive, and chiptrip.gdf file.
<i>Italic Type with Initial Capital Letters</i>	Indicate document titles. For example, <i>Stratix IV Design Guidelines</i> .
<i>italic type</i>	Indicates variables. For example, $n + 1$. Variable names are enclosed in angle brackets (< >). For example, <file name> and <project name>.pof file.

Visual Cue	Meaning
Initial Capital Letters	Indicate keyboard keys and menu names. For example, the Delete key and the Options menu.
“Subheading Title”	Quotation marks indicate references to sections within a document and titles of Quartus II Help topics. For example, “Typographic Conventions.”
Courier type	Indicates signal, port, register, bit, block, and primitive names. For example, <code>data1</code> , <code>tdi</code> , and <code>input</code> . The suffix <code>n</code> denotes an active-low signal. For example, <code>resetn</code> . Indicates command line commands and anything that must be typed exactly as it appears. For example, <code>c:\qdesigns\tutorial\chiptrip.gdf</code> . Also indicates sections of an actual file, such as a Report File, references to parts of files (for example, the AHDL keyword <code>SUBDESIGN</code>), and logic function names (for example, <code>TRI</code>).
	An angled arrow instructs you to press the Enter key.
1., 2., 3., and a., b., c., and so on	Numbered steps indicate a list of items when the sequence of the items is important, such as the steps listed in a procedure.
	Bullets indicate a list of items when the sequence of the items is not important.
	The hand points to information that requires special attention.
	A question mark directs you to a software help system with related information.
	The feet direct you to another document or website with related information.
	A caution calls attention to a condition or possible situation that can damage or destroy the product or your work.
	A warning calls attention to a condition or possible situation that can cause you injury.
	The envelope links to the Email Subscription Management Center page of the Altera website, where you can sign up to receive update notifications for Altera documents.