Upgrading Nios Processor Systems to the Nios II Processor

July 2006 - ver 1.1 Application Note 350

Overview

The purpose of this document is to guide you through the process of migrating to the Nios® II CPU in an existing embedded system with the Nios embedded processor. This document discusses all necessary hardware and software changes to use the Nios II CPU, as well as optional changes that can be made to further enhance system performance and functionality.

Audience

This document is intended for both hardware and software developers who have used the Altera® Nios development kit and have created one or more functioning designs with the first-generation Nios processor.

Software developers should note that this document discusses topics such as:

- Minimal C and assembly source code modifications
- GNU tool chain behavior, such as the treatment of the `volatile` keyword and its effect on compiler behavior
- Software development tool flow for the Nios processor (such as the use of integrated development environments (IDE), and command line tools such as `nios-build`, `nios-run`, `nios-debug`, etc.)

Hardware developers should note that this document discusses topics such as:

- Use of SOPC Builder (adding and removing components in a design)
- Possible interrupt request (IRQ) reassignments
- Possible modification of any previously-designed custom instruction hardware
- Simulation using an RTL simulator such as ModelSim

Readers who may not be proficient in the above topics are encouraged to review documents such as the *Nios II Hardware Development Tutorial* and the online *Nios II Software Development Tutorial* or other relevant Altera® documentation to re-familiarize themselves with the above bulleted topics.

Throughout this document it is assumed that you have an existing first-generation Nios system that is functional and that you have all SOPC Builder and Quartus II project files necessary to successfully recompile your existing system.

Before You Begin

Before beginning you should ensure that the following tools are installed on the computer where you develop with Nios/Nios II:

- Quartus II 4.0 Service Pack 1, or higher
- SOPC Builder 4.0, or higher (included with Quartus II Service Pack 1)
- Nios II Development Kit, 1.0, or higher
- (*Optional*) ModelSim® Altera 5.7e or higher, or ModelSim PE,SE,EE

Additionally, it is assumed that you have basic familiarity of the Nios II processor and the contents of the following documents:

- Nios II Development Kit Getting Started Guide
- Nios II Hardware Development Tutorial
- Nios II IDE Software Development Tutorials (online)

Introduction

The Nios development kit and embedded processor have been adopted by engineering teams worldwide in part because of its ease of use in development and implementation of system-on-a-programmable-chip (SOPC) designs.

Nios II represents the next revolutionary step in embedded design. Compared to the first-generation Nios processor it delivers higher performance, lower FPGA resource utilization, closer integration with real time operating systems (RTOS) such as Micrium MicroC/OS-II, and the Nios II IDE. To make these features possible, the Nios II processor introduces significant architectural changes in the microprocessor core, compiler and tool chain, and development methodology.

Upgrading a system from the first-generation Nios processor requires certain system changes that will be covered in detail later in this document. These steps include replacing the Nios CPU with a Nios II CPU in SOPC Builder, possible interrupt request (IRQ) reassignments, and upgrading certain other peripherals for Nios II compatibility.

Overview of Migration to Nios II Features

The Nios II processor introduces many new advances, including:

- The Nios II IDE for integrated code development and debugging.
- The hardware abstraction layer (HAL) system library replaces the Nios software development kit (SDK). The HAL provides a robust runtime environment, including support for the familiar ANSI C standard library functions, such as `printf()`, `fopen()`, `gettimeofday()`, etc.
- An instruction set simulator

Migration involves updating your existing application code to take advantage of the new features provided by the Nios II processor.

This upgrade has the following benefits:

- Full use of Nios II IDE, tailored to support the new Altera hardware abstraction layer (HAL) software development flow, allowing automatic creation and management of new software development projects, and close integration of MicroC-OS/II RTOS and LWIP network stack software.
- ANSI C standard library support through the Altera HAL.
- Support of new common flash interface (CFI), JTAG UART, and System ID peripherals, as well as new peripherals in future Nios II releases.
- Support for all Nios II cores, including the **Nios II/f core**.

This upgrade has the following limitations:

- Existing Nios source code will need to be modified using a five-step process (with examples) described in this document.
- On-chip memory contents and off-chip memory simulation model contents are no longer generated in SOPC Builder. You will need to build software in the Nios II IDE to generate memory content files.

Requisite Upgrade Steps

This section describes the steps necessary to upgrade. These steps will largely discuss the minimal hardware changes required to replace existing Nios CPU(s) with the Nios II CPU.

Preliminary Steps — Backup & Open Project Files

Before you begin, you should backup your existing Quartus II project folder containing your existing Nios design, including all sub-directories. This backup will allow you to refer to your design later if needed.

1. Rename or delete any existing **SDK** directories within your Quartus project folder.

2. Open your Quartus project file (**.quartus** or **.qpf**).

   If your project was created prior to Quartus II 4.0, and you have not since opened it, you will be prompted to upgrade your Quartus project and associated files to be compatible with Quartus II 4.0.

3. Open SOPC Builder to view and edit your design containing Nios CPU(s).

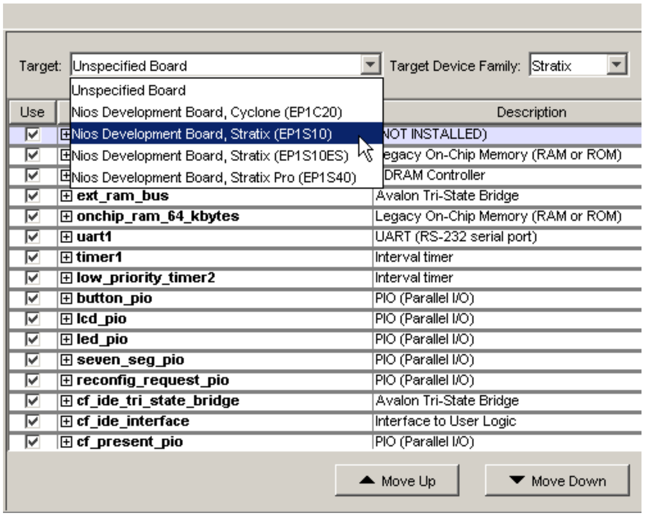
SOPC Builder — Hardware Modifications

Once you have opened your design in SOPC Builder, any existing Nios CPUs need to be replaced with Nios II CPUs. The following steps will guide you through first adding Nios II CPU core(s) to replace existing Nios cores; and then removing previously instantiated Nios core(s). This flow is helpful in copying over various system settings in SOPC Builder. Steps 1 through 9 in this section should be performed to complete this process.

Step 1

Select your target board from the Target pull-down menu. See Figure 4–1.

Figure 4–1. Selecting Target Development Board



Target: Unspecified Board   Target Device Family: Stratix

Unspecified Board
Nios Development Board, Cyclone (EP1C20)
Nios Development Board, Stratix (EP1S10)
Nios Development Board, Stratix (EP1S10ES)
Nios Development Board, Stratix Pro (EP1S40)

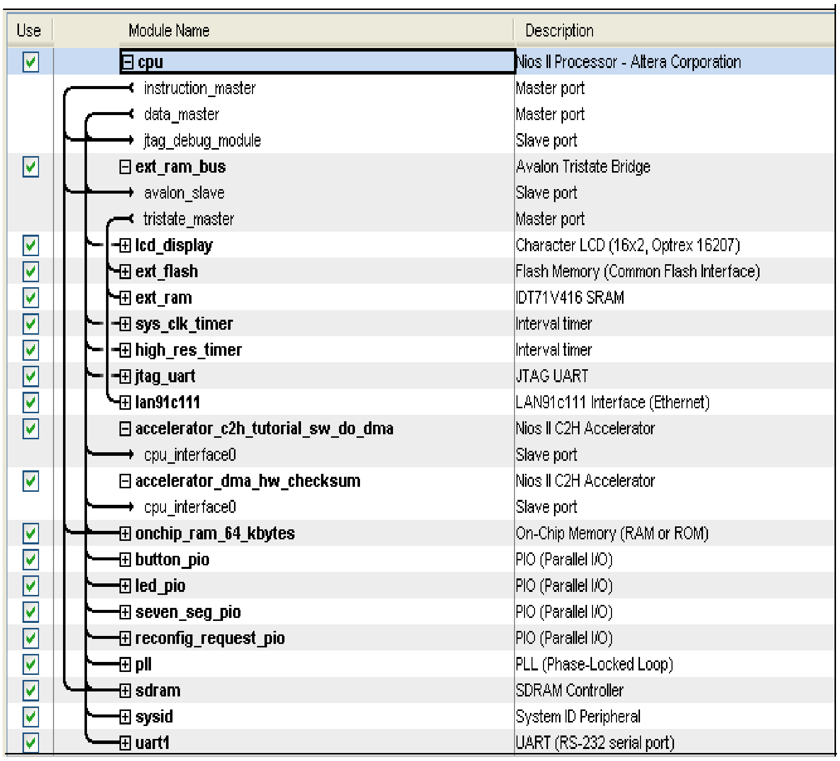
Use		Description
✓	⊞	NOT INSTALLED)
✓	⊞	egacy On-Chip Memory (RAM or ROM)
✓	⊞	DRAM Controller
✓	⊞ **ext_ram_bus**	Avalon Tri-State Bridge
✓	⊞ **onchip_ram_64_kbytes**	Legacy On-Chip Memory (RAM or ROM)
✓	⊞ **uart1**	UART (RS-232 serial port)
✓	⊞ **timer1**	Interval timer
✓	⊞ **low_priority_timer2**	Interval timer
✓	⊞ **button_pio**	PIO (Parallel I/O)
✓	⊞ **lcd_pio**	PIO (Parallel I/O)
✓	⊞ **led_pio**	PIO (Parallel I/O)
✓	⊞ **seven_seg_pio**	PIO (Parallel I/O)
✓	⊞ **reconfig_request_pio**	PIO (Parallel I/O)
✓	⊞ **cf_ide_tri_state_bridge**	Avalon Tri-State Bridge
✓	⊞ **cf_ide_interface**	Interface to User Logic
✓	⊞ **cf_present_pio**	PIO (Parallel I/O)

Step 2

Note the Nios CPU's master port connections to other peripherals in your system. From the **View** menu, select **Show Connections** and expand the Nios CPU to view master port connections to its Avalon slaves as shown in Figure 4–2 on page 4–5. If you have made any arbitration priority assignments between the Nios CPU and other peripherals, choose **Show Arbitration Priorities** from the **View** menu.

You will need to connect the Nios II CPU's master ports to the same Avalon slave ports as the original Nios CPU(s) being replaced.

Figure 4–2. Viewing Avalon Connections for existing Nios CPU



Use	Module Name	Description
✓	⊟ cpu	Nios II Processor - Altera Corporation
	instruction_master	Master port
	data_master	Master port
	jtag_debug_module	Slave port
✓	⊟ ext_ram_bus	Avalon Tristate Bridge
	avalon_slave	Slave port
	tristate_master	Master port
✓	⊞ lcd_display	Character LCD (16x2, Optrex 16207)
✓	⊞ ext_flash	Flash Memory (Common Flash Interface)
✓	⊞ ext_ram	IDT71V416 SRAM
✓	⊞ sys_clk_timer	Interval timer
✓	⊞ high_res_timer	Interval timer
✓	⊞ jtag_uart	JTAG UART
✓	⊞ lan91c111	LAN91c111 Interface (Ethernet)
✓	⊟ accelerator_c2h_tutorial_sw_do_dma	Nios II C2H Accelerator
	cpu_interface0	Slave port
✓	⊟ accelerator_dma_hw_checksum	Nios II C2H Accelerator
	cpu_interface0	Slave port
✓	⊞ onchip_ram_64_kbytes	On-Chip Memory (RAM or ROM)
✓	⊞ button_pio	PIO (Parallel I/O)
✓	⊞ led_pio	PIO (Parallel I/O)
✓	⊞ seven_seg_pio	PIO (Parallel I/O)
✓	⊞ reconfig_request_pio	PIO (Parallel I/O)
✓	⊞ pll	PLL (Phase-Locked Loop)
✓	⊞ sdram	SDRAM Controller
✓	⊞ sysid	System ID Peripheral
✓	⊞ uart1	UART (RS-232 serial port)

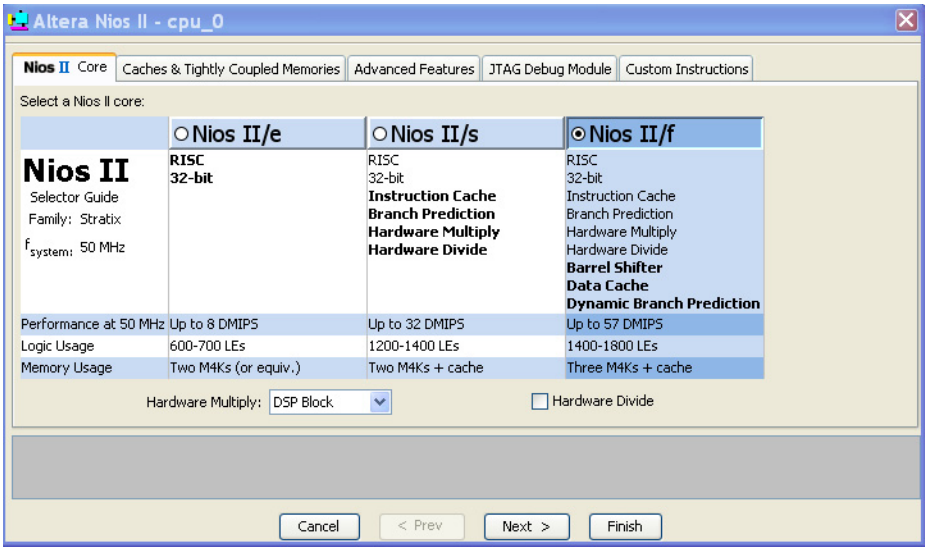
Step 3

Add Nios II CPU(s) to your system to replace each Nios CPU. The Nios II Wizard allows you to select a number of CPU architecture options that vary in performance and logic utilization.

1. In the Nios II core (see Figure 4–3), choose the Nios II core appropriate to your design. If appropriate, select the cache and hardware divider options as well.

If you are not sure of the core that meets your design needs, refer to the "**Nios II Core Implementation Details"** chapter in the *Nios II Processor Handbook* for details on each Nios II CPU core.

Figure 4–3. Nios II Wizard - CPU Selector



Select a Nios II core:

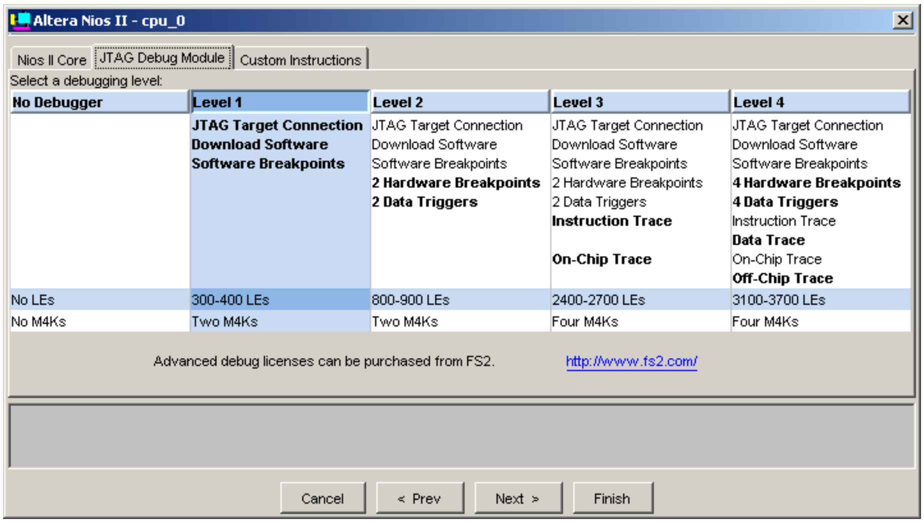
Nios II Selector Guide Family: Stratix f_{system}: 50 MHz	Nios II/e	Nios II/s	[✓ Nios II/f]
	RISC **32-bit**	RISC 32-bit **Instruction Cache** **Branch Prediction** **Hardware Multiply** **Hardware Divide**	RISC 32-bit Instruction Cache Branch Prediction Hardware Multiply Hardware Divide **Barrel Shifter** **Data Cache** **Dynamic Branch Prediction**
Performance at 50 MHz	Up to 8 DMIPS	Up to 32 DMIPS	Up to 57 DMIPS
Logic Usage	600-700 LEs	1200-1400 LEs	1400-1800 LEs
Memory Usage	Two M4Ks (or equiv.)	Two M4Ks + cache	Three M4Ks + cache

Hardware Multiply: DSP Block

☐ Hardware Divide

2. In the **JTAG Debug Module** (see Figure 4–4 on page 4–7), select the debug core applicable to your system's debug needs.

Figure 4–4. Nios II Wizard – Debug Configuration



Select a debugging level:

No Debugger	Level 1	Level 2	Level 3	Level 4
	JTAG Target Connection	JTAG Target Connection	JTAG Target Connection	JTAG Target Connection
	Download Software	Download Software	Download Software	Download Software
	Software Breakpoints	Software Breakpoints	Software Breakpoints	Software Breakpoints
		2 Hardware Breakpoints	2 Hardware Breakpoints	**4 Hardware Breakpoints**
		2 Data Triggers	2 Data Triggers	**4 Data Triggers**
			Instruction Trace	Instruction Trace
				Data Trace
			On-Chip Trace	On-Chip Trace
				Off-Chip Trace
No LEs	300-400 LEs	800-900 LEs	2400-2700 LEs	3100-3700 LEs
No M4Ks	Two M4Ks	Two M4Ks	Four M4Ks	Four M4Ks

Advanced debug licenses can be purchased from FS2. http://www.fs2.com/

Debug levels 1 — 4 are available with the Nios II development kit and are as follows:

- Level 1 — unlimited software breakpoints
- Level 2 — hardware breakpoints and limited software trace capability
- Level 3 — hardware breakpoints and full software trace capability
- Level 4 — additional hardware breakpoints, off-chip instruction, and data trace capability

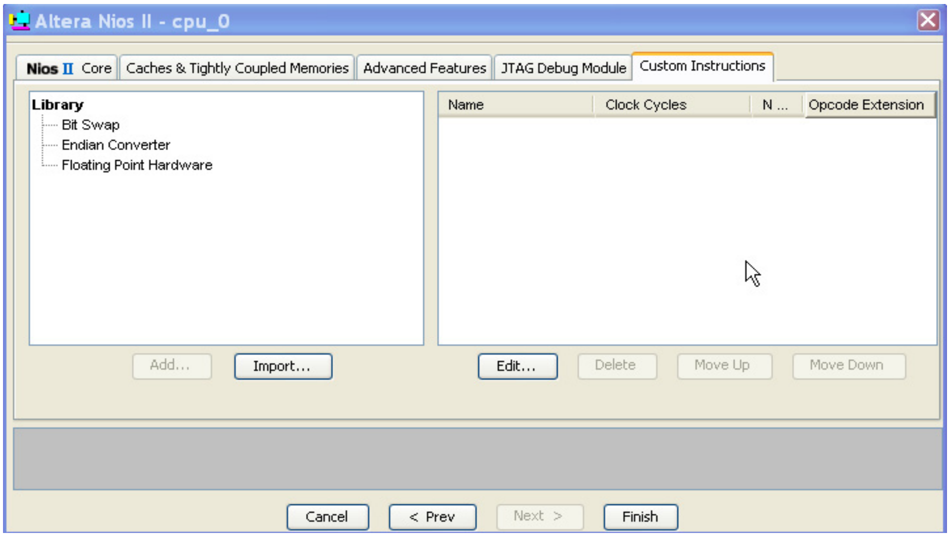
For the full software trace capacity of debug level 3 and all the features of debug level 4, an additional license and/or hardware from First Silicon Solutions is required. For more information, see the First Silicon Solutions website at **www.fs2.com**.

Extensive debugging tools, including trace capabilities and analysis tools are also available from Lauterbach. For more information, see the Lauterbach website at **www.lauterbach.com**.

On the first-generation Nios processor, the on-chip instrumentation (OCI) debug module used for JTAG-based debugging could also be used for host communication. For Nios II systems, JTAG host communication uses the new JTAG UART peripheral included with the Nios II development kit.

3. If necessary, re-add any custom instructions to your Nios II CPU that were present in your previous Nios CPU (see Figure 4–5).

Figure 4–5. Nios II Wizard – Custom Instructions



Refer to the *Nios II Custom Instruction User Guide* for details on upgrading your existing Nios custom instructions.

4. Once you have configured each Nios II CPU core, click **Finish** to add the CPU core to your system.

   If your Nios II CPU has the JTAG debug core enabled, the CPU core will contain an Avalon slave port. Compared to the first-generation Nios CPU, the debug core's addressable span has increased to `0x800`. If you notice any address conflicts with other peripherals, re-assign the Nios II CPU base address, or use the SOPC Builder **Auto-Assign Base Addresses** feature (accessible by right-clicking on the list of peripherals in the system) to resolve any address conflicts.

   This change will not alter the behavior of automatically-generated software because references to peripherals and memory are based on symbols rather than fixed addresses.

Step 4

After adding each Nios II CPU to your system, give each CPU a unique name in the table of components in your design.

Step 5

Reassign IRQ numbers to slave peripherals as necessary. IRQ settings for all slave peripherals must be in the range of 0-31, or **NC** for no connect.

The Nios II CPU's method of handling interrupts has changed significantly. Previously, a Nios CPU had 64 vectored interrupts, numbered 0-63, of which 0-15 were reserved for the system. Nios II CPUs allow a maximum of 32 vectored interrupts, ranging from 0-31. `IRQ 0` remains the highest priority interrupt when using the built-in Nios II software interrupt support.

For additional details on Nios II interrupt handling, see the "Exception Handling" chapter in the *Nios II Software Developer's Handbook*.

Step 6

Update memory components (on-chip memory, SRAM, and Flash). Note that these changes are not binding and can be changed if you re-generate the system.

1. You must replace existing memory components (on-chip memory and SRAM) with updated components that do not include memory contents. To do so, delete each existing memory peripheral from the system and re-add the component from the pool of SOPC Builder peripherals. You may elect to first add a replacement memory component and then delete the old one. Ensure that the same memory settings are used.

2. Delete any peripherals in your system used only for the GERMS boot monitor.

3. Replace flash memory components. Delete existing flash memory instantiations from your system and use the **Flash Memory (Common Flash Interface)** component from SOPC Builder to add replacement flash memory interfaces.

Refer to the "Common Flash Interface Controller Core with Avalon Interface" in the *Nios II Processor Handbook* for details on the new flash interface.

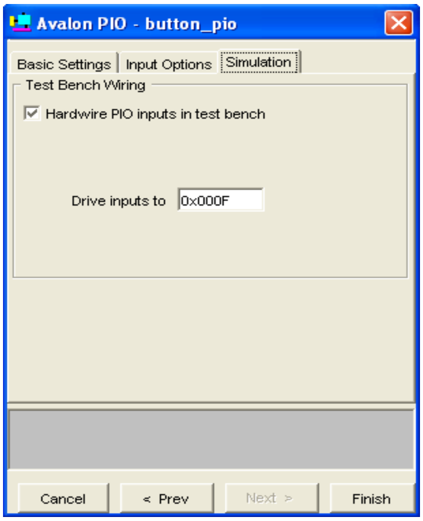
Memory peripheral replacement is necessary because memory contents, whether actual contents for on-chip memory, or simulation model contents for off-chip memory, are now generated by the Nios II IDE software build process. As a result, the SOPC Builder components that define the various memory interfaces have been updated.

It is no longer necessary to have a boot monitor present in your system. See "Overview of Migration to Nios II Features" on page 4–2. New utilities included with the Nios II IDE allow you to send code to the CPU and begin execution, as well as program flash, without the use of a GERMS monitor. Unless explicitly required, you may delete the on-chip boot memory present in your previous system to save FPGA memory resources.

Step 7

Double-click on any existing PIO peripherals in your system. Specify simulation stimulus in the **Stimulation** tab as shown in Figure 4–6.

Figure 4–6. PIO Simulation Settings



☑ Hardwire PIO inputs in test bench

Drive inputs to 0x000F

Step 8

Re-establish the Avalon interconnections that were present before removing the old Nios CPU(s) and any memory peripherals. During this step, re-enter any arbitration priorities settings from "Step 2" and apply them to the new Nios II Avalon master ports.

SOPC Builder will automatically re-connect many Avalon interconnections. Typically, the Nios II instruction master will be connected to all memory slaves, and the data master will be connected to all Avalon slaves in the system. You may need to add/remove connections for your system's architecture.

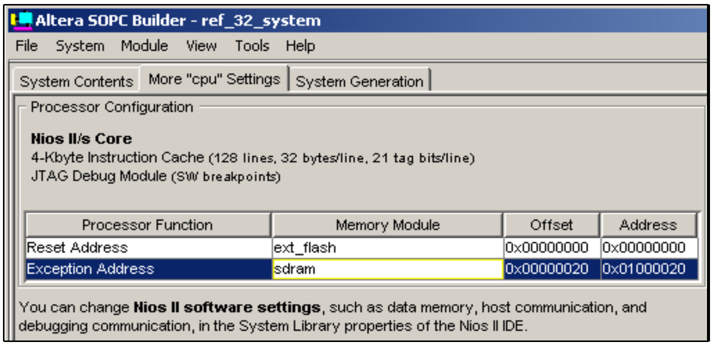
Step 9

To remove the previous Nios CPU(s) from your system, select the CPU and then click **Delete**, or right-click on the CPU and click **Delete**.

SOPC Builder – Additional Nios II CPU Settings

After replacing the CPU and any other peripherals in your system, proceed to the SOPC Builder tab **More <CPU Name> Settings** (see Figure 4–7). There will be a separate tab for each Nios II CPU in your system.

Figure 4–7. Nios II CPU Settings for Reset & Exception Locations



Nios II/s Core
4-Kbyte Instruction Cache (128 lines, 32 bytes/line, 21 tag bits/line)
JTAG Debug Module (SW breakpoints)

Processor Function	Memory Module	Offset	Address
Reset Address	ext_flash	0x00000000	0x00000000
Exception Address	sdram	0x00000020	0x01000020

You can change **Nios II software settings**, such as data memory, host communication, and debugging communication, in the System Library properties of the Nios II IDE.

1. Set the **Reset Location** to the memory device in your system that you wish to boot from. You may optionally specify an offset within this device; this is helpful if booting from a specific area of flash memory containing a boot copier.

CAUTION The Nios II CPU **Reset Location** offset, if any, ***must*** be at a 32-byte (0x20) aligned boundary. Unless your system forces reset code to a custom address during software link, offset 0x0 should be selected.

For systems that have the JTAG debug core enabled, you can set the reset location to an area of blank memory (such as erased flash). Once software is compiled, the **Nios II IDE**, or `nios2-download` command line utility may be used to send executable files to Nios II.

2. Set the **Exception Location** (and offset from base address, if applicable) to the memory device where your interrupt exception handler code will be placed. Typically this is where program memory also resides, but it may be located in a separate, lower-latency memory device such as an on-chip Memory peripheral for lower interrupt latency.

CAUTION The Nios II **Exception Location** offset, ***must*** be at a 32-byte (0x20) aligned boundary (e.g., offsets 0x0, 0x20, 0x40…). Again, the default offset is recommended for most users.

Upon completion of the above steps, the hardware portion of your upgrade is complete. You can safely re-generate your system's HDL in the SOPC Builder **System Generation** tab.

The following sections will discuss the software portion of the upgrade process.

Software Migration to Nios II

This section describes the software upgrade process to fully migrate your application to Nios II and associated features. Before completing this section, ensure that the "Requisite Upgrade Steps " on page 4–3 are completed. Before proceeding, ensure that you have access to the directory where your SOPC Builder project resides. You need not have a hardware target (Nios development or other board) available, but having one accessible will prove helpful in demonstrating runtime features as you progress.

The upgrade process to migrate to full Nios II support involves modifying your existing C/C++ source code (assembly code must be re-written as the Nios and Nios II instruction sets are different) to use the Nios II HAL. The HAL provides access to hardware in the system via the ANSI C standard library functions, such as `printf()` and `fopen()`. The actual source code modifications that need to be performed fall into a five-step process:

1. Replace existing header file inclusion (**excalibur.h**, etc.) with new header files for the system, peripherals, and any standard C libraries required for use of the ANSI C library use in conjunction with the HAL.

2. Change existing API calls to legacy SDK functions with their HAL equivalents (these are usually the `nr_()` functions provided with Nios and associated peripherals in the legacy SDK).

3. (*Optional*) Change generic data types (`unsigned char`, `short`, etc.) to Nios II-specific data types.

4. Replace peripheral registers accesses declared as `structs` and pointers with I/O macros for each register. With the HAL, each peripheral has a simple read and write macro to address each register. Generic I/O macros exist for accessing user-defined peripherals.

5. Update memory buffer access for Nios II data cache operation (if applicable to your system).

Later in this application note, source code examples developed using legacy SDK support will be modified to use the HAL environment. See "Upgrading Your Application Code to Nios II & HAL" on page 4–15.

An Introduction to the HAL Environment

The Altera HAL, is a lightweight runtime environment for software running on the Nios II processor. The HAL provides a simpler device driver interface for applications to communicate with hardware. This interface has been closely integrated with the Newlib standard C library such that devices and files can be accessed using the ANSI C I/O functions.

Throughout this section, references will be made to certain HAL concepts and implementations. Details are documented in the *Nios II Software Developer's Handbook*. It is highly recommended that you refer to this document to assist you in explaining how to migrate code from legacy SDK support to the HAL.

Before proceeding, please see "Overview of the HAL System Library" in the *Nios II Software Developer's Handbook* for a detailed introduction to the HAL.

Definition of System Peripheral Addresses & Parameters

In the legacy Nios SDK, the file **excalibur.h** was automatically generated and contained definitions of each peripheral's address, data structure and interrupt request (IRQ) numbers. With the HAL, system library projects are generated with a single system include file, **system.h**, which defines all system peripherals and base addresses. The content of **system.h** parallels **excalibur.h** in many ways, but with additional information for enabling ANSI C file-descriptor-based access to hardware resources, and additional definitions of the parameterized hardware for each peripheral.

Some of the key differences between **system.h** and **excalibur.h** include the following (In these examples, a fictitious peripheral named `MY_PERIPHERAL` in SOPC Builder is assumed):

- Base address
  - Legacy SDK: `na_my_peripheral`
  - HAL: `MY_PERIPHERAL_BASE`

- IRQ number (if applicable):
  - Legacy SDK: `na_my_peripheral_irq`
  - HAL: `MY_PERIPHERAL_IRQ`

- Name (for file descriptor use):
  - Legacy SDK: n/a
  - HAL: `/dev/my_peripheral`

In addition to the above differences, **system.h** defines specific features of your system that can be very useful in ensuring that your application is running on the correct target platform, or information that may be useful in establishing conditional compilation statements to make your application code generic, and run on multiple Nios II system variants. This information includes (to name a few): FPGA family targeted, CPU architecture, cache sizes, clock speed, the types of peripherals in the system (regardless of their assigned names in SOPC Builder), and a unique system ID (for use with the Altera System ID peripheral) that allow runtime verification of software application versus hardware target.

The **system.h** file does not include several of the features formerly present in **excalibur.h.** They have been replaced with functionality in the HAL in the form of additional C header files. See "Upgrading Your Application Code to Nios II & HAL" on page 4–15 for details. Examples of these include:

- Register bit-masks and `struct` definitions for each peripheral. The register map and any bit-masks for each Altera Avalon peripheral are now defined in a peripheral-specific **regs.h** file included with each peripheral. For example, the Altera Avalon Timer peripheral registers are declared by including **altera_avalon_timer_regs.h**

- Peripheral driver function prototypes. The HAL equivalents for each driver are now prototyped in the appropriate header file for each peripheral. For example, the Altera Avalon UART peripheral's driver prototype are declared by including `altera_avalon_uart.h`.

Software Development Tool & RTOS Use with the HAL

Included with the introduction of Nios II is the Nios II IDE. The Nios II IDE includes functionality that creates and manages software development projects. Each source code project must have an associated library project, which references relevant libraries from the HAL and system description files. For software build, GNU makefiles are automatically generated to build both the user's application code and associated library (HAL).

Developers using existing third-party software development and debug tools may still use the new software functionality provided by Nios II and the full migration path. All source files for system libraries (the HAL environment) are included with the Nios II kit and the HAL may be used to build an application provided that the user or tool being used for development refers to these source files (for example, using a Makefile and GNU make). You should contact your third-party software development tool vendor to see whether updates have been provided to automate Nios II software development.

Developers using third-party real time operating systems (RTOSs) should contact the RTOS vendor for a port supporting Nios II. Included with the Nios II release is a fully-functional port of the Micrium MicroC/OS-II RTOS.

Upgrading Your Application Code to Nios II & HAL

This section discusses the process of converting your existing C/C++ code from legacy Nios SDK support to work with the HAL environment. The examples show several source code excerpts from the `peripheral_test` example application to change it from relying on the legacy Nios SDK to the HAL environment. The `peripheral_test` application is an example program provided with the first-generation Nios development kit's SDK. It is a simple program that exercises several hardware components on the Nios development board, making it perfect for demonstrating the process of converting legacy SDK-based programs to use the HAL.

As stated in "Overview of Migration to Nios II Features" on page 4–2 the process to fully migrate software from legacy Nios to Nios II consists of five steps. In the following section we will discuss these steps in greater detail.

Step 1: Update Included Header Files

The HAL system library projects are generated with a single system include file, **system.h**, which defines all system peripherals, base addresses, etc. The system file is similar to **excalibur.h** or **nios.h** in the legacy SDK; you should first replace these old include files with **system.h**.

Each peripheral in your system will have a corresponding header file that defines the associated HAL device driver routines. These associated routines are documented in the "Developing Device Drivers for the HAL" in the *Nios II Software Developer's Handbook*. Register offsets and I/O access are defined in a second include file. For example, if your system has a UART, **altera_avalon_uart.h** should be included for the HAL device driver access, and **altera_avalon_uart_regs.h** included for register access. If your system has a DMA controller **altera_avalon_dma.h** and **altera_avalon_dma_regs.h** would be included.

In addition, the file **alt_types.h** is included to allow for directly referencing 8, 16, and 32-bit signed and unsigned data types "Step 3: Optional -- Change Generic Data Types to Nios II Specific Data Types" on page 4–18 describes the associated changes.

Code example – Include File Modification:
In the example below, **excalibur.h** is removed and **system.h** is added. Later in the source, the board's LEDs are accessed directly (via a PIO peripheral), so the PIO registers include file is added as well:

Application include files – Legacy SDK:
```
#include "excalibur.h"
#include "peripheral_test.h"
```

Application include files – using Nios II & the HAL:
```
#include "system.h"
#include "peripheral_test.h"
#include "altera_avalon_pio_regs.h"
#include "alt_types.h"
```

Step 2: Replace Legacy API Calls with HAL & ANSI C Calls

Next, existing calls to library routines provided in the legacy SDK are replaced with their HAL and ANSI C counterparts. The most commonly used API calls from the legacy SDK include `nr_installuserisr()`, which installs and enables an interrupt service routine for a particular IRQ number, and UART routines to send and receive characters, such as `nr_uart_rxchar()` and `nr_uart_txchar()`.

The next step is to locate and replace any legacy `nr_` calls with their HAL equivalents. Table 4–2 in "Nios II Legacy SDK vs. the HAL API" on page 4–23 summarizes each legacy SDK API call provided with previous releases of the Nios kit. Refer to it, in conjunction with the *Nios II Software Developer's Handbook* and related peripheral chapters in the *Nios II Processor Reference Manual* to determine the exact changes necessary in your application.

Code example – Pausing Application Execution:
The legacy `nr_delay()` routine pauses program execution by the specified number of milliseconds. In this example, it is replaced with `usleep()`, which pauses for the specified number of microseconds.

Pausing application execution – Legacy SDK:

```
nr_delay(100);// Pause 100 milliseconds
```

Pausing application execution – using Nios II & the HAL:

```
usleep(100000);// Pause 100,000 microseconds
```

Code example – Interrupt Service Routine Setup:
The legacy `nr_installuserisr()` was called to associate an ISR with a specific IRQ number. This is replaced with the HAL equivalent, `alt_irq_register()`:

ISR initialization – Legacy SDK:

```
nr_installuserisr(na_timer1_irq,MyTimerISR,(long)&gC);
```

ISR Initialization – using Nios II & the HAL:

```
alt_irq_register(TIMER1_IRQ, (long)&gC, MyTimerISR);
```

Note that the order of the parameters has changed, with the context being passed before ISR routine. Additionally, `na_timer1_irq`, previously defined in the legacy SDK, is replaced with `TIMER1_IRQ`, defined in **system.h.**

Step 3: Optional -- Change Generic Data Types to Nios II Specific Data Types

To accommodate ANSI C, the HAL includes type definitions specific to Nios II for accessing signed and unsigned 8, 16, and 32 bit values explicitly. Altera recommends using these types in place of `short`, `long`, etc. The benefit of this recommendation is that migration to future architectures will ensure that data accesses which must be 8, 16, or 32 bits continue to operate as expected. These definitions are declared by including the **alt_types.h** file in your application source code. The HAL source code and device drivers have been created following this convention and may be referred to for many examples. Table 4–1 below summarizes the data types and their meanings.

Table 4–1. Nios II Data Type Definitions

Nios II alt_type.h Definition	Data Type Meaning
`alt_8`	Signed 8-bit value
`alt_u8`	Unsigned 8-bit value
`alt_16`	Signed 16-bit value
`alt_u16`	Unsigned 16-bit value
`alt_32`	Signed 32-bit value
`alt_u32`	Unsigned 32-bit value

The ANSI C standard recognizes types `int`, `unsigned int`, and `signed int` (the same as `int`). Other commonly-used data types, such as `signed` and `unsigned char`, `short`, and `long` are machine-dependant. For example, a `long` on one processor architecture may mean 32 bits, while it may mean 64 bits on another. The `int` type is recognized as the data type providing the "best fit" for the architecture in question (32 bits for Nios II).

Step 4: Change Peripheral Register Access to use HAL I/O Macros

Access to Nios peripherals with the legacy SDK was usually performed by declaring a data structure, defined for each peripheral in the system, and assigning the pointer representing the start of that structure to the peripheral's base address. In the HAL environment, this mechanism is being replaced with I/O macros that define each register of a peripheral explicitly. This change is being made, in part so that access to peripheral registers uses Nios II CPU I/O-specific instructions. These instructions bypass the CPU's data cache, if any, and provide for uniform operation across all Nios II CPU cores.

The next step is to remove code that declares a structure, such as `np_uart *uart`. Access to individual structure members, such as `uart->rxdata` will be replaced with an I/O macro defined in **altera_avalon_uart_regs.h**, such as `IORD_ALTERA_AVALON_UART_RXDATA`. The two methods of accessing a peripheral register yield the same result: the register is simply read from or written to as appropriate. In addition, the defined I/O macros account for peripheral register width (byte vs. half-word vs. word).

Code Example – Avalon PIO Peripheral Data Register Access
This example writes to a PIO peripheral's data register using the `np_pio` structure method. This code will be replaced by calling the I/O macro for writing to the PIO's data register:

PIO Register access – Legacy SDK:

```
np_pio *pio = na_seven_seg_pio;
pio->np_piodata = bits;
```

PIO Register access – using Nios II & the HAL:

```
IOWR_ALTERA_AVALON_PIO_DATA(SEVEN_SEG_PIO_BASE, bits);
```

The `IOWR_ALTER_AVALON_PIO_DATA` macro is defined in **altera_avalon_pio_regs.h**, included during "Step 1: Update Included Header Files" on page 4–16. `SEVEN_SEG_PIO_BASE` is defined in **system.h** as the base address of the `seven_seg_pio` peripheral. This was previously defined as `na_seven_seg_pio` in the legacy SDK.

Reading from a peripheral register is accomplished in a similar fashion. This example reads the data register of the `button_pio` peripheral:

PIO register access – Legacy SDK:

```
buttons = pio->np_piodata;
```

PIO register access – Using Nios II & the HAL:

```
buttons = IORD_ALTERA_AVALON_PIO_DATA(BUTTON_PIO_BASE);
```

The above example illustrates direct register access to peripherals included with Nios II. Additional routines provided with the HAL and its ANSI C support may be better suited to your application's needs. For example, the HAL supports UART access with ANSI C `printf()`, `getchar()`, `fopen()`, `and fprintf()`. Use of these routines is highly recommended in favor of direct peripheral register access for most applications because accessing peripheral registers directly in one area of your application may interfere with the HAL driver access elsewhere (for example, in another RTOS thread, or in an interrupt service routine). For more information on the HAL and ANSI C support included with Nios II, refer to the *Nios II Software Developer's Handbook*.

Code Example – User-Defined Peripheral Register Access

Accessing registers in user logic is also straight-forward and done via the `IORD()` and `IOWR()` macros. Example 4–1 illustrates reading from, modifying, and writing back the contents of a 16-bit peripheral register in a user-defined peripheral named `my_peripheral`.

In this example, the register in question is the second from the base address; it is therefore **offset one word into** the peripheral's address space (as per Avalon Native addressing) by adding `0x4` to the base address. When replacing such a register access with the HAL I/O access macros, only the register number needs to be specified; the offset is calculated in the macro automatically.

Example 4–1. User-defined register access – Legacy SDK:

```
unsigned char *reg = na_my_peripheral + 0x4; // Set offset in peripheral
unsigned short temp = 0;

temp = *reg;                                // Read 16-bit register to temp
temp |= 0x8000;                             // Set most significant bit
*reg = temp;                                // Write data back to peripheral
```

Example 4–2. User-Defined peripheral register access – using Nios II & the HAL:

```
alt_u16 temp = 0;

temp = IORD(MY_PERIPHERAL_BASE, 1);         // Read 16-bit register to temp
temp |= 0x8000;                             // Set most significant bit
IOWR(MY_PERIPHERAL_BASE, 1, temp);          // Write data back to peripheral
```

Step 5: Update Memory Buffer Access for Nios II Data Cache Operation

This step is ***required*** for all users of Nios II CPU cores with Data Cache, such as the Nios II/f core. This step is *suggested* for all Nios II CPU core types for consistent operation if the CPU is later upgraded.

Accesses to explicit areas of memory (previously done with `volatile` pointer dereferencing in first-generation Nios systems containing data cache) must be updated to ensure that data written to main memory does not remain in the Nios II data cache. In the first-generation Nios CPU, `volatile` meant *do not cache* – in Nios II, `volatile` follows its ANSI C definition of *do not optimize*.

CAUTION This change is of vital importance in systems where data is shared between multiple masters. For example, a memory buffer that is accessed by a second CPU, DMA controller, or other external master needs assurance that data "touched" by the Nios II CPU is written back to memory after modification. Additionally, memory that may be modified outside of the CPU must not be cached so that the processor fetches the correct data rather than a *private* cached copy.

There are three basic tools available to the user for managing data cache coherency in Nios II. Use of these tools will depend on the requirements of your software (the details of all of these can be found in the Cache Memory chapter of the *Nios II Software Developer's Handbook*. The three basic tools are:

1. `alt_dcache_flush(void* start, alt_u32 len)` — Flushes the specified range of cached memory in the CPU data cache. This will write out cache information to external memory.

2. I/O Read and Write macros for access to memory:
   `IORD_<8|16|32>DIRECT & IOWR_<8|16|32>DIRECT`.

   These macros are designed to load or store 8, 16, or 32 bits of memory while bypassing the data cache. Care must be taken with these macros if some variable or data structure in your code is already cached. In such cases, the data cache should be flushed.

3. The HAL includes non-cacheable `alt_uncached_malloc()` and `alt_uncached_free()` routines to obtain an un-cacheable region of memory which you can then refer to later in your application without worrying about cache. These regions will not be cached and you will suffer a performance loss accordingly.

Use of these tools will be dependent on your application code and system architecture. Therefore no source-code examples are being provided. You should familiarize yourself with general concepts of working with a write-back data cache if you are not already familiar with its operation. However, the following general concepts apply:

- Data accessed only by the Nios II CPU, residing on the stack or heap with other application data will not need any special modification to work with the data cache.
- Accessing specific peripheral registers must be performed with the I/O access macros described in step #4 above. These macros use special I/O instructions in the Nios II CPU to bypass the data cache.

- A buffer that is constructed in your application and then accessed by another master in the system may be created and modified without special modifications for the data cache; however, the data cache must be flushed before any external master accesses the data. This flush allows the performance benefits of using the data cache to construct and manipulate data.
- If the un-cached malloc/free routines are used to obtain a region of memory that is not cached, no other special modifications are required. However, accesses to this region of memory will reduce system performance as the data cache is bypassed.

HAL Migration Summary

Following the above five-step procedure, you may build your code and run on the Nios II CPU making full use of the HAL environment. Refer to the *Nios II Software Developer's Handbook* and related peripheral chapters in the *Nios II Processor Reference Handbook* to determine the exact changes necessary in your application.

Importing Source Code to the Nios II IDE

This section discusses how to import your existing source code into a Nios II IDE project where the source modifications will be made.

It is recommended that you complete the steps in the online *Nios II Software Tutorial* to gain familiarity with Nios II IDE before proceeding.

☞ The instructions below assume that you will edit your source code in Nios II IDE, and as such, the first steps are to import your existing source code to a Nios II IDE project. However, you can use the editor of your choice for the source code modifications described later. You may elect to import your source code to a Nios II IDE project at any time, and continue to edit and develop in the environment of your choice.

1. Create a new Nios II IDE project to build and run your application. You may elect to choose one of the software example templates, such as the "Hello World" or "Hello LED" template to get a ready-to build starting point.

   When creating a project, you will also need a system library project for your hardware system's library files; a HAL system library project is typically created automatically when you create your application project.

2. Copy (using the console, Windows Explorer, etc.) your existing source code and header files into the directory where your project exists. You may keep subdirectories intact, as long as all source and header files are copied into the new project folder.

   When importing source code from a previously-generated Nios SDK directory structure, copy in only *your* source and include files; library files and any previously-generated software from SOPC Builder should not be imported.

3. In Nios II IDE, press **F5** or right-click your project and select **refresh**. to complete the source code import process and automatically add your source code to the managed makefile of the Nios II IDE project.

You can now use the Nios II IDE editor to modify your code and build your application. You may continue to edit the source files in other applications as well.

Nios II Legacy SDK vs. the HAL API

Table 4–2, below, describes API differences between the legacy SDK and the HAL. Refer to it, in conjunction with the *Nios II Software Developer's Handbook* and related peripheral chapters in the *Nios II Processor Reference Handbook* when updating your system's source code.

In addition, the HAL equivalent functions shown, include files (specific to the HAL or ANSI C) will need to be added to your application. Refer to the *Nios II Software Developer's Handbook* and a good ANSI C reference for information on each ANSI C library's contents.

Table 4–2. Nios Legacy SDK API vs. HAL API (Part 1 of 8)

Legacy SDK Call	HAL Equivalent(s)	API Changes & Notes
Altera Nios CPU – Cache Controls		
`nr_icache_invalidate_lines`	`alt_icache_flush`	**Legacy:** address range to invalidate passed in **HAL**: start address and length (bytes) passed in
`nr_dcache_invalidate_lines`	`alt_dcache_flush`	**Legacy**: address range passed in **HAL**: start address and length passed in
Altera Nios CPU -- Interrupt Control		

Table 4–2. Nios Legacy SDK API vs. HAL API (Part 2 of 8)

Legacy SDK Call	HAL Equivalent(s)	API Changes & Notes
`nr_installuserisr`	`alt_irq_register`	**Legacy**: IRQ, ISR address, context **HAL**: IRQ, context, ISR address
`nr_setirqenable`	`alt_irq_disable_all` `alt_irq_enable_all`	**Legacy**: Passed argument controls enable/disable **HAL**: See *Nios II Software Developer's Handbook*.
	Nios "C Stubs"	
`atexit`	atexit	Properly handled in Legacy SDK and in the HAL
`close`	`close`	Did nothing in legacy SDK; properly handled in HAL
`exit`	`exit`	Did nothing in legacy SDK; properly handled in HAL
`fstat`	`fstat`	Did nothing in legacy SDK; properly handled in HAL
`getpid`	`getpid`	Did nothing in legacy SDL; does nothing in HAL
`isatty`	`isatty`	Did nothing in legacy SDK; properly handled in HAL
`kill`	`kill`	Did nothing in legacy SDK; properly handled in HAL
`lseek`	`lseek`	Did nothing in legacy SDK; properly handled in HAL
`read`	`read`	UART-specific in legacy SDK; properly handled in the HAL.
`write`	`write`	UART-specific in legacy SDK; properly handled in the HAL.

Table 4–2. Nios Legacy SDK API vs. HAL API (Part 3 of 8)

Legacy SDK Call	HAL Equivalent(s)	API Changes & Notes
`sbrk`	`sbrk`	Same in both legacy SDK & the HAL.
Altera Nios CPU – Misc.		
`nr_copyrange`	memcpy	Included in ANSI C library
`nr_zerorange`	memset	Included in ANSI C library
`nr_delay`	`usleep`	**Legacy**: Delay in *milliseconds* passed in **HAL**: Delay in *microseconds* passed in
`nr_fprintf`	`fprintf`	**Legacy**: Implementation is UART-specific; properly handled in the HAL.
`nr_jumptoreset`	N/A	**HAL**: not implemented
`nr_callfromreset`	N/A	**HAL**: not implemented
`nr_jumptostart`	N/A	**HAL**: Handled in `crt0.s`
`nr_printf`	`printf`	Included in ANSI C library
`nr_sprintf`	`sprintf`	Included in ANSI C library
First-Generation Altera Nios CPU-Specific – (does not apply to Nios II)		
nr_icache_init	N/A	With the HAL, Nios II instruction cache (if present) is initialized in the default boot-strap code in `crt0.s`
nr_dcache_init	N/A	With the HAL, Nios II instruction cache (if present) is initialized in the default boot-strap code in `crt0.s`
nr_icache_enable	N/A	The Nios II instruction cache (if present) is always enabled

Table 4–2. Nios Legacy SDK API vs. HAL API (Part 4 of 8)

Legacy SDK Call	HAL Equivalent(s)	API Changes & Notes
nr_icache_disable	N/A	The Nios II instruction cache (if present) is always enabled
nr_dcache_enable	N/A	The Nios II instruction cache (if present) is always enabled
nr_dcache_disable	N/A	The Nios II instruction cache (if present) is always enabled
Various (**nios_debug.c**)	N/A	Used with SRAM-debug add-on to the original Nios and APEX™ boards
Various (**nios_gdb_stub.c**)	N/A	Replaced with GDB server and JTAG debug core
Various (**nios_gdb_stub_io.c**)	N/A	Replaced with GDB server and JTAG debug core
Various (**nios_gdb_stub_isr.s**)	N/A	Replaced with GDB server and JTAG debug core
Everything in nios_germs_monitor.s	N/A	GERMS monitor not supported outside of legacy SDK

`nr_getctlreg`	`NIOS2_READ_STATUS`
	`NIOS2_WRITE_STATUS`
	`NIOS2_READ_ESTATUS`
	`NIOS2_READ_BSTATUS`
	`NIOS2_READ_IENABLE`
	`NIOS2_WRITE_IENABLE`
	`NIOS2_READ_IPENDING`
		Direct access via macros to all Nios II CPU registers (`status`, `estatus`, `bstatus`, `ienable`, `ipending`)

Various (**nios_gprof.c**)	N/A	Replaced with GDB server & JTAG debug core
Various (**nios_math.s**)	Handled in `crt0.s`	Handles optional HW multiplication and division. Not user-called
Altera Avalon UART		
HAL device drivers model maps standard ANSI C routines to control the UART peripheral. Refer to the *Nios II Software Development Handbook* & the *Nios II Processor Reference Handbook* for details.		

Table 4–2. Nios Legacy SDK API vs. HAL API (Part 5 of 8)

Legacy SDK Call	HAL Equivalent(s)	API Changes & Notes
`nr_rxchar`	`getchar()`	**Legacy**: nr_rxchar () returns NULL if no RX data waiting **HAL**: getchar() blocks until RX data is valid; STDIN device used
`nr_txchar`	`putchar()`	**HAL**: STDOUT device used
`nr_txcr`	`printf(`*“\n”);*	Included in ANSI C library
`nr_uart_txhex`	`printf(“%2x”, val_8);`	Included in ANSI C library
`nr_uart_txhex16`	`printf(“%4x”, val_16);`	Included in ANSI C library
`nr_uart_txhex32`	`printf(“%8x”, val_32);`	Included in ANSI C library
`nr_uart_txstring`	`printf(“%s”, *str);`	Included in ANSI C library
Old OCI UART/New Altera Avalon JTAG UART		
HAL device drivers model maps standard ANSI C routines to control the JTAG UART peripheral. Refer to the *Nios II Software Development Handbook* & the *Nios II Processor Reference Handbook* for details.		
`nr_jtag_rxchar`	`getchar()`	**Legacy**: nr_rxchar () returns NULL if no RX data waiting **HAL**: getchar() blocks until RX data is valid; STDIN device used
`nr_jtag_txchar`	`putchar()`	**HAL**: STDOUT device used
`nr_jtag_tx_ready`	N/A	First-generation Nios OCI-core specific
`nr_jtag_txcr`	N/A	Included in ANSI C library
`nr_jtag_txhex`	N/A	Included in ANSI C library
`nr_jtag_txhex16`	N/A	Included in ANSI C library
`nr_jtag_txhex32`	N/A	Included in ANSI C library
`nr_jtag_txstring`	N/A	Included in ANSI C library.

Table 4–2. Nios Legacy SDK API vs. HAL API (Part 6 of 8)

Legacy SDK Call	HAL Equivalent(s)	API Changes & Notes
Altera Avalon Timer		
`nr_timer_milliseconds`	`alt_nticks`	HAL gives more control – it can set ticks per second.HAL also includes timestamp (high-resolution) and alarm (callback) timing features
Altera Avalon SPI		
`nr_spi_rxchar`	`alt_avalon_spi_command`	Refer to the “SPI Core with Avalon Interface” chapter in the *Nios II Processor Reference Handbook.*
`nr_spi_txchar`	`alt_avalon_spi_command`	Refer to the “SPI Core with Avalon Interface” chapter in the *Nios II Processor Reference Handbook.*
Altera Avalon DMA		
`nr_dma_copy_1_to_1`	N/A	Refer to the “DMA Core with Avalon Interface” chapter in the *Nios II Processor Reference Handbook*
`nr_dma_copy_1_to_range`	N/A	Refer to the “DMA Core with Avalon Interface” chapter in the *Nios II Processor Reference Handbook*
`nr_dma_copy_range_to_range`	N/A	Refer to the “DMA Core with Avalon Interface” chapter in the *Nios II Processor Reference Handbook*
`nr_dma_copy_range_to_1`	N/A	Refer to the “DMA Core with Avalon Interface” chapter in the *Nios II Processor Reference Handbook*
Altera Avalon PIO		

Table 4–2. Nios Legacy SDK API vs. HAL API (Part 7 of 8)

Legacy SDK Call	HAL Equivalent(s)	API Changes & Notes
`nr_pio_showhex`	N/A	**HAL**: Not implemented
Altera Avalon ASMI		
HAL device driver model maps standard ANSI C routines to control the EPCS Serial Flash controller peripheral. Refer to the *Nios II Software Development Handbook* & the *Nios II Processor Reference Handbook* for details.		
`nr_asmi_protect_region`	N/A	Supported with HAL flash routines. Refer to the *Nios II Software Developer's Handbook*
`nr_asmi_lowest_protected_address`	N/A	Supported with HAL flash routines. Refer to the *Nios II Software Developer's Handbook*
`nr_asmi_read_byte`	N/A	Supported with HAL flash routines. Refer to the *Nios II Software Developer's Handbook*
`nr_asmi_write_byte`	N/A	Supported with HAL flash routines. Refer to the *Nios II Software Developer's Handbook*
`nr_asmi_erase_sector`	N/A	Supported with HAL flash routines. Refer to the *Nios II Software Developer's Handbook*
`nr_asmi_erase_bulk`	N/A	Supported with HAL flash routines. Refer to the *Nios II Software Developer's Handbook*
`nr_asmi_read_buffer`	N/A	Supported with HAL flash routines. Refer to the *Nios II Software Developer's Handbook*

Table 4–2. Nios Legacy SDK API vs. HAL API (Part 8 of 8)

Legacy SDK Call	HAL Equivalent(s)	API Changes & Notes
`nr_asmi_write_buffer`	N/A	Supported with HAL flash routines. Refer to the *Nios II Software Developer's Handbook*
`nr_asmi_past_config`	N/A	Supported with HAL flash routines. Refer to the *Nios II Software Developer's Handbook*

Conclusion

Using this document, you can migrate a first-generation Nios embedded system design to one using the Nios II CPU. The upgrade process involves making hardware and software changes to use the Nios CPU, as well as optional changes that you can make to further enhance your system's performance and functionality.

101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
www.altera.com
Applications Hotline:
(800) 800-EPLD
Literature Services:
lit_req@altera.com