



Nios Embedded Processor

Software Development Reference Manual



101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
<http://www.altera.com>



Document Version: 2.2
Document Date: July 2002

Copyright © 2002 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, mask work rights, and copyrights. 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 Corporation. 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.





This document provides information for programmers developing software for the Nios[®] embedded soft core processor. Primary focus is given to code written in the C programming language; however, several sections discuss the use of assembly code as well.

The terms Nios processor or Nios embedded processor are used when referring to the Altera[®] soft core microprocessor in a general or abstract context.

The term Nios CPU is used when referring to the specific block of logic, in whole or part, that implements the Nios processor architecture.

Table 1 shows the reference manual revision history.

Table 1. Revision History	
Date	Description
July 2002	Changes to nr_debug_dump_trace, nr_debug_isr_halt, nr_debug_isr_continue, and nios-elf-gdb. Updated PDF - version 2.2
April 2002	Updated PDF - version 2.1
January 2002	Minor amendments. Added DMA and Debug core routines, nios-elf-gprof and tracelink utilities.
March 2001	Nios Embedded Processor Software Development Reference Manual - printed

How to Find Information

- Adobe Acrobat's Find feature lets you to search the contents of a PDF file. Click the binoculars toolbar icon to open the Find dialog box.
- Bookmarks serve as an additional table of contents.
- Thumbnail icons, which provide miniature previews of each page, provide a link to the pages.
- Numerous links, shown in green text, allow you to jump to related information.

How to Contact Altera

For the most up-to-date information about Altera products, go to the Altera world-wide web site at <http://www.altera.com>.

For technical support on this product, go to <http://www.altera.com/mysupport>. For additional information about Altera products, consult the sources shown in Table 2.

Table 2. How to Contact Altera






Information Type	USA & Canada	All Other Locations
Product literature	http://www.altera.com	http://www.altera.com
Altera literature services	lit_req@altera.com (1)	lit_req@altera.com (1)
Non-technical customer service	(800) 767-3753	(408) 544-7000 (7:30 a.m. to 5:30 p.m. Pacific Time)
Technical support	(800) 800-EPLD (3753) (7:30 a.m. to 5:30 p.m. Pacific Time)	(408) 544-7000 (1) (7:30 a.m. to 5:30 p.m. Pacific Time)
	http://www.altera.com/mysupport/	http://www.altera.com/mysupport/
FTP site	ftp.altera.com	ftp.altera.com

Note

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

Typographic Conventions

The *Nios Embedded Processor Software Development Reference Manual* uses the typographic conventions shown in [Table 3](#).

<i>Table 3. Conventions</i>	
Visual Cue	Meaning
Bold Type with Initial Capital Letters	Command names, dialog box titles, checkbox options, and dialog box options are shown in bold, initial capital letters. Example: Save As dialog box.
bold type	External timing parameters, directory names, project names, disk drive names, filenames, filename extensions, and software utility names are shown in bold type. Examples: f_{MAX} , \QuartusII directory, d: drive, chiptrip.gdf file.
<i>Bold italic type</i>	Book titles are shown in bold italic type with initial capital letters. Example: <i>1999 Device Data Book</i> .
<i>Italic Type with Initial Capital Letters</i>	Document titles are shown in italic type with initial capital letters. Example: <i>AN 75 (High-Speed Board Design)</i> .
<i>Italic type</i>	Internal timing parameters and variables are shown in italic type. Examples: <i>t_{PIA}</i> , <i>n + 1</i> . Variable names are enclosed in angle brackets (< >) and shown in italic type. Example: < <i>file name</i> >, < <i>project name</i> >.pof file.
Initial Capital Letters	Keyboard keys and menu names are shown with initial capital letters. Examples: Delete key, the Options menu.
"Subheading Title"	References to sections within a document and titles of Quartus II Help topics are shown in quotation marks. Example: "Configuring a FLEX 10K or FLEX 8000 Device with the BitBlaster™ Download Cable."
Courier type	Signal and port names are shown in lowercase Courier type. Examples: data1, tdi, input. Active-low signals are denoted by suffix n, such as resetn. Anything that must be typed exactly as it appears is shown in Courier type. For example: c:\quartusII\qdesigns\tutorial\chiptrip.gdf. Also, sections of an actual file, such as a Report File, references to parts of files (such as the AHDL keyword SUBDESIGN), as well as logic function names (such as TRI) are shown in Courier.
1., 2., 3., and a., b., c.,...	Numbered steps are used in a list of items when the sequence of the items is important, such as the steps listed in a procedure.
	Bullets are used in a list of items when the sequence of the items is not important.
	The checkmark indicates a procedure that consists of one step only.
	The hand points to information that requires special attention.
	The angled arrow indicates you should press the Enter key.
	The feet direct you to more information on a particular topic.



Notes:



About this Manual	iii
How to Find Information	iii
How to Contact Altera	iv
Typographic Conventions	v
Overview	1
Project Considerations	1
Development Flow	2
GERMS Monitor	8
Monitor Commands	9
GERMS Boot Process for the Default 32-Bit Nios Design	10
Bootting From Flash Memory	11
SDK Tree Overview	12
The Include ("inc") Directory	12
The Library ("lib") Directory	16
Nios Program Structure	19
Nios Library Routines	19
C Runtime Support	20
System-Level Services	21
High-Level C Support	25
Routines	27
Nios Peripheral Routines	27
Debug Core	29
Debug Core Register Access	30
Debug Core Trace Data	30
Debug Core Interrupt	30
Debug Core Software Routines and Macros	31
DMA	34
DMA Software Data Structure	34
DMA Software Routines	35
PIO	39
PIO Software Data Structure	39
PIO Software Routine: nr_pio_showhex	40
SPI	41
SPI Software Data Structure	41
SPI Software Routines	42
Timer	43

Timer Software Data Structure	43
Timer Software Routine: nr_timer_milliseconds	44
UART	45
UART Software Data Structure	45
UART Software Routines	45
Utilities	51
Nios Software Development Utilities	51
hexout2flash	52
Usage	52
Options	52
Example	52
nios_bash	53
Usage	53
nios-build	54
Usage	54
Options	55
Example	55
nios-convert	56
Usage	56
Options	56
Example	56
nios_csh	57
Usage	57
Example	57
nios-elf-as	58
Usage	58
Options	58
nios-elf-gcc	60
Usage	60
Options	60
nios-elf-gdb	64
Usage	64
Options	64
nios-elf-gprof	66
Usage	66
Options	66
Example	67
Advanced Usage	69
nios-elf-ld	70
Usage	70
Options	70
nios-elf-nm	75
Usage	75
Options	75
Example	76

nios-elf-objcopy	77
Usage	77
Options	77
nios-elf-objdump	79
Usage	79
Options	79
Example	81
nios-elf-size	82
Usage	82
Options	82
nios-run	83
Usage	83
Options	83
Example	83
srec2flash	84
Usage	84
Example	84
tracelink	86
Usage	86
Example	86
Appendix	89
Appendix A: GERMS Monitor Usage	89
Hardware Considerations (32-Bit Nios CPU only)	91
Appendix B: Assembly Language Macros	93
Index	95

The Nios embedded processor is a soft core CPU optimized for programmable logic and system-on-a-programmable-chip (SOPC) designs. SOPC designs are created using the MegaWizard® Plug-In Manager included in the Quartus® II development software. When the SOPC Builder generates a design, several results occur:

1. The system memory map is checked for consistency. Peripheral addresses and interrupt priorities are verified to be unique, and fall within the range of valid entries for the CPU. If not, appropriate errors are reported and corrections must be made before continuing.
2. A custom software development kit (SDK) is generated for the new Nios system. The SDK consists of a compiled library of software routines for the SOPC design, a Makefile for rebuilding the library, and C header files containing structures for each peripheral.
3. An HDL that describes the custom SOPC Module is generated. This HDL is (optionally) synthesized into a single netlist, which can then be integrated as a component in a larger system.

This document covers the SDK generated in step 2 above. All directories and files mentioned are assumed to be part of the SDK unless otherwise specified.

Project Considerations

Many design scenarios are possible in Nios processor-based systems. Before beginning development, it is helpful to make some decisions based on application requirements. Consider the following issues before starting the SOPC design:

- **Memory Model**
Application code can reside in on-chip RAM or ROM, in external memory devices, or both. The amount of available on-chip memory depends on the Altera programmable logic device (PLD). When targeting a device with a small amount of on-chip memory (that is, 1 to 20 Kbyte), assembly code may require hand optimization to maintain a small memory footprint.

Off-chip devices may be added to increase the addressable memory space, up to a maximum of 4 Gbytes. Consider memory access time and board layout.

■ CPU Core Size

The Nios CPU can be configured with a variety of options that affect the amount of logic and memory resources required to implement the SOPC module, including the CPU, peripherals and bus logic. These options allow the designer to make swaps between CPU size (that is, cost) and performance.

The Nios CPU allows 32-bit and 16-bit configurations. The 16-bit CPU consumes fewer logic elements (LEs, a unit of logic resources) and executes faster for software that does not require 32-bit operations. Regardless of data path width, the 32-bit Nios CPU can restrict the width of the address bus to conserve LEs. For example, if 4 MByte of address space is required, the address bus can be restricted to 22 bits wide.

■ Software Execution Speed and Hardware Acceleration

All options affecting the Nios CPU's configuration affect the CPU's performance in some respect. Many options offer a dramatic improvement in code execution time for a moderate trade-off in the CPU core size. Consider these cases:

- Software operating on 32-bit data executes more efficiently on the 32-bit Nios CPU.
- Instructions and data can usually be fetched from on-chip memory with less latency than off-chip memory.
- The 32-bit Nios CPU offers two hardware-accelerated multiply instructions, which achieve up to ten times the performance of a software-only implementation.
- Custom instructions can be added to the Nios instruction set. Custom instructions replace a complex sequence of standard instructions with a fast hardware implementation. Iterative, arithmetic algorithms can achieve greater performance by implementing the inner loop in hardware.

Development Flow

The following outline describes a typical development flow used when creating a Nios processor-based design from scratch. It is assumed initial development is accomplished using the development board and software tools included in the Nios development kit.

Developing applications using the Nios embedded processor is slightly different than that of traditional processors, since the designer can configure the processor architecture and specify the peripheral content. That is, a designer can build a microcontroller according to system design requirements, as opposed to selecting a pre-built microcontroller with a fixed set of peripherals, on-chip memory, and external interfaces.

The Nios development board included in the kit comes with a 32-bit reference design (processor, on-chip memory with monitor, and peripherals), and application code pre-loaded in on-board flash memory. This reference design helps you quickly familiarize yourself with the development tools prior to starting your custom design (see the *Nios Tutorial*). If possible, begin your software design using the Nios development board as your target hardware platform.

Step 1: Define the Processor

Based on your system needs, decide the following:

CPU data path

Does your application require a 32-bit or 16-bit data path? If a 32-bit data path is not needed, a 16-bit data path generates a smaller, faster CPU core.

Data Path	LEs Used	Address Range
16-bits	900	64 K
32-bits	1250	4 GB

Register File Size

Specify the size of the Nios CPU's internal register file to suit the system requirements. Valid configurations are 128, 256, or 512 registers. The width of each register is the width of the CPU data path. A larger register file consumes more on-chip memory resources.

The Nios processor implements a windowed register file. 32 registers are visible to the CPU at any given time. A window pointer into the register file makes the register file behave like a stack. This improves performance of subroutine calls by eliminating the need to load and store processor context and subroutine variables to slow external memory devices.

Multiplier

If your code performs few multiplication operations, does not contain time critical multiplication, or you want to make the CPU core as small as possible, use the software math libraries included with the C compiler. On the other hand, if your code performs numerous multiplication operations or must be optimized for speed, choose one of the dedicated hardware multipliers (MSTEP or MUL).

Table 1. Multiplication Options			
Option	Additional LEs Used	Clock Cycles 16x16>32	Clock Cycles 32x32>32
None (software)	0	80	250
MSTEP	+20	18	80
MUL	+400	2	16

On-Chip Memory

Determine how much on-chip ROM and RAM your system requires. The Nios processor uses embedded system blocks (ESBs) for on-chip memory. There are practical limits to the number of ESBs used for on-chip memory (see the *Altera Device Data Book* for details on the number of ESBs available in particular devices). The SOPC Builder imposes a maximum limit of 20 Kbytes per on-chip memory device.

Off-Chip Memory

Interfaces to off-chip memory are provided for flash memory, SRAM, SSRAM, and SDRAM. Any user-defined interface may be created to connect other off-chip memory devices. The GERMS monitor included in the development kit contains software routines for writing to and erasing Advanced Micro Devices (AMD) flash devices. See [“GERMS Monitor” on page 8](#) for details.

Peripherals

Decide the type and number of peripherals to connect to the Nios processor. A number of peripherals, listed in [Table 2](#), are included with the Nios development kit. You can also create interfaces to off-chip or custom on-chip peripherals using either the parallel input/output (PIO) peripheral or user-defined interface.

Table 2. Nios Peripherals

Peripheral Name	Description
DMA	Direct memory access: enables high-speed data transfer
PIO	1- to 32-bit parallel input/output and edge capture
SDRAM Controller	Interface to synchronous dynamic random-access memory (SDRAM)
SPI	Serial peripheral interface, 3-wire, master/slave
Timer	32-bit general-purpose timer
UART	Universal asynchronous receiver/transmitter
User-defined interface	Custom interface to on-chip and off-chip peripherals
Off-chip shared bus	Shared interface to off-chip peripherals and memory

Step 2: Build the Processor

Using the Quartus II development software and the MegaWizard Plug-In Manager, generate a custom processor system based on the choices you made in Step 1. As you build the processor, you will:

- Configure the CPU hardware options, including data path width (32 or 16 bits), multiplier acceleration, and custom instruction usage
- Add required peripherals and configure peripheral hardware options
- Specify the processor boot address
- Assign peripheral memory addresses and alignment
- Assign interrupt priorities for peripherals and external interfaces as needed
- Specify peripheral setup and hold requirements as needed
- Assign peripheral and memory wait states as needed
- Enable dynamic bus-sizing to narrow memory or peripheral interfaces as needed
- Specify files containing instruction or data memory to initialize on-chip ROM and/or RAM

Once the Nios system is created, it may optionally be combined with other user-defined logic. The top-level design must be synthesized and fit into an Altera device using the Quartus II software. Quartus II outputs a device configuration file of type **sof** or **hexout**, which must be downloaded to the development board. You can use the Quartus II software and the ByteBlaster MV™ download cable to configure the Altera device directly from a host PC. Another option is to burn the device configuration file into on-board flash, and then reset the board.

The GERMS monitor program included in the Nios development kit allows you to run executable code, read from and write to memory, download blocks of code (or data) to memory, and erase flash. See [“GERMS Monitor” on page 8](#) for details. By assigning the GERMS monitor to the processor boot address (typically on-chip ROM), you can immediately begin code development, download, and debug.



See the *Nios Tutorial* for instructions on creating a Nios processor-based SOPC design.

Step 3: Save the Processor Configuration to FLASH

The Altera PLD that implements the Nios CPU and other logic is volatile and therefore must be configured each time the board is reset by pushing the RESET button (SW2) or by cycling power. This configuration data (the hardware design) is stored in on-board flash. The development board contains logic that supports a dual configuration scheme as follows:

By default, the APEX™ device is configured from a “User” section of flash memory (address range 0x180000–0x1BFFFF). If the APEX device fails to configure due to corrupt or empty User section, it is automatically configured from the “Factory” section of flash memory (address range 0x1C0000–0x1FFFFFF). When jumper JP2 is shorted and the RESET button is pressed, the APEX device is forced to configure from the Factory section of flash memory.

During development, it is recommended you always store a new design to the User section of flash memory. This way, if a hardware bug occurs you can reconfigure the APEX device with the known good reference design stored in the Factory section of flash memory. Altera loads the factory section of flash memory with a 32-bit Nios system design. See [“hexout2flash” on page 52](#) for details on downloading device configuration files to flash memory.

Step 4: Create and Compile the Application Software

Using a text editor (xemacs and vi editors are included with Cygwin), write the application source code in C/C++ or assembly (.c or .s).

Compile your source code into executable code using the **nios-build** utility or a make file. The resultant binary code is stored in S-record format (.srec).

For small- to medium-sized software projects, use **nios-build** to generate executable code. See [“nios-build” on page 54](#) for details.

For large projects, use hand-crafted make files. For details on using **make**, see the online GNU documentation by choosing **Programs > Cygwin > Cygwin Documentation** (Windows Start Menu). In the help window that appears, click **Using make**.

Step 5: Download the Executable Code to the Development Board

Use **nios-run** to download and run the application on the development board. See [“nios-run” on page 83](#) for details.

Step 6: Debug the Code

If you use `printf()` to debug your code, messages are sent to the STDIO (such as UART). The **nios-run** utility acts as a dumb terminal to display these messages on your development system terminal.

If more sophisticated debugging is called for, rebuild the code with the compiler debugging option set ON, then use the GNU debugger (GDB) or other integrated development environment (IDE) to step through the code, examine memory and register contents, and so on. See [“nios-elf-gdb” on page 64](#) for details.

Step 7: Transition to Auto-Booting Code

Once the application code is sufficiently debugged, you may wish to store the executable code on the development board. The Nios CPU then automatically executes the application upon reset. The options for storing nonvolatile code on the board are:

Store Code in On-Chip Memory

To store program data in on-chip ROM or RAM, specify a file to initialize the memory. (See [“Step 2: Build the Processor” on page 5](#).) In this case, you remove the GERMS monitor completely and replace it with your application code.

Store Code in Off-Chip Memory

Store the program in flash memory so the GERMS monitor automatically executes it after initialization. Use **srec2flash** to convert the executable code in **.srec** format to a **.flash** format that can be easily burned into the on-board flash. **srec2flash** also adds a software routine that copies the executable code from flash memory to SRAM at start time. See [“srec2flash” on page 84](#) for details.

or

Remove the GERMS monitor entirely and change the Nios CPU reset address to point to the program in flash memory. Use **srec2flash** to add a routine that copies the executable code from flash memory to SRAM at start time.

Step 8: Transition Design From Nios Development Board to Target Hardware

If possible, continue using the GERMS monitor to download code to RAM on the target hardware. Iteratively compiling and downloading new software without burning a new ROM or recompiling the hardware design is quite useful.

GERMS Monitor

The GERMS monitor is a simple monitor program that provides basic development facilities for the Nios development board, and can also be used in an end system. The GERMS monitor is included in the default design stored in flash memory of the development board. On power-up, the GERMS monitor is the first code to execute, and it controls the boot process. Once booted, it provides a way to read from and write to the on-board SRAM or flash memories.

“GERMS” is a mnemonic for the minimal command set of the monitor program included in the Nios development kit:

G Go (run a program)
E Erase flash
R Relocate next download
M Memory set and dump
S Send S-records
: Send I-Hex records



For details on GERMS monitor usage, see [“Appendix A: GERMS Monitor Usage” on page 89](#).

Monitor Commands

When the monitor is running, it is always waiting for commands. The user sends commands as text characters to a “host communication UART” when building the Nios hardware. Commands consist of a letter followed by an address, or two addresses separated by a hyphen. The M command contains an address followed by a colon, followed by data to write.

Commands are executed as they are typed. For example, if you write to memory, each word is stored as soon as it is entered. GERMS does not recognize backspace or delete. If you type a mistaken character, press the Escape key to restart the monitor.

All numbers and addresses entered into the monitor are in hexadecimal.

Table 3. GERMS Monitor Commands

Syntax	Example	Description
G<base address>	G40000	GO—Execute a CALL instruction to the specified address
E<base address>	E180000	Erase flash memory. If the address is within the range of the “flash” ROM, the sector containing that address is erased.
R <offset>	R1C0000	Relocate command. Specifies the offset for the next downloaded S-record or I-Hex. When two parameters are provided, the offset is the <from address> subtracted from the <to address>. See “ Appendix A: GERMS Monitor Usage ” on page 89 for an example.
R<from address>-<to address>	R0-180000	
M<address>	M50000	Display memory starting from the address
M<address>-<address>	M40000-40100	Display a range of memory. Press ↵ again to show the same number of bytes, starting where the last M command ended.
M<address>:<value> <value>...	M50000:1 2 3 4	Write successive 16-bit words to memory until the end of line.
M<address>-<address>:<value>	M50000-50100:AA55	Fill a range of memory with a 16-bit word.
↵	↵	Display the next 64 bytes of memory.
S<S-record data>	S21840000...	Write S-record to next memory location.
:<I-hex record data>	:80000004...	Write I-hex record to next memory location.
Escape key	—	Restart the monitor.

GERMS Boot Process for the Default 32-Bit Nios Design

This section describes the boot process used by the GERMS monitor running on the “Factory” APEX design that ships on the Nios development board. Custom SOPC designs are not required to use GERMS, but probably perform many of the same operations to prepare the system for software execution.

The monitor is located at address zero, 0x0000, in the Nios development board default configuration.

There are several ways the monitor may come to be executed. When the Altera PLD is configured with the default Factory design, execution begins at address zero, which is the monitor. Later, if any unexpected TRAP or interrupt occurs for which the vector table is not initialized, the monitor is executed.

When the monitor starts running, it performs the system initialization:

1. Disables interrupts so that interrupt requests from the UART, Timer, switch PIO, and other peripherals do not interrupt the initialization process.
2. Sets current window pointer (CWP) to HI_LIMIT to initialize the register file window.
3. Sets interrupt priority (IPRI) to 63, so that when interrupts are re-enabled, all interrupt requests are serviced.
4. Initializes the stack pointer by setting %sp to 0x80000 (nasys_stack_top).

It then looks for code to execute out of flash memory:

5. Examines the two bytes at 0x14000C (nasys_flash + 0x04000C).
6. Examines button 0 on the switch PIO (SW4).
7. If the button is not pressed and the two bytes contain “N” and “i”, the monitor executes a CALL to location 0x140000 (nasys_flash + 0x040000).

If the code is not executed in step 7 or that code returns, then:

8. Prints an 8-digit version number to STDOUT, of the form “#vvvvPPPP” followed by a carriage return, where “vvvv” is a monitor pseudo-version—it is different but not necessarily consecutive for different builds of the monitor—and PPPP is the processor version number, as retrieved from processor register CTL 6.
9. Waits for user commands from STDIN.

Booting From Flash Memory

User software applications can be stored in flash and executed automatically on system power-up or reset. This is particularly useful when developing application code targeted for flash memory.

During the boot process, the GERMS monitor checks for the existence of application software in flash memory (step 5 in “GERMS Boot Process for the Default 32-Bit Nios Design”). If found, the processor immediately executes the code. Use the software utility **srec2flash** to prepare programs for this style of operation (see “**srec2flash**” on page 84). **srec2flash** adds a piece of code to the beginning of the program that copies the application code from flash (slow memory) to SRAM (fast memory) and runs from SRAM.

To return program execution to the GERMS monitor (that is, avoid running code stored in flash memory):

1. Hold down SW4.
2. Press then release the RESET button (SW2).
3. Release SW4.



For more details on this process, see the Altera white paper *Converting .srec Files to .flash Files for Nios Embedded Processor Applications* at http://www.altera.com/literature/wp/wp_srec_to_flash.pdf.

SDK Tree Overview

The SDK is generated as a subdirectory of your Quartus II (or MAX+PLUS® II) project. It is given the name of the SOPC Module (the Nios system) appended with “_sdk”. For example, the 32-bit Factory reference design (ref_32_system) directory structure is:

```
.../ref_32_system_cpu_sdk/
|
+--- inc/
|
+--- lib/
|
+--- src/
```

The Include (“inc”) Directory

```
[bash] ...inc/: ls -l
total 17
-rw-r--r--  1 niosuser Administ  12413 Oct  24 15:01 nios.h
-rw-r--r--  1 niosuser Administ   7088 Oct  24 15:01 nios.s
-rw-r--r--  1 niosuser Administ   8998 Oct  24 15:01 nios_macros.s
-rw-r--r--  1 niosuser Administ    688 Oct  24 15:01 pio_lcd16207.h
```

The SDK include directory, **inc**, contains several files intended for inclusion from your application programs. These files define the peripheral addresses, interrupt priorities, register structures, and other useful constants and macros. To use these features in a program, include **nios.h** in each file if the file is written in C or C++, or **nios.s** if the file is written in assembly language.

nios.h (and nios.s)

This file contains register maps for each peripheral in your system. Additionally, it contains C prototypes for library routines available for each peripheral.

For C programs, the register maps are provided as structures. For example, the Timer peripheral's structure is:

```
typedef volatile struct
{
    int np_timerstatus; // read only, 2 bits (any write to clear TO)
    int np_timercontrol; // write/readable, 4 bits
    int np_timerperiodl; // write/readable, 16 bits
    int np_timerperiodh; // write/readable, 16 bits
    int np_timersnapl; // read only, 16 bits
    int np_timersnaph; // read only, 16 bits
} np_timer;

enum
{
    np_timerstatus_run_bit = 1, // timer is running
    np_timerstatus_to_bit = 0, // timer timed out

    np_timercontrol_stop_bit = 3, // stop the timer
    np_timercontrol_start_bit = 2, // start the timer
    np_timercontrol_cont_bit = 1, // continuous mode
    np_timercontrol_ito_bit = 0, // enable time out interrupt

    np_timerstatus_run_mask = (1<<1), // timer is running
    np_timerstatus_to_mask = (1<<0), // timer timed out

    np_timercontrol_stop_mask = (1<<3), // stop the timer
    np_timercontrol_start_mask = (1<<2), // start the timer
    np_timercontrol_cont_mask = (1<<1), // continuous mode
    np_timercontrol_ito_mask = (1<<0) // enable timeout interrupt
};
```

The prefix `np_` stands for “Nios peripheral”.

Each register is included as an integer (int) structure field, so software that uses the structure can be targeted transparently to both 32-bit and 16-bit Nios processors.

For registers with sub-fields or control bits, additional constants are defined to reference those fields by both mask and bit number. (Bit numbers are useful for the Nios assembly language instructions SKP0 and SKP1.)

nios.h and **nios.s** also provide addresses for all peripherals, interrupt numbers, and other useful constants. Following is an excerpt:

```
#define na_timerl ((np_timer *) 0x00000440)
#define na_timerl_irq 25
#define na_led_pio ((np_pio *) 0x00000460)
#define na_button_pio ((np_pio *) 0x00000470)
#define na_button_pio_irq 27
#define nasys_printf_uart ((np_uart *) 0x00000400)
#define nasys_printf_uart_irq 26
```

The name `na_timer1` is derived from the peripheral's name "Timer". The prefix `na_` stands for "Nios address". It is defined as a number cast to the type of "`np_timer *`". This allows the symbol "`na_timer1`" to be treated as a pointer to a timer structure. Following is an example of code written to access the Timer:

```
int status = na_timer1->np_timerstatus;    /* get status of timer1 */
```

Switches

The following switches are defined in the **nios.h** file:

```
#define __nios_catch_irqs__      1
#define __nios_use_constructors__ 1
#define __nios_use_cwpmgr__     1
#define __nios_use_fast_mul     1
#define __nios_use_small_printf__ 1
```

`__nios_catch_irqs__`

When `__nios_catch_irqs__` is set to 1, a default interrupt handler is installed for every interrupt. Changing this setting to 0 saves a small amount of code space.

`__nios_use_constructors__`

When `__nios_use_constructors__` is set to 1, the Nios library contains startup code to call any initializing code for statically allocated C++ classes. By default, this is set to 1. Changing this setting to 0 reduces the compiled software's code footprint if static initialization of C++ classes is not needed. This is useful for creating software that requires a small ROM memory footprint.

`__nios_use_cwpmgr__`

When `__nios_use_cwpmgr__` is set to 1, the Nios library contains code for handling register window underflows. Changing this setting to 0 reduces the code footprint of the compiled software. Do this only when the code does not call to a subroutine depth that exceeds the register file size. See the *Nios 16-Bit Programmer's Reference Manual* or *Nios 32-Bit Programmer's Reference Manual* for details on the CWP register and managing the windowed register file.

`__nios_use_fast_mul__`

This setting is defined in the `nios.h` file but works in conjunction with the `NIOS_USE_MULTIPLY` and `NIOS_USE_MSTEP` settings in `.../lib/Makefile`. For details, see “`__nios_use_fast_mul__`” on page 17.

`__nios_use_small_printf__`

The standard `printf()` routine in the GNU libraries takes about 40 Kbytes of Nios code. It contains support for the complete ANSI `printf()` specification, including floating point numbers. When `__nios_use_small_printf__` is set to 1, a more minimal implementation is linked into the Nios library, which takes about 1 Kbyte of Nios code. This “small printf” supports only integers and the formats `%c`, `%s`, `%d`, `%x`, and `%X`. This setting is useful for sending debug messages to STDIO (a UART) without significantly increasing the executable code size.

`nios_macros.s`

This file includes various useful assembly language macros. See “Appendix B: Assembly Language Macros” on page 93 for details.

The Library (“lib”) Directory

```
[bash] ...lib/: 11
total 262
-rw-r--r-- 1 niosuser Administ 5353 Nov  6 15:02 Makefile
-rw-r--r-- 1 niosuser Administ 6564 Oct 24 15:01 flash_AMD29LV800.c
-rw-r--r-- 1 niosuser Administ 139016 Nov 13 15:59 libnios32.a
-rw-r--r-- 1 niosuser Administ 139706 Nov 13 15:59 libnios32_debug.a
-rw-r--r-- 1 niosuser Administ 711 Oct 24 15:01 nios_copyrange.s
-rw-r--r-- 1 niosuser Administ 3133 Oct 24 15:01 nios_cstubs.s
-rw-r--r-- 1 niosuser Administ 7932 Oct 24 15:01 nios_cwpmanager.s
-rw-r--r-- 1 niosuser Administ 561 Oct 24 15:01 nios_delay.s
-rw-r--r-- 1 niosuser Administ 11111 Oct 24 15:01 nios_emulator.s
-rw-r--r-- 1 niosuser Administ 392 Oct 24 15:01 nios_gdb_standalone.c
-rw-r--r-- 1 niosuser Administ 27310 Oct 24 15:02 nios_gdb_standalone.srec
-rw-r--r-- 1 niosuser Administ 26151 Oct 24 15:01 nios_gdb_stub.c
-rw-r--r-- 1 niosuser Administ 1886 Oct 24 15:01 nios_gdb_stub.h
-rw-r--r-- 1 niosuser Administ 443 Oct 24 15:01 nios_gdb_stub_io.c
-rw-r--r-- 1 niosuser Administ 9815 Oct 24 15:01 nios_gdb_stub_isr.s
-rw-r--r-- 1 niosuser Administ 23245 Oct 24 15:01 nios_germes_monitor.s
-rw-r--r-- 1 niosuser Administ 7740 Oct 24 15:02 nios_germes_monitor.s.o
-rw-r--r-- 1 niosuser Administ 22284 Nov  2 09:44 nios_gprof.c
-rw-r--r-- 1 niosuser Administ 7088 Oct 24 15:01 nios_isrmanager.s
-rw-r--r-- 1 niosuser Administ 720 Oct 24 15:01 nios_jumptostart.s
-rw-r--r-- 1 niosuser Administ 3281 Oct 24 15:01 nios_math1.s
-rw-r--r-- 1 niosuser Administ 960 Oct 24 15:01 nios_printf.c
-rw-r--r-- 1 niosuser Administ 2282 Oct 24 15:01 nios_setjmp.s
-rw-r--r-- 1 niosuser Administ 4260 Oct 24 15:01 nios_setup.s
-rw-r--r-- 1 niosuser Administ 5481 Oct 24 15:01 nios_sprintf.c
-rw-r--r-- 1 niosuser Administ 764 Oct 24 15:01 nios_zerorange.s
drwxr-xr-x 2 niosuser Administ 12288 Nov 13 15:59 obj32/
drwxr-xr-x 2 niosuser Administ 12288 Nov 13 15:59 obj32_debug/
-rw-r--r-- 1 niosuser Administ 5859 Oct 24 15:01 pio_lcd16207.c
-rw-r--r-- 1 niosuser Administ 1218 Oct 24 15:01 pio_showhex.s
-rw-r--r-- 1 niosuser Administ 2392 Oct 24 15:01 timer_milliseconds.s
-rw-r--r-- 1 niosuser Administ 770 Oct 24 15:01 uart_rxchar.s
-rw-r--r-- 1 niosuser Administ 764 Oct 24 15:01 uart_txchar.s
-rw-r--r-- 1 niosuser Administ 450 Oct 24 15:01 uart_txcr.s
-rw-r--r-- 1 niosuser Administ 901 Oct 24 15:01 uart_txhex.s
-rw-r--r-- 1 niosuser Administ 794 Oct 24 15:01 uart_txhex16.s
-rw-r--r-- 1 niosuser Administ 796 Oct 24 15:01 uart_txhex32.s
-rw-r--r-- 1 niosuser Administ 692 Oct 24 15:01 uart_txstring.s
[bash] ...lib/:
```

The SDK library directory, **lib**, contains a Makefile, and archive, source, and object files for libraries usable by your Nios system.

Some source files are in assembly language, and others are in C. The archive contains assembled (or compiled) versions of routines from each file, suitable for linking to your program. See [“Routines” on page 27](#) for details.

The command line tool **nios-build** uses the **libnios32.a** library directory when building for a 32-bit system, or **libnios16.a** when building for a Nios 16-bit system.

The Makefile contains instructions for rebuilding the archive file. The beginning of the Makefile contains several settings to enable or disable various features of the Nios library. Following is an excerpt from a typical Nios library Makefile:

```
#
# Nios SDK Generated Makefile
# 2002.01.24 01:19:30
# //d/niosbuild/src/tree/Delta/SWDev/bin/nios_reference32.ptf
#
NIOS_USE_MSTEP = 1 # CPU option (shift, test, & add)
NIOS_USE_MULTIPLY = 0 # CPU option (16x16->32)
NIOS_SYSTEM_NAME = nios_system_module

M = 32 # Nios 32
```

You can change each of these settings to customize the Nios library. After changing a setting, enter `make -s all` at the command line to rebuild the library.

The following sections describe each setting.

__nios_use_fast_mul__

This variable is defined in the **nios.h** file but works in conjunction with **NIOS_USE_MULTIPLY** and **NIOS_USE_MSTEP**.

If __nios_use_fast_mul__ is set to 0, a standard software multiply routine which is slow and short, is linked into the Nios library. To instruct the library to perform integer multiplications with either optional instruction **MUL** or **MSTEP**, set __nios_use_fast_mul__ to 1. When __nios_use_fast_mul__ is set to 1, and both **NIOS_USE_MULTIPLY** and **NIOS_USE_MSTEP** are set to 0, a hand-optimized integer multiplication routine, which is faster and larger, is linked into the Nios library. See [Table 4 on page 18](#) for a comparison of the possible settings

NIOS_USE_MSTEP

If **NIOS_USE_MSTEP** is set to 1, the Nios library overrides the standard multiplication routine with a faster one that uses the **MSTEP** instruction. This is set to 1 automatically if the **MSTEP** feature is selected in the SOPC Builder software. Use this setting in conjunction with __nios_use_fast_mul__.

NIOS_USE_MULTIPLY

If NIOS_USE_MULTIPLY is set to 1, the Nios library overrides the standard multiplication routine with a faster one that uses the MUL instruction, which runs even faster than MSTEP multiplication. This is set to 1 automatically if the MULTIPLY feature is selected in the SOPC Builder software. Use this setting in conjunction with `__nios_use_fast_mul__`.

Table 4 shows how the above settings work together:

<i>Table 4. Size Versus Speed Multiplication</i>			
<code>__nios_use_fast_mul__</code> Value	NIOS_USE_MSTEP Value	NIOS_USE_MULTIPLY Value	Use
0	0 or 1	0 or 1	Slow and short multiplication routine
1	0	0	Fast and large multiplication routine
1	1	0	Routine with MSTEP instructions
1	0	1	Routine with MUL instruction

NIOS_SYSTEM_NAME

This is a string with the name of the Nios system.

M

This setting is either 16 or 32, to match the width of the Nios CPU. Also, **nios-build** uses this value to set the appropriate compiler and assembler options when building.

Nios Program Structure

In the typical case of a C program built with **nios-build**, the memory layout represented in the resultant S-record file is:

Table 5. Memory Layout

Address, ascending	Contents
nasys_program_mem + 0x00	A preamble consisting of a JUMP instruction to the symbol “_start” and the four characters “N”, “I”, “O”, and “S”. This is always at the beginning of the S-record output file. It comes from the library file nios_jumptostart.o .
nasys_program_mem + 0x10	The program's “main()” is in here, as well as all other routines, in the order shown below. The command nios-build passes nios_jumptostart.o to the GNU linker as its first file and the user program as its second.
(A higher address)	A routine labeled “_start”. This comes from the library file nios_setup.o . It performs some initialization, then calls “main()”.
(A higher address)	Two routines for handling “register window underflow” and “register window overflow” required by the Nios embedded processor to execute arbitrarily deep calling chains. These come from the nios_cwpmanager.o library file.
(A higher address)	Any other Nios library routines the program references. The linker extracts only the required routines from the file libnios32.a and includes them in the final program.
(A higher address)	Any read-only data from the program, such as strings or numeric constants.
(A higher address)	Any static variables in the program.

Nios Library Routines

The SDK for your Nios system includes the pre-built library **libnios32.a** (for a 32-bit Nios system) or **libnios16.a** (for a 16-bit Nios system); either is referred to here as the Nios library. The routines available vary depending on the peripherals in the Nios system. This section describes routines that are always present. Optional peripheral routines are discussed in [“Routines” on page 27](#).

C Runtime Support

Before a compiled program is run, certain initializations must take place. When **nios-build** is used to compile and link a program, the first routine executed is “_start”, which performs this initialization, then calls the “main()” routine. Furthermore, the standard C libraries rely on several low-level platform-specific routines.

Table 6 lists the low-level C runtime support provided by the Nios library, always present in the Nios library:

Table 6. C Runtime Support Routines		
Routine	Source File	Description
_start	nios_setup.s	Performs initialization prior to calling main()
_exit	nios_cstubs.s	Execute a JMP to nasys_reset_address
_sbrk	nios_cstubs.s	Increments “RAMLimit” by the requested amount and returns its previous value, unless the new value is within 256 bytes of the current stack pointer, in which case it returns 0. This is the low-level routine used by malloc() to allocate more heap space.
isatty	nios_cstubs.s	Returns “1”, indicating to the C library that there is a tty
_close	nios_cstubs.s	Returns “0”; not used by Nios software without a file system, but necessary to link
_fstat	nios_cstubs.s	Returns “0”; not used by Nios software without a file system, but necessary to link
_kill	nios_cstubs.s	Returns “0”; not used by Nios software without a file system, but necessary to link
_getpid	nios_cstubs.s	Returns “0”; not used by Nios software without a file system, but necessary to link
_read	nios_cstubs.s	Calls nr_uart_rxchar() to read a single character from a UART. The “fd” parameter is treated as the base address of a UART.
_write	nios_cstubs.s	Calls nr_uart_txchar() to print characters to a UART. The “fd” parameter is treated as the base address of a UART. This allows the routine fprintf() to print to any UART by passing a UART address in place of the file handle argument.
__mulsi3 ¹	nios_math1.s	Overrides the standard signed 32-bit multiplication routine in the GNU C library.
__mulhi3 ¹	nios_math1.s	Overrides the standard unsigned 32-bit multiplication routine in the GNU C library.

Note

- (1) This routine is faster than the standard routine, uses the MUL or MSTEP instructions (if present), and does not use a register window level. It uses more code space than the standard routine.

_start

The first instructions executed by a Nios CPU upon start-up are the preamble instructions to jump to `_start`, followed by the actual `_start` code instructions. Before compiled software can run, system initialization must be performed by the `_start` routine. The initialization steps are:

1. Initialize the stack pointer to “`nasys_stack_top`”.
2. Zero program storage between “`__bss_start`” and “`_end`”.
3. Set an internal variable named “`RAMLimit`” to “`_end`” (malloc claims memory upwards from here).
4. Optionally install the CWP Manager.
5. Optionally call the C++ static constructors.
6. Execute a `CALL` to the routine “`main()`”, which normally is the main entry point of your C routine.
7. If “`main()`” returns, ignore its return value and execute a `TRAP 0`. This usually results in restarting the monitor.

System-Level Services

The system-level service routines discussed in this section are always present in the Nios library, and are called automatically unless disabled in the Makefile.

Interrupt Service Routine Handler

The Nios processor allows up to 64 prioritized, vectored interrupts numbered 0 to 63. The lower the interrupt number, the higher the priority. Interrupt vectors 0 through 15 are reserved for system services, leaving 48 interrupt vectors for user applications.



See the *Nios 16-Bit Programmer's Reference Manual* or *Nios 32-Bit Programmer's Reference Manual* for details on Nios CPU exception handling.

nr_installuserisr

This routine installs a user interrupt service routine for a specific interrupt number. If `nr_installuserisr()` is used to set up the interrupt vector table, standard compiled C functions can be specified as interrupt service routines. This is useful for software designers who are not familiar with the low-level details of the Nios interrupt vector table. This function is declared in the include file **nios.h**.



If you manipulate the vector table directly, you must completely understand the mechanisms of the Nios register window, control registers, and so on, so that interrupt requests execute and return properly.

The user interrupt service routine receives the context value as its only argument when called. The interrupt service routine itself must clear any interrupt condition for a peripheral it services.

Syntax:

```
void nr_installuserisr(int trapNumber, void  
*nios_isrhandlerproc, int context);
```

Parameters

Parameter Name	Description
trapNumber	Interrupt number to be associated with a user interrupt service routine
nios_isrhandlerproc	User-supplied routine with the prototype: <code>typedef void (*nios_isrhandlerproc)(int context);</code>
context	A value passed to the routine specified by <code>nios_isrhandlerproc</code>

nr_installuserisr2

This routine is similar to `nr_installuserisr`, except when the user interrupt service routine is called, the interrupt number and the interrupted PC are passed by the funnel routine to the user interrupt handler, as well as the context.

Syntax:

```
void nr_installuserisr2(int trapNumber, void
*nios_isrhandlerproc2, int context);
```

Parameters

Parameter Name	Description
trapNumber	Interrupt number to be associated with a user interrupt service routine
nios_isrhandlerproc2	User-supplied routine with the prototype: <pre>typedef void (*nios_isrhandlerproc2)(int context, int irq_number, int interruptee_pc);</pre>
context	A value passed to the routine specified by <code>nios_isrhandlerproc</code>
irq_number	Interrupt request number (trapNumber)
interruptee_pc	Return address from the interrupt

CWP Manager

A detailed understanding of the windowed register file is not required to write Nios software. The CWP Manager routine handles the details of manipulating the register file during subroutine calls. This section describes the CWP Manager since it becomes part of most users' final software.

The Nios embedded processor contains 128, 256, or 512 general-purpose registers. Of these, 32 are visible to the software at any particular moment. They are named `%r0–%r31`, and can also be referred to as `%g0–%g7` (*global*), `%o0–%o7` (*out*), `%L0–%L7` (*local*), and `%i0–%i7` (*in*).

The CWP bits of the Nios STATUS register (`%ctl0`, readable via the RDCTL instruction) determines which 32 registers are visible. See the *Nios 16-Bit Programmer's Reference Manual* or *Nios 32-Bit Programmer's Reference Manual* for details.

Subroutines execute a SAVE instruction, which decrements the CWP by one, revealing 16 “new” registers. The “caller’s” %o registers are visible to the “callee” as %i registers. Eventually, however, there are no more registers to reveal, and the CWP points to the lowest registers.

When the supply of registers is exhausted and a SAVE is executed, it induces a software exception that is handled by the CWP Manager’s underflow handler. This handler saves every register onto the stack, and repositions the CWP back to the top.

Conversely, subroutines execute a RESTORE instruction when they are ready to return. If the CWP is already at the top of the register file, a trap is induced, which is handled by the CWP Manager’s overflow handler. This handler restores the register contents from memory where they were saved earlier by the corresponding underflow condition.

nr_installcwpmanager

This routine is called automatically by `_start()` if the library was built with `__nios_use_cwpmgr__ = 1`. It installs service routines for the Nios CPU underflow and overflow exceptions. This function is declared in the include file **nios.h**.

Syntax:

```
void nr_installcwpmanager(void);
```

General-Purpose System Routines

The following sections describe the routines that perform general-purpose operations.

nr_delay

This routine causes program execution to pause for the number of milliseconds specified in milliseconds. During this delay, the function executes a tight countdown loop a fixed number of iterations, based on the system clock frequency specified at the time the Nios CPU was defined. This function is declared in the include file **nios.h**.

Syntax:

```
void nr_delay(int milliseconds);
```

The `milliseconds` parameter is the length of time, in milliseconds, for program execution to be suspended.

nr_zerorange

This routine writes zero to a range of bytes starting at rangeStart and counting up, writing rangeByteCount number of zero bytes. This function is declared in the include file **nios.h**.

Syntax:

```
void nr_zerorange(char *rangeStart, int rangeByteCount);
```

Parameters

Parameter Name	Description
rangeStart	First byte to set to zero
rangeByteCount	Number of consecutive bytes to set to zero

High-Level C Support

These routines are always present in the Nios library, unless disabled in the Makefile:

Table 7. High-Level C Support Routines		
Routine	Source File	Description
printf	nios_printf.c	This version of the standard C printf() function omits all support for floating point numbers, and supports only %d, %x, %X, %c, and %s formats. The Nios library includes this version of printf() because the standard library routine takes about 40 Kbytes of Nios code. This large footprint is primarily for floating point support, and the Nios CPU is often used for applications that do not require floating point. The Nios library version of printf() is about 1 Kbyte of Nios code.
sprintf	nios_printf.s	This routine uses the Nios library's version of printf() to print to a string in memory.

Nios Peripheral Routines



Table 8 summarizes C (or assembly) callable peripheral routines and macros that are automatically added to the custom SDK library when the corresponding peripherals are included in the Nios system design.

See the *Nios Embedded Processor Peripherals Reference Manual* for more information on the DMA, PIO, SPI, Timer, and UART peripherals, including details on registers, bits, and peripheral template file (PTF) assignments.

Table 8. Peripheral Routines Summary

Peripheral	Routine	Description
Debug Core	nr_debug_start	Initializes the debug core and begins monitoring instruction and data busses
	nr_debug_stop	Halts debug core
	nr_debug_dump_trace	Dumps the full contents of trace memory in text format
	nr_debug_isr_halt	Interrupt service routine that dumps trace memory and returns to the GERMS monitor
	nr_debug_isr_continue	Interrupt service routine that dumps trace memory and continues normal execution
	nm_debug_get_reg	Gets a debug core register value
	nm_debug_set_reg	Sets a debug core register value
	nm_debug_set_bp0	Sets up hardware breakpoint 0
Direct Memory Access (DMA)	nr_dma_copy_1_to_1	Transfers a range of bytes, half-words, or words between the source address and destination address.
	nr_dma_copy_1_to_range	
	nr_dma_copy_range_to_range	
	nr_dma_copy_range_to_1	
Parallel Input/Output (PIO)	nr_pio_showhex	Converts a 16-bit value to display as two hex digits on a seven-segment LED connected to a PIO.
Serial Peripheral Interface (SPI)	nr_spi_rxchar	Reads a character from the SPI peripheral whose address is passed as an argument.
	nr_spi_txchar	Sends a single character to the SPI peripheral whose address is passed as an argument.

Table 8. Peripheral Routines Summary

Peripheral	Routine	Description
Timer	nr_timer_milliseconds	Installs an interrupt service routine and returns zero the first time it is called. For each subsequent call, returns the number of milliseconds elapsed since the first call.
Universal Asynchronous Receiver/Transmitter (UART)	nr_uart_rxchar	Reads a character from the UART whose address is passed as an argument.
	nr_uart_txcr	Sends a carriage return and line feed to the UART at address <i>nasys_printf_UART</i> .
	nr_uart_txchar	Sends a single character to the UART whose address is passed as an argument.
	nr_uart_txhex	Prints an integer value, in hexadecimal, to the UART at address <i>nasys_printf_UART</i> .
	nr_uart_txhex16	Prints the value of a short integer, in hexadecimal, to the UART at address <i>nasys_printf_UART</i> .
	nr_uart_txhex32	Prints the value of a long integer, in hexadecimal, to the UART at address <i>nasys_printf_UART</i> .
	nr_uart_txstring	Prints a null-terminated string to the UART at address <i>nasys_printf_UART</i> .

Debug Core

Table 9. Debug Core Register Map

Index	Register Name	R/W	Description/Register Bits
0	interrupt	RO	Bit 0: dbp0 (data breakpoint 0) Bit 1: dbp1 (data breakpoint 1) Bit 2: ibp0 (instruction breakpoint 0) Bit 3: ibp1 (instruction breakpoint 1) Bit 4: mem (memory breakpoint)
1	n_samples_lsb	RO	16 least significant bits (LSBs) of the number of samples
2	n_samples_msb	RO	16 most significant bits (MSBs) of the number of samples
3	data_valid	RO	True when a trace sample is loaded into the trace registers
4	trace_address	RO	Most recently read trace address
5	trace_data	RO	Most recently read trace data
6	trace_code	RO	Bit 0: skp = 0 (skip) Bit 1: fifo_full Bit 2: bus (instruction/data) Bit 3: rw (read/write) Bit 4: intr (interrupt)
7	write_status	RO	Bit 0: skp = 1 (skip) Bit 1: fifo_full Bits 2-8: skp_cnt (skip count)
8	start	WO	Any write: start debug core
9	stop	WO	Any write: stop debug core
10	read_sample	WO	Any write: initiate transfer of next available trace sample to the trace registers
11	trace_mode	WO	Write 1: enable extended trace mode. Write 0: disable extended trace mode
12	mem_int_enable	WO	Control number of trace samples accumulated before a memory interrupt is generated If 0: no memory interrupt If > 0: shift left by 2 for number of samples to collect
13	ext_break_enable	WO	Write 1: enable external break signal. Write 0: disable external break signal
14	sw_reset	WO	Any write: reset debug trace
16	address_pattern_0	WO	Store breakpoint 0's address pattern
17	address_mask_0	WO	Store breakpoint 0's address mask
18	data_pattern_0	WO	Store breakpoint 0's data pattern
19	data_mask_0	WO	Store breakpoint 0's data mask
20	code_0	WO	Store breakpoint 0's break code Bit 0: read Bit 1: write Bit 2: fetch
24	address_pattern_1	WO	Store breakpoint 1's address pattern
25	address_mask_1	WO	Store breakpoint 1's address mask
26	data_pattern_1	WO	Store breakpoint 1's data pattern
27	data_mask_1	WO	Store breakpoint 1's data mask
28	code_1	WO	Store breakpoint 1's break code Bit 0: read Bit 1: write Bit 2: fetch

Debug Core Register Access

The debug core registers described in [Table 9](#) are accessed through Nios Control registers 3 and 4 using the WRCTL and RDCTL instructions. Control register 3 acts as an index register, while control register 4 acts as the data register.

To read a debug core register, you:

1. Write its index from the table above into control register 3.
2. Read its value from control register 4.

To write to a debug core register, you:

1. Write its index from the table above into control register 3.
2. Write the value to control register 4.

Debug Core Trace Data

Trace data is read from the core in reverse time order. That is, the first sample read from the core is the most recent sample stored and the last sample read from the core is the earliest sample stored.

Debug Core Interrupt

The debug core has a hard coded IRQ number. This number is defined in `nios.h` and can be referenced as `nasys_debug_core_irq`.

Debug Core Software Routines and Macros

The debug core routines are always present in the Nios library. The functions and macros are declared in the include file **nios.h**.

Trace data is also compressed, so the trace dumps provided by the routines below must be processed with the **tracelink** utility (see [“tracelink” on page 86](#)).

nr_debug_start

This routine resets the debug core and instructs it to begin monitoring the instruction and data bus transactions.

Syntax

```
nr_debug_start();
```

nr_debug_stop

This routine causes the debug core to stop monitoring the instruction and data busses.

Syntax

```
nr_debug_stop();
```

nr_debug_dump_trace

This routine dumps all accumulated trace samples in ASCII format out the specified serial port. The output from this function can be used by the **tracelink** utility to create a full instruction and data trace.

Syntax

```
nr_debug_dump_trace (void *uart)
```

Parameter

This parameter is maintained for compatibility with previous releases, but is now ignored. This routine always prints out the default printf uart.

nr_debug_isr_halt

This interrupt service routine for the Nios debug core can be installed with the `nr_installuserisr` routine (see “[nr_installuserisr](#)” on page 22). The `nr_installuserisr context` parameter is ignored. Trace dumps are sent out the default `printf` uart.

Once installed, this routine executes on any debug core break condition, dumping the interrupt cause followed by all trace samples accumulated to the point the interrupt service routine was called. When all trace samples are dumped, the interrupt service routine returns control to the GERMS monitor.

nr_debug_isr_continue

This interrupt service routine for the Nios debug core can be installed with the `nr_installuserisr` routine (see “[nr_installuserisr](#)” on page 22). The `nr_installuserisr context` parameter is ignored. Trace dumps are sent out the default `printf` uart.

Once installed, this routine executes on any debug core break condition, dumping the cause of the interrupt followed by all trace samples accumulated to the point the interrupt service routine was called. When all trace samples are dumped, the interrupt service routine returns control to the user program.

nm_debug_get_reg

This macro reads the value of a debug core register.

Syntax

```
nm_debug_get_reg (value,  
                  offset);
```

Parameters

Parameter Name	Description
value	The specified register's value
offset	Debug core register index

nm_debug_set_reg

This macro writes a value to a debug core register.

Syntax

```
nm_debug_set_reg (value,  
                  offset);
```

Parameters

Parameter Name	Description
value	Value to be written to the specified register's value
offset	Debug core register index

nm_debug_set_bp0 and nm_debug_set_bp1

These macros store all the register values for breakpoint 0 and 1, respectively.

Syntax

```
nm_debug_set_bp0 (address_pattern, address_mask,  
                  data_pattern, data_mask,  
                  break_code);  
nm_debug_set_bp1 (address_pattern, address_mask,  
                  data_pattern, data_mask,  
                  break_code);
```

Parameters

The debug core uses the above parameters to determine when to trigger a hardware break. The equation used is:

(Actual Bus Address & address_mask = address_pattern) &
(Actual Bus Data & data_mask = data_pattern) &
(Actual Bus Transaction & break_code!= 0)

DMA

Table 10. DMA Register Map

A2..A0	Register Name	R/W	Description/Register Bits											
			31	...	9	8	7	6	5	4	3	2	1	0
0	status ¹	RW								len	weop	reop	busy	done
1	readaddress	RW	Read master start address											
2	writeaddress	RW	Write master start address											
3	length	RW	Length in bytes											
4	reserved1	–	Reserved											
5	reserved2	–	Reserved											
6	control	RW			wcon	rcon	leen	ween	reen	i_en	go	word	hw	byte
7	reserved3	–	Reserved											

Notes

- (1) A write operation to the status register clears the len, weop, reop, and done bits.

DMA Software Data Structure

```
typedef volatile struct
{
    int np_dmastatus;           // status register
    int np_dmareadaddress;     // read address
    int np_dmawriteaddress;    // write address
    int np_dmalength;          // length in bytes
    int np_dmareserved1;       // reserved
    int np_dmareserved2;       // reserved
    int np_dmacontrol;         // control register
    int np_dmareserved3;       // reserved
} np_dma;
```

DMA Software Routines

The DMA routines are present in the Nios library when one or more DMA peripherals are present in the Nios system. These functions are declared in the include file **nios.h**.

nr_dma_copy_1_to_1

This routine transfers “transfer_count” units of data between the unchanging source address and destination address.

Syntax

```
nr_dma_copy_1_to_1
(
    np_dma *dma,
    int bytes_per_transfer,
    void *source_address,
    void *destination_address,
    int transfer_count
);
```

Parameters

Parameter Name	Description
<code>dma</code>	Which DMA peripheral to use
<code>bytes_per_transfer</code>	Must be 1, 2, or 4, but does not have to match the bus size
<code>source_address</code>	Address to transfer data from
<code>destination_address</code>	Address to transfer data to
<code>transfer_count</code>	Number of individual data transfers to perform

nr_dma_copy_1_to_range

This routine transfers “transfer_count” units of data between the source address and destination address. The same source address is used repeatedly, while the destination address increments by “bytes_per_transfer” each transaction.

Syntax

```
nr_dma_copy_1_to_range
(
    np_dma *dma,
    int bytes_per_transfer,
    void *source_address,
    void *first_destination_address,
    int transfer_count
);
```

Parameters

Parameter Name	Description
<code>dma</code>	Which DMA peripheral to use
<code>bytes_per_transfer</code>	Must be 1, 2, or 4, but does not have to match the bus size
<code>source_address</code>	Address to transfer data from
<code>first_destination_address</code>	Address to transfer data to
<code>transfer_count</code>	Number of individual data transfers to perform

nr_dma_copy_range_to_range

This routine transfers “transfer_count” units of data between the source address and destination address. Both the source address and the destination address increment by “bytes_per_transfer” each transaction.

Syntax

```
nr_dma_copy_range_to_range
(
    np_dma *dma,
    int bytes_per_transfer,
    void *first_source_address,
    void *first_destination_address,
    int transfer_count
);
```

Parameters

Parameter Name	Description
dma	Which DMA peripheral to use
bytes_per_transfer	Must be 1, 2, or 4, but does not have to match the bus size
first_source_address	Address to transfer data from
first_destination_address	Address to transfer data to
transfer_count	Number of individual data transfers to perform

nr_dma_copy_range_to_1

This routine transfers “transfer_count” units of data between the source address and destination address. The source address increments by “bytes_per_transfer” each transaction, while the same destination address is used repeatedly.

Syntax

```
nr_dma_copy_range_to_1
(
    np_dma *dma,
    int bytes_per_transfer,
    void *first_source_address,
    void *destination_address,
    int transfer_count
);
```

Parameters

Parameter Name	Description
dma	Which DMA peripheral to use
bytes_per_transfer	Must be 1, 2, or 4, but does not have to match the bus size
first_source_address	Address to transfer data from
destination_address	Address to transfer data to
transfer_count	Number of individual data transfers to perform

PIO

Table 11. PIO Register Map

A1..A0	Register Name		R/W	Variable Size—1 to 32 bits
0	data	read	RO	Data value currently on PIO inputs
		write	WO	New value to drive on PIO outputs
1	direction		RW	Data direction (optional): Individual control for each PIO bit
2	interruptmask		RW	Interrupt mask (optional): Per-bit IRQ enable/disable
3	edgecapture ¹		RW	Edge capture (optional): Per-bit synchronous edge detect and hold

Note

(1) A write operation to the edgecapture register clears all bits in register 0.

PIO Software Data Structure

```
typedef volatile struct
{
    int np_piodata;           // read/write, up to 32 bits
    int np_pioidirection;    // write/readable, up to 32 bits,
                             // 1->output bit
    int np_piointerruptmask; // write/readable, up to 32 bits,
                             // 1->enable interrupt
    int np_pioedgecapture;   // read, up to 32 bits,
                             // cleared by any write
} np_pio;
```

Example: Direct access to PIO

```
void TurnOnLEDs(void)
{
    // the reference design has a PIO named na_led_pio
    // that controls two LEDs on the development board

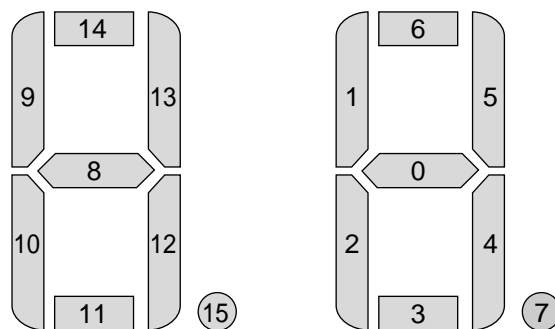
    na_led_pio->np_pioidirection = 3; // Set direction: output
    na_led_pio->np_piodata = 0;        // both LEDs off
    nr_delay(1000);                    // wait 1 second
    na_led_pio->np_piodata = 1;        // turn on first led
    nr_delay(1000);                    // wait 1 second
    na_led_pio->np_piodata = 3;        // both LEDs on
}
```

PIO Software Routine: nr_pio_showhex

The `nr_pio_showhex` routine is present in the Nios library when one or more PIO peripherals are present in the Nios system. This function is declared in the include file `nios.h`.

The `nr_pio_showhex` routine assumes a 16-bit wide PIO named "na_seven_seg_pio" is attached to a two-digit seven-segment display, in which segments are illuminated when the corresponding bits are set to 0. PIO bits are assigned to the seven-segment display elements as shown:

Figure 1. Seven-Segment Display



Syntax

```
void nr_pio_showhex(int value);
```

Parameter

The `value` parameter indicates the data to be sent to the seven-segment display.

Example

```
#include "nios.h"

void main(void)
{
    int c;

    printf("Please enter a character:\n");

    while((c = nr_uart_rxchar(0)) == -1); // wait for valid input

    nr_pio_showhex(c);
    printf("Your character is:\t%c, in hex:0x%02x\n", c, c);
}
```

SPI

Table 12 shows a register map for SPI master and slave devices with an n -bit transmit/receive shift register operating as master and slave devices.

Table 12. SPI Register Map

A2..A0	Register Name	R/W	Description/Register Bits										
			15	...	8	7	6	5	4	3	2	1	0
0	rxdata	RO	rxdata(n-1..0)										
1	txdata	WO	txdata(n-1..0)										
2	status ¹	RW			e	rrdy	trdy	tmt	toe	roe			
3	control	RW			ie	irrdy	itrdy		itoe	iroe			
4	reserved	—	Present only on master										
5	slaveselct	RW	Slave select mask—present only on master										

Notes

- (1) A write operation to the status register clears the roe, toe, and e bits.

SPI Software Data Structure

```
typedef volatile struct
{
    int np_spirxdata;           // Read-only, 1-16 bit
    int np_spitxdata;          // Write-only, 1-16 bit
    int np_spistatus;          // Read-only, 9-bit
    int np_spicontrol;         // Read/Write, 9-bit
    int np_spireserved;        // reserved
    int np_spislaveselct;      // Read/Write, 1-16 bit, master only
} np_spi;
```

SPI Software Routines

The SPI routines are present in the Nios library when one or more SPI peripherals are present in the Nios system. These functions are declared in the include file **nios.h**.

nr_spi_rxchar

This routine reads a character from the SPI peripheral whose address is passed as pSPI.

Syntax

```
int nr_spi_rxchar(np_spi *pSPI);
```

Parameter

The pSPI parameter is a pointer to the SPI peripheral.

nr_spi_txchar

This routine sends a single character, i, to the SPI peripheral whose address is passed as pSPI.

Syntax

```
int nr_spi_txchar(int i, np_spi *pSPI);
```

Parameters

Parameter Name	Description
i	Character to be sent
pSPI	Pointer to the SPI peripheral

Timer

Table 13. Timer Register Map

A2..A0	Register Name	R/W	Description/Register Bits					
			15	...	3	2	1	0
0	status	RW					run	to
1	control	RW			stop	start	cont	ito
2	periodl	RW	Timeout Period – 1 (bits 15..0)					
3	periodh	RW	Timeout Period – 1 (bits 31..16)					
4	snapl ¹	RW	Timeout Counter Snapshot (bits 15..0)					
5	snaph ¹	RW	Timeout Counter Snapshot (bits 31..16)					

Notes

- (1) A write operation to either the `snapl` or `snaph` registers updates both registers with a coherent snapshot of the current internal counter value.

Timer Software Data Structure

```
typedef volatile struct
{
    int np_timerstatus; // read only, 2 bits (any write to clear T0)
    int np_timercontrol; // write/readable, 4 bits
    int np_timerperiodl; // write/readable, 16 bits
    int np_timerperiodh; // write/readable, 16 bits
    int np_timersnapl; // read only, 16 bits
    int np_timersnaph; // read only, 16 bits
} np_timer;
```

Example: Direct access to Timer

```
#include "nios.h"

int main(void)
{
    int t = 0;

    // Set timer for 1 second
    na_timer1->np_timerperiodl = (short)(nasys_clock_freq & 0x0000ffff);
    na_timer1->np_timerperiodh = (short)((nasys_clock_freq >> 16) & 0x0000ffff);

    // Set timer running, looping, no interrupts
    na_timer1->np_timercontrol = np_timercontrol_start_mask + np_timercontrol_cont_mask;

    // Poll timer forever, print once per second
    while(1)
    {
        if(na_timer1->np_timerstatus & np_timerstatus_to_mask)
        {
            printf("A second passed! (%d)\n",t++);

            // Clear the to (timeout) bit
            na_timer1->np_timerstatus = 0; // (any value)
        }
    }
}
```

Timer Software Routine: `nr_timer_milliseconds`

The `nr_timer_milliseconds` routine is present in the Nios library when one or more Timer peripherals are present in the Nios system. This function is declared in the include file **`nios.h`**.

This routine requires the existence of a Timer called `timer1`, with a base address defined by `na_timer1` and an interrupt number defined by `na_timer1_irq`. The first time this routine is called, it installs an interrupt service routine for the Timer and returns zero. For each subsequent call, the number of milliseconds elapsed since the first call is returned.

Syntax

```
int nr_timer_milliseconds(void);
```

UART

Table 14. UART Register Map

A2..A0	Register Name	R/W	Description/Register Bits															
			15	...	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	rxdata	RO									RxData							
1	txdata	WO									TxData							
2	status ¹	RW			eop	cts	dcts	–	e ²	rrdy	trdy	tmt	toe	roe	brk	fe	pe	
3	control	RW			ieop	rts	idcts	trbk	ie	irrdy	itrdy	itmt	itoe	iroe	ibrk	ife	ipe	
4	divisor	RW	Baud Rate Divisor (optional)															
5	endofpacket	RW									End-packet value							

Notes

- (1) A write operation to the status register clears the dcts, e, toe, roe, brk, fe, and pe bits.
- (2) status register bit 8 (e) is the logical OR of the toe, roe, brk, fe, and pe bits.

UART Software Data Structure

```
typedef volatile struct
{
    int np_uartrxdata;        // Read-only, 8-bit
    int np_uarttxdata;        // Write-only, 8-bit
    int np_uartstatus;        // Read-only, 9-bit
    int np_uartcontrol;       // Read/Write, 9-bit
    int np_uartdivisor;       // Read/Write, 16-bit, optional
    int np_uartendofpacket;   // Read/Write, end of packet character
} np_uart;
```

UART Software Routines

The UART routines are present in the Nios library when one or more UART peripherals are present in the Nios system. These functions are declared in the include file **nios.h**.

nr_uart_rxchar

This routine reads a character from the UART peripheral whose address is passed in `uartBase`. If no character is waiting, `nr_uart_rxchar` returns -1. If zero is passed for the peripheral address, `nr_uart_rxchar` reads a character from the UART at location `nasys_printf_uart` (**nios.h**).

Syntax

```
int nr_uart_rxchar(np_uart *uartBase);
```

Parameter

The `uartBase` parameter is a pointer to the UART peripheral.

Example

```
#include "nios.h"

void main(void)
{
    int c;

    printf("Please enter a character:\n");

    while((c = nr_uart_rxchar(nasys_printf_UART)) == -1)
        ; // wait for valid input

    printf("Your character is:\t%c\n", c);
}
```

nr_uart_txchar

This routine sends a single character, `c`, to the UART peripheral whose address is passed as `uartBase`. If zero is passed for the peripheral address, `nr_uart_txchar` sends a character to the UART at location `nasys_printf_uart` (defined in **nios.h**).

Syntax

```
int nr_uart_txchar(int c, np_uart *uartBase);
```

Parameters

Parameter Name	Description
<code>c</code>	Character to be sent
<code>uartBase</code>	Pointer to the UART peripheral

Example

```
#include "nios.h"

#define kLineWidth 77
#define kLineCount 100

void SendLots(void)
{
    char c;
    int i,j;
    int mix;

    printf("\n\nPress character, or <space> for mix: ");
    while((c = nr_rxchar(0)) < 0);

    printf("%c\n\n",c);

    // Don't show unprintables

    if(c < 32)
        c = '.';

    mix = c==' ';

    for(i = 0; i < kLineCount; i++)
    {
        for(j = 0; j < kLineWidth; j++)
        {
            if(mix)
            {
                c++;
                if(c >= 127)
                    c = 33;
            }
            nr_uart_txchar(c,nasys_printf_UART);
            // send character to UART
        }
        nr_uart_txcr();
        // send carriage return and new line
    }
    printf("\n\n");
}
```

nr_uart_txcr

This routine sends a carriage return and line feed to the UART at location `nasys_printf_uart` (defined in **nios.h**).

Syntax

```
int nr_uart_txcr(void);
```

nr_uart_txhex

This routine prints the integer value of `x` in hexadecimal to the UART at location `nasys_printf_uart` (defined in **nios.h**). The range for a 16-bit Nios CPU is 0000 to FFFF, and for a 32-bit Nios CPU is 00000000 to FFFFFFFF.

Syntax

```
int nr_uart_txhex(int x);
```

Parameter

The `x` parameter is an integer value to be sent to UART.

nr_uart_txhex16

This routine prints the 16-bit value of `x` in hexadecimal to the UART at location `nasys_printf_uart` (defined in **nios.h**). The range is from 0000 to FFFF.

Syntax

```
int nr_uart_txhex16(short x);
```

Parameter

The `x` parameter is a 16-bit integer value to be sent to UART.

nr_uart_txhex32

This routine prints the 32-bit value of *x* in hexadecimal to the UART at location `nasys_printf_uart` (defined in **nios.h**). The range is from 00000000 to FFFFFFFF. This routine is not available on a 16-bit Nios CPU.

Syntax

```
int nr_uart_txhex32(long x);
```

Parameter

The *x* parameter is a 32-bit integer value to be sent to UART.

nr_uart_txstring

This routine prints the null-terminated string *s* to the UART at location `nasys_printf_uart` (defined in **nios.h**).

Syntax

```
int nr_uart_txstring(char *s);
```

Parameter

The *s* parameter is a pointer to a null-terminated character string.

Nios Software Development Utilities

The GNUPro software tools included in the Nios development kit contain several general-purpose software development utilities, including the Nios SDK Shell command line. Nios software is developed in the bash environment. The SDK Shell provides a UNIX-like environment on a PC platform, including most of the commands and utilities UNIX users are accustomed to using. For details, enter “man bash” at the shell prompt found at C:\Altera\Excilibur\sopc_builder_2_5\Nios SDK Shell.

Additionally, many Nios-specific utilities are included in the development kit for generating and debugging software. This chapter provides detailed descriptions of these utilities.

Table 15. Nios Utilities

Utility Name	Description
hexout2flash	Perl script that converts a Quartus II .hexout file (device configuration file) to a .hexout.flash file suitable for writing to flash memory on the Nios development board
nios_bash	Startup script to set the bash environment for Nios development (bash shell)
nios-build	Perl script that performs compilation and assembly of source files, links to Nios library, and generates .srec file, suitable to download to the Nios development board (see nios-run)
nios-convert	Perl script to convert .srec files to .mif or .dat file format suitable for initializing on-chip memory
nios_csh	Startup script to set the bash environment for Nios development (C shell)
nios-elf-as	GNU assembler for Nios
nios-elf-gcc	GNU C/C++ compiler for Nios
nios-elf-gdb	GNU debugger for Nios
nios-elf-gprof	GNU C program execution profiler
nios-elf-ld	GNU linker for Nios
nios-elf-nm	GNU tool to extract symbols from Nios object files
nios-elf-objcopy	GNU utility to convert linker output (.out) to S-records (.srec)
nios-elf-objdump	GNU tool to disassemble Nios object files
nios-elf-size	GNU utility to produce an object file size report for code (text), data (data), and uninitialized storage (bss).
nios-run	Utility for downloading and running a user .srec file, by performing terminal I/O
srec2flash	Perl script that converts a .srec file to a .flash file, suitable for writing to the Nios development board flash (software)
tracelink	Associates a Nios object file and a trace dump file to generate an assembly listing of all instructions traced, including all data accesses, skipped instructions, and interrupts.

hexout2flash

The Quartus II and MAX+PLUS II software generate configuration files for download to an Altera PLD. One configuration file format generated by Quartus II is **.hexout**. **hexout2flash** converts a **.hexout** file to a **.flash** file, suitable for writing to the flash device on the Nios development board. **hexout2flash** creates a sequence of GERMS monitor commands to erase a section of flash memory and relocate the **.hexout** file to the erased section.



See the *Nios Development Board Data Sheet* for details on the Nios development board.

Usage

```
hexout2flash [options] <filename>[.hexout]
```

Options

Table 16. hexout2flash Options

Option	Description
-b <base address>	Location in flash to write file (default 0x180000)
--help	Print help

Example

1. For a file called **my_design.hexout**, enter:

```
hexout2flash my_design.hexout
```

hexout2flash converts **my_design.hexout** to **my_design.hexout.flash**.

2. To download the **.flash** file to the development board, enter:

```
nios-run my_design.hexout.flash
```

The design is written into flash memory at location 0x180000 and becomes the default booting design for the development board.

nios_bash

nios_bash is a startup script that properly sets the **bash** shell environment for software development using **nios-build**. **nios-build** requires two shell variables to exist and be exported. A normal Windows install of the Nios development utilities sets up these variables automatically. The shell variables are:

- `niosgnu = <Nios GNU tools location>`
The default location is
`/altera/excalibur/sopc_builder_2_5/bin/nios-gnupro`
- `niosbin = <Nios bin location>`
The default location is `/altera/excalibur/sopc_builder_2_5/bin`

Usage

Source this script from the **.bash_profile** at shell startup time. It adds the paths and shell variables needed to use the Nios tools.

nios-build

nios-build is a Perl script that invokes the tools to compile, assemble, and link Nios source code. It ensures the standard C libraries and standard Nios libraries are linked with the user source code, and the associated “include” paths are available. Most programs compile with no command line options; reasonable defaults are assumed.



nios-build is a simple alternative to the Makefile. Use of Makefiles is fully supported by the Nios software development environment. For an example Makefile, see [.../lib/Makefile](#). For details on using Makefiles, see the GNU on-line documentation by choosing **Programs > Cygwin > Cygwin Documentation** (Windows Start Menu). In the help window that appears, click **Using make**.

nios-build produces a file with the base name of the last source file on the command line and the suffix **.srec**. The file is ready for downloading to the Nios development board, which must have the GERMS monitor running.

Source files are listed on the command line following the options. If only one source file is specified, **nios-build** searches the current directory for files with the same base name and underscore extensions.

Files ending with **.s** or **.asm** are passed to **nios-elf-as**. Files ending with **.c** are passed to **nios-elf-gcc**. Files ending with **.o** are passed to **nios-elf-ld**.

Usage

```
nios-build [options] <sourcefile>.[sco]
```


Options

Table 17. nios-build Options

Option	Description
-b <base address>	Set code base address
-m16	Generate code for Nios 16
-m32	Generate code for Nios 32 (default)
-as <quoted string>	Pass command line options to assembler
-cc <quoted string>	Pass command line options to compiler
-ld <quoted string>	Pass command line options to linker
-d	Set NIOS_GDB=1 and generate debug script
-s	Silent mode (only print errors)
-l <file name>	Include system library
-o <file name>	Output file name
--help	Print help
--help 1	Print more help

Example

```
nios-build foo.c bar.s
```

Multiple files listed in the command line, as shown above, generate the executable file **bar.srec**.

```
nios-build helloworld.c
```

If the files **helloworld_2.c** and **helloworld_3.s** are in the same directory, they are included in the build and the result is **helloworld.srec**.

nios-convert

nios-convert is a Perl script that converts files from one format to another. Source files can be **.srec** or **.mif**; destination files can be **.mif** or **.dat**.

nios-convert's primary functions are:

- Convert executable software code or data files (**.srec** format) to initialization files for on-chip memory (**.mif** format) in the Altera PLD.
- Convert the data width. This is useful, for example, to store 32-bit data in an off-chip 16-bit flash.
- Break wide data into multiple byte lanes. This is useful, for example, to break 32-bit data into two lanes of 16-bit data to write into two off-chip 16-bit flash memories used in parallel.

Destination files are named the same as the source file if no destination file name is specified.

Usage

```
nios-convert [options] <source file> [destFile]
```

Options

Table 18. nios-convert Options

Option	Description
--lanes=x	Break into multiple output files lane_0 .. _lane_(x-1) appended
--width=x	Set output width to 8, 16, or 32
--oformat=f	Format can be mif or dat
--comments=b	Comments in mif file enabled (1) or disabled (0). Default is enabled.
--help	Print help

Example

```
nios-convert bootcode.srec bootcode.mif
```

converts file **bootcode.srec** to **bootcode.mif**.

nios_csh

nios_csh is a startup script that properly sets the C shell environment for software development using **nios-build**.

Usage

Source this script from the **.login** at shell startup time.

Example

```
source /altera/excalibur/sopc_builder_2_5/bin/nios_csh
```

If the **.../altera/** directory is at a location other than **/usr/altera**, assign that location to the shell variables “altera”. For example:

```
set altera = /downloads/altera
source /downloads/altera/excalibur/nios-sdk/nios_bash
```

nios-elf-as

nios-elf-as is a Nios assembler that produces a relocatable object file from assembly language source code. The object file contains the binary code and debug symbols.

If you use **nios-build** to generate executable code from assembly source, **nios-elf-as** is invoked automatically. It may be useful, however, to have a working knowledge of the assembler command line options to help optimize your assembly source code.

Usage

```
nios-elf-as [option...] [asmfile...]
```

Options

Table 19. nios-elf-as Options

Option	Description
-a[sub-option...]	Turn on listings
Sub-Options	
c	Omit false conditionals
d	Omit debugging directives
h	Include high-level source
l	Include assembly
m	Include macro expansions
n	Omit forms processing
s	Include symbols
L	Include line debug statistics
=file	Set listing file name (must be last sub-option)
-D	Produce assembler debugging messages
--defsym SYM=VAL	Define symbol SYM to given value
-f	Skip white space and comment preprocessing
--gstabs	Generate STABS debugging information
--gdwarf2	Generate DWARF2 debugging information
--help	Show this message and exit
-I DIR	Add DIR to search list for .include directives
-J	Do not warn about signed overflow
-K	Warn when differences altered for long displacements
-L	Keep local symbols (such as starting with "L")
--keep-locals	

Table 19. nios-elf-as Options

Option	Description
-M --mri	Assemble in MRI compatibility mode
--MD <file>	Write dependency information in <file> (default none)
-nocpp	Ignored
-o <objfile>	Name the object file output <objfile> (default a.out)
-R	Fold data section into text section
--statistics	Print various measured statistics from execution
--strip-local-absolute	Strip local absolute symbols
--traditional-format	Use same format as native assembler when possible
--version	Print assembler version number and exit
-W --no-warn	Suppress warnings
--warn	Do not suppress warnings
--fatal-warnings	Treat warnings as errors
--itbl <insttbl>	Extend instruction set to include instructions matching the specifications defined in file <insttbl>
-w	Ignored
-X	Ignored
-Z	Generate object file even after errors
--listing-lhs-width	Set width in words of the output data column of the listing
--listing-lhs-width2	Set width in words of the continuation lines of the output data column; ignored if smaller than first line's width
--listing-rhs-width	Set max width in characters of the lines from the source file
--listing-cont-lines	Set maximum number of continuation lines used for the output data column of the listing

Table 20. Nios-Specific Command Line Options

Option	Description
-m16	Nios-16 processor (16-bit)
-m32	Nios-32 processor (32-bit)



For more information on using the GNU assembler, see the on-line documentation by choosing **Programs > Cygwin > Cygwin Documentation** (Windows Start Menu). In the help window that appears, click **Using as**.

nios-elf-gcc

The GNU compiler invokes the necessary utilities:

Utility	Description
cpp	C preprocessor that processes all the header files and macros the target requires
gcc	Compiler that produces assembly language code from the processed C files
as	Assembler that produces binary code from the assembly language source code and puts it in an object file
ld	Linker that binds the code to addresses, links the startup file and libraries to the object code, and produces the executable binary image

If you use **nios-build** to generate executable code, **nios-elf-gcc** is invoked automatically. It may be useful, however, to have a working knowledge of the C compiler command line options to help optimize your C code.

Usage

```
nios-elf-gcc [options] file...
```

Options

Table 21. nios-elf-gcc Options

Option	Description
-pass-exit-codes	Exit with highest error code from a phase
--help	Display this information (Enter “-v --help” to display command line options of sub-processes)
-dumpspecs	Display all built-in Spec strings
-dumpversion	Display compiler version
-dumpmachine	Display compiler's target processor
-print-search-dirs	Display directories in the compiler's search path
-print-libgcc-file-name	Display compiler's companion library name
-print-file-name=<lib>	Display full path to library <lib>
-print-prog-name=<prog>	Display full path to compiler component <prog>
-print-multi-directory	Display root directory for versions of libgcc
-print-multi-lib	Display mapping between command line options and multiple library search directories
-Wa,<options>	Pass comma-separated <options> onto assembler
-Wp,<options>	Pass comma-separated <options> onto preprocessor

Table 21. nios-elf-gcc Options

Option	Description
-Wl,<options>	Pass comma-separated <options> onto linker
-Xlinker <arg>	Pass <arg> onto linker
-save-temps	Do not delete intermediate files
-pipe	Use pipes rather than intermediate files
-time	Time the execution of each subprocess
-specs=<file>	Override built-in specs with contents of <file>
-std=<standard>	Assume input sources are for <standard>
-B <directory>	Add <directory> to compiler's search paths
-b <machine>	Run gcc for target <machine>, if installed
-V <version>	Run gcc version number <version>, if installed
-v	Display programs invoked by compiler
-E	Preprocess only; do not compile, assemble, or link
-S	Compile only; do not assemble or link
-c	Compile and assemble, but do not link
-o <file>	Place output into <file>
-x <language>	Specify language of the following input files. Permissible languages are "c", "c++", "assembler", and "none" (deduce language based on file extension).
-pg	Compile with profiling

Options starting with -g, -f, -m, -O or -W are automatically passed onto the sub-processes invoked by **nios-elf-gcc**. To pass other options onto these processes, the -W<letter> options must be used.

Table 22 lists Nios-specific options for **nios-elf-gcc**. The variable *x* used in this table can be 0, 1, 2, 3, or 4. The variable *w* can be any number from -1024 through 2047.

Table 22. Nios-Specific Options for nios-elf-gcc

Option	Description
-m16 -m32	Generate output for Nios 16 or Nios 32.
-mfewer-opcodes	Do not generate the opcodes LDS, LDP, STS, STP, STS8S, ST8S, STS16S, and ST16S.
-mmax-address=HEXADDR	Do not generate unnecessary PFX 0 and/or MOVHI 0 opcodes. When HEXADDR <= 0xffff, PFX/MOVHI instruction pairs will not be generated for addresses. When 0x10000 <= HEXADDR <= 0x1fffff, PFX instructions will not be generated before MOVHI instructions for addresses. HEXADDR is a hexadecimal address between 0 and fffffff, optionally prefixed by "0x".
-muser-opcode-mul=pf _{x_w} usr _x -muser-opcode-mul=usr _x	Generate USR _x instructions for signed integer multiplication. For example: <pre>int result, dataa, datab; result = dataa * datab;</pre> A non-zero prefix is optional.
-muser-opcode-div=pf _{x_w} usr _x -muser-opcode-div=usr _x	Generate USR _x instructions for signed integer division. For example: <pre>int result, dataa, datab; result = dataa / datab;</pre> A non-zero prefix is optional.
-muser-opcode-udiv=pf _{x_w} usr _x -muser-opcode-udiv=usr _x	Generate USR _x instructions for unsigned integer division. For example: <pre>unsigned int result, dataa, datab; result = dataa / datab;</pre> A non-zero prefix is optional.
-muser-opcode-mod=pf _{x_w} usr _x -muser-opcode-mod=usr _x	Generate USR _x instructions for signed integer modulus. For example: <pre>int result, dataa, datab; result = dataa % datab;</pre> A non-zero prefix is optional.
-muser-opcode-umod=pf _{x_w} usr _x -muser-opcode-umod=usr _x	Generate USR _x instructions for unsigned integer modulus. For example: <pre>unsigned int result, dataa, datab; result = dataa % datab;</pre> A non-zero prefix is optional.
-muser-opcode-extv=usr _x	Generate prefixed USR _x instructions for signed bit-field extraction. The prefix instruction's 11-bit immediate value is 1wwwppppp, where ppppp is the bit position of the rightmost bit of the field to extract (LSB = 0), and wwwwww is the field's width in bits. USR _x must sign extend the extracted bit-field value to a full integer.

Table 22. Nios-Specific Options for nios-elf-gcc

Option	Description
-muser-opcode-extzv=usr _x	Generate prefixed USR _x instructions for unsigned bit-field extraction. The prefix instruction's 11-bit immediate value is 0wwwwwppppp, where ppppp is the bit position of the rightmost bit of the field to extract (LSB = 0), and wwwwww is the field's width in bits. USR _x must zero-extend the extracted bit-field value to a full integer.
-muser-opcode-insv=usr _x	Generate prefixed USR _x instructions for bit-field insertion. The prefix instruction's 11-bit immediate value is 0wwwwwppppp, where ppppp is the bit position of the rightmost bit of the field to insert (LSB = 0), and wwwwww is the field's width in bits.
-muser-opcode-umax=pfx _w usr _x -muser-opcode-umax=usr _x	Generate USR _x instructions for unsigned integer maximum. C++ only.
-muser-opcode-smax=pfx _w usr _x -muser-opcode-smax=usr _x	Generate USR _x instructions for signed integer maximum. C++ only.
-muser-opcode-umin=pfx _w usr _x -muser-opcode-umin=usr _x	Generate USR _x instructions for unsigned integer minimum. C++ only.
-muser-opcode-smin=pfx _w usr _x -muser-opcode-smin=usr _x	Generate USR _x instructions for signed integer minimum. C++ only.
-muser-opcode-ffs=pfx _w usr _x -muser-opcode-ffs=usr _x	Generate USR _x instructions for the internal ffs() function. ffs() finds the first set bit of its int operand starting from the right, and returns that bit position, incremented by 1. The USR _x instruction must return 0 for an input of 0.



For more details on using the GNU compiler, refer to the on-line documentation by choosing **Programs > Cygwin > Cygwin Documentation** (Windows Start Menu). In the help window that appears, click **Using GNU CC**.

nios-elf-gdb

The GNU debugger (GDB) shows either what is going on inside another program while it executes, or what another program was doing the moment it stopped. GDB can:

- Start the program and specify anything that might affect its behavior
- Stop the program based on a set of specific conditions
- Examine what happened once the program is stopped
- Change the program to fix bugs and continue testing

Use GDB to debug programs written in assembly, C, and C++.

Usage

To debug a program using **nios-build** and **nios-elf-gdb**:

- ✓ Use **nios-build** with the “-d” command line option.

nios-build produces a file with the extension **.gdb**, which is a shell script for downloading the program and running **nios-elf-gdb**. If your design includes separate serial ports for host communication and debug communications, **nios-build -d** will assign COM1 for host communication and COM2 for serial communication. You can override these assignments using the following command line options:

-d=<debug com port>

-p=<host com port>



In previous versions of the Nios processor, to debug a program using **nios-build** and **nios-elf-gdb**, a line with “**NIOS_GDB_SETUP**” was required as the first statement in the **main()** routine.

Options

Table 23. nios-elf-gdb Options

Option	Description
--[no]async	Enable (disable) asynchronous version of CLI
-b <baudrate>	Set serial port baud rate used for remote debugging
--batch	Exit after processing options
--cd=<dir>	Change current directory to <dir>
--command=<file>	Execute GDB commands from <file>
--core=<corefile>	Analyze the core dump <corefile>

Table 23. nios-elf-gdb Options

Option	Description
--dbx	DBX compatibility mode
-d [=<COM port>]	Set NIOS_GDB=1, generate debug script and optionally assign the com port for debug communication
--directory=<dir>	Search for source files in <dir>
--epoch	Output information used by epoch emacs-GDB interface
--exec=<execfile>	Use <execfile> as the executable
--fullname	Output information used by emacs-GDB interface
--help	Print this message
--interpreter=<interp>	Select a specific interpreter/user interface
--mapped	Use mapped symbol files if supported on this system
--nw	Do not use a window interface
--nx	Do not read gdb.ini file
-p=<COM port>	Only use with -d option to set COM port for host communication
--quiet	Do not print version number on startup
--readnow	Fully read symbol files on first access
--se=<file>	Use <file> as symbol file and executable file
--symbols=<symfile>	Read symbols from <symfile>
--tty=<tty>	Use <tty> for input/output by the program being debugged
--version	Print version information and then exit
-w	Use a window interface
--write	Set writing into executable and core files
--xdb	XDB compatibility mode



For more information, type **help** from within GDB, or consult the GDB manual (available as on-line information or a printed manual).

For more details on using the GNU debugger, refer to the on-line documentation by choosing **Programs > Cygwin > Cygwin Documentation** (Windows Start Menu). In the help window that appears, click **Debugging with GDB**.

nios-elf-gprof

nios-elf-gprof produces an execution profile of a C program.

Usage

```
nios-elf-gprof [option(s)] [objfile] gmon.out
```

Options

Table 24. nios-elf-gprof Options

Option	Description
-a	Suppresses printing of statically declared functions
-b	Suppresses printing of a description of each field in the profile
-c	The static call graph of the program is discovered by a heuristic that examines the text space of the object file. Static-only parents or children are shown with call counts of 0.
-e <name>	Suppresses printing of the graph profile entry for routine name and all its descendants (unless they have other unsuppressed ancestors)
-E <name>	Suppresses printing of graph profile entry for routine name and its descendants, and excludes the time spent in <name> and its descendants from the total and percentage time computations
-f <name>	Prints graph profile entry of only the specified routine name and its descendants
-F <name>	Prints graph profile entry of only the routine name and its descendants, and uses only the times of the printed routines in total time and percentage computations
-k <fromname> <toname>	Deletes any arcs from routine <fromname> to routine <toname>
-s	Produces a profile file gmon.sum that represents the sum of the profile information in all specified profile files
-v	Prints gprof version number and exits
-z	Displays routines with zero usage (as shown by call counts and accumulated time)

Example

1. To see a profile for a C program, compile it with the “-pg” gcc option. For example, to profile **hello_world.c**, enter:

```
nb hello_world.c -cc -pg
```

2. In addition to the normal text output when it is run, the program outputs hexadecimal bytes preceded by “###”. These bytes must be converted into a binary file to be fed to the **nios-elf-gprof** profiling program. For example, enter:

```
nr hello_world.srec | tee hello_world.txt
```

The output is:

```
nios-run: Downloading.....
.....
.....
.....
.....
.....
.....
nios-run: Terminal mode (Control-C exits)
-----

Hello from Nios.

### 00 01 04 00 8c 2e 04 00 d2 16 00 00 00 00 00 00
### 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
### 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
### 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
### 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
### 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
### 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
...
### 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
### 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
### 00 00 *
### 88 01 04 00 20 01 04 00 03 00 00 00 00 *
### dc 01 04 00 20 01 04 00 01 00 00 00 00 *
### f4 13 04 00 5c 01 04 00 01 00 00 00 00 *
```

The file **hello_world.txt**, which captures the above text, is created.

The utility **nios-gprof-convert** is a Perl script that strips away the numbers followed by **###** and converts them to binary. The result is saved in the file **gmon.out**. This utility takes one argument (*<filename>*) as its only input.

3. Enter:

```
nios-gprof-convert hello_world.txt
```

The output is:

```
Nios Gprof Conversion Utility
Input file: hello_world.txt
Output file: gmon.out
```

4. **nios-elf-gprof** requires the files **hello_world.out** and **gmon.out**.
Enter:

```
nios-elf-gprof -C -q hello_world.out --traditional gmon.out > hello_world.profile
```

This command uses the objfile **hello_world.out** to gather the symbols and interpret **gmon.out**.

The end result is in **hello_world.profile**. The output is:

```
time is in ticks, not seconds
```

```
call graph profile:
```

```
The sum of self and descendents is the major sort for this listing.
function entries:
```

```

      .
      .
granularity: each sample hit covers 4 byte(s) for 4.76% of 21.00 seconds

```

index	%time	self called/total	descendents called/total	called+self children	name	index
[1]	61.9	13.00	0.00		<spontaneous> txCharWait [1]	
[2]	23.8	5.00	0.00		<spontaneous> nr_uart_txchar [2]	
[3]	9.5	2.00	0.00		<spontaneous> PrivatePrintf [3]	
[4]	4.8	1.00	0.00		<spontaneous> profile_on [4]	
[5]	0.0	0.00	0.00	1/1	done_calling_constructors [20]	
				1	main [5]	

```

-----
Index by function name
[3] PrivatePrintf      [2] nr_uart_txchar      [1] txCharWait
[5] main              [4] profile_on

```



The Perl script **nios-run-gprof**, which automatically executes all the above steps, is also provided in the Nios SDK.

Advanced Usage

In some instances, the default settings are not suitable to profile user code. You can manipulate the sampling rate and code chunk size parameters by editing `nios_gprof.c` in the `.../sdk/lib` directory. To do this, some understanding of the profiling method is necessary.

In profiling, the compiler adds a call to `_mcount` at the beginning of each function, including `main()`, and interrupts the user code at a specific rate. The first call to `_mcount()` sets up all the data structures, buffers, and the interrupt service routine. Subsequent calls trace the calling sequence. While the profiled program is running, the timer (`timer1`) interrupts at its rate (default is 10,000 interrupts per second), and increments a counter corresponding to the interruptee's PC. Each code chunk of size `HISTFRACTION` has a corresponding 16-bit counter. The counter starts initialized to zero and increments each time the timer interrupt routine interrupts this code chunk (the `HISTFRACTION` default value is 2, which will at least double the memory size requirement).

Some potential problems and their possible solutions are:

- Interrupt rate is too fast, causing counters to overflow
Solution: decrease sampling rate (`TIMER_SAMPLE_RATE` constant) specified in interrupts per second
- Interrupt rate is too slow, causing non-repeatable, coarse results.
Solution: increase sampling rate
- Out of memory message appears when allocating buffers
Solution: increase code chunk size (`HISTFRACTION` constant) by powers of 2



As code chunk size increases, resolution decreases, thus the wrong counter may be incremented.

To implement changes, edit `.../sdk/lib/nios_gprof.c`. Recreate the library by typing `"make all"`.



You can exclude code from profiling by compiling different modules with or without the `-pg` option. For example, if a program consists of `my_main.c`, `mod_1.c`, and `mod_2.c`, and the critical elements to profile are in `mod_1.c`, compile the modules `my_main.c` and `mod_2.c` *without* the `-pg` option, and compile `mod_1.c` *with* the `-pg` option.

nios-elf-ld

The GNU linker resolves the code addresses and debug symbols, links the startup code and additional libraries to the binary code, and produces an executable binary image.

If you use **nios-build** to generate executable code, **nios-elf-ld** is invoked automatically. It may be useful, however, to have a working knowledge of the linker command line options.

Usage

```
nios-elf-ld [options] file...
```

Options

Table 25. nios-elf-ld Options

Option	Description
-a <keyword>	Shared library control for HP/UX compatibility
-A <arch> --architecture <arch>	Set architecture
-b <target> --format <target>	Specify target for following input files
-c <file> --mri-script <file>	Read MRI format linker script
-d -dc -dp	Force common symbols to be defined
-e <address> --entry <address>	Set start address
-E --export-dynamic	Export all dynamic symbols
-EB	Link big-endian objects
-EL	Link little-endian objects
-f <shlib> --auxiliary <shlib>	Auxiliary filter for shared object symbol table objects
-F <shlib> --filter <shlib>	Filter for shared object symbol table
-g	Ignored
-G <size> --gpsize <size>	Small data size (if no size, same as --shared)
-h <filename> -soname <filename>	Set internal name of shared library

Table 25. nios-elf-ld Options

Option	Description
-l <libname> --library <libname>	Search for library <libname>
-L <directory> --library-path <directory>	Add <directory> to library search path
-m <emulation>	Set emulation
-M --print-map	Print map file on standard output
-n --nmagic	Do not page align data
-N --omagic	Do not page align data, do not make text read only
-o <file> --output <file>	Set output file name
-O	Optimize output file
-Qy	Ignored for SVR4 compatibility
-r -i --relocateable	Generate relocatable output
-R <file> --just-symbols <file>	Just link symbols (if directory, same as --rpath)
-s --strip-all	Strip all symbols
-S --strip-debug	Strip debugging symbols
-t --trace	Trace file opens
-T <file> --script <file>	Read linker script
-u <symbol> --undefined <symbol>	Start with undefined reference to <symbol>
-Ur	Build global constructor/destructor tables
-v --version	Print version information
-V	Print version and emulation information
-x --discard-all	Discard all local symbols
-X --discard-locals	Discard temporary local symbols

Table 25. nios-elf-ld Options

Option	Description
-y <symbol> --trace-symbol <symbol>	Trace mentions of <symbol>
-Y <path>	Default search path for Solaris compatibility
-z <keyword>	Ignored for Solaris compatibility
-(--start-group	Start a group
) --end-group	End a group
-assert <keyword>	Ignored for SunOS compatibility
-Bdynamic -dy -call_shared	Link against shared libraries
-Bstatic -dn -non_shared -static	Do not link against shared libraries
-Bsymbolic	Bind global references locally
--check-sections	Check section addresses for overlaps (default)
--no-check-sections	Do not check section addresses for overlaps
--cref	Output cross reference table
--defsym <symbol>=<expression>	Define a symbol
--demangle	Demangle symbol names
--dynamic-linker <program>	Set the dynamic linker to use
--embedded-relocs	Generate embedded relocs
--errors-to-file <file>	Save errors to <file> instead of printing to stderr
-fini <symbol>	Call <symbol> at unload-time
--force-exe-suffix	Force generation of file with .exe suffix
--gc-sections	Remove unused sections (on some targets)
--no-gc-sections	Do not remove unused sections (default)
--help	Print option help
-init <symbol>	Call <symbol> at load-time
-Map <file>	Write a map file
--no-demangle	Do not demangle symbol names
--no-keep-memory	Use less memory and more disk I/O
--no-undefined	Allow no undefined symbols
--no-warn-mismatch	Do not warn about mismatched input files
--no-whole-archive	Turn off --whole-archive
--noinhibit-exec	Create an output file even if errors occur

Table 25. nios-elf-ld Options

Option	Description
--oformat <target>	Specify target of output file
-qmagic	Ignored for Linux compatibility
--relax	Relax branches on certain targets
--retain-symbols-file <file>	Keep only symbols listed in <file>
-rpath <path>	Set runtime shared library search path
-rpath-link <path>	Set link time shared library search path
-shared	Create a shared library
-Bshareable	
--sort-common	Sort common symbols by size
--split-by-file	Split output sections for each file
--split-by-reloc <count>	Split output sections every <count> relocs
--stats	Print memory usage statistics
--task-link <symbol>	Do task level linking
--traditional-format	Use same format as native linker
-Tbss <address>	Set address of .bss section
-Tdata <address>	Set address of .data section
-Ttext <address>	Set address of .text section
--verbose	Output lots of information during link
--version-script <file>	Read version information script
--version-exports-section <symbol>	Take export symbols list from .exports, using <symbol> as the version
--warn-common	Warn about duplicate common symbols
--warn-constructors	Warn if global constructors/destructors are seen
--warn-multiple-gp	Warn if the multiple GP values are used
--warn-once	Warn only once per undefined symbol
--warn-section-align	Warn if start of section changes due to alignment
--whole-archive	Include all objects from following archives
--wrap <symbol>	Use wrapper functions for <symbol>
--mpc860c0 =<words>	Modify problematic branches in last <words> (1–10, default 5) words of a page

The **nios-elf-ld** supported targets are:

- elf32-nios
- elf32-little
- elf32-big
- srec
- symbolsrec
- tekhex
- binary
- ihex

The **nios-elf-ld** supported emulations are:

- elfnios16
- elfnios32

There are no **nios-elf-ld** emulation-specific options.



For more details on using the GNU linker, refer to the on-line documentation by choosing **Programs > Cygwin > Cygwin Documentation**. In the help window that appears, click **Using ld**.

nios-elf-nm

nios-elf-nm lists public symbols and their values from object files.

Usage

```
nios-elf-nm [options] [file...]
```

Options

Table 26. nios-elf-nm Options

Option	Description
-A -o --print-file-name	Precede each symbol with the name of the input file where it was found
-a --debug-syms	Display debugger-only symbols
-B	Same as --format=bsd
-C --demangle	Decode low-level symbol names into user-level names
-D --dynamic	Display dynamic symbols rather than the normal symbol
-f <format>	Use output format <format> ("bsd", "sysv", or "posix")
-g --extern-only	Display only external symbols
-n -v --numeric-sort	Sort symbols numerically by address, not alphabetically
-p --no-sort	Do not sort symbols
-P --portability	Use POSIX.2 standard output format instead of default format
-s --print-arnmap	When listing symbols from archive members, include index
-r --reverse-sort	Reverse sort order
--size-sort	Sort symbols by size
-t <radix> --radix=<radix>	Use <radix> ("d" for decimal, "o" for octal, or "x" for hexadecimal) as the radix for printing symbol values
--target=<bfdname>	Specify object code format other than default format
-u --undefined-only	Display only undefined symbols

Table 26. nios-elf-nm Options

-l --line-numbers	For each symbol, use debugging information to find a filename and line number
-V --version	Display nm's version number and exit
--help	Display a summary of nm's options and exit

Example

```
nios-elf-nm hello_world.out > hello_world.nm
```

creates **hello_world.nm**, which includes a list of all symbols in the program.

```
hello_world.out:
000406b0 t CWPOverflowTrapHandler
000405fc t CWPUnderflowTrapHandler
000402d6 T PrivatePrintf
00040244 T RAMLimit
00040ae8 A __bss_start
000408ca T __divsi3
000408fc T __modsi3
00040796 T __mulhi3
00040796 T __mulsi3
00000001 a __nios32__
.
.
.
```



For details on GNU **nm**, refer to the on-line documentation by choosing **Programs > Cygwin > Cygwin Documentation** (Windows Start Menu). In the help window that appears, click **Using binutils**, then **nm**.

nios-elf-objcopy

nios-elf-objcopy converts executable binary files (**.out**) to S-records, which are suitable for ROM images and for download images to embedded systems.

If you use **nios-build** to generate executable code, **nios-elf-objcopy** is invoked automatically.

Usage

```
nios-elf-objcopy <switches> in-file [out-file]
```

Options

Table 27. nios-elf-objcopy Options

Option	Description
-I <bfdname>	Assume input file is in format <bfdname>
-O <bfdname>	Create an output file in format <bfdname>
-F <bfdname>	Set both input and output format to <bfdname>
--debugging	Convert debugging information, if possible
-p	Copy modified/access timestamps to output
-j <name>	Only copy section <name> into output
-R <name>	Remove section <name> from output
-S	Remove all symbol and relocation information
-g	Remove all debugging symbols
--strip-unneeded	Remove all symbols not needed by relocations
-N<name>>	Do not copy symbol<name>
-K <name>	Only copy symbol <name>
-L <name>	Force symbol <name> to be marked as a local
-W <name>	Force symbol <name> to be marked as a weak
--weaken	Force all global symbols to be marked as weak
-x	Remove all non-global symbols
-X s	Remove compiler-generated symbols
-i <number>	Only copy one out of every <number> bytes
-b <num>	Select byte <num> in every interleaved block
--gap-fill <val>	Fill gaps between sections with <val>
--pad-to <addr>	Pad the last section up to address <addr>
--set-start <addr>	Set the start address to <addr>
--change-start <incr>	Add <incr> to start address
--change-addresses <incr>	Add <incr> to LMA, VMA, and start addresses
--change-section-address <name>[= + -]<val>	Change LMA and VMA of section <name> by <val>

Table 27. nios-elf-objcopy Options

Option	Description
--change-section-lma <name>{= + -}<val>	Change LMA of section <name> by <val>
--change-section-vma <name>{= + -}<val>	Change VMA of section <name> by <val>
--[no-]change-warnings	Warn if a named section does not exist
--set-section-flags <name>=<flags>	Set section <name>'s properties to <flags>
--add-section <name>=<file>	Add section <name> found in <file> to output
--change-leading-char	Force output format's leading character style
--remove-leading-char	Remove leading character from global symbols
--redefine-sym <old>=<new>	Redefine symbol name <old> to <new>
-v	List all object files modified
--verbose	
-V	Display this program's version number
--version	
-h	Display help for this utility
--help	



For details on GNU **objcopy**, refer to the on-line documentation by choosing **Programs > Cygwin > Cygwin Documentation** (Windows Start Menu). In the help window that appears, click **Using binutils**, then **objcopy**.

nios-elf-objdump

This utility displays information about one or more object files. The options control which information to display, thus allowing users to see routine locations or code types produced by the compiler.

Usage

```
nios-elf-objdump <switches> file(s)
```

Options

Use at least one switch listed in [Table 28](#). [Table 29](#) lists optional switches.

Table 28. nios-elf-objdump Switches

Switch	Description
-a --archive-headers	Display archive header information
-f --file-headers	Display contents of the overall file header
-p --private-headers	Display object format specific file header contents
-h --[section-]headers	Display contents of the section headers
-x --all-headers	Display contents of all headers
-d --disassemble	Display assembler contents of executable sections
-D --disassemble-all	Display assembler contents of all sections
-S --source	Intermix source code with disassembly
-s --full-contents	Display full contents of all sections requested
-g --debugging	Display debug information in object file
-G --stabs	Display STABS contents of an ELF format file
-t --syms	Display contents of the symbol table(s)
-T --dynamic-syms	Display contents of the dynamic symbol table
-r --reloc	Display relocation entries in the file

Table 28. nios-elf-objdump Switches

Switch	Description
-R --dynamic-reloc	Display dynamic relocation entries in the file
-V --version	Display this program's version number
-i --info	List object formats and architectures supported
-H --help	Display this information

Table 29. nios-elf-objdump Optional Switches

Switch	Description
-b <bfdname> --target=<bfdname>	Specify target object format as <bfdname>
-m <machine> --architecture <machine>	Specify target architecture as <machine>
-j <name> --section=<name>	Only display information for section <name>
-M --disassembler-options <O>	Pass text <O> onto disassembler section
-EB --endian=big	Assume big endian format when disassembling
-EL --endian=little	Assume little endian format when disassembling
--file-start-context	Include context from start of file (with -S)
-l --line-numbers	Include line numbers and filenames in output
-C --demangle	Decode mangled/processed symbol names
-w --wide	Format output for more than 80 columns
-z --disassemble-zeroes	Do not skip blocks of zeroes when disassembling
--start-address <address>	Start displaying data at <address>
--stop-address <address>	Stop displaying data at <address>
--prefix-addresses	Print complete address alongside disassembly
--[no-]show-raw-insn	Display hex alongside symbolic disassembly
--adjust-vma <offset>	Add <offset> to all displayed section addresses

The **nios-elf-objdump** supported targets are:

- elf32-nios
- elf32-little
- elf32-big
- srec
- symbolsrec
- tekhex
- binary
- ihex

Example

```
nios-elf-objdump -D hello_world.out > hello_world.objdump
```

disassembles the object file **hello_world.out** and creates a disassembly output file **hello_world.objdump**:

```
hello_world.out:      file format elf32-nios
```

```
Disassembly of section .text:
```

```
00040100 <nr_jumptostart>:
 40100: 06 98      pfx %hi(0xc0)
 40102: 40 35      movi %g0,0xa
 40104: 00 98      pfx %hi(0x0)
 40106: 40 6c      movhi %g0,0x2
 40108: c0 7f      jmp %g0
 4010a: 00 30      nop
 4010c: 4e 69      extl6d %sp,%o2
 4010e: 6f 73      *unknown*
00040110 <main>:
 40110: 17 78      save %sp,0x17
 40112: 4a 98      pfx %hi(0x940)
 40114: 88 35      movi %o0,0xc
 40116: 00 98      pfx %hi(0x0)
 40118: 88 6c      movhi %o0,0x4
 4011a: 04 98      pfx %hi(0x80)
 4011c: a1 36      movi %g1,0x15
 4011e: 00 98      pfx %hi(0x0)
 40120: 41 6c      movhi %g1,0x2
 40122: e1 7f      call %g1
 40124: 00 30      nop
 40126: df 7f      ret
 40128: a0 7d      restore
.
.
.
```



For details on GNU **objdump**, see the on-line documentation by choosing **Programs > Cygwin > Cygwin Documentation** (Windows Start Menu). In the help window that appears, click **Using binutils**, then **objdump**.

nios-elf-size

The **nios-elf-size** utility analyzes **.out**, **.o**, or **.a** files and produces a report of code (text), data (data), and uninitialized storage (bss) sizes.

Usage

```
nios-elf-size [options] [file...]
```

Options

Table 30. nios-elf-size Options

Option	Description
-A	Output from GNU size resembles output from System V size
-B	Output from GNU size resembles output from Berkeley size
--format <compatibility>	Output from GNU size resembles output from <compatibility> size ("sysv" or "berkeley")
-d	Display section size in decimal
-o	Display section size in octal
-x	Display section size in hexadecimal
--radix <number>	Display section size in <number> ("10" for decimal, "8" for octal, "16" for hexadecimal)
--target <bfdname>	Specify an object code format for objfile as <bfdname>
-V	Display version number information on size itself
--version	
--help	Display a summary of arguments and options



For details on GNU **size**, refer to the on-line documentation by choosing **Programs > Cygwin > Cygwin Documentation** (Windows Start Menu). In the help window that appears, click **Using binutils**, then **size**.

nios-run

The **nios-run** utility downloads code and/or data to the Nios development board with the GERMS monitor running. **nios-run** is also used as a terminal I/O program to interact with the GERMS monitor or any other software running on the Nios development board.

When given a filename as a parameter, **nios-run** sends characters from the file to the host communication UART on the Nios development board. Usually these files are of type **.srec** with data to write into SRAM, or **.flash** with data to burn into flash.

Usage

```
nios-run [option(s)] [filename]
```

Options

Table 31. nios-run Options

Option	Description
-b <baud-rate>	Set the serial port baud rate (default = 115200)
-d	Provide additional debugging information during download
-e "<command>"	Execute a monitor command before entering terminal mode (experimental)
-o <seconds>	Quit after <seconds> seconds in terminal mode
-p <port-name>	Specify serial port (default = COM1:)
-s <millisecs>	Specify a per-character delay (useful for reluctant flash)
-t	Enter terminal mode without downloading code
-x	Exit immediately after downloading code
-z	Display timestamp for each line (useful for benchmarking)

Example

```
nios-run -p com2 hello_world.srec
```

downloads the executable file **hello_world.srec** to the development board via COM2.

srec2flash

srec2flash converts executable code in a **.srec** file to a **.flash** file, which can then be downloaded and burned into flash on the Nios development board.

At system start-up, the GERMS monitor looks for code in flash memory at location 0x140000. If user code is detected in flash, GERMS executes the code. **srec2flash** takes code in a **.srec** file targeted for location 0x40100 (SRAM on the development board) and creates a **.flash** file. The **.flash** file is a sequence of GERMS commands that prepare flash to be written, then burn the contents of the **.srec** file into flash at location 0x140000.

However, because the executable code is assumed to be executed from 0x40100 in SRAM, some additional pre-processing is required. **srec2flash** adds a small routine at the head of the user software. When executed, this routine copies the user software (and itself) stored in flash at 0x140000 into SRAM at 0x40100. After code is copied from flash to SRAM, program execution begins from SRAM.

Usage

```
srec2flash <srec file> [filename]
```

Example

```
srec2flash hello_world.srec
```

Generates the file **hello_world.flash** (partial listing follows):

```

# This file generated by srec2flash, part of
# the Nios SDK. This file contains a short
# program to run out of flash memory which
# copies the main program down to RAM, and
# executes it there.
#
# Original file: hello_world.srec
#
# Loader program
r0
#
# Erase flash sector 140000
#
# This address is checked by germsMon at startup
#
e140000
#
S219140000009800350098406DC07F00304E696F73089810349044
S2191400156E1134116F08981234926C005A50048074015A500455
S21914002A8174011E0140415E92043012E27EF387003021981009
S21914003F340098106CB2993135115E08981234926C3224D27FA0
S206140054003061
#
# Main program
#
r40100-140100
S013000068656C6C6F5F776F726C642E7372656376
S219040100069840350098406CC07F00304E696F7317784A988889
S219040115350098886C0498A1360098416CE17F0030DF7FA07D48
S21904012A17781298D95F1398DA5F1498DB5F1598DC5F1698DD09
S21904013F5F0833169849370098496CCB3302980B050B986135AC
.
.
.

```

To burn flash on the development board, use the **nios-run** utility:

```
nios-run -x hello_world.flash
```



For more details on this process, see the Altera white paper *Converting .srec Files to .flash Files for Nios Embedded Processor Applications* at http://www.altera.com/literature/wp/wp_srec_to_flash.pdf.

tracelink

tracelink associates a Nios object file (generated with **nios-elf-ld**) and a trace dump file (generated with the debug core routines described in “[Debug Core](#)” on page 29) to generate an assembly listing of all instructions traced, including all data accesses, skipped instructions, and interrupts.

Usage

```
tracelink objectfile tracedump
```

Example

```
nios-run hello_debug.srec > hello_debug.dump
```

hello_debug sends its trace output to the serial port, which is directed to the file **hello_debug.dump**.

```
tracelink hello_debug.out hello_debug.dump > hello_debug.trace
```

generates the output:

```
Reading in object and trace information...100%
Organizing trace stream.....100%
Generating instruction sequence.....100%
Aligning skip information.....100%
Aligning data information.....100%
Generating report.....100%
```

`hello_debug.trace` contains a full trace listing:

```

.
.
.
0x4009a  br 40090 <main+0x66>
0x4009c  nop
0x40090  bsr 40010 <do_write>
0x40092  mov %o0,%i0
0x40010  save %sp,0x17
0x40012  mov %l0,%i0
0x40014  lsli %l0,0x2
0x40016  pfx %hi(0x10c0)
0x40018  movi %g1,0x10
0x4001a  pfx %hi(0x0)
0x4001c  movhi %g1,0x4
0x4001e  add %l0,%g1
0x40020  movi %g1,0x1
0x40022  lsl %g1,%i0
0x40024  stp [%l0,0x0],%g1  WRITE  0x4114c -> 0x80000000
0x40026  ret
0x40028  restore
0x40094  inc %i0
0x40096  cmpi %i0,0x1f
0x40098  skps cc_gt
SKIPPED 0x4009a  br 40090 <main+0x66>
0x4009c  nop
0x4009e  movi %i0,0x0
0x400a0  bsr 40010 <do_write>
0x400a2  mov %o0,%i0
0x40010  save %sp,0x17
0x40012  mov %l0,%i0
.
.
.

```


Appendix A: GERMS Monitor Usage

The GERMS prompt is a plus sign, “+”. Unknown commands are answered with a question mark, “?”, followed by a new prompt “+”. The examples in this section assume the Nios development board is configured with the default Factory hardware image, and the GERMS monitor is running. The vector table is located at 0x40000 and the program code starts at 0x40100. The program used is **hello_world.c**. The vector table can be examined as follows:

```
+m40000
#00040000: 0A52 0002 0FBA 0002 1014 0002 0A52 0002
#00040010: 0A52 0002 0A52 0002 0A52 0002 0A52 0002
#00040020: 0A52 0002 0A52 0002 0A52 0002 0A52 0002
#00040030: 0A52 0002 0A52 0002 0A52 0002 0A52 0002
+
```

Each time ↵ is pressed, the next memory segment is displayed. For example, press ↵ five times to display:

```
#00040040: 0A52 0002 0A52 0002 0A52 0002 0A52 0002
#00040050: 0A52 0002 0A52 0002 0A52 0002 0A52 0002
#00040060: 0A52 0002 08E9 0002 0A52 0002 0A52 0002
#00040070: 0A52 0002 0A52 0002 0A52 0002 0A52 0002
+
#00040080: 0A52 0002 0A52 0002 0A52 0002 0A52 0002
#00040090: 0A52 0002 0A52 0002 0A52 0002 0A52 0002
#000400A0: 0A52 0002 0A52 0002 0A52 0002 0A52 0002
#000400B0: 0A52 0002 0A52 0002 0A52 0002 0A52 0002
+
#000400C0: 0A52 0002 0A52 0002 0A52 0002 0A52 0002
#000400D0: 0A52 0002 0A52 0002 0A52 0002 0A52 0002
#000400E0: 0A52 0002 0A52 0002 0A52 0002 0A52 0002
#000400F0: 0A52 0002 0A52 0002 0A52 0002 0A52 0002
+
#00040100: 984E 3780 9800 6C40 7FC0 3000 694E 736F
#00040110: 7817 9970 3408 9800 6C88 984C 35E1 9800
#00040120: 6C41 7FE1 3000 981A 3708 9800 6C48 7FE8
#00040130: 3000 7FDF 7DA0 0000 7817 9811 5FD8 9811
+
#00040140: 5BC1 7EC1 8004 3000 3438 8011 3000 9811
#00040150: 5BC1 9FFF 37E2 9FFF 6FE2 1041 7EE2 8004
#00040160: 3000 3438 8004 3000 3418 8001 3000 7FDF
#00040170: 7DA0 781C 9811 5FD8 9812 5FD9 9811 5BC1
+
```

The vector table starts at 0x40000 and ends at 0x400FF. Each four bytes represent an address for each of the 64 possible interrupts. In the case of `hello_world`, no interrupts are set, so the above display is the address of the routine `r_spurious_irq_handler` divided by 2. That is, `r_spurious_irq_handler` is at 0x414a4. Divided by 2 it is at 0x020a52, seen as 0x0A52 (lower half-word) and 0x0002 (higher half-word).

The program starts at 0x40100. Following is an excerpt from the start of **hello_world.objdump**:

```
hello_world.out:      file format elf32-nios

Disassembly of section .text:

00040100 <nr_jumptostart>:
  40100:      4e 98          pfx %hi(0x9c0)
  40102:      80 37          movi %g0,0x1c
  40104:      00 98          pfx %hi(0x0)
  40106:      40 6c          movhi %g0,0x2
  40108:      c0 7f          jmp %g0
  4010a:      00 30          nop
  4010c:      4e 69          extl6d %sp,%o2
  4010e:      6f 73          usr0 %o7,%i3

00040110 <main>:

#include "nios.h"

int main(void)
{
  40110:      17 78          save %sp,0x17
  //
  // This will not work without a UART!
  //

  NIOS_GDB_SETUP

  printf ("\n\nHello from Nios.\n\n");
  40112:      70 99          pfx %hi(0x2e00)
  40114:      08 34          movi %o0,0x0
```

Use the G command to start execution at a certain address. For example, after **hello_world.srec** is run, it exits to the GERMS monitor. To run `hello_world` again without exiting the monitor:

```

ref_32_2.1
+
g40100

Hello from Nios.

*
#415B1234
ref_32_2.1
+

```

To write to a particular peripheral, only the address is required. For example, by default, the Nios development board's seven-segment LED is at address 0x420. Write 0x3649 to display two vertical and three horizontal lines:

```

+
m420:3649
+

```

Hardware Considerations (32-Bit Nios CPU only)

When using the M command to write to memory, the monitor can use either an ST8 (8-bit store), ST16 (16-bit store), or ST (32-bit store) instruction. This is useful to know, as ST8 uses one byte enable, ST16 uses two byte enables, and ST uses four byte enables to write. The following examples demonstrate writing with each method.

To write a single byte:

```

+
m50000:30
+
m50000
#00050000: 0030 0000 0000 0000 0000 0000 0000 0000
#00050010: 0000 0000 0000 0000 0000 0000 0000 0000
#00050020: 0000 0000 0000 0000 0000 0000 0000 0000
#00050030: 0000 0000 0000 0000 0000 0000 0000 0000

```

To write a single half-word:

```

+
m50000:1234
+
m50000
#00050000: 1234 0000 0000 0000 0000 0000 0000 0000
#00050010: 0000 0000 0000 0000 0000 0000 0000 0000
#00050020: 0000 0000 0000 0000 0000 0000 0000 0000
#00050030: 0000 0000 0000 0000 0000 0000 0000 0000

```

To write a full word:

```

+
m50000:ABCD9876
+
m50000
#00050000: 9876 ABCD 0000 0000 0000 0000 0000 0000
#00050010: 0000 0000 0000 0000 0000 0000 0000 0000
#00050020: 0000 0000 0000 0000 0000 0000 0000 0000
#00050030: 0000 0000 0000 0000 0000 0000 0000 0000
+

```

In the next example, a new design is manually loaded in the default flash area, such as **hexout2flash** does. Assuming the design **my_user_design.hexout** is created, start the monitor from the bash shell:

```

[bash] ../mydir: nr -t
nios-run: Terminal mode (Control-C exits)
-----
+
e180000
+
e190000
+
e1A0000
+
e1B0000
+
r180000
+
<CTRL-C>

```

The “e” commands erase the different blocks needed from the flash and the “r” command relocates whatever is downloaded next, starting at that address. The final step is to send the design:

```
[bash] ../mydir: nr my_user_design.hexout
```

Appendix B: Assembly Language Macros

The file **nios_macros.s** located in the **.../inc/** directory provides several assembly language macros useful for low-level programming and debugging.

See the *Nios 16-Bit Programmer's Reference Manual* or *Nios 32-Bit Programmer's Reference Manual* for details on assembly language programming.

Table 32. Assembly Language Macros

Macro	Description
MOVIP %reg,value	Acts similarly to the Nios instruction MOVI, but allows any size constant. It automatically uses a combination of BGEN, MOVI, MOVHI, and PFX to load the value into the register. MOVIP uses as few of these instructions as possible. MOVIP can only be used with defined constants; it generates an error if the constant is not defined at assembly time.
MOVIA %reg,value	Load a native-sized value into the register. The native word size is 16 or 32 bits; 16-bit or 32-bit Nios CPU, respectively. The value need not be defined at assembly time; the linker fills in the value later.
ADDIP %reg,value	Acts similarly to ADDI, but works for any 16-bit constant. It does not work for constants greater than 16 bits.
SUBIP %reg,value	Acts similarly to SUBI, but works for any 16-bit constant. It does not work for constants greater than 16 bits.
CMPIP %reg,value	Acts similarly to CMPI, but works for any 16-bit constant. It does not work for constants greater than 16 bits.
ANDIP %reg,value	Acts similarly to ANDI, but works for any 16-bit constant. It does not work for constants greater than 16 bits.
ANDNIP %reg,value	Acts similarly to ANDNI, but works for any 16-bit constant. It does not work for constants greater than 16 bits.
ORIP %reg,value	Acts similarly to ORI, but works for any 16-bit constant. It does not work for constants greater than 16 bits.
_BR address	Acts similarly to BR, but uses %g7 to load the target address. The target address is therefore not limited to the short branch range.
_BSR address	Acts similarly to BSR, but uses %g7 to load the target address. The target address is therefore not limited to the short branch range.
nm_print string	Prints the quoted string to the default UART. It uses %o0 and %g registers.
nm_println string	Like nm_print, but prints the string followed by a carriage return and line feed.

Table 32. Assembly Language Macros

Macro	Description
<code>nm_d_txchar char</code>	<p>Expands out to a large block of code that transmits a character to the default UART without altering any registers or requiring the CWP to move. It does use stack space.</p> <p>Because this macro does not affect any registers or the CWP, it is useful for debugging interrupt handlers and low-level services, such as task switchers.</p>
<code>nm_d_txreg char1,char2,%reg</code>	<p>Expands out to a large block of code that transmits the two characters followed by the register's hexadecimal value. It prints erroneous values for the stack pointer register.</p> <p>Because this macro does not affect any registers or the CWP, it can be useful for debugging interrupt handlers and low-level services, such as task switchers.</p>

Symbols

__mulhi3 routine 20
 __mulsi3 routine 20
 __nios_use_constructors__ setting 14
 __nios_use_cwpmgr__ setting 14
 __nios_use_fast_mul__ setting 17
 __nios_use_small_printf__ setting 15
 _BR macro 93
 _BSR macro 93
 _close routine 20
 _exit routine 20
 _fstat routines 20
 _getpid routine 20
 _kill routine 20
 _read routine 20
 _sbrk routine 20
 _start routine 20, 21
 _write routine 20

A

ADDIP macro 93
 ANDIP macro 93
 ANDNIP macro 93
 Application software, creating and compiling 6
 Assembly language macros 93

B

bash 51
 Boot process, GERMS 10

C

C runtime support 20
 CMPIP macro 93
 Code
 auto-booting transition 7

debugging 7
 executable, downloading 7

CPU

 core size 2
 data path 3
 Current window pointer. See CWP Manager.
 CWP Manager 23

D

Data path, CPU 3
 Data structures
 DMA 34
 PIO 39
 SPI 41
 Timer 43
 UART 45
 Debug core peripheral
 interrupt 30
 register access 30
 registers 29
 routines 31
 trace data 30
 Debugging code 7
 Development flow 2
 DMA peripheral
 data structure 34
 registers 34
 routines 35

E

Execution speed, software 2

F

Flash memory
 booting from 11
 saving to 6

G

General-purpose system routines 24
GERMS monitor 8–11
 boot process 10
 building processor 6
 commands 9
 usage examples 89
GNUPro utilities 51

H

Hardware acceleration 2
hexout2flash utility 51, 52

I

inc directory 12
Include directory 12
isatty routine 20

L

lib directory 16
libnios16.a library 19
libnios32.a library 19
Library
 directory 16
 routines 19

M

M setting 18
Macros
 _BR 93
 _BSR 93
 ADDIP 93
 ANDIP 93
 ANDNIP 93
 CMPIP 93
 MOVIA 93
 MOVIP 93
 nm_d_txchar 94
 nm_d_txreg 94
 nm_print 93
 nm_println 93
 ORIP 93

SUBIP 93

Makefile settings 17
 __nios_use_constructors__ 14
 __nios_use_cwpmgr__ 14
 __nios_use_fast_mul__ 17
 __nios_use_small_printf__ 15
M 18
NIOS_SYSTEM_NAME 18
NIOS_USE_MSTEP 17
NIOS_USE_MULTIPLY 18

Memory
 model 1
 off-chip 4
 on-chip 4
MOVIA macro 93
MOVIP macro 93
MSTEP multiplier 4
MUL multiplier 4
Multipliers 4

N

Nios
 library routines 19
 program structure 19
 utilities 51–87
nios_bash utility 51, 53
nios_csh utility 51, 57
NIOS_SYSTEM_NAME setting 18
NIOS_USE_MSTEP setting 17
NIOS_USE_MULTIPLY setting 18
nios-build utility 51, 54
nios-convert utility 51, 56
nios-elf-as utility 51, 58
nios-elf-gcc utility 51, 60
nios-elf-gdb utility 51, 64
nios-elf-gprof utility 51, 66
nios-elf-ld utility 51, 70
nios-elf-nm utility 51, 75
nios-elf-objcopy utility 51, 77
nios-elf-objdump utility 51, 79
nios-elf-size utility 51, 82
nios-run utility 51, 83
nm_d_txchar macro 94
nm_d_txreg macro 94
nm_debug_get_reg routine 27, 32

nm_debug_set_bp0 routine 27, 33
 nm_debug_set_bp1 routine 27, 33
 nm_debug_set_reg routine 27, 33
 nm_print macro 93
 nm_println macro 93
 nr_debug_dump_trace routine 27, 31
 nr_debug_isr_continue routine 27, 32
 nr_debug_isr_halt routine 27, 32
 nr_debug_start routine 27, 31
 nr_debug_stop routine 27, 31
 nr_delay routine 24
 nr_dma_copy_1_to_1 routine 27, 35
 nr_dma_copy_1_to_range routine 27, 36
 nr_dma_copy_range_to_1 routine 27, 38
 nr_dma_copy_range_to_range routine 27, 37
 nr_installcwpmanager routine 24
 nr_installuserisr routine 22
 nr_installuserisr2 routine 23
 nr_pio_showhex routine 27, 40
 nr_spi_rxchar routine 27, 42
 nr_spi_txchar routine 27, 42
 nr_timer_milliseconds routine 28, 44
 nr_uart_rxchar routine 28, 46
 nr_uart_txchar routine 28, 46
 nr_uart_txc routine 28, 48
 nr_uart_txhex routine 28, 48
 nr_uart_txhex16 routine 28, 48
 nr_uart_txhex32 routine 49
 nr_uart_txstring routine 28, 49
 nr_zerorange routine 25

O

Off-chip
 memory 4
 shared bus 5
 On-chip memory 4
 ORIP macro 93

P

Peripherals 5
 Debug core 29
 DMA 34
 Off-chip shared bus 5
 PIO 39

 register maps 12
 routines 27–49
 SPI 41
 Timer 43
 UART 45
 user defined interface 5
 PIO peripheral
 data structure 39
 registers 39
 routines 40
 printf routine 25
 Processor
 building 5
 saving configuration 6
 Program structure, Nios 19

R

Register file, size considerations 3
 Registers

 Debug core 29
 DMA 34
 PIO 39
 SPI 41
 Timer 43
 UART 45

Routines

 __mulhi3 20
 __mulsi3 20
 _close 20
 _exit 20
 _fstat 20
 _getpid 20
 _kill 20
 _read 20
 _sbrk 20
 _start 20, 21
 _write 20
 Debug core 31
 DMA 35
 general purpose 24
 isatty 20
 nm_debug_get_reg 27, 32
 nm_debug_set_bp0 27, 33
 nm_debug_set_bp1 27, 33
 nm_debug_set_reg 27, 33

- nr_debug_dump_trace 31
- nr_debug_isr_continue 27, 32
- nr_debug_isr_halt 27, 32
- nr_debug_start 27
- nr_debug_stop 27, 31
- nr_debug_trace 27
- nr_delay 24
- nr_dma_copy_1_to_1 27, 35
- nr_dma_copy_1_to_range 27, 36
- nr_dma_copy_range_to_1 27, 38
- nr_dma_copy_range_to_range 27, 37
- nr_installcwpmanager 24
- nr_installuserisr 22
- nr_installuserisr2 23
- nr_pio_showhex 27, 40
- nr_spi_rxchar 27, 42
- nr_spi_txchar 27, 42
- nr_timer_milliseconds 28, 44
- nr_uart_rxchar 28, 46
- nr_uart_txchar 28, 46
- nr_uart_txcr 28, 48
- nr_uart_txhex 28, 48
- nr_uart_txhex16 28, 48
- nr_uart_txhex32 49
- nr_uart_txstring 28, 49
- nr_zerorange 25
- peripherals 27–49
- PIO 40
- printf 25
- rn_debug_start 31
- service 21
- SPI 42
- sprintf 25
- Timer 44
- UART 45
- uart_txchar32 28
- Runtime support, C 20

S

- SDK 1, 12–18
- Software
 - data structures. See Data structures.
 - execution speed 2
 - routines. See Routines.
- SOPC design considerations 1

- SPI peripheral
 - data structure 41
 - registers 41
 - routines 42
- sprintf routine 25
- srec2flash utility 11, 51, 84
- SUBIP macro 93
- System-level services 21

T

- Timer peripheral
 - data structure 43
 - registers 43
 - routines 44
- tracelink utility 51, 86

U

- UART peripheral
 - data structure 45
 - registers 45
 - routines 45
- uart_txchar32 routine 28
- User defined interface 5
- Utilities 51–87
 - hexout2flash 51, 52
 - nios_bash 51, 53
 - nios_csh 51, 57
 - nios_elf-as 51, 58
 - nios-build 51, 54
 - nios-convert 51, 56
 - nios-elf-gcc 51, 60
 - nios-elf-gdb 51, 64
 - nios-elf-gprof 51, 66
 - nios-elf-ld 51, 70
 - nios-elf-nm 51, 75
 - nios-elf-objcopy 51, 77
 - nios-elf-objdump 51, 79
 - nios-elf-size 51, 82
 - nios-run 51, 83
 - srec2flash 51, 84
 - tracelink 51, 86