



Nios Software Development Tutorial



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

Document Version: 1.3
Document Date: July 2003

Copyright © 2003 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.



I.S. EN ISO 9001

This document provides a software tutorial for the Altera® Nios® embedded processor.

Table 1 shows the tutorial revision history.

Table 1. Tutorial Revision History	
Date	Description
July 2003	Reflects new directory structure for SOPC Builder 3.0 and Nios Development Kit version 3.1.
May 2003	Minor edits and additions.
March 2003	Updated the document so that it applies to all Nios development kits.
January 2003	First publication.

How to Find Information

- The Adobe Acrobat Find feature allows 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
Technical support	http://www.altera.com/mysupport/	http://www.altera.com/mysupport/
	(800) 800-EPLD (3753) (7:00 a.m. to 5:00 p.m. Pacific Time)	(408) 544-7000 (1) (7:00 a.m. to 5:00 p.m. Pacific Time)
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)
FTP site	ftp.altera.com	ftp.altera.com






Note:

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

Typographic Conventions

This document uses the typographic conventions shown in [Table 3](#).

Table 3. Conventions

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} , lqdesigns directory, d: drive, chiptrip.gdf file.
<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: <file name>, <project name>. .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 on-line help topics are shown in quotation marks. Example: "Typographic Conventions."
Courier type	Signal and port names are shown in lowercase Courier type. Examples: data1, tdi, input. Active-low signals are denoted by suffix n, e.g., resetn. Anything that must be typed exactly as it appears is shown in Courier type. For example: c:\qdesigns\tutorial\chiptrip.gdf. Also, sections of an actual file, such as a Report File, references to parts of files (e.g., the AHDL keyword SUBDESIGN), as well as logic function names (e.g., 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 Tutorial	iii
How to Find Information	iii
How to Contact Altera	iv
Typographic Conventions	v
Tutorial Overview	9
Hardware/ Software Development Flow	9
Hardware & Software Requirements	11
Tutorial Files	12
Software Tools Used	12
GNUPro Tools	12
Nios On-Chip Instrumentation (OCI) Debug Module	12
Nios OCI Debug Console	13
Nios SDK Shell	13
About Configurable Processor Hardware Features	13
Multipliers	13
Data & Instruction Caches	14
Custom Instructions	14
Simultaneous Multi-Master Bus Architecture & DMA	14
Peripherals & Memory Interfaces	14
Nios SDK	14
inc Directory	15
lib Directory	17
src Directory	17
Tutorial	19
Before You Begin	19
Open the Nios SDK Shell	20
Compile a Program	21
Download, Run & Debug the Program Using Insight	22
Download, Run & Debug the Program Using the Nios OCI Debug Console	29
Rebuild the Software	33
Download the Software to Flash	34
Third-Party Development & Debug Tools	35
Appendix—Using a .hexout	37

GERMS Monitor	39
Insight: GNU Debugger	39
Appendix—Other Board Communication & Debug	39
Gprof: GNU Profiler	41
Requirements	42
Debugging Code with Gprof	42
Changing Gprof Settings	48
References	48
Appendix—Nios SDK Shell Tips	49
Changing to the SDK Directory	49
Keystroke Shortcuts	49
Utility Usage Shortcuts	49
Index	51

Hardware/ Software Development Flow

This tutorial introduces software development for the Nios processor and walks you through the basic software development flow for compiling, debugging, and prototyping embedded software. Although this tutorial uses the Nios development kit as a demonstration platform, the general concepts introduced in this tutorial are valid for any Nios processor system. The instructions in this tutorial are designed to work with the following Nios development kits:

- Nios Development Kit, Stratix Edition
- Nios Development Kit, Cyclone Edition
- Nios Development Kit, Stratix Professional Edition

Figure 1 shows a complete design flow for creating a Nios system and prototyping it on the Nios development board. The diagram includes both the hardware and software design tasks required to create a working system. The right side shows the software development flow while the left side shows the hardware design flow. This tutorial walks you through the steps “Software Development” and “Software Prototype on the Development Board” shown in Figure 1.



Refer to the *Nios Hardware Development Tutorial* for a complete explanation of the hardware flow.

Figure 1. Hardware/Software Development Flow for a Nios Processor System

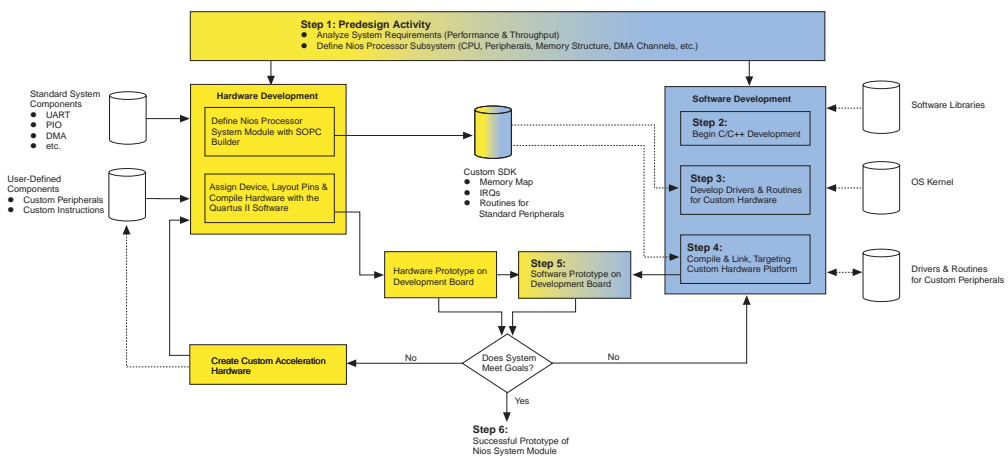


Figure 1 on page 9 shows where the hardware and software flows intersect. To obtain a complete, working system, it is important to know what each side must provide for the other. Even if your development involves separate teams for hardware and software design, it is helpful to understand the design flow on both sides of the hardware-software divide.

The development flow begins with predesign activity (step 1 in Figure 1), which includes an analysis of the system requirements:

- What computational performance does the design require?
- How much bandwidth or throughput must the system handle?

Based on the answers to these questions, you can determine the concrete system requirements:

- Will the CPU need a hardware-accelerated multiplier?
- Which peripherals, and how many of each, does the design require?
- Could DMA channels be used to free up CPU cycles spent copying data?

These decisions involve both the hardware and software teams.

Based on the system requirements, the hardware design begins with the SOPC Builder system integration software. At this point, you can begin writing device-independent C/C++ software, such as arithmetic algorithms or control programs (step 2 in Figure 1). You can use existing software libraries and/or an OS kernel to speed the development process.

After the hardware designer defines the custom Nios processor hardware system using SOPC Builder, SOPC Builder generates a custom software development kit (SDK) that forms the foundation for the software development flow. With the SDK (step 3 in Figure 1), you can begin writing software that interacts at a low level with hardware components. The SDK defines the software view of the custom hardware, including the memory map and the data structures for accessing hardware components in the system. The SDK provides software routines for accessing standard peripherals such as UARTs, PIOs, and DMA controllers. With very little development effort, you can use the SDK to communicate successfully with fundamental system components.

The GNUPro Toolkit is used to compile and link software with the SDK generated routines, header files, and other software libraries (step 4 in [Figure 1](#)). After the hardware designers prototype the basic Nios processor hardware working on the development board, the software team can download the software to the development board and prototype the software executing in hardware (step 5 in [Figure 1](#)). The Nios development kit provides several utilities for compiling software projects and downloading programs to the development board using an Altera ByteBlasterMV or ByteBlaster II download cable or using a serial cable. The available GNUPro Toolkit and Nios development utilities provide rapid edit-compile-download-debug iterations while designing your system (step 6 in [Figure 1](#)).

The configurable Nios processor offers flexible options for performance enhancements if the system does not initially achieve its performance goals. Refer to [“About Configurable Processor Hardware Features” on page 13](#) for more information about these features.



If the system includes custom hardware components, then you must create low-level C or Assembly language routines to interface to the custom hardware. These routines can be simple, such as writing values to a PIO peripheral to communicate control data to the outside world; or more complex, such as using interrupt service routines to monitor and process stimulus from the outside world continuously. In many cases, the hardware engineer writes the most low-level software functions, and these functions become an integrated part of the SDK. This tutorial does not describe how to write software for custom peripherals.

Hardware & Software Requirements

This tutorial requires the following hardware and software:

- A PC with the Nios processor version 3.0 and related software development tools installed.
- Nios development board configured with the factory-programmed “safe” reference design. The board should be set up and connected as described in the getting started user guide for your kit.
- The ByteBlaster download cable driver, installed as described in the *Quartus II Installation & Licensing for PCs* manual.



Refer to the getting started user guide for your kit for details on the Nios development board and the Nios development tools.

Tutorial Files

The Nios embedded processor software installation creates the directories shown in [Table 4](#) in the `\altera\kits\nios` directory by default.

<i>Table 4. Directory Structure</i>	
Directory Name	Description
bin	Contains tools required for developing Nios hardware & software designs, including the GNU tool chain.
components	Contains all of the SOPC Builder peripheral components. Each peripheral has its own subdirectory with a class.ptf file that describes the component.
documents	Contains documentation for the Nios embedded processor, Nios development board, SOPC Builder, and GNUPro Toolkit.
examples	Contains subdirectories of Nios sample designs, including the standard_32 project on which the design for this tutorial is based.
tutorials	Contains subdirectories of files that you can use with various Nios tutorials. The directory for this tutorial is Nios_SW_Tutorial_<device family>_<device> .

Software Tools Used

The following sections describe the software tools that you will use in this tutorial.

GNUPro Tools

The Nios development kit includes the GNUPro Toolkit, which includes a compiler, debugger (command-line GDB and GUI-based Insight), and general-purpose software development utilities. For more information, refer to the Nios documentation in the `<SOPC Builder>/documents` directory and other Red Hat documents in the `<SOPC Builder>/documents/gnu_tools` directory.

Nios On-Chip Instrumentation (OCI) Debug Module

The Nios On-Chip Instrumentation (OCI) Debug Module is a debug core created by First Silicon Solutions. The Nios OCI Debug Module provides a JTAG interface to the Nios CPU, and implements an in-circuit emulator feature set including run control, memory and register visibility, and complex breakpoints.

Nios OCI Debug Console

The Nios OCI Debug Console is a command-line interface for communicating with the Nios OCI Debug Module. Commands are included for system configuration, emulation control, memory access (including an assembler and disassembler), register access, trace and trigger access, file downloading, and status indication. The commands are detailed in Appendix A of First Silicon Solutions' *Getting Started: System Analyzer for Nios Processor Core*. The Nios development kit includes a version of the debug console that works with the Altera ByteBlasterMV or ByteBlaster II cable.



Nios Development Kit, Stratix Professional Edition users can also set complex data breakpoint triggers like breaking execution when a specific data value is stored to a specific address.

Nios SDK Shell

You use this bash environment to develop Nios software and communicate with the Nios development board. The Nios SDK Shell provides a UNIX-like environment on a PC platform. The Nios development kit includes many Nios-specific utilities that you can run in the Nios SDK Shell to generate and debug software. You can also use the Nios SDK Shell to run test programs on the Nios development board.

Designing with a soft-core processor allows you to configure the processor hardware based on your system requirements. The following sections describe some of the available features you can use to increase the performance of your Nios system.

Multipliers

The 32-bit Nios CPU offers two hardware-accelerated multiply instructions, which achieve up to ten times the performance of a software-only implementation. 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). For more information, refer to the *Nios 3.0 CPU Data Sheet*.

About Configurable Processor Hardware Features

Data & Instruction Caches

You can configure the Nios processor to use on-chip memory as instruction cache (I-cache) and data cache (D-cache), which can improve overall system performance. The processor stores copies of off-chip memory in the on-chip I-cache and D-cache to decrease memory access time. If you are using slower off-chip memory and want better performance, add I-cache and/or D-cache.

Custom Instructions

You can add custom instructions to the Nios processor to improve performance dramatically. If you find that a particular operation prevents the system from meeting your performance goals, consider implementing that operation in hardware as a custom instruction.

Simultaneous Multi-Master Bus Architecture & DMA

The Nios processor uses a multi-master bus architecture, which allows multiple masters to perform bus transactions simultaneously. For example, the simultaneous multi-master architecture allows a DMA peripheral to transfer data between peripherals and/or memories while the Nios CPU executes instructions simultaneously, as long as the DMA and the Nios processor do not access the same peripheral. This feature increases system throughput without increasing the clock frequency. For more information, refer to *AN 184: Simultaneous Multi-Mastering with the Avalon Bus*.

Peripherals & Memory Interfaces

You can create a Nios processor hardware system with any combination of peripherals, memory interfaces, or CPUs. Often, it is more efficient to duplicate hardware resources than to make software share a single resource. The only limiting factor for the number of CPUs, peripherals, and memory interfaces in a system is the amount of logic inside the Altera FPGA. The SOPC Builder tool makes it easy to add intellectual property (IP) as well as user-defined logic to the peripheral library. Visit the Altera web site for a list of the SOPC Builder ready blocks.

Nios SDK

In addition to the GNUPro tools, the SDK is the foundation for developing Nios software. SOPC Builder generates a custom SDK for every Nios processor system generated. If you do not use SOPC Builder to generate hardware, then you must provide your own SDK.

The SDK is a collection of files, and the top-level directory is of the format `<CPU name>_sdk` (where `<CPU name>` is the CPU name given in SOPC Builder). If your system has more than one Nios CPU, you should receive an SDK for each CPU.

The three main directories in the Nios SDK are:

- inc
- lib
- src



For more information on the SDK, refer to the *Nios Software Development Reference Manual*.

inc Directory

The **inc** (include) directory has include files (**.h** and **.s**) that contain peripheral addresses, register structures, interrupt numbers, function prototypes for some useful routines, and Assembly language macros. The **excalibur.h** and **excalibur.s** files contain `#define` statements that describe the memory map and interrupt priorities. The directory also includes function prototypes for software routines and C structures for accessing peripheral registers. Because the Nios processor architecture and memory map are configurable, you should only reference the hardware using the symbols defined in files in the **inc** directory. [Figure 2](#) is an excerpt from an example **excalibur.h** file.

Figure 2. Example *excalibur.h*

```
.
.
.

// The Memory Map

#define na_boot_monitor_rom      ((void *)      0x00000000) //
altera_avalon_onchip_memory
#define na_boot_monitor_rom_base      0x00000000
#define na_boot_monitor_rom_end      ((void *)      0x00000400)
#define na_boot_monitor_rom_size      0x00000400
#define na_uart1                  ((np_uart *)      0x00000400) //
altera_avalon_uart
#define na_uart1_base                  0x00000400
#define na_uart1_irq                    26
#define na_timer1                  ((np_timer *)      0x00000440) //
altera_avalon_timer
#define na_timer1_base                  0x00000440
#define na_timer1_irq                    25
.
.
.
```

Continued on next page

```
// -----
// Timer Peripheral

// Timer Registers
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;

// Timer Register Bits
enum
{
    np_timerstatus_run_bit = 1, // timer is running
    np_timerstatus_to_bit = 0, // timer has 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 has 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 time out interrupt
};

// Timer Routines
int nr_timer_milliseconds(void); // Starts on first call, hogs timer1.
.
.
.
```

The prefixes `na_` and `np_` stand for Nios address and Nios peripheral, respectively. The base addresses of the peripherals are derived from the user-defined name assigned in SOPC Builder. You should refer to the base addresses as `na_<peripheral name>`. Similarly, refer to the names of the registers in the C structure as `np_<register name>`. Always use these names instead of absolute addresses so that you will not have to change the code if the memory map changes.

lib Directory

The **lib** (library) directory contains a makefile, and archive, source, and object files for libraries usable by your Nios system. The archive contains assembled (or compiled) versions of routines from each file, suitable for linking to your program.

src Directory

All source code should be placed in the **src** (source) directory when using the **nios-build** command. The SDK **src** directory includes software code examples for you to test the system. You can download these examples to the development board and run them without any modifications.



Notes:

Before You Begin

This section walks you through the process of compiling software for the Nios processor, downloading software to the Nios development board, and debugging software.

To begin developing software for the Nios processor, you must have the following items:

- SDK for your design. The SDK contains header files, library files, and example source code.



This tutorial uses the SDK generated by SOPC Builder for the factory reference design on the Nios development board. The files are located in the *<installation path>/tutorials/Nios_SW_Tutorial_<device family>_<device>/cpu_sdk* directory. These files are an example of what is required before starting software development.

- A development board with a serial cable and a ByteBlasterMV or ByteBlaster II cable attached. Set up the Nios development board and connect the serial, ByteBlasterMV, or ByteBlaster II cables as described in the getting started user guide for your kit. This tutorial assumes you have the factory image loaded on the Nios development board. The factory image is loaded if the Safe LED is illuminated.
- The hardware image for the development board. Usually the image is a Hexadecimal (Intel-Format) Output File (.hexout), an output of the Quartus II software, containing an Altera FPGA image that is ready to be loaded into flash memory on the board.
- The data sheet for each peripheral used in the design. The data sheet provides a description of the registers and software routines in the SDK, which you can use in your code. The peripheral data sheets are located in the *<installation path>/documents* directory.

Open the Nios SDK Shell

As described in “[Nios SDK Shell](#)” on page 13, the Nios SDK Shell is a UNIX-like command shell that allows you to build software, download software to the Nios development board, and run utilities and various test programs on the board. [Table 5](#) briefly describes some of the utilities that you can invoke in the Nios SDK Shell.

<i>Table 5. Nios SDK Shell Commands</i>	
Command	Description
nios-build	Compiles and links the source code (C and Assembly).
nios-run	Downloads software to a development board and runs it. It is also used as a terminal program to interact with the development board.
nios-console	Launches the Nios OCI Debug Console.
nios-debug	Launches the software debugger.
hexout2flash	Converts a hardware design .hexout file to a .flash file that can be downloaded to a flash device.
srec2flash	Converts a compiled software program S-Record file (.srec) into a flash image that will be executed automatically when the development board is reset.
nios-elf-size	Prints the size of the code, data, and uninitialized storage.
nios-elf-objdump	Creates a .objdump file with disassembly of the .out file.
nios-elf-gprof	Creates the execution profile of a C program.



For more detailed information about these and other software utilities, refer to the *Nios Software Development Reference Manual*.



For helpful hints on using the Nios SDK Shell, refer to “[Appendix—Nios SDK Shell Tips](#)” on page 49.

1. Choose **Programs > Altera > Nios <version> > Nios SDK Shell** (Windows Start menu). The **Nios SDK Shell** window opens and displays some text, including path information and messages about sample programs. By default, the shell opens in the **/Altera/kits/nios/examples** directory. See [Figure 3](#).

Figure 3. Nios SDK Shell

```

SOPC Builder 3.00

Welcome To Altera SOPC Builder
Version 3.00, Built Tue Jun 10 18:21:38 PDT 2003

Welcome To Nios Development Kit
Version 3.10, Built Tue Jul 29 02:09:35 PDT 2003

Example designs can be found in
/cygdrive/c/altera/kits/nios/examples

<You may add a startup script: c:/altera/kits/nios/user.bashrc>
/cygdrive/c/altera/kits/nios/examples
[SOPC Builder]$ _

```

Compile a Program

In this section you will compile a program using the Nios SDK Shell. To compile your program, perform the following steps.

1. Change to the software tutorial source directory, *<installation path>/tutorials/Nios_SW_Tutorial_<device family>_<device>/cpu_sdk/src*, by typing the following command at the Nios SDK Shell prompt:

```
cd ../tutorials/Nios_SW_Tutorial_<device family>_<device>
/cpu_sdk/src ↵
```

2. Compile **lcd_demo1.c** by typing the following command:

```
nios-build lcd_demo1.c ↵
```

The **nios-build** command compiles and links the source code in **lcd_demo1.c**, creates an executable S-record file (**.srec**), and creates the files shown in [Table 6](#). The **nios-build** command is an alternative to the makefile, which is fully supported in Nios development kit. For more information on makefiles, refer to the *Nios Embedded Processor Software Development Reference Manual*.



You can use the abbreviated command **nb** instead of **nios-build**.

Table 6. Files Created by nios-build

File	Description
lcd_demo1.c.o	Linked object file (.s.o for Assembly language files).
lcd_demo1.nm	Provides a list of all symbols in the program.
lcd_demo1.out	Binary output.
lcd_demo1.objdump	Interleaved source code and disassembly from .out.
lcd_demo1.srec	Binary output in S-record format.

3. You can determine the amount of memory the program consumes by typing the following command:

```
nios-elf-size lcd_demo1.out ↵
```

This command lists the size of the code (text), data (data), and uninitialized storage sizes (bss). The decimal and hexadecimal columns printed by the nios-elf-size command are the decimal and hexadecimal values for the text size.

Download, Run & Debug the Program Using Insight

In this section, you will download the **lcd_demo1.srec** to the development board, launch Insight (the front end to the GNUPro Debugger), and perform basic debugging.



Using the Nios OCI debug console, Nios Development Kit, Stratix Professional Edition users can also set complex data breakpoint triggers like breaking execution when a specific data value is stored to a specific address. See the First Silicon Solutions' *Getting Started: System Analyzer for Nios Processor Core* for details.

The design files for this tutorial use the serial port for basic host communication and the JTAG port for debugging communication and for downloading source code (.srec). You must use the JTAG port to communicate with the debug module. You will use the Nios SDK Shell you opened in [“Open the Nios SDK Shell” on page 20](#) to launch the debug console. You will open a second Nios SDK Shell for host communication.



If Insight does not launch properly, verify that the Safe LED is illuminated on the Nios development board, indicating that the factory reference design is loaded into the Altera FPGA.

Before starting this section of the tutorial, reset the system by pressing and holding SW0 on the Nios development board while pressing and releasing the CPU reset button. This action brings up the system with the GERMS monitor waiting for commands (instead of running the web server from the factory reference design).

Perform the following steps to download, run, and debug the program.

1. Choose **Programs > Altera > Nios <version> > Nios SDK Shell** (Windows Start menu) to open a second Nios SDK Shell.
2. Type the following command in the shell to enter terminal mode and connect to the serial port on the host PC:

```
nios-run -t -r ↵
```

The `-t` option changes to terminal mode and the `-r` option forces a connection to the serial port.



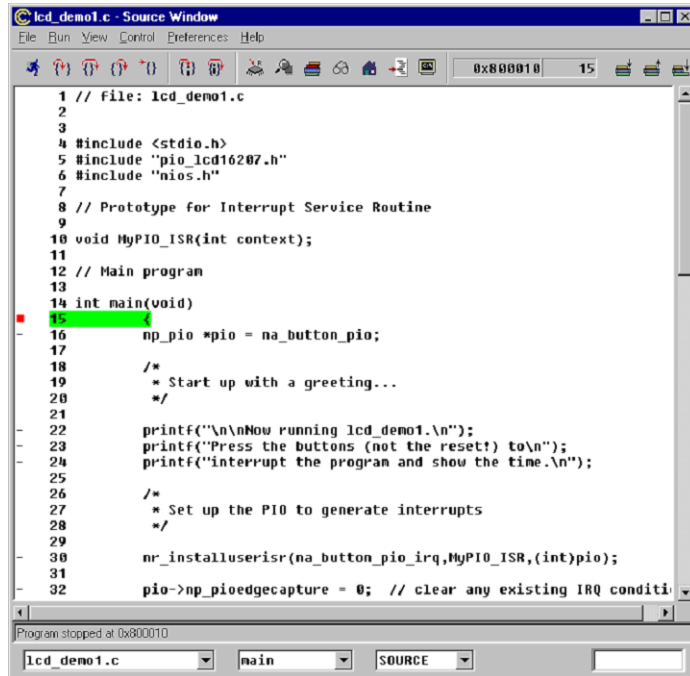
COM1 is the default port. If you are using COM2, use the command `nios-run -t -p com2`.

3. Switch to the other SDK Shell, which is in command mode (not terminal mode).
4. Download **lcd_demo1.srec** (via JTAG) to the Nios development board and launch Insight by typing one of the following commands (Figure 4 shows the GUI):

```
nios-debug lcd_demo1.srec ↵  
nios-debug lcd_demo1.out ↵
```

lcd_demo1 is loaded into the program memory, which is at address 0x800000 for Stratix and Cyclone example designs. The **nios-debug** command automatically sets a software breakpoint at `main()` denoted by the red square in the left margin.

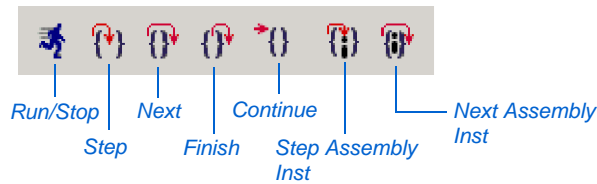
You can use the **nd** command instead of **nios-debug**. To learn more about the **nios-debug** command options, type `nios-debug --help` ↵ at the Nios SDK Shell prompt or refer to the *Nios Embedded Processor Software Development Reference Manual*.

Figure 4. GnuPro Insight with lcd_demo1 Loaded

The line highlighted in green is the next line of code that will be executed. Green indicates that Insight is properly connected to the CPU.

If the highlight is purple, there is a communication problem and you will not be able to perform debugging. Close Insight, verify that the board and cables are connected properly, and relaunch Insight using the **nios-debug** command. When launching Insight, the **nios-debug** utility prints status messages that can help you determine why it cannot connect.

Commands for executing and stepping through code are located in the Control menu. Alternatively, you can use the task bar buttons (see [Figure 5](#)).

Figure 5. Insight Task Bar Button for Executing Code

The Insight task bar buttons are described below.

- **Run**—The **Run** button lets you select a new program to debug. For Nios debugging, a script is always used to select the program; therefore, you do not need to use this button.
 - **Stop**—The **Run** button turns into the **Stop** button during code execution. The **Stop** button stops the code execution.
 - **Step**—The **Step** button steps to the next executable line of the source code; steps into called functions.
 - **Next**—The **Next** button steps to the next executable line of the source code in the current file; steps over called functions.
 - **Finish**—The **Finish** button finishes execution of the current subroutine. If the **Finish** button is selected while in a function, it finishes the function and returns to the line that called the function.
 - **Continue**—The **Continue** button continues execution until a breakpoint, watchpoint, or other exception is encountered, or when execution is complete.
 - **Step Assembly Inst**—The **Step Assembly Inst** button steps to the next assembler instruction. It steps into subroutines.
 - **Next Assembly Inst**—The **Next Assembly Inst** button steps to next assembler instruction. It executes subroutines and steps to the subsequent instruction.
5. Breakpoints signal (to the debugger) a stopping point in the code. Set a software breakpoint on line 23 by clicking the hash mark in the left margin next to line 23. A red square appears, which indicates that the breakpoint is set.

You can set a breakpoint on any line that has a '-' in the left margin.

You can disable breakpoints in the **Breakpoints Window** (View menu). When a breakpoint is disabled, the square in the left margin is black.

6. Begin program execution by clicking **Continue**. This action causes the processor to run until a breakpoint, or other exception is reached or execution completes. The processor stops at the breakpoint you set at line 23 and this line is highlighted in the GUI. The following line displays in the Nios SDK Shell (in terminal mode):

```
Now running lcd_demo1.
```

7. Click **Step** to step into the `printf` function on line 23. The source window opens the **nios_printf.c** file and highlights the first executable line of code in the `nr_printf` routine.
8. Click **Step** to step into the `PrivatePrintf` routine. The source window opens **nios_sprintf.c** and highlights the first line of executable code in the `PrivatePrintf` routine.
9. Click **Next** to step through a few instructions. Unlike **Step**, **Next** does not step into subroutine calls.
10. Click **Continue** to continue running the program. The following lines display in the other Nios SDK Shell (the one in terminal mode not command mode):

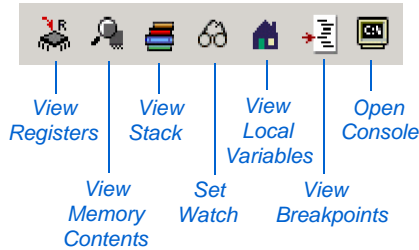
```
Press the buttons (not the reset!) to  
interrupt the program and show the time.
```

After these messages, the * character prints continuously while the program runs. The following 3 messages continuously scroll across the LCD screen:

```
Hello! This is Nios.  
Your message goes here!  
You could show another message, too.
```

Because there are no more breakpoints and the program is in a continuous while loop, the program runs until you click **Stop**.

Insight provides many views into your system. You can access the views via the View menu or the View task bar. [Figure 6](#) shows the View task bar and the function of each button.

Figure 6. Insight Task Bar View Options

The Insight task bar view windows are described below:

- **Open Console**—The **Open Console** window provides the traditional command-line interface to GDB, and is very similar to the command-line you see when GDB is run with the `-nw` option.
- **View Breakpoints**—The **View Breakpoints** window lists the breakpoints that exist in the program and facilitates modifying (i.e., making them temporary or normal, disabled or enabled) and removing breakpoints.
- **View Local Variables**—The **View Local Variables** window displays all local variables in scope. You can use it to visualize and edit local variables.
- **Set Watch**—You can use the **Set Watch** expressions window to inspect and edit any expression, i.e., including global variables, static variables, local variables, function arguments, and registers.
- **View Stack**—The **View Stack** window allows you to view the call stack frame and jump between levels of the stack frame.
- **View Memory Contents**—The **View Memory Contents** window allows you to display and edit memory contents.
- **View Registers**—The **View Registers** window lists all of the registers and their contents for the selected stack frame. It permits viewing the contents of registers in different formats, and editing register values and some display customizations. The **View Registers** window updates the register contents in the display to match the stack frame currently being viewed in the **Source Window** and **Stack Window**. While the program is running you cannot open a new view window and the value of the items in the windows will not change. You must stop the program to open a new window or examine the current value of any item in a window.

11. Stop the program by clicking **Stop**.

12. Choose **Registers** (View menu). The **Registers** window opens showing all of the registers and their current values.
13. Choose **Memory** (View menu). The **Memory** window opens showing the contents of the memory at the current instruction address.
14. Choose **Local Variables** (View menu). The **Local Variables** window opens showing all of the local variables and their current values.
15. Step through a few lines of code (if the program stops in Assembly Language code, click the **Step Assembly** or **Next Assembly** button) and observe the values of the registers, memory and local variables change. Any item that changes value is highlighted in blue.

In the following steps, you will use the **Memory** window to write to LEDs D0 through D7 on the Nios development board. You can find the base address (0x920980) of the PIO peripheral for the LEDs (**na_led_pio**) in the **excalibur.h** file in the `<CPU name>_sdk/inc` directory.



Refer to the *Nios PIO Peripheral Data Sheet* for more information on the PIO registers.

1. Type 0x920980 in the **Memory** window's **Address** box.
2. Press Enter to display the contents of this memory location.
3. Choose **Preferences** (Addresses menu). The **Preferences** window opens
4. Choose **Byte** from the **Size** list box and click **OK**. The **Memory** window displays the contents in byte alignment. This view makes it easier to edit the memory contents for the 8-bit PIO.
5. Type 0x55 in address location 0x920980, which is the address of the data register for **na_led_pio**.
6. Press Enter to write the value to memory.



After you press Enter, the value in address location 0x920980 reverts to 0xff. The value reverts because the PIO is output only and you cannot read the contents, you can only write to them.

The even-numbered LEDs from D0 to D7 should be illuminated.

7. Click **Continue**.

8. Press any of the pushbuttons SW0 through SW3 on the Nios development board and watch the LCD screen display the following message:

The time is now <time in milliseconds>

9. Close the Insight debugger to continue with the rest of the tutorial. The **Nios SDK Shell** returns to command mode. Leave both **Nios SDK Shell** windows open to perform the steps in the following section.



If you wish to reload the program and restart the debugging session, you should exit Insight and use the `nios-debug` command again from the Nios SDK Shell prompt. See [Step 4](#) on page 23.

Download, Run & Debug the Program Using the Nios OCI Debug Console

In this section, you will download the `lcd_demo1.srec` to the development board, launch the Nios OCI Debug Console (debug console), and perform basic debugging from the debug console. The Nios OCI Debug Console allows you to perform basic debugging using a command-line interface, similar to a debug monitor interface.

In addition to performing basic debugging, Nios Development Kit, Stratix Professional Edition users can also set complex data breakpoint triggers like breaking execution when a specific data value is stored to a specific address. See the First Silicon Solutions' *Getting Started: System Analyzer for Nios Processor Core* for details.

The design files for this section use the serial port for basic host communication and the JTAG port for debugging communication. You must use the JTAG port to communicate with the Nios OCI Debug Module. You will use the Nios SDK Shell you opened in ["Open the Nios SDK Shell" on page 20](#), which is in command mode to launch the debug console. You will open a second Nios SDK Shell for host communication.



You should have two **Nios SDK Shell** windows open. If you do not, choose **Programs > Altera > Nios <version> > Nios SDK Shell** (Windows Start menu) twice to open two Nios SDK shells.

Perform the following steps to download, run, and debug the program.

1. Type the following command in the shell to enter terminal mode and connect to the serial port on the host PC:

```
nios-run -t -r ↵
```



COM1 is the default port. If you are using COM2, use the command `nios-run -t -p com2`.

2. Switch to the other SDK Shell, which is in command mode (not terminal mode).
3. Download **lcd_demo1.srec** (via JTAG) to the Nios development board and launch the debug console by typing the following command (Figure 7 shows the debug console):

```
nios-console lcd_demo1.srec ←
```

The **nios-console** command downloads the **lcd_demo1** program to the Nios development board and sets the program counter (pc) to the start address indicated in the **.srec**, which is 0x800278 for this design.



You can type **nc** instead of **nios-console**. To learn more about the **nios-console** command options, type `nios-console --help` at the Nios SDK Shell prompt or refer to the *Nios Embedded Processor Software Development Reference Manual*.

Figure 7. Nios OCI Debug Console

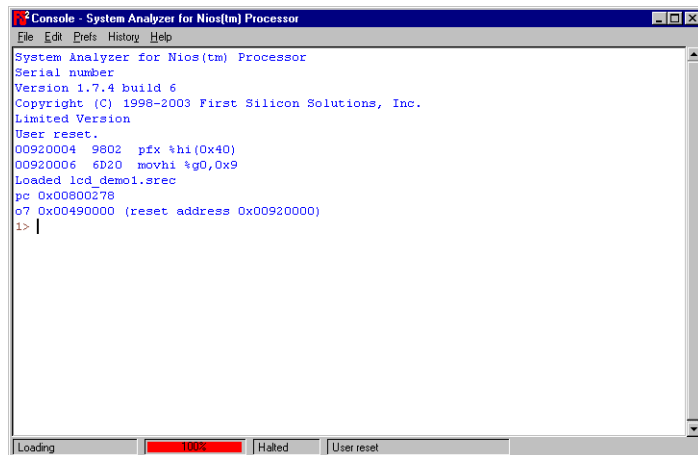


Table 7 describes some useful debug console commands with a brief description. You will use some of these commands in this tutorial.

Table 7. Debug Console Commands

Command	Description
go	Starts executing instructions at the current program counter (pc) address.
help	Used to get help on any command. The usage is <code>help <command></code> .
halt	Stops the processor.
reset	Resets the processor.
pc	Allows you to view the current value of the program counter and change its value.
regs	Allows you to view the registers and change their values.
step	Steps through the code.
bkpt	Sets breakpoints.
dump	Displays the memory contents.
byte, half, word	Writes to memory.

Unless stated otherwise, you should execute the commands in the following steps at the debug console prompt.

- To run **lcd_demo1**, type the following commands at the debug console:

```
go ←
```

After you type the commands, the following lines print to the Nios SDK Shell in terminal mode:

```
Now running lcd_demo1.
Press the buttons (not the reset!) to
interrupt the program and show the time.
```

After these messages, the * character prints continuously while the program runs. The following 3 messages continuously scroll across the LCD screen:

```
Hello!           This is Nios.
Your message goes here!
You could show  another message, too.
```

- Press any of the pushbuttons SW0 through SW3 on the Nios development board and watch the LCD screen display the following message:

```
The time is now <time in milliseconds>
```

6. Set a software breakpoint at 0x80002e by typing the following command at the debug console prompt:

```
bkpt setsw 0x80002e ↵
```

7. Using your favorite text editor, open the **lcd_demo1.objdump** file in the SDK **src** directory. You should see that address 0x80002e contains the first instruction for printing the text “Press the buttons (not the reset!) to”. Therefore, the next time you run **lcd_demo1**, the system should stop immediately before printing this text.

8. Stop the processor by typing the following command:

```
halt ↵
```

9. Move the pc to the beginning of **lcd_demo1** and rerun **lcd_demo1** by typing the following commands:

```
pc 0x800000 ↵  
go ↵
```

The following text appears in the debug console:

```
Emulation started.  
Software breakpoint.  
0080002E 988A pfx %hi(0x1140)  
00800030 3608 movi %o0,0x10
```

The following text appears in the Nios SDK Shell:

```
Now running lcd_demo1.
```

The processor has been stopped at the breakpoint.

10. Examine the registers by typing the following command:

```
regs ↵
```

11. Step through the code by executing the **step** command 4 times in the debug console. After every **step** command, the debug console shows the instruction to which you are stepping. You can compare this output to the **lcd_demo1.objdump** to follow the program execution. Sometimes stepping causes the pc to move ahead 2 instructions because the PFX instruction is combined with the following instruction; therefore, the 2 instructions act as one. See the *Nios Embedded Processor 32-bit Programmer's Reference Manual* for details.



You can step a specific number of instructions by typing “step <number>” (1 is the default).

12. Type the following command to continue running the program:

```
go ↵
```

13. Stop the processor by typing `halt` ↵ at the debug console prompt.
14. Type the following command to clear the breakpoint:

```
bkpt clear 0x80002e ↵
```

15. You can confirm that the breakpoint was cleared by typing `bkpt` ↵ at the prompt. This command list all of the breakpoints.



The command `bkpt clear all` ↵ clears all breakpoints.

16. Write the half word value 0xe8cf to the 7-segment LED by typing the following command at the debug console prompt:

```
half 0x920990 0xe8cf ↵
```

The 7-segment LED display shows the pattern `hI`. This command writes the half word (16-bit) value 0xe8cf to the memory location 0x920990, which is the data register for the PIO peripheral connected to the 7-segment LED.

You can find the base address (0x920990) of the PIO peripheral for the 7-segment LED peripheral (**na_seven_seg_pio**) in the **excalibur.h** file in the `<CPU>_sdk/inc` directory. Refer to the *Nios PIO Peripheral Data Sheet* for more information on the PIO registers.

Rebuild the Software

If you want to change your software, perform the following steps. You can leave the Nios SDK Shell that is in terminal mode open while going through these steps.

1. Close the debug console window by typing `exit` ↵.
2. Modify the code. For this tutorial design, you can modify the **lcd_demo1.c** file and change the messages that display on the LCD screen.
3. Rebuild the code using **nios-build** (step 3 in “[Compile a Program](#)” on page 21).

Download the Software to Flash

4. Redownload the **.srec** and start the debugger using the **nios-debug** or **nios-console** command.

In this section you will download the program to flash so that it executes after the system boots. You should place your code in the user software portion of the flash memory, which is located at 0x40000 (flash base address plus 0x40000). The **srec2flash** utility creates a flash file containing your code and a routine that copies your code to SRAM (the program memory address) before it is executed.

1. Convert the **.srec** to a **.flash** by typing the following command at the Nios SDK shell prompt:

```
srec2flash lcd_demo1.srec ↵
```

This command creates the file **lcd_demo1.flash**, which you can download to the user software portion of the flash memory on the Nios development board.



You can also use a Hexadecimal (Intel-Format) Output File (**.hexout**). Refer to [“Appendix—Using a .hexout” on page 37](#) for more information.

2. Download the flash file by typing the following command:

```
nios-run -x lcd_demo1.flash ↵
```

The **-x** option causes the Nios SDK Shell to exit terminal mode when the download completes.

If you receive a message about not being able to open COM1, type Ctrl-C to make sure you have exited any other terminal mode you may have running Nios SDK Shells.

3. Reset the system by pressing the CPU Reset button on the Nios development board. After the system boots, **lcd_demo1** is copied to SRAM and executed. The **lcd_demo1** program execute in the same manner as when you downloaded it directly to SRAM using the **nios-run** command with the **-x** option.



For more information on the **srec2flash** utility, refer to the *Nios Embedded Processor Software Development Reference Manual*.

Third-Party Development & Debug Tools

Several third-party development and debugging tools are available for use with the Nios development kit. For more information, refer to the *Quick Start Guide for Third-Party Development Tools* that was included with the Nios development kit. More information is also available at www.altera.com/nios.



Notes:

A Hexadecimal (Intel-Format) Output File (**.hexout**) can be stored in flash and can configure an Altera device. Upon power-up, a configuration controller on the Nios development board attempts to load the Altera device with the user hardware portion of the flash, 0x600000 - 0x6FFFFF. If this attempt is unsuccessful, it loads the Altera device with the factory image at 0x700000 - 0x7AFFFF. Therefore, you should always place your hardware image beginning at 0x600000 - 0x6FFFFF. The **hexout2flash** utility automatically creates a flash file, which places the hardware image at this address.

The following steps illustrate how to download a **.hexout** of the **standard_32** reference design to the flash device on the Nios development board.

1. Change to the software tutorial directory by typing the following command at the Nios SDK Shell prompt:

```
cd ../tutorials/Nios_SW_Tutorial_<device family>_<device> ↵
```

2. Convert the **standard_32.hexout** to **.hexout.flash** by typing the following command:

```
hexout2flash standard_32.hexout ↵
```

The **hexout2flash** utility converts the **.hexout** to a **.hexout.flash**, which you can download to the flash on the Nios development board. After you execute the **hexout2flash** command, it creates **standard_32.hexout.flash** in the current directory.

3. Download the **standard_32.hexout.flash** file to the user hardware portion of flash by typing the following command:

```
nios-run -x standard_32.hexout.flash ↵
```

It takes a few minutes to download the file. The **-x** option exits terminal mode after the download completes

4. Reset the Nios development board so the configuration controller loads the new image into the Altera device.

5. Verify that the new image was loaded into the FPGA by checking that the user LED on the Nios development board is illuminated.

You can use other methods besides the ones described in this tutorial, to communicate with the Nios development board and debug Nios systems. You can use the serial or JTAG ports for communication; the hardware design determines which one you use. The Nios SDK Shell **nios-run** utility checks the **Primary Serial Port** setting in SOPC Builder to determine which serial port (or JTAG) to use for stdio and GERMS communication.

nios-debug defaults to using JTAG for downloading if the Nios OCI Debug Module is enabled. You can override this default with the `-gdb = comX` and `-p = comX` options, where X is the COM port number.

If you only have a serial port on your host machine, you can use the GERMS monitor with the GNU Debugger (GDB) (or the GUI version called Insight) for debugging. This appendix describes how to connect to GDB using serial communication. The steps for debugging are the same once the host and target are connected.

GERMS Monitor

The Nios development kit includes the GERMS monitor program, which provides basic facilities for communicating with any development board, including reading and writing to memory, downloading software code and instructing the CPU to begin executing code at a particular address. The GERMS monitor also controls the boot process. The Altera-provided Nios reference designs contain a GERMS monitor in on-chip memory.




Refer to the *Nios Embedded Processor Software Development Reference Manual* for more information.

Insight: GNU Debugger

Insight is the visual debugger included with the GNUPro tools, which are installed with the Nios development kit. Insight provides a graphical user-interface alternative to the command-line interface of GDB. The Insight debugger works with GDB and lets you see inside a program while it executes. This section describes how to use GDB and Insight using serial communication. To view host communication during debugging, open a Nios SDK Shell and enter terminal mode using the command `nios-run -t`.

To drive Insight using the menu, follow these steps:

 Set the debug communications UART to a baud rate of 115,200.

1. Build your code with the GDB stub using the following command:

```
nios-build -d hello_nios.c ↵
```

2. Download a standalone version of GDB with a breakpoint at main:

```
nios-run -x ../lib/nios_gdb_standalone.srec ↵
```


The `-x` option exits terminal mode.

3. Invoke Insight:

```
nios-elf-gdb ↵
```

4. Set up your project:

- a. Choose **Open** (File menu).
- b. Select **hello_nios.out**.
- c. Choose **Source** (File menu).
- d. Select **setup.gdb**.

 An example **setup.gdb** script is shown in [Figure 8](#). It sets the architecture and adds the **inc** and **lib** directories. Altera recommends that you create a **setup.gdb** script, which ensures that you have installed and are using the correct Cygwin version. (The **nios-debug -s** option creates a **.gdb** script in the current directory.)

If you are targeting a 16-bit Nios system, you should change the first line to target **nios16** and lines 2 and 3 should point to the **m16** directory. The directory on line 3 may change for different versions of SOPC Builder (if a new version of Cygwin is included).

Figure 8. Setup GDB Script

```

set architecture nios32
dir /cygdrive/c/altera/kits/nios/bin/nios-gnupro/nios-elf/lib/m32
dir /cygdrive/c/altera/kits/nios/bin/nios-gnupro/lib/gcc-lib/nios-elf/2.9-nios-
010801-20030227/m32
dir ../lib
dir ../inc
dir .

```

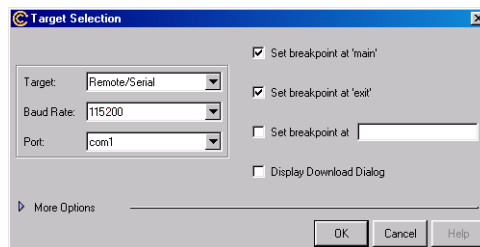
- e. Choose **Target** (File menu).
- f. In the **Target Selection** dialog box, ensure that the following parameters are appropriately set (see [Figure 9](#)):

Target: Remote/Serial

Baud Rate: <baud rate of debug port>

Port: <debug port>

- g. Click **OK**.

Figure 9. Target Selection Dialog Box

You are now ready to debug your code.



For information on using Insight and GDB, refer to the GNUPro Toolkit documentation in the Nios Development Kit documents directory.

Gprof: GNU Profiler

Gprof is the GNU profiler. You can use profiling to learn where the program spends time and which function called another function during execution. This information reveals the sections of code that are slower than expected and that you may want to rewrite for faster code execution. Gprof can also show which functions are being called more or less often than you expected, helping you find bugs that may otherwise go unnoticed.

Because Gprof uses information collected during the actual execution of your program, you can use it on programs that are too large or too complex to analyze by reading the source code. However, how you run your program affects the information that shows up in the profile data. For example, if you don't use a program's feature while it is being profiled, profile information is not generated for that feature.



For more information on Gprof, go to:

www.sources.redhat.com/binutils/docs-2.10/gprof.html.

Requirements

Gprof requires the following items:

- A full-featured timer, named `timer1`, which cannot be used anywhere else in your code.
- A UART to download profiling information to your PC. The UART can be shared because it is only needed after the user's program is complete.
- Because the profiling data is downloaded after the user's program is complete, the program must exit normally.
- Extra memory to store the profiling data. The amount of memory needed depends on your profile settings. Using the default settings, a 32-bit Nios system needs an extra 100% of code size of memory and a 16-bit Nios system would need an extra 25%.

Debugging Code with Gprof

This section describes an example using Gprof. The example assumes you are attempting to profile the Dhrystone benchmark program, `dhry.c`. To use Gprof, first compile the program in the Nios SDK Shell with the `gcc` compiler option `-pg`:

```
nios-build dhry.c -cc -pg ↵
```

The file `nios_gprof.c` is included in your compilation. The program provides two operations.

- The profiling code keeps track of the function calls. Using the `-pg` command, 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.

- The profiling code keeps track of the amount of time spent in each function using a timer named `timer1` that periodically samples the program counter. The rate of sampling is set in **nios_gprof.c** by the constant `TIMER_SAMPLE_RATE` (the default sampling rate is 10,000), in interrupts per second. Each timer interrupt increments one of many *buckets*, which represent ranges of code memory (i.e., code chunk size).

To use Gprof, perform the following steps:

1. Use the `| tee` option to download the program and run the **.srec** while copying the screen output from the PC to a text file. For example:

```
nios-run dhry.srec | tee dhry.txt ↵
```

Besides the standard output of your code, profiling data is also output, preceded by three pound symbols (i.e., `###`).



Your program will run much slower due to the large number of interrupts caused by the profiling code.

2. Convert the text file into a **gmon.out** file using the Perl script **nios-gprof-convert**. This script converts the data preceded by `###` to binary and saves it to the file **gmon.out**. For example:

```
nios-gprof-convert dhry.txt ↵
```

3. Run **nios-elf-gprof** on the **objfile** and **gmon.out**, directing the output to the profile file:

```
nios-elf-gprof -C -q dhry.out gmon.out >
dhry.profile ↵
```

The Gprof example uses the following options:

- **-C**—If you include this option, Gprof prints a list of functions and the number of times each was called.
- **-q**—If you use this option, the call graph shows which functions called it, which other functions it called, and how many times (see [Figure 10](#)). It also estimates how much time was spent in the subroutines of each function. You can use this information to find function calls that use a lot of time.

Gprof provides two additional output formats. To use either one, replace the `-q` option with one of the following options (see Figures 10 through 12):

- **-p**—The flat profile option shows how much time your program spent in each function, and how many times that function was called. If you simply want to know which functions burn most of the cycles, it is stated concisely here.
- **-A**—The annotated source listing option is a copy of the program's source code, labeled with the number of times each line of the program was executed.



For more information and a full description of all the options available, refer to the GNU Gprof manual, at www.sources.redhat.com/binutils/docs-2.10/gprof.html.

Keep the following items in mind when reading the profiling data:

- Time is calculated in ticks, not seconds. Divide time by `TIMER_SAMPLE_RATE` (default is 10,000) for seconds.
- Functions from **nios_gprof.c** are included in your profile, for example:
 - `-internal_mcount`
 - `-sbrk`

Figure 10. Profile Data for dhry.c Using Call Graph (-C Option)

granularity: each sample hit covers 4 byte(s) for 0.37% of 267.00 seconds

index	%time	self	descendents	called/total called+self called/total	parents name index children
[1]	48.3	129.00	0.00		<spontaneous> internal_mcount [1]

		16.00	42.50	1/1	_past_main [3]
[2]	21.9	16.00	42.50	1	main [2]
		9.00	8.50	100/100	Proc_1 [5]
		6.00	1.67	100/100	Func_2 [9]
		6.00	0.00	100/100	Proc_8 [10]
		4.00	0.00	100/100	Proc_2 [14]
		3.33	0.00	200/300	Func_1 [12]
		2.50	0.00	100/100	Proc_4 [20]
		1.00	0.00	100/100	Proc_5 [22]
		0.50	0.00	100/300	Proc_7 [21]

[3]	21.9	0.00	58.50		<spontaneous> _past_main [3]
		16.00	42.50	1/1	main [2]

[4]	7.5	20.00	0.00		<spontaneous> nios_mcount [4]

		9.00	8.50	100/100	main [2]
[5]	6.6	9.00	8.50	100	Proc_1 [5]
		3.50	0.50	100/100	Proc_3 [15]
		4.00	0.00	100/100	Proc_6 [16]
		0.50	0.00	100/300	Proc_7 [21]

[6]	6.4	17.00	0.00		<spontaneous> _sbrk [6]

[7]	4.1	11.00	0.00		<spontaneous> sbrk [7]

[8]	3.7	10.00	0.00		<spontaneous> strcmp [8]

		6.00	1.67	100/100	main [2]
[9]	2.9	6.00	1.67	100	Func_2 [9]
		1.67	0.00	100/300	Func_1 [12]

		6.00	0.00	100/100	main [2]
[10]	2.2	6.00	0.00	100	Proc_8 [10]

[11]	2.1	5.50	0.00		<spontaneous> sbrk_done [11]

Figure 11. Profile Data Using Flat Profile (-p Option)

granularity: each sample hit covers 4 byte(s) for 0.06% of 1733.00 seconds

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
81.5	1412.00	1412.00				txCharWait [1]
6.5	1524.00	112.00				internal_mcount [2]
2.0	1558.00	34.00				PrivatePrintf [5]
1.6	1586.00	28.00				nios_mcount [6]
1.3	1608.00	22.00				_sbrk [7]
0.9	1623.00	15.00				sbrk [9]
0.9	1638.00	15.00				udivmodsi4 [10]
0.8	1652.00	14.00	1	14000.00	54000.00	main [3]
0.8	1666.00	14.00				strcmp [11]
0.6	1676.00	10.00	100	100.00	163.33	Proc_1 [8]
0.5	1684.00	8.00	100	80.00	90.00	Func_2 [12]
0.4	1691.50	7.50				nr_uart_txchar [13]
0.3	1696.50	5.00	100	50.00	50.00	Proc_8 [14]
0.3	1701.50	5.00				__mulhi3 [15]
0.2	1705.50	4.00				__divsi3 [16]
0.2	1708.50	3.00	300	10.00	10.00	Func_1 [18]
0.2	1711.50	3.00	100	30.00	30.00	Proc_2 [19]
0.2	1714.50	3.00	100	30.00	36.67	Proc_3 [17]
0.2	1717.50	3.00				strlen [20]
0.1	1719.50	2.00	300	6.67	6.67	Proc_7 [21]
0.1	1721.50	2.00	100	20.00	20.00	Proc_4 [22]
0.1	1723.50	2.00	100	20.00	20.00	Proc_5 [23]
0.1	1725.50	2.00	100	20.00	20.00	Proc_6 [24]
0.1	1726.50	1.00				DoDisableTimerInterrupt [25]
0.1	1727.50	1.00				__udivsi3 [29]
0.1	1728.50	1.00				printf [26]
0.1	1729.50	1.00				sbrk_done [27]
0.1	1730.50	1.00				sbrk_stillRoom [28]
0.0	1731.00	0.50				rxCharA [30]
0.0	1731.00	0.00	100	0.00	0.00	Func_3 [31]

Figure 12. Profile Data in Annotated Source (-A Option)

```

Proc_2 (Int_Par_Ref)
/*****
    /* executed once */
    /* *Int_Par_Ref == 1, becomes 4 */
    One_Fifty *Int_Par_Ref;
100 -> {
    One_Fifty Int_Loc;
    Enumeration Enum_Loc;

    Int_Loc = *Int_Par_Ref + 10;
    do /* executed once */
        if (Ch_1_Glob == 'A')
            /* then, executed */
            {
                Int_Loc -= 1;
                *Int_Par_Ref = Int_Loc - Int_Glob;
                Enum_Loc = Ident_1;
            } /* if */
        while (Enum_Loc != Ident_1); /* true */
    } /* Proc_2 */

Proc_3 (Ptr_Ref_Par)
/*****
    /* executed once */
    /* Ptr_Ref_Par becomes Ptr_Glob */

    Rec_Pointer *Ptr_Ref_Par;

100 -> {
    if (Ptr_Glob != Null)
        /* then, executed */
        *Ptr_Ref_Par = Ptr_Glob->Ptr_Comp;
    Proc_7 (10, Int_Glob, &Ptr_Glob->variant.var_1.Int_Comp);
    } /* Proc_3 */

Proc_4 () /* without parameters */
/*****
    /* executed once */
100 -> {
    Boolean Bool_Loc;

    Bool_Loc = Ch_1_Glob == 'A';
    Bool_Glob = Bool_Loc | Bool_Glob;
    Ch_2_Glob = 'B';
    } /* Proc_4 */

Proc_5 () /* without parameters */
/*****
    /* executed once */
100 -> {
    Ch_1_Glob = 'A';
    Bool_Glob = false;
    } /* Proc_5 */

```

Changing Gprof Settings

In some instances, the Gprof 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 *<installation directory>/sdk/lib* directory.

The sampling rate is set by the constant `TIMER_SAMPLE_RATE` (default is 10,000) in interrupts per second. If the interrupt rate is too fast (i.e., causing counters to overflow), decrease the sampling rate. If the interrupt rate is too slow (i.e., causing non-repeatable, coarse results), increase the sampling rate.

Each timer interrupt increments one of many buckets, which represent ranges of code memory (code chunk size), defined by `HISTFRACTION`.

- The default for a 32-bit Nios system is 2 bytes for a code chunk size (i.e., `HISTFRACTION` is 2), which is a single Nios instruction. This setting increases your data memory footprint by 100% of your code size.
- The default for a 16-bit Nios system is 8 bytes (due to memory restrictions) for a code chunk size, which is four Nios instructions. This setting causes the counter buffer to equal 25% of code size.

You can use a larger code chunk size (i.e., set `HISTFRACTION` to 16); however, the negative effect is that the profile occasionally attributes a sample to a different function than the program counter was actually in. However, using a larger code chunk size has the positive effect of reducing the amount of required memory.

To implement these changes, edit the **sdk/lib/nios_gprof.c** file, and re-create the library by typing `make all` in the **lib** directory.



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.

References

Reference documents for this appendix include:

- *GNU Gprof Profiler Manual, Rev.*
- *Debugging with GDB: The GNU Source-Level Debugger*

The following tips make using the Nios SDK Shell easier.

Changing to the SDK Directory

The Nios SDK Shell opens to the `/altera/kits/nios/examples` directory by default. To change to SDK `src` directory for this tutorial, type the following command:

```
cd ../tutorials/nios_sw_tutorial_cyclone_1c20/cpu_sdk/src ↵
```

Keystroke Shortcuts

The Nios SDK Shell supports command completion of unique commands with the Tab key and pattern matching with the * key. Therefore, instead of typing a whole string, you can type a few letters. For example:

- Instead of typing the word `tutorials`, type `tut` and press the Tab key.
- Instead of typing `<CPU>_sdk`, type `*sdk`.

Using these keyboard shortcuts, the keystrokes needed to change to the Nios **tutorial** directory are:

```
cd ../tut<press the Tab key>Nios_SW<press the Tab key>/*sdk/src ↵
```

Utility Usage Shortcuts

You can use the command `nb` instead of `nios-build`. For example:

```
nb hello_nios.c ↵
```

You can use the command `nr` instead of `nios-run`. For example:

```
nr hello_nios.srec ↵
```

You can use the command `nc` instead of `nios-console`. For example:

```
nc hello_nios.srec ↵
```

You can use the command **nd** instead of **nios-debug**. For example:

```
nd hello_nios.srec ↵
```

B

- before you begin
 - software tutorial 19
- breakpoints
 - Insight 25

C

- commands
 - debug console 30
- configurable processors
 - about 13
 - custom instructions 14
 - data and instruction caches 14
 - DMA 14
 - memory interfaces 14
 - multipliers 13
 - peripherals 14
 - simultaneous multi-master architecture 14

D

- debug
 - board communication 37, 39, 49
 - JTAG 22, 29
- debug console
 - bkpt 31
 - byte 31
 - commands 30
 - dump 31
 - go 31
 - hal 31
 - half 31
 - help 31
 - pc 31
 - regs 31
 - reset 31

- step 31
 - word 31
- debug tools
 - third-party 35
- debugging
 - Insight 22
 - Nios OCI Debug Console 29
 - other methods 39
 - via Insight GUI 40
 - with Gprof 42
- design flow 9
 - predesign activity 10
- development board
 - communicating with 37, 39, 49
- development flow
 - hardware/software 9
- downloading
 - flash 34

F

- flash
 - downloading to 34

G

- GERMS monitor
 - debug 39
- GNUPro debugger 39
- GNUPro Toolkit 10, 12
- GNUprofiler 41
- Gprof 41
 - debugging code with 42
 - requirements 42
 - settings 48

H

hardware and software requirements
 software tutorial 11
hardware development flow 9

I

Insight 22
 breakpoints 25
 Continue task bar button 25
 Finish task bar button 25
 GNU Debugger 39
 Next Assembly Inst task bar button 25
 Next task bar button 25
 Open Console task bar button 27
 Run task bar button 25
 Set Watch task bar button 27
 Step Assembly Inst task bar button 25
 Step task bar button 25
 Stop task bar button 25
 task bar 25
 task bar view options 27
 View Breakpoints task bar button 27
 View Local Variables task bar button 27
 View Memory Contents task bar button 27
 View Registers task bar button 27
 View Stack task bar button 27

J

JTAG communication 22, 29

N

nb 49
nc 49
nd 50
Nios development board
 communicating with 37, 39, 49
Nios OCI Debug Console 13
Nios OCI debug console 29
Nios OCI Debug Module 12
Nios SDK Shell 13
 commands 20
 hexout2flash 20
 nios-build 20

 nios-console 20
 nios-debug 20
 nios-elf-gprof 20
 nios-elf-objdump 20
 nios-elf-size 20
 nios-run 20
 srec2flash 20
 tips 49
 using 20
nios-build
 shortcut 49
 software tutorial 21
nios-console
 software tutorial 23, 30
nios-run
 shortcut 49, 50
 software tutorial 23, 29
nr 49

P

predesign activity 10
programs
 running 31

R

rebuilding code
 software tutorial 33

S

SDK 19
 using 19
software development flow 9
software development kit 19
software tutorial
 before you begin 19
 compile program 21
 debug program 22, 29
 download program 22, 29
 download to flash 34
 files 12
 hardware and software requirements 11
 nios-build 21
 nios-console 23, 30
 nios-run 23, 29

- opening Nios SDK Shell 20
- rebuilding code 33
- run program 22, 29
- software tools 12
- third-party tools 35
- using Nios SDK Shell 20
- software used
 - software tutorial 12
- SOPC Builder 10, 12
- system definition 10
 - CPU 10
 - peripherals 10
- system requirements 10

T

- third-party
 - debug tools 35
- tips
 - Nios SDK Shell 49

U

- using SOPC Builder output 19



Notes: