

Nios[®] II

Creating Multiprocessor Nios II Systems

Tutorial



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

Copyright © 2007 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, maskwork 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.



Printed on recycled paper

TU-N2033005-1.3



I.S. EN ISO 9001

About This Tutorial

Revision History	v
How to Contact Altera	v
Referenced Documents	vi
Typographic Conventions	vi

Chapter 1. Creating Multiprocessor Nios II Systems

Introduction	1-1
Benefits of Multiprocessor Systems	1-2
Nios II Multiprocessor Systems	1-2
Hardware Design Considerations	1-3
Autonomous Multiprocessors	1-3
Multiprocessors that Share Resources	1-4
Sharing Resources in a Multiprocessor System	1-4
Sharing Memory	1-6
The Hardware Mutex Core	1-7
Nios II Systems Without a Mutex Core	1-8
Sharing Peripherals Between Multiple Processors	1-8
Multiprocessors and Overlapping Address Space	1-9
Software Design Considerations	1-10
Program Memory	1-10
Boot Addresses	1-14
Running and Debugging Multiprocessor Systems from the Nios II IDE	1-16
Design Example	1-17
Hardware and Software Requirements	1-17
Creating the Hardware System	1-17
Creating Software for the Multiprocessor System	1-26
Starting the Nios II IDE	1-26
Creating a Software Project for cpu1	1-27
Creating a Software Project for cpu2	1-29
Creating a Software Project for cpu3	1-30
Building the Software Projects	1-31
Setting up the Nios II IDE for Multiprocessor Debug	1-31
Creating a Run/Debug Configuration for Each Processor	1-32
Creating a Multiprocessor Collection	1-33
Starting the Multiprocessor Collection	1-35
Debugging the Software Projects on the Board	1-36

This tutorial describes the features of the Altera® Nios® II processor and SOPC Builder tool that are useful for creating systems with two or more processors. The tutorial provides an example design that guides you through a step-by-step process for building a multiprocessor system containing three processors that all share a memory buffer. It shows you how to use the Nios II Integrated Development Environment (IDE) to create and debug three software projects, one for each processor in the system.



Refer to the *Nios II Embedded Design Suite Release Notes* and *Nios II Embedded Design Suite Errata* for the latest features, enhancements, and known issues in the current release.

Revision History

The following table shows the revision history for this tutorial.

Date and Document Version	Changes Made	Summary of Changes
December 2007 v1.3	Update for Quartus II 7.2 release: minor text changes.	—
May 2007 v1.2	Updated for Quartus II 7.1 release.	—
May 2006 v1.1	Updated for Quartus II 6.0 release.	—
April 2005 v1.0	Initial release.	—

How to Contact Altera

For the most up-to-date information about Altera® products, refer to the following table.

Contact (1)	Contact Method	Address
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	custrain@altera.com
Product literature	Website	www.altera.com/literature

Contact (1)	Contact Method	Address
Altera literature services	Email	literature@altera.com
Non-technical support (General)	Email	nacomp@altera.com
(Software Licensing)	Email	authorization@altera.com

Note to table:

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

Referenced Documents








This tutorial references the following documents:

- [Mutex Core chapter in volume 5 of the Quartus II Handbook](#)
- [Nios II Embedded Design Suite Errata](#)
- [Nios II Embedded Design Suite Release Notes](#)
- [Nios II Flash Programmer User Guide](#)
- [Nios II Hardware Development Tutorial](#)
- [Nios II Software Developer's Handbook](#)

Typographic Conventions

This document uses the typographic conventions shown in the following table.

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>.pdf 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."

Visual Cue	Meaning
Courier type	Signal and port names are shown in lowercase Courier type. Examples: <code>data1</code> , <code>tdi</code> , <code>input</code> . Active-low signals are denoted by suffix <code>n</code> , e.g., <code>resetn</code> . Anything that must be typed exactly as it appears is shown in Courier type. For example: <code>c:\qdesigns\tutorial\chiptrip.gdf</code> . Also, sections of an actual file, such as a Report File, references to parts of files (e.g., the AHDL keyword <code>SUBDESIGN</code>), as well as logic function names (e.g., <code>TRI</code>) are shown in Courier.
1., 2., 3., and a., b., c., etc.	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.
	A caution calls attention to a condition or possible situation that can damage or destroy the product or the user's work.
	A warning calls attention to a condition or possible situation that can cause injury to the user.
	The angled arrow indicates you should press the Enter key.
	The feet direct you to more information about a particular topic.

Introduction

Any system which incorporates two or more microprocessors working together to perform a task is commonly referred to as a multiprocessor system. Developers using the Altera® Nios®II processor and SOPC Builder tool can quickly design and build multiprocessor systems that share resources. SOPC Builder is a system development tool for creating SOPC design systems based on processors, peripherals, and memories. A Nios II processor system typically refers to a system with a processor core, a set of on-chip peripherals, on-chip memory and interfaces to off-chip memory all implemented on a single Altera device.

This document describes the features of the Nios II processor and SOPC Builder tool that are useful for creating systems with two or more processors. This document provides an example design that guides you through a step-by-step process for building a multiprocessor system containing three processors that all share a memory buffer. Using the Nios II Integrated Development Environment (IDE), you create and debug three software projects, one for each processor in the system.

After completing this document, you will have the knowledge to perform the following:

- Build an SOPC Builder system containing more than one Nios II processor.
- Safely share resources between processors, avoiding data corruption.
- Build software projects for multiprocessor systems using the Nios II IDE.
- Debug multiple software projects running on multiple processors using the Nios II IDE.

This chapter assumes that you are familiar with reading and writing embedded software and that you have read and followed the step-by-step procedures for building a microprocessor system in the *Nios II Hardware Development Tutorial*.



The *Nios II Hardware Development Tutorial* can be found on the Nios II Processor Literature page at www.altera.com/literature/lit-nio2.jsp.

Benefits of Multiprocessor Systems

Multiprocessor systems possess the benefit of increased performance, but nearly always at the price of significantly increased system complexity. For this reason, the use of multiprocessor systems has historically been limited to workstation and high-end PC computing using a complex method of load-sharing often referred to as symmetric multiprocessing (SMP). While the overhead of SMP is typically too high for most embedded systems, the idea of using multiple processors to perform different tasks and functions on different processors in embedded applications (asymmetrical) is gaining popularity. Altera FPGAs provide an ideal platform for developing asymmetric embedded multiprocessor systems, because the hardware can easily be modified and tuned using the SOPC Builder tool to provide optimal system performance. Furthermore, with a powerful integration tool like SOPC Builder, different system configurations can be designed, built, and evaluated very quickly.

Nios II Multiprocessor Systems

The Nios II IDE version 7.1 and higher includes features to help with the creation and debugging of multiprocessor systems. Multiple Nios II processors are able to efficiently share system resources thanks to the multimaster friendly slave-side arbitration capabilities of the system interconnect fabric. Since the capabilities of SOPC Builder now allow users to almost effortlessly add as many processors to a system as desired, the design challenge of building multiprocessor systems no longer lies in the arranging and connecting of hardware components. The design challenge in building multiprocessor systems now lies in writing the software for those processors so they operate efficiently together, and do not conflict with one another.

To aid in the prevention of multiple processors interfering with each other, a hardware mutex core is included in the Nios II Embedded Design Suite (EDS). The hardware mutex core allows different processors to claim ownership of a shared resource for a period of time. This temporary ownership of a resource by a processor prevents the shared resource from becoming corrupted by the actions of another processor.



For more information about the hardware mutex core, see the [Mutex Core chapter in volume 5 of the *Quartus II Handbook*](#).

Performing software debug on multiprocessor systems is made easier with the Nios II IDE, allowing users to launch and stop software debug sessions on different processors with a single operation.

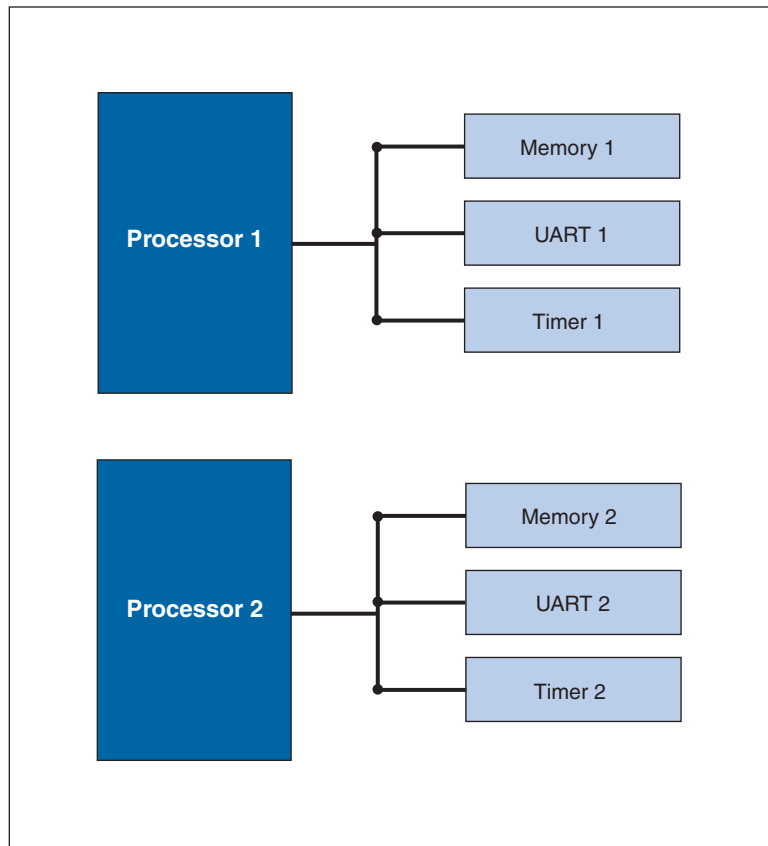
Hardware Design Considerations

Nios II multiprocessor systems are split into two main categories, those that share resources, and those in which each processor is autonomous and does not share resources with other processors.

Autonomous Multiprocessors

While autonomous multiprocessor systems contain multiple processors, these processors are completely autonomous and do not communicate with the others, much as if they were completely separate systems. Systems of this type are typically less complicated and pose fewer challenges because by design, the system's processors are incapable of interfering with each other's operation. Figure 1–1 shows a block diagram of two autonomous processors in a multiprocessor system.

Figure 1–1. Autonomous Multiprocessor System

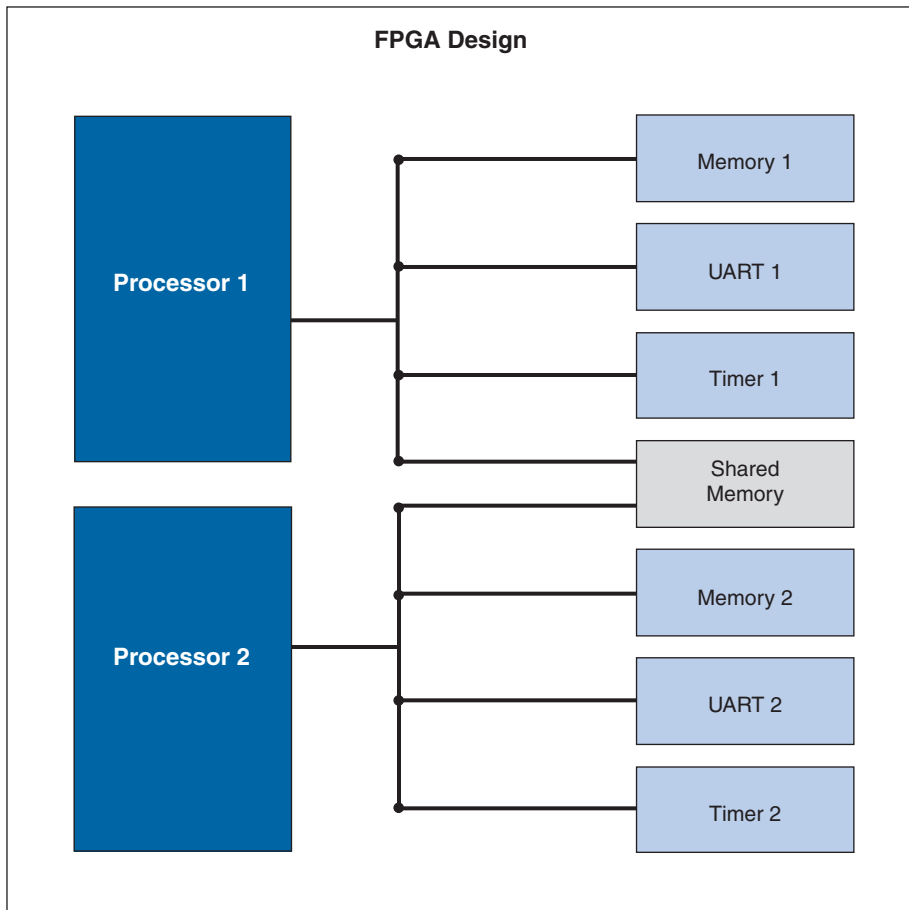


Multiprocessors that Share Resources

Multiprocessor systems that share resources can pose many more challenges. While the Nios II EDS includes features making it possible to reliably implement multiprocessor systems that share resources, the creation of such systems is not necessarily a straightforward venture. Altera recommends that you complete this tutorial and fully understand its recommendations before attempting to create a resource-sharing multiprocessor system.

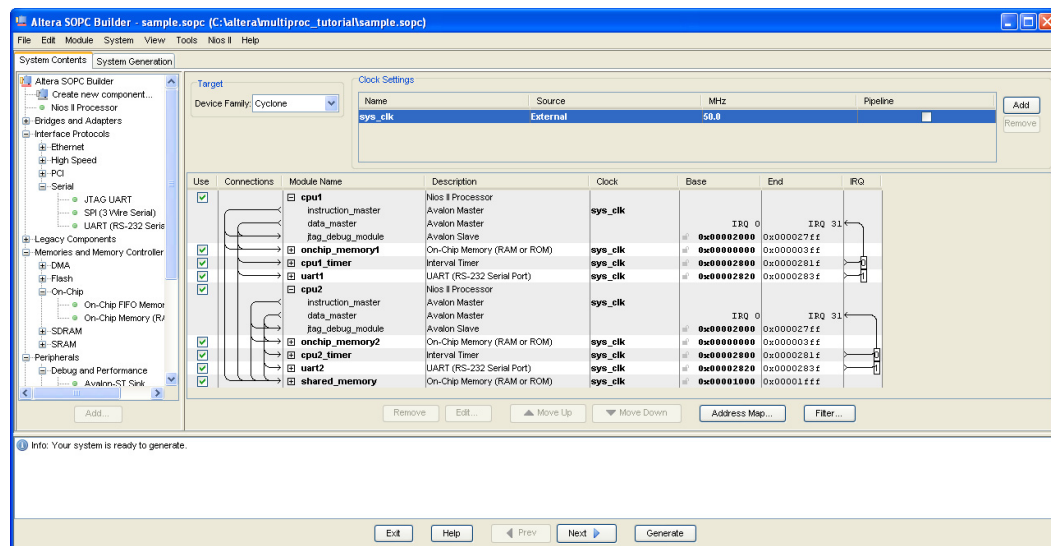
Sharing Resources in a Multiprocessor System

Resources are considered shared when they are available to be accessed by more than one processor. Shared resources can be a very powerful aspect of multiprocessor systems, but care must be taken when deciding which system resources are shared, and how the different processors will cooperate regarding the use of resources. [Figure 1–2](#) shows a block diagram of a sample multiprocessor system in which two processors share an on-chip memory.

Figure 1–2. Multiprocessor System with Shared Resource

Resources can be made shareable by simply connecting them to multiple processor bus masters in the connection matrix of SOPC Builder, but that in no way guarantees that the processors that share them will do so non-destructively. The software running on each processor is responsible for coordinating access to shared resources with the system's other processors. [Figure 1–3](#) shows the sample multiprocessor system in SOPC Builder. The on-chip memory is considered shared because the data master ports of both processors are connected to the same slave port of the memory. Since **cpu1** and **cpu2** are both physically capable of writing blocks of data to the shared memory at the same time, the software for those processors must be written carefully to protect the integrity of the data stored in the shared memory.

Figure 1–3. Multiprocessor System Sharing On-Chip Memory



Sharing Memory

The most common type of shared resource in multiprocessor systems is memory. Shared memory can be used for anything from a simple flag whose purpose is to communicate status between processors, to complex data structures that are collectively computed by many processors simultaneously.

If a memory component is to contain the program memory for more than one processor, each processor sharing the memory is required to use a separate area for code execution. The processors cannot share the same area of memory for program space. Each processor must have its own unique `.text`, `.rodata`, `.rdata`, `.heap`, and `.stack` sections. See [“Software Design Considerations” on page 1–10](#) for information on how to make sure each processor sharing a memory component for program space uses a dedicated area within that memory.

If a memory component is to be shared for data purposes, its slave port must be connected to the data masters of the processors that are sharing the memory. Sharing data memory between multiple processors can be trickier than sharing instruction memory because data memory can be written to as well as read. If one processor is writing to a particular area

of shared data memory at the same time another processor is reading or writing to that area, data corruption will likely occur, causing application errors at the very least, and possibly a system crash.

The processors sharing memory need a mechanism to inform one another when they are using a shared resource, so the other processors do not interfere.

The Hardware Mutex Core

The Nios II processor provides protection of shared resources with its hardware mutex core feature. This hardware mutex core is not an internal feature of the Nios II processor, but a small SOPC Builder component named **mutex**.

The term mutex stands for mutual exclusion, and a mutex does exactly as its name suggests. A mutex allows cooperating processors to agree that one of them should be allowed mutually exclusive access to a hardware resource in the system. This is useful for the purpose of protecting resources from data corruption that can occur if more than one processor attempts to use the resource at the same time.

The mutex core acts as a shared resource, providing an atomic test-and-set operation that allows a processor to test if the mutex is available and if so, to acquire the mutex lock in a single operation. When the processor is finished using the shared resource associated with the mutex, the processor releases the mutex lock. Now another processor may acquire the mutex lock and use the shared resource. Without the mutex, this kind of function would normally require the processor to execute two separate instructions, test and set, between which another processor could also test for availability and succeed. This situation would leave two processors both thinking they successfully acquired mutually exclusive access to the shared resource when clearly they did not.

It is important to note that the mutex core does not physically protect resources in the system from being accessed at the same time by multiple processors. The software running on the processors is responsible for abiding by the rules. The software must be designed to always acquire the mutex before accessing its associated shared resource.

Another kind of mutex, called a software mutex is common in many operating systems for providing the same protection of resources. The difference is that a software mutex is purely a software construct that is used to protect hardware resources from being corrupted by multiple processes running on the same processor. A hardware mutex core is an SOPC Builder component with an Avalon interface that uses logic to

guarantee only one processor is granted the lock of the mutex at any given time. Therefore, if every processor waits until it locks the mutex before using the associated shared resource, the resource is protected from corruption due to simultaneous access by multiple processors. Each processor must first request a lock of the mutex core before accessing the associated shared resource.

Nios II Systems Without a Mutex Core

In most cases, a mutex core should be used to protect any resource shared between multiple processors. However, in some limited cases a mutex core might not be necessary. Such cases might include one-way or circular message buffer arrangements in which only one processor ever writes to a particular set of memory locations. However, sharing resources safely without a mutex core can be complicated. When in doubt, use the mutex core.

Sharing Peripherals Between Multiple Processors

In general, with the exception of the mutex core, Nios II EDS does not support sharing non-memory peripherals between multiple processors.

Sharing peripherals in multiprocessor systems presents some difficult challenges, and is generally considered to lead to inefficient system designs. The biggest problems arise for peripherals with interrupts. If a peripheral is allowed to interrupt all the processors that share it, there is no reliable way to guarantee which processor will respond first and service that interrupt. Additionally, if the peripheral is used as an input device for multiple processors, it becomes difficult to determine which processor is supposed to collect given input from the device. While it is conceivable that a complex system of handshaking could be created to handle these scenarios, such a system is beyond the scope of this document, and is unsupported by the Nios II hardware abstraction layer (HAL) library.



For more information on the Nios II HAL Library, refer to the *Nios II Software Developer's Handbook*.

Altera recommends that each non-memory peripheral be accessible by only one processor in the system. If other processors require use of the peripheral, they should use a message buffer that is either mutex-protected or otherwise multiprocessor-safe when communicating with the processor that is connected to that peripheral.

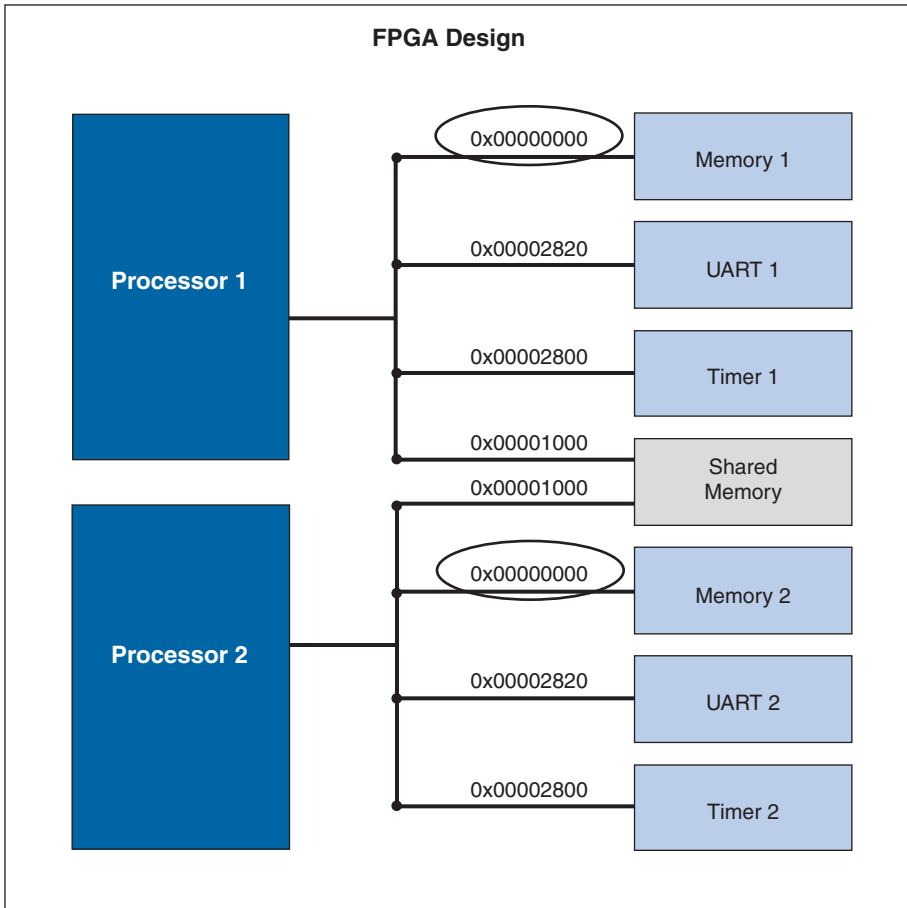
When building any system, especially a multiprocessor system, it is advisable to only make connections between peripherals that require communication. For instance, if a processor runs from and uses only one

on-chip memory, there is no need to connect that processor to any other memory in the system. Physically disconnecting the processor from memories it is not using both saves FPGA resources and guarantees the processor will never corrupt those memories.

In single processor systems, SOPC Builder will usually make intelligent default choices for connecting master and slave components. However, in multiprocessor systems the need to connect different components is very design dependent. Therefore, when designing multiprocessor systems, you should explicitly verify that each component is connected appropriately.

Multiprocessors and Overlapping Address Space

Single-processor systems typically prohibit more than one slave peripheral from occupying the same address space because this arrangement causes conflicts. In multiprocessor systems however, separate slave peripherals can occupy the same base address and not conflict, as long as each of those peripherals is exclusively mastered by a different processor. Because not every slave peripheral is necessarily mastered by every processor, each processor might have a different view of the system. If processor A is connected to a slave peripheral mapped to address 0×4000 , processor B may connect to a separate slave peripheral, also mapped to address 0×4000 , as long as processor A is not connected to processor B's slave peripheral and processor B is not connected to processor A's slave peripheral. In effect, the point-to-point connectivity allows the two processors to have separate address spaces. [Figure 1-4](#) shows a block diagram of the sample multiprocessor system with different slave components mapped to the same base address. [Figure 1-3](#) shows the different slave components mapped to the same base address in SOPC Builder.

Figure 1–4. Multiprocessor Slave Peripherals Mapped to the Same Base Address

Software Design Considerations

Creating and running software on multiprocessor systems is much the same as for single-processor systems, but requires the consideration of a few additional points. Many of the software design issues described in this section are dictated by the system's hardware architecture.

Program Memory

When creating multiprocessor systems, you might want to run the software for more than one processor out of the same physical memory device. Software for each processor must be located in its own unique region of memory, but those regions are allowed to reside in the same

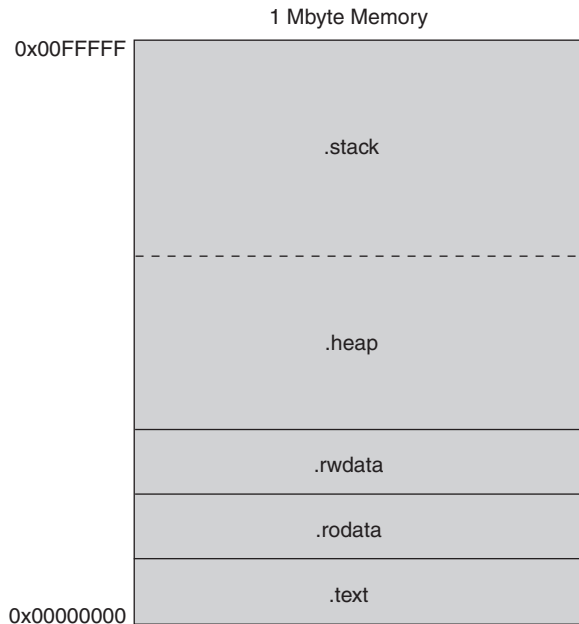
physical memory device. For instance, imagine a two-processor system where both processors run out of SDRAM. The software for the first processor requires 128 KBytes of program memory, and the software for the second processor requires 64 KBytes. The first processor could use the region between 0×0 and $0 \times 1\text{FFFF}$ in SDRAM as its program space, and the second processor could use the region between 0×20000 and $0 \times 2\text{FFFF}$.

Nios II and SOPC Builder provide a simple scheme of memory partitioning that allows multiple processors to run their software out of different regions of the same physical memory. The partitioning scheme uses the exception address for each processor, which is set in SOPC Builder, to determine the region of memory from which each processor will be allowed to run its software. Although the Nios II IDE is ultimately responsible for the linking of the processors' software and determining where the software will reside in memory, the Nios II IDE looks at the exception addresses that were set for each processor in SOPC Builder to calculate where the different code sections will be linked. The Nios II IDE provides each processor its own section within memory from which it can run its software. If the software for two different processors is linked to the same physical memory, then the exception address of each processor is used to determine the base address of the region which that processor's software can occupy. The end address of the region is determined by the next exception address found in that physical memory, or the end of that physical memory, whichever comes first.

Each processor in a single or multiprocessor system has five primary code sections that need to be linked to fixed addresses in memory. These sections are:

- `.text` — the actual executable code
- `.rodata` — any read-only data used in the execution of the code
- `.rwdata` — where read-write variables and pointers are stored
- `.heap` — where dynamically allocated memory is located
- `.stack` — where function-call parameters and other temporary data is stored

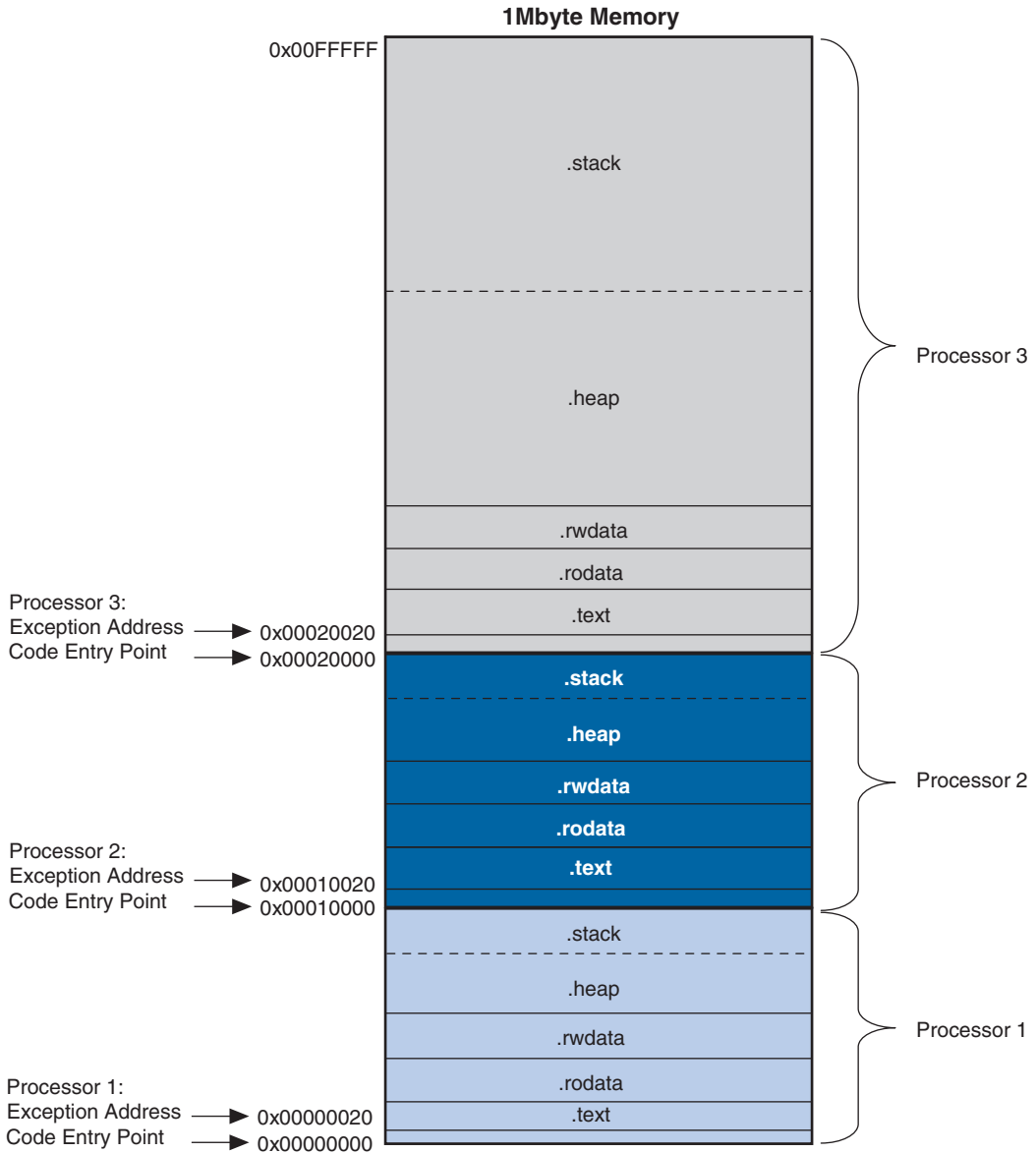
See [Figure 1–5](#) for a memory map showing how these sections are typically linked in memory for a single processor Nios system.

Figure 1–5. Single Processor Code Linked in Memory Map

In a multiprocessor system, it might be advantageous to use a single memory to store all the code sections for each processor. In this case, the exception address set for each processor in SOPC Builder is used to define the boundaries between where one processor's code sections end and where the next processor's code sections begin.

For instance, imagine a system where SDRAM occupies the address range 0x0–0xFFFFF and processors A, B and C each require 64 KBytes of SDRAM to run their software. If you use SOPC Builder to set their exception addresses 64 KBytes apart in SDRAM, the Nios II IDE automatically partitions SDRAM based on those exception addresses. See [Figure 1–6](#) for a memory map showing how the SDRAM is partitioned in this example system.

Figure 1–6. Partitioning of SDRAM Memory Map for Three Processors



The lower six bits of the exception address are always set to 0x20. Offset 0x0 is where the Nios II processor must run its reset code, so the exception address must be placed elsewhere. The offset of 0x20 is used

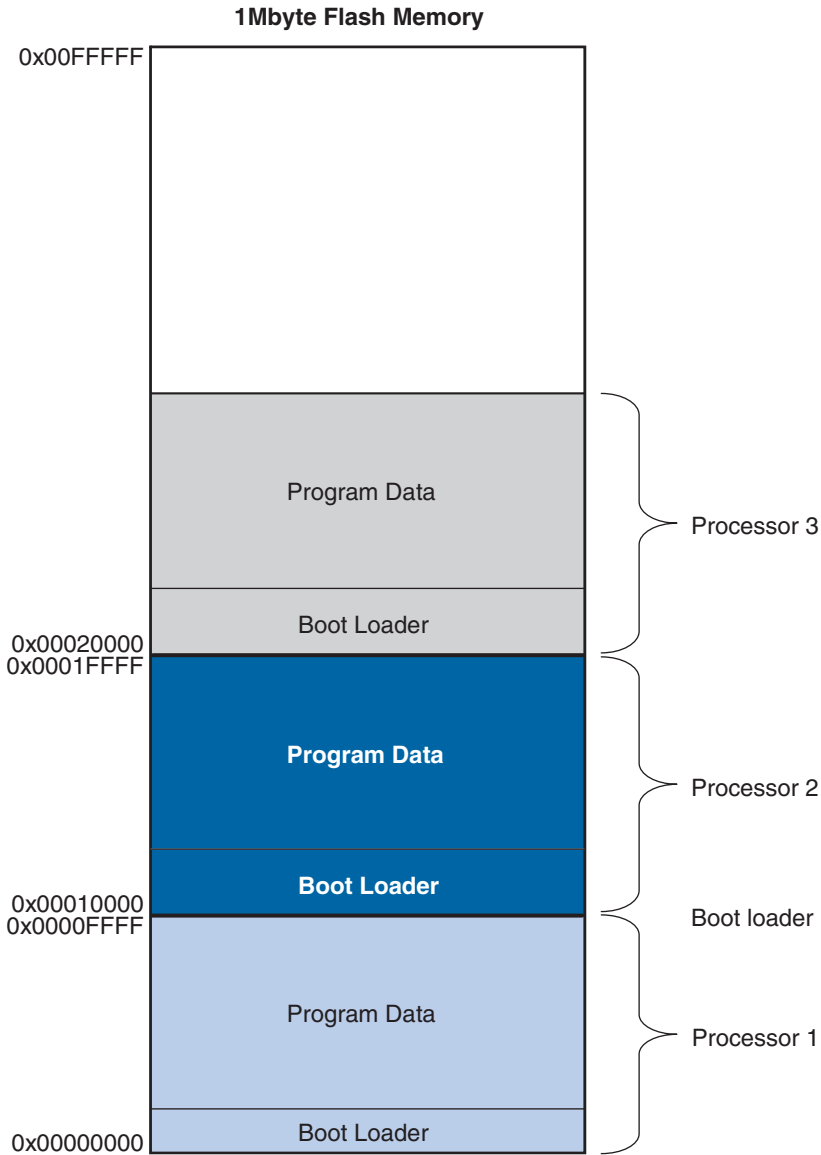
because it corresponds to one instruction cache line. The 0x20 bytes of reset code initialize the instruction cache, and then branch around the exception section to the system startup code.

Care must be taken when partitioning a physical memory to contain the code sections of multiple processors. There are no safeguards in SOPC Builder or the Nios II IDE that guarantee you have provided enough code space for each processor's stack and heap in the partition. If inadequate code space is allotted in memory, the stack and heap might overflow and corrupt the processor's code execution.

Boot Addresses

In multiprocessor systems, each processor must boot from its own piece of memory. Multiple processors might not boot successfully from the same bit of executable code at the same address in the same non-volatile memory. Boot memory can also be partitioned, much like program memory can, but the notion of sections and linking is not a concern as boot code typically just copies the real program code to where it has been linked in RAM, and then branches to the program code. To boot multiple processors out of separate regions with the same non-volatile memory device, simply set each processor's reset address to the location from where you wish to boot that processor. Be sure you leave enough space between boot addresses to hold the intended boot payload. See [Figure 1-7](#) for a memory map of one physical flash device from which three processors can boot.

Figure 1-7. Flash Device Memory Map with Three Processors Booting



The Nios II flash programmer is able to program bootable code for multiple processors into a single flash device. The flash programmer looks at the reset address of each processor and then uses that reset address to calculate the offset within the flash memory where the code is programmed.



For details about the flash programmer, refer to the *Nios II Flash Programmer User Guide*.

Running and Debugging Multiprocessor Systems from the Nios II IDE

The Nios II IDE includes a number of features that can help in the development of software for multiprocessor systems. Most notable is the ability of the Nios II IDE to perform simultaneous on-chip debug for multiple processors. Multiple debug sessions can run at the same time on a multiprocessor system and can pause and resume each processor independently. Breakpoints can also be set individually per processor. If one processor hits a breakpoint, it does not halt or affect the operation of the other processors. Debug sessions can be launched and stopped independently.

Debug sessions for multiple processors can also be launched in a single operation with the Nios II IDE, using a feature called multiprocessor collections. Multiprocessor collections are groups of debug configurations for individual processors that are combined under one configuration name. The benefit of a multiprocessor collection is that any time the collection is launched; the Nios II IDE individually launches each of the single debug configurations in the background. This allows users to launch debug sessions for multiprocessor systems without having to manually launch a session for each processor. Multiprocessor collections can also be stopped with one operation, however pausing and resuming multiprocessor collections together is not currently supported.

The launching and stopping of multiprocessor collections is not simultaneous, meaning the processors in the collection do not start executing code on the same clock cycle. In fact, there might be a delay of a few seconds between the individual processors being started. The purpose of multiprocessor collections is to make it more convenient to launch debug sessions for multiprocessor systems, not to synchronize the processors. If you require the multiple processors to start within a shorter period of time, a separate hardware or software mechanism must be constructed.

Design Example

The following exercise shows you how to build a three-processor Nios II system with SOPC Builder, starting with the **standard** example design as a template. You create three software projects in the Nios II IDE, one for each processor. The software for all three CPUs generates messages to be displayed and uses the hardware mutex core to put those messages in a shared message buffer. **cpu1** continually checks the message buffer for new messages, and if it finds one, prints it using the **jtag_uart**.

Hardware and Software Requirements

To use this design example you must have the following:

- Quartus®II Software version 7.1 or higher – Both Quartus II Web Edition and the fully licensed version work with the example design.
- Nios II Development Kit version 7.1 or higher – Each of the five available kits includes a Nios development board and an Altera USB Blaster download cable. You can use any of the following Nios II Development Kits:
 - Stratix II Edition
 - Stratix Edition
 - Stratix Professional Edition
 - Cyclone II Edition
 - Cyclone Edition

If you do not have a development board, you can follow the hardware development steps, but you will not be able to download the complete system to a working board.



You can download the Quartus II Web Edition software and the Nios II EDS for free from the Altera Download Center at www.altera.com/download. Before you begin creating the design, you must install both the Quartus II software and the Nios II EDS.

Creating the Hardware System

In the following steps you create a multiprocessor system by starting with the **standard** hardware example design included in the Nios II EDS, and adding two additional processors, two additional timers, and a hardware mutex component. You can use the **standard** hardware example design for any of the Nios development boards, and the resulting system runs on that development board. If you do not have a Nios development board, you can still follow these steps to learn how to design multiprocessor hardware.

Getting Started with a Standard Example Design

To begin building a multiprocessor system sharing resources, perform the following steps:

2. Using an external file management tool (such as Windows Explorer), browse to the examples directory for your board. Each board-specific project file resides in the `<Nios II EDS install path>/examples/<hdl>/<development board>/standard` directory.
3. Copy the **standard** example design project directory for the board you are using to a working directory of your choice. Make sure the pathname has no spaces.
4. Open the Quartus II software.
5. On the File menu, click **Open Project** (not Open).
6. Browse and load the Quartus II Project File (.qpf) from the newly-created directory.
7. On the Tools menu, click **SOPC Builder**.



In this tutorial, you must name the hardware components exactly according to the instructions. If your component names differ from the names printed here, the software example will not work.

8. Right-click **cpu** and click **Rename**.
9. Type `cpu1` to rename the processor then press Enter.
10. Right-click **sys_clk_timer** and click **Rename**.
11. Type `cpu1_timer` and press Enter. This is the timer for **cpu1**.
12. If **cpu1_timer** is not immediately under **cpu1**, click **Move Up** several times to move **cpu1_timer** under **cpu1**.

Adding a Second Processor

In the next series of steps, you add a second Nios II processor to the system. You use a Nios II/s processor because it is a good general-purpose choice.

To add a second processor, perform the following steps:

1. In the list of available components (on the left-hand side of the **System Contents** tab), select **Nios II Processor**.
2. Click **Add**. The Nios II Processor MegaWizard interface appears, displaying the **Nios II Core** page.
3. Specify the following settings:
 - **Nios II Core:** Nios II/s
 - **Hardware Multiply:** None
 - **Hardware Divide:** Off
 - **Reset Vector: Memory:** ext_flash **Offset:** 0x100000
 - **Exception Vector: Memory:** sdram **Offset:** 0x100020



Recall from “[Program Memory](#)” on page 1–10 that the exception addresses determine how code memory is partitioned between processors. In this tutorial, each of the three processors runs its software from 1 Mbyte of SDRAM, so you set each processor's exception address within SDRAM, each separated by 0x100000 (1 Mbyte).

4. Click **JTAG Debug Module**. The **JTAG Debug Module** page appears.
5. Select **Level 1** as the debugging level for this processor.
6. Click **Finish**. You return to the SOPC Builder **System Contents** tab, and an instance of the Nios II core named **cpu** now appears in the table of available components.



Error messages appear in the SOPC Builder messages window. This is because SOPC Builder does not know that you plan to connect this processor with other components in the system. Ignore the error messages for now. You will fix these errors in later steps.

7. Right-click the newly-added processor and click **Rename**.
8. Type **cpu2** and press Enter.
9. Click **Move Up** several times to move **cpu2** under **cpu1_timer**.

Adding a Third Processor

In the next series of steps, you add a third Nios II processor to the system. Use a Nios II/e processor to demonstrate that you can use any combination of Nios II processors in a multiprocessor system.

To add the third processor, perform the following steps:

1. In the list of available components, select **Nios II Processor**.
2. Click **Add**. The Nios II Processor MegaWizard interface appears, displaying the **Nios II Core** page.
3. Specify the following settings:
 - **Nios II Core:** Nios II/e
 - **Hardware Multiply:** None
 - **Hardware Divide:** Off
 - **Reset Vector: Memory:** ext_flash **Offset:** 0x200000
 - **Exception Vector: Memory:** sdram **Offset:** 0x200020
4. Click **JTAG Debug Module**. The **JTAG Debug Module** page appears.
5. Select **Level 1** as the debugging level for this processor.
6. Click **Finish**. You return to the SOPC Builder **System Contents** tab, and an instance of the Nios II core named **cpu** now appears in the table of available components.
7. Right-click the newly-added processor and click **Rename**.
8. Type **cpu3** and press Enter.
9. Click **Move Up** several times to move **cpu3** under **cpu2**.

Adding a Timer for cpu2

As mentioned earlier, it is typically not recommended for multiple processors to share non-memory peripherals, so in this section you add separate timer peripherals for each processor in this system.

To add a timer for **cpu2**, perform the following steps:

1. In the list of available components, expand **Peripherals**, expand **Microcontroller Peripherals**, and then click **Interval Timer**.
2. Click **Add**. The Interval Timer MegaWizard interface appears.
3. In the **Presets** list, select **Full-featured**.

4. Click **Finish**. You return to the SOPC Builder **System Contents** tab, and an instance of the interval timer named **timer** now appears in the table of available components.
5. Right-click **timer** and click **Rename**.
6. Type `cpu2_timer` and press Enter. This is the timer for **cpu2**.
7. Click **Move Up** to move **cpu2_timer** under **cpu2**.
8. Using the connection matrix, connect **cpu2_timer** to the data master for **cpu2** only. Disconnect **cpu2_timer** from all other masters.



If you do not see the connection matrix when you move the mouse over the SOPC Builder connections, click **Show Connections Column** on the View menu.

9. Type 0 in the IRQ column for the **cpu2/cpu2_timer connection**. This value allows **cpu2_timer** to interrupt **cpu2** with a priority setting of 0, which is the highest priority.

Adding a Timer for cpu3

To add a timer for **cpu3**, perform the following steps:

1. In the list of available components, expand **Peripherals**, expand **Microcontroller Peripherals**, and then click **Interval Timer**.
2. Click **Add**. The Interval Timer MegaWizard interface appears.
3. In the **Presets** list, select **Full-featured**.
4. Click **Finish**. You return to the SOPC Builder **System Contents** tab, and an instance of the interval timer named **timer** now appears in the table of available components.
5. Right-click **timer** and click **Rename**.
6. Type `cpu3_timer` and press Enter. This is the timer for **cpu3**.
7. Click **Move Up** to move **cpu3_timer** under **cpu3**.
8. Using the connection matrix, connect **cpu3_timer** to the data master for **cpu3** only. Disconnect **cpu3_timer** from all other masters.

9. Type 0 in the **IRQ** column for the **cpu3/cpu3_timer connection**. This allows **cpu3_timer** to interrupt **cpu3** with a priority setting of 0, which is the highest priority.

Adding a Hardware Mutex

You are building a multiprocessor system that shares a data memory between processors, so it is essential that you include a hardware mutex component to protect that memory from data corruption.

To add the hardware mutex, perform the following steps:

1. In the list of available components, expand **Peripherals**, expand **Multiprocessor Coordination**, and then click **Mutex**.
2. Click **Add**. The Mutex MegaWizard interface appears.
3. Click **Finish** to accept the defaults. You return to the SOPC Builder **System Contents** tab, and an instance of the mutex named **mutex** now appears in the table of available components.
4. Right-click **mutex** and click **Rename**.
5. Type `message_buffer_mutex` and press Enter.

Adding a Message Buffer Memory

In this section, you add an on-chip memory to the system that is used as a message buffer to pass messages between processors. This memory is shared by all processors in the system. The processors use the mutex core added in the previous steps to protect the memory's contents from corruption.

To add a message buffer memory perform the following steps:

1. In the list of available components, expand **Memories and Memory Controllers**, expand **On-Chip**, and then click **On-Chip Memory (RAM or ROM)**.
2. Click **Add**. The On-Chip Memory (RAM or ROM) MegaWizard interface appears.
3. In the **Total memory size** box, type 1 and select **KBytes** to specify a memory size of 1 KByte.

4. Click **Finish**. You return to the SOPC Builder **System Contents** tab, and an instance of the on-chip memory named **onchip_mem** now appears in the table of available components.
5. Right-click **onchip_mem** and click **Rename**.
6. Type `message_buffer_ram` and press Enter. This memory is used as a message buffer for the three processors in your multiprocessor system.

Connecting Shared Resources

Now you need to properly connect all the resources that are shared between processors in the system using SOPC Builder's connection matrix and IRQ connection matrix.

To properly connect all the resources in the system shared by the multiple processors, perform the following steps:

1. In the connection matrix, ensure that each `cpu_timer` is connected only to the data master for its CPU component.
2. Using the connection matrix, connect **sdram** to the instruction and data masters for each processor, allowing all three processors to access **sdram**. All the connection dots for **sdram** should be solid black.
3. Using the connection matrix, connect **ext_ram_bus** to the instruction and data masters for each processor, allowing all three processors to access external RAM and flash memory. All the connection dots for **ext_ram_bus** should be solid black.
4. Using the connection matrix, connect **message_buffer_mutex** to the data masters for all three processors and disconnect all three instruction masters, allowing all three processors to access **message_buffer_mutex**.
5. Using the connection matrix, connect **message_buffer_ram** to the data masters for all three processors and disconnect all three instruction masters, allowing all three processors to access that memory only as data memory. No software instructions run from **message_buffer_ram**.
6. Using the connection matrix, disconnect **high_res_timer** from all instruction and data masters except for the **cpu1** data master. The **high_res_timer** should only be connected to the data master for **cpu1**.

7. Using the connection matrix, disconnect **uart1** from all instruction and data masters except for the **cpu1** data master. The **uart1** should only be connected to the data master for **cpu1**.
8. Using the connection matrix, disconnect **led_pio** from all instruction and data masters except for the **cpu1** data master. The **led_pio** should only be connected to the data master for **cpu1**.



In practice, none of the I/O components should be connected to multiple CPUs. This tutorial runs successfully without further component disconnecting because the tutorial code does not attempt to access the other I/O components. Additionally, unused connections consume Avalon resources and FPGA logic elements, possibly affecting system f_{MAX} .

9. Using the IRQ connection matrix (on the right-hand side of the **System Contents** tab), erase the default IRQ numbers to disconnect all lines involving **cpu2**, **cpu3**, **cpu2_timer** and **cpu3_timer** from all resources, except for the two **cpu2/cpu2_timer** and **cpu3/cpu3_timer** IRQs you set in step 9 in “Adding a Timer for cpu2” on page 1–20 and in step 9 in “Adding a Timer for cpu3” on page 1–21.
10. On the System menu, click **Auto-Assign Base Addresses** to give every peripheral a unique base address.

Figure 1–8 shows a system in SOPC Builder after these changes. It shows the new components that implement the message buffer and the required connectivity for the system. Because this tutorial runs on several different development boards, the complete component list might not match yours.

Figure 1–8. Shared Resource Connections

Use	Connections	Module Name	Description	Clock	Base	End	IRQ	
<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/> connector_pll	PLL	sys_clk	0x03222460	0x0322247f		
<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/> cpu1	Nios II Processor		connector_pll_c0			
		instruction_master	Avalon Master					
		data_master	Avalon Master					
		jtag_debug_module	Avalon Slave		0x03221000	0x032217ff	IRQ 0	
<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/> cpu1_timer	Interval Timer		connector_pll_c0			
		<input checked="" type="checkbox"/> cpu2	Nios II Processor		connector_pll_c0			
		instruction_master	Avalon Master					
		data_master	Avalon Master					
		jtag_debug_module	Avalon Slave		0x00000800	0x00000fff	IRQ 0	
<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/> cpu2_timer	Interval Timer		connector_pll_c0			
		<input checked="" type="checkbox"/> cpu3	Nios II Processor		connector_pll_c0			
		instruction_master	Avalon Master					
		data_master	Avalon Master					
		jtag_debug_module	Avalon Slave		0x00001800	0x00001fff	IRQ 0	
<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/> cpu3_timer	Interval Timer		connector_pll_c0			
		<input checked="" type="checkbox"/> sysid	System ID Peripheral		connector_pll_c0			
		reconfig_request_pio	PIO (Parallel I/O)		connector_pll_c0	0x032224a0	0x032224af	IRQ 31
<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/> jtag_uart	JTAG UART		connector_pll_c0			
		<input checked="" type="checkbox"/> ext_ram_bus	Avalon-MM Tristate Bridge					
		avalon_slave	Avalon Slave			0x032224f8	0x032224ff	IRQ 31
		tristate_master	Avalon Tristate Master					
<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/> high_res_timer	Interval Timer		connector_pll_c0			
		<input checked="" type="checkbox"/> ext_flash	Flash Memory (CFI)		connector_pll_c0			
		ext_ram	IDT71V416 SRAM		connector_pll_c0	0x02800000	0x02ffff	IRQ 31
<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/> lan91c111	LAN91C111 Interface		connector_pll_c0			
		<input checked="" type="checkbox"/> lcd_display	Character LCD		connector_pll_c0			
		uart1	UART (RS-232 Serial Port)		connector_pll_c0	0x03221000	0x0321ffff	IRQ 31
<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/> button_pio	PIO (Parallel I/O)		connector_pll_c0			
		<input checked="" type="checkbox"/> led_pio	PIO (Parallel I/O)		connector_pll_c0			
		seven_seg_pio	PIO (Parallel I/O)		connector_pll_c0	0x032224b0	0x032224bf	IRQ 31
<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/> sdram	SDRAM Controller		connector_pll_c0			
		<input checked="" type="checkbox"/> sdram_pll	PLL		PLD_CLKFB	0x032224c0	0x032224cf	IRQ 31
		<input checked="" type="checkbox"/> epcs_controller	EPCS Serial Flash Controller		connector_pll_c0	0x032224d0	0x032224df	IRQ 31
<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/> message_buffer_mux	Mutex		connector_pll_c0			
		<input checked="" type="checkbox"/> message_buffer_ram	On-Chip Memory (RAM or ROM)		connector_pll_c0	0x032224e0	0x032224ef	IRQ 31
					connector_pll_c0	0x03222500	0x03222507	IRQ 31
				connector_pll_c0	0x03222000	0x032223ff	IRQ 31	

Generating and Compiling the System

In this section, you generate HDL for the system you just constructed in SOPC Builder, and then compile the project in the Quartus II software to produce a programming file. To generate and compile the system, perform the following steps:

1. Click the **System Generation** tab.
2. Turn off **Simulation. Create project simulator files**. System generation executes much faster when simulation is off.
3. Click **Generate**. This might take a few moments. A **Stop** button replaces the **Generate** button, indicating generation is taking place.
4. When generation is complete, the **Generate** button replaces the **Stop** button, and a **SUCCESS: SYSTEM GENERATION COMPLETED** message displays. Click **Exit** in SOPC Builder to return to the Quartus II software.

5. On the Quartus II Processing menu, click **Start Compilation** to compile the project in the Quartus II software.
6. When compilation completes and displays the **Full compilation was successful** message box, click **OK**.
7. Click **Programmer** on the Tools menu.
8. Turn on the **Program/Configure** checkbox for the SRAM Object File (.sof) in the Quartus II Programmer.
9. Click **Start** to download the FPGA configuration data to your target hardware.

Creating Software for the Multiprocessor System

In the following steps you create one application project and one system library project for each processor in the system using the Nios II IDE, a total of six separate software projects for the multiprocessor system. You then build, run and debug the software projects.

The software you run on this system uses the hardware mutex to share a message buffer. All three processors write messages to the message buffer. `cpu1` then reads the messages and prints them to the `jtag_uart`. The same executable file runs on each processor, but the processors are doing slightly different things. In this particular application, the software running on each CPU decides what to do based on whether or not the CPU is connected to the `jtag_uart` component. In Nios II processor systems, a processor locks the mutex by writing the value of its `cpuid` control register to the `OWNER` field of the `mutex` register. The `cpuid` register holds a static value that uniquely identifies the processor in a multi-processor system. The software checks the processor's `cpuid` before executing any functions that are specific to a particular processor. If the `cpuid` is correct, it executes the function.

Starting the Nios II IDE

In this section, you start the Nios II IDE and begin creating software projects for the three processors in the system. To start the Nios II IDE from SOPC Builder, perform the following steps:

1. On the Tools menu in the Quartus II software, click **SOPC Builder**.
2. In SOPC Builder, click the **System Generation** tab.
3. Click **Nios II IDE**. The Nios II IDE starts.



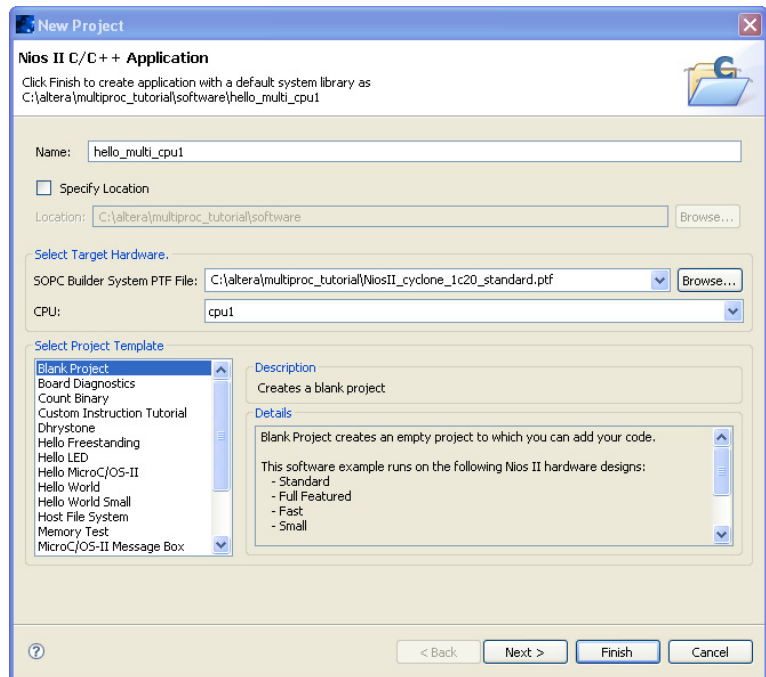
If the **Workspace Launcher** dialog box appears, click **OK** to accept the default workspace. If the Nios II IDE welcome screen appears, click **Workbench** to continue.

Creating a Software Project for cpu1

To create a software project for **cpu1**, perform the following steps:

1. On the File menu, point to **New**, and then click **Nios II C/C++ Application**. The **New Project** wizard for Nios II C/C++ application projects appears, pre-selecting the newly-created **SOPC Builder System PTF File** for you.
2. In the **Name** field, type `hello_multi_cpu1`.
3. Under **Select Target Hardware**, select **cpu1** as the **CPU**.
4. In **Select Project Template** list, select **Blank Project** as shown in [Figure 1–9](#).

Figure 1–9. New Project for cpu1




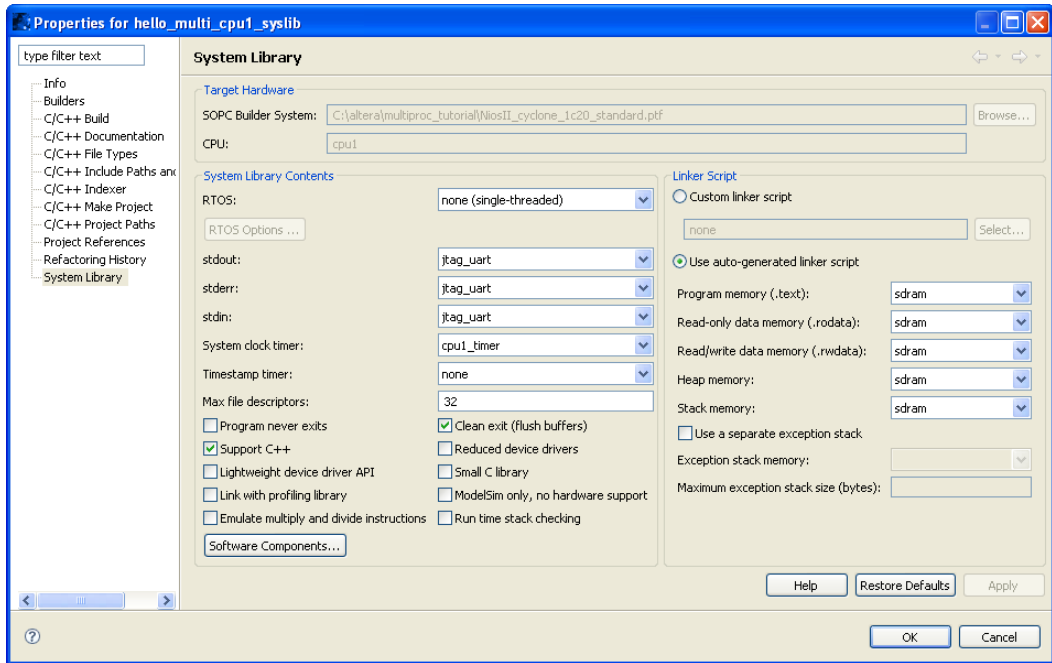
5. Click **Finish**. The Nios II IDE generates a new Nios II C/C++ application project, and a corresponding system library project for **cpu1**.
6. Download the file **hello_world_multi.c** to a known location on your host PC.
 You can find this file with this tutorial on the Nios II Processor Literature page at www.altera.com/literature/lit-nio2.jsp.
7. Using an external file management tool (such as Windows Explorer), drag **hello_world_multi.c** from its known location into the **Nios II C/C++ Projects** view of the Nios II IDE, and drop it onto the **hello_multi_cpu1** project folder.
8. Right-click the system library project **hello_multi_cpu1_syslib**.
9. Select **Properties**.
10. In the left-hand pane, select **System Library**.
11. Verify **jtag_uart** is selected for **stdin**, **stderr**, and **stdout**.
12. Verify **cpu1_timer** is selected for **System clock timer**.
13. Verify that **sdram** is selected for **Program Memory**, **Read-only data memory**, **Read/write data memory**, **Heap memory**, and **Stack memory**. See [Figure 1–10](#) for an example of system library property settings.
14. Click **OK**.

Figure 1–10. System Library Property Settings



Creating a Software Project for cpu2

To create a software project for `cpu2`, perform the following steps:

1. On the File menu, point to **New**, and then click **Nios II C/C++ Application**. The **New Project** wizard for Nios II C/C++ application projects appears, pre-selecting the newly-created **SOPC Builder System PTF File** for you.
2. In the **Name** field, type `hello_multi_cpu2`.
3. Under **Select Target Hardware**, select `cpu2` as the **CPU**.
4. In **Select Project Template**, choose **Blank Project**.
5. Click **Finish**. The Nios II IDE generates a new Nios II C/C++ application project, and a corresponding system library project for `cpu2`.

6. In the **Nios II C/C++ Projects** view, expand the **hello_multi_cpu1** project folder. Right-click **hello_world_multi.c** and click **Copy**.
7. Right-click the **hello_multi_cpu2** project folder and click **Paste**. A copy of **hello_world_multi.c** appears under the **hello_multi_cpu2** project.
8. Right-click the system library project **hello_multi_cpu2_syslib**.
9. Select **Properties**.
10. Select **System Library** in the left-hand pane.
11. Verify **null** is selected for **stdin**, **stderr**, and **stdout**. Only **cpu1** is connected to the **jtag_uart**.
12. Select **cpu2_timer** as **System clock timer**.
13. Verify that **sdram** is selected for **Program Memory**, **Read-only data memory**, **Read/write data memory**, **Heap memory**, and **Stack memory**.
14. Click **OK**.

Creating a Software Project for cpu3

To create a software project for **cpu3**, perform the following steps:

1. On the **File** menu, point to **New**, and then click **Nios II C/C++ Application**. The **New Project** wizard for Nios II C/C++ application projects appears, pre-selecting the newly-created **SOPC Builder System PTF File** for you.
2. In the **Name** field, type **hello_multi_cpu3**.
3. Under **Select Target Hardware**, select **cpu3** as the **CPU**.
4. In **Select Project Template**, choose **Blank Project**.
5. Click **Finish**. The Nios II IDE generates a new Nios II C/C++ application project, and a corresponding system library project for **cpu3**.
6. In the **Nios II C/C++ Projects** view, expand the **hello_multi_cpu1** project folder. Right-click **hello_world_multi.c** and click **Copy**.

7. Right-click the **hello_multi_cpu3** project folder and click **Paste**. A copy of **hello_world_multi.c** appears under the **hello_multi_cpu3** project.
8. Right-click system library project **hello_multi_cpu3_syslib**.
9. Select **Properties**.
10. Select **System Library** in the left-hand pane.
11. Verify **null** is selected for **stdin**, **stderr**, and **stdout**. Only **cpu1** is connected to the **jtag_uart**.
12. Select **cpu3_timer** as **System clock timer**.
13. Verify that **sdram** is selected for **Program memory**, **Read-only data memory**, **Read/write data memory**, **Heap memory**, and **Stack memory**.
14. Click **OK**.

Building the Software Projects

In this section, you build the three software projects you just created so they can be run on the processors in the system.

To build the three software projects, perform the following steps:

1. In the **Nios II C/C++ Projects** view, right-click the project **hello_multi_cpu1** and click **Build Project**.
2. Right-click the project **hello_multi_cpu2** and click **Build Project**.
3. Right-click the project **hello_multi_cpu3** and click **Build Project**.

If you encounter any errors in the builds, you must correct them and rebuild before continuing.

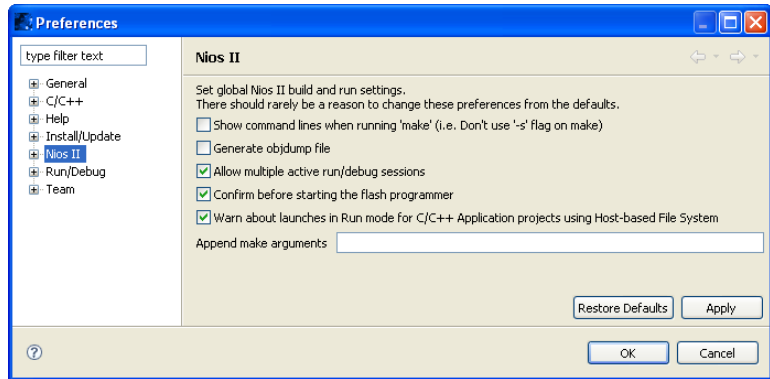
Setting up the Nios II IDE for Multiprocessor Debug

By default, the Nios II IDE is set to not allow multiple active debug sessions. To enable multiple debug sessions, perform the following steps:

1. In the Nios II IDE, on the Window menu, click **Preferences**.
2. Select **Nios II** and turn on **Allow multiple active run/debug sessions** as shown in [Figure 1–11](#).

3. Click **OK**.

Figure 1–11. Multiple Active Run/Debug Sessions



Creating a Run/Debug Configuration for Each Processor

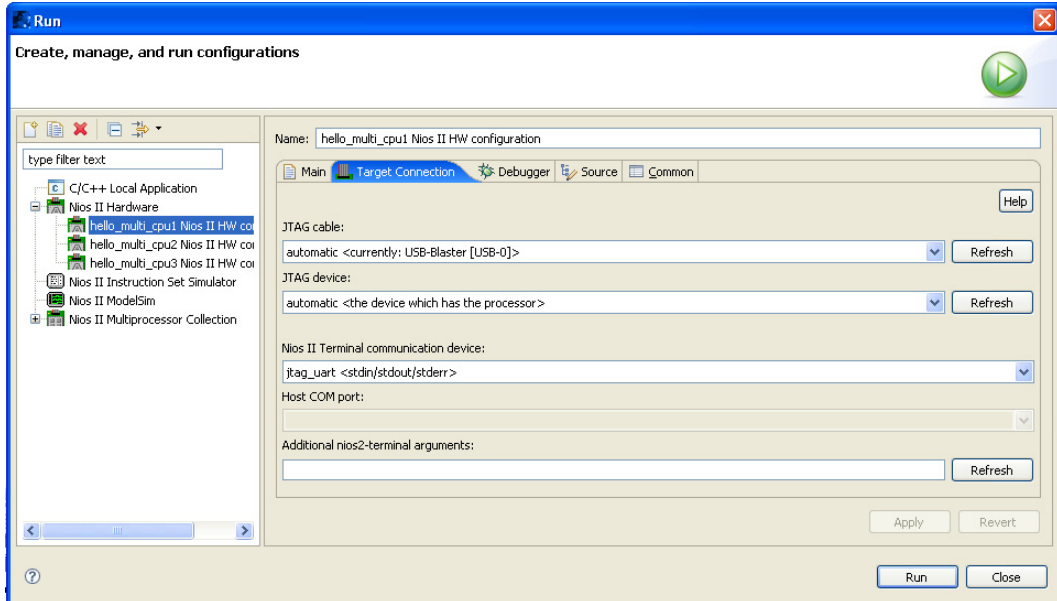
In this section, you create a run/debug configuration for each of the target processors. These configurations enable you to run and debug the three software projects you just built on the processors in the system.

To create run/debug configurations for each processor, perform the following steps:


1. In the **Nios II C/C++ Projects** view, click the **hello_multi_cpu1** project.
2. On the Run menu, click **Run**.
3. Right-click **Nios II Hardware** in the **configurations** list.
4. Click **New**. A new run/debug configuration is created for the project.
5. On the **Main** tab, click **Load JDI File** and browse to the system's JDI file located in the Quartus II project directory.
6. Click the **Target Connection** tab.
7. Ensure the download cable you are using is selected in the **JTAG cable** field as shown in [Figure 1–12](#).

If the field reads **Automatic**<currently (your correct download cable)>, you do not need to change it.

Figure 1–12. Run/Debug Configuration



8. Click **Close**.
9. Repeat steps 1–8 to create a run/debug configuration for each of the target processors.

 Be sure you have selected the appropriate project when you create the run/debug configuration.

You have created a run/debug configuration for each processor in the system. You can now download, execute, and debug code on each of the processors individually, using the normal flow for running or debugging.

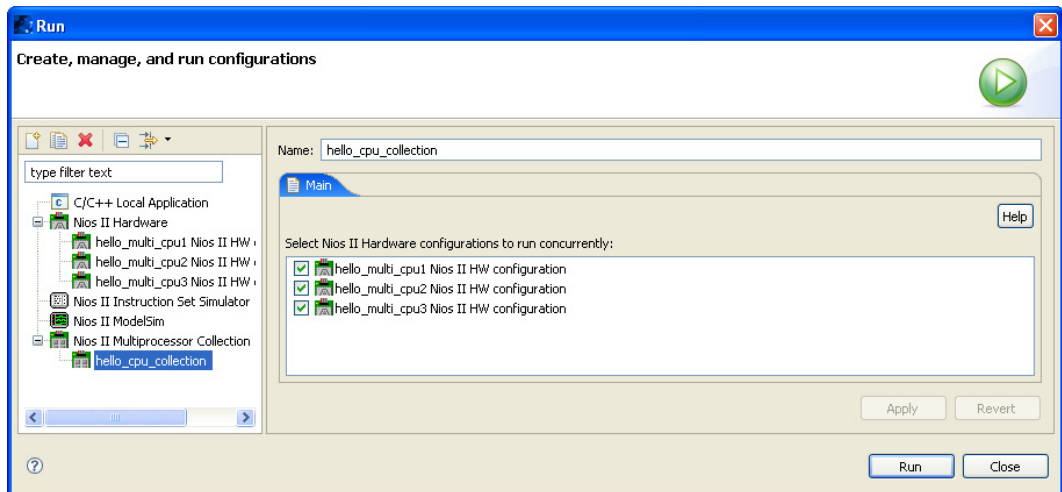
Creating a Multiprocessor Collection

In this section, you create a multiprocessor collection which enables the launching and stopping of multiple processors as a single unit.

To create this multiprocessor collection, perform the following steps:

1. On the Run menu, click **Run**.
2. In the **configurations** list, right-click **Nios II Multiprocessor Collection**.
3. Click **New**.
4. In the **Name** field, type `hello_cpu_collection` as the name for this new multiprocessor collection.
5. Turn on **hello_multi_cpu1 Nios II HW configuration**, **hello_multi_cpu2 Nios II HW configuration**, and **hello_multi_cpu3 Nios II HW configuration** as shown in [Figure 1–13](#).
6. Click **Apply**.

Figure 1–13. Multiprocessor Collection Example



Starting the Multiprocessor Collection

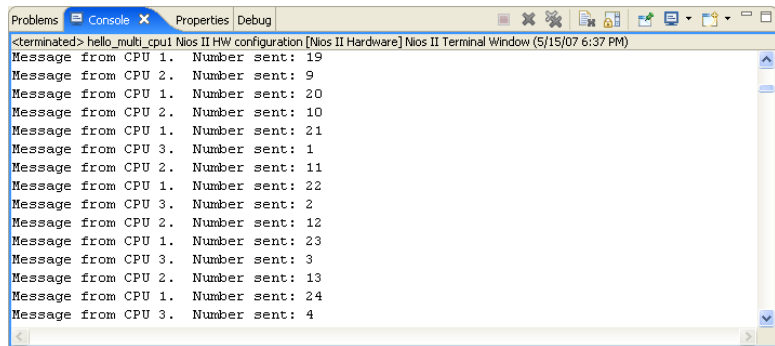
Now you can start all the processors with a single mouse click. To start all the processors, perform the following steps:

1. Select the **hello_cpu_collection** configuration, and click **Run**. The Nios II IDE downloads the software to each processor, and then runs the software.

Each processor begins executing code as soon as its code is downloaded; the processors do not start in unison.

2. After the launch finishes, you should see messages from all three processors displaying in the Console view as shown in [Figure 1-14](#).

Figure 1-14. Multiprocessor Collection Messages



3. When you are done observing the Console output, click **Terminate** (the square red button on the Console view toolbar) to close the terminal connection.
4. If no Console output appeared, erase the flash memory and try again:
 - a. In SOPC Builder, on the System Contents tab, verify the base address of flash memory.
 - b. Open the Nios II IDE Command Shell.
 - c. In the Nios II IDE Command Shell, type

```
nios2-flash-programmer --base=<flash base address> --erase-all --instance "0"
```

- d. Repeat steps 1-3.

Debugging the Software Projects on the Board

In this section, you start all the processors using the multiprocessor collection, and set breakpoints on individual processors. To start the processors and set individual breakpoints, perform the following steps:

1. On the Run menu, click **Debug**.
2. In the **configurations** list, select the new collection you created under **Nios II Multiprocessor Collection** in the previous section.
3. Click **Debug**.



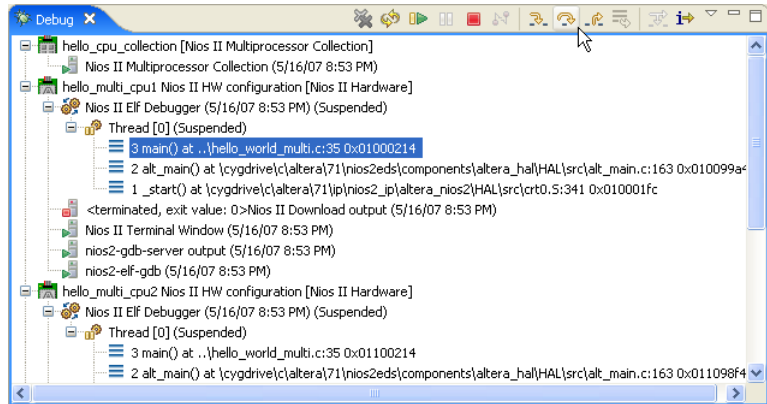
If a dialog box appears and asks you to switch to the Debug perspective, click **Yes**.

Again, the Nios II IDE downloads and launches each software project on its respective processor, then pauses each one at a breakpoint set on `main()`.

In the **Debug** view, you see the processor collection listed at the top with each individual debug session listed below it, including the call stack.

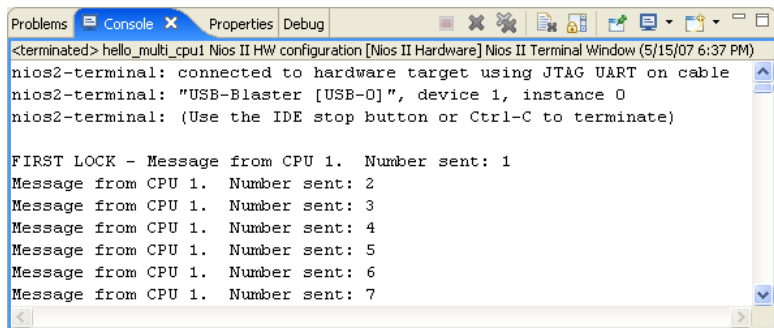
4. Click the **main()** call stack entry under the **cpu1** debug session.
5. Click **Step Over** in the toolbar menu to see **cpu1** step through the software code.

Figure 1–15 shows the Debug view with the **main()** call stack entry highlighted and the mouse pointer pointing to the **Step Over** icon in the toolbar menu.

Figure 1–15. Debug View

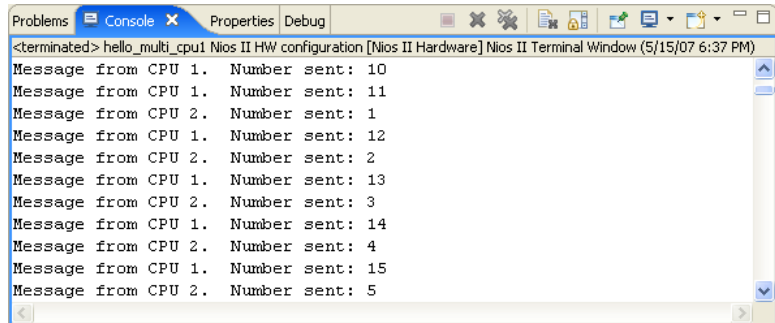
- Click the **Resume** icon in the toolbar menu to let **cpu1** run freely.

You see that only messages from **cpu1** appear on the terminal as shown in [Figure 1–16](#).

Figure 1–16. cpu1 Debug Messages

- Click the **main()** call stack entry under the **cpu2** debug session.
- Click the **Resume** icon in the toolbar menu to let **cpu2** run uninterrupted.

You now see messages from both **cpu1** and **cpu2** appear on the terminal as shown in [Figure 1–17](#).

Figure 1–17. *cpu1 and cpu2 Debug Messages*A screenshot of a Nios II Terminal Window. The window title is "hello_multi_cpu1 Nios II HW configuration [Nios II Hardware] Nios II Terminal Window (5/15/07 6:37 PM)". The terminal output shows a sequence of messages: "<terminated> hello_multi_cpu1 Nios II HW configuration [Nios II Hardware] Nios II Terminal Window (5/15/07 6:37 PM)", followed by "Message from CPU 1. Number sent: 10", "Message from CPU 1. Number sent: 11", "Message from CPU 2. Number sent: 1", "Message from CPU 1. Number sent: 12", "Message from CPU 2. Number sent: 2", "Message from CPU 1. Number sent: 13", "Message from CPU 2. Number sent: 3", "Message from CPU 1. Number sent: 14", "Message from CPU 2. Number sent: 4", "Message from CPU 1. Number sent: 15", and "Message from CPU 2. Number sent: 5". The window has tabs for "Problems", "Console", "Properties", and "Debug".

```
<terminated> hello_multi_cpu1 Nios II HW configuration [Nios II Hardware] Nios II Terminal Window (5/15/07 6:37 PM)
Message from CPU 1. Number sent: 10
Message from CPU 1. Number sent: 11
Message from CPU 2. Number sent: 1
Message from CPU 1. Number sent: 12
Message from CPU 2. Number sent: 2
Message from CPU 1. Number sent: 13
Message from CPU 2. Number sent: 3
Message from CPU 1. Number sent: 14
Message from CPU 2. Number sent: 4
Message from CPU 1. Number sent: 15
Message from CPU 2. Number sent: 5
```

9. Repeat steps 7 and 8 for **cpu3** to see messages from all three CPUs.
10. Click **Terminate** (the square red button) to stop the debug sessions for all three processors.

You're done! You've now constructed, built software projects for, and debugged software on your first Nios II multiprocessor system. You have also learned how to use the `Mutex` component to share system resources between processors. Feel free to experiment with the system you've created and find interesting new ways of using multiple processors in an Altera FPGA.

Altera recommends saving this system to use as a starting point next time you wish to create a multiprocessor system.