



# **Creating Multiprocessor Nios II Systems**

---

## **Tutorial**



101 Innovation Drive  
San Jose, CA 95134  
[www.altera.com](http://www.altera.com)

TU-N2033005-2.0

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

11.0  
June 2011



[Subscribe](#)

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



## Chapter 1. Creating Multiprocessor Nios II Systems

Introduction to Nios II Multiprocessor Systems .....	1-1
Benefits of Hierarchical Multiprocessor Systems .....	1-2
Nios II Multiprocessor Systems .....	1-2
Multiprocessor Tutorial Prerequisites .....	1-3
Hardware Designs for Peripheral Sharing .....	1-3
Autonomous Multiprocessors .....	1-3
Multiprocessors that Share Peripherals .....	1-4
Sharing Peripherals in a Multiprocessor System .....	1-4
Sharing Memory .....	1-6
The Hardware Mutex Core .....	1-7
Sharing Peripherals .....	1-8
Overlapping Address Space .....	1-8
Software Design Considerations for Multiple Processors .....	1-9
Program Memory .....	1-9
Boot Addresses .....	1-13
Debugging Nios II Multiprocessor Designs .....	1-15
Design Example: The Dining Philosophers' Problem .....	1-15
Hardware and Software Requirements .....	1-16
Installation Notes .....	1-17
Creating the Hardware System .....	1-17
Getting Started with the multiprocessor_tutorial_start Design Example .....	1-17
Viewing a Philosopher System .....	1-18
Philosopher System Pipeline Bridges .....	1-19
Adding Philosopher Subsystems .....	1-21
Connecting the Philosopher Subsystems .....	1-22
Viewing the Complete System .....	1-27
Generating and Compiling the System .....	1-28
Creating Software for the Multiprocessor System .....	1-29
Building and Running the Applications from the Command Line .....	1-29
Building and Launching the Applications .....	1-29
Viewing and Controlling Applications from the Command Line .....	1-31
Debugging the Applications in the Nios II SBT for Eclipse .....	1-33
Starting the Nios II SBT for Eclipse .....	1-33
Importing the Software Projects .....	1-34
Building the Software Projects .....	1-34
Launching nios2-terminal for stdio Connections .....	1-34
Creating and Running a Launch Configuration for Each Processor .....	1-35
Debugging the Software Projects on the Board .....	1-36
Conclusion .....	1-38

## Additional Information

Document Revision History .....	Info-1
How to Contact Altera .....	Info-1
Typographic Conventions .....	Info-2



This tutorial demonstrates the features of the Altera® Nios® II processor and Qsys system integration tool that are useful for creating systems with multiple processors. The tutorial provides a design example that guides you through stitching together subsystems in a hierarchical design. Using Qsys, you build a multiprocessor system containing six processors. Each processor is in a subsystem, creating a hierarchy with six subsystems with a shared memory map, coordinated with pipeline bridges. This system demonstrates a solution for the classic Dining Philosophers' Problem.

This tutorial shows you how to use the Nios II Software Build Tools (SBT) to create, build, download, and view `stdio` output in a console for six applications, using shell scripts. It includes steps to import and debug those applications in the Nios II SBT for Eclipse.



Refer to the [Nios II Embedded Design Suite Release Notes and Errata](#) and the [MegaCore IP Library Release Notes and Errata](#) for the latest features, enhancements, and known issues in the current release.

## Introduction to Nios II Multiprocessor Systems

Any system that incorporates multiple microprocessors working together to perform one or more related tasks is commonly referred to as a multiprocessor system. Using the Altera Nios II processor and Qsys tool, you can quickly design and build multiprocessor systems that share peripherals safely. Qsys is a system development tool for creating FPGA designs that can include 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 Qsys tool that are useful for creating systems with multiple processors. This document provides a design example that guides you through stitching together subsystems, each containing a Nios II processor, timer, clock, JTAG UART, and hardware mutex component. You build a multiprocessor system containing six processors that share mutex peripherals in a hierarchical design. You execute scripts that invoke the Nios II SBT to build and download software for each processor, and view each processor's `stdio` output in a console. Then you import six software application and board support package (BSP) projects to the Nios II SBT for Eclipse, and use the Nios II SBT for Eclipse to debug them.

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

- Create hierarchical Qsys systems containing multiple Nios II processors, using pipeline bridges to access peripherals in neighboring subsystems.
- Ensure integrity by safely sharing peripherals between processors, preventing data corruption.

- Build, download, and interact with software for multiprocessor systems using the Nios II SBT with shell scripts.
- Debug multiple software projects running on multiple processors simultaneously using the Nios II SBT for Eclipse.

## Benefits of Hierarchical Multiprocessor Systems

Multiprocessor systems possess the benefit of increased performance, but nearly always at the price of significantly increased system complexity for both hardware and software. The idea of using multiple processors to perform different tasks and functions on different processors in real-time embedded applications is gaining popularity. Altera FPGAs provide an ideal platform for developing embedded multiprocessor systems, because the hardware can easily be modified and tuned using the Qsys tool to provide optimal system performance. Increases in the size of Altera FPGAs make possible system designs with many Nios II processors on a single chip. Furthermore, with a powerful integration tool like Qsys, different system configurations can be designed, built, and evaluated very quickly. Qsys enables hierarchical designs, reducing system complexity through compartmentalization of the design into discrete subsystems. Each subsystem exports user-defined interfaces, linking the subsystem hierarchy together.

## Nios II Multiprocessor Systems

The Nios II SBT for Eclipse includes features to help with the creation and debugging of multiprocessor systems. Multiple Nios II processors are able to efficiently share peripherals thanks to the multimaster-friendly slave-side arbitration capabilities of the Qsys interconnect. Because the capabilities of Qsys allow you to almost effortlessly add as many processors to a system as desired, the design focus in building multiprocessor systems no longer lies in the arranging and connecting of hardware components. The challenge in building multiprocessor systems 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 for Qsys. The hardware mutex core allows different processors to claim ownership of a shared peripheral for a period of time. This temporary ownership of a peripheral by a processor protects the shared peripheral from corruption by the actions of another processor.

To prevent corruption, you must write software that waits to acquire the mutex before it accesses the shared peripheral, ensuring mutually exclusive access.

A nonatomic test-and-set operation has a serious risk: two processors can simultaneously test the flag, each confirming that no processor currently has ownership. If both processors then acquire the peripheral, they violate mutual exclusion.

An atomic test-and-set operation avoids this risk, because it cannot be interrupted. An atomic test-and-set allows a processor to check for ownership and acquire ownership in a single operation.

The fact that the operation cannot be interrupted also ensures that an operating system task switch cannot occur while the processor is testing and acquiring or releasing the mutex.


The hardware mutex core provides a semaphore for mutually exclusive access to any peripheral. The software determines that peripheral and is responsible for uniform use of the mutex API to ensure mutually exclusive access every time the peripheral is accessed.

For more information about mutually exclusive access to shared memory, refer to [“The Hardware Mutex Core” on page 1-7](#).

The Nios II SBT for Eclipse supports software debugging on multiprocessor systems, by allowing you to start and stop multiple software debug sessions on simultaneously running processors.

## Multiprocessor Tutorial Prerequisites

This chapter assumes that you are familiar with the following topics:

- Reading and writing embedded software for the Nios II Processor
  -  Read and follow the step-by-step procedures for building a microprocessor system in the *Nios II Hardware Development Tutorial*, found on the [Literature: Nios II Processor](#) page of the Altera website.
- Multiprocessing, especially the following concepts:
  - Mutual exclusion and mutex usage
  - Concurrency
  - Synchronization
- Hierarchical system design in Qsys

## Hardware Designs for Peripheral Sharing

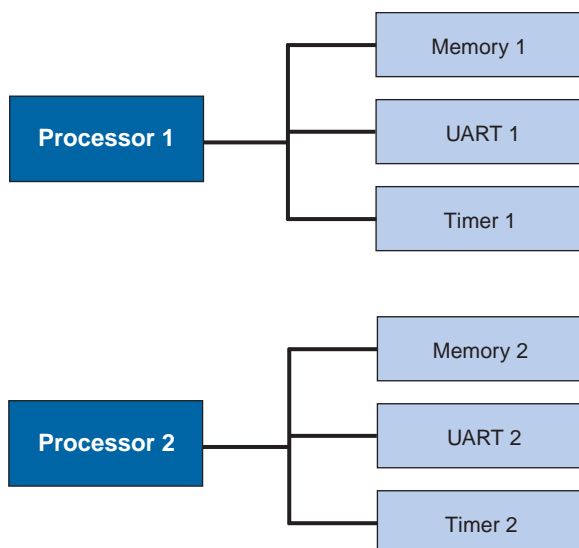
Nios II multiprocessor systems are split into two main categories: those that share peripherals, and those in which each processor is autonomous and does not share peripherals with other processors.

### 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. By design, systems of this type do not share peripherals, and so the processors cannot interfere with each other. Therefore, such systems are typically less complicated and pose fewer challenges.

Figure 1–1 shows a block diagram of two autonomous processors in a multiprocessor system.

**Figure 1–1. Autonomous Multiprocessor System**



## Multiprocessors that Share Peripherals

Multiprocessor systems that share peripherals can pose many challenges. There are features in Qsys that make it possible to reliably implement multiprocessor systems that share peripherals. However, creating such a system is not always straightforward.

Figure 1–2 shows a block diagram of a sample multiprocessor system in which two processors share an on-chip memory.

The next section discusses shared peripherals in detail.

## Sharing Peripherals in a Multiprocessor System

Peripherals are considered shared when they can be accessed by multiple processors. The Qsys connections panel controls which hardware components can be accessed by each individual Nios II processor.

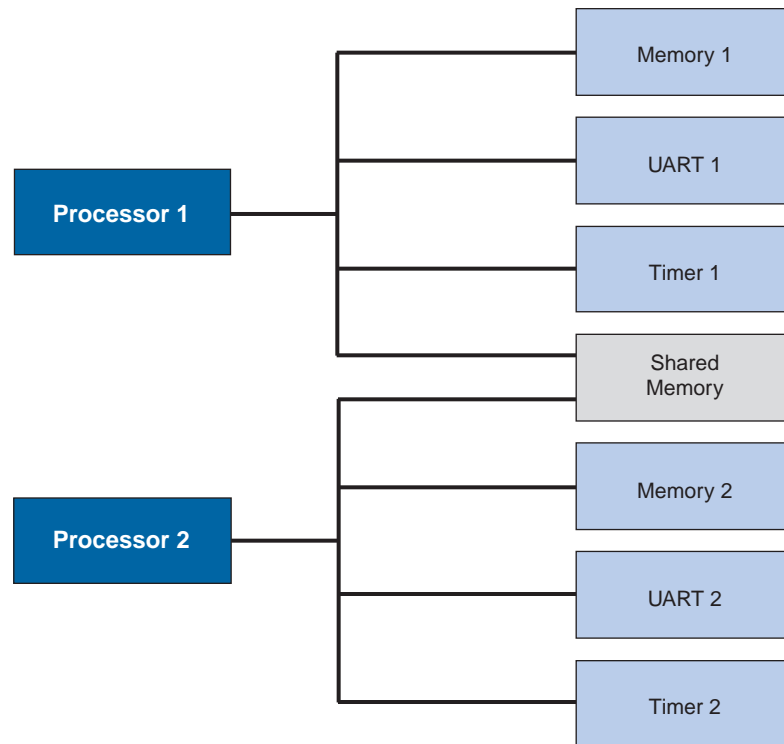
Shared peripherals can be a very powerful feature of multiprocessor systems, but care must be taken when deciding which system peripherals are shared, and how the different processors cooperate regarding the use of peripherals.

In a nonhierarchical system, peripherals can be made shareable by simply connecting them to multiple processor master interfaces in the connection matrix of Qsys. In a hierarchical system, peripherals can also be made shareable to processors outside of the subsystem containing the peripheral by exporting the slave interface of the peripheral. Processor master interfaces gain access to the peripheral through connection in the Qsys connection matrix to the exported interface of the subsystem containing the peripheral. A processor master interface located in a subsystem of the hierarchy can gain access to a peripheral located in a parent system through



connection to an Avalon™ Memory-Mapped (Avalon-MM) pipeline bridge. An Avalon-MM pipeline bridge also provides a mechanism for simultaneous connection of a slave interface to both a processor master local to the subsystem and an external processor master elsewhere in the hierarchy. In that case, the pipeline bridge exports the slave interface, instead of the peripheral exporting the slave interface directly.

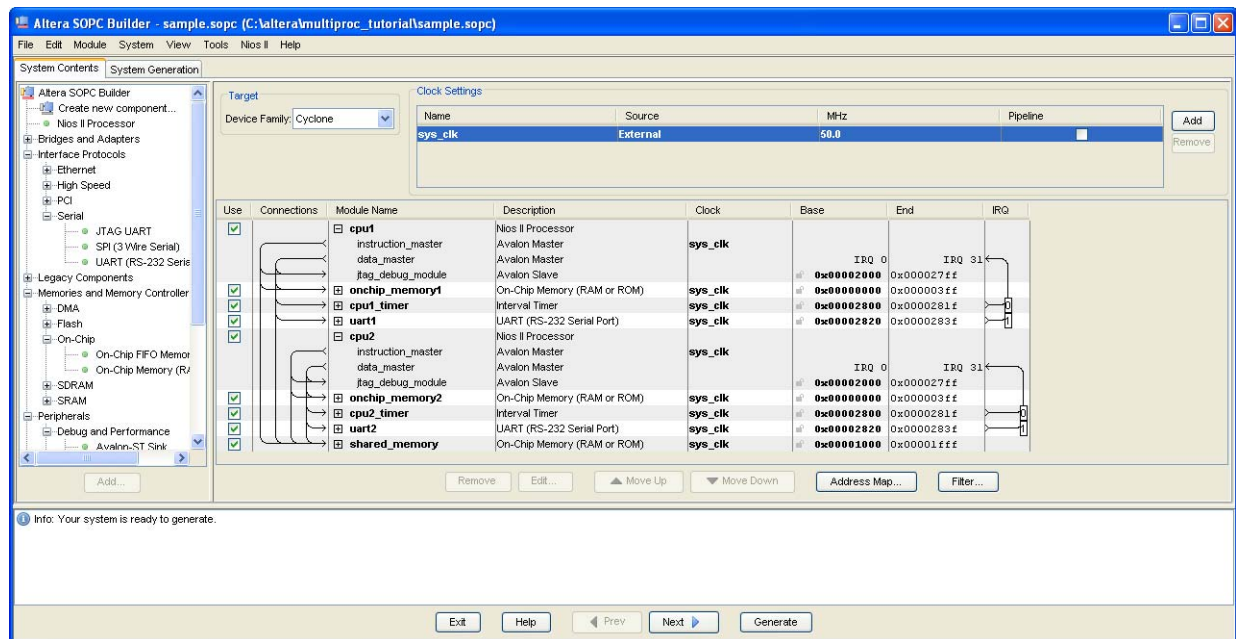
**Figure 1–2. Multiprocessor System with Shared Peripheral**



The software running on each processor is responsible for coordinating mutually exclusive access to shared peripherals with the system's other processors through employment of mutex peripherals.

Figure 1–3 shows a sample multiprocessor system in SOPC Builder, the predecessor to Qsys. The component listed at the bottom, **shared\_memory**, is considered shared because the data and instruction master ports of both processors are connected to the same slave port of the memory. Because **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 peripheral 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 multiple processors, 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 for Multiple Processors](#)” on page 1–9 for information on how to make sure each processor sharing a memory component for program space uses a dedicated area in that memory.

If a memory component is to be shared for data purposes, you must connect its slave port to the data masters of the processors that are sharing the memory. In a nonhierarchical system, make the connection directly in the connection panel. In a hierarchical system, make a logical connection to each subsystem’s exported slave interface.

Sharing data memory among multiple processors can be tricky because data memory can be written 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 is likely to 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 peripheral, so the other processors do not interfere. The following section discusses such a mechanism: the Altera hardware mutex core.

## The Hardware Mutex Core

The Nios II processor provides protection of shared peripherals by accessing the hardware mutex core, which ensures only one processor has ownership of the mutex at any given time. The hardware mutex core is not an internal feature of the Nios II processor. It is a simple Qsys component.

The term mutex stands for mutual exclusion, and a mutex does exactly as its name suggests. A mutex allows cooperating processors to agree that only one processor at a time is allowed access to a particular hardware peripheral. This is useful for the purpose of protecting peripherals from data corruption that can occur if multiple processors attempt to use the peripheral at the same time.

The mutex core acts as a shared peripheral, 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 peripheral associated with the mutex, the processor releases the mutex lock. Thereafter, another processor can acquire the mutex lock and use the shared peripheral. 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 peripheral when they did not.



The mutex core does not physically protect peripherals 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 written to always acquire the mutex before accessing its associated shared peripheral.



For more information about the hardware mutex core, refer to the *Mutex Core* chapter in the *Embedded Peripherals IP User Guide*.

Another kind of mutex, called a software mutex, is common in many operating systems for providing the same protection of peripherals. The difference is that a software mutex is purely a software construct that is used to protect software or hardware peripherals from being corrupted by multiple processes running on the same processor. A hardware mutex core is a Qsys component with an Avalon interface that uses logic to guarantee only one processor is granted the lock of the mutex at any given time. If every processor waits until it acquires the appropriate mutex before using the associated shared peripheral, the peripheral is protected from potential corruption caused by simultaneous access by multiple processors. The hardware mutex core itself has no connection to the shared peripheral; it merely provides a semaphore.

In some cases a mutex core might not be necessary. For example, no mutex is needed with a one-way message buffer, where one processor exclusively writes to the buffer, and all other processors exclusively read. However, sharing peripherals safely without a mutex core can be complicated. When in doubt, use the mutex core.

## Sharing Peripherals

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 responds first and services 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 not provided by the Nios II hardware abstraction layer (HAL) library.



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

Memory peripherals and mutex peripherals can be accessed by multiple processors. Altera recommends that you restrict all other peripherals to be accessible by only one processor in the system. If other processors require use of the peripheral, it is better to use a hardware FIFO, or a message buffer that is mutex-protected, to communicate with the single processor that is connected to that peripheral. That single processor acts as a server for the other processor clients of that peripheral.

When building any system, especially a multiprocessor system, it is advisable to only make connections between processors and peripherals that require direct 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 never corrupts those memories.

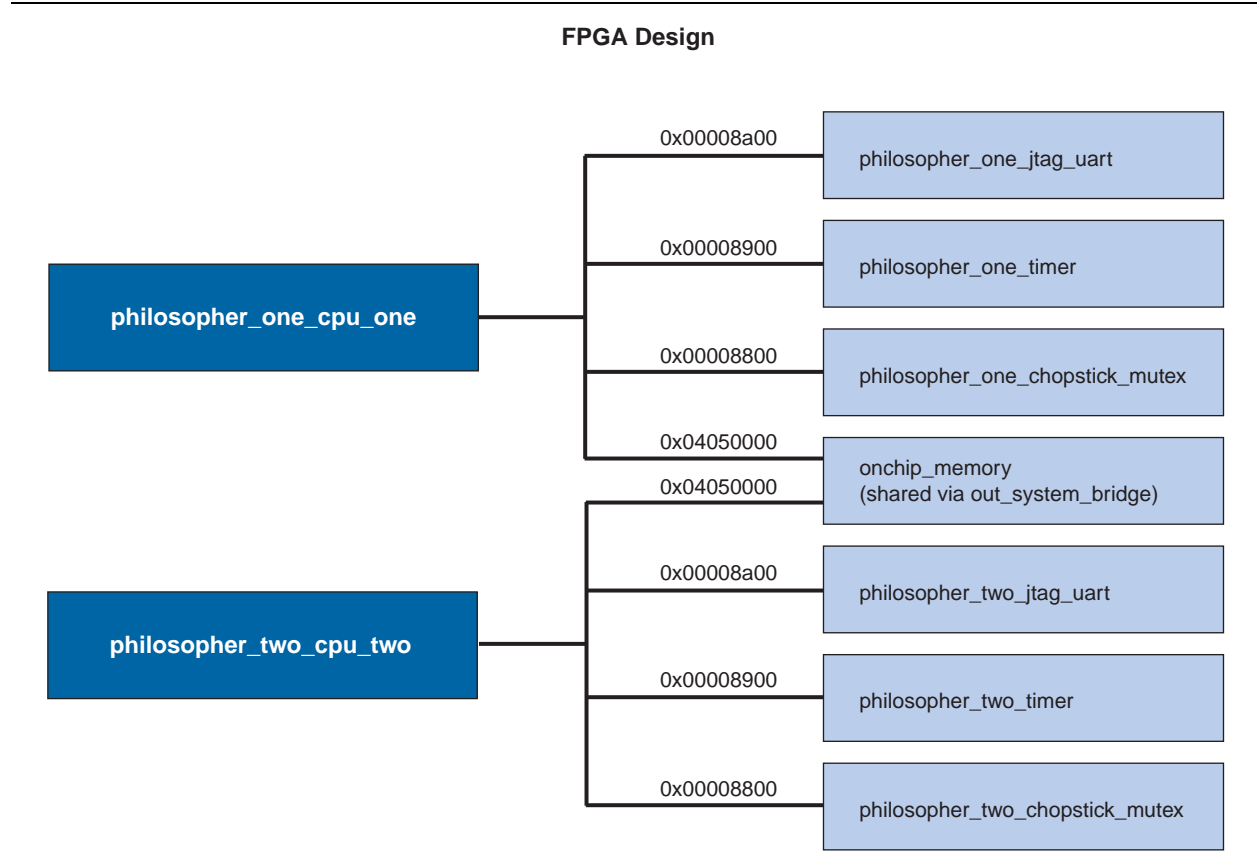
In multiprocessor systems, the need to connect various components is very design-dependent. Therefore, when designing multiprocessor systems, verify explicitly that each component is connected to the desired processor. Most components are best managed by a single processor. If processor A requires the services of a peripheral that is connected to and managed by processor B, processor A must request of processor B that it perform operations with the peripheral on behalf of processor A. You can use shared on-chip memory protected by a mutex for communication between the two processors for this purpose.

## Overlapping Address Space

Single-processor systems typically prohibit more than one slave peripheral from occupying the same address space because this arrangement causes conflicts. However, in multiprocessor systems, separate slave peripherals can occupy the same base address without conflict, as long as each peripheral 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 0x8a00, processor B can connect to a separate slave peripheral, also mapped to address 0x8a00, 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 a sample multiprocessor system with different slave components mapped to the same base address.

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



## Software Design Considerations for Multiple Processors

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 multiple processors from 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 multiprocessor system where all processors execute from on-chip memory. The

software for each processor requires eight kilobytes (KB) of memory for program code and data. The first processor could use the region between 0x0 and 0x1FFF in on-chip memory as its program space, and a second processor could use the region between 0x2000 and 0x3FFF. [Figure 1-6 on page 1-12](#) shows an example of this type of memory sharing.

The Nios II SBT provides a simple scheme of memory partitioning that allows multiple processors to run their software from different regions of the same physical memory. The SBT uses the exception address for each processor to determine the region of memory from which each processor can run its code. The system designer sets the exception address for each processor independently in Qsys.

The Nios II SBT ensures that the processors' software is linked and determines where the software resides in memory. It uses the exception addresses to calculate where each code section is linked. The Nios II SBT positions each processor's code region in the memory component containing the exception address.

If the software for multiple processors is linked to the same physical memory component, then the SBT uses the exception address of each processor to determine the base address of the region. The code region ends at the next exception address for a different processor found in that physical memory component. The processor with the highest exception address is assigned a code region that extends to the end of the physical memory component's address range.

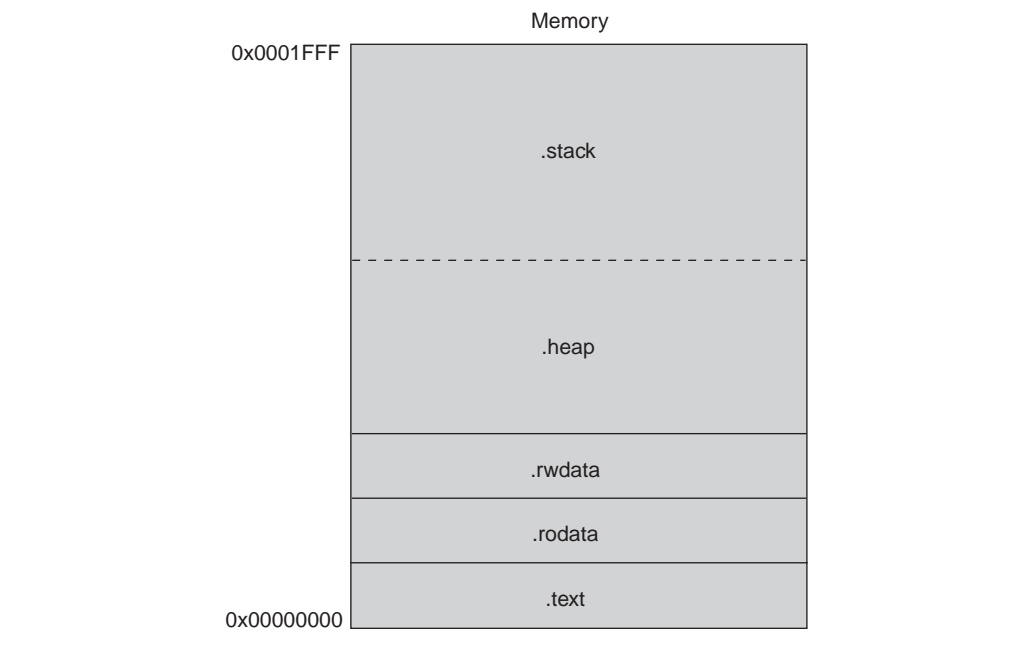
Each processor has five default linker sections. Regardless whether the processor is in a single-processor or a multiprocessor system, the default linker sections are as follows:

- `.text`—the 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

The SBT ensures that these sections are linked and located at fixed addresses in memory.

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

**Figure 1-5. Single Processor Code Linked in Memory Map**

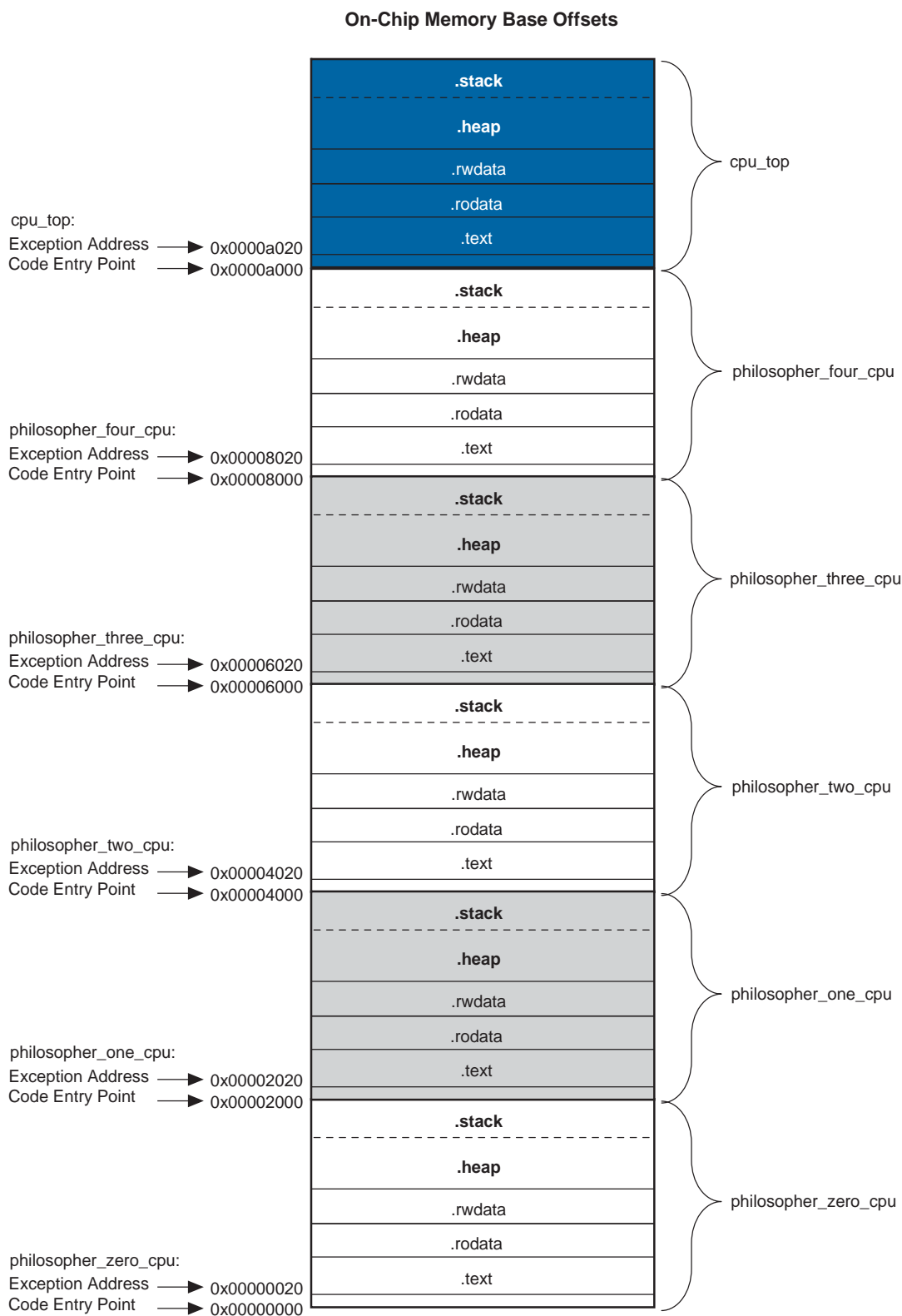


In a multiprocessor system, it might be advantageous to use a single memory to store all the code sections for each processor. In this case, the exception address set for each processor in Qsys is used to define the boundaries between where one processor's code section ends and where the next processor's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

For instance, imagine a system where on-chip memory occupies the following processor-specific address ranges:

- 0x00050000 to 0x0005FFFF—**cpu\_top** processor
- 0x04000000 to 0x0405FFFF—Nios II processor in any philosopher subsystem

Processor **cpu\_top** and the processors in each philosopher subsystem are each allocated eight KB of on-chip memory to run their software. If you use Qsys to set their exception addresses eight KB apart in on-chip memory, the Nios II SBT automatically partitions on-chip memory based on those exception addresses. See [Figure 1-6](#) for a memory map showing how the on-chip memory is partitioned in this example system. This figure depicts the memory map offsets to the base of on-chip memory in the top level of the hierarchy. The on-chip memory is seen by processors in each subsystem at a different address location than is seen by **cpu\_top** in the top level of the hierarchy. This address is obtained by adding the base address of the on-chip memory, as defined in the top level, to the base address of **out\_system\_bridge**, the Avalon-MM pipeline bridge, in the philosopher subsystem used to access components in the top level.

**Figure 1-6. Partitioning of On-Chip Memory for Six 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 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 for that processor.



Care must be taken when partitioning a physical memory to contain the code sections of multiple processors. There are no safeguards in Qsys or the Nios II SBT 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 region 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 because boot code typically just copies the real program code to where it is linked in RAM, and then branches to the program code. To boot multiple processors from separate regions with the same non-volatile memory device, simply set each processor's reset address to the location from which you need 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.

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



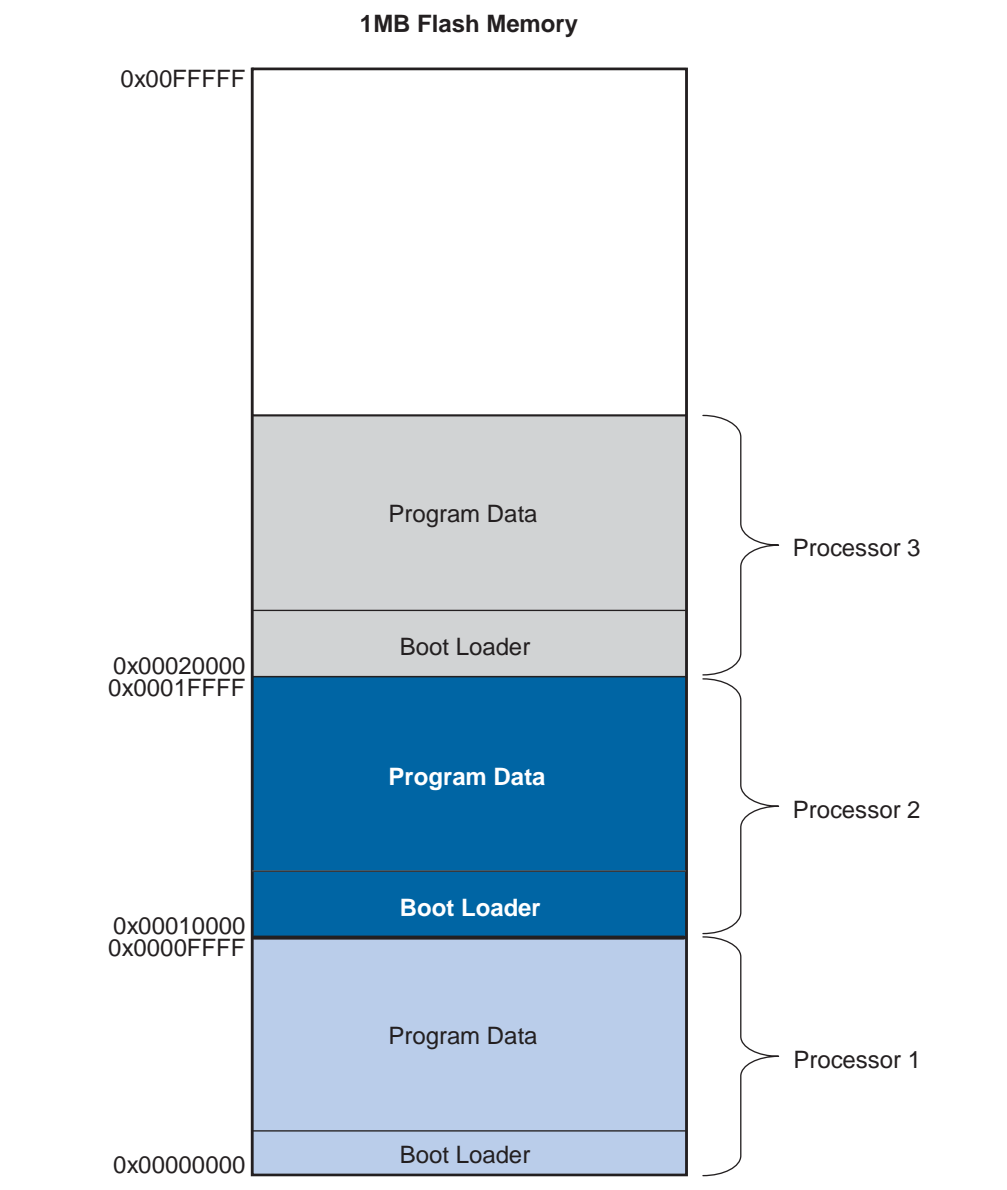
For details about the Nios II flash programmer, refer to the [Nios II Flash Programmer User Guide](#).



You must exercise caution when connecting multiple Nios II processors to a single CFI flash memory device. Because no support mechanism exists in the CFI flash driver to allow a processor to confirm that another processor is not currently accessing the flash memory device, a read operation can return corrupted data. Specifically, if a processor attempts to read from a CFI flash memory device currently not in read mode, the read operation does not access the data on the flash correctly. If another processor issues a query to the flash memory device immediately prior to the first processor's read attempt, the flash memory device is in command mode while it processes the query, and the read operation cannot read the data correctly. For this reason, Altera

recommends that you designate one Nios II processor as the flash master, and allow only the flash master to read from or write to the flash memory device, in any system that connects multiple Nios II processors to a single flash memory device. The designated processor can read the application images from the flash memory device for the other processors.

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



If you allow multiple Nios II processors to boot from the same CFI flash memory device, to ensure safe access to the CFI flash memory, you must remove the CFI flash memory driver initialization from the `alt_main()` function for all but one processor, and that processor must confirm boot completion by all the other processors before proceeding with the CFI flash memory driver initialization.



For information about complex boot procedures, refer to *AN458: Alternative Nios II Boot Methods*.

## Debugging Nios II Multiprocessor Designs

The Nios II SBT for Eclipse 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 SBT for Eclipse to perform simultaneous 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.

For more information about debugging multiprocessor systems, refer to “*Debugging the Software Projects on the Board*” on page 1–36.

## Design Example: The Dining Philosophers' Problem

The following exercise shows you how to build a hierarchical six processor Nios II system with Qsys, starting with the **multiprocessor\_tutorial\_start** design example as a template. You launch scripts to create and execute six applications and six BSPs, one project pair for each processor.

The Nios II multiprocessor design example demonstrates the use of multiple Nios II processors in an Altera FPGA. Although this example is primarily aimed at demonstrating a properly constructed hierarchical hardware system, it also contains the software to exercise the interprocessor coordination capabilities of the system.

This example implements the classic Dining Philosophers' Problem, illustrating resource sharing and synchronization. Imagine five philosophers seated at a round table. A single chopstick is positioned between each philosopher. Each philosopher tries first to grab the chopstick to her left, and then the chopstick to her right. If both chopsticks are acquired, the philosopher can eat. After a small delay, which represents the eating time, the philosopher drops both chopsticks, making them available to her neighboring philosophers. After another small delay, which represents thinking time, the cycle repeats. To prevent deadlock, if any philosopher cannot grab the right chopstick immediately after grabbing the left chopstick, she must drop the left chopstick and try again later.



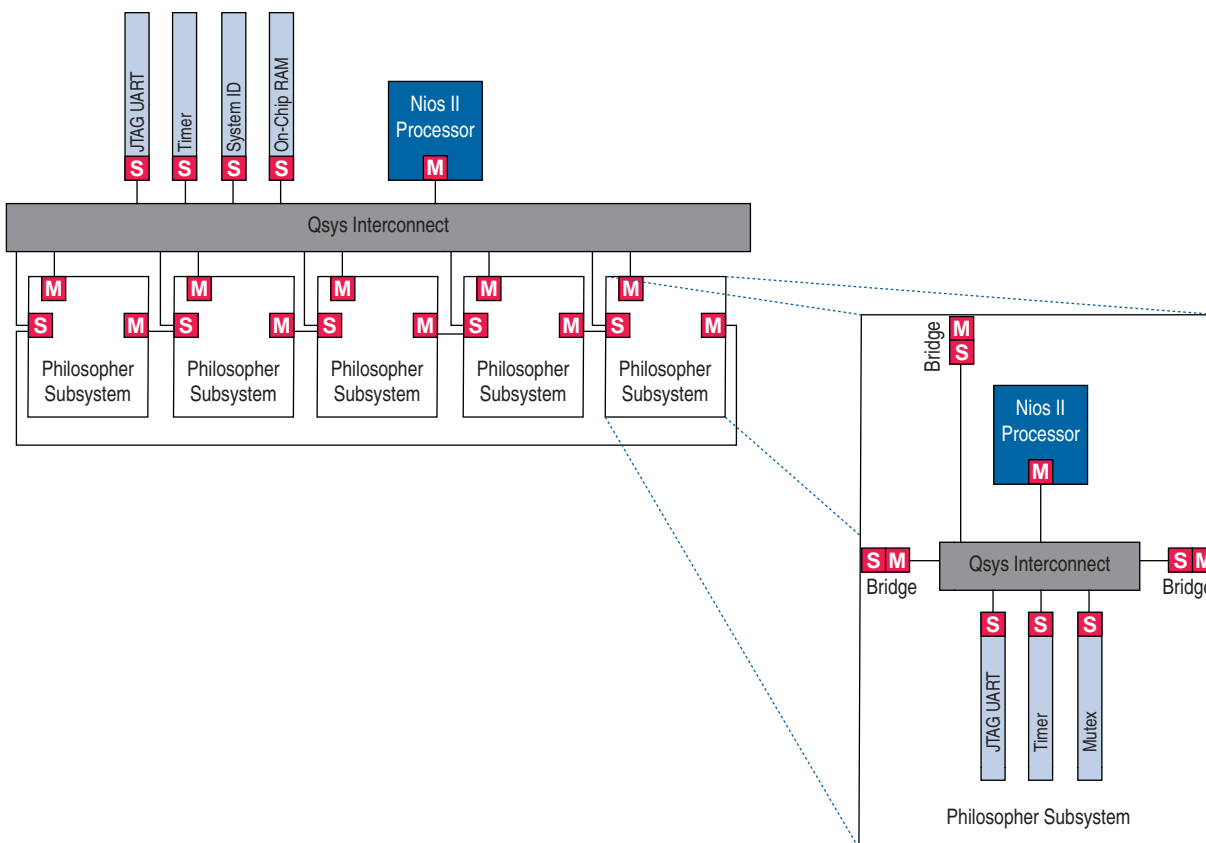
For a description of the Dining Philosophers' Problem, refer to the *Dining philosophers problem* article at [en.wikipedia.org](http://en.wikipedia.org). “Resource hierarchy solution” corresponds to the solution presented in this tutorial.

Created with Qsys, the hierarchical hardware design dedicates five processors to implement each of five dining philosophers, and five hardware mutexes to implement each of five chopsticks. A sixth Nios II processor and one on-chip RAM reside in the top level, along with a JTAG UART and timer. Each of the five subsystems shares the top-level on-chip RAM, and contains a processor, JTAG UART, timer, and mutex. Avalon-MM pipeline bridges enable communication between subsystem and top-level components, and between processors and mutexes located in logically adjacent subsystems connected in a ring.

The `dining_philosophers.c` software runs on each of five subsystem processors, implementing the thinking, eating, and chopsticks acquisition and release. The top-level processor executes `philosophers_monitor.c`, accepting numeric commands to acquire any mutex, preventing both logically adjacent “philosopher” processors from eating until that “chopstick” mutex is released.

Figure 1-8 shows the topology of the complete multiprocessor system. The Qsys interconnect, including all connections to individual components, is provided by Qsys.

**Figure 1-8. Hierarchical Nios II Multiprocessor System Block Diagram—System Level**



## Hardware and Software Requirements

To use this design example you must have the following:

- Quartus® II Software version 11.0 or higher
- Cyclone® III FPGA development board, connected through a USB-Blaster™ connection to the host computer



For information about the Cyclone III FPGA development board, refer to the [Altera Embedded Systems Development Kit, Cyclone III Edition](#) page on the Altera website.

If you do not have a Altera Cyclone III 3C120 development board, you can follow the hardware development steps, but you cannot download the complete system to a working board. The design example does not use any hardware unique to the Altera Cyclone III 3C120 development board. All peripherals used in the hardware design are soft IP; therefore, the design can be ported to any Altera FPGA.

## Installation Notes

You can download the Dining Philosophers' Problem design example from the [Nios II Multiprocessor Design Example](#) page on the Altera website.



For installation notes specific to Altera software versions, refer to the **readme.txt** file included in your **tt\_nios2\_multiprocessor\_design.zip** installation.

## Creating the Hardware System

In the following steps you create a multiprocessor system by starting with the **multiprocessor\_tutorial\_start** hardware design example available with this tutorial in **tt\_nios2\_multiprocessor\_design.zip**, and add six subsystems, each containing a Nios II processor, a timer, a clock, a JTAG UART, and a hardware mutex component. Your final system should be identical to that in the **multiprocessor\_tutorial\_final** hardware design available with this tutorial in **tt\_nios2\_multiprocessor\_design.zip**, for comparison purposes. If you do not have an Altera Cyclone III 3C120 development board, you can still follow these steps to learn how to design multiprocessor hardware.

### Getting Started with the multiprocessor\_tutorial\_start Design Example

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

1. Unzip the **tt\_nios2\_multiprocessor\_design.zip** file.
2. Copy the **Multiprocessor\_Tutorial\_start** folder to a working directory of your choice. Make sure the path name has no spaces. The remainder of this tutorial refers to your working directory as *<working directory>*.
3. Open the Quartus II software.
4. On the File menu, click **Open Project** (not Open).
5. Browse and load the Quartus II Project File (**multiprocessor\_tutorial.qpf**) from *<working directory>*.
6. On the Tools menu, click **Qsys**.
7. Select **philosopher\_zero.qsys**. Click **Open**

## Viewing a Philosopher System

First we'll examine one of the five Philosopher Systems to be added as subsystems to the Qsys hierarchical system.

**Figure 1–9. philosopher\_zero System**

Use	Connections	Name	Description	Export	Clock	Base	End
<input checked="" type="checkbox"/>		clk	Clock Source	philosopher_clk_in	clk		
<input checked="" type="checkbox"/>		clk_in	Clock Input	philosopher_clk_reset_in	clk		
<input checked="" type="checkbox"/>		clk_in_reset	Reset Input				
<input checked="" type="checkbox"/>		clk	Clock Output				
<input checked="" type="checkbox"/>		clk_reset	Reset Output				
<input checked="" type="checkbox"/>		cpu_zero	Nios II Processor		clk		
<input checked="" type="checkbox"/>		reset_n	Reset Input		clk		
<input checked="" type="checkbox"/>		data_master	Avalon Memory Mapped Master		clk		
<input checked="" type="checkbox"/>		instruction_master	Avalon Memory Mapped Master		clk		
<input checked="" type="checkbox"/>		jtag_debug_module_reset	Reset Output		clk		
<input checked="" type="checkbox"/>		jtag_debug_module	Avalon Memory Mapped Slave		clk		
<input checked="" type="checkbox"/>		custom_instruction_master	Custom Instruction Master		clk		
<input checked="" type="checkbox"/>		chopstick_mutex	Mutex		clk		
<input checked="" type="checkbox"/>		reset	Reset Input		clk		
<input checked="" type="checkbox"/>		s1	Avalon Memory Mapped Slave		clk	0x00008800	0x00008807
<input checked="" type="checkbox"/>		out_system_bridge	Avalon-MM Pipeline Bridge		clk		
<input checked="" type="checkbox"/>		reset	Reset Input		clk		
<input checked="" type="checkbox"/>		s0	Avalon Memory Mapped Slave		clk	0x04000000	0x07ffffff
<input checked="" type="checkbox"/>		m0	Avalon Memory Mapped Master		clk		
<input checked="" type="checkbox"/>		in_philo_bridge	Avalon-MM Pipeline Bridge		clk		
<input checked="" type="checkbox"/>		reset	Reset Input		clk		
<input checked="" type="checkbox"/>		s0	Avalon Memory Mapped Slave		clk		
<input checked="" type="checkbox"/>		m0	Avalon Memory Mapped Master		clk		
<input checked="" type="checkbox"/>		out_philo_bridge	Avalon-MM Pipeline Bridge		clk		
<input checked="" type="checkbox"/>		reset	Reset Input		clk		
<input checked="" type="checkbox"/>		s0	Avalon Memory Mapped Slave		clk	0x00008800	0x00008807
<input checked="" type="checkbox"/>		m0	Avalon Memory Mapped Master		clk		
<input checked="" type="checkbox"/>		timer	Interval Timer		clk		
<input checked="" type="checkbox"/>		reset	Reset Input		clk		
<input checked="" type="checkbox"/>		s1	Avalon Memory Mapped Slave		clk	0x00008900	0x0000891f
<input checked="" type="checkbox"/>		jtag_uart	JTAG UART		clk		
<input checked="" type="checkbox"/>		reset	Reset Input		clk		
<input checked="" type="checkbox"/>		avalon_jtag_slave	Avalon Memory Mapped Slave		clk	0x00008a00	0x00008a07

Messages

Description	Path
1 Info Message	
CPUID control register value is 0	System.cpu_zero

0 Errors, 0 Warnings

Each philosopher system is identical, except for a few of the Nios II processor configuration settings.

Each system contains a processor, clock, mutex, timer, JTAG UART, and three pipeline bridges. Table 1–1 illustrates the Nios II parameter settings differences.

In [Table 1-1](#), the **Vector Memory** and **Vector Offset** rows represent values entered in the Nios II Core parameter editor in Qsys, when configuring each Nios II processor. In a hierarchical system, if your processor defines exception and reset vectors in a memory component external to the subsystem, you must set the memory component to **Absolute**. Specify the absolute address in the **Reset Vector Offset** and **Exception Vector Offset** fields instead of the memory component base offset.

**Table 1-1. Nios II Parameter Settings**

Parameter	Values					
Processor Name	cpu_top	cpu_zero	cpu_one	cpu_two	cpu_three	cpu_four
<b>cpuid Control Register</b>	5	0	1	2	3	4
<b>Processor instance</b>	0	1	2	3	4	5
<b>JTAG UART instance</b>	0	1	2	3	4	5
<b>Reset Vector Memory</b>	onchip_memory.s1	Absolute	Absolute	Absolute	Absolute	Absolute
<b>Reset Vector Offset and Value</b>	0x0000A000	0x04050000	0x04052000	0x04054000	0x04056000	0x04058000
	0x0005A000	0x04050000	0x04052000	0x04054000	0x04056000	0x04058000
<b>Exception Vector Memory</b>	onchip_memory.s1	Absolute	Absolute	Absolute	Absolute	Absolute
<b>Exception Vector Offset and Value</b>	0x0000A020	0x04050020	0x04052020	0x04054020	0x04056020	0x04058020
	0x0005A020	0x04050020	0x04052020	0x04054020	0x04056020	0x04058020

Recall from “[Program Memory](#)” on [page 1-9](#) that the exception addresses determine how code memory is partitioned between processors. In this tutorial, each of the six processors runs its software from eight KB of on-chip memory, so you set each processor's exception address in on-chip memory, separated by 0x2000 (eight KB).

The first row contains the **CPUID Control Register** value, located under the **Advanced Features** tab. The rest of the rows of parameters can be found in the **Core Nios II** tab.

## Philosopher System Pipeline Bridges

Each philosopher subsystem has three Avalon-MM pipeline bridges. They enable each of the Nios II processors to access components in other levels of the hierarchy, creating a peripheral memory map unique to each processor. The three bridges are named **out\_system\_bridge**, **in\_philo\_bridge**, and **out\_philo\_bridge**.

**out\_system\_bridge** and **out\_philo\_bridge** each provide an address range window for accessing components in an external system. The address range for the bridge spans from the bridge base to a bridge base offset derived from the address width in bits specified with an addressable unit size of symbols (a.k.a. bytes).

**out\_system\_bridge** provides the processor with access to top-level components, specifically on-chip memory. At the top level, on-chip memory has a base address of 0x50000. **out\_system\_bridge** has a base address of 0x0400000. The base of the bridge is added to the top-level component base address to find the starting location in the memory map where on-chip memory appears for a processor in a philosopher subsystem, at 0x0405000. An address width of 26 creates a memory window provided by **out\_system\_bridge** spanning 0x03FFFFFF bytes, ranging from 0x0400000 to 0x07FFFFFF.

**out\_philo\_bridge** provides access for mutex peripherals located in adjacent subsystems. **out\_philo\_bridge** has a base of 0x80000. **out\_philo\_bridge** exports a master, which gets connected to a neighboring philosopher subsystem at the top level. To derive the address of the mutex in the neighboring subsystem, first take the base address of **out\_philo\_bridge** of 0x80000. Add to that the base address of a neighboring philosopher subsystem, for example **philosopher\_two**, with a base address of 0x020000. Finally, add the base address of the mutex itself at 0x8800, to get the address of 0x000A8800 for accessing **philosopher\_two**'s mutex from **philosopher\_one**. An address width of 19 bits enables an address window spanning 0x7FFFF bytes, ranging from 0x00080000 to 0x000FFFF.

**in\_philo\_bridge** is needed to provide an exported slave interface for the connection of remote processors to the local chopstick mutex. The **in\_philo\_bridge** master can then be connected to the local chopstick mutex at the same time that local philosopher processor master connects to the local chopstick mutex, providing simultaneous local and remote processor access to the chopstick mutex. This configuration is necessary because the chopstick mutex cannot simultaneously export its slave interface directly while connecting its slave interface locally. An address width of 16 bits enables an address window spanning 0xFFFF bytes.

In this multiprocessor tutorial's hierarchical design, each “philosopher” processor is connected to only two of the five total mutexes available in the design. The two mutex connections represent one chopstick on the left and one chopstick on the right.



One of the two connections, representing the left chopstick, is made directly in the Qsys connection panel from the processor's data master to the mutex slave local to the same subsystem as the processor. The other connection, representing the right chopstick, connecting the processor's data master to the slave port of a mutex located in a remote subsystem, is made in the following three stages:

1. The processor data master is connected in the Qsys connection panel to a pipeline bridge, named **out\_philo\_bridge**. **out\_philo\_bridge** exports its Avalon-MM master interface, named **outgoing\_philo\_master**. This connection is already made in the philosopher subsystem.
2. In the remote subsystem containing the mutex, the mutex slave interface is connected in the Qsys connection panel to another pipeline bridge, named **in\_philo\_bridge**. This connection is already made in the philosopher subsystem.  
**in\_philo\_bridge** exports its Avalon-MM slave interface, named **incoming\_philo\_slave**. Any exported interface cannot also be directly connected to components local to the subsystem. The mutex must connect to both a local processor master and a remote processor master. Therefore, instead of the mutex exporting its slave interface directly, the mutex slave interface is connected to both the local processor master and the local pipeline bridge master through the Qsys connection panel. Any remote processor's master can then be connected to the mutex slave indirectly through this pipeline bridge.
3. To provide the indirect connection between the local processor's data master to the slave port of the remote mutex, connect the two bridges at the top level of the hierarchy. In the Qsys connection panel of the top-level design, connect the **outgoing\_philo\_master** interface of the philosopher subsystem containing the processor to the **incoming\_philo\_slave** exported interface of the philosopher subsystem containing the mutex.

## Adding Philosopher Subsystems

In the next series of steps, you add five Philosopher Subsystems to the Qsys hierarchical system. Open the top-level design in Qsys, by performing the following steps:

1. On the Qsys File menu, click **Open**.
2. Select **multiprocessor\_tutorial\_main\_system.qsys** and click **Open**.
3. To add philosopher zero, perform the following steps:
  - a. In the Component Library on the left side of the **System Contents** tab, under **Project**, expand **System**, and select **philosopher\_zero**.
  - b. Click **Add**. The **philosopher\_zero** parameter editor appears. Click **Finish**.
  - c. Right-click the added component name, **zero\_0**, and click **Rename**.
  - d. Type **philosopher\_zero** ↵.
4. Add philosophers one through four, in that order, by repeating steps **a** through **d**. In place of zero, substitute one, two, three and four, in turn.
5. For each philosopher subsystem, in the **Export** column, click the automatically exported names (**clk** for the clock input, and **reset** for the reset input), and press backspace to delete the exported names.

## Connecting the Philosopher Subsystems

Using the Qsys connection matrix, make the following port connections.



Some of the connections to **cpu\_top** already exist in the multiprocessor\_tutorial\_start hardware design, but are listed again in this section for completeness.

1. Connect the clock input for all five philosopher subsystem's clock input as shown in [Table 1-2](#).

**Table 1-2. Connecting Philosopher Clock Inputs**

Connect From	
	<b>clk.clk</b>
	<b>Connect To</b>
✓	philosopher_zero.philosopher_clk_in
✓	philosopher_one.philosopher_clk_in
✓	philosopher_two.philosopher_clk_in
✓	philosopher_three.philosopher_clk_in
✓	philosopher_four.philosopher_clk_in

2. Connect the reset inputs for all processors and **clk** to each of the **clk** component's reset output, and the **cpu\_jtag\_debug\_module\_reset** reset output of every processor (**cpu\_top** and each philosopher subsystem), for a total of 42 reset connections, as shown in [Table 1-3](#).

**Table 1-3. Connecting Philosopher Reset Inputs**

Connect From							Connect To
clk.clk_reset	cpu_top.jtag_debug_module_reset	philosopher_zero.cpu_jtag_debug_module_reset	philosopher_one.cpu_jtag_debug_module_reset	philosopher_two.cpu_jtag_debug_module_reset	philosopher_three.cpu_jtag_debug_module_reset	philosopher_four.cpu_jtag_debug_module_reset	
✓	✓	✓	✓	✓	✓	✓	philosopher_zero.philosopher_clk_reset_in
✓	✓	✓	✓	✓	✓	✓	philosopher_one.philosopher_clk_reset_in
✓	✓	✓	✓	✓	✓	✓	philosopher_two.philosopher_clk_reset_in
✓	✓	✓	✓	✓	✓	✓	philosopher_three.philosopher_clk_reset_in
✓	✓	✓	✓	✓	✓	✓	philosopher_four.philosopher_clk_reset_in
✓	✓	✓	✓	✓	✓	✓	philosopher_five.philosopher_clk_reset_in

3. Connect the reset inputs for the system ID peripheral and on-chip memory to the reset output of each processor, as shown in [Table 1-4](#).

Table 1-4. Connecting Peripheral Reset Inputs

Connect From						Connect To
cpu_top.jtag_debug_module_reset	philosopher_zero.cpu_jtag_debug_module_reset	philosopher_one.cpu_jtag_debug_module_reset	philosopher_two.cpu_jtag_debug_module_reset	philosopher_three.cpu_jtag_debug_module_reset	philosopher_four.cpu_jtag_debug_module_reset	
✓	✓	✓	✓	✓	✓	onchip_memory.reset1
✓	✓	✓	✓	✓	✓	sysid_qsys.reset

In the next two steps, you connect the two Avalon-MM masters in each philosopher subsystem to the components in other subsystems and components at the top level that each philosopher processor accesses.

4. Connect the Avalon-MM master exported pipeline bridge connection, named **outgoing\_master**, for each philosopher subsystem, to the two components in the top level that every philosopher processor accesses, which are the system ID peripheral and on-chip memory, as shown in [Table 1-5](#).

Table 1-5. Connecting Philosopher Masters to Peripherals

Connect From						Connect To
cpu_top.data_master	philosopher_zero.cpu_outgoing_master	philosopher_one.cpu_outgoing_master	philosopher_two.cpu_outgoing_master	philosopher_three.cpu_outgoing_master	philosopher_four.cpu_outgoing_master	
✓	✓	✓	✓	✓	✓	onchip_memory.control_slave
✓	✓	✓	✓	✓	✓	sysid_qsys.s1

5. Connect the Avalon-MM master exported pipeline bridge connection, named **outgoing\_philo\_master**, for each philosopher subsystem, to the Avalon-MM slave exported pipeline bridge connection, named **incoming\_philo\_slave**, in a logically adjacent philosopher subsystem, granting access to the chopstick mutex in that neighboring philosopher subsystem, as shown in [Table 1-6](#).

**Table 1-6. Connecting Philosopher Masters and Slaves**

Connect From					Connect To
philosopher_zero.outgoing_philo_master	philosopher_one.outgoing_philo_master	philosopher_two.outgoing_philo_master	philosopher_three.outgoing_philo_master	philosopher_four.outgoing_philo_master	
				✓	philosopher_zero.incoming_philo_slave
✓					philosopher_one.incoming_philo_slave
	✓				philosopher_two.incoming_philo_slave
		✓			philosopher_three.incoming_philo_slave
			✓		philosopher_four.incoming_philo_slave

6. Finally, connect the **cpu\_top** Nios II processor's Avalon-MM master, named **data\_master**, to each philosopher subsystem's **incoming\_philo\_slave** Avalon-MM slave, as shown in [Table 1-7](#).

**Table 1-7. Connecting cpu\_top Master to Philosopher Slaves**

Connect From	Connect To
cpu_top.data_master	
✓	philosopher_zero.incoming_philo_slave
✓	philosopher_one.incoming_philo_slave
✓	philosopher_two.incoming_philo_slave
✓	philosopher_three.incoming_philo_slave
✓	philosopher_four.incoming_philo_slave

- After all the connections are specified, assign the base addresses for each philosopher subsystem. In the **Base** column, double-click and type each address shown in Table 1-8.

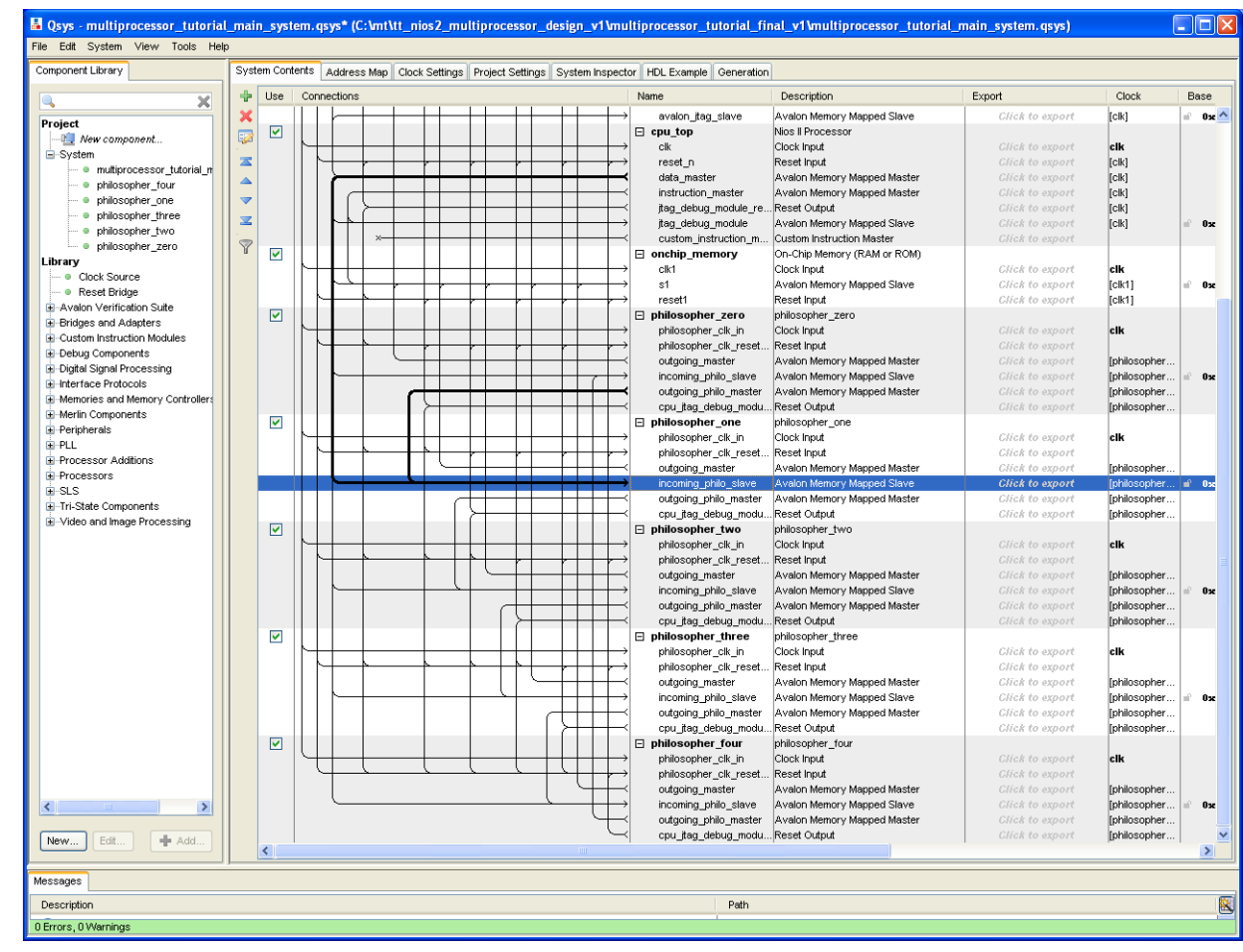
**Table 1-8. Philosopher Subsystem Base Addresses**

Processor Name	Base Address
philosopher_zero	0x00000000
philosopher_one	0x00010000
philosopher_two	0x00020000
philosopher_three	0x00030000
philosopher_four	0x00040000

## Viewing the Complete System

Figure 1-10 shows the system in Qsys after you complete the changes in the previous section. It shows the new connected philosopher subsystem components and the required connectivity for the system.

**Figure 1-10. Shared Peripheral Connections**



The **multiprocessor\_tutorial\_final** design example is also available in the **tt\_nios2\_multiprocessor\_design.zip** file. You can compare your completed system to the predefined system located in the **multiprocessor\_tutorial\_final** subdirectory of the extracted directory.

## Generating and Compiling the System

In this section, you generate HDL for the system you just constructed in Qsys, 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 **Generation** tab.
2. Turn off **Simulation**. Set **Create simulator model** to **None**. Set **Create testbench Qsys system** to **None**. 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, a **GENERATE COMPLETED** message appears. Click **Close**.
5. Exit Qsys to return to the Quartus II software.
6. On the Quartus II Processing menu, click **Start Compilation** to compile the project in the Quartus II software.
7. When compilation completes and displays the **Full compilation was successful** message, click **OK**.

## Generating Hierarchical Systems in Qsys

Generate only the top-level system in the Qsys hierarchy. In this tutorial, the top-level system is in **multiprocessor\_tutorial\_main\_system.qsys**. Generating the top-level Qsys system automatically generates any subsystems in the top-level system.



If you create a **.sopcinfo** file for a subsystem, and attempt to build a BSP based on that **.sopcinfo** file, when the BSP or application tries to refer to components in the top level, those references fail. Instead, you must base the BSP on the **.sopcinfo** file for the top-level Qsys system.

For example, for the Multiprocessor Tutorial, if you create **philosopher\_zero.sopcinfo** by generating **philosopher\_zero.qsys** directly, and create a BSP based on **philosopher\_zero.sopcinfo**, BSP creation fails. The failure results in a severe error indicating that absolute address 0x04050000 does not reference a device in the master group **cpu\_zero**. The absolute address 0x04050000 is intended to reference a location in the address span of the on-chip memory declared in the top level of the Multiprocessor Tutorial hardware design. Therefore BSP creation fails because on-chip memory is located in the top level, outside of the **philosopher\_zero.qsys** subsystem.

Instead, you must generate a **multiprocessor\_tutorial\_main\_system.sopcinfo** from the top-level system, and base the BSP on this **.sopcinfo** file.



## Creating Software for the Multiprocessor System

In the following steps you build one application and one BSP project for each processor in the system using the Nios II SBT through scripts, creating a total of twelve separate software projects for the multiprocessor system. You also download the contents of .elf files to each processor, and launch **nios2-terminal** sessions for each JTAG UART through these scripts. You then import and debug the software projects using the Nios II SBT for Eclipse.



If you encounter any difficulties executing the software on your new hardware design, Altera recommends that you try running the software on the `multiprocessor_tutorial_final` hardware design to validate your execution of these procedures. After you are successful running software on the `multiprocessor_tutorial_final` hardware design, compare the Qsys system of your new hardware design with the `multiprocessor_tutorial_final` hardware design's Qsys system, and look for any differences that might be responsible for incorrect behavior of your new hardware design.

## Building and Running the Applications from the Command Line

In this section, you build and launch the `philosopher` and `philosophers_monitor` applications, view their output in **nios2-terminal**, and interact with them.

### Building and Launching the Applications

To build the applications for this tutorial, perform the following steps:

1. Start six Nios II Command Shells.
2. Change directories to *<working directory>* in each shell.
3. Change directories to **software**.
4. Download the **multiprocessor\_tutorial.sof** file that you just built, by typing the following command in one shell:

```
./multiprocessor_tutorial_hw.sh ↵
```



If you are using more than one USB-Blaster cable, specify the numeric cable number as the last parameter to each of these scripts. You can see a description of command usage by typing `philosopher.sh` with no arguments.

5. Build, download, and start the project to be run on the **cpu\_top** processor by typing the following command in the same shell you used in the previous step:

```
./philosophers_monitor.sh ↵
```

6. Build, download, and start the project to be run on the **cpu\_zero** processor by typing the following command in another shell:

```
./philosopher.sh 0 ↵
```

7. Build, download, and start the projects to be run on the **cpu\_one**, **cpu\_two**, **cpu\_three**, and **cpu\_four** processors by repeating step 6 with command-line arguments 1, 2, 3, and 4.

These scripts build and download each application to its respective target. Application software projects and BSP projects are created and built.

If you make hardware design changes in Qsys, you can easily regenerate the application and BSP projects to reflect those changes by rerunning these scripts. The scripts rebuild the projects each time they are executed.

The scripts also start **nios2-terminal** for viewing output from each application. In each of the shells, observe the five philosophers eating, thinking, and waiting for chopsticks to become available.



The Quartus II software automatically assigns instance numbers for components. You cannot directly control how the instance numbers are assigned. However, they are assigned systematically, and a subsequent Quartus II compile of any particular hardware design is guaranteed to produce the same instance numbers as previous compiles of the identical hardware design, for both JTAG UARTs and Nios II processors. So, after a Quartus II compile, you just need to try the different instance numbers. One way to tell which particular Nios II processor is executing the software following a download to a particular Nios II processor instance is to print `ALT_CPU_CPU_ID_VALUE`, after specifying unique values in Qsys for each Nios II processor. However, that number is not the same as an instance number.

**nios2-download** allows you to specify the target processor by name, instead of by instance number. To specify the processor by name, use the `--cpu-name` switch. Drop the subsystem portion of the derived hierarchical processor name, and only specify the local name of a processor in a subsystem.

For example, to target **philosopher\_one\_cpu\_one**, use `--cpu-name=cpu_one`.

**nios2-download** also accepts a switch, `--jdi`, for specifying the jdi file for the hardware design. Use of the `--jdi` switch is required with `--cpu-name`. For the **multiprocessor\_tutorial\_start** design example, specify `<working directory>/multiprocessor_tutorial.jdi`.

JTAG UART instance numbers are also automatically assigned independently from the Nios II processors by the Quartus II software, and do not necessarily match the instance number of the Nios II processor to which the JTAG UART is connected. You must specify the JTAG UART by its instance number, using the `--instance` switch on the command line. Use `jtagconfig -n` to display all the instances of Nios II processors and JTAG UARTs available in your hardware design. To check which JTAG UART instance matches which Nios II processor in your Qsys hardware design, perform the following steps:

1. Open **nios2-terminal** sessions in separate windows for all JTAG UART instances.
2. Download code to one of the Nios II processors
3. Make a note of the **nios2-terminal** that receives the output, with its JTAG UART instance number.
4. Download code to each Nios II processor in turn, until you have a list of **nios2-terminal** windows connected and processor names.

PHILOSOPHER\_DOWNLOAD\_CPU\_NAME and JTAG\_UART\_INSTANCE\_NUMBER are variables set and used by these scripts to specify both the processor name and the JTAG UART instances. In **philosophers\_monitor.sh**, JTAG\_UART\_INSTANCE\_NUMBER is set to 0, and PHILOSOPHER\_DOWNLOAD\_CPU\_NAME is set to cpu\_top. In **philosopher.sh**, these two variable values are derived from the first parameter philosopher number that you type on the command line. If you customize the Qsys hardware design provided with this tutorial, modify the assignment for the JTAG\_UART\_INSTANCE\_NUMBER variable in the scripts if you find that the JTAG UART instance numbers no longer correspond to the processor names in [Table 1-1 on page 1-19](#).

Philosophers are seated counterclockwise, so philosopher 4 sits to the right of philosopher 3, and philosopher 0 sits to the right of philosopher 4. Philosopher 4 grabs the chopstick in his own subsystem, representing the left chopstick. Philosopher 4 grabs the chopstick in his neighboring philosopher 0's subsystem, representing the right chopstick from the point of view of philosopher 4.

**Figure 1–11. Monitor and Five Philosopher Applications Running**



In the following steps, you interact with the philosophers\_monitor application. You demonstrate peripheral contention by grabbing chopsticks from each philosopher subsystem, and observing the results.

1. Grab chopstick 1 by pressing 1 in the philosophers\_monitor shell, as shown in [Figure 1-12](#).

**Figure 1-12. philosophers\_monitor Running in nios2-terminal**

```
C:\ Altera Nios II EDS 11.0 [gcc4]
Executing nios2-terminal --cable=1 --instance=5
nios2-terminal: connected to hardware target using JTAG UART on cable
nios2-terminal: "USB-Blaster [USB-0]", device 1, instance 5
nios2-terminal: <Use the IDE stop button or Ctrl-C to terminate>

Hello from top CPU 5.
To alternately grab or drop a chopstick, enter a chopstick id <0 - 4>:
1
Chopstick 1 selected.
Attempting to grab chopstick 1.
Chopstick 1 grabbed.
4
Chopstick 4 selected.
Attempting to grab chopstick 4.
Chopstick 4 grabbed.
0
Chopstick 0 selected.
Attempting to grab chopstick 0.
Chopstick 0 grabbed.
4
Chopstick 4 selected.
Chopstick 4 dropped.
```

Philosopher 4 continues to eat and think, as shown in [Figure 1-13](#).

**Figure 1–13. Philosopher 4 Eating and Thinking**

[illegible]

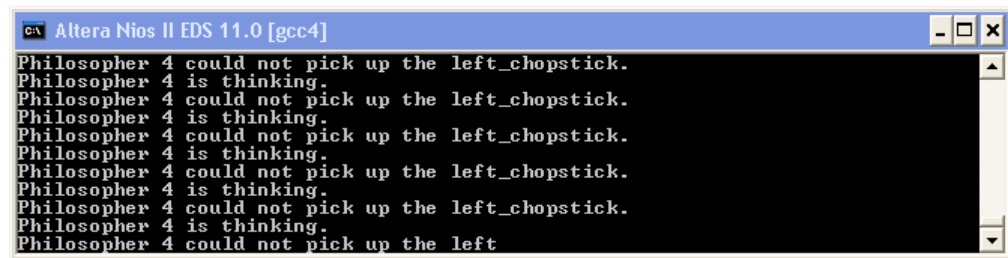
- Grab chopstick 0 by pressing 0 in the philosophers\_monitor shell. Philosopher 4 cannot pick up the right chopstick, as shown in [Figure 1-14](#).

**Figure 1–14. Philosopher 4 with Chopstick 0 Grabbed**

```
C:\ Altera Nios II EDS 11.0 [gcc4]
Philosopher 4 could not pick up the right_chopstick.
Philosopher 4 is thinking.
Philosopher 4 could not pick up the right_chopstick.
Philosopher 4 is thinking.
Philosopher 4 could not pick up the right_chopstick.
Philosopher 4 is thinking.
Philosopher 4 could not pick up the right_chopstick.
Philosopher 4 is thinking.
Philosopher 4 could not pick up the right_chopstick.
Philosopher 4 is thinking.
Philosopher 4 could no
```

3. Grab chopstick 4 by pressing 4 in the philosophers\_monitor shell. Philosopher 4 cannot pick up the left chopstick, as shown in Figure 1-15.

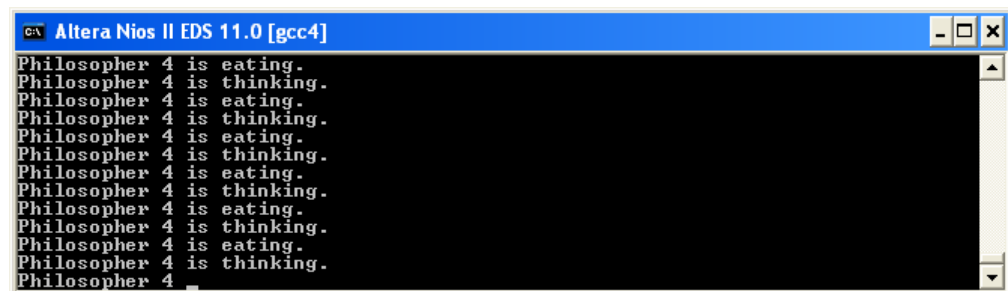
**Figure 1-15. Philosopher 4 with Chopstick 4 Grabbed**



```
Altera Nios II EDS 11.0 [gcc4]
Philosopher 4 could not pick up the left_chopstick.
Philosopher 4 is thinking.
Philosopher 4 could not pick up the left_chopstick.
Philosopher 4 is thinking.
Philosopher 4 could not pick up the left_chopstick.
Philosopher 4 is thinking.
Philosopher 4 could not pick up the left_chopstick.
Philosopher 4 is thinking.
Philosopher 4 could not pick up the left_chopstick.
Philosopher 4 is thinking.
Philosopher 4 could not pick up the left
```

4. Drop both chopsticks 0 and 4 by pressing 0 and 4 in the philosophers\_monitor shell. Philosopher 4 can eat again, as shown in Figure 1-16.

**Figure 1-16. Philosopher 4 with No Chopsticks Grabbed**



```
Altera Nios II EDS 11.0 [gcc4]
Philosopher 4 is eating.
Philosopher 4 is thinking.
Philosopher 4 is eating.
Philosopher 4 is thinking.
Philosopher 4 is eating.
Philosopher 4 is thinking.
Philosopher 4 is eating.
Philosopher 4 is thinking.
Philosopher 4 is eating.
Philosopher 4 is thinking.
Philosopher 4 is eating.
Philosopher 4 is thinking.
Philosopher 4 is eating.
```

## Debugging the Applications in the Nios II SBT for Eclipse

In this section, you set up a debugging environment in the Nios II SBT for Eclipse, and begin debugging the applications.

### Starting the Nios II SBT for Eclipse

In this section, you start the Nios II SBT for Eclipse and begin importing software projects for the two Nios II processors in the system. In the Windows operating system, start the Nios II SBT for Eclipse from Qsys by performing the following steps:

1. On the Tools menu in the Quartus II software, click **Qsys**.
2. In Qsys, click **Cancel** when prompted to open a Qsys system.
3. On the Tools menu, click **Nios II Software Build Tools for Eclipse**. The Nios II SBT for Eclipse starts.



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

## Importing the Software Projects

In this section, you import the following four projects to the Nios II SBT for Eclipse:

- **app\_monitor**
- **bsp\_monitor**
- **app4**
- **bsp4**

To import the software projects, perform the following steps:

1. In the Nios II SBT for Eclipse, on the File menu, click **Import**.
2. Expand **Nios II Software Build Tools Project**. The **Import Nios II Software Build Tools Project** option appears.
3. Highlight **Import Nios II Software Build Tools Project**, and click **Next**. The **Import Software Build Tools Project** window opens.
4. For **Project location**, browse to one of the four projects that you built by running the **philosophers\_monitor** and **philosopher** scripts. For example, browse to **software**, highlight the **app4** folder, and click **OK**.
5. For **Project name**, enter your project name, for example **app4**, and click **Finish**.



If a dialog box appears with the message **Do you want the Nios II Software Build Tools for Eclipse to manage your makefile for you?**, click **Yes**.

6. Repeat steps 1 to 5 to import the remaining three projects: **bsp4**, **app\_monitor**, and **bsp\_monitor**.

## Building the Software Projects

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

To build the software projects, perform the following steps:

1. In the Nios II perspective, right-click the project **app4** and click **Build Project**.
2. Right-click the project **app\_monitor** and click **Build Project**.

## Launching nios2-terminal for stdio Connections

In this section, you launch two **nios2-terminal** sessions to connect to the **stdout** and **stdin** devices on each processor outside of Nios II SBT for Eclipse. These **nios2-terminal** sessions receive output from and send input to application projects launched in multiple Nios II SBT for Eclipse debug sessions.


Launch **nios2-terminal** in two different Nios II command shells, one for each debug configuration, specifying the JTAG UART instances with the `--instance` switch. For this Multiprocessor Tutorial hardware design, each processor uses (philosopher number + 1) as the assigned instance IDs for both processors and JTAG UARTs. For example, project **app4** running on **cpu\_four** uses JTAG UART instance 5, the same instance ID as **cpu\_four**. Project **AppMonitor**, running on **cpu\_top**, uses JTAG UART instance 0, the same instance ID as **cpu\_top**.

```
nios2-terminal --instance=5 ←  
nios2-terminal --instance=0 ←
```

## Creating and Running a Launch Configuration for Each Processor

In this section, you create a launch configuration for each of the target processors and start debugging with those configurations. These configurations enable you to run and debug the multiple software projects you just built for the processors in the system.

To create a debug configuration for one of the processors, perform the following steps:

1. In the **Nios II** perspective, click the **app4** project.
  2. On the Run menu, click **Debug Configurations**.
  3. In the configurations list, right-click **Nios II Hardware**.
  4. Click **New**. A new debug configuration is created for the project.
  5. In the Name field, type **App4**.
  6. Open the **Project** tab.
  7. For **Project Name**, select **app4**.
  8. Open the **Target Connection** tab.
  9. Ensure that the **Name** column under **Processors** is populated. If no processor names appear, perform the following steps:
    - a. Click **Refresh Connections**.
    - b. Select one of the processor rows, then click **Resolve Names**.
    - c. If the **Name** column remains unpopulated, perform the following additional steps:
      - In the **Project** tab, click **Advanced**. The **Nios II ELF Section Properties** dialog box appears.
      - Under **Other**, turn off **Use default JDI File extracted from ELF file** and set **JTAG Debugging Information File name** to `<working directory>/multiprocessor_tutorial.jdi`.
      - Click **Close**.
-  The processor name listed in Nios II ELF Section Properties shows the full hierarchical name. The SBT does not use the full hierarchical processor names.
10. Under **Processors**, ensure that the row with **Name** value **cpu\_four** is selected.



11. If you made changes to the debug configuration, the **Apply** button is enabled. Click **Apply**.



Because the Nios II SBT does not use the full hierarchical processor names, you specify **cpu\_four**, not **philosopher\_four\_cpu\_four**, when creating the debug configuration for project **app4**. When selecting processor instances, you can ignore the following message at the top of the Debug Configuration window:

The expected CPU name does not match the selected target CPU name.

12. Under Byte Stream Devices, turn on **Disable 'Nios II Console' view**. This setting lets you use **nios2-terminal** to connect to the stdout and stdin device on each processor outside of Nios II SBT for Eclipse.
13. Click **Debug**. The Nios II SBT for Eclipse downloads and launches the **app\_four** software project on the **cpu\_four** processor, then pauses **cpu\_four** at a breakpoint set on `main()`.
14. If the **Debug Configurations** dialog box does not close automatically, click **Close** to return to the Nios II perspective.
15. If you are prompted to enter the Nios II Debug perspective, click **Yes**.
16. If you are not already in the Nios II Debug perspective, click the Debug perspective icon in the top right corner of your Nios II SBT for Eclipse window.
17. Check that the **App4** debug session, including the call stack, appears in the Nios II Debug perspective.
18. To return to the Nios II perspective, click the Nios II perspective icon in the top right corner of your Nios II SBT for Eclipse window. If the Nios II perspective icon is not visible, click the yellow plus-sign Open Perspective button.
19. Repeat steps 1 through 17 to create and run a debug configuration to execute the **app\_monitor** project on the **cpu\_top** target processor, substituting **app\_monitor** for **app4** and **cpu\_top** for **cpu\_four**.

You have created, downloaded, and started a debug configuration for each processor in the system. At this point, you can resume the code execution and debug code on each of the processors individually, using the normal flow for running or debugging.

Each processor begins executing code immediately after its code is downloaded to the FPGA; the processors do not start in unison. Although each processor begins running the code as soon as it is downloaded, the debug configuration ensures that each processor stops at a breakpoint set on `main()`.



For many multiprocessor designs, the Launch Group feature can save steps by downloading all **.elf** files to all processors at once. However, the Nios II SBT for Eclipse does not support Nios II console connections to multiple JTAG UARTs with the Launch Group feature.

## Debugging the Software Projects on the Board

After you download both debug configurations to the Altera Cyclone III 3C120 development board, you must resume code execution on each processor.

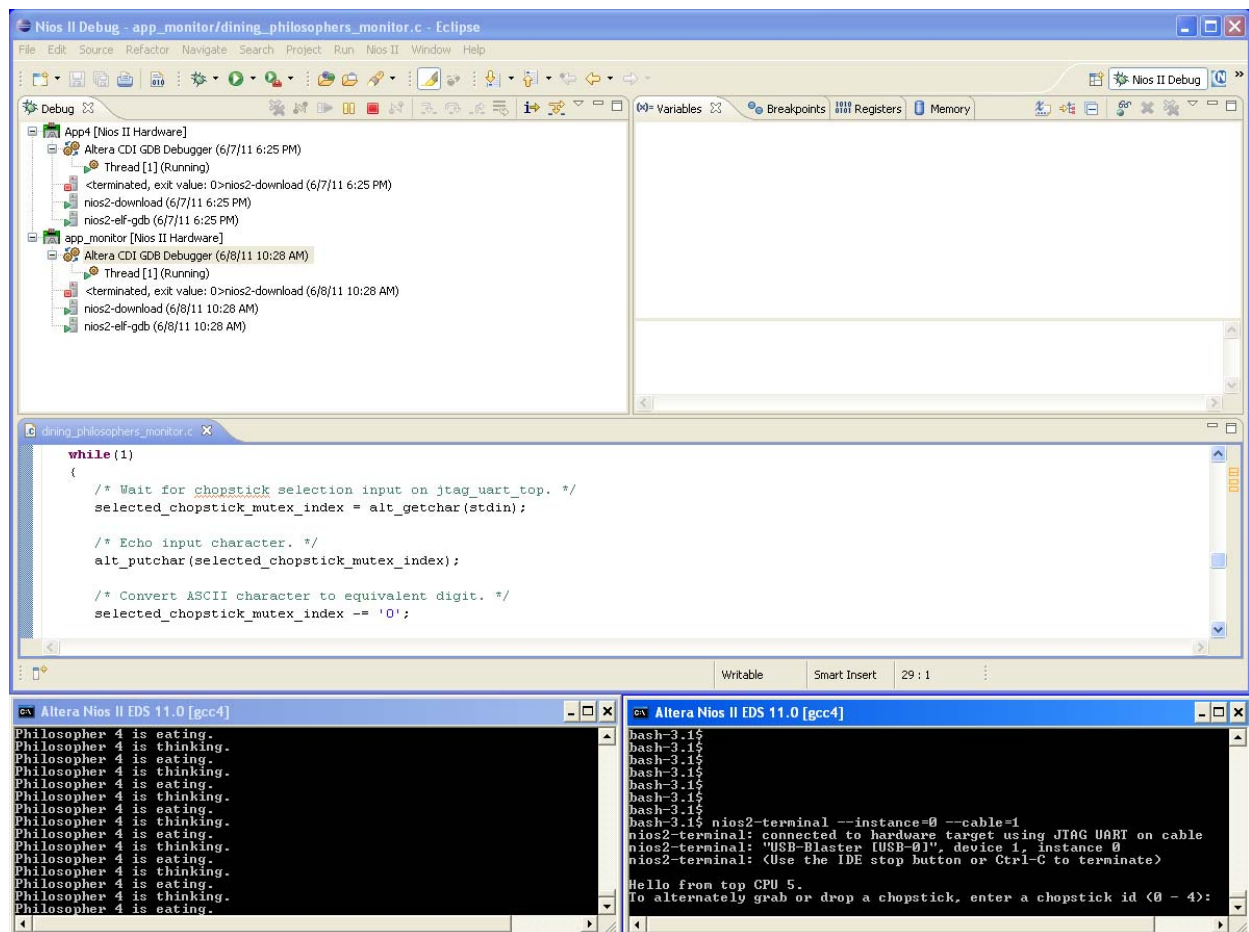


To run the design example on the Altera Cyclone III 3C120 development board, after the two debug configurations are invoked, perform the following steps:

1. Ensure that both software projects are paused at the beginning of `main()`.
2. To continue the **app\_monitor** debug run past the initial breakpoint, perform the following steps:
  - a. To use the debugger to step through code, click the `main()` call stack entry under the **app\_monitor** debug session.
  - b. Click the Step Over icon in the toolbar menu to see **app\_monitor** step through the software code.
  - c. To let **app\_monitor** run freely, click the green arrow Resume icon in the toolbar menu. Output and input for debug session **app\_monitor** are exchanged in the **nios2-terminal** session with instance 0, connected to the **cpu\_top** Nios II processor.

Figure 1-17 shows multiple threads running in the Nios II SBT for Eclipse.

**Figure 1-17. Multiple Processors in the Nios II SBT for Eclipse**



3. To continue debugging the app4 project past the initial breakpoint, perform the following steps:
  - a. Click the `main()` call stack entry under the **App4** debug session.
  - b. Click the green arrow Resume icon in the toolbar menu. The software running in the App4 launch configuration on the **cpu\_four** processor sends messages to `stdout` describing the eating and thinking activity for philosopher 4. This output appears on the **nios2-terminal** session with instance 5, connected to the `cpu_four` processor.
4. In the first **nios2-terminal** session (monitoring **cpu\_top**), grab and drop the 0 and 4 chopsticks. Observe the **cpu\_four** eating and thinking. After each chopstick grab, watch the behavior in the second **nios2-terminal** session (monitoring **cpu\_four**). After grabbing chopstick 0, philosopher 4 could not pick up the right chopstick. After grabbing chopstick 4, philosopher 4 could not pick up the left chopstick.

## Conclusion

In this tutorial, you constructed, built software projects for, and debugged software on a Nios II multiprocessor system. You have also learned how to use the Altera mutex core to synchronize system peripheral control among multiple processors. Feel free to experiment with the system you have 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 need to create a multiprocessor system.

This chapter provides additional information about the document and Altera.

## Document Revision History

The following table shows the revision history for this document.

Date	Version	Changes
June 2011	2.0	<ul style="list-style-type: none"> <li>■ Updated for Qsys</li> <li>■ Added new example design based on the Dining Philosophers' problem</li> </ul>
February 2010	1.4	Updated for Nios II Software Build Tools for Eclipse.
December 2007	1.3	Updated for Quartus II 7.2 release: minor text changes.
May 2007	1.2	Updated for Quartus II 7.1 release.
May 2006	1.1	Updated for Quartus II 6.0 release.
April 2005	1.0	Initial release.

## How to Contact Altera

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








Contact (1)	Contact Method	Address
Technical support	Website	<a href="http://www.altera.com/support">www.altera.com/support</a>
Technical training	Website	<a href="http://www.altera.com/training">www.altera.com/training</a>
	Email	<a href="mailto:custrain@altera.com">custrain@altera.com</a>
Product literature	Website	<a href="http://www.altera.com/literature">www.altera.com/literature</a>
Non-technical support (General) (Software Licensing)	Email	<a href="mailto:nacomp@altera.com">nacomp@altera.com</a>
	Email	<a href="mailto:authorization@altera.com">authorization@altera.com</a>

**Note to Table:**

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

## Typographic Conventions

The following table shows the typographic conventions this document uses.

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