Linux® Device Driver Programmer's Guide – Porting to SGI® Altix® Systems

007–4520–006

COPYRIGHT
© 2003, 2004, 2005, Silicon Graphics, Inc. All rights reserved; provided portions may be copyright in third parties, as indicated elsewhere herein. No permission is granted to copy, distribute, or create derivative works from the contents of this electronic documentation in any manner, in whole or in part, without the prior written permission of Silicon Graphics, Inc.

LIMITED RIGHTS LEGEND
The software described in this document is "commercial computer software" provided with restricted rights (except as to included open/free source) as specified in the FAR 52.227-19 and/or the DFAR 227.7202, or successive sections. Use beyond license provisions is a violation of worldwide intellectual property laws, treaties and conventions. This document is provided with limited rights as defined in 52.227-14.

TRADEMARKS AND ATTRIBUTIONS
Silicon Graphics, SGI, the SGI logo, Altix, IRIX, and Origin are registered trademarks and NUMAlink is a trademark of Silicon Graphics, Inc., in the United States and/or other countries worldwide.

Intel and Itanium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Linux is a registered trademark of Linux Torvalds. Motorola is a registered trademark of Motorola, Inc. All other trademarks mentioned herein are the property of their respective owners.

The information in Appendix A, “Memory Operation Ordering on SGI Altix Systems,” was originally authored by Jesse Barnes.

New Features in This Guide

This release provides the following:

- Updated the steps for building a new kernel in "Building a New Linux Kernel" on page 95.
- Updated information about how to boot a new Linux kernel in "Building a New Linux Kernel" on page 95.
- Replaced session trace information in "Downloading SGI Altix RPMs" on page 97.
- Added information about the `pgprot_noncached`() marcro in "PCI-X Memory Resource Address" on page 67.

Record of Revision

Version	Description
001	February 2003 Original publication
002	June 2003 Updated to support the SGI ProPack v2.2 for Linux release.
003	August 2003 Updated to support the SGI ProPack v2.2.1 for Linux release.
004	May 2004 Updated to support the SGI ProPack v3.0 for Linux release.
005	January 2005 Updated to support the SGI ProPack 3 for Linux Service Pack 3 release.
006	July 2005 Updated to support the SGI ProPack 3 for Linux Service Pack 6 release.

Contents

About This Guide . . . . . **xvii**
Related Resources xvii
Developer Program xvii
Internet Resources xvii
SGI Altix System Documentation xviii
Standards Documents xviii
Intel Compiler Documentation xix
Other Intel Documentation xix
Additional Reading xix
Obtaining Publications xx
Reader Comments xx

1. Introduction **1**
Legacy Functionality 2
Special Architectural Considerations 4
Programmable I/O Write Operations 4
Direct Memory Access 4
Device Interrupts and Posted DMAs 5
PIO Reads and Posted DMAs 5
Polling Memory for Completion and Posted DMA Data 5
Porting a Driver to Altix Systems — A Quick Start Guide 5
Device Driver Registration and Special Device Nodes 6
Itanium 2 Processors and Altix System Addresses 6
System Physical Memory Addresses 7

Bus Addresses - PCI/PCI-X Buses . . . 7
Programmable IO Read/Write Addresses . . . 8
Device Driver Interrupt Registration - IRQs . . . 9
Direct Memory Access Addresses (DMA) . . . 10
Posted PIO `write` Calls . . . 11

2. Architecture . . . 13
System Components . . . 13
Compute/Processor Node (SC-brick) . . . 16
PCI-X with BaseIO (IX-brick) . . . 18
PCI-X with Expansion (PX-brick) . . . 19
System Memory Address Space . . . 20
Physical Address Space . . . 21
Global MMR Space . . . 22
AMO Space . . . 23
Cacheable Memory Space . . . 25
Cache Use . . . 25
Cache Coherency . . . 26
SHub Physical Address Map . . . 26
PIO Addresses and DMA Addresses . . . 27
PIO Addressing . . . 29
PIO Addressing Extension . . . 29
DMA Addressing . . . 30
DMA Addressing Extension . . . 30
Linux Kernel and User Virtual Address Management . . . 30
Memory Access . . . 31
CPU Access to Memory or I/O Address Space . . . 31

CPU Access to Memory 31
CPU Access to I/O Address Space – Programmable I/O (PIO) 33
Device Access to System Physical Memory Space – Direct Memory Access 35

3. PCI-X Device Attachment 37
PX-brick with BaseIO (IX-brick) 37
PX-brick Expansion 38
PCI-X Implementation 40
Latency and Operation Order 40
Configuration Register Initialization 41
Unsupported PCI-X Signals 42
Address Spaces Supported 42
64-bit Address and Data Support 42
PIO Address Mapping 43
DMA Address Mapping 43
Bus Arbitration 44
Interrupt Signal Distribution 44

4. PCI System Initialization 47

5. Finding Your PCI Device 51
Physical Location of Your PCI Device 51
Logical Address of Your PCI Device 52
Physically Locating Your PCI Device Information 53
Programmatically Locating Your PCI Device 55

6. PCI/PCI-X Configuration Space 57

7. PCI-X I/O and Memory Resources 61

Anatomy of a PIO Address 61
PIO Addresses 61
Flow of PIO Operation 62
Targeting a PCI-X Device on a Local Node 62
Targeting a PCI-X Device on a Remote Node 64
PIO Address Translation from CPU to PCI Bus 65
PCI-X PIO Resource Management 66
PCI-X I/O Resource Address 66
PCI-X Memory Resource Address 67
PCI-X I/O Resource Reservation 68
PCI-X I/O Resource Use Macros 69
PCI-X Memory Resource Use Macros 70
PIO Write (Posted) Synchronization 70
PIO Read Flushing Posted DMA Buffers 73

8. PCI-X Interrupt Mechanism 75
Interrupt Architecture 75
Interrupt Request (IRQ) Management 75
Driver Interrupt Registration 75

9. PCI-X Direct Memory Access (DMA) 77
Types of DMA Mappings 79
Consistent DMA Mappings 79
Streaming DMA Mappings 79
Anatomy of a Mapped DMA Address 80
Format of 32-bit Direct Mapped DMA Addresses 80
Format of 32-bit DMA Page Mapped Addresses 82
Format of a 64-bit DMA Mapped Adddress 83

PCI-X DMA Address Management 83
PCI-X DMA Mapped Routines 84

10. Device Driver Memory Usage 87

Device Driver Memory Allocation 87
Allocating Page Boundary Memory 87
Allocating Page Boundary Memory on Specific Nodes 88
Allocating Byte-Range Memory 88
Accessing the User Memory Area 88
Disabling Validity Checking 90
Directly Mapping User Virtual Addresses 90

11. Time Management 93

Interval Timer Counter (ITC) 93
Delaying Execution — Short Delay 93
Delaying Execution — Long Delay 94

12. Building Linux Kernels and Modules 95

Default Configuration File 95
Building a New Linux Kernel 95
Booting Your New Linux Kernel 95
Rebuilding Modules 96
Downloading SGI Altix RPMs 97
Building New Modules 98

Appendix A. Memory Operation Ordering on SGI Altix Systems 101

Memory Ordering 101
Release Semantics 102

Acquire Semantics 103
Memory Fencing 105

Index 107

Figures

Figure 2-1 Links Between Bricks 15
Figure 2-2 SC-brick Block Diagram 17
Figure 2-3 IX-brick (PX-brick with BaseIO Card) 19
Figure 2-4 PX-brick - PCI-X Expansion Brick 20
Figure 2-5 Address Decoding for Physical Memory Access 21
Figure 2-6 Bit Values for Global MMR Space 23
Figure 2-7 Bit Values for AMO Space 24
Figure 2-8 SHub Physical Address Map 27
Figure 2-9 Device Access through a Bus Adapter 28
Figure 2-10 CPU Access to Memory 32
Figure 2-11 CPU Access to Device Registers (Programmable I/O) 34
Figure 2-12 Device Access to Memory 35
Figure 3-1 PX-brick with BaseIO 38
Figure 3-2 PX-brick PCI-X Expansion 39
Figure 3-3 PCI-X Implementation 40
Figure 5-1 Physical Address Components 52
Figure 6-1 PCI-X Configuration Space 58
Figure 7-1 PIO Address Format 62
Figure 7-2 PIO to a Local PCI-X Device 63
Figure 7-3 PIO to a Remote PCI-X Device 64
Figure 7-4 PIO Address from the CPU 65
Figure 9-1 DMA to Memory on A Local Node 77
Figure 9-2 DMA to Memory on A Remote Node 78

Figure 9-3 PCI Direct Mapped Register (one per PCI bridge) 81
Figure 9-4 32–bit Direct Mapped Address As Returned by the System 81
Figure 9-5 50-bit System Memory Address 81
Figure 9-6 32-bit DMA Mapped Address 82
Figure 9-7 64-bit DMA Mapped Address 83
Figure A-1 Release Semantics One-Directional Fence 103
Figure A-2 Acquire Semantics One-Directional Fence 104
Figure A-3 Two-dimensional Memory Fence (`mf`) 105

Tables

Table 3-1 Bandwidth Characteristics of the IX-brick 37
Table 3-2 Bandwidth Characteristics of the PX-brick 39
Table 3-3 Shared Interrupts 44
Table 7-1 Memory Locks 71
Table 7-2 Correct Memory Lock Usage 72

About This Guide

This guide describes the ways in which hardware devices are integrated into and controlled from an SGI Altix series system running the Linux operating system. This guide provides an overview of the unique elements of writing drivers for SGI systems, a description of the SGI Altix architecture, and a summary of the SGI Linux kernel resources.

To write a **process-level** driver, you must be an experienced C programmer with a thorough understanding of the use of Linux system services and, of course, detailed knowledge of the device to be managed.

To write a **kernel-level** driver, you must be an experienced C programmer who knows Linux system administration and who understands the concepts of Linux device management.

Related Resources

The resources listed in this section contain additional information that might be helpful.

Developer Program

Information and support are available through the SGI Developer Program. To join the program, contact the Developer Response Center at 800-770-3033 or e-mail devprogram@sgi.com.

Internet Resources

A great deal of useful material can be found on the Internet. Some starting points are in the following list.

http://docs.sgi.com	SGI technical manuals to read or download

Note: Make sure you search in the entire Technical Publications Library (TPL) to view Linux and Altix systems documentation.

http://www.pcisig.com	Home page of the PCI bus standardization organization

SGI Altix System Documentation

For additional information on SGI Altix system documentation, see the following:

- *SGI ProPack for Linux Start Here*

  Provides a comprehensive list of SGI Altix system hardware and software documentation

- *SUSE LINUX for SGI Altix Systems*

  Provides a comprehensive list of SGI Altix system hardware and software documentation

- http://docs.sgi.com

  SGI technical manuals to read or download

Note: Make sure you search in the entire Technical Publications Library (TPL) to view Linux and Altix systems documentation.

Standards Documents

The following documents are the official standard descriptions of buses:

- *PCI Local Bus Specification, Version 2.1*, available from the PCI Special Interest Group, P.O. Box 14070, Portland, OR 97214 (fax: 503-234-6762).
- *ANSI/IEEE standard 1014-1987* (VME Bus), available from IEEE Customer Service, 445 Hoes Lane, PO Box 1331, Piscataway, NJ 08855-1331.

Intel Compiler Documentation

Documentation for the Intel compilers is located on your system in the `/docs` directory of the directory tree where your compilers are installed. If you have installed the Intel compilers, the following documentation is available:

- *Intel C++ Compiler User's Guide* (`c_ug_lnx.pdf`)
- *Intel Fortran Compiler User's Guide* (`for_ug_lnx.pdf`)
- *Intel Fortran Programmer's Reference* (`for_prg.pdf`)
- *Intel Fortran Libraries Reference* (`for_lib.pdf`)

Other Intel Documentation

The following documents describe the Itanium (previously called "IA-64") architecture and other topics of interest:

- *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization*, available online at the following location:

http://developer.intel.com/design/itanium2/manuals/251110.htm

- *Intel Itanium Architecture Software Developer's Manual*, available online at the following location:

http://developer.intel.com/design/itanium/manuals/iiasdmanual.htm

- *Introduction to Itanium Architecture*, available online at the following location:

http://shale.intel.com/softwarecollege/CourseDetails.asp?courseID=13

(secure channel required)

Additional Reading

The following additional publications are referenced in this manual:

- David Mosberger and Stephane Eranian, *IA-64 Linux Kernel Design and Implementation*. Prentice Hall, http: //www.phptr.com. ISBN 0–13–061014–3.
- Alessandro Rubini and Jonathan Corbet, *Linux Device Drivers*. Second edition, June 2001. O'Reilly, 0–59600–008–1, order number: 0081. Also available at http://www.xml.com/ldd/chapter/book/index.html.
- Tom Shanley, *PCI-X System Architecture*. First edition, 2001. Mindshare Inc. Addison-Wesley, ISBN 0-2-1-72682-3.
- Tom Shanley and Don Anderson, *PCI System Architecture*. Third edition. Mindshare Inc.

Obtaining Publications

You can obtain SGI documentation as follows:

- See the SGI Technical Publications Library at http://docs.sgi.com. Various formats are available. This library contains the most recent and most comprehensive set of online books, release notes, man pages, and other information.
- If it is installed on your SGI system, you can use InfoSearch, an online tool that provides a more limited set of online books, release notes, and man pages. With an IRIX system, enter `infosearch` at a command line or select **Help > InfoSearch** from the Toolchest.
- On IRIX systems, you can view release notes by entering either `grelnotes` or `relnotes` at a command line.
- On Linux systems, you can view release notes on your system by accessing the `README.txt` file for the product. This is usually located in the `/usr/share/doc/`*productname* directory, although file locations may vary.
- You can view man pages by typing `man` *title* at a command line.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this publication, contact SGI. Be sure to include the title and document number of the publication with your comments. (Online, the document number is located in the

front matter of the publication. In printed publications, the document number is located at the bottom of each page.)

You can contact SGI in any of the following ways:

- Send e-mail to the following address:

  techpubs@sgi.com

- Use the Feedback option on the Technical Publications Library Web page:

  http://docs.sgi.com

- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system.

- Send mail to the following address:

  Technical Publications
  SGI
  1500 Crittenden Lane, M/S 535
  Mountain View, California 94043–1351

SGI values your comments and will respond to them promptly.

Introduction

This document provides a description of issues that affect Linux device drivers executing on SGI Altix series systems. SGI Altix systems use a global-address-space cache-coherent multiprocessor that can scale up to 512 processors in a cache-coherent domain. For more information on system components, see "System Components" on page 13.

This document does not provide a tutorial on how to write a Linux device driver on this or any other Linux platform, but assumes that you can write a Linux device driver. It provides information you need for porting a Linux device driver, rewriting a Linux device driver, or writing a new Linux device driver for this platform. If you have never written a Linux device driver, you should start with *Linux Device Drivers*, second edition. This book provides an excellent background, with many examples, for writing a Linux device driver. Similarly, the book, *IA-64 Linux Kernel Design and Implementation*, provides details on the implementation of IA-64 Linux on the Intel Itanium family of processors, which is the architecture on which the SGI Altix is based. Authors and publishers of these books are listed in the Preface (“About This Guide”).

Caution: Drivers developed by using the information contained in this guide are the responsibility of the user. SGI does not extend any warranty to devices not officially supported by SGI. For information on devices officially supported on SGI and the support terms associated with them, see your support agreement.

Topics discussed in this document are as follows:

- Chapter 2, "Architecture" on page 13
- Chapter 3, "PCI-X Device Attachment" on page 37
- Chapter 4, "PCI System Initialization " on page 47
- Chapter 5, "Finding Your PCI Device" on page 51
- Chapter 6, "PCI/PCI-X Configuration Space" on page 57
- Chapter 7, "PCI-X I/O and Memory Resources" on page 61
- Chapter 8, "PCI-X Interrupt Mechanism" on page 75
- Chapter 9, "PCI-X Direct Memory Access (DMA)" on page 77

- Chapter 10, "Device Driver Memory Usage" on page 87
- Chapter 11, "Time Management" on page 93
- Chapter 12, "Building Linux Kernels and Modules" on page 95
- Appendix A, "Memory Operation Ordering on SGI Altix Systems" on page 101

This chapter addresses the following topics:

- "Legacy Functionality" on page 2
- "Special Architectural Considerations" on page 4
- "Porting a Driver to Altix Systems — A Quick Start Guide" on page 5

Legacy Functionality

Certain “legacy” methods are available to device drivers on other Linux systems that SGI Altix systems do not support (for example, using legacy I/O port numbers 0 through 64K, reading and using peripheral component interconnect (PCI) configuration base address registers (BARs) or interrupt requests (IRQs) directly from the card’s configuration space, and so on). Drivers that use legacy methods are not portable and they will not execute correctly on SGI Altix systems.

The SGI Altix system does not always impose upon a Linux device driver the use of additional or different sets of Linux DKIs to function correctly on this platform. However, the SGI Altix system is a large, complex system, and for drivers to successfully invoke the full parallelism of the hardware and hence achieve optimal performance, it might well be necessary to invoke services and paradigms that are not available in the standard Linux DKI. **For specific information, see your SGI support representative.**

SGI Altix systems, which run Linux, provide the same I/O capabilities as the SGI Origin series systems, which run IRIX, except for the Intel processor and the little endian platform. The following list describes the legacy functionalities that are **not** available on the SGI Altix platform.

Legacy Functionality	**Description**
I/O ports	SGI Altix I/O subsystems do not support legacy I/O ports from either the Linux kernel or user level applications. If you use legacy I/O port numbers 0 –

	65K in I/O port macros such as `inb()` and `outb()`, the system will generate an exception.
Expansion ROM	SGI Altix systems do not read and execute basic input/output systems (BIOS) in expansion read-only memory (ROM), even if the ROM is present. Drivers and cards that depend on initialization by these BIOS might not function correctly on this platform. All initialization must be done by the drivers when the Linux kernel calls them to initialize.
RAM VGA video memory	SGI Altix systems do not support legacy video random access memory (RAM).
IRQs in PCI configuration space	SGI Altix systems provide IRQs greater than 256. The device driver cannot use the IRQ byte in the PCI configuration space. Device drivers are required to retrieve the IRQ number initialized by the kernel for that device in the `pci_dev` structure.
Base Address Registers	SGI Altix I/O subsystem PCI bridges cannot generate a "dual address" cycle for programmable I/O addresses on the PCI-X bus. As such, only 32 bits of the BARs can be initialized. However, the platform also requires a PIO address to be 64 bits wide. As such, the values in the BARs are not the same as the addresses that the device driver uses on the CPU. The addresses on the CPU have been mapped. Reading the BAR and using it in any I/O macros will cause an exception. PCI-X I/O and memory addresses for the devices are provided in the `pci_dev` structure. These values are already mapped and using them will correctly target the relevent PCI-X device.
Peripheral buses	The only peripheral buses that SGI Altix systems support are PCI-X buses. SGI Altix systems do not support traditional legacy I/O space such as I/O ports. PCI-X I/O resource space and memory resource space are presented to the device driver as uncached virtual

addresses. For more details, see Chapter 3, "PCI-X Device Attachment" on page 37.

Special Architectural Considerations

The following sections describe special architectural characteristics of SGI Altix systems.

Programmable I/O Write Operations

Programmable I/O (PIO) write operations on SGI Altix I/O subsystems can be cached in various components of the system, from the CPU to the PCI-X bridges. PIO write requests from the same CPU are guaranteed to be issued in program order. However, they are not synchronous. PIO write operations on this platform are posted. To guarantee that PIO write operations have completed, device drivers are required to push all prior PIO write operations out to the device by issuing a PIO read operation to the same controller after the last write operation before releasing a semaphore. This will prevent another CPU from acquiring the semaphore and having its PIO transactions complete before the previous holder of the semaphore.

PIO access and system memory access use different paths and hardware components on SGI Altix I/O subsystems. A `get/release` operation on a memory-based lock can complete before a PIO write request.

You are strongly advised to program device drivers to flush all relevent PIO write operations with a PIO read operation to the same controller prior to releasing the relevent memory-based locks.

PIO write operation caching is a performance feature. Making each PIO write operation synchronous incurs unneccessary performance penalty. Other Linux based platforms also require the device driver to explicitly execute PIO write flushing for correct operation.

Direct Memory Access

SGI Altix I/O subsystems provide support for posted direct memory access (DMA). With posted DMA capability, the host bridge can respond to the requester that the request is complete prior to actually transferring the data to target memory. This is a performance feature. DMA data is not guaranteed to arrive in memory “in-order”.

Device Interrupts and Posted DMAs

SGI Altix I/O subsystems use the interrupt mechanism to flush all posted DMA data to target memory. This is the only mechanism currently available to ensure that all posted DMAs are flushed into the target memory.

PIO Reads and Posted DMAs

PCI-X bridge chipsets on SGI Altix systems do not automatically flush Posted DMA writes on any PIO reads. For information regarding software flushing of posted DMA write buffers, see "PIO Read Flushing Posted DMA Buffers" on page 73.

Polling Memory for Completion and Posted DMA Data

On SGI Altix systems, direct memory access (DMA) data from controller cards to system memory is not guaranteed to arrive "in-order". If a device driver is polling a memory location for completion status and the completion status is the last DMA operation by the controller card, it is not guaranteed that all the prior DMA data will arrive in memory before the DMA completion status word. If you are polling memory for completion status, you must use the equivalent mapping routines for this "Completion Status". The equivalent mapping routine provides a DMA handle to flush all DMA data.

Using the appropriate mapping routine will ensure that all prior DMA data has arrived in memory before the "Completion Status" DMA data.

The Linux operating system provides two separate DMA mapping interfaces:

- Consistent mapping
- Streaming mapping

For information on these mapping interfaces, see "Types of DMA Mappings" on page 79.

Porting a Driver to Altix Systems — A Quick Start Guide

This section provides a summary of major items to consider when porting your driver to an SGI Altix system and Itanium 2 processors, as follows:

- "Device Driver Registration and Special Device Nodes" on page 6

- "Itanium 2 Processors and Altix System Addresses" on page 6
- "System Physical Memory Addresses" on page 7
- "Bus Addresses - PCI/PCI-X Buses" on page 7
- "Programmable IO Read/Write Addresses" on page 8
- "Device Driver Interrupt Registration - IRQs" on page 9
- "Direct Memory Access Addresses (DMA)" on page 10
- "Posted PIO `write` Calls" on page 11

Device Driver Registration and Special Device Nodes

If your Altix system has `devfs` configured and activated, you need to ensure that your calls to these functions:

```
register_chrdev()
unregister_chrdev()

register_blkdev()
unregister_blkdev()
```

can be conditionally compiled to call `devfs`, as follows:

```
devfs_mk_dir()
devfs_register()
devfs_unregister()
```

Itanium 2 Processors and Altix System Addresses

All addresses on an Altix system are 64 bits long . Drivers have to ensure that any structures that are allocated to store any addresses must be 64 bits long.

Note: It is very important to note that if you want to translate a virtual memory address into a bus address (DMA for the card), using the following macros for translation **WILL NOT** work:

```
bus_to_virt()
virt_to_bus()
```

An **Example** such as the following, will **not** work:

```
/* This will NOT work ... */
dmabuf = kmalloc(size, GFP_KERNEL);
writel(virt_to_bus(dmabuf), card's_dma_addr_reg);
```

For more information, see "Direct Memory Access Addresses (DMA)" on page 10

System Physical Memory Addresses

An SGI Altix system does not have system physical memory smaller than or equal to 32 bits. To the device driver, system physical memory addresses are always 64 bits long.

The following macros will provide proper translation from physical-to-virtual or virtual-to-physical:

```
phys_to_virt()
virt_to_phys()
```

Note: A system physical address is not the same as a bus address. Therefore, system physical addresses cannot be used by the card for DMA, as is (see "Bus Addresses - PCI/PCI-X Buses" on page 7).

Bus Addresses - PCI/PCI-X Buses

Bus addresses are addresses that allow the device to perform DMA operations from the card into system physical memory. An SGI Altix system supports either a 64–bit bus address or a 32–bit bus address. These bus addresses must be obtained from the various `pci_map_XXX()` routines. See the section on direct memory access ddresses

(DMA). Legacy macros like `virt_to_bus`() and `bus_to_virt`() do **not** provide the correct translation.

Note: On an Altix system, there is no way to translate a bus address to virtual address. Moreover, there are no reasons why a driver needs to translate a bus Address to a virtual Address. Drivers are responsible to save the the corresponding virtual address to the mapped DMA address. For more information, see "Direct Memory Access Addresses (DMA)" on page 10.

Programmable IO Read/Write Addresses

The following legacy routines do almost no work on Intel Itanium 2 platforms:

- `ioremap`() — Adds the IA64 uncache OFFset
- `iounmap`() — Does nothing
- `ioremap_nocache`() — Calls the `ioremap`() function

Drivers must use the IO addresses provided in the `pci_dev` structure for the device.

An **Example** such as the following, will **not** work:

```
/* This will not work .. */
pci_read_config_dword(pci_dev, PCI_BASE_ADDRESS_0, &ioaddr);
cards_regs = ioremap(ioaddr, 0x1000);
writel(0x60002,(cards_regs + (PCI_INT_CFG/PltfMsk)));
```

Base address registers in the PCI Configuration Space of a card cannot be used, as is, by the device driver for PIO. Device Drivers have to use addresses initialized in the `pci_dev` structure allocated by the system for that device via this routine, as follows:

```
pci_resource_start(dev,bar)
```

Other resource routines of interest are, as follows:

```
pci_resource_end(dev,bar)
pci_resource_flags(dev,bar)
pci_resource_len(dev,bar)
```

On an SGI Altix system, these addresses are 64 bits long, regardless of whether they are PCI IO or memory resources. PCI IO resource addresses can then be used in the following macros:

```
inb/inw/inl/outb/outw/outl
insb/insw/insl/outsb/outsw/outsl
```

Note: Hardcoded legacy addresses for example, IO Port Number 0x360, used in IN/OUT macros will not work, for example, `inb(0x360)`, and so on.

PCI memory resource addresses can then be used in the following macros:

```
readb/readw/readl/readq/writeb/writew/writel/writeq
```

Device Driver Interrupt Registration - IRQs

Device drivers register their interrupt handling routines by calling the following code:

```
int request_irq(unsigned int irq,
         void (*handler)(int, void *, struct pt_regs *),
         unsigned long irqflags,
         const char * devname,
         void *dev_id)
```

Of particular interest here is `irq` integer, which traditionally is the interrupt line in the PCI configuration space. Device drivers should **not** be reading this value from the PCI configuration space to get the `irq` value for the `request_irq`(). Instead, device drivers should use the `irq` number as allocated in the `pci_dev` structure by the Linux operating system, as follows:

```
pci_dev->irq
```

An **Example** such as the following, will **not** work:

```
/* This will not work .. */
pci_read_config_byte(pci_dev, PCI_INTERRUPT_LINE, &irq);
request_irq(irq, ...);
```

Direct Memory Access Addresses (DMA)

DMA addresses (bus addresses) on Altix system are either 64 bits or 32 its, nothing in-between. Requests for DMA addresses between 33 and 63 bits are given 32 bits DMA addresses.

Device drivers cannot use legacy macros, such as the following:

```
bus_to_virt()
virt_to_bus()
```

Before calling any of the DMA mapping routines, a device driver should query the system for the DMA address size that the platform supports, using the following:

```
pci_dma_supported()
```

By default, the `dma_mask` is set by the Linux operating system to be `0xffffffff`, which means 32 bits.

On an Altix system, there no calls to convert addresses from bus-to-virtual or virtual-to-bus. If the driver requires the corresponding virtual address of a bus address, it should save the virtual address.

Linux provides the following routines for mapping Virtual Address to DMA address:

```
pci_alloc_consistent()
pci_free_consistent()
pci_map_single()
pci_unmap_single()
pci_map_sg()
pci_unmap_sg()
pci_dma_sync_single()
pci_dma_sync_sg()
```

See `linux/Documentation/DMA-mapping.txt` for more details.

The `pci_alloc_consistent`() routine, by default, returns a 32 bit DMA address to the caller. On an Altix system, there is an exception. If your card is a PCIX card running in PCIX mode, only 64–bit DMA addresses are returned. For cards running in PCIX mode, please use the following: `pci_set_consistent_dma_mask`() to set the consistent mask bits to `0xffffffffffffffff`. Otherwise, your call to `pci_alloc_consistent`() will fail.

Posted PIO `write` Calls

For performance reasons, PIO write calls are posted. That is, on return from a PIO `write` call for example, `outb(X)`, an Altix system does not guarantee that the PIO has arrived and been received by the designated device. To ensure that a PIO write has actually been delivered and received by the designated device, device drivers are required to perform a PIO read to a safe register on the device, for example reading the vendor's identification, and so on:

```
outb(X);
outb(XX);
inb(safe register address);
```

Note: IO writes are delivered as soon as possible. In a ccNUMA architecture like used in an Altix system, if the system is very busy, a PIO write can be buffered by the IO chipsets.

The same rules apply to PIOs using the `readb`() family of macros.

For more information on synchronization issues regarding PIOs and memory references, see the *Linux Device Drivers Guide*.

Architecture

This chapter gives an overview of system components and the management of physical and virtual memory in SGI Altix series systems, which are based on the Itanium Processor Family (IPF) of processors. This chapter also provides background information to help you understand the limitations and special conventions used by some kernel functions.

The following main topics are covered in this chapter:

- "System Components" on page 13
- "System Memory Address Space" on page 20
- "Memory Access" on page 31

System Components

The SGI Altix servers are a family of multiprocessor distributed shared memory (DSM) computer systems. The SGI Altix systems use a global-address-space cache-coherent multiprocessor that can scale up to 512 processors in a cache-coherent domain. The processors are housed in a 3-U high brick called the SC-brick. The SC-brick contains two processor nodes. A processor node consists of two processors, each with 1.5- or 3-MB on-chip, private tertiary (L3) cache, connected to the scalable hub (SHub) ASIC via the front side bus (FSB). The SHub ASIC acts as a crossbar between the processors, local SDRAM memory, the network interface, and the I/O interface. Each processor node is interconnected by a NUMAlink 4 channel. The modularity of the DSM approach combines the advantages of low entry-level cost with global scalability in processors, memory, and I/O. The SGI Altix systems are based on the Intel Itanium 2 processor. The Intel Itanium 2 processor is a 64-bit processor that is initially offered at 900 MHz clock speed with a 1.5 MB L3 cache size.

The SGI Altix has a PCI-X-based I/O system. (For more details on PCI-X devices, see Chapter 3, "PCI-X Device Attachment" on page 37). The I/O components are housed in an I/O brick. Following are the two types of I/O bricks:

IX-brick	An IX-brick consists of six PCI-X buses. One slot is preloaded with the BaseIO card, plus a drive module containing a DVD-ROM and one or two system disks.
PX-brick	A PX-brick consists of six PCI-X buses, each with two PCI-X slots.

Figure 2-1 on page 15, shows the links between the various bricks of the SGI Altix system.

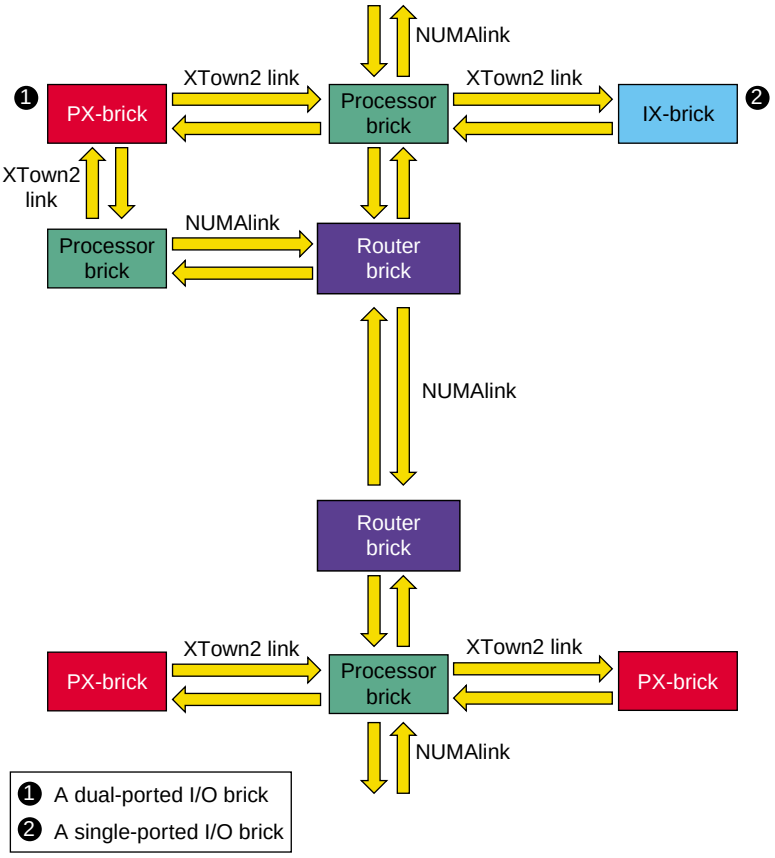

Figure 2-1 Links Between Bricks

The following sections provide additional information of the various system bricks. These sections describe the following system components:

- Compute/processor node (SC-brick)
- PCI-X with BaseIO (IX-brick)
- PCI-X with expansion (PX-brick)

Compute/Processor Node (SC-brick)

The SC-brick is a 3U (4.5″), 1U==1.5″, rackmountable enclosure that contains the following components:

- Two processor nodes, each containing two 64-bit processors with 1.5- or 3-MB secondary caches.
- Two SHub chipsets.
- Sixteen DIMM slots per SHub; one or two memory banks per four DIMMs.
- Node electronics.
- One L1 controller.

The node electronics, L1 controller, and power regulators are contained on a single half-panel power board (PCB). The two SHubs, four processors, and processor power pods are housed on separate half-panel boards. Four memory daughtercards house the memory DIMMs. Each daughtercard supports eight memory DIMMs. Figure 2-2 on page 17, shows the block diagram of an SC-brick.

Note: All transfer rates in Figure 2-2 on page 17, are peak rates.

The $ in Figure 2-2 on page 17, means “cache.”

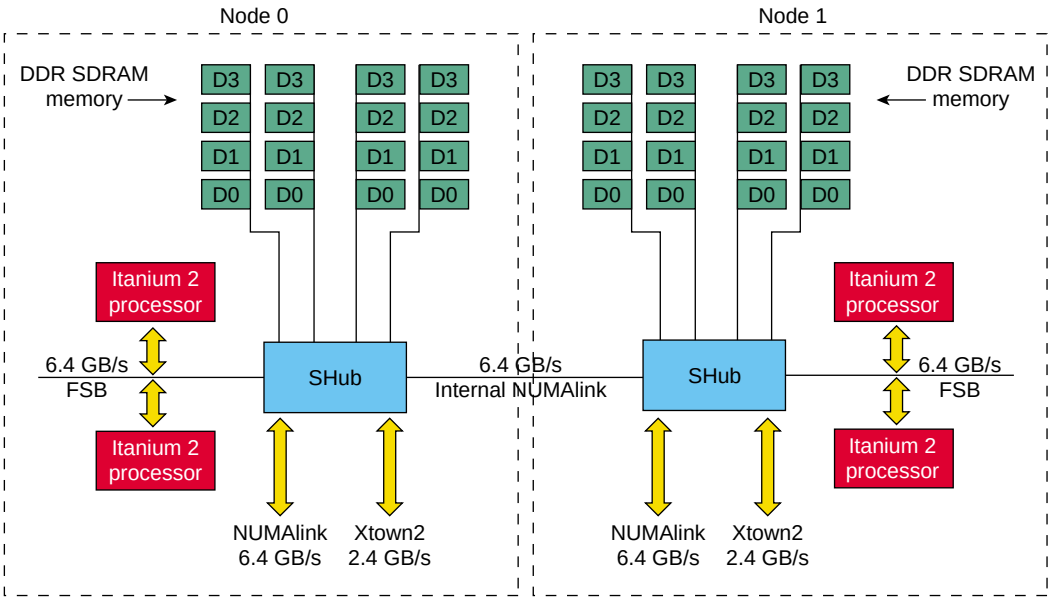

Figure 2-2 SC-brick Block Diagram

The SC-brick has the following features:

- Two 64-bit processors
- Contains one 1.5- or 3-MB secondary cache per processor (integrated within the processor)
- Configurable from 2.0 GB to 16 GB of main memory (minimum 8 DIMMs)
- Contains two 6.4-GB/s (each direction) NUMAlink channels
- Contains two 2.4-GB/s (each direction) Xtown2 channels
- Contains one connection port to the L2 controller
- Contains one DB9 console port

PCI-X with BaseIO (IX-brick)

The IX-brick is actually a PX-brick with a BaseIO card in PCI-X bus Q, slot Q, plus a drive module. The BaseIO card consists of the following components:

- IOC4 components:

  ATA bus connected to DVD-ROM
  NVRAM
  Real-time clock
  Real-time input/output ports
  Serial ports
  PS/2 keyboard and mouse ports

- Ethernet network chipset
- SCSI controller

Figure 2-3 on page 19, shows an IX-brick.

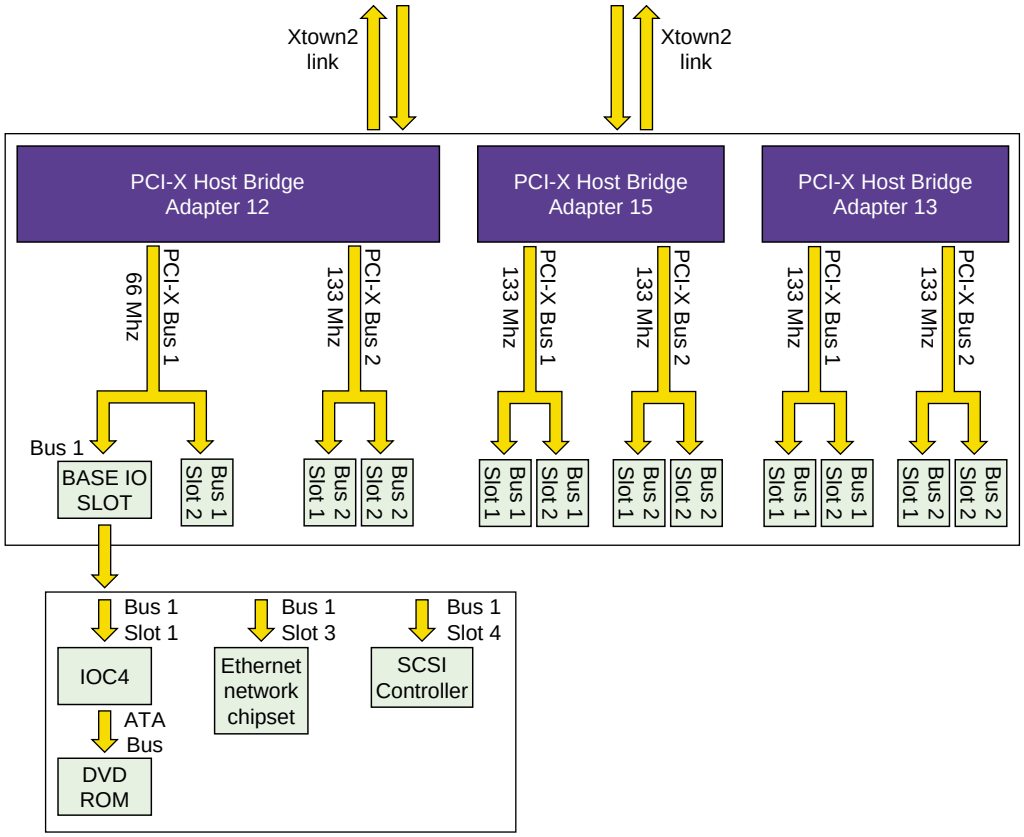

Figure 2-3 IX-brick (PX-brick with BaseIO Card)

PCI-X with Expansion (PX-brick)

The PX-brick contains six PCI-X buses with two slots per bus to make a total of 12 PCI-X slots. PX-bricks can be connected to the system via two Xtown2 links. The PX-brick PCI-X expansion is shown in Figure 2-4 on page 20.

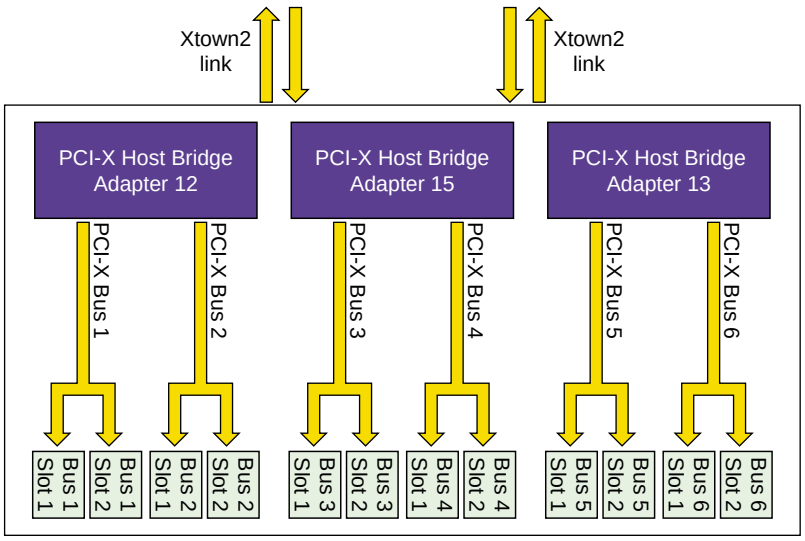

Figure 2-4 PX-brick - PCI-X Expansion Brick

System Memory Address Space

SGI Altix systems support 64-bit mode addressing. This section refers to the 64-bit address spaces provided by the SGI Altix system microprocessor (see Figure 2-8 on page 27). This architecture uses addresses that are 64-bit unsigned integers from 0x0000 0000 0000 0000 to 0xFFFF FFFF FFFF FFFF. This is an immense span of numbers—if it were drawn to a scale of 1 millimeter per terabyte, the drawing would be 16.8 kilometers long (just over 10 miles).

The following types of space are described in this section:

- Physical address
- Global Memory mapped register (MMR)
- Atomic memory operation (AMO)
- Cacheable memory
- SHub physical address map

Physical Address Space

This section provides physical address space information that is normally used by device drivers. SGI Altix systems support 50-bit physical addressing, as shown in Figure 2-5 on page 21.

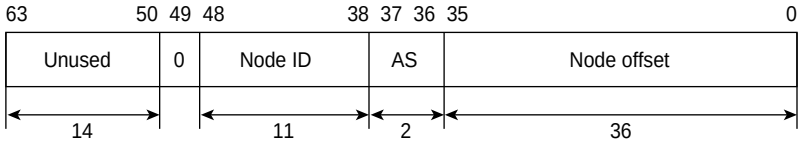

Figure 2-5 Address Decoding for Physical Memory Access

Fields in Figure 2-5 on page 21, are defined as follows:

Bits	Description
63:50	Unused and reserved for future use. The value of these bits should always be zero. This leaves 512 terabytes of addressing for SGI Altix systems implemented with SHub.
49:38	Node ID bits. SGI Altix systems implemented with SHub support up to 1024 processor nodes (2048 CPUs per system). Bit 38 indicates the node type. A value of 0 indicates a processor node. Bit 49 is always 0.
37: 36	Address space (AS). Each SHub is allocated 256 GB of physical address space. Bits 37:36 divide the 256 GB into four 64-GB spaces, as follows:

Bits [37:36]	Description
00	Local resource space and global MMR space
01	GET space
10	AMO space
11	Cacheable memory space

The AS bits are analogous to the uncached attribute bits of the SGI Origin series systems; however, since Itanium 2 processors do not support uncached attribute bits in the translation lookaside buffer (TLB), physical address bits are used to perform the equivalent function.

35:0 — Node offset. These bits point to a specific byte location within one of the four 64-GB spaces of the SHub. When the value of bits 37:36 is 0b00, the 64-GB local resource space and global MMR space is really split into two 32-GB regions: 32 GB of local resource space and 32 GB of global MMR space. Bit 35 selects between these two regions. When the value of bits 37:35 is 0b000, the request targets the local resource space. When the value of bits 37:35 is 0b001, the request targets the global MMR space.

The following sections describe global MMR space, AMO space, and cacheable memory space.

Global MMR Space

A node's global memory mapped register (MMR) space provides all processor nodes in the system with access to a node's MMRs (see Figure 2-6 on page 23). Notice the position of the global MMR space in the physical address map shown in Figure 2-8 on page 27. Following are the values of the bits for global MMR space:

Bit	Value
49	0
48:38	Node ID (remember, SHubs are even nodes)
37:36	00 (AS bits)
35	1

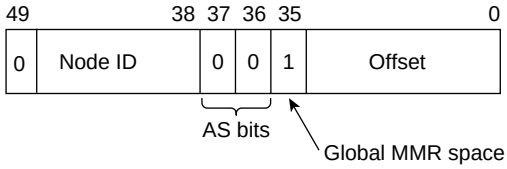

Figure 2-6 Bit Values for Global MMR Space

Note: Programmable I/O addresses reside in this space (for example, SHub systems, registers set, PCI configuration space, PCI I/O and memory space, I/O brick registers, and so on).

AMO Space

When the address space (AS) bits are set to 10, the reference is to atomic memory operation (AMO) space. An AMO read operation (`AMOR`) or AMO write operation (`AMOW`) request is issued to the SHub that is identified by the number in the node ID (see Figure 2-7 on page 24). Notice the position of the AMO space in the physical address map shown in Figure 2-8 on page 27. The node offset bits specify a 36-bit offset within the SHub address space, as follows:

Bit	Value
49	0
48:38	Node ID (remember, SHubs are even nodes)
37:36	10
35:0	Node offset

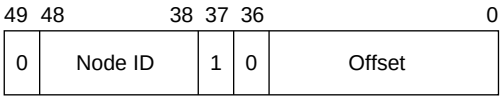

Figure 2-7 Bit Values for AMO Space

A number of fetch-and-op style AMOs are supported to optimize common synchronization primitives such as locks, tickets, and barriers. These AMOs operate on a read-modify-write basis. AMOs are defined only for word and doubleword data sizes and are performed using uncached loads and stores to the AMO address space. In addition, operations are allowed only on the first doubleword of each 64-byte block (half cache line) in memory. The AMO variable can be accessed either as one 64-bit AMO variable or as two 32-bit AMO variables.

In the AMO address space, bits 5:3 of the node offset (the three address bits above the doubleword offset) determine the type of AMO to perform.

The following AMO read operations are supported:

Fetch	Simple uncached read of the location.
Fetch and Increment	The location's current value is returned and then the location's value is incremented. This operation is followed by a write operation.
Fetch and Decrement	The location's current value is returned and then the location's value is decremented. This operation is followed by a write operation.
Fetch and Clear	The location's current value is returned and then the location's value is cleared. This operation is followed by a write operation.

The following AMO write operations are supported:

Initialize	Simple uncached write of the location.
Increment	The location's value is incremented.
Decrement	The location's value is decremented.
Logical AND	Stored data is logically AND'd with the location's current value.

Logical OR	Stored data is logically OR'd with the location's current value.

Cacheable Memory Space

When the AS bits are set to 11, the reference is to cacheable memory space. A memory request is issued to the SHub that is identified by the number in the node ID. The node offset bits specify a 36-bit offset within the SHub address space. UC, WB, and WC attributes are supported for cacheable memory space. Notice the position of the cacheable memory space in the physical address map shown in Figure 2-8 on page 27. The 50-bit physical address has a 36-bit offset within the SHub address space, as follows:

Bit	Value
49	0
48:38	Node ID (remember, SHubs are even nodes)
37:36	11 (AS bits)
35:0	Node offset

Note: Direct memory access (DMA) addresses reside in cacheable memory space.

Cache Use

The primary, secondary, and tertiary caches shown in Figure 2-10 on page 32, are essential to CPU performance. There is an order of magnitude difference in the speed of access between cache memory and main memory. Execution speed remains high only as long as a very high proportion of memory accesses are satisfied from the primary, secondary, or tertiary cache.

The use of caches means that there are often multiple copies of data: a copy in main memory, a copy in the secondary cache (when one is used), and a copy in the primary cache. Moreover, a multiprocessor system has multiple CPU modules like the one shown in Figure 2-10 on page 32, and there can be copies of the same data in the cache of each CPU.

Cache Coherency

The problem of *cache coherency* is to ensure that all cache copies of data are true reflections of the data in main memory. Different SGI systems use different hardware designs to achieve cache coherency.

Multiprocessor systems have more complex cache coherency protection because it is possible to have data in multiple caches. In an SGI Altix multiprocessor system, the hardware ensures that cache coherency is maintained under all conditions, including DMA input and output, without action by the software.

SHub Physical Address Map

Figure 2-8 on page 27 shows the SHub physical address map. On SHub, AMO space and global MMR space must be accessed uncached, and GET space must be accessed cached. Cacheable memory space can be accessed cached or uncached, subject to operating system constraints.

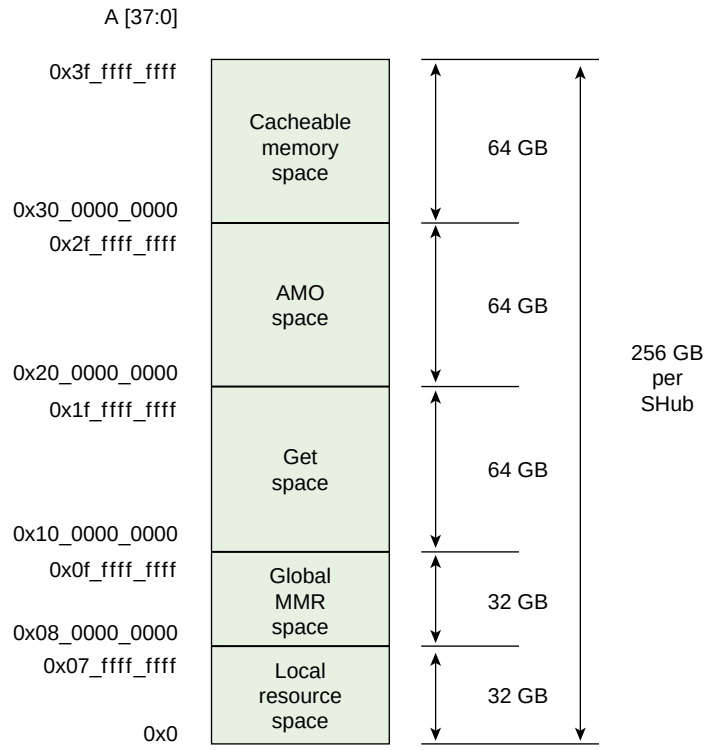

Figure 2-8 SHub Physical Address Map

Note: Linux drivers run in virtual mode (TLBs enabled for all addresses) all the time. Therefore, the address space they see depends not only on behavior of the SHub, but also on the TLB mapping conventions of the operating system.

PIO Addresses and DMA Addresses

Figure 2-12 on page 35, is too simple for some devices that are attached through a bus adapter. A bus adapter connects a bus of a different type to the system bus, as shown in Figure 2-9 on page 28.

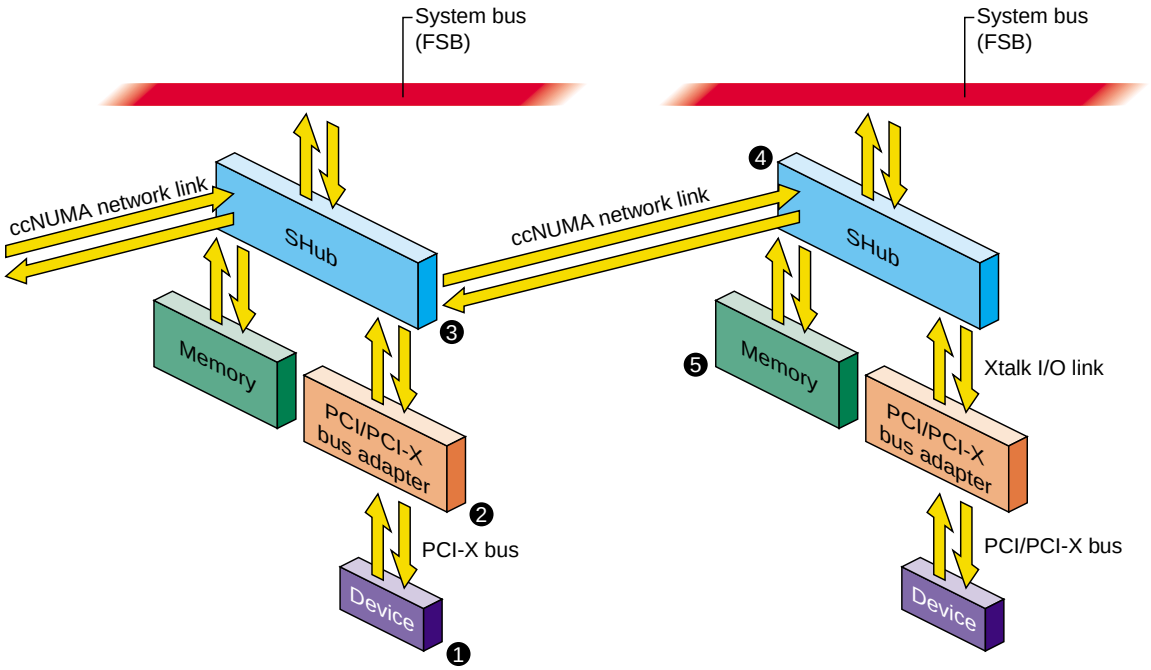

Figure 2-9 Device Access through a Bus Adapter

For example, the PCI/PCI-X bus adapter connects a PCI/PCI-X bus to the Xtalk I/O interface of SHub. Multiple PCI/PCI-X devices can be plugged into the PCI/PCI-X bus and use the bus to read and write. The bus adapter translates the PCI/PCI-X bus protocol into the system Xtalk protocol.

Each PCI/PCI-X bus has address lines that carry the address values used by devices on that PCI/PCI-X bus. These bus addresses are not related to the physical addresses used on the system front side bus (FSB). The issue of bus addressing is made complicated by three facts:

- Bus-master devices independently generate memory-read and memory-write commands that are intended to access system memory.
- The bus adapter can translate addresses between addresses on the bus it manages, and different addresses on the system bus it uses.

- The translation done by the bus adapter can be programmed dynamically (mapped), and can change from one I/O operation to another.

This subject can be simplified by dividing it into two distinct subjects: PIO addressing, used by the CPU to access a device, and DMA addressing, used by a bus master to access memory. These addressing modes need to be treated differently.

PIO Addressing

Programmable I/O (PIO) is the term for a load or store instruction executed by the CPU that names an I/O device space as its operand. The CPU places a physical address on the system bus. The bus adapter repeats the read or write command on its bus, but not necessarily using the same address bits as the CPU put on the system bus.

One task of a bus adapter is to translate between the physical addresses used on the system bus and the addressing scheme used within the proprietary bus. The address placed on the target bus is not necessarily the same as the address generated by the CPU. The translation is done differently with different bus adapters and in different system models.

With the more sophisticated PCI and PCI-X buses, the translation is dynamic. Both of these buses support bus address spaces that are as large or larger than the physical address space of the system bus. It is impossible to hard-wire a translation of the entire bus address space. Furthermore, SGI Altix architecture provides multiple system buses. For more details, see "Address Spaces Supported" on page 42.

The PCI/PCI-X resource addresses in the `pci_dev` structure are PIO mapped addresses that the device driver can use in their existing state.

PIO Addressing Extension

To use a dynamic PIO address, a device driver can create a software object called a PIO map that represents that portion of bus address space that contains the device registers the driver uses. When the driver wants to use the PIO map, the kernel dynamically sets up a translation from an unused part of physical address space to the needed part of the bus address space. The driver extracts an address from the PIO map and uses it as the base for accessing the device registers. This is an extension that SGI provides.

DMA Addressing

A bus-master device on the PCI bus can be programmed to perform transfers to or from memory independently and asynchronously. A bus master is programmed using PIOs with a starting bus address and a length. The bus master generates a series of memory-read or memory-write operations to successive addresses. But what bus addresses should it use in order to store into the proper memory addresses?

The bus adapter translates the addresses used on the proprietary bus to corresponding addresses on the system bus. As shown in Figure 2-9 on page 28, the operation of a DMA device is as follows:

1. The device places a bus address and data on the PCI or PCI-X bus.
2. The bus adapter translates the address to a meaningful physical address, and places that address and the data on the system Xtalk I/O link.
3. The memory modules store the data.

The translation of bus virtual to physical addresses is done by the bus adapter and programmed by the kernel. A device driver requests the kernel to set up a dynamic mapping from a designated memory buffer to bus addresses. For more information, see Chapter 9, "PCI-X Direct Memory Access (DMA)" on page 77.

Linux device drivers on SGI Altix systems must use the standard Linux `pci_dma` map routines. For more information, see Chapter 9, "PCI-X Direct Memory Access (DMA)" on page 77.

DMA Addressing Extension

The driver calls kernel functions to establish the range of memory addresses that the bus master device will need to access—typically the address of an I/O buffer. When the driver calls one of the `pci_dma` map routines, the kernel sets up the bus adapter hardware to translate between some range of bus addresses and the desired range of memory space. The driver uses PIO to program this bus address into the bus master device registers. SGI software supports 64- and 32-bit DMA addresses. For more information on 64- and 32-bit DMA map addresses, see Chapter 9, "PCI-X Direct Memory Access (DMA)" on page 77.

Linux Kernel and User Virtual Address Management

The SGI Altix system uses the same virtual memory manager as any IA-64 Linux system with ccNUMA and discontiguous memory support. For more information on

Linux kernel and user virtual address management, see *IA-64 Linux Kernel Design and Implementation*.

Memory Access

The following sections describe CPU and device access to memory.

CPU Access to Memory or I/O Address Space

Each SGI computer system has one or more CPU modules and one or more I/O modules. A CPU reads data from memory or a device by placing an address on a system bus and receiving data back from the addressed memory or device. An address can be translated more than once as it passes through multiple layers of I/O chipsets and bus adapters. Access to memory can also pass through multiple levels of cache.

CPU Access to Memory

The CPU generates the address of data that it needs—the address of an instruction to fetch, or the address of an operand of an instruction. It requests the data through a mechanism that is depicted in simplified form in Figure 2-10 on page 32.

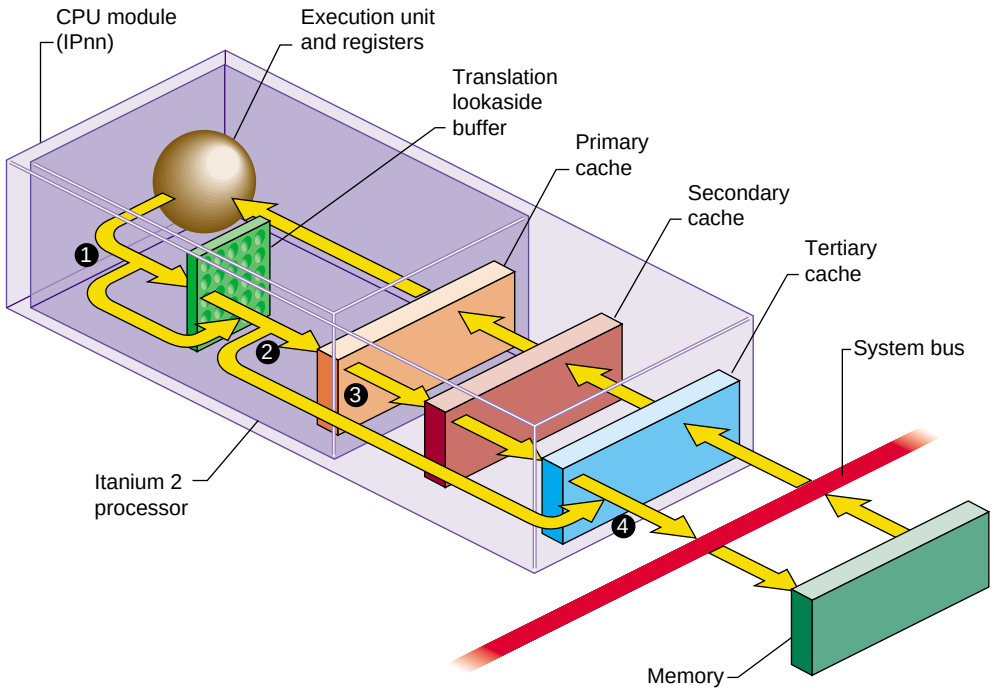

Figure 2-10 CPU Access to Memory

The process is as follows:

1. The address of the needed data is formed in the processor execution or instruction-fetch unit. Most addresses are then mapped from virtual to real through the translation lookaside buffer (TLB). On Itanium 2 processors, all addresses go through the TLBs if TLBs are enabled. With some very small exceptions, TLBs are always enabled.

2. Most addresses are presented to the L1 cache, a cache in the processor chip. If a copy of the data with that address is found, it is returned immediately. Certain address ranges are never cached; these addresses pass directly to the bus.

3. If the L1 cache does not contain the data, the address is presented to the L2 cache. If it contains a copy of the data, the data is returned immediately. The size and the architecture of the secondary cache differ from one CPU model to another.

4. If L2 does not contain the data, the address is presented to the L3 cache. The address is placed on the system bus. The memory module that recognizes the address places the data on the bus.

The process in Figure 2-10 on page 32 is correct for an SGI Altix system when the addressed data is in the local node.

Note: When the address applies to memory in another node, the address passes out through the connection fabric to a memory module in another node, from which the data is returned.

CPU Access to I/O Address Space – Programmable I/O (PIO)

The CPU accesses a device register using *programmable I/O* (PIO), a process illustrated in Figure 2-11 on page 34. Access to device registers is always uncached. It is not affected by considerations of memory cache coherency in any system (see "Cache Use" on page 25).

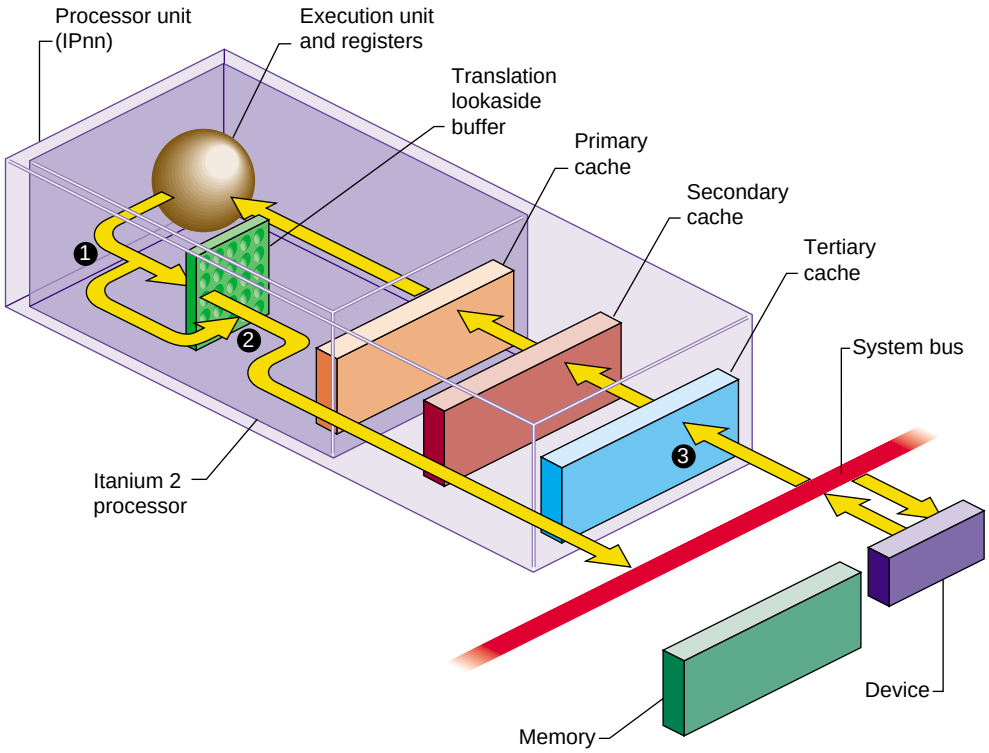

Figure 2-11 CPU Access to Device Registers (Programmable I/O)

The process is as follows:

1. The address of the device is formed in the execution unit. It is not usually an address that is mapped by the TLB.
2. A device address, after mapping if necessary, always falls in one of the ranges that is not cached, so it passes directly to the system bus.
3. The device or system component (such as SHub) recognizes its physical address and responds with data.

The PIO process shown in Figure 2-11 on page 34, is correct for an SGI Altix system when the addressed device is attached to the same node. When the device is attached to a different node, the address passes through the connection fabric to that node, and the data returns the same way.

Device Access to System Physical Memory Space – Direct Memory Access

Some devices can perform *direct memory access* (DMA), in which the device itself, not the CPU, reads or writes data into memory. A device that can perform DMA is called a *bus master* because it independently generates a sequence of bus accesses without help from the CPU.

To read or write a sequence of memory addresses, the bus master has to be told the proper physical address (bus address) range to use. This is done by using PIO to store a bus address and length into the device's registers from the CPU. When the device has the DMA information, it can access memory through the system bus as shown in Figure 2-12 on page 35.

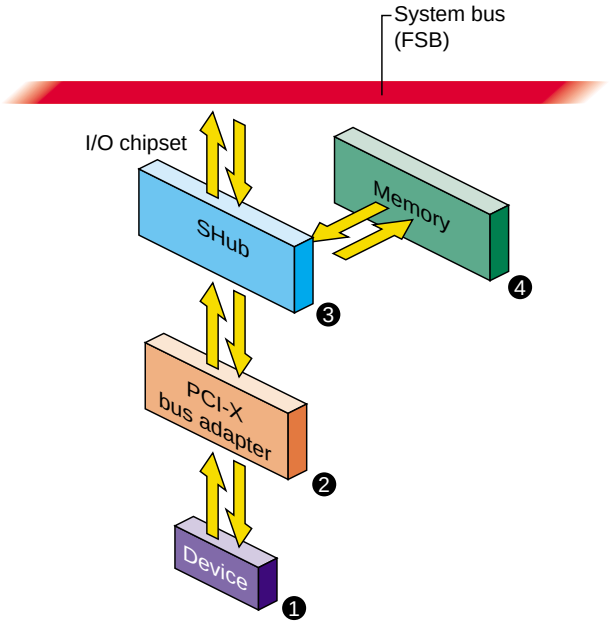

Figure 2-12 Device Access to Memory

The process is as follows:

1. The device makes a request on the PCI/PCI-X bus.

2. The PCI/PCI-X bus adapter translates the PCI/PCI-X bus request and generates a request to the I/O chipset (SHub).
3. The local SHub forwards the request to the requested memory controllers (local or remote).
4. The memory module stores the data.

In an SGI Altix system, the device and the memory module can be in different nodes, with address and data passing through the connection fabric (NUMAlink) between nodes.

When a device is programmed with an invalid physical address, the result is a bus error interrupt. The interrupt occurs on some CPU that is enabled for bus error interrupts. These interrupts are not simple to process for two reasons. First, the CPU that receives the interrupt is not necessarily the CPU from which the DMA operation was programmed. Second, the bus error can occur a long time after the operation was initiated.

PCI-X Device Attachment

The peripheral component interconnect (PCI) bus, initially designed at Intel, is standardized by the PCI special interest group, a nonprofit consortium of vendors. PCI-X is the successor to PCI. Both PCI and PCI-X devices can be used on a PCI-X bus. SGI Altix system architecture supports the PCI-X bus. All peripheral devices on SGI Altix systems are connected via PCI-X buses. The PCI-X bus is designed as a high-performance local bus to connect peripherals to memory and a microprocessor.

This chapter describes PCI-X implementation in larger architectures such as the SGI Altix systems.

For more information about PCI-X system architecture, see *PCI-X System Architecture*.

PX-brick with BaseIO (IX-brick)

The PX-brick with BaseIO is a Crosstalk-to-PCI-X based I/O subsystem. It has two 1200-MB/s Xtown2 connectors and one or two of them can be used to connect to SC-bricks. There are 12 PCI slots that are configured on 6 buses and 2 hard disk bays that support SCSI disk drives and a DVD-ROM. The DVD-ROM is not SCSI; it is connected through parallel ATA. For peak bandwidth values, see Table 3-1 on page 37. Figure 3-1 on page 38, depicts the PX-brick with BaseIO.

Table 3-1 Bandwidth Characteristics of the IX-brick

Description	Peak Bandwidth
Xtown2 ports — A and B	1200 MB/s
PCI-X bus frequency:	*64–bit mode:*
33 MHz	256 MB/s
66 MHz	512 MB/s
100 MHz	800 MB/s
133 MHz	1024 MB/s

Note 1: PCI-X mode achieves a higher percentage of theoretical peak versus PCI mode.

Note 2: To run the bus at 133 MHz requires only one card on that bus and it must be a 133-MHz capable card.

Note3: The IO9 is a 66-MHz PCI card, so bus 1 runs at 66 MHz PCI in the IX-brick.

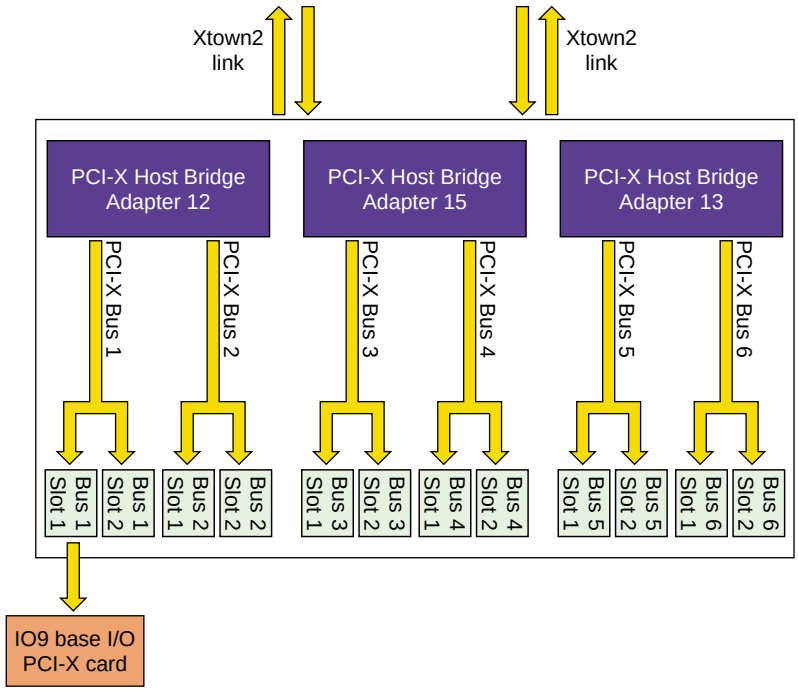

Figure 3-1 PX-brick with BaseIO

PX-brick Expansion

The PX-brick has two 1200-MB/s Xtown2 ports that connect SC-bricks. There are 12 PCI slots that are configured on 6 buses. For peak and sustained bandwidth values, see Table 3-2 on page 39. Figure 3-2 on page 39, depicts a PX-brick PCI-X expansion.

Table 3-2 Bandwidth Characteristics of the PX-brick

Description	Peak Bandwidth
Xtown2 ports — A and B	1200 MB/s
PCI-X bus frequency:	*64–bit mode:*
33 MHz	256 MB/s
66 MHz	512 MB/s
100 MHz	800 MB/s
133 MHz	1024 MB/s

Note 1: PCI-X mode achieves a higher percentage of theoretical peak versus PCI mode.

Note 2: To run the bus at 133 MHz requires only one card on that bus and it must be a 133-MHz capable card.

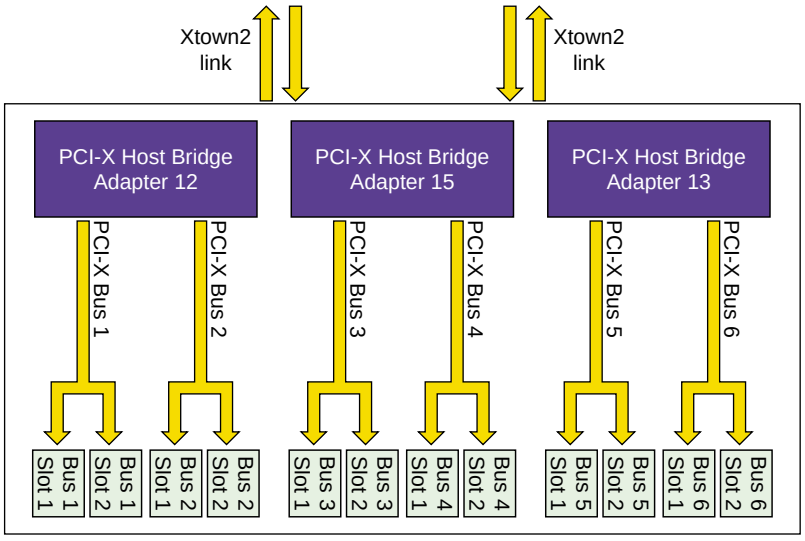

Figure 3-2 PX-brick PCI-X Expansion

PCI-X Implementation

In SGI Altix systems, the PCI-X adapter connects to the high-speed XIO bus. This bridge joins the PCI-X bus into the connection fabric, so that any PCI-X bus can be addressed from any module, and any PCI-X bus can access memory that is physically located in any module, as shown in Figure 3-3 on page 40.

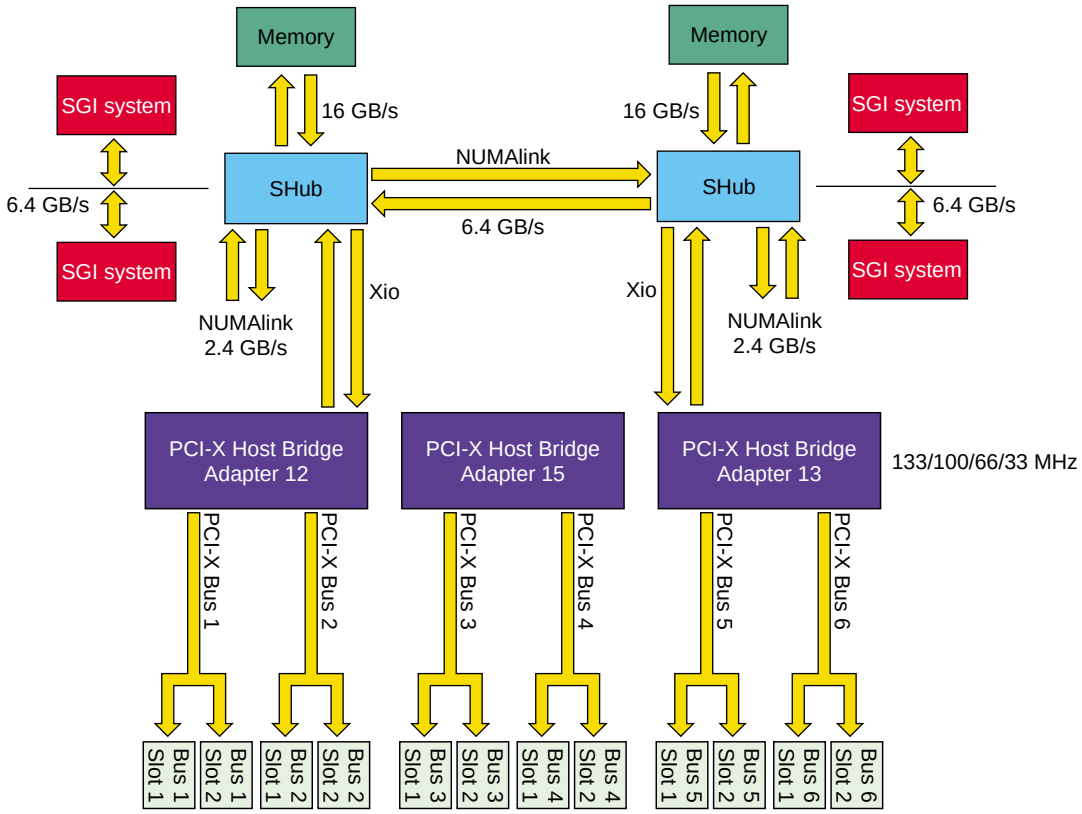

Figure 3-3 PCI-X Implementation

Latency and Operation Order

In SGI Altix systems, the multimedia features have substantial local resources, so contention with multimedia for the use of main memory is lower. However, these

systems also have multiple CPUs and multiple layers of address translation, and these factors can introduce latencies in PCI-X transactions.

It is important to understand that there is **no guaranteed order of execution between separate PCI-X transactions in these systems**. There can be multiple hardware layers between the CPU, memory, and the device. One or more data transactions can be "in flight" for durations that are significant. For example, suppose that a PCI-X bus-master device completes the last transfer of a DMA write of data to memory, and then executes a DMA write to update a status flag elsewhere in memory.

Under circumstances that are unusual but not impossible, the status in memory can be updated and acted upon by software, while the data transaction is still "in flight" and has not completely arrived in memory. The same can be true of a programmable I/O (PIO) read that polls the device. It can return "complete" status from the device while data sent by DMA has yet to reach memory.

Ordering is guaranteed when interrupts are used. An interrupt handler is not executed until all writes initiated by the interrupting device have completed.

Configuration Register Initialization

When the Linux kernel scans the PCI-X buses on an SGI Altix system and finds an active device, it initializes the device configuration registers as follows:

Command register	The enabling bits for I/O access, memory access, and master are set to 1. Other bits, such as memory write, invalidate, and fast back-to-back are left at 0.
Cache line size	Set at 0x20 (32, 32-bit words, or 128 bytes).
Latency timer	Setting depends on bus speed and the device's `Min_Gnt` (minimum grant) register. If the device's `Min_Gnt` value is 0, the latency timer is set to 1 microsecond. Otherwise, it is set to (`min_gnt_mult * Min_Gnt`). Values for `min_gnt_mult`depend on bus speed, as follows:

Bus Speed	`min_gnt_mult` **Value**
133 MHz	32
100 MHz	24
66 MHz	16
33 MHz	8

Base address registers	Each register that requests PCI memory or PCI I/O address space is programmed with a starting address.

Note: This address is valid only for this PCI-X bus.

When attaching a device, the device driver can set any other configuration parameters.

Caution: If the driver changes the contents of a base address register, the results are unpredictable. Do not do this.

Unsupported PCI-X Signals

The following optional signal lines are not supported:

- The LOCK# signal is ignored; atomic access to memory is not supported.
- The cache-snoop signals SBO# and SDONE are ignored. Cache coherency is ensured by the PCI-X adapter and the memory architecture, with assistance by the driver.

Address Spaces Supported

In SGI Altix systems, addresses are translated not once but at least twice and sometimes more often between the CPU and the device, or between the device and memory. Also, some of the logic for features, such as prefetching and byte-swapping, is controlled by the use of high-order address bits. There is no simple function on a physical memory address that yields a PCI-X bus address (nor vice-versa). The device driver must use the PIO addresses presented in the `pci-dev` structure. For more information, see Chapter 7, "PCI-X I/O and Memory Resources" on page 61.

It is also necessary for the device driver to call the relevant DMA mapping routines for DMA addresses (PCI-X bus addresses). For more information, see Chapter 9, "PCI-X Direct Memory Access (DMA)" on page 77.

64-bit Address and Data Support

SGI Altix systems support 64-bit data transactions. Use of 64-bit data transactions results in best performance. The PCI-X adapter accepts 64-bit addresses produced by a bus-master device. The PCI-X adapter does not generate 64-bit addresses itself.

(This is because the PCI-X adapter generates addresses only to implement PIO transactions, and PIO targets are always located in 32-bit addresses).

In SGI Altix systems, the PCI-X adapter implements a standard, 64-bit PCI-X bus operating as follows:

- One 133-MHz PCI-X card in one slot of the bus with the other slot empty and configured down.
- Two 100-MHz PCI-X cards or two 66-MHz PCI-X cards on one bus.
- Two 66-MHz PCI cards or two 33-MHz PCI cards on one bus.
- For mixed-speed cards, the PCI-X bus will downgrade to the lowest card speed.

PIO Address Mapping

For PIO purposes, memory space defined by each PCI-X device in its configuration registers is allocated in the lowest gigabyte of PCI-X address space, below 0x400 0000. These addresses are allocated dynamically, based on the contents of the configuration registers of active devices. The I/O address space requested by each PCI-X device in its configuration registers is also allocated dynamically as the system comes up. For further information on PIO address use, see Chapter 7, "PCI-X I/O and Memory Resources" on page 61.

DMA Address Mapping

Any part of physical address space can be mapped into PCI-X bus address space for purposes of DMA access from a PCI-X bus-master device. The SGI Altix system architecture uses a 50-bit physical address, of which some bits designate a node board. The PCI-X adapter sets up a translation between an address in PCI-X memory space and a physical address, which can refer to a different node from the one to which the PCI-X bus is attached.

The device driver ensures correct mapping through the use of PCI DMA map routines.

If the PCI-X device supports only 32-bit addresses, DMA addresses can be established in 32-bit PCI-X space. When this is requested, extra mapping hardware is used to map a window of 32-bit space into the 50-bit memory space.

Caution: The number of mapping registers is limited, so it is possible that a request for DMA translation could fail.

Because of the possibility of the failure of a DMA translation request, it is preferable to use 64-bit DMA mapping when the device supports it. When the device supports 64-bit PCI-X bus addresses for DMA, the PCI-X adapter can use a simpler mapping method from a 64-bit address into the target 50-bit address, and there is no contention for mapping hardware. The device driver must request a 64-bit DMA map, and must program the device with 64-bit values. For further information on DMA mapping, see Chapter 9, "PCI-X Direct Memory Access (DMA)" on page 77.

Bus Arbitration

The PCI-X adapter maintains two priority groups, the real-time group and the low-priority group. Both groups are arbitrated in round-robin style. Devices in the real-time group always have priority for use of the bus. There is no kernel interface for changing the priority of a device.

Interrupt Signal Distribution

Each PCI-X bus contains two unique interrupt signals. The `INTA#` and `INTC#` signals are wired together, and the `INTB#` and `INTD#` signals are wired together. A PCI-X device that uses two distinct signals must use `INTA#` and `INTB#`, or `INTC#` and `INTD#`. A device that needs more than two signals can use the additional signal lines, but such a device must also provide a register from which the device driver can learn the cause of the interrupt.

The PCI-X bus adapter chip that is used on all SGI Altix systems has eight input interrupts. PCI-X cards, however, can implement up to four different interrupts (A, B, C, and D), which might create a shared condition. Table 3-3 on page 44, shows how interrupts can be shared on an SGI Altix system.

Table 3-3 Shared Interrupts

PCI-X slots	PCI–X Interrupt line A	PCI-X Interrupt line B	PCI-X Interrupt line C	PCI-X Interrupt line D
Slot 0	0	4	0	4
Slot 1	1	5	1	5
Slot 2	2	6	2	6

PCI-X slots	PCI–X Interrupt line A	PCI-X Interrupt line B	PCI-X Interrupt line C	PCI-X Interrupt line D
Slot 3	3	7	3	7
Slot 4	4	0	4	0
Slot 5	5	1	5	1
Slot 6	6	2	6	2
Slot 7	7	3	7	3

For example, if a card in slot 0 uses `INTA#` and a card in slot 4 uses `INTB#`, there will be a conflict. In this case, the interrupt service routines (ISRs) of both cards will be called when the bridge interrupt pin 0 changes to active. If you try to connect to all four interrupt lines from the card, you will create a shared condition. This is called interrupt overloading.

Because SGI Altix systems support two slots and line A and line C are wired together and line B and line D are wired together, devices on the same bus will never share the same interrupt line.

PCI System Initialization

As part of the Linux kernel boot process, all PCI buses and devices are scanned, configuration space is initialized, and kernel data structures are created to map to these devices. This process is initiated well before any PCI device drivers are called to initialize their devices.

The Linux kernel creates a data structure to map each discovered PCI bus and a PCI device data structure to map each PCI device on that PCI bus. The following structures are examples of a `pci_bus` structure created for each PCI bus (Example 4-1, page 47) and a `pci_dev` structure created for each PCI device (Example 4-2, page 48), respectively.

Example 4-1 `pci_bus` structure created for each PCI bus

```
struct pci_bus {
        struct list_head node;          /* node in list of buses */
        struct pci_bus  *parent;        /* parent bus this bridge is on */
        struct list_head children;      /* list of child buses */
        struct list_head devices;       /* list of devices on this bus */
        struct pci_dev  *self;          /* bridge device as seen by parent */
        struct resource *resource[4];   /* address space routed to this bus */

        struct pci_ops  *ops;           /* configuration access functions */
        void            *sysdata;       /* hook for sys-specific extension */
        struct proc_dir_entry *procdir; /* directory entry in /proc/bus/pci */

        unsigned char   number;         /* bus number */
        unsigned char   primary;        /* number of primary bridge */
        unsigned char   secondary;      /* number of secondary bridge */
        unsigned char   subordinate;    /* max number of subordinate buses */

        char            name[48];
        unsigned short  vendor;
        unsigned short  device;
        unsigned int    serial;         /* serial number */
```

```
        unsigned char   pnpver;         /* Plug & Play version */
        unsigned char   productver;     /* product version */
        unsigned char   checksum;       /* if zero - checksum passed */
        unsigned char   pad1;
};
```

Example 4-2 `pci_dev` structure created for each PCI device

```
struct pci_dev {
        struct list_head global_list;   /* node in list of all PCI devices */
        struct list_head bus_list;      /* node in per-bus list */
        struct pci_bus  *bus;           /* bus this device is on */
        struct pci_bus  *subordinate;   /* bus this device bridges to */

        void            *sysdata;       /* hook for sys-specific extension */
        struct proc_dir_entry *procent; /* device entry in /proc/bus/pci */

        unsigned int    devfn;          /* encoded device & function index */
        unsigned short  vendor;
        unsigned short  device;
        unsigned short  subsystem_vendor;
        unsigned short  subsystem_device;
        unsigned int    class;          /* 3 bytes: (base,sub,prog-if) */
        u8              hdr_type;       /* PCI header type (‘multi’ flag masked out) */
        u8              rom_base_reg;   /* which config register controls the ROM */

        struct pci_driver *driver;      /* which driver has allocated this device */
        void            *driver_data;   /* data private to the driver */
        u64             dma_mask;       /* Mask of the bits of bus address this
                                           device implements.  Normally this is
                                           0xffffffff.  You only need to change
                                           this if your device has broken DMA
                                           or supports 64-bit transfers.  */

        u32             current_state;  /* Current operating state. In ACPI-speak,
                                           this is D0-D3, D0 being fully functional,
                                           and D3 being off. */

        /* device is compatible with these IDs */
        unsigned short vendor_compatible[DEVICE_COUNT_COMPATIBLE];
        unsigned short device_compatible[DEVICE_COUNT_COMPATIBLE];
```

```
        /*
         * Instead of touching interrupt line and base address registers
         * directly, use the values stored here. They might be different!
         */
        unsigned int    irq;
        struct resource resource[DEVICE_COUNT_RESOURCE]; /* I/O, memory, ROMS regions*/
        struct resource dma_resource[DEVICE_COUNT_DMA];
        struct resource irq_resource[DEVICE_COUNT_IRQ];

        char            name[80];       /* device name */
        char            slot_name[8];   /* slot name */
        int             active;         /* ISAPnP: device is active */
        int             ro;             /* ISAPnP: read only */
        unsigned short  regs;           /* ISAPnP: supported registers */

        int (*prepare)(struct pci_dev *dev);    /* ISAPnP hooks */
        int (*activate)(struct pci_dev *dev);
        int (*deactivate)(struct pci_dev *dev);
};
```

A PCI bus is uniquely identified by a PCI bus number (`pci_`*dev.number*) A PCI device is uniquely identified by its device and function number (`pci_`*dev.devfn*) and the bus it is on (`pci_`*dev.bus->number*).

As you can observe from the data structures in Example 4-1, page 47, and Example 4-2, page 48, the PCI buses are linked together and each `pci_bus` structure is also linked to its own PCI devices. All the PCI devices are also linked.

At the end of the system PCI initialization, the Linux kernel has the following initialization status:

- Software representation of each PCI physical bus
- Software representation of each PCI device
- Any required initialization of the PCI configuration space registers, such as base address registers, and so on

Finding Your PCI Device

The SGI Altix system can support many more PCI-X I/O devices than other traditional Linux servers. Each PX-brick has 6 PCI-X buses and 12 slots..

When you have more than one PX-brick on your SGI Altix system, you must be able to physically and programmatically locate your devices. This chapter provides the information about physical and logical Linux addresses that you need to find your PCI-X device.

Physical Location of Your PCI Device

When your PCI device is slotted into a PCI-X slot, it has a physical address. The components of the physical address are as follows:

- Function number (0 to 7)
- Physical device/slot number on the PCI-X bus (1 or 2)
- PCI-X bus number on the host bridge adapter (1 or 2)
- Host bridge adapter number (also known as widgets) (12, 13, or 15)
- Compute brick slab number (0 or 1)
- Compute brick module identification number
- PX-brick/IX-brick module identification number

Figure 5-1 on page 52, shows physical address components.

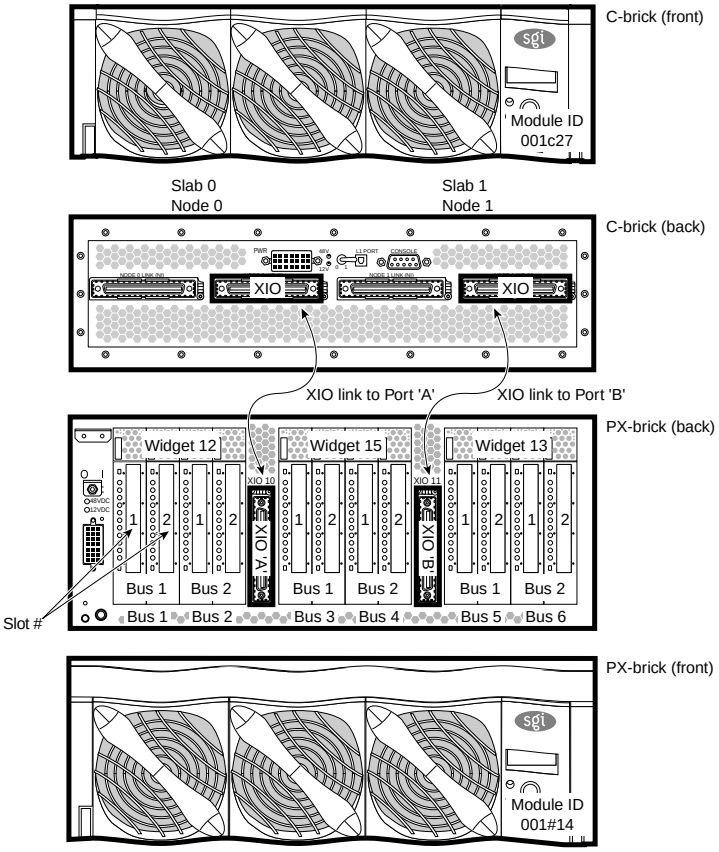

Figure 5-1 Physical Address Components

Logical Address of Your PCI Device

When the Linux PCI subsystem encounters your PCI device, it is given a logical address. The components of the logical address are as follows:

- Function number
- Device number
- Logical PCI bus number

On SGI Altix systems, because of the vast number of PCI-X buses that the system can support, SGI has provided a hardware graph topology that maps all PCI devices in the system from their Linux logical address to the actual hardware address. This facility allows you to correctly identify a particular device on the system for any purpose, one of which may be to find the physical failing component that has been reported by the system.

During SGI Altix platform initialization, as the system hardware probes outwards from the compute bricks to initialize the various components attached to the system, the system software uses the Linux device filesystem (`devfs`) to create a hierachical topology of all the components discovered and initialized.

An example:

```
/dev/hw/module/001c11/slab/0/Pbrick/xtalk/12/pci-x/0/1
```

This topology describes the actual hardware location of a specific PCI-X device. The following section describes how to use this topology information to physically find your device.

Physically Locating Your PCI Device Information

The `/proc/pci` file provides a user level display of all PCI devices seen and initialized by the Linux kernel. Consider the following example:

```
[root@pumpkin root]# cat /proc/pci

       ....

  Bus  3, device   1, function  0:
    Fibre Channel: QLogic Corp. QLA2300 64-bit FC-AL Adapter (#5) (rev 1).
      IRQ 567.
      Master Capable.  Latency=64.  Min Gnt=64.
      I/O at 0xc00000080fa00000 [0xc00000080fa000ff].
      Non-prefetchable 64 bit memory at 0xc00000080fe00000 [0xc00000080fe00fff].

       ....
```

Note: The bus number (3) in the preceding example is in decimal notation and the bus numbers in `/dev/hw/linux/bus/pci-x/*` are in hexadecimal notation.

You can locate your device physically from the logical Linux PCI bus number and device number by using the `/proc/pci` output. The PCI device in the previous example is on logical PCI bus 3, device 1. The device number is the same as the slot number on the PCI bus. Therefore, you can determine that the device is on slot 1 of that PCI bus.

SGI Altix systems provide topology information (known as the hwgraph) for all system components. The hwgraph depicts the connections among these components. The hwgraph topology information is rooted at `/dev/hw`. The following example shows how to find the physical location of your device. It shows the physical location of Linux bus numbers as provided by the `/proc/pci` display.

```
[root@pumpkin root]# ls -al /dev/hw/linux/bus/pci-x/*
lr-xr-xr-x    1 root     root        56 Dec 31  1969 /dev/hw/linux/bus/pci-x/1-> ../../
      ../hw/module/001c27/slab/0/Pbrick/xtalk/12/pci-x/0
lr-xr-xr-x    1 root     root        56 Dec 31  1969 /dev/hw/linux/bus/pci-x/2-> ../../
      ../hw/module/001c27/slab/0/Pbrick/xtalk/12/pci-x/1
lr-xr-xr-x    1 root     root        56 Dec 31  1969 /dev/hw/linux/bus/pci-x/3-> ../../
      ../hw/module/001c27/slab/0/Pbrick/xtalk/15/pci-x/0
lr-xr-xr-x    1 root     root        56 Dec 31  1969 /dev/hw/linux/bus/pci-x/4-> ../../
      ../hw/module/001c27/slab/0/Pbrick/xtalk/15/pci-x/1
```

Note: The bus numbers in the preceding example are in hexadecimal notation.

The components are defined as follows:

`module/001c27`	Indicates the module identification number. Each module on the system has a module identification number. This number is displayed on the LED on the front of the module.
`/slab/0`	Indicates the slab number. Each compute brick has two slabs (also known as nodes) — slab 0 and slab 1.
`Pbrick`	Indicates the I/O brick type. All IX-bricks and PX-bricks are identified as `Pbrick` in the topology.
`xtalk/15`	Indicates the PCI-X host bridge adapter number. There are three PCI-X host bridge adapters in each I/O brick. They are numbered 12, 13, and 15. Each PCI-X host bridge adapter number is "stamped" at the back of the I/O brick.

`pci-x/0` Indicates the logical PCI-X bus number on the host bridge adapter. Each PCI-X host bridge adapter has two PCI-X buses — bus 1 and bus 2. This number is also "stamped" on the back panel of the I/O brick. Unfortunately, the hwgraph assigns the PCI bus numbers as 0 and 1 instead of 1 and 2. Therefore, `pci-x/0` is actually physical bus 1 and `pci-x/1` is physical bus 2.

To physically locate your device, use the hwgraph topology information, as follows: If, from the `/proc/pci` display, your Linux logical bus number is 3, your device number is 1, and `/dev/hw/linux/busnum` points at `../../../hw/module/0012c27/slab/0/Pbrick/xtalk/15/pci-x/1`, you know that your device (device 1) is in slot 1, on physical bus number 2 (`pcix-x/1`), on PCI-X host bridge adapter number 15, that is connected to slab 0 (node 0) of the compute brick with module identification number 0012c27.

Programmatically Locating Your PCI Device

PCI services provided by the Linux kernel driver kernel interface (DKI) require a handle that maps to your PCI device. This handle is actually a pointer to your device `pci_dev` structure. Therefore, before the device driver can call the Linux kernel for any PCI services, it must first locate its PCI device handle.

You can locate the PCI device `pci_dev` structure by using the following methods:

```
 struct pci_dev *
               pci_find_device(unsigned int vendor, unsigned int device,
                       const struct pci_dev *from)
```

This routine scans from `*from` to locate the `pci_dev` structure that matches the given vendor and device identification. If `*from` is `NULL`, it starts from the beginning of the `pci_dev` list. If a match is found, the address of `pci_dev` is returned in `*from`.

Other search routines of interest are:

```
struct pci_dev *pci_find_subsys (unsigned int vendor,
                        unsigned int device,
                        unsigned int ss_vendor, unsigned int ss_device,
                        const struct pci_dev *from);
```

```
struct pci_dev *pci_find_class (unsigned int class,
                        const struct pci_dev *from);


struct pci_dev *pci_find_slot (unsigned int bus, unsigned int devfn);
```

The device driver can also scan through the PCI device list by searching the list in reverse order (see Example 5-1 on page 56) and scanning the list in order (see Example 5-2 on page 56). The advantage of these scanning mechanisms is that they allow the device driver to match any attributes required for locating its PCI device.

Example 5-1 Scanning the list in reverse order

```
#define pci_for_each_dev_reverse(dev)\
               for(dev = pci_dev_g(pci_devices.prev); \
                        dev != pci_dev_g(&pci_devices); \
                        dev = pci_dev_g(dev->global_list.prev))
```

Example 5-2 Scanning the list in order

```
#define pci_for_each_dev(dev) \
                for(dev = pci_dev_g(pci_devices.next); \
                        dev != pci_dev_g(&pci_devices); \
                        dev = pci_dev_g(dev->global_list.next))
```

PCI/PCI-X Configuration Space

Each PCI/PCI-X device has 256 bytes of configuration address space. Sixty-four bytes of this area are standardized as shown in Figure 6-1 on page 58. For more details regarding these registers, see *PCI-X System Architecture* or *PCI System Architecture*.

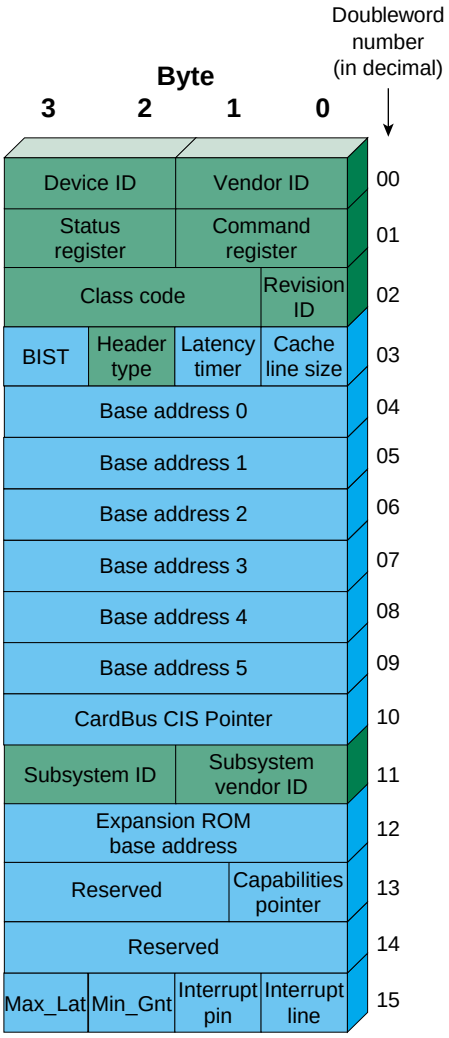

Figure 6-1 PCI-X Configuration Space

On SGI Altix systems, the following registers in the PCI configuration space should not be used directly by device drivers:

- Base address registers. These register values are not usable on the CPU as PIO addresses.

- Expansion ROM base address registers. SGI Altix systems do not support loading and executing basic input/output systems (BIOS) code resident in PCI ROM. This register is not initialized. Drivers should ensure that all initialization is performed in the driver.
- Interrupt Line. On legacy systems, this is the IRQ. SGI Altix systems support IRQs greater than 8 bits (greater than 256 IRQs). The interrupt line value is not used on SGI Altix systems.

Device drivers have to use the addresses and values provided in the `pci_dev` structure. For methods of programmatically finding your device `pci_dev` information, see Chapter 5, "Finding Your PCI Device" on page 51.

You can read and write PCI/PCI-X configuration space for your device by using the following PCI configuration space routines:

`/usr/include/linux/pci.h:`

```
int pci_read_config_byte(struct pci_dev *dev,int where,u8 *ptr);

int pci_read_config_word(struct pci_dev *dev,int where,u16 *ptr);

int pci_read_config_dword(struct pci_dev *dev,int where, u32 *ptr);

int pci_write_config_byte (struct pci_dev *dev,int where, u8 val);

int pci_write_config_word (struct pci_dev *dev,int where, u16 val);

int pci_write_config_word (struct pci_dev *, int where, u32 val);

int pci_write_config_dword (struct pci_dev *dev,int where, u32 val);
```

Variables are as follows:

Variable	Description
`*dev`	Pointer to your device `pci_dev` structure
`where`	Byte offset into the PCI configuration space of your device
`*ptr`	Address of the location to store the byte read
`val`	Value to write into the PCI configuration space of your device

PCI-X I/O and Memory Resources

This chapter describes programmable I/O (PIO) architecture, first describing the PIO address and then describing the flow of PIO operations.

Anatomy of a PIO Address

On SGI Altix 3000 systems, the PIO address that the device driver and CPU encounter is different from the PCI-X addresses that are initialized on the base address registers (BARs). PCI-X host bridge adapters on SGI Altix 3000 systems can generate only single address cycles for PIO read and write operations. This limits the size of the PCI address on a PCI bus to 32 bits.

PCI-X host adapters on SGI Altix 3000 systems provide a set of device registers that help maintain PIO attributes. Two of the most common PIO attributes are as follows:

`DEV_IO_MEM`	Enables device memory or I/O space. When set, the request generated on the PCI bus is for the PCI memory resource. Otherwise, the request is generated for the PCI I/O resource.
`DEV_OFF`	Specifies PCI-X address offset bits. These 12 bits replace bits 31 to 20 of the PIO address that the PCI-X host bridge adapter obtains from the CPU.

PIO Addresses

The diagram in Figure 7-1 on page 62, provides the breakdown of a “mapped” PIO address as seen by the device driver on the CPU. Note that this address is quite different from the addresses that are initialized on the BARs.

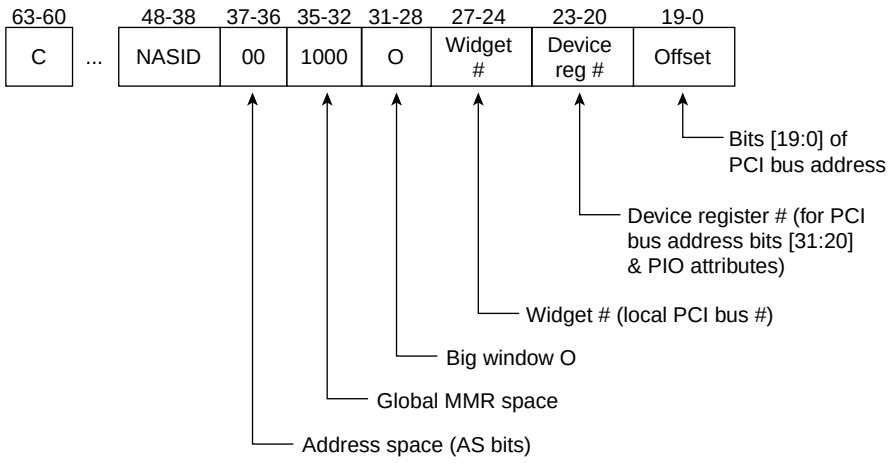

Figure 7-1 PIO Address Format

Flow of PIO Operation

The following sections describe the flow of PIO operation to local and remote PCI-X devices.

Targeting a PCI-X Device on a Local Node

The flow of PIO to a local PCI-X device is depicted in Figure 7-2 on page 63. Following the figure is an explanation of the numbered components.

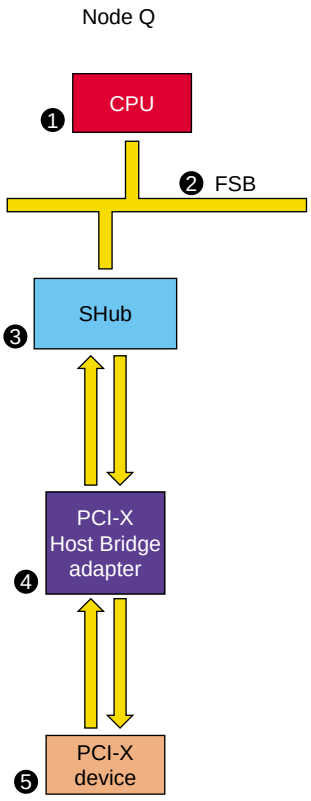

Figure 7-2 PIO to a Local PCI-X Device

1. CPU sees that the address is "uncached." It forwards the address on the front side bus (FSB).
2. The SHub receives the request and determines from the NASID (bits 48 to 38) that it is targeted to itself.
3. The SHub forwards the request to the attached PCI-X host bridge adapter via the Xtown2 link.
4. The PCI-X host bridge adapter receives the request, parses the addresses, and places the modified PIO address (PCI-X bus address) on the PCI-X bus.
5. The PCI-X device with the matching BARs responds to the request.

Targeting a PCI-X Device on a Remote Node

The flow of PIO to a remote PCI-X device is depicted in Figure 7-3 on page 64. Following the figure is an explanation of the numbered components.

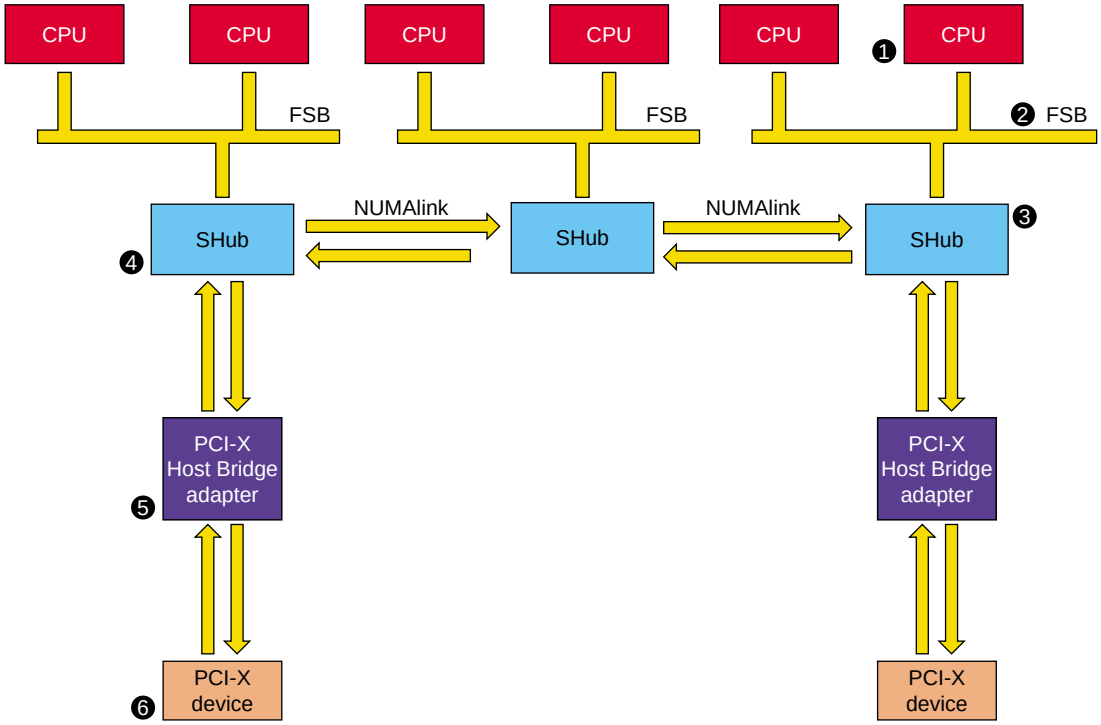

Figure 7-3 PIO to a Remote PCI-X Device

1. The CPU places the PIO mapped address on the FSB.
2. The local SHub receives the request and determines from the NASID that it is **not** targeted to itself.
3. The local SHub forwards the request via the NUMAlink to the targeted remote SHub.
4. The PCI-X host bridge adapter receives the request, parses the addresses, and places the modified PIO Address (PCI-X bus address) on the PCI-X bus.

5. The PCI-X device with the matching BARs responds to the request.

PIO Address Translation from CPU to PCI Bus

The PIO address that the CPU issues does not look anything like the PCI address on the targeted base address register (BAR). Consider the following example:

Example 7-1 Address Translation

A PCI-X device has requested for I/O a resouce of 512 bytes. It is connected via NASID (Node ID) 0x0 and it is on that node's local PCI bus (widget identifier) 0xe.

At boot time the system has initialized the BARs to 0x1fff_0001. Given the previous information, the "mapped" PIO address looks like the following to the device driver and the CPU:

```
0xc000_0008_0e4f_0000
```

The diagram in Figure 7-4 on page 65, provides the breakdown of an address that the CPU issues. This is the address you get from the `pci_dev` structure.

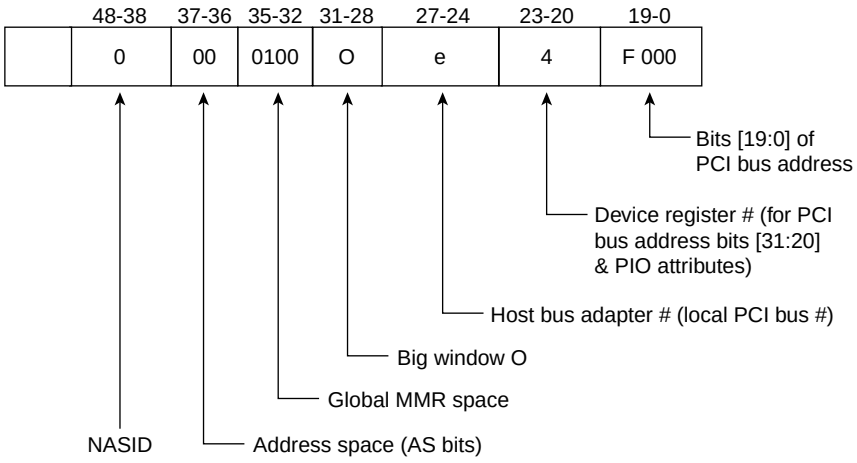

Figure 7-4 PIO Address from the CPU

The targeted PCI-X host bridge adapter gets the PIO address as the following:

```
0x0e4f_0000
```

The device register contents of 0x11ff specifies the following:

```
DEV_IO_MEM == IO
DEV_OFF == 0x1ff
```

The relevent device register is the one identified by PCI bus widgets 0xe and 0x4. With this information from the device register for this address, the PCI-X host bridge adapter places the following PCI-X bus PCI address on widget 0xe as a PCI I/O transaction (read or write operation):

```
0x1fff_0000
```

The PCI-X host bridge adapter strips the 0xe4 (from 0x0e4f_0000 to form 0xf_0000) and prepends 0x1ff (from the device register `DEV_OFF` value) to 0xf_0000 to make the PCI-X bus address 0x1fff_0000. This value matches the value as initialized in the BAR.

Note: Reading the BARs for an address to use as a PIO will definitely not work on SGI Altix 3000 systems. It might or might not work on any other systems. Most importantly, it makes your code **not** portable.

PCI-X PIO Resource Management

Device drivers on SGI Altix 3000 systems must use the PCI resource routines described in the following sections to obtain either the I/O or memory PIO addresses that are initialized by the platform. Device drivers must not read or use the BARs directly.

PCI-X I/O Resource Address

Linux provides the following PCI resource interfaces to obtain the PCI I/O resource address.

To retrieve the start I/O resource address:

```
pci_resource_start(dev,bar)
```

To retrieve the ending address of an I/O resource address:

```
pci_resource_end(dev,bar)
```

To obtain the length of an I/O resource address:

```
pci_resource_len(dev,bar)
```

For example:

```
reg_base = pci_resource_start(pdev, 0);
reg_len = pci_resource_len(pdev, 0);
flags = pci_resource_flags(dev,bar);
if (flags & IORESOURCE_IO) {
        // This is an I/O resource.
}
```

PCI-X Memory Resource Address

Linux provides the following PCI resource interfaces to obtain PCI memory resource addresses.

To retrieve the start memory resource address:

```
pci_resource_start(dev,bar)
```

To retrieve the ending address of a memory resource address:

```
pci_resource_end(dev,bar)
```

To obtain the length of a memory resource address:

```
pci_resource_len(dev,bar)
```

For example:

```
reg_base = pci_resource_start(pdev, 0);
reg_len = pci_resource_len(pdev, 0);
flags = pci_resource_flags(dev,bar);
if (flags & IORESOURCE_MEM) {
        // This is a memory resource.
}
```

If the device driver provides the ability to memory–map the memory resource address into user space, the `pgprot_noncached`() macro must be used to set appropriate caching attributes on the corresponding virtual memory area.

For example:

```
static int
my_dev_mmap(struct file *filp, struct vm_area_struct *vma) {
        unsigned long my_dev_page;
        struct pci_dev *dev;

        /* Determine dev by methods specific to your driver, then... */
        /* Check validity of input arguments, then... */

        my_dev_page = pci_resource_start(dev, 0) + MY_DEV_PAGE_OFFSET;
        vma->vm_flags |= VM_IO | VM_RESERVED;
        vma->vm_page_prot = pgprot_noncached(vma->vm_page_prot);
        return io_remap_page_range(vma, vma->vm_start, my_pci_page,
                                   MY_DEV_PAGE_LEN, vma->vm_page_prot);
 }
```

PCI-X I/O Resource Reservation

It is strongly recommended that device drivers call the following PCI-X resource reservation routines to ensure that no other drivers are currently using a resource by mistake.

To reserve a PCI I/O resource region:

```
request_region(start,n,name)
```

To reserve a PCI memory resource region:

```
request_mem_region(start,n,name)
```

To release the PCI I/O resource:

```
release_region(start,n);
```

To release the PCI memory resource region:

```
release_mem_region(start,n);
```

For example:

```
request_region(reg_base, reg_len, "any_id");
       .....
release_region(reg_base, reg_len);
```

PCI-X I/O Resource Use Macros

You should reference PCI-X I/O resource addresses by using the following macros.

Single byte access macros:

```
inb(address);
outb(value, address);
```

Single word access macros:

```
inw(address);
outw(value, address);
```

Single long access macros:

```
inl(address);
outl(value, address);
```

Multiple byte access macros:

```
insb(address, value_address, byte_count);
outsb(address, value_address, byte_count);
```

Multiple word access macros:

```
insw(address, value_address, word_count);
outsw(address, value_address, word_count);
```

Multiple long access macros:

```
insl(address, value_address, long_count);
outsl(address, value_address, long_count);
```

Note: Even though on SGI Altix 3000 systems, PCI-X I/O resource addresses are mapped addresses and can be referenced without using any of the macros in the preceding list, it is recommended that you use these macros so that your code is portable.

PCI-X Memory Resource Use Macros

PCI-X memory resource addresses should not be used alone. Use the following platform-independent macros with PCI-X memory resource addresses.

Single byte access macros:

```
readb(address);
writeb(value, address);
```

Single word access macros:

```
readw(address);
writew(value, address);
```

Single long access macros (4 bytes):

```
readl(address);
writel(value, address);
```

Single unsigned long access macros (8 bytes):

```
readq(address);
writeq(value, address);
```

PIO Write (Posted) Synchronization

PIO write operations on SGI Altix 3000 systems can be cached in the various system components prior to actual arrival at the device. These PIO write operations are called “posted” operations. To explicitly flush these write operations, the device driver is required to perform a PIO read operation (also known as a “PIO flush”) after the last significant PIO write operation.

The need to perform PIO flushes becomes apparent when you consider a multithreaded driver. Multithreaded drivers use a memory lock for synchronization, as shown in the example sequence in Table 7-1 on page 70.

Table 7-1 Memory Locks

Time	CPU 0	CPU 1
n	(1) Grab lock (This CPU wins the race for the lock)	(1) Grab lock (This CPU must wait, as CPU 0 has the lock)
n + 1	(2) PIO write of Oxa to device *x*	(2) Waiting
n + 2	(3) Release lock (but no guarantee that #2 has completed)	(3) Receive lock
n + 3	(4) No activity	(4) PIO write of Oxb to device *x*
n + 4	(5) Device can receive Oxb before Oxa	

To avoid the releasing of the memory lock before the PIO write has completed, drivers for SGI Altix 3000 systems can be programmed to issue an additional operation (a read operation to the same controller, called a PIO flush) to force the data to be delivered to the device before the memory lock is released and a second thread can issue a read operation. The sequence shown in Table 7-2 on page 72, illustrates the correct usage.

Table 7-2 Correct Memory Lock Usage

Time	CPU 0	CPU 1
n	(1) Grab lock (This CPU wins the race for the lock)	(1) Grab lock (This CPU must wait, as CPU 0 has the lock)
n + 1	(2) PIO write of Oxa to device *x*	(2) Waiting
n + 2	(3) PIO read to the same controller (4) Device receives Oxa	(3) Waiting
n + 3	(5) Release lock	(4) Receive lock
n + 4	(6) No activity	(5) PIO write of Oxb to device *x* (6) PIO read to the same controller (7) Device receives Oxb

Even though at *n* + 1 CPU 0 issued the PIO write, it does not guarantee that the device will have received the data (Oxa) before *n* + 3. Similarly, it does not guarantee that the PIO write from CPU 1 at *n* + 3 does not arrive at the device before the operation that was issued by CPU 0 at *n* + 1.

Following is a more concrete example from a hypothetical device driver:

```
 ...
CPU A:  spin_lock_irqsave(&dev_lock, flags)
CPU A:  val = readl(my_status);
CPU A:  ...
CPU A:  writel(newval, ring_ptr);
CPU A:  spin_unlock_irqrestore(&dev_lock, flags)
...
CPU B:  spin_lock_irqsave(&dev_lock, flags)
CPU B:  val = readl(my_status);
CPU B:  ...
CPU B:  writel(newval2, ring_ptr);
CPU B:  spin_unlock_irqrestore(&dev_lock, flags)
...
```

In the case above, the device may receive `newval2` before it receives `newval`, which could cause problems. Following is a fix for the problem:

```
...
CPU A:  spin_lock_irqsave(&dev_lock, flags)
CPU A:  val = readl(my_status);
CPU A:  ...
CPU A:  writel(newval, ring_ptr);

 (***The following line fixes the previous problem***)
CPU A:  (void)readl(safe_register); /* maybe a config register? */

CPU A:  spin_unlock_irqrestore(&dev_lock, flags)
...
CPU B:  spin_lock_irqsave(&dev_lock, flags)
CPU B:  val = readl(my_status);
CPU B:  ...
CPU B:  writel(newval2, ring_ptr);
CPU B:  (void)readl(safe_register); /* maybe a config register? */
CPU B:  spin_unlock_irqrestore(&dev_lock, flags)
```

Here, the read operations from `safe_register` cause the I/O chipset to flush any pending write operations before actually posting the read operation to the chipset, thus preventing possible data corruption.

For more informaton, see Appendix A, "Memory Operation Ordering on SGI Altix Systems" on page 101.

PIO Read Flushing Posted DMA Buffers

SGI Altix system hardware provides the capability to buffer write DMA buffers. These buffers are flushed only when the device generates an interrupt.

PCI specification requires that any bridge that can buffer DMA write buffers must ensure that these posted buffers are flushed whenever a PIO read is issued to the device. Because this specification is not supported on SGI Altix hardware, all of the PIO read macros (for example, `inX`() and `readX`()) have been enhanced to perform a DMA write flush before returning to the caller. However, on some devices and device drivers, this enhancement can cause a negligible performance degradation. Because of this potential performance implication, a "fast" PIO call procedure is available. These calls do not perform any DMA write buffer flushing. For devices that do not depend

on a PIO read to flush posted write DMA buffers, you can use the following set of interfaces:

```
sn_inb_fast (unsigned long port)
sn_inw_fast (unsigned long port)
sn_inl_fast (unsigned long port)
sn_readb_fast (void *addr)
sn_readw_fast (void *addr)
sn_readl_fast (void *addr)
```

These calls are defined in the `include/asm-ia64/sn/sn2/io.h` file.

PCI-X Interrupt Mechanism

This chapter describes the interrupt mechanism for SGI Altix systems and provides information about interrupt architecture, interrupt requests (IRQs), and interrupt registration.

Interrupt Architecture

The interrupt architecture of SGI Altix systems might differ from other Itanium platforms. However, the interfaces presented to the device driver are the same. Linux interrupt support on SGI Altix systems is no different from support on any other Intel Itanium 2-based platforms.

Interrupt Request (IRQ) Management

Some aspects of interrupt configuration on the SGI Altix platform might differ from those on other Linux platforms. However, as for any Linux platforms, configuration is performed either in system BIOS or in Linux platform specific code. Both of these types of codes are transparent to Linux device drivers. Linux device drivers on SGI Altix systems install and register their interrupt handlers in exactly the same way as Linux device drivers on any other Linux platform. The interrupt flow is as follows:

1. The device pulls its configured interrupt pin (INTA, INTB, and so on).
2. The bridge uses the IRQ number to signal the SHub to interrupt the specified CPU.
3. The SHub delivers this interrupt with the IRQ number to the targeted CPU.
4. The targeted CPU delivers the interrupt to the driver that is registered with the IRQ number.

Driver Interrupt Registration

SGI Altix architecture can support more than 256 interrupt requests (IRQs). The 1-byte field in the PCI configuration register is not sufficient to map all of the possible IRQs that can exist. Therefore, a device driver that is retrieving the IRQ from the PCI

configuration space for interrupt installation (registration) is not portable to any platforms that can support more than 256 IRQs. On SGI Altix systems, you **must not** use the contents of the IRQ from the PCI configuration space. The proper procedure to use is as follows:

1. Get the IRQ number from the `pci_dev` structure initialized by the Linux PCI infrastructure during boot.
2. Call the `request_irq` method with the IRQ obtained in step 1.

Following is an example code sequence:

```
static void
intr_handler(int irq, void *private_data, struct pt_regs *regs);

my_irq = pci_dev->irq;
request_irq(my_irq, intr_handler,
        SA_INTERRUPT | SA_SHIRQ, "My Driver", private_data);
```

For more information on interrupt handling, see *Linux Device Drivers*, chapter 9, "Interrupt Handling."

Caution: An interrupt is the only mechanism in which posted DMA data are flushed from the PCI-X bridge to target memory.

PCI-X Direct Memory Access (DMA)

This chapter describes direct memory access (DMA) architecture, first describing the flow of DMA from the PCI-X bus into the system, and then describing the DMA mapped address. Figure 9-1 on page 77, and Figure 9-2 on page 78, show the targeting of local and remote node memory, respectively.

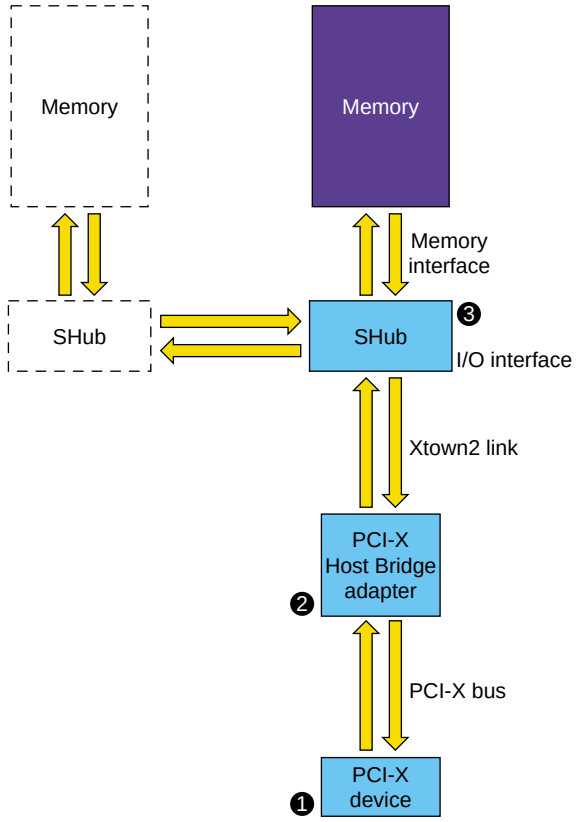

Figure 9-1 DMA to Memory on A Local Node

The flow depicted in Figure 9-1 on page 77, is as follows:

1. PCI-X device places the DMA request on the bus.
2. PCI-X host bridge adapter places the DMA request to the directly connected SHub's I/O interface.
3. Since this is a request to locally attached memory, the request is satisfied by the local SHub's memory.

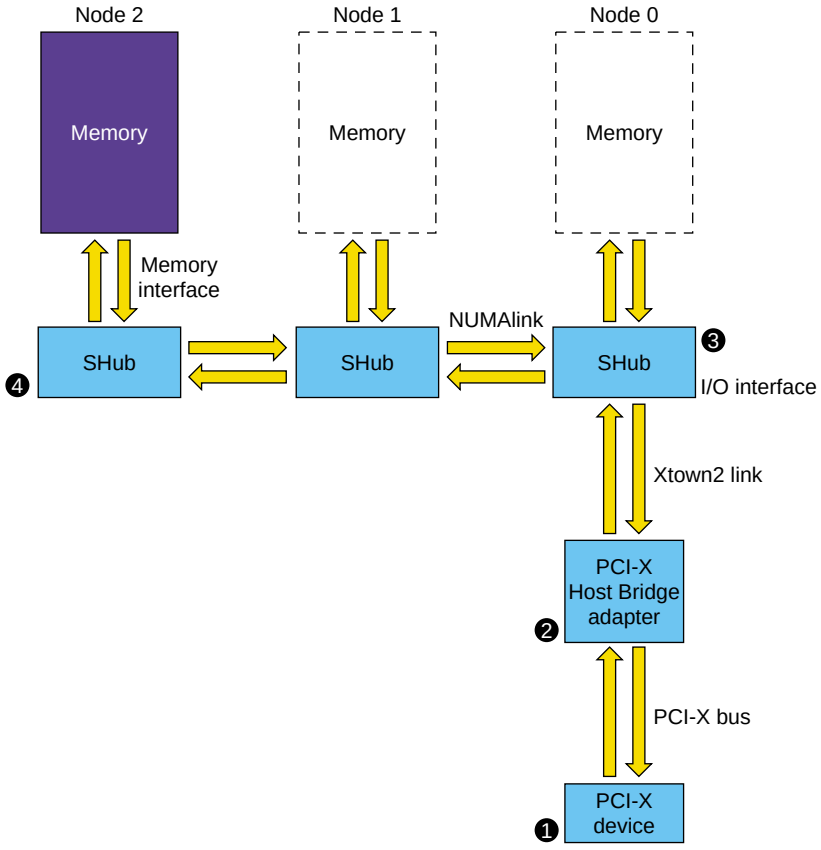

Figure 9-2 DMA to Memory on A Remote Node

The flow depicted in Figure 9-2 on page 78, is as follows:

1. PCI-X device places the DMA request on the PCI-X bus.

2. PCI-X host bridge adapter forwards the DMA request to the I/O interface on the directly attached SHub.
3. Because this request is targeted to a remote node's memory, the I/O interface (SHub) forwards this request to the remote SHub via the NUMAlink.
4. Remote SHub satisfies request with its memory.

Types of DMA Mappings

SGI Altix systems support consistent DMA mappings and streaming DMA mappings. The following sections describe each of these types.

Consistent DMA Mappings

Consistent DMA mappings are synchronous and coherent. These mappings are usually mapped by the driver at initialization time and unmapped only when the device is no longer running. Consistent DMA mappings guarantee that the device and the CPU can access the data in parallel and can see each other's updates without explicit software flushing.

Consistent DMA mappings always return a mapped DMA address that is 32-bit single address cycle (SAC) addressable, regardless of the DMA capability of the device.

Examples of memory areas that require consistent DMA mappings are the following device driver control structures shared between the host and controller (PCI device):

- Network card DMA ring descriptors
- SCSI adapter mailbox command data structures

Streaming DMA Mappings

Streaming DMA mappings are asynchronous and can be buffered (prefetched) by various hardware components along the DMA path.

Streaming DMA mappings are usually mapped for one DMA transfer and unmapped directly after the transfer. The unmap operation usually guarantees that the DMA data is coherent, but not on SGI Altix systems.

Examples of memory that can use streaming DMA mappings are as follows:

- Networking buffers transmitted or received by a device
- Filesystem buffers written or read by a SCSI device

Anatomy of a Mapped DMA Address

On SGI Altix systems, the mapped DMA address as seen and issued by the PCI-X device controller can be either 64 bits (dual address cycle (DAC)) or 32 bits (single address cycle (SAC)). This platform does not support drivers that cannot handle at least 32-bit DMA addresses. Devices that can handle DMA addresses from 33 bits to 63 bits are assigned 32-bit DMA addresses. Devices that are capable of handling 64-bit addresses are always assigned 64-bit DMA addresses.

On SGI Altix systems, the mapped DMA address as seen and issued by the PCI-X device controller can be either 64 bits (dual address cycle or DAC) or 32 bits (single address cycle or SAC). This platform supports only devices that are capable of using 32-bit or greater DMA addresses. Devices that can handle DMA addresses from 33 bits to 63 bits are assigned 32-bit direct- or page-mapped DMA addresses. Devices that are capable of handling 64-bit addresses are assigned 64-bit DMA addresses for 'streaming' type mappings (see `pci_map_single` and `pci_map_sg`), and 32-bit DMA addresses for so-called 'consistent' mappings (see `pci_alloc_consistent`).

Direct-mapped addressing is per-bus. That is, when a device is on a bus by itself, the PCI-X bridge chip can be programmed to map 32-bit DMAs by the device into a particular 2-GB window of system memory. Page-mapped addressing operates on a per-address basis, which allows the driver to program the device to DMA into multiple 2-GB windows of system memory. These modes of addressing will be transparent to the driver, however, and are only described here for informational purposes. The following sections describe direct mapped DMA addresses and page mapped DMA addresses, respectively.

Format of 32-bit Direct Mapped DMA Addresses

Figure 9-3 on page 81, shows the format of a PCI direct mapped register. Figure 9-4 on page 81, shows the format of a direct mapped DMA address.

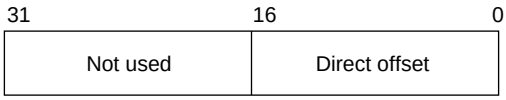

Figure 9-3 PCI Direct Mapped Register (one per PCI bridge)

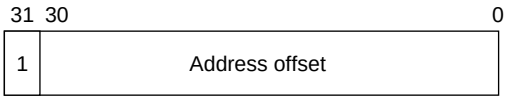

Figure 9-4 32–bit Direct Mapped Address As Returned by the System

A DMA address is a 32-bit direct mapped DMA address when bit 31 of the 32-bit DMA address is set. When a DMA address is direct mapped, the actual system physical address (PCI bus address) is formulated from both the direct map register and the 32-bit DMA mapped address.

When a direct map register is a 17-bit constant value and when a 32-bit mapped DMA address is a 31-bit variable value, a 32-bit direct mapped mechanism can map any 2-GB range of system physical memory space. Figure 9-5 on page 81, shows the bits that comprise a 50-bit system address.

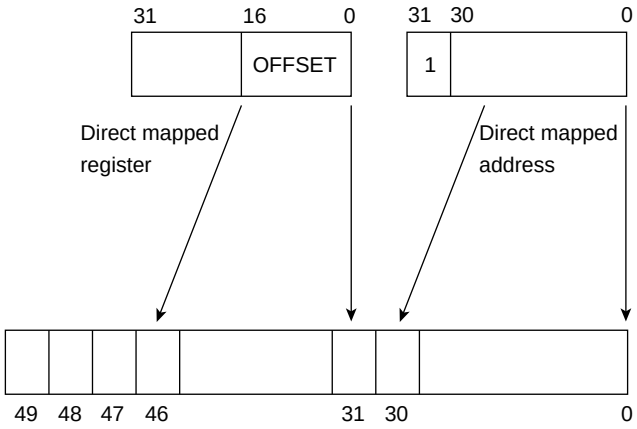

Figure 9-5 50-bit System Memory Address

In Figure 9-5 on page 81, the physical memory space is mapped as follows:

- Bits 30 – 0 from the 32-bit mapped DMA address become bits 30 – 0 of the system physical address.
- Bits 16 – 0 from the direct mapped register becomes bits 47 – 31 of the system physical address.

Note: There is only one direct mapped register per PCI-X bus.

Format of 32-bit DMA Page Mapped Addresses

The type is returned by `pci_alloc_consistent`. The type is also returned by `pci_map_*` when `pci_dev.dma_mask` is less than 64 bits. Figure 9-6 on page 82, shows the format of a 32-bit DMA page mapped address.

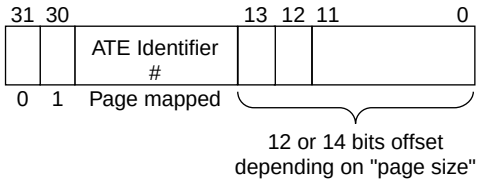

Figure 9-6 32-bit DMA Mapped Address

If bit 31 is set, it is a 32-bit direct mapped address. If bit 30 is set, it is a 32-bit page mapped address.

If the system page size is 4 KB, bits 11 to 0 are the page offset and the address translation entry (ATE) index is in bits 29 to 12. If the system page size is 16 KB, bits 13 to 0 are the page offset and the ATE index is in bits 29 to 14.

DMA mapped addresses 0x4000_0000 to 0x700_0000 are paged mapped addresses. This means that the address offset and the DMA attributes come from the targeted ATE register and direct map register.

DMA mapped addresses 0x8000_0000 to 0xffff_ffff are direct mapped adresses. This means that the address offset and the DMA attributes come from the device registers and the direct map register.

Format of a 64-bit DMA Mapped Adddress

The type is returned by `pci_map_*` when `pci_dev.dma_mask` is 64 bits (see `pci_set_dma_mask`). Figure 9-7 on page 83, shows the format of a 64-bit DMA mapped address.

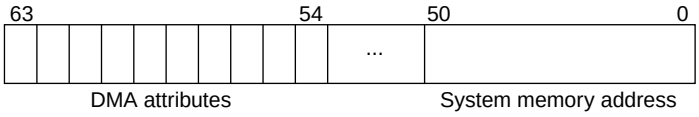

Figure 9-7 64-bit DMA Mapped Address

In 64-bit DMA mapped addresses, bits 63 to 54 are set. These are DMA attributes. Bits 50 to 0 target the actual system memory location.

Caution: Devices asking for 64-bit DMA addresses without the capacity to generate dual address cycles will cause unknown hangs and data corruption.

PCI-X DMA Address Management

Device drivers should follow these rules when using DMA resources on SGI Altix systems:

- Query that the platform can support your DMA capability. To determine whether the system supports the indicated DMA address size, use the following command:

  ```
  pci_dma_supported(dev, mask);
  ```

- Inform the system of your DMA capability. To inform the system that the device can support a DMA address only up to the indicated size, enter the following command:

  ```
  pci_set_dma_mask(dev, mask);
  ```

Example:

```
/* This driver can support up to 64-bit DMA Address. */
if (!pci_set_dma_mask(pdev, (u64) 0xffffffffffffffff)) {
                // Yes, system can support 64-bit DMA addresses.
```

```
                .. code ..
} else {
                // Okay, can system support 32-bit DMA addresses?
                err = pci_set_dma_mask(pdev, (u64) 0xffffffff);
                if (err) {
                        // Need to abort as this driver cannot support
                        // DMA addresses smaller than 32 bits!
                        .. code ..
                }
                // Alright, we are running with 32-bit DMA addresses.
                .. code ..
       }
```

On SGI Altix systems, the platform can support either 64-bit or 32-bit DMA addresses. Cards that can support 32-bit to 63-bit DMA addresses are given 32-bit DMA addresses. Cards that can support 64-bit DMA addresses are always given 64-bit DMA addresses. Cards that can support only DMA addresses that have less than 32 bits are not supported on this platform. 32-bit DMA addresses can be used in dual address cycles as long as bits 63 to 32 are zero.

PCI-X DMA Mapped Routines

On SGI Altix systems, DMA memories are required to be mapped or unmapped via the following routines:

For consistent DMA mappings:

```
void * pci_alloc_consistent(struct pci_dev *hwdev, size_t size, dma_addr_t *dma_handle)
void pci_free_consistent(struct pci_dev *hwdev, size_t size, void *vaddr, dma_addr_t dma_handle)
```

For streaming DMA mappings:

```
dma_addr_t pci_map_single(struct pci_dev *hwdev, void *ptr,size_t size, int direction)

void pci_unmap_single(struct pci_dev *hwdev, dma_addr_t dma_addr, size_t size, int direction)

int pci_map_sg(struct pci_dev *hwdev, struct scatterlist *sg,int nents, int direction)
void pci_unmap_sg(struct pci_dev *hwdev, struct scatterlist *sg,int nents, int direction)
```

On SGI Altix platforms, all DMA operations prior to the controller card's issue of an interrupt on the PCI-X bus are guaranteed to be "completed and coherent" before the

interrupt is forwarded to the CPU. In other words, the interrupt "pushes" any and all DMA cache data out into the system.

On SGI Altix systems, memory coherency between processor caches and memory buses is guaranteed, and use of the following routines is unnecessary:

```
void pci_dma_sync_single(struct pci_dev *hwdev,dma_addr_t dma_handle,size_t size, int direction)

void pci_dma_sync_sg(struct pci_dev *hwdev, struct scatterlist *sg, int nents, int direction)
```

For more details, see the documentation in the Linux source tree at the following location:

`linux/documentation/DMA-mapping.txt`

Device Driver Memory Usage

Device drivers can dynamically allocate memory via the many memory management interfaces that Linux provides. This chapter describes memory interfaces that return addresses that can be mapped as direct memory access (DMA) addresses.

Device Driver Memory Allocation

Device drivers should never try to use kernel text or data for DMA. It is also inappropriate to use memory areas allocated via the `vmalloc` routine as a DMA area, even though it can be done. The `vmalloc` routine allows the driver to allocate a large contiguous virtual address range. However, the address cannot be used as input to any of the DMA map routines.

Allocating Page Boundary Memory

Device drivers can use the `__get_free_page()` routine to allocate a page of memory that can be mapped as a DMA address, as in the following example:

```
{
        char *buf;
        buf = (char*) __get_free_page(GFP_KERNEL);

}
```

This routine returns a value of type `unsigned long`. Memory is always allocated from the node that the thread is executing first.

Memory areas allocated via this interface are region 7 addresses and identity mapped addresses. These kernel virtual addresses can be used as input to any of the PCI DMA map routines.

Allocating Page Boundary Memory on Specific Nodes

Device drivers can request memory on a specific node. The rationale for a specific request would be data placement for performance reasons. If you need to allocate memory on a specific node, use the following interface:

```
struct page * alloc_pages_node(int node_id, unsigned int gfp_mask, unsigned int order)
```

The address returned in `struct page *` can be used as input to any of the PCI DMA map routines, as in the following example:

```
struct page * my_page;
my_page = alloc_pages_node(node_id, gfp_mask, order);
dma_addr = pci_map_single(my_dev, my_page -> virtual, size, direction);
```

Note: The interface is available only if `CONFIG_NUMA` is configured on, as in SGI Altix 3000 systems.

Allocating Byte-Range Memory

Device drivers can ask for memory with any arbitrary length by using the following routine:

```
void * kmalloc (size_t size, int flags);
```

This routine returns region 7 and identity mapped kernel virtual addresses. This kernel virtual address can also be used as input into any of the PCI mapping routines.

The advantage of this routine is that it provides a single starting virtual and physical address that is contiguous. However, if the kernel cannot find a contiguous area large enough, this call will fail.

Accessing the User Memory Area

It is very important for drivers, when reading or writing into the user area, to verify first that the specified user address is valid. Linux provides two routines that enable kernel level code for verification.

For verification within the user virtual area:

```
int access_ok(type, user_address, size);
```

For verification when the area is actually mapped with the correct access rights:

```
int verify_area(type, user_address, size);
```

Where:

`type`:	`VERIFY_READ` - Verify for read access
	`VERIFY_WRITE` - Verify for read/write access
`user_address`:	User's virtual address
`size`:	Size of the area in bytes

The following routines write or read a a single value to or from the user area. The size of the single value depends on the size of (`*pointer`):

```
int put_user(value, pointer)
int get_user(value, pointer)
```

The following optimized routines write or read the given number of byte count to or from the user area:

```
int copy_to_user(user_address, kernel_address, byte_count);
int copy_from_user(kernel_address, user_address, byte_count);
```

The following routines provide string manipulation from the user area:

```
long strlen_user(user_address);
long strnlen_user(user_address, count);
long strncpy_from_user(kernel_address, user_address, count);
```

An important point to note is that all of these user area access routines actually perform access verification first. The following corresponding set of routines are available to the driver, which do not perform any access verifications. The names of these routines are the same as the precedint routines but with a double underscore (__) prepended.

```
int __put_user(value, pointer)
int __get_user(value, pointer)

int __copy_to_user(user_address, kernel_address, byte_count)
int __copy_from_user(kernel_address, user_address, byte_count)
```

```
long __strlen_user(user_address)
long __strnlen_user(user_address, count)
long __strncpy_from_user(kernel_address, user_address, count)
```

Disabling Validity Checking

Sometimes, kernel level routines require calls to system call handlers, as do user level routines. However, all of these handlers expect addresses in the call to be user addresses. A kernel level routine calling a system call interface with a kernel virtual address will definitely fail when the system call interface performs address validation. For example, the following call will fail when `sys_open` performs any address verification:

```
 sys_open("/dev/mydriver_config_file", O_RDONLY, 0);
```

Linux provides the following set of routines that, in effect, disable address validity checking:

```
current_segment = get_fs();
        set_fs(active_segment);
```

The `current_segment` and `active_segment` values are either `KERNEL_DS` or `USER_DS`. If the current data segment is `KERNEL_DS`, the access verification routines do not perform any verification. If the current data segment is `USER_DS`, the access routines verify that the specified user addresses are valid.

Therefore, if your driver requires negation of all address verification, use the following code:

```
mm_segment_t current_segment;
current_segment = get_fs(); /* Save current segment */
set_fs(KERNEL_DS);
fd = sys_open("/dev/mydriver_config_file", O_RDONLY, 0);

set_fs(current_segment) /* Restore saved segment */
```

Directly Mapping User Virtual Addresses

In general, device drivers allocate kernel buffers as intermediate transfer areas to move data between the user area and an external device. Sometimes, but not always, it can be more performance-efficient to transfer data directly between the user area

and the external device, without incurring the additional CPU time for copying between kernel buffers and the user area, and other such activities. Linux provides the Kernel IO Buffer (`kiobuf`) facility, which allows a user area to be mapped to kernel virtual space. Once this mapping is performed, kernel virtual addresses can be DMA mapped to allow external devices to write directly into the mapped user area.

`kiobuf` provides the following interfaces:

- Allocate and free kiobuf structures:

```
int alloc_kiovec(int number_pages, struct kiobuf **iovec);
void free_kiovec(int number_pages, struct kiobuf **iovec);
```

- Map/unmap user addresses to kernel virtual addresses:

```
int map_user_kiobuf(int rw, struct kiobuf *iobuf, unsigned long va, size_t len)
void unmap_kiobuf (struct kiobuf *iobuf)
```

- Lock the pages in the `kiobuf`:

```
int lock_kiovec(int nr, struct kiobuf *iovec[], int wait)
int unlock_kiovec(int nr, struct kiobuf *iovec[])
```

For more information regarding this feature, see *Linux Device Drivers*, chapter 13, "MMap and Dma."

Time Management

All device drivers must eventually deal with issues of time and timing. Chapter 6 of *Linux Device Drivers* provides extensive time management information for Linux drivers. This chapter highlights the architectural differences of SGI Altix systems in the interval timer counter (ITC) and provides brief information about execution delays in drivers.

Interval Timer Counter (ITC)

The ITC register is a free running, 64-bit counter that counts up at a fixed relationship to the processor clock. To retrieve elapsed cycles, Itanium 2 processors provide this register to programs via the `ia64_get_itc (void)` routine. Drivers should use the **cycle_t get_cycles (void);** interface in order to be portable. On SGI systems, these registers are not synchronized among the other CPUs on the system. Callers of this routine must be very careful that these calls are made within the same CPU. For more information regarding the ITC register, see the *Intel IA-64 Architecture Software Developer's Manual*, volume 2, "IA-64 System Architecture."

For system-wide synchronized timing information, you can use the `do_gettimeofday()` function.

Delaying Execution — Short Delay

The Linux kernel also provides the following routines to allow callers to delay for a specified amount of time:

`void udelay` *(unsigned long usecs)*
`void mdelay` *(unsigned long msecs)*
`void ndelay` *(unsigned long nsecs)*

The `udelay`() routine pauses for the specified number of microseconds while the `mdelay`() routine pauses for the specified number of milliseconds. The `ndelay`() routine pauses for the specified number of nanoseconds. Because these routines simply spin in the CPU, they should be used only for pausing small amounts of time.

Delaying Execution — Long Delay

If your driver requires longer delays than the delays that the simple `udelay`() and `mdelay`() commands provide, you must use other facilities provided by Linux that will provide execution of your tasks at a later time without depending on interrupts or spinning in a loop. Linux provides the following interfaces for this purpose:

- Task queues
- Tasklets
- Kernel timers

For documentation on these features, see *Linux Device Drivers*, chapter 6, "Flow of Time." All of these interfaces are supported on SGI Altix systems.

Building Linux Kernels and Modules

This section provides the location of the default configuration file, steps for building and booting a new Linux kernel on an SGI Altix series system, and steps for rebuilding a module. The commands and procedures for recompiling or rebuilding the Linux kernel are the same as for any other Linux platform.

Default Configuration File

The configuration file used to build the Linux kernel that is running on your SGI Altix series system is `/usr/src/linux/arch/ia64/sn/configs/defconfig-sn2`. To build a new kernel, use this configuration file as a base. It should not be necessary to make a new configuration file.

Building a New Linux Kernel

The steps for building a new kernel are as follows:

```
cd /usr/src/linux
make clean
cp configs/kernel-2.4.21-ia64.config .config
make oldconfig
make dep
make
make compressed
```

Booting Your New Linux Kernel

To boot your new Linux kernel, copy or move `vmlinuz` to `/boot/efi/efi/sgi` and reboot, as follows:

```
% cp arch/ia64/boot/vmlinuz /boot/efi/efi/sgi/my_vmlinuz reboot
```

Enter the following at the EFI shell command:

```
elilo my_vmlinuz root=/dev/sda3 console=ttyS0
```

For more details regarding rebuilding the Linux kernel, see the `/usr/src/linux/README` file . This file also contains information on how to rebuild a new Linux kernel from sources other than those released by SGI that you have downloaded on your machine.

Rebuilding Modules

To rebuild the modules on your system, use the following commands:

```
cd /usr/src/linux
make modules
```

Example output:

```
make[1]: Entering directory ‘/usr/src/linux-2.4.20/lib’
make -C zlib_deflate modules
make[2]: Entering directory ‘/usr/src/linux-2.4.20/lib/zlib_deflate’
gcc -D__KERNEL__ -I/usr/src/linux-2.4.20/include -Wall -Wstrict-prototypes
-Wno-trigraphs -O2 -fno-strict-aliasing -fno-common -g -fomit-frame-pointer -pipe
-ffixed-r13 -mfixed-range=f10-f15,f32-f127 -falign-functions=32 -DSGI_SN_EXECUTABLE_STACKS
-DBRINGUP -DBRINGUP2 -DMODULE -I /usr/src/linux-2.4.20/lib/zlib_deflate -nostdinc
-iwithprefix include -DKBUILD_BASENAME=deflate  -c -o deflate.o deflate.c

gcc -D__KERNEL__ -I/usr/src/linux-2.4.20/include -Wall -Wstrict-prototypes
-Wno-trigraphs -O2 -fno-strict-aliasing -fno-common -g -fomit-frame-pointer -pipe
-ffixed-r13 -mfixed-range=f10-f15,f32-f127 -falign-functions=32 -DSGI_SN_EXECUTABLE_STACKS
-DBRINGUP -DBRINGUP2 -DMODULE -I /usr/src/linux-2.4.20/lib/zlib_deflate -nostdinc
-iwithprefix include -DKBUILD_BASENAME=deftree -c -o deftree.o deftree.c

gcc -D__KERNEL__ -I/usr/src/linux-2.4.20/include -Wall -Wstrict-prototypes
-Wno-trigraphs -O2 -fno-strict-aliasing -fno-common -g -fomit-frame-pointer -pipe
-ffixed-r13 -mfixed-range=f10-f15,f32-f127 -falign-functions=32 -DSGI_SN_EXECUTABLE_STACKS
-DBRINGUP -DBRINGUP2 -DMODULE -I /usr/src/linux-2.4.20/lib/zlib_deflate -nostdinc
-iwithprefix include -DKBUILD_BASENAME=deflate_syms -DEXPORT_SYMTAB -c deflate_syms.c
rm -f zlib_deflate.o
ld  -r -o zlib_deflate.o deflate.o deftree.o deflate_syms.o
```

To install these newly created modules, use the following command:

```
make modules_install
```

The following step copies all the newly created modules to the directory:

```
/lib/modules/[kernel_version]/
```

Example output:

```
make[1]: Entering directory '/usr/src/linux-2.4.20/lib'
make -C zlib_deflate modules_install
make[2]: Entering directory '/usr/src/linux-2.4.20/lib/zlib_deflate'
mkdir -p /lib/modules/2.4.20-sgi220a31/kernel/lib/zlib_deflate/
cp zlib_deflate.o /lib/modules/2.4.20-sgi220a31/kernel/lib/zlib_deflate/
make[2]: Leaving directory '/usr/src/linux-2.4.20/lib/zlib_deflate'
make[1]: Leaving directory '/usr/src/linux-2.4.20/lib'
```

Note: The SGI Altix series system might come with binary-only release modules. These modules cannot be remade on your machine.

For more information on Linux kernel modules, see the following link:
http://tldp.org/HOWTO/Module-HOWTO/

Downloading SGI Altix RPMs

Open source rpms for the SGI Altix system can be downloaded from oss.sgi.com. These rpms can be downloaded with your favorite web browser at the following location:

http://oss.sgi.com/projects/sgi_propack/

You can also download these sources via Anonymous FTP, as follows:

1. % **ftp oss.sgi.com**
2. % **cd /projects/sgi_propack/download/2.2**
3. % **get kernel-2.4.20-sgi220rp03062622_10017.src.rpm**

If you have the correct directories and also root access set up to manage Red Hat rpms, you can use the various rpm commands to retrieve and build the sources from the source rpm. Otherwise, you can use the following procedure:

1. Use the `rpm2cpio` command to extract the files from the rpm.
2. Use the `unzip`(1) command to unzip the tar file.
3. Use the `tar`(1) command to untar the tar file.
4. Perform your build.

The following provides a session trace of the above steps:

```
$ rpm2cpio kernel-2.4.21-sgi303r2.src.rpm | cpio -iduvm
kernel-2.4.21-ia64.config
linux-2.4.21.tar.gz
linux-common.spec
linux-sn2.spec
module-info
77535 blocks
$ gzip -dc
$ gzip -dc
```

You can perform this task on any platform with Itanium processors.

Building New Modules

If you are writing and developing modules for the SGI Altix system on either an Altix platform or any other platform with Itanium processors, you can use the steps outlined in "Downloading SGI Altix RPMs" on page 97 to download the kernel sources onto your platform and then build and test your modules. You may consider using the following procedure to build your modules, (The following procedures assumes that you are using `/usr/src/example` as the directory of your module development) as follows:

1. Create a Makefile for 2.4 similar to the following:

```
obj-m += module_example.o
include $(TOPDIR)/Rules.make
```

2. Run the make(1) command from any location, as follows:

   % **make -C /usr/src/linux-2.4.20 SUBDIRS=/usr/src/example modules**

   An example session is, as follows:

```
Example session:

[root@rappel linux-2.4.20]# make -C /usr/src/linux-2.4.20 SUBDIRS=/usr/src/example modules
make: Entering directory '/usr/src/linux-2.4.20'
make -C  /usr/src/example CFLAGS="-D__KERNEL__ -I/usr/src/linux-2.4.20/include -Wall -Wstrict-prototypes -W
no-trigraphs -O2 -fno-strict-aliasing -fno-common -g -fomit-frame-pointer -pipe  -ffixed-r13 -mfixed-range=
f10-f15,f32-f127 -falign-functions=32 -DMODULE" MAKING_MODULES=1 modules
make[1]: Entering directory '/usr/src/example'
gcc -D__KERNEL__ -I/usr/src/linux-2.4.20/include -Wall -Wstrict-prototypes -Wno-trigraphs -O2 -fno-strict-a
liasing -fno-common -g -fomit-frame-pointer -pipe  -ffixed-r13 -mfixed-range=f10-f15,f32-f127 -falign-funct
ions=32 -DMODULE  -nostdinc -iwithprefix include -DKBUILD_BASENAME=module_example  -c -o module_example.o m
odule_example.c
make[1]: Leaving directory '/usr/src/example'
make: Leaving directory '/usr/src/linux-2.4.20'

[root@rappel linux-2.4.20]# ls /usr/src/example
Makefile  module_example.c  module_example.o
```

Note: SGI strongly recommends that you use this method because it ensures that all the appropriate defines and flags are used to build your modules.

Memory Operation Ordering on SGI Altix Systems

Memory operation ordering is a complicated set of rules with issues that are not specific to SGI Altix systems but rather to any Linux platforms with Intel Itanium 2-based processors. Similarly, this topic is not related to PIO posted operations described in "PIO Write (Posted) Synchronization" on page 70.

The compiler can reorder instructions and also optimize away instructions that appear to be superflous or are not used. One technique it might use is to preload some registers, whose contents might or might not be valid by the time they are needed and used.

One optimization feature of Intel Itanium 2 processors is that they can reorder instructions such that some instructions are scheduled and completed not exactly in the order that they appear in your program. For more information regarding memory ordering, memory fences, and so on, see the *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization* and *Intel Itanium Architecture Software Developer's Manual* on for additional information on MP coherence and synchronization.

This appendix describes the following memory operation aspects of SGI Altix systems:

- Memory ordering
- Release semantics
- Acquire semantics
- Memory fencing

Memory Ordering

Memory load and store operations on SGI Altix platforms will not necessarily complete (that is, be visible in memory to other CPUs) in program order. For example, consider the following code snippet (program order):

```
1:  ld r1=[r2]  // r1 = *r2
2:  st [r4]=r6   // *r4 = r6
3:  ld r8=[r9]   // r8 = *r9
4:  st [r22]=r3  // *r22 = r3
```

This code could actually execute in the following order:

1. Register `r1` is set to the value at memory address `r2`.
2. Register `r8` is set to the value at memory address `r9`.
3. The address in `r22` is set to the value in `r3`.
4. The address in `r4` is set to the value in `r6`.

Note: This is a separate issue from compiler reordering, as it occurs at runtime. This also assumes that the pointers in question point to non-overlapping addresses. The kind of reordering shown in the previous example can expose bugs of various types, some of them very similar to the PIO ordering and coherency issues explained in this document.

Release Semantics

Using release semantics on an Intel Itanium 2 processor, the programmer can ensure that all previous memory accesses are made visible prior to the `st.rel` process, though subsequent memory accesses may "float up" above `st.rel`. For example, consider the following code sample:

```
1:  st [r1]=r2        // cannot move below 2
2:  st.rel [r4]=r6    // will be visible only after 1 is visible
3:  ld r8=[r9]        // may be reordered
4:  st [r22]=r3       // may be reordered
```

The processor will guarantee that the memory reference on line 1 is visible before the the `st.rel` on line 2; that is, the following sequence could be the actual execution order:

1. The address in `r1` is set to the value in `r2`.
2. The address in `r22` is set to the value in `r3`.
3. The address in `r4` is set to the value in `r6` (will happen after one register `r8` is set to the value at memory address `r9`).

In other words, no prior memory references (in program order) are allowed to propagate below a store with release semantics, but memory references following an `st.rel` **might** "float up" above the `st.rel` instruction.

Release semantics is a one-directional fence that prevents "Downward" drift as shown in Figure A-1 on page 103.

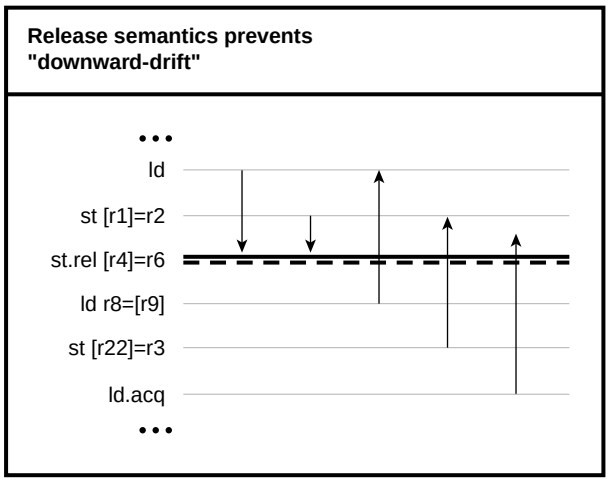

Figure A-1 Release Semantics One-Directional Fence

For more information on release semantics, see the *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization* and *Intel Itanium Architecture Software Developer's Manual*.

Acquire Semantics

Using so-called "acquire" semantics, the programmer can ensure that a load is made visible before all subsequent data accesses, though previous memory accesses can propagate below an `ld.acq` process. For example, consider the following code sample:

```
1:  ld r44=[r23]      // *can* move below 2
2:  ld.acq r1=[r2]    // will be visible before 3
3:  ld r8=[r9]        // cannot move above 2
4:  st [r4]=r6        // cannot move above 2
5:  st [r22]=r3       // cannot move above 2
```

The processor will ensure that the memory accesses prior to line 3 (in program order) are made visible before any subsequent accesses. So the following sequence could be executed by the processor:

1. Register `r1` is set to the value at memory address `r2` (will happen before 2).
2. Register `r8` is set to the value at memory address `r9`.
3. The address in `r4` is set to the value in `r6`.
4. Register `r44` is set to the value at memory address `r23`.
5. The address in `r22` is set to the value in `r3`.

Acquire semantics is a one-directional fence that prevents “Upward” drift as shown in Figure A-2 on page 104.

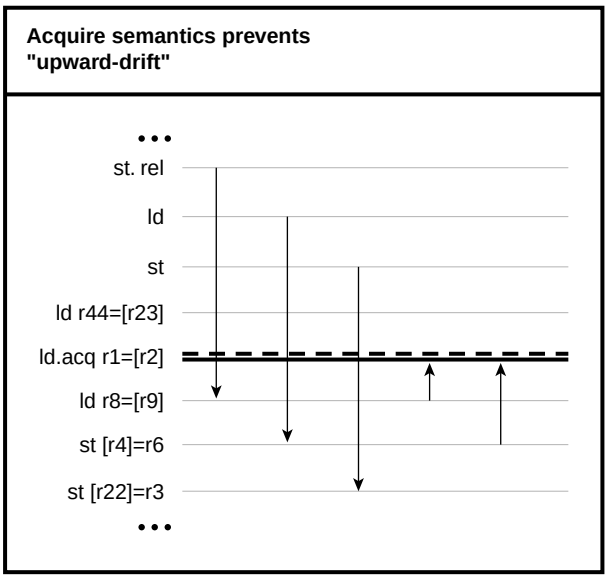

Figure A-2 Acquire Semantics One-Directional Fence

.

For more information on acquire semantics, see the *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization* and *Intel Itanium Architecture Software Developer's Manual*.

Memory Fencing

A memory fence acts as a simple, two-way barrier for memory operations as shown in Figure A-3 on page 105. For example, consider the following snippet:

```
1:  ld r1=[r2] <--\
2:  st [r4]=r6 <--- neither can move below 3
3:  mf
4:  ld.acq r8=[r9] <-- neither can move above 3
5:  st [r22]=r3 <----/
```

Lines 1 and 2 are guaranteed to be visible before any subsequent memory accesses (like those on lines 4 and 5), and memory accesses following the fence **will not** be visible to instructions before the memory fence (in program order).

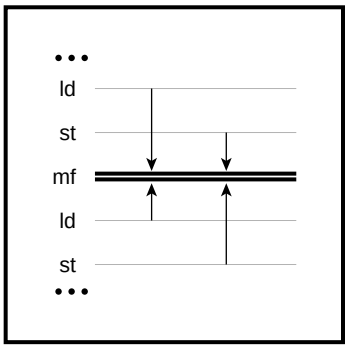

Figure A-3 Two-dimensional Memory Fence (`mf`)

For more information on memory fencing semantics, see the *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization* and *Intel Itanium Architecture Software Developer's Manual*.

Index

64-bit address support, 42

A

Acquire semantics, 103
Address management, 30
Address mapping
  DMA, 43
  PIO, 43
Address space
  64-bit, 20
  AMO, 23
  bus virtual, 27
  cacheable memory, 25
  CPU access, 31, 33
  DMA, 80
  global MMR, 22
  physical, 21
  SHub physical, 26
  supported, 42
  system memory, 20
  user virtual mapping, 90
AMO space, 23
Architecture
  interrupt, 75
  PIO, 61
  system components, 13
  system memory address space, 20
Atomic memory operation (AMO), 23

B

BaseIO, 37
Bus
  adapter, 28
  arbitration, 44
  virtual address, 27

C

Cache coherency, 25
Cacheable memory space, 25
Compute/processor node, 16
Configuration register, 41

D

Device driver
  byte-range memory allocation, 88
  disabling validity checking, 90
  memory allocation, 87
    for specific nodes, 88
    page boundary, 87
  memory usage, 87
  user memory area access, 88
Direct Memory Access (DMA)
  See "DMA", 35
DMA
  32–bit direct mapped addresses, 80
  32–bit page mapped addresses, 82
  64–bit mapped addresses, 83
  access to system physical memory space, 35
  address management, 83
  addresses, 27
  addressing, 30
  addressing extension, 30
  architecture, 77
  consistent mappings, 79
  mapped address anatomy, 80
  mapped routines, 84

streaming mappings, 79
types of mappings, 79

E

Execution delay
long, 94
short, 93

G

Global MMR space, 22

I

Interrupt request
See "IRQ", 75
Interrupt signals, 44
Interval Timer Counter (ITC), 93
IRQ
driver registration, 75
management, 75
IX-brick, 14, 18, 37

L

Latency and operation order, 40
Legacy functionality, 2

M

Memory
allocation for device drivers, 87
allocation for page boundary, 87
allocation for specific nodes, 88
byte-range allocation, 88
fencing, 105
ordering, 101
user access, 88
Memory access
from CPU, 31
from devices, 35
to device registers, 33

N

Node, compute/processor, 16

P

PCI-X
with BaseIO, 18
with expansion, 19
PCI/PCI-X
configuration space, 57
device attachment, 37
DMA address management, 83
DMA mapped routines, 84
I/O and memory resources, 61
I/O resource management, 66
I/O resource use macros, 69
implementation, 40
interrupt mechanism, 75
local node, 62
locating device programmatically, 55
locating information physically, 53
logical address, 52
memory resource address, 67
memory resource use macros, 70
origin, 37
physical location, 51
remote node, 64
resource reservation, 68
system initialization, 47
unsupported signals, 42
Peripheral Component Interconnect

See "PCI/PCI-X", 37
Physical address space, 21
PIO
address anatomy, 61
address mapping, 43
address translation, 65
addresses, 27
addressing, 29
addressing extension, 29
architecture, 61
CPU access, 33
mapped addresses, 61
operation flow, 62
resource management, 66
write operations, 70
Programmable I/O
See "PIO", 29
PX-brick, 14, 19, 37
PX-brick expansion, 38

R

Registers
ITC, 93
Release semantics, 102
Routines, DMA mapped, 84

S

SC-brick, 16, 38
SHub physical address map, 26
System physical memory space, 35

T

Time management
interval timer counter (ITC), 93

U

User virtual addresses, 90

V

Validity checking, 90

W

Write operations, 4