Origin2000™ Power-On Diagnostics

Document Number 108-0161-001

Contributors

Written by Darrin Goss
Illustrated by Darrin Goss
Edited by Cindi Leiser
Production by Mike Dixon
Engineering contributions by Chris Satterlee, Curt McDowell, Dawn Maxon, Mohan Natarajan, Curt Chambliss, Steve Whitney, Sue Liu, and Tom Kenville.

© 1996, Silicon Graphics, Inc.— All Rights Reserved

This document contains proprietary and confidential information of Silicon Graphics, Inc. The contents of this document may not be disclosed to third parties, copied, or duplicated in any form, in whole or in part, without the prior written permission of Silicon Graphics, Inc.

Restricted Rights Legend

Use, duplication, or disclosure of the technical data contained in this document by the Government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 52.227-7013 and/or in similar or successor clauses in the FAR, or in the DOD or NASA FAR Supplement. Unpublished rights reserved under the Copyright Laws of the United States. Contractor/manufacturer is Silicon Graphics, Inc., 2011 N. Shoreline Blvd., Mountain View, CA 94043-1389.

Origin2000 Power-On Diagnostics
Document Number 108-0161-001

Silicon Graphics, Inc.
Mountain View, California

Silicon Graphics and the Silicon Graphics logo are registered trademarks and IRIX and Origin2000 are trademarks of Silicon Graphics, Inc. R10000 is a trademark of MIPS Technologies, Inc.

Contents

About This Guide ..... **xix**

Notational Conventions xx

Terminology xx

1. Introduction **1-1**

1.1 Overview of Power-On Diagnostics 1-1

1.2 Modifying How the Power-On Diagnostics are Run 1-6

1.3 Console Input to the Power-On Diagnostics 1-7

1.4 Output From the Power-On Diagnostics 1-8

1.4.1 Node Board LEDs 1-8

1.4.2 Console Output 1-13

2. Boot Sequence **2-1**

2.1 Boot Sequence Steps in IP27 PROM Code 2-1

2.2 Boot Sequence Steps in BaseIO PROM Code 2-4

3. Invoking Power-On Diagnostics Manually **3-1**

3.1 Power-On Diagnostics From the POD Prompt 3-1

3.2 Power-On Diagnostics From the BaseIO Command Prompt 3-2

4. IP27 PROM Tests **4-1**

4.1 R10000 Processor Tests 4-1

4.1.1 Register Test 4-1

4.1.1.1 How to Run This Test 4-1

4.1.1.2 Test Description 4-2

4.1.1.3 Failure Information 4-2

4.1.2 Primary Cache Tests 4-2

4.1.2.1 How to Run These Tests 4-2

4.1.2.2 Test Descriptions 4-2

4.1.2.3 Failure Information 4-2

4.1.3 Interrupt Test 4-3

4.1.3.1 How to Run This Test 4-3

4.1.3.2 Test Description 4-3

4.1.3.3 Failure Information 4-3

4.2 Second-level Cache Tests 4-3
4.2.1 Secondary Cache Tests 4-3
4.2.1.1 How to Run These Tests 4-3
4.2.1.2 Test Descriptions 4-3
4.2.1.3 Failure Information 4-4
4.3 HUB Tests 4-4
4.3.1 GClk Logic Test 4-4
4.3.1.1 How to Run This Test 4-4
4.3.1.2 Test Description 4-4
4.3.1.3 Failure Information 4-4
4.3.2 BTE Test 4-4
4.3.2.1 How to Run This Test 4-5
4.3.2.2 Test Description 4-5
4.3.2.3 Failure Information 4-5
4.4 Local IO (Junk IO) Bus Device Test 4-5
4.4.1 PROM Checksum Test 4-5
4.4.1.1 How to Run This Test 4-5
4.4.1.2 Test Description 4-6
4.4.1.3 Failure Information 4-6
4.5 Memory/Directory Tests 4-6
4.5.1 Memory Configuration Test 4-6
4.5.1.1 How to Run This Test 4-6
4.5.1.2 Test Description 4-6
4.5.1.3 Failure Information 4-6
4.5.2 Backdoor Directory Space Test 4-7
4.5.2.1 Prerequisites to Running This Test 4-7
4.5.2.2 How to Run This Test 4-7
4.5.2.3 Test Description 4-7
4.5.2.4 Failure Information 4-7
4.5.3 Memory Test 4-8
4.5.3.1 Prerequisites to Running This Test 4-8
4.5.3.2 How to Run This Test 4-8
4.5.3.3 Test Description 4-8
4.5.3.4 Failure Information 4-8
4.6 CrayLink Tests 4-9
4.6.1 Local Link Connection Test 4-9
4.6.1.1 Prerequisites to Running This Test 4-9
4.6.1.2 How to Run This Test 4-9
4.6.1.3 Test Description 4-9
4.6.1.4 Failure Information 4-9

4.6.2 HUB Send Data Error Test 4-10
4.6.2.1 Prerequisites to Running This Test 4-10
4.6.2.2 How to Run This Test 4-10
4.6.2.3 Test Description 4-10
4.6.2.4 Failure Information 4-10
4.6.3 HUB Error Detection Test 4-10
4.6.3.1 How to Run This Test 4-11
4.6.3.2 Test Description 4-11
4.6.3.3 Failure Information 4-11
4.6.4 Router Send Data Error Test 4-11
4.6.4.1 Prerequisites to Running This Test 4-11
4.6.4.2 How to Run This Test 4-11
4.6.4.3 Test Description 4-12
4.6.4.4 Failure Information 4-12
4.6.5 Router Error Detection Test 4-12
4.6.5.1 How to Run This Test 4-12
4.6.5.2 Test Description 4-12
4.6.5.3 Failure Information 4-13
4.6.6 Get Chipid Test 4-13
4.6.6.1 How to Run This Test 4-13
4.6.6.2 Test Description 4-13
4.6.6.3 Failure Information 4-13
4.7 XBOW Test 4-13
4.7.1 Prerequisites to Running This Test 4-14
4.7.2 How to Run This Test 4-14
4.7.3 Test Description 4-14
4.7.4 Failure Information 4-14
4.8 Bridge Test 4-15
4.8.1 Prerequisites to Running This Test 4-15
4.8.2 How to Run This Test 4-15
4.8.3 Test Description 4-16
4.8.4 Failure Information 4-16
4.9 PCI Tests 4-17
4.9.1 PCICONFIG Test 4-17
4.9.1.1 Prerequisites to Running This Test 4-17
4.9.1.2 How to Run This Test 4-17
4.9.1.3 Test Description 4-17
4.9.1.4 Failure Information 4-17

4.9.2 PCI Bus Test 4-18
4.9.2.1 Prerequisites to Running This Test 4-18
4.9.2.2 How to Run This Test 4-18
4.9.2.3 Test Description 4-19
4.9.2.4 Failure Information 4-19
4.10 Serial Port Tests 4-19
4.10.1 Serial Programmed I/O Test 4-19
4.10.1.1 Prerequisites to Running This Test 4-19
4.10.1.2 How to Run This Test 4-20
4.10.1.3 Test Description 4-20
4.10.1.4 Failure Information 4-20
4.10.2 Serial DMA Test 4-21
4.10.2.1 Prerequisites to Running This Test 4-21
4.10.2.2 How to Run This Test 4-21
4.10.2.3 Test Description 4-22
4.10.2.4 Failure Information 4-22

5. BaseIO PROM Tests 5-1
5.1 SCSI Tests 5-1
5.1.1 SCSI SSRAM Test 5-2
5.1.1.1 Prerequisites to Running This Test 5-2
5.1.1.2 How to Run This Test 5-2
5.1.1.3 Test Description 5-2
5.1.1.4 Failure Information 5-3
5.1.2 SCSI DMA Test 5-3
5.1.2.1 Prerequisites to Running This Test 5-3
5.1.2.2 How to Run This Test 5-3
5.1.2.3 Test Description 5-3
5.1.2.4 Failure Information 5-3
5.1.3 SCSI Self-Test 5-4
5.1.3.1 Prerequisites to Running This Test 5-4
5.1.3.2 How to Run This Test 5-4
5.1.3.3 Test Description 5-4
5.1.3.4 Failure Information 5-4
5.2 Ethernet Tests 5-5
5.2.1 Comprehensive Ethernet Test 5-6
5.2.1.1 Prerequisites to Running This Test 5-7
5.2.1.2 How to Run This Test 5-7
5.2.1.3 Test Description 5-7
5.2.1.4 Failure Information 5-8

5.2.2 Ethernet SSRAM Test 5-9
5.2.2.1 Prerequisites to Running This Test 5-9
5.2.2.2 How to Run This Test 5-9
5.2.2.3 Test Description 5-9
5.2.2.4 Failure Information 5-10
5.2.3 PHY Register Test 5-10
5.2.3.1 Prerequisites to Running This Test 5-11
5.2.3.2 How to Run This Test 5-11
5.2.3.3 Test Description 5-11
5.2.3.4 Failure Information 5-11
5.2.4 TX Clock Test 5-12
5.2.4.1 Prerequisites to Running This Test 5-13
5.2.4.2 How to Run This Test 5-13
5.2.4.3 Test Description 5-13
5.2.4.4 Failure Information 5-13
5.2.5 NIC Test 5-13
5.2.5.1 Prerequisites to Running This Test 5-13
5.2.5.2 How to Run This Test 5-14
5.2.5.3 Test Description 5-14
5.2.5.4 Failure Information 5-14
5.2.6 IOC3 Internal Loopback Test 5-15
5.2.6.1 Prerequisites to Running This Test 5-15
5.2.6.2 How to Run This Test 5-15
5.2.6.3 Test Description 5-15
5.2.6.4 Failure Information 5-16
5.2.7 PHY Chip Internal Loopback Test 5-18
5.2.7.1 Prerequisites to Running This Test 5-18
5.2.7.2 How to Run This Test 5-18
5.2.7.3 Test Description 5-18
5.2.7.4 Failure Information 5-19
5.2.8 Twister Chip Internal Loopback Test 5-20
5.2.8.1 Prerequisites to Running This Test 5-21
5.2.8.2 How to Run This Test 5-21
5.2.8.3 Test Description 5-21
5.2.8.4 Failure Information 5-22
5.2.9 External Loopback DMA Test 5-22
5.2.9.1 Prerequisites to Running This Test 5-22
5.2.9.2 How to Run This Test 5-23
5.2.9.3 Test Description 5-23
5.2.9.4 Failure Information 5-24

5.2.10 External Loopback DMA (10 Mb/s) Test 5-24
5.2.10.1 Prerequisites to Running This Test 5-24
5.2.10.2 How to Run This Test 5-25
5.2.10.3 Test Description 5-25
5.2.10.4 Failure Information 5-26
5.2.11 External Loopback DMA (100 Mb/s) Test 5-26
5.2.11.1 Prerequisites to Running This Test 5-26
5.2.11.2 How to Run This Test 5-26
5.2.11.3 Test Description 5-27
5.2.11.4 Failure Information 5-27
5.2.12 XTALK Stress Test 5-28
5.2.12.1 Prerequisites to Running This Test 5-28
5.2.12.2 How to Run This Test 5-28
5.2.12.3 Test Description 5-28
5.2.12.4 Failure Information 5-29

6. POD Mode 6-1
6.1 Entering POD Mode 6-1
6.2 Dirty Exclusive, Uncached, and Cached Modes 6-1
6.2.1 Dirty Exclusive Mode 6-2
6.2.2 Uncached Mode 6-2
6.2.3 Cached Mode 6-2
6.3 POD Mode Prompt 6-2
6.4 Data Types 6-4
6.5 Constants 6-5
6.6 Expressions 6-5
6.7 Address Modifiers 6-6
6.8 Hardware Registers 6-7
6.9 Symbols 6-7
6.10 POD Mode Commands 6-8
6.10.1 Conventions 6-8
6.10.2 Quick Reference Descriptions 6-8
6.10.3 Printing a String 6-13
6.10.3.1 echo Command 6-13
6.10.4 Printing Expressions 6-14
6.10.4.1 px Command 6-14
6.10.4.2 pd Command 6-14
6.10.4.3 po Command 6-14
6.10.4.4 pb command 6-14
6.10.4.5 pa Command 6-14
6.10.4.6 nm Command 6-15

6.10.5 Displaying Registers 6-15
6.10.5.1 pr Command 6-15
6.10.5.2 pf Command 6-16
6.10.6 Loading From Memory and Storing to Memory 6-17
6.10.6.1 lb Command 6-17
6.10.6.2 lh Command 6-17
6.10.6.3 lw Command 6-17
6.10.6.4 ld Command 6-18
6.10.6.5 la Command 6-18
6.10.6.6 sb Command 6-18
6.10.6.7 sh Command 6-19
6.10.6.8 sw Command 6-19
6.10.6.9 sd Command 6-20
6.10.6.10 sdv Command 6-20
6.10.7 Setting Registers 6-20
6.10.7.1 sr Command 6-20
6.10.7.2 sf Command 6-21
6.10.8 Performing Operations Between Modules 6-21
6.10.8.1 vr Command 6-21
6.10.8.2 vw Command 6-22
6.10.8.3 vx Command 6-22
6.10.8.4 route Command 6-23
6.10.9 Controlling POD Command Execution 6-23
6.10.9.1 repeat Command 6-23
6.10.9.2 loop Command 6-24
6.10.9.3 while Command 6-24
6.10.9.4 for Command 6-24
6.10.9.5 if Command 6-25
6.10.9.6 delay Command 6-25
6.10.9.7 sleep Command 6-25
6.10.9.8 time Command 6-26
6.10.9.9 jump Command 6-26
6.10.9.10 call Command 6-26
6.10.9.11 reset Command 6-26
6.10.10 Working With Nonmaskable Interrupts (NMIs) 6-27
6.10.10.1 nmi Command 6-27
6.10.10.2 why Command 6-27
6.10.10.3 altregs Command 6-28
6.10.10.4 traplog Command 6-28

6.10.11 Getting Help Information 6-28
6.10.11.1 help Command 6-28
6.10.11.2 ? Command 6-28
6.10.11.3 version Command 6-28
6.10.12 Manipulating the Instruction, Data, and Secondary Caches 6-29
6.10.12.1 inval Command 6-29
6.10.12.2 flush Command 6-29
6.10.12.3 dtag Command 6-29
6.10.12.4 itag Command 6-29
6.10.12.5 stag Command 6-30
6.10.12.6 dline Command 6-30
6.10.12.7 iline Command 6-30
6.10.12.8 sline Command 6-30
6.10.12.9 adtag Command 6-30
6.10.12.10 aitag Command 6-31
6.10.12.11 astag Command 6-31
6.10.12.12 adline Command 6-31
6.10.12.13 ailine Command 6-31
6.10.12.14 asline Command 6-31
6.10.13 Manipulating the TLB 6-32
6.10.13.1 tlbc Command 6-32
6.10.13.2 tlbr Command 6-32
6.10.14 Setting the Operating Mode 6-32
6.10.14.1 go Command 6-32
6.10.15 Running the HUB BIST and Router BIST 6-33
6.10.15.1 bist Command 6-33
6.10.15.2 rbist Command 6-33
6.10.16 Enabling and Disabling Hardware 6-33
6.10.16.1 disable Command 6-33
6.10.16.2 enable Command 6-34
6.10.16.3 tdisable Command 6-34

6.10.17 Testing Memory 6-34
6.10.17.1 dirinit Command 6-34
6.10.17.2 meminit Command 6-35
6.10.17.3 dirtest Command 6-35
6.10.17.4 memtest Command 6-36
6.10.17.5 santest Command 6-37
6.10.17.6 maxerr Command 6-38
6.10.17.7 qual Command 6-38
6.10.17.8 ecc Command 6-38
6.10.17.9 error Command 6-38
6.10.17.10 clear Command 6-39
6.10.17.11 im Command 6-39
6.10.18 Manipulating the Memory Region 6-39
6.10.18.1 memset Command 6-39
6.10.18.2 memcpy Command 6-39
6.10.18.3 memcmp Command 6-40
6.10.18.4 memsum Command 6-40
6.10.18.5 reconf Command 6-40
6.10.19 Manipulating the Directory Region 6-40
6.10.19.1 scandir Command 6-40
6.10.19.2 dirstate Command 6-41
6.10.20 Running the Local Link Connection Test 6-41
6.10.20.1 chklink Command 6-41
6.10.21 Running the HUB Send Data Error Test 6-41
6.10.21.1 hubsde Command 6-41
6.10.22 Running the Router Send Data Error Test 6-41
6.10.22.1 rtrsde Command 6-41
6.10.23 Testing the XBOW 6-42
6.10.23.1 dgxbow Command 6-42
6.10.24 Testing the Bridge 6-42
6.10.24.1 dgbrdg Command 6-42
6.10.25 Testing the PCI Hardware 6-42
6.10.25.1 dgconf Command 6-42
6.10.25.2 dgpci Command 6-43
6.10.26 Testing the Serial Ports 6-43
6.10.26.1 dgspio Command 6-43
6.10.26.2 dgsdma Command 6-43
6.10.27 Accessing the Slave (Launch Loop) 6-43
6.10.27.1 slave Command 6-43
6.10.27.2 cpu Command 6-44

6.10.28 Accessing the BaseIO PROM 6-44
6.10.28.1 segs Command 6-44
6.10.28.2 exec Command 6-44
6.10.29 Choosing a Console 6-45
6.10.29.1 ioc3 Command 6-45
6.10.29.2 junk Command 6-45
6.10.29.3 msc Command 6-45
6.10.29.4 talk Command 6-45
6.10.30 Accessing the CrayLink Interconnect 6-46
6.10.30.1 disc Command 6-46
6.10.30.2 pcfg Command 6-46
6.10.30.3 node Command 6-46
6.10.30.4 rtab Command 6-46
6.10.30.5 rstat Command 6-47
6.10.31 Debugging I/O 6-47
6.10.31.1 crb Command 6-47
6.10.31.2 crbx Command 6-47
6.10.32 Flashing the PROM 6-47
6.10.32.1 flash Command 6-47
6.10.33 Reading the NIC Values 6-48
6.10.33.1 nic Command 6-48
6.10.33.2 rnic Command 6-48
6.10.33.3 modnic Command 6-49
6.10.34 Manipulating the System Controller 6-49
6.10.34.1 scw Command 6-49
6.10.34.2 scr Command 6-49
6.10.34.3 sc Command 6-49
6.10.34.4 dips Command 6-51
6.10.34.5 dbg Command 6-52
6.10.34.6 pas Command 6-52
6.10.34.7 module Command 6-52
6.10.35 Disassembling Memory 6-53
6.10.35.1 dis Command 6-53
6.10.35.2 kern_sym Command 6-53
6.10.36 Dumping Error Information 6-53
6.10.36.1 dumpspool Command 6-53
6.10.36.2 error_dump Command 6-53
6.10.36.3 edump_bri Command 6-54

6.10.37 Manipulating the PROM Log 6-54
6.10.37.1 initlog Command 6-56
6.10.37.2 initalllogs Command 6-57
6.10.37.3 log Command 6-57
6.10.37.4 rlog Command 6-57
6.10.37.5 setenv Command 6-57
6.10.37.6 unsetenv Command 6-58
6.10.37.7 printenv Command 6-58
6.10.38 Running the FRU Analyzer 6-59
6.10.38.1 fru Command 6-59
6.10.39 Debugging the Kernel 6-59
6.10.39.1 kdebug Command 6-59
6.10.40 Setting the Verbosity Level of the Boot Sequence 6-59
6.10.40.1 verbose Command 6-59

A. Register Names in POD Mode A-1
A.1 HUB Processor Interface Registers A-1
A.2 HUB Memory/Directory Registers A-3
A.3 HUB I/O Interface Registers A-4
A.4 HUB Network Interface Registers A-6
A.5 HUB Internal Crossbar Control Registers A-7
A.6 Midplane Crossbow Registers A-7
A.7 Router Registers A-8
A.8 R10000 Coprocessor 0 Registers A-9

Figures

Figure 1-1 Reading the LED Patterns (Example) 1-8

Tables

Table 1-1 POD Modes 1-2
Table 1-2 IP27 PROM Tests 1-3
Table 1-3 BaseIO PROM Tests 1-5
Table 1-4 DIP Switch Assignments 1-6
Table 1-5 Consoles From Which the Power-On Diagnostics Can Receive Input 1-7
Table 1-6 Node Board LED Values 1-9
Table 1-7 LED Status Values for Exceptions 1-12
Table 2-1 System Maintenance Menu Options 2-5
Table 3-1 Power-On Diagnostics That Run From the POD Prompt 3-1
Table 3-2 Power-On Diagnostics That Run From the BaseIO Command Prompt 3-2
Table 5-1 IOC3 Loopback Test Failure Information Based on the EISR Values 5-16
Table 5-2 PHY Chip Internal Loopback Test Failure Information Based on the EISR Values 5-20
Table 6-1 Components of the POD Mode Prompt 6-2
Table 6-2 POD Mode Data Types 6-4
Table 6-3 Possible Suffixes for POD Mode Constants 6-5
Table 6-4 POD Mode Address Modifiers 6-6
Table 6-5 POD Mode Commands (Quick Reference) 6-8
Table 6-6 MSC Commands 6-50
Table 6-7 PROM Environment Variables 6-54

About This Guide

This document describes the power-on diagnostics (POD) for the Origin2000 series computer systems. It is written for System Support Engineers (SSEs) to use as a reference document in training and on-site.

The information contained in this document is organized into the following chapters:

- Chapter 1, "Introduction," provides an overview of and introduction to the power-on diagnostics. This Chapter introduces the various POD modes, the IP27 PROM tests, the BaseIO PROM tests, the debug DIP switches, how to send input to the power-on diagnostics, and how the power-diagnostics return output.
- Chapter 2, "Boot Sequence," describes the Origin2000 boot sequence and what the power-on diagnostics do during the boot sequence.
- Chapter 3, "Invoking Power-On Diagnostics Manually," lists the power-on diagnostics that you can manually run from the PROM monitor (either from the POD prompt or from the BaseIO command prompt).
- Chapter 4, "IP27 PROM Tests," describes the power-on diagnostic tests that are included in the IP27 PROM.
- Chapter 5, "BaseIO PROM Tests," describes the power-on diagnostic tests that are included in the BaseIO PROM.
- Chapter 6, "POD Mode," describes POD mode, an interactive mode that uses the command line interpreter that is in the PROM.
- Appendix A, "Register Names in POD Mode," lists all of the available register names that you can use in expressions or commands in POD mode.

Note: This document supports PROM version 5.2; the POD mode *version* command shows the PROM version that is loaded on your system.

Notational Conventions

This document uses the following notational conventions:

- Information displayed on the screen is shown in `Courier` type.
- Commands that you should enter are shown in **`Courier bold`** type.
- Commands in text are shown in *italic* type.
- Variables are shown in *italic* type.

Terminology

This document uses the following terms interchangeably:

- Bridge, Bridge ASIC, and Bridge chip
- HUB, HUB ASIC, and HUB chip
- Crossbow, XBOW, XBOW ASIC, and XBOW chip

Chapter 1

Introduction

This chapter provides an overview of the power-on diagnostics (POD) for Origin2000 systems. The power-on diagnostics test the components necessary to load the operating system or the micro-diagnostic kernel (MDK). These diagnostics run as part of the normal boot sequence; you can also run several of these diagnostics from the IP27 PROM POD mode command line or the BaseIO PROM command monitor.

1.1 Overview of Power-On Diagnostics

The power-on diagnostics (POD) are programmable read-only memory (PROM) resident routines that run automatically during the boot sequence. You can also run several of the power-on diagnostics manually from the IP27 PROM POD mode command line or the BaseIO PROM command monitor.

Power-on diagnostics focus on the hardware needed to boot the operating system or the micro-diagnostic kernel (MDK). The routines that the power-on diagnostics use are stored in the IP27 PROM and the BaseIO PROM.The POD diagnostics run in four modes: no diags mode, normal mode, heavy mode, and manufacturing mode. Table 1-1 describes the modes and the DIP switch settings you can use to select the modes. Refer to Section 1.2, "Modifying How the Power-On Diagnostics are Run," for more information about how to set the DIP switches.

Table 1-1 POD Modes

Mode	Description[a]
No Diags	This mode performs minimal testing; only the initialization functions of the power-on diagnostics (for example, clearing memory) are performed. (This mode also checks the memory configurations and the real-time clock.) To set this mode, put DIP switch one in the On position and DIP switch two in the Off position.
Normal	This mode performs testing to ensure that the operating system or MDK can run. Because of the time constraints necessary in the normal boot process, this mode performs a subset of the testing done by heavy mode. To select this mode, put both DIP switches in the Off position.
Heavy	This mode performs more extensive and time-consuming testing than normal mode. Use this mode the first time that you power on a system in the field or when the normal mode detects errors. To select this mode, put DIP switch one in the Off position and DIP switch two in the On position.
Manufacturing	This mode provides more detailed status and failure information than heavy mode. This mode may run more extensive tests than heavy mode. (Normally, this mode is used only in the manufacturing environment.) To select this mode, put both DIP switches in the On position.

a. The DIP switches described in this table are located on a blue block next to the module system controller (MSC) keyswitch. Numbering of the DIP switches starts from the left and starts with number one, so DIP switch number one is the first switch on the left and DIP switch number two is the second switch from the left. Refer to Section 1.2, "Modifying How the Power-On Diagnostics are Run."

The power-on diagnostics are divided into two sections: IP27 tests and BaseIO tests. Table 1-2 lists the IP27 PROM tests by the functional unit that they test. Table 1-3 lists the BaseIO PROM tests by the functional unit that they test. The "Page" column in the tables indicates the page on which you can find a detailed description of the test.

Table 1-2 IP27 PROM Tests

Functional Unit	Test	Failure LED Pattern	Failure Message	Failing FRU	Page
R10000 processors	Register test	0x81 (FLED_CP1)	n/a	Node board	4-1
	Primary instruction cache test	0x83 (FLED_ICACHE)	n/a	Node board	4-2
	Primary data cache test	0x84 (FLED_DCACHE)	n/a	Node board	4-2
	Interrupt test	0x90 (FLED_HUBINTS)	`RSLT hub_int_diag FAIL`	Node board	4-3
Second-level cache	Secondary cache tests	0x85 (FLED_SCACHE)	`RSLT cache_test_s FAIL`	Node board	4-3
HUB	GClk logic test	n/a	`RSLT rt_clock_test FAIL`	Node board	4-4
	BTE test	n/a	`RSLT hub_bte_diag FAIL`	Node board	4-4
Local IO (junk IO) bus	PROM checksum test	n/a	`RSLT prom_checksum FAIL`	Node board	4-5
Memory and directory	Memory configuration test	0x98 (FLED_NOMEM) or 0xa9 (FLED_BADMEM)	`No Memory Found`, `*** Mixed standard and premium memory`, or `nasid` *node_number*`: disabling bank` *bank_number*	DIMM	4-6
	Backdoor directory space test	n/a	`RSLT dirtest FAIL` or `MEMORY FAILURE`	DIMM	4-7
	Memory test	n/a	`RSLT memtest FAIL` or `MEMORY FAILURE`	DIMM	4-8
CrayLink	Local link connection test	n/a	`RSLT chk_local_link FAIL`	Midplane, Node board, or Router	4-9
	HUB send data error test	n/a	`RSLT hub_send_data_error FAIL`	Router, Node board, or midplane	4-10

Table 1-2 (continued) IP27 PROM Tests

Functional Unit	Test	Failure LED Pattern	Failure Message	Failing FRU	Page
CrayLink (continued)	HUB error detection test	n/a	`RSLT test_hub_error_detection FAIL`	Node board, Router, or midplane	4-10
	Router send data error test	n/a	`RSLT rtr_send_data_error FAIL`	Node board, Router, or midplane/cable	4-11
	Router error detection test	n/a	`RSLT test_rtr_error_detection FAIL`	Router, Node board, or midplane/cable	4-12
	Get chipid test	n/a	`RSLT get_chipid FAIL`	Node board or Router	4-13
XBOW	XBOW test	n/a	`RSLT xbow_sanity FAIL`	Midplane or Node board	4-13
Bridge	Bridge test	n/a	`RSLT bridge_sanity FAIL`	BaseIO board	4-15
PCI	PCICONFIG test	n/a	`RSLT io6config_space FAIL`	BaseIO board	4-17
	PCI bus test	n/a	`RSLT pcibus_sanity FAIL`	BaseIO board	4-18
Serial port	Serial programmed I/O test	n/a	`RSLT serial_pio FAIL`	BaseIO board	4-19
	Serial DMA test	n/a	`RSLT serial_dma FAIL`	BaseIO board	4-21

Table 1-3 BaseIO PROM Tests

Functional Unit	Test	Failure Message	Failing FRU	Page
SCSI	SCSI SSRAM test	`RSLT scsi_ram FAIL`	BaseIO board	5-2
	SCSI DMA test	`RSLT scsi_dma FAIL`	BaseIO board	5-3
	SCSI self-test	`RSLT scsi_controller`	BaseIO board	5-4
Ethernet	Comprehensive Ethernet test	`RSLT enet_all FAIL`	BaseIO board	5-6
	Ethernet SSRAM test	`RSLT enet_ssram FAIL`	BaseIO board	5-9
	PHY register test	`RSLT enet_phy_reg FAIL`	BaseIO board	5-10
	TX clock test	`RSLT enet_tx_clk FAIL`	BaseIO board	5-12
	NIC test	`RSLT enet_nic FAIL`	BaseIO board	5-13
	IOC3 internal loopback DMA test	`RSLT enet_ioc3_loop FAIL`	BaseIO board	5-15
	PHY chip internal loopback test	`RSLT enet_phy_reg FAIL`	BaseIO board	5-18
	Twister chip internal loopback test	`RSLT enet_tw_loop FAIL`	BaseIO board	5-20
	External loopback DMA test	`RSLT enet_ext_loop FAIL`	BaseIO board	5-22
	External loopback DMA (10 Mb/s) test	`RSLT enet_10_ext_loop FAIL`	BaseIO board	5-24
	External loopback DMA (100 Mb/s) test	`RSLT enet_100_ext_loop FAIL`	BaseIO board	5-26
	XTALK stress test	`RSLT enet_xtalk_stress FAIL`	BaseIO board	5-28

1.2 Modifying How the Power-On Diagnostics are Run

Use the debug DIP switches that are mounted on a blue block next to the module system controller (MSC) keyswitch to modify how the power-on diagnostics are run. The switch numbering is 1 to 8, going from left to right. Table 1-4 describes the function of each switch. You should set all switches to Off for normal system operations. Any changes you make to the DIP switch settings take effect the next time you boot the system.

Table 1-4 DIP Switch Assignments

Switch	Description[a]
1	Selects the diagnostic mode (with switch 2)
2	Selects the diagnostic mode (with switch 1) Switches 1 and 2 set the mode as follows: Turning switch 1 Off and switch 2 Off selects "normal" mode. Turning switch 1 Off and switch 2 On selects "heavy" mode. Turning switch 1 On and switch 2 Off selects "no diags" mode. Turning switch 1 On and switch 2 On selects "manufacturing" mode. For more information about the diagnostic modes, refer to Table 1-1.
3	Selects the boot information level When switch 3 is set to On, the PROM code displays detailed information messages during the boot sequence. When switch 3 is set to Off, the PROM code displays only normal boot status messages.
4	Selects the boot stop point (with switch 5)
5	Selects the boot stop point (with switch 4), Switches 4 and 5 set the boot stop point as follows: Turning switch 4 Off and switch 5 Off sets the boot stop point to "never." This setting causes the boot process to completely boot the operating system. Turning switch 4 Off and switch 5 On sets the boot stop point to "local." This setting causes the boot process to stop at the point where it would normally enter the BaseIO PROM code. All Nodes enter cached (Cac) POD mode. This setting affects all modules in the system. Turning switch 4 On and switch 5 Off sets the boot stop point to "global." This setting causes the boot process to stop at the point where it normally would jump to the BaseIO PROM code. Then, the master Node enters cached (Cac) POD mode, and the slave Nodes enter the slave loop. This setting affects all modules in the system. Turning switch 4 On and switch 5 On sets the boot stop point to "memoryless." This setting causes the boot process to stop as soon as the system is able to enter POD mode. All Nodes enter dirty exclusive (Dex) POD mode.
6	Selects the default environment When switch 6 is set to On, the PROM code ignores all PROM log environment variables and all BaseIO NVRAM settings; instead, the PROM code uses the system defaults. When switch 6 is set to Off, the PROM code uses all PROM log environment variables and all BaseIO NVRAM settings. It is useful to set switch 6 to On if any of the variable storage devices contain data that prevents the system from booting.

Table 1-4 (continued) DIP Switch Assignments

Switch	Description[a]
7	Specifies whether or not to bypass the first BaseIO board in the system If switch 7 is to On, the PROM code bypasses the first BaseIO board that it finds in the system and tries to boot from the second BaseIO board. If switch 7 is set to Off, the PROM code tries to boot from the first BaseIO board. It is useful to set switch 7 to On if the first BaseIO board in the system is not working; this enables you to boot the system without removing the failing BaseIO board.
8	Specifies whether or not to bypass the global master If switch 8 is set to On, the Node that would normally become the global master becomes the slave Node; the next CPU in line becomes the global master.

a. You should set all switches to Off for normal operation.

The MSC NVRAM includes two sets of debug switch values: eight physical switch values (numbered 1 to 8) and eight virtual debug switch values (numbered 9 to 16). You can view and change the debug switch settings with the MSC *dbg* command (refer to Table 6-6 in Section 6.10.34.3) and with the POD mode *dips* command (refer to Section 6.10.34.3, "sc Command,") and *dbg* command (refer to Section 6.10.34.5, "dbg Command,").

1.3 Console Input to the Power-On Diagnostics

The power-on diagnostics can interact with different types of consoles, depending on how your system is configured. Table 1-5 describes the three console types that are available.

Table 1-5 Consoles From Which the Power-On Diagnostics Can Receive Input

POD Console	Description
IOC3 UART	If a BaseIO board is available with a terminal connected to a port marked for the system console, POD expects input from the IOC3 UART on the BaseIO board. (LED pattern: 0x80 0x8c)
MSC UART	If no IOC3 is available and the MSC is responsive, POD expects input from the alternate diagnostic console that is connected to the MSC UART. (LED pattern: 0x80 0xbc)
Netuart	If none of the physical UARTs are available, POD expects input from a netuart. A netuart emulates a physical UART to enable one HUB to talk to another. When a CPU uses a netuart, it monitors shared memory locations and interrupt lines that it uses for communication. From any CPU that has a UART and is at the POD prompt, you can use the *talk* command to communicate with CPUs that are using a netuart.

POD mode checks for the consoles in the order shown in Table 1-5. When a CPU is at the POD prompt, the LED pattern for the selected console blinks at 0.5 Hz on the LEDs for the CPU.

1.4 Output From the Power-On Diagnostics

The power-on diagnostics send output to the Node board LEDs and to a console.

1.4.1 Node Board LEDs

Each Node board has two side-by-side columns of eight LEDs. While the system is booting, the Node board LEDs indicate the progress of the boot. If the system crashes while it is booting, you can use the value shown by the LEDs to determine where in the boot process the crash occurred. The left column of LEDs indicates the status of CPU A on the Node board; the right column of LEDs indicates the status of CPU B.

The columns of LEDs have the most significant bit at the top. An illuminated LED indicates a zero value for that position; an unilluminated LED indicates a one value. Figure 1-1 shows several examples of reading the LED values.

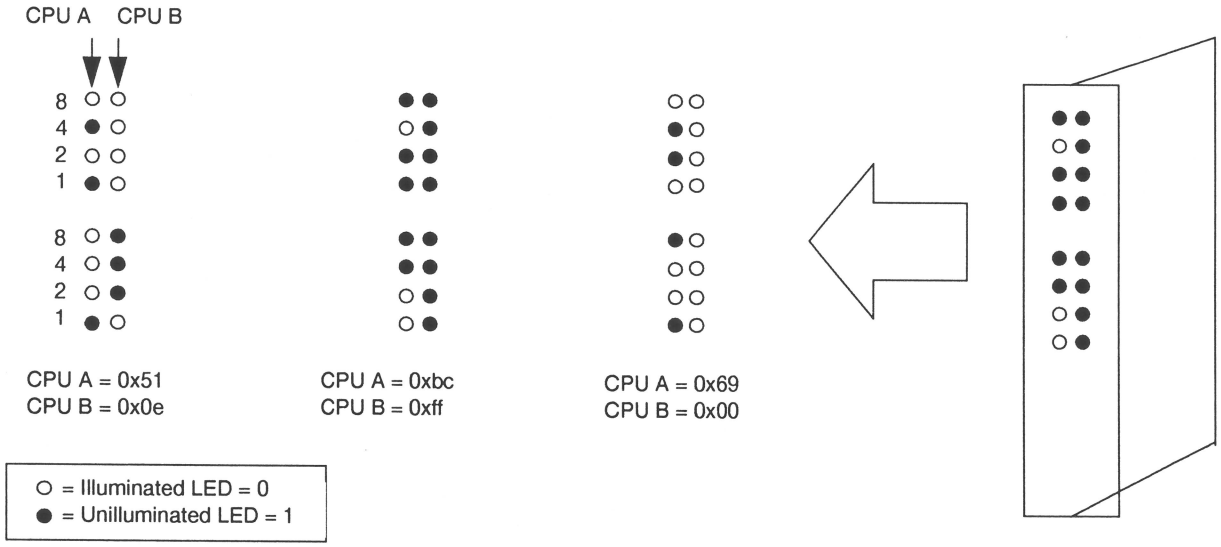

Figure 1-1 Reading the LED Patterns (Example)

Notice that Figure 1-1 illustrates how the values for the LEDs are inverted from what you might expect them to be. An illuminated LED has a value of 0, and an unilluminated LED has a value of 1.

Table 1-6 describes the values that appear on a Node Board's LEDs during the boot process and while the power-on diagnostics are running.

Table 1-6 Node Board LED Values

LED Value	Name	Description
0x01	PLED_INITCPU	Initializing the R10000™ general-purpose registers (GPRs), floating-point registers (FPR), and COP0 registers
0x02	PLED_TESTCP1	Performing the register test
0x03	PLED_RUNTLB	Switching to mapped mode
0x04	PLED_TESTICACHE	Performing the primary instruction cache test
0x05	PLED_TESTDCACHE	Performing the primary data cache test
0x06	PLED_TESTSCACHE	Performing the secondary cache test
0x07	PLED_FLUSHCACHES	Flushing all caches
0x0a	PLED_INVICACHE	Invalidating R10000 primary instruction cache
0x0b	PLED_INVDCACHE	Invalidating R10000 primary data cache
0x0c	PLED_INVSCACHE	Invalidating secondary cache
0x0d	PLED_INMAIN	Successfully jumped to the **main()** function
0x0e	PLED_SPEEDUP	Preparing to increase PROM access speed
0x0f	PLED_SPEEDUPOK	Successfully increased PROM access speed
0x1a	PLED_ELSCPROBE	Preparing to probe for presence of MSC
0x1c	PLED_DONEPROBE	Done probing for presence of MSC
0x1d	PLED_UARTINIT	Preparing to initialize selected UART
0x1e	PLED_UARTINITDONE	Done initializing selected UART
0x21	PLED_PODLOOP	Preparing to enter POD mode (C code portion)
0x22	PLED_PODPROMPT	Preparing to enter the POD prompt loop
0x23	PLED_PODMODE	Preparing to enter POD mode (assembly code portion)
0x24	PLED_LOCALARB	Performing local arbitration (between CPU A and CPU B)
0x28	PLED_BARRIER	Preparing to perform first local barrier
0x2a	PLED_MAKESTACK	Preparing to configure Dex mode stack and data
0x2b	PLED_MAIN	Code execution has reached the **main()** function
0x2f	PLED_INSLAVE	Waiting for commands from master CPU
0x31	PLED_NMI	NMI occurred
0x35	PLED_RTCINIT	Preparing to initialize the HUB real-time counter

Table 1-6 (continued) Node Board LED Values

LED Value	Name	Description
0x36	PLED_RTCINITDONE	Done initializing the HUB real-time counter
0x38	PLED_BARRIEROK	Successfully completed the first local barrier operation
0x3c	PLED_JUMPRAMU	Preparing to jump to UALIAS space
0x3d	PLED_JUMPRAMUOK	Successfully jumped to UALIAS space
0x3e	PLED_JUMPRAMC	Preparing to jump to cached space
0x3f	PLED_JUMPRAMCOK	Successfully jumped to cached space
0x40	PLED_STACKRAM	Preparing to test the stack area of memory
0x41	PLED_STACKRAMOK	Successfully tested the stack area of memory
0x42	PLED_SLAVEINT	Received interrupt message from master CPU
0x43	PLED_SLAVECALL	Preparing to execute a command from the master CPU
0x44	PLED_SLAVEREND	Completed executing command from the master CPU
0x45	PLED_LAUNCHLOOP	Preparing to enter the slave launch loop
0x46	PLED_LAUNCHINTR	Received a launch interrupt
0x47	PLED_LAUNCHCALL	Calling the **launched()** function
0x48	PLED_LAUNCHDONE	Returned from the **launched()** function
0x4a	PLED_MDIRINIT	Preparing to initialize the HUB MD and SIMM controls
0x4b	PLED_MDIRCONFIG	Preparing to determine and configure the memory size
0x4c	PLED_I2CINIT	Preparing to initialize the PCF8584 I2C chip
0x4d	PLED_I2CDONE	Completed initializing the PCF8584 I2C chip
0x4f	PLED_IODISCOVER	Preparing to discover HUB I/O
0x50	PLED_HUB_CONFIG	Updating config structures for HUBs and for Routers without HUBs
0x51	PLED_ROUTER_CONFIG	Preparing to write the Router configuration data into the KLCONFIG structure
0x52	PLED_INITIO	Preparing to initialize the I/O interface (II) on the HUB
0x53	PLED_CONSOLE_GET	Preparing to probe the I/O section to locate the console
0x54	PLED_CONSOLE_GET_OK	Successfully located the console
0x56	PLED_INITIODONE	Successfully initialized the II on the HUB

Table 1-6 (continued) Node Board LED Values

LED Value	Name	Description
0x57	PLED_STASH2	Preparing to clear the HUB error registers
0x58	PLED_STASH3	Preparing to enable error checking
0x59	PLED_STASH4	Completed enabling error checking
0x5a	PLED_IODISCOVER_DONE	Completed discovering HUB I/O
0x5b	PLED_NMI_INIT	Preparing to initialize the NMI handler area
0x5c	PLED_TEST_INTS	Preparing to test HUB interrupts
0x5d	PLED_IORESET	Performing early reset of HUB I/O section
0x81	FLED_CP1	Register test failed (The failing FRU is the Node board; refer to Section 4.1.1, "Register Test," for more information.)
0x82	FLED_RESTART	Restart master was unable to load the BaseIO PROM
0x83	FLED_ICACHE	Primary instruction cache test failed (The failing FRU is the Node board; refer to Section 4.1.2, "Primary Cache Tests," for more information.)
0x84	FLED_DCACHE	Primary data cache test failed (The failing FRU is the Node board; refer to Section 4.1.2, "Primary Cache Tests," for more information.)
0x85	FLED_SCACHE	Secondary cache tests failed (The failing FRU is the Node board; refer to Section 4.2.1, "Secondary Cache Tests," for more information.)
0x87	FLED_RTC	Real-time counter is not working correctly
0x8f	FLED_OS	OS requested LEDs
0x90	FLED_HUBINTS	Interrupt test failed (The failing FRU is the Node board; refer to Section 4.1.3, "Interrupt Test," for more information.)
0x91	FLED_HUBLOCAL	HUB local failed
0x93	FLED_PREM_DIR_REQ	Some Node does not have premium memory
0x97	FLED_MAINRET	Returned from **main()** function
0x98	FLED_NOMEM	Node Board does not have local memory
0x9a	FLED_DISABLED	CPU is disabled by an environment variable
0x9b	FLED_DOWNLOAD	Memory download failed

Table 1-6 (continued) Node Board LED Values

LED Value	Name	Description
0x9c	FLED_COREDEBUG	Boot process cannot set the CORE debug registers
0x9d	FLED_IODISCOVER	I/O discovery process failed
0x9e	FLED_HUB_CONFIG	Failure occurred while writing the HUB information into the KLCONFIG structure
0x9f	FLED_ROUTER_CONFIG	Failure occurred while writing the Router information into the KLCONFIG structure
0xa0	FLED_HUBIO_INIT	Failure occurred while trying to initialize the HUB I/O interface
0xa1	FLED_CONFIG_INIT	Failure occurred while trying to initialize the KLCONFIG structure
0xa4	FLED_HUBBIST	HUB LBIST or HUB ABIST test failed (The failing FRU is the Node board.)
0xa6	FLED_RESETWAIT	Waiting for a reset
0xa7	FLED_LLP_FAIL	LLP failed after a reset
0xa8	FLED_LLP_NORESET	LLP did not come up after a reset
0xa9	FLED_BADMEM	Local memory is bad
0xab	FLED_NET_DISCOVER	Network discovery failed
0xac	FLED_NASID_CALC	Calculation of NASID failed
0xad	FLED_ROUTE_CALC	Calculation of route failed
0xae	FLED_ROUTE_DIST	Distribution of route failed
0xaf	FLED_NASID_DIST	Distribution of NASID failed
0xb0	FLED_NO_NASID	Master was not assigned a NASID
0xb1	FLED_NO_MODULEID	Module ID arbitration failed
0xb2	FLED_MIXED_SN00	SN0 mixed with SN00

Table 1-7 shows the LED values that blink on the Node board LEDs when an exception occurs before there is a tty port available to receive information about the exception. If the LEDs on a Node board display any of these patterns, disable or replace the Node board because it contains failing hardware.

Table 1-7 LED Status Values for Exceptions

LED Value	Name	Exception Type
0xf2	EXC_GENERAL	General exception
0xf3	EXC_ECC	ECC exception
0xf4	EXC_TLB	TLB exception

Table 1-7 (continued) LED Status Values for Exceptions

LED Value	Name	Exception Type
0xf5	EXC_XTLB	XTLB exception
0xf6	EXC_UNIMPL	Unimplemented exception
0xf7	EXC_CACHE	Cache error exception

1.4.2 Console Output

The power-on diagnostics also return failure messages to the alternate console port (ACP) on the MSC and to the system console. For more information about the messages, refer to the descriptions of the tests later in this document.

Chapter 2

Boot Sequence

This chapter describes the boot sequence. The boot sequence occurs whenever you power up or reset the system. It uses code from the IP27 PROM and, optionally, the BaseIO PROM. The appropriate power-on diagnostics run automatically as the boot sequence discovers each hardware component in the system.

2.1 Boot Sequence Steps in IP27 PROM Code

The code in the IP27 PROM performs the following actions.

1. Initialize the CPU(s).
   - Set up the R10000 status registers.
   - Clear the register files.
2. Test the CPU caches.
   - Test the instruction cache.
   - Test the data cache.
3. Set up Dex mode.
   - Start treating the CPU data cache as memory.
   - Initialize the stack.
   - Execute the boot procedure.
4. Disable CPU A and/or CPU B according to the DisableA and DisableB environment variables.
5. Arbitrate the local master CPU (one per Node board).
   - Check for the presence of another CPU on the Node board.
   - Time out if the other CPU does not respond.
   - Disable the other CPU if it does not respond.
6. Read the debug DIP switch settings from the module system controller (MSC).
7. Determine the initial console device and initialize it.

   Normally the initial console port is the alternate console port (ACP) on the MSC.

Once the initial console device is initialized, the boot sequence can print detailed boot status information to the console device.

8. Record error information that may indicate the cause of a prior crash.
   - Save the state of HUB error registers in the cache.
   - Clear the state of HUB error registers.
   - Enable SYSAD error checking.
9. Initialize I/O.
   - Initialize the I/O interface (II) section of the HUB.
   - Initialize the Bridge.
   - Initialize the PCI bus.
10. Determine if there is a BaseIO card with a console port. If there is, switch the console to the terminal connected to the BaseIO console port.
11. Display the PROM boot banner on the console.

    The PROM boot banner includes the following information: IP27 PROM version and size, chip revisions, slot ID values, any CPUs that are present in the system, and other miscellaneous information.
12. Display which Debug Switch settings are set to other than the default; this information appears on the console.
13. Determine the serial number of the Node and send it to the other Nodes.
14. Configure local memory.
    - Initialize the SIMM control registers.
    - Determine the amount and type (premium or standard) of memory in the system.
    - Program the HUB memory configuration registers.
15. Perform basic memory tests.
    - Make sure address 0 of each bank is accessible.
    - Disable any banks that are not accessible.
    - If bank 0 is bad, swap it with a good bank (use the SwapBank environment variable).
16. Download the PROM to memory
    - Perform a memory test using a small portion of PROM data.
    - If the memory test passes, download a copy of the entire PROM to memory.
    - Verify the download checksum to ensure that the PROM data was downloaded successfully.
17. Transfer the program counter to uncached memory.

    Execution from memory is significantly faster than running out of the PROM. (The boot process still does not require the secondary cache.)
18. Move crucial data structures from the cache into uncached memory.
19. Test and invalidate the secondary cache.

20. Transfer the stack to uncached memory.
    - Discard the data (from Dex mode) contained in the data cache.
21. Transfer the program counter and stack to cached memory.

    Execution of the PROM code now runs at top speed.
22. Initialize the first 32 MB of bank 0 memory for use by the PROM.
23. Initialize permanent low-memory system data structures: KLDIR (the indirection table for accessing all other low-memory structures), NMI (the nonmaskable interrupt vector area), and KLCONFIG (the system topology and configuration information).
24. Run diagnostics on the local CrayLink port.
25. Discover the CrayLink Interconnect topology.
    - Perform a depth-first search using vector routing operations.
    - Build the promcfg data structure in low cached memory.
26. Verify that all PROMs are running the same firmware version.
27. Arbitrate the global master. (Use a protocol that uses vector routing operations and allows one global master per partition.)
28. Have the global master configure the CrayLink Interconnect.
    - Determine the Node ID for each Node (not yet assigned).
    - Program the CrayLink routing tables in all of the Hubs and Routers.
    - Tell each Node what its Node ID will be.
29. Switch all Nodes back to uncached memory, which is required to change Node IDs.
    - Flush the caches.
    - Closely synchronize all Nodes.
    - Quiesce bus activity with tight loops that are internal to the R10000 processor.
30. Change the Node IDs.
    - Update data structures that were assuming the Node ID was zero.
    - Reinitialize coherence directories for the first 32 MB of memory
31. Switch back to cached memory in the new Node ID space.
32. Test and initialize all of the rest of local memory (above 32 MB).
33. Initialize any headless Nodes (Nodes without functional CPUs).
    - Disable the CPUs.
    - Read the NIC.
    - Probe, configure, and initialize memory.
    - Initialize the low-memory data structures.
    - Initialize the I/O.
34. If the system is booting because of a reset and not because it was just powered on, display the error state information from the error registers.

- Decode the HUB error registers and display detailed information about their contents.

35. Transfer control to the BaseIO PROM code.

    Note: This step may be inhibited by the debug DIP switch settings. Refer to Section 1.2, "Modifying How the Power-On Diagnostics are Run," for more information about the debug DIP switches.

    - Retrieve a copy of the BaseIO PROM image from the master BaseIO board. If the master BaseIO board is not available, retrieve the copy of the BaseIO PROM image that is stored in the IP27 PROM.
    - Decompress the BaseIO PROM image and load it into memory.
    - Continue code execution at the entry point of the BaseIO PROM code.

2.2 Boot Sequence Steps in BaseIO PROM Code

When code execution reaches the entry point of the BaseIO PROM code, the code from the BaseIO PROM performs the following actions.

1. Initialize the global data structures.
2. Number the Nodes in the system.
3. Synchronize the Node clocks.
4. Compare the current system inventory with the stored system inventory.
5. Initialize and test the SCSI buses, console port, Ethernet port, graphics tube, and audio.
6. Fetch the environment variables and boot the IRIX™ operating system. When the system is ready to boot IRIX, the BaseIO PROM code displays the following menu:

```
System Maintenance Menu

1) Start System
2) Install System Software
3) Run Diagnostics
4) Recover System
5) Enter Command Monitor

Option?
```

The selection that you make from the System Maintenance Menu determines what happens next in the boot process. Table 2-1 describes each of the menu options.

Table 2-1 System Maintenance Menu Options

Option	Description
`1) Start System`	This option causes the system to boot IRIX in the default way.
`2) Install System Software`	This option is used when system software needs to be installed or upgraded. The PROM first attempts to find a tape drive on the system and if one is found, it prompts the user to insert the installation tape in it. If a tape device is not found, then installation is expected to take place through an Ethernet connection. In this case, the PROM prompts the user for the name of the system that will be used as the server.
`3) Run Diagnostics`	This option invokes the extended hardware diagnostic program, which performs a thorough test of the CPU board and any graphics boards present. It reports a summary.
`4) Recover System`	This option can be used to perform special system administration tasks such as restoring a system disk from backup tapes. It follows a sequence similar to installing system software, but instead of starting the installation program, it invokes an interactive restoration tool.
`5) Enter Command Monitor`	Additional functions can be performed from an interactive command monitor. This option puts the PROM into a manual mode of operation. It displays the BaseIO command prompt, which is the >> prompt.

Chapter 3

Invoking Power-On Diagnostics Manually

Besides having the power-on diagnostics run automatically during the boot sequence, you can also run severally manually in an interactive mode. This mode, called POD mode, lets you select which tests run; you normally use POD mode to debug a crashed system. (For more information about POD mode, refer to Chapter 6, "POD Mode.")

This chapter lists the power-on diagnostics that you can run manually (either from the POD prompt or from the BaseIO command prompt).

3.1 Power-On Diagnostics From the POD Prompt

Table 3-1 lists the power-on diagnostics that you can run from the POD prompt and indicates the command(s) to use to start the tests. For more information about each of the tests (including descriptions of the command line options), refer to the page shown in the "Page" column.

Table 3-1 Power-On Diagnostics That Run From the POD Prompt

Diagnostic Test	Command(s) to Run the Test	Page
Local link connection test	*chklink*	4-9
HUB send data error test	*hubsde*	4-10
Router send data error test	*rtrsde*	4-11
XBOW test	*dgxbow* [mn | mh | mm] [*nNODE*]	4-13
Bridge test	*dgbrdg* [mn | mh | mm] [*nNODE*] [*sSLOT*]	4-15
PCICONFIG test	*dgconf* [mn | mh | mm] [*nNODE*] [*sSLOT*]	4-17
PCI bus test	*dgpci* [mn | mh | mm] [*nNODE*] [*sSLOT*] [p*PCINUM*]	4-18
Serial programmed I/O test	*dgspio* [mn | mh | mm | mx] [*nNODE*] [s*SLOT*] [p*PCINUM*]	4-19
Serial DMA test	*dgsdma* [mn | mh | mm | mx] [*nNODE*] [s*SLOT*] [p*PCINUM*]	4-21

3.2 Power-On Diagnostics From the BaseIO Command Prompt

The power-on diagnostics that reside in the BaseIO PROM do not run from the `POD>` prompt; you must run these diagnostics from the BaseIO command prompt, which is >>. You can access the >> prompt by selecting the `5. Enter Command Monitor` option of the System Maintenance Menu.

Table 3-2 lists the power-on diagnostics that run from the BaseIO command prompt. For more information about each of the tests, refer to the page shown in the "Page" column.

Table 3-2 Power-On Diagnostics That Run From the BaseIO Command Prompt

Diagnostic Test	Command to Run the Test	Page
Comprehensive SCSI test	*diag_scsi*	n/a
SCSI SSRAM test	*diag_scsi* -t ram	5-2
SCSI SSRAM DMA test	*diag_scsi* -t dma	5-3
SCSI self-test	*diag_scsi* -t controller	5-4
Comprehensive Ethernet test	*diag_enet*	5-6
Ethernet SSRAM test	*diag_enet* -t ssram	5-9
PHY register test	*diag_enet* -t phyreg	5-10
TX clock test	*diag_enet* -t txclk	5-12
NIC test	*diag_enet* -t nic	5-13
IOC3 internal loopback test	*diag_enet* -t ioc3_loop	5-15
PHY chip internal loopback test	*diag_enet* -t phy_loop	5-18
Twister chip internal loopback test	*diag_enet* -t tw_loop	5-20
External loopback DMA test	*diag_enet* -t ext_loop	5-22
External loopback DMA test (10 Mb/s)	*diag_enet* -t ext_loop_10	5-24
External loopback DMA test (100 Mb/s)	*diag_enet* -t ext_loop_100	5-26
XTALK stress test	*diag_enet* -t xtalk	5-28

Chapter 4

IP27 PROM Tests

This Chapter describes the power-on diagnostic tests that are included in the IP27 PROM.

Each test description includes the following information:

- A short description of the test
- Any prerequisites that you should perform before you run the test (if you can manually run the test)
- Information about how to run the test (if it runs during the boot sequence and if you can run it manually)
- A description of the actions that the test performs
- Failure information for the test (including the failure indication and the failing hardware components)

All IP27 PROM tests print out pass and fail messages. The tests send the messages to the system controller. The system controller sends the messages to the console that is connected to the alternate console port (ACP). Failure messages use the following format:

`RSLT` *test_name*`FAIL diag_rc =` *xxx failure_description*

Example:

```
RSLT hub_intrpt_diag FAIL diag_rc = 157 hub interrupt test failed.
```

4.1 R10000 Processor Tests

4.1.1 Register Test

The register test checks the COP1 register portion of the R10000 processor.

4.1.1.1 How to Run This Test

This test runs automatically during the boot process. You cannot run this test from the POD prompt.

4.1.1.2 Test Description

This test checks for stuck bits in the processor registers. This test uses the following algorithm:

1. Write 0xa5a5 and 0x0000 patterns to the BSR registers.
2. Read the data back from the BSR registers.
3. Compare the data that was read back with the data that was written.

4.1.1.3 Failure Information

If this test fails, the value FLED_CP1 (0x81) flashes on the LEDs of the failing CPU.

If this test fails, the failing FRU is the Node board; the cause is most likely the R10000 chip.

4.1.2 Primary Cache Tests

The primary instruction cache test writes to and reads from the primary instruction cache. The primary data cache test writes to and reads from the primary data cache.

4.1.2.1 How to Run These Tests

These tests run automatically during the boot process. You cannot run these tests from the POD prompt.

4.1.2.2 Test Descriptions

The primary instruction cache test uses the following algorithm:

1. Write 0x5555 and 0x0000 data patterns to the primary instruction cache.
2. Read the data back from the primary instruction cache.
3. Compare the data that was read with the data that was written.

The primary data cache test uses the following algorithm:

1. Write 0x5555 and 0x0000 data patterns to the primary data cache.
2. Read the data back from the primary data cache.
3. Compare the data that was read with the data that was written.

4.1.2.3 Failure Information

If the primary instruction cache test fails, the value FLED_ICACHE (0x83) illuminates on the LEDs of the failing CPU. If primary data cache test fails, the value FLED_DCACHE (0x84) illuminates on the LEDs of the failing CPU.

If either of these tests fails, the failing FRU is the Node board; the cause of the failure is most likely the R10000 chip.

4.1.3 Interrupt Test

The interrupt test verifies that the processors can correctly receive interrupts.

4.1.3.1 How to Run This Test

This test runs automatically during the boot process. You cannot run this test from the POD prompt.

4.1.3.2 Test Description

This test checks the hardware in the R10000 processors that is dedicated to handling any interrupts or exceptions. It also tests the HUB chip interrupt registers.

4.1.3.3 Failure Information

If this test fails, the value FLED_HUBINTS (0x90) illuminates on the LEDs of the failing CPU, and the test prints a `RSLT hub_int_diag FAIL` message on the console connected to the ACP.

If this test fails for both CPUs on a Node board, the failing FRU is the Node board; the cause is most likely the HUB chip on the Node board. If this test fails for only one of the CPUs, the failing FRU is the Node board; the cause of the failure is most likely the CPU that failed; however, it is still possible that the Node board caused the failure.

4.2 Second-level Cache Tests

The second-level cache tests test the secondary data cache and the secondary tag cache.

4.2.1 Secondary Cache Tests

The secondary data and tag cache tests walk data patterns across the secondary data cache to detect stuck bits.

4.2.1.1 How to Run These Tests

These tests run automatically during the boot process. You cannot run these tests from the POD prompt.

4.2.1.2 Test Descriptions

These tests check for stuck bits in the secondary cache SRAMs. These tests use the following algorithm:

1. Write data patterns (0xaaaa and 0x5555 in normal mode; 0xaaaa, 0x5555, 0x0000, and 0xffff in heavy and manufacturing modes) to the secondary cache SRAMs that are located on the HIMM board.
2. Read the data back from the secondary cache SRAMs.
3. Compare the data that was read with the data that was written.

4.2.1.3 Failure Information

If these tests fail, the value FLED_SCACHE (0x85) illuminates on the LEDs of the failing CPU, and the tests print a `RSLT cache_test_s FAIL` message on the console connected to the ACP.

If these tests fail, the failing FRU is the Node board; the failure is caused by a soft or hard failure in an SRAM chip, a bad interconnect on the HIMM board between the R10000 processor and the SRAM, or a bad R10000 processor. If a failure occurs, these tests use the syndrome information to decode the bad bit; this process enables the test to identify which of the five secondary cache chips has the bad bit.

4.3 HUB Tests

4.3.1 GClk Logic Test

The GClk logic test checks the functionality of the HUB global clock logic.

4.3.1.1 How to Run This Test

This test runs automatically during the boot process. You cannot run this test from the POD prompt.

4.3.1.2 Test Description

This test checks the GCLK logic that is internal to the HUB chip.

4.3.1.3 Failure Information

If a failure occurs, this test prints out a `RSLT rt_clock_test FAIL` failure message on the console connected to the ACP.

If this test fails, the failing FRU is the Node board.

4.3.2 BTE Test

The BTE test checks the block transfer engine (BTE) logic in the HUB.

4.3.2.1 How to Run This Test

This test runs automatically during the boot process. You cannot run this test from the POD prompt.

4.3.2.2 Test Description

This test uses the following algorithm:

1. Program both BTEs on the HUB chip.
2. Have both BTEs on the HUB chip perform local transfer operations.
3. Compare the results.

In all modes, this test performs a simple block copy transfer and then checks all basic modes of the BTE logic, including: basic copy, termination of a copy in progress, zero fill mode, poison mode, interrupt mode, and notification mode.

4.3.2.3 Failure Information

If this test fails, the test prints a `RSLT hub_bte_diag FAIL` message on the console connected to the ACP. The boot process continues even if this test detects an error.

If this test fails due to memory problems, the failing FRU is the failing DIMM. If the failure is not a memory problem, the failure is caused by one of the following conditions: the HUB has an internal logic failure, there is a multibit error in memory, or some of the HUB-to-memory interconnects are broken. If you test memory before you run this test, then the most likely cause of failure of this test is a bad HUB chip.

4.4 Local IO (Junk IO) Bus Device Test

4.4.1 PROM Checksum Test

The PROM checksum test verifies the copy of PROM code in memory. If it fails when it is verifying the copy in memory, this test verifies the copy of the PROM code that is stored in the PROM.

4.4.1.1 How to Run This Test

This test runs automatically during the boot process. You cannot run this test from the POD prompt.

4.4.1.2 Test Description

This test uses the following algorithm:

1. Add up the number of words in the PROM code.
2. Compare the result with an expected value.

4.4.1.3 Failure Information

If this test fails, it flashes FLED_DOWNLOAD (0x9b) on the LEDs and prints a `RSLT prom_checksum FAIL` message on the console connected to the ACP.

If this test fails, the failing FRU is either the Node board or a memory DIMM. Possible causes are a bad PROM, a bad copy of the software image in the PROM, bad bit(s) in memory, a transient error that occurred when the PROM image was copied from the PROM into memory, or a transient error in the R10000 processor that occurred when the processor computed the checksum. The most likely cause is one or more bad bits in memory.

4.5 Memory/Directory Tests

4.5.1 Memory Configuration Test

The memory configuration test verifies that the system's memory configuration is valid.

4.5.1.1 How to Run This Test

This test runs automatically during the boot process. You cannot run this test from the POD prompt.

4.5.1.2 Test Description

This test checks the memory configuration for valid memory combinations, valid memory sizes, etc.

4.5.1.3 Failure Information

If this test fails, it illuminates FLED_NOMEM (0x98) or FLED_BADMEM (0xa9) on the LEDs. The test also prints one of the following messages on the console connected to the ACP.

```
No Memory Found

*** Mixed standard and premium memory
*** Treating all as standard.

nasid node_number: disabling bank bank_number
```

If this test fails, the failing FRU is the DIMM that the test indicates. A failure of this test most likely indicates a bad memory configuration. It could also indicate a hardware failure in a HUB-to-memory interconnect. The system will not boot if this test fails.

Note: This test always prints out the following message for each Node in the system: `Total memory configured:` *number* `MB.`

4.5.2 Backdoor Directory Space Test

The backdoor directory space test checks the memory that contains the backdoor directory space.

4.5.2.1 Prerequisites to Running This Test

You should verify that the R10000 processor, the system interface bus, and the HUB registers are functional before you run this test.

4.5.2.2 How to Run This Test

This test runs automatically during the boot process. You can also use the *dirtest* command at the POD prompt to run this test. For more information about the *dirtest* command, refer to Section 6.10.17.3, "dirtest Command,".

4.5.2.3 Test Description

This test uses the following algorithm:

1. Write data patterns to the backdoor directory space. (In normal mode, this test uses 0x55 and 0xaa data patterns. In heavy and manufacturing modes, this test uses 0x55, 0xaa, and additional patterns.)
2. Read the data back.
3. Compare the data that was read back to the data that was written.
4. Test the address path by writing an address-dependent pattern to the backdoor directory space.
5. Read the data back.
6. Compare the data that was read back with the data that was written.

4.5.2.4 Failure Information

If this fails, it prints one of the following messages on the console connected to the ACP.

```
-----MEMORY FAILURE (caught exception node slot slot_number) -----
Address address
CAUSE cause_register_value
```

```
-----MEMORY FAILURE (miscompare node slot slot_number) -----
Address address
CAUSE cause_register_value
```

If this test fails, the failing FRU is the DIMM that the test specifies. Failure of this test indicates one of the following hardware failures (listed most likely to least likely): a directory DIMM had a soft or hard failure, a HUB-to-directory memory interconnect failed, or the HUB failed.

4.5.3 Memory Test

The memory test checks the memory address and data lines.

4.5.3.1 Prerequisites to Running This Test

You should verify that the R10000 processors, the system interface bus and the HUB registers are functional before you run this test.

4.5.3.2 How to Run This Test

This test runs automatically during the boot process. You can also use the *memtest* command from the POD prompt to run this test. For more information about the *memtest* command, refer to Section 6.10.17.4, "memtest Command,".

4.5.3.3 Test Description

This test uses the following algorithm:

1. Write data patterns to memory.
2. Read the data back from memory.
3. Compare the data that was read with the data that was written.

4.5.3.4 Failure Information

If this test fails, it prints one of the following messages on the console connected to the ACP.

```
-----MEMORY FAILURE (caught exception node slot slot_number) -----
Address address
CAUSE cause_register_value

-----MEMORY FAILURE (miscompare node slot slot_number) -----
Address address
Actual actual_value
Expected expected_value
Difference difference_value
```

If this test fails, the failing FRU is the DIMM that the test specifies. A failure in this test indicates a soft or hard failure in a memory chip (most likely), a failing HUB chip-to-memory interconnect, or a bad HUB chips (least likely).

4.6 CrayLink Tests

4.6.1 Local Link Connection Test

The local link connection test checks the connection between the local HUB and an attached HUB or Router.

4.6.1.1 Prerequisites to Running This Test

You should verify that the R10000 processor and the HUB are functional before you run this test.

4.6.1.2 How to Run This Test

This test runs automatically during the boot process, and you can run it from the POD prompt. The following example shows how you can run this test from the POD prompt:

```
POD Dex 000 0A> chklink
```

4.6.1.3 Test Description

This test uses the following algorithm:

1. Check the ni_port error register
2. Perform a series of vector write and read operations; verify the results.

Note: This test runs before the boot process performs network discovery operations; this process isolates any errors to the local Node.

4.6.1.4 Failure Information

If this test fails, the test displays a `RSLT chk_local_link FAIL` message on the console that is connected to the ACP; the message includes complete link isolation information.

If this test fails, the failing FRU is the midplane (most likely), Node board, or Router (least likely). The failure is caused by a failing local HUB, a failing attached HUB or Router, a failing local HUB-to-attached HUB interconnect, or a failing local HUB-to-attached Router interconnect.

4.6.2 HUB Send Data Error Test

The HUB send data error test verifies that the HUB or Router attached to the local HUB can detect link errors.

4.6.2.1 Prerequisites to Running This Test

You should ensure that the R10000 processors and the HUB are functional before you run this test.

4.6.2.2 How to Run This Test

This test runs automatically during the boot process, and you can run it from the POD prompt. The following example shows how you can run this test from the POD prompt:

```
POD Dex 000 0A> hubsde
```

4.6.2.3 Test Description

This test uses the following algorithm:

1. Check the remote port error register.
2. Send an error in a packet across the link.
3. Verify that the hardware detected and logged the error.

This test runs the same in normal, heavy, and manufacturing modes.

Note: This test runs before the boot process performs network discovery operations; this process isolates any errors to the local Node.

4.6.2.4 Failure Information

If this test fails, the test displays a `RSLT hub_send_data_error FAIL` message on the console connected to the ACP; the message includes complete link isolation information.

If this test fails, the failing FRU is the Router, Node board, or midplane. A failure in this test indicates a bad local HUB, a bad attached HUB or Router, a bad local HUB-to-attached HUB interconnect, or a bad HUB-to-attached Router interconnect.

4.6.3 HUB Error Detection Test

The HUB error detection test is similar to the HUB send data error test, but the HUB error detection test is called recursively during the discovery process.

4.6.3.1 How to Run This Test

This test runs automatically during the boot process. You cannot run this test from the POD prompt.

4.6.3.2 Test Description

This test uses the following algorithm:

1. Check the remote port error register.
2. Send an error in a packet across the link.
3. Verify that the hardware detected and logged the error.

This test runs only in heavy and manufacturing modes. The algorithm is performed recursively during the boot process.

4.6.3.3 Failure Information

If this test fails, the test displays a `RSLT hub_error_detection_error FAIL` message on the console that is connected to the ACP; the message includes complete link isolation information.

The failing FRU is the Node board (most likely), Router, or midplane (least likely). A failure in this test indicates a bad Local HUB, a bad attached HUB or Router, or a bad local HUB-to-attached HUB or Router interconnect. Given the testing that has been performed by the time this test runs in the boot process, the most likely cause of failure is a failing attached link or a failing attached board or chip.

4.6.4 Router Send Data Error Test

The Router send data error test verifies that the local HUB can detect link errors.

4.6.4.1 Prerequisites to Running This Test

You should ensure that the R10000 processors and the HUB are functional before you run this test.

4.6.4.2 How to Run This Test

This test runs automatically during the boot process, and you can run it from the POD prompt. The following example shows how you can run this test from the POD prompt:

```
POD Dex 000 0A> rtrsde
```

4.6.4.3 Test Description

This test uses the following algorithm:

1. Check the local port error register.
2. Send an error in a packet across the link.
3. Verify that the hardware detected and logged the error.

This test runs identically in normal, heavy, and manufacturing modes.

Note: This test runs before the boot process performs network discovery operations; this process isolates any errors to the local Node.

4.6.4.4 Failure Information

If this test detects an error, the test displays a `RSLT rtr_send_data_error FAIL` message on the console connected to the ACP; the message includes complete link isolation information.

The failing FRU is the Node board (most likely), Router, or midplane/cable (least likely). A failure in this test indicates a bad local HUB, a bad attached HUB or Router, or a bad local HUB-to-attached HUB or Router interconnect. Given the testing that has been performed by the time this test runs in the boot process, the most likely cause of failure is a failing attached link or a failing attached board or chip.

4.6.5 Router Error Detection Test

The Router error detection test is similar to the Router send data error test except that you cannot run it from the POD prompt and that it runs recursively during the discovery process.

4.6.5.1 How to Run This Test

This test runs automatically during the boot process. You cannot manually run this test from the POD prompt.

4.6.5.2 Test Description

This test uses the following algorithm:

1. Check the local port error register.
2. Send an error in a packet across the link.
3. Verify that hardware detects and logs the error.

This test runs only in heavy and manufacturing modes.

4.6.5.3 Failure Information

If this test fails, the test displays a `RSLT test_rtr_error_detection FAIL` message on the console connected to the ACP; the message includes complete link isolation information.

The failing FRU is the Router (most likely), Node board, or midplane/cable (least likely). A failure in this test indicates a bad local HUB, a bad attached HUB or Router, or a bad local HUB-to-attached HUB or Router interconnect. Given the testing that has been performed by the time this test runs in the boot process, the most likely cause of failure is a failing attached link or a failing attached board or chip.

4.6.6 Get Chipid Test

The get chipid test runs during the discovery process to obtain the ID of a HUB or Router chip that is attached across an LLP link.

4.6.6.1 How to Run This Test

This test runs automatically during the boot process. You cannot manually run this test from the POD prompt.

4.6.6.2 Test Description

The get chipid test runs during the discovery process to obtain the ID of a HUB or Router chip that is attached across an LLP link.

4.6.6.3 Failure Information

If this test fails, it displays a `RSLT get_chipid FAIL` failure message on the console connected to the ACP; the message includes complete isolation information for the failing links.

The failing FRU is the Node board or Router. A failure in this test indicates a bad source chip, a bad destination chip, or a bad link between these chips. The most likely cause of a failure is a bad or misconnected link.

4.7 XBOW Test

The XBOW test checks for valid reset values in the XBOW registers and tests the XBOW interrupt hardware.

Note: If a HUB-link error occurs during the boot sequence, this test may not run because the link was not detected.

4.7.1 Prerequisites to Running This Test

There are none.

4.7.2 How to Run This Test

This test runs automatically during the boot process, and you can run it from the POD prompt. To run this test from the POD prompt, enter the following command:

dgxbow [mn | mh | mm] [n*NODE*]

The **m** option specifies the mode in which the test should run. Enter **mn** to run the test in normal mode, enter **mh** to run the test in heavy mode, and enter **mm** to run the test in manufacturing mode. (The default mode is normal mode.)

The **n** option specifies the Node ID (NASID value) of the Node on which the test should run. (The default Node is the Node from which you entered the command.)

For example, enter **dgxbow mh n1** to run the XBOW test in heavy mode on Node 1.

4.7.3 Test Description

This test uses the following algorithm:

1. Read the XBOW registers and verify that they contain valid reset values.
2. Create an illegal access by writing to a read-only register, which causes an XBOW interrupt.
3. Verify that the interrupt occurred on the XBOW and the HUB.

4.7.4 Failure Information

The XBOW test returns the following failure messages:

- `xbow_sanity: Bad xbow id reg value` *value*

  The XBOW test failed.

- `xbow_sanity: Bad Xbow id Exp:` *expected* `Recv:` *received*

  The XBOW test failed.

- `xbow_sanity: Reg write Read miscompare. Exp:` *expected*`:: Recv:` *received*

  This message indicates a failing XBOW or XBOW-to-HUB connection.

- `xbow_sanity: Register access violation not detected.`

  This message indicates a failing XBOW chip.

- `xbow_sanity: Status Register Clear on Read not working!!`

  The XBOW test failed.

- `xbow_sanity: Interrupt did not reach PI_INT_PEND0. exp:` *expected* `recv:` *received*

  This message indicates a failing XBOW, a failing XBOW-to-HUB connection, or a failing HUB.

- `RSLT xbow_sanity FAIL`

  This message follows any of the previous XBOW failure messages.

When the XBOW test fails, the failing FRU is either the midplane or a Node board.

4.8 Bridge Test

The Bridge test checks for valid reset values in the Bridge registers and tests the Bridge interrupt hardware.

4.8.1 Prerequisites to Running This Test

There are none.

4.8.2 How to Run This Test

This test runs automatically during the boot process, and you can run it from the POD prompt. To run this test from the POD prompt, enter the following command:

dgbrdg [mn | mh | mm] [n*NODE*] [s*SLOT*]

The **m** option specifies the mode in which the test should run. Enter **mn** to run the test in normal mode, enter **mh** to run the test in heavy mode, and enter **mm** to run the test in manufacturing mode. (The default mode is normal mode.)

The **n** option specifies the Node ID (NASID value) of the Node on which the test should run. (The default Node is the Node from which you entered the command.)

The **s** option specifies the IO slot number of the board with the Bridge that you want to test. (The default IO slot number is 1, which selects slot IO1.)

For example, enter **dgbrdg mh s4** to run this test in heavy mode on the Bridge ASIC on the I/O board in slot IO4 of the module that contains the Node from which you entered the command.

4.8.3 Test Description

This test uses the following algorithm:

1. Read the Bridge registers and verify that they contain valid reset values.
2. Force illegal accesses to the SSRAM, which causes Bridge interrupts because the SSRAM is not located on the BaseIO board.
3. Verify that the interrupts occurred on both the Bridge and the HUB.

4.8.4 Failure Information

The Bridge test returns the following failure messages:

- `bridge_sanity: Bad bridge id register value (LSB != 1)`

  This message indicates a failing Bridge ASIC. The failing FRU is the BaseIO board.

- `bridge_sanity: Bad bridge id Exp:` *expected* `Recv:` *received*

  This message indicates a failing Bridge ASIC. The failing FRU is the BaseIO board.

- `bridge_sanity: PCI bit set to 0 sn status reg.`

  The Bridge test failed. The failing FRU is the BaseIO board.

- `bridge_sanity: Timeout val reg exp:` *expected* `recv:` *received*.

  The Bridge test failed. The failing FRU is the BaseIO board.

- `bridge_sanity: Bridge control reg exp:` *expected* `recv:` *received*.

  The Bridge test failed. The failing FRU is the BaseIO board.

- `bridge_sanity: Bad reset val in dev reg exp:` *expected* `recv:` *received*.

  The Bridge test failed. The failing FRU is the BaseIO board.

- `bridge_sanity: Interrupt status register value non-zero:` *value*.

  The Bridge test failed. The failing FRU is the BaseIO board.

- `bridge_sanity: Bridge(wr): Exp:` *expected* `Recv:` *received*.

  The Bridge test failed. The failing FRU is the BaseIO board.

- `bridge_sanity: Interrupt status reg bit not set: exp:` *expected* `recv:` *received*.

  The Bridge test failed. The failing FRU is the BaseIO board.

- `bridge_sanity: Interrupt` *interrupt* `did not reach PI_INT_PEND0: exp:` *expected* `recv:` *received*.

  The Bridge test failed. The failing FRU is the BaseIO board.

- `bridge_sanity: Interrupt` *interrupt* `did not occur.`

  This message indicates a failing Bridge, a failing XBOW chip, or a failing HUB.

- `RSLT bridge_sanity FAIL`

  This message follows any of the previous bridge test failure messages.

4.9 PCI Tests

4.9.1 PCICONFIG Test

The PCICONFIG test verifies the PCI configuration space on the BaseIO board.

4.9.1.1 Prerequisites to Running This Test

There are none.

4.9.1.2 How to Run This Test

This test runs automatically during the boot process, and you can run it from the POD prompt. To run this test from the POD prompt, enter the following command:

dgconf [mn | mh | mm] [n*NODE*] [s*SLOT*]

The **m** option specifies the mode in which the test should run. Enter **mn** to run the test in normal mode, enter **mh** to run the test in heavy mode, and enter **mm** to run the test in manufacturing mode. (The default mode is normal mode.)

The **n** option specifies the Node ID (NASID value) of the Node on which the test should run. (The default Node is the Node from which you entered the command.)

The **s** option specifies the IO slot number of the board that you want to test. (The default IO slot number is 1, which selects slot IO1.)

For example, enter **dgconf mh s7** to run the PCICONFIG test in heavy mode on the BaseIO board in slot IO7 of the module that contains the Node from which you entered the command.

4.9.1.3 Test Description

The PCICONFIG test verifies the PCI configuration space on the BaseIO board.

4.9.1.4 Failure Information

The PCICONFIG test returns the following failure messages:

- `io6config_space: IOC3 PCI_SCR reg reset val exp:` *expected* `recv:` *received*

  The PCICONFIG test failed; the failing FRU is the BaseIO board.

- `io6config_space: IOC3 PCI_ADDR reg reset val exp:` *expected* `recv:` *received*

  The PCICONFIG test failed; the failing FRU is the BaseIO board.

- `io6config_space: IOC3 PCI_ADDR reg write read error. wrote:` *value* `read:` *value*

  The PCICONFIG test failed; the failing FRU is the BaseIO board.

- `io6config_space: PCI device` *device* `QL command status reg reset val exp:` *expected* `recv:` *received*

  The PCICONFIG test failed; the failing FRU is the BaseIO board.

- `io6config_space: PCI device` *device* `QL class code and revision read as` *value*

  The PCICONFIG test failed; the failing FRU is the BaseIO board.

- `io6config_space: PCI device` *device* `QL base address reg reset val exp:` *expected* `recv:` *received*

  The PCICONFIG test failed; the failing FRU is the BaseIO board.

- `io6config_space: PCI device` *device* `QL base addr write read error. wrote:` *value* `read:` *value*

  The PCICONFIG test failed; the failing FRU is the BaseIO board.

- `RSLT io6config_space FAIL`

  This message follows any of the previous failure messages.

4.9.2 PCI Bus Test

The PCI bus test checks the PCI bus with walking data tests.

4.9.2.1 Prerequisites to Running This Test

There are none.

4.9.2.2 How to Run This Test

This test runs automatically during the boot process, and you can run it from the POD prompt. To run this test from the POD prompt, enter the following command:

dgpci [mn | mh | mm] [n*NODE*] [s*SLOT*] [p*PCINUM*]

The **m** option specifies the mode in which the test should run. Enter **mn** to run the test in normal mode, enter **mh** to run the test in heavy mode, or enter **mm** to run the test in manufacturing mode. (The default mode is normal mode.)

The **n** option specifies the Node ID (NASID value) of the Node on which the test should run. (The default Node is the Node from which you entered the command.)

The **s** option specifies the IO slot number of the BaseIO board that you want to test. (The default IO slot number is 1, which selects slot IO1.)

The **p** option specifies the PCI device number of the IOC3 ASIC that you want to test. (The default PCI device number is 2, which selects the IOC3 chip on the BaseIO board.)

For example, enter `dgpci mn s3` to run the PCI bus test in normal mode on the Node from which you entered the command. This example uses the IOC3 ASIC on the BaseIO board in slot number IO3 of the module that contains the Node from which you started the test.

4.9.2.3 Test Description

This test uses the following algorithm:

1. Write a walking one pattern in the 32-bit Ethernet low address register.
2. Read the data back.
3. Compare the data that was read back with the data that was written.
4. Write a walking zero pattern in the 32-bit Ethernet low address register.
5. Read the data back.
6. Compare the data that was read back with the data that was written.

4.9.2.4 Failure Information

The PCI bus test returns the following failure messages:

- `pcibus_sanity: IOC3 enet hashed low addr reg reset val exp:` *expected* `recv:` *received*

  This message indicates a failing PCI or failing IOC3 chip. The failing FRU is the BaseIO board.

- `pcibus_sanity: PCI walk one test. exp:` *expected* `recv:` *received*

  This message indicates a failing PCI bus. The failing FRU is the BaseIO board.

- `pcibus_sanity: PCI walk zero test. exp:` *expected* `recv:` *received*

  This message indicates a failing PCI bus. The failing FRU is the BaseIO board.

- `RSLT pcibus_sanity FAIL`

  This message follows any of the previous PCI bus failure messages. The failing FRU is the BaseIO board.

4.10 Serial Port Tests

4.10.1 Serial Programmed I/O Test

The serial programmed I/O (PIO) test uses the internal loopback features of the SuperIO chip to test the serial PIO functionality of the BaseIO board.

4.10.1.1 Prerequisites to Running This Test

Before you can run this test in external loopback mode (by using the **mx** command line option), you must attach loopback connectors to the serial ports that you will test.

4.10.1.2 How to Run This Test

This test runs automatically during the boot process, and you can run it from the POD prompt. To run this test from the POD prompt, enter the following command:

dgspio [mn | mh | mm | mx] [n*NODE*] [s*SLOT*] [p*PCINUM*]

The **m** option specifies the mode in which the test should run. Enter `mn` to run the test in normal mode, enter `mh` to run the test in heavy mode, enter `mm` to run the test in manufacturing mode, or enter `mx` to run the test in external loopback mode. (The default mode is normal mode.)

The **n** option specifies the Node ID (NASID value) of the Node on which the test should run. (The default Node is the Node from which you entered the command.)

The **s** option specifies the IO slot number of the BaseIO board that you want to test. (The default IO slot number is 1, which selects slot IO1.)

The **p** option specifies the PCI device number of the IOC3 ASIC that you want to test. (The default PCI device number is 2, which selects the IOC3 ASIC on the BaseIO board.)

For example, enter `dgspio mh` to run the serial PIO test in heavy mode on the Node from which you entered the command. The test uses the IOC3 chip on the BaseIO board in slot IO1 of the module that contains the Node from which you started the test.

4.10.1.3 Test Description

This test uses the following algorithm:

1. Reset the SuperIO chip.
2. Fill the transmit FIFO with data.
3. Read the receive FIFO in loopback mode.
4. Compare the data in the receive FIFO with the data in the transmit FIFO.

4.10.1.4 Failure Information

The serial PIO test returns the following failure messages:

- `serial_pio: UARTA read timed out on LSR<Data Ready>`

  The serial PIO test failed; the failing FRU is the BaseIO board.

- `serial_pio: UARTB read timed out on LSR<Data Ready>`

  The serial PIO test failed; the failing FRU is the BaseIO board.

- `serial_pio: UARTA error. Chars in FIFO after reset .....`

  This message indicates a failing SuperIO chip or PCI.

- `serial_pio: UARTB error. Chars in FIFO after reset .....`

  This message indicates a failing SuperIO chip. The failing FRU is the BaseIO board.

- `serial_pio: UARTA sent:` *value* `recv:` *value*

  This message indicates that the data that was received is different from the data that was sent. This message indicates a failing IOC3-to-SuperIO interface, a failing SuperIO chip, or a failing IOC3 chip.

  This message also appears if you do not install external loopback plugs on the ports before you run the test in external loopback mode.

- `serial_pio: UARTB sent:` *value* `recv:` *value*

  This message indicates that the data that was received is different from the data that was sent. This message indicates a failing IOC3-to-SuperIO interface, a failing SuperIO chip, or a failing IOC3 chip.

  This message also appears if you do not install external loopback plugs on the ports before you run the test in external loopback mode.

- `serial_pio failed with` *number* `errors`

  The serial PIO test failed; the failing FRU is the BaseIO board.

- `RSLT serial_pio FAIL`

  This message follows any of the previous serial PIO failure messages.

4.10.2 Serial DMA Test

The serial DMA test verifies that serial DMA can be performed.

4.10.2.1 Prerequisites to Running This Test

The CPU from which you run this test must be in Cac mode. If you attempt to run this test from a CPU that is in Dex mode or Unc mode, the test displays a message that indicates you should enter Cac mode (type `go cac`) before you run the test.

Before you can run this test in external loopback mode (by using the **mx** command line option), you must attach loopback connectors to the serial ports that you will test.

4.10.2.2 How to Run This Test

This test runs automatically during the boot process, and you can run it from the POD prompt. To run this test from the POD prompt, enter the following command:

dgsdma [mn | mh | mm | mx] [n*NODE*] [s*SLOT*] [p*PCINUM*]

The **m** option specifies the mode in which the test should run. Enter **mn** to run the test in normal mode, enter **mh** to run the test in heavy mode, enter **mm** to run the test in manufacturing mode, or enter **mx** to run the test in external loopback mode. (The default mode is normal mode.)

The **n** option specifies the Node ID (NASID value) of the Node on which the test should run. (The default Node is the Node from which you entered the command.)

The **s** option specifies the IO slot number of the BaseIO board that you want to test. (The default IO slot number is 1, which selects slot IO1.)

The **p** option specifies the PCI device number of the IOC3 ASIC that you want to test. (The default PCI device number is 2, which selects the IOC3 ASIC on the BaseIO board.)

For example, enter `dgsdma mh` to run the serial DMA test in heavy mode on the Node from which you entered the command. The test uses the IOC3 chip on the BaseIO board in slot IO1 of the module that contains the Node from which you started the test.

4.10.2.3 Test Description

This test uses the following algorithm:

1. Set up the ring buffers.
2. Set up DMA in loopback mode.
3. Perform DMA operations simultaneously through UART channels A and B.
4. Verify the results.

4.10.2.4 Failure Information

The serial DMA test returns the following failure messages:

- `serial_dma: PCI: could not get the ioc3 base addr`

  This message indicates a failing PCI or failing IOC3 chip. The failing FRU is the BaseIO board.

- `serial_dma: Serial DMA exp:` *expected* `recv:` *received*

  This message indicates that the data that was received is different from the data that was sent. This message indicates a failing SuperIO chip, a failing IOC3 chip, a failing Bridge, or a failing HUB.

  This message also appears if you do not install external loopback plugs on the ports before you run the test in external loopback mode.

- `serial_dma: Timeout on uartA dma ....`

  The serial DMA test failed; the failing FRU is the BaseIO board.

- `serial_dma: Timeout on uartB dma ....`

  The serial DMA test failed; the failing FRU is the BaseIO board.

- `RSLT serial_dma FAIL`

  This message follows any of the previous serial DMA failure messages.

Chapter 5

BaseIO PROM Tests

This chapter describes the power-on diagnostic tests that are included in the BaseIO PROM. These diagnostics test the SCSI controller and the Ethernet port and its related hardware.

Each test description includes the following information:

- Any prerequisites that you should perform before you run the test
- Information about how to run the test
- A description of actions that the test performs
- Failure information for the test (including the failure messages and the failing hardware components).

5.1 SCSI Tests

The BaseIO PROM's SCSI driver automatically calls the SCSI tests when the boot sequence installs the BaseIO SCSI devices. You can also manually run them from the >> prompt, which you can access by selecting option `5. Enter Command Monitor` of the System Maintenance Menu. You cannot run these tests from the `POD>` prompt.

When the tests run automatically, they use the mode set by the debug DIP switches. When you run the tests manually, the command line options that you specify override the mode specified by the debug DIP switch settings.

The *diag_scsi* command runs the SCSI tests from the >> prompt and uses the following syntax:

diag_scsi [-N *nasid*] [-m *moduleid* -s *slotid*] [-p *pcidev*] [-h] [-v] [-t *testname*] [-r *count* | -R *count*]

where:

-N *nasid* — Specifies the NASID number of the module to test (defaults to the console BaseIO module). This option supersedes the NASID implied by the module and slot values.

-m *moduleid* — Specifies the module number of the module to test (defaults to the console BaseIO module).

-s *slotid*	Specifies the name of the slot location that contains the module to test. (The default is the console BaseIO slot).
-p *pcidev*	Specifies the PCI device number of the target IOC3. (The default is 0, which is the device number of the first SCSI controller on an BaseIO board.)
-h	Runs the diagnostic in heavy mode.
-v	Runs the diagnostic in verbose mode, which returns detailed status and error messages.
-t *testname*	Runs only the specified test. Valid values for *testname* are ram, dma, and controller. (The default is to run all tests.)
-r *count*	Repeats the test *count* times. The test stops when it detects a failure.
-R *count*	Repeats the test *count* times. The test continues to execute when it detects a failure.

5.1.1 SCSI SSRAM Test

The SCSI SSRAM test verifies that data can be loaded into SCSI SSRAM memory and read back successfully. It also verifies the "mailbox" mechanism before it performs this test.

5.1.1.1 Prerequisites to Running This Test

There are none.

5.1.1.2 How to Run This Test

To run the SCSI SSRAM test from the >> prompt, enter the `diag_scsi -t ram` command and any other command line options that you want to use. For example:

```
>> diag_scsi -t ram [options]
```

Refer to Section 5.1 for descriptions of all available *options*.

5.1.1.3 Test Description

This test uses the following algorithm:

1. Verify the SCSI "mailbox" mechanism.
2. Write data into the SCSI SSRAM (one word at a time).
3. Read data back from the SCSI SSRAM (one word at a time).
4. Compare the data that was read back with the data that was written.

5.1.1.4 Failure Information

The SCSI SSRAM test returns the following failure messages:

- `scsi_ram: SCSI mailbox error. erp:` *expected* `recv:` *received*

  This message indicates a failing SCSI controller. The failing FRU is the BaseIO board.

- `scsi_ram: QL: wrote:` *write_value* `read` *read_value* `at addr:` *address*

  This message indicates failing SCSI SSRAM. The failing FRU is the BaseIO board.

- `RSLT scsi_dma FAIL`

  This message follows any of the previous SCSI SSRAM failure messages.

5.1.2 SCSI DMA Test

The SCSI DMA test verifies that the SCSI DMA mechanism is functional.

5.1.2.1 Prerequisites to Running This Test

The CPU from which you run this test must be in Cac mode. If you attempt to run this test from a CPU that is in Dex mode or Unc mode, the test displays a message that indicates you should enter Cac mode (type `go cac`) before you run the test.

5.1.2.2 How to Run This Test

To run the SCSI DMA test from the >> prompt, enter the `diag_scsi -t dma` command and any other command line options that you want to use. For example:

```
>> diag_scsi -t dma [options]
```

Refer to Section 5.1 for descriptions of all available *options*.

5.1.2.3 Test Description

The SCSI DMA test uses the following algorithm:

1. Use a SCSI DMA operation to load the SCSI SSRAM with data.
2. Use a SCSI DMA operation to dump the data in the SCSI SSRAM.
3. Compare the data that was dumped with the data that was loaded.

5.1.2.4 Failure Information

The SCSI DMA test returns the following failure messages:

- `scsi_dma: LOAD SSRAM command failed`

  This message indicates a failing SCSI controller. The failing FRU is the BaseIO board.

- `scsi_dma: DUMP SSRAM command failed`

  This message indicates a failing SCSI controller. The failing FRU is the BaseIO board.

- `scsi_dma: DMA failure exp` *expected* `recv` *received* `at` *location*

  This message indicates a failing SCSI controller, Bridge chip, or HUB chip.

- `RSLT scsi_dma FAIL`

  This message follows any of the previous SCSI DMA failure messages.

5.1.3 SCSI Self-Test

The SCSI self-test uses the SCSI self-test firmware to test the SCSI controller.

5.1.3.1 Prerequisites to Running This Test

There are none.

5.1.3.2 How to Run This Test

To run the SCSI self-test from the >> prompt, enter the `diag_scsi -t controller` command and any other command line options that you want to use. For example:

```
>> diag_scsi -t controller [options]
```

Refer to Section 5.1 for descriptions of all available *options*.

5.1.3.3 Test Description

This test uses the following algorithm:

1. Load the SCSI self-test firmware.
2. Perform the self-test on the SCSI controller. The self-test tests the RISC unit, the SXP FIFOs, and the SCSI bus.

5.1.3.4 Failure Information

The SCSI self-test returns the following failure messages:

- `scsi_controller: EXECUTE FIRMWARE Command Failed.`

  This message indicates a failing SCSI controller. The failing FRU is the BaseIO board.

- `scsi_controller: write data failure at` *count* `word`

  This message indicates a failing SCSI controller or SCSI bus. The failing FRU is the BaseIO board.

- `scsi_controller: Bad status val ==>` *value* `....`

  This message indicates a failing SCSI controller. The failing FRU is the BaseIO board.

- `RSLT scsi_controller FAIL`

  This message follows any of the previous SCSI self-test failure messages.

5.2 Ethernet Tests

The BaseIO PROM's Ethernet driver automatically calls the Ethernet tests when the boot sequence installs the BaseIO Ethernet device. You can also manually run them from the >> prompt, which you can access by selecting option `5. Enter Command Monitor` of the System Maintenance Menu. You cannot run these tests from the `POD>` prompt.

When the tests run automatically, they use the mode set by the debug DIP switches. When you run the tests manually, the command line options that you specify override the mode specified by the debug DIP switch settings.

The *diag_enet* command runs the Ethernet tests from the >> prompt and uses the following syntax:

diag_enet [-N *nasid*] [-m *moduleid* -s *slotid*] [-p *pcidev*] [-h] [-e | -x] [-v] [-t *testname*] [-r *count* | -R *count*]

where:

-N *nasid*	Specifies the NASID number of the module to test (defaults to the console BaseIO module). This option supersedes the NASID implied by the module and slot values.
-m *moduleid*	Specifies the module number of the module to test (defaults to the console BaseIO module).
-s *slotid*	Specifies the name of the slot location of the module to test. (The default is the console BaseIO slot).
-p *pcidev*	Specifies the PCI device number of the target IOC3. (The default is 2, which is the device number of the IOC3 on a BaseIO board.)
-h	Runs the diagnostic in heavy mode.
-e or -x	Runs the diagnostic in external loopback mode.
-v	Runs the diagnostic in verbose mode, which returns detailed status and error messages.
-t *testname*	Runs only the specified test. Valid values for testname are ssram, txclk, phyreg, nic, ioc3_loop, phy_loop, tw_loop, ext_loop, ext_loop_10, ext_loop_100, and xtalk. (The default is to run all tests.)
-r *count*	Repeats the test *count* times. The test stops when it detects a failure.
-R *count*	Repeats the test *count* times. The test continues executing when it detects a failure.

The Ethernet diagnostics are written with the assumption that the following hardware has been tested and is functional:

- Non-Ethernet-specific logic on the IOC3 ASIC
- PCI bus address and data
- Bridge ASIC
- XTALK interface between the Bridge and XBOW chips
- XBOW, midplane, and all IP27 board components

Note: When the boot sequence runs the Ethernet diagnostics, all of these components have already been tested by other POD diagnostics. However, the actions that the Ethernet diagnostics perform stress these components harder than the other diagnostics, so it is possible that a previously undetected failure in one of these components could cause the Ethernet diagnostics to fail.

The Ethernet diagnostic tests are:

- Comprehensive Ethernet test
- Ethernet SSRAM test
- PHY register test
- TX clock test
- NIC test
- IOC3 internal loopback test
- PHY chip internal loopback test
- Twister chip internal loopback test
- External loopback DMA test
- External loopback DMA (10 Mb/s) test
- External loopback DMA (100 Mb/s) test
- XTALK stress test

5.2.1 Comprehensive Ethernet Test

The comprehensive Ethernet test runs all of the Ethernet tests in an order that optimizes fault isolation. The ef_install routine in the BaseIO PROM Ethernet driver calls this test during the boot sequence.

Note: If a failure of one of the diagnostics implies failures in subsequent diagnostics, this test does not run the diagnostics that will fail.

5.2.1.1 Prerequisites to Running This Test

Ensure that the following hardware is functional before you run this test:

- Non-Ethernet-specific logic on the IOC3 ASIC
- PCI bus address/data
- Bridge ASIC
- XTALK interface between the Bridge ASIC and XBOW chip
- XBOW chip, midplane, and all IP27 board components

5.2.1.2 How to Run This Test

This test runs automatically as part of the boot sequence. To manually run this test from the >> prompt, enter `diag_enet [`*options*`]`.

Refer to Section 5.2 for descriptions of all available *options*.

5.2.1.3 Test Description

This test uses the following algorithm to control the execution of the other Ethernet tests:

```
Run the SSRAM test
Run the TX clock test
Run the PHY register test
Run the NIC test
If the SSRAM test passed
  {
      Run the IOC3 internal loopback DMA test
      If the IOC3 internal loopback DMA test passed AND the TX clock test passed AND
      the PHY register test passed
      {
          Run the PHY internal loopback DMA test
          If the PHY internal loopback DMA test passed
          {
              Run the Twister chip internal loopback DMA test
              If the PHY internal loopback DMA test passed AND external loopback
              mode is selected
              {
                  Run the external loopback DMA test
              }
              If the PHY internal loopback DMA test passed AND external loopback DMA
              test passed (or was not run) AND heavy mode or external loopback mode
              was selected
              {
                  Run the XTALK stress test
              }
          }
      }
  }
```

For detailed information about each of the tests, refer to the corresponding description later in this document.

5.2.1.4 Failure Information

The comprehensive Ethernet diagnostic generates the following failure messages:

- `enet_all: SSRAM test failed.`

  The SSRAM test failed. Refer to Section 5.2.2, "Ethernet SSRAM Test," for more information about the failure messages that the SSRAM test returns.

- `enet_all: Ethernet PHY chip register test failed.`

  The PHY register test failed. Refer to Section 5.2.3, "PHY Register Test," for more information about the failure messages that the PHY register test returns.

- `enet_all: Ethernet TX clock test failed.`

  The TX clock test failed. Refer to Section 5.2.4, "TX Clock Test," for more information about the failure messages that the TX clock test returns.

- `enet_all: Ethernet NIC test failed.`

  The NIC test failed. Refer to Section 5.2.5, "NIC Test," for more information about the failure messages that the NIC test returns.

- `enet_all: IOC3 internal loopback DMA test failed.`

  The IOC3 internal loopback DMA test failed. Refer to Section 5.2.6, "IOC3 Internal Loopback Test," for more information about the failure messages that the IOC3 internal loopback test returns.

- `enet_all: Ethernet PHY chip internal loopback DMA test failed.`

  The PHY chip internal loopback DMA test failed. Refer to Section 5.2.7, "PHY Chip Internal Loopback Test," for more information about the failure messages that the PHY chip internal loopback DMA test returns.

- `enet_all: Ethernet Twister chip internal loopback DMA test failed.`

  The Twister chip internal loopback DMA test failed. Refer to Section 5.2.8, "Twister Chip Internal Loopback Test," for more information about the failure messages that the Twister chip internal loopback DMA test returns.

- `enet_all: Ethernet external loopback DMA test failed.`

  The external loopback DMA test failed. Refer to Section 5.2.9, "External Loopback DMA Test," for more information about the failure messages that the external loopback DMA test returns.

- `enet_all: Xtalk stress test failed.`

  The XTALK stress test failed. Refer to Section 5.2.12, "XTALK Stress Test," for more information about the failure messages that the XTALK stress test returns.

For fault isolation information, refer to the individual test descriptions.

5.2.2 Ethernet SSRAM Test

The Ethernet SSRAM test checks the 128 KB Ethernet packet buffering SSRAM, the SSRAM's interface to the IOC3 ASIC, and the parity generation and checking logic in the IOC3.

5.2.2.1 Prerequisites to Running This Test

Ensure that the following hardware is functional before you run this test:

- Non-Ethernet-specific logic on the IOC3 ASIC
- PCI bus address/data
- Bridge ASIC
- XTALK interface between the Bridge ASIC and XBOW chip
- XBOW chip, midplane, and all IP27 board components

5.2.2.2 How to Run This Test

To run the Ethernet SSRAM test from the >> prompt, enter the `diag_enet -t ssram` command and any other command line options that you want to use. For example:

```
>> diag_enet -t ssram [options]
```

Refer to Section 5.2 for descriptions of all available *options*.

5.2.2.3 Test Description

This test uses the following algorithm:

1. Perform an address walk test.
   - Write a test pattern to addresses 0, 1, 2, 4, 8, ...
   - Read the pattern back and verify that the data that was read back equals the data that was written.
2. Perform a parity test.
   - Write a value with good parity = 0 to memory location 0.
   - Read the value back, verify that the value that was read back equals the value that was written, and verify that the error bit did not set.
   - Write a value with good parity = 1 to memory location 0.
   - Read the value back, verify that the value that was read back equals the written value, and verify that the error bit did not set.
   - Write a value with bad parity = 0 to memory location 0.
   - Read the value back, verify that the value that was read back equals the value that was written, and verify that the error bit set.
   - Write a value with bad parity = 1 to memory location 0.

- Read the value back, verify that the value that was read back equals the value that was written, and verify that the error bit set.

3. In heavy and manufacturing modes only, perform a data pattern test.
   - Write a test pattern to all memory addresses, alternating between low and high addresses.
   - Read the data back and verify that the data that was read back equals the data that was written.

In normal mode, the address walk test and parity tests are the only tests that run.

In heavy and manufacturing modes, all tests run.

5.2.2.4 Failure Information

The Ethernet SSRAM test generates the following failure messages:

- `enet_ssram: could not get the ioc3 base addr`

  This indicates a programming error in the test; you should never see this error message in the field.

- `enet_ssram: ssram address walk error. Ex:` *expected_value* `Recv:` *received_value*

  The address walk test failed; the data that was read back (*received_value*) did not equal the data that was written (*expected_value*).

  The failing FRU is the BaseIO board. The failing components are the IOC3-to-SSRAM wires (most likely), the SSRAM, or the IOC3 ASIC (least likely).

- `enet_ssram: ssram parity test error. Exp:` *expected_value* `Recv:` *received_value*

  The parity test failed; the data that was read back (*received_value*) did not equal the data that was written (*expected_value*).

  The failing FRU is the BaseIO board. The failing component might be the IOC3 ASIC or the SSRAM.

- `enet_ssram: IOC3 ssram test failed...`

  This message will be removed from a future version of the test.

- `enet_ssram: **Data miscompare(A)** offset:` *offset_value* `exp:` *expected_value* `recv:` *received_value*

  The data pattern test failed. The failing FRU is the BaseIO board. The most likely failing component is the SSRAM. Other (less likely) failing components include the IOC3-to-SSRAM wires and the IOC3 ASIC.

5.2.3 PHY Register Test

The PHY register test checks the reset values of registers on the PHY chip (a National Semiconductor DP83840 chip). This test also verifies the capability to read and write

selected bits in selected registers on the PHY chip; this test checks only the bits and registers that the Ethernet driver and the other Ethernet diagnostics use.

Note: Some documentation, including the BaseIO board schematics, refers to the PHY chip as the X-chip.

5.2.3.1 Prerequisites to Running This Test

Ensure that the following hardware is functional before you run this test:

- Non-Ethernet-specific logic on the IOC3 ASIC
- PCI bus address/data
- Bridge ASIC
- XTALK interface between the Bridge ASIC and XBOW chip
- XBOW chip, midplane, and all IP27 board components

5.2.3.2 How to Run This Test

To run the PHY register test from the >> prompt, enter the `diag_enet -t phyreg` command and any other command line options that you want to use. For example:

```
>> diag_enet -t phyreg [options]
```

Refer to Section 5.2 for descriptions of all available *options*.

5.2.3.3 Test Description

This test uses the following algorithm:

1. Verify the presence of the PHY chip.
2. If the PHY chip is present, perform the following actions:
   - Reset the PHY chip with a software reset.
   - Verify the reset values of all registers in the PHY chip, and generate failure messages for any miscompares that are detected.
   - Read and write data to selected bits in registers 0, 23, and 24. Compare the data that is read back to the written data, and generate failure messages for any miscompares that are detected.

5.2.3.4 Failure Information

The PHY register test generates the following failure messages:

- `enet_phy_reg: Read of PHY register 0 failed (returned 0xffff).`

  The PHY presence test failed. The failing FRU is the BaseIO board. The most likely failing component is the IOC3-to-PHY wiring (the MII interface wires or the reset wire). Other possible failing components are the PHY chip and its clocks or the IOC3 ASIC.

- `enet_phy_reg: Timed out waiting for PHY to rest.`

  The software reset of the PHY chip did not complete. The failing FRU is the BaseIO board. The most likely failing hardware component is the PHY chip and its clocks. Other less likely failing components are the IOC3-to-PHY wiring and the IOC3 ASIC.

- `enet_phy_reg: Timed out on MICR_BUSY before read.`
  `enet_phy_reg: Timed out on MICR_BUSY during read.`

  A read operation of a PHY register did not complete. The failing FRU is the BaseIO board. The most likely failing hardware component is the IOC3-to-PHY wiring. Other possible failing components include the PHY chip and the IOC3 ASIC.

- `enet_phy_reg: Timed out on MICR_BUSY before write.`
  `enet_phy_reg: Timed out on MICR_BUSY during write.`

  A write operation to a PHY register did not complete. The failing FRU is the BaseIO board. The most likely failing hardware component is the IOC3-to-PHY wiring. Other possible failing components include the PHY chip and the IOC3 ASIC.

- `enet_phy_reg: Warning - PHY Identifier register #1 value` *value* `is not supported by this diag!`
  `enet_phy_reg: Warning - PHY Identifier register #2 value` *value* `is not supported by this diag!`
  `enet_phy_reg: Warning - PHY revision register value` *value* `is not supported by this diag!`

  The test failed while checking a reset value in one of the registers that might change in future revisions of the PHY chip. This most likely indicates that the BaseIO board includes a newer version of the PHY chip and the diagnostic software has not been updated with this information. It could also indicate a failing PHY chip.

- `enet_phy_reg: Warning - unexpected Reg` *register* `reset value, exp:` *expected* `mask:` *mask_value* `got:` *received*

  The test failed while checking the register reset values. The failing FRU is the BaseIO board. The most likely failing hardware component is the PHY chip. Another possible failing component is the wiring from the IOC3 ASIC to the PHY chip.

- `enet_phy_reg: Reg 0 value wrong, exp:` *expected* `mask:` *mask_value* `got:` *received*
  `enet_phy_reg: Reg 23 value wrong, exp:` *expected* `mask:` *mask_value* `got:` *received*
  `enet_phy_reg: Reg 24 value wrong, exp:` *expected* `mask:` *mask_value* `got:` *received*

  A failure occurred during the write and read test to register 0, 23, or 24. The failing FRU is the BaseIO board. The most likely failing hardware component is the PHY chip. Another possible failing component is the wiring from the IOC3 ASIC to the PHY chip.

5.2.4 TX Clock Test

The TX clock test verifies the presence of the TX clock sent from the PHY chip to the IOC3 ASIC. This test reads the IOC3 ETCSR register and checks the NO_TX_CLK bit.

5.2.4.1 Prerequisites to Running This Test

Ensure that the following hardware is functional before you run this test:

- Non-Ethernet-specific logic on the IOC3 ASIC
- PCI bus address/data
- Bridge ASIC
- XTALK interface between the Bridge ASIC and XBOW chip
- XBOW chip, midplane, and all IP27 board components

5.2.4.2 How to Run This Test

To run the TX clock test from the >> prompt, enter the `diag_enet -t txclk` command and any other command line options that you want to use. For example:

```
>> diag_enet -t txclk [options]
```

Refer to Section 5.2 for descriptions of all available *options*.

5.2.4.3 Test Description

This test uses the following algorithm:

1. Read the IOC3 ETCSR register.
2. Check the NO_TX_CLK bit.

This test runs the same in normal and heavy modes. Manufacturing mode returns more status information than normal and heavy modes.

5.2.4.4 Failure Information

The TX clock test returns the following failure message:

- `enet_tx_clk: IOC3 is not seeing the PHY TX clock.`

  The TX clock test failed. The failing FRU is the BaseIO board. The most likely failing hardware component is the TX clock wire. Other (less likely) failing components are the PHY chip and the IOC3 ASIC.

5.2.5 NIC Test

The NIC test checks the number in a can (NIC) device (a Dallas Semiconductor chip) on the BaseIO board; the NIC device stores the Ethernet MAC address. This test reads the NIC and then performs further testing if the results of the read are incorrect.

5.2.5.1 Prerequisites to Running This Test

Ensure that the following hardware is functional before you run this test:

- Non-Ethernet-specific logic on the IOC3 ASIC
- PCI bus address/data
- Bridge ASIC
- XTALK interface between the Bridge ASIC and XBOW chip
- XBOW chip, midplane, and all IP27 board components

5.2.5.2 How to Run This Test

To run the NIC test from the >> prompt, enter the `diag_enet -t nic` command and any other command line options that you want to use. For example:

```
>> diag_enet -t nic [options]
```

Refer to Section 5.2 for descriptions of all available *options*.

5.2.5.3 Test Description

This test uses the following algorithm:

1. Call an external function that reads the NIC.
2. Provide further testing if the NIC value is incorrect.

This test runs the same in normal and heavy modes. Manufacturing mode returns more status information than normal and heavy modes.

5.2.5.4 Failure Information

The NIC test returns the following failure messages:

- `nic_eaddr: No presence pulse from NIC`
  `nic_eaddr: (check for an empty NIC socket)`

  The NIC test failed. The failing FRU is the BaseIO board. The most likely cause is a missing NIC device. Other possible causes are a failing socket, failing NIC device, failing wiring from the IOC ASIC to the socket, or a failing IOC3 ASIC.

- `nic_eaddr: Invalid enet MAC command CRC byte` *value*`, s/b 0x8d`
  `nic_eaddr: Invalid enet crc16` *value*`, s/b 0xb001`
  `nic_eaddr: Invalid enet length byte` *value* `found in NIC, s/b 0x0a`
  `nic_eaddr: Invalid enet MAC crc16` *value*`, s/b 0xb001`
  `nic_eaddr: Failed to find readable MAC NIC, rc` *value*

  The NIC test failed. The failing FRU is the BaseIO board. The failing hardware component is the NIC device.

5.2.6 IOC3 Internal Loopback Test

The IOC3 internal loopback test verifies Ethernet DMA by using the internal loopback features of the IOC3 ASIC. (The IOC3 loopback DMA runs at 66.7 Mb/s by using its internal clock as a TX clock.)

5.2.6.1 Prerequisites to Running This Test

Ensure that the following hardware is functional before you run this test:

- Non-Ethernet-specific logic on the IOC3 ASIC
- PCI bus address/data
- Bridge ASIC
- XTALK interface between the Bridge ASIC and XBOW chip
- XBOW chip, midplane, and all IP27 board components

5.2.6.2 How to Run This Test

To run the IOC3 internal loopback test from the >> prompt, enter the `diag_enet -t ioc3_loop` command and any other command line options that you want to use. For example:

```
>> diag_enet -t ioc3_loop [options]
```

Refer to Section 5.2 for descriptions of all available *options*.

5.2.6.3 Test Description

This test uses the following algorithm:

1. Allocate memory for the TX descriptor ring and write the base address to the ETBR registers in the IOC3.
2. Allocate memory for the TX data buffers and write their addresses to the descriptor ring entries and to the command and buffer count fields.
3. Generate test data and write it to the TX buffers. (The test pattern is designed to stress PCI SSO.)
4. Allocate memory for the RX buffer pointer ring and write the base address to the ERBR registers on the IOC3.
5. Allocate memory for the RX buffers.
6. Enable error interrupts on the Bridge ASIC.
7. Reset the IOC3 Ethernet DMA engines, initialize the IOC3 registers for IOC3 loopback DMA, and enable the TX_EMPTY interrupt.
8. Write to the ring pointers in a way that causes the DMA to occur in one burst of 10 packets (in normal mode) or one burst of 495 packets (in heavy and manufacturing modes) and then stop.

9. Write to the EMCR register on the IOC3 to enable DMA.
10. Execute a polling loop to wait for the IOC3 to post TX_EMPTY in the EISR. During the polling loop, monitor for unexpected error conditions. Time out if this takes too long.
11. Check the Bridge ASIC ISR to verify that the interrupt went from the IOC3 to the Bridge.
12. Wait for the valid bit to set in the RX buffer for the last packet. While waiting for this to occur, the test monitors for unexpected error conditions.
13. Time out if it takes too long for the valid bit to set in the RX buffer for the last packet.
14. Check all received packets: Check the status field for each packet and also verify that the data matches the data that was transmitted.
15. Clean up: Free the allocated memory, restore interrupt enables, and reset the PHY chip.

In normal mode, this test transmits and receives 10 packets; each packet contains 301 bytes of data. In heavy and manufacturing modes, this test transmits and receives 495 packets; each packet contains 1,501 bytes of data.

5.2.6.4 Failure Information

The IOC3 internal loopback test returns the following failure messages:

- `enet_ioc3_loop: Timed out on EMCR_ARB_DIAG_IDLE trying to reset enet.`

  The IOC3 PCI arbiter was stuck with a request to the Bridge before this test could start. The failing FRU is the BaseIO board. Failing hardware component information is not available because the test could not run.

- `enet_ioc3_loop: Unexpected EISR condition while waiting for TX_EMPTY (`*`string1`*`: `*`string2`*`).`
  `enet_ioc3_loop: EISR value = `*`e_value`*
  `enet_ioc3_loop: Elapsed time = `*`time`*` us`

  The IOC3 ASIC detected an error during a DMA operation. The EISR value (shown in *e_value*) indicates the failing hardware. Refer to Table 5-1 for descriptions of the EISR values.

Table 5-1 IOC3 Loopback Test Failure Information Based on the EISR Values

EISR Value	Failure Information
0x00000010	RX_MEMERR RX DMA access to onboard PCI bus failed The failing FRU is the BaseIO board. The cause could be a PCI parity error.
0x00000020	RX_PARERR SSRAM parity error during RX DMA The failing FRU is the BaseIO board. The most likely cause is failing SSRAM. Other possible causes are failing IOC3-to-SSRAM wiring or a failing IOC3 ASIC.
0x00200000	TX_BUF_UFLO This message is a by-product of an SSRAM parity error.

Table 5-1 (continued) IOC3 Loopback Test Failure Information Based on the EISR Values

EISR Value	Failure Information
0x02000000	TX_MEMERR TX DMA access to onboard PCI bus failed. The failing FRU is the BaseIO board. The cause could be a PCI parity error or an unrecoverable error in the memory subsystem.
0x04000000	TX_PARERR SSRAM parity error during TX DMA. The failing FRU is the BaseIO board. The most likely cause is failing SSRAM. Other possible causes are failing IOC3-to-SSRAM wiring or a failing IOC3 ASIC.

- `enet_ioc3_loop: Timed out waiting for TX_EMPTY interrupt (`*string*`: `*string*`).`

  The DMA operation did not complete. The failing FRU is the BaseIO board. The failing hardware component is difficult to isolate.

- `enet_ioc3_loop: Bridge ISR bit `*value*` did not set after TX_EMPTY interrupt (`*string*`: `*string*`).`
  `enet_ioc3_loop: ==> Bridge ISR = `*value*

  The interrupt between the IOC3 and the Bridge was not successful. The failing FRU is the BaseIO board. The most likely cause is a failing interrupt wire. Other possible causes are a failing IOC3 ASIC or a failing Bridge ASIC.

- `enet_ioc3_loop: Timed out polling RX buffer valid bit. (`*string*`: `*string*`).`
  `enet_ioc3_loop: `*value*` of `*value*` packets were received.`

  The RX buffers in main memory did not receive all of the packets that were sent. The failing FRU is the BaseIO board. The failing hardware component is difficult to isolate.

- `enet_ioc3_loop: Unexpected EISR condition while waiting for RX buffer valid bit (`*string*`: `*string*`)`
  `enet_ioc3_loop: EISR value = `*e_value*

  The IOC3 ASIC detected an error during a DMA operation. The EISR value (shown in *e_value*) indicates the failing hardware. Refer to Table 5-1 for descriptions of the EISR values.

- `enet_ioc3_loop: RX data miscompare on packet `*value*` first word.`
  `enet_ioc3_loop: ==> exp: `*expected*` recv: `*received*` (`*string*`: `*string*`).`

  The first word (which contains the valid bit, the byte count, and the IP checksum) of the RX data is corrupted. The failing FRU is the BaseIO board. The failing hardware component is difficult to isolate.

- `enet_ioc3_loop: RX data miscompare on packet `*value*` status word.`
  `enet_ioc3_loop: ==> exp: `*expected*` recv: `*received*` (`*string*`: `*string*`).`

  The status word of the RX data does not have the expected value. (This should never happen in IOC3 loopback mode.) The failing FRU is the BaseIO board.

- `enet_ioc3_loop: RX data miscompare on packet `*value*` DW `*value*`.`
  `enet_ioc3_loop: ==> exp: `*expected*` recv: `*received*` (`*string*`: `*string*`).`

  The specified doubleword of data is corrupted. The failing FRU is the BaseIO board. The failing hardware component is difficult to isolate.

- `enet_ioc3_loop: RX data miscompare on packet` *value* `byte` *value*.
  `enet_ioc3_loop: ==> exp:` *expected* `recv:` *received* `(`*string*`:` *string*`)`.

  One of the last bytes of data is corrupted. The failing FRU is the BaseIO board. The failing hardware component is difficult to isolate.

5.2.7 PHY Chip Internal Loopback Test

The PHY chip loopback test checks Ethernet DMA by using the internal loopback features of the PHY chip on the BaseIO board. The PHY chip loopback DMA runs at 10 Mb/s and also at 100 Mb/s.

5.2.7.1 Prerequisites to Running This Test

Ensure that the following hardware is functional before you run this test:

- Non-Ethernet-specific logic on the IOC3 ASIC
- PCI bus address/data
- Bridge ASIC
- XTALK interface between the Bridge ASIC and XBOW chip
- XBOW chip, midplane, and all IP27 board components

Also ensure that the following Ethernet tests pass before you run this test:

- IOC3 internal loopback test
- TX clock test
- PHY register test

5.2.7.2 How to Run This Test

To run the PHY chip internal loopback test from the >> prompt, enter the **`diag_enet -t phy_loop`** command and any other command line options that you want to use. For example:

```
>> diag_enet -t phy_loop [options]
```

Refer to Section 5.2 for descriptions of all available *options*.

5.2.7.3 Test Description

This test uses the following algorithm twice (once for 10 Mb/s testing and then again for 100 Mb/s testing):

1. Allocate memory for the TX descriptor ring and write the base address to the ETBR registers on the IOC3.
2. Allocate memory for the TX data buffers and write their addresses to the descriptor ring entries and to the command and buffer count fields.

3. Generate test data and write it to the TX buffers. (The test pattern is designed to stress PCI SSO.)
4. Allocate memory for the RX buffer pointer ring and write the base address to the ERBR registers on the IOC3.
5. Allocate memory for the RX buffers.
6. Enable error interrupts on the Bridge ASIC.
7. Reset the IOC3 Ethernet DMA engines, initialize the IOC3 registers for PHY loopback DMA, and enable the TX_EMPTY interrupt.
8. Write to the ring pointers in a way that causes the DMA to occur in one burst of 10 packets (in normal mode) or one burst of 495 packets (in heavy and manufacturing modes) and then stop.
9. Reset the PHY chip and initialize it for internal loopback at 10 Mb/s (during first test sequence) or 100 Mb/s (during second test sequence).
10. Write to the EMCR register on the IOC3 to enable DMA.
11. Wait for the IOC3 to post TX_EMPTY in the EISR. During the polling loop, monitor for unexpected error conditions. Time out if this takes too long.
12. Check the Bridge ASIC ISR to verify that the interrupt got from the IOC3 to the Bridge.
13. Wait for the valid bit to set in the RX buffer for the last packet. While waiting for this to occur, the test monitors for unexpected error conditions.
14. Time out if it takes too long for the valid bit to set in the RX buffer for the last packet.
15. Check all received packets: Check the status field for each packet and also verify that the data matches the data that was transmitted.
16. Clean up: Free the allocated memory, restore interrupt enables, and reset the PHY chip.

In normal mode, this test transmits and receives 10 packets; each packet contains 301 bytes of data. In heavy and manufacturing modes, this test transmits and receives 495 packets; each packet contains 1,501 bytes of data.

5.2.7.4 Failure Information

The PHY chip internal loopback test returns the same messages as the IOC3 internal loopback test (refer to Section 5.2.6.4, "Failure Information,"). If the IOC3 internal loopback test passed, any failures that are listed as "difficult to isolate" are likely to be caused by a fault in the IOC3 chip to PHY chip wiring, the PHY chip, or the IOC3 ASIC.

This test also returns the following failure messages:

- `enet_phy_loop: Unexpected EISR condition while waiting for TX_EMPTY (`*`string`*`: `*`string`*`).`
  `enet_phy_loop: EISR value = `*`e_value`*
  `enet_phy_loop: Elapsed time = `*`value`*` us`

The IOC3 detected an error during the DMA operation. The EISR value (shown in the output above as *e_value*) indicates the failing hardware. Refer to Table 5-2 for descriptions of the EISR values.

Table 5-2 PHY Chip Internal Loopback Test Failure Information Based on the EISR Values

EISR Value	Failure Information
0x00000010	RX_MEMERR RX DMA access to onboard PCI bus failed The failing FRU is the BaseIO board. The cause could be a PCI parity error.
0x00000020	RX_PARERR SSRAM parity error during RX DMA The failing FRU is the BaseIO board. The most likely cause is failing SSRAM. Other possible causes are failing IOC3-to-SSRAM wiring or a failing IOC3 ASIC.
0x00020000	TX_RTRY There were excessive collisions. The failing FRU is the BaseIO board. The cause is a stuck MII_COL signal from the PHY chip to the IOC3 ASIC.
0x00040000	TX_EXDEF There were excessive deferrals. The failing FRU is the BaseIO board. The cause is a stuck MII_CRS signal from the PHY chip to the IOC3 ASIC.
0x00200000	TX_BUF_UFLO This message is a by-product of an SSRAM parity error.
0x02000000	TX_MEMERR TX DMA access to onboard PCI bus failed. The failing FRU is the BaseIO board. The cause could be a PCI parity error or an unrecoverable error in the memory subsystem.
0x04000000	TX_PARERR SSRAM parity error during TX DMA. The failing FRU is the BaseIO board. The most likely cause is failing SSRAM. Other possible causes are failing IOC3-to-SSRAM wiring or a failing IOC3 ASIC.

Any failures in this test are strong evidence that the failing FRU is the BaseIO board.

5.2.8 Twister Chip Internal Loopback Test

The Twister chip internal loopback test verifies Ethernet DMA by using the internal loopback features of the Twister chip (a National Semiconductor DP83223 chip). The Twister chip loopback DMA runs at 100 Mb/s.

5.2.8.1 Prerequisites to Running This Test

Ensure that the following hardware is functional before you run this test:

- Non-Ethernet-specific logic on the IOC3 ASIC
- PCI bus address/data
- Bridge ASIC
- XTALK interface between the Bridge ASIC and XBOW chip
- XBOW chip, midplane, and all IP27 board components

Also ensure that the PHY chip internal loopback test runs without errors before you run this test.

5.2.8.2 How to Run This Test

To run the Twister chip internal loopback test from the >> prompt, enter the `diag_enet -t tw_loop` command and any other command line options that you want to use. For example:

```
>> diag_enet -t tw_loop [options]
```

Refer to Section 5.2 for descriptions of all available *options*.

5.2.8.3 Test Description

This test uses the following algorithm:

1. Allocate memory for the TX descriptor ring and write the base address to the ETBR registers on the IOC3.
2. Allocate memory for the TX data buffers and write their addresses to the descriptor ring entries and to the command and buffer count fields.
3. Generate test data and write it to the TX buffers. (The test pattern is designed to stress PCI SSO.)
4. Allocate memory for the RX buffer pointer ring and write the base address to the ERBR registers on the IOC3.
5. Allocate memory for the RX buffers.
6. Enable error interrupts on the Bridge ASIC.
7. Reset the IOC3 Ethernet DMA engines, initialize the IOC3 registers for Twister loopback DMA, and enable the TX_EMPTY interrupt.
8. Write to the ring pointers in a way that causes the DMA to occur in one burst of 10 packets (in normal mode) or one burst of 495 packets (in heavy and manufacturing modes) and then stop.
9. Reset the PHY chip and initialize it for Twister loopback at 100 Mb/s.
10. Write to the EMCR register on the IOC3 to enable DMA.

11. Wait for the IOC3 to post TX_EMPTY in the EISR. During the polling loop, monitor for unexpected error conditions. Time out if this takes too long.
12. Check the Bridge ASIC ISR to verify that the interrupt got from the IOC3 to the Bridge.
13. Wait for the valid bit to set in the RX buffer for the last packet. While waiting for this to occur, the test monitors for unexpected error conditions.
14. Time out if it takes too long for the valid bit to set in the RX buffer for the last packet.
15. Check all received packets: Check the status field for each packet and also verify that the data matches the data that was transmitted.
16. Clean up: Free the allocated memory, restore interrupt enables, and reset the PHY chip.

In normal mode, this test transmits and receives 10 packets; each packet contains 301 bytes of data. In heavy and manufacturing modes, this test transmits and receives 495 packets; each packet contains 1,501 bytes of data.

5.2.8.4 Failure Information

The Twister chip internal loopback test returns the same messages that the PHY chip internal loopback test returns (refer to Section 5.2.7.4, "Failure Information,"). Failures that were listed as "difficult to isolate" are likely caused by faulty wiring between the PHY chip and the Twister chip.

5.2.9 External Loopback DMA Test

The external loopback DMA test verifies Ethernet DMA by using an external loopback cable or connector. This test verifies the autonegotiation feature and verifies the DMA at 10 Mb/s and 100 Mb/s.

5.2.9.1 Prerequisites to Running This Test

Ensure that the following hardware is functional before you run this test:

- Non-Ethernet-specific logic on the IOC3 ASIC
- PCI bus address/data
- Bridge ASIC
- XTALK interface between the Bridge ASIC and XBOW chip
- XBOW chip, midplane, and all IP27 board components

Also ensure that the Twister chip internal loopback test runs without errors before you run this test.

Install a loopback cable or connector before you run this test.

5.2.9.2 How to Run This Test

This test does not run during the boot sequence. It runs only from the command line. To run this test from the >> prompt, enter `diag_enet -t ext_loop [`*options*`]`. For example:

```
>> diag_enet -t ext_loop [options]
```

Refer to Section 5.2 for descriptions of all available *options*.

5.2.9.3 Test Description

This test uses the following algorithm twice (once at 10 Mb/s and then again at 100 Mb/s):

1. Allocate memory for the TX descriptor ring and write the base address to the ETBR registers on the IOC3.
2. Allocate memory for the TX data buffers and write their addresses to the descriptor ring entries and to the command and buffer count fields.
3. Generate test data and write it to the TX buffers. (The test pattern is designed to stress PCI SSO.)
4. Allocate memory for the RX buffer pointer ring and write the base address to the ERBR registers on the IOC3.
5. Allocate memory for the RX buffers.
6. Enable error interrupts on the Bridge ASIC.
7. Reset the IOC3 Ethernet DMA engines, initialize the IOC3 registers for external loopback DMA, and enable the TX_EMPTY interrupt.
8. Write to the ring pointers in a way that causes the DMA to occur in one burst of 10 packets (in normal mode) or one burst of 495 packets (in heavy and manufacturing modes) and then stop.
9. Reset the PHY chip.
10. Test the autonegotiation features of the hardware.
11. Write to the EMCR register on the IOC3 to enable DMA.
12. Wait for the IOC3 to post TX_EMPTY in the EISR. During the polling loop, monitor for unexpected error conditions. Time out if this takes too long.
13. Check the Bridge ASIC ISR to verify that the interrupt went from the IOC3 to the Bridge.
14. Wait for the valid bit to set in the RX buffer for the last packet. While waiting for this to occur, the test monitors for unexpected error conditions.
15. Time out if it takes too long for the valid bit to set in the RX buffer for the last packet.
16. Check all received packets: Check the status field for each packet and also verify that the data matches the data that was transmitted.
17. Clean up: Free the allocated memory, restore interrupt enables, and reset the PHY chip.

In normal mode, this test transmits and receives 10 packets; each packet contains 301 bytes of data. In heavy mode, this test transmits and receives 495 packets; each packet contains 1,501 bytes of data. This test does not run in manufacturing mode.

5.2.9.4 Failure Information

This test returns the same failure information as the Twister loopback test (refer to Section 5.2.8.4, "Failure Information,"). If the Twister loopback test passed, then failures that were "difficult to isolate" are most likely caused by one of the following conditions.

- A faulty transformer on the BaseIO board.
- The wires between the PHY and the magnetics (for autonegotiation or 10 Mb/s DMA test).
- The wires between the Twister chip and the magnetics (for 100 Mb/s DMA test).
- The RJ45 connector.
- The loopback cable/connector.

If the loopback cable and connector are good, the failing FRU is the BaseIO board.

The following messages, which indicate that the autonegotiation test failed, will be seen only with the external loopback test and indicate a failure of the autonegotiation test:

- `enet_ext_loop: Phy did not complete auto-negotiation with itself via external loopback cable.`
- `enet_ext_loop: PHY reg 6 LP_AN_ABLE bit not set after auto-negotiation.`
  `==> Reg 0 =` *value*
  `==> Reg 4 =` *value*
  `==> Reg 5 =` *value*
  `==> Reg 6 =` *value*
  `*** Are you sure there is an external loopback? ***`
- `enet_ext_loop: PHY reg 5 mode support bits (9:5) do not match reg 4 mode support bits after autonegotiation. Reg 4:` *value* `Reg 5:` *value*

5.2.10 External Loopback DMA (10 Mb/s) Test

The external loopback DMA (10 Mb/s) test is a version of the external loopback DMA test; this version has the following differences: it does not test autonegotiation and it runs only at 10 Mb/s.

5.2.10.1 Prerequisites to Running This Test

Ensure that the following hardware is functional before you run this test:

- Non-Ethernet-specific logic on the IOC3 ASIC
- PCI bus address/data
- Bridge ASIC

- XTALK interface between the Bridge ASIC and XBOW chip
- XBOW chip, midplane, and all IP27 board components

Also ensure that the Twister chip internal loopback test runs without errors before you run this test.

Install a loopback cable or connector before you run this test.

5.2.10.2 How to Run This Test

This test does not run during the boot sequence. It runs only from the command line. To run this test from the >> prompt, enter `diag_enet -t ext_loop_10 [`*options*`]`. For example:

```
>> diag_enet -t ext_loop_10 [options]
```

Refer to Section 5.2 for descriptions of all available *options*.

5.2.10.3 Test Description

This test uses the following algorithm:

1. Allocate memory for the TX descriptor ring and write the base address to the ETBR registers on the IOC3.
2. Allocate memory for the TX data buffers and write their addresses to the descriptor ring entries and to the command and buffer count fields.
3. Generate test data and write it to the TX buffers. (The test pattern is designed to stress PCI SSO.)
4. Allocate memory for the RX buffer pointer ring and write the base address to the ERBR registers on the IOC3.
5. Allocate memory for the RX buffers.
6. Enable error interrupts on the Bridge ASIC.
7. Reset the IOC3 Ethernet DMA engines, initialize the IOC3 registers for external loopback DMA, and enable the TX_EMPTY interrupt.
8. Write to the ring pointers in a way that causes the DMA to occur in one burst of 10 packets (in normal mode) or one burst of 495 packets (in heavy and manufacturing modes) and then stop.
9. Reset the PHY chip.
10. Write to the EMCR register on the IOC3 to enable DMA.
11. Wait for the IOC3 to post TX_EMPTY in the EISR. During the polling loop, monitor for unexpected error conditions. Time out if this takes too long.
12. Check the Bridge ASIC ISR to verify that the interrupt got from the IOC3 to the Bridge.
13. Wait for the valid bit to set in the RX buffer for the last packet. While waiting for this to occur, the test monitors for unexpected error conditions.

14. Time out if it takes too long for the valid bit to set in the RX buffer for the last packet.
15. Check all received packets: Check the status field for each packet and also verify that the data matches the data that was transmitted.
16. Clean up: Free the allocated memory, restore interrupt enables, and reset the PHY chip.

In normal mode, this test transmits and receives 10 packets; each packet contains 301 bytes of data. In heavy mode, this test transmits and receives 495 packets; each packet contains 1,501 bytes of data. This test does not run in manufacturing mode.

5.2.10.4 Failure Information

This test returns the same failure messages as the external loopback DMA test (refer to Section 5.2.9.4, "Failure Information,").

5.2.11 External Loopback DMA (100 Mb/s) Test

The external loopback DMA (100 Mb/s) test is a version of the external loopback DMA test; the differences are that it does not test autonegotiation and it runs only at 100 Mb/s.

5.2.11.1 Prerequisites to Running This Test

Ensure that the following hardware is functional before you run this test:

- Non-Ethernet-specific logic on the IOC3 ASIC
- PCI bus address/data
- Bridge ASIC
- XTALK interface between the Bridge ASIC and XBOW chip
- XBOW chip, midplane, and all IP27 board components

Also ensure that the Twister chip internal loopback test runs without errors before you run this test.

You should install a loopback cable or connector before you run this test.

5.2.11.2 How to Run This Test

This test does not run during the boot sequence. It runs only from the command line. To run this test from the >> prompt, enter `diag_enet -t ext_loop_100 [`*options*`]`. For example:

```
>> diag_enet -t ext_loop_100 [options]
```

Refer to Section 5.2 for descriptions of all available *options*.

5.2.11.3 Test Description

This test uses the following algorithm:

1. Allocate memory for the TX descriptor ring and write the base address to the ETBR registers on the IOC3.
2. Allocate memory for the TX data buffers and write their addresses to the descriptor ring entries and to the command and buffer count fields.
3. Generate test data and write it to the TX buffers. (The test pattern is designed to stress PCI SSO.)
4. Allocate memory for the RX buffer pointer ring and write the base address to the ERBR registers on the IOC3.
5. Allocate memory for the RX buffers.
6. Enable error interrupts on the Bridge ASIC.
7. Reset the IOC3 Ethernet DMA engines, initialize the IOC3 registers for external loopback DMA, and enable the TX_EMPTY interrupt.
8. Write to the ring pointers in a way that causes the DMA to occur in one burst of 10 packets (in normal mode) or one burst of 495 packets (in heavy and manufacturing modes) and then stop.
9. Reset the PHY chip.
10. Write to the EMCR register on the IOC3 to enable DMA.
11. Wait for the IOC3 to post TX_EMPTY in the EISR. During the polling loop, monitor for unexpected error conditions. Time out if this takes too long.
12. Check the Bridge ASIC ISR to verify that the interrupt went from the IOC3 to the Bridge.
13. Wait for the valid bit to set in the RX buffer for the last packet. While waiting for this to occur, the test monitors for unexpected error conditions.
14. Time out if it takes too long for the valid bit to set in the RX buffer for the last packet.
15. Check all received packets: Check the status field for each packet and also verify that the data matches the data that was transmitted.
16. Clean up: Free the allocated memory, restore interrupt enables, and reset the PHY chip.

In normal mode, this test transmits and receives 10 packets; each packet contains 301 bytes of data. In heavy mode, this test transmits and receives 495 packets; each packet contains 1,501 bytes of data. This test does not run in manufacturing mode.

5.2.11.4 Failure Information

This test returns the same failure messages as the external loopback DMA test (refer to Section 5.2.9.4, "Failure Information,").

5.2.12 XTALK Stress Test

The XTALK stress test is a 100 Mb/s DMA test that uses a continuous stream of DMA for 1 to 5 seconds, instead of a single burst of 10 or 495 packets like the other DMA tests. This test does not check the received data, but it does check the IOC3 and Bridge interrupt registers for errors. This test uses a data pattern that is designed to stress SSO on the LLP links in the XTALK path.

5.2.12.1 Prerequisites to Running This Test

Ensure that the following hardware is functional before you run this test:

- Non-Ethernet-specific logic on the IOC3 ASIC
- PCI bus address/data
- Bridge ASIC
- XTALK interface between the Bridge ASIC and XBOW chip
- XBOW chip, midplane, and all IP27 board components

Also ensure that the Twister chip internal loopback test runs without errors before you run this test.

5.2.12.2 How to Run This Test

This test does not run as part of the boot sequence in normal mode; it does run during the boot sequence in heavy and manufacturing modes.

To run the XTALK stress test from the >> prompt, enter the `diag_enet -t xtalk` command and any other command line options that you want to use. For example:

```
>> diag_enet -t xtalk [options]
```

Refer to Section 5.2 for descriptions of all available *options*. If you use the **-e** option, this test uses the external loopback path instead of the Twister internal loopback. If you use the **-r** option, this test prints the retry counts every 50 seconds.

When you run this test manually from the command line, the command reads and clears out the LLP retry count registers on the HUB, XBOW, and Bridge before and after the test runs.

5.2.12.3 Test Description

This test uses the following algorithm:

1. Allocate memory for the TX descriptor ring and write the base address to the ETBR registers on the IOC3.
2. Allocate memory for the TX data buffers and write their addresses to the descriptor ring entries and to the command and buffer count fields.

3. Generate test data and write it to the TX buffers. (The test pattern is designed to stress LLP.)
4. Allocate memory for the RX buffer pointer ring and write the base address to the ERBR registers on the IOC3.
5. Allocate memory for the RX buffers.
6. Enable error interrupts on the Bridge ASIC.
7. Reset the IOC3 Ethernet DMA engines, initialize the IOC3 registers for Twister loopback DMA, and enable the TX_EMPTY interrupt.
8. Write to the ring pointers in a way that causes the DMA to occur in one burst of 10 packets (in normal mode) or one burst of 495 packets (in heavy and manufacturing modes) and then stop.
9. Reset the PHY chip and initialize it for Twister loopback at 100 Mb/s.
10. Write to the EMCR register on the IOC3 to enable DMA.
11. Check for errors in the IOC3 EISR and the Bridge ISR and chase the consume pointers. (In normal mode, repeat this step for 1 second. In heavy and manufacturing modes, repeat this step for 5 seconds.)
12. Wait for the IOC3 to post TX_EMPTY in the EISR. During the polling loop, monitor for unexpected error conditions. Time out if this takes too long.
13. Check the Bridge ASIC ISR to verify that the interrupt got from the IOC3 to the Bridge.
14. Wait for the valid bit to set in the RX buffer for the last packet. While waiting for this to occur, the test monitors for unexpected error conditions.
15. Time out if it takes too long for the valid bit to set in the RX buffer for the last packet.
16. Check all received packets: Check the status field for each packet and also verify that the data matches the data that was transmitted.
17. Clean up: Free the allocated memory, restore interrupt enables, and reset the PHY chip.

In normal mode, this test performs 1 second of DMA operations. In heavy and manufacturing modes, this test performs 5 seconds of DMA operations.

5.2.12.4 Failure Information

The failure messages are the same as the failure messages for the Twister chip internal loopback test (refer to Section 5.2.8.4, "Failure Information,"). This test also includes the following messages:

- `enet_xtalk_stress: RX CONSUME did not increment from Xtalk stress loop` *value* `to` *value*.
  `enet_xtalk_stress: TX CONSUME did not increment from Xtalk stress loop` *value* `to` *value*.

  The DMA hung. The failing hardware component is difficult to isolate.

- `enet_xtalk_stress: Unexpected EISR condition in Xtalk stress loop` *value*.
  `enet_xtalk_stress:  EISR value =` *e_value*

  The IOC3 detected an error during the DMA operation. The EISR value (shown in the output above as *e_value*) indicates the failing hardware. Refer to Table 5-1 for descriptions of the EISR values.

- `enet_xtalk_stress: Unexpected Bridge ISR condition in Xtalk stress loop` *value*.
  `enet_xtalk_stress:  Bridge ISR value =` *value*

  An unexpected Bridge ISR condition occurred. Most cases are difficult to isolate.

Chapter 6

POD Mode

This chapter describes POD mode. POD mode is an interactive mode that uses the command line interpreter that is in the PROM. You can use POD mode on a crashed system to:

- examine the contents of the CPU registers
- examine the contents of the support chip registers
- examine the contents of memory
- enable or disable Node board components (for example, CPUs and memory banks)
- manually run several of the IP27 power-on diagnostics

6.1 Entering POD Mode

The system enters POD mode under the following conditions:

- A power-on diagnostic fails.
- An unexpected interrupt occurs while the IP27 PROM code or BaseIO PROM code is executing.
- The boot sequence detects that the boot stop point debug DIP switches (switches 4 and 5) are set to Off and On, On and Off, or On and On.

  Refer to Table 1-4 in Section 1.2, "Modifying How the Power-On Diagnostics are Run," for more information about the debug DIP switches. The boot sequence stops and ends POD mode at the specified time.
- You issue a nonmaskable interrupt (NMI) at the PROM level.
- You issue two NMIs at the kernel level. The first takes the system to the PROM level; the second takes the system to the POD prompt.
- You type the *pod* command at the BaseIO command prompt (for example, `>>pod`).

6.2 Dirty Exclusive, Uncached, and Cached Modes

POD mode can run in three modes: dirty exclusive (Dex) mode, uncached (Unc) mode, or cached (Cac) mode.

6.2.1 Dirty Exclusive Mode

Dirty exclusive (Dex) mode is the mode in which the power-on diagnostics require the fewest system resources to run. Dirty exclusive mode does not require memory; it accesses the program code directly from the PROM and uses the data cache in the R10000 processors as the memory for its stack. Dirty exclusive mode does not use the secondary cache. Nonmaskable interrupts and uncaught exceptions typically return you to the Dex mode prompt.

Dex mode is very slow because PROM instruction fetches are slow. Avoid performing long memory tests or flashing remote PROMs from Dex mode.

When you are loading or storing data in memory or running memory tests in Dex mode, ensure that you do not access cached memory addresses.

6.2.2 Uncached Mode

Uncached (Unc) mode is the mode in which power-on diagnostics place the program code, data, and stack into uncached memory. This mode is similar to Cac mode, but Unc mode runs slower; however, program execution is quicker in Unc mode than in Dex mode.

6.2.3 Cached Mode

Cached (Cac) mode is the mode in which the power-on diagnostics place the program code, data, and stack into cached memory. Cached mode is available only after memory has been probed and configured. POD execution is much quicker in Cac mode than in Dex mode or Unc mode.

6.3 POD Mode Prompt

The POD mode prompt displays information about the current state of POD mode. An example POD mode prompt is:

```
POD Dex MSC 001 MDerr 3A>
```

Table 6-1 uses this example to describe the contents of the POD prompt.

Table 6-1 Components of the POD Mode Prompt

Prompt Component	Description
`POD`	This field is always `POD`. It indicates that you are using POD mode.
`Dex`	This field indicates the mode in which POD mode is running. Possible values are `Dex`, `Cac`, and `Unc`.

Table 6-1 (continued) Components of the POD Mode Prompt

Prompt Component	Description
`MSC`	This field indicates the UART on which the POD prompt is being displayed. Possible values with matching descriptions are: `MSC`--The MSC alternate console port is displaying the POD prompt. `Junk`--A junk UART is displaying the POD prompt. `IOC3`--An IOC3 UART is displaying the POD prompt. `Talk`--A netuart is displaying the POD prompt.
`001`	This field displays the NASID value for the current Node, which is the Node ID.
`MDerr`	This field occasionally indicates errors or alternate operating modes Possible values with matching descriptions are: `MDerr`--MD errors are pending. (Enter the *error* command to view the errors. Enter the *clear* command to clear the errors.) `AltRegs`--POD mode is using the kernel register set instead of the PROM register set.
`3A`	The number in this field indicates the slot location of the Node board that contains the CPU (1 to 4) that is displaying the POD mode prompt. The letter in this field indicates which CPU on the Node board (A or B) is displaying the POD mode prompt.

Many POD mode commands (including memory test commands, PROM flashing commands, and loop or repeat statements) can be aborted by typing Ctrl+C.

You can use the following editing characters at the POD prompt.

Character	Description
Ctrl+C	Abort the last POD mode command (works for many POD mode commands, including running the memory tests, flashing the PROM, and executing the *loop* and *repeat* commands).
Ctrl+H	Backspace one character.
Backspace	Backspace one character.
Delete	Backspace one character.
Ctrl+R	Redisplay the line.
Ctrl+U	Erase the line.
!!	Use the contents of previous command line.
!$	Use the last word from previous command line.
Ctrl+*x* Ctrl+*y*	Use the contents of previous command but substitute first occurrence of string *x* with string *y*.

Environment variables set with the *setenv* command are substituted wherever the name of an environment variable appears in backward single quotation marks (` and `). For example:

```
Set "range" to? n:1 u: 0x3ff0000 2m
POD Dex Hub 000 1A> dirtest `range`
POD Dex Hub 000 1A> dirinit `range`
POD Dex Hub 000 1A> memtest `range`
```

You can enter more than one command on a line by separating the commands with semicolons (for example: `POD Dex Hub 000 1A> ld pi_rt_cout; sd pi_rt_count 0`).

For commands that take other commands as arguments (for example, *loop*), you can enter a single command or you can enclose multiple commands in curly braces { }. The following examples show both methods:

```
POD Dex Hub 000 1A> loop pi_rt_count
POD Dex Hub 000 1A> loop {ld pi_rt_count; sd pi_rt_count 0}
```

6.4 Data Types

Table 6-2 describes the data types that are available for POD commands.

Table 6-2 POD Mode Data Types

Data Type	Description
ADDR	A memory address (either a number or an arithmetic expression enclosed in parentheses)
BITNO	A bit number (0 to 63)
COUNT	Count or length
CMD	A POD mode command
EXPR	An arithmetic expression
GPRNAME	A general-purpose register name (either text or $*NUM* format)
LEN	The length of a range of memory (in bytes)
LINE	A cache line (either as a line value or the address of a line)
NODE	A Node identification value (also called a Numa Address Space IDentifier, or NASID)
PCINUM	A PCI device number
REGNO	A register number (0 to 31)
SLICE	A CPU (either A or B)
SLOT	A slot number
START	The starting address for a range of memory

Table 6-2 (continued) POD Mode Data Types

Data Type	Description
VADDR	A CrayLink vector address (an offset value or an NI or RR register name)
VAL	A value (either number or an arithmetic expression in parentheses)
VEC	A CrayLink vector path (for example 0x15)

6.5 Constants

POD mode constants are similar to C language constants. You must include a 0x prefix for hexadecimal numbers (for example, 0xbead). You must include a 0 prefix for octal numbers (for example, 03777). You must include a 0b prefix for binary numbers (for example, 0b010010). POD mode treats all other numbers as decimal numbers (for example, 123).

POD mode provides several suffixes that you can use as shortcuts for large numbers. Table 6-3 describes the POD mode suffixes that are available.

Table 6-3 Possible Suffixes for POD Mode Constants

Suffix	Description	Example	Value
k	Multiplies the value by 1 KB (1,024)	100k	102,400
m	Multiplies the value by 1 MB (1,048,576)	32m	33,554,432
g	Multiplies the value by 1 GB (1,073,741,824)	1g	1,073,741,824

6.6 Expressions

You can use arithmetic expressions within POD commands. Many commands require that you enclose expressions within parentheses. The following examples show expressions within POD commands:

```
POD Dex Hub 000 1A> px (0xeeadbbef >> 6) & 0xff
POD Dex Hub 000 1A> sd md_memory_config (ld md_memory_config | 077777777)
```

Commands that require strings may require quotation marks around the strings. Within quotation marks, standard C language escape sequences (\b, \t, \n, \r, \\, and \ddd) are permitted. Strings must be 31 characters or less.

6.7 Address Modifiers

When you are using POD commands that require addresses, you can use the address modifiers shown in Table 6-4 to use constants to enter addresses (the modifier should come before the constant).

Table 6-4 POD Mode Address Modifiers

Modifier	Description	Example	Equivalent Value
h:	Set the top byte to 0x90 (HSPEC).	h:256m	0x9000000010000000 (LBOOT)
i:	Set the top byte to 0x92 (IO).	i:(16m+32)	0x9200000001000020
m:	Set the top byte to 0x94 (MSPEC).	m:0	0x9400000000000000
u:	Set the top byte to 0x96 (UNCAC).	u:0	0x9600000000000000
c:	Set the top byte to 0xa8 (CAC).	c:0	0xa800000000000000
p:	Set the top byte to 0x00 (PHYS).	p:c:0x100	0x100
s:	Sign extend the word.	s:0xbfc00000	0xffffffffbfc00000
n:#	Add the NASID, where # is the NASID. (You can use this modifier only if CrayLink routing is configured.)	n:2 c:1	0xa800000200000001
w:#	Set the top byte to 0x92. Set bit 24 and add widget, where # is the widget. If it is remote, add bit 23.	w:1 0x200 n:1 w:1 0x200	0x9200000001000200 0x9200000101800200
b:#	Add the offset for the memory bank (# multiplied by 512 MB).	n:3 b:5 u:2k	0x96000003a0000800
L:	Convert an address to the corresponding directory entry LO in HSPEC.	L:0x2100560	0x90000000c08402a0
H:	Convert an address to the corresponding directory entry HI in HSPEC.	H:u:0x2100560	0x90000000c08402a8
P:#	Convert an address to the corresponding protection entry, where # is the region.	P:5 0x2100560	0x90000000c0840028
E:	Convert an address to the corresponding ECC word address.	E:0x2100560	0x9000000080840158

6.8 Hardware Registers

POD mode recognizes register names for most of the registers in the system. This lets you use register names in POD commands instead of having to manually look up and type the register address. When you use a name in an expression, POD replaces it with the appropriate address.

POD mode recognizes the names of R10000 COP0 registers (Cause, Status, and so on) and R10000 general-purpose registers (v0, a1, ta0, gp, r5, and so on). You can also refer to general-purpose registers as $0, $1, $2, and so on. You can refer to floating-point registers as $f0, $f1, $f2, and so on.

The names of HUB registers are prefixed according to the five sections of the HUB with PI_, MD_, II_, NI_, and CORE_.

You can use the names of HUB NI and Router registers in place of the vector address in some vector commands. Router registers are prefixed with RR_. Crossbow (XBOW) registers are prefixed with XB_.

Refer to Appendix A, "Register Names in POD Mode," for a complete list of available register names.

Register names are case-insensitive; you may abbreviate a name to the extent that the name can be distinguished from other register names. For example, you can abbreviate md_refresh_control as md_ref, as the following example shows:

```
POD Dex IOC3 000 4A> px (md_refresh_control + 8)
0x9200000001200028
POD Dex IOC3 000 4A> pr n:1 md_ref
Register: MD_REFRESH_CONTROL (0x9200000101a00020)
Value   : 0x0000000000000000 (loaded from remote register)
<63> RW  ENABLE 0x1
<23:12> RW  COUNTER 0x2e0
<11:00> RW  CNT_THRESH 0x504
```

6.9 Symbols

POD mode recognizes symbols within the PROM (and the kernel if you loaded one). You can use these symbols in expressions; POD mode replaces them with their corresponding addresses.

Examples:

```
POD Dex IOC3 000 4A> px main
0xc00000001fc07784
POD Dex IOC3 000 4A> dis (main+0x1c) 4
[main+0x001c, 0x1fc077a0]:      0001083c        dsll32  at,at,#0
[main+0x0020, 0x1fc077a4]:      64420000        daddiu  v0,v0,0x0
[main+0x0024, 0x1fc077a8]:      0039082c        dadd    at,at,t9
[main+0x0028, 0x1fc077ac]:      0002103c        dsll32  v0,v0,#0
```

```
POD Dex IOC3 000 4A> px "memcpy"
0xc00000001fc1db14
```

Notice that you must enclose some symbols in quotation marks (like memcpy in the preceding example) because they are reserved words (POD commands) in POD mode.

6.10 POD Mode Commands

6.10.1 Conventions

The following conventions apply to the POD commands that are described in this document.

- Square brackets [ ] indicate optional command line arguments.
- A vertical bar | separates an either/or choice.
- Command arguments are shown as all upper-case, italic text and are called variables (for example, the *VEC* variable). The names used for the variables indicate the type of data with which you can replace the argument (refer to Section 6.4, "Data Types," for more information about the available data types).

6.10.2 Quick Reference Descriptions

Table 6-5 provides an alphabetical listing of quick-reference descriptions of the POD mode commands. The number in the "Page" column indicates the page on which you can find a more detailed description of the command.

Table 6-5 POD Mode Commands (Quick Reference)

Command	Description	Syntax	Page
?	Display help information.	*? [CMDNAME]*	6-28
adline	Dump the specified data cache line.	*adline LINE*	6-31
adtag	Dump the specified data cache tag.	*adtag LINE*	6-30
ailine	Dump the specified instruction cache line.	*ailine LINE*	6-31
aitag	Dump the specified instruction cache tag.	*aitag LINE*	6-31
altregs	Use the alternate registers.	*altregs NUM*	6-28
asline	Dump the specified secondary cache line.	*asline LINE*	6-31
astag	Dump the specified secondary cache tag.	*astag LINE*	6-31
bist	Run the HUB self-test.	*bist* le | ae | lr | ar [n:*NODE*]	6-33
call	Call a subroutine.	*call ADDR [A0 [A1* [A2 [A3]]]]	6-26
chklink	Run the local link connection test.	*chklink*	6-41

Table 6-5 (continued) POD Mode Commands (Quick Reference)

Command	Description	Syntax	Page
clear	Clear errors.	*clear*	6-39
cpu	Switch to the specified CPU.	*cpu* [[n:*NODE*] a | b]	6-44
crb	Dump I/O interface (II) CRBs.	*crb* [n:*NODE*]	6-47
crbx	Dump the CRBs in a 137-column-wide format.	*crbx* [n:*NODE*]	6-47
dbg	Display or change the virtual and physical debug DIP switch settings.	*dbg* [*VAL VAL*]	6-52
delay	Delay command execution.	*delay MICROSEC*	6-25
dgbrdg	Run the Bridge test.	*dgbrdg* [mn | mh | mm] [n*NODE*] [s*SLOT*]	6-42
dgconf	Run the PCICONFIG test.	*dgconf* [mn | mh | mm] [n*NODE*] [s*SLOT*]	6-42
dgpci	Run the PCI bus test.	*dgpci* [mn | mh| mm] [n*NODE*] [s*SLOT*] [p*PCINUM*]	6-43
dgsdma	Run the serial DMA test.	*dgsdma* [mn | mh | mm | mx] [n*NODE*] [s*SLOT*] [p*PCINUM*]	6-43
dgspio	Run the serial programmed I/O test.	*dgspio* [mn | mh | mm | mx] [n*NODE*] [s*SLOT*] [p*PCINUM*]	6-43
dgxbow	Run the XBOW test.	*dgxbow* [mn | mh | mm] [n*NODE*]	6-42
dips	Read and display the debug DIP switch settings.	*dips*	6-51
dirinit	Initialize a range of memory to the "Unowned" state.	*dirinit START LEN*	6-34
dirstate	Scan a range of addresses for the directory state that you specify.	*dirstate* [*STATE* [*BASE* [*LEN*]]]	6-41
dirtest	Perform directory test and initialization.	*dirtest START LEN*	6-35
dis	Disassemble the data stored in the specified memory locations.	*dis ADDR* [*COUNT*]	6-53
disable	Disable the specified CPU.	*disable* n:*NODE* [*SLICE*]	6-33
disc	Discover the CrayLink interconnect.	*disc*	6-46
dline	Dump the specified data cache line.	*dline LINE*	6-30
dtag	Dump the specified data cache tag.	*dtag LINE*	6-29
dumpspool	Dump the PI error spool.	*dumpspool* [n:*NODE SLICE*]	6-53

Table 6-5 (continued) POD Mode Commands (Quick Reference)

Command	Description	Syntax	Page
ecc	Enable or disable ECC mode.	*ecc* [on | off]	6-38
echo	Display a string on the console.	*echo "STRING"*	6-13
edump_bri	Dump Bridge error information to memory.	*edump_bri* [n:*NODE*]	6-54
enable	Enable the specified CPU.	*enable* n:*NODE* [*SLICE*]	6-34
error	Display errors.	*error*	6-38
error_dump	Dump the error information.	*error_dump*	6-53
exec	Load and execute a segment.	*exec* [*SEGNAME* [*FLAG*]]	6-44
flash	Program the remote PROM.	*flash NODE* [...]	6-47
flush	Flush and invalidate caches.	*flush*	6-29
for	Repeat the specified command(s) for the number of iterations determined by the specified expression.	*for* (*CMD;EXPR;CMD*) *CMD*	6-24
fru	Run the FRU analyzer program.	*fru* [1 | 2]	6-59
go	Set memory mode.	*go* dex | unc | cac	6-32
help	Display help information.	*help* [*CMDNAME*]	6-28
hubsde	Run the HUB send data error test.	*hubsde*	6-41
if	Execute the specified command(s) depending on the value of a conditional.	*if* (*EXPR*) *CMD*	6-25
iline	Dump the specified instruction cache line.	*iline LINE*	6-30
im	Set the R10000 processor interrupt mask.	*im* [*BYTE*]	6-39
initalllogs	Initialize all PROM logs for all modules in the system.	*initalllogs*	6-57
initlog	Initialize the PROM log.	*initlog* [n:*NODE*]	6-56
inval	Invalidate cache(s).	*inval* [i] [d] [s]	6-29
ioc3	Use the IOC3 UART.	*ioc3*	6-45
itag	Dump the specified instruction cache tag.	*itag LINE*	6-29
jump	Invalidate cache and continue execution at a specific address.	*jump ADDR* [A0 [A1]]	6-26
junk	Use the JunkBus UART.	*junk*	6-45
kdebug	Debug the kernel.	*kdebug* [*STACKADDR*]	6-59
kern_sym	Use the kernel symbol table.	*kern_sym*	6-53
la	Load byte data and display it as ASCII characters.	*la ADDR* [*COUNT*]	6-18

Table 6-5 (continued) POD Mode Commands (Quick Reference)

Command	Description	Syntax	Page
lb	Load a byte.	*lb ADDR* [*COUNT*]	6-17
ld	Load a doubleword.	*ld ADDR* [*COUNT*]	6-18
lh	Load a halfword.	*lh ADDR* [*COUNT*]	6-17
log	Display entries from the PROM log for the specified Node.	*log* [n:*NODE*] [*SKIP* [*COUNT*]]	6-57
loop	Repeat the specified command(s) forever.	*loop CMD*	6-24
lw	Load a word.	*lw ADDR* [*COUNT*]	6-17
maxerr	Test the error limit.	*maxerr COUNT*	6-38
memcmp	Compare data (by bytes) in one memory location to data in another memory location.	*memcmp DST SRC LEN*	6-40
memcpy	Copy memory (by bytes) from one memory location to another memory location.	*memcpy DST SRC LEN*	6-39
meminit	Clear memory.	*meminit START LEN*	6-35
memset	Fill memory with a byte pattern.	*memset DST BYTE LEN*	6-39
memsum	Add data (by bytes).	*memsum SRC LEN*	6-40
memtest	Perform memory sanity test.	*memtest START LEN*	6-36
modnic	Display the NIC value of the current module.	*modnic*	6-49
module	Display or change a module number.	*module* [*VAL*]	6-52
msc	Use the system controller UART. (This command was previously *elsc*.)	*msc*	6-45
nic	Display the HUB NIC.	*nic* [n:*NODE*]	6-48
nm	Look up a PROM address.	*nm ADDR*	6-15
nmi	Send a nonmaskable interrupt (NMI) to a Node.	*nmi NODE* [a | b]	6-27
node	Get or set a Node.	*node* [[*VEC*] *ID*]	6-46
pa	Print an address.	*pa ADDR* [*BITNO*]	6-14
pas	Display or change the MSC password.	*pas* ["*STRING*"]	6-52
pb	Print the value of an expression in binary.	*pb EXPR*	6-14
pcfg	Dump the pcfg structure.	*pcfg* [n:*NODE*]	6-46
pd	Print the value of an expression in decimal.	*pd EXPR*	6-14
pf	Print the contents of an R10000 COP1 floating point register.	*pf* [*REGNO*]	6-16
po	Print the value of an expression in octal.	*po EXPR*	6-14

Table 6-5 (continued) POD Mode Commands (Quick Reference)

Command	Description	Syntax	Page
pr	Print the contents of a register.	*pr* [*GPRNAME* [*VAL* | ~]]	6-15
printenv	Print the value of the specified environment variable(s).	*printenv* [n:*NODE*] [*VAR*]	6-58
px	Print the value of an expression in hexadecimal.	*px EXPR*	6-14
qual	Enable or disable quality mode.	*qual* [on | off]	6-38
rbist	Run the Router self-test.	*rbist* le | ae | lr | ar *VEC*	6-33
reconf	Initialize the memory/directory section (MD) of the HUB.	*reconf*	6-40
repeat	Repeat count.	*repeat COUNT CMD*	6-23
reset	Reset the system.	*reset*	6-26
rlog	Display entries from the PROM log of the specified Node (most recent entry to least recent entry).	rlog [n:NODE] [SKIP [COUNT]]	6-57
rnic	Read the Router NIC.	*rnic* [*VEC*]	6-48
route	Set up a route.	*route* [*VEC NODE*]	6-23
rstat	Dump and clear the Router status.	*rstat VEC*	6-47
rtab	Dump the route table.	*rtab* [*VEC*]	6-46
rtrsde	Run the Router send data error test.	*rtrsde*	6-41
santest	Perform the memory sanity test.	*santest ADDR*	6-37
sb	Store a byte.	*sb ADDR* [*VAL* [*COUNT*]]	6-18
sc	Send a command to the MSC.	*sc* ["*COMMAND*"]	6-49
scandir	Scan directory states.	*scandir ADDR* [*LEN*]	6-40
scr	Read from the system controller NVRAM.	*scr ADDR* [*COUNT*]	6-49
scw	Write to the system controller NVRAM.	*scw ADDR* [*VAL* [*COUNT*]]	6-49
sd	Store a doubleword.	*sd ADDR* [*VAL* [*COUNT*]]	6-20
sdv	Store a doubleword and verify it.	*sdv ADDR COUNT*	6-20
segs	List the segments.	*segs* [*FLAG*]	6-44
setenv	Set an environment variable.	*setenv* [n:*NODE*] *VAR* ["*STRING*"]	6-57
sf	Store the contents of an R10000 COP1 floating point register.	*sf REGNO VAL*	6-21
sh	Store a halfword.	*sh ADDR* [*VAL* [*COUNT*]]	6-19
slave	Go into slave mode.	*slave*	6-43

Table 6-5 (continued) POD Mode Commands (Quick Reference)

Command	Description	Syntax	Page
sleep	Sleep for a specified number of seconds.	*sleep SEC*	6-25
sline	Dump the specified secondary cache line.	*sline LINE*	6-30
softreset	Perform a soft reset on a Node.	*softreset* n:*NODE*	n/a
sr	Store the contents of a register.	*sr GPRNAME VAL*	6-20
stag	Dump the specified secondary cache tag.	*stag LINE*	6-30
sw	Store a word.	*sw ADDR* [*VAL* [*COUNT*]]	6-19
talk	Use the net UART.	*talk* [n:*NODE* a | b]	6-45
tdisable	Temporarily disable the specified CPU.	*tdisable* n:*NODE* [*SLICE*]	6-34
time	Perform benchmark timing.	*time CMD*	6-26
tlbc	Clear the TLB.	*tlbc* [*INDEX*]	6-32
tlbr	Read the TLB.	*tlbr* [*INDEX*]	6-32
traplog	Trap log.	*traplog*	6-28
unsetenv	Remove an environment variable.	*unsetenv* [n:*NODE*] *VAR*	6-58
verbose	Specify whether or not the power-on diagnostics run in verbose mode.	*verbose* [0 | 1]	6-59
version	Display the PROM version.	*version*	6-28
vr	Perform a vector read operation.	*vr VEC VADDR*	6-21
vw	Perform a vector write operation.	*vw VEC VADDR VAL*	6-22
vx	Perform a vector exchange operation.	*vx VEC VADDR VAL*	6-22
while	Repeat the specified command(s) while the specified expression evaluates to true.	*while* (*EXPR*) *CMD*	6-24
why	Display the current exception and NMI status.	*why*	6-27

6.10.3 Printing a String

6.10.3.1 echo Command

The *echo* command displays a string on the console. This command uses the following syntax:

echo "STRING"

The *STRING* variable specifies the string that this command should display.

6.10.4 Printing Expressions

6.10.4.1 px Command

The *px* command evaluates an arithmetic expression and prints the result as a hexadecimal number. This command uses the following syntax:

px EXPR

6.10.4.2 pd Command

The *pd* command evaluates an arithmetic expression and prints the result as a decimal number. This command uses the following syntax:

pd EXPR

6.10.4.3 po Command

The *po* command evaluates an arithmetic expression and prints the result as an octal number. This command uses the following syntax:

po EXPR

6.10.4.4 pb command

The *pb* command evaluates an arithmetic expression and prints the result as a binary number. This command uses the following syntax:

pb EXPR

6.10.4.5 pa Command

The *pa* command decodes an individual memory bit to print the NASID value, the physical DIMM slot where the bit resides, and the data line number of the bit. This command uses the following syntax:

pa ADDR [*BITNO*]

If you do not specify a bit number with the *BITNO* variable, this command uses bit 0.

Example:

```
POD Dex IOC3 000 4A> pa 0x40000 21
Address 0x0000000000040000 bit 21 is on NASID 0 MMXL0 DIMML line 39
```

6.10.4.6 nm Command

The *nm* command prints the function in the PROM symbol table (or kernel symbol table) that contains the address that you specify. This command prints the name of the function and the offset into the function of the memory location that you specify. This command uses the following syntax:

nm ADDR

Example:

```
POD Dex Hub 000 1A> nm 0xc00000001fc07c04
main+0x488
```

6.10.5 Displaying Registers

6.10.5.1 pr Command

The *pr* command prints the contents of an R10000 register or a HUB register. This command uses the following syntax:

pr [*GPRNAME* [*VAL* | ~]]

If you do not specify any arguments, this command prints out the contents of all of the R10000 general-purpose registers (GPRs). If you specify a register name with the *GPRNAME* argument, this command loads a value from the register and decodes it into bit fields. If you specify a register name with the *GPRNAME* argument and a value with the *VAL* argument, this command decodes the specified value into bit fields.

If you specify a register name with the *GPRNAME* argument and also specify a tilde (~), this command decodes the register's reset default value into bit fields.

If you specify a value with the *VAL* argument or specify the tilde (~), you can use this command with router registers (RR_*) instead of R10000 GPRs.

Examples:

- `pr`

  This example prints the contents of all 32 R10000 general-purpose registers (GPRs) with their numbers and symbolic names.

- ```
  pr 2
  pr $2
  pr r2
  pr r02
  pr v0
  ```

  All of these examples print the value of GPR 2.

- `pr cause`

  This example prints the value of the R10000 COP0 CAUSE register.
  ```

- `pr pi`

  This example prints all HUB PI registers because the specified register name is ambiguous.

- `pr md_memory_config`

  This example prints the address, contents, and individually decoded bit fields for the specified HUB register.

- `pr n:1 md_memory_config`

  This example prints the address, contents, and individually decoded bit fields for the specified HUB register located on the remote Node that has a NASID value of 1.

- `pr md_memory_config 0x001010004fffffff`

  This example decodes the specified constant into individual bit fields for the specified HUB register.

- `pr md_memory_config ~`

  This example decodes the reset default value of the specified HUB register into individual bit fields for the register.

- `pr rr_status_rev_id ~`

  This example decodes the reset default value of the specified Router register into individual bit fields for the register.

6.10.5.2 pf Command

The *pf* command prints the contents of R10000 COP1 floating point registers (FPRs). This command uses the following syntax:

pf [*REGNO*]

Examples:

- `pf`

  This example prints the contents of all 32 R10000 floating point registers (FPRs).

- `pf 3`

  This example prints the contents of the $f3 FPR.

Note: You can also print the contents of the FPRs by using the *px*, *pd*, *po*, and *pb* commands, which accept floating point register names in expressions.

6.10.6 Loading From Memory and Storing to Memory

6.10.6.1 lb Command

The *lb* command loads byte data from the memory address that you specify and displays the result as a hexadecimal number. This command uses the following syntax:

lb ADDR [COUNT]

The *ADDR* variable specifies the memory address from which the command should load the data. The *COUNT* variable specifies the number of bytes that the command should load from memory; if you specify a *COUNT*, this command loads the specified number of bytes and displays the data in columns.

An exception may occur if the address that you specify is not properly aligned. Do not try to access cached memory if you are using Dex mode.

6.10.6.2 lh Command

The *lh* command loads halfword data from the memory address that you specify and displays the result as a hexadecimal number. This command uses the following syntax:

lh ADDR [COUNT]

The *ADDR* variable specifies the memory address from which the command should load the data. The *COUNT* variable specifies the number of halfwords that the command should load from memory; if you specify a *COUNT*, this command loads the specified number of halfwords and displays the data in columns.

An exception may occur if the address that you specify is not properly aligned. Do not try to access cached memory if you are using Dex mode.

6.10.6.3 lw Command

The *lw* command loads word data from the memory address that you specify and displays the result as a hexadecimal number. This command uses the following syntax:

lw ADDR [COUNT]

The *ADDR* variable specifies the memory address from which the command should load the data. The *COUNT* variable specifies the number of words that the command should load from memory; if you specify a *COUNT*, this command loads the specified number of words and displays the data in columns.

An exception may occur if the address that you specify is not properly aligned. Do not try to access cached memory if you are using Dex mode.

6.10.6.4 ld Command

The *ld* command loads doubleword data from the memory address that you specify and displays the result as a hexadecimal number. This command uses the following syntax:

ld ADDR [COUNT]

The *ADDR* variable specifies the memory address from which the command should load the data. The *COUNT* variable specifies the number of doublewords that the command should load from memory; if you specify a *COUNT*, this command loads the specified number of doublewords and displays the data in columns.

An exception may occur if the address that you specify is not properly aligned. Do not try to access cached memory if you are using Dex mode.

6.10.6.5 la Command

The *la* command loads byte data from the memory address that you specify. This command displays the result as ASCII characters (instead of hexadecimal values like the *lb* command). This command uses the following syntax:

la ADDR [COUNT]

The *ADDR* variable specifies the memory address from which the command should load the data. The *COUNT* variable specifies the number of bytes that the command should load from memory. Do not try to access cached memory if you are using Dex mode.

Note: This command displays bytes that are not between 0x20 (the ASCII code for a space) and 0x7e (the ASCII code for a tilde) as octal values.

6.10.6.6 sb Command

The *sb* command stores byte data in the address that you specify. This command uses the following syntax:

sb ADDR [VAL [COUNT]]

The *ADDR* variable specifies the memory address at which the command should store the data. The *VAL* variable specifies the data value that the command should store at the specified address. The *COUNT* variable specifies the number of times the data value should be written into memory: The command starts writing the data at the specified address and continues writing for *COUNT* number of writes.

If you do not specify the optional *VAL* and *COUNT* variables, the command enters an interactive mode. The interactive mode shows the current contents of the memory location that you specified with the *ADDR* variable. If you want to change the contents, type a hexadecimal value (without the leading 0x) and press the Enter key. If you do not want to change the contents of the memory location, press the Enter key. When you press the Enter key, the interactive mode displays the next location in memory. Interactive mode continues until you type a period (.) and press the Enter key. Do not try to access cached memory if you are using Dex mode.

6.10.6.7 sh Command

The *sh* command stores halfword data in the address that you specify. This command uses the following syntax:

sh ADDR [*VAL* [*COUNT*]]

The *ADDR* variable specifies the memory address at which the command should store the data. The *VAL* variable specifies the data value that the command should store at the specified address. The *COUNT* variable specifies the number of times the data value should be written into memory: The command starts writing the data at the specified address and continues writing for *COUNT* number of writes.

If you do not specify the optional *VAL* and *COUNT* variables, the command enters an interactive mode. The interactive mode shows the current contents of the memory location that you specified with the *ADDR* variable. If you want to change the contents, type a hexadecimal value (without the leading 0x) and press the Enter key. If you do not want to change the contents of the memory location, press the Enter key. When you press the Enter key, the interactive mode displays the next location in memory. Interactive mode continues until you type a period (.) and press the Enter key.

Do not try to access cached memory if you are using Dex mode.

6.10.6.8 sw Command

The *sw* command stores word data in the address that you specify. This command uses the following syntax:

sw ADDR [*VAL* [*COUNT*]]

The *ADDR* variable specifies the memory address at which the command should store the data. The *VAL* variable specifies the data value that the command should store at the specified address. The *COUNT* variable specifies the number of times the data value should be written into memory: The command starts writing the data at the specified address and continues writing for *COUNT* number of writes.

If you do not specify the optional *VAL* and *COUNT* variables, the command enters an interactive mode. The interactive mode shows the current contents of the memory location that you specified with the *ADDR* variable. If you want to change the contents, type a hexadecimal value (without the leading 0x) and press the Enter key. If you do not want to change the contents of the memory location, press the Enter key. When you press the Enter key, the interactive mode displays the next location in memory. Interactive mode continues until you type a period (.) and press the Enter key.

Do not try to access cached memory if you are using Dex mode.

6.10.6.9 sd Command

The *sd* command stores doubleword data in the address that you specify. This command uses the following syntax:

sd ADDR [VAL [COUNT]]

The *ADDR* variable specifies the memory address at which the command should store the data. The *VAL* variable specifies the data value that the command should store at the specified address. The *COUNT* variable specifies the number of times the data value should be written into memory: The command starts writing the data at the specified address and continues writing for *COUNT* number of writes.

If you do not specify the optional *VAL* and *COUNT* variables, the command enters an interactive mode. The interactive mode shows the current contents of the memory location that you specified with the *ADDR* variable. If you want to change the contents, type a hexadecimal value (without the leading 0x) and press the Enter key. If you do not want to change the contents of the memory location, press the Enter key. When you press the Enter key, the interactive mode displays the next location in memory. Interactive mode continues until you type a period (.) and press the Enter key.

Do not try to access cached memory if you are using Dex mode.

6.10.6.10 sdv Command

The *sdv* command stores doubleword data at the address that you specify, reads the data back, and verifies the result. This command prints an error message if the data read back does not match the data that was written. This command uses the following syntax:

sdv ADDR COUNT

The *ADDR* variable specifies the memory address at which the command should store the data. The *VAL* variable specifies the data value that the command should store at the specified address. The *COUNT* variable specifies the number of times the data value should be written into memory: The command starts writing the data at the specified address and continues writing for *COUNT* number of writes.

Do not try to access cached memory if you are using Dex mode.

6.10.7 Setting Registers

6.10.7.1 sr Command

The *sr* command sets an R10000 general-purpose register (GPR) to a specified value. This command uses the following syntax:

sr GPRNAME VAL

The *GPRNAME* variable specifies the general-purpose register that you want to modify. The *VAL* variable specifies the value that you want to store in the register.

Note: This command modifies the register storage area, where the registers are saved in the event of an exception or NMI.

6.10.7.2 sf Command

The *sf* command sets an R10000 floating point register (FPR) to the value that you specify. This command uses the following syntax:

sf REGNO VAL

The *REGNO* specifies the floating point register that you want to modify; refer to Appendix A, "Register Names in POD Mode," for a list of the available registers. The *VAL* variable specifies the value that you want to store in the register.

6.10.8 Performing Operations Between Modules

Early in the system boot, before the PROM has configured the CrayLink Interconnect, Node IDs are not yet assigned (they are all zero) and Router tables are not yet programmed. In this state, it is not valid to access the memory and I/O spaces of remote Nodes using regular remote memory accesses (where the Node ID has been placed into physical address bits 40:32, such as by using the n: modifier).

The Origin2000 HUB and Routers have a feature called vector addressing that allows access to any remote HUB's Network Interface (NI_) register subset and to all Router (RR_) registers. Reading or writing a remote HUB or Router register requires you to specify a path called a vector.

A vector contains a string of output port numbers from 1 to 6 (corresponding directly to the hardware labeling A through F), one per nibble, appearing in reverse order. For example, the vector 0x512 indicates the request should leave the current HUB and go into a Router, leave port 2 of that Router and go into another Router, leave port 1 of that Router and go into another Router, and finally leave port 5 of that Router and go into either a HUB or Router, depending on what appears on that last port. The response automatically follows the reverse path and arrives back at the local HUB. The vector 0x0 accesses the device immediately connected to the local HUB, which is usually a Router but may be a HUB if a Null Router is being used.

The following commands enable you to perform vector operations.

6.10.8.1 vr Command

The *vr* command performs a CrayLink vector read operation. This command uses the following syntax:

vr VEC VADDR

The *VEC* variable specifies the CrayLink vector of the HUB or Router on which you want to perform the read operation. The *VADDR* variable is the address of the register (on the specified HUB or Router) that the command should read.

For HUB operations, you can use a numerical value or the symbolic HUB register names for the *VADDR* variable. For Router operations, you can use a numerical value or the symbolic Router register names for the *VADDR* variable.

Examples:

- `vr 0 0`

  This example reads address 0 from the device that is immediately connected to the local HUB. (Address 0 is the NI_STATUS_REV_ID on a HUB and RR_STATUS_REV_ID on a Router. The status register was deliberately placed at address 0 on both HUBs and Routers and contains a bit field that you can use to tell which type is connected.)

- `vr 1 rr_scratch_reg0`

  This example goes out of the HUB into the connected Router, out port 1 of that Router into another Router, and reads one of the scratch registers on that Router.

6.10.8.2 vw Command

The *vw* command performs a CrayLink vector write operation. This command uses the following syntax:

vw VEC VADDR VAL

The *VEC* variable specifies the CrayLink vector of the HUB or Router on which you want to perform the write operation. The *VADDR* variable is the address of the register (on the specified HUB or Router) to which the command should write. The *VAL* variable specifies the data that the command should write.

For HUB operations, you can use a numerical value or the symbolic HUB register names for the *VADDR* variable. For Router operations, you can use a numerical value or the symbolic Router register names for the *VADDR* variable.

Example:

- `vw 0x512 ni_port_reset 0x80`

  This example goes through three Routers to reach a HUB and writes 0x80 (warm reset code) into the NI_PORT_RESET register of the HUB.

6.10.8.3 vx Command

The *vx* command performs a CrayLink vector exchange operation. This command uses the following syntax:

vx VEC VADDR VAL

The *VEC* variable specifies the CrayLink vector of the HUB or Router on which you want to perform the exchange operation. The *VADDR* variable is the address of the register (on the specified HUB or Router) that the command should exchange. The *VAL* variable specifies the data that the command should exchange.

For HUB operations, you can use a numerical value or the symbolic HUB register names for the *VADDR* variable. For Router operations, you can use a numerical value or the symbolic Router register names for the *VADDR* variable.

6.10.8.4 route Command

The *route* command programs a minimum subset of the router tables in the local HUB, remote HUB, and all intervening Routers. This command also sets the remote Node's Node ID (NASID). This command causes the remote Node to crash; however, it enables the local Node to access the entire memory and disk space of the remote Node. This command uses the following syntax:

route [*VEC NODE*]

The *VEC* variable specifies a vector route from the local HUB to the remote HUB. The *NODE* variable specifies the Node ID to assign to the remote HUB.

If you do not specify a *VEC* and *NODE*, the PROM consults the internal pcfg structure (see the *disc* and *pcfg* command descriptions) and lists all of the Nodes that are present in the system and the vector routes from the current Node to each of the other Nodes. Then, this command asks whether or not it should program the routes and NASIDs of the remote Nodes. If you specify that the command should program Nodes, the command uses the NASIDs found in the pcfg structure. If there is not an NASID in the pcfg structure, the command uses incrementing NASIDs, starting at one plus the current Node's NASID.

Example:

```
POD Cac Hub 000 1A> vr 0x53 ni_status_rev_id
POD Cac Hub 000 1A> route 0x53 1
POD Cac Hub 000 1A> pr n:1 ni_status_rev_id
POD Cac Hub 000 1A> ld n:1 u:0 10
POD Cac Hub 000 1A> flash 1
```

6.10.9 Controlling POD Command Execution

6.10.9.1 repeat Command

The *repeat* command executes the POD commands that you specify for the number of times that you specify. This command uses the following syntax:

repeat COUNT CMD

The *COUNT* variable specifies the number of times the commands should be executed. The *CMD* variable specifies one or more POD commands.

You can nest the repeated loops. You can stop a loop by typing Ctrl+C.

Example:

- `repeat 20 vr 0 0`

  This example repeats the *vr* 0 0 command 20 times.

6.10.9.2 loop Command

The *loop* command executes the POD commands that you specify indefinitely or until you type Ctrl+C. This command uses the following syntax:

loop CMD

The *CMD* variable specifies one or more POD commands.

Examples:

- `loop pr pi_rt_count`

  This example loops the specified *pr* command.

- `loop { sd md_led0 0x55; sleep 1; sd md_led0 0xaa; sleep 1 }`

  This example loops the specified *sd* and *sleep* commands. Notice that multiple commands are enclosed in braces { } and separated by semicolons.

6.10.9.3 while Command

The *while* command executes POD commands while a specified expression evaluates to true. This command uses the following syntax:

while (EXPR) CMD

The *EXPR* variable provides the expression that the command will evaluate. The *CMD* variable specifies the POD commands to execute while the expression evaluates to true. If *CMD* contains more than one POD command, you must separate the POD commands with semicolons and enclose the entire string of commands in curly braces { }.

Example:

- `while ((ld md_ureg8 & 0x40) != 0) px md_ureg8`

  This example prints the contents of md_ureg8 while the expression (ld md_ureg8 & 0x40) != 0 is true.

6.10.9.4 for Command

The *for* command iteratively executes a sequence of POD commands while stepping a variable through a sequence of values. This command uses the following syntax:

for (CMD; EXPR; CMD) CMD

The *for* command evaluates the first *CMD* variable once, at the start of the loop. This variable is typically used for variable initialization.

The *for* command evaluates the *EXPR* variable at the beginning of each pass through the loop; if it evaluates to 0 (or false), the *for* command exits the loop. If it evaluates to 1 (or true), the *for* command continues the loop.

The *for* command evaluates the second *CMD* variable at the end of each iteration of the loop. This variable typically specifies the increment of variables within the loop.

The last *CMD* variable contains one or more POD commands that the *for* command will execute during each iteration of the loop. If *CMD* contains more than one POD command, you must separate the POD commands with semicolons and enclose the entire string of commands in curly braces { }.

Example:

- `for (v0=0;v0<5;v0=v0+1) {px v0}`

  This example prints 0x00, 0x01, 0x02, 0x03, and 0x04.

6.10.9.5 if Command

The *if* command executes a POD command if the expression that you specify evaluates to true. This command uses the following syntax:

if (EXPR) CMD

The *EXPR* variable contains the expression that the *if* command evaluates. If the expression evaluates to true, the *if* command executes the command(s) specified with the *CMD* variable.

6.10.9.6 delay Command

The *delay* command pauses command execution for a specified number of microseconds. You cannot interrupt a *delay* command by typing Ctrl+C. This command uses the following syntax:

delay MICROSEC

The *MICROSEC* variable specifies the number of microseconds that this command should pause command execution.

6.10.9.7 sleep Command

The *sleep* command pauses command execution for a specified number of seconds. You cannot interrupt a *sleep* command by typing Ctrl+C. This command uses the following syntax:

sleep SEC

The *SEC* variable specifies the number of seconds that this command should pause command execution.

6.10.9.8 time Command

The *time* command determines and prints the length of time it takes for a POD command to execute. This command uses the following syntax:

time CMD

The *CMD* variable specifies the command(s) that the *time* command should evaluate.

Example:

- `time memtest u: 32 m 32m`

  This example returns the amount of time that it takes the specified *memtest* command to execute.

6.10.9.9 jump Command

The *jump* command causes the program counter to switch to a specific address. This command uses the following syntax:

jump ADDR [*A0* [*A1*]]

The *ADDR* variable specifies the address that this command should load into the program counter. The *A0* and *A1* variables specify values that this command should load into the R10000 a0 and a1 registers.

6.10.9.10 call Command

The *call* command performs a subroutine call to a specified address. This command uses the following syntax:

call ADDR [*A0* [*A1* [*A2* [*A3*]]]]

The *ADDR* variable specifies the address to which this command will make the subroutine call. You can pass data in the a0, a1, a2, and a3 registers by using the *A0*, *A1*, *A2*, and *A3* variables.

The return value of the subroutine is stored in the register area as v0; enter `pr v0` to access it.

6.10.9.11 reset Command

The *reset* command resets the local Node. The reset signal goes over the CrayLink Interconnect to other Nodes in the system, causing those Nodes to reset. However, the reset signal does not go across partition boundaries where the operating system in the Routers blocks the reset signal. This command uses the following syntax:

reset

6.10.10 Working With Nonmaskable Interrupts (NMIs)

6.10.10.1 nmi Command

The *nmi* command sends a nonmaskable interrupt to one or both CPUs on a specific Node. This command uses the following syntax:

nmi NODE [a | b]

The *NODE* variable indicates the Node to which this command should send the nonmaskable interrupt. Option **a** specifies that the command should send the nonmaskable interrupt to CPU A. Option **b** specifies that the command should send the nonmaskable interrupt to CPU B. If you do not specify **a** or **b**, this command sends the nonmaskable interrupt to both CPUs on the Node.

6.10.10.2 why Command

The *why* command displays the current exception and NMI status information, which includes all general-purpose registers, the program counter, several COP0 registers, and the exception vector address. (If you issue an NMI or an exception occurs while the Node is executing the PROM code, the Node saves this information and returns to the POD prompt in Dex mode.) This command uses the following syntax:

why

Example:

```
POD Dex Hub 000 1A> ld u:1      # (Misaligned memory address)
9600000000000001:
8A: *** General Exception on node 0
8A: *** EPC: 0xffffffffbfc2a9cc (load+0x48)
8A: *** Press ENTER to continue.
POD Dex Hub 000 1A> why
EPC    : 0xffffffffbfc2a394 (load+0x48)
ERREPC : 0x0000000000000000 (0x0)
CACERR : 0x0000000000000000
Status : 0x0000000024407c80
BadVA  : 0x0000000000000001 (0x1)
RA     : 0xffffffffbfc2a500 (exec_load+0x114)
SP     : 0xa800000000103660
A0     : 0x0000000000000001
Cause  : 0xffffffff80000010 (BDSLOT INT:-------- <Load Addr Err>)
BusAdr : 0x0000000000000001
Reason : 242 (Unexpected General Exception.)
POD mode was called from: 0xc00000001fc01910 (bevPanic+0xe0)
```

6.10.10.3 altregs Command

The *altregs* command specifies which set of R10000 registers should be used. This command uses the following syntax:

altregs NUM

If you set *NUM* to 1, the PROM code stops using the set of R10000 registers that were saved on PROM entry; instead the PROM code uses the set of registers saved by the kernel debugger (Symmon). If you set *NUM* to 0, the PROM switches back to the registers saved on PROM entry.

6.10.10.4 traplog Command

The *traplog* command displays entries from the kernel trap log. This command uses the following syntax:

traplog

6.10.11 Getting Help Information

6.10.11.1 help Command

The *help* command displays usage information for other POD commands. This command uses the following syntax:

help [*CMDNAME*]

You specify a single command with the *CMDNAME* variable. If you do not specify a command, the *help* command prints out a summary of usage for all commands.

6.10.11.2 ? Command

The *?* command displays usage information for other POD commands. This command uses the following syntax:

? [*CMDNAME*]

You specify a single command with the *CMDNAME* variable. If you do not specify a command, the *?* command prints out a summary of usage for all commands.

6.10.11.3 version Command

The *version* command displays the version string of the PROM that you are using. This command uses the following syntax:

version

Example:

```
POD Dex Hub 000 1A> version
IP27 PROM
SGI Version 1.4 Bring-up built 05:08:35 PM Jul 25, 1996
```

6.10.12 Manipulating the Instruction, Data, and Secondary Caches

6.10.12.1 inval Command

The *inval* command invalidates the instruction cache, the data cache, or the secondary cache. This command uses the following syntax:

inval [i] [d] [s]

The **i** option specifies instruction cache. The **d** option specifies data cache. The **s** option specifies the secondary cache. You can specify any combination of these options.

Note: This command does not write the contents of the caches back into memory.

6.10.12.2 flush Command

The *flush* command flushes the contents of the primary and secondary data caches to memory. This command uses the following syntax:

flush

6.10.12.3 dtag Command

The *dtag* command dumps tag data from one or more cache lines in the data cache. This command uses the following syntax:

dtag LINE

The *LINE* variable specifies the cache line(s) to use.

6.10.12.4 itag Command

The *itag* command dumps tag data from one or more cache lines in the instruction cache. This command uses the following syntax:

itag LINE

The *LINE* variable specifies the cache line(s) to use.

6.10.12.5 stag Command

The *stag* command dumps tag data from one or more cache lines in the secondary cache. This command uses the following syntax:

stag LINE

The *LINE* variable specifies the cache line(s) to use.

6.10.12.6 dline Command

The *dline* command dumps line data from one or more cache lines in the data cache. This command uses the following syntax:

dline LINE

The *LINE* variable specifies the cache line(s) to use.

6.10.12.7 iline Command

The *iline* command dumps line data from one or more cache lines in the instruction cache. This command uses the following syntax:

iline LINE

The *LINE* variable specifies the cache line(s) to use.

6.10.12.8 sline Command

The *sline* command dumps line data from one or more cache lines in the secondary cache. This command uses the following syntax:

sline LINE

The *LINE* variable specifies the cache line(s) to use.

6.10.12.9 adtag Command

The *adtag* command dumps tag data for an address in the data cache. This command uses the following syntax:

adtag LINE

The *LINE* variable specifies the cache line(s) to use.

6.10.12.10 aitag Command

The *aitag* command dumps tag data for an address in the instruction cache. This command uses the following syntax:

aitag LINE

The *LINE* variable specifies the cache line(s) to use.

6.10.12.11 astag Command

The *astag* command dumps tag data for an address in the secondary cache. This command uses the following syntax:

astag LINE

The *LINE* variable specifies the cache line(s) to use.

6.10.12.12 adline Command

The *adline* command dumps line data for an address in the data cache. This command uses the following syntax:

adline LINE

The *LINE* variable specifies the cache line(s) to use.

6.10.12.13 ailine Command

The *ailine* command dumps line data for an address in the instruction cache. This command uses the following syntax:

ailine LINE

The *LINE* variable specifies the cache line(s) to use.

6.10.12.14 asline Command

The *asline* command dumps line data for an address in the secondary cache. This command uses the following syntax:

asline LINE

The *LINE* variable specifies the cache line(s) to use.

6.10.13 Manipulating the TLB

6.10.13.1 tlbc Command

The *tlbc* command clears the TLB entry that you specify or the entire TLB. This command uses the following syntax:

tlbc [*INDEX*]

The *INDEX* variable specifies a specific entry in the TLB. If you do not use the *INDEX* variable, this command clears the entire TLB.

6.10.13.2 tlbr Command

The *tlbr* command reads a TLB entry or the entire TLB. This command uses the following syntax:

tlbr [*INDEX*]

The *INDEX* variable specifies a specific entry in the TLB. If you do not use the *INDEX* variable, this command reads the entire TLB.

6.10.14 Setting the Operating Mode

6.10.14.1 go Command

The *go* command sets the operating mode for POD mode. This command uses the following syntax:

go dex | unc | cac

The *go* dex format flushes the data cache, invalidates all of the caches, initializes the primary data cache as a stack and data area, and then returns to the POD prompt in dirty exclusive (Dex) mode, which does not use memory.

The *go* unc format returns to the POD prompt in uncached (Unc) mode with all POD code, data structures, and stack in uncached memory. This command tests the area of memory that PROM will use. It then copies the contents of the PROM into memory and verifies the PROM code contents in memory before jumping to the PROM code copied into memory.

The *go* cac format returns to the POD prompt in cached (Cac) mode with all POD code, data structures, and stack in cached memory. This command tests the area of memory that PROM will use. It then copies the contents of the PROM into memory and verifies the PROM code contents in memory before jumping to the PROM code copied into memory.

6.10.15 Running the HUB BIST and Router BIST

6.10.15.1 bist Command

The *bist* command runs the HUB built-in self-test (BIST) on the local Node or a remote Node. This command uses the following syntax:

bist le | ae | lr | ar [n:*NODE*]

The *NODE* variable specifies that this command should run BIST on a remote HUB; if you do not specify a *NODE*, this command runs BIST on the local HUB.

The *bist* le format executes the logic built-in self-test and then resets the Node. When the Node returns to POD prompt, you can use the *bist* lr command to obtain the results of the logic BIST.

The *bist* ae format executes the address built-in self-test and then resets the Node. When the Node returns to the POD prompt, you can use the *bist* ar command to obtain the results of the address BIST.

6.10.15.2 rbist Command

The *rbist* command runs the Router built-in self-test (BIST) on a router. This command uses the following syntax:

rbist le | ae | lr | ar *VEC*

The *VEC* variable specifies a CrayLink vector that indicates the Router on which the test will run.

The *rbist* le format executes the logic built-in self-test and then resets the Router. When the Router is reset, you can use the *rbist* lr command to obtain the results of the logic BIST.

The *rbist* ae format executes the address built-in self-test and then resets the Router. When the Router is reset, you can use the *rbist* ar command to obtain the results of the address BIST.

6.10.16 Enabling and Disabling Hardware

6.10.16.1 disable Command

The *disable* command prevents a CPU from participating in the boot process and prevents a CPU from making HUB requests. This command uses the following syntax:

disable n:*NODE* [*SLICE*]

The *NODE* variable specifies the Node on which the CPU is located. The *SLICE* variable specifies which CPU (A or B) should be disabled. If you do not specify a *SLICE*, this command disables both CPUs.

To disable the CPUs, this command sets the environment variables DisableA and DisableB in the PROM log for the specified Node.

6.10.16.2 enable Command

The *enable* command removes the DisableA and/or DisableB environment variables from the PROM log of the Node that you specify. This enables the CPUs to participate in the boot sequence and to make HUB requests. This command uses the following syntax:

enable n:*NODE* [*SLICE*]

The *NODE* variable specifies the Node on which the CPUs are located. The *SLICE* variable specifies the CPU that the command should enable (A or B). If you do not specify a *SLICE*, this command enables both CPUs.

6.10.16.3 tdisable Command

The *tdisable* command temporarily prevents a CPU from participating in the boot sequence and making HUB requests (until the next time you reset the system). This command uses the following syntax:

tdisable n:*NODE* [*SLICE*]

The variable specifies the Node on which the CPU is located. The SLICE variable specifies which CPU (A or B) should be disabled. If you do not specify a SLICE, this command disables both CPUs.

The CPUs are reenabled when you reset the Node.

6.10.17 Testing Memory

6.10.17.1 dirinit Command

The *dirinit* command initializes a range of directory memory to the "Unowned" state. This command uses the following syntax:

dirinit START LEN

The *START* variable specifies the ordinary memory address that this command masks for the lower 40 bits and translates them to a corresponding directory address. The *LEN* variable specifies the length (in bytes) of the range of directory memory. *START* and *LEN* must be multiples of 4,096 (0x1000).

6.10.17.2 meminit Command

The *meminit* command initializes a range of memory to zeroes. This command uses the following syntax:

meminit START LEN

The *START* variable specifies the starting memory address (either cached or uncached) of the range of memory to initialize. The *LEN* variable specifies the length of the region to initialize (in bytes). *START* and *LEN* must be multiples of 4,096 (0x1000).

6.10.17.3 dirtest Command

The *dirtest* command tests a range of directory memory. This command uses the following syntax:

dirtest START LEN

The *START* variable specifies a memory address that this command uses by masking off the lower 40 bits and translating the value into a corresponding directory address. The *LEN* command specifies the size of the area to test (in bytes). *START* and *LEN* must be multiples of 4,096 (0x1000).

Note: Turn off ECC checking (enter `ecc off`) before you test directory memory.

The directory memory test uses the following algorithm:

1. Clear the error registers.
2. Test the store and load functions. For each doubleword address in the memory range that is being tested, perform the following actions:
   - Perform a store operation with a data pattern that contains all 0x5 values, immediately perform load operations to get the data back, and compare the loaded data with the stored data.
   - Perform a store operation with a data pattern that contains all 0xa values, immediately perform load operations to get the data back, and compare the loaded data with the stored data.

   This test may find problems with RAM cells that take too long to change states.
3. Test memory with alternating 0x5 and 0xa data patterns. (Write the data patterns, read the data back, and compare the data that was read back with the data that was written.)
4. Test memory with alternating 0xa and 0x5 data patterns. (Write the data patterns, read the data back, and compare the data that was read back with the data that was written.)
5. Test memory with alternating 0xc3c3c3c3c3c3c3c3 and 0x3c3c3c3c3c3c3c3c doubleword patterns. (Write the data patterns, read the data back, and compare the data that was read back with the data that was written.)

6. Test memory with alternating 0x3c3c3c3c3c3c3c3c and 0xc3c3c3c3c3c3c3c3 doubleword patterns. (Write the data patterns, read the data back, and compare the data that was read back with the data that was written)
7. Test memory with random data patterns. (Write a random data pattern, read the data back, and compare the data that was read back with the data that was written.)
8. Test memory with address data patterns. (Write doublewords with the address of the location being written, read the data back, and compare the data that was read back with the data that was written.)

Memory is left filled with a data pattern of incrementing doublewords. The *maxerr* command specifies the number of errors that the test should display for each phase before it stops the current phase and continues with the next phase of the test.

6.10.17.4 memtest Command

The *memtest* command tests a range of memory. This command uses the following syntax:

memtest START LEN

The *START* variable specifies the starting address of the memory range; this address can be in cached or uncached memory. The *LEN* command specifies the size of the range of memory to test (in bytes). *START* and *LEN* must be multiples of 4,096 (0x1000).

The memory test uses the following algorithm:

1. Clear the error registers.
2. Test the store and load functions. For each doubleword address in the memory range that is being tested, perform the following actions:
   - Perform a store operation with a data pattern that contains all 0x5 values, immediately perform load operations to get the data back, and compare the loaded data with the stored data.
   - Perform a store operation with a data pattern that contains all 0xa values, immediately perform load operations to get the data back, and compare the loaded data with the stored data.

   This test may find problems with RAM cells that take too long to change states.
3. Test memory with alternating 0x5 and 0xa data patterns. (Write the data patterns, read the data back, and compare the data that was read back with the data that was written.)
4. Test memory with alternating 0xa and 0x5 data patterns. (Write the data patterns, read the data back, and compare the data that was read back with the data that was written.)
5. Test memory with alternating 0xc3c3c3c3c3c3c3c3 and 0x3c3c3c3c3c3c3c3c doubleword patterns. (Write the data patterns, read the data back, and compare the data that was read back with the data that was written.)
6. Test memory with alternating 0x3c3c3c3c3c3c3c3c and 0xc3c3c3c3c3c3c3c3 doubleword patterns. (Write the data patterns, read the data back, and compare the data that was read back with the data that was written)

7. Test memory with random data patterns. (Write a random data pattern, read the data back, and compare the data that was read back with the data that was written.)
8. Test memory with address data patterns. (Write doublewords with the address of the location being written, read the data back, and compare the data that was read back with the data that was written.)

Memory is left filled with a data pattern of incrementing doublewords. The *maxerr* command specifies the number of errors that the test should display for each phase before it stops the current phase and continues with the next phase of the test.

Examples:

- ```
  memtest u:32m 1m
  memtest 0x9600000002000000 0x100000
  ```

  These examples are equivalent. These examples test 1 MB of uncached memory space, starting at an offset of 32 MB.

- ```
  memtest c:32m 32m
  ```

  This example tests the second 32 MB of cached memory.

- ```
  memtest u:b:3 0 64m
  ```

  This example tests 64 MB of uncached memory in bank 3.

- ```
  memtest n:7 b:3 c: 1m 63m
  memtest (n:7 b:3 c: 1m) 63m
  memtest 0xa800000760100000 0x3f00000
  ```

  These examples are equivalent. These examples test 63 MB of cached memory on Node 7, beginning at the 1 MB position of bank 3.

6.10.17.5 santest Command

The *santest* command performs a sanity test on a memory address. The sanity test verifies that the address can be written to and read from; the test performs these operations with full error register monitoring and exception catching. This command uses the following syntax:

santest ADDR

The *ADDR* variable specifies the memory address to test.

The sanity test uses the following algorithm:

1. Clear the error registers and verify that they are clear.
2. Verify that no bits in the corresponding directory memory entry LO/HI are stuck at 1 or 0 (with error register checking).
3. Initialize the corresponding directory memory entry LO/HI to the "Unowned" state.
4. Verify that no bits in the corresponding directory memory entry LO/HI are stuck at 1 or 0 (with error register checking).

6.10.17.6 maxerr Command

The *maxerr* command sets the maximum number of errors a memory test will report before it aborts the failing portion of the test and moves to the next phase of the test. This command uses the following syntax:

maxerr COUNT

The *COUNT* variable specifies the number of errors; the default is 32 errors.

6.10.17.7 qual Command

The *qual* command turns the data quality checking features of an R10000 processor on or off by modifying the contents of the PI_SYSAD_ERRCHK_EN register. This command uses the following syntax:

qual [on | off]

The **on** option turns the data quality checking features on; the **off** option turns the data quality checking features off. If you do not specify **on** or **off**, this command toggles the state of the data quality checking features.

6.10.17.8 ecc Command

The *ecc* command turns ECC checking on or off. This command uses the following syntax:

ecc [on | off]

The **on** option turns ECC checking on by setting the IGNORE_ECC bit to off. When you enable ECC checking, exceptions occur when the ECC for directory memory or regular memory does not match the data stored in directory memory or regular memory.

The **off** option turns ECC checking off by setting the IGNORE_ECC bit to on. When you disable ECC checking, exceptions do not occur when the ECC for directory memory or regular memory does not match the data stored in directory memory or regular memory. This option does not stop the HUB from correcting errors.

6.10.17.9 error Command

The *error* command checks the MD section of the HUB for any error bits that are set, decodes the bits, and displays the decoded information. This command uses the following syntax:

error

Use this command when the POD prompt includes `MDerr`, which indicates that there are memory or directory errors pending in the HUB.

6.10.17.10 clear Command

The *clear* command clears any pending error bits in the HUB MD section. This command uses the following syntax:

clear

6.10.17.11 im Command

The *im* command sets the R10000 interrupt mask field in the COP0 Status register. The interrupt mask specifies which errors can cause an interrupt. This command uses the following syntax:

im [*BYTE*]

The *BYTE* variable contains a bitmask that represents the interrupt mask.

6.10.18 Manipulating the Memory Region

6.10.18.1 memset Command

The *memset* command fills a range of memory with a byte value. This command uses the following syntax:

memset DST BYTE LEN

The *DST* variable specifies the starting address for the range of memory to fill. The *BYTE* variable specifies the byte value to use. The *LEN* variable specifies the length of the memory range to fill (in bytes).

6.10.18.2 memcpy Command

The *memcpy* command copies the contents of a range of memory from one location in memory to another location in memory. This command uses the following syntax:

memcpy DST SRC LEN

The *DST* variable specifies the first address of the range of memory that will receive the copied data. The *SRC* variable specifies the first address of the range of memory from which this command will copy the data. The *LEN* variable specifies the amount of memory that this command will copy (in bytes).

6.10.18.3 memcmp Command

The *memcmp* command compares the data in one range of memory with the data in another range of memory; then, it prints a message that indicates whether or not the two ranges of memory contain the same data. This command uses the following syntax:

memcmp DST SRC LEN

The *DST* variable specifies the starting address of one of the ranges of memory. The *SRC* variable specifies the starting address of the other range of memory. The *LEN* variable specifies the amount of memory to compare (in bytes).

6.10.18.4 memsum Command

The *memsum* command adds up the data contained in a range of memory (by adding the individual bytes together). This command uses the following syntax:

memsum SRC LEN

The *SRC* variable specifies the starting address of the memory range to use. The *LEN* variable specifies the amount of memory to use (in bytes).

You can use this command to compare the ranges of memory on different systems by comparing the byte sums for the memory ranges.

6.10.18.5 reconf Command

The *reconf* command initializes the memory/directory (MD) section of the HUB. (It runs a test to determine the true amount of physical memory on the local Node and then uses this value to configure the HUB.) This command uses the following syntax:

reconf

6.10.19 Manipulating the Directory Region

6.10.19.1 scandir Command

The *scandir* command scans a range of memory and displays the directory state of each memory location. (To reduce the amount of output, this command prints only the start of the region where the directory state changes.) This command uses the following syntax:

scandir ADDR [*LEN*]

The *ADDR* variable specifies a cache-line aligned memory address; this command masks the *ADDR* value to get the lower 40 bits and translates the value into a corresponding directory address. The *LEN* variable specifies the size of the memory area that this command should test (in bytes).

6.10.19.2 dirstate Command

The *dirstate* command scans a range of addresses for the directory state that you specify. This command uses the following syntax:

dirstate [*STATE* [*BASE* [*LEN*]]]

The *STATE* variable specifies the state for which this command should search. The *BASE* variable specifies the starting address. The *LEN* variable specifies the amount of memory to check.

6.10.20 Running the Local Link Connection Test

6.10.20.1 chklink Command

The *chklink* command checks the local link by running the local link connection test. (It attempts to communicate with the Router or HUB attached to the local link and measures the error rate.) This command uses the following syntax:

chklink

Refer to Section 4.6.1, "Local Link Connection Test," for more information about the local link connection test.

6.10.21 Running the HUB Send Data Error Test

6.10.21.1 hubsde Command

The *hubsde* command runs the HUB send data error test, which verifies that the HUB or Router attached to the local HUB can detect errors. This command uses the following syntax:

hubsde

Refer to Section 4.6.2, "HUB Send Data Error Test," for more information about the HUB send data error test.

6.10.22 Running the Router Send Data Error Test

6.10.22.1 rtrsde Command

The *rtrsde* command runs the Router send data error test, which verifies that the local HUB can detect link errors. This command uses the following syntax:

rtrsde

Refer to Section 4.6.4, "Router Send Data Error Test," for more information about the Router send data error test.

6.10.23 Testing the XBOW

6.10.23.1 dgxbow Command

The *dgxbow* command runs the XBOW test, which checks for valid reset values in the XBOW registers and tests the XBOW interrupt hardware. This command uses the following syntax:

dgxbow [mn | mh | mm] [n*NODE*]

Refer to Section 4.7.2, "How to Run This Test," for descriptions of the command line options.

6.10.24 Testing the Bridge

6.10.24.1 dgbrdg Command

The *dgbrdg* command runs the Bridge test, which checks for valid reset values in the Bridge registers and tests the Bridge interrupt hardware. This command uses the following syntax:

dgbrdg [mn | mh | mm] [n*NODE*] [s*SLOT*]

Refer to Section 4.8.2, "How to Run This Test," for descriptions of the command line options.

6.10.25 Testing the PCI Hardware

6.10.25.1 dgconf Command

The *dgconf* command runs the PCICONFIG test, which verifies the PCI configuration space on the BaseIO board. This command uses the following syntax:

dgconf [mn | mh | mm] [n*NODE*] [s*SLOT*]

Refer to Section 4.9.1.2, "How to Run This Test," for descriptions of the command line options.

6.10.25.2 dgpci Command

The *dgpci* command runs the PCI bus test, which checks the PCI bus with walking data tests. This command uses the following syntax:

dgpci [mn | mh | mm] [n*NODE*] [s*SLOT*] [p*PCINUM*]

Refer to Section 4.9.2.2, "How to Run This Test," for descriptions of the command line options.

6.10.26 Testing the Serial Ports

6.10.26.1 dgspio Command

The *dgspio* command runs the serial programmed I/O (PIO) test, which uses the internal loopback features of the SuperIO chip to test the serial PIO functionality of the BaseIO board. This command uses the following syntax:

dgspio [mn | mh | mm | mx] [n*NODE*] [s*SLOT*] [p*PCINUM*]

Refer to Section 4.10.1.2, "How to Run This Test," for descriptions of the command line options.

6.10.26.2 dgsdma Command

The *dgsdma* command runs the serial DMA test, which verifies that serial DMA can be performed. This command uses the following syntax:

dgsdma [mn | mh | mm | mx] [n*NODE*] [s*SLOT*] [p*PCINUM*]

Refer to Section 4.10.2.2, "How to Run This Test," for descriptions of the command line options.

6.10.27 Accessing the Slave (Launch Loop)

6.10.27.1 slave Command

The *slave* commands causes a CPU to jump into the IP27 PROM slave launch loop, where the CPU waits to be assigned a task and to receive an interrupt from another CPU. This command uses the following syntax:

slave

6.10.27.2 cpu Command

The *cpu* command launches the CPU that you specify from the slave loop into POD mode and places the CPU from which you called this command into the slave launch loop; this process exchanges the active CPU. This command uses the following syntax:

cpu [[n:*NODE*] a | b]

The *NODE* variable specifies the Node board that contains the CPU to launch from the slave loop. Option **a** indicates CPU A on the Node board. Option **b** indicates CPU B on the Node board.

6.10.28 Accessing the BaseIO PROM

6.10.28.1 segs Command

The *segs* command reads the BaseIO PROM and prints information about it (the segments it contains, the size of the segments, and so on). This command uses the following syntax:

segs [*FLAG*]

The *FLAG* variable specifies which version of the BaseIO PROM to use. If you specify 0, this command uses the copy of the BaseIO PROM that is stored in the IP27 PROM, instead of the actual BaseIO PROM. If you specify 1, this command uses the actual BaseIO PROM. If you do not specify a *FLAG* value, this command uses the actual BaseIO PROM.

6.10.28.2 exec Command

The *exec* command reads the BaseIO PROM and then jumps to the segment that you specify. This command uses the following syntax:

exec [*SEGNAME* [*FLAG*]]

The *SEGNAME* variable specifies the segment to which this command should jump; if you do not specify a *SEGNAME* value, this command defaults to the BaseIO PROM. The *FLAG* variable specifies which version of the BaseIO PROM to use.

- If you specify 0, this command uses the copy of the BaseIO PROM that is stored in the IP27 PROM, instead of the actual BaseIO PROM.
- If you specify 1, this command uses the actual BaseIO PROM.
- If you specify 2, this command loads the IP27 PROM version of the BaseIO PROM but does not jump to it; the command only displays the entry point address and returns to POD mode.
- If you specify 3, this command loads the actual BaseIO PROM but does not jump to it.

If you do not specify a *FLAG* value, this command uses the actual BaseIO PROM.

6.10.29 Choosing a Console

6.10.29.1 ioc3 Command

The *ioc3* command moves the POD prompt to the console connected to the IOC3. Before moving the prompt, this command verifies that the console is accessible. This command uses the following syntax:

ioc3

6.10.29.2 junk Command

The *junk* command moves the POD prompt to the console connected to the Junk UART. Before moving the prompt, this command verifies that the console is accessible. This command uses the following syntax:

junk

This command is normally used only in the bring-up environment.

6.10.29.3 msc Command

The *msc* command moves the POD prompt to the console connected to the module system controller (MSC). Before moving the prompt, this command verifies that the console is accessible. This command uses the following syntax:

msc

Note: This command was previously *elsc*.

6.10.29.4 talk Command

The *talk* command moves the POD prompt to a netuart. This command uses the following syntax:

talk [n:*NODE* a | b]

The *NODE* variable specifies the Node board where the netuart is located. Option **a** specifies CPU A, and option **b** specifies CPU B. If you do not specify the *NODE* variable and CPU option, this command attempts to emulate a terminal and contact another Node that is using a virtual netuart.

6.10.30 Accessing the CrayLink Interconnect

6.10.30.1 disc Command

The *disc* command runs the CrayLink Interconnect discovery algorithm. This command uses the following syntax:

disc

This command runs only in Cac mode. It builds the internal promcfg data structure, which contains a graph of all connections for HUBs and Routers in the system. Use the *pcfg* command to view the results of this command.

6.10.30.2 pcfg Command

The *pcfg* command displays the contents of the promcfg data structure, which is created by the *disc* command. This command uses the following syntax:

pcfg [n:*NODE*]

The *NODE* variable specifies the Node.

Use this command only in Cac mode and only after you have run the *disc* command.

6.10.30.3 node Command

The *node* command sets the NASID value (Node ID) for a Node. This command uses the following syntax:

node [[*VEC*] *ID*]

The *VEC* variable specifies a remote Node that this command will access through a CrayLink vector write operation. The *ID* variable specifies the value to which the NASID will be set. If you do not specify a *VEC* value, this command sets the NASID on the local Node, which causes the local Node to crash; however, the local Node returns to POD mode after the crash.

6.10.30.4 rtab Command

The *rtab* command displays the routing tables for the HUB that you specify, which include routing tables in the HUB and routing tables in the Router. (The routing tables in the HUB contain 16 local table entries and 32 metatable entries, each containing a single 4-bit direction. The routing tables in the Router contain 16 local table entries and 32 metatable entries, each containing 24 bits of routing information that are organized as six 4-bit directions.) This command uses the following syntax:

rtab [*VEC*]

The *VEC* variable specifies the vector of the HUB to use; if you omit the *VEC* variable, this command uses the local HUB.

6.10.30.5 rstat Command

The *rstat* command displays the link error and packet histogram information for the Router that you specify. This command uses the following syntax:

rstat VEC

The *VEC* variable specifies the vector of the Router.

6.10.31 Debugging I/O

6.10.31.1 crb Command

The *crb* command dumps the CRB registers from the I/O section of the HUB chip in a hexadecimal format. This command uses the following syntax:

crb [n:*NODE*]

The *NODE* variable specifies the Node that this command should use.

6.10.31.2 crbx Command

The *crbx* command dumps the CRB registers from the I/O section of the HUB chip, decodes the dumped data, and displays the results in a 132-column table. This command uses the following syntax:

crbx [n:*NODE*]

The *NODE* variable specifies the Node that this command should use.

6.10.32 Flashing the PROM

6.10.32.1 flash Command

The *flash* command flashes a remote Node's IP27 Flash PROM with a copy of the local Node's IP27 Flash PROM. This command uses the following syntax:

flash NODE [...]

The *NODE* variable specifies the location of the remote Node that this command should flash. You may specify more than one remote Node at a time.

The *flash* command accesses the remote Node through memory space, so you can use this command only if you have programmed the CrayLink routing tables and assigned the remote Node ID (you can use the *route* command to do this).

You can use the flash command in Dex mode; however, this is extremely slow. If you are using Dex mode (for example, through an NMI), try to enter Cac mode (type `go cac`) before flashing the remote PROM(s).

Example:

```
POD Dex Hub 000 1A> go cac
POD Cac Hub 000 1A> route 0x53 1
POD Cac Hub 000 1A> flash 1
```

6.10.33 Reading the NIC Values

6.10.33.1 nic Command

The *nic* command displays the NIC value of the local Node or the Node that you specify; it displays the Node board's unique ID number. This command uses the following syntax:

nic [n:*NODE*]

The *NODE* variable specifies the Node from which this command should read the NIC. If you do not specify a *NODE*, this command reads the NIC from the local Node.

Example:

```
POD Dex Hub 000 1A> nic
0x000000000001fcba
```

6.10.33.2 rnic Command

The *rnic* command reads the NIC value of the Router that you specify. This command uses the following syntax:

rnic [*VEC*]

The *VEC* variable specifies a CrayLink vector to the Router that this command should use. If you do not specify a *VEC*, this command defaults to 0.

Example:

```
POD Dex Hub 000 1A> rnic 1
0x0000000000020300
```

6.10.33.3 modnic Command

The *modnic* command displays the NIC value of the current module; it displays the module (midplane) serial number. This command uses the following syntax:

modnic

6.10.34 Manipulating the System Controller

6.10.34.1 scw Command

The *scw* command writes bytes of data to the module system controller (MSC) nonvolatile RAM (NVRAM). This command uses the following syntax:

scw ADDR [VAL [COUNT]]

The *ADDR* variable specifies the address in MSC NVRAM to which the command should write the data; the address must be between 0 and 2047, and the address cannot be between 1792 and 2047, which is memory that is reserved for MSC use. The *VAL* variable specifies the value that the command should write to the address. The *COUNT* variable specifies the number of times the value should be written to memory.

If you omit the *VAL* and *COUNT* variables, this command enters an interactive mode. The current contents of the specified address are displayed on the screen. To modify the contents, type a hexadecimal value (without the leading 0x), and press Enter. The contents of the next memory location are shown. To modify the contents, type a hexadecimal value (without the leading 0x), and press Enter. To go to the next address without changing the data, press Enter. To exit interactive mode, type period (.) and then press Enter.

6.10.34.2 scr Command

The *scr* command reads byte data from the MSC NVRAM. This command uses the following syntax:

scr ADDR [COUNT]

The *ADDR* variable specifies the address in MSC NVRAM from which the command should read the data; the address must be between 0 and 2047, and the address cannot be between 1792 and 2047, which is memory that is reserved for MSC use. The *COUNT* variable specifies the number of bytes the command should read.

6.10.34.3 sc Command

The *sc* command sends a command to the MSC. This command uses the following syntax:

sc ["COMMAND"]

The *COMMAND* variable specifies the command that this command should send to the MSC. Table 6-6 describes the commands that are available.

Table 6-6 MSC Commands

Command	Description
clr	Resets all MSC options (including echo mode, no heartbeats, and so on) to their power-on defaults
dbg	Displays the virtual and physical debug DIP switch bytes
dbg V P	Sets the virtual debug DIP switch setting byte to *V* and the physical debug DIP switch byte to *P*
dsp M	Displays message *M* on the eight-character alphanumeric display The message *M* can contain up to eight ASCII characters (other than NUL). For example, `dsp testing` overwrites the first seven characters with "testing" and does not change the eighth character.
dsc N C	Changes one character on the eight-character alphanumeric display The variable *N* specifies which character (from 0 to 7) on the display to change on the display. The variable *C* specifies the new value for the character.
echo 0	Turns off echoing mode Commands are not echoed as they are entered.
echo 1	Turns on echoing mode Commands are echoed as they are entered.
ech	Toggles echoing mode
fan	Returns the speed of the fans (n for normal or h for high)
fan n	Sets the fans to normal speed
fan h	Sets the fans to high speed
key	Returns the status of the keyswitch (off, on, or diag)
mod	Displays the number of the module that includes the MSC
mod xx	Sets the number of the module that contains the MSC to *xx*, where *xx* is a hexadecimal number for 01 to ff)
nmi	Sends an NMI to all Node boards that are in the same module as the MSC
pas xxxx	Places the MSC in supervisor mode if you specify the proper password with the *xxxx* value. The MSC must be in supervisor mode for you to use restricted commands. The MSC is automatically in supervisor mode if the keyswitch is in the diag position.
pas s *xxxx*	Sets the password to the value that you specify. The password is stored in the NVRAM. This is a restricted command.
pwr	Returns the power status of the system (u = the system is powered on (up); d = the system is powered off (down))
pwr u	Powers the system on (up)

Table 6-6 (continued) MSC Commands

Command	Description
pwr d	Powers the system off (down)
pwr d *N*	Waits *N* seconds and then powers the system off The variable *N* must be a hexadecimal value from 5 to 258 (5 to 600 decimal).
pwr c *N*	Powers the system off, waits *N* seconds, and then powers the system back on The variable *N* must be a hexadecimal value from 5 to 258 (5 to 600 decimal).
rst	Sends a hardware reset to all Node boards that are in the same Node as the MSC
rsw	Returns the current Debug Switch settings as a hexadecimal byte
sel cpu	Specifies the CPU that will receive input from the ACP (anything that you type that is not an escaped command is sent through the CPU specified with the *cpu* variable) For example, `sel 2a` selects CPU A in slot number 2.
sel	Displays which CPU is currently selected to receive ACP or `none` if no CPU is selected
sel auto	Specifies that the last CPU that output data should receive input from the ACP (this is the power-on default)
sel none	Causes no CPU to be selected to receive the ACP input
see cpu	Causes output from all CPUs except the CPU specified by *cpu* to be discarded Usually, the output from all CPUs is displayed, one line after another. For example, `see 2a` selects output from CPU A in slot number 2.
see	Displays the CPU for which output is currently being shown (or `all` if output from all CPUs is being shown)
see all	Specifies that output from all CPUs should be shown (this is the power-on default)
tmp	Returns the current temperature state of the system (o = overtemperature condition; h = dangerously high temperature; n = normal temperature)
ver	Displays the version number of the MSC firmware

6.10.34.4 dips Command

The *dips* command displays the debug DIP switch setting values, which are exactly as the settings appear on the MSC. This command uses the following syntax:

dips

6.10.34.5 dbg Command

The *dbg* command displays or changes the virtual and physical debug DIP switch settings. This command uses the following syntax:

dbg [*VAL VAL*]

The first *VAL* variable specifies the value to which the virtual debug DIP switch settings should be set. The second *VAL* variable specifies the value to which the physical DIP switch settings should be set. This command uses decimal numbers by default; if you want to use a hexadecimal number, you must use the 0x prefix.

If you do not specify the *VAL* variables, this command displays the current values of the virtual and physical debug DIP switch settings.

Example:

```
POD Dex Hub 000 1A> dbg 0 0x18
```

This example turns off all of the virtual debug DIP switch settings because the first VAL variable is set to 0. This example turns on physical debug DIP switches 4 and 5 (because 0x18 has the fourth and fifth bits set).

Note: The system performs an exclusive OR (XOR) operation on the physical settings and the actual hardware settings to determine the actual physical settings. You can use this feature to remotely override the hardware debug DIP switch settings.

6.10.34.6 pas Command

The *pas* command displays or changes the MSC password directly from POD mode. This command uses the following syntax:

pas [*"STRING"*]

The optional *STRING* variable sets a new MSC password and must be four characters. If you do not specify a *STRING*, this command displays the current MSC password. The default password is "none."

6.10.34.7 module Command

The *module* command displays or changes the module number of a module directly from POD mode. This command uses the following syntax:

module [*VAL*]

The optional *VAL* variable sets the module number to the value that you specify. If you do not specify a *VAL*, this command displays the current module number for the module. *VAL* must be a number from 0 to 255. This command uses decimal numbers by default; if you want to use a hexadecimal number, you must use the 0x prefix.

6.10.35 Disassembling Memory

6.10.35.1 dis Command

The *dis* command disassembles a region of memory into R10000 instructions. This command uses the following syntax:

dis ADDR [*COUNT*]

The *ADDR* variable specifies the starting address of the region of memory to disassemble. The *COUNT* variable specifies the number of instructions to disassemble (not including branch delay slots). If you do not specify a *COUNT*, this commands stops disassembling instructions at the end of the function containing the start address, if applicable; otherwise, this command disassembles 12 instructions.

6.10.35.2 kern_sym Command

The *kern_sym* command toggles the set of symbols that the PROM code uses to translate from addresses to symbols and from symbols to addresses. The two sets of symbols are the IP27 PROM symbols and the kernel symbols (which are available only if you have loaded a kernel since the system boot). This command uses the following syntax:

kern_sym

6.10.36 Dumping Error Information

6.10.36.1 dumpspool Command

The *dumpspool* command dumps the PI error spool from the CPU that you specify. This command uses the following syntax:

dumpspool [n:*NODE SLICE*]

The *NODE* variable specifies the Node that contains the CPU. The *SLICE* variable specifies the CPU (CPU A or CPU B) that this command should use.

6.10.36.2 error_dump Command

The *error_dump* command dumps the error state of the local HUB. This command decodes and displays detailed information about all of the error bits in the HUB. This command uses the following syntax:

error_dump

6.10.36.3 edump_bri Command

The *edump_bri* command dumps Bridge error information to memory. This command uses the following syntax:

edump_bri [n:*NODE*]

The *NODE* variable specifies the Node that contains the Bridge from which you want to dump the error information.

6.10.37 Manipulating the PROM Log

The IP27 PROM includes 1 MB of data storage, which is organized as 16 pages that contain 64 KB of data storage. The first 14 pages store the IP27 PROM code; the last 2 pages contain the PROM log. You can independently flash (erase) the PROM sectors, so that new code can be flashed into the PROM without erasing the PROM log; when you initialize the PROM log with the *initlog* command, the PROM code remains intact.

The PROM log stores three types of data: PROM environment variables, PROM environment lists, and error log messages.

Table 6-7 describes the available PROM environment variables.

Table 6-7 PROM Environment Variables

Variable	Description[a]
DisableA	Specifies whether or not CPU A should be used. If you define this variable with the *setenv* command, it prevents CPU A from participating in the boot process and from making HUB requests. When you modify this variable, provide a string that specifies the reason. Enter one of the following commands to modify this variable: **setenv DisableA** *STRING* **unsetenv DisableA** *STRING*
DisableB	Specifies whether or not CPU B should be used. If you define this variable with the *setenv* command, it prevents CPU B from participating in the boot process and from making HUB requests. When you modify this variable, provide a string that specifies the reason. Enter one of the following commands to modify this variable: **setenv DisableB** *STRING* **unsetenv DisableB** *STRING*
DisableNode	Specifies whether or not the Node should be used. If you define this variable with the *setenv* command, the system functions as if the Node is physically not part of the configuration. When you modify this variable, provide a string that specifies the reason. Enter one of the following commands to modify this variable: **setenv DisableNode** *STRING* **unsetenv DisableNode** *STRING*

Table 6-7 (continued) PROM Environment Variables

Variable	Description[a]
DisableMem	Specifies memory banks that should not be used. When you modify this variable, provide a string that specifies the reason. Enter one of the following commands to modify this variable: **setenv DisableMem** *NUM* **[,** *NUM*, **...]** *STRING* **unsetenv DisableMem** *STRING*
DisableBTE	Specifies that BTE should not be used. Set this variable if the BTE test fails. When you modify this variable, provide a string that specifies the reason. Enter one of the following commands to modify this variable: **setenv DisableBTE** *STRING* **unsetenv DisableBTE** *STRING*
DisableIO	Specifies that I/O probing should be disabled. When you modify this variable, provide a string that specifies the reason. Enter one of the following commands to modify this variable: **setenv DisableIO** *STRING* **unsetenv DisableIO** *STRING*
SwitchOn	Specifies debug DIP switch settings that should be forced to On. The number that you specify is a bitmask, where bit 0 corresponds to debug DIP switch 0, bit 1 corresponds to debug DIP switch 1, and so on. Enter the following command to set this variable: **setenv SwitchOn** *NUM*
SwitchOff	Specifies debug DIP switch settings that should be forced to Off. The number that you specify is a bitmask, where bit 0 corresponds to debug DIP switch 0, bit 1 corresponds to debug DIP switch 1, and so on. Enter the following command to set this variable: **setenv SwitchOff** *NUM*
BootCount	Specifies the value of the counter that increments when IRIX is booted. Enter the following command to set this variable: **setenv BootCount** *NUM*
DiagLevel	Specifies the mode in which the diagnostics will run. Enter **setenv DiagLevel normal** to always use normal mode. Enter **setenv DiagLevel heavy** to always use heavy mode. Enter **setenv DiagLevel mfg** to always use manufacturing mode. If the DiagLevel variable is not set, the power-on diagnostics run in heavy mode; and if the diagnostics pass in heavy mode, the DiagLevel is set to normal mode.
SwapBank	Specifies a bank that will be swapped with bank 0. Enter the following command to set this variable: **setenv SwapBank** *NUM*
BootLevel	Specifies when the boot process should stop. Enter **setenv BootLevel dex** to have the boot process stop when the process reaches Dex POD mode. Enter **setenv BootLevel pod** to have the boot process stop when the process reaches memory POD mode.

Table 6-7 (continued) PROM Environment Variables

Variable	Description[a]
LoadSegment	Specifies the segment to load. Enter the following command to set this variable: `setenv LoadSegment` *STRING*
OverrideNIC	Specifies that the PROM should override the NIC and use the value that you specify. Enter the following command to set this variable: `setenv OverrideNIC` *NUM*
GlobalMaster	Specifies that the Node should be the global master. Enter the following command to set this variable: `setenv GlobalMaster` *STRING*
LastSlot	Specifies the last slot that contains a board. Enter the following command to set this variable: `setenv LastSlot` *NUM*
LastModule	Specifies the last module number that is plugged into the system. Enter the following command to set this variable: `setenv LastModule` *NUM*
LedDelay	Specifies the delay between the boot LED updates. Enter the following command to set this variable: `setenv LedDelay` *NUM*

a. The *STRING* variables shown in the "Description" column can contain up to 127 ASCII characters.

PROM environment lists store multiple value strings under the same key string. Currently, the only environment list is the DisableRouter list, which contains a NIC value; if you set DisableRouter with a *setenv* command, the Router with the NIC value that you specify is disabled.

Error log messages indicate errors that occur in the PROM. To avoid filling up the PROM log, the PROM code does not generate any more error log messages than necessary. There are three types of messages: `Fatal` messages, which indicate fatal errors; `Error` messages, which indicate error message; and `Info` messages, which provide important information.

The following commands let you manipulate the PROM log and the PROM environment variables.

6.10.37.1 initlog Command

The *initlog* command initializes the PROM log by clearing all variable entries and all log entries. This command uses the following syntax:

initlog [n: *NODE*]

The *NODE* variable specifies that the command should initialize the PROM log for a remote Node. If you do not specify a *NODE*, this command initializes the PROM log for the local Node.

Warning: **Executing *initlog* on a remote PROM crashes the remote Node if the Node is running in Dex mode.**

You cannot use this command on the local PROM in Dex mode because the PROM cannot be programmed while you are running the code from it.

6.10.37.2 initalllogs Command

The *initalllogs* command initializes all PROM logs for all modules in the system. (It clears all variable entries and all log entries from each PROM log in the system.) This command uses the following syntax:

initalllogs

Note: Run this command only after the CrayLink discovery is completed.

6.10.37.3 log Command

The *log* command displays entries from the PROM log of the Node that you specify. It displays information from the beginning of the log to the end of the log (least recent information to most recent information). This command uses the following syntax:

log [n:*NODE*] [*SKIP* [*COUNT*]]

If you specify a value with the *SKIP* variable, this command skips the first *SKIP* number of values, which lets you start at a certain point in the log. The *COUNT* variable specifies the number of entries that the command should show (the default is 24 entries).

6.10.37.4 rlog Command

The *rlog* command displays entries from the PROM log of the Node that you specify (in reverse order). It displays information from the end of the log to the beginning of the log (most recent information to least recent information). This command uses the following syntax:

rlog [n:*NODE*] [*SKIP* [*COUNT*]]

If you specify a value with the *SKIP* variable, this command skips the first *SKIP* number of values, which lets you start at a certain point in the log. The *COUNT* variable specifies the number of entries that the command should show (the default is 24 entries).

6.10.37.5 setenv Command

The *setenv* command sets an environment variable in the PROM log. This command uses the following syntax:

setenv [n:*NODE*] *VAR* ["*STRING*"]

The *NODE* variable specifies that the command should set the variable in a remote Node's PROM. The *VAR* variable specifies the name of the environment variable; the name must

be from 1 to 15 ASCII characters. If you do not specify a *NODE*, this command uses the local Node.

The *STRING* variable is the value to which this command should set the environment variable. *STRING* must contain from 0 to 127 ASCII characters. You enclose variable names that are PROM reserved words in quotation marks (such as "ld" or "v0"). If you omit the *STRING* variable, the PROM code prompts you for a string. You must also enclose numeric values in quotation marks.

You cannot use this command on the local PROM in Dex mode because the PROM cannot be programmed while you are running the code from it.

Warning: **Executing *setenv* on a remote PROM crashes the remote Node if it is running in Dex mode (out of the PROM).**

6.10.37.6 unsetenv Command

The *unsetenv* command removes an environment variable from the PROM log. This command uses the following syntax:

unsetenv [n:*NODE*] *VAR*

The *NODE* variable specifies that the command should remove the variable in a remote Node's PROM. If you do not specify a *NODE*, this command uses the local Node. The *VAR* variable specifies the name of the environment variable; the name must be from 1 to 15 ASCII characters. You must enclose variable names that are PROM reserved words in quotation marks (such as "ld" or "v0").

You cannot use this command on the local PROM in Dex mode because the PROM cannot be programmed while you are running the code from it.

Warning: **Executing *unsetenv* on a remote PROM crashes the remote Node if it is running in Dex mode (out of the PROM).**

6.10.37.7 printenv Command

The *printenv* command displays the environment variables in the PROM log. This command uses the following syntax:

printenv [n:*NODE*] [*VAR*]

The *NODE* variable specifies that the command should display the variables in a remote Node's PROM. If you do not specify a *NODE*, this command prints the variables contained in the local HUB's PROM. The *VAR* variable specifies the name of the environment variable that the command should print. The name must be from 1 to 15 ASCII characters. You must enclose variable names that are PROM reserved words in quotation marks (such as "ld" or "v0"). If you do not specify a *VAR*, this command displays all environment variables.

6.10.38 Running the FRU Analyzer

6.10.38.1 fru Command

The *fru* command runs the FRU analyzer program. This command uses the following syntax:

fru [1 | 2]

If you specify **1**, this command runs the FRU analyzer on the local Node. If you specify **2**, this command runs the FRU analyzer on all Nodes in the system.

Refer to the *FRU Analyzer Reference Guide*, publication number 108-0166-001, for more information about the FRU analyzer program.

6.10.39 Debugging the Kernel

6.10.39.1 kdebug Command

The *kdebug* command lets you debug the kernel when the normal kernel debugger is not available. (This command moves the stack to the address that you specify or an appropriate place in lower memory, configures POD mode to use the kernel symbol table, and configures POD mode to use the kernel's stored registers.) This command uses the following syntax:

kdebug [*STACKADDR*]

The *STACKADDR* variable specifies the location in lower memory to which this command should move the stack. If you do not specify a *STACKADDR* value, this command moves the stack to an appropriate place in lower memory.

Note: You must load and initialize the kernel before you can use this command.

6.10.40 Setting the Verbosity Level of the Boot Sequence

6.10.40.1 verbose Command

The *verbose* command sets the verbosity level of the boot sequence. This command corresponds to debug DIP switch 3 (the boot information level); refer to Section 1.2, "Modifying How the Power-On Diagnostics are Run," for more information about debug DIP switch 3. This command uses the following syntax:

verbose [0 | 1]

If you specify **0**, the PROM does not display detailed information messages during the boot sequence. If you specify **1**, the PROM does display detailed information messages during the boot sequence. By default, this command is set to the value of debug DIP switch 3.

Appendix A

Register Names in POD Mode

This Appendix lists all of the available register names that you can use in expressions or commands in POD mode. POD mode automatically replaces the register names with the addresses that are necessary to access the registers.

A.1 HUB Processor Interface Registers

You can use the following names to access the HUB processor interface (PI) registers:

```
PI_BIST_COUNT_TARGET
PI_BIST_ENTER_RUN
PI_BIST_RDY
PI_BIST_READ_DATA
PI_BIST_SHIFT_LOAD
PI_BIST_SHIFT_UNLOAD
PI_BIST_WRITE_DATA
PI_CALIAS_SIZE
PI_CC_MASK
PI_CC_PEND_CLR_A
PI_CC_PEND_CLR_B
PI_CC_PEND_SET_A
PI_CC_PEND_SET_B
PI_CPU_ENABLE_A
PI_CPU_ENABLE_B
PI_CPU_NUM
PI_CPU_PRESENT_A
PI_CPU_PRESENT_B
PI_CPU_PROTECT
PI_CRB_SFACTOR
PI_CRB_TIMEOUT_A
PI_CRB_TIMEOUT_B
PI_ERR_INT_MASK_A
PI_ERR_INT_MASK_B
PI_ERR_INT_PEND
PI_ERR_STACK_ADDR_A
PI_ERR_STACK_ADDR_B
PI_ERR_STACK_FORMAT
PI_ERR_STACK_SIZE
PI_ERR_STATUS0_A
PI_ERR_STATUS0_A_CLEAR
PI_ERR_STATUS0_B
```

```
PI_ERR_STATUS0_B_CLEAR
PI_ERR_STATUS1_A
PI_ERR_STATUS1_A_CLEAR
PI_ERR_STATUS1_B
PI_ERR_STATUS1_B_CLEAR
PI_FORCE_BAD_CHECK_BIT_A
PI_FORCE_BAD_CHECK_BIT_B
PI_GFX_BIAS_A
PI_GFX_BIAS_B
PI_GFX_CREDIT_CNTR_A
PI_GFX_CREDIT_CNTR_B
PI_GFX_INT_CMP_A
PI_GFX_INT_CMP_B
PI_GFX_INT_CNTR_A
PI_GFX_INT_CNTR_B
PI_GFX_PAGE_A
PI_GFX_PAGE_B
PI_INT_MASK0_A PI_INT_MASK0_B
PI_INT_MASK1_A PI_INT_MASK1_B
PI_INT_PEND0
PI_INT_PEND1
PI_INT_PEND_MOD
PI_IO_PROTECT
PI_MAX_CRB_TIMEOUT
PI_NACK_CMP
PI_NACK_CNT_A
PI_NACK_CNT_B
PI_NMI_A PI_NMI_B
PI_PROFILE_COMPARE
PI_PROF_INT_EN_A
PI_PROF_INT_EN_B
PI_PROF_INT_PEND_A
PI_PROF_INT_PEND_B
PI_PROT_OVRRD
PI_REGION_PRESENT
PI_REPLY_LEVEL
PI_RTC_INT_EN_A
PI_RTC_INT_EN_B
PI_RT_COMPARE_A
PI_RT_COMPARE_B
PI_RT_COUNTER
PI_RT_FILTER_CTRL
PI_RT_INT_PEND_A
PI_RT_INT_PEND_B
PI_RT_LOCAL_CTRL
PI_SOFTRESET
PI_SPOOL_CMP_A
PI_SPOOL_CMP_B
PI_SYSAD_ERRCHK_EN
PI_UNUSED
```

A.2 HUB Memory/Directory Registers

You can use the following names to access the HUB memory/directory (MD) registers:

```
MD_DIR_DIMM_INIT
MD_DIR_ERROR
MD_DIR_ERROR_CLR
MD_FANDOP_CAC_STAT
MD_HSPEC_PROTECT
MD_IO_PROTECT
MD_IO_PROT_OVRRD
MD_LED0 MD_LED1
MD_MEMORY_CONFIG
MD_MEM_DIMM_INIT
MD_MEM_ERROR
MD_MEM_ERROR_CLR
MD_MIG_CANDIDATE
MD_MIG_CANDIDATE_CLR
MD_MIG_DIFF_THRESH
MD_MIG_VALUE_THRESH
MD_MISC_ERROR
MD_MISC_ERROR_CLR
MD_MLAN_CTL
MD_MOQ_SIZE
MD_PERF_CNT0
MD_PERF_CNT1
MD_PERF_CNT2
MD_PERF_CNT3
MD_PERF_CNT4
MD_PERF_CNT5
MD_PERF_SEL
MD_PROTOCOL_ERROR
MD_PROTOCOL_ERROR_CLR
MD_REFRESH_CONTROL
MD_SLOT_ID_USTATUS
MD_UREG0_0
MD_UREG0_1
MD_UREG0_2
MD_UREG0_3
MD_UREG0_4
MD_UREG0_5
MD_UREG0_6
MD_UREG0_7
MD_UREG1_0
MD_UREG1_1
MD_UREG1_10
MD_UREG1_11
MD_UREG1_12
MD_UREG1_13
MD_UREG1_14
MD_UREG1_15
MD_UREG1_2
MD_UREG1_3
MD_UREG1_4
MD_UREG1_5
```

```
MD_UREG1_6
MD_UREG1_7
MD_UREG1_8
MD_UREG1_9
```

A.3 HUB I/O Interface Registers

You can use the following names to access the HUB I/O interface (II) registers:

```
II_IBCT0
II_IBCT1
II_IBDA0
II_IBDA1
II_IBIA0
II_IBIA1
II_IBLS0
II_IBLS1
II_IBNA0
II_IBNA1
II_IBSA0
II_IBSA1
II_ICCR
II_ICDR
II_ICMR
II_ICRB0_A
II_ICRB0_B
II_ICRB0_C
II_ICRB0_D
II_ICRB1_A
II_ICRB1_B
II_ICRB1_C
II_ICRB1_D
II_ICRB2_A
II_ICRB2_B
II_ICRB2_C
II_ICRB2_D
II_ICRB3_A
II_ICRB3_B
II_ICRB3_C
II_ICRB3_D
II_ICRB4_A
II_ICRB4_B
II_ICRB4_C
II_ICRB4_D
II_ICRB5_A
II_ICRB5_B
II_ICRB5_C
II_ICRB5_D
II_ICRB6_A
II_ICRB6_B
II_ICRB6_C
II_ICRB6_D
II_ICRB7_A
II_ICRB7_B
II_ICRB7_C
```

```
II_ICRB7_D
II_ICRB8_A
II_ICRB8_B
II_ICRB8_C
II_ICRB8_D
II_ICRB9_A
II_ICRB9_B
II_ICRB9_C
II_ICRB9_D
II_ICRBA_A
II_ICRBA_B
II_ICRBA_C
II_ICRBA_D
II_ICRBB_A
II_ICRBB_B
II_ICRBB_C
II_ICRBB_D
II_ICRBC_A
II_ICRBC_B
II_ICRBC_C
II_ICRBC_D
II_ICRBD_A
II_ICRBD_B
II_ICRBD_C
II_ICRBD_D
II_ICRBE_A
II_ICRBE_B
II_ICRBE_C
II_ICRBE_D
II_ICTO
II_ICTP
II_IECLR
II_IFDR
II_IGFX0
II_IGFX1
II_IIAP
II_IIDEM
II_IIDSR
II_IIWA
II_ILAPO
II_ILAPR
II_ILCSR
II_ILLR
II_IMEM
II_IOWA
II_IPCA
II_IPCR
II_IPDR
II_IPPR
II_IPRB0
II_IPRB8
II_IPRB9
II_IPRBA
II_IPRBB
II_IPRBC
II_IPRBD
II_IPRBE
```

```
II_IPRBF
II_IPRTE
II_IPRTE0
II_IPRTE1
II_IPRTE2
II_IPRTE3
II_IPRTE4
II_IPRTE5
II_IPRTE6
II_IPRTE7
II_IPTP0
II_IPTP1
II_ITTE1
II_ITTE2
II_ITTE3
II_ITTE4
II_ITTE5
II_ITTE6
II_ITTE7
II_IXCC
II_IXTT
II_NOT_DONE
II_WCR
II_WID
II_WSTAT
```

A.4 HUB Network Interface Registers

You can use the following names to access the HUB network interface (NI) registers:

```
NI_AGE_CPU0_MEMORY
NI_AGE_CPU0_PIO
NI_AGE_CPU1_MEMORY
NI_AGE_CPU1_PIO
NI_AGE_GBR_MEMORY
NI_AGE_GBR_PIO
NI_AGE_IO_MEMORY
NI_AGE_IO_PIO NI_DIAG_PARMS
NI_ERROR_CLEAR
NI_GLOBAL_PARMS
NI_IO_PROTECT
NI_LOCAL_TABLE<00>
NI_META_TABLE<00>
NI_PORT_ERROR
NI_PORT_PARMS
NI_PORT_RESET
NI_PROTECTION_CONFIG
NI_PROT_OVRRD
NI_RETURN_VECTOR
NI_SCRATCH_REG0
NI_SCRATCH_REG1
NI_STATUS_REV_ID
NI_VECTOR
NI_VECTOR_CLEAR
NI_VECTOR_DATA
```

```
NI_VECTOR_PARMS
NI_VECTOR_READ_DATA
NI_VECTOR_STATUS
```

A.5 HUB Internal Crossbar Control Registers

You can use the following names to access the HUB internal crossbar control (CORE) registers:

```
CORE_CACR
CORE_CDBGSEL
CORE_CFDR
CORE_CMDAB
CORE_CNDAB
CORE_CRPQD
CORE_CRQQD
```

A.6 Midplane Crossbow Registers

You can use the following names to access the midplane Crossbow (XBOW) registers:

```
XB_ARB_RELOAD
XB_CTRL
XB_ERR_CMDWORD
XB_ERR_LOWER
XB_ERR_UPPER
XB_ID XB_INT_LOWER
XB_INT_UPPER
XB_LINK_ARB_LOWER_8
XB_LINK_ARB_LOWER_9
XB_LINK_ARB_LOWER_A
XB_LINK_ARB_LOWER_B
XB_LINK_ARB_LOWER_C
XB_LINK_ARB_LOWER_D
XB_LINK_ARB_LOWER_E
XB_LINK_ARB_LOWER_F
XB_LINK_ARB_UPPER_8
XB_LINK_ARB_UPPER_9
XB_LINK_ARB_UPPER_A
XB_LINK_ARB_UPPER_B
XB_LINK_ARB_UPPER_C
XB_LINK_ARB_UPPER_D
XB_LINK_ARB_UPPER_E
XB_LINK_ARB_UPPER_F
XB_LINK_AUX_STAT_8
XB_LINK_AUX_STAT_9
XB_LINK_AUX_STAT_A
XB_LINK_AUX_STAT_B
XB_LINK_AUX_STAT_C
XB_LINK_AUX_STAT_D
XB_LINK_AUX_STAT_E
XB_LINK_AUX_STAT_F
```

```
XB_LINK_CTRL_8
XB_LINK_CTRL_9
XB_LINK_CTRL_A
XB_LINK_CTRL_B
XB_LINK_CTRL_C
XB_LINK_CTRL_D
XB_LINK_CTRL_E
XB_LINK_CTRL_F
XB_LINK_IBUF_FLUSH_8
XB_LINK_IBUF_FLUSH_9
XB_LINK_IBUF_FLUSH_A
XB_LINK_IBUF_FLUSH_B
XB_LINK_IBUF_FLUSH_C
XB_LINK_IBUF_FLUSH_D
XB_LINK_IBUF_FLUSH_E
XB_LINK_IBUF_FLUSH_F
XB_LINK_RESET_8
XB_LINK_RESET_9
XB_LINK_RESET_A
XB_LINK_RESET_B
XB_LINK_RESET_C
XB_LINK_RESET_D
XB_LINK_RESET_E
XB_LINK_RESET_F
XB_LINK_STAT_8
XB_LINK_STAT_9
XB_LINK_STAT_A
XB_LINK_STAT_B
XB_LINK_STAT_C
XB_LINK_STAT_CLR_8
XB_LINK_STAT_CLR_9
XB_LINK_STAT_CLR_A
XB_LINK_STAT_CLR_B
XB_LINK_STAT_CLR_C
XB_LINK_STAT_CLR_D
XB_LINK_STAT_CLR_E
XB_LINK_STAT_CLR_F
XB_LINK_STAT_D
XB_LINK_STAT_E
XB_LINK_STAT_F
XB_LLP_CTRL
XB_NIC
XB_PERF_CTR_A
XB_PERF_CTR_B
XB_PKT_TO
XB_STAT
XB_STAT_CLR
```

A.7 Router Registers

You can use the following names to access the Router registers:

```
RR_BIST_COUNT_TARGET
RR_BIST_DATA
RR_BIST_ENTER_RUN
```

```
RR_BIST_READY
RR_BIST_SHIFT_LOAD
RR_BIST_SHIFT_UNLOAD
RR_DIAG_PARMS
RR_ERROR_CLEAR1
RR_ERROR_CLEAR2
RR_ERROR_CLEAR3
RR_ERROR_CLEAR4
RR_ERROR_CLEAR5
RR_ERROR_CLEAR6
RR_GLOBAL_PARMS
RR_HISTOGRAM1
RR_HISTOGRAM2
RR_HISTOGRAM3
RR_HISTOGRAM4
RR_HISTOGRAM5
RR_HISTOGRAM6
RR_LOCAL_TABLE<00>
RR_META_TABLE<00>
RR_NIC_ULAN
RR_PORT_PARMS1
RR_PORT_PARMS2
RR_PORT_PARMS3
RR_PORT_PARMS4
RR_PORT_PARMS5
RR_PORT_PARMS6
RR_PORT_RESET
RR_PROTECTION_CONFIG
RR_RESET_MASK1
RR_RESET_MASK2
RR_RESET_MASK3
RR_RESET_MASK4
RR_RESET_MASK5
RR_RESET_MASK6
RR_SCRATCH_REG0
RR_SCRATCH_REG1
RR_STATUS_ERROR1
RR_STATUS_ERROR2
RR_STATUS_ERROR3
RR_STATUS_ERROR4
RR_STATUS_ERROR5
RR_STATUS_ERROR6
RR_STATUS_REV_ID
```

A.8 R10000 Coprocessor 0 Registers

You can use the following names to access the R10000 COP0 registers:

```
C0_23
C0_24
C0_7
C0_BADVADDR
C0_BRDIAG
C0_CACHEERR
C0_CAUSE
```

```
C0_COMPARE
C0_CONFIG
C0_CONTEXT
C0_COUNT
C0_ECC
C0_ENTRYHI
C0_ENTRYLO0
C0_ENTRYLO1
C0_EPC
C0_ERROREPC
C0_FRAMEMASK
C0_INDEX
C0_LLADDR
C0_PAGEMASK
C0_PERFCOUNT
C0_PRID
C0_RANDOM
C0_STATUS
C0_TAGHI
C0_TAGLO
C0_WATCHHI
C0_WATCHLO
C0_WIRED
C0_XCONTEXT
```

Glossary

cached (Cac) mode

The mode in which the power-on diagnostics place program code, data, and stack into cached memory. Cac mode is available only after memory has been probed and configured. POD execution is much quicker in Cac mode than in Dex mode or Unc mode. *See also* dirty exclusive (Dex) mode and uncached (Unc) mode.

dirty exclusive (Dex) mode

The mode in which the power-on diagnostics require the fewest system resources to run. Dex mode does not require memory; it accesses program code directly from the PROM and uses the data cache in the R10000 processors as the memory for its stack. Dex mode does not use the secondary cache. Nonmaskable interrupts and uncaught exceptions typically return you to the Dex mode prompt.

Dex mode is very slow because PROM instruction fetches are slow. Avoid performing long memory tests or flashing remote PROMs from Dex mode. *See also* cached (Cac) mode and uncached (Unc) mode.

headless Node

A Node without functional CPUs.

NASID value

A number that indicates the Numa Address Space IDentifier. A NASID value is a unique number that specifies a Node in the system.

nonmaskable interrupt (NMI)

An interrupt that forces a CPU to stop processing and jump to a fixed address in the PROM. The CPU then runs the code, called the NMI handler, which is located at that address. Nonmaskable interrupts are used to debug a system that is hung (usually because of a PROM or Kernel bug).

You can issue an NMI by pressing the NMI button on the MSC or by executing the MSC *nmi* command.

NMI handler

The code at a fixed address in the PROM that the CPU accesses when it receives a nonmaskable interrupt (NMI). This code attempts to bring the system to a PROM or kernel monitor so you can examine the state of the system and determine why it hung.

number in a can (NIC)

A Dallas Semiconductor chip that contains a 48-bit number that is permanently burned in by a laser when the chip is manufactured. The 48-bit number is different for each board in the system and provides a way for software to determine which boards are in the system.

An NIC also contains several pages of nonvolatile memory that the system can read and write. This memory contains the following information: board type, board revision number, board serial number, and board-specific information.

PROM environment list

A group of PROM environment variable values that are stored under one name.

PROM environment variable

A variable stored in the PROM log that changes the way in which the PROM environment operates.

R10000 processor

The MIPS processor used on the IP27 board. This processor is also called the T5.

T5

The internal code name for the R10000 processor. Some diagnostics still refer to R10000 processors as the T5 processors.

uncached (Unc) mode

The mode in which power-on diagnostics place their program code, data, and stack into uncached memory. This mode is similar to Cac mode, but Unc mode runs slower; however, program execution is quicker in Unc mode than in Dex mode. *See also* cached (Cac) mode and dirty exclusive (Dex) mode.

Widget

A generic name for any device that can connect to the PCI interface.