Origin200™/2000™ and Onyx2™ MDK-Based Field Diagnostics

Document Number 108-0162-001

Contributors

Written by Darrin Goss
Edited by Cindi Leiser
Production by Mike Dixon
Engineering contributions by Judy Young, Imran Pasha, Curt Chambliss, and Mike Galles

Origin 200™/2000™ and Onyx2™ MDK-Based Field Diagnostics
Document Number 108-0162-001

Silicon Graphics, Inc.
Mountain View, California

Contents

Figures

Tables

About This Guide

This document describes the micro-diagnostic kernel (MDK) and the MDK-based diagnostics that you can use to troubleshoot Origin200™, Origin2000™, and CRAY® Origin2000™ and Onyx2™ computer systems. System Support Engineers (SSEs) and Field Engineers (FEs) may use this reference document during training and on-site.

The document organizes information into the following chapters:

- Chapter 1, "Introduction to MDK," provides an overview of MDK and the diagnostic tests that you can run from MDK.
- Chapter 2, "R10000 Processor Tests," describes the diagnostic tests that you can use to test the R10000 processors.
- Chapter 3, "Secondary Cache Test," describes the diagnostic test that you can use to test the secondary caches on the Node boards.
- Chapter 4, "Memory Tests," describes the diagnostic tests that you can use to test memory.
- Chapter 5, "Router SSO Test," describes the diagnostic test that you can use to test Router links.
- Chapter 6, "Cache Thrasher With I/O Test," describes the diagnostic test that you can use to stress test the CrayLink™ network and the HUB-to-XBOW links.
- Appendix A, "Troubleshooting Link Failures," provides general information to help you isolate link failures.

Note: This document supports MDK version 1.20. To determine the version of MDK that you are using, look at the MDK start-up banner:

```
**********************************************************
SGI Nanos Version 1.20 SN0 built 11:25:03 AM Feb 21, 1997
**********************************************************
```

Typographical Conventions

This document uses the following typographical conventions and symbols:

- Information displayed on the screen is shown in `Courier` type.
- Commands that you should enter are shown in **`Courier bold`** type.
- Commands in text and filenames are shown in *italic* type.
- Variables are shown in *italic* type.
- The >> symbol indicates the PROM monitor (or BaseIO command) prompt.
- The `MDK>` symbol indicates the MDK prompt.

Terminology

This document uses the following terms interchangeably:

- HUB and HUB chip
- crossbow, XBOW, and XBOW chip
- diagnostic, diagnostic test, and test

Chapter 1

Introduction to MDK

This chapter describes the micro-diagnostic kernel (MDK) and provides an overview of the MDK-based diagnostics that you can use to troubleshoot Origin200, Origin2000, CRAY Origin2000, and Onyx2 systems.

1.1 About the Micro-Diagnostic Kernel

The micro-diagnostic kernel (MDK) is a standalone diagnostic environment that runs in kernel mode on Origin200, Origin2000, CRAY Origin2000, and Onyx2 systems. MDK lets you load and run diagnostic tests in an environment that provides access to hardware components that cannot be accessed from a user program that is running under the IRIX™ operating system.

MDK-based diagnostics require full use of the system: you must bring down (reboot) the system to boot MDK. Therefore, no other kernel (that is, the IRIX operating system) can be running when you use the MDK-based diagnostics. You should use the MDK-based diagnostics to isolate failures for which the IRIX based diagnostics do not provide enough error information or to test hardware that the IRIX based diagnostics cannot test.

MDK provides features that enable MDK-based diagnostic tests to operate like normal user programs; these features include memory allocation, console output, exception handling, and multiprocessing. MDK also provides system configuration information (memory size and number of CPUs, Nodes, and Routers) for the diagnostic tests.

This document describes MDK version 1.20, which includes several test packages: *t5r.mdk*, *memory.mdk*, and *stress.mdk*. Each test package binary includes the micro-diagnostic kernel and a set of diagnostic tests. (Because MDK does not support a filesystem, it is not possible to load individual tests from the disk.) The following test packages are available:

- The *t5r.mdk* test package contains the micro-diagnostic kernel, the R10000 processor tests, and the Router SSO test.
- The *memory.mdk* test package contains the micro-diagnostic kernel, the secondary cache test, and several memory tests.
- The *stress.mdk* package contains the micro-diagnostic kernel and the cache thrasher with I/O test.

A major advantage of test packages is that you do not need to reboot the system to run another test from the same test package; you simply start the second test from the `MDK>` prompt that appears after the first test completes execution. However, you do need to reboot the system to run a test in a different test package.

Earlier versions of MDK were called Nanos, so you may see references to Nanos in the output from the MDK-based tests. For example, the MDK start-up banner includes the word `Nanos`:

```
**********************************************************
SGI Nanos Version 1.20 SN0 built 11:25:03 AM Feb 21, 1997
**********************************************************
```

MDK-based tests were developed in two modes: mapped and unmapped.

1.1.1 Mapped Mode

In mapped mode, MDK provides memory allocation and uses the translation lookaside buffer (TLB) for mapping virtual addresses to physical memory addresses. Mapped tests are limited to 1 Gbyte of user address space.

1.1.2 Unmapped Mode

MDK-based tests that run in unmapped mode do not use the TLB, so these tests are not limited to 1 Gbyte of user address space. Unmapped tests manage the physical memory themselves.

1.2 Available Diagnostic Tests

Two types of diagnostic tests are available in MDK: directed tests and stress tests.

1.2.1 Directed Tests

Directed tests focus on one area of the system. The areas tested include the R10000™ processors, secondary cache, memory, and the Router boards.

1.2.1.1 R10000 Processor Tests

The R10000 processor tests are directed tests that focus on the components within each R10000 processor. These tests check the following areas:

- the functionality of the external interface to each R10000 processor
- the arithmetic logic unit (ALU) in each R10000 processor
- the floating-point unit (FPU) in each R10000 processor
- the primary cache in each R10000 processor
- the instruction pipeline in each R10000 processor
- the register-renaming functionality of each R10000 processor

If an R10000 processor test fails, the failing hardware is the processor in which the test detected the errors. The failing field replaceable unit (FRU) is the Node board that contains the failing R10000 processor.

To run the R10000 processor tests, you must load the *t5r.mdk* test package.

Refer to Chapter 2, "R10000 Processor Tests," for more information.

1.2.1.2 Secondary Cache Test

The secondary cache test verifies the functionality of the secondary cache located on each Node board.

If this test fails, the failing hardware component is the secondary cache or memory. The failing FRU is the Node board that contains the failing secondary cache or the failing memory dual inline memory module (DIMM).

To run the secondary cache test, you must load the *memory.mdk* test package.

Refer to Chapter 3, "Secondary Cache Test," for more information.

1.2.1.3 Memory Tests

The memory tests verify that memory actually contains data that was written to it. The memory tests range from testing memory with 1 or 2 CPUs to testing memory with all Nodes and CPUs in the system.

If a memory test fails, the failing hardware is one or more memory DIMMs, the secondary cache (HIMM), a Node board, a midplane, or a Router board. The failing FRU is either a DIMM, Node board, midplane, or Router board.

To run the memory tests, you must load the *memory.mdk* test package.

Refer to Chapter 4, "Memory Tests," for more information.

1.2.1.4 Router SSO Test

The Router SSO test checks the Router links.

If this test detects a Router failure, the failing hardware is a compression (CPOP) connector, a terminator, or a Router chip. The failing FRU is a Node board, midplane, Router board, or connecting cable. If this test detects a HUB failure, the failing FRU is a Node board, Router board, or midplane.

Note: Before you replace any FRUs, reseat the suspected boards and reconnect the suspected cables, and then run the test again. If the test still fails, you will need to replace one or more of the FRUs.

To run the Router SSO, you must load the *t5r.mdk* test package.

Refer to Chapter 5, "Router SSO Test," for more information.

1.2.2 Stress Tests

The stress tests use many hardware components to test a larger portion of the system. There is one stress test available for use in the field: the cache thrasher with I/O test.

1.2.2.1 Cache Thrasher With I/O Test

The cache thrasher with I/O test stress tests the CrayLink network and the HUB-to-XBOW links. It also stress tests cache coherency from the processor and from I/O.

If this test fails, check the Router board LEDs and Crossbow LEDs on the midplane for lost links.

The cache thrasher with I/O test is part of the stress.mdk test package; you must load the *stress.mdk* package to run the cache thrasher with I/O test.

Refer to Chapter 6, "Cache Thrasher With I/O Test," for more information.

1.3 Installing MDK

The *Internal Support Tools 1.1* CD includes version 1.20 of MDK and the MDK-based diagnostics. Refer to the CD booklet for information about how to install the MDK software in the */stand* directory on your system disk.

Once the MDK software is installed on your system disk, you can use the *boot* command to load MDK into the system you want to test. Refer to Table 1-1 and the individual diagnostic descriptions later in this document for the specific commands used to load each diagnostic.

You can use the ***bootp*** command to run MDK from another system on the network. To do this, perform the following steps:

- On the bootp server, copy the *t5r.mdk*, *memory.mdk*, and *stress.mdk* binaries to */usr/local/boot*.
- On the client, boot MDK with the following command:

  >>bootp()server_name_or_IP_address:mdk_test_package

  (for example, bootp()hui.csd.sgi.com:memory.mdk)
- At the `MDK>` prompt that appears, enter the name of the MDK diagnostic that you want to run.

Any future updates to the MDK-based diagnostics will be available from the Internal Support Tools website (`http://ist.csd.sgi.com/Tools/Home_pages/products.html`) or on future CD releases.

1.4 Commands Available at the MDK Prompt

When you start one of the MDK packages from the PROM monitor (at the BaseIO command prompt, >>), the `MDK>` prompt appears. From this prompt, you can enter commands that control MDK or commands that run diagnostic tests.

1.4.1 Commands for Controlling MDK

1.4.1.1 help Command

The *help* command prints a description of the command that you specify or prints a list of available commands if you do not specify a command. This command uses the following syntax:

`help [`*command*`]`

1.4.1.2 kill Command

The *kill* command terminates the process that you specify. This command uses the following syntax:

`kill` *pid*

Use the *pid* variable to specify the process ID (PID) of the process that you want to terminate.

When a diagnostic process takes an unexpected exception, MDK places the process in a suspended, or frozen, state. Use the *kill* command to terminate this "frozen" process before you run any other diagnostic tests. (Use the *ps* command to determine the PID for the frozen process.)

1.4.1.3 killall Command

The *killall* command terminates all active diagnostic processes. (Using this command is equivalent to using the *kill* command with the PID from every active test.) This command uses the following syntax:

```
killall
```

1.4.1.4 ls Command

The *ls* command lists the names of all available diagnostic tests in the MDK test package that is currently loaded. This command uses the following syntax:

```
ls
```

For example, when the *memory.mdk* package is loaded, the *ls* command returns the following information:

```
MDK> ls

scache
memum.nb
memum.lo
memum.in
memum.qs
mem.lo
mem.qs

MDK>
```

1.4.1.5 ps Command

The *ps* command returns process status information, including the PID value, for all active diagnostic processes. This command uses the following syntax:

```
ps
```

1.4.1.6 reset Command

The *reset* command performs a soft reset of the system. (Using this command is equivalent to entering the *reset* command from the PROM monitor [at the BaseIO command prompt, >>].) This command uses the following syntax:

```
reset
```

1.4.2 Commands for Loading a Diagnostic Test

To run an MDK-based diagnostic test, enter the name of the diagnostic test at the `MDK>` prompt. When the test completes, the `MDK>` prompt reappears.

Remember, you can run only the diagnostic tests that are part of the currently loaded MDK package (*t5.mdk*, *memory.mdk*, or *stress.mdk*). Table 1-1 lists the commands that are used to run each of the diagnostic tests.

1.5 Diagnostic Test Output

All MDK-based diagnostics return output to the BaseIO console (the console that connects to the BaseIO board). The diagnostics print pass and fail messages as they run; all MDK-based diagnostics print a *testname* `test has completed` message when they complete testing.

1.5.1 Output From Successful Completion

When an MDK-based diagnostic completes successfully, the diagnostic displays one or more messages that indicate that it completed without errors.

For example, the Router SSO test returns the following messages when it completes testing without detecting any hardware failures:

```
TEST RESULT: **** TEST PASSED  ****
MDK Router_sso test run is complete.
```

1.5.2 Output When a Test Detects a Failure

When an MDK-based diagnostic detects a hardware failure, it prints a failure message and any information related to the failure.

For example, the `ERROR: Test Failed` message in the following output is from a test that detected a hardware failure:

```
NODE0 CPU1 Pid 2 FPU3 Test Started
NODE0 CPU0 Pid 1 FPU3 Test Started
NODE0 CPU1 Pid 2 FPU3 Test Passed
NODE0 CPU0 Pid 1 ERROR: Test Failed
```

1.5.3 Output When an Unexpected Exception Occurs

When an unexpected exception occurs, MDK prints out the contents of the Coprocessor 0 registers, the general-purpose registers, and the following information about the exception: the type of exception, the address where the exception occurred, and any other information that is relevant to the exception.

After MDK prints out this information, the `MDK>` prompt appears. At this point, you can load another diagnostic test, reset the system, or issue a non-maskable interrupt (NMI).

The following example shows MDK output from an unexpected exception:

```
ERROR: Unexpected Exception Occured.
 Nasid 1: Local CPU A: Global CPU 2: PID 3: Multiple exceptions at
0xa800000000a0c4e8: sr = 0xb4007fe6: cause = 0x881c
 Nanos: Frozen 22A 001:
 Status: 0xffffffffb4007fe6        XCtxt: 0xffffffe000002a30
 Hi:     0x                0       Lo:    0x                0
 epc:    0xa800000000a0c4e8        cause: 0x             881c
 count:  0x         18afb669       comp:  0xffffffffffffffff
 Enhi:   0x                0       CacErr: 0xffffffffd442aa80
 R01/AT: 0x            1ff00       R02/V0: 0xffffffffb4007fe6
 R03/V1: 0x           a257b0       R04/A0: 0xa800000000a27000
 R05/A1: 0x           a1dd38       R06/A2: 0x         8bfdfdf0
 R07/A3: 0x                8       R08/A4: 0x                4
 R09/A5: 0xa800000183fdf978        R10/A6: 0x         3e17aad8
 R11/A7: 0xa800000300caaad8        R12/T0: 0x               38
 R13/T1: 0x                0       R14/T2: 0x                0
 R15/T3: 0xa800000300caaa98        R16/S0: 0xa800000183fdf978
 R17/S1: 0xa8000000005fbdc8        R18/S2: 0x                0
 R19/S3: 0x                0       R20/S4: 0x                2
 R21/S5: 0xa800000008000000        R22/S6: 0x                3
 R23/S7: 0x               e0       R24/T8: 0x          8000000
 R25/T9: 0xa800000000a07ed8        R28/GP: 0x                0
 R29/SP: 0xa800000183fdd928        R30/S8: 0xa800000000a27000
 R31/RA: 0xa800000000a08eac
 Nanos: Frozen 3
```

1.6 Debugging Tips

If a hang or unexpected exception occurs while you are running MDK-based diagnostics, use the following POD mode commands to gather more information that you can use to isolate the problem:

- *nmi*, which issues a nonmaskable interrupt (NMI)
- *why*, which displays the current exception and NMI status information
- *error*, which displays error information from the MD section of the HUB
- *error_dump*, which displays detailed information about all error bits in the HUB
- *dumpspool*, which dumps a CPU's PI error spool
- *pr*, which prints the contents of an R10000 or HUB register
- *crb*, which dumps the I/O interface (II) CRBs
- *crbx*, which dumps the II CRBs in a 133-column-wide format

Refer to *Origin2000 Power-On Diagnostics*, SGI part number 108-0161-001, for more information about POD mode and these commands.

1.7 Quick-Reference List of MDK-Based Diagnostics

Table 1-1 provides quick-reference information about the MDK-based tests. This information includes the following items:

- the functional unit that each test checks
- the name of each test
- the commands that you use to invoke each test from the PROM monitor (at the BaseIO command prompt, >>)
- the failure message for each test
- the possible failing FRUs for each test
- the page in this document on which you can find more information about each test

Table 1-1 Quick Reference List of MDK-Based Diagnostic Tests

Functional Unit	Test	Invocation	Failure Message	Failing FRU	Page
R10000 processors	*t5-1*	>>**boot dksc(0,1,0)/stand/t5r.mdk** MDK>**t5-1**	ERROR: Test Failed	Node board	2-2
	t5-2	>>**boot dksc(0,1,0)/stand/t5r.mdk** MDK>**t5-2**	*testname* ERROR	Node board	2-5
Secondary cache	*scache*	>>**boot dksc(0,1,0)/stand/memory.mdk** MDK>**scache**	Scache data path test FAILED Scache cell test FAILED Scache random address test FAILED	Node board	3-1
Memory	*mem.qs*	>>**boot dksc(0,1,0)/stand/memory.mdk** MDK>**mem.qs**	NODE*x* CPU*y* PID*z* *testname* test FAILED	DIMM, Node board, midplane, or Router board	4-7
	memum.qs	>>**boot dksc(0,1,0)/stand/memory.mdk** MDK>**memum.qs**	NODE*x* CPU*y* PID*z* *testname* test FAILED	DIMM, Node board, midplane, or Router board	4-7
	memum.nb	>>**boot dksc(0,1,0)/stand/memory.mdk** MDK>**memum.nb**	NODE*x* CPU*y* PID*z* *testname* test FAILED	DIMM, midplane, or Node board	4-16
	memum.in	>>**boot dksc(0,1,0)/stand/memory.mdk** MDK>**memum.in**	NODE*x* CPU*y* PID*z* *testname* test FAILED	DIMM, Node board, midplane, or Router board	4-21

Table 1-1 (continued) Quick Reference List of MDK-Based Diagnostic Tests

Functional Unit	Test	Invocation	Failure Message	Failing FRU	Page
Memory (continued)	*mem.lo*	`>>boot dksc(0,1,0)/stand/memory.mdk` `MDK>mem.lo`	`NODE`*x* `CPU`*y* `PID`*z* *testname* `test FAILED`	DIMM, Node board, midplane, or Router board	4-27
	memum.lo	`>>boot dksc(0,1,0)/stand/memory.mdk` `MDK>memum.lo`	`NODE`*x* `CPU`*y* `PID`*z* *testname* `test FAILED`	DIMM, Node board, midplane, or Router board	4-27
Router links	*router*	`>>boot dksc(0,1,0)/stand/t5r.mdk` `MDK>router`	`ERROR`, `TEST RESULT: **** WARNING`, or `TEST RESULT: ****FAIL`	Router board, Node board, midplane, or cable	5-1
CrayLink network and HUB-to-XBOW links	*ct_io_turbo*	`>>boot dksc(0,1,0)/stand/stress.mdk` `MDK>ct_io_turbo`	No messages for several minutes (hang), `ERROR: Unexpected Exception Occurred`, or some type of `ERROR` message	Router board, Node board, midplane, or cable	6-1

Chapter 2

R10000 Processor Tests

The chapter describes the diagnostics that you can use to test the R10000 processors.

2.1 About the R10000 Processor Tests

The R10000 processor tests are directed tests that focus on the components within the R10000 processors. To run these diagnostic tests, you must load the *t5r.mdk* test package.

There are two R10000 processor tests: *t5-1* and *t5-2*. Table 2-1 provides quick-reference descriptions of these tests. The number in the Page column indicates the page on which a detailed description of each test begins.

Table 2-1 R10000 Processor Tests

Diagnostic Test	Description	Page
t5-1	Tests the arithmetic logic unit (ALU) and portions of the floating-point unit (FPU)	2-2
t5-2	Tests portions of the FPU, the external interface to the R10000 processor, the primary cache, the instruction issue queue, instruction issuing, register renaming, and several R10000 processor branch conditions	2-5

If either of these tests fails, the failing FRU is the Node board that contains the failing R10000 processor, with the following exception: the tests could encounter an uncorrectable memory error while attempting to access data from secondary cache or memory; if this occurs, the secondary cache test or memory tests should also detect the error.

2.2 *t5-1* Test

The *t5-1* test checks the arithmetic logic unit (ALU) hardware and portions of the floating-point unit (FPU) hardware.

2.2.1 How to Run the *t5-1* Test

Use the following procedure to load the test from the local system disk:

- Use the *boot* command from the PROM monitor (at the BaseIO command prompt, >>) to start the *t5r.mdk* test package:

  ```
  >>boot dksc(0,1,0)/stand/t5r.mdk
  ```

  MDK displays boot status information and the `MDK>` prompt.

- Load the *t5-1* test from the `MDK>` prompt:

  ```
  MDK> t5-1
  ```

  The *t5-1* test loads into the system and begins testing the R10000 processors.

2.2.2 *t5-1* Test Description

The *t5-1* test focuses on two areas of the R10000 processors: the ALU and the FPU. The ALU section of the *t5-1* test checks the ALU by executing a variety of ALU instructions and comparing the results with expected values. The FPU, FPU2, FPU3, FPU4, and FPU5 sections of the *t5-1* test check the FPU with a variety of vectors.

Note: The FPU6 section of the *t5-2* test provides additional FPU testing.

2.2.3 *t5-1* Test Output

The *t5-1* test returns output to the BaseIO console.

2.2.3.1 Pass Output

The *t5-1* test prints a `Test Passed` message for each section that passes without detecting hardware failures. The test prints the message `MDK R10000 combined test run is complete` when all sections have completed testing.

For example, the following messages indicate that the FPU5 test passed and all tests have completed testing:

```
NODE1 CPU2 Pid 3 FPU5 Test Started
NODE1 CPU2 Pid 3 FPU5 Test Passed
MDK R10000 combined test run is complete.
```

2.2.3.2 Failure Output

When a section of this test detects a failure, the test prints an error message that indicates which test section failed.

For example, the following failure output indicates that the FPU3 test section detected a failure (because the FPU3 test section was running in the CPU that reported the error):

```
NODE0 CPU1 Pid 2 FPU3 Test Started
NODE0 CPU0 Pid 1 FPU3 Test Started
NODE0 CPU1 Pid 2 FPU3 Test Passed
NODE0 CPU0 Pid 1 ERROR: Test Failed
```

2.2.4 *t5-1* Test Example

In the following example, the *t5-1* test runs on a 2-Node system (4 CPUs) without detecting any errors.

```
>>boot dksc(0,1,0)/stand/t5r.mdk
.Booting Nanos.........

************************************************************
SGI Nanos Version 1.10 SN0 built 01:01:56 PM Mar 11, 1997
************************************************************

CPU 0: Total no. of CPUs = 4
Launched CPU 1
Launched CPU 2
Launched CPU 3
MDK>t5-1
Created successfully pid = 1
CPU 0: User mem starts from 0xa800000000641000
NODE1 CPU3 Pid 4 ALU Test Started
NODE0 CPU0 Pid 1 ALU Test Started
NODE0 CPU1 Pid 2 ALU Test Started
NODE1 CPU2 Pid 3 ALU Test Started
NODE0 CPU0 Pid 1 ALU Test Passed
NODE0 CPU1 Pid 2 ALU Test Passed
NODE1 CPU2 Pid 3 ALU Test Passed
NODE1 CPU3 Pid 4 ALU Test Passed
NODE1 CPU3 Pid 4 FPU Test Started
NODE1 CPU3 Pid 4 FPU Test Passed
NODE0 CPU0 Pid 1 FPU Test Started
NODE0 CPU1 Pid 2 FPU Test Started
NODE0 CPU0 Pid 1 FPU Test Passed
NODE0 CPU1 Pid 2 FPU Test Passed
NODE1 CPU2 Pid 3 FPU Test Started
NODE1 CPU2 Pid 3 FPU Test Passed
NODE1 CPU3 Pid 4 FPU2 Test Started
NODE1 CPU3 Pid 4 FPU2 Test Passed
NODE0 CPU0 Pid 1 FPU2 Test Started
NODE0 CPU1 Pid 2 FPU2 Test Started
NODE0 CPU0 Pid 1 FPU2 Test Passed
NODE0 CPU1 Pid 2 FPU2 Test Passed
NODE1 CPU2 Pid 3 FPU2 Test Started
```

```
NODE1 CPU2 Pid 3 FPU2 Test Passed
NODE1 CPU3 Pid 4 FPU3 Test Started
NODE1 CPU3 Pid 4 FPU3 Test Passed
NODE0 CPU1 Pid 2 FPU3 Test Started
NODE0 CPU0 Pid 1 FPU3 Test Started
NODE0 CPU1 Pid 2 FPU3 Test Passed
NODE0 CPU0 Pid 1 FPU3 Test Passed
NODE1 CPU2 Pid 3 FPU3 Test Started
NODE1 CPU2 Pid 3 FPU3 Test Passed
NODE1 CPU3 Pid 4 FPU4 Test Started
NODE1 CPU3 Pid 4 FPU4 Test Passed
NODE0 CPU0 Pid 1 FPU4 Test Started
NODE0 CPU1 Pid 2 FPU4 Test Started
NODE0 CPU0 Pid 1 FPU4 Test Passed
NODE0 CPU1 Pid 2 FPU4 Test Passed
NODE1 CPU2 Pid 3 FPU4 Test Started
NODE1 CPU2 Pid 3 FPU4 Test Passed
NODE1 CPU3 Pid 4 FPU5 Test Started
NODE1 CPU3 Pid 4 FPU5 Test Passed
NODE0 CPU0 Pid 1 FPU5 Test Started
NODE0 CPU1 Pid 2 FPU5 Test Started
NODE0 CPU0 Pid 1 FPU5 Test Passed
NODE0 CPU1 Pid 2 FPU5 Test Passed
NODE1 CPU2 Pid 3 FPU5 Test Started
NODE1 CPU2 Pid 3 FPU5 Test Passed
MDK R10000 combined test run is complete.
```

2.3 *t5-2* Test

The *t5-2* test checks portions of the FPU, the external interface to the R10000 processor, the primary cache, the instruction issue queue, instruction issuing, register renaming, and several R10000 processor branch conditions.

2.3.1 How to Run the *t5-2* Test

Use the following procedure to load the test from the local system disk:

- Use the *boot* command from the PROM monitor (at the BaseIO command prompt, >>) to start the *t5r.mdk* test package:

  ```
  >>boot dksc(0,1,0)/stand/t5r.mdk
  ```

  MDK displays boot status information and the `MDK>` prompt.

- Load the *t5-2* test from the `MDK>` prompt:

  ```
  MDK> t5-2
  ```

  The *t5-2* test loads into the system and begins testing the R10000 processors.

2.3.2 *t5-2* Test Description

The *t5-2* test has six sections; Table 2-2 describes each of the sections.

Table 2-2 t5-2 Test Sections

Section	Description
FPU6	Performs additional tests on the FPU with test vectors that are not used in the *t5-1* test
PCACHE	Checks the primary cache by writing various data patterns to the primary cache, reading the data back, and comparing the data that was read back with the data that was written
QISSUE QISSUE2	Tests the instruction queue and instruction issue timing in the R10000 pipeline by placing different instruction sequences in the pipe that cause boundary conditions (such as queue full) to occur
RENAME	Checks the functionality of register renaming and the functionality of several branch conditions in an R10000 processor
CORECTL	Checks the external interface on an R10000 processor with the following actions: Cause writebacks from primary data cache to the secondary cache with a writeback to memory. Verify that the resulting data and cache states match expected values. Cause a store hit to a clean cache line. Verify that the resulting data and cache states match expected values. Cause multiple store misses to return with primary writebacks pending. Verify that the resulting data and cache states match expected values

2.3.3 *t5-2* Test Output

The *t5-2* test returns output to the BaseIO console.

2.3.3.1 Pass Output

The *t5-2* test prints the message `Test Passed` for each section that passes without detecting hardware failures. The test prints the message `MDK R10000 combined test run is complete` when all sections have completed testing.

For example, the following messages indicate that the CORECTL test passed and all tests have completed testing:

```
NODE1 CPU2 Pid 3 CORECTL Test Started
NODE1 CPU2 Pid 3 CORECTL Test Passed
MDK R10000 combined test run is complete.
```

2.3.3.2 Failure Output

When a section of this test detects a failure, the test prints an error message that indicates which test section failed.

For example, the following failure output indicates that the rename test section detected a failure in CPU0:

```
NODE0 CPU0 Pid 1 RENAME Test Started
NODE0 CPU0 Pid 1 T5 Rename Test ERROR
NODE1 CPU3 Pid 4 RENAME Test Started
NODE1 CPU3 Pid 4 RENAME Test Passed
```

When this test fails, the failing FRU is normally the Node board that contains the failing R10000 processor. However, the test could encounter an uncorrectable memory error while attempting to access data from secondary cache or memory; if this occurs, the secondary cache test or memory tests should also detect the error.

2.3.4 *t5-2* Test Example

In the following example, the *t5-2* test runs on a 2-Node system (4 CPUs) without detecting any errors.

```
>>boot dksc(0,1,0)/stand/t5r.mdk
Booting Nanos.........

************************************************************
SGI Nanos Version 1.10 SN0 built 01:01:56 PM Mar 11, 1997
************************************************************

CPU 0: Total no. of CPUs = 4
Launched CPU 1
Launched CPU 2
Launched CPU 3
MDK>t5-2
```

```
Created successfully pid = 1
CPU 0: User mem starts from 0xa800000000641000
NODE1 CPU3 Pid 4 FPU6 Test Started
NODE0 CPU1 Pid 2 FPU6 Test Started
NODE0 CPU0 Pid 1 FPU6 Test Started
NODE0 CPU1 Pid 2 FPU6 Test Passed
NODE0 CPU0 Pid 1 FPU6 Test Passed
NODE1 CPU2 Pid 3 FPU6 Test Started
NODE1 CPU3 Pid 4 FPU6 Test Passed
NODE1 CPU2 Pid 3 FPU6 Test Passed
NODE1 CPU3 Pid 4 PCACHE Test Started
NODE0 CPU1 Pid 2 PCACHE Test Started
NODE0 CPU0 Pid 1 PCACHE Test Started
NODE0 CPU1 Pid 2 PCACHE Test Passed
NODE0 CPU0 Pid 1 PCACHE Test Passed
NODE1 CPU2 Pid 3 PCACHE Test Started
NODE1 CPU3 Pid 4 PCACHE Test Passed
NODE1 CPU2 Pid 3 PCACHE Test Passed
NODE1 CPU3 Pid 4 QISSUE Test Started
NODE1 CPU3 Pid 4 QISSUE Test Passed
NODE0 CPU0 Pid 1 QISSUE Test Started
NODE0 CPU1 Pid 2 QISSUE Test Started
NODE0 CPU0 Pid 1 QISSUE Test Passed
NODE0 CPU1 Pid 2 QISSUE Test Passed
NODE1 CPU2 Pid 3 QISSUE Test Started
NODE1 CPU2 Pid 3 QISSUE Test Passed
NODE1 CPU3 Pid 4 QISSUE2 Test Started
NODE1 CPU3 Pid 4 QISSUE2 Test Passed
NODE0 CPU1 Pid 2 QISSUE2 Test Started
NODE0 CPU0 Pid 1 QISSUE2 Test Started
NODE0 CPU1 Pid 2 QISSUE2 Test Passed
NODE0 CPU0 Pid 1 QISSUE2 Test Passed
NODE1 CPU2 Pid 3 QISSUE2 Test Started
NODE1 CPU2 Pid 3 QISSUE2 Test Passed
NODE1 CPU3 Pid 4 RENAME Test Started
NODE1 CPU3 Pid 4 RENAME Test Passed
NODE0 CPU0 Pid 1 RENAME Test Started
NODE0 CPU1 Pid 2 RENAME Test Started
NODE1 CPU2 Pid 3 RENAME Test Started
NODE0 CPU0 Pid 1 RENAME Test Passed
NODE0 CPU1 Pid 2 RENAME Test Passed
NODE1 CPU2 Pid 3 RENAME Test Passed
NODE1 CPU3 Pid 4 CORECTL Test Started
NODE1 CPU3 Pid 4 CORECTL Test Passed
NODE0 CPU1 Pid 2 CORECTL Test Started
NODE0 CPU0 Pid 1 CORECTL Test Started
NODE0 CPU1 Pid 2 CORECTL Test Passed
NODE0 CPU0 Pid 1 CORECTL Test Passed
NODE1 CPU2 Pid 3 CORECTL Test Started
NODE1 CPU2 Pid 3 CORECTL Test Passed
MDK R10000 combined test run is complete.
```

Chapter 3

Secondary Cache Test

This chapter describes the diagnostic test that you can use to test the secondary caches in the system.

3.1 About the Secondary Cache Test

The secondary cache test is a directed test that checks the secondary caches that are located on the Node boards. To run the secondary cache test, you must load the *memory.mdk* test package.

It this test fails, the failing hardware is one or more of the secondary caches (HIMMs) in the system. The failing FRU is the Node board that contains the failing secondary cache.

3.2 How to Run the Secondary Cache Test

Use the following procedure to load the test from the local system disk:

- Use the *boot* command from the PROM monitor (at the BaseIO command prompt, >>) to start the *memory.mdk* test package:

  ```
  >>boot dksc(0,1,0)/stand/memory.mdk
  ```

  MDK displays boot status information and the `MDK>` prompt.

- Load the secondary cache test, *scache*, from the `MDK>` prompt:

  ```
  MDK>scache
  ```

  The secondary cache test loads into the system and begins testing the secondary caches.

3.3 Secondary Cache Test Description

The secondary cache test runs separately in each CPU to enable the test to check each CPU's local secondary cache (located on the HIMM).

The secondary cache test writes data to the secondary cache, ensuring that data is written from the primary cache into the secondary cache. The test then reads the data back and compares the data that was read back with expected data.

The secondary cache test includes three subtests that use different data patterns to test the secondary caches: a data path test, a cell test, and a random address and data test.

3.3.1 Data Path Test

The data path test uses data patterns that cause simultaneous switching on the data lines. This test uses the following algorithm:

- Fill the secondary cache with data vectors that cause simultaneous switching on the secondary cache lines. This test writes data in the following sequence:

```
0x0000 0000 0000 0000 0000 0000 0000 0001
0xffff ffff ffff ffff ffff ffff ffff fffe
0x0000 0000 0000 0000 0000 0000 0000 0002
0xffff ffff ffff ffff ffff ffff ffff fffd
. . .
. . .
0x8000 0000 0000 0000 0000 0000 0000 0000
0x7fff ffff ffff ffff ffff ffff ffff ffff
```

- Read the data back and compare the data that was read back with the data that was written.

3.3.2 Cell Test

There are two parts to the cell test: a data vectors cell test and a random data cell test.

3.3.2.1 Data Vectors Cell Test

The data vectors cell test writes specific data vectors to each memory cell in the secondary cache to test the cache as an array. This test uses the following array of data vectors:

```
0xffffffff, 0xaaaaaaaa, 0xcccccccc, 0xffff0000, 0x00000000, 0x55555555,
0x33333333, 0x0000ffff
```

The test fills the secondary cache with this data, reads the data back, and compares the data that was read back with the data that was written. The test repeats this process for several passes; each pass starts at a different value in the data array.

3.3.2.2 Random Data Cell Test

The random data cell test moves lines of data between the primary cache and secondary cache and between the secondary cache and memory. This test uses random data. The number of tag field bits that the test can manipulate is limited by the size of the user address space that MDK allows or by the amount of physical memory in the system.

3.3.3 Random Address and Data Test

The random address test selects a random address and a random range of address bits to change. The test creates new addresses to use by randomly modifying the address bits in the selected range.

This process creates addresses that may cause store operations to a stride of cache lines within the cache. This process also creates addresses that cause data lines in the cache to be written back to memory.

3.4 Secondary Cache Test Output

The secondary cache test returns output to the BaseIO console.

3.4.1 Pass Output

The secondary cache test prints the following messages if it completes testing without detecting any errors:

```
NODEx CPUy PIDz Scache data path test PASSED
NODEx CPUy PIDz Scache cell test PASSED
NODEx CPUy PIDz Scache random address test PASSED
Secondary cache test has completed
```

3.4.2 Failure Output

The secondary cache test prints the following messages if it detects errors:

```
NODEx CPUy PIDz Scache data path test FAILED
NODEx CPUy PIDz Scache cell test FAILED
NODEx CPUy PIDz Scache random address test FAILED

Virtual addr virtual_address Physical addr physical_address
Expected data expected_data Actual data actual_data
Syndrom syndrome_value
```

If this test fails, the failing hardware is the secondary cache (HIMM), memory, or CPU. The failing FRU is the Node board that contains the failing secondary cache.

3.5 Secondary Cache Test Example

The following example shows the secondary cache test running on a 2-Node system (4 CPUs). In this example, the test does not detect any hardware failures.

```
>>boot dksc(0,1,0)/stand/memory.mdk
Booting Nanos.........
************************************************************
SGI Nanos Version 1.10 SN0 built 02:51:05 PM Mar 25, 1997
************************************************************

CPU 0: Total no. of CPUs = 4
Launched CPU 1
Launched CPU 2
Launched CPU 3
MDK>scache
Created successfully pid = 1
CPU 0: User mem starts from 0xa800000000641000
NODE0 CPU0 PID1 In1B 000:  main
NODE0 CPU1 PID2 In child
NODE0 CPU0 PID1 Scache data pat1B 000: h test
NODE0 CPU1 PID2 Scache data path test
CPU0: sCDataBusTest: Loop 00000000
CDataBusTest: Loop 00000000
NODE1 CPU3 PID4 2A 001: In child
NODE1 CPU2 PID3 In child
NODE1 CPU3 PID4 Scache data path test
NODE1 CPU2 PID3 Scache data path test
CPU3: sCDataBusTest: Loop 00000000
CDataBusTest: Loop 00000000
CPU0: sCDataBusTest: Loop 00000500
CPU2: sCDataBusTest: Loop 00000500
CPU3: sCDataBusTest: Loop 00000500
CPU1: sCDataBusTest: Loop 00000500
NODE0 CPU0 PID1 Scache data path test PASSED
NODE0 CPU0 PID1 Scache cell test
NODE0 CPU0 PID1 sCCellRandTest Loop 00000000
NODE1 CPU2 PID3 Scache data path test PASSED
NODE1 CPU2 PID3 Scache cell test
NODE1 CPU2 PID3 sCCellRandTest Loop 00000000
NODE1 CPU3 PID4 Scache data path test PASSED
NODE1 CPU3 PID4 Scache cell test
NODE1 CPU3 PID4 sCCellRandTest Loop 00000000
NODE0 CPU1 PID2 Scache data path test PASSED
NODE0 CPU1 PID2 Scache cell test
NODE0 CPU1 PID2 sCCellRandTest Loop 00000000
NODE0 CPU0 PID1 sCCellRandTest Loop 00000500
NODE1 CPU2 PID3 sCCellRandTest Loop 00000500
NODE1 CPU3 PID4 sCCellRandTest Loop 00000500
NODE0 CPU1 PID2 sCCellRandTest Loop 00000500
NODE0 CPU0 PID1 Scache cell test PASSED
NODE0 CPU0 PID1 Scache random address test
CPU0: sCRandAddrTest: Loop 00000000
CPU0: sCRandAddrTest: Loop 00000500
NODE0 CPU0 PID1 Scache random address test PASSED
NODE0 CPU0 PID1 PASSED
NODE1 CPU2 PID3 Scache cell test PASSED
```

```
NODE1 CPU2 PID3 Scache random address test
CPU2: sCRandAddrTest: Loop 00000000
CPU2: sCRandAddrTest: Loop 00000500
NODE1 CPU2 PID3 Scache random address test PASSED
NODE1 CPU2 PID3 child PASSED
NODE1 CPU3 PID4 Scache cell test PASSED
NODE1 CPU3 PID4 Scache random address test
CPU3: sCRandAddrTest: Loop 00000000
CPU3: sCRandAddrTest: Loop 00000500
NODE1 CPU3 PID4 Scache random address test PASSED
NODE1 CPU3 PID4 child PASSED
NODE0 CPU1 PID2 Scache cell test PASSED
NODE0 CPU1 PID2 Scache random address test
CPU1: sCRandAddrTest: Loop 00000000
CPU1: sCRandAddrTest: Loop 00000500
NODE0 CPU1 PID2 Scache random address test PASSED
NODE0 CPU1 PID2 child PASSED
**** Secondary cache test status summary ****
CNODE0 NASID0 CPU0 PID1 PASSED
CNODE0 NASID0 CPU1 PID2 PASSED
CNODE1 NASID1 CPU2 PID3 PASSED
CNODE1 NASID1 CPU3 PID4 PASSED
Secondary cache test has completed
```

Chapter 4

Memory Tests

This chapter describes the diagnostics that you can use to test memory.

4.1 About the Memory Tests

The memory tests are directed tests that verify that memory properly stores data that is written to it. Four different types of memory tests are available: quick screen memory tests, Node board memory tests, internode memory tests, and long memory tests. There are mapped and unmapped versions of most of these tests.

To run the memory tests, you must load the *memory.mdk* test package.

Table 4-1 provides quick-reference descriptions of the memory tests. The number in the Page column indicates the page on which a detailed description of each test begins.

Table 4-1 Memory Tests

Type	Name	Description	Page
Quick screen memory tests	*mem.qs*	Writes data patterns to memory, reads the data back, and verifies the results (mapped version)	4-7
	memum.qs	Writes data patterns to memory, reads the data back, and verifies the results (unmapped version)	4-7
Node board memory test	*memum.nb*	Uses one or two CPUs to write two different sets of data to the cache lines, reads the data back, and verifies the results (unmapped version)	4-16
Internode memory test	*memum.in*	Uses all Nodes and CPUs in the system to write data to the cache lines, reads the data back, and verifies the results (unmapped version)	4-21
Long memory tests	*mem.lo*	Runs extended versions of the *mem.qs*, *mem.nb*, and *mem.in* tests (mapped version)	4-27
	memum.lo	Runs extended versions of the *memum.qs*, *memum.nb*, and *memum.in* tests (unmapped version)	4-27

If any of the memory tests fails, the failing hardware could be a memory DIMM, a secondary cache (a HIMM), a Node board, or a Router board. The failing FRU could be the memory DIMM, Node board, or Router board that contains the failing hardware component.

If the memory tests detect failing DIMMs, they print out the location of each DIMM (for example, MMXL0, MMXH0, or MMYH7). For the main memory DIMM slots, use the last two digits (either an L and a number or an H and a number) to determine which FRU to replace. Figure 4-1 shows the location of the DIMM memory slots on the Node board.

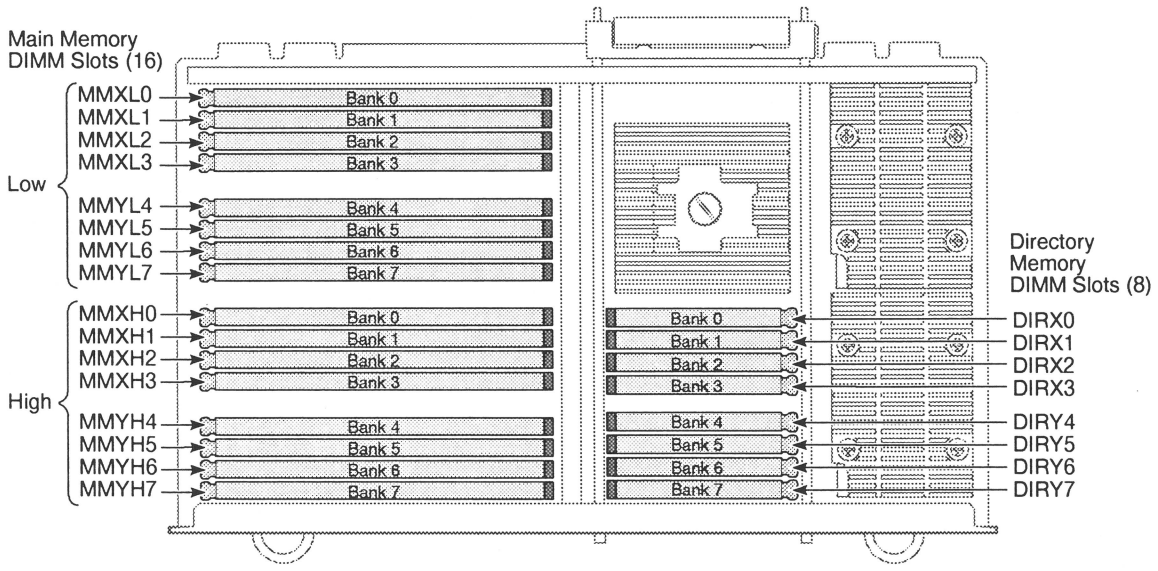

Figure 4-1 Memory DIMM Locations

4.2 Common Error Output

The memory tests print the same messages for several types of errors, including when the tests detect exceptions, when the tests find data miscompares, and when unmapped memory tests encounter single-bit ECC errors. The examples in Section 4.2.1, Section 4.2.2, and Section 4.2.3 show the output that the tests return for these errors.

4.2.1 Cache Error Exception Example

The following example shows a portion of the output from an unmapped or mapped memory test that detects an exception. In this example, the exception is a cache error exception.

```
Status: 0xffffffffb40080e5        XCtxt: 0xffffffe088888880
Hi:     0x                0       Lo:    0x                36
epc:    0xa800000000a07e28        cause: 0x              4020
count:  0x            60a3be        comp:  0x          c800000
Enhi:   0x                0       CacErr: 0xffffffffd879e100
R01/AT: 0x                1       R02/V0: 0xa8000000026bdf80
R03/V1: 0xa800000001f9e108        R04/A0: 0xa800000001f9c988
R05/A1: 0x                0       R06/A2: 0x                0
R07/A3: 0x          34a9ff8       R08/A4: 0x                0
R09/A5: 0xa800000000b14000        R10/A6: 0xa800000000b14000
R11/A7: 0x                1       R12/T0: 0xa8000000026bdf80
R13/T1: 0xa8000000026bdf80        R14/T2: 0x                1
R15/T3: 0x                0       R16/S0: 0x                0
R17/S1: 0x                0       R18/S2: 0x                0
R19/S3: 0x                0       R20/S4: 0x                0
R21/S5: 0x                0       R22/S6: 0x                0
R23/S7: 0x                0       R24/T8: 0x                0
R25/T9: 0x7777777711111111        R28/GP: 0x                0
R29/SP: 0xa800000003fde688        R30/S8: 0x                0
R31/RA: 0xa800000001f9e108
Nasid 0: Local CPU A: Global CPU 0: PID 1: Cac Err exception at
0xa800000000a05640: sr = 0xb40080e5
Nanos: Fr
MSC> nmi
        ok

*** NMI while in Kernel and no NMI vector installed on node 0
*** NMI while in Kernel and no NMI vector installed on node 0
*** Error EPC: 0xa800000000a008cc (0xa800000000a008cc)
*** Error EPC: 0xa800000000025354 (0xa800000000025354)
*** Press ENTER to continue.
*** Press ENTER to continue.
POD MSC Dex MDerr> error
Uncorrectable memory ECC error, with overrun
   Address       : 0x1f9e100
   Bad syndrome  : 0xc6 (multi)
   Physical loc  : MMXH0 and/or MMXL0
POD MSC Dex MDerr> pr MD_MEM_ERROR
Register: MD_MEM_ERROR (0x9200000001200070)
Value   : 0xc000007e01f20a81 (loaded from register)
   <63> R   UCE_VALID                0x1
```

```
   <62> R   CE_VALID                  0x1
<39:32> R   BAD_SYN                   0x7e
<31:03> R   ADDRESS<31:03>            0x003f3c20 << 3 = 0x01f9e100
   <01> R   UCE_OVERRUN               0x0
   <00> R   CE_OVERRUN                0x1
POD MSC Dex MDerr> ld u:0x1f9e100
9600000001f9e100 75757777111111111111
```

In this example, the following actions were taken to get more information about the exception:

- A nonmaskable interrupt was issued from the `MSC>` prompt after the exception occurred to bring the system into POD mode.
- The POD *error* command was issued to get more information about memory or directory errors (from the MD section of the HUB) because the `MDerr` text in the POD prompt indicates that there are MD errors pending.
- The POD *pr MD_MEM_ERROR* command was issued to print the contents of the memory/directory memory error register. (This register shows more detail about the ECC error.)

  Note: Refer to *Origin2000 Power-On Diagnostics*, SGI part number 108-0161-001, for more information about the registers that you can print with the *pr* command.
- The POD *ld u:0x1f9e100* command was issued to view the contents of the memory address that the *pr* command reported. (This shows the data miscompare: The memory address contains 0x75757777111111111111 when it should contain 0x77777777111111111111.)

To summarize, the cache error exception was caused by an uncorrectable ECC memory error at address 0x1f9e100; the failing DIMM(s) are in location MMXH0 and/or MMXL0 (refer to the `Physical loc` field of the *error* command output).

4.2.2 Data Miscompare Example

The following example shows a portion of the output from an unmapped or mapped memory test that detects a data miscompare:

```
Node 0 Bank 0 Physical addr 0xa800000002016838
Expected data 0x7777777712345678 Actual data 0x7777777711111111
Syndrom 0x0000000003254769
Address 0000000002016838 bit 0 is on NASID 0
MMXL0 DIMML line 18
Address 0000000002016838 bit 3 is on NASID 0
MMXL0 DIMML line 21
Address 0000000002016838 bit 5 is on NASID 0
MMXL0 DIMML line 23
Address 0000000002016838 bit 6 is on NASID 0
MMXL0 DIMML line 24
Address 0000000002016838 bit 8 is on NASID 0
MMXL0 DIMML line 26
Address 0000000002016838 bit 9 is on NASID 0
MMXL0 DIMML line 27
Address 0000000002016838 bit 10 is on NASID 0
MMXL0 DIMML line 28
Address 0000000002016838 bit 14 is on NASID 0
MMXL0 DIMML line 32
Address 0000000002016838 bit 16 is on NASID 0
MMXL0 DIMML line 34
Address 0000000002016838 bit 18 is on NASID 0
MMXL0 DIMML line 54
Address 0000000002016838 bit 21 is on NASID 0
MMXL0 DIMML line 57
Address 0000000002016838 bit 24 is on NASID 0
MMXL0 DIMML line 60
Address 0000000002016838 bit 25 is on NASID 0
MMXL0 DIMML line 61
**** MDK memory unmapped test status summary ****
CNODE0 NASID0 CPU0 PID1 Mem test FAILED
CNODE0 NASID0 CPU1 PID2 Mem test PASSED
ERROR Memory unmapped test failed
```

The output indicates that the failing DIMM is located in DIMM slot MMXL0. (Refer to Table 4-1 on page 4-2 for an illustration of the DIMM slot locations on a Node board.)

4.2.3 Single-Bit ECC Error Example (Unmapped Memory Tests Only)

The following example shows the output when an unmapped memory test encounters a single-bit ECC error:

```
ECC_ERR: Correctable ecc err detected
Nasid 2 bank 0x0 address 0xa800000201d6a030
MD_MEM_ERR    0x4000001501d6a030
UCE_VLD       0x0
CE_VLD        0x1
RSVD1         0x0
BAD_SYN       0x15
ADDRESS       0x3ad406 << 3 0x1d6a030
RSVD2         0x0
UCE_OVR       0x0
CE_OVR        0x0
```

When testing is completed, the unmapped memory test prints a summary of all single-bit ECC errors that occurred:

```
################ DIR SBECC SUMMARY ################
No directory ECC errs detected


################ MEM SBECC SUMMARY ################
Module 1 Slot 1 Address 0xa800000001fdb238
Encountered MULTIPLE SBECC errs
SBECC err reproducable - YES
Address 0xa800000001fdb238 bit 26 is on NASID 0
MMXL0 == DIMM BANK 0 L - line 62
syndrome 0x4c == data_18
-----------------------------------------
Module 1 Slot 3 Address 0xa800000201ea7038
Encountered MULTIPLE SBECC errs
SBECC err reproducable - NO
Address 0xa800000201ea7038 bit 26 is on NASID 2
MMXL0 == DIMM BANK 0 L - line 62
syndrome 0x4c == data_18
-----------------------------------------

**** MDK memory unmapped test status summary ****
CNODE0 NASID0 CPU0 PID1 Mem single bit ECC ERROR - Mem test FAILED
CNODE0 NASID0 CPU1 PID2 Mem single bit ECC ERROR - Mem test FAILED
CNODE1 NASID1 CPU2 PID3 Mem test PASSED
CNODE1 NASID1 CPU3 PID4 Mem test PASSED
CNODE2 NASID3 CPU4 PID5 Mem test PASSED
CNODE2 NASID3 CPU5 PID6 Mem test PASSED
CNODE3 NASID2 CPU6 PID7 Mem single bit ECC ERROR - Mem test FAILED
CNODE3 NASID2 CPU7 PID8 Mem single bit ECC ERROR - Mem test FAILED
Memory unmapped test has completed
```

4.3 Quick Screen Memory Tests

The quick screen memory tests write data patterns to memory, read the data back, and compare the data that was read back with the data that was written. The quick screen memory tests are *mem.qs* (mapped version) and *memum.qs* (unmapped version).

4.3.1 How to Run the Quick Screen Memory Tests

Use the following procedure to load the quick screen memory tests from the local system disk:

- Use the *boot* command from the PROM monitor (at the BaseIO command prompt, >>) to start the *memory.mdk* test package:

  ```
  >>boot dksc(0,1,0)/stand/memory.mdk
  ```

  MDK displays boot status information and the `MDK>` prompt.

- Load the mapped version (*mem.qs*) or unmapped version (*memum.qs*) from the `MDK>` prompt:

  1. To load the mapped version, enter *mem.qs* at the `MDK>` prompt:

     ```
     MDK>mem.qs
     ```

  2. To load the unmapped version, enter *memum.qs* at the `MDK>` prompt:

     ```
     MDK>memum.qs
     ```

  The specified quick screen memory test loads into the system and begins testing memory.

4.3.2 Test Description

Both quick screen memory tests use the following algorithm:

- Acquire the system configuration:
  1. Determine the amount of memory that is populated in the system.
  2. Determine the number of CPUs and Nodes in the system.
  3. Scale the test to the system configuration.
- Select one CPU to perform only read operations: This CPU causes memory traffic while the test is running.
- Select one CPU to perform read and write operations: This CPU does the actual memory testing.

Note: If there is only one CPU in the system, it runs two processes. One of the processes performs only read operations; the other process performs read and write operations.

- Perform a memory pattern test:
  1. Repeat the following actions until all of the memory being tested is filled with the data patterns:
     - Write 0xc33cc33cc33cc33c to a cache line.
     - Write the complement data pattern (0x3cc33cc33cc33cc3) to the next cache line.
  2. Read the data back and compare it to expected values.
  3. Repeat the following actions until all of the memory being tested is filled with the data patterns:
     - Write 0xffffffffffffffff to a cache line.
     - Write the complement data pattern (0x0000000000000000) to the next cache line.
  4. Read the data back and compare it to expected values.
- Perform a simultaneous switching (SSO) over cache line test:
  1. Fill the available memory with data vectors that cause simultaneous switching on the secondary cache lines. This test writes data in the following sequence:

     0x0000 0000 0000 0000 0000 0000 0000 0001
     0xffff ffff ffff ffff ffff ffff ffff fffe
     0x0000 0000 0000 0000 0000 0000 0000 0002
     0xffff ffff ffff ffff ffff ffff ffff fffd
     . . .
     . . .
     0x8000 0000 0000 0000 0000 0000 0000 0000
     0x7fff ffff ffff ffff ffff ffff ffff ffff
  2. Read the data back and compare the data that was read back with the data that was written.
- Perform a data bus test:
  1. Write data to memory addresses that are generated using the following information:
     - Node ID
     - DIMM bank ID
     - physical bank ID
     - logical bank ID
  2. Read the data back and compare it to expected values.
- Perform a background test:
  1. Write all memory being tested with all 0's.
  2. Read back one doubleword at a time and compare it to the expected value of 0.
  3. Write the complement of 0 to the doubleword location that was just read.
  4. Repeat Steps 2 and 3 until all available memory has been used.
  5. Read back one doubleword of data at a time and compare it to the expected value of the complement of 0.

6. Write 0 to the doubleword location that was just read.
7. Repeat the steps 5 and 6 until all available memory has been used.

- Perform an address bus test:
  1. Write memory with the following data patterns: 0xc33cc33cc33cc33c and 0x3cc33cc33cc33cc3.
  2. Read the data back and compare it to expected values.
- Repeat the following actions several times to perform a random data test:
  1. Write all memory being tested with a random data pattern.
  2. Readthe data back and compare it to expected values.

4.3.3 Output From the Quick Screen Memory Tests

The quick screen memory tests return output to the BaseIO console.

4.3.3.1 Pass Output

As the tests run, they print out `PASSED` messages for each test passing section. The following examples show some of these messages:

```
NODE0 CPU0 PID1 memory pattern "0xc33cc33cc33cc33c 0x3cc33cc33cc33cc3" test PASSED
NODE1 CPU2 PID3 memory pattern "0xc33cc33cc33cc33c 0x3cc33cc33cc33cc3" test PASSED
NODE0 CPU0 PID1 memory pattern "0xffffffffffffffff 0x0000000000000000" test PASSED
NODE1 CPU2 PID3 memory pattern "0xffffffffffffffff 0x0000000000000000" test PASSED
NODE0 CPU0 PID1 SSO over cache line test PASSED
NODE1 CPU2 PID3 SSO over cache line test PASSED
NODE1 CPU2 PID3 memory data bus test PASSED
NODE0 CPU0 PID1 memory data bus test PASSED
NODE0 CPU0 PID1 memory background test PASSED
NODE1 CPU2 PID3 memory background test PASSED
NODE0 CPU0 PID1 memory address bus test PASSED
NODE1 CPU2 PID3 memory address bus test PASSED
NODE0 CPU0 PID1 random data test PASSED
NODE1 CPU2 PID3 random data test PASSED
```

When all test sections have completed successfully, the tests print out a summary for each CPU in the system, as shown in the following example messages:

```
**** MDK memory test status summary ****
CNODE0 NASID0 CPU0 PID1 Mem test PASSED
CNODE0 NASID0 CPU1 PID2 Mem test PASSED
CNODE1 NASID1 CPU2 PID3 Mem test PASSED
CNODE1 NASID1 CPU3 PID4 Mem test PASSED
MDK memory mapped test has completed
```

Note: The last line is for the *mem.qs* test. It changes to `Memory unmapped test has completed` for the *memum.qs* test.

4.3.3.2 Failure Output

If these tests detect errors, they print an error message and related output, similar to the following example messages:

```
ERROR NODEx CPUy PIDx testname FAILED
Data type type
Virtual addr 0x000000000cdb1a07  Physical addr 0xa800000181f05a07
Expected data 0xf9  Actual data 0x79
Syndrom 0x80
Address 0000000081f05a07 bit 7 is on NASID 1
MMYH4 DIMML line 7
```

If either of the quick screen memory tests fails, the failing hardware could be a memory DIMM, a secondary cache (a HIMM), a Node board, a midplane, a cable, or a Router board. The failing FRU could be the memory DIMM, Node board, midplane, cable, or Router board that contains the failing hardware component.

4.3.4 Examples of Running the Quick Screen Memory Tests

This section contains examples of running the *mem.qs* and *memum.qs* quick screen memory tests.

4.3.4.1 Example of Running the *mem.qs* Test

The following example output shows the *mem.qs* test running on a 2-Node system (4 CPUs). In this example, the test completes successfully without detecting any hardware failures.

```
>>boot dksc(0,1,0)/stand/memory.mdk
Booting Nanos.........
**********************************************************
SGI Nanos Version 1.10 SN0 built 02:53:28 PM Mar 25, 1997
**********************************************************

CPU 0: Total no. of CPUs = 4
Launched CPU 1
Launched CPU 2
Launched CPU 3
MDK>mem.qs
Created successfully pid = 1
CPU 0: User mem starts from 0xa800000000641000
CPU0 PID1 Allocating 0x000000000705f000 bytes of memory on NODE0
NODE0 CPU0 PID1 Trying to malloc 0x000000000705f000 bytes
NODE0 CPU0 PID1 malloc SUCCESSFUL for 0x000000000705f000 bytes
vmem_addri 0x000000000060a000 vmem_addrf 0x0000000007669000 size 0x000000000705f000
CPU0 PID1 Allocating 0x0000000007c80000 bytes of memory on NODE1
NODE1 CPU0 PID1 Trying to malloc 0x0000000007c80000 bytes
NODE1 CPU0 PID1 malloc SUCCESSFUL for 0x0000000007c80000 bytes
vmem_addri 0x0000000007669000 vmem_addrf 0x000000000f2e9000 size
0x0000000007c80000
CNODE0 NASID0 CPU0 ppid1
SCACHE_CONFIG_T:
     sc_size     1
```

```
    sc_size_max 6
    sc_bsize    1
    qwo[sbit]   00      qwo[size]   04      qwo[mask]   0x0000000f
    qw[sbit]    04      qw[size]    02      qw[mask]    0x00000003
    index[sbit] 06      index[size] 13      index[mask] 0x00001fff
    data[sbit]  04      data[size]  15      data[mask]  0x00007fff
    tag[sbit]   19      tag[size]   21      tag[mask]   0x001fffff
    way[sbit]   07      way[size]   13      way[mask]   0x00000fff
Config_t:
    numCpus    4
    numNodes   2
    numRouters 1
        CNODE0 NASID0:
            numCpus  2
            mem size 0x0000000008000000
        CNODE1 NASID1:
            numCpus  2
            mem size 0x0000000008000000
NODE_MEM_PTR_T:
        CNODE0 NASID0:
        vmem_addri 0x000000000060a000
        vmem_addrf 0x0000000007669000
        pmem_addri 0xa800000000664000
        pmem_addrf 0xa8000000076c3000
        size       0x000000000705f000
        CNODE1 NASID1:
        vmem_addri 0x0000000007669000
        vmem_addrf 0x000000000f2e9000
        pmem_addri 0xa800000100020000
        pmem_addrf 0xa800000107ca0000
        size       0x0000000007c80000
PROCESS_T:
    Cpu0:
        type   0  READ_WRITE
        status 0  IDLE
    Cpu1:
        type   1  READ_ONLY
        status 0  IDLE
    Cpu2:
        type   0  READ_WRITE
        status 0  IDLE
    Cpu3:
        type   1  READ_ONLY
        status 0  IDLE
CPU0 PID1 Sproc on CPU1
CPU0 PID1 Sproc on CPU2
NODE0 CPU1 PID2 In child
CPU0 PID1 Sproc on CPU3
NODE0 CPU1 PID2 Read only test
NODE1 CPU3 PID4 In child
NODE1 CPU2 PID3 In child
NODE1 CPU3 PID4 Read only test
NODE0 CPU0 PID1 memory pattern "0xc33cc33cc33cc33c 0x3cc33cc33cc33cc3" test PASSED
NODE1 CPU2 PID3 memory pattern "0xc33cc33cc33cc33c 0x3cc33cc33cc33cc3" test PASSED
NODE0 CPU0 PID1 memory pattern "0xffffffffffffffff 0x0000000000000000" test PASSED
NODE1 CPU2 PID3 memory pattern "0xffffffffffffffff 0x0000000000000000" test PASSED
NODE0 CPU0 PID1 SSO over cache line test PASSED
```

```
NODE1 CPU2 PID3 SSO over cache line test PASSED
NODE0 CPU1 PID2 Read only test
NODE0 CPU0 PID1 memory data bus test
NODE1 CPU2 PID3 memory data bus test
NODE1 CPU3 PID4 Read only test
NODE1 CPU2 PID3 memory data bus test PASSED
NODE0 CPU0 PID1 memory data bus test PASSED
NODE1 CPU2 PID3 memory background test
NODE0 CPU0 PID1 memory background test
NODE0 CPU0 PID1 memory background test PASSED
NODE0 CPU0 PID1 memory address bus test
NODE0 CPU0 PID1 memory address bus test "Ulong_nec" index 0
NODE1 CPU2 PID3 memory background test PASSED
NODE1 CPU2 PID3 memory address bus test
NODE1 CPU2 PID3 memory address bus test "Ulong_nec" index 0
NODE0 CPU0 PID1 memory address bus test "Ulong_nec" index 1
NODE1 CPU2 PID3 memory address bus test "Ulong_nec" index 1
NODE0 CPU0 PID1 memory address bus test "Ulong_NEC" index 0
NODE1 CPU2 PID3 memory address bus test "Ulong_NEC" index 0
NODE0 CPU0 PID1 memory address bus test "Ulong_NEC" index 1
NODE1 CPU2 PID3 memory address bus test "Ulong_NEC" index 1
NODE0 CPU0 PID1 memory address bus test "Ulong_NEC" index 2
NODE1 CPU2 PID3 memory address bus test "Ulong_NEC" index 2
NODE0 CPU0 PID1 memory address bus test "Ulong_NEC" index 3
NODE1 CPU2 PID3 memory address bus test "Ulong_NEC" index 3
NODE0 CPU0 PID1 memory address bus test "Ulong_NEC" index 4
NODE1 CPU2 PID3 memory address bus test "Ulong_NEC" index 4
NODE0 CPU0 PID1 memory address bus test "Ulong_NEC" index 5
NODE1 CPU2 PID3 memory address bus test "Ulong_NEC" index 5
NODE0 CPU0 PID1 memory address bus test "Ulong_NEC" index 6
NODE1 CPU2 PID3 memory address bus test "Ulong_NEC" index 6
NODE0 CPU0 PID1 memory address bus test "Ulong_NEC" index 7
NODE0 CPU0 PID1 memory address bus test PASSED
NODE0 CPU0 PID1 random data test
NODE0 CPU0 PID1 Loop 0 data type 3
NODE1 CPU2 PID3 memory address bus test "Ulong_NEC" index 7
NODE1 CPU2 PID3 memory address bus test PASSED
NODE1 CPU2 PID3 random data test
NODE1 CPU2 PID3 Loop 0 data type 3
NODE0 CPU0 PID1 random data test PASSED
NODE1 CPU2 PID3 random data test PASSED
**** MDK memory test status summary ****
CNODE0 NASID0 CPU0 PID1 Mem test PASSED
CNODE0 NASID0 CPU1 PID2 Mem test PASSED
CNODE1 NASID1 CPU2 PID3 Mem test PASSED
CNODE1 NASID1 CPU3 PID4 Mem test PASSED
MDK memory mapped test has completed
```

4.3.4.2 Example of Running the *memum.qs* Test

The following example output shows the *memum.qs* test running on a 2-Node system (4 CPUs). In this example, the test completes successfully without detecting any hardware failures.

```
>>boot dksc(0,1,0)/stand/memory.mdk
Booting Nanos.........
************************************************************
SGI Nanos Version 1.10 SN0 built 02:53:28 PM Mar 25, 1997
************************************************************

CPU 0: Total no. of CPUs = 4
Launched CPU 1
Launched CPU 2
Launched CPU 3
Created successfully pid = 1
MDK>memum.qs
CPU 0: User mem starts from 0xa800000000641000
CPU0 PID1 Allocating 0x000000000705f000 bytes of memory on NODE0
NODE0 CPU0 PID1 Trying to malloc 0x000000000705f000 bytes
NODE0 CPU0 PID1 malloc SUCCESSFUL for 0x000000000705f000 bytes
vmem_addri 0x000000000060a000 vmem_addrf 0x0000000007669000 size 0x000000000705f000
CPU0 PID1 Allocating 0x0000000007c80000 bytes of memory on NODE1
NODE1 CPU0 PID1 Trying to malloc 0x0000000007c80000 bytes
NODE1 CPU0 PID1 malloc SUCCESSFUL for 0x0000000007c80000 bytes
vmem_addri 0x0000000007669000 vmem_addrf 0x000000000f2e9000 size
0x0000000007c80000
CNODE0 NASID0 CPU0 ppid1
Config_t:
   numCpus    4
   numNodes   2
   numRouters 1
   CNODE000 NASID000:
     numCpus  2
     mem size 0x0000000008000000
     Bank0    0x0000000008000000
     Bank1    0x0000000000000000
     Bank2    0x0000000000000000
     Bank3    0x0000000000000000
     Bank4    0x0000000000000000
     Bank5    0x0000000000000000
     Bank6    0x0000000000000000
     Bank7    0x0000000000000000
   CNODE001 NASID001:
     numCpus  2
     mem size 0x0000000008000000
     Bank0    0x0000000008000000
     Bank1    0x0000000000000000
     Bank2    0x0000000000000000
     Bank3    0x0000000000000000
     Bank4    0x0000000000000000
     Bank5    0x0000000000000000
     Bank6    0x0000000000000000
     Bank7    0x0000000000000000
INFO_T addr of ppid      0xa800000007fde788
INFO_T addr of sysconfig 0xa800000007fde790
```

```
INFO_T addr of procinfo  0xa800000000b11000
INFO_T addr of nodeinfo  0xa800000000b12000
INFO_T addr of addrinfo  0xa800000000b13000
NODE000 BANK0 mem   0x0000000008000000
NODE000 BANK0 saddr 0xa800000000b14000
NODE000 BANK0 eaddr 0xa800000007fbdff8
NODE000 BANK0 size  0x00000000074a9ff8
```

[configuration information for Node 0, banks 1 through 6 (not shown)]

```
NODE000 BANK7 mem   0x0000000000000000
NODE000 BANK7 saddr 0x0000000000000000
NODE000 BANK7 eaddr 0x0000000000000000
NODE000 BANK7 size  0x0000000000000000
NODE001 BANK0 mem   0x0000000008000000
NODE001 BANK0 saddr 0xa800000100020000
NODE001 BANK0 eaddr 0xa800000107fbdff8
NODE001 BANK0 size  0x0000000007f9dff8
```

[configuration information for Node 1, banks 1 through 6 (not shown)]

```
NODE001 BANK7 mem   0x0000000000000000
NODE001 BANK7 saddr 0x0000000000000000
NODE001 BANK7 eaddr 0x0000000000000000
NODE001 BANK7 size  0x0000000000000000
CPU0 Type 0 READ_WRITE Status 0 IDLE
CPU1 Type 1 READ_ONLY Status 0 IDLE
CPU2 Type 0 READ_WRITE Status 0 IDLE
CPU3 Type 1 READ_ONLY S1B 000: tatus 0 IDLE
NODE0 CPU1 PID2 In child
NODE0 CPU0 PID1 unmapped memory phase zero test
NODE0 CPU1 PID2 unmapped memory phase zero test
NODE0 CPU0 PID1 unmapped memory pattern test
NODE0 CPU1 PID2 Read only test
NODE1 CPU2 PID3 In child
NODE1 CPU3 PID4 In child
NODE1 CPU2 PID3 unmapped memory phase zero test
NODE1 CPU3 PID4 unmapped memory phase zero test
NODE1 CPU2 PID3 unmapped memory pattern test
NODE1 CPU3 PID4 Read only test
NODE0 CPU0 PID1 memory pattern "0xc33cc33cc33cc33c 0x3cc33cc33cc33cc3" test PASSED
NODE0 CPU0 PID1 unmapped memory pattern test
NODE1 CPU2 PID3 memory pattern "0xc33cc33cc33cc33c 0x3cc33cc33cc33cc3" test PASSED
NODE1 CPU2 PID3 unmapped memory pattern test
NODE0 CPU0 PID1 memory pattern "0xffffffffffffffff 0x0000000000000000" test PASSED
NODE0 CPU0 PID1 unmapped memory SSO over cahe line test
NODE1 CPU2 PID3 memory pattern "0xffffffffffffffff 0x0000000000000000" test PASSED
NODE1 CPU2 PID3 unmapped memory SSO over cahe line test
NODE0 CPU0 PID1 SSO over cache line test PASSED
NODE0 CPU0 PID1 unmapped memory phase zero test PASSED
NODE1 CPU2 PID3 SSO over cache line test PASSED
NODE1 CPU2 PID3 unmapped memory phase zero test PASSED
NODE0 CPU1 PID2 unmapped memory phase zero test PASSED
NODE1 CPU3 PID4 unmapped memory phase zero test PASSED
NODE0 CPU1 PID2 unmapped memory phase one test
NODE0 CPU0 PID1 unmapped memory phase one test
NODE0 CPU1 PID2 Read only test
```

```
NODE0 CPU0 PID1 unmapped memory background test
NODE1 CPU3 PID4 unmapped memory phase one test
NODE1 CPU2 PID3 unmapped memory phase one test
NODE1 CPU3 PID4 Read only test
NODE1 CPU2 PID3 unmapped memory background test
NODE0 CPU0 PID1 unmapped memory background test PASSED
NODE0 CPU0 PID1 unmapped memory cell test
NODE0 CPU0 PID1 unmapped memory cell test Ulong_NEC bank 0
NODE1 CPU2 PID3 unmapped memory background test PASSED
NODE1 CPU2 PID3 unmapped memory cell test
NODE1 CPU2 PID3 unmapped memory cell test Ulong_NEC bank 0
NODE0 CPU0 PID1 unmapped memory cell test PASSED
NODE0 CPU0 PID1 unmapped memory phase one test PASSED
NODE1 CPU2 PID3 unmapped memory cell test PASSED
NODE1 CPU2 PID3 unmapped memory phase one test PASSED
NODE0 CPU1 PID2 unmapped memory phase one test PASSED
NODE1 CPU3 PID4 unmapped memory phase one test PASSED
**** MDK memory unmapped test status summary ****
CNODE0 NASID0 CPU0 PID1 Mem test PASSED
CNODE0 NASID0 CPU1 PID2 Mem test PASSED
CNODE1 NASID1 CPU2 PID3 Mem test PASSED
CNODE1 NASID1 CPU3 PID4 Mem test PASSED
Memory unmapped test has completed
```

4.4 Node Board Memory Test

The Node board memory test uses cache line sharing without internode access. This enables it to test cache coherence and to perform cache thrashing. This test does not generate internode traffic. There is only an unmapped version of the Node board memory test (*memum.nb*).

4.4.1 How to Run the Node Board Memory Test

Use the following procedure to load the test from the local system disk:

- Use the *boot* command from the PROM monitor (at the BaseIO command prompt, >>) to start the *memory.mdk* test package:

  ```
  >>boot dksc(0,1,0)/stand/memory.mdk
  ```

  MDK displays boot status information and the MDK> prompt.

- Load the *memum.nb* test from the MDK> prompt:

  ```
  MDK>memum.nb
  ```

  The Node board memory test loads into the system and begins testing memory.

4.4.2 Test Description

The Node board memory test uses the following algorithm:

- Acquire the system configuration.
  1. Determine the amount of memory that is populated in the system.
  2. Determine the number of CPUs and Nodes in the system.
  3. Scale the test to the system configuration.
- Select one CPU to perform only read operations: This CPU causes memory traffic while the test is running.
- Select one CPU to perform read and write operations: This CPU does the actual memory testing.

Note: If there is only one CPU in the system, it runs two processes. One of the processes performs only read operations; the other process performs read and write operations.

- Generate random addresses and data that the test will use.
- Split each cache line in half: CPU0 uses the lower 64 bytes of the cache line, and CPU1 uses the upper 64 bytes of the cache line.
- Have each CPU write data to its half of each cache line.
- Have each CPU read the data back from its half of the cache lines.
- Compare the data that was read back to the data that was written and verify that both sets of data are the same.

4.4.3 Output From the Node Board Memory Test

The Node board memory test returns output to the BaseIO console.

4.4.3.1 Pass Output

As the test runs, it prints out a `PASSED` message for each test section that completes without detecting hardware failures. The following examples show some of these messages:

```
NODE1 CPU3 PID4 unmapped memory random address and data test PASSED
NODE1 CPU2 PID3 unmapped memory random address and data test PASSED
NODE0 CPU0 PID1 unmapped memory random address and data test PASSED
NODE0 CPU1 PID2 unmapped memory random address and data test PASSED
```

When all test sections have completed successfully, the test prints out a summary for each CPU in the system, as shown in the following example messages:

```
**** MDK memory unmapped test status summary ****
CNODE0 NASID0 CPU0 PID1 Mem test PASSED
CNODE0 NASID0 CPU1 PID2 Mem test PASSED
CNODE1 NASID1 CPU2 PID3 Mem test PASSED
CNODE1 NASID1 CPU3 PID4 Mem test PASSED
Memory unmapped test has completed
```

4.4.3.2 Failure Output

If this test detects errors, it prints an error message and related output, similar to the following example messages:

```
ERROR NODEx CPUy PIDx testname FAILED
Data type type
Virtual addr 0x000000000cdb1a07  Physical addr 0xa800000181f05a07
Expected data 0xf9  Actual data 0x79
Syndrom 0x80
Address 0000000081f05a07 bit 7 is on NASID 1
MMYH4 DIMML line 7
```

If the Node board memory test fails, the failing hardware could be a memory DIMM, a secondary cache (a HIMM), or a Node board. The failing FRU could be the memory DIMM, Node board, or Router board that contains the failing hardware component.

4.4.4 Example of Running the Node Board Memory Test

The following example output shows the Node board memory test running on a 2-Node system (4 CPUs). In this example, the test completes successfully without detecting any hardware failures.

```
>>boot dksc(0,1,0)/stand/memory.mdk
Booting Nanos.........
************************************************************
SGI Nanos Version 1.10 SN0 built 02:53:28 PM Mar 25, 1997
************************************************************

CPU 0: Total no. of CPUs = 4
Launched CPU 1
Launched CPU 2
Launched CPU 3
MDK>memum.nb
Created successfully pid = 1
CPU 0: User mem starts from 0xa800000000641000
CPU0 PID1 Allocating 0x000000000705f000 bytes of memory on NODE0
NODE0 CPU0 PID1 Trying to malloc 0x000000000705f000 bytes
NODE0 CPU0 PID1 malloc SUCCESSFUL for 0x000000000705f000 bytes
vmem_addri 0x000000000060a000 vmem_addrf 0x0000000007669000 size
0x000000000705f000
CPU0 PID1 Allocating 0x0000000007c80000 bytes of memory on NODE1
NODE1 CPU0 PID1 Trying to malloc 0x0000000007c80000 bytes
NODE1 CPU0 PID1 malloc SUCCESSFUL for 0x0000000007c80000 bytes
vmem_addri 0x0000000007669000 vmem_addrf 0x000000000f2e9000 size
0x0000000007c80000
CNODE0 NASID0 CPU0 ppid1
Config_t:
   numCpus    4
   numNodes   2
   numRouters 1
   CNODE000 NASID000:
     numCpus  2
     mem size 0x0000000008000000
     Bank0    0x0000000008000000
     Bank1    0x0000000000000000
     Bank2    0x0000000000000000
     Bank3    0x0000000000000000
     Bank4    0x0000000000000000
     Bank5    0x0000000000000000
     Bank6    0x0000000000000000
     Bank7    0x0000000000000000
   CNODE001 NASID001:
     numCpus  2
     mem size 0x0000000008000000
     Bank0    0x0000000008000000
     Bank1    0x0000000000000000
     Bank2    0x0000000000000000
     Bank3    0x0000000000000000
     Bank4    0x0000000000000000
     Bank5    0x0000000000000000
     Bank6    0x0000000000000000
     Bank7    0x0000000000000000
INFO_T addr of ppid      0xa800000007fde788
```

```
INFO_T addr of sysconfig 0xa800000007fde790
INFO_T addr of procinfo  0xa800000000b11000
INFO_T addr of nodeinfo  0xa800000000b12000
INFO_T addr of addrinfo  0xa800000000b13000
NODE000 BANK0 mem   0x0000000008000000
NODE000 BANK0 saddr 0xa800000000b14000
NODE000 BANK0 eaddr 0xa800000007fbdff8
NODE000 BANK0 size  0x00000000074a9ff8
```

[configuration information for Node 0, banks 1 through 6 (not shown)]

```
NODE000 BANK7 mem   0x0000000000000000
NODE000 BANK7 saddr 0x0000000000000000
NODE000 BANK7 eaddr 0x0000000000000000
NODE000 BANK7 size  0x0000000000000000
NODE001 BANK0 mem   0x0000000008000000
NODE001 BANK0 saddr 0xa800000100020000
NODE001 BANK0 eaddr 0xa800000107fbdff8
NODE001 BANK0 size  0x0000000007f9dff8
```

[configuration information for Node 1, banks 1 through 6 (not shown)]

```
NODE001 BANK7 mem   0x0000000000000000
NODE001 BANK7 saddr 0x0000000000000000
NODE001 BANK7 eaddr 0x0000000000000000
NODE001 BANK7 size  0x0000000000000000
NODE0 CPU1 PID2 unmapped memory phase two test
NODE0 CPU0 PID1 unmapped memory phase two test
NODE0 CPU1 PID2 Unmapped memory random address and data test
NODE0 CPU0 PID1 Unmapped memory random address and data test
NODE0 CPU0 PID1 unmapped memory random addr and data test Loop 0
NODE0 CPU1 PID2 unmapped memory random addr and data test Loop 0
NODE1 CPU3 PID4 unmapped memory phase two test
NODE1 CPU2 PID3 unmapped memory phase two test
NODE1 CPU3 PID4 Unmapped memory random address and data test
NODE1 CPU2 PID3 Unmapped memory random address and data test
NODE1 CPU2 PID3 unmapped memory random addr and data test Loop 0
NODE1 CPU3 PID4 unmapped memory random addr and data test Loop 0
NODE1 CPU3 PID4 unmapped memory random addr and data test Loop 5000
NODE1 CPU3 PID4 unmapped memory random addr and data test Loop 10000
NODE0 CPU0 PID1 unmapped memory random addr and data test Loop 5000
NODE0 CPU1 PID2 unmapped memory random addr and data test Loop 5000
NODE1 CPU3 PID4 unmapped memory random addr and data test Loop 15000
NODE1 CPU3 PID4 unmapped memory random addr and data test Loop 20000
NODE1 CPU2 PID3 unmapped memory random addr and data test Loop 5000
NODE1 CPU3 PID4 unmapped memory random addr and data test Loop 25000
NODE0 CPU1 PID2 unmapped memory random addr and data test Loop 10000
NODE0 CPU0 PID1 unmapped memory random addr and data test Loop 10000
NODE1 CPU3 PID4 unmapped memory random addr and data test Loop 30000
NODE1 CPU2 PID3 unmapped memory random addr and data test Loop 10000
NODE1 CPU3 PID4 unmapped memory random addr and data test Loop 35000
NODE0 CPU1 PID2 unmapped memory random addr and data test Loop 15000
NODE0 CPU0 PID1 unmapped memory random addr and data test Loop 15000
NODE1 CPU3 PID4 unmapped memory random addr and data test Loop 40000
NODE1 CPU3 PID4 unmapped memory random addr and data test Loop 45000
NODE1 CPU3 PID4 unmapped memory random address and data test PASSED
NODE1 CPU2 PID3 unmapped memory random addr and data test Loop 15000
```

```
NODE0 CPU1 PID2 unmapped memory random addr and data test Loop 20000
NODE0 CPU0 PID1 unmapped memory random addr and data test Loop 20000
NODE1 CPU2 PID3 unmapped memory random addr and data test Loop 20000
NODE0 CPU1 PID2 unmapped memory random addr and data test Loop 25000
NODE0 CPU0 PID1 unmapped memory random addr and data test Loop 25000
NODE1 CPU2 PID3 unmapped memory random addr and data test Loop 25000
NODE1 CPU2 PID3 unmapped memory random addr and data test Loop 30000
NODE0 CPU1 PID2 unmapped memory random addr and data test Loop 30000
NODE0 CPU0 PID1 unmapped memory random addr and data test Loop 30000
NODE1 CPU2 PID3 unmapped memory random addr and data test Loop 35000
NODE0 CPU1 PID2 unmapped memory random addr and data test Loop 35000
NODE0 CPU0 PID1 unmapped memory random addr and data test Loop 35000
NODE1 CPU2 PID3 unmapped memory random addr and data test Loop 40000
NODE1 CPU2 PID3 unmapped memory random addr and data test Loop 45000
NODE0 CPU0 PID1 unmapped memory random addr and data test Loop 40000
NODE0 CPU1 PID2 unmapped memory random addr and data test Loop 40000
NODE1 CPU2 PID3 unmapped memory random address and data test PASSED
NODE0 CPU0 PID1 unmapped memory random addr and data test Loop 45000
NODE0 CPU1 PID2 unmapped memory random addr and data test Loop 45000
NODE0 CPU0 PID1 unmapped memory random address and data test PASSED
NODE0 CPU1 PID2 unmapped memory random address and data test PASSED
**** MDK memory unmapped test status summary ****
CNODE0 NASID0 CPU0 PID1 Mem test PASSED
CNODE0 NASID0 CPU1 PID2 Mem test PASSED
CNODE1 NASID1 CPU2 PID3 Mem test PASSED
CNODE1 NASID1 CPU3 PID4 Mem test PASSED
Memory unmapped test has completed
```

4.5 Internode Memory Test

The internode memory test uses all Nodes and CPUs in the system to achieve very aggressive interaction of cache coherence protocol. There is only an unmapped version of the internode memory test (*memum.in*).

4.5.1 How to Run the Internode Memory Test

Use the following procedure to load the test from the local system disk:

- Use the *boot* command from the PROM monitor (at the BaseIO command prompt, >>) to start the *memory.mdk* test package:

  ```
  >>boot dksc(0,1,0)/stand/memory.mdk
  ```

  MDK displays boot status information and the `MDK>` prompt.

- Load the *memum.in* test from the `MDK>` prompt:

  ```
  MDK>memum.in
  ```

  The internode memory test loads into the system and begins testing memory.

4.5.2 Internode Memory Test Description

The internode memory test uses the following algorithm:

- Acquire the system configuration.
  1. Determine the amount of memory that is populated in the system.
  2. Determine the number of CPUs and Nodes in the system.
- Determine the size of the data block and divide the data block between the CPUs.
  1. If the number of CPUs in the system is greater than 128, the size of the data block equals the number of CPUs (in bytes). Each CPU owns a byte of the data block to which the CPU can write data.
  2. If the number of CPUs in the system is less than or equal to 128, the size of the data block is 128 bytes. Each CPU owns (128 divided by the number of CPUs) bytes of the data block to which the CPU can write data.
- Have each CPU write data to its portion of the data block.
  1. Randomly select a Node.
  2. Repeat the following actions until unique addresses can no longer be generated:
     - Randomly select a variable number of bits in the address that will randomly change. The remaining bits in the address do not change.
     - Fill the selected bits with random data to generate a random address.
     - Write random data to the memory address.
- Have each CPU read data back from its portion of the data block.
- Verify that the data that was read back is the same as the data that was written.

4.5.3 Output From the Internode Memory Test

The internode memory test returns output to the BaseIO console.

4.5.3.1 Pass Output

As the test runs, it prints out a PASSED message for each test section that completes without detecting hardware failures. The following examples show some of these messages:

```
NODE0 CPU0 PID1 inter node test PASSED
NODE1 CPU3 PID4 inter node test PASSED
NODE0 CPU1 PID2 inter node test PASSED
NODE1 CPU2 PID3 inter node test PASSED
```

When all test sections have completed successfully, the test prints out a summary for each CPU in the system, as shown in the following example messages:

```
**** MDK memory test status summary ****
CNODE0 NASID0 CPU0 PID1 Mem test PASSED
CNODE0 NASID0 CPU1 PID2 Mem test PASSED
CNODE1 NASID1 CPU2 PID3 Mem test PASSED
CNODE1 NASID1 CPU3 PID4 Mem test PASSED
Memory unmapped test has completed
```

4.5.3.2 Failure Output

If this test detects errors, it prints an ERROR message and related output, similar to the following example messages:

```
ERROR NODEx CPUy PIDx testname FAILED
Data type type
Virtual addr 0x000000000cdb1a07  Physical addr 0xa800000181f05a07
Expected data 0xf9  Actual data 0x79
Syndrom 0x80
Address 0000000081f05a07 bit 7 is on NASID 1
MMYH4 DIMML line 7
```

If the internode memory test fails, the failing hardware could be a memory DIMM, a secondary cache (a HIMM), a Node board, a midplane, a cable, or a Router board. The failing FRU could be the memory DIMM, Node board, midplane, cable, or Router board that contains the failing hardware component.

4.5.4 Example of Running the Internode Memory Test

The following example shows a test that is running and has not detected any errors:

```
>>boot dksc(0,1,0)/stand/memory.mdk
Booting Nanos.........

************************************************************
SGI Nanos Version 1.10 SN0 built 09:41:44 PM Jan 24, 1997
************************************************************

CPU 0: Total no. of CPUs = 8
Launched CPU 1
Launched CPU 2
Launched CPU 3
Launched CPU 4
Launched CPU 5
Launched CPU 6
Launched CPU 7
MDK>memum.in
Created successfully pid = 1
CPU 0: User mem starts from 0xa800000000691000
CPU0 PID1 Allocating 0x0000000006a0f000 bytes of memory on NODE0
NODE0 CPU0 PID1 Trying to malloc 0x0000000006a0f000 bytes
NODE0 CPU0 PID1 malloc SUCCESSFUL for 0x0000000006a0f000 bytes
vmem_addri 0x000000000060a000 vmem_addrf 0x0000000007019000 size
0x0000000006a0f000
CPU0 PID1 Allocating 0x0000000007c80000 bytes of memory on NODE1
NODE1 CPU0 PID1 Trying to malloc 0x0000000007c80000 bytes
NODE1 CPU0 PID1 malloc SUCCESSFUL for 0x0000000007c80000 bytes
vmem_addri 0x0000000007019000 vmem_addrf 0x000000000ec99000 size
0x0000000007c80000
CPU0 PID1 Allocating 0x0000000007c80000 bytes of memory on NODE2
NODE2 CPU0 PID1 Trying to malloc 0x0000000007c80000 bytes
NODE2 CPU0 PID1 malloc SUCCESSFUL for 0x0000000007c80000 bytes
vmem_addri 0x000000000ec99000 vmem_addrf 0x0000000016919000 size
0x0000000007c80000
CPU0 PID1 Allocating 0x0000000007c80000 bytes of memory on NODE3
NODE3 CPU0 PID1 Trying to malloc 0x0000000007c80000 bytes
NODE3 CPU0 PID1 malloc SUCCESSFUL for 0x0000000007c80000 bytes
vmem_addri 0x0000000016919000 vmem_addrf 0x000000001e599000 size
0x0000000007c80000
CNODE0 NASID0 CPU0 ppid1
SCACHE_CONFIG_T:
    sc_size     1
    sc_size_max 6
    sc_bsize    1
    qwo[sbit]   00      qwo[size]   04      qwo[mask]   0x0000000f
    qw[sbit]    04      qw[size]    02      qw[mask]    0x00000003
    index[sbit] 06      index[size] 13      index[mask] 0x00001fff
    data[sbit]  04      data[size]  15      data[mask]  0x00007fff
    tag[sbit]   19      tag[size]   21      tag[mask]   0x001fffff
    way[sbit]   07      way[size]   13      way[mask]   0x00000fff
Config_t:
    numCpus    8
    numNodes   4
    numRouters 2
```

```
        CNODE0 NASID0:
            numCpus  2
            mem size 0x0000000008000000
        CNODE1 NASID1:
            numCpus  2
            mem size 0x0000000008000000
        CNODE2 NASID3:
            numCpus  2
            mem size 0x0000000008000000
        CNODE3 NASID2:
            numCpus  2
            mem size 0x0000000008000000
NODE_MEM_PTR_T:
        CNODE0 NASID0:
        vmem_addri 0x000000000060a000
        vmem_addrf 0x0000000007019000
        pmem_addri 0xa8000000006b4000
        pmem_addrf 0xa8000000070c3000
        size       0x0000000006a0f000
        CNODE1 NASID1:
        vmem_addri 0x0000000007019000
        vmem_addrf 0x000000000ec99000
        pmem_addri 0xa800000100020000
        pmem_addrf 0xa800000107ca0000
        size       0x0000000007c80000
        CNODE2 NASID3:
        vmem_addri 0x000000000ec99000
        vmem_addrf 0x0000000016919000
        pmem_addri 0xa800000300020000
        pmem_addrf 0xa800000307ca0000
        size       0x0000000007c80000
        CNODE3 NASID2:
        vmem_addri 0x0000000016919000
        vmem_addrf 0x000000001e599000
        pmem_addri 0xa800000200020000
        pmem_addrf 0xa800000207ca0000
        size       0x0000000007c80000
PROCESS_T:
    Cpu0:
        type   0  READ_WRITE
        status 0  IDLE
    Cpu1:
        type   1  READ_ONLY
        status 0  IDLE
    Cpu2:
        type   0  READ_WRITE
        status 0  IDLE
    Cpu3:
        type   1  READ_ONLY
        status 0  IDLE
    Cpu4:
        type   0  READ_WRITE
        status 0  IDLE
    Cpu5:
        type   1  READ_ONLY
        status 0  IDLE
    Cpu6:
```

```
        type   0  READ_WRITE
        status 0  IDLE
    Cpu7:
        type   1  READ_ONLY
        status 0  IDLE
CPU0 PID1 Sproc on CPU1
CPU0 PID1 Sproc on CPU2
NODE0 CPU1 PID2 In child
CPU0 PID1 Sproc on CPU3
NODE0 CPU1 PID2 inter node test
CPU0 PID1 Sproc on CPU4
NODE0 CPU1 PID2 Loop 0
CPU0 PID1 Sproc on CPU5
NODE1 CPU2 PID3 In child
CPU0 PID1 Sproc on CPU6
NODE1 CPU2 PID3 inter node test
CPU0 PID1 Sproc on CPU7
NODE1 CPU2 PID3 Loop 0
NODE0 CPU0 PID1 inter node test
NODE1 CPU3 PID4 In child
NODE0 CPU0 PID1 Loop 0
NODE1 CPU3 PID4 inter node test
NODE3 CPU7 PID8 In child
NODE1 CPU3 PID4 Loop 0
NODE3 CPU7 PID8 inter node test
NODE2 CPU4 PID5 In child
NODE3 CPU7 PID8 Loop 0
NODE3 CPU6 PID7 In child
NODE2 CPU5 PID6 In child
NODE3 CPU6 PID7 inter node test
NODE2 CPU5 PID6 inter node test
NODE3 CPU6 PID7 Loop 0
NODE2 CPU5 PID6 Loop 0
NODE2 CPU4 PID5 inter node test
NODE2 CPU4 PID5 Loop 0
NODE3 CPU7 PID8 Loop 5000
NODE0 CPU1 PID2 Loop 5000
NODE0 CPU0 PID1 Loop 5000
NODE1 CPU2 PID3 Loop 5000
NODE1 CPU3 PID4 Loop 5000
NODE2 CPU4 PID5 Loop 5000
NODE2 CPU5 PID6 Loop 5000
NODE3 CPU6 PID7 Loop 5000
NODE3 CPU7 PID8 Loop 10000
NODE0 CPU0 PID1 Loop 10000
NODE0 CPU1 PID2 Loop 10000
NODE1 CPU2 PID3 Loop 10000
NODE1 CPU3 PID4 Loop 10000
NODE2 CPU4 PID5 Loop 10000
NODE3 CPU7 PID8 Loop 15000
NODE3 CPU6 PID7 Loop 10000
NODE3 CPU7 PID8 Loop 20000
NODE0 CPU1 PID2 Loop 15000
NODE0 CPU1 PID2 Loop 20000
NODE1 CPU2 PID3 Loop 15000
NODE1 CPU3 PID4 Loop 15000
NODE2 CPU4 PID5 Loop 15000
```

```
NODE1 CPU2 PID3 Loop 20000
NODE1 CPU3 PID4 Loop 20000
NODE2 CPU4 PID5 Loop 20000
NODE2 CPU5 PID6 Loop 15000
NODE3 CPU6 PID7 Loop 15000
NODE0 CPU1 PID2 Loop 25000
NODE2 CPU5 PID6 Loop 20000
NODE3 CPU7 PID8 Loop 25000
NODE3 CPU6 PID7 Loop 20000
NODE0 CPU1 PID2 Loop 30000
NODE3 CPU7 PID8 Loop 30000
Memory internode test has completed
```

4.6 Long Memory Tests

The long memory tests run extended versions of the quick screen, Node board, and internode memory tests. The long memory tests are *mem.lo* (mapped version) and *memum.lo* (unmapped version).

4.6.1 How to Run the Long Memory Tests

Use the following procedure to load the long memory tests from the local system disk:

- Use the *boot* command from the PROM monitor (at the BaseIO command prompt, >>) to start the *memory.mdk* test package:

  ```
  >>boot dksc(0,1,0)/stand/memory.mdk
  ```

  MDK displays boot status information and the MDK> prompt.

- Load the mapped version (*mem.lo*) or unmapped version (*memum.lo*) from the MDK> prompt:

  1. To load the mapped version, enter *mem.lo* at the MDK> prompt:

     ```
     MDK>mem.lo
     ```

  2. To load the unmapped version, enter *memum.lo* at the MDK> prompt:

     ```
     MDK>memum.lo
     ```

  The specified long memory test loads into the system and begins testing memory.

4.6.2 Test Description

Both long memory tests use the following algorithm:

- Acquire the system configuration.
  1. Determine the amount of memory that is populated in the system.
  2. Determine the number of CPUs and Nodes in the system.
  3. Scale the test to the system configuration.
- Run an enhanced version of the quick screen memory test.

  The address bus test uses the following additional data patterns:

  0xc33cc33cc33cc33c 0x3cc33cc33cc33cc3 0x6996699669966996 0x9669966996699669
  0xaaaaaaaaaaaaaaaa 0x5555555555555555 0x0000000000000000 0xffffffffffffffff
- Run 1,000 loops of the Node board memory test.
- Run 5,000 loops of the internode memory test.

4.6.3 Output From the Long Memory Tests

The long memory tests return output to the BaseIO console.

4.6.3.1 Pass Output

As the tests run, they print out a `PASSED` message for each test section that completes without detecting hardware failures. The following examples show some of these messages:

```
NODE0 CPU0 PID1 SSO over cache line test PASSED
NODE1 CPU2 PID3 SSO over cache line test PASSED
NODE1 CPU2 PID3 memory data bus test PASSED
NODE0 CPU0 PID1 memory data bus test PASSED
NODE0 CPU0 PID1 memory background test PASSED
NODE1 CPU2 PID3 memory background test PASSED
NODE0 CPU0 PID1 memory address bus test PASSED
NODE1 CPU2 PID3 memory address bus test PASSED
NODE0 CPU0 PID1 random data test PASSED
NODE1 CPU2 PID3 random data test PASSED
NODE0 CPU0 PID1 random address and data test PASSED
NODE0 CPU1 PID2 random address and data test PASSED
NODE1 CPU3 PID4 random address and data test PASSED
NODE1 CPU2 PID3 random address and data test PASSED
NODE0 CPU0 PID1 inter node test PASSED
NODE1 CPU3 PID4 inter node test PASSED
NODE0 CPU1 PID2 inter node test PASSED
NODE1 CPU2 PID3 inter node test PASSED
```

When all test sections have completed successfully, the tests print out a summary for each CPU in the system, as shown in the following example messages:

```
**** MDK memory test status summary ****
CNODE0 NASID0 CPU0 PID1 Mem test PASSED
CNODE0 NASID0 CPU1 PID2 Mem test PASSED
CNODE1 NASID1 CPU2 PID3 Mem test PASSED
CNODE1 NASID1 CPU3 PID4 Mem test PASSED
MDK memory mapped test has completed
```

Note: The last line is for the *mem.lo* test. It changes to `Memory unmapped test has completed` for the *memum.lo* test.

4.6.3.2 Failure Output

If these tests detect errors, they print an error message and related output, similar to the following example messages:

```
ERROR NODEx CPUy PIDx testname FAILED
Data type type
Virtual addr 0x000000000cdb1a07  Physical addr 0xa800000181f05a07
Expected data 0xf9  Actual data 0x79
Syndrom 0x80
Address 0000000081f05a07 bit 7 is on NASID 1
MMYH4 DIMML line 7
```

If either of the long memory tests fails, the failing hardware could be a memory DIMM, a secondary cache (a HIMM), a Node board, a midplane, a cable, or a Router board. The failing FRU could be the memory DIMM, Node board, midplane, cable, or Router board that contains the failing hardware component.

4.6.4 Examples of Running the Long Memory Tests

This section contains examples of running the *mem.lo* and *memum.lo* long memory tests.

4.6.4.1 Example of Running the *mem.lo* Test

The following example output shows the *mem.lo* test running on a 2-Node system (4 CPUs). In this example, the test completes successfully without detecting any hardware failures.

```
>>boot dksc(0,1,0)/stand/memory.mdk
Booting Nanos.........
************************************************************
SGI Nanos Version 1.10 SN0 built 02:53:28 PM Mar 25, 1997
************************************************************

CPU 0: Total no. of CPUs = 4
Launched CPU 1
Launched CPU 2
Launched CPU 3
MDK>mem.lo
Created successfully pid = 1
CPU 0: User mem starts from 0xa800000000641000
CPU0 PID1 Allocating 0x000000000705f000 bytes of memory on NODE0
NODE0 CPU0 PID1 Trying to malloc 0x000000000705f000 bytes
NODE0 CPU0 PID1 malloc SUCCESSFUL for 0x000000000705f000 bytes
vmem_addri 0x000000000060a000 vmem_addrf 0x0000000007669000 size 0x000000000705f000
CPU0 PID1 Allocating 0x0000000007c80000 bytes of memory on NODE1
NODE1 CPU0 PID1 Trying to malloc 0x0000000007c80000 bytes
NODE1 CPU0 PID1 malloc SUCCESSFUL for 0x0000000007c80000 bytes
vmem_addri 0x0000000007669000 vmem_addrf 0x000000000f2e9000 size
0x0000000007c80000
CNODE0 NASID0 CPU0 ppid1
SCACHE_CONFIG_T:
     sc_size     1
     sc_size_max 6
     sc_bsize    1
     qwo[sbit]   00      qwo[size]   04      qwo[mask]   0x0000000f
     qw[sbit]    04      qw[size]    02      qw[mask]    0x00000003
     index[sbit] 06      index[size] 13      index[mask] 0x00001fff
     data[sbit]  04      data[size]  15      data[mask]  0x00007fff
     tag[sbit]   19      tag[size]   21      tag[mask]   0x001fffff
     way[sbit]   07      way[size]   13      way[mask]   0x00000fff
 Config_t:
     numCpus    4
     numNodes   2
     numRouters 1
         CNODE0 NASID0:
             numCpus  2
             mem size 0x0000000008000000
         CNODE1 NASID1:
             numCpus  2
             mem size 0x0000000008000000
 NODE_MEM_PTR_T:
         CNODE0 NASID0:
         vmem_addri 0x000000000060a000
         vmem_addrf 0x0000000007669000
```

```
        pmem_addri 0xa800000000664000
        pmem_addrf 0xa8000000076c3000
        size       0x000000000705f000
        CNODE1 NASID1:
        vmem_addri 0x0000000007669000
        vmem_addrf 0x000000000f2e9000
        pmem_addri 0xa800000100020000
        pmem_addrf 0xa800000107ca0000
        size       0x0000000007c80000
PROCESS_T:
    Cpu0:
        type   0  READ_WRITE
        status 0  IDLE
    Cpu1:
        type   1  READ_ONLY
        status 0  IDLE
    Cpu2:
        type   0  READ_WRITE
        status 0  IDLE
    Cpu3:
        type   1  READ_ONLY
        status 0  IDLE
NODE0 CPU0 PID1 memory pattern "0xc33cc33cc33cc33c 0x3cc33cc33cc33cc3" test PASSED
NODE1 CPU2 PID3 memory pattern "0xc33cc33cc33cc33c 0x3cc33cc33cc33cc3" test PASSED
NODE0 CPU0 PID1 memory pattern "0xffffffffffffffff 0x0000000000000000" test PASSED
NODE1 CPU2 PID3 memory pattern "0xffffffffffffffff 0x0000000000000000" test PASSED
NODE0 CPU0 PID1 SSO over cache line test PASSED
NODE1 CPU2 PID3 SSO over cache line test PASSED
NODE1 CPU3 PID4 Read only test
NODE0 CPU0 PID1 memory data bus test
NODE0 CPU1 PID2 Read only test
NODE1 CPU2 PID3 memory data bus test
NODE1 CPU2 PID3 memory data bus test PASSED
NODE0 CPU0 PID1 memory data bus test PASSED
NODE1 CPU2 PID3 memory background test
NODE0 CPU0 PID1 memory background test
NODE0 CPU0 PID1 memory background test PASSED
NODE0 CPU0 PID1 memory address bus test
NODE0 CPU0 PID1 memory address bus test "Ulong_nec" index 0
NODE1 CPU2 PID3 memory background test PASSED
NODE1 CPU2 PID3 memory address bus test
NODE1 CPU2 PID3 memory address bus test "Ulong_nec" index 0
NODE0 CPU0 PID1 memory address bus test "Ulong_nec" index 1
NODE1 CPU2 PID3 memory address bus test "Ulong_nec" index 1
NODE0 CPU0 PID1 memory address bus test "Ulong_NEC" index 0
NODE1 CPU2 PID3 memory address bus test "Ulong_NEC" index 0
NODE0 CPU0 PID1 memory address bus test "Ulong_NEC" index 1
NODE1 CPU2 PID3 memory address bus test "Ulong_NEC" index 1
NODE0 CPU0 PID1 memory address bus test "Ulong_NEC" index 2
NODE1 CPU2 PID3 memory address bus test "Ulong_NEC" index 2
NODE0 CPU0 PID1 memory address bus test "Ulong_NEC" index 3
NODE1 CPU2 PID3 memory address bus test "Ulong_NEC" index 3
NODE0 CPU0 PID1 memory address bus test "Ulong_NEC" index 4
NODE1 CPU2 PID3 memory address bus test "Ulong_NEC" index 4
NODE0 CPU0 PID1 memory address bus test "Ulong_NEC" index 5
NODE1 CPU2 PID3 memory address bus test "Ulong_NEC" index 5
NODE0 CPU0 PID1 memory address bus test "Ulong_NEC" index 6
```

```
NODE1 CPU2 PID3 memory address bus test "Ulong_NEC" index 6
NODE0 CPU0 PID1 memory address bus test "Ulong_NEC" index 7
NODE0 CPU0 PID1 memory address bus test "Ulong" index 0
NODE1 CPU2 PID3 memory address bus test "Ulong_NEC" index 7
NODE0 CPU0 PID1 memory address bus test "Ulong" index 1
NODE1 CPU2 PID3 memory address bus test "Ulong" index 0
NODE0 CPU0 PID1 memory address bus test "Ulong" index 2
NODE1 CPU2 PID3 memory address bus test "Ulong" index 1
NODE0 CPU0 PID1 memory address bus test "Ulong" index 3
NODE1 CPU2 PID3 memory address bus test "Ulong" index 2
NODE0 CPU0 PID1 memory address bus test "Ulong" index 4
NODE1 CPU2 PID3 memory address bus test "Ulong" index 3
NODE0 CPU0 PID1 memory address bus test "Ulong" index 5
NODE1 CPU2 PID3 memory address bus test "Ulong" index 4
NODE0 CPU0 PID1 memory address bus test "Ulong" index 6
NODE1 CPU2 PID3 memory address bus test "Ulong" index 5
NODE0 CPU0 PID1 memory address bus test PASSED
NODE0 CPU0 PID1 random data test
NODE0 CPU0 PID1 Loop 0 data type 3
NODE1 CPU2 PID3 memory address bus test "Ulong" index 6
NODE1 CPU2 PID3 memory address bus test PASSED
NODE1 CPU2 PID3 random data test
NODE1 CPU2 PID3 Loop 0 data type 3
NODE0 CPU0 PID1 Loop 10 data type 3
NODE1 CPU2 PID3 Loop 10 data type 3
NODE0 CPU0 PID1 Loop 20 data type 3
NODE1 CPU2 PID3 Loop 20 data type 3
NODE0 CPU0 PID1 Loop 30 data type 3
NODE1 CPU2 PID3 Loop 30 data type 3
NODE0 CPU0 PID1 Loop 40 data type 3
NODE1 CPU2 PID3 Loop 40 data type 3
NODE0 CPU0 PID1 random data test PASSED
NODE1 CPU2 PID3 random data test PASSED
NODE0 CPU0 PID1 random address and data test
NODE0 CPU1 PID2 random address and data test
NODE0 CPU0 PID1 Loop 0
NODE0 CPU1 PID2 Loop 0
NODE1 CPU3 PID4 random address and data test
NODE1 CPU2 PID3 random address and data test
NODE1 CPU3 PID4 Loop 0
NODE1 CPU2 PID3 Loop 0
NODE0 CPU0 PID1 Loop 5000
NODE1 CPU3 PID4 Loop 5000
NODE0 CPU1 PID2 Loop 5000
NODE1 CPU2 PID3 Loop 5000
NODE0 CPU0 PID1 Loop 10000
NODE1 CPU3 PID4 Loop 10000
NODE0 CPU1 PID2 Loop 10000
NODE1 CPU2 PID3 Loop 10000
NODE0 CPU0 PID1 Loop 15000
NODE1 CPU3 PID4 Loop 15000
NODE0 CPU1 PID2 Loop 15000
NODE0 CPU0 PID1 Loop 20000
NODE1 CPU3 PID4 Loop 20000
NODE1 CPU2 PID3 Loop 15000
NODE0 CPU1 PID2 Loop 20000
NODE0 CPU0 PID1 Loop 25000
```

```
NODE1 CPU3 PID4 Loop 25000
NODE0 CPU0 PID1 Loop 30000
NODE1 CPU2 PID3 Loop 20000
NODE0 CPU1 PID2 Loop 25000
NODE0 CPU0 PID1 Loop 35000
NODE1 CPU3 PID4 Loop 30000
NODE0 CPU1 PID2 Loop 30000
NODE1 CPU2 PID3 Loop 25000
NODE0 CPU0 PID1 Loop 40000
NODE1 CPU3 PID4 Loop 35000
NODE0 CPU1 PID2 Loop 35000
NODE0 CPU0 PID1 Loop 45000
NODE1 CPU2 PID3 Loop 30000
NODE1 CPU3 PID4 Loop 40000
NODE0 CPU1 PID2 Loop 40000
NODE0 CPU0 PID1 random address and data test PASSED
NODE1 CPU3 PID4 Loop 45000
NODE1 CPU2 PID3 Loop 35000
NODE0 CPU1 PID2 Loop 45000
NODE0 CPU1 PID2 random address and data test PASSED
NODE1 CPU3 PID4 random address and data test PASSED
NODE1 CPU2 PID3 Loop 40000
NODE1 CPU2 PID3 Loop 45000
NODE1 CPU2 PID3 random address and data test PASSED
NODE0 CPU1 PID2 inter node test
NODE0 CPU0 PID1 inter node test
NODE0 CPU1 PID2 Loop 0
NODE1 CPU3 PID4 inter node test
NODE0 CPU0 PID1 Loop 0
NODE1 CPU2 PID3 inter node test
NODE1 CPU3 PID4 Loop 0
NODE1 CPU2 PID3 Loop 0
NODE0 CPU0 PID1 Loop 5000
NODE1 CPU3 PID4 Loop 5000
NODE0 CPU1 PID2 Loop 5000
NODE1 CPU2 PID3 Loop 5000
NODE0 CPU0 PID1 Loop 10000
NODE1 CPU3 PID4 Loop 10000
NODE0 CPU1 PID2 Loop 10000
NODE1 CPU2 PID3 Loop 10000
NODE0 CPU0 PID1 Loop 15000
NODE1 CPU3 PID4 Loop 15000
NODE0 CPU0 PID1 Loop 20000
NODE1 CPU3 PID4 Loop 20000
NODE0 CPU1 PID2 Loop 15000
NODE1 CPU2 PID3 Loop 15000
NODE0 CPU1 PID2 Loop 20000
NODE1 CPU2 PID3 Loop 20000
NODE0 CPU0 PID1 Loop 25000
NODE1 CPU3 PID4 Loop 25000
NODE0 CPU0 PID1 Loop 30000
NODE1 CPU3 PID4 Loop 30000
NODE0 CPU1 PID2 Loop 25000
NODE0 CPU0 PID1 Loop 35000
NODE1 CPU2 PID3 Loop 25000
NODE1 CPU3 PID4 Loop 35000
NODE0 CPU1 PID2 Loop 30000
```

```
NODE1 CPU2 PID3 Loop 30000
NODE0 CPU0 PID1 Loop 40000
NODE0 CPU1 PID2 Loop 35000
NODE1 CPU3 PID4 Loop 40000
NODE0 CPU0 PID1 Loop 45000
NODE1 CPU2 PID3 Loop 35000
NODE1 CPU3 PID4 Loop 45000
NODE0 CPU0 PID1 inter node test PASSED
NODE0 CPU1 PID2 Loop 40000
NODE0 CPU1 PID2 Loop 45000
NODE1 CPU3 PID4 inter node test PASSED
NODE1 CPU2 PID3 Loop 40000
NODE1 CPU2 PID3 Loop 45000
NODE0 CPU1 PID2 inter node test PASSED
NODE1 CPU2 PID3 inter node test PASSED
**** MDK memory test status summary ****
CNODE0 NASID0 CPU0 PID1 Mem test PASSED
CNODE0 NASID0 CPU1 PID2 Mem test PASSED
CNODE1 NASID1 CPU2 PID3 Mem test PASSED
CNODE1 NASID1 CPU3 PID4 Mem test PASSED
MDK memory mapped test has completed
```

4.6.4.2 Example of Running the *memum.lo* Test

The following example output shows the *memum.lo* test running on a 2-Node system (4 CPUs). In this example, the test completes successfully without detecting any hardware failures.

```
>>boot dksc(0,1,0)/stand/memory.mdk
Booting Nanos.........
************************************************************
SGI Nanos Version 1.10 SN0 built 02:53:28 PM Mar 25, 1997
************************************************************

CPU 0: Total no. of CPUs = 4
Launched CPU 1
Launched CPU 2
Launched CPU 3
MDK>memum.lo
Created successfully pid = 1
CPU 0: User mem starts from 0xa800000000641000
CPU0 PID1 Allocating 0x000000000705f000 bytes of memory on NODE0
NODE0 CPU0 PID1 Trying to malloc 0x000000000705f000 bytes
NODE0 CPU0 PID1 malloc SUCCESSFUL for 0x000000000705f000 bytes
vmem_addri 0x000000000060a000 vmem_addrf 0x0000000007669000 size 0x000000000705f000
CPU0 PID1 Allocating 0x0000000007c80000 bytes of memory on NODE1
NODE1 CPU0 PID1 Trying to malloc 0x0000000007c80000 bytes
NODE1 CPU0 PID1 malloc SUCCESSFUL for 0x0000000007c80000 bytes
vmem_addri 0x0000000007669000 vmem_addrf 0x000000000f2e9000 size
0x0000000007c80000
CNODE0 NASID0 CPU0 ppid1
Config_t:
   numCpus    4
   numNodes   2
   numRouters 1
   CNODE000 NASID000:
```

```
    numCpus  2
    mem size 0x0000000008000000
    Bank0    0x0000000008000000
    Bank1    0x0000000000000000
    Bank2    0x0000000000000000
    Bank3    0x0000000000000000
    Bank4    0x0000000000000000
    Bank5    0x0000000000000000
    Bank6    0x0000000000000000
    Bank7    0x0000000000000000
  CNODE001 NASID001:
    numCpus  2
    mem size 0x0000000008000000
    Bank0    0x0000000008000000
    Bank1    0x0000000000000000
    Bank2    0x0000000000000000
    Bank3    0x0000000000000000
    Bank4    0x0000000000000000
    Bank5    0x0000000000000000
    Bank6    0x0000000000000000
    Bank7    0x0000000000000000
INFO_T addr of ppid      0xa800000007fde788
INFO_T addr of sysconfig 0xa800000007fde790
INFO_T addr of procinfo  0xa800000000b11000
INFO_T addr of nodeinfo  0xa800000000b12000
INFO_T addr of addrinfo  0xa800000000b13000
NODE000 BANK0 mem   0x0000000008000000
NODE000 BANK0 saddr 0xa800000000b14000
NODE000 BANK0 eaddr 0xa800000007fbdff8
NODE000 BANK0 size  0x00000000074a9ff8
```

[configuration information for Node 0, banks 1 through 6 (not shown)]

```
NODE000 BANK7 mem   0x0000000000000000
NODE000 BANK7 saddr 0x0000000000000000
NODE000 BANK7 eaddr 0x0000000000000000
NODE000 BANK7 size  0x0000000000000000
NODE001 BANK0 mem   0x0000000008000000
NODE001 BANK0 saddr 0xa800000100020000
NODE001 BANK0 eaddr 0xa800000107fbdff8
NODE001 BANK0 size  0x0000000007f9dff8
```

[configuration information for Node 1, banks 1 through 6 (not shown)]

```
NODE001 BANK7 mem   0x0000000000000000
NODE001 BANK7 saddr 0x0000000000000000
NODE001 BANK7 eaddr 0x0000000000000000
NODE001 BANK7 size  0x0000000000000000
NODE0 CPU0 PID1 unmapped memory phase zero test
NODE0 CPU1 PID2 unmapped memory phase zero test
NODE0 CPU0 PID1 unmapped memory pattern test
NODE1 CPU2 PID3 unmapped memory phase zero test
NODE1 CPU3 PID4 unmapped memory phase zero test
NODE1 CPU2 PID3 unmapped memory pattern test
NODE0 CPU0 PID1 memory pattern "0xc33cc33cc33cc33c 0x3cc33cc33cc33cc3" test PASSED
NODE0 CPU0 PID1 unmapped memory pattern test
NODE1 CPU2 PID3 memory pattern "0xc33cc33cc33cc33c 0x3cc33cc33cc33cc3" test PASSED
```

```
NODE1 CPU2 PID3 unmapped memory pattern test
NODE0 CPU0 PID1 memory pattern "0xffffffffffffffff 0x0000000000000000" test PASSED
NODE0 CPU0 PID1 unmapped memory SSO over cahe line test
NODE1 CPU2 PID3 memory pattern "0xffffffffffffffff 0x0000000000000000" test PASSED
NODE1 CPU2 PID3 unmapped memory SSO over cahe line test
NODE0 CPU0 PID1 SSO over cache line test PASSED
NODE0 CPU0 PID1 unmapped memory phase zero test PASSED
NODE1 CPU2 PID3 SSO over cache line test PASSED
NODE1 CPU2 PID3 unmapped memory phase zero test PASSED
NODE0 CPU1 PID2 unmapped memory phase zero test PASSED
NODE1 CPU3 PID4 unmapped memory phase zero test PASSED
NODE0 CPU1 PID2 unmapped memory phase one test
NODE0 CPU0 PID1 unmapped memory phase one test
NODE0 CPU0 PID1 unmapped memory background test
NODE1 CPU2 PID3 unmapped memory phase one test
NODE1 CPU3 PID4 unmapped memory phase one test
NODE1 CPU2 PID3 unmapped memory background test
NODE0 CPU0 PID1 unmapped memory background test PASSED
NODE0 CPU0 PID1 unmapped memory cell test
NODE0 CPU0 PID1 unmapped memory cell test Ulong_NEC bank 0
NODE1 CPU2 PID3 unmapped memory background test PASSED
NODE1 CPU2 PID3 unmapped memory cell test
NODE1 CPU2 PID3 unmapped memory cell test Ulong_NEC bank 0
NODE0 CPU0 PID1 unmapped memory cell test Ulong bank 0
NODE1 CPU2 PID3 unmapped memory cell test Ulong bank 0
NODE0 CPU0 PID1 unmapped memory cell test PASSED
NODE0 CPU0 PID1 random data test
NODE0 CPU0 PID1 random data test loop 0
NODE1 CPU2 PID3 unmapped memory cell test PASSED
NODE1 CPU2 PID3 random data test
NODE1 CPU2 PID3 random data test loop 0
```

[additional loop count status output (not shown)]

```
NODE0 CPU0 PID1 random data test loop 40
NODE1 CPU2 PID3 random data test loop 40
NODE0 CPU0 PID1 random data test PASSED
NODE0 CPU0 PID1 unmapped memory phase one test PASSED
NODE1 CPU2 PID3 random data test PASSED
NODE1 CPU2 PID3 unmapped memory phase one test PASSED
NODE0 CPU1 PID2 unmapped memory phase one test PASSED
NODE1 CPU3 PID4 unmapped memory phase one test PASSED
NODE1 CPU2 PID3 unmapped memory phase two test
NODE0 CPU1 PID2 unmapped memory phase two test
NODE0 CPU0 PID1 unmapped memory phase two test
NODE0 CPU1 PID2 Unmapped memory random address and data test
NODE0 CPU1 PID2 unmapped memory random addr and data test Loop 0
NODE0 CPU0 PID1 Unmapped memory random address and data test
NODE0 CPU0 PID1 unmapped memory random addr and data test Loop 0
NODE1 CPU3 PID4 unmapped memory phase two test
NODE1 CPU2 PID3 Unmapped memory random address and data test
NODE1 CPU2 PID3 unmapped memory random addr and data test Loop 0
NODE1 CPU3 PID4 Unmapped memory random address and data test
NODE1 CPU3 PID4 unmapped memory random addr and data test Loop 0
NODE1 CPU3 PID4 unmapped memory random addr and data test Loop 5000
NODE0 CPU1 PID2 unmapped memory random addr and data test Loop 5000
NODE0 CPU0 PID1 unmapped memory random addr and data test Loop 5000
```

```
NODE1 CPU2 PID3 unmapped memory random addr and data test Loop 5000
```

[additional loop count status output (not shown)]

```
NODE1 CPU3 PID4 unmapped memory random addr and data test Loop 45000
NODE1 CPU3 PID4 unmapped memory random address and data test PASSED
NODE1 CPU2 PID3 unmapped memory random addr and data test Loop 45000
NODE1 CPU2 PID3 unmapped memory random address and data test PASSED
NODE0 CPU1 PID2 unmapped memory random addr and data test Loop 45000
NODE0 CPU0 PID1 unmapped memory random addr and data test Loop 45000
NODE0 CPU1 PID2 unmapped memory random address and data test PASSED
NODE0 CPU0 PID1 unmapped memory random address and data test PASSED
NODE0 CPU0 PID1 Unmapped memory inter node test
NODE0 CPU1 PID2 Unmapped memory inter node test
NODE0 CPU0 PID1 Loop 0
NODE0 CPU1 PID2 Loop 0
NODE1 CPU2 PID3 Unmapped memory inter node test
NODE1 CPU3 PID4 Unmapped memory inter node test
NODE1 CPU2 PID3 Loop 0
NODE1 CPU3 PID4 Loop 0
NODE1 CPU3 PID4 Loop 5000
NODE0 CPU0 PID1 Loop 5000
NODE0 CPU1 PID2 Loop 5000
NODE1 CPU2 PID3 Loop 5000
```

[more loop count status output (not shown)]

```
NODE1 CPU3 PID4 Loop 45000
NODE1 CPU3 PID4 Unmapped memory inter node test PASSED
NODE0 CPU1 PID2 Loop 45000
NODE0 CPU0 PID1 Loop 45000
NODE1 CPU2 PID3 Loop 45000
NODE0 CPU1 PID2 Unmapped memory inter node test PASSED
NODE0 CPU0 PID1 Unmapped memory inter node test PASSED
NODE1 CPU2 PID3 Unmapped memory inter node test PASSED
**** MDK memory unmapped test status summary ****
CNODE0 NASID0 CPU0 PID1 Mem test PASSED
CNODE0 NASID0 CPU1 PID2 Mem test PASSED
CNODE1 NASID1 CPU2 PID3 Mem test PASSED
CNODE1 NASID1 CPU3 PID4 Mem test PASSED
Memory unmapped test has completed
```

Chapter 5

Router SSO Test

This chapter describes the diagnostic that you can use to test the Router links.

5.1 About the Router SSO Test

The Router simultaneous switching (Router SSO) test is a directed test that focuses on testing the Router links.

To run the Router SSO test, you must load the *t5r.mdk* test package.

If this test detects a Router failure, the failing hardware is a CPOP connector, a midplane, a cable, a terminator, or a Router chip. The failing FRU is a Node board, Router board, midplane, or connecting cable. If this test detects a HUB failure, the failing FRU is a Node board, Router board, or midplane.

Note: Before you replace any FRUs, reseat the suspected boards and reconnect the suspected cables, and then run the test again. If the test still fails, you will need to replace one or more of the FRUs.

5.2 How to Run the Router SSO Test

Use the following procedure to load the test from the local system disk:

- Use the *boot* command from the PROM monitor (at the BaseIO command prompt, >>) to start the *t5r.mdk* test package:

  ```
  >>boot dksc(0,1,0)/stand/t5r.mdk
  ```

  MDK displays boot status information and the `MDK>` prompt.

- Load the Router SSO test, *router*, from the `MDK>` prompt:

  ```
  MDK>router
  ```

  The Router SSO test loads into the system and begins testing the Router links.

5.3 Router SSO Test Description

The Router SSO test performs 96 passes of the following procedure:

- Select a data pattern from a predefined set of data patterns.
- Initialize the 16-Mbyte data area in memory on each Node board to the selected data pattern.

  Note: During this step, the test performs operations that ensure that the data contained in each secondary cache is written back to memory.

- Have one CPU on each Node board execute 50 iterations of the following procedure:

  1. Select a different Node board from which the CPU will obtain data.

     The CPU selects a different Node board for each iteration to ensure that data flows through all Router links in the system.

  2. Fetch data from memory on the selected Node board.
  3. The CPU performs multiple reads until the CPU reads the entire 16 Mbytes of data from the data area of memory on the selected Node board. Each read occurs at an address that is a cache line offset of the address used in the previous read: This ensures that the test performs as many back-to-back read operations as possible.
  4. Read the Router error register on each Router. If there are more than 10 check-bit errors, print out an error message. If there are 10 check-bit errors or less, write the check-bit error count into a data structure that the test will use to generate a report at the end of the test.
  5. Read the HUB network interface (NI) error register on each HUB. If there are more than 10 check-bit errors in a HUB NI, print out an error message. If there are 10 check-bit errors or less in a HUB NI, write the check-bit error count into a data structure that the test will use to generate a report at the end of the test.

- Have CPU0 gather the data from the data structure, summarize it, and print status information for the current loop.

Note: The total amount of data moving through the system during each pass equals 16 Mbytes multiplied by 50 (the number of iterations) multiplied by the number of Node boards in the system.

5.4 Router SSO Test Output

The Router SSO test returns output to the BaseIO console. All output comes from CPU0, which summarizes the status data from all other CPUs in the system.

5.4.1 Pass Output

When the test completes all 96 passes, it prints a `TEST RESULT` message.

For a test that completes all 96 passes without detecting any failures, the `TEST RESULT` message is a `TEST PASSED` message:

```
TEST RESULT: **** TEST PASSED  ****
MDK Router_sso test run is complete.
```

5.4.2 Failure Output for Router Failures

When an individual pass detects that more than 10 check-bit errors have occurred on a Router, it prints an error message and a table that indicates which port had the errors. If a pass detects that fewer than 10 check-bit errors occurred, it updates the data in the table but does not print out the table.

The following output shows an example error message and table when there are more than 10 check-bit errors on a Router:

```
ERROR: CPU0: pid 1: Checkbit error count threshold of 10 exceeded.

Checkbit_err count on ROUTER:
 ----------------------------

 Nasid  Module  Node  Router      port:
                slot   slot        1     2     3     4     5     6
 -----  ------  ----  ------      ----  ----  ----  ----  ----  ----
    0       1     1       1         0     0     0     0     0     0
    3       1     4       2         0     0     0     0     0     0
    2       1     3       2         0     0     0     0     0     0
    7       2     4       2         0     0     0     0     0     0
    6       2     3       2       190     0     0     0     0     0
    5       2     2       1         0     0     0     0     0     0
    4       2     1       1         0     0     0     0     0     0
    1       1     2       1         0     0     0     0     0     0
```

This example shows that 190 check-bit errors occurred on Router port 1 of the Router board in slot 2 of module 2.

When the test completes all 96 passes, it prints a test result message, which is either a warning message or a fail message. When one of these messages appears, check the `Checkbit_err count on router` portion of the output that follows the message to determine the ports on which the errors occurred.

5.4.2.1 Warning Message for Router Failures

A warning message indicates that the test has detected check-bit errors on one or more Router ports, but each port had fewer than the threshold of 10 check-bit errors per pass. The following example output includes a warning message because there are two ports that each have one check-bit error:

```
TEST RESULT: **** WARNING: CHECKBIT_error_count > 0,              ****
TEST RESULT: ****          BUT LESS THAN THRESHOLD OF 10 PER PASS ****

Checkbit_err count on router:
-----------------------------

Nasid  Module  Node  Router      port:
               slot   slot        1     2     3     4     5     6
-----  ------  ----  ------      ----  ----  ----  ----  ----  ----
   0       1     1       1         0     0     0     0     0     0
   3       1     4       2         0     0     0     0     1     0
   2       1     3       2         0     0     0     0     0     0
   7       2     4       2         0     0     0     0     0     0
   6       2     3       2         0     0     0     0     0     0
   5       2     2       1         0     0     0     0     0     0
   4       2     1       1         0     0     0     0     1     0
   1       1     2       1         0     0     0     0     0     0
```

If the test returns a `WARNING` message for a Router, the failing hardware is hard to isolate. The failing hardware might be the Router chip for the port that has one to nine errors. The failing FRU could be the Router board with the failing Router chip. The failing FRU could also be a CrayLink cable or midplane.

Note: Before you replace any FRUs, reseat the suspected boards and reconnect the suspected cables, and then run the test again. If the test still fails, you will need to replace one or more of the FRUs.

Refer to Appendix A, "Troubleshooting Link Failures," for general information about troubleshooting link failures.

5.4.2.2 Fail Message for Router Failures

A fail message indicates that the test has detected more than 10 check-bit errors on one or more Routers on an individual pass. The following example output includes a fail message because Router port 1 of the Router board in slot 2 of module 2 has 190 check-bit errors:

```
TEST RESULT: **** FAIL: NUMBER OF CHECKBIT ERRORS EXCEEDED THRESHOLD ****
TEST RESULT: ****       OF 10 ON AT LEAST ONE PASS OF THE TEST ****

Checkbit_err count on router:
-----------------------------

Nasid  Module  Node  Router      port:
               slot   slot        1     2     3     4     5     6
-----  ------  ----  ------      ----  ----  ----  ----  ----  ----
   0       1     1       1         0     0     0     0     0     0
   3       1     4       2         0     0     0     0     0     0
   2       1     3       2         0     0     0     0     0     0
   7       2     4       2         0     0     0     0     0     0
   6       2     3       2       190     0     0     0     0     0
   5       2     2       1         0     0     0     0     0     0
   4       2     1       1         0     0     0     0     0     0
   1       1     2       1         0     0     0     0     0     0
```

If the test returns a fail message for a Router, check the `Checkbit_err count on router` portion of the output to determine more information about the failure:

- If the errors are on ports 1, 2, or 3; check the cables that connect the Routers on those ports.
- If the errors are on ports 4, 5, or 6; the failing hardware components are the CPOP connectors on the Router board or Node board. The failing FRU is the Router board or Node board.
- If the check-bit error count is a value in that ranges from the hundreds to the thousands, the failing hardware is most likely a CPOP connector on the Router board or Node board, a midplane, or a CrayLink cable. The failing FRU is the Router board or Node board.
- If the check-bit error count is a value that ranges from the tens to hundreds, the failing hardware is most likely a terminator on the Router board. The failing FRU is the Router board.
- If the check-bit error count is less than 10, the failing hardware is hard to isolate, but it could be a Router chip. The failing FRU could be the Router board with the failing Router chip.

Note: Before you replace any FRUs, reseat the suspected boards and reconnect the suspected cables, and then run the test again. If the test still fails, you will need to replace one or more of the FRUs.

Refer to Appendix A, "Troubleshooting Link Failures," for general information about troubleshooting link failures.

5.4.3 Failure Output for HUB Failures

When an individual pass detects that more than 10 check-bit errors have occurred on a HUB, it prints an error message and a table that indicates which HUB had the errors. If a pass detects that fewer than 10 check-bit errors occurred, it updates the data in the table but does not print out the table.

The following output shows an example error message and table when there are more than 10 check-bit errors on a HUB:

```
ERROR: CPU0: pid 1: Checkbit error count threshold of 10 exceeded.

 Checkbit_err count on HUB Network Interface:
 ----------------------------------------

 Nasid  Module  Node  Router    Checkbit_err
                slot   slot         count
 -----  ------  ----  ------    -----------
    0       1     1       1             0
    3       1     4       2             0
    2       1     3       2             0
    7       2     4       2             0
    6       2     3       2            11
    5       2     2       1             0
    4       2     1       1             0
    1       1     2       1             0
```

This example shows that there were 11 check-bit errors detected on the HUB that is located on the Node board in slot 3 of module 2.

When the test completes all 96 passes, it prints a test result message, which is either a warning message or a fail message. When one of these messages appears, check the `Checkbit_err count on HUB Network Interface` portion of the output that follows the message to determine the Hubs on which the errors occurred.

5.4.3.1 Warning Message for HUB Failures

A warning message indicates that the test has detected check-bit errors on one or more HUB NI registers, but each HUB NI register had fewer than the threshold of 10 check-bit errors per pass. The following example output includes a warning message because there is one HUB NI register with one check-bit error:

```
TEST RESULT: **** WARNING: CHECKBIT_error_count > 0,              ****
TEST RESULT: ****          BUT LESS THAN THRESHOLD OF 10 PER PASS ****

 Checkbit_err count on HUB Network Interface:
 -------------------------------------------

 Nasid  Module  Node  Router    Checkbit_err
                slot   slot         count
 -----  ------  ----  ------    ------------
    0       1     1       1             0
    3       1     4       2             0
    2       1     3       2             0
    7       2     4       2             0
    6       2     3       2             1
    5       2     2       1             0
    4       2     1       1             0
    1       1     2       1             0
```

If the test returns a `WARNING` message for a HUB, the failing hardware could be a Node board, Router board, or midplane.

5.4.3.2 Fail Message for HUB Failures

A fail message indicates that the test has detected more than 10 check-bit errors in one or more HUB NI registers on an individual pass. The following example output includes a fail message because Router port 1 of the Router board in slot 2 of module 2 has 190 check-bit errors:

```
TEST RESULT: **** FAIL: NUMBER OF CHECKBIT ERRORS EXCEEDED THRESHOLD ****
TEST RESULT: ****       OF 10 ON AT LEAST ONE PASS OF THE TEST ****

 Checkbit_err count on HUB Network Interface:
 -------------------------------------------

 Nasid  Module  Node   Router    Checkbit_err
                slot    slot         count
 -----  ------  ----  ------    ------------
    0       1     1       1             0
    3       1     4       2             0
    2       1     3       2             0
    7       2     4       2             0
    6       2     3       2            11
    5       2     2       1             0
    4       2     1       1             0
    1       1     2       1             0
```

If the test returns a fail message for a HUB, check the `Checkbit_err count on HUB Network Interface` portion of the output to determine more information about the failure. The failing hardware could be a Node board, Router board, or midplane.

In the example output shown above, the check-bit errors occurred on the Node board in slot 3 of module 2. The failing hardware is either the Node board in slot 3 of module 2, the Router board in slot 2 of module 2, or the midplane.

5.5 Example of Running the Router SSO Test

The following example shows passes 0, 94, and 95 from running the test on a system with 16 CPUs (2 modules with a total of 8 Nodes, 16 CPUs, and 4 Routers). In this example, pass 0 does not detect any errors, pass 94 detects more than 10 check-bit errors, and pass 95 (the final pass) does not detect any errors.

In this example, the test reports 190 check-bit errors on Router port 1 of the Router board in slot 2 of module 2. The Router board is most likely causing this failure because the check-bit errors are on port 1, which is a Router board connection.

Pass 94 of the test also reports 11 check-bit errors in the HUB NI register on the Node board in slot 3 of module 2. This failure is caused by the Node board in slot 3 of module 2, the Router board in slot 2 of module 2, or the midplane.

```
>>boot dksc(0,1,0)/stand/t5r.mdk
Booting Nanos.........

************************************************************
SGI Nanos Version 1.10 SN0 built 09:02:48 PM Dec 30, 1996
************************************************************

CPU 0: Total no. of CPUs = 16
Launched CPU 1
Launched CPU 2
Launched CPU 3
Launched CPU 4
Launched CPU 5
Launched CPU 6
Launched CPU 7
Launched CPU 8                                    Initial parameters
Launched CPU 9
Launched CPU 10
Launched CPU 11
Launched CPU 12
Launched CPU 13
Launched CPU 14
Launched CPU 15
MDK>router
Created successfully pid = 1
CPU 0: User mem starts from 0xa800000000732000

Starting Router SSO test.

Checkbit Error Threshold Count = 10 per pass.

System Configuration:

   Number of nodes:      8
   Number of cpus:      16
   Number of routers:    4
```

Figure 5-1 Router SSO Test Sample Output (Part 1 of 4)

```
Table of module, nasid, and slot numbers:
     Module       Nasid       Node       Router
                              Slot        slot
     ------       -----       ----       ------
        1           0           1          1
        1           3           4          2
        1           2           3          2
        2           7           4          2
        2           6           3          2
        2           5           2          1
        2           4           1          1
        1           1           2          1

Number of initialized data patterns = 96


CPU0: pid 1: PASS = 0 Started
CPU0: pid 1: Initializing memory on each node
             to data pattern = 5aaaaaaaaaaaaaaa 5aaaaaaaaaaaaaaa
             to test router link data line = 0
CPU0: pid 1: Starting test code on all nodes.
CPU0: pid 1: PASS = 0 Completed




CPU0: pid 1: PASS = 94 Started
CPU0: pid 1: Initializing memory on each node
             to data pattern = 3333333333333373 ccccccccccccc8c
             to test router link data line = 14
CPU0: pid 1: Starting test code on all nodes.

ERROR: CPU0: pid 1: Checkbit error count threshold of 10 exceeded.

Checkbit_err count on router:
-----------------------------


Nasid  Module  Node  Router      port:
               slot   slot        1     2     3     4     5     6
-----  ------  ----  ------      ----  ----  ----  ----  ----  ----
   0       1     1       1         0     0     0     0     0     0
   3       1     4       2         0     0     0     0     0     0
   2       1     3       2         0     0     0     0     0     0
   7       2     4       2         0     0     0     0     0     0
   6       2     3       2       190     0     0     0     0     0
   5       2     2       1         0     0     0     0     0     0
   4       2     1       1         0     0     0     0     0     0
   1       1     2       1         0     0     0     0     0     0
```

Initial parameters

Pass 0 information

Passes 2 through 93 are not shown in this example.

Pass 94 information

Failure for pass 94:

Router port 1 on the Router board in slot 2 of module 2 has 190 check-bit errors

Figure 5-2 Router SSO Test Sample Output (Part 2 of 4)

```
ERROR: CPU0: pid 1: Checkbit error count threshold of 10 exceeded.

 Checkbit_err count on HUB Network Interface:
 ----------------------------------------

 Nasid  Module  Node  Router    Checkbit_err
               slot   slot         count
 -----  ------  ----  ------    ------------
    0       1     1     1             0
    3       1     4     2             0
    2       1     3     2             0
    7       2     4     2             0
    6       2     3     2            11
    5       2     2     1             0
    4       2     1     1             0
    1       1     2     1             0

CPU0: pid 1: PASS = 94 Completed

CPU0: pid 1: PASS = 95 Started
CPU0: pid 1: Initializing memory on each node
             to data pattern = 3333333333333337 cccccccccccccc8
             to test router link data line = 15
CPU0: pid 1: Starting test code on all nodes.

CPU0: pid 1: PASS = 95 Completed
TEST RESULT: **** FAIL: NUMBER OF CHECKBIT ERRORS EXCEEDED THRESHOLD ****
TEST RESULT: ****       OF 10 ON AT LEAST ONE PASS OF THE TEST ****

Checkbit_err count on router:
----------------------------

Nasid  Module  Node  Router      port:
               slot   slot        1     2     3     4     5     6
-----  ------  ----  ------      ----  ----  ----  ----  ----  ----
   0       1     1     1          0     0     0     0     0     0
   3       1     4     2          0     0     0     0     0     0
   2       1     3     2          0     0     0     0     0     0
   7       2     4     2          0     0     0     0     0     0
   6       2     3     2        190     0     0     0     0     0
   5       2     2     1          0     0     0     0     0     0
   4       2     1     1          0     0     0     0     0     0
   1       1     2     1          0     0     0     0     0     0
```

Failure for pass 94: HUB NI register on Node board in slot 3 of module 2 has 11 check-bit errors

Pass 95 information

Test result message and check-bit error counts for Router failure

Figure 5-3 Router SSO Test Sample Output (Part 3 of 4)

```
TEST RESULT: **** FAIL: NUMBER OF CHECKBIT ERRORS EXCEEDED THRESHOLD ****
TEST RESULT: ****       OF 10 ON AT LEAST ONE PASS OF THE TEST       ****

Checkbit_err count on HUB Network Interface:
-------------------------------------------

Nasid  Module  Node   Router     Checkbit_err
               slot    slot          count
-----  ------  ----   ------     ------------
   0       1       1       1             0
   3       1       4       2             0
   2       1       3       2             0
   7       2       4       2             0
   6       2       3       2            11
   5       2       2       1             0
   4       2       1       1             0
   1       1       2       1             0

MDK Router_sso test run is complete.
```

Test result message and check-bit error counts for HUB failure

Failures for this test:

Router port 1 on the Router board in slot 2 of module 2 has 190 check-bit errors and HUB NI register on the Node board in slot 3 of module 2 has 11 check-bit errors

Figure 5-4 Router SSO Test Sample Output (Part 4 of 4)

Chapter 6

Cache Thrasher With I/O Test

This chapter describes the cache thrasher with I/O test, which stress tests the CrayLink network and the HUB-to-XBOW links.

6.1 About the Cache Thrasher With I/O Test

The cache thrasher with I/O test stress tests the CrayLink network and the HUB-to-XBOW links. This test also stress tests cache coherency from the processors and from I/O.

To run the cache thrasher with I/O test, you must load the *stress.mdk* package.

If the test fails, check the Router board LEDs and Crossbow LEDs on the midplane for lost links.

6.2 How to Run the Cache Thrasher With I/O Test

Use the following procedure to load the test from the local system disk:

- Use the *boot* command from the PROM monitor (at the BaseIO command prompt, >>) to start the *stress.mdk* test package:

  ```
  >>boot dksc(0,1,0)/stand/stress.mdk
  ```

  MDK displays boot status information and the `MDK>` prompt.

- Load the cache thrasher with I/O test, *ct_io_turbo*, from the `MDK>` prompt:

  ```
  MDK> ct_io_turbo
  ```

  The cache thrasher with I/O test loads into the system and begins testing the system.

6.3 Cache Thrasher With I/O Test Description

The cache thrasher with I/O test initializes several memory arrays and then loops through the following test algorithm ten times:

- Select random values for all parameters.

- Perform a random number of iterations of the test sequence. During each iteration, every CPU in the system performs the following actions:
  1. Write data to the memory arrays, read data back, and compare the data that was read back with the data that was written. (Occasionally perform operations that write back data from cache into memory.)
  2. Perform block transfer engine (BTE) data transfers.
  3. Write data to the graphics buffer and verify the data.
  4. Write data to the fetch-and-op address space.
  5. Write to and read from the local HUB registers.
  6. Read instructions from the code and text area of the test, and write the instructions into memory.

Note: The cache thrasher with I/O test writes data to and reads data from all types of address space in the system (except I/O address space). The test also causes all CPUs to write to and read from the same data areas so that line sharing occurs between the CPUs.

- Report status information about the Router ports. Table 6-1 describes the Router port status information; the test obtains this information from the Router local block registers.

Table 6-1 Cache Thrasher With I/O Test Router Port Status Information

Heading	Description
`Port`	Port on the Router chip
`oflw`	Number of overflow errors for the port
`bdDir`	Number of bad direction errors for the port
`dedTO`	Number of deadlock time-out errors for the port
`tailTO`	Number of tail time-out errors for the port
`retry`	Number of LLP transmit retries for the port
`cb_err`	Number of check-bit errors for the port
`sn_err`	Number of sequence number errors for the port
`Bkt3`	Histogram bucket 3 (shows the free running count)
`Bkt2`	Histogram bucket 2 (shows how many sent packets have occurred)
`Bkt1`	Histogram bucket 1 (shows how many received packets have occurred)
`Bkt0`	Histogram bucket 0 (shows how many bypass packets have occurred)

This test takes 30 minutes to run to a successful completion.

6.4 Cache Thrasher With I/O Test Output

The cache thrasher with I/O test returns output to the BaseIO console. It also continually updates the Node board LEDs and turns the amber LEDs on the Router board on or off.

6.4.1 Pass Output

If the cache thrasher with I/O test completes successfully without detecting any failures, each CPU in the system prints the following message:

```
**********************************************
*   Cache thrasher: TEST COMPLETED.          *
*                                            *
* MDK Cache thrasher test run is complete.   *
* You may now reset the machine.             *
**********************************************
```

Once all CPUs have displayed this message, use the *reset* command at the `MDK>` prompt to reset the system:

```
MDK> reset
```

6.4.2 Failure Output

The cache thrasher with I/O test indicates failures three ways:

- The test hangs.
- The test takes an unexpected exception.
- The test reports that it detected a hardware error.

6.4.2.1 Failure Output When the Test Hangs

Normally, while the test is running, you should see messages similar to the following messages continually appear on the BaseIO console:

```
PID(1) 1485 fetch_op: 0x1  bte xfers = 356
PID(2) 1485 fetch_op: 0x2  bte xfers = 356
PID(3) 1485 fetch_op: 0x3  bte xfers = 789
PID(4) 1485 fetch_op: 0x4  bte xfers = 791
```

You will also see the Node board LEDs continually change patterns.

When this test hangs, there will be no output on the BaseIO console and no activity on the Node board LEDs. This usually indicates that an I/O link went down because of a hard failure.

Note: Wait for at least 3 minutes of inactivity before deciding that a hang condition has occurred.

Refer to Appendix A, "Troubleshooting Link Failures," for general information about troubleshooting link failures.

6.4.2.2 Failure Output When the Test Takes an Unexpected Exception

If an unexpected exception occurs, the MDK exception handler code takes control of the system. MDK prints `ERROR: Unexpected Exception Occurred`, followed by a register dump. The register dump contains Coprocessor 0 registers, the general-purpose registers, and the following information about the exception: the type of exception, the address where the exception occurred, and any other information that is relevant to the exception.

The following example shows the output from MDK when the cache thrasher with I/O test takes an unexpected exception:

```
ERROR: Unexpected Exception Occured.
 Nasid 1: Local CPU A: Global CPU 2: PID 3: Multiple exceptions at
0xa800000000a0c4e8: sr = 0xb4007fe6: cause = 0x881c
 Nanos: Frozen 22A 001:
 Status: 0xffffffffb4007fe6        XCtxt: 0xffffffe000002a30
 Hi:     0x                0       Lo:    0x                0
 epc:    0xa800000000a0c4e8        cause: 0x              881c
 count:  0x          18afb669      comp:  0xffffffffffffffff
 Enhi:   0x                0       CacErr: 0xffffffffd442aa80
 R01/AT: 0x            1ff00       R02/V0: 0xffffffffb4007fe6
 R03/V1: 0x           a257b0       R04/A0: 0xa800000000a27000
 R05/A1: 0x           a1dd38       R06/A2: 0x          8bfdfdf0
 R07/A3: 0x                8       R08/A4: 0x                 4
 R09/A5: 0xa800000183fdf978        R10/A6: 0x          3e17aad8
 R11/A7: 0xa800000300caaad8        R12/T0: 0x                38
 R13/T1: 0x                0       R14/T2: 0x                 0
 R15/T3: 0xa800000300caaa98        R16/S0: 0xa800000183fdf978
 R17/S1: 0xa8000000005fbdc8        R18/S2: 0x                 0
 R19/S3: 0x                0       R20/S4: 0x                 2
 R21/S5: 0xa800000008000000        R22/S6: 0x                 3
 R23/S7: 0x               e0       R24/T8: 0x           8000000
 R25/T9: 0xa800000000a07ed8        R28/GP: 0x                 0
 R29/SP: 0xa800000183fdd928        R30/S8: 0xa800000000a27000
 R31/RA: 0xa800000000a08eac
 Nanos: Frozen 3
```

An unexpected exception usually indicates that a Router link went down with a hard failure (there is no longer a connection).

Refer to Appendix A, "Troubleshooting Link Failures," for general information about troubleshooting link failures.

6.4.2.3 Failure Output When the Test Detects an Error

If this test detects an error, it prints out an error message, some error information, a `Hitting the brakes` message, and an `MDK Cache thrasher test run is complete` message.

The following example shows the output from a bus error:

```
PID(2) - handler grabbed plock
PID(2) BUS ERROR
PID(2) data bus error exception handler
PID(2) os_ef_badvaddr(a800000000a0dc50)
PID(2) returning from data bus error handler epc(a800000000a04084)
PID(2) Hitting the brakes
PID(2) MDK Cache thrasher test run is complete.
```

In this example, the failing hardware could be a memory error or a Router link connection that is failing but not completely down. To determine whether the Router link connection is the failing hardware, check the output that displays the Router port status. An excessive number (more than ten) of check-bit errors in the `cb_err` column indicates that the Router link connection may be the failing hardware.

The following example shows 150 check-bit errors on port 5:

```
Port    oflw bdDir dedTO tailTO retry cb_err sn_err  Bkt3  Bkt2  Bkt1  Bkt0
 Port1:  0     0     0     0     0     0     255    a528     1     0     0
 Port2:  0     0     0     0     0     0       0    a963     1     0     0
 Port3:  0     0     0     0     0     0     255    a992     1     0     0
 Port4:  0     0     0     0     1     0       0    a9b0  1ac0  137f   f31
 Port5:  0     0     0     0     0   150     255    aa45  137e  1afc  1213
 Port6:  0     0     0     0     0     0       0    aab2     1     0     0
```

Refer to A., "Troubleshooting Link Failures," for general information about troubleshooting link failures.

6.4.3 Example of Running the Cache Thrasher With I/O Test

Figure 6-1 through Figure 6-4 show example output from the first loop of the cache thrasher with I/O test. In this example, the test runs on a 2-Node system (4 CPUs and 1 Router) system without detecting any errors.

```
>>boot dksc(0,1,0_/stand/stress.MDK
1027024+856 entry: 0xa800000000019180
Booting Nanos.........
************************************************************
SGI Nanos Version 1.10 SN0 built 09:44:07 PM Mar 10, 1997
************************************************************

CPU 0: Total no. of CPUs = 4
Launched CPU 1
Launched CPU 2
Launched CPU 3
MDK>ct_io_turbo
Created successfully pid = 1
CPU 0: User mem starts from 0xa800000000641000
cons buf addr = 0xa800000000100000
Entering cache thrasher sweep program.
debug_level = 2
malloc 27262976 bytes on the master node.
malloc starts at 0xa800000000a27000.
sproc'ing 4 processes on 2 nodes.
PID(1) node 0x0 XBOW reports wid 0xa link is down
PID(3) node 0x1 XBOW reports wid 0x9 link is down
PID(1) io_peer_wid = 0x0
PID(1) WARNING: io_peer_wid = 0
PID(3) io_peer_wid = 0x0
PID(3) WARNING: io_peer_wid = 0
PID(1) completed NASID assignment loop. nasid[0] = 0
PID(3) completed NASID assignment loop. nasid[1] = 1
RR_STATUS_REV_ID = 0x16a56d0000f21
new RR_STATUS_REV_ID = 0x16a56d0034f21
Clearing router stats.
RR_STATUS_REV_ID = 0x12a56d0034f21
PID(1) GLOBAL_PARMS = 0x3fffffe2001f0ffb
new RR_STATUS_REV_ID = 0x12a56d0034f21
PID(1) Set router_hist to 0x405f0
local status: 16a56d0034f21
Port   oflw bdDir dedTO tailTO retry cb_err sn_err  Bkt3  Bkt2  Bkt1 Bkt0
Port1:  0    0     0     0      0     0     255    d2f1    0     0 0
Port2:  0    0     0     0      0     0       0    d2f9    0     0 0
Port3:  0    0     0     0      0     0     255    d2f9    0     0 0
Port4:  0    0     0     0      0     0       0    d2f9   17    18 18
Port5:  0    0     0     0      0     0       0    d2f8   36    37 17
Port6:  0    0     0     0      0     0       0    d2f8    0     0 0
PID(1) Pick initial random seed of 0xb21d0704.
-----------------------------------------------
-
-           LOOP NUMBER 1 OF 10
-
-----------------------------------------------
using random seed of 0xb21d0704.
num_memories is 1
PID(1) Setting graphics array base = 0xa27180.
PID(1) Total io writes = 0x0.
PID(1) Total io reads = 0x0.
```

Initial parameters

Start of current loop

Parameters for the current loop

Figure 6-1 Cache Thrasher With I/O Sample Output (Part 1 of 4)

```
PID(1) Graphics array base = 0xa27180.
PID(1) Clear 6 ct_turbo arrays..
PID(1) Done clearing.
PID(1) Beginning ct_turbo loop with 4 cpus.
     master_quiet = 0x0
     slave_quiet = 0x0
     array size = 0xac80
     iterations = 10000
     stride = 9
     nlocs = 306
     force_wb = 0
     write_text = 36
     prefetch = 1
     write_local = 0
     read_local = 0
     read_rtc = 0
     read_vector = 0
     interleave_reads = 0
     reverse_odd = 1
     use_sc = 0
     use_uncached = 0
     reader unroll depth = 16
     writer unroll depth = 16
     print_freq is every 495 iterations.
     led_freq is every 61 iterations.
     Using 6 arrays, 3 deep.
     array[ 0] begins at      0xa800000000a271a4
        d2[ 1] begins at      0xa800000000aa71a4
        d2[ 2] begins at      0xa800000000b271a4
     array[ 1] begins at      0xa800000000c271a4
        d2[ 1] begins at      0xa800000000ca71a4
        d2[ 2] begins at      0xa800000000d271a4
     array[ 2] begins at      0xa800000000e271a4
        d2[ 1] begins at      0xa800000000ea71a4
        d2[ 2] begins at      0xa800000000f271a4
     array[ 3] begins at      0xa8000000010271a4
        d2[ 1] begins at      0xa8000000010a71a4
        d2[ 2] begins at      0xa8000000011271a4
     array[ 4] begins at      0xa8000000012271a4
        d2[ 1] begins at      0xa8000000012a71a4
        d2[ 2] begins at      0xa8000000013271a4
     array[ 5] begins at      0xa8000000014271a4
        d2[ 1] begins at      0xa8000000014a71a4
        d2[ 2] begins at      0xa8000000015271a4
     fetch_op array begins at 0x9400000002027200 (stride 16)
     bte copy size is 0x10000
PID(1) bte source array begins at 0xa800000000a27180
          target array begins at 0xa800000002227180
PID(2) bte source array begins at 0xa800000000a27180
          target array begins at 0xa800000002227180
PID(3) bte source array begins at 0xa800000100a27180
          target array begins at 0xa800000102227180
PID(4) bte source array begins at 0xa800000100a27180
```

Parameters for the current loop

Figure 6-2 Cache Thrasher With I/O Sample Output (Part 2 of 4)

```
          target array begins at 0xa800000102227180
PID(1) entering iter loop
PID(2) entering iter loop
PID(1) 0 fetch_op: 0x1  bte xfers = 0
PID(2) 0 fetch_op: 0x2  bte xfers = 0
PID(4) entering iter loop
PID(3) entering iter loop
PID(4) 0 fetch_op: 0x4  bte xfers = 0
PID(3) 0 fetch_op: 0x3  bte xfers = 0
PID(1) 495 fetch_op: 0x1  bte xfers = 117
PID(2) 495 fetch_op: 0x2  bte xfers = 117
PID(4) 495 fetch_op: 0x4  bte xfers = 262
PID(3) 495 fetch_op: 0x3  bte xfers = 262
PID(1) 990 fetch_op: 0x1  bte xfers = 238
PID(2) 990 fetch_op: 0x2  bte xfers = 238
PID(3) 990 fetch_op: 0x3  bte xfers = 528
PID(4) 990 fetch_op: 0x4  bte xfers = 529
PID(1) 1485 fetch_op: 0x1  bte xfers = 356
PID(2) 1485 fetch_op: 0x2  bte xfers = 356
PID(3) 1485 fetch_op: 0x3  bte xfers = 789
PID(4) 1485 fetch_op: 0x4  bte xfers = 791
PID(1) 1980 fetch_op: 0x1  bte xfers = 477
PID(2) 1980 fetch_op: 0x2  bte xfers = 478
PID(3) 1980 fetch_op: 0x3  bte xfers = 1054
PID(4) 1980 fetch_op: 0x4  bte xfers = 1059
PID(1) 2475 fetch_op: 0x1  bte xfers = 693
PID(2) 2475 fetch_op: 0x2  bte xfers = 489
PID(3) 2475 fetch_op: 0x3  bte xfers = 1318
PID(4) 2475 fetch_op: 0x4  bte xfers = 1324
PID(1) 2970 fetch_op: 0x1  bte xfers = 928
PID(3) 2970 fetch_op: 0x3  bte xfers = 1585
PID(2) 2970 fetch_op: 0x2  bte xfers = 489
PID(4) 2970 fetch_op: 0x4  bte xfers = 1593
PID(1) 3465 fetch_op: 0x1  bte xfers = 1157
PID(3) 3465 fetch_op: 0x3  bte xfers = 1851
PID(2) 3465 fetch_op: 0x2  bte xfers = 489
PID(4) 3465 fetch_op: 0x4  bte xfers = 1859
PID(1) 3960 fetch_op: 0x1  bte xfers = 1392
PID(3) 3960 fetch_op: 0x3  bte xfers = 2119
PID(2) 3960 fetch_op: 0x2  bte xfers = 489
PID(4) 3960 fetch_op: 0x4  bte xfers = 2130
PID(1) 4455 fetch_op: 0x1  bte xfers = 1627
PID(3) 4455 fetch_op: 0x3  bte xfers = 2377
PID(2) 4455 fetch_op: 0x2  bte xfers = 489
PID(4) 4455 fetch_op: 0x4  bte xfers = 2391
PID(1) 4950 fetch_op: 0x1  bte xfers = 1852
PID(3) 4950 fetch_op: 0x3  bte xfers = 2640
PID(2) 4950 fetch_op: 0x2  bte xfers = 489
PID(4) 4950 fetch_op: 0x4  bte xfers = 2654
PID(1) 5445 fetch_op: 0x1  bte xfers = 2087
PID(3) 5445 fetch_op: 0x3  bte xfers = 2907
PID(2) 5445 fetch_op: 0x2  bte xfers = 489
PID(4) 5445 fetch_op: 0x4  bte xfers = 2922
PID(1) 5940 fetch_op: 0x1  bte xfers = 2322
```

Messages from the current loop as it runs

Figure 6-3 Cache Thrasher With I/O Sample Output (Part 3 of 4)

```
PID(3) 5940 fetch_op: 0x3  bte xfers = 3167
PID(2) 5940 fetch_op: 0x2  bte xfers = 489
PID(4) 5940 fetch_op: 0x4  bte xfers = 3186
PID(1) 6435 fetch_op: 0x1  bte xfers = 2553
PID(3) 6435 fetch_op: 0x3  bte xfers = 3432
PID(2) 6435 fetch_op: 0x2  bte xfers = 489
PID(4) 6435 fetch_op: 0x4  bte xfers = 3452
PID(1) 6930 fetch_op: 0x1  bte xfers = 2782
PID(3) 6930 fetch_op: 0x3  bte xfers = 3696
PID(2) 6930 fetch_op: 0x2  bte xfers = 489
PID(4) 6930 fetch_op: 0x4  bte xfers = 3716
PID(1) 7425 fetch_op: 0x1  bte xfers = 3011
PID(3) 7425 fetch_op: 0x3  bte xfers = 3964
PID(2) 7425 fetch_op: 0x2  bte xfers = 489
PID(4) 7425 fetch_op: 0x4  bte xfers = 3985
PID(1) 7920 fetch_op: 0x1  bte xfers = 3246
PID(3) 7920 fetch_op: 0x3  bte xfers = 4228
PID(2) 7920 fetch_op: 0x2  bte xfers = 489
PID(4) 7920 fetch_op: 0x4  bte xfers = 4251
PID(1) 8415 fetch_op: 0x1  bte xfers = 3476
PID(3) 8415 fetch_op: 0x3  bte xfers = 4492
PID(2) 8415 fetch_op: 0x2  bte xfers = 489
PID(4) 8415 fetch_op: 0x4  bte xfers = 4516
PID(1) 8910 fetch_op: 0x1  bte xfers = 3711
PID(3) 8910 fetch_op: 0x3  bte xfers = 4757
PID(2) 8910 fetch_op: 0x2  bte xfers = 489
PID(4) 8910 fetch_op: 0x4  bte xfers = 4782
PID(1) 9405 fetch_op: 0x1  bte xfers = 3945
PID(3) 9405 fetch_op: 0x3  bte xfers = 5026
PID(2) 9405 fetch_op: 0x2  bte xfers = 489
PID(4) 9405 fetch_op: 0x4  bte xfers = 5052
PID(1) 9900 fetch_op: 0x1  bte xfers = 4180
PID(3) 9900 fetch_op: 0x3  bte xfers = 5293
PID(2) 9900 fetch_op: 0x2  bte xfers = 489
PID(4) 9900 fetch_op: 0x4  bte xfers = 5319
PID(1) waiting to sync inner loop...
PID(3) waiting to sync inner loop...
PID(2) waiting to sync inner loop...
PID(4) waiting to sync inner loop...
PID(4) Fetch & Op result correct: 0x4
PID(1) Fetch & Op result correct: 0x1
PID(2) Fetch & Op result correct: 0x2
PID(3) Fetch & Op result correct: 0x3
PID(1) Random parameter ct_turbo 0 complete.
local router master pid 1 status from mid-round:
local status: 16a56d0034f21
Port   oflw bdDir dedTO tailTO retry cb_err sn_err  Bkt3  Bkt2  Bkt1 Bkt0
Port1:  0    0     0     0      0      0    255    a528     1     0 0
Port2:  0    0     0     0      0      0      0    a963     1     0 0
Port3:  0    0     0     0      0      0    255    a992     1     0 0
Port4:  0    0     0     0      1      0      0    a9b0  1ac0  137f f31
Port5:  0    0     0     0      0    150    255    aa45  137e  1afc 1213
Port6:  0    0     0     0      0      0      0    aab2     1     0 0
```

Messages from the current loop as it runs

Status of Router ports at the end of the current loop

Figure 6-4 Cache Thrasher With I/O Sample Output (Part 4 of 4)

Appendix A

Troubleshooting Link Failures

This appendix provides some general information for troubleshooting link failures. This information may help you to fix failures that are detected by the Router SSO test or the cache thrasher with I/O test.

A.1 About the Green LEDs on the Router Boards

A properly working link illuminates the green LED on the Router board. Failures are indicated by the following conditions:

- The green LED will not illuminate if nothing is connected to the link.
- The green LED will not illuminate if the link is broken.
- A poor link may negotiate successfully and illuminate the green LED, but the link may fail when it is being stress tested: This causes the link to shut down and turns off the green LED for the link.
- A marginal link may never fail (shut down), but it will report excessive errors in the Node NI error register, the Router error register, or the XBOW error register. The green LED will remain illuminated, but the diagnostics will read the error registers and report an excessive number of errors.

A.2 Troubleshooting Internal (CPOP) Link Failures

Internal link failures are most often caused by poor connections through one or more CPOP connectors. For Router link failures, the failing CPOP connectors will be on the Node boards or Router boards. For I/O link (XBOW) failures, the failing CPOP connectors will be on the Node boards.

If you suspect that an internal link problem is causing a link failure, perform the following procedure:

- If the green LED on a Router board is not illuminated, perform the following actions to check the CPOP connectors on the Router board:
  1. Remove the Router board and inspect the CPOP connector for damage. Also inspect the midplane connector area for foreign material.

2. Reinstall the Router board.

Caution: When you reinstall the Router board, screw each of the hex rods in halfway before you screw both rods in completely. Screwing in one rod completely before starting to screw in the other rod can damage the CPOP connector, which will cause more link failures.

- If the green LED on the Router board still does not illuminate or the corresponding XBOW green LED is not illuminated, perform the following actions to check the CPOP connectors on the Node board used in the link:

  1. Remove the corresponding Node board and inspect the CPOP connector for damage. Also inspect the midplane connector area for foreign material.
  2. Reinstall the Node board.

  Caution: When you reinstall the Node board, screw each of the hex rods in halfway before you screw both rods in completely. Screwing in one rod completely before starting to screw in the other rod can damage the CPOP connector, which will cause more link failures.

- If the link problem still exists, swap the Node board into a different slot.

  If the link error follows the Node board, the failing FRU is most likely the Node board.

- If the link error does not follow the Node board, swap the Router board into a different slot.

  If the link error follows the Router board, the failing FRU is most likely the Router board.

  If the link error does not follow the Router board, the midplane is most likely the failing FRU.

A.3 Troubleshooting External (Cable) Failures

External link failures apply only to Router link failures. Poor mating between a cable and the Router is the most likely cause of external link failures.

If the Router link green LED is not illuminated and you suspect that an external failure is the problem, perform the following procedure:

- Inspect the failing cable connections for even mating.
- Remove and inspect the cables for damage.
- Re-attach the cables.

Caution: When you re-attach the cable, support the connector with one hand and carefully tighten the top and bottom screws. Ensure that there is no play in the mated cable connection.

- If the problem persists, replace the cable.
- If the problem persists with a different cable, try connecting one end of the cable to an alternate Router port. This will narrow the failure to a single port of a single board.