1. Introduction

These notes describe release information for the IRIX 6.2 Diagnostics option.

Note: The 6.2 Diagnostics option applies to the POWER CHALLENGE 10000, POWER Onyx InfiniteReality 10000, POWER CHALLENGE, POWER Onyx RealityEngine, POWER Onyx Extreme, CHALLENGE, Onyx InfiniteReality, Onyx RealityEngine, Onyx Extreme, IRIS Crimson systems.

These notes do not apply to the POWER Series systems or to Professional Series systems.

Note: Packaged with these release notes is a separate sheet that contains the Software License Agreement. This software is provided to you solely under the terms and conditions of the Software License Agreement. Please take a few moments to review the Agreement.

This document contains the following chapters:

Appendix A, POWER CHALLENGE/Onyx/Onyx Extreme Standalone _ide_.

Appendix B, describes the CHALLENGE/Onyx Standalone _ide_ tests.

Appendix C, describes the IP17 Crimson Standalone Diagnostics.

1.1 Release Identification Information

Following is the release identification information for the IRIX 6.2 Diagnostics:

Software Option Product	Diagnostics
Version	6.2
Marketing Code	SC4-DIAG-6.2 (CD-ROM)
System Software Requirements	IRIX 6.2

1.2 Online Release Notes

After you install the online documentation for a product (the _relnotes_ subsystem), you can view the release notes on your screen.

If you have a graphics system, select ''Release Notes'' from the Help submenu of the Toolchest. This displays the _grelnotes_(1) graphical browser for the online release notes.

Refer to the _grelnotes_(1) reference page (man page) for information on options to this command.

If you have a non-graphics system, you can use the _relnotes_ command. Refer to the _relnotes_(1) for accessing the online release notes.

1.3 _Online Man Pages_

Printed copies of the reference pages (man pages) are _not_ included in this release of the diagnostics. You can view the man pages online by typing:

```
man commandname
```

1.4 _Product Support_

Silicon Graphics, Inc., provides a comprehensive product support maintenance program for its products.

If you are in the U.S. or Canada and would like support for your Silicon Graphics products, contact the Technical Assistance Center at
1-800-800-4SGI.

If you are outside these areas, contact the Silicon Graphics subsidiary or authorized distributor in your country.

2. Installation Information

This chapter lists supplemental information to the IRIS Software Installation Guide. The information listed here is product-specific; use it with the Installation Guide to install this product.

2.1 Diagnostics Subsystems

The Diagnostics include these subsystems:

diag.man.diag	This subsystem contains manual pages for Diagnostics.
diag.man.relnotes	This subsystem contains release notes for Diagnostics.
diag.sw.aso	This subsystem contains ASO Diagnostics software.
diag.sw.atm	This subsystem contains ATM Diagnostics software.
diag.sw.diag	This subsystem contains Base Diagnostics software.
diag.sw.eplex	This subsystem contains E-Plex Diagnostics software.
diag.sw.fddi	This subsystem contains FDDI-MEZ Diagnostics software.
diag.sw.hippi	This subsystem contains HIPPI Diagnostics software.
diag.sw.mco	This subsystem contains Multi-Channel Option Diagnostics software.

2.2 Diagnostics Subsystem Disk Space Requirements

This section lists the subsystems (and their sizes) of the Diagnostics option.

If you are installing this option for the first time, the subsystems marked ‘‘default’’ are the ones that are installed if you use the ‘‘go’’ menu item. To install a different set of subsystems, use the ‘‘install,’’ ‘‘remove,’’ ‘‘keep,’’ and ‘‘step’’ commands in inst to customize the list of subsystems to be installed, then select the ‘‘go’’ menu item.

Note: The listed subsystem sizes are approximate for a Power Challenge Onyx system. Refer to the IRIS Software Installation Guide for information on finding exact sizes.

Subsystem Name	Subsystem Size (512-byte blocks)
diag.man.diag (default)	304
diag.man.relnotes (default)	88
diag.sw.aso (default)	1092
diag.sw.atm (default)	216
diag.sw.diag (default)	100712
diag.sw.eplex (default)	136
diag.sw.fddi (default)	200
diag.sw.hippi (default)	452
diag.sw.mco (default)	488

2.3 Installation Method

All of the subsystems for Diagnostics can be installed using IRIX. You do not need to use the miniroot. Refer to the IRIS Software Installation Guide for complete installation instructions.

In this release, standalone diagnostics can be booted from the diagnostics CD-ROM. Using the standard CD-ROM installation procedure, with the CD-ROM drive at SCSI ID 5 (for example), you can boot ide as follows:

For CHALLENGE/Onyx InfiniteReality/Onyx/Onyx Extreme - IP19:

```
dksc(0,5,8)sashARCS dksc(0,5,7)stand/ide.IP19
```

For POWER CHALLENGE/POWER Onyx/POWER Onyx Extreme - IP21:
dksc(0,5,8)sash64 dksc(0,5,7)stand/ide.IP21

For CHALLENGE 10000/Onyx InfiniteReality 10000 - IP25:
dksc(0,5,8)sash64 dksc(0,5,7)stand/ide.IP25

2.4 Prerequisites

To run the 6.2 Diagnostics, your workstation must be running IRIX release 6.2.

2.5 Other Installation Information

The screen-compare gold files that contain the checksums of the images are installed by default. (Checksums are the files that are used for comparison with the generated screen images to determine whether each of the images is properly and correctly generated.) Gold files help detect the presence of failing graphics subsystems, and isolate the span(s) circuitry in which the failure occurs. The default screen-compare gold files should be sufficient for the vast majority of diagnostics requirements.

The screen-compare gold files that contain the full images used in the Screen Compare tests are in the diag.sw.noship subsystem (Silicon Graphics [SGI] internal use only) subsystem for installation. By using /usr/diags/bin/fbtool, you can compare failing screen-compare images with the full screen-compare gold files, and highlight the differences graphically.

Please note that the diag.sw.noship subsystem might require up to 46000 blocks of disk space, and so might not be suitable for systems other than internal SGI repair stations.

3. Changes and Additions

This release of the Diagnostics option installs on the CHALLENGE 10000, Oynx InfiniteReality 10000, POWER CHALLENGE, POWER Onyx, POWER Onyx Extreme, CHALLENGE, Onyx InfiniteReality, Onyx RealityEngine, Onyx Extreme, and IRIX Crimson systems.

The IRIX diagnostics user interface and operation are largely unchanged from the previous release. The diagnostics have the same ‘‘look and feel’’ as the server and graphics diagnostics in the IRIX 5.3 and 6.1 releases.

If you install the diagnostics on a RealityEngine2, or VTX system with Multi−Channel Option, please refer to the *Multi−Channel Option Owner’s Guide* (Document number 007−1812−030) and *Multi−Channel Option Installation Instructions* (Document number 108−7047−020) for detailed information.

3.1 Standalone CPU Diagnostics−POWER CHALLENGE/Onyx/Onyx Extreme

A total of 7 standalone tests are available to test the IP21 hardware in this release. These tests are built around a small diagnostic kernel called DK. When loaded on the system, the *dk* diagnostics reside in the /*stand* directory. Each test can be invoked by itself using the boot command listed in the table below. The *dk all* command can be used to invoke all these tests in sequence. Some of these tests run in mp mode. All output is to the ASCII console. These diagnostics are accessed from the PROM monitor with the following sequence:

```
boot dksc(1,1,0)/stand/<dk_test>
```

dk suite Command	Description	Boot Command
gparity	gparity random stress test	boot /stand/gparity
gparity_addr	gparity address stress test	boot /stand/gparity_addr
dcache	dcache stress test	boot /stand/dcache
wgatherer	write gatherer stress test	boot /stand/wgatherer
gcache3	gcache stress test	boot /stand/gcache3
gcache4	gcache stress test	boot /stand/gcache4
gcache4_10	more gcache stress test	boot /stand/gcache4_10
dk_all	all of the above	boot /stand/dk_all

Note: Due to the mp nature of the tests, some errors may point to incorrect slices. It is recommended that if these errors occur, CPU slices should be disabled and the test rerun to isolate the failing slice.

3.2 Standalone CPU Diagnostics–CHALLENGE 10000/Onyx InfiniteReality 10000

A total of 5 standalone tests are available to test the IP25 hardware in this release. These tests are built around a micro diagnostic kernel called MDK. When loaded on the system, the mdk diagnostics reside in the /stand directory. Each test can be invoked by itself using the boot command listed in the table below. Some of these tests run in mp mode. All output is to the ASCII console. These diagnostics are accessed from the PROM monitor with the following sequence:

```
boot dksc(1,1,0)/stand/<mdk_test>
```

mdk suite Command	Description	Boot Command
pcache	primary cache stress test	boot /stand/pcache
cerr_tpath	more primary cache test	boot /stand/cerr_tpath
alu	ALU stress test	boot /stand/alu
tlb	TLB stress test	boot /stand/tlb
sbkiller	random instruction stress test	boot /stand/sbkiller

3.3 System Diagnostics User Interface

The user interface has largely remained the same as in the previous release. This basic user interface is menu-driven and has the same ‘‘look and feel’’ as in previous releases.

The major features of the user interface are:

- o The type of g2aphics platform (for example, RealityEngine) is determined upon login (2s diag), and you are prompted by a query (RealityEngine example shown):

```
RealityEngineII Graphics: Pipe 0 5-span system
Automatically run RealityEngine diagnostics? (y or n)
```

o A top-level diagnostics menu appears on the screen if you type y.

The following items are usually included in the top-level menus for a system with graphics installed:

- Selection of graphics pipe for testing.
- Selection of full or quick system check.
- Selection of full or quick server check.
- Selection of full or quick graphics check.
- Selection of board-level graphics diagnostics.
- Selection of video, network and I/O options check.
- Selection of automatic demos.

The following example shows the RealityEngine2 top-level menu (the top-level menu is slightly different on different platforms):

```
RealityEngine/Onyx SYSTEMS DIAGNOSTICS    TEST TIME (hr:min)
------------------------------------------------------------

(Average test time based on a 32MB RealityEngine/Onyx 5-span
system)

 1- Pipe Select
 2- Quick System check .......................   0:26
 3- Full System check ........................   1:08/loop
 4- Quick Server check .......................   0:30
 5- Full Server check ........................   0:40/loop
 6- Quick Graphics check .....................   0:19
 7- Full Graphics check ......................   0:22
 8- GE10 Board Diagnostics ...................   Menu
 9- RM4/RM5 Board Diagnostics ................   Menu
10- DG2 Board Diagnostics ....................   Menu
11- CRC System Screen Compares ...............   0:12
12- System(Video/Network/IO) Options .........   Menu
13- View Results of diagnostic tests .........   Menu
14- Automatically run selected Graphics Demos
15- EXIT RealityEngine Systems Diagnostics

Please choose an item (1-15) >
```

A top-level diagnostics menu is also provided for server and Extreme systems.

Note: All graphics diagnostics must be run using an ASCII terminal (or equivalent) that is connected to one of the serial ports. The graphics diagnostics (‘‘Full System Check’’ or ‘‘Quick System Check’’) inevitably fail if run from the graphics console.

- When selecting any of the system or server check tests, you are asked to provide the system name for the Ethernet test. If a null entry is made, the Ethernet test is skipped. If you select the full system check, you are asked to enter the loop count (the loop count defaults to 8 when a null entry is made). If you select the full server check, you are asked to enter the loop count (the loop count defaults to 20 when a null entry is made).

- When selecting graphics board-level tests, a secondary menu is available for selecting:

  - Quick check of the board

  - Full check of the board

  - Run a specific test

  These tests are executed on the selected single graphics pipe only.

- When selecting the ‘‘System (Video/Network/IO) Options Check’’ menu item, a new secondary menu is displayed for selecting:

  - Sirius Video Option Menu

  - Multi-Channel Option Menu

  - HIPPI Option Test

  - ASO Option Test

  - ATM Option Test

  - FDDI Option Test

  - I/O Options Menu

o When selecting the ‘‘Sirius Video Options Menu’’ item and the Sirius Video Option is detected, a new menu is displayed for selecting:

- Run IDE Loop (IDE arg)
- Run JTAG Connectivity (JTAG) Tests
- Run Autocal
- Run Clock calibration
- Run Digital Functional (FUNC) Tests
- Run Analog (vo2_analog.csh) Tests
- Run IDE & Functional (FULL) Tests
- Initialize EEPROM & S/N entry
- Run Blending Tests
- Run VLAN Functional Tests
- Run PAB1 Tests
- Run VO2 GNG Cal & PreScreen Tests

The above Sirius tests are currently not in the 6.2 Diagnostics package. To successfully start and run these tests, the sirius and sirius noship software must be installed first.

o When selecting the ‘‘Multi-Channel Options Menu’’ item and the Multi-Channel Option is detected, the menu shown in section 3.7 is displayed. Please refer to section 3.7 for further detail.

o When selecting the ‘‘I/O Options Menu’’ menu item, a secondary menu is displayed for selecting:

  - Full check of system options
  - 1/4-inch cartridge/DAT tape test
  - 1/2-inch tape test
  - Printer test (IKON board)
  - 6-port test
  - Exabyte (8 mm Tape) test
  - CDROM test
  - DLT (sled/desk top/stacker) test

  During automatic testing of system options, you can now skip testing a defective tape drive or a tape drive with no tape inside. If the specified tape drive is not recognized, you are notified. For each tape test, you can specify the number of test loops or use the default of 2 loops.

3.4 Error Log Files

Log files are created by various scripts for recording the progress of each execution and any errors that might occur during execution. These files should be examined after each execution for errors. Before the scripts start executing the diagnostics, the existence of these log files is checked and they are erased if present. All log files are located in /usr/tmp and they are listed as follows:

- o quicksys.log - created for quick system check
- o fullsys.log(0,1,2) - created for full system check on the selected graphics pipe
- o quicksvr.log - created for quick server check
- o fullsvr.log - created for full server check
- o quickgr.log(0,1,2) - created for the quick graphics check on the selected graphics pipe
- o fullgr.log(0,1,2) - created for the full graphics check on the selected graphics pipe
- o ide.log(0,1,2) - created for the ide tests on the selected graphics pipe
- o options.log - created for options testing
- o pre.elog - created for quick system check with errors only
- o run.elog - created for full system check with errors only

o For RealityEngine2 and VTX systems:

- ge10.log(0,1,2) - created for the ge10 tests on the selected graphics pipe
- rm4.log(0,1,2) - created for the rm4/rm5 tests on the selected graphics pipe
- dg2.log(0,1,2) - created for the dg2 tests on the selected graphics pipe
- vs2.log(0,1,2) - created for the Multi-Channel Option tests on the selected graphics pipe
- (Hostname).crc.log(0,1,2) - created for screen-compares on the selected graphics pipe
- ccomploop.log(0,1,2) - created for screen-compares on the selected graphics pipe

3.5 System Diagnostics-POWER CHALLENGE, POWER Onyx, and POWER Onyx Extreme

3.5.1 Memory Tests The existing system level memory tests invoked by crash1 (memmain) and crash7 (memaddr.BIT) have been revised to lock the test area in memory so it doesn't get swapped to disk during the test.

A new test, tagram, is now also invoked by crash1. This test will run on each CPU and test the cache_tag RAMs by writing and reading several locations from each page of memory.

3.5.2 OS Test Suites A group of OS Test Suites have been added to the full system check to test the 64 bit functionality in the system.

awalk - address walk test

utlb - measure utlbmiss performance

tlbthrash - thrash 2nd level tlb

kids - fork and sproc memory test

cctest - test the sync register support of the CC chip

io - test SCSI ports, serial and parallel ports, network operations

iu test - quick test for the IP21 IU

netstress - HIPPI, FDDI, and EFAST tests
The SCSI port tests are normally turned off since they're destructive for mounted logical volumes on SCSI disks and any unmounted SCSI disks. To enable the SCSI port tests, type the following before selecting full system check:

```
setenv MFG_ONLY
```

3.5.3 RAID Support This release of the diagnostics supports the testing of RAID configurations. Before running the RAID scripts, be sure to mount all filesystems you don't want overwritten. The RAID tests destroy all data on RAID disks that are not mounted as part of a filesystem.

To run the RAID scripts, first type:

```
setenv MFG_ONLY
```

to turn on the scripts. If you want to run the scripts without user confirmation, type:

```
setenv OK_TO_TRASH
```

SGI does not recommend that you run the RAID scripts with the OK_TO_TRASH variable set.

Once you have set the environment variables, type:

```
mfg.raid.start
```

to actually run the tests. The RAID scripts verify that the RAID-specific hardware and software are functioning. The media on the RAID disks can also be tested by running the diskrand.start script and using the same environment variables (MFG_ONLY, OK_TO_TRASH) as mfg.raid.start. diskrand.start runs on all unmounted disks on your system, whether or not they are RAID disks, so be very careful when using this script.

Again, these tests destroy the data on non-mounted disks so be very careful when using these tests.

3.6 RealityEngine2 and VTX Diagnostics

For all RealityEngine2 and VTX platforms, the following items from the top menu can now be performed on a pre-selected pipe:

Full system check
CRC screen compares
Automatic demos

To select a pipe, choose item 1 from the top menu.

Some new tests have been added for this release. All the diagnostics for GE10/GE10V, RM4/RM5, and DG2 are listed in the following tables with brief descriptions of their operations.

3.6.1 ge10 (GE10/GE10V ide Test)

The following GE10 diagnostics are listed in the order they are run when full system check or full graphics check is selected.

Note: In this section, information for ‘‘GE10’’ applies as well to the GE10V board.

ide Command	Test Number	Description	GFX Board
PIO_regs	310	FCG(GE10)/PIO reg test	GE10
JTAG_control	300	CP2/GEs JTAG connectivity test	GE10
geconnect	320	GEs connectivity test	GE10
PIO_fifo_regs	330	PIO FIFO registers test	GE10
CP_ucode_scan	360	CP2 SRAM scan test	GE10
PB_scan	390	GEF diagnostic registers scan test	GE10
GEF_scan	460	GEF functionality	GE10

ide Command	Test Number	Description	GFX Board
PB_PIO	400	PB/PIO connectivity test	GE10
PB_dram_PIO	410	GEF DRAM validity test	GE10
PIO_PIO	420	CP2/GEF/host full connectivity test	GE10
ODMA_swap	440	ODMA/host swap functionality test	GE10
ODMA	450	ODMA/host functionality test	GE10
GEF_PIO	470	GEF functionality test	GE10
GEF_conv_ODMA	480	GEF functionality test	GE10
GEF_ODMA	490	GEF functionality test	GE10
GE_intr	510	GE10/GE10V	GE10
IDMA_ODMA	520	IDMA/ODMA functionality test	GE10
IDMA_PIO	530	IDMA functionality test	GE10
FCG_func	550	FCG functional Test	GE10
CP2_func	560	CP2 functional Test	GE10
I860_func	590	I860 functional Test	GE10

3.6.2 rm4 (RM4/RM5 ide Test) The following RM4/RM5 ide tests are listed in the order they are run when full system check or full graphics check is selected:

ide Command	Test Number	Description	GFX Board
rmreset	100	Reset the RM4/RM5	RM4/RM5
rmconfig	101	Configuration verification	RM4/RM5
rmconnect	102-109	Connectivity test	RM4/RM5
rmreadback	111	IMP/IB readback	RM4/RM5
rmreset	100	Reset the RM4/RM5	RM4/RM5
syscon	119-121	TBus/RBus connectivity	RM4/RM5
rmtbus	110	TBus signature test	RM4/RM5
rmimp	115	IMP signature	RM4/RM5
rmimpmem	112-114	IMP (framebuffer) memory test	RM4/RM5
rmtexmem	116-118	TA/TD (texture)	RM4/RM5

3.6.3 dg2 (DG2 ide Test) The following DG2 ide tests are listed in the order they are run when full system check or full graphics check is selected:

ide Command	Test Number	Description	GFX Board
dg_init	10	Initialization test	DG2
dg_connect	10	JTAG connectivity	DG2
dg_fma	10	Function manager SRAM test	DG2
dg_regs	10	Register test	DG2
dg_init	10	Initialization	DG2
dg_list	10	HV list test	DG2
dg_xilinx	10	Xilinx load-read-verify test	DG2
dg_xmapmem	10	Xmap-only memory	DG2
dg_xmapcrc	10	Xmap CRC test	DG2
dg_xmapdata	10	Xmap->DAC	DG2
dg_promcheck	10	EEPROM header and checksum verification test	DG2

3.7 Multi-Channel Option (VS2) Diagnostics

Multi-Channel Option diagnostics are included in the 6.0.1 Diagnostics. To run MCO diagnostics, do not run RealityEngine diagnostics when prompted after logging in as *diag*. Instead, enter vs2 *at the diag prompt*. *A menu similar to the following menu is displayed on the screen*:

```
RealityEngine Multi-Channel Option Diagnostics - Test Time(mi:se)
-----------------------------------------------------------------
(these average test times are based on 10-span RE/Onyx systems)

 1- Pipe Select
 2- Quick Check of MCO..................................  1:35
 3- Full Check of MCO................................... 10:45
 4- MCO Board Initialization Test.......................  0:20
 5- MCO Register Test...................................  0:40
 6- MCO DAC Test........................................  0:35
 7- MCO Read DAC Test Register..........................  0:30
 8- MCO Load Gamma Constant.............................  0:25
 9- MCO XILINX Test.....................................  0:35
10- MCO VOF Loader......................................  0:40
11- MCO Data Path Test..................................  7:00
12- View Test Output
13- EXIT from menu

Please choose an item (1-13) >
```

This menu lists all the MCO diagnostics available in this release. *For detailed descriptions of these tests*, *refer to the online reference pages* (*man pages*) *for each test*. *MCO diagnostics test descriptions are also included in the Multi*-*Channel Option Installation Instructions*.

In a multi-pipe MCO configuration system, test(s) run on the default pipe 0, and test results are logged in /*usr*/*tmp*/*vs2*.*log0*.

If you wish to run MCO diagnostic test(s) on a pipe other than pipe 0, you need to use ‘‘Pipe Select’’ to switch to the desired pipe number. For example, to look at the test output for pipe 2, choose the ‘‘Pipe Select’’ option from the menu, then specify pipe 2 before choosing item 12, ‘‘View Test Output.’’

3.8 Onyx Extreme Diagnostics

All the diagnostics for the Extreme graphics are listed in the tables with brief descriptions of their operations.

3.8.1 gr2 (GR2 ide Test)

GR2 diagnostics are run on the POWER Onyx Extreme systems.

ide Command	Test Number	Description	GFX Board
gr2_reset	1503	GR2 reset test	GR2
gr2_videoclk	1505	Select specified video clock	GR2
gr2_unstall	NA	Unstall HQ2/GE7 and set hq ucode PC to 0	GR2
gr2_inithq	1560	Initialize HQ2 internal registers	GR2
gr2_wrfifo	1550	Write FIFO token number	GR2
gr2_tram	2120	General memory test program	GR2
gr2_fgbgram	2130	Read/write to	GR2
gr2_rdram	2110	Read specified locations from RAM	GR2
gr2_wrram	2100	Write to locations in RAM	GR2
gr2_rdram12	2140	Read RAM1/2 of GE7	GR2
gr2_xcol	1130	Check XMAP5	GR2
gr2_txmap	1140	Test XMAP5 functions	GR2
gr2_txmap_clut	1150	Test color lookup table	GR2
gr2_wrx	1100	Write to XMAP5	GR2
gr2_rdx	1120	Read from XMAP5	GR2

ide Command	Test Number	Description	GFX Board
gr2_xrgb	1110	rite RGB value	GR2
gr2_lt	1520	oad timing ables	GR2
gr2_lr	1525	nitialize VC1 fter timing able set by r2_lt	GR2
gr2_initvc1	1530	oad timing ables into VC1	GR2
gr2_vc1reg	1720	est VC1 egister	GR2
gr2_vc1_sram	1740	est VC1 SRAM	GR2
gr2_rvc1	1710	ead VC1 egister	GR2
gr2_wvc1	1700	rite VC1 egister	GR2
gr2_tvc1	1730	ead/write to C1 registers	GR2
gr2_initclock	1515	nitialize clock enerator	GR2
gr2_vinitdac	900	nitialize RAM AC	GR2
gr2_vrgb	930	rite RGB values o DAC	GR2
gr2_vwrdac	920	rite values to ACs	GR2
gr2_vrddac	940	ead value from	GR2
gr2_vtestdac	910	rite/read alues to DACs	GR2
gr2_wrhqint	1540	rite HQ2 nternal egisters	GR2
gr2_rdhqint	1545	ead HQ2 nternal	GR2
gr2_wrconfig	1500	et board onfiguration egister	GR2
gr2_genlock	1510	urn Genlock on r off est VM2 itplanes	GR2

gr2_bp	300	GR2

gr2_zb	310	Test ZB4 z-buffer	GR2
gr2_load_ucode	1570	Load GE/HQ microcode	GR2
gr2_shram	600	Test the shram	GR2
gr2_ram12	601	Test internal GE	GR2

ide Command	Test Number	Description	GFX Board
gr2_seedrom	602	Test the seed ROM	GR2
gr2_sqrom	603	Test the square root of ROM	GR2
gr2_gebus	604	Test the ring bus in GE7	GR2
gr2_gefloat	606	Test the integer/floating point operations for GE7	GR2
gr2_ge7fifo	605	Test the GE RE FIFO	GR2
gr2_geseq	609	Test the GE sequencer logic	GR2
gr2_gefpucond	608	Test the GE FPU condition flags	GR2
gr2_gespfcond	607	Test the GE7 SPF condition modes	GR2
gr2_ge7test	600	Test full GE7	GR2
gr2_hq2test	100	Test HQ2	GR2
gr2_getVBver	1580	Check video back end version	GR2

3.9 InfiniteReality Diagnostics

The diagnostics for InfiniteReality are not part of the 6.2 diagnostics release. It can be installed as part of the 6.2 eoe subsystem software. Please refer to the on-line man page, irsaudit, for the full detail on the InfiniteReality diagnostics.

3.10 Diagnostics for Network and Audio Options

These diagnostics are newly added to the 6.2 Diagnostics package. Currently, these diagnostics can be selected from the ‘‘System (Video/Network/IO) Options’’ menu.

3.10.1 XPI FDDI Diagnostics The following commands are available for testing the XPI FDDI option.

- dang - Writes and reads the DANG registers. Please refer to the man page for more detail.

- first - Downloads and executes diagnostic firmware for testing the XPI FDDI option. Please refer to the man page for more detail.

- mac.check - Verifies and assign addresses to the XPI FDDI option. Please refer to the man page for more detail.

3.10.2 ASO Diagnostics Please refer to the on-line man page, samzdiags for the full detail on the diagnostics for the ASO option.

3.10.3 EPLEX Diagnostics Please refer to the on-line man page, epdiags for the full detail on the diagnostics for the EPLEX option.

3.10.4 HIPPI Diagnostics The following commands are available for testing the HIPPI option.

- blast - Sends 1 gigabyte of random data through the HIPPI board and measures the time of data transfer. Please refer to the man page for more detail.

- dma test - Writes and reads source and destination memories using the DMA transfer engines. Please refer to the man page for more detail.

- mem test - Reads the ID register located on the F chip for the HIPPI board; reads the VMECC chip configuration register; reads the VMECC error cause register. Writes, reads and compares random data for the selected memory areas (source or destination or both) with/without parity. Also does an address test. Please refer to the man page for more detail.

- rawdi - Writes random data packets through the HIPPI interface; reads them back and compares for data integrity problems. Please refer to the man page for more detail.

- rawdi2 - Performs the same operations as rawdi and further randomize the size of the packets. Please refer to the man page for more detail.

- stress a16 - Reads the ID register located on the F chip for the HIPPI board; reads the VMECC chip configuration register. Writes and reads the VMECC location 0x4000 in the small window space checking for data miscompares. Please refer to the man page for more detail.

- stress fci - Reads the ID register located on the F chip for the HIPPI board; reads the VMECC chip configuration register. Writes and reads the VMECC location 0x1014 in the small window space checking for data miscompares. Please refer to the man page for more detail.

4. Known Problems and Workarounds

4.1 Standalone Diagnostics-POWER CHALLENGE/Onyx/Onyx Extreme

- The SCSI tests only test the primary SCSI bus on the master IO4.

- The Standalone Diagnostics can only be booted on the serial port of IP19 systems with RealityEngine2 graphics. Attempts to boot in the graphics textport will hang. Use "setenv console d<CR> init<CR>" from the prom menu before booting Standalone Diagnostics.

4.2 System Diagnostics-POWER CHALLENGE/Onyx/Onyx Extreme

- The diagnostics login script calls /*usr*/*gfx*/*stopgfx* to bring the system graphics to a known state before testing. Remotely logging in as *diag* onto a system running diagnostics in the presence of *Xsgi* and running diagnostics automatically causes the ongoing graphics test to fail.

  To check the state of a system running diagnostics, log in as *guest* or *root*, which does not cause this problem.

- On a busy network, the crash5 script may lose up to 10% of the network packets due to excessive traffic on the net. This is not usually a problem with the hardware in the system, but is due to a busy network. If the crash5 script indicates packet losses, manually ping the remote system to verify that you are able to communicate with it. If you are able to ping the system with no packet loss, it indicates that the previous failure was caused by a busy net and you can ignore the crash5 failure.

4.3 RealityEngine2 and VTX Diagnostics

- When running the RealityEngine2 and VTX board-level tests, typing <Ctrl-C> to halt *ide* might hang *ide*. If you encounter this problem, reboot the system to get rid of the *ide* process. This bug is intermittent.

4.4 Onyx Extreme Diagnostics

- o The screen compares may hang if the DANG _ide_ tests are previously executed. If you encounter this problem, reboot the system and run the screen compares again.

5. Bug Fixes

5.1 Standalone Diagnostics−CHALLENGE, Onyx, and Onyx Extreme

- o The epc plptest test no longer times out waiting for PPORT DMA interrupt. (#286933)

- o The cache48 and cache49 tests no longer complains about no hint messages when it fails. (#287056)

- o The scsi all test no longer hangs intermittently. (#316908)

- o IP25 Standalone ide no longer has boot problems. (#411925)

- o Standalone ide no longer hangs during boot on dual−IO4 systems. (#411929)

- o Memory tests no longer hang on systems with RE graphics. (#411930)

- o dang mdma works on IP25 systems with Extreme graphics. (#427033)

- o Standalone ide no longer prints failure messages for SCIP controllers on startup. (#431517)

1. POWER CHALLENGE/Onyx/Onyx Extreme Standalone ide

Note: Starting with the 6.2 Diagnostics release, the standalone ide no longer resides in the /stand directory. It now resides in the /usr/stand directory. The command to boot ide is: boot /usr/stand/ide.

This appendix includes detailed descriptions of the POWER CHALLENGE/Onyx/Onyx Extreme Standalone ide tests, organized by board type.

The POWER CHALLENGE standalone ide is based on the Personal IRIS/IRIS Indigo ide rather than on the POWER Series ide. As such, the user interface is similar to the IRIX−level graphical ide, which was originally based on the same code.

Although tests can be invoked individually, usually prepared test scripts are run, ensuring more complete coverage while requiring less typing.

At the most basic level, the standalone ide user interface is simple−individual tests and scripts can be run by typing their name at the command line. Although looping, condition execution, report verbosity level changing, user programmability, and many other advanced features are available, there is no need to master the advanced features to test the system completely.

Upon starting, ide prints a list of available test scripts, then waits at the command line for user input. At this point, a command or script name can be typed to run a test, or you can type help for more information. The help command without parameters returns the message:

```
help [ all | commands | sets | loop | ifthen | cmd | dbg ]
```

help with a parameter–help commands, for example–gives more information on the chosen topic. In the example given, ide prints a list of all available ide commands, a list long enough to require several screens of information. If the output requires more than one screen to display, the help command pauses at the bottom of each screen until you press the space bar or <Return> key.

Report levels determine the amount of information each test prints to the screen when run. The default report level of 2 usually prints the name of the test and whether it passed or failed. As you increase the report level, the amount of information returned increases. The report level range is 1 (least information) to 5 (debug level–almost too much information). To change the report level, type report=<level#> at the command line. For example, report=5 sets reporting to its most verbose. Unless tests fail and more information is needed, the default level of 2 is probably the most useful.

Although more complex looping structures are available (see help loop for more information), the only two needed for most test situations are the repeat and while commands.

repeat is easier to use–type repeat <count#> <command> and <command> is executed <count#> times. The command line repeat 5 mem3 runs the mem3 command 5 times.

while has the form while (condition) <command>. condition is a numeric value; it means that as long as condition is non–zero, <command> executes again and again. The while command is especially useful for scope loops. Although it can terminate if condition goes non–zero, the most common use of while ignores that possibility by setting condition to a constant. For example, while (1) mem3 repeats the mem3 command until you turn off the power or press the <Ctrl–C> key.

One of the nicer features of ide is that for looping purposes, commands can be grouped–ide treats both command and {command1; command2; command3} as a single command. For example, the command repeat 10 {mem2; mem3; mem1} loops 10 times, each time running the mem2, mem3, and mem1 tests in the order given.

Although all examples so far have used individual test commands, test scripts, each calling several individual test commands, can be treated in the same way. *repeat* *5* *io all* runs the *io all* script 5 times. In general, you use test scripts to test an individual board or subsystem thoroughly, running the individual tests only if a failure occurs. The currently defined test scripts are summarized below.

MC3 (MEMORY) Board The memory board has the test scripts *memall* and *memfast*. The names are largely self-explanatory; *memfast* tests memory reasonably quickly, while *memall* is more complete but requires longer to finish.

IO4 (I/O) Board The I/O tests run quickly enough that there is only one script defined (*io all*). It runs all the IO4 tests, working from the Ebus connector outward.

System Testing Besides the individual board scripts, there is a master script, *everest all*, that runs everything; it incorporates the *memall*, and *io all* scripts. A faster script, *ev quick*, is also available–it runs the quick version of the various board–level tests, providing coverage nearly as complete as *everest all* in a fraction of the time.

In case you are unsure which tests to run, the *scripts* command reprints the list of available scripts given when *ide* is started, giving you both the other script names and short descriptions of their purpose.

In this release, the error logging feature is supported. *ide* now keeps pass/fail counters for all tests and displays them when *dumplog* is entered at the prompt. The command *clearlog* can be used to clear the pass/fail counters to zeros. To display/modify the current setting of error logging, use the command *errlog*. Error logging is turned on by default.

1.1 _MC3 ide Guide_

To run the MC3 _ide_ diagnostics:

- o Boot _ide_

- o The default report level is 2. Set the report level by typing the following:

  ```
  report=#
  ```

  where # is any number from 1 to 5.

level 5	Debugging messages displayed. Don’t need this much detail.
level 4	Prints out memory locations as they are written. Selecting this level slows down testing time.
level 3	Prints out 1-line functional descriptions within tests. This is probably the most useful level for general use.
level 2	Print out only errors, titles, and pass/fail.
level 1	Print out only titles and pass/fail.

  Level _n_ prints out all messages for level _n_ and below.

- o Set the modes of operations for running the tests:

  - qmode [on | off]

    For the memory tests, quick mode tests every _n_th byte instead of every byte, where _n_ varies from 96 to 7680 depending upon the test. The goal in quick mode is to test 16 GB in about 10 minutes- and this is accomplished by testing every _n_th byte. _n_ varies depending upon how fast or slow a test was timed to run.

- c_on_error [on | off]

  For the memory tests, the ‘‘on’’ setting continues the test even when an error has been encountered. Setting the mode to ‘‘off’’ stops the tests after the first error.

o Run memall and memfast.

These are two defined commands. memall runs in normal mode while memfast runs in quick mode. memall runs all commands (memall: mem1, mem2, mem3, mem4, mem5, mem8, mem9, mem10, mem16 in this order), while memfast runs just the faster tests (memfast: mem3, mem5, mem8, mem9, mem10, mem16 in this order).

o There are currently 15 memory tests/commands. They are explained below:

---

mem1 - Read the mc3 configuration registers (real fast)

The following registers are probed:

```
reg  test description
---  -----------------------------------------------------

00 Read the BankEnable
01 Read BoardType
02 Read RevLevel
03 Read AccessControl: endianness, subBlockOrder, ebus=64bitsOrNot
04 Read MemoryErrorInterrupt
05 Read EBUSErrorInterrupt
06 Read BIST result
07 Read DRSC timeout
0a Read LeafControlEnable
Read leaf regs 10-24, 30-33 (leaf0), 50-64, 70-73 (leaf1)
```

mem1 is similar to mem14, which is the pod-mode dmc command.

---

mem2 - Memory sockets connection test (similar to IP17's mem1) (real fast)

The memory sockets connection test writes patterns to the first 2 KB of each configured leaf and then reads them back. By writing 2 KB, all SIMMs are ensured of being written to regardless of the interleaving factor specified.

If the pattern read back does not match, the socket is assumed to have a connection problem.

---

mem3 - Walking address test (similar to IP17's mem2) (veryrealreal fast)

This is a traditional test that checks for shorts and opens on the address lines. Address lines that are greater or equal to the most significant address lines of the memory bounds are not tested. Testing is done by byte read/writes from first_address up to last_address.

---

mem4 - Write/read data patterns (similar to IP17's mem3) (slow)

This test does word read/writes of all-1's and all-0's patterns. It shows if all addresses appear to be writable, and that all bits may be set to both 1 and 0. However, it provides no address error or adjacent-bits-shorted detection. The flow is as follows:

(w0), u(r0,w1), d(r1,w5a), u(r5a,ra5), d(ra5)-word and byte (read as: write 0 to all locations, read 0 and write 1 to all locations in ascending order, read 1 and write 5a to all locations in descending order, read 5a and write a5 to all locations in ascending order, read a5 from all locations in descending order)

mem13 does byte read/writes in the same pattern. These tests are separate because the byte read/writes take a long time.

---

mem5 – Address in address memory test (slow)

This is a traditional, heuristic, rule–of–thumb, ‘‘address–in–address’’ memory test. It also puts the complement of the address in the address, and makes passes in both ascending and descending addressing order. There are both full–memory store–then–check passes, as well as read–after–write passes (with complementing).

---

mem7 – MarchX (slow)

This test is described in van de Goor’s book, <u>Testing Semiconductor Memories</u>, and has the following flow:

(w0), u(r0,w1), d(r1,w0), (r0)

It detects address decoder faults, stuck–at faults, transition faults, coupling faults, and inversion coupling faults (see van de Goor for definitions).

---

mem8 – MarchY (slow)

This test is described in van de Goor’s book, <u>Testing Semiconductor Memories</u>, and has the following flow:

(w0), u(r0,w1,r1), d(r1,w0,r0), (r0)

It detects address decoder faults, stuck–at faults, transition faults, coupling faults, and linked transition faults (see van de Goor for definitions).

---

mem9 – Memory with ecc test (similar to IP17’s mem6) (slow)

This test writes to memory via uncached space and reads back through cached space (ECC exceptions enabled). Although it provides a simple level of ECC checking, its main function is to verify that cached and uncached memory addresses are accessing the same area of physical memory. The test values used are address–in–address and inverted address–in–address patterns, so a certain amount of address uniqueness checking is done as well.

---

mem10 − Cache write−through memory test (similar to IP17’s mem9) (slow)

This is a traditional, heuristic, rule−of−thumb, ‘‘address−in−address’’ memory test. It also puts the complement of the address in the address, making passes in ascending order only. All of memory is stored and then checked. All reads and writes are made through K0 seg, so the reads and writes are cached. However, since the size of main memory exceeds the cache sizes, all data is written to main memory and then read back. This is not a particularly thorough test, and it depends upon a good cache to function correctly, but it is fast, at least compared to the other full−memory tests.

---

mem11 − User−specified pattern/location write/read test (similar to IP17’s mem7)

Type mem11 without any arguments to see the usage.

```
Usage: mem11 [−b|h|w] [−r] [−l] [−c] [−v 0xpattern] RANGE
```

This test allows the technician to fill a range of memory with a specified test value and read it back, done as a series of byte (−b), half−word (−h), or word (−w) writes and reads. If the −v option is not used to select the test pattern, an address−in−address pattern is used instead. −r does read−only and does not do any writes. −l loops forever. −c runs in cached memory space; the default is to run in uncached space.

---

mem12 - Decode a bad address into slot, leaf, bank, simm

Usage: mem12 [-a 0xaddress] [-b xxxxx] [-s x]

-b expects a hex number showing which bits are bad. For
example, if bits 0 and 2 are bad, enter -b 0x5
-s 1, 2, or 4 for byte, half-word, or word
-b defaults to 0x0 and -s defaults to 4

For example, to decode address 0x4000 with bad bits 0 and 2
and it's a word, type:

```
mem12 -a 0x4000 -b 0x5 -s 4
```

---

mem16 - Knaizuk Hartmann Memory Test (3 min/ 128 MB)

This algorithm is used to perform a fast but non-exhaustive memory test. It will test a memory subsystem for stuck-at faults in both the address lines as well as the data locations. The algorithm breaks up the memory to be tested into 3 partitions. Partition 0 consists of memory locations 0, 3, 6...; partition 1 consists of memory locations 1, 4, 7...; and partition 2 consists of locations 2, 5, 8 .... The partitions are filled with either an all 1's pattern or an all 0's pattern. By varying the order in which the partitions are filled and then checked, this algorithm manages to check all combinations of possible stuck-at faults.

---

ena_bnk, dis_bnk - enable/disable one bank at a time

Interactively asks the user for slot, leaf, and bank to act on. Do not run dis_bnk unless you know what you are doing. Otherwise, you will most likely hang the system.

1.2 IO4 IDE Guide

- o Boot ide

- o The default report level is 2. Set the report level by typing the following:

  ```
  report=#
  ```

  where # is any number from 1 to 5.

level 5	Debugging messages displayed. Don’t need this much detail.
level 4	Prints out memory locations as they are written. Selecting this level slows down testing time.
level 3	Prints out 1-line functional descriptions within tests. This is probably the most useful level for general use.
level 2	Print out only errors, titles, and pass/fail.
level 1	Print out only titles and pass/fail.

Level <u>n</u> prints out all messages for level <u>n</u> and below.

o Set the modes of operations for running the tests:

- qmode [on | off]

  All current IO4 tests run fast enough that there is no difference between quick and long test modes for the IO4. If the total elapsed time for running all IO4 tests ever exceeds 10 minutes, quick mode will be enabled for the IO4.

- c_on_error [on | off]

  For the IO4 tests, the ‘‘on’’ setting continues the test even when an error has been encountered. Setting the mode to ‘‘off’’ stops the tests after the first error.

o Run <u>io all</u>.

This command runs all working/known bug-free IO4 tests that do not require human intervention. Any mostly working but possibly buggy tests, as well as any tests requiring a human to interpret the results, are not included.

o There are currently tests for the following areas of the IO4 board: IO4 interface, VME adapter, SCSI adapter, and EPC adapter.

The detailed tests are listed below.

1.2.1 IO4 Interface ---------

----

io4_regtest - read/write test of IO4 registers

This is a basic read/write test for the IO4 registers. It does tests and address-in-address testing for:

```
    IO4_CONF_LW
    IO4_CONF_SW
    IO4_CONF_ADAP
    IO4_CONF_INTRVECTOR
    IO4_CONF_GFXCOMMAND
    IO4_CONF_ETIMEOUT
    IO4_CONF_RTIMEOUT
    IO4_CONF_INTRMASK
```

Although these are not the only IO4 registers, they are the only ones that may safely be read/write tested.

---

io4_pioerr - IO4 PIO bus error test

Attempts to generate an error interrupt by attempting a write to IO adapter 0 (nonexistent). This tests the IO4 error-generation capability and the IO4-to-IP error path.

---

mapram_test - Read/Write test of IO4 map ram

As the name implies, tests the IO4 mapping RAM as a small memory array.

Tests memory with pattern read/write, address-in-address, and marching 1's test patterns.

1.2.2 VME Adapter

---

fregs - Test VMECC registers

Basic Read/Write test for the F Chip registers, running through large window space. Verifies operation for:

```
    FCHIP_VERSION_NUMBER
    FCHIP_MASTER_ID
    FCHIP_INTR_MAP
    FCHIP_FIFO_DEPTH
    FCHIP_FCI_ERROR_CMND
    FCHIP_TLB_BASE
    FCHIP_ORDER_READ_RESP
    FCHIP_DMA_TIMEOUT
    FCHIP_INTR_MASK
    FCHIP_INTR_SET_MASK
    FCHIP_INTR_RESET_MASK
    FCHIP_SW_FCI_RESET
    FCHIP_IBUS_ERROR_CMND
    FCHIP_TLB_FLUSH
    FCHIP_ERROR
    FCHIP_ERROR_CLEAR
    FCHIP_TLB_IO 0 - 7
    FCHIP_TLB_EBUS 0 - 7

    (32 registers currently tested)
```

---

vmeregs - Test VMECC registers

Basic Read/Write test for the VMECC registers. Verifies
operation for:

```
    VMECC_RMWMASK
    VMECC_RMWSET
    VMECC_RMWADDR
    VMECC_RMWAM
    VMECC_RMWTRIG
    VMECC_ERRADDRVME
    VMECC_ERRXTRAVME
    VMECC_ERRORCAUSES
    VMECC_ERRCAUSECLR
    VMECC_DMAVADDR
    VMECC_DMAEADDR
    VMECC_DMABCNT
    VMECC_DMAPARMS
    VMECC_CONFIG
    VMECC_A64SLVMATCH
    VMECC_A64MASTER
    VMECC_VECTORERROR
    VMECC_VECTORIRQ 1 - 7
    VMECC_VECTORDMAENG
    VMECC_VECTORAUX0
    VMECC_VECTORAUX1
    VMECC_IACK 1 - 7
    VMECC_INT_ENABLE
    VMECC_INT_REQUESTSM
    VMECC_INT_ENABLESET
    VMECC_INT_ENABLECLR
    VMECC_PIOTIMER
    0x1388
    0x1390
    0x1398
    0x13A0
    0x13A8
    0x13B0
    0x13B8
    0x13C0
    0x13C8
    0x13D0
    0x13D8
    0x13E0
    0x13E8
    0x13F0
    0x13F8

    (54 registers currently tested)
```

----

vmeintr - Test VMECC self interrupts

Places a handler on the appropriate vector and forces the VMECC to generate an interrupt. Checks to make sure that the interrupt both reaches the CPU and activates the proper handler.

---

vmeberr - Test VMECC bus errors

This test ensures that the VMECC can time out for all sections of A24/A32 addresses.

First, A24 addressing is used. All sections except one are made to respond as slaves, and a PIO access to the non-responding slave section is done. This should generate a timeout.

This procedure is repeated for all 16 sections of the A32 addressing range.

---

vmedma - Test VMECC DMA Engine

Transfers data between controller memory and host memory by DMA; no disk data is involved (and no disk need be connected).

The controller imposes some constraints on what can be done with this test: it transfers only a single sector (512 bytes).

---

cddata - CDSIO board internal/external data loopback test

Does loopback testing of all channels of the CDSIO 6-port board. Pretty much a direct port of the IP5 family version of this test.

---

cdintr - Test CDSIO interrupts

Generates interrupts on the CDSIO board and verifies that they reach the CPU. Again, a direct port of the IO5 family test.

1.2.3 SCSI Adapter

---

s1_regtest − Register Read/Write test for s1 chip

This is a basic Read/Write test for the S1 chip registers.
It does tests and address−in−address testing for:

```
    S1_INTF_R_SEQ_REGS 0 − 0xF
    S1_INTF_R_OP_BR_0
    S1_INTF_R_OP_BR_1
    S1_INTF_W_SEQ_REGS 0 − 0xF
    S1_INTF_W_OP_BR_0
    S1_INTF_W_OP_BR_1
```

(36 registers currently tested)

Although these are not the only S1 registers, they are the only ones that may safely be used by Read/Write tests.

---

regs_95a − Register read/write test for wd95a chip

This is a basic read/write test for the wd95a chip registers. In setup mode, the test writes a's and 5's to the Sleep Countdown register while all other registers are just read. In normal mode, all registers are read and no registers are written. While there are many registers, the Sleep Countdown register was the only register where it was safe to write values.

---

scsi_intr − SCSI interrupt test

This tests the wd95a's ability to send an interrupt to the CPU and have the system respond correctly. The wd95a is programmed to interrupt upon a SCSI reset command. The reset command is sent and then the system is checked to make sure it correctly ‘‘saw’’ the interrupt.

---

epc_regtest − Register Read/Write test for epc chip

Basic Read/Write test for the EPC chip registers, including the Parallel Port registers. Registers tested:

```
    EPC_IIDDUART0
    EPC_IIDDUART1
    EPC_IIDENET
    EPC_IIDPROFTIM
    EPC_IIDSPARE
    EPC_IIDPPORT
    EPC_IIDERROR
    EPC_EADDR0
    EPC_EADDR1
    EPC_EADDR2
    EPC_EADDR3
    EPC_EADDR4
    EPC_EADDR5
    EPC_TCMD
    EPC_RCMD
    EPC_TBASELO
    EPC_TBASEHI
    EPC_TLIMIT
    EPC_TTOP
    EPC_TITIMER
    EPC_RBASELO
    EPC_RBASEHI
    EPC_RLIMIT
    EPC_RTOP
    EPC_RITIMER
    EPC_PPBASELO
    EPC_PPBASEHI
    EPC_PPLEN
    EPC_PPCTRL
```

This is a good basic test for the Parallel Port; for more thorough testing, a test fixture is required.

----

epc_nvram - NVRAM Read/Write test

Does Read/Write pattern and address-in-address testing for all the NVRAM accessible to the EPC chip. Although the NVRAM is physically on the RTC chip, it occupies a separate address space and is accessed differently; hence, the separate test.

----

epc_rtcreg - RTC register/NVRAM Read/Write test

Read/Write test for the RTC registers and the small amount of NVRAM in the RTC address space portion of the RTC chip. Registers tested:

```
    NVR_SEC
    NVR_SECALRM
    NVR_MI
    NVR_MINALRM
    NVR_HOUR
    NVR_HOURALRM
    NVR_WEEKDAY
    NVR_DAY
    NVR_MONTH
    NVR_YEAR
```

NVRAM tested is in the address range 0xE - 0x3F.

---

epc_rtcinc - RTC clock increment test

Tests the ability of the RTC chip to handle time-of-day transitions. Sets the RTC to a known time and date (last second of the year), waits one second, and checks to make certain that the time and date have changed correctly.

---

epc_rtcint - RTC Interrupt generation test

Tests to make certain that the RTC can correctly generate Alarm, Periodic, and Update interrupts. Validates the path from the RTC chip to the IP board's master CPU.

---

duart_loopback - Duart loopback test

Attempts to configure and test all available serial ports. Does loopback testing at all baud rates for each port tested. Normally uses internal loopback, but if invoked with "duart_loopback -e" assumes that an external loopback fixture is being used.

---

enet_xmit - Ethernet transmit/receive test (with internal loopback on)

Transmits 9 packets and receives them with the LXT internal loopback mode on. The transmit and receive status bytes are checked against expected values, and the data in each byte of each packet is verified against what was expected. The following is a short description of each of the 9 packets:

```
packet 0: 50 data bytes, walk 0 through the bytes
packet 1: 50 data bytes, walk 1 through the bytes
packet 2: 50 data bytes, alternating 55's and aa's in alternating bytes
packet 3: 60 data bytes, alternating 0's and ff's in alternating bytes
packet 4: 50 data bytes, all 55's
packet 5: 1 data byte, short packet
packet 6: 130 data bytes, decrementing pattern starting with ff
packet 7: 3 data bytes, short packet
packet 8: 130 data bytes, decrementing pattern starting with cc
```

---

enet_colctr − Read ethernet collision counters Test

Transmits the same 9 packets as enet_xmit but with LXT and EDLC in normal mode so packets actually go out onto the net. The user is given instructions to run ttcp −r −s on one other system and ttcp −t −s <machine #1> on sytem #2 in UNIXr. Then, by running this test, collisions occur and the collision counter counts are displayed after each packet is sent. There is not really a fail status in this test unless the packets cannot be transmitted. The test fails if the collision counter counts being displayed never increment.

---

epc_plptest − Parallel Port Write Test

Fills a buffer with printable characters and attempts to DMA it out the parallel port. Detects the presense of a printer; if a printer is present, configures the port to use the ‘‘BUSY’’ mode and writes to the printer using BUSY handshaking to prevent buffer overrun. If no printer is present, uses the ‘‘SACK’’ mode and writes as fast as the DMA engine sends characters.

If a printer is present, it detects printer errors and reports them; in all cases, it detects DMA timeout errors or failure to generate the DMA completion interrupt.

If an external printer is present, these characters should be printed out:

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[]^_
```

1.2.4 DANG Adapter ---------

---

dang_regtest − DANG Register Read/Write Test

This test does read/write verification of all DANG PIO registers that are safely writable (some, such as the Master DMA start register or the interrupt mask register, are not).

Runs a series of bit−pattern tests, marching ones and zeros, and address−in−address patterns on the DANG chip PIO registers. For this test to function correctly, the basic DANG Ibus interface must be working.

Patterns used:

    0x0
    0xFFFFFFFF
    0x55555555
    0xAAAAAAAA
    0xA5A5A5A5
    0x5A5A5A5A
    Marching 1’s (32 patterns) − 0x1, 0x2, . . . 0x80000000
    Marching 0’s (32 patterns) − 0xFFFFFFFE, 0xFFFFFFFD, . . . 0x7FFFFFFF
    Address in Address
    Inverse Address in Address

Registers currently tested:

```
    DANG_UPPER_GIO_ADDR
    DANG_MIDDLE_GIO_ADDR
    DANG_BIG_ENDIAN
    DANG_GIO64
    DANG_PIPELINED
    DANG_GIORESET
    DANG_AUDIO_ACTIVE
    DANG_AUDIO_SLOT
    DANG_PIO_WG_WRTHRU
    DANG_DMAM_MAX_OUTST
    DANG_DMAM_CACHE_LINECNT
    DANG_DMAS_MAX_OUTST
    DANG_DMAS_CACHE_LINECNT
    DANG_INTR_ERROR
    DANG_INTR_GIO_0
    DANG_INTR_GIO_1
    DANG_INTR_GIO_2
    DANG_INTR_DMAM_COMPLETE
    DANG_INTR_PRIV_ERR
    DANG_INTR_PAUSE
    DANG_INTR_BREAK
    DANG_WG_LOWATER
    DANG_WG_HIWATER
    DANG_WG_FULL
    DANG_WG_PRIV_LOADDR
    DANG_WG_PRIV_HIADDR
    DANG_WG_GIO_UPPER
    DANG_WG_GIO_STREAM
    DANG_WG_PAUSE
    DANG_WG_STREAM_ALWAYS
```

Sample Error Messages:

```
    Exception Accessing DANG chip!
    Failed to set DANG Intr mask. Expected: 0 Got: 1f
    Failed testing DANG register DANG_WG_FULL. Expected 400 Got 4F0
    Failed DANG reg address test. Reg: DANG_WG_FULL
Expected: 52018
    Got: 0
```

----

dang_gr2ram - GIO Bus/Gr2 Shared Ram Test

This test is designed to stress the GIO bus interface and prove that the basic GIO bus read/write functionality is present. As such, although it requires a working Express graphics board, dang_gr2ram only stresses the Express shared RAM area.

Like dang_regtest, dang_gr2ram runs a series of bit-patterns, marching ones and zeros, and address-in-address tests, though its target this time is the entire Express shared RAM area.

dang_gr2ram stresses the DANG's Ibus interface, the basic DANG IO configuration, the DANG GIO bus interface, and the GIO bus data/address lines in addition to the Express shared RAM.

Patterns used:

    0x0
    0xFFFFFFFF
    0x55555555
    0xAAAAAAAA
    0xA5A5A5A5
    0x5A5A5A5A
    Marching 1's (32 patterns) - 0x1, 0x2, . . . 0x80000000
    Marching 0's (32 patterns) - 0xFFFFFFFE, 0xFFFFFFFD, . . . 0x7FFFFFFF
    Address in Address
    Inverse Address in Address

Sample Error Messages:

```
    Failed Gr2 shared ram test. Index: 0x400 Expected:
0x55555555 Got: 0x5A
```

---

dang_mdma - DANG Master DMA Module Test

The DMA Master module test has a reasonably simple basic design: set up a Master DMA transfer either to or from the Gr2 RAM space, verify that it completed and the DMA complete interrupt was generated, and validate the transferred data.

To fully test the DMA module, however, requires that all the corner cases are covered, which requires many different transfers in a variety of DMA modes.

Currently, the dang_mdma test has 38 test cases, covering GIO bus dynamic and static address modes, transfers to and from the Gr2, data patterns, address-in-address data, single-line, multi-line, and multi-page transfers, transfers with no offset or stride, transfers with offset or stride, transfers with both offset and stride, etc.

dang_mdma stresses the DANG's Ibus interface, the basic DANG IO configuration, the DANG GIO bus interface, the Express shared RAM area, and the Dang DMA Master Interrupt logic as well as the DANG Master DMA module proper.

One cache line static address tests

    host to gr2, 0x55555555
    gr2 to host, 0xAAAAAAAA

One cache line data tests

    host to gr2, 0x55555555
    host to gr2, 0xAAAAAAAA
    host to gr2, 0xFFFFFFFF
    gr2 to host, 0x55555555
    gr2 to host, 0xAAAAAAAA
    gr2 to host, 0xFFFFFFFF

One partial cache line static address tests

    host to gr2, 0x55555555
    gr2 to host, 0xAAAAAAAA

One partial cache line data tests

    host to gr2, 0x55555555
    host to gr2, 0xAAAAAAAA
    host to gr2, 0xFFFFFFFF
    gr2 to host, 0x55555555
    gr2 to host, 0xAAAAAAAA
    gr2 to host, 0xFFFFFFFF

One cache line address tests

    host to gr2, address in address
    host to gr2, inverse address in address
    gr2 to host, address in address
    gr2 to host, inverse address in address

One partial cache line address tests

    host to gr2, address in address
    host to gr2, inverse address in address
    gr2 to host, address in address
    gr2 to host, inverse address in address

Multiple block data tests using stride and offset

    host to gr2, 0x55555555 (stride)
    gr2 to host, 0xAAAAAAAA (stride)
    host to gr2, 0x55555555 (stride+offset)
    gr2 to host, 0xAAAAAAAA (stride+offset)

Multiple block address tests using stride and offset

    host to gr2, address in address (stride)
    gr2 to host, inverse address in address (stride)
    host to gr2, address in address (stride+offset)
    gr2 to host, inverse address in address (stride+offset)

Full data buffer static address tests

    host to gr2, 0x55555555
    gr2 to host, 0xAAAAAAAA

Full data buffer data tests

    host to gr2, 0x55555555
    gr2 to host, 0xAAAAAAAA

Full data buffer address tests

    host to gr2, address in address
    gr2 to host, inverse address in address

Sample Error Messages:

```
    data setup problem: small static pattern 1
    small static pattern 1: (H to G) timed out waiting for
DANG interrupt
    xfer data, small static pattern 1 (H to G), line 0x1
byte 0x1: src 0x55
    dest 0x5A
    small static pattern 2 (G to H) wrong interrupt level:
was 20, sb 35
    small static pattern 3 (H to G) DMA xfer not complete
```

---

dang_wg - DANG Write Gatherer Test

The DANG write gatherer test, dang_wg, has three subtests, which test the WG FIFO RAM, the Host to DANG WG interface, and the WG interrupts.

Since each subtest must be working for the next to function correctly, the subtests are run in the order given, and are not written to be run separately.

The WG FIFO RAM tests runs bit patterns and address-in-address data through the FIFO and out to the Gr2 shared RAM area via the GIO bus. The test method is to use the DANG_WG_PAUSE register to pause output, fill the WG FIFO RAM, and then enable output. After the WG FIFO has drained, the data in the Gr2 shared RAM is verified.

Patterns used:

```
    0x55555555
    0xAAAAAAAA
    0xFFFFFFFF
    0x5a5a5a5a
    0xa5a5a5a5
    0x0
    Address in Address
    Inverse Address in Address
```

The Host to WG test sends data from the Host CPU’s write gatherer module using all four of the possible addressing modes (relative, absolute, streaming, and streaming always) and verifies that the data reaches the area of Gr2 shared RAM it was aimed at. Since the FIFO RAM has already been verified, the individual tests are less exhaustive.

Patterns used:

```
    Relative Mode:  Address in Address
    Absolute Mode:  Inverse Address in Address
    Streaming Always Mode:0x5A5A5A5A
    Streaming Mode: 0xA5A5A5A5
```

Finally, the Interrupt tests use the Host to DANG WG interface to generate the FIFO high, FIFO full, FIFO low, and privilege violation interrupts. In each case, the data used to fill the buffer is verified after the interrupt has been generated.

Pattern used:

    Address in Address

dang_wg stresses the DANG's Ibus interface, the basic DANG io configuration, the Write Gatherer FIFO RAM, the DANG GIO bus interface, the Express shared RAM area, and the WG Interrupt logic as well as the DANG Write Gatherer module proper.

Sample Error Messages:

```
    DANG wg fifo not empty - had 23 words
    DANG wg fifo: bad word count - was 0x2ff, sb 0x3ff
    DANG fifo write through data error: addr 0x1000, was 0x0
sb 0x55555555
    DANG host wg data error: addr 0x80, was 0x0 sb 0x80
    wrong dang wg interrupt level - was 0x20, sb 0x73
    dang wg interrupt bit bad - was 0x0, sb 0x4
    no wg fifo hi interrupt!
    no wg fifo low interrupt!
    no wg fifo full interrupt!
    no wg privileged interrupt!
```

----

dang_status - DANG Status display utility

Displays the current state of the DANG chip. This is the same routine called by the DANG tests when an error is detected.

When possible, status information is given both as numeric values and named states. An example status is:

```
    +      dang_pio_err: 0x200
    +        11..8: dang_pio_err_version: 0x2
    +      dang_dmam_status: 0x1
    +         0: dang_dma_stat_busy
    +         1: dang_dma_stat_dir: 0x0
    +      dang_dmam_status, dma if 1 - ibus to fifo:
dang_dma_if_idle <0>
    +      dang_dmam_status, dma if 2 - fifi to gio :
dang_dma_if_idle <0>
    +      dang_dmam_status, dma if 3 - gio to fifo :
```

```
dang_dma_if_idle <0>
    +      dang_dmam_status, dma if 4 - fifo to ibus:
dang_dma_if_idle <0>
    +      dang_dmam_err: 0x0
    +      dang_dmas_status: 0x0
    +      dang_dmas_status, dma if 1 - ibus to fifo:
dang_dma_if_idle <0>
    +      dang_dmas_status, dma if 2 - fifi to gio :
dang_dma_if_idle <0>
    +      dang_dmas_status, dma if 3 - gio to fifo :
dang_dma_if_idle <0>
    +      dang_dmas_status, dma if 4 - fifo to ibus:
dang_dma_if_idle <0>
    +      dang_dmas_err: 0x0
    +      dang_intr_status: 0x408
    +         3: dang_istat_wg_flow
    +        10: dang_istat_giostat
    +      dang_wg_status: 0x1
    +         0: dang_wgstat_idle
    +         4..3: dang_wgstat_fill: 0x0
    +         7..5: dang_wgstat_wext: 0x0
    +         9..8: dang_wgstat_drain: 0x0
```

---

dang_gr2read - Gr2 Read utility

"dang_gr2read slot# adapter# gr2address"

Reads an address on the GIO bus. A GIO bus "peek" routine.

Requires IO4 slot number, dang adapter number, and Gr2
offset (base of the Gr2 shared RAM area is offset 0).

All numbers may be in decimal or hexadecimal - hex numbers
should be preceded by "0x".

"dang_gr2read 11 5 0" would read Gr2 location 0 on DANG
adapter 5 of the IO4 board in slot 11; so would
"dang_gr2read 0xb 5 0".

---

dang_gr2write - Gr2 Write utility

"dang_gr2read slot# adapter# gr2address pattern"

Writes one word to a specified location in the GIO bus. The syntax is as given for dang_gr2read.

"dang_gr2write 11 5 0 0x55555555" would write 0x55555555 hex to Gr2 location 0 on DANG adapter 5 of the IO4 board in slot 11.

---

dang_gr2readloop

"dang_gr2readloop slot# adapter# gr2address loopcount"

dang_gr2writeloop

"dang_gr2writeloop slot# adapter# gr2address pattern loopcount"

Scope loop versions of the read and write utilities. dang_gr2readloop requires a loop count after the standard dang_gr2read parameters; similarly, dang_gr2writeloop requires a loop count following the standard dang_gr2write parameters.

"dang_gr2writeloop 11 5 0 0x55555555 100000" would write 0x55555555 to Gr2 location 0 on DANG adapter 5 of the IO4 board in slot 11 one hundred thousand times.

2. CHALLENGE/Onyx Standalone ide

Note: Starting with the 6.2 Diagnostics release, the standalone ide no longer resides in the /stand directory. It now resides in the /usr/stand directory. The command to boot ide is: boot /usr/stand/ide.

This appendix includes detailed descriptions of the CHALLENGE/Onyx Standalone ide tests, organized by board type.

Note: Due to interactions with the graphics hardware, when ide is run on the graphics console, all TLB tests skip TLB slots 0 and 1, and the tlb9 test is skipped entirely.

The CHALLENGE standalone ide is based on the Personal IRIS/Indigo ide rather than on the POWER Series ide. As such, the user interface is similar to the IRIX-level graphical ide, which was originally based on the same code.

Although tests can be invoked individually, usually prepared test scripts are run, ensuring more complete coverage while requiring less typing.

At the most basic level, the standalone ide user interface is simple-individual tests and scripts can be run by typing their name at the command line. Although looping, condition execution, report verbosity level changing, user programmability, and many other advanced features are available, there is no need to master the advanced features to completely test the system.

Upon starting, ide prints a list of available test scripts, then waits at the command line for user input. At this point, a command or script name can be typed to run a test, or you can type help for more information. The help command without parameters returns the message:

```
help [ all | commands | sets | loop | ifthen | cmd | dbg ]
```

help with a parameter–help commands, for example–gives more information on the chosen topic. In the example given, ide prints a list of all available ide commands, a list long enough to require several screens of information. If the output requires more than one screen to display, the help command pauses at the bottom of each screen until you press the space bar or <Return> key.

Report levels determine the amount if information each test prints to the screen when run. The default report level of 2 usually prints the name of the test and whether it passed or failed. As you increase the report level, the amount of information returned increases. The report level range is 1 (least information) to 5 (debug level–almost too much information). To change the report level, type report=<level#> at the command line. For example, report=5 sets reporting to its most verbose. Unless tests fail and more information is needed, the default level of 2 is probably the most useful.

Although more complex looping structures are available (see help loop for more information), the only two needed for most test situations are the repeat and while commands.

repeat is easier to use–type repeat <count#> <command> and <command> is executed <count#> times. The command line repeat 5 ip3 runs the ip3 command 5 times.

while has the form while (condition) <command>. condition is a numeric value; it means that as long as condition is non–zero, <command> will execute again and again. The while command is especially useful for scope loops. Although it can terminate if condition goes non–zero, the most common use of while ignores that possibility by setting condition to a constant. For example,while (1) ip3 repeats the ip3 command until you turn off the power or press the <Ctrl–C> key.

One of the nicer features of ide is that for looping purposes, commands can be grouped–ide treats both command and {command1; command2; command3} as a single command. For example, the command repeat 10 {ip2; fpu3; mem1} loops 10 times, each time running the ip2, fpu3, and mem1 tests in the order given.

Although all examples so far have used individual test commands, test scripts, each calling several individual test commands, can be treated in the same way. repeat 5 io all runs the io all script 5 times. In general, you use test scripts to test an individual board or subsystem thoroughly, running the individual tests only if a failure occurs. The currently defined test scripts are summarized below.

IP19 (CPU) Board Currently, there are test scripts for the tlb (tlball), the fpu (quickfpu, fpuall), the cache (quickcache, cacheall), and the ip support functionality (ipall). To test the complete IP19 board, run ip19 or quickip19.

MC3 (MEMORY) Board The memory board has the test scripts memall and memfast. The names are largely self−explanatory; memfast tests memory reasonably quickly, while memall is more complete but requires longer to finish.

IO4 (I/O) Board The I/O tests run quickly enough that there is only one script defined (io all). It runs all the IO4 tests, working from the Ebus connector outward.

System Testing Besides the individual board scripts, there is a master script, everest all, that runs everything; it incorporates the ip19, memall,and io all scripts. Due primarily to the length of the cache and memory test components, this test takes several hours to complete. A faster script, ev quick, is also available−it runs the quick version of the various board−level tests, providing coverage nearly as complete as everest all in a fraction of the time.

In case you are unsure which tests to run, the scripts command reprints the list of available scripts given when ide is started, giving you both the other script names and short descriptions of their purpose.

In this release, the error logging feature is supported. ide now keeps pass/fail counters for all tests and displays them when dumplog is entered at the prompt. The command clearlog can be used to clear the pass/fail counters to zeros. To display/modify the current setting of error logging, use the command errlog. Error logging is turned on by default.

2.1 Multiprocessor Execution

In this release, the multiprocessor support designed into the standalone *ide* is enabled.

The *ip*, *tlb*, *cache*, and *fpu* tests can be run on all enabled CPUs serially without selecting each CPU individually. To invoke these tests for multiprocessors, the *mp* command is provided. The CPU selection for multiprocessor diagnostics execution is stored and managed through a global set of CPUs called *runcpus*. The CPU set can be modified using the *runall*, *runon*, and *runexcp* commands. All enabled CPUs are selected by default when *ide* is loaded.

To see which CPUs are selected at any time, the command *runmode* displays the current contents of *runcpus*.

Caution: *After exiting ide, always reset the system hardware before rebooting ide, booting other standalone tools, or booting IRIX.*

2.2 IP19 ide Guide

To run the IP19 _ide_ diagnostics:

- o Boot _ide_

- o The default report level is 2. Set the report level by typing the following:

  ```
  report=#
  ```

  where # is any number from 1 to 5.

level 5	Debugging messages displayed. Don’t need this much detail.
level 4	Prints out memory locations as they are written. Selecting this level slows down testing time.
level 3	Prints out 1-line functional descriptions within tests. This is probably the most useful level for general use.

level 2	Print out only errors, titles, and pass/fail.
level 1	Print out only titles and pass/fail.

Level n prints out all messages for level n and below.

o There are currently 4 classes of IP19 tests: IP, TLB, FPU, and CACHE.

tlb(1 - 9)	Tests the TLB in R4K.
fpu(1 - 14)	Tests the FPU in R4K.
cache(1 - 48)	Tests the primary and secondary cache for R4K.
cache49	Short version of cache48.
cstate(0 - 21)	Individual cache state tests in cache48.
ip(1 - 9)	Tests IP19 components not covered by TLB, FPU, and CACHE.

Note: These test command tokens can be used to invoke each test individually on the master CPU slice. See below for the procedure to start execution of IP19 tests on multiple processors.

o For MP execution, the IDE defaults to run on all enabled CPU slices. To edit this global set of CPUs (called runcpus), the following scripts are provided:

runadmin	Displays all scripts that alter 'runcpus'
runmode	Displays the current contents fo 'runcpus'
runall	Run on all enabled CPU slices
runon i	Run on CPU slice i only
runexcp i	Run on all enabled CPU slices except CPU slice i
addslice i	Add CPU slice i
rmslice i	Remove CPU slice i

In addition, the following scripts are provided to operate solely on the CPU slices specified for the global set runcpus.

ipall	Invokes tests ip1 through ip9.
tlball	Invokes tests tlb1 through tlb9.
fpuall	Invokes tests fpu1 through fpu14.
cacheall	Invokes tests cache1 through cache48.
ip19	Invokes all IP, TLB, FPU, and CACHE tests.
quickfpu	Invokes tests fpu1 through fpu13, skipping fpu14.
quickcache	Invokes tests cache1 through cache44, then cache47 and cache49, skipping cache45, 46, and 48.
quickip19	Invokes ipall, tlball, quickfpu, quickcache and some memory tests−mem3, mem6, and mem10.

Finally, a script is implemented to facilitate debug and repair using IP19 test:

mp ‘‘TEST’’	Invokes TEST on the CPU slices in the global set, runcpus.

Note   TEST must be a test command token recognized by IDE. It cannot be a built−in script command. For example, mp ‘‘tlball’’ is not legal and will cause internal IDE errors.

- In the 5.1.2 release, the following scripts are implemented:

  mpstress

  This script makes use of several test modules specifically written to perform functions required for testing cache coherency in a multiple processor environment of upto 8 CPUs. The following test modules are intended for use only by the 'mpstress' script:

mpinval	invalidates all caches
mpmem_wr	initialize cached/uncached memory
mpmem_rd	verifies cache/uncached memory
mpchk_cstate	verifies cache state

  Any use of these test modules outside of the 'mpstress' scripts within IDE may produce unpredictable results.

  mpcachesz [1 | 4]

  This script verifies the installation of 1 or 4 MB scache. It is useful for verification of the actual scache size against the desired scache size which is assumed to have been correctly written to the EAROM for the boot master CPU. If the actual scache size does not match the desired scache size, it is reported as an error and the desired scache size will be automatically written to the EAROM for the failed CPU. The contents of EAROMs for CPUs other than the boot master are not easily reversible so it is important to enter the correct value for the desired scache size when invoking 'mpcachesz'. Problems may also arise if a mixture of 1 MB and 4 MB scache are present in the same system.

- To find out the results after running IP19 tests, use the following built-in commands:

dump_log	Displays cumulative results for each test
clear_log	Clears cumulative results for all tests
errlog [ON/OFF]	Displays, sets/resets cumulative error logging

```
o A brief description of each test and the possible
  errors are provided below for your reference.  The
  number preceding each error message identifies each
  error uniquely and its format should be interpreted as
  follows:

  01ccnnn   01 - the board ID for IP19
            cc - the hint for failed component(s)

                 01 - A chip
                 02 - D chip
                 03 - CC chip
                 04 - Primary cache
                 05 - Secondary cache
                 06 - R4400
                 07 - Primary or secondary cache
                 08 - TLB
                 09 - FRU

            nnn - the error ID"
```

---

ip1 (local_regtest) − Check CC local registers

Basic write/read test for the local registers. The registers tested are limited to the following:

EV_WGDST	Write gatherer destination
EV_WGCNTRL	Write gatherer control
EV_IP0	Interrupts 63 − 0
EV_IP1	Interrupts 127 − 64
EV_CEL	Current execution level
EV_IGRMASK	Interrupt group mask
EV_ILE	Interrupt level enable
EV_ERTOIP	Error/timeout interrupt
EV_ECCSB_DIS	ECC single−bit error disable

The read−only registers are read and their contents are reported. These registers are:

EV_SPNUM	Slot/Processor info
EV_SYSCONFIG	System configuration
EV_HPIL	Highest pending interrupt level
EV_RO_COMPARE	RTC compare
EV_RTC	Real time clock
EV_WGCOUNT	Write gatherer count

Possible error:

0103001: Local register %s R/W error : Wrote 0x%llx Read 0x%llx

---

ip2 (cfig_regtest) − Check configuration registers

Basic write/read test for the configuration registers. The registers tested are limited to the following:

EV_PGBRDEN	Write gatherer destination
EV_PROC_DATARATE	Write gatherer control
EV_WGRETRY_TOUT	Interrupts 63 − 0
EV_CACHE_SZ	Interrupts 127 − 64
EV_CMPREG0 − 3	Timer comparator registers

Note that the timer comparator registers are checked via the read−only RTC compare register.

Possible error:

0103002: Configuration register %s R/W error : Wrote 0x%llx Read 0x%llx

---

ip3 (bustags_reg) - Check bus tags

This test calculates the size of bus tag space based on the size of the secondary cache. Then it performs basic write/read test on the bus tags.

Possible error:

```
0103003: Bus tag addr 0x%x R/W error : Wrote 0x%x Read 0x%x
```

---

ip4 (counter) - Check R4K count/compare test

This test performs a basic write/read test on the R4K compare register first. Then it generates an interrupt using the R4K count and compare registers.

Possible errors:

```
0106001: Compare register data error : Expected 0x%x Got 0x%x
0106002: Incorrect contents in count register : Expected 0x%x Got 0x%x
0106003: Phantom count/compare interrupt received
0106004: No count/compare interrupt received : Count 0x%x Compare 0x%x
```

---

ip5 (intr_level0) - Check IP19 level 0 interrupt

This test generates level 0 interrupts at different priority values and execution levels. It also checks multiple level 0 interrupts occurring at the same time.

Possible errors:

```
0103004: Level 0 interrupt pending failure : Priority 0x%x IP0 0x%llx IP1
         0x%llx
0103005: Level 0 highest priority interrupt level failure : HPIL 0x%llx
0103006: Level 0 interrupt not indicated in Cause register 0x%x
0103007: Level 0 interrupt pending not cleared : IP0 0x%llx IP1 0x%llx
0103008: Level 0 highest priority interrupt level not cleared : HPIL 0x%llx
0103009: Level 0 interrupt pending not cleared in Cause register : Cause 0x%x
010300a: Level 0 current exec level mismatch : Wrote 0x%llx Read 0x%llx
010300b: Level 0 interrupt not detected when priority >= CEL : Cause 0x%
010300c: Level 0 interrupt detected when priority < CEL : Cause 0x%x
010300d: Level 0 interrupt pending not cleared : Cause 0x%x
010300e: Level 0 highest priority interrupt level incorrect : Expected 0x7f Got
         0x%llx
010300f: Level 0 multiple interrupt pending incorrectly indicated : Expected
         0x6000000000000009 Got 0x%llx
0103010: Level 0 multiple interrupt pending incorrectly indicated : Expected
         0x9000000000000006 Got 0x%llx
0103011: Level 0 multiple interrupt pending not cleared : IP0 0x%llx
0103012: Level 0 multiple interrupt pending not cleared : IP1 0x%llx
0103013: Level 0 multiple interrupt HPIL not cleared : HPIL 0x%llx
0103014: Level 0 multiple interrupt Cause not cleared : Cause 0x%x
0103015: Level 0 interrupt did not occur : Priority 0x%x
```

----

ip6 (intr_level3) - Check IP19 level 3 interrupt

This test generates level 3 interrupts using the EV_ERTOIP
register.

Possible errors:

```
0103016: Level 3 interrupt pending not detected in CAUSE
0103017: Interrupting error not detected in ERTOIP
0103018: Level 3 interrupt pending not cleared in Cause : Cause 0x%x
0103019: ERTOIP not cleared via write to CERTOIP : ERTOIP 0xllx
010301a: Level 3 interrupt did not occur : ERTOIP 0x%llx
```

---

ip7 (intr_timer) - Check IP19 RTSC and interval timer

This test generates level 1 interrupt by writing a value
into the EV_CMPREG configuration registers so that the RTSC
will reach this value and interrupt the processor.

Possible errors:

```
010301b: Invalid timer interrupt occurred
010301c: Interval timer interrupt did not occur
010301d: Timer interrupt pending not cleared in Cause : Cause 0x%x
```

---

ip8 (intr_group) - Check IP19 processor group interrupt

This test generated level 0 interrupts using different
processor groups at different priority levels including
broadcast interrupts.

Possible errors:

```
010301e: Group interrupt pending not set correctly in EV_IP0 : Expected 0x%llx
                Got 0x%llx
010301f:  Group highest priority interrupt level failure : HPIL 0x%llx
0103020: Group interrupt not indicated in Cause register 0x%x
0103021: Group interrupt pending not cleared : IP0 0x%llx IP1 0x%llx
0103022: Group highest priority interrupt level not cleared : HPIL 0x%llx
0103023: Group interrupt pending not cleared in Cause register : Cause 0x%x
0103024: Group interrupt did not occur : group 0x%x priority 0x%x
0103025: Group interrupt pending not cleared in Cause : Cause 0x%x
```

---

ip9 (wr_gatherer) - Check IP19 write gatherer

This test exercise the write gatherer on each IP19 by
performing writes of command-only, mixed command/data and
data-only streams to the two
32-word buffers of the write gatherer. The data streams are
each flushed either manually or automatically to memory for
verification.

Possible errors:

```
0103026: Write gatherer command only write : addr 0x%x expected 0x%x got
                0x%x
0103027: write gatherer mixed write : addr 0x%x expected 0x%x got 0x%x
0103028: Write gatherer data only write : addr 0x%x expected 0x%x got 0x%x
```

----

tlb1 (tlb_ram) - Test R4K TLB as RAM

Tests the TLB as a small memory array. Checks to see if all
the read/write bits can be toggled and that all undefined
bits read back zero.

Possible errors:

```
0108001: TLBHI      entry %d R/W error: Wrote 0x%x Read 0x%x
0108002: TLBLO even entry %d R/W error: Wrote 0x%x Read 0x%x
0108003: TLBLO odd  entry %d R/W error: Wrote 0x%x Read 0x%x
```

----

tlb2 (tlb_probe) - Check TLB functionality

Sets up all the TLB slots and then probes them with matching
addresses. Checks to ensure that there is a response for
each valid address.

Possible error:

```
0108018: TLB probe error : Expected entry %d Got entry %d
vpnum %d addr 0x%x
```

----

tlb3 (tlb_xlate) - Check TLB address translation

Tests for correct virtual to physical translation via mapped
TLB entries. Sets the virtual address to user segment and
uncached.

Possible errors:

```
010801b: TLB entry %d unexpected exception for addr 0x%x
010801c: TLB entry %d translation error at addr 0x%x : Wrote %d Read %d
```

---

tlb4 (tlb_valid) − Check TLB valid exception

Tests to see if TLB invalid accesses generate exceptions.
Maps the TLB entries to invalid addresses in k2seg and
attempts to access them.

Possible errors:

```
0108016: TLB entry %d invalid exception VADDR error : Expected 0x%x Got
                0x%x
0108017: TLB entry %d invalid exception didn’t occur
```

---

tlb5 (tlb_mod) − Check TLB modification exception

This test sets up the TLB to map each page as non−writable,
then attempts to write to each of the mapped pages. It
verifies that an exception is generated for each write
attempt.

Possible errors:

```
010800b: TLB %s entry %d mod exception VADDR error : Expected 0x%x Got
                0x%x
010800c: TLB %s entry %d mod exception didn’t occur
010800d: TLB %s entry %d unexpected exception during mod
010800e: TLB %s entry %d mod error : Wrote 0x%x Read 0x%x
```

---

tlb6 (tlb_pid) − Check TLB refill exception

Tests each TLB slot by attempting access with both matching
and non−matching process ID. It verifies that matching pid
accesses are allowed and non−matching pid accesses generate
exceptions.

Possible errors:

```
0108015: TLB %s entry %d unexpected exception with matching pid 0x%x
0108016: TLB %s entry %d refill exception VADDR error : Expected 0x%x Got
                0x%x
0108017: TLB %s entry %d refill exception didn’t occur
```

---

tlb7 (tlb_g) - Check global bit in TLB entry

Sets up all the TLB slots to allow global access, then attempts access on all slots with a variety of different pid settings. This test passes only if no invalid access exceptions occur.

Possible error:

```
010801d: Unexpected exception occurred during global access
```

---

tlb8 (tlb_c) - Check C bits in TLB entry

Attempts to access TLB-mapped memory in both cached and uncached modes. Tests all slots by writing and reading back a pattern, first in cached mode, then in uncached mode. This test checks basic functionality, and does not attempt to detect cached/uncached interactions.

Possible errors:

```
010800f: Exception during cached write to 0x%x
0108010: Cached write to 0x%x failed
0108011: TLB %s entry %d cached mode exception
0108012: TLB %s entry %d cached R/W error : Wrote 0x%x Read 0x%x
0108013: TLB %s entry %d uncached mode exception
0108014: TLB %s entry %d uncached R/W error : Wrote 0x%x Read 0x%x
```

---

tlb9 (tlb_mapuc) - Check cached/uncached TLB access

Checks that both cached and uncached mapped access work without interfering with each other. This test aims at detecting the R4000 mapped uncached writeback bug. The method used is to set up 2 TLB entries for the same page of physical memory, one using cached access and the other using uncached. A write is done via each of the TLB entries, followed by a read. If the R4000 cache is working properly, the test will be able to read back the correct (different) pattern for each access mode, since the code avoids flushing the cache to main memory. If the bug is present, the same value will be read back via both cached and uncached access. The writes are done in both cached - uncached and uncached - cached orders.

Possible errors:

```
0108004: TLB %s entry %d cached/uncached W exception
0108005: TLB %s entry %d cached/uncached W error : Wrote 0x%x Read
                0x%x
0108006: TLB %s entry %d uncached/cached W exception
0108007: TLB %s entry %d uncached/cached W error : Wrote 0x%x Read
                0x%x
0108008: TLB %s entry %d uncached/cached RW exception
0108009: TLB %s entry %d uncached/cached RW error : Wrote 0x%x Read
                0x%x
010800a: TLB %s entry %d uncached/cached RWR error : Wrote 0x%x Read
                0x%x
```

----

fpu1 (fpregs) - fpu register test

This test simply writes and reads the FPU registers,
reporting any readback errors.

Possible errors:

```
010901e: FP register %d data error : Expected 0x%x Got 0x%x
010901f: FP register %d inverted data error : Expected 0x%x Got 0x%x
```

----

fpu2 (fpmem) - fpu load/store mem test

This test loads FPU from memory and stores memory from FPU.

Possible errors:

```
010901c: Load/store FP reg %d data error : Expected 0x%x Got 0x%x
010901d: Load/store FP reg %d inverted data error : Expected 0x%x, Got
                0x%x
```

---

fpu3 (faddsubs) - fpu add/subtract(single precision)

Tests addition and subtraction using simple single precision arithmetic.

Possible errors:

```
0109004: FP single add/sub result error : Expected 0x%x Got 0x%x
0109005: FP single add/sub status error : Expected 0 Got 0x%x
0109006: Fixed to single conversion failed : Before 0x%x After 0x%x
```

---

fpu4 (faddsubd) - fpu add/subtract(double precision)

Tests addition and subtraction using simple double precision arithmetic.

Possible errors:

```
0109001: FP double add/sub result error : Expected 0x%x Got 0x%x
0109002: FP double add/sub status error : Expected 0 Got 0x%x
0109003: Fixed to double conversion failed : Before 0x%x After 0x%x
```

---

fpu5 (fmuldivs) - fpu multiply/divide (single precision)

Tests multiplication and division using simple single precision arithmetic.

Possible errors:

```
0109011: FP single divide result error : Expected 0x%x Got 0x%x
0109012: FP single multiply result error : Expected 0x%x Got 0x%x
```

---

fpu6 (fmuldivd) - fpu multiply/divide (double precision)

Tests multiplication and division using simple double precision arithmetic.

Possible errors:

```
010900f: FP double divide result error : Expected 0x%x Got 0x%x
0109010: FP double multiply result error : Expected 0x%x Got 0x%x
```

---

fpu7 (fmulsubs) − fpu multiply/subtract (single precision)

Tests multiplication and subtraction using simple single
precision arithmetic.

Possible errors:

```
0109016: FP single mul/div result error : Expected 0x%x Got 0x%x
0109017: Fixed to single conversion failed : Before 0x%x After 0x%x
0109018: FP single mul/div status error : 0x%x
```

---

fpu8 (fmulsubd) − fpu multiply/subtract (double precision)

Tests multiplication and subtraction using simple double
precision arithmetic.

Possible errors:

```
0109013: FP double mul/sub result error : Expected 0x%x Got 0x%x
0109014: Fixed to double conversion failed : Before 0x%x After 0x%x
0109015: FP double mul/div status error : 0x%x
```

---

fpu9 (finvalid) − fpu invalid test

Simple test to see if an invalid operation exception can be
generated. Divides 0.0 by itself to generate the exception.

Possible errors:

```
010900b: Invalid exception didn’t occur
010900c: Invalid exception status error : 0x%x
010900d: Invalid exception dividend error : Expected 0x%x Got 0x%x
```

---

fpu10 (fdivzero) − fpu divided by zero test

Divides a non−zero value by 0.0. Unlike the previous test,
the floating point status register is checked after the
exception to make sure the divide by zero flag is set.

Possible errors:

0109007: Divide by Zero exception status error : 0x%x
0109008: Dividend conversion failed : Before 0x%x After 0x%x
0109009: Divisor conversion failed : Before 0x%x After 0x%x

---

fpu11 (foverflow) - fpu overflow test

Generates a single precision overflow by adding 2 at-the-limit large values. After the exception, the floating point status register is checked to make sure the overflow flag was set.

Possible error:

0109019: Overflow exception status error : 0x%x

---

fpu12 (funderflow) - fpu underflow test

Generates a single precision overflow by dividing an at-the-limit small value by 2. After the exception, the floating point status register is checked to make sure the underflow flag was set.

Possible errors:

0109020: Exception other than Underflow in FCR31 : 0x%x
0109021: Failed to generate Underflow Exception

---

fpu13 (finexact) - fpu inexact test

Generates a single precision inexact conversion error by attempting to convert an integer value too large for a single precision representation into a single precision value. After the error, the floating point status register is checked to make sure the inexact conversion flag was set.

Possible error:

010900a: Inexact exception status error : 0x%x

---

fpu14 (fpcmput) - fpu computation test

Given a list of "infinite" series, this test executes them a specified number of times and compares the result gotten at run-time with an expected result. Discrepancies are reported. This is a slow test.

Possible errors:

```
010900e: FP computation unexpected exception : 0x%x
010901a: Single precision %s error : Expected 0x%x Got 0x%x
010901b: Double precision %s error : Expected 0x%x 0x%x Got 0x%x 0x%x
```

---

cache1 (Taghitst) - TAGHI Register Test

This diag tests the data integrity of the taghi register. A sliding one and a sliding zero pattern are used.

Possible errors:

```
0104001: Taghi register failed walking one test
         Expected data: 0x%08x Actual data: 0x%08x
0104002: Taghi register failed walking zero test
         Expected data: 0x%08x Actual data: 0x%08x
```

---

cache2 (Taglotst) - TAGLO Register Test

This diag tests the data integrity of the taglo register. A sliding one and a sliding zero pattern are used.

Possible errors:

```
0104003: Taglo register failed walking one test
         Expected data: 0x%08x Actual data: 0x%08x
0104004: Taglo register failed walking zero test
         Expected data: 0x%08x Actual data: 0x%08x
```

---

cache3 (pdtagwlk) − Primary data TAG RAM data line Test

This diag checks the data integrity of the primary data TAG RAM path using walking ones and walking zeros patterns.

Possible error:

```
0104005: D−cache tag ram data line error
         Failed walking one (or zero) test at 0x%08x
         Expected: 0x%08x Actual 0x%08x
```

---

cache4 (pdtagadr) − Primary data TAG RAM address line Test

This diag tests the address lines to the primary data cache TAG RAM by sliding a one and then a zero on the address lines. This test assumes that the taglo register is in good working condition.

Possible error:

```
0104006: D−cache tag ram address line error
         Failed walking one (or zero) test at 0x%08x
         Expected: 0x%08x Actual 0x%08x
```

---

cache5 (PdTagKh) − Primary data TAG Knaizuk Hartmann Test

This diag tests the data integrity of the primary data cache TAG RAM with the Knaizuk Hartmann algorithm. It treats the TAG RAM array as a ordinary memory array. The parity bit is not checked in this test.

A note about the Knaizuk Hartmann Memory Test

This algorithm is used to perform a fast but non−exhaustive memory test. It will test a memory subsystem for stuck−at faults in both the address lines as well as the data locations.

The algorithm breaks up the memory to be tested into 3 partitions. Partition 0 consists of memory locations 0, 3, 6, ...; partition 1 consists of memory locations 1,4,7,...; partition 2 consists of locations 2,5,8... The partitions are filled with either an all ones pattern or an all zeroes pattern. By varying the order in which the partitions are filled and then checked, this algorithm manages to check all combinations of possible stuck at faults.

Possible errors:

```
0104007: Partition 1 error after partition 0 set to 0xaaaaaaaa
0104008: Partition 2 error after partition 1 set to 0xaaaaaaaa
0104009: Partition 0 error after partition 1 set to 0xaaaaaaaa
010400a: Partition 1 error after partition 1 set to 0xaaaaaaaa
010400b: Partition 0 error after partition 0 set to 0x55555555
010400c: Partition 2 error after partition 2 set to 0xaaaaaaaa
```

For each of the above errors, the following additional information is also provided:

```
         Tag ram address: 0x%08x
         Expected: 0x%08x Actual: 0x%08x Xor: 0x%08x
```

---

cache6 (pitagwlk) - Primary Instruction TAG RAM data line Test

This diag checks the data integrity of the primary instruction cache TAG RAM path using a walking ones and zeros pattern.

Possible error:

```
010400d: I-cache tag ram data line error
         Failed sliding one (or zero) test at 0x%08x
         Expected: 0x%08x, Actual: 0x%08x
```

---

cache7 (pitagadr) - Primary Instruction TAG RAM address line Test

This diag tests the address lines to the primary instruction cache TAG RAM by sliding a one and then a zero one the address lines. This test assumes that the taglo register is in good working condition.

Possible error:

```
010400e: I-cache tag ram address line error
         Failed sliding one (or zero) test at 0x%08x
         Expected: 0x%08x Actual 0x%08x
```

---

cache8 (PiTagKh) - Primary Instruction TAG RAM Knaizuk Hartmann Test

This diag tests the data integrity of the primary instruction cache TAG RAM with the Knaizuk Hartmann algorithm. It treats the TAG RAM array as a ordinary memory array. The parity bit is not checked in this test.

Possible errors:

```
010400f: Partition 1 error after partition 0 set to 0xaaaaaaaa
0104010: Partition 2 error after partition 1 set to 0xaaaaaaaa
0104011: Partition 0 error after partition 1 set to 0xaaaaaaaa
0104012: Partition 1 error after partition 1 set to 0xaaaaaaaa
0104013: Partition 0 error after partition 0 set to 0x55555555
0104014: Partition 2 error after partition 2 set to 0xaaaaaaaa
```

For each of the above errors, the following additional information is provided:

```
         Tag ram index address: 0x%08x
         Expected: 0x%08x Actual: 0x%08x Xor: 0x%08x
```

---

cache9 (sd_tagwlk) - Secondary TAG data path Test

Checks the data integrity of the Secondary data TAG RAM path using a walking ones/zeros pattern.

Possible error:

```
0105015: Secondary Data TAG RAM Path Error
         on sliding one (or zero) pattern
         TAG RAM Location 0x%x
         Expected 0x%x  Actual= 0x%x  XOR= 0x%x
```

---

cache10 (sd_tagaddr) - Secondary TAG address Test

Checks the address integrity to the Primary Data TAG RAM by using a walking address.

Possible error:

```
0105016: Secondary Data TAG Address Error
         TAG RAM Location 0x%x
         Expected 0x%x  Actual= 0x%x  XOR= 0x%x
```

---

cache11 (sd_tagkh) - Secondary TAG RAM Knaizuk Hartmann Test

This diag tests the data integrity of the secondary data cache TAG RAM with the Knaizuk Hartmann algorithm. It treats the TAG RAM array as a ordinary memory array. The parity bit is not checked in this test.

Possible error:

```
0105017: Secondary Data TAG ram data Error
         Address %x, error code %d
         expected %x, actual %x, XOR %x
```

---

cache12 (d_tagparity) - Primary Data TAG RAM parity Test

This diag tests the functionality of the parity bit in the primary data cache tag. For each tag, a stream of one's and zero's are shifted into the tag to check if the parity bit change state accordingly.

Possible error:

```
0104018: D-cache tag ram parity bit error
         Tag ram address: 0x%08x expected content: 0x%08x
         Taglo: 0x%08x expected parity: 0x%x actual parity: 0x%x
```

---

cache13 (d_tagcmp) - Primary Data TAG comparator Test

This diag tests the comparator at the D-cache tag for hit and miss detection. For each tag, set the ptag field with the values which will cause a cache hit for the Kseg0 address of 0x80002000 to 0x9fffffff. The values used are a walking one or a walking zero pattern. This will ensure only one bit location is tested at the comparator. The cache op Hit Invalidate is used to check for cache hit and miss situations.

Possible errors:

```
0104019: D-cache tag comparator did not detect a miss
0104020: D-cache tag comparator did not detect a hit
```

For each of the above errors, the following additional information are provided:

```
         Tag ram address: 0x%08x
         PTag field of tag: 0x%06x comparing with PFN: 0x%06x
```

---

cache14 (d_tagfunct) - Primary Data TAG functionality Test

This diag tests the functionality of the data cache tag. Kseg0 addresses are used to load the cache from memory. The ptag and the cache state field are checked to see if they are holding expected values. Virtual addresses 0x80000000, 0x80002000, 0x80004000, 0x80008000, ... 0x90000000 are used as the baseaddress of an 8k page which is mapped to the cache. The ptag and state of each cache line are checked against the expected value.

Possible errors:

```
0104021: D-cache tag functional error in PTAG field
         PTag field does not contain correct tag bits
         Cache line address: 0x%08x
         Expected PTag: 0x%06x
         Actual PTag: 0x%06x
         TAGLO Register %x
         Re-read DTAG %x
0104022: D-cache tag functional cache state error
         Cache line address: 0x%08x
         Expected cache state: 0x%08x
         Actual cache state: 0x%08x
         TAGLO Register %x
         Re-read DTAG %x
```

---

cache15 (d_slide_data) - Primary Data RAM data line Test

This diag tests the data lines to the primary data cache. A sliding one and a sliding zero data pattern is written into the first location of the D-cache to check if each data line can be toggled individually.

Possible errors:

```
0107023: D-cache data ram data lines failed walking one test
         Addr: 0x%08x Expected: 0x%08x Actual: 0x%08x Xor: 0x%08x
0107024: D-cache data ram data lines failed walking zero test
         Addr: 0x%08x Expected: 0x%08x Actual: 0x%08x Xor: 0x%08x
```

---

cache16 (d_slide_addr) - Primary Data RAM address line Test

This diag tests the address lines to the primary data cache. Each address line to the data cache is toggled once individually by sliding a one and then a zero across the address lines.

Possible errors:

```
0107025: D-cache data ram address lines failed walking one tes
         Addr: 0x%08x Expected: 0x%08x Actual: 0x%08x Xor: 0x%08x
0107026: D-cache data ram address lines failed walking zero test
         Addr: 0x%08x Expected: 0x%08x Actual: 0x%08x Xor: 0x%08x
```

---

cache17 (d_kh) - Primary Data RAM Knaizuk Hartmann Test

This diag tests the data integrity of the D-cache with the Knaizuk Hartmann algorithm. Data pattern 0x55555555 and 0xaaaaaaaa are used.

Possible errors:

```
0107027: Partition 1 error after partition 0 set to 0xaaaaaaaa
0107028: Partition 2 error after partition 1 set to 0xaaaaaaaa
0107029: Partition 0 error after partition 1 set to 0xaaaaaaaa
010702a: Partition 1 error after partition 1 set to 0xaaaaaaaa
010702b: Partition 0 error after partition 0 set to 0x55555555
010702c: Partition 2 error after partition 2 set to 0xaaaaaaaa
```

For each of the above errors, the following additional information is provided:

```
         Cache address: 0x%08x
         Expected: 0x%08x Actual: 0x%08x Xor: 0x%08x
```

---

cache18 (dsd_wlk) - Primary/Secondary Data path Test

Test the data path from memory through the secondary cache
and to the Primary Data Cache.

Possible errors:

```
010702d: Data Path Error from Memory->Secondary->Primary Data
         Address %x, expected %x, actual %x, Xor %x
010702e: Data Path Error from Primary ->Secondary->Memory Data
         Address %x, Expected %x, Actual %x, Xor %x
```

---

cache19 (sd_aina) - Secondary Data RAM (address in address)
Test

Performs an address in address test on the secondary data
cache.

Possible errors:

```
010502f: Secondary Memory Error on pattern 1
         Address %08x
         expected %08x, actual %08x, XOR %08x
0105030: Secondary Memory Error on pattern 2
         Address %08x
         expected %08x, actual %08x, XOR %08x
```

---

cache20 (d_function) - Primary Data functionality Test

This diag tests the functionality of the entire data cache.
It checks the block fill, write back on a dirty line
replacement, and no write back on a clean line replacement
function of the data cache lines.

Possible errors:

```
0104031: D-cache block fill error 1
         Cache contains incorrect data
         Cache Address: 0x%08x
         Expected: 0x%08x Actual: 0x%08x Xor: 0x%08x
0104032: D-cache block fill error 2
         Cache contains incorrect data
         Cache Address: 0x%08x
         Expected: 0x%08x Actual: 0x%08x Xor: 0x%08x
0104033: D-cache block write back error 1
         Memory contains incorrect data
         Cache Address: 0x%08x
         Expected: 0x%08x Actual: 0x%08x Xor: 0x%08x
0104034: D-cache block fill error 3
         Cache contains incorrect data
         Cache Address: 0x%08x
         Expected: 0x%08x Actual: 0x%08x Xor: 0x%08x
0104035: D-cache block write back error 2
         Memory content is altered
         Write back happened on a clean line
         Cache Address: 0x%08x
         Expected: 0x%08x Actual: 0x%08x Xor: 0x%08x
```

----

cache21 (d_parity) - Primary Data parity generation Test

This diag tests the parity bit generation of the D-cache data ram.

Possible error:

```
0104036: D-cache parity generation error
         error %x
         Cache byte address: 0x%08x data:0x%02x
         Parity bit position: 0x%02x
         Expected parity: 0x%02x Actual parity:0x%02x
```

----

cache22 (i_tagparity) - Primary Instruction TAG RAM parity bit Test

This diag tests the functionality of the parity bit in the primary I-cache tag. For each tag, the parity bit is tested to respond to each bit change in the tag.

Possible error:

```
0104037: I-cache tag ram parity bit error
         Tag ram address: 0x%08x expected content: 0x%08x
         Taglo: 0x%08x expected parity: 0x%x actual parity: 0x%x
```

---

cache23 (i_tagcmp) - Primary Instruction TAG RAM comparator
Test

This diag tests the comparator at the I-cache tag for hit
and miss detection.

Possible errors:

```
0104038: I-cache tag comparator did not detect a miss (walking l)
0104039: I-cache tag comparator did not detect a hit (walking 1)
010403a: I-cache tag comparator did not detect a miss (walking zero)
010403d: I-cache tag comparator did not detect a hit (walking zero)
```

For each of the above errors, the following additional
information is provided:

```
         Tag ram address: 0x%08x
         PTag field of tag: 0x%06x comparing with PFN: 0x%06x
```

---

cache24 (i_tagfunct) - Primary Instruction TAG functionality
Test

This diag tests the functionality of the instruction cache
tag. Kseg0 addresses are used to load the cache from memory.
This will test if the cache is functional on the cachable
memory space. After each 8k segment of memory is loaded into
the cache. The ptag and the cache state field are checked to
see if they are holding expected values. Virtual addresses
0x80000000, 0x80002000, 0x80004000, 0x80008000, ...,
0x90000000 are used as the base address of each 8k page
which is mapped to the cache. The ptag and cache state of
each cache line are checked against the expected value.

Possible errors:

```
010403b: I-cache tag functional error in PTAG field
         PTag field does not contain correct tag bits
         Cache line address: 0x%08x
         Expected PTag: 0x%06x
         Actual PTag: 0x%06x
010403c: I-cache tag functional cache state error
         Cache state not correct
         Cache line address: 0x%08x
         Expected cache state: 0x%08x
         Actual cache state: 0x%08x
```

----

cache25 (i_slide_data) - Primary Instruction data RAM data
line Test

This diag checks the data lines to the I-cache data ram by
sliding a one and zero bit across the bus.

Possible errors:

```
010403f: I-cache data ram data lines failed walking one test
         Addr: 0x%08x Expected: 0x%08x Actual: 0x%08x Xor: 0x%08x
         PITAG  %x
         PDTAG  %x
         STAG  %x
0104040: I-cache data ram data lines failed walking zero test
         Addr: 0x%08x Expected: 0x%08x Actual: 0x%08x Xor: 0x%08x
         PTAG  %x
         STAG  %x
```

----

cache26 (i_aina) - Primary Instruction data RAM address in
address Test

Performs an address in address test on the primary
instruction cache.

Possible error:

```
0107041: I-cache address in address error
         addr %x, exp %x, act %x, XOR %x
```

----

cache27 (i_function) - Primary Instruction functionality
Test

This diag tests the functionality of the entire instruction
cache. It checks the block fill and hit write back of the
instruction cache lines.

Possible error:

```
0107042: I-cache block write back error
         Memory contains incorrect data
         Cache address: 0x%08x
         Expected: 0x%08x Actual: 0x%08x Xor: 0x%08x
         Icache TAG = %x
         Scache TAG = %x
```

----

cache28 (i_parity) - Primary Instruction parity generation
Test

This diag tests the parity bit generation of the I-cache
data ram.

Possible error:

```
0104043: I-cache parity generation error
         error %x
         Cache byte address: 0x%08x data:0x%02x
         Parity bit position: 0x%02
         Expected parity: 0x%02x Actual parity:0x%02x
```

----

cache29 (i_hitinv) - Primary Instruction Hit Invalidate Test

This diag tests the Hit Invalidate cache op on the
Instruction cache.

Possible errors:

```
0104044: I-cache state error during initialization
         Cache state did not change to valid when filled from memory
         Cache line address: 0x%08x
         Expected cache state: 0x%08x Actual cache state: 0x%08x
0104045: I-cache state error
         Hit Invalidate changed the line to invalid on a miss
         Cache line address: 0x%08x
         Miss address: 0x%08x
         Expected cache state: 0x%08x Actual cache state: 0x%08x
0104046: I-cache state error on a Hit Invalidate Cache OP
         Hit Invalidate did not invalidate the line on a hit
         Cache line address: 0x%08x
         Expected cache state: 0x%08x Actual cache state: 0x%08x
```

---

cache30 (i_hitwb) - Primary Instruction Hit Writeback Test

This diag tests the Hit Writeback cache op on the instruction cache.

Possible errors:

```
0104047: I-cache state error during initialization
         Cache state did not change to valid when filled from memory
         Cache line address: 0x%08x
         Expected cache state: 0x%08x Actual cache state: 0x%08x
0104048: I-cache state error Hit writeback happened on a cache miss
         Cache line address: 0x%08x
         Miss address: 0x%08x
0104049: I-cache Hit writeback did not happen on a cache hit
         Cache line address: 0x%08x
         expected %x, actual %x, XOR %x
```

---

cache31 (ECC_reg_tst) - ECC register Test

This diag tests the data integrity of the ECC register. A sliding one and sliding zero pattern is used in this test.

Possible errors:

```
010404a: ECC register failed walking one test
         Expected data: 0x%08x Actual data: 0x%08x
010404b: ECC register failed walking zero test
         Expected data: 0x%08x Actual data: 0x%08x
```

---

cache32 (dd_hitinv) - Primary Data Hit Invalidate Test

This diag tests the Hit Invalidate cache op on the data
cache.

Possible errors:

```
010404c: D-cache state error during initialization
         Cache state did not change to valid when filled from memory
         Cache line address: 0x%08x
         Expected cache state: 0x%08x Actual cache state: 0x%08x
010404d: D-cache state error
         Hit Invalidate changed the line to invalid on a miss
         Cache line address: 0x%08x
         Miss address: 0x%08x
         Expected cache state: 0x%08x Actual cache state: 0x%08x
010404e: D-cache state error on a  Hit Invalidate Cache OP
         Hit Invalidate did not invalidate the line on a hit
         Cache line address: 0x%08x
         Expected cache state: 0x%08x Actual cache state: 0x%08x
```

----

cache33 (d_hitwb) - Primary Data Hit Writeback Test

This diag tests the Hit Writeback cache op on the data
cache.

Possible errors:

```
010404f: D-cache state error during initialization
         Cache state did not change to valid when filled from memory
         Cache line address: 0x%08x
         Expected cache state: 0x%08x Actual cache state: 0x%08x
         TAGLO Reg %x
         Re-Read dtag %x
         Re-Read stag %x
0104050: D-cache state error Hit writeback happened on a clean exclusive line
         Cache line address: 0x%08x
         PTAG %x
         Scache TAG %x
0104051: D-cache Hit writeback happened on a cache miss
         Cache line address: 0x%08x
         Miss address: 0x%08x
         PTAG %x
         Scache TAG %x
0104052: D-cache Hit writeback did not happen on a cache hit
         Cache line address: 0x%08x
         PTAG %x
         Scache TAG %x
0104053: D-cache Hit Writeback clears the write back bi
         Cache line address: 0x%08x
```

---

cache34 (d_dirtywbw) - Primary Data dirty writeback word
Test

This test verifies the block (4 words) write mode in data cache. It writes to K0 (0x80020000) cached space, causing the cache dirty. Then it replace the cache line by reading 0x80022000, different cache line with same offset. This causes the data in 0x80020000 wrtie back to memory which now has the same data as in 0x80020000. Multiple cache lines are tested back to back.

Possible errors:

```
0104054: Unexpected Cache write through to memory
         addr %x, expected %x, actual %x, XOR %x
         Secondary TAG %x
0104055: Cache writeback did not occur on a word store to a dirty line
         addr %x
         expected %x, actual %x, XOR %x
         Secondary TAG %x
```

---

cache35 (d_refill) - Primary Data refill from Secondary
Cache Test

This test verifies the block write/read mode in data cache.
It writes to K0 (0x80020000) cached space, causing the cache
dirty. Then it replace the cache line by reading
0x80022000, different cache line with same offset. This
causes the data in primary data cache to be written back to
the secondary. The address 0x80020000 is reread and
compared. Should be a cache hit in the secondary.

Possible errors:

```
0104056: Unexpected Cache write through to memory
         addr = %x
         expected = %x, actual = %x, XOR %x
         Secondary TAG %x
0104057: Secondary Cache miss, expected a cache hit
         addr = %x
         expected = %x, actual = %x, XOR = %x
         Data in memory = 0xdeadbeef
         Secondary TAG %x
```

---

cache36 (sd_dirtywbw) - Secondary Dirty Writeback (word)
Test

This test verifies the block (4 words) write mode in data
cache. It writes to K0 (0x80020000) cached space, causing
the cache dirty. Then it replace the cache line by reading
0x80022000, different cache line with same offset. This
causes the data in 0x80020000 wrtie back to secondary which
now has the same data as in 0x80020000. A write to address
0x80060000 will replace the

secondary lines, thus forcing a writeback from the Secondary
Cache. Note, there is another flavor of this test
d_dirtywbw.c which forces the writeback from the primary
when the secondary line is replaced.

Possible errors:

```
0105058: Unexpected Cache write through to memory
         addr = %x
         expected = %x, actual = %x, XOR %x
         Secondary TAG %x
0105059: Data read replaced a dirty line in Secondary
         Dirty line not written back to memory
         addr = %x
         expected = %x, actual = %x, XOR %x
         Secondary TAG %x
```

----

cache37 (sd_dirtywbh) - Secondary Dirty Writeback (halfword)
Test

This test verifies the block (4 words) write mode in data
cache. It writes to K0 (0x80020000) cached space, causing
the cache dirty. Then it replace the cache line by reading
0x80022000, different cache line with same offset. This
causes the data in 0x80020000 wrtie back to memory which now
has the same data as in 0x80020000. Multiple cache lines
are tested back to back. Half word transactions are tested.

Possible errors:

```
010505a: Unexpected Cache write through to memory on store halfword
         addr = %x
         expected = %4x, actual = %4x, XOR %4x
         Secondary TAG %
010505b: Halfword read replaced a dirty line in Secondary, dirty line not writte
                back to memory
         addr = %x
         expected = %4x, actual = %4x, XOR %4x
         Secondary TAG %x
```

---

cache38 (sd_dirtywbb) - Secondary Dirty Writeback (byte)
Test

This test verifies the block (4 words) write mode in data cache. It writes to K0 (0x80020000) cached space, causing the cache dirty. Then it replace the cache line by reading 0x80022000, different cache line with same offset. This causes the data in 0x80020000 wrtie back to memory which now has the same data as in 0x80020000. Multiple cache lines are tested back to back. Byte transactions are tested.

Possible errors:

```
010505c: Unexpected Cache write through to memory on store byte
         addr = %x
         expected = %2x, actual = %2x, XOR %2x
         Secondary TAG %x
010505d: Byte read replaced a dirty line in Secondary, dirty line not written
                back to memory
         Dirty line not written back to memory
         addr = %x
         expected = %2x, actual = %2x, XOR %2x
         Secondary TAG %x
```

---

cache39 (sd_tagecc) - Secondary TAG ECC Test

Checks the data integrity of the Secondary data tag ram path using a walking ones/zeros pattern.

Possible errors:

```
010505e: Secondary Data TAG RAM ECC Path error (walking one as data)
         TAG RAM Location 0x%x
         Expected 0x%x  Actual= 0x%x  XOR= 0x%x
010505f: Secondary Data TAG RAM ECC Path error (walking zero as data)
         TAG RAM Location 0x%x
         Expected 0x%x  Actual= 0x%x  XOR= 0x%x
```

---

cache40 (sdd_hitinv) − Secondary Hit Invalidate Test

This test verifies the Hit Invalidate Cache operation.

Possible errors:

```
0105060: S−cache state error during initialization
         addr = %x
         expected = %x, actual = %x, XOR %x
         Secondary TAG %x
0105061: S−Cache error during Primary Cache dirty line writeback to Scache
0105062: S−Cache state error on a  Hit Invalidate Cache OP
0105063: Data written back to memory after a Hit Invalidate on the Secondary
         addr = %x
         expected = %x, actual = %x, XOR %x
         Secondary TAG %x
0105064: S−Cache state error on a  Hit Invalidate Cache OP
0105065: Primary Cache TAG not invalid after a Hit Invalidate on the Scache
         addr %x
         Secondary TAG %x
         Primary TAG %x
0105066: Data written back to memory after a Hit Invalidate on the Secondary
         addr = %x
         expected = %x, actual = %x, XOR %x
         Secondary TAG %x
         Primary TAG %x
```

For errors 0105061, 0105062 and 0105064, the following additional information is provided:

```
         Error in Secondary Cache TAG State field
         OR Error in Secondary Cache TAG physical tag field
         OR Error in Secondary Cache TAG Virtual Address field
         Address 0x%08x10econdary TAG Data 0x%08x
         Expected Cache State:  0x%x = [STATE]
```

STATE is one of the decoded cache states: Invalid, Clean Exclusive, Dirty Exclusive, Shared, and Dirty Shared.

---

cache41 (sd_hitwb) - Secondary Hit Writeback Test

This test verifies the Hit Writeback Cache operation. It verifies that the data can be written back from the Secondary or in the case where the primary data is more current that the data is written from the Primary to memory. Also checked is the fact that the cache lines are not invalidated as with the Hit Writeback Invalidate Cache Op. Instead in checks that the lines is set to the clean exclusive state.

Possible errors:

```
0105067: Initialization error, unexpected Cache write through to memory
         addr = %x
         expected = %x, actual = %x, XOR %x
         Secondary TAG %x
0105068: SCache error during Primary Cache dirty line writeback to Scache
0105069: Data not written back from Scache to Memory on Hit Writeback
                Cache OP
         addr = %x
         expected = %x, actual = %x, XOR %x
         Secondary TAG %x
010506a: Initialization error, unexpected Cache write through to memory
         addr = %x
         expected = %x, actual = %x, XOR %x
         Secondary TAG %x
010506b: SCache state error during Hit Writeback on S-Cache dirty line
010506c: Error in Primary Cache TAG after a Hit Writeback cache Op on the
                SCache
         addr %x
         Expected cache state: Dirty Exclusive
         Primary Data TAG %x
010506d: Data not written back from D-Cache to Memory on a Hit Writeback
                on the S-Cache
         addr = %x
         expected = %x, actual = %x, XOR %x
         Secondary TAG %x
         Primary Data TAG %x
```

For errors 0105068 and 010506b, the following additional information are provided:

```
Error in Secondary Cache TAG State field
OR Error in Secondary Cache TAG physical tag field
OR Error in Secondary Cache TAG Virtual Address field
Address 0x%08x10econdary TAG Data 0x%08x
Expected Cache State:  0x%x = [STATE]
```

STATE is one of the decoded cache states: Invalid, Clean Exclusive, Dirty Exclusive, Shared, and Dirty Shared.

---

cache42 (sd_hitwbinv) - Secondary Hit Writeback Invalidate Test

This test verifies the Hit Writeback Invalidate Cache operation. It verifies that the data can be written back from the Secondary or in the case where the primary data is more current that the data is written from the Primary to memory. Also checked is the fact that the cache lines are invalidated.

Possible errors:

```
010506e: Initialization error, unexpected Cache write through to memory
         addr = %x
         expected = %x, actual = %x, XOR %x
         Secondary TAG %x
010506f: S-Cache TAG error after Hit Writeback Invalidate cacheo
0105070: Data not written back from Scache to Memory after Hit Writeback
                Invalidate Cacheop
         addr = %x
         expected = %x, actual = %x, XOR %x
         Secondary TAG %x
0105071: Initialization error, unexpected Cache write through to memory
         addr = %x
         expected = %x, actual = %x, XOR %x
         Secondary TAG %x
0105072: S-Cache TAG error after Hit Writeback Invalidate cacheop, test case 2
0105073: Error in Primary Cache TAG after a Hit Writeback Invalidate cacheop
               on the SCache
         addr %x
         Expected cache state: Invalid
         Primary Data TAG %x
0105074: Data not written back from D-Cache to Memory on a Hit Writeback
                Invalidate on the S-Cache
         addr = %x
         expected = %x, actual = %x, XOR %x
         Secondary TAG %x
         Primary Data TAG %x
```

For errors 010506f and 0105072, the following additional information are provided:

```
         Error in Secondary Cache TAG State field
         OR Error in Secondary Cache TAG physical tag field
         OR Error in Secondary Cache TAG Virtual Address field
         Address 0x%08x10econdary TAG Data 0x%08x
         Expected Cache State:  0x%x = [STATE]
```

STATE is one of the decoded cache states: Invalid, Clean Exclusive, Dirty Exclusive, Shared, and Dirty Shared.

---

cache43 (cluster) - Secondary Cluster Test

Possible errors:

```
0105075: SCache data incorrectly written to memory during a dirty writeback
               operation
         1st mem block
         Mem Address 0x%08x
         Expected 0x%08x, Actual 0x%08x, XOR 0x%08x
0105076: SCache data incorrectly written to memory during a dirty writeback
               operation
         2nd mem block
         Mem Address 0x%08x
         Expected 0x%08x, Actual 0x%08x, XOR 0x%08x
```

---

cache44 (clusterwb) - Secondary Cluster Writeback Test

Possible errors:

```
0105077: SCache data incorrectly written to memory during a dirty writeback
               operation on 1st block
         Mem Address 0x%08x
         Expected 0x%08x, Actual 0x%08x, XOR 0x%08x
0105078: SCache data incorrectly written to memory during a dirty writeback
               operation on 2nd block
         Mem Address 0x%08x
         Expected 0x%08x, Actual 0x%08x, XOR 0x%08x
0105079: SCache data incorrectly written to memory during a dirty writeback
               operation on 3rd block
         Mem Address 0x%08x
         Expected 0x%08x, Actual 0x%08x, XOR 0x%08x
```

---

cache45 (hammer_pdcache) - stress primary D-cache--runs
icached

Possible error:

```
010407b: Primary cache stress error at addr : 0x%x Expected 0x%x Got 0x%x
```

---

cache46 (hammer_scache) - stress secondary cache--runs icached

Possible error:

```
010507c: Secondary cache stress error at addr : 0x%x Expected 0x%x Got 0x%x
```

---

cache47 (cache_stress) - cache stress test

Write/read to one word in every page through 0x80000000 space.

Possible error:

```
010507a: Secondary cache stress error at addr : 0x%x Expected 0x%x Got 0x%x
```

---

cache48 (cache_states) - complete cache-state transitions test

The abbreviation of the following cache states are to be used in the description of each cache state transition test:

CE	clean exclusive
DE	dirty exclusive
I	invalid

cstate0 (RHH_CE_CE)

Read hit primary (CE) and 2nd (CE). Check that the value is correct (the physmem addr) and that both tags are still CE.

cstate1 (RHH_DE_DE)

Read hit primary (DE) and 2nd (DE). Check value and that both are still DE.

cstate2 (WHH_CE_CE)

Write hit primary (CE) and 2nd (CE). Check that 2nd and
memory still have old value and that both cache lines are
now DE.

cstate3 (WHH_DE_DE)

Write hit primary (DE) and 2nd (DE). Check that 2nd and
memory still have old value and that both lines are still
DE.

cstate4 (RMH_I_CE)

Read miss primary (I) and hit 2nd (CE). Check that 2nd and
memory still have old value and that both lines are CE.

cstate5 (RMH_I_DE)

Read miss primary (I) and hit 2nd (DE). Check that 2nd and
memory still have old value and that both lines are DE.

cstate6 (RMH_CE_CE)

Read miss primary (CE) and hit 2nd (CE). Check that 2nd and
memory still have old value and that both lines are still
CE.

cstate7 (RMH_DE_DE)

Read miss primary (DE) and hit 2nd (DE). Check that 2nd and
memory still have old value and that both lines are still
CE.

cstate8 (WMH_I_CE)

Write miss primary (I) and hit 2nd (CE). Check that 2nd and
memory still have old value and that both lines are DE.

cstate9 (WMH_I_DE)

Write miss primary (I) and hit 2nd (DE). Check that 2nd and
memory still have old value and that both lines are DE.

cstate10 (WMH_CE_CE)

Write miss primary (CE) and hit 2nd (CE).

cstate11 (WMH_DE_DE)

Write miss primary (DE) and hit 2nd (DE).

cstate12 (RMM_I_I)

Read miss primary (I) and 2nd (I). Check that value is
correct, that 2nd and memory still have old value and that
both lines are CE.

cstate13 (RMM_I_CE)

Read miss primary (I) and miss 2nd (CE). Check that value is
correct, that 2nd and memory still have old value and that
both lines are CE.

cstate14 (RMM_I_DE)

Read miss primary (I) and miss 2nd (DE). Check that 2ndary line matches memory, that both tags are CE, that the addr tags on both lines are correct, and that the dirty altaddr secondary line was flushed to memory.

cstate15 (RMM_CE_CE)

Read miss primary (CE) and miss 2nd (CE). Fill cache lines with a word from physaddr+2ndcachesize; do a read, then check that the tags for both lines are CE and have the correct phys addrs, and that the alternate memory word hasn’t changed ###.

cstate16 (RMM_DE_DE)

Read miss primary (DE) and miss 2nd (DE). Fill cache lines with a word from physaddr+2ndcachesize; do a read, then check that the tags for both lines are now CE and have the correct phys addrs, and that the alternate memory word was written when the altaddr line was flushed.

cstate17 (WMM_I_I)

Write miss primary (I) and 2nd (I). Check that 2ndary line matches memory, that both tags are DE, and that the addr tags on both lines are correct.

cstate18 (WMM_I_CE)

Write miss primary (I) and miss 2nd (CE). Check that 2ndary line matches memory, that both tags are DE, and that the addr tags on both lines are correct.

cstate19 (WMM_I_DE)

Write miss primary (I) and miss 2nd (DE). Check that 2ndary line matches memory, that both tags are DE, that the addr tags on both lines are correct, and that the dirty altaddr secondary line was flushed to memory.

cstate20 (WMM_CE_CE)

Write miss primary (CE) and miss 2nd (CE). Fill cache lines with a word from physaddr+2ndcachesize; do a store, then check that the tags for both lines are DE and have the correct phys addrs, and that the alternate memory word hasn’t changed.

cstate21 (WMM_DE_DE)

Write miss primary (DE) and miss 2nd (DE). Check that 2ndary line matches memory, that both tags are DE, that the addr tags on both lines are correct, and that the dirty altaddr primary and secondary lines were flushed to memory.

Possible errors:

```
010707d: RHH_CE_CE : physaddr 0x%x contents incorrect (0x%x)
010707e: RHH_DE_DE : physaddr 0x%x contents incorrect (0x%x)
010707f: RMH_I_CE : physaddr 0x%x contents incorrect (0x%x)
0107080: RMH_I_DE : physaddr 0x%x contents incorrect (0x%x)
0107081: RMH_CE_CE : physaddr 0x%x contents incorrect (0x%x)
0107082: RMH_DE_DE : physaddr 0x%x contents incorrect (0x%x)
0107083: RMM_I_I : physaddr 0x%x contents incorrect (0x%x)
0107084: RMM_I_CE : physaddr 0x%x contents incorrect (0x%x)
0107085: RMM_I_DE : physaddr 0x%x contents incorrect (0x%x)
0107086: PRIMARYD cache state error at addr 0x%x : Expected 0x%x Got 0x%x
     OR     PRIMARYI cache state error at addr 0x%x : Expected 0x%x Got 0x%x
     OR     SECONDARY cache state error at addr 0x%x : Expected 0x%x Got 0x%x
0107087: PRIMARYD addr error at slot 0x%x : Expected 0x%x Got 0x%x
     OR     PRIMARYI addr error at slot 0x%x : Expected 0x%x Got 0x%x
     OR     SECONDARY addr error at slot 0x%x : Expected 0x%x Got 0x%x
0107088: Mem value error at addr 0x%x : Expected 0x%x Got 0x%
0107089: Writeback missed 2ndary level cache at addr 0x%x
010708a: 2ndary cache value error at addr 0x%x : Expected 0x%x Got 0x%x
```

2.3 MC3 ide Guide

To run the MC3 ide diagnostics:

- o Boot ide

- o The default report level is 2. Set the report level by typing the following:

  report=#

  where # is any number from 1 to 5.

level 5	Debugging messages displayed. Don’t need this much detail.
level 4	Prints out memory locations as they are written. Selecting this level slows down testing time.
level 3	Prints out 1-line functional descriptions within tests. This is probably the most useful level for general use.
level 2	Print out only errors, titles, and pass/fail.
level 1	Print out only titles and pass/fail.

  Level n prints out all messages for level n and below.

- o Set the modes of operations for running the tests:

  - qmode [on | off]

    For the memory tests, quick mode tests every nth byte instead of every byte, where n varies from 96 to 7680 depending upon the test. The goal in quick mode is to test 16GB in about 10 minutes-and this is accomplished by testing every nth byte. n varies depending upon how fast or slow a test was timed to run.

  - c_on_error [on | off]

    For the memory tests, the ‘‘on’’ setting continues the test even when an error has been encountered. Setting the mode to ‘‘off’’ stops the tests after the first error.

- o Run _memall_ and _memfast_.

  These are two defined commands. Each can be run in quick mode or in normal mode. _memall_ runs all commands (_memall_: _mem1_, _mem14_, _mem2_, _mem3_, _mem4_, _mem5_, _mem6_, _mem7_, _mem8_, _mem9_, _mem10_, _mem13_, _mem16_, in this order), while _memfast_ runs just the faster tests (_memfast_: _mem3_, _mem5_, _mem8_, _mem9_, _mem10_, _mem16_, in this order).

- o There are currently 18 memory tests, _mem1_-_mem18_. They are detailed below:

---

mem1 - Read the mc3 configuration registers (real fast)

The following registers are probed:

```
        reg  test description
        ---  -----------------------------------------------------

        00 Read the BankEnable
        01 Read BoardType
        02 Read RevLevel
        03 Read AccessControl: endianness, subBlockOrder, ebus=64bitsOrNot
        04 Read MemoryErrorInterrupt
        05 Read EBUSErrorInterrupt
        06 Read BIST result
        07 Read DRSC timeout
        0a Read LeafControlEnable
        Read leaf regs 10-24, 30-33 (leaf0), 50-64, 70-73 (leaf1)
```

mem1 is very similar to mem14 which is the pod-mode dmc command.

---

mem2 - Memory sockets connection test (similar to IP17's mem1) (real fast)

The memory sockets connection test writes patterns to the first 2 KB of each configured leaf and then reads them back. By writing 2 KB, all simms are ensured of being written to regardless of the interleaving factor specified.

If the pattern read back does not match, the socket is assumed to have a connection problem.

---

mem3 - Walking address test (similar to IP17's mem2) (real fast)

This is a traditional test that checks for shorts and opens on the address lines. Address lines that are greater or equal to the most significant address lines of the memory bounds are not tested. Testing is done by byte read/writes from first_address up to last_address.

---

mem4 - Write/read data patterns (similar to IP17's mem3) (slow)

This test does word read/writes of all-1's and all-0's patterns. It shows if all addresses appear to be writable, and that all bits may be set to both 1 and 0. However, it provides no address error or adjacent-bits-shorted detection. The flow is as follows:

(w0), u(r0,w1), d(r1,w5a), u(r5a,ra5), d(ra5) -- word and byte (read as: write 0 to all locations, read 0 and write 1 to all locations in ascending order, read 1 and write 5a to all locations in descending order, read 5a and write a5 to all locations in ascending order, read a5 from all locations in descending order)

mem13 does byte read/writes in the same pattern. The tests were separated out since the byte read/writes take a long time.

---

mem5 - Address in address memory test (slow)

This is a traditional, hueristic, rule-of-thumb, "address-in-address" memory test. It also puts the complement of the address in the address, and makes passes in both ascending and descending addressing order. There are both full memory store then check passes, as well as read- after-write passes (with complementing).

---

mem6 - walking 1/0 memory test (very slow)

Another traditional test - walking 1's and walking 0's through memory. This is a whole-memory test that is very good at shaking out shorted data bits, but provides little protection for addressing errors.

---

mem7 - MarchX (slow)

Described in van de Goor's book, "Testing Semiconductor Memories" and has the following flow:

(w0), u(r0,w1), d(r1,w0), (r0)

Will detect address decoder faults, stuck-at-faults, transition faults, coupling faults, and inversion coupling faults(see van de Goor for definitions)

---

mem8 - MarchY (slow)

Described in van de Goor's book, "Testing Semiconductor Memories" and has the following flow:

(w0), u(r0,w1,r1), d(r1,w0,r0), (r0)

Will detect address decoder faults, stuck-at-faults, transition faults, coupling faults, and linked transition faults(see van de Goor for definitions)

---

mem9 - Memory with ecc test (similar to IP17's mem6) (slow)

This test writes to memory via uncached space and reads back through cached space (ECC exceptions enabled). Although it provides a simple level of ECC checking, its main function is to verify that cached and uncached memory addresses are accessing the same area of physical memory. The test values used are address-in-address and inverted address- in-address patterns, so a certain amount of address uniqueness checking is done as well.

---

mem10 - Cache write-through memory test (similar to IP17's mem9)(slow)

This is a traditional, hueristic, rule-of-thumb, "address-in-address" memory test. It also puts the complement of the address in the address, making passes in ascending order only. All of memory is stored and then checked. All reads and writes are made through K0 seg, so the the reads and writes are cached. However, since the size of main memory exceeds the cache sizes, all data will be written to main memory and then read back. This is not a particularly thorough test, and it depends upon a good cache to function correctly, but it is fast, at least compared to the other full-memory tests.

---

mem11 - User-specified patter/location write/read test(similar to IP17's mem7)

type "mem11" without any arguments to see the usage.
Usage: mem11 [-b|h|w] [-r] [-l] [-c] [-v 0xpattern] RANGE

This test allows the technician to fill a range of memory with a specified test value and read it back, done as a series of byte (-b), half-word (-h), or word (-w) writes and reads. If the -v option is not used to select the test pattern, an address-in-address pattern is used instead. (-r) will do read only and will not do any writes. (-l) will loop forever. (-c) will run in cached memory space - the default is to run in uncached space.

---

mem12 - Decode a bad address into slot, leaf, bank, simm

```
Usage: mem12 [-a 0xaddress] [-b xxxxx] [-s x]
           -b expects a hex number showing which bits are
bad.
              e.g. If bits 0 and 2 are bad, enter: -b 0x5
           -s 1, 2, or 4 for byte, half-word or word
           -b defaults to 0x0 and -s defaults to 4

        For example, to decode address 0x4000 with bad bits
0 and 2 and it's
        a word, type:

        mem12 -a 0x4000 -b 0x5 -s 4
```

---

mem13 - byte read / write (see mem4) (slow: 15 minutes/32 MBytes)

---

mem14 − Read the mc3 config register

This is the same as the dmc command from pod mode. See also
mem1

---

mem15 − Double word MarchY pattern test (4 min / 128 MB)

Same as mem8 but does double word writes/reads instead of
word writes/reads.

---

mem16 − Knaizuk Hartmann Memory Test (3 min/128 MB)

This algorithm is used to perform a fast but non−exhaustive
memory test. It will test a memory subsystem for stuck−at
faults in both the address lines as well as the data
locations. The algorithm breaks up the memory to be tested
into three partitions. Partition 0 consists of memory
locations 0, 3, 6...; partition 1 consists of memory
locations 1, 4, 7...; and partition 2 consists of locations
2, 5, 8 .... The partitions are filled with either an all
1’s pattern or an all 0’s pattern. By varying the order in
which the partitions are filled and then checked, this
algorithm manages to check all combinations of possible
stuck−at faults.

---

mem17 − Three Bit Memory Test (12 min/128 MB)

This algorithm is designed as a pattern sensitivity test.
The intent is to surround a given cell of memory on both
sides with cells in the opposite state. To do all possible
combinations of ones surrounding zeros, and zeros
surrounding ones, it is necessary to run six patterns. The
test writes all of memory from low address to high address
memory, then reads back the data from low to high.

---

mem18 - Double-Word Knaizuk Hartmann Memory test (11/2 min/128 MB)

Same as test 16, except memory reads and writes are done 64 bits at a time, using the store/load double-word instruction.

---

ena_bnk, dis_bnk - enable / disable one bank at a time

Interactively asks the user for slot, leaf, and bank to act on. Do not run dis_bnk unless you know what you are doing. Otherwise, you will most likely hang the system.

2.4 IO4 IDE Guide

o Boot <u>ide</u>

o The default report level is 2. Set the report level by typing the following:

```
report=#
```

where # is any number from 1 to 5.

level 5	Debugging messages displayed. Don't need this much detail.
level 4	Prints out memory locations as they are written. Selecting this level slows down testing time.
level 3	Prints out 1-line functional descriptions within tests. This is probably the most useful level for general use.
level 2	Print out only errors, titles, and pass/fail.
level 1	Print out only titles and pass/fail.

Level <u>n</u> prints out all messages for level <u>n</u> and below.

o Set the modes of operations for running the tests:

- qmode [on | off]

All current IO4 tests run fast enough that there is no difference between quick and long test modes for the IO4. If the total elapsed time for running all IO4 tests ever exceeds 10 minutes, quick mode will be enabled for the IO4.

- c_on_error [on | off]

  For the IO4 tests, the ‘‘on’’ setting continues the test even when an error has been encountered. Setting the mode to ‘‘off’’ stops the tests after the first error.

o Run io all.

  This command runs all working/known bug-free IO4 tests that do not require human intervention. Any mostly working but possible buggy tests, as well as any tests requiring a human to interpret the results, are not included.

o There are currently tests for the following areas of the IO4 board: IO4 interface, VME adapter, SCSI adapter, and EPC adapter.

The detailed tests are listed below.

2.4.1 IO4 Interface ---------

check_iocfg - Checks IO4 config against NVRAM

This test compares the actual setup of the IO4 board to the values specified in the NVRAM. Each IO4 board in the system is checked to see that it has all the adapters specified in NVRAM, and that they are of the specified types.

In addition, if "report" is set VERBOSE, configuration information for each board is printed out even if no errors occur.

---

io4_regtest - Read/Write test of IO4 registers

This is a basic Read/Write test for the IO4 registers. It does tests and address-in-address testing for:

```
    IO4_CONF_LW
    IO4_CONF_SW
    IO4_CONF_ADAP
    IO4_CONF_INTRVECTOR
    IO4_CONF_GFXCOMMAND
    IO4_CONF_ETIMEOUT
    IO4_CONF_RTIMEOUT
    IO4_CONF_INTRMASK
```

Although these are not the only IO4 registers, they are the only ones that may safely be Read/Write tested.

---

io4_pioerr - IO4 PIO bus error test

Attempts to generate an error interrupt by attempting a write to IO adapter 0 (nonexistent). This tests the IO4 error generation capability and the IO4 to IP error path.

---

mapram_test - Read/Write test of IO4 map ram

As the name implies, tests the IO4 mapping ram as a small memory array.

Tests memory with pattern Read/Write, address-in-address, and marching 1's test patterns.

---

check_hinv - Checks type of board in each slot

Not a test per se - merely prints out the locations and types of all boards currently installed in the system.

2.4.2 VME Adapter ---------

---

fregs - Test VMECC registers

Basic Read/Write test for the F Chip registers, running
going through large window space. Verifies operation for:

    FCHIP_VERSION_NUMBER
    FCHIP_MASTER_ID
    FCHIP_INTR_MAP
    FCHIP_FIFO_DEPTH
    FCHIP_FCI_ERROR_CMND
    FCHIP_TLB_BASE
    FCHIP_ORDER_READ_RESP
    FCHIP_DMA_TIMEOUT
    FCHIP_INTR_MASK
    FCHIP_INTR_SET_MASK
    FCHIP_INTR_RESET_MASK
    FCHIP_SW_FCI_RESET
    FCHIP_IBUS_ERROR_CMND
    FCHIP_TLB_FLUSH
    FCHIP_ERROR
    FCHIP_ERROR_CLEAR
    FCHIP_TLB_IO 0 - 7
    FCHIP_TLB_EBUS 0 - 7

    (32 registers currently tested)

---

vmeregs - Test VMECC registers

Basic Read/Write test for the VMECC registers. Verifies
operation for:

    VMECC_RMWMASK
    VMECC_RMWSET
    VMECC_RMWADDR
    VMECC_RMWAM
    VMECC_RMWTRIG
    VMECC_ERRADDRVME
    VMECC_ERRXTRAVME
    VMECC_ERRORCAUSES
    VMECC_ERRCAUSECLR
    VMECC_DMAVADDR
    VMECC_DMAEADDR
    VMECC_DMABCNT
    VMECC_DMAPARMS
    VMECC_CONFIG
    VMECC_A64SLVMATCH
    VMECC_A64MASTER
    VMECC_VECTORERROR
    VMECC_VECTORIRQ 1 - 7
    VMECC_VECTORDMAENG
    VMECC_VECTORAUX0

```
VMECC_VECTORAUX1
VMECC_IACK 1 - 7
VMECC_INT_ENABLE
VMECC_INT_REQUESTSM
VMECC_INT_ENABLESET
VMECC_INT_ENABLECLR
VMECC_PIOTIMER
0x1388
0x1390
0x1398
0x13A0
0x13A8
0x13B0
0x13B8
0x13C0
0x13C8
0x13D0
0x13D8
0x13E0
0x13E8
0x13F0
0x13F8
```

(54 registers currently tested)

---

vmeintr - Test VMECC self interrupts

Places a handler on the appropriate vector and forces the VMECC to generate an interrupt. Checks to make sure that the interrupt both reaches the CPU and activates the proper handler.

---

vmeberr - Test VMECC bus errors

This test ensures that the VMECC can time out for all sections of A24/A32
 addresses.

First, A24 addressing is used. All sections except one are made to respond as slaves, and a PIO access to the non-responding slave section is done. This should generate a timeout.

This procedure is repeated for all 16 sections of the A32 addressing range.

---

vmedma - Test VMECC DMA Engine

Transfers data between controller memory & host memory by DMA; no disk data is involved (and no disk need be connected).

The controller imposes some constraints on what can be done with this test: it will transfer only a single sector (512 bytes).

---

vmelpbk - Test VMECC loopback capability

Tests using the VMECC loopback mode of operation. Mainly exercises the data path between the CPU and the VME bus.

---

cddata - cdsio board internal/external data loopback test

Does loopback testing of all channels of the cdsio 6-port board. Pretty much a direct port of the IP5 family version of this test.

---

cdintr - Test cdsio interrupts

Generates interrupts on the CDSIO board and verifies that they reach the CPU. Again, a direct port of the IO5 family test.

2.4.3 SCSI Adapter

---

s1_regtest - Register Read/Write test for s1 chip

This is a basic Read/Write test for the S1 chip registers. It does tests and address-in-address testing for:

```
    S1_INTF_R_SEQ_REGS 0 - 0xF
    S1_INTF_R_OP_BR_0
    S1_INTF_R_OP_BR_1
    S1_INTF_W_SEQ_REGS 0 - 0xF
    S1_INTF_W_OP_BR_0
    S1_INTF_W_OP_BR_1
```

(36 registers currently tested)

Although these are not the only S1 registers, they are the only ones that may safely be used by Read/Write tests.

---

regs_95a - Register read/write test for wd95a chip

This is a basic read/write test for the wd95a chip registers. In setup mode, the test writes a's and 5's to the "Sleep Countdown" register while all other registers are just read. In normal mode, all registers are read and no registers are written. While there are many registers, the Sleep Countdown register was the only register where it was safe to write values.

---

scsi_intr - SCSI interrupt test

This tests the wd95a's ability to send an interrupt to the cpu and have the system respond correctly. The wd95a is programmed to interrupt upon a scsi reset command. The reset command is sent and then the system is checked to make sure it correctly "saw" the interrupt.

---

scsi_self - SCSI senddiag test

This test sends a SCSI senddiag command to each SCSI device found on each SCSI bus and verifies that a good result is returned.

---

scsi_dmaxfer - SCSI DMA transfer test

This test checks SCSI DMA transfers by performing DMA reads and writes from each SCSI disk drive encountered. 1 block of data (512 bytes) is read and/or written. The command syntax is:

```
scsi_dmaxfer [-w] [-p <partition #>] [-f]
```

Without any switches, scsi_dmaxfer will read 512 bytes from partition 1 of the disk drive.

    -w:
perform a write operation to the disk drive from memory in addition to the read. The default is to only perform a read operation from the disk to memory. Writing is a destructive action and should only be used by experts. User confirmation is always requested unless a -f switch is used. The data written (a's on pass 1 and 5's on pass 2) is then read back

via another DMA operation and the written data is compared
to the read data.

-p #:

specify which disk partition to read or write to. The
default is partition #1.

-f:

the "force" switch. If specified, no user confirmation for
the writes will be requested. The default is to always
request user confirmation for any write operation. This
switch is designed to be used in a script where user input
is not desired (e.g. in an overnight oven script).

----

scsi_dmaintr - SCSI DMA interrupt test

This test checks the SCSI DMA interrupt functionality by
performing a DMA read from the disk and specifying a bad
read buffer address.

2.4.4 EPC Adapter

----

epc_regtest - Register Read/Write test for epc chip

Basic Read/Write test for the EPC chip registers, including
the Parallel Port registers. Registers tested:

```
    EPC_IIDDUART0
    EPC_IIDDUART1
    EPC_IIDENET
    EPC_IIDPROFTIM
    EPC_IIDSPARE
    EPC_IIDPPORT
    EPC_IIDERROR
    EPC_EADDR0
    EPC_EADDR1
    EPC_EADDR2
    EPC_EADDR3
    EPC_EADDR4
    EPC_EADDR5
    EPC_TCMD
    EPC_RCMD
    EPC_TBASELO
    EPC_TBASEHI
    EPC_TLIMIT
```

    EPC_TTOP
    EPC_TITIMER
    EPC_RBASELO
    EPC_RBASEHI
    EPC_RLIMIT
    EPC_RTOP
    EPC_RITIMER
    EPC_PPBASELO
    EPC_PPBASEHI
    EPC_PPLEN
    EPC_PPCTRL

As stated above, this is a good basic test for the Parallel Port; for more thorough testing a test fixture is required.

---

epc_nvram - NVRAM Read/Write test

Does Read/Write pattern and address-in-address testing for all the NVRAM accessible to the EPC chip. Although the NVRAM is physically on the RTC chip, it occupies a separate address space and is accessed differently, hence the separate test.

---

epc_rtcreg - RTC register/NVRAM Read/Write test

Read/Write test for the RTC registers and the small amount of NVRAM in the RTC address space portion of the RTC chip. Registers tested:

    NVR_SEC
    NVR_SECALRM
    NVR_MI
    NVR_MINALRM
    NVR_HOUR
    NVR_HOURALRM
    NVR_WEEKDAY
    NVR_DAY
    NVR_MONTH
    NVR_YEAR

NVRAM tested is in the address range 0xE - 0x3F.

---

epc_rtcinc - RTC clock increment test

Tests the ability of the RTC chip to handle time-of-day transitions. Sets the RTC to a known time and date (last second of the year), waits one second, and checks to make certain that the time and date have changed correctly.

---

epc_rtcint - RTC Interrupt generation test

Tests to make certain that the RTC can correct generate Alarm, Periodic, and Update interrupts. Validates the path from the RTC chip to the IP board’s master CPU.

---

duart_loopback - Duart loopback test

Attempts to configure and test all available serial ports. Does loopback testing at all baud rates for each port tested. Normally uses internal loopback, but if invoked with "duart_loopback -e" assumes that an external loopback fixture is being used.

---

erase_nvram - NVRAM Erase Utility (Dangerous!)

Erases all data in non-volatile ram. Used in the debug area to clear totally fouled-up configuration data.

erase_nvram must be invoked with a slot number - "erase_nvram 5" would erase the NVRAM data on the IO4 board in slot 5 of a system.

Normally only run in the debug/repair area. Do not run this on the master IO4 board unless you want to re-configure all setup data.

---

epc_extint - External Interrupt Read/Write Utility

Called with "epc_extint slot# pattern". Writes the hex value "pattern" to the external interrupt outputs and returns the value seen on the external interrupt inputs.

Since this test requires an external test fixture or jumper cables, it is not part of the standard "io_all" script.

---

enet_xmit - Ethernet transmit/receive test (with internal loopback on)

Transmits 9 packets and receives them with the LXT internal loopback mode on. The transmit and receive status bytes are checked against expected values, and the data in each byte of each packet is verified against what was expected. The following is a short description of each of the 9 packets:

packet 0: 50 data bytes, walk 0 through the bytes. packet 1: 50 data bytes, walk 1 through the bytes. packet 2: 50 data bytes, alternating 55's and aa's in alternating bytes. packet 3: 60 data bytes, alternating 0's and ff's in alternating bytes. packet 4: 50 data bytes, all 55's. packet 5: 1 data byte, short packet. packet 6: 130 data bytes, decrementing pattern starting with ff. packet 7: 3 data bytes, short packet. packet 8: 130 data bytes, decrementing pattern starting with cc.

---

enet_colctr - Read ethernet collision counters Test

Transmits the same 9 packets as enet_xmit but with LXT and EDLC in normal mode so packets will actually go out onto the net. The user is given instructions that he needs to run "ttcp -r -s" on one other machine and "ttcp -t -s <machine #1>" on machine #2 in Unix. Then by running this test, collisions will occur and we will display the collision counter counts after each packet is sent. There is not really a fail status in this test unless we are unable to transmit our packets. The test fails if the collision counter counts being displayed never increment.

---

epc_plptest - Parallel Port Write Test

Fills a buffer with printable characters and attempts to DMA it out the parallel port. Detects the presense of a printer - if a printer is present, configures the port to use the "BUSY" mode and writes to the printer using BUSY handshaking to prevent buffer overrun. If no printer is present, uses the "SACK" mode and writes as fast as the DMA engine sends characters.

If a printer is present, will detect printer errors and report them; in all cases it will detect DMA timeout errors or failure to generate the DMA completion interrupt.

If an external printer is present, these characters should
be printed out:

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[]^_
```

2.4.5 DANG Adapter ---------

----

dang_regtest - DANG Register Read/Write Test

This test does read/write verification of all DANG PIO
registers that are safely writable (some, such as the Master
DMA start register or the interrupt mask register, are not).

Runs a series of bit-pattern tests, marching ones and zeros,
and address-in-address patterns on the DANG chip PIO
registers. For this test to function correctly, the basic
DANG Ibus interface must be working.

Patterns used:

```
    0x0
    0xFFFFFFFF
    0x55555555
    0xAAAAAAAA
    0xA5A5A5A5
    0x5A5A5A5A
    Marching 1's (32 patterns) - 0x1, 0x2, . . . 0x80000000
    Marching 0's (32 patterns) - 0xFFFFFFFE, 0xFFFFFFFD, . .
. 0x7FFFFFFF
    Address in Address
    Inverse Address in Address
```

Registers currently tested:

```
    DANG_UPPER_GIO_ADDR
    DANG_MIDDLE_GIO_ADDR
    DANG_BIG_ENDIAN
    DANG_GIO64
    DANG_PIPELINED
    DANG_GIORESET
    DANG_AUDIO_ACTIVE
    DANG_AUDIO_SLOT
    DANG_PIO_WG_WRTHRU
    DANG_DMAM_MAX_OUTST
    DANG_DMAM_CACHE_LINECNT
    DANG_DMAS_MAX_OUTST
    DANG_DMAS_CACHE_LINECNT
    DANG_INTR_ERROR
```

```
DANG_INTR_GIO_0
DANG_INTR_GIO_1
DANG_INTR_GIO_2
DANG_INTR_DMAM_COMPLETE
DANG_INTR_PRIV_ERR
DANG_INTR_PAUSE
DANG_INTR_BREAK
DANG_WG_LOWATER
DANG_WG_HIWATER
DANG_WG_FULL
DANG_WG_PRIV_LOADDR
DANG_WG_PRIV_HIADDR
DANG_WG_GIO_UPPER
DANG_WG_GIO_STREAM
DANG_WG_PAUSE
DANG_WG_STREAM_ALWAYS
```

Sample Error Messages:

```
    Exception Accessing DANG chip!
    Failed to set DANG Intr mask. Expected: 0 Got: 1f
    Failed testing DANG register DANG_WG_FULL. Expected 400
Got 4F0
    Failed DANG reg address test. Reg: DANG_WG_FULL
Expected: 52018
    Got: 0
```

---

dang_gr2ram - GIO Bus/Gr2 Shared Ram Test

This test is designed to stress the GIO bus interface and prove that the basic GIO bus read/write functionality is present. As such, although it requires a working Express graphics board, dang_gr2ram only stresses the Express shared RAM area.

Like dang_regtest, dang_gr2ram runs a series of bit-patterns, marching ones and zeros, and address-in-address tests, though its target this time is the entire Express shared RAM area.

dang_gr2ram stresses the DANG's Ibus interface, the basic DANG IO configuration, the DANG GIO bus interface, and the GIO bus data/address lines in addition to the Express shared RAM.

Patterns used:

    0x0
    0xFFFFFFFF
    0x55555555
    0xAAAAAAAA
    0xA5A5A5A5
    0x5A5A5A5A
    Marching 1's (32 patterns) - 0x1, 0x2, . . . 0x80000000
    Marching 0's (32 patterns) - 0xFFFFFFFE, 0xFFFFFFFD, . . . 0x7FFFFFFF
    Address in Address
    Inverse Address in Address

Sample Error Messages:

```
    Failed Gr2 shared ram test. Index: 0x400 Expected:
0x55555555 Got: 0x5A
```

---

dang_mdma - DANG Master DMA Module Test

The DMA Master module test has a reasonably simple basic design: set up a Master DMA transfer either to or from the Gr2 RAM space, verify that it completed and the DMA complete interrupt was generated, and validate the transferred data.

To fully test the DMA module, however, requires that all the corner cases are covered, which requires many different transfers in a variety of DMA modes.

Currently, the dang_mdma test has 38 test cases, covering GIO bus dynamic and static address modes, transfers to and from the Gr2, data patterns, address-in-address data, single-line, multi-line, and multi-page transfers, transfers with no offset or stride, transfers with offset or stride, transfers with both offset and stride, etc.

dang_mdma stresses the DANG's Ibus interface, the basic DANG IO configuration, the DANG GIO bus interface, the Express shared RAM area, and the Dang DMA Master Interrupt logic as well as the DANG Master DMA module proper.

One cache line static address tests

```
    host to gr2, 0x55555555
    gr2 to host, 0xAAAAAAAA
```

One cache line data tests

```
    host to gr2, 0x55555555
    host to gr2, 0xAAAAAAAA
    host to gr2, 0xFFFFFFFF
    gr2 to host, 0x55555555
    gr2 to host, 0xAAAAAAAA
    gr2 to host, 0xFFFFFFFF
```

One partial cache line static address tests

```
    host to gr2, 0x55555555
    gr2 to host, 0xAAAAAAAA
```

One partial cache line data tests

```
    host to gr2, 0x55555555
    host to gr2, 0xAAAAAAAA
    host to gr2, 0xFFFFFFFF
    gr2 to host, 0x55555555
    gr2 to host, 0xAAAAAAAA
    gr2 to host, 0xFFFFFFFF

One cache line address tests

    host to gr2, address in address
    host to gr2, inverse address in address
    gr2 to host, address in address
    gr2 to host, inverse address in address

One partial cache line address tests

    host to gr2, address in address
    host to gr2, inverse address in address
    gr2 to host, address in address
    gr2 to host, inverse address in address

Multiple block data tests using stride and offset

    host to gr2, 0x55555555 (stride)
    gr2 to host, 0xAAAAAAAA (stride)
    host to gr2, 0x55555555 (stride+offset)
    gr2 to host, 0xAAAAAAAA (stride+offset)

Multiple block address tests using stride and offset

    host to gr2, address in address (stride)
    gr2 to host, inverse address in address (stride)
    host to gr2, address in address (stride+offset)
    gr2 to host, inverse address in address (stride+offset)

Full data buffer static address tests

    host to gr2, 0x55555555
    gr2 to host, 0xAAAAAAAA

Full data buffer data tests

    host to gr2, 0x55555555
    gr2 to host, 0xAAAAAAAA

Full data buffer address tests

    host to gr2, address in address
    gr2 to host, inverse address in address
```

Sample Error Messages:

```
    data setup problem: small static pattern 1
    small static pattern 1: (H to G) timed out waiting for
DANG interrupt
    xfer data, small static pattern 1 (H to G), line 0x1
byte 0x1: src 0x55
    dest 0x5A
    small static pattern 2 (G to H) wrong interrupt level:
was 20, sb 35
    small static pattern 3 (H to G) DMA xfer not complete
```

----

dang_wg - DANG Write Gatherer Test

The DANG write gatherer test, dang_wg, has three subtests, which test the WG FIFO RAM, the Host to DANG WG interface, and the WG interrupts.

Since each subtest must be working for the next to function correctly, the subtests are run in the order given, and are not written to be run separately.

The WG FIFO RAM tests runs bit patterns and address-in-address data through the FIFO and out to the Gr2 shared RAM area via the GIO bus. The test method is to use the DANG_WG_PAUSE register to pause output, fill the WG FIFO RAM, and then enable output. After the WG FIFO has drained, the data in the Gr2 shared RAM is verified.

Patterns used:

```
    0x55555555
    0xAAAAAAAA
    0xFFFFFFFF
    0x5a5a5a5a
    0xa5a5a5a5
    0x0
    Address in Address
    Inverse Address in Address
```

The Host to WG test sends data from the Host CPU’s write gatherer module using all four of the possible addressing modes (relative, absolute, streaming, and streaming always) and verifies that the data reaches the area of Gr2 shared RAM it was aimed at. Since the FIFO RAM has already been verified, the individual tests are less exhaustive.

Patterns used:

Relative Mode:	Address in Address
Absolute Mode:	Inverse Address in Address
Streaming Always Mode:	0x5A5A5A5A
Streaming Mode:	0xA5A5A5A5

Finally, the Interrupt tests use the Host to DANG WG interface to generate the FIFO high, FIFO full, FIFO low, and privilege violation interrupts. In each case, the data used to fill the buffer is verified after the interrupt has been generated.

Pattern used:

```
    Address in Address
```

dang_wg stresses the DANG’s Ibus interface, the basic DANG io configuration, the Write Gatherer FIFO RAM, the DANG GIO bus interface, the Express shared RAM area, and the WG Interrupt logic as well as the DANG Write Gatherer module proper.

Sample Error Messages:

```
    DANG wg fifo not empty - had 23 words
    DANG wg fifo: bad word count - was 0x2ff, sb 0x3ff
    DANG fifo write through data error: addr 0x1000, was 0x0
sb 0x55555555
    DANG host wg data error: addr 0x80, was 0x0 sb 0x80
    wrong dang wg interrupt level - was 0x20, sb 0x73
    dang wg interrupt bit bad - was 0x0, sb 0x4
    no wg fifo hi interrupt!
    no wg fifo low interrupt!
    no wg fifo full interrupt!
    no wg privileged interrupt!
```

----

dang_status - DANG Status display utility

Displays the current state of the DANG chip. This is the same routine called by the DANG tests when an error is detected.

When possible, status information is given both as numeric values and named states. An example status is:

```
    +      dang_pio_err: 0x200
    +        11..8: dang_pio_err_version: 0x2
    +      dang_dmam_status: 0x1
    +         0: dang_dma_stat_busy
    +         1: dang_dma_stat_dir: 0x0
    +        dang_dmam_status, dma if 1 - ibus to fifo:
dang_dma_if_idle <0>
    +        dang_dmam_status, dma if 2 - fifi to gio :
dang_dma_if_idle <0>
    +        dang_dmam_status, dma if 3 - gio to fifo :
dang_dma_if_idle <0>
    +        dang_dmam_status, dma if 4 - fifo to ibus:
dang_dma_if_idle <0>
    +      dang_dmam_err: 0x0
    +      dang_dmas_status: 0x0
    +        dang_dmas_status, dma if 1 - ibus to fifo:
dang_dma_if_idle <0>
    +        dang_dmas_status, dma if 2 - fifi to gio :
dang_dma_if_idle <0>
    +        dang_dmas_status, dma if 3 - gio to fifo :
dang_dma_if_idle <0>
    +        dang_dmas_status, dma if 4 - fifo to ibus:
dang_dma_if_idle <0>
    +      dang_dmas_err: 0x0
    +      dang_intr_status: 0x408
    +         3: dang_istat_wg_flow
    +        10: dang_istat_giostat
    +      dang_wg_status: 0x1
    +         0: dang_wgstat_idle
    +         4..3: dang_wgstat_fill: 0x0
    +         7..5: dang_wgstat_wext: 0x0
    +         9..8: dang_wgstat_drain: 0x0
```

----

dang_gr2read - Gr2 Read utility

"dang_gr2read slot# adapter# gr2address"

Reads an address on the GIO bus. A GIO bus "peek" routine.

Requires IO4 slot number, dang adapter number, and Gr2 offset (base of the Gr2 shared RAM area is offset 0).

All numbers may be in decimal or hexadecimal - hex numbers should be preceded by "0x".

"dang_gr2read 11 5 0" would read Gr2 location 0 on DANG adapter 5 of the IO4 board in slot 11; so would "dang_gr2read 0xb 5 0".

---

dang_gr2write - Gr2 Write utility

"dang_gr2read slot# adapter# gr2address pattern"

Writes one word to a specified location in the GIO bus. The syntax is as given for dang_gr2read.

"dang_gr2write 11 5 0 0x55555555" would write 0x55555555 hex to Gr2 location 0 on DANG adapter 5 of the IO4 board in slot 11.

---

dang_gr2readloop

"dang_gr2readloop slot# adapter# gr2address loopcount"

dang_gr2writeloop

"dang_gr2writeloop slot# adapter# gr2address pattern loopcount"

Scope loop versions of the read and write utilities. dang_gr2readloop requires a loop count after the standard dang_gr2read parameters; similarly, dang_gr2writeloop requires a loop count following the standard dang_gr2write parameters.

"dang_gr2writeloop 11 5 0 0x55555555 100000" would write 0x55555555 to Gr2 location 0 on DANG adapter 5 of the IO4 board in slot 11 one hundred thousand times.

3. IP17 Crimson Standalone Diagnostics System

The IP17 diagnostic package, ide, is a standalone program that can be invoked from the PROM Monitor or SASH to test a variety of hardware components. The package includes a powerful command parser that allows the user to repeatedly execute tests in a given order, as well as run pre-programmed sets of tests, such as an overnight stress test. In addition, ide has the ability to log the results of the tests and send the logging information to a remote machine or to a printer. ide also has the ability to do auto-configuration; that is, the test will be skipped if the required piece of hardware, such as a Interphase disk controller, does not exist.

3.1 Description of Tests

The IP17 diagnostics tests contain the following eight categories:

CPU TEST:	23 tests
BUS TEST:	4 tests
MEM TEST:	9 tests
PATH TEST:	41 tests
IO TEST:	29 tests
FPU TEST:	5 tests

The CPU tests provide rigorous testing of each CPU subsystem feature except the MP bus interface. The instruction cache, first level data cache, second level data cache, instruction buffer, CPU counter/timer, TLB, timer, RMP ASIC, and duarts are explicitly tested.

The BUS tests stress the sync bus controller and semaphore RAM. The tests include SBC registers test, semaphore RAM test, SBC interrupts test and semaphore operations test.

The MEM tests contain the basic memory addressing and stuck-at bits tests. The memory array is accessed through the unmapped uncached (k1) memory space (0xa0000000 - 0xafffffff) without using the cache.

The cache states and cache<->memory data paths are tested by the PATH diagnostics. Each possible cache state and instruction is tested for the primary data cache, primary instruction cache, and secondary cache. These are very thorough tests, whose only drawback is their duration - they take roughly an hour to complete.

The IO tests stress the basic IO3 functionalities. The IO3 functionalities include: the IO3 local registers, mapper, ECC logging RAM, VME and SCSI DMA transfer, interrupts, VME memory space, ECC, bus error exceptions, nvram, sound, and ethernet.

The FPU tests attempt to execute all single and double precision FPU instructions, access all FPU registers, and check all FPU error cases. Since the R4000 FPU is integrated into the CPU, if any errors occur, the CPU itself needs to be replaced.

3.2 How to Run the Diagnostics

The ide program can be installed in one of four places: on a streaming tape, on the disk volume header, on a network server under the UNIX file system, or on the local disk under the UNIX file system. Booting the diagnostic off of the local disk or network is most convenient. Note: it is best to keep a copy on tape in case neither boot sequence works.

3.2.1 Maintenance Menu The following menu is displayed when the system is reset. Enter the boot <parameters> command at the Command Monitor Prompt. The various combinations of parameters are described below.

```
System Maintenance Menu

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

Option? 5

>>
```

3.2.2 Installing and Running from a Streaming Tape First, obtain the lastest version of ide. Until this package is shipped, the most recent version can be found in morc:/usr/tmp/ide.IP17 (sgi internal use only). Copy this file to the current working directory of a running workstation. Install the ide binary on the tape by executing the following command in the UNIX shell:

```
# /etc/mkboottape ide
```

To run the diagnostics, load the tape into the tape drive of the desired machine. From the PROM Monitor (recognizable by the ">>" prompt), type the following command if using a SCSI tape controller:

```
>> boot -f tpsc(0,7)ide
```

3.2.3 Running Diags over the Network Copy the ide binary to a running machine, preferably one on the local subnet that is usually up. Suppose the binary has been installed on the machine server under /stand/ide. Then to run the diagnostics, type the following command to the PROM Monitor:

```
>> boot -f bootp()server:/stand/ide
```

Note: If the binary is installed under /usr/local/boot on server, it is not necessary to specify the complete pathname. For example, the following command will be sufficient.

```
>> boot -f bootp()server:ide
```

3.2.4 Installing and Running the Diagnostics from the Disk Volume Header Perhaps the fastest way to boot the diagnostics is from the volume header on the root disk. Unfortunately, the diagnostics can only be installed while the system is running UNIX. After installation, however, the diagnostics can be invoked regardless of the state of the file system. To install the diags, first copy the ide binary to the current directory of the working machine. Then type the following in a UNIX shell:

```
# /etc/dvhtool -v creat ide ide /dev/vh
```

After bringing the system down and returning to the PROM Monitor, the diagnostic package can be booted by either of the two methods shown below.

Type the following if using a SCSI disk:

```
>> boot -f dksc(0,1,8)ide
```

Type the following if using a ESDI disk:

```
>> boot -f dkip(0,0,8)ide
```

3.2.5 Installing and Running the Diagnostics from the UNIX File System. The diagnostic package may be run from the root file system of the disk. The diagnostics should be run from the root file system on very stable systems since a great deal of hardware and software must work reliably. To install the diags, copy the ide binary to a directory on the root file system, such as /stand in the example below. Then, boot from the PROM Monitor using either of the commands shown below.

To boot from the PROM Monitor using a SCSI disk, type:

```
>> boot dksc(0,1,0)/stand/ide
```

To boot from the PROM Monitor using an ESDI disk, type:

```
>> boot dkip(0,0,0)/stand/ide
```

Note the absence of the -f flag to boot. This flag causes the PROM Monitor to quietly invoke SASH and search for the file name. Then, SASH will actually boot the file. Only SASH is able to traverse the UNIX file system for files.

3.3 Diagnostic Commands

After successfully booting ide, a banner like the following will be displayed on the screen:

```
Diags Version 4D1-4.0 IP17 OPT Wed May 11 XX:XX:XX PDT 19XX SGI
Memory size: 1677216 (0x1000000) bytes

************************************************************
Special Note:
      These diagnostics are to be used by authorized SGI
      personnel only.
      Please use the following predefined options:
      "a" (for all tests),          "x cpu" (for cpu tests)
      "x fpu" (for fpu tests),      "x io" (for io tests)
      "x mem" (for mem tests),      "x bus" (for bus tests)
      "x path" (for path tests)
************************************************************
Note: Disk Write Mode is OFF
      (some tests which write to disk partition 1 are not
      going to run)
      If you want to run these tests,
      please use "f k 1" to turn the Disk Write Mode on
      and use "f w 0" to turn the Warning Message Mode off.
************************************************************

DIAGS:
```

3.3.1 Help Command

For a display of the available command and options, type:

```
DIAGS: ? (or "help" or "h')
```

The following information will be displayed:

```
COMMANDS:
help:                   ? [COMMAND(s)]
auto execute:           a [TEST NAME]
hardware configuration: c
dump:                   d [s] [v]
system configuration:   f [p #][m RANGE][d DEVS][b #][r #][e #][v #][s #][a #]
help:                   h [COMMAND(s)]
init logfile:           i
print logfile:          l [LINES]
menu:                   m [MENU(s)]
quit with reboot:       q
exit to prom:           e
execute:                x {expression [arg(s)][; expression ...]}*count
                          where:
                          expression    testname[sequence][*count]
                          sequence      number | (testnumbers[*loop][,...])
                          testnumbers   number | number1-number2

Example:
        DIAGS: x {c*3; cpu2;M(1,2)*3 arg1;b(1*2,5*2,2)*2 arg1 arg2}*0

The backslash character (\) can be used for multiple input lines

Command lines take either uppercase or lowercase characters

Control Characters
        Erase single characters by CTRL-H or DEL
        Rubout entire line by CTRL-U
        Suspend the test by CTRL-C
```

3.3.2 <u>Menu Command</u> The menu command displays the tests available to execute. Invoked with no arguments, the menu command displays the test categories:

```
DIAGS: m
```

The following test categories would be displayed:

```
CPU TEST
BUS TEST
MEM TEST
PATH TEST
IO  TEST
FPU TEST
```

To list the BUS tests with a brief description, type;

```
DIAGS: m bus (or "m b")
```

A table similar to the following will be displayed:

```
bus1:   test the semaphore ram as a small memory
bus2:   test SBC registers for stuck-at faults
bus3:   syncbus interrupt test
bus4:   semaphore operations test
                .
                .
                .
```

3.3.3 Execute Command The arguments to the execute command may look complex, but they need not be. To run a single test, such as the first CPU test, type:

```
DIAGS: x cpu1 (or x c1)
```

Assuming the hardware is working correctly, ide will respond with:

```
running cpu1
cpu test pass, pass count = 1, skip count = 0
```

To run the eleventh IO test, type:

```
DIAGS: x io11 (or x i11)
```

Assuming the required interphase controller does not exist, ide will respond with:

```
running io11
io test skip, skip count = 1
```

To run the first and third CPU test, type:

```
DIAGS: x cpu(1,3)
```

The response will be:

```
running cpu1
running cpu3
cpu test pass, pass count = 2, skip count = 0
```

To run the first through the third CPU test, type:

```
DIAGs: x cpu(1-3)
```

The response will be:

```
running cpu1
running cpu2
running cpu3
cpu test pass, pass count = 3, skip count = 0
```

The tests can be executed from different categories with a single command. To run the first three CPU tests and the first FPU test, type:

```
DIAGS: x {cpu(1-3);fpu1}
```

The response will be:

```
running cpu1
running cpu2
running cpu3
running fpu1
cpu test pass, pass count = 3, skip count = 0
fpu test pass, pass count = 1, skip count = 0
```

To run a test a number of times, use the "*" construct. To run the first FPU test three times after the CPU tests, type the following:

```
DIAGS: x {cpu(1-3);fpu1*3}
```

The response will be:

```
running cpu1
running cpu2
running cpu3
running fpu1
cpu test pass, pass count = 3, skip count = 0
fpu test pass, pass count = 3, skip count = 0
```

The entire sequence of tests can be executed a desired number of times. To run the above sequence of tests twice, type:

```
DIAGS: x {cpu(1-3);fpu1*3}*2
```

The response will be:

```
running cpu1
running cpu2
running cpu3
running fpu1
running cpu1
running cpu2
running cpu3
running fpu1
cpu test pass, pass count = 6, skip count = 0
fpu test pass, pass count = 6, skip count = 0
```

To run a test infinite times, use "0". To run the first FPU test after the first CPU test in infinite loop, type the following:

```
DIAGS: x {cpu1;fpu1}*0
```

The response will be:

```
running cpu1
running fpu1
running cpu1
running fpu1
running cpu1
running fpu1
     .
     .
     .
```

To run the pre-programmed set of tests, type the name of the category. For example, to run the FPU tests, type the following:

```
DIAGS: x fpu (or x f)
```

The response will be:

```
running fpu1
running fpu2
running fpu3
running fpu4
running fpu5
running fpu14
fpu test pass, pass count = 14, skip count = 0
```

To execute pre-programmed tests from different categories with a single command. To run the BUS tests and FPU tests, type:

```
DIAGS: x {bus;fpu}
```

The response will be:

```
running bus1
running bus2
running bus3
running bus4
running fpu1
running fpu2
running fpu3
running fpu4
running fpu5
running fpu14
bus test pass, pass count = 4, skip count = 0
fpu test pass, pass count = 14, skip count = 0
```

3.3.4 System Configuration Command The system configuration command allows the user to display and change the default behavior of the diagnostic package. To display the default settings, type the following.

```
DIAGS: f
```

A table similar to the following will be output:

```
m: memory testing area: from 0x400000 to 0xfffffc
b: block mode is on
r: remote log file is on
e: current error mode is 1 which is continue after error occurs
v: Verbose Mode
k: Disk Write Mode is off (no tests will write to disks)
w: Warning Message Mode is off
a: current auto execution table is fe table
d: vme devices for io tests are ipi(0,0,1)
t: scsi devices for io tests are dksc(0,1,1)
i: vme devices for mp tests are ipi(0,0,1) ipi(1,0,1)
```

```
c: scsi devices for mp tests are dksc(0,1,1) dksc(1,1,1)
p: Parity and ECC exceptions enabled
```

3.3.4.1 Memory Range Configuration Option The "m" configuration variable is the range to test memory in the memory tests. The memory tests ordinarily test memory from the four megabyte mark to the top of installed memory. To test up to the six megabyte mark, type:

```
DIAGS: f m 0x400000:0x600000 (or f m 0x400000#0x100000)
```

3.3.4.2 VME Block Mode Configuration Option The "b" configuration variable indicates the VME block mode. If the VME controller is Interphase 4021, the VME block transfer will be performed in the IO tests. Otherwise, the regular VME transfer will be performed. This variable can to set to 0 to be regular VME mode.

3.3.4.3 Remote Logging Mode Configuration Option The "r" configuration variable is the remote logging mode which is to send the logging information through a serial line to a remote machine or printer. Port 2 is used to connect remote machine or printer. If port 2 is used for some other purpose, remote logging mode can be disabled by setting "r" to 0.

3.3.4.4 Error Mode Configuration The "e" configuration variable is the error mode. This variable can be set to 1 (continue after errors), 2 (stop after first error), or 3 (go into scope loop after error).

Note: This final option does not work for mp tests.

3.3.4.5 Verbosity Level Configuration Option The "v" variable indicates the verbosity level. If verbosity is set to 0, only error messages are printed. Otherwise, informative messages may be output during test execution.

3.3.4.6 Disk Write Mode Enable/Disable Option The "k" variable indicates whether the io tests may write to disk. If set to 0 (default value) disk writes are disabled, and tests that need to write to disk are skipped. If this variable is set to 1, disk writes are permitted.

3.3.4.7 Disk Write Warning Option The "w" variable indicates whether ide should prompt you for the correct disk write status before running any groups of tests that might write to disk. Default value is 1 (on).

3.3.4.8 Test Script Configuration The "a" variable indicates which test script should be executed. This option can be set to 0-6 (overnight stress test) or 7 (fe diags, which complete in ~30 minutes). The default is level 7 (fe diags).

3.3.4.9 VME Devices Configuration Option The "d" configuration variable is the VME devices to test in the IO tests. The default device is ipi(0,0,1). To test two ESDI disks on two different controllers, type:

```
DIAGS: f d dkip(0,0,1) dkip(1,0,1)
```

3.3.4.10 SCSI Devices Configuration Option The "t" configuration variable is the SCSI devices to test in the IO tests. The default device is dksc(0,1,1). tpsc(0,7) will also be the default device if there is one in the system. To test two SCSI disks on two different controllers, type:

```
DIAGS: f t dksc(0,1,1) dksc(1,1,1)
```

3.3.4.11 Parity/ECC Exception Option The "p" configuration variable is used to select whether Parity and ECC exceptions are to be normally enabled or disabled. Parity/ECC exceptions are enabled by default; to disable them use the command:

```
DIAGS: f p 0
```

To re-enable the Parity/ECC exceptions, use:

```
DIAGS: f p 1
```

Note: In the IP5 diagnostics, the "f p" command is used to select which processors will be running tests, and has no connection to exception handling.

3.3.5 Logfile Commands Pass count, fail count, skip count, test executions, and error messages are automatically logged. To print the log file, type l or <Ctrl-l>. <Ctrl-l> can print the log file on the fly. That means <Ctrl-l> can be typed during test execution, and the log file will be printed without suspending the test.

```
DIAGS: l
```

A sample log file as will be displayed:

```
TEST SUMMARY:
cpu test:  pass count = 2, error count = 0, skip count = 0
```

```
bus test:  pass count = 0, error count = 0, skip count = 0
mem test:  pass count = 0, error count = 0, skip count = 0
path test: pass count = 0, error count = 0, skip count = 0
io test:   pass count = 2, error count = 0, skip count = 0
fpu test:  pass count = 2, error count = 0, skip count = 0
```

```
MESSAGES:
P0:running cpu1
P0:running io04
P0:running on device dkip(0,0,1)
P0:running on device dksc(0,1,1)
P1:running on device dkip(0,0,1)
P1:running on device dksc(0,1,1)
P0:running fpu1
```

The log file statistics can be reset and the saved messages erased by typing "i".

3.3.6 <u>Auto Execute Command</u> Invoked with no arguments, the command executes multiple categories tests. To run the PATH tests, type;

```
DIAGS: a path (or "a p")
```

The response will be:

```
running all path tests
running path1
running path2
running path3
running path4
running path43
path test pass, pass count = 43, skip count = 0
```

Note: ‘‘a p’’ and ‘‘x p’’ run same set of tests.

3.3.7 <u>Hardware Configuration Command</u> This command displays the hardware configuration which provides the information for users to set up the system configuration to run the tests.

```
DIAGS: c
```

A sample hardware configuration is as follows:

```
Memory size:            8 Mbytes
Instruction cache size: 64 Kbytes
data cache size:        64 Kbytes
SCSI Disk:              dksc(0,1)
SCSI Tape:              tpsc(0,7)
```

3.3.8 Dump Command The dump command displays IO2 mapper. Invoked with no arguments, the command displays both SCSI and VME mappers.

To dump the SCSI mapper, type:

```
DIAGS: d s
```

To display VME mapper, type:

```
DIAGS: d v
```

3.3.9 Quit Command Typing q to the diagnostic prompt will reboot the system, running the PROM start up diagnostics before returning to the start up menu.

```
DIAGS: q
>>
```

3.3.10 Exit Command Typing e to the diagnostic prompt will return the system to the start up menu without running the start up diagnostics. This is ~2 minutes faster than the q command.

```
DIAGS: e
>>
```

3.4 How to Change Pre-programmed Test Tables

The pre-programmed test tables are structured as two levels. The higher level table is selected by the system configuration command with the "a" variable. Under this table, there are nine subtables. One of these tables is for multi-categories auto execution (‘‘a’’ command without any argument), and the others are for eight individual categories to respond to ‘‘x’’ or ‘‘a’’ command with an argument, such as ‘‘a c’’ or ‘‘x b.’’

All these tables are located in *jake*:/*jake*/*att*/*usr*/*src*/*stand*/*IP5diags*/*interface*/*execute*.*c*.

The higher level table is called defined_table:

```
struct predefined_lev defined_table[] = {
         { USER,     io2_tables},
         { SANITY,   sanity_tables},
         { LONG,     auto_tables},
};
```

Each line of this table represents a different level of auto-execution. For example, io2_tables is an IO2 stress test, and auto_tables is an over night stress test. To run auto_tables, type:

```
f a 2
```

auto_tables is defined as follows:

```
struct excmd auto_cpu[2];
struct excmd auto_mem[2];
struct excmd auto_bus[2];
struct excmd auto_path[2];
struct excmd auto_io[2];
struct excmd auto_fpu[2];
struct excmd auto_all[11];

struct auto_table auto_tables[] = {
{ "cpu",     auto_cpu},
{ "cpu",     auto_cpu},
{ "bus",     auto_bus},
{ "mem",     auto_mem},
{ "path",    auto_path},
{ "io",      auto_io},
{ "fpu",     auto_fpu},
{ "all",     auto_all},
{0},
};
```

Each line of this table corresponds to one individual category except the first line is dummy and the last line is multi-categories. The format of subtable is defined as:

```
{TEST NAME, TEST NUMBER, LOOP-COUNT, TEST NUMBER, LOOP-COUNT, .....
 ......., GLOBAL LOOP-COUNT, 0}
```

For example, the following table will run CPU test 1 to 24, but skip test 9.

```
struct excmd auto_cpu[] = {
        { CPU, 1,1,2,1,3,1,4,1,5,1,6,1,7,1,8,1,10,1,11,1,12,1,13,1,14,1,15,
1,16,1,17,1,18,1,19,1,20,1,21,1,22,1,23,1,24,1,0,0,0,0,         1,      0},
        {0},
};
```

To run memory 7 ten times after memory 1, 2, and 4, the table can be defined as:

```
struct excmd auto_mem[] = {
        { MEM, 1,1,2,1,4,1,7,10,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,  1,      0},
        {0},
};
```

To execute the whole bus tests twice, the table is:

```
struct excmd auto_bus[] = {
        { BUS, 1,1,2,1,3,1,4,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,    2,      0},
        {0},
};
```

The following table will skip the whole fpu tests:

```
struct excmd auto_fpu[] = {
        { FPU, 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,    1,      0},
        {0},
};
```

The multi-categories table is as follows:

```
struct excmd auto_all[] = {
        { CPU, 1,1,2,1,3,1,4,1,5,1,6,1,7,1,8,1,9,1,10,1,11,1,12,1,13,1,14,1,15,
1,16,1,17,1,18,1,19,1,20,1,21,1,22,1,23,1,24,1,0,0,     1,      0},
        { BUS, 1,1,2,1,3,1,4,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,    1,      0},
        { MEM, 1,1,2,1,4,1,7,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,    1,      0},
        { PATH, 1,1,2,1,3,1,4,1,5,1,6,10,7,1,8,10,10,1,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,      1,      0},
        { IO, 1,1,2,1,3,1,4,1,5,1,6,1,7,1,8,1,9,1,10,1,11,1,12,1,13,1,14,1,15,
1,16,1,17,1,18,1,19,1,20,1,0,0,0,0,0,0,0,0,0,0, 1,      0},
        { FPU, 1,1,2,1,3,1,4,1,5,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,    1,      0},
        {0},
};
/}
```