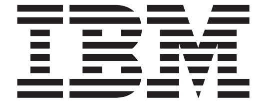
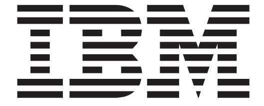
AIX 5L Version 5.1



Understanding the Diagnostic Subsystem for AIX

AIX 5L Version 5.1



Understanding the Diagnostic Subsystem for AIX

Fifth Edition (April 2001)

Before using the information in this book, read the general information in “Appendix. Notices” on page 221.

This edition applies to AIX 5L Version 5.1 and to all subsequent releases of this product until otherwise indicated in new editions.

A reader’s comment form is provided at the back of this publication. If the form has been removed, address comments to Publications Department, Internal Zip 9561, 11400 Burnet Road, Austin, Texas 78758-3493. To send comments electronically, use this commercial Internet address: aix6kpub@austin.ibm.com. Any information that you supply may be used without incurring any obligation to you.

© Copyright International Business Machines Corporation 1997, 2001. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About This Book . . . xi
Who Should Use This Book . . . xi
Highlighting . . . xi
ISO 9000 . . . xi
Related Publications . . . xi
Prerequisite Software . . . xi
Trademarks . . . xi

Chapter 1. Introduction . . . 1
Structure of Diagnostics . . . 1
Strategy for Diagnostics . . . 3
Staging the Impact of Diagnostics . . . 4
Option Checkout . . . 4
System Checkout . . . 4
Diagnostic Commands . . . 5
diag Command . . . 5
diagrpt Command . . . 5

Chapter 2. Operating Environments . . . 7
Online Diagnostics . . . 7
Concurrent Mode . . . 7
Service Mode . . . 8
Maintenance Mode . . . 8
Standalone Diagnostics (POWER-based only) . . . 8
Tasks not Supported in Standalone Diagnostics . . . 8
Console Configuration Diskette . . . 8
NIM Diagnostics . . . 9

Chapter 3. Diagnostic Components . . . 11
Diagnostic Controller . . . 11
Control Flow of the Diagnostic Controller . . . 12
Return Status . . . 13
Diagnostic Applications . . . 13
Device Configuration . . . 14
Determining the Level of Tests to Execute . . . 14
Drivers Used for Diagnostic Purposes . . . 14
Acquiring a Greater Share of the Resource . . . 15
Error Log Analysis . . . 15
Enhanced Error Handling (EEH) Option . . . 16
Persistent Variables . . . 16
Field Replaceable Units (FRUs) . . . 17
Specifying a Text Conclusion . . . 17
Library Restrictions for Diagnostic Programs . . . 17
Guidelines for Writing Diagnostic Programs using C++ . . . 18
Completion Status for Diagnostic Applications . . . 18
Control Flow of a Diagnostic Application . . . 18
SRN Architecture . . . 19
Diagnostic Controller Generated SRNs . . . 19
Source Numbers . . . 20
Diagnostic Application Code Checklist . . . 20
Other test scenarios include: . . . 21
Tasks and Service Aids . . . 22
Creating a Task . . . 22

© Copyright IBM Corp. 1997, 2001

Performing a Task . . . 22
Task List. . . . 23
Add or Delete Drawer Configuration. . . . 24
Add Resource to Resource List . . . 24
Shell Prompt . . . 25
Analyze Adapter Internal Log (Device Specific) . . . 25
Backup and Restore Media . . . 25
Certify Media . . . 25
Change Hardware Vital Product Data . . . 25
Configure Dials and LPFKeys . . . 26
Configure ISA Adapter. . . . 26
Configure Reboot Policy (CHRP). . . . 26
Configure Remote Maintenance Policy (CHRP) . . . 27
Configure Ring Indicate Power On (RSPC) . . . 29
Configure Ring Indicate Power On Policy (CHRP) . . . 29
Configure Service Processor (RSPC) . . . 29
Surveillance Setup . . . 29
Modem Configuration . . . 30
Call In/Out Setup . . . 30
Site Specific Call In/Out Setup. . . . 30
Configure Surveillance Policy (CHRP) . . . 30
Create Customized Configuration Diskette . . . 31
Delete Resource from Resource List . . . 31
Disk Maintenance (SCSI Disks) . . . 31
Display Checkstop Analysis Results. . . . 32
Display Configuration and Resource List . . . 32
Display Firmware Device Node Information (CHRP) . . . 32
Display Hardware Error Report . . . 33
Display Hardware Vital Product Data . . . 33
Display Machine Check Error Log . . . 33
Display Microcode Level . . . 33
Display or Change Bootlist . . . 33
Display or Change BUMP Configuration . . . 33
Display or Change Diagnostic Run Time Options . . . 34
Display or Change Electronic Mode Switch . . . 34
Display or Change Multiprocessor Configuration (Multiprocessor Service) . . . 35
Display Previous Diagnostic Results . . . 35
Display Resource Attributes. . . . 35
Display Service Hints . . . 35
Display Software Product Data . . . 35
Display System Environmental Sensors (CHRP) . . . 36
Display Test Patterns . . . 37
Download Microcode . . . 37
ESCON Bit Error Rate Service Aid . . . 38
Fibre Channel RAID Service Aids (Device Specific) . . . 38
Flash SK-NET FDDI Firmware. . . . 38
Format Media . . . 39
Generic Microcode Download . . . 39
Hot Plug Task . . . 39
Local Area Network Analyzer . . . 39
PCI RAID Physical Disk Identify . . . 39
Periodic Diagnostics . . . 40
Process Supplemental Media . . . 40
Run Diagnostics . . . 40
Run Error Log Analysis . . . 40
Save or Restore Hardware Management Policies (CHRP) . . . 40

Save or Restore Service Processor Configuration (RSPC) . . . 41
SCSD Tape Drive Service Aid . . . 41
SCSI Bus Analyzer . . . 42
Service Aids for use with Ethernet . . . 42
Spare Sector Availability . . . 43
SSA Service Aids . . . 43
Update Disk Based Diagnostics . . . 43
Update System Flash (RSPC) . . . 43
Update System or Service Processor Flash (CHRP). . . . 44
7135 RAIDiant Array Service Aid . . . 44
Application Test Units . . . 45
Test Unit Definition . . . 46
Hardware Functional Coverage . . . 46
Test Unit Numbering . . . 46
Test Unit Code Device Open and Close . . . 46
Portability . . . 47
In-Service versus Out-of-Service Test Units . . . 48
Recommended General Structure of Test Unit Code. . . . 48
Designing for Multitasking Environments . . . 49
Persistent Data and the TU_INFO_HANDLE . . . 49
Test Unit Call Interface . . . 50
Definition of TU_TYPE Input Structure . . . 50
Definition of TU_RETURN_TYPE Output Structure . . . 51
Return Codes . . . 51
major_rc . . . 51
minor_rc . . . 52
Interrupt Handler Call Interface . . . 52
Syntax . . . 52
Parameters . . . 52
Interrupt Handling in Test Units . . . 52
Using the Interrupt Flag Bit Mask. . . . 54
Example . . . 54
Programming Interfaces for TUs and Interrupt Handlers . . . 55
Configuration Services Device Attributes . . . 55
Message Handling . . . 56
Signal Handling . . . 57
Definition of EXECTU() . . . 57
Purpose . . . 57
Syntax . . . 57
Description . . . 57
Parameters . . . 57
Return Value . . . 58
PCI Configuration Space for I/O Devices . . . 58
Test Unit 64-bit Porting Guide . . . 60
C Language Data Model . . . 60
Makefile . . . 60
Makefile Source . . . 60
SLIH Conversion Tips . . . 61
SLIH Conversion Required Changes . . . 62
Related Information . . . 63
Microcode Download/Display Requirements for Test Units . . . 63
Enhanced Error Handling Option . . . 63
Diagnostic Kernel Extension . . . 63
Overview . . . 64
Device Configuration . . . 64
Loading PDIAGEX . . . 64

Second-Level Interrupt Handlers . . . 64
Programming Interfaces for libpdiag.a . . . 65
pdiag_diagnose_state . . . 65
pdiag_restore_state . . . 66
pdiag_cs_open . . . 66
pdiag_cs_close . . . 66
pdiag_cs_get_attr . . . 67
pdiag_cs_free_attr . . . 68
pdiag_open . . . 68
pdiag_close . . . 69
pdiag_pcicfg_read . . . 70
pdiag_pcicfg_write . . . 71
Programming Interfaces for PDIAGEX . . . 72
pdiag_dd_watch_for_interrupt . . . 72
pdiag_dd_interrupt_notify . . . 73
pdiag_dd_write, pdiag_dd_write_64 . . . 74
pdiag_dd_read, pdiag_dd_read_64 . . . 76
pdiag_dd_dma_setup . . . 79
pdiag_dd_dma_complete . . . 80
pdiag_dd_dma_enable . . . 82
pdiag_eeh_errinjct . . . 83
pdiag_read_slot_reset . . . 83
pdiag_set_eeh_option . . . 84
pdiag_set_slot_reset . . . 85
Data Dictionary . . . 86
PDIAGEX Data Structures . . . 86
pdiagex_dds_t . . . 86
pdiagex_opflags_t . . . 87
dma_struct . . . 87
aioo_struct_t . . . 88
diag_struc_t . . . 88
Kernel Services . . . 89
Programmed I/O Services . . . 90
Diagnostic Library . . . 91
diag_add_obj . . . 93
diag_change_obj . . . 93
diag_close_class . . . 94
diag_free_list . . . 94
diag_get_list . . . 95
diag_lock . . . 95
diag_open_class . . . 96
diag_rm_obj . . . 96
diag_unlock . . . 97
init_dgodm, term_dgodm . . . 97
configure_device, initial_state . . . 97
diagex_cfg_state . . . 98
diagex_initial_state . . . 99
get_device_status . . . 99
addfrub . . . 100
insert_frub . . . 102
diag_catopen . . . 102
diag_cat_gets . . . 103
diag_resource_screen . . . 103
diag_popup . . . 106
diag_task_screen . . . 106
diag_progress . . . 109

diag_read . . . 109
diag_asl_clear_screen . . . 110
diag_asl_init . . . 110
diag_asl_msg . . . 111
diag_asl_read . . . 112
diag_asl_quit . . . 112
diag_display . . . 113
diag_display_menu . . . 114
diag_emsg . . . 115
diag_msg, diag_msg_nw . . . 115
diag_get_device_flag . . . 116
diag_get_property . . . 117
diag_get_sid_lun . . . 117
get_cpu_model . . . 118
get_dev_desc . . . 118
get_diag_att . . . 119
dlog_close . . . 120
dlog_find_first . . . 120
dlog_find_next . . . 121
dlog_find_sequence . . . 121
dlog_formatElogResults . . . 122
dlog_freeEntry . . . 122
dlog_getTestMode . . . 123
dlog_open . . . 123
dlog_read . . . 124
dlog_same_elogId . . . 125
dlog_setEntryType . . . 126
dlog_write . . . 126
save_davars_ela . . . 127
copy_text . . . 128
DA_SETRC_XXXXXX, DA_CHECKRC_XXXXXX, DA_EXIT . . . 128
diag_asl_beep . . . 130
diag_asl_execute . . . 131
diag_exec_source . . . 131
diag_execute . . . 132
dt . . . 132
error_log_get . . . 133
file_present . . . 135
get_DApp . . . 135
getdainput, clrdainput . . . 136
getdavar, putdavar . . . 136
ipl_mode . . . 137
getELAdates . . . 137
has_diag_authority . . . 138
menugoal . . . 138
schedule_ela . . . 139
Diagnostic Object Classes . . . 139
Predefined Diagnostic Resource Object Class . . . 140
Predefined Diagnostic Attribute Device Object Class . . . 143
Predefined Diagnostic Task Object Class . . . 145
Customized Diagnostic Attribute Object Class . . . 146
Test Mode Input Object Class . . . 147
Menu Goal Object Class . . . 150
FRU Bucket Object Class . . . 150
FRU Reporting Object Class . . . 151
Diagnostic Application Variables Object Class . . . 152

Predefined Diagnostic Devices Object Class . . . 153
Diagnostic Supervisor Menu Options Object Class . . . 155
Diagnostic Header Files . . . 156
Diagnostic User Interface . . . 156
Diagnostic Applications . . . 156
Diagnostic Tasks . . . 159
Diagnostic Menu Examples . . . 165
Diagnostic Operating Instructions Menu . . . 165
Function Selection Menu . . . 165
Define Terminal Menu . . . 165
Missing Resource Selection Menu . . . 166
Missing Resource Menu . . . 166
New Resource Menu. . . . 167
Diagnostic Mode Selection Menu . . . 167
Resource Selection Menu . . . 167
Resource Selection Menu - Display Common Tasks . . . 168
Test Method Menu . . . 168
No Trouble Found Menu . . . 169
Problem Report Menu . . . 169
Additional Resources Menu . . . 170
Task Selection List Menu . . . 170
Task Selection List Menu - Display Supported Resources . . . 170
Run Time Options Menu . . . 171

Chapter 4. Diagnostic Features . . . 173
Missing Options Resolution . . . 173
Online Concurrent Diagnostics . . . 173
Online Service Diagnostics . . . 174
Standalone Diagnostics (POWER-based only) . . . 175
Missing Options Procedure Steps . . . 175
Error Log Analysis. . . . 177
Running Problem Determination Mode in Diagnostics. . . . 177
Periodic Diagnostics . . . 177
AIX Version 3 . . . 177
AIX Version 4 and Itanium-based platform . . . 177
Technical Description. . . . 178
Automatic Error Log Analysis (DIAGELA) . . . 178
Loop Testing . . . 180

Chapter 5. Diagnostic Packaging . . . 181
Hardfile Packaging . . . 181
Software Packages and Filesets . . . 181
Directory Structure Organization . . . 181
CDROM Packaging (POWER-based only) . . . 182
Diagnostic Supplemental Media . . . 182
Diagnostic Supplemental Diskette Contents . . . 182
Example ODM Stanzas . . . 183
Example diagstartS Script File . . . 184
Example diagstart3S Script File . . . 185
Diagnostic Supplemental Diskette Label. . . . 186

Chapter 6. Diagnostic Debugging Hints . . . 187
Debugging Hints for Diagnostic Applications . . . 187
Debugging Hints for Diagnostic Kernel Extension . . . 187
Starting Trace for Diagnostic Kernel Extension . . . 187
Running Trace for Diagnostic Kernel Extension in the Background . . . 188

Finding the Right Address . . . 188
Looking at an Illegal Trap . . . 189
Diagnostic Patch Diskette Procedure . . . 190
Diagnostic Configuration Diskette . . . 191
Diagnostic Patch Diskette . . . 191
Diagnostic Debug Diskette. . . . 191

Chapter 7. Code Examples . . . 193
Example {DEVICE}_ERR_DETAIL.H: TU Specific Outputs . . . 193
Example {DEVICE}_INPUT_PARAMS.H: TU Specific Inputs . . . 194
Example TU Local Header File . . . 194
Example TU exectu Function. . . . 195
Example TU Open/Close Device Interface . . . 197
Example TU Makefiles . . . 203
Example C Source File for TU Interrupt Handler. . . . 203
Example TU Interrupt Handler Makefile . . . 204
Example Diagnostic Application . . . 206
Example Diagnostic Application Message File . . . 215

Chapter 8. Diagnostic Task Matrix . . . 217

Appendix. Notices . . . 221

Index . . . 223

About This Book

This publication describes the hardware diagnostic subsystem for the POWER-based platform and the Itanium-based platform.

Who Should Use This Book

The book is intended for developers of diagnostic applications, application test units, device-driver test units, the diagnostic controller, and the diagnostic user interface.

Highlighting

The following highlighting conventions are used in this guide:

Bold	Identifies commands, subroutines, keywords, files, structures, directories, and other items whose names are predefined by the system.
italics	Identifies parameters whose actual names or values are to be supplied by the user.
`monospace`	Identifies examples of specific data values, examples of text similar to what you might see displayed, examples of portions of program code similar to what you might write as a programmer, messages from the system, or information you should actually type.

ISO 9000

ISO 9000 registered quality systems were used in the development and manufacturing of this product.

Related Publications

- *AIX 5L Version 5.1 General Programming Concepts: Writing and Debugging Programs*
- *AIX 5L Version 5.1 Kernel Extensions and Device Support Programming Concepts*
- *AIX 5L Version 5.1 Commands Reference*
- *AIX 5L Version 5.1 Technical Reference: Base Operating System and Extensions Volume 1*

Prerequisite Software

For the POWER-based platform, the **bos.diag** 4.3.3+ package is required.

For the Itanium-based platform, the **bos.diag** 5.1.0 package is required.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

- IBM
- Micro Channel
- PowerPC
- RS/6000

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be the trademarks or service marks of others.

© Copyright IBM Corp. 1997, 2001

Chapter 1. Introduction

This chapter contains the following topics:
- Structure
- Strategy
- Diagnostic Commands

The Structure section gives an overview of the diagnostic system. Key application modules are described and their relationships to one another is shown.

Also, a figure is displayed that shows the relationship between the Diagnostic Controller, Diagnostic Applications, and Application Test Units.

The Strategy section gives an overview of the strategy used by the diagnostic system to discover and analyze problems on the system.

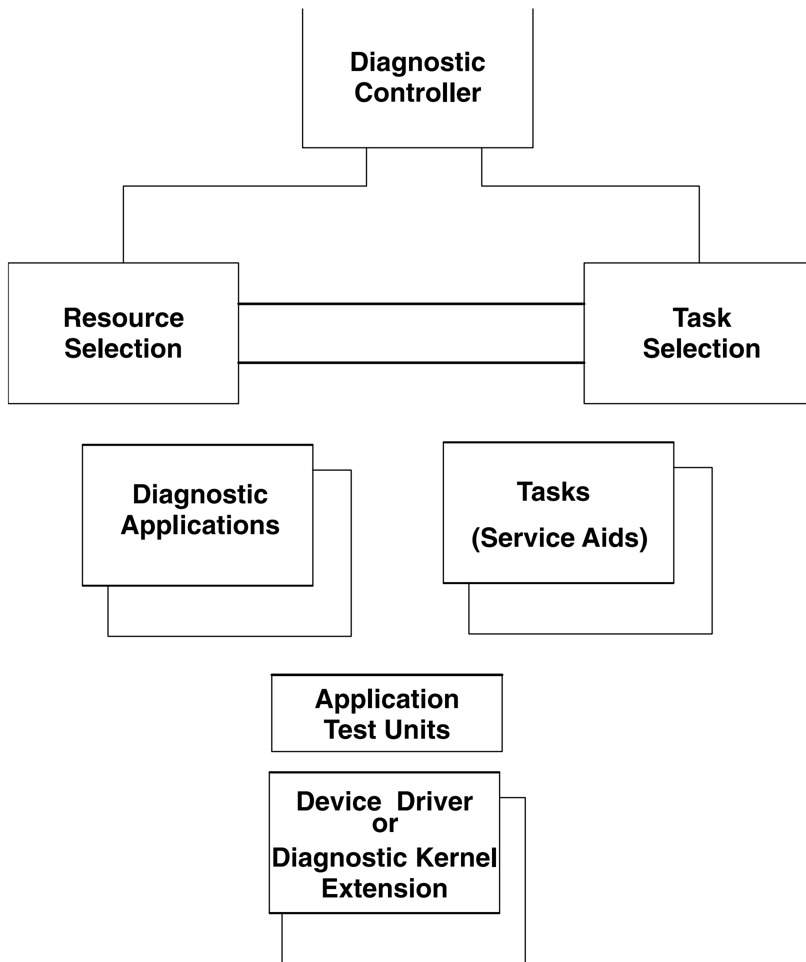
The Diagnostic Commands section gives the usage and command line flags for the **diag** and **diagrpt** commands.

Structure of Diagnostics

The Diagnostic System is a collection of application modules that work together to perform some software or hardware action. This collection of application modules are comprised of various distinct components.

© Copyright IBM Corp. 1997, 2001

The following figure illustrates the diagnostic architecture:



Diagnostic Architecture

The architecture shows that the Diagnostic Controller has two main functions:

- Resource Selection
- Task Selection

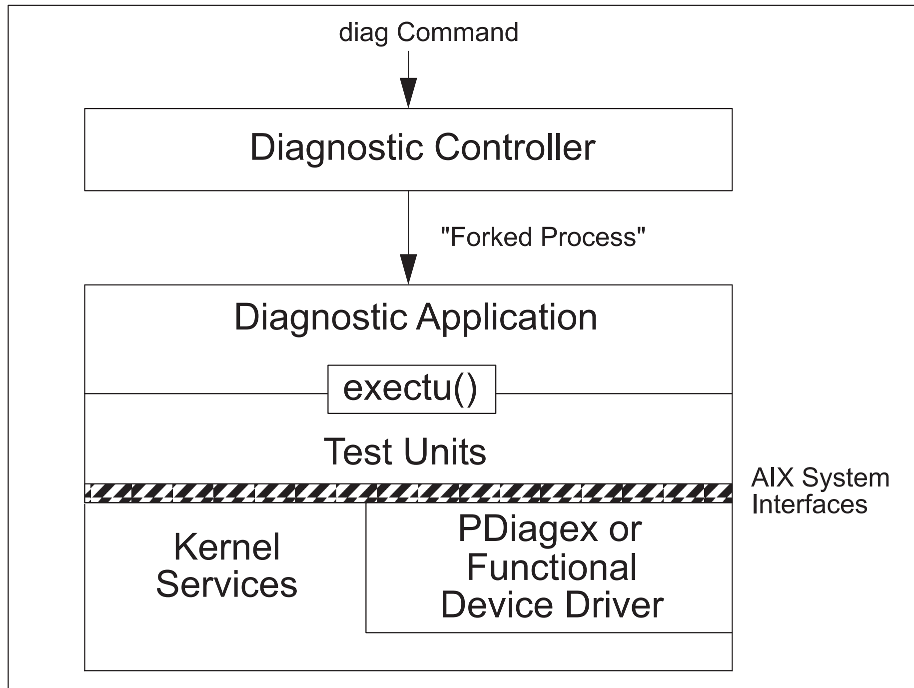
Tasks are operations that can be performed on a resource. Running Diagnostics, Displaying VPD, or Formatting a Resource, are examples of tasks. Service aids are also considered as tasks.

Resources are devices contained by the system unit. The diskette drive and CD ROM drive are examples of resources.

The Function Selection Menu contains selections allowing either resources or tasks to be displayed. When *Task Selection* is made and a task has been selected, a list of resources supporting that task is displayed. Alternatively, when *Resource Selection* is made, and a resource or group of resources are selected, then a list of tasks supporting the selected resources is displayed.

A Diagnostic Application or Task, may involve the use of Application Test Unit code, which in turn may involve the use of a Diagnostic Kernel Extension, or a Device Driver to gain access to the hardware.

The figure below illustrates the current diagnostic structure that allows access to diagnostic function concurrent with system operation. Diagnostics for a given resource consists of an executable file containing Diagnostic Application code, which controls the execution of one or more Application Test Units. This executable is started by the Diagnostic Controller, which allows the user to select diagnostic modes and devices to test. To properly execute the Application Test Units, the Diagnostic Application currently must have detailed specific knowledge about each of the Application Test Units.



Diagnostic Structure

The **executu()** interface is the call interface for Application Test Units, and contains all the information necessary to run the Application Test Unit against a particular device and return results. **PDiagex** is a special generic device driver written for use by Application Test Units, which can be used in place of the functional device driver to provide a simple direct interface to the device under test. Doing so places a greater requirement on the Application Test Unit to directly manipulate the device hardware, but in doing so, it provides earlier use of the Application Test Unit during the hardware bring-up and debug phase, since the Application Test Unit is not dependent on the availability of a working functional device driver.

Strategy for Diagnostics

The strategy for diagnostics is founded on:

- Staging diagnostics based on underlying hardware capabilities according to three levels of testing:
  - Shared
  - Subtest
  - Full-test
- Isolating defective field replaceable units (FRUs) such that there is the least impact to the system. This is accomplished by either:
  - Option Checkout
  - System Checkout

Staging the Impact of Diagnostics

The impact of diagnostics is staged. There are three levels of tests supported by diagnostics:

Shared
: The tests in this category are nondisruptive. Diagnostics does not need exclusive access to run these tests. All Diagnostic Applications (DA) should support the shared testing category since DAs perform error-log analysis. Other possible shared tests are error circuitry testing, cyclic redundancy checks of Loadable ROS, On Board Self Tests (provided the appropriate recovery procedures are included), and selected functional testing such as diagnostic reads and writes.

Subtest
: The tests in this category apply to multiplexed resources such as Native I/O Planar and multiport async cards. The sub-tests are disruptive, but only to a portion of the resource. To run these tests, diagnostics needs exclusive access to the portion of the resource that is being tested.

Full-test
: The tests in this category impact the entire resource. Diagnostics must have exclusive access to the entire resource to run these tests.

Option Checkout

If the configuration is viewed as a tree structure, diagnostics starts testing at the leaves of the tree, and moves vertically and horizontally down the tree toward the root. The leaves represent terminal devices, and the root is the processor.

The following algorithm generally describes the isolation strategy. It starts at an arbitrary node in the tree and isolates to the correct FRU bucket based on the good or bad status of siblings and parent resources.

The steps are:

1. Test resource *x*. If no problems are detected, no further isolation is required.
2. Test a sibling of resource *x*, called resource *y*. If no problems are found, the fault of resource *x* is isolated to resource *x*.
3. Test the parent of resources *x* and *y*. If no problems are detected, the problem has not been isolated to a single failing resource. The FRU buckets associated with resources *x* and *y* will both be reported. No further isolation is required. However, if the parent fails its tests, disregard the failures of resources *x* and *y* and continue isolating the problem for the parent.

This general process of testing siblings and parents is repeated until a resource passes its tests or until a DA indicates that no further testing is required.

The diagnostic subsystem attempts to isolate to a single failing device. When multiple child devices fail their tests, the fault most likely lies with the parent. Thus the DA testing the parent in step 3 should name the parent as being defective and indicate that no more devices should be tested, in which case the diagnostic controller would only report the parent. The status of the child devices that have been tested is identified in the DA's control block.

System Checkout

Each resource in the system that has not been deleted from the resource selection list is tested during system checkout. System Checkout selection is accomplished by selecting All Resources from the Resource Selection Menu. User interaction is not allowed unless a problem has been detected and a question needs to be asked to isolate the problem.

Configuration processing for system checkout is different from that for option checkout, which impacts the effectiveness of the FRU Callout. Option checkout is the specification of an individual resource to test. When option checkout is chosen, the option chosen is tested first, and if a problem is found, it is traced back through its siblings and parents until it has been isolated. The configuration is processed from the outside in. When system checkout is chosen, the configuration is processed from the inside out. For example, the configuration is processed starting with the system planar, and works its way out on a

per-card basis. First a card is tested, then the devices attached to the card are tested, and then the devices attached to the device attached to the card are tested, and so on. This process is repeated for each card attached to the system planar.

Option Checkout is more effective because the children are tested before the parent, which allows the parent to determine its own culpability above and beyond its own test results. The parent can implicate itself for no other reason than that its children are failing.

Diagnostic Commands

This chapter describes the commands available in the Diagnostic Subsystem.

- diag Command
- diagrpt Command

diag Command

The **diag** command performs hardware problem determination. When you suspect there is a problem, this command assists you in finding it. The command has the following syntax:

diag [-a] [-c] [-d *name***] [-e] [-s] [-v] [-A] [-B] [-E** *days***] [-T** *taskstring***] [-S** *testsuite***]**

Most users should enter the **diag** command without any flags. The following flags perform various actions:

-a	Processes the changes in the hardware configuration. For example, missing and/or new resources.
-c	Indicates that the machine will not be attended. No questions will be asked. Results are written to standard output. Normally used by shell scripts.
-d *name*	Names the resource that should be tested. The *name* parameter is a resource name displayed by the **lscfg** command.
-e	Causes the device's Diagnostic Application to be run in Error Log Analysis mode.
-s	Causes the system to be tested in System Checkout mode.
-v	System Verification Mode. Default is Problem Determination mode.
-A	Advanced mode. Default is non-advanced mode.
-B	Tests the base system devices, such as planar, memory, processor.
-E *days*	Number of days used to search the error log.
-T *taskstring*	Specifies a particular Task to execute. The *taskstring* depends on the particular task to be executed. See Tasks for more information.
-S *testsuite*	Tests the Test Suite Group:

1	Base System
2	I/O Devices
3	Async Devices
4	Graphics Devices
5	SCSI Devices
6	Storage Devices
7	Commo Devices
8	Multimedia Devices

diagrpt Command

Displays the conclusions made by diagnostics. The **diagrpt** command has the following syntax:

/usr/lpp/diagnostics/bin/diagrpt [-o] [-s *startdate***] [-a] [-r]**

The **diagrpt** command reports the conclusions made by diagnostics.

If the user does not specify a flag, a scrollable menu with all diagnostic conclusion reports is displayed.

-o	Displays the lastest diagnostic conclusion.
-s *startdate*	Reports diagnostic conclusions made after the date specified (*mmddyy*).
-a	Displays the long version of the Diagnostic Event Log.
-r	Displays the short version of the Diagnostic Event Log.

Chapter 2. Operating Environments

This chapter contains the following topics:
- Online Diagnostics
- Standalone Diagnostics
- NIM Diagnostics

The Diagnostics operating environment consists of online and standalone diagnostics. The two environments differ in the way they are packaged, installed, and executed. Diagnostics is a collection of applications, the majority of which are device specific. These applications are packaged as filesets, with each fileset associated with a device.

Online diagnostics is commonly referred to as running diagnostics from an installed hardfile. This implies that the operating system, and the various device related packages have been installed.

Standalone diagnostics are packaged on removable media. The removable media contains the operating system, and all device related applications, device drivers, ODM stanzas, etc. supported at a particular release level. Third party devices and other devices not available for inclusion on the removable media at release time are supported by Diagnostic Supplemental Media.

Hardware Diagnostics can also be performed on NIM clients using a diagnostic boot image from a NIM server, rather than booting from removable media or hardfile. Not only does this eliminate the need for diagnostic boot media, it eliminates the need to have diagnostics installed on the local hardfiles of the client machines.

Diagnostics are a secure application. The user must know the appropriate password to run diagnostics. Diagnostics are inherently destructive, but this destructiveness is managed. The run-time status of each device identifies the level of diagnostics that can be safely executed. In addition, the testing has been structured so that some tests can only be executed in standalone mode.

Online Diagnostics

Online diagnostics can be run in three modes:

Concurrent Mode Allows the normal system functions to continue while selected resources are being checked.

Service Mode Allows checking of most system resources.

Maintenance Mode Allows checking of most system resources.

Concurrent Mode

Concurrent mode provides a way to run online diagnostics on some of the system resources while the system is running normal system activity. Because the system is running in normal operation, devices such as the following may require additional actions by the user or diagnostic application before testing can be done.
- SCSI adapters connected to paging devices
- Disk drive(s) used for paging, or are part of the *rootvg*
- LFT devices and graphic adapters if a Windowing system is active
- Memory
- Processor

© Copyright IBM Corp. 1997, 2001

Service Mode

Service mode provides the most complete checkout of the system resources. This mode also requires that no other programs be running on the system. All system resources, except the SCSI adapter and the disk drives used for paging, can be tested. However, note that the memory and processor are only tested during Power On Self Tests (POSTs).

Service Mode is entered by booting the operating system in service mode.

Maintenance Mode

Maintenance mode provides the exact same test coverage as Service Mode. The difference between the two modes is the way they are invoked. Maintenance mode requires that all activity on the operating system be stopped.

The **shutdown -m** command is used to stop all activity on the operating system and put the operating system into maintenance mode. After setting the terminal type, use the diag command to start Diagnostics.

Standalone Diagnostics (POWER-based only)

Standalone diagnostics provide a method to test the system when the online diagnostics are not installed and a method of testing the disk drives that cannot be tested by the online diagnostics.

Standalone diagnostics are currently packaged on CDROM. They are run by placing the Standalone Diagnostic CDROM into the cdrom drive, then booting the system in service mode.

The Standalone Diagnostic CDROM file system is mounted over a RAM-file system for execution. Because of this, the CDROM drive (and the SCSI controller that controls it) cannot be tested by the standalone diagnostics.

Device support that is not on the Diagnostic CDROM must be supported by Diagnostic Supplemental Media.

Tasks not Supported in Standalone Diagnostics

Some tasks and service aids are not supported in standalone diagnostics. This is due to the fact that Standalone diagnostics runs from a RAM-file system, they have no direct access to the hardfile.

See the Diagnostic Task Matrix for the list of supported tasks and their operating environments.

Console Configuration Diskette

The Standalone Diagnostic Package allows the use of a Console Configuration Diskette to accomplish two tasks:

- Use a Different Async Terminal as the Console
- Set the Refresh rate on a High-Function Terminal

The Create Customized Configuration Diskette task allows this diskette to be created.

Different Async Terminal for Console

The Standalone Diagnostic Package allows a terminal attached to any RS232 or RS422 adapter to be selected as a console device. The default device is an RS232 tty attached to the first native serial port. However, a file is provided allowing the console device to be changed. The file name is **/etc/consdef**. The format of the file is:

```
# COMPONENT_NAME: (cfgmeth) Device Configuration Methods
#
# FUNCTIONS: consdef
#
# ORIGINS: 27, 28
```

```
#
# (C) COPYRIGHT International Business Machines Corp. YYYY,YYYY
# All Rights Reserved
# Licensed Materials - Property of IBM
#
# US Government Users Restricted Rights - Use, duplication or
# disclosure restricted by GSA ADP Schedule Contract with IBM Corp
#
#
# The console definition file is used for defining async terminal
# devices, which are the console candidates at system boot. During
# system boot,all natively attached graphic displays, any async
# terminal on native serial port s1, and async terminals defined in # this file will display the "Select System Console" message. Only
# one terminal may be selected as console. If the terminal
# attributes are not specified in this file, default values from the # odm database are assumed. However, the location and connection
# attributes are mandatory. The location value may be displayed with # the lsdev command.
#
# The entries must be in the following format:
#
#ALTTTY:
#       connection=value
#       location=nn-nn-ss-nn
#       attribute=value
#       .
#       .
#ALTTTY:
#       connection=value
#       location=nn-nn-ss-nn
#       attribute=value
#       .
#       .
# Lines in this file must not exceed 80 characters. All comments
# must be preceded by a pound sign (#) in the first column.
#
# For backward compatibility, the "ALTTTY:" keyword is not required
# for the first entry.
#
#
# For example, to display the console selection message on the ttys # attached to the S1 and S2 ports, uncomment the following stanzas:
#
#ALTTTY:
#       connection=rs232
#       location=00-00-S1-00
#       speed=9600
#       bpc=8
#       stops=1
#       xon=yes
#       parity=none
#       term=ibm3163
#ALTTTY:
#       connection=rs232
#       location=00-00-S2-00
#       speed=9600
#       bpc=8
#       stops=1
#       xon=yes
#       parity=none
#       term=ibm3151
```

High-Function Terminals 60/77-Mhz Refresh Rate

Certain high-function terminals may be set to run at a different refresh rate. The Console Configuration Diskette may be created setting the appropriate refresh rate for the terminal used as the console. The Standalone Diagnostic Package uses the default 60-Mhz rate. The Create Customized Configuration Diskette task allows this value to be changed, and a new Console Configuration Diskette to be created.

NIM Diagnostics

Hardware diagnostics can be performed on all NIM clients using a diagnostic boot image from a NIM server, rather than booting from removable media or hard disk. This is useful for standalone clients, because the diagnostics do not have to be installed on the local hardfile. Diagnostic support comes from a SPOT resource.

In addition, diskless and dataless clients have another way of loading diagnostics from the network. You can boot a diskless or dataless client from the network the same way you do for normal use, but with the machine’s key mode switch in the Service position. If the client’s key mode switch is in the Service

position at the end of the boot process, hardware diagnostics from the server's SPOT are loaded. If a standalone client boots with the key in the Service position, the diagnostics (if installed) are loaded from the hard disk.

Running diagnostics in a NIM environment is very similar to running in Standalone mode.

See the Network Installation Management Guide and Reference for more information on the NIM environment.

Chapter 3. Diagnostic Components

This chapter contains information on the various components that make up the Diagnostic Subsystem environment.

- Diagnostic Controller
- Diagnostic Applications
- Tasks & Service Aids
- Application Test Units
- Diagnostic Kernel Extension
- Diagnostic Library
- Diagnostic Object Classes
- Diagnostic Header Files
- User Interface
- Diagnostic Menu Examples

Diagnostic Controller

The Diagnostic Controller function is started when the root user enters the **diag** command. Various flags that allow operations to be performed directly may be specified as input. For example, a flag may specify that the system or a particular resource is to be tested or that the system is to be run unattended. If no flags are specified, then the Diagnostic Controller presents menus to determine what the user wants to do.

Diagnostic object classes define the resources and tasks available for the Diagnostic Controller to work with. Predefined data in these object classes specify various attributes about the resources and tasks that may be available on the system.

The **Customized Device** object class (**CuDv**)contains information describing the resource instances actually defined to the system. A defined resource instance may or may not have a corresponding device driver that is used to control it. A resource may be a rack, drawer, adapter, disk, memory card, floating point chip, planar, bus, and so on.

The Diagnostic Controller is a data-driven program. It uses information found in both the **CuDv** and the Predefined Diagnostic Resources object class (**PDiagRes**) to generate a list of supported resources. This list of supported resources is used to build the Resource Selection menu.

Given the user's selection from the Resource Selection Menu, the Diagnostic Controller employs the **PDiagRes** object class to determine the appropriate Diagnostic Application (DA) to start. The Diagnostic Controller waits for the DA to complete. Diagnostic Application status is returned by the exit system call.

The Diagnostic Controller employs a system-wide view of the configuration enabling the Diagnostic Controller to walk through the configuration database testing resources. For example, if a resource fails its tests, the Diagnostic Controller may attempt to test other resources until the problem has been isolated. The Diagnostic Controller understands the dependencies between the resources. The term ″resource″ is used in a generic sense and includes adapters, as well as terminal devices.

The Diagnostic Controller analyzes the conclusions made by the Diagnostic Applications and generates a Problem Report. The Problem Report lists the field replaceable units (FRUs) that should be replaced, the probability of failure associated with each FRU, and the reason why the diagnosis was made.

The Diagnostic Controller writes its analysis to the directory **/etc/lpp/diagnostics/data**, and the **diagrpt** command, or ″Display Previous Diagnostic Results″ task, can be used at a later date to retrieve these results.

© Copyright IBM Corp. 1997, 2001

Control Flow of the Diagnostic Controller

Invoking the diag command without any flags starts the Diagnostic Controller which performs the following:

1. Displays the Operating Instructions menu. The version number will reflect the version of the Diagnostic code installed.
2. Displays the Function Selection menu, and starts the command associated with the user's selection.

Invoking the diag command with flags starts the Diagnostic Controller and passes the flags on to the Controller.

The Diagnostic Controller performs the following tasks:

1. Initialize the user interface. It is assumed that if there is no display and keyboard, then the initialization will fail.
   - If **-a**, then performs configuration management.
   - If **-s**, then performs system checkout once.
   - If **-S#**, then runs diagnostics on the resources indicated by the Test Suite ID.
   - If a flag was not specified, Diagnostics prompts the user.
2. From the Function Selection Menu, allows the user to select one of the following:
   - Select **Diagnostics**
   - Select **Advanced Diagnostics**
   - Select **Task Selection Menu**
   - Select **Resource Selection Menu**
3. If **Diagnostics** or **Advanced Diagnostics** is selected, then the following happens:
   - The Diagnostic Mode Selection menu is displayed, to determine if **System Verification** or **Problem Determination** should be run.
   - If **Problem Determination** is chosen, then the Diagnostic Controller automatically scans the error log for any PERMANENT HARDWARE errors that have been logged within the last 7 days to determine if any devices should be automatically tested. A problem report may be generated.
   - Walks the configuration database to determine which resources in the current configuration can be tested. This information is presented in the Resource Selection Menu.
   - If **Advanced Diagnostics Routines** is chosen, and the system is in Online Service mode of operation, the Diagnostic Controller will display the **Test Method** menu to determine if the tests should be repeated.
   - Initializes the input parameters to the Diagnostic Application (DA), which are contained in the TMInput - Test Mode Input object class.
   - Runs the Diagnostic Application (DA) of the resource to be tested.
   - Waits for the DA to complete.
   - The Diagnostic Controller then:
     – Performs isolation process.
     – Presents conclusions to the screen.
     – If no trouble is found, diagnostics exits with a return value of 0. Otherwise, a value of 1 is returned if the hardware was tested bad.
4. If **Task Selection Menu** is selected, then the following happens:
   - The Diagnostic Controller displays a list of Tasks that are available for the system.
   - After a task has been selected, a Resource Selection Menu will appear if the selected task supports a resource selection. After selection of a Resource, the task is called with the selected resource name as a command-line argument.
   - If the selected task does not support resource selection, then the task is invoked.
5. If Resource Selection Menu is selected, then the following happens:

- The Diagnostic Controller displays a list of Resources available on the system.
- After a Resource has been selected, a Task Selection Menu will appear containing the commonly supported tasks for each selected Resource. After selection of a task, the task is invoked.

Return Status

The Diagnostic Controller returns the following values:

Diagnostic Controller Return Values		
DIAG_EXIT_GOOD	0	No problems found
DIAG_EXIT_DEVICE_ERROR	1	Error running diagnostics
DIAG_EXIT_INTERRUPT	2	Received an interrupt while running diagnostics
DIAG_EXIT_NO_DEVICE	3	Device to test was not found in system configuration
DIAG_EXIT_BUSY	4	Another **Dctrl** program is running
DIAG_EXIT_LOCK_ERROR	5	Cannot create lock file for diagnostic controller
DIAG_EXIT_OBJCLASS_ERROR	6	Error accessing ODM database
DIAG_EXIT_USAGE	7	Usage error
DIAG_EXIT_SCREEN	8	Screen size incorrect
DIAG_EXIT_NoPDiagDev	9	Device not supported by diagnostics
DIAG_EXIT_NO_DIAGSUPPORT	10	Diagnostics is not supported
DIAG_EXIT_NOT_MISSING	11	Device is not missing
DIAG_EXIT_NO_AUTHORIZATION	12	User is not authorized to run diagnostics
DIAG_EXIT_KERNSUPPORT	13	Device is not supported on the 64-bit kernel

Diagnostic Applications

Note: The Diagnostic subsystem supports 32-bit diagnostic applications only.

Most resources in a system have a Diagnostic Application (DA), started by the Diagnostic Controller, that tests an area. DAs are associated with each resource supported by diagnostics in the configuration database.

DAs analyze the error log, display prompts and questions to the user, control which tests are run, call **Application Test Units**, and analyze test results.

The following topics are discussed in detail:

- Device Configuration
- Determining the Level of Tests to Execute
- Drivers Used for Diagnostic Purposes
  - Production Driver Used for Diagnostic Purposes
  - Separate Diagnostic Driver Used for Diagnostic Purposes
  - Diagnostic Kernel Extension Used for Diagnostic Purposes
- Acquiring a Greater Share of the Resource
- Error Log Analysis
- Enhanced Error Handling Option
- Persistent Variables
- Field Replaceable Units (FRUs)
- Specifying a Text Conclusion

- Library Restrictions for Diagnostic Programs
- Guidelines for Writing Diagnostic Programs using C++
- Completion Status for Diagnostic Applications
- Control Flow of a Diagnostic Application
- SRN Architecture
- Diagnostic Application Code Checklist

Device Configuration

In some cases, the DA will have to configure a device in order to test it. If the Configuration Method associated with the device does not contain the code that is required to load the device driver into the kernel and initialize it, then the DA will have to perform this function.

However, in most cases, the DA may use one of the diagnostic library functions provided to perform the configuration. The following library functions aid in the configuration/unconfiguration process:

- configure_device
- initial_state
- diagex_cfg_state
- diagex_initial_state

If a resource is reconfigured, then it must be restored to its initial state before the DA exits. Also, **never** assume that the parent resource(s) are always configured.

Determining the Level of Tests to Execute

Each DA is responsible for determining the level of tests that can be safely executed. This determination is a function of how the underlying device drivers support access to the device.

For nonshared, nonmultiplexed devices, the DA should attempt to **open()** the device with read/write privileges and thus determine its access privileges. For shared or multiplexed devices, a more complicated strategy needs to be developed. Perhaps the simplest method - at least from an application standpoint - is to add support for an **openx()** system call to the device driver, where the *ext* parameter distinguishes between port-level and card-level diagnostics.

Drivers Used for Diagnostic Purposes

There are different scenarios for configuring a resource to test. Depending on the relationship the resource to be tested has with other resources, it may be desirable to use one method over another. For instance, to unconfigure a resource in order to load a separate diagnostic driver or kernel extension, it will also be necessary to unconfigure all of the children resources connected to the particular resource, if any. This could cause a problem if the child resources are in use. In this case, it is desirable to use the production driver for diagnostic purposes. In all cases, it is important to restore the resource (and child resources) to their original state after testing.

Production Driver Used for Diagnostic Purposes

If the resource is in the **DEFINED** state, the resource must be configured before testing. After the resource is configured, tests can be performed on the resource, and then the resource must be put back into its original state.

Separate Diagnostic Driver Used for Diagnostic Purposes

If the resource is in the **DEFINED** state, the diagnostic driver may be loaded for testing, then unloaded after testing. If the resource is in the **AVAILABLE** state because the production driver is loaded, it is necessary to unload the production driver, load the diagnostic driver, perform the tests, unload the diagnostic driver, and then reload the production driver. Any child resources must be unconfigured before the resource under test can be unconfigured.

Diagnostic Kernel Extension Used for Diagnostic Purposes

If the resource is in the **DEFINED** state, the resource must be put into the **DIAGNOSE** state for testing. If the resource is in the **AVAILABLE** state because the production driver is loaded, it is necessary to unconfigure the resource and all its children, reconfigure the resource into the **DIAGNOSE** state, test it, and then reconfigure the resource and all its children back to their original states.

Acquiring a Greater Share of the Resource

If further testing is required, then the DA should assist the user in determining if the user should proceed with the testing.

For some devices, it may be best to ask the user to switch to another window and vary the device offline before continuing. For others, it may be best to send software-terminate signals. And for still others, it may be best to start the commands that have been specifically provided to gracefully degrade the system.

Error Log Analysis

If the *dmode* field in the **TMInput Test Mode Input** object class is set to either **DMODE_ELA** or **DMODE_PD**, then Error Log Analysis should be performed. Error log analysis should be considered a shared test.

The getdainput subroutine is used to get the test mode input parameters.

resource_alias Attribute

When a DA needs to analyze error logs from multiple resources, like the base system DA and system planar, memory and l2 cache resources, or a DA wants to analyze error logs that are logged against hardware events, like machine checks or environmental and power warnings (EPOW), then a PDiagAtt stanza must be used to define the alias between the device under test and the additional resources.

For example, the DA for the system planar on the RSPC platform performs error log analysis for machine checks that are logged by the RSPC Machine Check Error Handler. The following PDiagAtt stanza must be used to define the alias between the resource, sysplanar0, and the machine check event, MACHCHECK.

```
PDiagAtt:
  DClass = "planar"
  DSClass = "sys"
  DType = "sysplanar_rspc"
  attribute = "resource_alias"
  value = "MACHCHECK"
  rep = "n"
  DApp = ""
```

Thus any error logged against ″MACHCHECK″ will be analyzed by the DA for the resource of the class, subclass and type of ″planar/sys/sysplanar_rspc″, which is typically ″sysplanar0″. Any repair action done for the resource (sysplanar0) will be associated with the error logged against ″MACHCHECK″.

Another example: The Diagnostic Application for the base system on the CHRP platform performs error log analysis for the firmware generated error logs for the system planar, memory and l2 cache resources. The following stanzas are used to invoke error log analysis from Problem Determination mode and to record the repair action in the error log after the system verification procedure.

```
PDiagAtt:
   DClass = "planar"
   DSClass = "sys"
   DType = "sysplanar_rspc"
   attribute = "resource_alias"
   value = "mem0"
   rep = "n"
   DApp = ""

PDiagAtt:
```

```
DClass = "planar"
DSClass = "sys"
DType = "sysplanar_rspc"
attribute = "resource_alias"
value = "l2cache0"
rep = "n"
DApp = ""
```

Enhanced Error Handling (EEH) Option

The Diagnostics Application interface includes the pdiag_set_eeh_option, pdiag_set_slot_reset, pdiag_eeh_errinjct, and pdiag_read_slot_reset subroutines. These subroutines provide the DA with the necessary tools for adequate testing on the EEH option. The DA Support for this feature requires that the DA perform the following sequence of instructions in order:

1. Open I/O Adapter Test Units (TU_OPEN).
2. Call pdiag_read_slot_reset.
   Verify that the EEH option is supported.
3. Call pdiag_set_eeh_option.
   Enable EEH option for test, and set error injection option.
4. Verify I/O Adapter and setup Test Unit for test.
   Execute the required Test Units to provide the proper environment for a read and write operation and to verify that the adapter is responding. (A read/write environment is adapter dependent, so this step may not be required for some applications.)
5. Call pdiag_eeh_errinjct.
   Open Inject error condition.
6. Execute I/O Adapter Test Unit; only one Test Unit is required.
   The required Test Unit must implement a write and a read to the I/O Adapter under test.
7. Check Test Unit return code.
   For a successful test result, the I/O Adapter Test Unit will report a unique return code. If the I/O Adapter Test Units execute successfully, the EEH option has failed the test. This unique return code is specific to a Test Units suite and Diagnostics Application. The return code is mutually agreed upon by the Test Unit and Diagnostics Application developers, and is documented in the Test Units Specification. When this test fails the Diagnostics Application will report an SRN.
8. Call pdiag_eeh_errinjct.
   Close Inject error condition.
9. Call pdiag_set_slot_reset.
   Set the PCI slot to reset state (reset active) for the I/O Adapter being tested.
10. Call pdiag_read_slot_reset.
    Verify that the selected PCI slot has its reset mode deactivated.
11. Execute full suite of Test Units (normal Test Units execution for affected component).
    If an EEH error is reported, retry the Diagnostics Application as follows:
    1. Reset the PCI slot.
    2. Verify the PCI slot state.
    3. Retry Test Units.
12. Disable EEH option (restore to original state).
13. Close I/O Adapter Test Units (TU_CLOSE).

Persistent Variables

DAs must store state variables in the **DAVars Diagnostic Application Variables** object class to support loop mode. DAs are executed for each pass of loop mode, and thus lose state.

The putdavar and getdavar subroutines are used to put or get persistent variables.

Field Replaceable Units (FRUs)

DAs report FRU Buckets to identify parts that need to be replaced. The addfrub subroutine is used to add a FRU bucket to the **FRU Bucket** object class in the configuration database.

As part of the FRU information, a FRU part number for a fru not in the ODM database can be returned by the DA. The FRU part number will be placed in the **DAVars** object class. Also, if the FRU bucket contains a sub-FRU (i.e., memory module, daughter cards, etc.), the DA must return its physical or logical location code as part of the FRU bucket.

Each DA should base its good or bad status on the status of its children. A resource may pass its tests and be labeled bad when it has multiple children that have been labeled bad.

If a problem is detected with resource *x*, which has a parent called resource *y* and a sibling called resource *z*, then two FRU Buckets should be output.

- FRU Bucket 1 should identify the resources *x* and *y*, and any cables that can be identified. If the cables cannot be uniquely identified, then the Service Repair Action should implicitly include any cables that may be needed.
- FRU Bucket 2 should only identify resource *x* and any cables if possible.

The Diagnostic Controller decides which FRU Bucket to use, based on the good/bad status of the sibling. If the sibling passes its tests, then FRU Bucket 2 is named.

Specifying a Text Conclusion

DAs can also specify a menu as a conclusion. A menu should be specified if the repair action can be performed by the customer. For example, if the problem can be solved by formatting a hard disk, then a menu should be specified.

The **menugoal** subroutine performs this function by adding the menu goal to the **Menugoal object class**.

Library Restrictions for Diagnostic Programs

Library **libc.a.min** is the libc included in the standalone diagnostic package. Do not use any function that is not part of **libc.a.min** in your application. If a function is used in a diagnostic program that is not an exported symbol of **libc.a.min**, then an immediate software error (803-xxx) will occur when attempting to run the diagnostic program in standalone diagnostic mode.

To ensure that all symbols used by your diagnostics application are included in the standalone environment, compile and link the application code with the **libc.a.min** library found in the **/usr/ccs/lib** directory.

One method is to create a directory containing the libraries needed for linking:

1. Copy libraries **libodm.a**, **libcfg.a**, and **libcrypt.a** to the new directory.
2. Make a link from **/usr/ccs/lib/libc.a.min** to **libc.a** in the new directory.
3. Make a link from **/usr/ccs/lib/libc.a.min** to **libbind.a** in the new directory.
4. Export **LIBPATH** to the new directory.
5. Compile and Link your application.

You can ignore any unresolved symbols coming from libasl, or others that you know about.

Errors found indicating unresolved symbols must be fixed before the program will properly execute in standalone diagnostics mode.

Guidelines for Writing Diagnostic Programs using C++

1. The standard library libC.a is not supported. Do not use this library's API.
2. All of the language support functions in libC.a need to be statically linked at compile time. Use `-lCns.a` and `-bI:/usr/lpp/xlC/lib/libC.imp` arguments to compile with xlC.
3. Use an exception only for exceptional cases. For example, an exception should not be used for a program's normal flow of control.
4. Never throw an exception across a shared library and executable boundaries.
5. No kernel extension shall be written in C++.

Completion Status for Diagnostic Applications

DAs must issue the macro **DA_EXIT**() to exit.

Individual values can be set by calling the appropriate **DA_SETRC_XXXXXX()** macro definition.

The following values are defined:

DA_STATUS_GOOD	No problems were found.
DA_STATUS_BAD	A FRU Bucket or a Menu Goal was reported.
DA_USER_NOKEY	No special function keys were entered.
DA_USER_EXIT	The Exit key was entered by the user.
DA_USER_QUIT	The Cancel key was entered by the user.
DA_ERROR_NONE	No errors were encountered performing a normal operation such as displaying a menu, accessing the object repository, and allocating memory.
DA_ERROR_OPEN	Could not open the device.
DA_ERROR_OTHER	An error was encountered performing a normal operation.
DA_TESTS_NOTEST	No tests were executed.
DA_TEST_FULL	The full tests were executed.
DA_TEST_SUB	The subtests were executed.
DA_TEST_SHR	The shared tests were executed.
DA_MORE_NOCONT	The isolation process is complete.
DA_MORE_CONT	The path to the device should be tested. The next DA to be called will be either the parent or sibling, depending on the value of *DNext* in the **Predefined Diagnostic Resources PDiagRes** object class.

Control Flow of a Diagnostic Application

The DA performs these tasks:

1. Displays first stand-by menu.
2. Obtains its input from the **TMInput** object class.
3. References the *state1* and *state2* variables in the **TMInput** object class to determine if the child devices which were tested during the current session are defective. If so, then the DA should name the parent as being bad.
4. Determines the level of tests to run.
5. Calls **TU_OPEN**.
6. Calls Application Test Units (TU).
7. Calls **TU_CLOSE**.
8. Reconfigures the device if DA caused it to be configured.
9. Performs error-log analysis if the *dmode* variable in the **TMInput** object class is equal to PD or ELA.
10. Returns status to the Diagnostic Controller through the **DA_EXIT**() macro call.

SRN Architecture

Diagnostic applications report problems through SRNs (Service Request Numbers). SRNs take the following forms:

- Six-digit SRNs consist of two grouping of three digits seperated by the character ″-″ (for example, 922-101, where the first group of three digits is referred to as the *source number*. The second group of three digits is referred to as the *reason code*. The source number is a unique number that identifies the diagnostic application that produced the SRN. The source number is usually synonymous with the LED field of the PdDV object class of the configuration database. For a diagnostic applications that can not use the LED value, for whatever reason, a value must be assigned to avoid duplication. The reason code can be used to identify a particular failure cause detected by the diagnostic application.
- Other SRN Types. See the **addfrub** subroutine for details.

Six-digit SRNs should be grouped so that each set of FRU callouts are grouped together. For example, if a Diagnostic Application callout consists of:

- 10 SRNs for FRU A
- 20 SRNs for FRU B
- 5 SRNs for FRU A most likely with FRU B next
- 6 SRNs for FRU B most likely with FRU A next

Then the SRNs should be grouped like the following:

- 921-111 to 921-120 FRU A
- 921-131 to 921-150 FRU B
- 921-211 to 921-215 FRU A FRU B
- 921-221 to 921-226 FRU B FRU A

The guidelines for the Reason Codes for SRN Source Numbers 700 to 799 and 811 to 999 that are not decoded from some type of special information are:

000 Reserved

001 Indicates that an adapter or device could not be found

002 to 100 Reserved

101 to 199 Reserved for non-ELA callouts with a single FRU

200 to 299 Reserved for non-ELA callouts with two FRUs

300 to 399 Reserved for non-ELA callouts with three FRUs

400 to 499 Reserved for non-ELA callouts with four or more FRUs

500 to 599 Reserved for non-ELA cases that require a special action such as waiting for a thermal device to cool or checking the level of a device.

600 to 699 Reserved for ELA callouts with a single FRU

700 to 799 Reserved for ELA callouts with two or more FRUs

800 to 899 Reserved for ELA cases that require a special action, such as waiting for a thermal device to cool or checking the level of a device.

900 to 999 Reserved

This is done to group the SRNs with like FRUs into one entry in the SRN Tables.

Diagnostic Controller Generated SRNs

The following table lists SRN generated by the diagnostic controller when the event shown in the description column occurs.

> **Note:** ″xxx″ in the following table represents the source number of the diagnostic application that executed.

SRN	Description
802-xxx	The diagnostic did not detect an installed device (Online Diagnostics).
803-xxx	An error not related to the diagnostic tests occurred.
804-xxx	A halt occurred in the diagnostic application.
801-101 801-102	The diagnostics did not detect an installed device (Standalone Diagnostics).

Source Numbers

The following source numbers are defined for use by third party vendors.

> **Note:** If the LED field of the **PdDV** object class for a particular device is different than the source number shown in the table below, the LED takes precedence. Source Numbers shown in the following table are hexadecimal values.

Source Number	Description
661	IDE Tape Drive
66a	USB Open Host Controller Type
66b	USB Universal Host Controller Type
74b	ATM Adapter
74d	Sound Card
74e	Fibre Channel Adapter
892	Graphics Display Adapter
893	Local Area Network (LAN) Adapter
894	Async Protocol Adapter
901	SCSI Protocol Device
902	Graphics Display
904	Parallel Port Attached Device
753	IDE CD ROM Drive
891	SCSI Device Adapter
752	IDE Disk Drive
805	CD Read/Write Drive
711	Generic Adapter (Not covered above)

Diagnostic Application Code Checklist

The following checklist can be helpful in ensuring successful Diagnostic Application (DA) code.

1. Code must execute Good Machine Path (GMP) testing without abending or returning an SRN under the following conditions:
   a. IPL Mode: Service from hard disk.
   b. Select Advanced mode.
   c. Select PD mode.
   d. Run a single time.

Follow all instructions presented by the DA. If the question presented on a screen is unclear, note the ambiguity and answer the question as you understand it.

Use wrap plugs where required. Unplug cables as required.

Look for:

a. Spelling errors
b. Grammatical errors

2. Code must execute GMP testing without abending or returning an SRN under the following conditions:
   a. IPL Mode: Service from CD-ROM.
   b. Select Advanced mode.
   c. Select PD mode.
   d. Run a single time.

   Use wrap plugs where required. Unplug cables as required.
3. Code must execute GMP testing without abending or returning an SRN under the following conditions:
   a. IPL Mode: Normal.
   b. Run diagnostics from command line in no-console mode.
      `diag -cd` *device*
   c. Run diagnostics from command line in no-console Advanced mode.
      `diag -Acd` *device*
4. Code must execute Good Machine Path (GMP) testing without abending or returning an SRN under the following conditions:
   a. IPL Mode: Service from hard disk.
   b. Select PD mode.
   c. Select Advanced mode.
   d. Select ALL Resources.

   Follow all instructions presented by the DA. If the question presented on a screen is unclear, note the ambiguity and answer the question as you understand it.

   Look for: No interactive menus displayed while the application is executing.

Other test scenarios include:

1. Bring the device to the DEFINED State; then run diagnostics to ensure the DA causes the device to be made available. After testing is completed, ensure adapter is placed back in the DEFINED State.
2. If microcode is used by the device, rename the microcode file, run the DA, and make sure the DA reports the absence of the file.
3. Run Advanced Diagnostics on the device. When a wrap plug is called for, do not use it. Make sure an SRN is generated. Alternatively, do anything that causes an SRN to be reported. Check the SRN for accuracy.
4. Try to cause an open error by renaming device driver. Ensure that a software error is reported.
5. Place the adapter in the DEFINED state. Cause the configuration to fail by renaming the method. Observe how the DA handles this. In most instances, an SRN should be generated stating that the device could not be configured.
6. Place the adapter in the second I/O planar of a supported system. Ensure the adapter is in the DEFINED state. Run diagnostics to ensure the DA causes the device to be made available. After testing is completed, ensure adapter is placed back in the DEFINED state.

Tasks and Service Aids

The Diagnostic Package contains programs that are called Tasks. Tasks can be thought of as ″performing a specific function on a resource″; for example, running diagnostics, or performing a Service Aid on a resource.

Creating a Task

Note: The diagnostic subsystem only supports 32-bit Tasks and Service Aids.

Tasks are represented by an entry in the Predefined Diagnostic Task object class (PDiagTask). To create a new task, a **PDiagTask** object is needed plus the binary executable of the task itself, as specified by the **PDiagTask->***Action* class member.

Some Task IDs are reserved for use by the Diagnostic Controller:

`Task ID 0`
: Built-in Controller Task

`Task ID 1000+`
: Reserved for Third-Party Use. Any number may be used above 999. A clash of task IDs by third-party tasks may occur if the same task ID is used. The problem may appear to the user as seeing a particular resource supported by a task, when in reality it is not. Each third-party supported task should be able to handle the condition of a nonsupported resource given as a command-line argument, if the **PDiagTask->***ResourceFlag* is set.

Performing a Task

Menu

Select the following from the Function Selection Menu:

```
Task Selection (Diagnostics, Advanced Diagnostics, Service Aids, etc.)
This selection will list the tasks supported by these procedures.
Once a task is selected, a resource menu may be presented showing
all resources supported by the task.
```

The displaying of the resource menu is dependent on the value of the **PDiagTask->***ResourceFlag* value.

Note: Many of these tasks work on all system model architectures. (The Diagnostic Task Matrix shows all current supported tasks and their supported platforms.) Some tasks are only accessible from Online Diagnostics in Service or Concurrent mode, others may be accessible only from Standalone Diagnostics. While still other tasks may only be supported on a particular system architecture, such as CHRP (Common Hardware Reference Platform), or RSPC (PowerPC Reference Platform).

Fastpath with Unknown Resource

A fastpath method is also available to perform a task by using the `-T` flag with the diag command. This means that the user does not have to go through most of the introductory menus just to get to a particular task. Instead the user is presented with a list of resources available that support the task specified.

The current fastpath tasks are:

format	Format Media
certify	Certify Media
download	Download Microcode
disp_mcode	Display Microcode Level
chkspares	Spare Sector Availability
identify	PCI RAID Physical Disk Identify

Fastpath with Known Resource

Each of these tasks can also be invoked directly from the command line specifying the resource and other task unique flags. This implies that the user already knows the resource to perform the task operation on. See publications *Diagnostic Information for Multiple Bus Systems* or *Diagnostic Information for Micro Channel Bus Systems* for more specific information on the tasks and flags.

Task List

The following is a list of all known supported tasks on the latest level of diagnostics. Tasks have been separated into one of six groups.

- Run Diagnostics
- Run Error Log Analysis
- Run Exercisers
- Display or Change Diagnostic Run Time Options
- 7135 RAIDiant Array Service Aid
- Add or Delete Drawer Configuration
- Add Resource to Resource List
- AIX Shell Prompt
- Analyze Adapter Internal Log
- Backup and Restore Media
- Certify Media
- Change Hardware Vital Product Data
- Configure Dials and LPFKeys
- Configure ISA Adapter
- Configure Reboot Policy (CHRP)
- Configure Remote Maintenance Policy (CHRP)
- Configure Ring Indicate Power On Policy (CHRP)
- Configure Ring Indicate Power On (RSPC)
- Configure Service Processor (RSPC)
  - Call In/Out Setup
  - Modem Configuration
  - Site Specific Call In/Out Setup
  - Surveillance Setup
- Configure Surveillance Policy (CHRP)
- Create Customized Configuration Diskette
- Delete Resource from Resource List
- Disk Maintenance
  - Disk to Disk Copy
  - Display/Alter Sector
- Display Checkstop Analysis Results
- Display Configuration and Resource List
- Display Firmware Device Node Information (CHRP)
- Display Hardware Error Report
- Display Hardware Vital Product Data
- Display Machine Check Error Log
- Display Microcode Level
- Display Previous Diagnostic Results

- Display Resource Attributes
- Display Service Hints
- Display Software Product Data
- Display System Environmental Sensors (CHRP)
- Display or Change Bootlist
- Display or Change BUMP Configuration
- Display or Change Electronic Mode Switch
- Display or Change Multiprocessor Configuration
- Display Test Patterns
- Download Microcode
- ESCON Bit Error Rate Service Aid
- Fibre Channel RAID Service Aids
- Flash SK-NET FDDI Firmware
- Format Media
- Generic Microcode Download
- Local Area Network Analyzer
- PCI RAID Physical Disk Identify
- Periodic Diagnostics
- Process Supplemental Media
- Save or Restore Hardware Management Policies (CHRP)
- Save or Restore Service Processor Configuration (RSPC)
- SCSD Tape Drive Service Aid
- SCSI Bus Analyzer
- SCSI Device Identification and Removal
- Service Aids for use with Ethernet
- Spare Sector Availability
- SSA Service Aids
- Update Disk Based Diagnostics
- Update System Flash (RSPC)
- Update System or Service Processor Flash (CHRP)

Add or Delete Drawer Configuration

Note: Not applicable to RSPC or CHRP systems, or to the Itanium-based platform.

This task invokes SMIT to provide the following options:
- List all Drawers
- Add a Drawer
- Remove a Drawer

The supported drawer types are:
- Media SCSI Device Drawer
- DASD SCSI DASD Drawer

Add Resource to Resource List

Use this task to add resources back to the resource list.

Note: Only resources that were previously detected by the diagnostics and deleted from the Diagnostic Test List are listed. If no resources are available to be added, then none are listed.

Shell Prompt

Note: Online Service Mode only.

This Service Aid allows access to the command line. To use this Service Aid the user must know the root password (when a root password has been established).

Do not use this task to install code, or change the configuration of the system. It is intended to be used to look at files, configuration, data, etc. Changing the system configuration, or installing code may produce problems after exiting back to the Diagnostic Controller.

Analyze Adapter Internal Log (Device Specific)

The PCI RAID adapter has an internal log that logs information about the adapter and the disk drives attached to the adapter. Whenever data is logged in the internal log, the device driver copies the entries to the system error log and clears the internal log.

The Analyze Adapter Internal Log Service Aid analyzes these entries in the system error log. The Service Aid displays the errors and the associated service actions. Entries that do not require any service actions are ignored.

Backup and Restore Media

This Service Aid allows verification of backup media and devices. It presents a menu of tape and diskette devices available for testing and prompts for selection of the desired device. It then presents a menu of available backup formats and prompts for selection of the desired format. The supported formats are tar, backup, and cpio. After the device and format are selected, the Service Aid backups a known file to the selected device, restores that file to /tmp, and compares the original file to the restored file. The restored file is also left in /tmp to allow for visual comparison. All errors are reported.

Certify Media

This task allows the selection of diskette or hardfiles to be certified. Hardfiles can be connected either to a SCSI adapter(non RAID) or a PCI SCSI RAID adapter. The usage and certify criteria for a hardfile connected to a non RAID SCSI adapter are different from those for a hardfile connected to a PCI SCSI RAID adapter.

Note: The certify function for devices attached to a PCI SCSI RAID adapter is supported for certain PCI SCSI RAID adapters only.

This task may be run directly from the command line. The following usage statement describes the syntax of the fastpath command:

Usage:
```
diag -T "certify"
```

Change Hardware Vital Product Data

Use this Service Aid to display the Display/Alter VPD Selection Menu. The menu lists all resources installed on the system. When a resource is selected, a menu displays all the VPD that are recognized by the operating system for that resource.

Note: The user cannot alter the VPD for a specific resource unless it is not machine readable.

Configure Dials and LPFKeys

This Service Aid provides a tool for configuring and removing dials/LPFKs to the asynchronous serial ports.

Since version 4.1.3 a tty must be defined on the async port before the Dials and LPFKs can be configured on the port. Before version 4.2 the Dials and LPFKs could only be configured on the standard serial ports. At version 4.2 the Dials and LPFKs can be configured on any async port.

This selection invokes the SMIT utility to allow Dials and LPFKs configuration. A tty must be in the available state on the async port before the Dials and LPFKs can be configured on the port. The task allows an async adapter to be configured, then a tty port defined on the adapter, and then Dials and LPFKs can be defined on the port.

Configure ISA Adapter

This task invokes SMIT to allow the identification and configuration of ISA adapters on systems that have an ISA bus and adapters.

Diagnostic support for ISA adapters not shown in the list may be supported from a Supplemental Diskette. ISA adapter support can be added from a Supplemental Diskette with the Process Supplemental Media task.

Whenever an ISA adapter is installed, this Service Aid must be run and the adapter configured before the adapter can be tested. This Service Aid must also be run (and the adapter removed) whenever an ISA adapter is physically removed from the system.

If diagnostics are run on an ISA adapter that has been removed from the system, the diagnostics fail.

ISA adapters cannot be detected by the system.

> **Note:** When using this Service Aid choose the option that places the adapter in the ″Defined State″. Do not select the option that places the device in the ″Available State″.

Configure Reboot Policy (CHRP)

> **Note:** Runs on CHRP systems units only.

This Service Aid controls how the system tries to recover from a system crash.

Use this Service Aid to display and change the following settings for the Reboot Policy.

> **Note:** Because of system capability, some of the following settings may not be displayed by this Service Aid.

- **Maximum Number of Reboot Attempts**

  Enter a number that is 0 or greater.

  > **Note:** A value of 0 indicates ’do not attempt to reboot’ to a crashed system.

  This number is the maximum number of consecutive attempts to reboot the system. The term ″reboot″, in the context of this Service Aid, is used to describe bringing system hardware back up from scratch, for example from a system reset or power on.

  When the reboot process completes successfully, the reboot attempts count is reset to 0, and a ″restart″ begins. The term ″restart″, in the context of this Service Aid, is used to describe the operating system activation process. Restart always follows a successful reboot.

When a restart fails, and a restart policy is enabled, the system attempts to reboot for the maximum number of attempts.

- **Use the O/S Defined Restart Policy (1=Yes, 0=No)**

  When 'Use the O/S Defined Restart Policy' is set to Yes, the system attempts to reboot from a crash if the operating system has an enabled Defined Restart or Reboot Policy.

  When 'Use the O/S Defined Restart Policy' is set to No, or the operating system restart policy is undefined, then the restart policy is determined by the 'Supplemental Restart Policy'.

- **Enable Supplemental Restart Policy (1=Yes, 0=No)**

  The 'Supplemental Restart Policy', if enabled, is used when the O/S Defined Restart Policy is undefined, or is set to False.

  When surveillance detects operating system inactivity during restart, an enabled 'Supplemental Restart Policy' causes a system reset and the reboot process begins.

- **Call-Out Before Restart (on/off)**

  When enabled, Call-Out Before Restart allows the system to call out (on a serial port that is enabled for call out) when an operating system restart is initiated. Such calls can be valuable if the number of these events becomes excessive, thus signaling bigger problems.

- **Enable Unattended Start Mode (1=Yes, 0=No)**

  When enabled, 'Unattended Start Mode' allows the system to recover from the loss of AC power.

  If the system was powered-on when the AC loss occurred, the system reboots when power is restored. If the system was powered-off when the AC loss occurred, the system remains off when power is restored.

This Service Aid may be accessed directly from the command line, by entering:

```
/usr/lpp/diagnostics/bin/uspchrp -b
```

Configure Remote Maintenance Policy (CHRP)

Note: Runs on CHRP systems units only.

The Remote Maintenance Policy includes modem configurations and phone numbers to use for remote maintenance support.

Use this Service Aid to display and change the following settings for the Remote Maintenance Policy.

Note: Because of system capability, some of the following settings may not be displayed by this Service Aid.

- **Configuration File for Modem on S1**
  Configuration File for Modem on S2

  Enter the name of a modem configuration file to load on either serial port 1 (S1) or serial port 2 (S2). The modem configuration files are located in the directory **/usr/share/modems**. If a modem file is already loaded, it is showed by `Modem file currently loaded`.

- **Modem file currently loaded on S1**
  Modem file currently loaded on S2

  This is the name of the file that is currently loaded on serial port 1 or serial port 2.

  Note: These settings are only shown when a modem file is loaded for a serial port.

- **Call In Authorized on S1 (on/off)**
  Call In Authorized on S2 (on/off)

  Call In allows the Service Processor to receive a call from a remote terminal.

- **Call Out Authorized on S1 (on/off)**
  Call Out Authorized on S2 (on/off)

  Call Out allows the Service Processor to place calls for maintenance.

- **S1 Line Speed**
  S2 Line Speed

  A list of line speeds is available by using 'List' on the screen.
- **Service Center Phone Number**

  This is the number of the service center computer. The service center usually includes a computer that takes calls from systems with call-out capability. This computer is referred to as "the catcher". The catcher expects messages in a specific format to which the Service Processor conforms. For more information about the format and catcher computers, refer to the README file in the **/usr/samples/syscatch** directory. Contact the service provider for the correct telephone number to enter here.
- **Customer Administration Center Phone Number**

  This is the number of the System Administration Center computer (catcher) that receives problem calls from systems. Contact the system administrator for the correct telephone number to enter here.
- **Digital Pager Phone Number In Event of Emergency**

  This is the number for a pager carried by someone who responds to problem calls from your system.
- **Customer Voice Phone Number**

  This is the number for a telephone near the system, or answered by someone responsible for the system. This is the telephone number left on the pager for callback.
- **Customer System Phone Number**

  This is the number to which your system's modem is connected. The service or administration center representatives need this number to make direct contact with your system for problem investigation. This is also referred to as the Call In phone number.
- **Customer Account Number**

  This number could be used by a service provider for record keeping and billing.
- **Call Out Policy Numbers to call if failure**

  This is set to either 'first' or 'all'. If the call out policy is set to 'first', call out stops at the first successful call to one of the following numbers in the order listed:

  1. Service Center
  2. Customer Admin Center
  3. Pager

  If Call Out Policy is set to 'all', call out attempts to call all of the following numbers in the order listed:

  1. Service Center
  2. Customer Admin Center
  3. Pager
- **Customer RETAIN Login ID**
  Customer RETAIN Login Password

  These settings apply to the RETAIN service function.
- **Remote Timeout, in seconds**
  Remote Latency, in seconds

  These settings are functions of the service provider's catcher computer.
- **Number of Retries While Busy**

  This is the number of times the system should retry calls that resulted in busy signals.
- **System Name (System Administrator Aid)**

  This is the name given to the system and is used when reporting problem messages.

  > **Note:** Knowing the system name aids the support team to quickly identify the location, configuration, history, etc. of your system.

This Service Aid may be accessed directly from the command line, by entering:

```
/usr/lpp/diagnostics/bin/uspchrp -m
```

Configure Ring Indicate Power On (RSPC)

Note: Runs on RSPC systems units only.

This Service Aid allows the user to display and change the NVRAM settings for the Ring Indicate Power On capability of the service processor.

The settings allows the user to:

- Enable/Disable power on from Ring Indicate
- Read/Set the number of rings before power on

Configure Ring Indicate Power On Policy (CHRP)

Note: Runs on CHRP systems units only.

This Service Aid allows the user to power on a system by telephone from a remote location. If the system is powered off, and Ring Indicate Power On is enabled, the system powers on at a predetermined number of rings. If the system is already on, no action is taken. In either case, the telephone call is not answered and the caller receives no feedback that the system has powered on.

Use this Service Aid to display and change the following settings for the Ring Indicate Power On Policy.

Note: Because of system capability, some of the following settings may not be displayed by this Service Aid.

- Power On Via Ring Indicate (on/off)
- Number of Rings Before Power On

This Service Aid may be accessed directly from the command line, by entering:

```
/usr/lpp/diagnostics/bin/uspchrp -r
```

Configure Service Processor (RSPC)

Note: Runs on RSPC systems units only.

This Service Aid allows you to display and change the NVRAM settings for the service processor.

This Service Aid supports the following functions:

- Surveillance Setup
- Modem Configuration
- Call In/Call Out Setup
- Site Specific Call In/Call Out Setup

Surveillance Setup

This selection allows you to display and change the NVRAM settings for the surveillance capability of the service processor.

The settings allow you to:

- Enable/disable surveillance
- Set the surveillance time interval, in minutes
- Set the surveillance delay, in minutes

The current settings are read from NVRAM and displayed on the screen. Any changes made to the data shown are written to NVRAM.

Modem Configuration

Use this selection when setting the NVRAM for a modem attached to any of the Service Processor's serial ports. The user inputs the file name of a modem configuration file and the serial port number. The formatted modem configuration file is read, converted for NVRAM than loaded into NVRAM. Refer to the ″Service Processor Installation and User's Guide″ for more information.

Call In/Out Setup

This selection allows the user to display and change the NVRAM settings for the Call In/Call Out capability of the service processor.

The settings allows the user to:

- Enable/Disable call in on either serial port.
- Enable/Disable call out on either serial port.
- Set the line speed on either serial port.

Site Specific Call In/Out Setup

This selection allows you to display and change the NVRAM settings that are site specific for the call in/call out capability of the service processor.

The site specific NVRAM settings allow you to:

- Set the phone number for the service center
- Set the phone number for the customer administration center
- Set the phone number for a digital pager
- Set the phone number for the customer system to call in
- Set the phone number for the customer voice phone
- Set the customer account number
- Set the call out policy
- Set the customer RETAIN id
- Set the customer RETAIN password
- Set the remote timeout value
- Set the remote latency value
- Set the number of retries while busy
- Set the system name

The current settings are read from NVRAM and displayed on the screen. Any changes made to the data shown are written to NVRAM.

Configure Surveillance Policy (CHRP)

Note: Runs on CHRP systems units only.

This Service Aid monitors the system for hang conditions, that is, hardware or software failures that cause operating system inactivity. When enabled, and surveillance detects operating system inactivity, a call is placed to report the failure.

Use this Service Aid to display and change the following settings for the Surveillance Policy.

Note: Because of system capability, some of the following settings may not be displayed by this Service Aid.

- **Surveillance (on/off)**
- **Surveillance Time Interval**

  This is the maximum time between heartbeats from the operating system.
- **Surveillance Time Delay**

  This is the time to delay between when the operating system is in control and when to begin operating system surveillance.
- **Changes are to take affect immediately**

  Set this to Yes if the changes made to the settings in this menu are to take place immediately. Otherwise the changes takes place beginning with the next system boot.

This Service Aid may be accessed directly from the command line, by entering:

```
/usr/lpp/diagnostics/bin/uspchrp -s
```

Create Customized Configuration Diskette

This selection invokes the Diagnostic Package Utility Service Aid which allows the user to Create a Standalone Diagnostic Package Configuration Diskette

The Standalone Diagnostic Package Configuration Diskette allows the following to be changed when running diagnostics from removable media:

- **High-Function Terminals 60/77-Mhz Refresh Rate**

  The refresh rate used by the standalone diagnostic package is 60Hz. If the display's refresh rate is 77Hz, then set the refresh rate to 77.
- **Different async terminal console**

  A console configuration file that allows a terminal attached to any RS232 or RS422 adapter to be selected as a console device can be created using this Service Aid. The default device is a RS232 tty attached to the first standard serial port (S1).

Delete Resource from Resource List

Use this task to delete resources from the resource list.

Note: Only resources that were previously detected by the diagnostics and have not been deleted from the Diagnostic Test List are listed. If no resources are available to be deleted, then none are listed.

Disk Maintenance (SCSI Disks)

- Disk to Disk Copy
- Display/Alter Sector

Disk to Disk Copy

This selection allows you to recover data from an old drive when replacing it with a new drive. The Service Aid only supports copying from a drive to another drive of similar size. This Service Aid cannot be used to update to a different size drive. The **migratepv** command should be used when updating drives. The Service Aid recovers all LVM software reassigned blocks. To prevent corrupted data from being copied to the new drive, the Service Aid aborts if an unrecoverable read error is detected. To help prevent possible problems with the new drive, the Service Aid aborts if the number of bad blocks being reassigned reaches a threshold.

Note: Use the **migratepv** command when copying the contents to other disk drive types. This command also works when copying SCSI disk drives or when copying to a different size SCSI disk drive. Refer to ″System Management Guide: Operating System and Devices″ for a procedure on Migrating the Contents of a Physical Volume.

The procedure for using this Service Aid requires that both the old and new disks be installed in or attached to the system with unique SCSI addresses. This requires that the new disk drive SCSI address must be set to an address that is not currently in use and the drive be installed in an empty location. If there are no empty locations, then one of the other drives must be removed. Once the copy is complete, only one drive may remain installed. Either remove the target drive to return to the original configuration, or perform the following procedure to complete the replacement of the old drive with the new drive.

1. Remove both drives.
2. Set the SCSI address of the new drive to the SCSI address of the old drive.
3. Install the new drive in the old drive's location.
4. Install any other drives that were removed into their original location.

To prevent problems that may occur when running this Service Aid from disk, it is suggested that this Service Aid be run from the diagnostics that are loaded from removable media when possible.

Display/Alter Sector

This selection allows the user to display and alter information on a disk sector. Care must be used when using this Service Aid because inappropriate modification to some disk sectors may result in total loss of all data on the disk. Sectors are addressed by their decimal sector number. Data is displayed both in hex and in ASCII. To prevent corrupted data from being incorrectly corrected, the Service Aid does not display information that cannot be read correctly.

Display Checkstop Analysis Results

This Service Aid analyzes checkstop files and displays the results. During a system reboot, following a checkstop, a data file is written to /usr/lib/ras that contains the state of the system at the time of the checkstop. The files have names that begin with ″checkstop″ and end with either ″.A″ or ″.B″.

The analysis of the file(s) will produce a description of the problem and provide an action plan with repair instructions or recommendations. Following the action plans, a detailed dump of the data that was saved for the checkstop is displayed.

The following options are provided:

- **Analyze Checkstop Files Created Within the Last 7 Days**

  Analyze and display the results of any checkstop file that was created in the last 7 days. This is the same file(s) that the system planar diagnostics analyzed, but will provide more detail.
- **Analyze All of the Checkstop Files**

  Analyze and display the results of all of checkstop files.

For either option, carefully read the results of the analysis and perform any recommended actions.

Display Configuration and Resource List

This Service Aid displays the item header only for all installed resources. Use this Service Aid when there is no need of seeing the VPD. (No VPD is displayed.)

Display Firmware Device Node Information (CHRP)

Note: Runs on CHRP systems units only.

This task displays the firmware device node information that appears on CHRP platforms. The format of the output data does not necessarily have to be the same between different levels of the operating system. It is intended to be used to gather more information about individual or particular devices on the system.

Display Hardware Error Report

This Service Aid provides a tool for viewing the hardware error log. It uses the **errpt** command.

The Display Error Summary and Display Error Detail selection provide the same type of report as the **errpt** command. The Display Error Analysis Summary and Display Error Analysis Detail selection provide additional analysis.

Display Hardware Vital Product Data

This Service Aid displays all installed resources along with any VPD that is recognized by the operating system for those resources. Use this Service Aid when you want to look at the VPD for a specific resource.

Display Machine Check Error Log

Note: The Machine Check Error Log Service Aid is available only on Standalone Diagnostics.

When a machine check occurs, information is collected and logged in a NVRAM error log before the system unit shuts down. This information is logged in the error log and cleared from NVRAM when the system is rebooted from either hard disk or LAN. The information is not cleared when booting from Standalone Diagnostics. When booting from Standalone Diagnostics, this Service Aid can take the logged information and turn it into a readable format that can be used to isolate the problem. When booting from the hard disk or LAN, the information can be viewed from the error log using the Hardware Error Report Service Aid. In either case the information is analyzed when running the **sysplanar0** diagnostics in Problem Determination Mode.

Display Microcode Level

This selection provides a way to display microcode on a device or adapter. Once invoked, a list of resources are available for selection that supports this function. Once a resource is selected, a specific application that supports that function on the resource is invoked. See the description on PDiagAtt for the stanza that is needed to achieve this.

This task may be run directly from the command line. The following usage statement describes the syntax of the fastpath command:

Usage:
```
diag -T ″disp_mcode″
```

Display or Change Bootlist

This Service Aid allows the bootlist to be displayed, altered, or erased.

The system attempts to perform an IPL from the first device in the list. If the device is not a valid IPL device or if the IPL fails, the system proceeds in turn to the other devices in the list to attempt an IPL.

Display or Change BUMP Configuration

This Service Aid is unique to the POWER-based SMP system units and provides the following functions:

- **Display or Change Remote Support Phone Number**

  This function allows the remote support phone number to be displayed or altered.
- **Display or Change Diagnostics Modes**

This function displays a dialog screen that lists the states of all the BUMP (Bringup Micro-Processor) Diagnostic Flags. The states can be changed via the dialog screen.

- **Save or Restore Diagnostics Modes and Remote Support Phone Number**

  This function allows the diagnostics modes and remote support phone number to be saved or restored. The location of the save area is to be defined.
- **Flash EPROM Download**

  This function updates the Flash EPROM.

Display or Change Diagnostic Run Time Options

The Display or Change Diagnostic Run Time Options task allows the diagnostic run time options to be set.

The run time options are:

- **Display Diagnostic Mode Selection Menus**

  This option allows the user to turn `on` or `off` displaying the DIAGNOSTIC MODE SELECTION MENU. The default value is `on`.
- **Include Advanced Diagnostics**

  This option allows the user to turn `on` or `off` including the Advanced Diagnostics. The default value is `off`.
- **Run Tests Multiple Times**

  This option allows the user to turn `on` or `off` running the diagnostic in Loop Mode. The default value is `off`.

  Note: This option is only displayed when running Online Diagnostics in Service Mode.
- **Include Error Log Analysis**

  This option allows the user to turn `on` or `off` including the Error Log Analysis (ELA). The default value is `off`.
- **Number of days used to search error log**

  This option allows the user to select the number of days to search the error log for errors when running Error Log Analysis. The default value is `7` days, but can be changed from `1` to `60` days.
- **Display Progress Indicators**

  This option allows the user to turn `on` or `off` the progress indicators shown when running Diagnostic Applications. The progress indicators are a popup box at the bottom of the screen indicating the test being run. The default value is `on`.
- **Diagnostic Event Logging**

  This option allows the user to turn `on` or `off` logging information to the Diagnostics Event Log. The default value is `on`.
- **Diagnostic Event Log file size**

  This option allows the user to select the maximum size of the Diagnostic Event Log. The default value is `100K`, but can be changed from `100K` to `1000K`.
- **Save changes to the database**

  This option allows the user to save any changes made to the run time options. Without saving the changes, any changes made are only applicable to that session of diagnostics. The default value is `no`.

Display or Change Electronic Mode Switch

This Service Aid is unique to the POWER-based SMP system units and displays the states of the Physical and Electronic Keys. It also allows the electronic keys to be set.

Display or Change Multiprocessor Configuration (Multiprocessor Service)

This Service Aid is unique to the POWER-based SMP system units and provides the following functions:

- **Display or Change Processor States**

  This function displays or changes the state of available processors.
- **Bind Process**

  This function provides a tool for binding a process and all its threads to a specified processor.

Display Previous Diagnostic Results

Note: This Service Aid is not available when you load the diagnostics from a source other than a disk drive or from a network.

This service aid will allow a service representative to display results from a previous diagnostic session. When the `Display Previous Results` option is selected, the user will be able to view up to 25 no trouble found (NTF) and service request number (SRN) results.

This service aid will also display diagnostic log information. The diagnostic log can be displayed in a short version or a long version. The diagnostic log contains information about events logged by a diagnostic session.

This service aid will display the information in reverse chronological order. If more information is available than what can be displayed on the screen, the Page Down and Page Up keys can be used to scroll through the information.

This information is not from the error log maintained by the operating system. This information is stored in the **/var/adm/ras directory**.

Display Resource Attributes

This task displays the Customized Device Attributes associated with a selected resource. This task is similar to running the **lsattr -E -l** ***resource*** command.

Display Service Hints

This Service Aid reads and displays the information in the **CEREADME** file from the diagnostics media. This file contains information that is not in the publications for this version of the diagnostics. It also contains information about using this particular version of diagnostics.

This Service Aid will present a menu if multiple **CEREADME** files are present in the **/usr/lpp/diagnostics/** directory. This allows other non-related **CEREADME** files to be displayed containing information about unrelated functions.

Use the arrow keys to scroll through the information in the file.

Display Software Product Data

This task invokes SMIT to display information about the installed software and provides the following functions:

- List Installed Software
- List Applied but Not Committed Software Updates
- Show Software Installation History
- Show Fix (APAR) Installation Status
- List Fileset Requisites
- List Fileset Dependents

- List Files Included in a Fileset
- List File Owner by Fileset

Display System Environmental Sensors (CHRP)

Note: Runs on CHRP systems units only.

This Service Aid displays the environmental sensors implemented on a CHRP system. The information displayed is the sensor name, physical location code, literal value of the sensor status, and the literal value of the sensor reading.

The sensor status can be any one of the following:

- **Normal**

  The sensor reading is within the normal operating range.
- **Critical High**

  The sensor reading indicates a serious problem with the device. Run diagnostics on sysplanar0 to determine what repair action is needed.
- **Critical Low**

  The sensor reading indicates a serious problem with the device. Run diagnostics on sysplanar0 to determine what repair action is needed.
- **Warning High**

  The sensor reading indicates a problem with the device. This could become a critical problem if action is not taken. Run diagnostics on sysplanar0 to determine what repair action is needed.
- **Warning Low**

  The sensor reading indicates a problem with the device. This could become a critical problem if action is not taken. Run diagnostics on sysplanar0 to determine what repair action is needed.
- **Hardware Error**

  The sensor could not be read because of a hardware error. Run diagnostics on sysplanar0 in problem determination mode to determine what repair action is needed.
- **Hardware Busy**

  The system has repeatedly returned a busy indication, and a reading is not available. Try the Service Aid again. If the problem continues, run diagnostics, on sysplanar0 in problem determination mode to determine what repair action is needed.

This Service Aid can also be run as a command. The command can be used to list the sensors and their values in a text format, list the sensors and their values in numerical format, or a specific sensor can be queried to return either the sensor status or sensor value.

The command can be run by entering one of the following:

```
/usr/lpp/diagnostics/bin/uesensor -l | -a
/usr/lpp/diagnostics/bin/uesensor -t token -i index [-v]
```

Flags

-l
List the sensors and their values in a text format.

-a
List the sensors and their values in a numerical format. For each sensor, the following numerical values are displayed as:

```
<token> <index> <status> <measured value> <location code>
```

-t *token*
Specifies the sensor token to query.

-i *index*
Specifies the sensor index to query.

-v
Indicates to return the sensor measured value. The sensor status is returned by default.

Examples

1. Display a list of the environmental sensors:

```
/usr/lpp/diagnostics/bin/uesensor -l
```

   Sensor Token = Fan Speed
   Status = Normal
   Value = 2436 RPM
   Location Code = F1

   Sensor Token = Power Supply
   Status = Normal
   Value = Present and operational
   Location Code = V1

   Sensor Token = Power Supply
   *Status = Critical low
   Value = Present and not operational
   Location Code = V2

2. Display a list of the environmental sensors in a numerical list:

```
/usr/lpp/diagnostics/bin/uesensor -a
```

   3 0 11 87 P1
   9001 0 11 2345 F1
   9004 0 11 2 V1
   9004 1 9 2 V2

3. Return the status of sensor 9004, index 1:

```
/usr/lpp/diagnostics/bin/uesensor -t 9004 -i 1

9
```

4. Return the value of sensor 9004, index 1:

```
/usr/lpp/diagnostics/bin/uesensor -t 9004 -i 1 -v

2
```

Display Test Patterns

This Service Aid provides a means of adjusting system display units by providing displayable test patterns. Through a series of menus the user selects the display type and test pattern. After the selections are made the test pattern is displayed.

Download Microcode

This selection provides a way to update microcode to a device or adapter. Once invoked, a list of resources are available for selection that supports this function. Once a resource is selected, a specific application that supports that function on the resource is invoked. See the description on PDiagAtt for the stanza that is needed to achieve this.

This task may be run directly from the command line. The following usage statement describes the syntax of the fastpath command:

Usage:

```
diag -T "download"
```

ESCON Bit Error Rate Service Aid

This Service Aid is used to check the bit error rate for an ESCON adapter to assure that the link to the host system is functioning properly. To run the ESCON Bit Error Rate Service Aid, the adapter must be connected, configured, and on-line. If the adapter is not configured properly, the Service Aid is not able to check the bit error rate.

Fibre Channel RAID Service Aids (Device Specific)

The Fibre Channel RAID Service Aids contain the following functions:

- **Certify LUN**

  This selection reads and checks each block of data in the LUN. If excessive errors are encountered the user is notified.

  This task may be run directly from the command line. The following usage statement describes the syntax of the fastpath command:

  Usage:
  `diag -T "certify"`
- **Certify Spare Physical Disk**

  This selection allows the user to certify (check the integrity of the data) on drives designated as spares.

  This task may be run directly from the command line. The following usage statement describes the syntax of the fastpath command:

  Usage:
  `diag -T "certify"`
- **Format Physical Disk**

  This selection is used to format a selected disk drive.

  This task may be run directly from the command line. The following usage statement describes the syntax of the fastpath command:

  Usage:
  `diag -T "format"`
- **Array Controller Microcode Download**

  This selection allows the microcode on the Fibre Channel RAID controller to be updated when required.

  This task may be run directly from the command line. The following usage statement describes the syntax of the fastpath command:

  Usage:
  `diag -T "download"`
- **Physical Disk Microcode Download**

  This selection is used to update the microcode on any of the disk drives in the array.

  This task may be run directly from the command line. The following usage statement describes the syntax of the fastpath command:

  Usage:
  `diag -T "download"`
- **Update EEPROM**

  This selection is used to update the contents of the EEPROM on a selected controller.
- **Replace Controller**

  Use this selection when it is necessary to replace a controller in the array.

Flash SK-NET FDDI Firmware

This task allows the Flash firmware on the SysKonnect SK-NET FDDI adapter to be updated.

Format Media

The Format Media task supports the selection of diskettes, SCSI hardfiles, or SCSI optical media.

This task may be run directly from the command line. The following usage statement describes the syntax of the fastpath command:

Usage:
```
diag -T ″format″
```

Generic Microcode Download

This Service Aid provides a means of executing a ″generic″ script from a diskette. The intended purpose for this ″generic″ script is to load microcode to a supported resource. This script is responsible for executing whatever program is required in order to download the microcode onto the adapter or device.

This Service Aid is supported in both concurrent and standalone modes from disk, LAN, or removable media.

On entry, the Service Aid displays information about what it does. It then asks for a ″Genucode″ diskette to be inserted into the diskette drive. The diskette must be in **tar** format. The Service Aid then restores the script file, ″**genucode**″, to the **/tmp** directory. Then the script is executed. The script must at that point then pull off any other needed files from the diskette. The script should then **exec** whatever program is necessary in order to perform its function. On completion, a status code is returned, and the user is returned to the Service Aid.

The **genucode** script should have a #!/usr/bin/ksh line at the beginning of the file. Return status of 0 should be returned if the program was successful, else a non-zero status should be returned.

Hot Plug Task

This Service Aid allows the user to choose a SCSI device or location from a menu and to identify a device, located in a 7027 system unit.

The Service Aid also does the following:

- Generates a menu displaying all SCSI devices.
- Lists the device and all of it’s sibling devices.
- List all SCSI adapters and their ports.
- List all SCSI devices on a port.

Local Area Network Analyzer

This selection is used to exercise the LAN communications adapters (Token-Ring, Ethernet, and (FDDI) Fiber Distributed Data Interface). The following services are available:

- **Connectivity testing between two network stations**

  Data is transferred between the two stations. This requires the user to input the Internet Addresses of both stations.
- **Monitoring ring (Token-Ring only)**

  The ring is monitored for a period of time. Soft and hard errors are analyzed.

PCI RAID Physical Disk Identify

This selection identifies physical disks connected to a PCI SCSI-2 F/W RAID adapter.

This task may be run directly from the command line. The following usage statement describes the syntax of the fastpath command:

Usage:

```
diag -T "identify"
```

Periodic Diagnostics

This selection provides a tool for configuring periodic diagnostics and automatic error log analysis. A hardware resource can be chosen to be tested once a day, at a user specified time. If the resource cannot be tested because it is busy, error log analysis is performed. Hardware errors logged against a resource can also be monitored by enabling Automatic Error Log Analysis. This allows error log analysis to be performed every time a hardware error is put into the error log. If a problem is detected, a message is posted to the system console and a mail message sent to the user(s) belonging to the system group with information about the failure such as Service Request Number.

The Service Aid provides the following functions:

- Add or delete a resource to the periodic test list
- Modify the time to test a resource
- Display the periodic test list
- Modify the error notification mailing list
- Disable or Enable Automatic Error Log Analysis

Process Supplemental Media

Diagnostic Supplemental Media contains all the necessary diagnostic programs and files required to test a particular resource. The supplemental is normally released and shipped with the resource as indicated on the diskette label. Diagnostic Supplemental Media must be used when the device support has not been incorporated into the latest Diagnostic CDROM.

This task processes the Diagnostic Supplemental Media. Insert the Supplemental Media when prompted, then press *Enter*. After processing has occurred, go to the Resource Selection list to find the resource to test.

Notes:

- This task is supported in Standalone Diagnostics only.
- Always process and test one resource at a time.
- Do not process multiple supplementals at a time.

More information on Diagnostic Supplemental Media can be found at the following link:Diagnostic Supplemental Media.

Run Diagnostics

The Run Diagnostics task invokes the Resource Selection List menu. When the commit key is pressed, Diagnostics are run on all selected resources.

The procedures for running the diagnostics depends on the state of the Diagnostics Run Time Options. See Display or Change Diagnostic Run Time Options section.

Run Error Log Analysis

The Run Error Log Analysis task invokes the Resource Selection List menu. When the commit key is pressed, Error Log Analysis is run on all selected resources.

Save or Restore Hardware Management Policies (CHRP)

Note: Runs on CHRP systems units only.

Use this Service Aid to save or restore the settings from Ring Indicate Power On Policy, Surveillance Policy, Remote Maintenance Policy and Reboot Policy.

- **Save Hardware Management Policies**

  This selection writes all of the settings for the hardware management policies to the file:

  `/etc/lpp/diagnostics/data/hmpolicies`

- **Restore Hardware Management Policies**

  This selection restores all of the settings for the hardware management policies from the contents of the file:

  `/etc/lpp/diagnostics/data/hmpolicies`

This Service Aid may be accessed directly from the command line, by entering:

`/usr/lpp/diagnostics/bin/uspchrp -a`

Save or Restore Service Processor Configuration (RSPC)

Note: Supported on RSPC system units only.

Use this Service Aid to save or restore the Service Processor Configuration to or from a file. The Service Processor Configuration includes the Ring Indicator Power On Configuration.

- **Save Service Processor Configuration**

  This selection will write all of the settings for the Ring Indicate Power On and the Service Processor to the file:

  `/etc/lpp/diagnostics/data/spconfig`

- **Restore Service Processor Configuration**

  This selection will restore all of the settings for the Ring Indicate Power On and the Service Processor from the file:

  `/etc/lpp/diagnostics/data/spconfig`

SCSD Tape Drive Service Aid

This Service Aid provides a means to obtain the status or maintenance information from a SCSD tape drive. Only some models of SCSI tape drive are supported.

The Service Aid provides the following options:

- **Display time since a tape drive was last cleaned.**

  The time since the drive was last cleaned is displayed onto the screen. In addition, a message whether the drive is recommended to be cleaned is also displayed.

- **Copy a tape drive's trace table.**
- **The trace table of the tape drive is written to diskettes.**

  The required diskettes must be formatted for DOS. Writing the trace table may require several diskettes. The actual number of required diskettes is determined by the Service Aid based on the size of the trace table. The names of the data files are of the following format:

  TRACE[*X*].DAT

  where *X* is the sequential diskette number. The complete trace table consists of the sequential concatenation of all the diskette data files.

- **Display or copy a tape drive's log sense information.**

  The Service Aid provides options to display the log sense information onto the screen, to copy it to a DOS formatted diskette or to copy it to a file. The file name **LOGSENSE.DAT** is used when the log sense data is written on the diskette. The Service Aid prompts for a file name when the log sense data is chosen to be copied to a file.

SCSI Bus Analyzer

This Service Aid provides a means to diagnose a SCSI Bus problem in a free-lance mode.

To use this Service Aid, the user should have an understanding of how a SCSI Bus works. This Service Aid should be used when the diagnostics cannot communicate with anything on the SCSI Bus and cannot isolate the problem. Normally the procedure for finding a problem on the SCSI Bus with this Service Aid is to start with a single device attached, ensure that it is working, then start adding additional devices and cables to the bus ensuring that each one works. This Service Aid works with any valid SCSI Bus configuration.

The SCSI Bus Service Aid transmits a SCSI Inquiry command to a selectable SCSI Address. The Service Aid then waits for a response. If no response is received within a defined amount of time, the Service Aid displays a timeout message. If an error occurs or a response is received, the Service Aid then displays one of the following messages:

- The Service Aid transmitted a SCSI Inquiry Command and received a valid response back without any errors being detected.
- The Service Aid transmitted a SCSI Inquiry Command and did not receive any response or error status back.
- The Service Aid transmitted a SCSI Inquiry Command and the adapter indicated a SCSI bus error.
- The Service Aid transmitted a SCSI Inquiry Command and an adapter error occurred.
- The Service Aid transmitted a SCSI Inquiry Command and a check condition occurred.

When the SCSI Bus Service Aid is entered a description of the Service Aid is displayed.

Pressing the Enter key displays the Adapter Selection menu. This menu allows the user to enter which address to transmit the SCSI Inquiry Command.

When the adapter is selected the SCSI Bus Address Selection menu is displayed. This menu allows the user to enter which address to transmit the SCSI Inquiry Command.

Once the address is selected the SCSI Bus Test Run menu is displayed. This menus allows the user to transmit the SCSI Inquiry Command by pressing the Enter key. The Service Aid then indicates the status of the transmission. When the transmission is completed, the results of the transmission are displayed.

Notes:

- A Check Condition can be returned when there is nothing wrong with the bus or device.
- The operating system does not allow the command to be sent if the device is in use by another process.

Service Aids for use with Ethernet

This selection provides a tool for diagnosing Ethernet problems. This Service Aid is used to exercise the Ethernet adapter and parts of the Ethernet network. The Service Aid works by transmitting a data block to itself. This Service Aid works with a wrap plug or with any valid Ethernet network and can be used as a tool to diagnose Ethernet network problems.

When the Ethernet Service Aid is executed, one of the following messages is returned:

- No errors occurred.
- An adapter error occurred.
- A transmit time-out occurred.
- A transmit error occurred.
- A receive time-out occurred.
- A receive error occurred.

- A system error occurred.
- Receive and transmit data did not match.
- An error occurred that could not be identified.
- The configuration indicates that there are no Ethernet adapters in this system unit.
- Another application is currently using the adapter.
- The resource could not be configured.

Spare Sector Availability

This selection checks the number of spare sectors available on the optical disk. The spare sectors are used to reassign when defective sectors are encountered during normal usage or during a format and certify operation. Low availability of spare sectors shows that the disk needs to be backed up and replaced. Formatting the disk does not improve the availability of spare sectors.

This task may be run directly from the command line. The following usage statement describes the syntax of the fastpath command:

Usage:
```
diag -T "chkspares"
```

SSA Service Aids

This Service Aid provides tools for diagnosing and resolving problems on SSA attached devices. The following tools are provided:

- Set Service Mode
- Link Verification
- Configuration Verification
- Format and Certify Disk

Update Disk Based Diagnostics

This Service Aid allows fixes (APARs) to be applied.

This task invokes the SMIT Update Software by Fix (APAR) task. The task allows the input device and APARs to be selected. Any APAR can be installed using this task.

Update System Flash (RSPC)

Note: Supported on RSPC system units only.

This selection updates the system flash for RSPC systems.

The user provides a valid binary image either on diskette or qualified path name. The diskettes can be in DOS or a backup format.

The flash update image is copied to the */var* file system. If there is not enough space in the file system for the flash update image file, an error will be reported. If this occurs, increase the file size of the */var* file system. The current flash image is not saved. The command automatically removes the */var/update_flash_image*.

After user confirmation, the command will reboot the system twice to complete the flash update.

Update System or Service Processor Flash (CHRP)

Note: Runs on CHRP system units only.

This selection updates the system or service processor flash for CHRP system units.

Further update and recovery instructions may be provided with the update. It is necessary to know the fully qualified path and file name of the flash update image file that was provided. If the flash update image file is on a diskette, the Service Aid can list the files on the diskette for selection.

Refer to the update instructions, or the system unit's service guide to determine the level of the system unit or service processor flash.

When run from online diagnostics, the flash update image file is copied to the **/var** file system. If there is not enough space in the **/var** file system for the flash update image file, an error is reported. If this occurs, exit the Service Aid, increase the size of the **/var** file system and retry the Service Aid. After the file is copied, a warning screen asks for confirmation to continue the update flash. Continuing the update flash reboots the system. The system does not return to diagnostics. The current flash image is not saved. After the reboot, the **/var/update_flash_image** can be removed.

When running from standalone diagnostics, the flash update image file is copied to the file system from diskette. The user needs to provide the image on a diskette since the user does not have access to remote file systems or any other files that are on the system. If enough space is not available, an error is reported stating additional system memory is needed. After the file is copied, a warning screen asks for confirmation to continue the update flash. Continuing the update flash reboots the system. The current flash image is not saved.

The **update_flash** command can be used in place of this Service Aid. It is located in the **/usr/lpp/diagnostics/bin** directory.

Attention: The **update_flash** command reboots the entire system. Do not use this command if more than one user is signed onto the system.

7135 RAIDiant Array Service Aid

The 7135 RAIDiant Array Service Aids contain the following functions:

- **Certify LUN**

  This selection reads and checks each block of data in the LUN. If excessive errors are encountered the user is notified.
- **Certify Spare Physical Disk**

  This selection allows the user to certify (check the integrity of the data) on drives designated as spares.
- **Format Physical Disk**

  This selection is used to format a selected disk drive.
- **Array Controller Microcode Download**

  This selection allows the microcode on the 7135 controller to be updated when required.
- **Physical Disk Microcode Download**

  This selection is used to update the microcode on any of the disk drives in the array.
- **Update EEPROM**

  This selection is used to update the contents of the EEPROM on a selected controller.
- **Replace Controller**

  Use this selection when it is necessary to replace a controller in the array.

Application Test Units

Application Test Units (TU) are used by the Diagnostic Applications to test a device. Typically, due to either their large size or their functional composition, TUs are more appropriately written as applications as opposed to being included within device drivers.

This chapter defines requirements for Application Test Unit code and provides guidance for TU Developers who need to develop code for multiple target environments. The TU code should be developed in ANSI C language and according to generally accepted good programming practices, including, but not limited to:

- Modularity
- Readability
- Self Documenting
- Maintainability
- Re-entrant Capability

The use of assembler-level code is strongly discouraged, but may be necessary in certain cases where performance is critical to the effectiveness of the test function. Such code would not be considered portable and would have to be rewritten for the target platform.

The following topics are discussed in detail:

- Test Unit Definition
- Hardware Functional Coverage
- Test Unit Numbering
- Test Unit Code Device Open and Close
- Portability
- In-Service versus Out-of-Service Test Units
- Recommended General Structure of Test Unit Code
- Designing for Multitasking Environments
- Persistent Data and the TU_INFO_HANDLE
- Test Unit Call Interface
- Definition of TU_TYPE Input Structure
- Definition of TU_RETURN_TYPE Output Structure
- Return Codes
- Interrupt Handler Call Interface
- Interrupt Handling in Test Units
- Using the Interrupt Flag Bit Mask
- Programming Interfaces for TUs and Interrupt Handlers
- Configuration Services Device Attributes
- Message Handling
- Signal Handling
- Definition of EXECTU()
- PCI Configuration Space for I/O Devices
- Test Unit 64-bit Porting Guide
- Microcode Download/Display Requirements for Test Units
- Enhanced Error Handling Option

Test Unit Definition

Fundamental to the Test Unit methodology is a basic, modular building block that is referred to as a Test Unit. A test unit is a single operation performed on the system or subsystem under test. Most often this is an individual function test, such as a register read/write test. Several basic assumptions are made for the test units:

- Only one modular test function is performed in each individual test unit.
- Test units are numbered, and the calling application specifies the number of the test unit it wishes to execute.
- No environmental specific code is allowed in a test unit. This specifically includes user interface calls. Also, device-access methods such as reads or writes are done with generic function calls, which can then be defined in a different source file and coded, if necessary, to meet the specific requirements of the target environments.
- Test units are grouped appropriately in source files. This allows custom building of executable libraries to meet the requirements of the target environments.
- In cases where the same test unit may be used to test hardware in different ways based on some control variables (for example, speed or mode settings), that test unit may be used to represent several ″logical″ test units, each with a different test unit number. When the test unit is called, it would interpret the test unit number requested and set the control variables appropriately.

Hardware Functional Coverage

The Test Unit package should be designed and implemented such that if the TUs are run in the recommended order as documented, then a minimum coverage of 95% of the hardware function is achieved.

Test Unit Numbering

Test Units should be numbered according to some logical sequence, which is determined by the TU Developer. Zero should not be used as a TU number. The allowable range for TU numbers is 1 through 61439 (1 through EFFF hex). This numbering requirement must be respected even though the TU member of the **tucb** header structure is defined as a 32 bit integer. It is desirable that a numbering scheme be developed by the TU Developer allowing TUs to be executed in sequential numerical order when executing them as designated. This might include spacing the TUs so that future TUs can be inserted into the number sequence, where appropriate.

Test Unit Code Device Open and Close

Before a device can be tested by one of the **test units**, it must be opened for access through the interfaces defined in ″Programming Interfaces for TUs and Interrupt Handlers″ . Also, when testing is complete, the device must be closed and restored to its original state. The opening and closing of the device for testing presents some problems that must be accounted for in the design of the Test Unit library for the device:

- Errors may occur on the open and close operation, and these must be presented back to the calling applications in a form those applications know how to handle; that is, test unit results.
- Since the calling application will typically run through all or most of the Test Units for a given device, the performance penalty of opening and closing the device for each call to a Test Unit is prohibitive.
- Under different conditions, test units may be run in different combinations and sequences, so the calling application must be able to call the functions which do device open and close independent of the other test functions.

Test Unit Conventions

To provide a standard solution for handling the above problems, the following conventions for Test Units within a specific device library are required:

1. There must be a Test Unit number 1, referred to as **TU_OPEN**, which includes functions to initialize data structures, place the device in the correct state for diagnostics, and open the device for testing. It

does not perform any other test functions. Any error conditions are returned as diagnostic results. The define value **TU_OPEN** should be used as the numerical identifier for this Test Unit.

Specifically, **TU_OPEN** performs the following:

a. Sees that the TU_INFO_HANDLE parameter is set to NULL, allocates a memory buffer to hold persistent data, and assigns TU_INFO_HANDLE to that address. For more information, see ″Persistent Data and the TU_INFO_HANDLE″.
b. Reads needed device attribute information by making calls to the configuration services (**pdiag_cs_get_attr**), and places appropriate information into the **pdiagex_dds_t** structure that is passed as a parameter on the **pdiag_open** call.
c. Calls **pdiag_diagnose_state** to place the device into a testable state.
d. Calls **pdiag_open** to open the device for testing, and loads the interrupt handler, if one exists.
e. Assuming all the above functions are performed without error, returns a value of ″0″ as the major return code.

2. There must be a Test Unit number 61439 (0xEFFF hex), referred to as **TU_CLOSE**, which closes the device and restores the device to the original state it was in prior to diagnostics being invoked. The define value **TU_CLOSE** should be used as the numerical identifier for this test unit.

   Specifically, **TU_CLOSE** performs the following:

   a. Calls **pdiag_close** to close the device, and unloads the interrupt handler.
   b. Calls **pdiag_restore_state** to return the device to the state it was in prior to **TU_OPEN**.
   c. Frees any memory buffers that were allocated by **TU_OPEN**. For the most part, the buffers that need to be freed are ″secondary″ persistent data buffers, pointed to by pointers in TU_INFO_HANDLE.
   d. Assuming all the above functions are performed without error, returns a value of ″0″ as the major return code.
   e. A valid diagnostic sequence consists of a call to Test Unit **TU_OPEN**, some arbitrary number of calls to Test Units other than **TU_OPEN** or **TU_CLOSE**, and then a final call to Test Unit **TU_CLOSE**.

Portability

With today's systems, multiple operating systems are typically supported on a single hardware platform. Since these systems usually share the same hardware features, diagnostics need to be written to support hardware failure analysis that works within any of these operating environments. For this reason, all TU packages must be designed with portability in mind.

Besides the operating environment differences, there is also the need for different types of user interfaces for the different execution environments. For instance, system diagnostics for the field may use a different interface than the hardware exerciser used in the design verification test.

By ensuring that the TU package performs no interaction with the user (output to screen and input from keyboard), one third of the problem will have been solved. Then all the invocations of the TUs will be made through one interface, and different types of user interfaces can be developed with no need to change the TU package.

Another third of the problem concerns how the device gets accessed through the operating environment. Since different operating environments have different device drivers (for example, UNIX drivers, DOS/WIN drivers, Firmware based, or generic I/O, there must be a way to isolate the functional test from the burden of knowing what driver/environment is being used for access. Therefore, standard device-access routines are needed to perform the device accesses on the functional test's behalf. The device accesses typically needed for functional tests are:

- Device Open
- Read

- Write
- Interrupt Setup and Handling
- Direct Memory Access (DMA) Setup and Cleanup
- Device Close

The interface of these routines must be independent of the underlying device-access method (that is, execution environment) by design, and must not change across operating environments. The internals of these routines will change per operating environment, using the appropriate system/driver calls to accomplish the device-access requests on the functional tests' behalf.

In-Service versus Out-of-Service Test Units

The architecture described in this document is primarily for the creation of ″out-of-service″ Test Units, meaning that the device being tested is not available for any other use by the operating system while it is under test. In high-availability systems, however, it is often desirable to have Test Units which can be used while the device is ″in-service.″ This may be especially true for devices which can have partial failures; for example, DASD media, RAID, memory/cache arrays, and multi-port adapters. A variation of In-Service diagnostics can sometimes be done with an Out-of-Service Test Unit that takes over the device for such a short period of time that no service outage is detected.

Test units designed to be run truly concurrently with other operations on the same hardware component will, in general, have to perform their testing through the ″normal″ functional device driver installed by the operating system. Because the device driver model tends to be unique to each operating system, the Test Unit written to that interface may not be easily portable to other operating systems. However, proper structuring of the Test Unit library, as discussed below in ″Recommended General Structure of Test Unit Code,″ will help isolate into a single source file those functions which must be modified.

Recommended General Structure of Test Unit Code

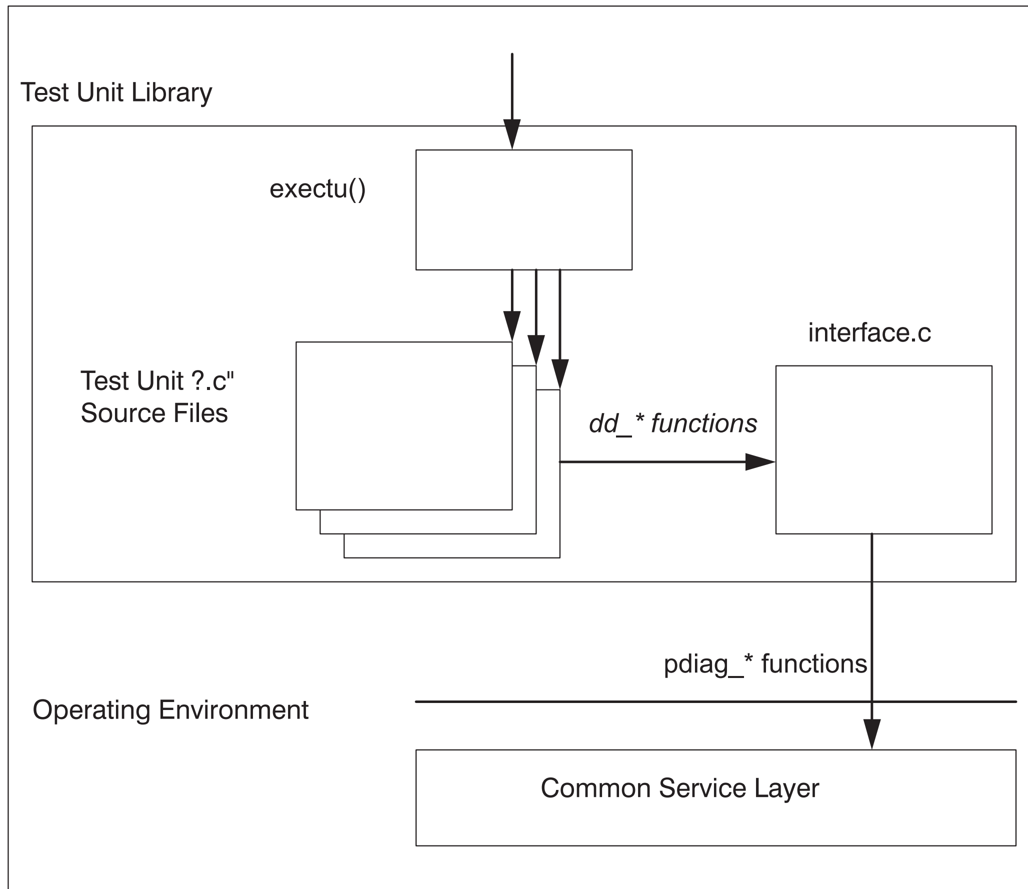
The TU environment specified in this document is designed to provide source code portability of TUs across multiple operating environments. TUs should only use the device and system interfaces specified in this document to ensure portability. However, experience has shown that it is good programming practice to isolate and abstract external functions so that any problems in porting can be corrected within a single source code file. For this reason, it is strongly recommended that TU developers include a special source file in their TU library for the purpose of providing that isolation and abstraction. The following describes a recommended implementation of that source file, given to help promote consistency in TU development. The consistency is very important for long-term maintenance of the Test Unit code.

TU libraries should include a C source file called **interface.c**, which provides a set of abstracted device functions that can be used by the actual TU functions. The following is a list of functions that should be implemented within the **interface.c**.

dd_open Prepares a device for testing and obtains needed device attributes.
dd_close Cleans up after testing.
dd_read Performs a read operation.
dd_write Performs a write operation.
dd_dma Initializes, pins, and cross-memory attaches the user buffer for a DMA operation.
dd_dma_enable Enables/Disables a DMA operation.
dd_dma_cleanup Deallocates any resources previously allocated for a DMA operation.
dd_interrupt Processes interrupt conditions.

As illustrated below, these functions should provide mappings to one or more of the services described in ″Programming Interfaces for TUs and Interrupt Handlers″ .

The figure also illustrates how TU libraries should include a C source file that implements the **executu**() interface, which provides the program entry point for the TU library, decodes the specified TU number to the correct internal function, and calls that function.



General Structure of a Test Unit Library

Designing for Multitasking Environments

Test units should be designed with rules of re-entrance in mind. Although it is unlikely that a given set of Test Units could be run simultaneously against the same device, it is possible that more than one of the same type of device (or devices which are tested by the same TU code) exists in the system. Since it may be desirable to run the Test Units concurrently as part of a system exerciser or a stress test for a specific subsystem, it is possible that the same TU code may be run in different threads under the same process. The use of static variables in this case could lead to data conflicts between the multiple instances of TU code execution.

Persistent Data and the TU_INFO_HANDLE

Because of the requirement to allow multi-threaded, simultaneous execution of Test Units, the TU functions must be written to be re-entrant, implying that statically defined variables or structure are not allowed.

Note: Static constant values are not a problem.

To illustrate the problem, imagine two threads of execution calling the same TU to run simultaneously against two device instances of the same type. Values stored in static variables would get changed in both threads of execution, probably leading to a program failure. Therefore, all variables and structures must be

either defined locally as stack variables, or created using allocated memory. Without static variables, it is difficult to retain any data around from one execution of a TU to the next.

The intent of the TU_INFO_HANDLE pointer in the **executu**()interface is to provide the TU writer with a pointer to a data buffer that will persist across multiple execution calls to specific Test Units. On the first call to a TU library, the TU_INFO_HANDLE pointer will be set to NULL. The first TU, TU_OPEN, must allocate the buffer and set the TU_INFO_HANDLE pointer. Data that the TU writer wants to have persist (for example, device attribute information) can then be placed within that buffer, and the pointer to the buffer will be passed back on each subsequent call to the TU library.

Because the data buffer remains allocated after the TU returns control to the calling application, it is the responsibility of the calling application to free the buffer any time that a premature termination is required, or after it calls the last TU (TU_CLOSE).

Data that should be kept in the persistent data buffer includes:

- The *pdiagex_dds_t* structure which contains several device attributes and is used as a parameter to the **pdiag_open** call.
- The PDIAG_INFO_HANDLE returned from the **pdiag_open** call, which is used as an input parameter to all the other device operation functions.
- An indicator of the state of the device (DIAGNOSE or NORMAL)
- Other device-attribute information obtained from Configuration Services using the **pdiag_cs_get_attr** function (to avoid the overhead of rerequesting it for each TU call).
- Any other information the TU writer would like to have persist from one call to the next.

Test Unit Call Interface

To execute test units, a C language function with the name **executu**() has been defined to provide the interface between the test unit code and the managing application. The definition of this interface has been developed to:

- Hide the complexity of the structures and protocols used in performing functional tests
- Provide a uniform interface for all the different management applications that may invoke the test unit code.

See the section ″Definition of EXECTU″.

Definition of TU_TYPE Input Structure

The **executu**() interface is dependent on the definition of a Test Unit Control Block (TUCB) structure. The TUCB is defined as a C language data type called TU_TYPE, and is located in the **diag/tucb.h** header file. This header file must be used without modification and included in each source file using the structure.

To make the test unit functions available to a wide range of managing applications, this TUCB structure must not deviate from the defined structure. No new data types or structures may be added. Each test unit should be self-sufficient in the function provided. The data types OUTPUT_DATA and INPUT_DATA are declared as 'void' in the **diag/tucb.h** file. If these structures are to be used, two header files are required to redefine these parameters:

- The {DEVICE}_err_detail.h File file should be used to define device specific error log detail output data (OUTPUT_DATA).
- The {DEVICE}_input_params.h File file should be used to define device specific input parameter data for a test unit (INPUT_DATA).

Both header files (if used) should be included before the **diag/tucb.h** file.

The TU_TYPE structure is specified as follows:

```
typedef struct tucb_t {
          char  *resource_name;
          TU_INPUT_TYPE parms;
} TU_TYPE;
```

The resource_name is a string containing the name of the hardware or physical device (as defined by the operating system) on which to run the test unit. TU_INPUT_TYPE is a substructure of TU_TYPE, and contains several input parameters, as specified in the following:

```
typedef struct tucb_in_t {
           ulong tu;
           ulong loop;
           OUTPUT_DATA *data_log;
           ulong data_log_length;
           INPUT_DATA *tu_data;
           ulong tu_data_length;
           FILE *msg_file;
} TU_INPUT_TYPE;
```

See ″Definition of EXECTU()″ for structure member definitions.

Note: For most applications, the TU number and loop count are the only parameters required. However, this interface allows for an open way of passing special parameters into the Test Units and receiving detailed data back out, to allow for specialized testing environments.

Using such data requires specific knowledge about the Test Unit design in the calling application, and does not allow for generic diagnostic handling, as would be required from a system management application. However, this design would allow a remote diagnostic application, which could have detailed diagnostic design knowledge, to work through a local agent function which only has generic diagnostic knowledge. The local agent would only have to allocate buffers of the requested size, and pass data between the Test Units and the remote diagnostic application.

Definition of TU_RETURN_TYPE Output Structure

The **exectu**() interface expects, as a return value, a unsigned long *major_rc* return code value. As an extension of this return value, a Test Unit Control Block (TUCB) return structure is included as a third argument to the **exectu**() function call. The TUCB return structure is defined as a C language data type called TU_RETURN_TYPE, and is defined in the **diag/tucb.h** header file. This header file must be used without modification and included in each source file where the structure is used.

```
typedef struct tucb_out_t {
           ulong             major_rc;
           ulong             minor_rc;
           ulong             actual_loop;
           ulong             data_log_length;
           ulong             severity;
} TU_RETURN_TYPE;
```

See ″Definition of EXECTU()″ for structure member definitions.

Return Codes

major_rc

The *major_rc* return value from the **exectu**() function should indicate the success or failure of the TU which was executed. If all testing is successful, it should return a value of zero (0), otherwise a non-zero value should be returned corresponding to a specific value. A managing application uses the *major_rc* return code to determine the flow of the diagnostic procedure, and to look up the appropriate card level Field Replaceable Unit (FRU) or FRUs to be replaced. To satisfy the failure-isolation requirements of all managing applications, the return codes should be designed to be as granular as possible to provide maximum fault isolation. For most purposes, this means attempting to isolate to a single FRU.

Attention: When defining *major_rc* return codes, keep the following in mind:

- Never return memory offset information in the return code.
- Do not return any detailed information, such as failing bits, through the return code. Instead, use the OUTPUT_DATA error log.

minor_rc

The *minor_rc* return value is used to pass back a more specific error indication, and would typically be provided as an aid for fault isolation within a FRU, perhaps down to modules or I/O lines. This information is intended for use in bring-up and debug, and in manufacturing, to point to a specific hardware defect. Used in conjunction with the OUTPUT_DATA error log, the TU writer should be able to pass back enough information to isolate to a failure to whatever level is needed. However, most management applications will only be interested in the *major_rc* return value.

Interrupt Handler Call Interface

The diagnostic interrupt handler function for a device must be packaged in an executable module separate from the Test Unit library. This module is loaded into the operating system and registered with the diagnostic system services when the TU_OPEN calls the **pdiag_open** function.

When the services receive an interrupt, control is passed to these ″second-level″ interrupt handlers in sequential order. Each interrupt handler reads the status of its respective adapter to see if it was the source of the interrupt. If the Test Unit is waiting for the interrupt by calling the **pdiag_dd_watch_for_interrupt** service, the *sleep_flag* will be set to 1, indicating that the interrupt handler should do a **pdiag_dd_interrupt_notify** when it has completed.

Interrupt handlers can use the device methods to read and write operations on the device. Typically, they will read registers on the device to obtain more information about the interrupt, and write registers (if necessary) to clear the interrupt condition. The content of any data passed back to the TU through the *data_area* buffer, and whether the TUs even wait for interrupts, is a decision left to the designer of the TUs and interrupt handler. That decision depends upon the operation of the specific device and how it is being tested.

Syntax

The function entry prototype for an interrupt handler is as follows:

```
int device_interrupt (
        PDIAG_INFO_HANDLE *handle,
        pdiag_addr_t data_area,
        int32 *interrupt_flag,
        uint32 sleep_flag,
        uint32 *sleep_word )
```

Parameters

handle	Pointer to a handle for use in device operations
data_area	Buffer area where the interrupt handler can store information that the Test Unit can review after interrupt processing is complete.
interrupt_flag	Bit field indicating which interrupt occurred
sleep_flag	Boolean value to indicate whether the waiting Test Unit should be notified
sleep_word	Semaphore that the Test Unit is waiting for, used as a parameter to the ***pdiag_dd_interrupt_notify service***

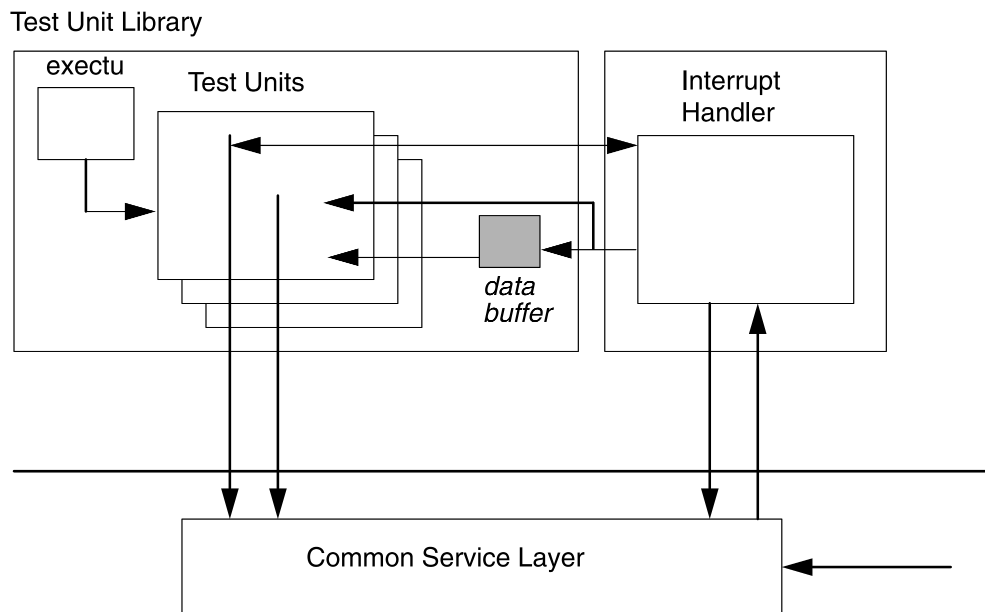
Interrupt Handling in Test Units

A typical sequence of events in the functional flow of a Test Unit is to set up a device operation through reads and writes to the device address space, and then wait to receive an interrupt from the device to

indicate that an operation has completed or needs attention. Since interrupt handling is device-specific and part of the test process, an interrupt handler function must be provided in addition to the Test Unit library. When a device is opened for testing by Test Unit 1 (TU_OPEN), an interrupt handler may be loaded (if one is needed) by passing an interrupt handler module name as one of the parameters on the **pdiag_open** system service. A data buffer address is also passed as part of the input to the **pdiag_open** function, so the device methods know which interrupt handler to use, as well as where to pass back data from the interrupt handler.

The purpose of the interrupt handler function is to receive the interrupt indication, possibly gather some information from the device, clear the interrupt condition on the device, and notify a waiting Test Unit that the interrupt has occurred. Clearing of the interrupt condition is critical, because the interrupt handler will be called continuously as long as the interrupt condition exists. Since this function is called to handle a specific device I/O interrupt, the information it gathers from the device is useful in diagnosing the device behavior. The interrupt handler puts this information into the data buffer area (defined at device-open time), where the waiting Test Unit can access it for analysis.

The basic flow of interrupt processing is shown in the ″Interrupt Processing in Test Units″ illustration. The flow of events is as follows:



Interrupt Processing in Test Units

1. An **exectu**() call is made to Test Unit 1 (TU_OPEN), which calls **pdiag_open** to open the device for testing. Included in the input information passed to **pdiag_open** is the name of the interrupt handler module and the address of a memory-allocated data buffer area.
2. A Test Unit is started, which performs some operations on the device, and then calls **pdiag_dd_watch_for_interrupt** to wait for a response in the form of a device interrupt (or a time-out if no interrupt occurs).
3. The device-methods layer receives an interrupt indication from the operating system.
4. The device-methods pass control to the registered interrupt handler.
5. The interrupt handler function gathers data from the device and places it in the data buffer area, clears the interrupt, and releases the Test Unit from its WAIT state.
6. The interrupt handler completes and returns to the caller (the device methods).
7. The Test Unit continues execution by processing the data returned from the interrupt handler.

8. When testing is completed, a call is made to Test Unit 0xEFFF (TU_CLOSE), which calls **pdiag_close** to close the device and unload the interrupt handler.

The cycle of device setup and wait for interrupt can be repeated as often as necessary during the execution of the Test Units. Registration of the interrupt handler only needs to be done once, at the time when the device is opened for testing. However, different interrupt handlers could be used (if necessary) by closing the device, then reopening the device with a different interrupt handler module-name parameter.

Using the Interrupt Flag Bit Mask

The *interrupt_flag* parameter to the interrupt handling routine and the *flag_mask* parameter on the **pdiag_dd_watch_for_interrupt** system service are used by the Test Unit and interrupt handler to communicate the type of interrupt that occurred, and which types of interrupts the Test Unit wants to know about. The bit fields within these words can be defined in whatever way the TU developer wants to assign them, based on the device involved and how many different interrupt types it can surface. However, it is important to understand how these parameters should be used.

When an interrupt handler is called as the result of an interrupt condition, it should examine its device to see which type of interrupt, if any, occurred on that device. If it detects no interrupt condition, the *interrupt_flag* should be set to 0 before it returns. If it does detect an interrupt condition, then it should set an appropriate bit equal to 1 in the *interrupt_flag* before it returns.

A TU waits for an interrupt condition to occur by calling the **pdiag_dd_watch_for_interrupt** service, and one of the parameters to that function is a *flag_mask* word. This is defined as a bit mask, using the same bit definitions as in the interrupt handler, to indicate the interrupt types for which the TU wants to watch. It does this by setting one or more bit values equal to 1, where each bit represents an interrupt type. The **pdiag_dd_watch_for_interrupt** will not return until either an appropriate interrupt is detected (essentially determined by a non-zero result when ″and″ing the *flag_mask* and the *interrupt_flag* 0 values), or until the time limit is reached.

> **Note:** If the Test Unit writer wants to watch for more than one interrupt type, but also needs to know which specific interrupt occurred, the writer should define a structure element in the *data_area* buffer where the interrupt handler can pass back that information.

Example

```
#define Int_A 80000000       /* Common defines used by both the TU and */
#define Int_B 40000000       /* interrupt handler                      */
#define Int_C 20000000
```

Assume TU calls **pdiag_dd_watch_for_interrupt** with:

```
flag_mask = Int_A | Int_B
```

Case 1:

1. Interrupt received
2. Interrupt handler reads device, sees Interrupt C, sets:
   ```
   interrupt_flag = Int_C
   ```
3. **pdiag_dd_watch_for_interrupt** does not return until timeout occurs.

Case 2:

1. Interrupt received
2. Interrupt handler reads device, sees interrupt A, sets:
   ```
   interrupt_flag = Int_A
   ```
3. **pdiag_dd_watch_for_interrupt** returns

Case 3:

1. Interrupt received
2. Interrupt handler reads device, sees both interrupt B and C, sets:
```
interrupt_flag = Int_B | Int_C
```
3. **pdiag_dd_watch_for_interrupt** returns

Programming Interfaces for TUs and Interrupt Handlers

System interface calls and use of header files should conform to the XPG4 standards. This ensures portability to other platforms meeting the same standards.

The following table lists the standard set of services available to TU developers. Using only these services provides portability of TUs to other platforms where this diagnostic infrastructure is supported. See ″Diagnostic Kernel Extension Interfaces″ for more information on these functions, their input parameters, and the function prototypes.

Function Name	Description	Usable by:
pdiag_open	Opens a device for testing	TU
pdiag_close	Frees up a device after testing	TU
pdiag_dd_read	Performs a read operation to a device	TU, 32-bit Interrupt
pdiag_dd_write	Performs a write operation to a device	TU, 32-bit Interrupt
pdiag_dd_dma_setup	Initializes, pins, and cross-memory attaches user buffer for a dma operation	TU
pdiag_dd_dma_enable	Enables/disables a dma operation	TU
pdiag_dd_dma_complete	Unpins the dma user buffer and detaches the cross-memory descriptor	TU
pdiag_dd_watch_for_interrupt	Waits for device interrupt to occur, or until a specified timeout is reached	TU
pdiag_dd_interrupt_notify	Notifies waiting test unit that an interrupt has been received	32-bit Interrupt
pdiag_diagnose_state	Places device under test into a testable state	TU
pdiag_restore_state	Places device under test into original state before testing	TU
pdiag_cs_open	Open/Initialize configuration data services	TU
pdiag_cs_close	Close/Terminate configuration data services	TU
pdiag_cs_get_attr	Obtain device attribute value	TU
pdiag_cs_free_attr	Free storage that was allocated by **pdiag_cs_get_attr**	TU
findmcode	Locate specific microcode file for loading	TU
pdiag_dd_read_64	Performs a read operation to a device	64-bit Interrupt
pdiag_dd_write_64	Performs a write operation to a device	64-bit Interrupt

Configuration Services Device Attributes

The configuration data services provided by the **pdiag_cs_*** functions (described in the previous table) define the interface by which the TU developer may obtain information about the device under test. The table below lists the standard attributes which may be available for a given device; however, not all attributes are supported for all devices, since some are specific to particular device types.

Normally, the TU developer should use this service to gather the required attribute information during the call to Test Unit TU_OPEN (the Test Unit which opens the device for testing), and save this device information for reference during subsequent Test Unit calls. This avoids the performance overhead of calling the configuration services many times during the execution of a set of Test Units.

Standard Attribute	Description
bus_id	Adapter I/O bus ID value
bus_intr_lvl	Bus interrupt level
bus_io_length	Base address of bus I/O area
bus_mem_addr	Base address of Shared Bus Memory area
bus_mem_length	Length of Shared Bus Memory area
bus_type	Type of bus (for example, Microchannel, PCI, 60X)
connwhere	Connwhere location as stored in CuDv
dms_bus_flags	Bus flags for DMA operation (PCI/ISA only)
dma_bus_length	Length of bus memory DMA area in bytes (MCA only)
dma_bus_mem	Address of bus memory used for DMA (MCA only)
dma_chan_id	DMA channel ID of device
dma_flags	Flags to indicate DMA actions (MCA only)
dma_lvl	DMA bus arbitration level (MCA only)
intr_flags	Interrupt flags
intr_priority	Interrupt priority
maxmaster	Maximum number of concurrent DMA master calls
parent_name	Parent device name
slot_num	Slot number of adapter (for MCA, actual slot number, for PCI, device number)

Message Handling

In general, there should be no **printf**() or **fprintf**() calls imbedded in TU code which is delivered for production use. This includes debug messages, execution-progress messages, and so on. However, it is understood that such practices are common and useful during the initial code development, and sometimes desirable at a later time when something breaks. Therefore, to satisfy both requirements, the messages should be allowed to be conditionally compiled in and out of the code. To allow the calling application to redirect the messages to any file, including **stdout**, only the **fprintf**() call should be used. Then, to conditionally compile the messages, the following convention should be followed:

In one of the include files, define the following PRINT macros conditionally with the standard conditional flag TU_DEBUG_MSG.

```
#ifdef TU_DEBUG_MSG
#define PRINT( args ) fprintf args
#else
#define PRINT( args )
#endif
```

Next, use the *"msg_file"* pointer in the TUCB structure definition which determines where messages will be sent.

Then, at any place in the code where a message should be output, use the PRINT macro. The calling application would then set the *"msg_file"* parameter to **stdout** in order to have messages directed to a terminal or monitor. Alternatively, to have messages directed to a file, the calling application would use the **fopen**() function to open a file and set *"msg_file"* to the pointer returned from this call.

For example, you want to print the message ″Hello, World number 1″, and **tucb_ptr** is a pointer to the TU_TYPE structure passed by the application, and **w_num** is a variable with a value of 1. You could then insert, at an appropriate place in the TU code, a line like the following:

```
PRINT((tucb_ptr->parms.msg_file, "Hello, World number %d",w_num));
```

> **Note:** The double parentheses are required to pass variable-length argument lists through the PRINT macro to the **fprintf**() function.

Signal Handling

In general, signal handling is the responsibility of the DA. When a signal to terminate is caught, the signal handler must start **TU_CLOSE** through the **exectu**() interface, so that a proper cleanup of the device is performed and a release of resources occurs. **TU_CLOSE** should be started only if **TU_OPEN** has already been called successfully.

Definition of EXECTU()

Purpose

Executes test unit (TU) bound into a Diagnostic Application (DA).

Syntax

```
#include   <diag/tucb.h>
ulong
exectu ( TU_TYPE *tucb_ptr, TU_INFO_HANDLE *tu_handle, TU_RETURN_TYPE *tu_rc)
```

Description

The **exectu** subroutine runs an TU referenced by the test unit control block. The test units are normally built and packaged as a loadable library. The device to be tested by the test unit is referenced by a character-string designator indicating the device instance.

Parameters

tucb_ptr
: Pointer to the test-unit control block. This structure is defined in **diag/tucb.h** file.

```
typedef struct tucb_t {
            char  *resource_name;
            TU_INPUT_TYPE parms;
      } TU_TYPE;
```

where `TU_INPUT_TYPE` is as follows:

```
typedef struct tucb_in_t {
            ulong tu;
            ulong loop;
            OUTPUT_DATA *data_log;
            ulong data_log_length;
            INPUT_DATA *tu_data;
            ulong tu_data_length;
            FILE *msg_file;
} TU_INPUT_TYPE;
```

tu
: Test-unit number of the test unit to run.

loop
: Indicates the number of times the test unit should be run provided that an error does not occur.

data_log
: Error details log and or output data log. This log is device specific and is defined by the **{device}_output_data.h** file. It should point to an empty array of structures and then filled in with output or error detail data by the test unit(s). This parameter should be initialized by the calling application if intended to be used.

data_log_length
: Size of the data_log structure. This field is used when passing the tucb data to a remote managing application. This number is initialized by the calling application by calculating the size of the data structure to be filled in and multiplying it by the number of records to be logged. The test unit calculates the number of records by dividing this number by the size of the intended OUTPUT_DATA structure to be used. A *data_log_length* value of zero results in no data being logged to the data_log.

tu_data
: Input parameter to be used to pass extra input data to the test units. This parameter must only be used as special case scenarios. It is intended for special applications such as manufacturing or hardware exercisers.

tu_data_length
: Size of the tu_data structure. This field is used when passing the tucb data to a remote managing application. This number is initialized by the calling application by calculating the size of the data structure to be filled in and multiplying it by the number of records to be logged. The test unit calculates the number of data records by dividing this number by the size of the intended INPUT_DATA structure to be used.

tu_handle
: Pointer to a block of data that the TUs need to have persist between subsequent calls to the TU library. Content and layout of the persistent data is a decision left to the TU writer, but there are certain data structures which should be kept here, as described in the next section. Pointer variable is defined in the diagnostic application, but it is set by TU_OPEN to point to a memory buffer allocated by the TU_OPEN code. This structure is defined in **diag/tucb.h** file.

tu_rc
: Pointer to the test-unit control block return code structure. This structure is defined in **diag/tucb.h** file.

```
typedef struct tucb_out_t {
          ulong                major_rc;
          ulong                minor_rc;
          ulong                actual_loop;
          ulong                data_log_length;
          ulong                severity;
} TU_RETURN_TYPE;
```

major_rc
: Major return code. Used for FRU isolation.

minor_rc
: Minor return code. Used for more granular detailed fault isolation.

actual_loop
: Indicates the number of times the test unit ran.

data_log_length
: Returns the total number of data log records that have been recorded.

severity
: Indicates the severity of a diagnostic failure.

Return Value

The *major_rc* return code is defined as the output from a test unit. This is the same value contained in the TU_RETURN_TYPE structure.

Upon successful completion with no failure, a value of 0 should be returned in the *major_rc* field.

PCI Configuration Space for I/O Devices

There are several writable fields in the standard PCI Configuration Header for PCI devices. They are:

- Command Register
- Latency Timer
- Cache Line Size
- Base Address Registers
- Expanded ROM Base Address

• Interrupt Line

Some of these are written by the firmware and should never be changed by the device driver. The PCI Configuration Header Programming Table must be followed when programming the PCI Configuration Header registers.

PCI Configuration Header Programming Table

<table>
<tr><th>Register/Bit Name</th><th>Firmware Action
(Boot or ibm, configure-connector call)</th><th>Software (Device Driver) Action</th></tr>
<tr><td>Command/Fast Back-to-Back Enable</td><td>Write to a value of 0 on platforms capable of PCI Hot Plug. May be written to a value of 1 on non-Hot-Plug capable platforms if all I/O devices on the same PCI bus are capable of Fast Back-to-Back transfers.</td><td rowspan="7">Preserve value</td></tr>
<tr><td>Command/SERR# enable</td><td>Write a value of 1</td></tr>
<tr><td>Command/Wait cycle control</td><td>Write to a value of 0 (may be hardwired to a 1, so may be a 1 when read even after being written to a 0)</td></tr>
<tr><td>Command/Parity Error Response</td><td>Write a value of 1</td></tr>
<tr><td>Command/VGA Palette snoop</td><td>Write a value of 0</td></tr>
<tr><td>Command/Memory Write and Invalidate Enable</td><td rowspan="2">Write to 0 (reset value)</td></tr>
<tr><td>Command/Special Cycles</td></tr>
<tr><td>Command/Bus Master</td><td>Write to 0 (reset value) unless boot device.</td><td>Must write to a 1 before the first DMA operation. Must write to a 0 before unconfiguring device driver.</td></tr>
<tr><td>Command/Memory Space</td><td rowspan="2">Write a value of 0 (reset value) unless boot device, in which case does not write a value of 1 until BARs and Expansion ROM Base Address are set. Only written to a 1 if that specific address space is used for that I/O device.</td><td>Must write to a 1 before the first operation (if any) to the I/O devices memory space. Must write to a 0 before unconfiguring device driver.</td></tr>
<tr><td>Command/IO Space</td><td>Must write to a 1 before the first operation (if any) to the I/O devices I/O space. Must write to a 0 before unconfiguring device driver.</td></tr>
<tr><td>Build-in Self Test (BIST)</td><td>Write a value of 0</td><td>If BIST is implemented, can write to a 1 to initiate BIST</td></tr>
<tr><td>Latency Timer</td><td rowspan="2">Initialize to a system-specific value</td><td rowspan="2">Preserve value</td></tr>
<tr><td>Cache Line Size</td></tr>
<tr><td>Base Address Registers</td><td rowspan="2">Initialize based on size requested and address space available</td><td>Writes based on the ODM M.n and O.n customized attributes</td></tr>
<tr><td>Expansion ROM Base Address</td><td>Writes based on the ODM M.n and O.n customized attributes. Write LSB to a 0 before enabling the Command/Memory Space if Expansion ROM not used by software.</td></tr>
<tr><td>Interrupt Line</td><td>Ignore</td><td>Ignore - get information from ODM</td></tr>
</table>

Test Unit 64-bit Porting Guide

Changes to the pdiagex kernel extension running under a 64-bit kernel were designed with the test unit developer in mind. Most of the changes required to port the test units are done at the Second Level Interrupt Handler (SLIH) level. For a test unit developer that has followed the architecture specified in this document, the changes are minor and will require minimal testing.

Before porting an existing set of test units, it is important to understand the test units application environment as well as the 64-bit C language data model and how it differs from the 32-bit model.

Test units execute as 32-bit applications under a 32-bit kernel and therefore only use 32-bit kernel extensions (pdiagex). This porting guide describes the required changes to the test units and SLIH in order to function under a 64-bit kernel. The test units will continue executing as 32-bit applications: only the SLIHs will be 64-bit applications.

C Language Data Model

The C language data model used in the 32-bit and 64-bit operating system environments are defined in the following table. You must consider the size of the data passed from the Test Units to the SLIHs and back, since sizes can change as they are passed from one environment to the other. Use special care when passing information in the form of structures or pointers.

C Type	32-bit Data Size	64-bit Data Size
char	8 bits	8 bits
short	16 bits	16 bits
int	32 bits	32 bits
long	32 bits	64 bits
long long	64 bits	64 bits
pointer	32 bits	64 bits

Makefile

To support 32-bit and 64-bit SLIHs, the SLIH Makefile has to be modified to build two executables; one for 32-bits that will remain named as it is today and one for the 64-bit SLIH which will have `64` appended to the name.

File Names	Syntax	Example
32-bit	*filename*	fcphal_intr
64-bit	*filename*`64`	fcpthal_intr64

Makefile Source

Here is an example of what a common source 32-bit and 64-bit SLIH Makefile might look like:

> **Note:** Replace the *environment variables* and *file names* with your own names to customize this example for your own use.

```
# @(#)17     1.1  src/idd/en_US/aixprggd/diagunsd/TU_64bit_port.htm, iddiagunsd, idd500 5/23/00 13:54:31
#

.include <${MAKETOP}bos/kernext/Kernext.mk>


TU_VPATH    = ${MAKETOP}/bos/diag/tu/tu_dir
VPATH       = ${MAKETOP}bos/kernel/exp:${MAKETOP}bos/kernext/exp:$TU_VPATH
```

```
# 32-bit version of load object
#
KERNEL_EXT      = your_intr

# 64-bit version of load object
#
KERNEL_EXT64    = your_intr64

IDIR            = /usr/lpp/diagnostics/slih/

# install list containing 32-bit and 64-bit version
#
ILIST           = your_intr your_intr64

OPT_LEVEL       =    -qlist -qsource

# entry point, import and export files for 32-bit version
#
your_intr_DEPENDS        = your_intr.exp
your_intr_ENTRYPOINT     = your_interrupt
your_intr_IMPORTS        = -bI:pdiagex.exp
your_intr_EXPORTS        = -bE:your_intr.exp

# entry point, import and export files for 64-bit version
# (common with 32-bit version)
your_intr64_DEPENDS      = your_intr.exp
your_intr64_ENTRYPOINT   = your_interrupt
your_intr64_IMPORTS      = -bI:pdiagex.exp \
                                   pdiagex64.exp
your_intr64_EXPORTS      = -bE:your_intr.exp

# object list definition for 32-bit version
#
your_intr_OFILES     =  your_intr.o

# object list definition for 64-bit version (common objects
# across 32-bit and 64-bit versions), with 64-bit objects
# renamed to .64o
#
your_intr64_OFILES   =  your_intr.64o

INCFLAGS   = -I${MAKETOP}/bos/diag/tu/tu_dir \
             -I${MAKETOP}bos/usr/include
LIBS       = ${KERNEXT_LIBS}


.include <${RULES_MK}>
```

SLIH Conversion Tips

To achieve a clean SLIH conversion, pay special attention to the following:

- Any source code that assumes that **int**, **long** and **pointer** types are the same size must be corrected (reshaped) for 64-bit environment.
- Review any type casting, since the underlying data types may have changed.
- Make sure that any data structures containing long types and pointers are checked for sizes, especially data passed between test units and SLIHs (data_area). Refer to the C Language Data Model table. Also see Interrupt Handler Call Interface to make sure the *data_area* contains the proper data types.

  When long types or pointers (or both) are passed in this structure, the structure must be reshaped before it is used by the SLIH.
- Use system-derived types for type declarations whenever possible.

SLIH Conversion Required Changes

The following required changes must be applied to all SLIHs being ported to 64-bit kernel:

1. **Performing Read Operations to a Device**

   All instances of **pdiag_dd_read** will have to be duplicated with **pdiag_dd_read_64** for 64-bit. Every place where **pdiag_dd_read** is used for a 32-bit SLIH, a **pdiag_dd_read_64** will be used for a 64-bit SLIH. This will be accomplished by using conditional preprocessor compiler statements (`#ifdef`).

   Here is an example of what a common source 32-bit and 64-bit read call might look like:

   ```
   #ifdef __64BIT_KERNEL
        rc = pdiag_dd_read_64(pdiagex_handle, IOSHORT16, io_addr, &datas,
   &flags);
   #else
        rc = pdiag_dd_read(pdiagex_handle, IOSHORT16, io_addr, &datas,
   &flags);
   #endif
   ```

   Note:
   - The __64BIT_KERNEL compiler directive is defined for 64-bit kernel compilers, therefore the user will not need to define it.
   - Special case for IOLONG32 reads, the data has to be shifted 32-bits right after the function call, such as, (`data = data >> 32;`).
   - The **pdiag_dd_read_64** function is used in kernel environment only, therefore the **intrlev** flag must always be set to `INTRKMEM`.

2. **Performing Write Operations to a Device**

   All instances of **pdiag_dd_write** have to be duplicated with **pdiag_dd_write_64** for 64-bit. Every place where **pdiag_dd_write** is used for a 32-bit SLIH, a **pdiag_dd_write_64** will be used for a 64-bit SLIH. This will be accomplished by using conditional preprocessor compiler statements (#ifdef).

   Here is an example of what a common source 32-bit and 64-bit write call might look like:

   ```
   #ifdef __64BIT_KERNEL
           rc = pdiag_dd_write_64(pdiagex_handle, IOLONG32, io_addr,
   &datal, &flags);
   #else
           rc = pdiag_dd_write(pdiagex_handle, IOLONG32, io_addr, &datal,
   &flags);
   #endif
   ```

   Note:
   - The __64BIT_KERNEL compiler directive is defined for 64-bit kernel compilers, therefore the user will not need to define it.
   - The **pdiag_dd_read_64** function is used in kernel environment only, therefore the **intrlev** flag must always be set to `INTRKMEM`.

3. **SLIH function prototype**

   The SLIH function prototype requires change in the type declaration for ***sleep_word** and **sleep_flag** as follows:

   ```
   int your_interrupt(pdiag_info_handle_t pdiagex_handle, char
   *data_area, int *interrupt_flag,
   #ifdef __64BIT_KERNEL
           long sleep_flag, long *sleep_word)
   #else
           int sleep_flag, int *sleep_word)
   #endif
   ```

Related Information

"Chapter 3. Diagnostic Components" on page 11 for general information on how to write interrupt handlers.

Interrupt Handler Call Interface

pdiag_dd_read, pdiag_dd_read_64 functions

pdiag_dd_write, pdiag_dd_write_64 functions

Microcode Download/Display Requirements for Test Units

Any adapter or device that has resident microcode or firmware that can be updated in the field has a separate Test unit for both the display of the installed microcode or firmware level and the installation of the microcode or firmware. Use a separate Test Unit for each specific function (display and install) as follows:

Microcode Display:
: This Test Unit provides the calling application with all the present microcode revision levels residing in the adapter or device under test. All device specific output resulting from a microcode device or adapter queries are passed to the calling application using OUTPUT_DATA (*data_log) as defined in TU_INPUT_TYPE. For more information refer to Definition of EXECTU().

Microcode Installation:
: This Test Unit provides a function to update the Adapter or Device Microcode. The Microcode file name is passed from the calling application using INPUT_DATA (*tu_data) as defined in TU_INPUT_TYPE. For more information refer to Definition of EXECTU().

Enhanced Error Handling Option

The Diagnostics Test Units Application interface consists of adapting all read functions as follows:

- All data reads for the adapter must be verified that the data read is other than all 1s, unless otherwise expected. Any data reads that result in all 1s produce a unique error, which is reported to the Diagnostics application.
- A test unit that expects all 1s as normal operation, because of a particular test's nature, does not report the error until that error is verified by the requesting data as being caused by all 1s.
- Diagnostics application developers and test unit developers must determine jointly a unique error code for enhanced error handling.

Diagnostic Kernel Extension

This section describes the use of and programming interfaces to the Diagnostic Kernel Extension (**PDIAGEX**) and device configuration services. The *pdiag_* calls are contained in **/usr/lib/libpdiag.a**. The *pdiag_dd_* calls are contained in **/usr/lib/drivers/pdiagex** kernel extension.

The following topics are discussed in detail:

- Overview
- Device Configuration
- Loading PDIAGEX
- Second-Level Interrupt Handlers
- Programming Interfaces for libpdiag.a
- Programming Interfaces for PDIAGEX
- Data Dictionary

Overview

The Portable Diagnostic Kernel Extension (**PDIAGEX**) is designed to allow a user-level application to exercise or test a device without requiring specialized diagnostic code to be added to the device driver. **PDIAGEX** is loaded and bound into the kernel by the Diagnostic Controller before the application is invoked.

PDIAGEX provides system calls for reading and writing device registers, performing Direct Memory Access (DMA), and handling interrupts.

To use **PDIAGEX** for exercising a device, make the device unavailable to the rest of the system by invoking device methods to move the device from the DEFINED or AVAILABLE state to the DIAGNOSE state. Once the device is in the DIAGNOSE state, the device may be exercised using **PDIAGEX.** This is accomplished by using the **libpdiag.a call** *pdiag_diagnose_state*.

Applications using **PDIAGEX** must be linked with the **pdiagex.exp** file specified as an import file.

Device Configuration

Using **PDIAGEX** requires that serialization be used to limit access to the adapters by the diagnostics and the normal device drivers. Serialization is provided by the device configuration software.

A device state, DIAGNOSE, is defined. The state is identified by **state=4** in the **CuDv** object for the device. A define statement:

```
#define  DIAGNOSE   4
```

has been added to the **/usr/include/sys/cfgdb.h** file.

This state can be entered only from the DEFINED state and only by running the **/usr/lib/methods/cfgdiag** method. From the DIAGNOSE state, a device can be changed back to the DEFINED state only by running the **/usr/lib/methods/ucfgdiag** method. Transitions between the AVAILABLE and DIAGNOSE states are not allowed. This provides a mechanism for serializing access to the devices that support this DIAGNOSE state. While in the AVAILABLE state, a device's normal device driver is loaded and operational, but while it is in the DIAGNOSE state, the **PDIAGEX** (or separate diagnostic device driver) is loaded and has control of the device.

The **/usr/lib/methods/cfgdiag** method checks that the parent of the device is in the correct state. If the device is a Micro Channel adapter, it verifies that the adapter is in the slot. **Busresolve** then runs to ensure that bus resources are allocated properly.

Two diagnostic library routines have been created to move the device and its children to their appropriate states for testing. The routines are **pdiag_diagnose_state** and **pdiag_restore_state**.

Loading PDIAGEX

The Diagnostic Controller coordinates the loading and unloading of the kernel extensions required before executing the Diagnostic Application. The **KernExt** field in the PDiagRes and PDiagTask object class is used to tell the Controller that the device requires a kernel extension. This is a ',' comma-separated list of required kernel extensions for the application. Each kernel extension is loaded before the application is invoked.

Second-Level Interrupt Handlers

All second-level interrupt handlers should reside in the directory **/usr/lpp/diagnostics/slih**. This directory is defined by environment variable **DIAGX_SLIH_DIR**. Avoid code names at all times. Use the component name if applicable.

Programming Interfaces for libpdiag.a

This section provides information on application programming interfaces to the Portable Diagnostic library.

- pdiag_diagnose_state
- pdiag_restore_state
- pdiag_cs_open
- pdiag_cs_close
- pdiag_cs_get_attr
- pdiag_cs_free_attr
- pdiag_open
- pdiag_close
- pdiag_pcicfg_read
- pdiag_pcicfg_write
- pdiag_set_eeh_option
- pdiag_read_slot_reset
- pdiag_set_slot_reset
- pdiag_eeh_errinjct

pdiag_diagnose_state

Purpose

Puts the device under test into the correct state for testing.

Syntax

```
#include <sys/pdiag_def.h>

int32  pdiag_diagnose_state ( char *device_instance )
```

Description

The **pdiag_diagnose_state** subroutine unconfigures the device, and its children if necessary, to set the device into the DIAGNOSE state. Original states of all devices changed will be saved. Use **pdiag_restore_state** to put the changed devices back to their original states.

This function is platform-implementation specific. Its main purpose is to make sure that the target device is in the correct state for diagnostic purposes. If the device is already in a diagnostic state, or any state allowed by the operating system for this purpose, then this function should return successful status. If an error occurs, then this function should return a non-zero.

The global variable **diag_cfg_errno** will be set to the return value of the method invoked for the device.

Parameters

device_instance	Name of the device under test.

Return Value

The **pdiag_diagnose_state** subroutine returns one of the following values:

0	Successful return.
-1	Software error.
1	Error putting device in diagnose state.

pdiag_restore_state

Purpose

Restores resource and children to their initial state before testing.

Syntax

```
#include    <sys/pdiag_def.h>

int32  pdiag_restore_state ( char *device_instance )
```

Description

The **pdiag_restore_state** subroutine puts the device, and its children if necessary, back to the original state before the **pdiag_diagnose_state** routine was called.

This function is platform-implementation specific. Its main purpose is to make sure that the target device is back in its original state before diagnostic functions were performed on the device. If the device is already in the correct state, then this function should return successful status. If an error occurs, then this function should return non-zero.

Parameters

device_instance	Name of the device under test.

Return Value

The **pdiag_restore_state** subroutine returns one of the following values:

0	Successful return.
-1	Software error.
1	Error restoring device to initial state.

pdiag_cs_open

Purpose

Opens and initializes the configuration services, which are used to obtain device information. This is the Object Data Manager (ODM).

Syntax

```
int32 pdiag_cs_open ( )
```

Description

The **pdiag_cs_open** subroutine issues an **odm_initialize** call to the Object Data Manager.

Parameters

Takes no parameters.

Return Value

A value of 0 is always returned.

pdiag_cs_close

Purpose

Closes the configuration services, which are used to obtain device information. This is the Object Data Manager (ODM).

Syntax

```
int32 pdiag_cs_close ( )
```

Description

The **pdiag_cs_close** subroutine issues an **odm_terminate** call to the Object Data Manager.

Parameters

Takes no parameters.

Return Value

A value of 0 is always returned.

pdiag_cs_get_attr

Purpose

Returns resource attribute information.

Syntax

```
int32 pdiag_cs_get_attr ( char *device_instance, char *attribute,
                char **cvalue, char *type )
```

Description

The **pdiag_cs_get_attr** subroutine searches the data configuration database to obtain the value of the attribute for the device. The value and type is returned to the calling application.

Parameters

device_instance
: Name of the device under test.

attribute
: Character string describing attribute to be retrieved. Supported device attribute names:

- alias
- alt_addr
- attn_mac
- beacon_mac
- bus_addr_start
- bus_id
- bus_intr_lvl
- bus_io_addr
- bus_io_length
- bus_mem_addr
- bus_mem_start
- bus_type
- dma1_start
- dma2_start
- dma3_start
- dma4_start
- dma_bus_mem
- dma_channel
- dma_lvl
- gd_frequency
- int_level
- intr_priority
- rcv_que_size
- ring_speed
- use_alt_addr
- vram_start
- xmt_que_size

cvalue Pointer to data buffer, set by this function to address of buffer allocated to hold the attribute data.

type Character set by this function to indicate the returned data type. Supported data types are:

s String

i Long integer

Return Value
A value of 0 is returned if successful.

pdiag_cs_free_attr

Purpose
Frees a buffer allocated by a **pdiag_cs_get_attr** request.

Syntax
```
int32 pdiag_cs_free_attr ( char *cvalue )
```

Description
The **pdiag_cs_free_attr** subroutine frees the buffer allocated by a previous **pdiag_cs_get_attr** call.

Parameters
cvalue Pointer to previously allocated data buffer.

Return Value
A value of 0 is returned if successful.

pdiag_open

Purpose
Prepares a resource for testing.

Syntax
```
#include <sys/pdiagex_dds.h>
#include <sys/pdiag_def.h>

int32 pdiag_open( device_instance, dds_ptr, int_handler, handle )

pdiag_addr_t         device_instance;
pdiagex_dds_t       *dds_ptr;
pdiag_addr_t         int_handler;
pdiag_info_handle_t *handle;
```

Description
The **pdiag_open**() function allocates memory for a handle for this particular resource. The *pdiagex_dds_t* structure contains information about the resource to be tested. The Test Unit code must initialize the data in this structure before calling **pdiag_open**. The returned *pdiag_info_handle_t* structure is the handle created for the resource. The Test Unit does not need to know any of the internal details of this structure, but must retain the pointer for use in subsequent function calls. The DMA channel is initialized by calling the **d_init** kernel service and then the DMA channel is unmasked for transfer; that is, you are not required to do a **pdiag_dd_dma_setup**(). For Micro Channel bus_types, it also initializes a DMA TCW management table to indicate that all buffers are available.

If a user interrupt-handler routine exists, it pins the handler, initializes this handler (using the **i_init** kernel service), and allocates memory for interrupt data.

Both this routine and pdiag_close() share a common lock while executing to prevent simultaneous resource allocation/deallocation. If a call is made to this routine or **pdiag_close**() while the lock is being held by a previous call, the calling process will sleep until the routine is available.

Note: In some instances, the members of the dds structure may not be necessary. For example, if dds->bus_type is equal to BUS_60X, the dds members, **bus_io_addr, bus_io_length, dma_bus_addr, dma_bus_length, dma_lvl, dma_flags**, and **dma_chan_id** are not used and are ignored by **PDIAGEX**. See “Programming Interfaces for libpdiag.a” on page 65.

Execution Environment

The **pdiag_open**() function can be called from the process environment only.

Parameters

device_instance	Pointer to the string name of the specific device to open.
dds_ptr	Points to a **pdiagex_dds_t** structure which should already be initialized with attributes for the particular resource described by the dds (see ″ Data Dictionary″).
int_handler	Pointer to the string name of the interrupt handler to be loaded.
handle	Returned pointer to diagnostic resource handle.

Return Value

The **pdiag_open** function returns one of the following values:

DGX_OK	The operation was successful. The errno is not set.
DGX_BOUND_FAIL	An input parameter is out of bounds (**dds.dma_bus_len** is not a multiple of PAGESIZE or zero) (Micro Channel bus type only). The errno is not set.
DGX_BADVAL_FAIL	An input parameter (**dds.bus_type**) is not valid. The errno is not set.
DGX_INVALID_HANDLE	Specified handle pointer is not valid. The errno is set to the **suword()** return code.
DGX_COPYDDS_FAIL	Application could not copy the dds information. The errno is set to the **copyin()/copyout()** return code.
DGX_DINIT_FAIL	Application could not initialize the DMA channel. The errno is set to the **d_init()** return code.
DGX_IINIT_FAIL	Application could not initialize the user’s interrupt handler. The errno is set to the **i_init()** return code.
DGX_KMOD_FAIL	Application could not locate the user’s interrupt handler in kernel space. The errno is set to the **kmod_entrypt()** return code.
DGX_PINCODE_FAIL	Application could not pin the user’s interrupt handler or the interrupt environment **PDIAGEX** functions. The errno is set to the **pincode()** return code.
DGX_PINU_FAIL	Application could not pin the specified user buffer. The errno is set to the **pinu()** return code.
DGX_XMALLOC_FAIL	Application could not allocate resources. The errno is set to the **xmalloc()** return code.
DGX_XMATTACH_FAIL	Application could not attach user buffer to the physical address. The errno is set to the **xmattach()** return code.

Related Information

pdiag_close() function.

pdiag_close

Purpose

Frees up **PDIAGEX** Kernel Extension resources.

Syntax

```
#include <sys/pdiagex_dds.h>
#include <sy/pdiag_def.h>

int pdiag_close( handle )
pdiag_info_handle_t  handle;
```

Description

pdiag_close() frees the DMA and interrupt channels, if they were initialized. This function also masks the DMA channel; that is, you are not required to do a **pdiag_dd_dma_complete**(). Any memory that was allocated, pinned, or cross-memory attached is detached, unpinned, and freed appropriately.

If this is the last use of the user's interrupt-handler routine, it is unloaded from kernel memory.

Both this routine and **pdiag_open**() share a common lock while executing to prevent simultaneous resource allocation and deallocation. If a call is made to this routine or **pdiag_open**() while the lock is being held by a previous call, the calling process will sleep until the routine is available.

> **Note:** All **pdiag_dd_dma_setup**()) calls should be matched with a **pdiag_dd_dma_complete**() call prior to calling this routine. Any outstanding DMA operations will result in the failure of this routine.

Execution Environment

The **pdiag_close**() function can be called from the process environment only.

Parameters

handle Pointer to **pdiag_info_handle_t** structure which is returned from **pdiag_open**().

Return Value

The **pdiag_close** function returns one of the following values:

DGX_OK	The operation was successful. The errno is not set.
DGX_INVALID_HANDLE	Specified handle has been closed or was not generated by the (**pdiag_open**) call. The errno is not set.
DGX_OUTSTANDINGDMA_FAIL	An outstanding DMA operation is preventing closure. The errno is not set.

Related Information

pdiag_open subroutine.

pdiag_pcicfg_read

Purpose

Reads a PCI Configuration register.

Syntax

```
#include <sys/pdiagex_dds.h>
#include <sys/pdiag_def.h>

int32 pdiag_pcicfg_read( device_instance, reg_offset, datasize, data )

pdiag_addr_t         device_instance;
ulong                reg_offset;
int                  datasize;
uchar                *data;
```

Description

The **pdiag_pcicfg_read**() function reads 8, 16, or 32 bits of a PCI Configuration register for this particular resource. The *reg_offset* parameter contains the register offset into the device's PCI configuration table. The calling application must provide a valid register offset before calling **pdiag_pcicfg_read**. The returned *data* is the 8, 16, or 32 bit value read from the PCI register configuration table. All the byte swapping required is performed internally by this function; the calling application must not alter the byte positioning of the data.

Execution Environment

The **pdiag_pcicfg_read**() function can be called from the process environment only.

Parameters

device_instance — Pointer to the string name of the specific device to read.

reg_offset — Contains the offset within the PCI configuration table register to be read.

datasize — The data size will be specified as follows:

Size	Type
8 bits	IOCHAR8
16 bits	IOSHORT16
32 bits	IOLONG32

data — Pointer to the data to be read within the PCI Configuration Table.

Note: The value read is the specified size on the datasize parameter.

Return Value

The **pdiag_pcicfg_read** function returns one of the following values:

0 — Successful return

-1 — Software error

Related Information

The **pdiag_pcicfg_write**() function.

pdiag_pcicfg_write

Purpose

Writes to a PCI Configuration register.

Syntax

```
#include <sys/pdiagex_dds.h>
#include <sys/pdiag_def.h>

int32 pdiag_pcicfg_write( device_instance, reg_offset, datasize, data )

pdiag_addr_t        device_instance;
ulong               reg_offset;
int                 datasize;
uchar               data;
```

Description

The **pdiag_pcicfg_write**() function writes 8, 16, or 32 bits to a PCI Configuration register for this particular resource. The *reg_offset* parameter contains the register offset into the device's PCI configuration table. The Test Unit code must provide a valid register offset when calling **pdiag_pcicfg_write**. The *data* value is the 8, 16, or 32 bit value to be written to the PCI register configuration table depending on the data size specified in the datasize parameter. All the byte swapping required is performed internally by this function; the calling application must not alter the byte positioning of the data.

Execution Environment

The **pdiag_pcicfg_write**() function can be called from the process environment only.

Parameters

device_instance
: Pointer to the string name of the specific device to write.

reg_offset
: Contains the offset within the PCI configuration table register to be written.

datasize
: The data size will be specified as follows:

Size	Type
8 bits	IOCHAR8
16 bits	IOSHORT16
32 bits	IOLONG32

data
: Contains the value to be written to a specific PCI register.

Note: The size of the value must be specified in the datasize parameter and must be IOCHAR8, IOSHORT16, or IOLONG32.

Return Value

The **pdiag_pcicfg_write** function returns one of the following values:

0	Successful return
-1	Software error

Related Information

The **pdiag_pcicfg_read** subroutine.

Programming Interfaces for PDIAGEX

This section provides information on application programming interfaces to the Portable Diagnostic Kernel Extension PDIAGEX.

Test unit developers should use these interfaces to ensure their code has maximum portability across platforms.

- pdiag_dd_watch_for_interrupt
- pdiag_dd_interrupt_notify
- pdiag_dd_write
- pdiag_dd_read
- pdiag_dd_dma_setup
- pdiag_dd_dma_complete
- pdiag_dd_dma_enable

pdiag_dd_watch_for_interrupt

Purpose

The **pdiag_dd_watch_for_interrupt**() function sleeps until a desired interrupt condition occurs, or a time-out occurs if the interrupt does not occur within the specified time.

Syntax

```
#include <sys/pdiagex_dds.h>
int32 pdiag_dd_watch_for_interrupt( handle, flag_mask, timeout_sec )
pdiag_info_handle_t  handle;
uint32 flag_mask;
uint32 timeout_sec;
```

Description

pdiag_dd_watch_for_interrupt() sleeps until a desired interrupt condition occurs or *timeout_sec* seconds pass. If the interrupt condition occurs before the routine is called, the function simply returns, without sleeping. To be awakened from the sleep state and get interrupt condition information, this routine is highly dependent on the interaction of the application's interrupt handler. This interaction is maintained by using the *handle.flag_word, handle.sleep_word*, and *handle.sleep_flag*.

The application's interrupt handler should update the *handle.flag_word* each time it receives an interrupt. The *handle.flag_word* and *flag_mask* format is determined by the application. The application's interrupt handler should also test the *handle.sleep_flag* each time it receives an interrupt to determine if the **pdiag_dd_watch_for_interrupt**() routine is sleeping. If *handle.sleep_flag* is TRUE, the application's interrupt handler should wake the **pdiag_dd_watch_for_interrupt**() routine using the **pdiag_dd_interrupt_notify**() service with *handle.sleep_word* as the sleep word.

Execution Environment

The **pdiag_dd_watch_for_interrupt**() function can be called from the process environment.

Parameters

handle	Points to **pdiag_info_handle_t** structure which is returned from **pdiag_open**().
flag_mask	32-bit flag mask which, when bitwise ANDed with the *handle.flag_word*, produces a nonzero result only when the *handle.flag_word* identifies the desired interrupt condition.
timeout_sec	Number of seconds to watch for the interrupt condition before timing out. (A value of zero will never time-out; possible hang condition).

Return Value

The **pdiag_dd_watch_for_interrupt** function returns one of the following values:

DGX_OK	The operation was successful. The errno is not set.
DGX_FAIL	The interrupt condition did not occur before *timeout_sec* seconds passed.
DGX_INVALID_HANDLE	Specified handle has been closed or was not generated by the **pdiag_open**() call. The errno is not set.

pdiag_dd_interrupt_notify

Purpose

The **pdiag_dd_interrupt_notify**() function can only be used by the interrupt handling function of the TU library. This function notifies a pending **pdiag_dd_watch_for_interrupt** call that an interrupt has been processed.

Syntax

```
#include <sys/pdiagex_dds.h>
int32 pdiag_dd_interrupt_notify( sleep_word )
uint32 sleep_word;
```

Description

pdiag_dd_interrupt_notify() is used to notify a previously pending call to **pdiag_dd_watch_for_interrupt** that an expected interrupt has been received and processed. This call is only used by the second-level interrupt-handler code provided in the TU library.

Execution Environment

The **pdiag_dd_interrupt_notify**() function can only be called from the interrupt environment.

Parameters

sleep_word	Semaphore handle that TU is waiting on, passed in as a parameter to the interrupt handler.

Return Value

The **pdiag_dd_interrupt_notify** function returns one of the following values:

DGX_OK	The operation was successful. The errno is not set.

pdiag_dd_write, pdiag_dd_write_64

Note: pdiag_dd_write_64 is only used in 64-bit kernel.

Purpose

The **pdiag_dd_write**() and the **pdiag_dd_write_64**() functions perform write operations on a resource.

Syntax for 32-Bit Kernel

```
#include <sys/pdiagex_dds.h>
int32 pdiag_dd_write( handle, type, offset, data, flags )
pdiag_info_handle_t  handle;
uint32  type;
uint32  offset;
pdiag_addr_t  data;
pdiagex_opflags_t  *flags;
```

Syntax for 64-Bit Kernel

```
#include <sys/pdiagex_dds.h>
int32 pdiag_dd_write_64( handle, type, offset, data, flags )
pdiag_info_handle_t  handle;
uint32  type;
uint32  offset;
pdiag_addr_t  data;
pdiagex_opflags_t  *flags;
```

Description

The **pdiag_dd_write**() and the **pdiag_dd_write_64**() functions write the specified data to the specified offset address. If the user enables the *times* variable, timing information for this function is also returned. Each write performed is dependent on the *memio* operation and *count* parameters.

PDIAG_IO_OP	If *count* is1, data is written to the specified bus I/O offset address.
PDIAG_MEM_OP	If *count* is1, data is written to the specified memory offset address.
PDIAG_POS_OP	If *count* is1, data is written to the specified POS offset address.

A specified number of write accesses to the offset address may be performed if *count* is greater than 1. The user may choose to write the data to one location (the offset address) *count* times, or write the data to *count* consecutive locations, starting at the offset address. In either case, the data to be written is supplied by consecutive locations of the *data* buffer starting at the specified buffer address.

> **Note:** When writing data, it is imperative that the write data buffer is at least the size of *count* * *type* (unless the write data buffer address is not being incremented) and filled with valid data for each write operation to be performed. If this is not done, meaningless data is written to the designated area. This may cause problems with your testing.

Execution Environment

The **pdiag_dd_write**() function can be called from the process or the interrupt environment. The **pdiag_dd_write_64**() function can only be called from the interrupt environment.

Parameters

handle	Points to **pdiag_info_handle_t** structure which is returned from **pdiag_open**().
type	Defines the data length (byte, word or long) read from the address specified when type is IOCHAR8, IOSHORT16, and IOLONG32 respectively.

offset
: Offset value that is dependent on the type of operation being performed. It can be one of the following values:

PDIAG_IO_OP
    Offset from base I/O address.

PDIAG_MEM_OP
    Offset from base memory address.

PDIAG_POS_OP
    offset from base POS address.

data
: Pointer to a block of information to be written to the specified address. This block will be of size:

count **for** *type* **IOCHAR8**
    (1 if not incrementing *data*)

    OR

count ***2 for** *type* **IOSHORT16**
    (2 if not incrementing *data*)

    OR

count ***4 for** *type* **IOLONG32**
    (4 if not incrementing *data*).

flags
: The *flags* structure contains the following members:

memio
: Indication of the type of read operation to perform.

PDIAG_IO_OP
    For I/O write operations.

PDIAG_MEM_OP
    For memory write operations.

PDIAG_POS_OP
    For I/O Configuration Space write operations.

count
: Number of accesses to perform.

PDIAG_IO_OP
    Number of write operations to be performed.

PDIAG_MEM_OP
    Number of times *data* is written.

PDIAG_POS_OP
    Count should be set to 1.

addr_incr_flag
: Determines whether the *data* buffer address and the *offset* address get incremented on each of *count* accesses:

PDIAG_SING_LOC_ACC
    Single-location accesses: neither address is incremented.

PDIAG_SING_LOC_BUF
    Single-location access for buffer: the *data* address is never incremented. The address referred to by *offset* is incremented by *type.*

PDIAG_SING_LOC_HW
    Single-location access for hardware: the *data* address is incremented by *type*. The address referred to by *offset* is not incremented.

PDIAG_MULT_LOC_ACC
    Multiple-location accesses: both addresses are incremented by *type.*

intrlev Indicates which environment the calling routine is in:

PROCLEV
If calling from the process level.

INTRKMEM
If calling from the interrupt level and the *data* buffer is in kernel memory.

Note: For the **pdiag_dd_write** function, the *intrlev* parameter may be set to either PROCLEV or INTRKMEM. For the **pdiag_dd_write_64** function, the *intrlev* parameter must always be set to INTRKMEM.

times Points to the **timestruc_t** structure which returns timing information. If *times* is a null pointer, no timing information will be returned back to the user.

Return Value

The **pdiag_dd_write** and the **pdiag_dd_write_64** functions return one of the following values:

DGX_OK	The operation was successful. The errno is not set.
DGX_INVALID_HANDLE	Specified handle has been closed or was not generated by the **pdiag_open**() call. The errno is not set.
DGX_BOUND_FAIL	*offset* given was larger than the width of the I/O address range. The errno is not set.
DGX_BADVAL_FAIL	Type field was not valid (that is, not IOCHAR8, IOSHORT16, or IOLONG32). The errno is not set.
DGX_FAIL	Error occurred during the I/O write access. The errno is set to BUS_PUT(L/S/C)X macro return code.
DGX_COPY_FAIL	User *data* buffer could not be copied to or from kernel memory. The errno is set to the xmemin/out or copyin/out return code.

Related Information

pdiag_dd_read, pdiag_dd_read_64 function.

pdiag_dd_read, pdiag_dd_read_64

Note: pdiag_dd_read_64 is only used in the 64-bit kernel.

Purpose

The **pdiag_dd_read**() and the **pdiag_dd_read_64**() functions perform read operations on a resource.

Syntax for 32-Bit Kernel

```
#include <sys/pdiagex_dds.h>
int32 pdiag_dd_read ( handle, type, offset, data, flags )
pdiag_info_handle_t  handle;
uint32  type;
uint32  offset;
pdiag_addr_t  data;
pdiagex_opflags_t  *flags;
```

Syntax for 64-Bit Kernel

```
#include <sys/pdiagex_dds.h>
int32 pdiag_dd_read_64 ( handle, type, offset, data, flags )
pdiag_info_handle_t  handle;
uint32  type;
uint32  offset;
pdiag_addr_t  data;
pdiagex_opflags_t  *flags;
```

Description

The **pdiag_dd_read**() and the **pdiag_dd_read_64**() functions read the data from the specified address. If the user enables the *times* variable, timing information for this function is also returned. Each read

performed is dependent on the *memio* operation and *count* parameters.

PDIAG_IO_OP	If count is 1, data is read from the specified bus I/O offset address.
PDIAG_MEM_OP	If count is 1, data is read from the specified memory offset address.
PDIAG_POS_OP	If count is 1, data is read from the specified POS offset address.

A specified number of read accesses from the offset address may be performed if *count* is greater than 1. The user may choose to read the data from one location (the offset address) *count* times, or read the data from *count* consecutive locations, starting at the offset address. In either case, the read data is stored in the *data* buffer starting at the specified buffer address.

> **Note:** When reading data, it is imperative that the read data buffer is at least the size of *count* * *type* (unless the read data buffer address is not being incremented). If this is not done, meaningless data is written to an area outside the read buffer. This may cause problems with your testing.

Execution Environment

The **pdiag_dd_read**() function can be called from the process or the interrupt environment. The **pdiag_dd_read_64**() function can only be called from the interrupt environment.

Parameters

handle
: Points to **pdiag_info_handle_t** structure which is returned from **pdiag_open**().

type
: Defines the data length (byte, word or long) read from the address specified when type is IOCHAR8, IOSHORT16, and IOLONG32 respectively.

offset
: Offset value that is dependent on the type of operation being performed. It can be one of the following values:

  PDIAG_IO_OP
  offset from base I/O address.

  PDIAG_MEM_OP
  offset from base memory address.

  PDIAG_POS_OP
  offset from base POS address.

data
: Address of the information read from the specified address.

  > **Note:** For PDIAG_IO_OP and PDIAG_MEM_OP: The value read from the specified *offset* will be placed at the specified *data* address in the form specified by *type*. If the *data* buffer is smaller than the specified *type*, the value will overwrite the bounds of your buffer. If the *data* buffer is larger than the specified *type*, the value will reside in the upper *type* bytes of the buffer.
  >
  > For PDIAG_POS_OP: The value read from the specified *offset* will be placed at the specified *data* address and will occupy 1 byte. If the *data* buffer is larger than 1 byte, the value will reside in the upper byte of the buffer.

flags
: The *flags* structure contains the following members:

memio
: Indication of the type of read operation to perform.

  PDIAG_IO_OP
  For I/O read operations.

  PDIAG_MEM_OP
  For memory read operations..

  PDIAG_POS_OP
  For I/O Configuration Space read operations.

count
: Number of accesses to perform.

  PDIAG_IO_OP
  : Number of read operations to be performed.

  PDIAG_MEM_OP
  : Number of times *data* is read.

  PDIAG_POS_OP
  : Count should be set to 1.

addr_incr_flag
: Determines whether the *data* buffer address and the *offset* address get incremented on each of *count* accesses:

  PDIAG_SING_LOC_ACC
  : Single-location accesses: neither address is incremented.

  PDIAG_SING_LOC_BUF
  : Single-location access for buffer: the *data* address is never incremented. The address referred to by *offset* is incremented by *type*.

  PDIAG_SING_LOC_HW
  : Single-location access for hardware: the *data* address is incremented by *type*. The address referred to by *offset* is not incremented.

  PDIAG_MULT_LOC_ACC
  : Multiple-location accesses: both addresses are incremented by *type*.

intrlev
: Indicates which environment the calling routine is in:

  PROCLEV
  : If calling from the process level

  INTRKMEM
  : If calling from the interrupt level and the *data* buffer is in kernel memory.

  Note: For the **pdiag_dd_read** function, the *intrlev* parameter may be set to either PROCLEV or INTRKMEM. For the **pdiag_dd_read_64** function, the *intrlev* parameter must always be set to INTRKMEM.

times
: Points to the **timestruc_t** structure which returns timing information. If *times* is a null pointer, no timing information will be returned back to the user.

Return Value

The **pdiag_dd_read** and the **pdiag_dd_read_64** functions return one of the following values:

DGX_OK	The operation was successful. The errno is not set.
DGX_INVALID_HANDLE	Specified handle has been closed or was not generated by the **pdiag_open**() call. The errno is not set.
DGX_BOUND_FAIL	*offset* given was larger than the width of the I/O address range. The errno is not set.
DGX_BADVAL_FAIL	Type field was not valid (that is, not IOCHAR, IOSHORT, or IOLONG). The errno is not set.
DGX_FAIL	Error occurred during the I/O read access. The errno is set to BUS_GET(L/S/C)X macro return code.
DGX_COPY_FAIL	User *data* buffer could not be copied to or from kernel memory. The errno is set to the xmemin/out or copyin/out return code.

Related Information

pdiag_dd_write, pdiag_dd_write_64 function.

pdiag_dd_dma_setup

Purpose

The **pdiag_dd_dma_setup**() function initializes, pins, and cross-memory attaches the user buffer for a DMA operation.

Syntax

```
#include <sys/pdiagex_dds.h>
#include <sys/dma.h>
int32 pdiag_dd_dma_setup( handle, dma_flags, baddr, users_daddr, count, minxfer,operation )
pdiag_info_handle_t  handle;
int32 dma_flags;
pdiag_addr_t baddr;
pdiag_addr_t users_daddr;
uint32 count;
uint32 minxfer;
uint32 operation;
```

Description

The following is performed by the **pdiag_dd_dma_setup** depending on the bus type and *operation*:

Where bus type = BUS_MICRO_CHANNEL or BUS_60X and operation is PDIAG_DMA_MASTER

- The DMA master function on Micro Channel and 60X bus sytems pins and cross-memory attaches the user buffer for the length of *count*.

  For Micro Channel bus type adapters, the DMA master function issues the **d_master** kernel call for the specified address and length. The DMA address space is managed for you, and the offset into the DMA buffer is supplied in the *daddr* parameter. For 60X bus type adapters, the DMA master function issues the **xmemdma** kernel call for each page referred to by the specified address and length.

  The flags for this call will be (XMEM_HIDE | XMEM_ACC_CHK). The DMA address space is *not* managed for you, and the offset into the DMA buffer is supplied in the *daddr* parameter.

  Note: The *dds* member, *maxmaster*, must be set to the maximum number of concurrent **pdiag_dd_dma_setup**( )s to be used (that is, maximum number of **pdiag_dd_dma_setup**()s called above the number of associated **pdiag_dd_dma_complete** ()s at any given time). *maxmaster* must be set to at least 1 (one) for this call to pass without a DGX_BOUND_FAIL error.

Where bus type = BUS_BID and operation is PDIAG_DMA_MASTER

- The **pdiag_dd_dma_setup**() function pins and cross-memory attaches the user buffer. The function allows for a transfer of 4k or 1 page. The transfer cannot cross a page boundary. Larger transfers are not allowed at this time.

  This function issues the **d_map_page** kernel call for the specified address. The DMA space is managed for the user, and the offset into the DMA buffer is supplied in the *users_daddr* parameter.

Where bus type = BUS_MICRO_CHANNEL and operation is PDIAG_DMA_SLAVE

- For slave operation on a Micro Channel, the **pdiag_dd_dma_setup**() function issues the **d_slave** kernel call for the specified length. Only one Micro Channel slave DMA may occur at a time.

  Note: The *dds* member, *maxmaster*, must be set to at least 1 (one) for this call to pass without a DGX_BOUND_FAIL error.

Execution Environment

The **pdiag_dd_dma_setup**() function can be called from the process environment only.

Parameters

handle	Points to **pdiag_info_handle_t** structure which is returned from **pdiag_open**().

dma_flags	This flag is ignored for 60X bus type adapters. The following refers only to Micro Channel bus type adapters. Use the DMA_READ flag for transferring data from the adapter to user memory. Use 0 (zero) for transferring data from the system to the adapter. See the header file **sys/dma.h** for more information on other DMA flags. If the user wants to read or modify data before calling **pdiag_dd_dma_complete**(), then DMA_NOHIDE should also be set. This may be useful for devices that set up long-term DMA mapping for purposes of communication (such as command blocks, status blocks, common buffer pools). Then the **pdiag_dd_dma_complete**() does not have to be called each time they want to let the application read/write, and then **pdiag_dd_dma_setup**() again for the next DMA transfer. If DMA_NOHIDE is set and the user wants to read data before calling **pdiag_dd_dma_complete**(), then call the **pdiag_dd_dma_enable**() routine to flush and read the data. If DMA_NOHIDE is set and the user wants to write data before calling **pdiag_dd_dma_complete**(), then after the user modifies the data, call the **pdiag_dd_dma_enable**() routine with a flush operation. Make sure that the adapter will not be transferring data to the same area that the user is manipulating.
baddr	Points to user's read or write buffer where DMA transfer should take place.
users_daddr	Points to an integer to be filled with the physical memory address of *baddr* upon successful completion of this call.
count	Number of bytes to be transferred.
minxfer	Minimum transfer length that the device will handle. (Slave transfer only on BUS_BID).
operation	Type of operation to perform: **PDIAG_DMA_MASTER** **PDIAG_DMA_SLAVE**

Return Value

The **pdiag_dd_dma_setup** function returns one of the following values:

DGX_OK	The operation was successful. The errno is not set.
DGX_INVALID_HANDLE	Specified handle has been closed or was not generated by the **pdiag_open**() call. The errno is not set.
DGX_BOUND_FAIL	Application tried to setup a DMA outside its resources, the resources are currently unavailable, or the *dds* member *dma_bus_length* (Micro Channel only) or *maxmaster* is set to zero. The errno is not set.
DGX_BADVAL_FAIL	**PDIAGEX** was unable to update the specified *daddr*. The errno is set to the suword() return code.
DGX_PINU_FAIL	Application could not pin the specified user buffer. The errno is set to the pinu() return code.
DGX_XMATTACH_FAIL	Application could not attach user buffer to the physical address. The errno is set to the xmattach() return code.

Related Information

The **pdiag_dd_dma_enable** and **pdiag_dd_dma_complete** subroutines.

pdiag_dd_dma_complete

Purpose

The **pdiag_dd_dma_complete**() function unpins and detaches the user space DMA buffer. If the handle's *dds.bus_type* is set for the Micro Channel, this function also calls the **d_complete**() kernel service, which checks for detected IOCC errors, flushes the IOCC buffer (unhides it if necessary) and sets the page table 'modified' bit if the information was modified.

Syntax

```
#include <sys/pdiagex_dds.h>
int32 pdiag_dd_dma_complete( handle, daddr, operation )
pdiag_info_handle_t  handle;
pdiag_addr_t  daddr;
uint32 operation;
```

Description

The following is performed by the **pdiag_dd_dma_complete** depending on the bus type and *operation*:

Where bus type = BUS_MICRO_CHANNEL and operation is PDIAG_DMA_MASTER or PDIAG_DMA_SLAVE

- The **pdiag_dd_dma_complete**() function cleans up after the DMA transfer. First, the specified *daddr* is used to retrieve the *baddr*, *count*, and *dma_flags* specified in the corresponding **pdiag_dd_dma_setup**() calls. **pdiag_dd_dma_complete**() then issues the **d_complete** kernel call using these parameters. The user address space used for the DMA transfer is then unpinned, detached, and made available for another DMA transfer.

  Where bus type = BUS_BID and operation is PDIAG_DMA_MASTER
- The **pdiag_dd_dma_complete**() should be called after I/O completion involving the area mapped by the prior **pdiag_dd_dma_setup**() function call.

  This function utilizes the **D_UNMAP_PAGE** macro to unmap the specified address.

  Where bus type = BUS_BID and operation is PDIAG_DMA_SLAVE
- The **pdiag_dd_dma_complete**() should be called after I/O completion involving the area mapped by the prior **pdiag_dd_dma_setup**() function call.

  This function utilizes the **D_UNMAP_SLAVE** macro to unmap the specified address.

Execution Environment

The **pdiag_dd_dma_complete**() function can be called from the process or the interrupt environment on a BUS_MICRO_CHANNEL system. The function can only be called from the process environment on a BUS_BID system.

Parameters

handle	Points to **pdiag_info_handle_t** structure which is returned from **pdiag_open**().
daddr	The offset into the user's physical DMA address. This is returned by **pdiag_dd_dma_setup** () routine. For DMA slave completes, this should be set to 0.
operation	Type of operation to perform: **PDIAG_DMA_MASTER** **PDIAG_DMA_SLAVE**

Return Value

The **pdiag_dd_dma_complete** function returns one of the following values:

DGX_OK	The operation was successful. The errno is not set.
DGX_INVALID_HANDLE	Specified handle has been closed or was not generated by the **pdiag_open**() call. The errno is not set.
DGX_BADVAL_FAIL	daddr value was not valid. The errno is not set.
DGX_DCOMPLETE_FAIL	Application received a DMA error detected by the system hardware. The errno is set to the d_complete() return code.
DGX_UNPINU_FAIL	Application could not unpin the specified user buffer. The errno is set to the unpinu() return code.
DGX_XMDETACH_FAIL	Application could not detach user space from the physical address. The errno is set to the xmdetach() return code.

Related Information

pdiag_dd_dma_setup() and **pdiag_dd_dma_enable**() functions.

pdiag_dd_dma_enable

Purpose

The **pdiag_dd_dma_enable**() function enables and disables a DMA operation. The actual function performed depends on the bus type and operation requested.

Syntax

```
#include <sys/pdiagex_dds.h>
int32 pdiag_dd_dma_enable( handle, daddr, operation )
pdiag_info_handle_t>  handle;
pdiag_addr_t  daddr;
uint32 operation;
```

Description

Where bus type = BUS_MICRO_CHANNEL and operation is PDIAG_DMA_FLUSH

- The PDIAG_DMA_FLUSH operation uses the specified *daddr* to retrieve the *baddr* and count specified in the corresponding **pdiag_dd_dma_setup**() call. Then the **d_cflush** and **d_bflush** kernel routines are called to do the processor cache and IOCC buffer flushes, respectively.

  If users need to change data in the DMA address space, they first change the data in their user space and then call **pdiag_dd_dma_enable**() with a PDIAG_DMA_FLUSH operation. If they need to read data in the DMA address space, they first call **pdiag_dd_dma_enable** () with a PDIAG_DMA_FLUSH operation, and then reads the data in the user space.
- The PDIAG_DMA_FLUSH operation flushes the processor cache and the IOCC buffer. This may be used if a user is required to look at or change the DMA area after a **pdiag_dd_dma_setup**() routine. This routine works only if **pdiag_dd_dma_setup**() is called with **dma_flags = DMA_NOHIDE**.

  This routine is required only if the user wants to read the data before doing **pdiag_dd_dma_complete**().

Where bus type = BUS_MICRO_CHANNEL or BUS_BID and operation is PDIAG_DMA_DISABLE

- The DMA channel for that handle is disabled.

Where bus type = BUS_MICRO_CHANNEL or BUS_BID and operation is PDIAG_DMA_ENABLE

- The DMA channel for that handle is enabled.

Execution Environment

The **pdiag_dd_dma_enable**() function can be called from the process or the interrupt environment on a BUS_MICRO_CHANNEL system. The function can only be called from the process environment on a BUS_BID system.

Parameters

handle	Points to **pdiag_info_handle_t** structure which is returned from **pdiag_open** ().
daddr	Pointer to the user's physical DMA address. This is returned by **pdiag_dd_dma_setup**() routine.
operation	Type of operation to perform: PDIAG_DMA_ENABLE PDIAG_DMA_DISABLE PDIAG_DMA_FLUSH

Return Value

The **pdiag_dd_dma_enable** function returns one of the following values:

DGX_OK	The operation was successful. The errno is not set.
DGX_INVALID_HANDLE	Specified handle has been closed or was not generated by the **pdiag_open**() call. The errno is not set.
DGX_BADVAL_FAIL	Specified daddr is not valid. The errno is not set.
DGX_FAIL	Application could not transfer data between the processor and the I/O controller (IOCC) data caches. The errno is set to the d_cflush or d_bflush return code.

Related Information

The **pdiag_dd_dma_setup** and **pdiag_dd_dma_complete** subroutines.

pdiag_eeh_errinjct

Purpose

Injects an error to the PCI slot of the I/O adapters under test.

Syntax

```
#include <sys/pdiagex_dds.h>
#include <sys/pdiag_def.h>

int32  pdiag_eeh_errinjct(char *device_instance, int32 operation_type)
```

Description

The **pdiag_eeh_errinjct** subroutine injects an error to the PCI slot using Run-Time Abstraction Service calls (RTAS). You must implement this function with the Enhanced Error Handling (EEH) option enabled. Run the **open** operation to automatically inject the error. Then run the **close** operation to reactivate the slot and adapter.

Parameters

device_instance
: Name of the device under test.

operation_type
: Integer indicating the function to be performed.

  0: Open Error inject mode
  This option opens and injects the error allowing the error injector to be in control until the close is issued. A busy return indicates that the device has an open error inject by another caller.

  1: Close Error inject mode
  This option closes the error inject.

Return Value

The **pdiag_eeh_errinjct** subroutine returns one of the following values:

0	The operation was successful.
-1	A hardware error occurred.
-2	A software error occurred.

Related Information

The pdiag_set_slot_reset, pdiag_read_slot_reset, and pdiag_set_eeh_option subroutines.

pdiag_read_slot_reset

Purpose

Queries the state of the physical reset signal to the I/O Adapter and the Enhanced Error Handling (EEH) slot's capabilities.

Syntax

```
#include <sys/pdiagex_dds.h>
#include <sys/pdiag_def.h>

int32 pdiag_read_slot_reset( char *device_instance, int32 operation_type )
```

Description

The **pdiag_read_slot_reset** subroutine issues a Run-Time Abstraction service (RTAS) call to query the state of the physical reset signal to the I/O Adapter and the EEH slot's capability.

Parameters

device_instance	Name of the device under test.
operation_type	Integer indicating the function to be performed.

0: Query Reset State
This option returns the slot reset state, indicating if the slot reset is activated or deactivated, and if the I/O adapter is in stopped state or not.

1: Query Slot Capabilities
This option returns the EEH I/O Adapter capabilities, indicating if EEH is supported or not.

Return Value

The **pdiag_read_slot_reset** subroutine returns one of the following values for the **Query Reset State** operation:

Return Code	Description
-2	Software error
-1	Hardware error
0	Reset deactivated and I/O Adapter is not in the EEH stopped state.
1	Reset activated and I/O Adapter is not in the EEH stopped state.
2	I/O Adapter is in the EEH stopped state with the reset signal deactivated and the Load/Store Path is disabled.
3	I/O Adapter is in the EEH stopped state with the reset signal deactivated and the Load/Store Path is enabled.
4	I/O Adapter is permanently unavailable.

The ***pdiag_read_slot_reset*** subroutine returns one of the following values for the **Query Slot Capabilities** operation:

Return Code	Description
-2	Software error.
-1	Hardware error.
0	EEH not supported.
1	EEH supported.

Related Information

The pdiag_set_slot_reset, pdiag_set_eeh_option, and pdiag_eeh_errinjct subroutines.

pdiag_set_eeh_option

Purpose

Enables and disables the Enhanced Error Handling (EEH) option for an I/O Adapter, for systems supporting the EEH option.

Syntax

```
#include <sys/pdiagex_dds.h>
#include <sys/pdiag_def.h>

int32 pdiag_set_eeh_option( char *device_instance, int32 operation_type )
```

Description

The **pdiag_set_eeh_option** subroutine issues Run-Time Abstraction Services (RTAS) calls to enable and disable the EEH option for an I/O Adapter.

Parameters

device_instance
: Name of the device under test.

operation_type
: Integer indicating the function to be performed.
Supported operations:

0: Disable EEH option:
This operation disables the EEH option for the selected I/O Adapter (freeze function is disabled). An error will be reported if the EEH function is not supported.

1: Enable EEH option:
This operation enables the EEH option for the selected I/O Adapter (freeze function enabled). An error will be reported if the EEH function is not supported.

Return Value

The **pdiag_set_eeh_option** subroutine returns one of the following values:

- -2 A software error occurred.
- -1 A hardware error occurred.
- 0 The operation was successful.

Related Information

The pdiag_set_slot_reset, pdiag_read_slot_reset, and pdiag_eeh_errinjct subroutines.

pdiag_set_slot_reset

Purpose

Activates and deactivates the physical reset signal to the I/O adapter for systems supporting the Enhanced Error Handling (EEH) option.

Syntax

```
#include <sys/pdiagex_dds.h>
#include <sys/pdiag_def.h>

int32 pdiag_set_slot_reset( char *device_instance )
```

Description

The **pdiag_set_slot_reset** subroutine resets a single PCI slot by activating and deactivating the slot specific physical reset signal line to the I/O adapter by issuing a Run-Time Abstraction Service (RTAS) call. All required timing parameters will be handled by this subroutine (such as the 100 millisecond minimum reset signal active time for PCI bus).

Parameters

device_instance
: Name of the device under test.

Return Value

The **pdiag_set_slot_reset** subroutine returns one of the following values:

-2 A software error occurred.
-1 A hardware error occurred.
0 The operation was successful.

Related Information

The pdiag_set_eeh_option, pdiag_read_slot_reset, and pdiag_eeh_errinjct subroutines.

Data Dictionary

This section provides information on the data structures and kernel services used by the Diagnostic Kernel Extension PDIAGEX.

- PDIAGEX Data Structures
- Kernel Services
- Programmed I/O Services

PDIAGEX Data Structures

This section describes the data structures used by **PDIAGEX**.

pdiagex_dds_t

The **pdiagex_dds_t** structure defines the device driver structure (dds) for **PDIAGEX**. The **pdiagex_dds_t** structure must be initialized with attributes for the resource before calling **pdiag_open**(). The **pdiagex_dds_t** structure is defined in **/usr/include/sys/pdiagex_dds.h** and contains the following fields:

```
/*----------------------------------------------------------------------*/
/*  PDIAGEX_DDS_T
/*    This structure MUST be filled in by the Calling Application (TU)
/*    This structure is passed to pdiagex in the pdiag_open() routine
/*----------------------------------------------------------------------*/
typedef struct {
   uint32      slot_num;          /* slot number of adapter

   /* BUS DATA  */
   uint32      bus_id;            /* Identifies the I/O bus that the DMA
                                  /* channel is to be allocated on.
   uint32      bus_type;          /* BUS_MICRO_CHANNEL, BUS_60X or BUS_BID
   uint32      bus_io_addr;       /* Base address of Bus I/O area
   uint32      bus_io_length;     /* Length (width) of Bus I/O area
   uint32      bus_mem_addr;      /* Base address of Shared Bus Memory area
   uint32      bus_mem_length;    /* Length (width) of Shared Bus Memory area

   /* DMA  /
   /* Next three are for BUS_MICRO_CHANNEL devices only  */
   uint32      dma_bus_mem;       /* Base address of Bus Memory DMA area
   uint32      dma_bus_length;    /* Length (multiple of PAGESIZE) of BUS
                                  /* Memory DMA area in bytes.
   uint32      dma_lvl;           /* Bus arbitration level

   uint32      maxmaster;         /* maximum number of concurrent
                                  /* dma_master calls
   uint32      dma_flags;         /* DMA flags as defined in sys/dma.h.
                                  /* These flags describe what actions to
                                  /* take ( master/slave, initialize the
                               /* channel, etc. Not used by 60X type devices)

   /* dma_bus_flags is for BUS_BID devices only */
   uint32      dma_bus_flags;     /* Bus flags specific for DMA operation

   uint32      dma_chan_id;       /* For BUS_MICRO_CHANNEL
```

```
                                /* Dma channel ID is returned as a result
                                /* of the DMA initialization.
                                /* For BUS_BID
                                /* Dma channel ID is the assigned DMA
                                /* channel for the device.
                                /* For BUS_60X
                                /* Dma channel ID is not used

   /* Interrupt Handler                            */
   pdiag_addr_t data_ptr;          /* Pointer for passing data to interrupt
   uint32       d_count;           /* Count of bytes of data for interrupt
   uint32       bus_intr_lvl;      /* Interrupt level
   uint32       intr_priority;     /* Interrupt priority
   uint32       intr_flags;        /* Interrupt flags as defined in intr.h

   /* Attribute Expansion Area  */
   pdiag_addr_t attributes;        /* Pointer to specific attributes

 }pdiagex_dds_t;
```

pdiagex_opflags_t

The **pdiagex_opflags_t** structure defines the device operations to be used. The **pdiagex_opflags_t** structure is defined in **/usr/include/sys/pdiagex_dds.h** and consists of the following:

```
/*---------------------------------------------------------------------------*/
/*  PDIAGEX_OPFLAGS_T
/*     This structure MUST be filled in by the Calling Application (TU)
/*     This structure is used for Read and Write Operations
/*---------------------------------------------------------------------------*/
typedef struct {
   uint32       memio;            /* Type of Memory Operation                                           /*PDIAG_M
   uint32       count;            /* Number of accesses to perform
   uint32       addr_incr_flag;   /* Flag that determines whether the data
                                  /* buffer address and/or the offset
                                  /* address gets incremented on each of
                                  /* count accesses.
                                  /* PDIAG_SING_LOC_ACC or
                                  /* PDIAG_SING_LOC_HW   or
                                  /* PDIAG_SING_LOC_BUF  or
                                  /* PDIAG_MULT_LOC_ACC
   uint32       intr_level;       /* Indicates which environment the
                                  /* calling application is in.
                                  /* PROCLEV or INTRKMEM or INTRPMEM
   struct timestruc_t *times;     /* Address of times structure, NULL if
                                  /* not used.
} pdiagex_opflags_t;
```

dma_struct

The **dma_struct** structure defines the DMA structure used by **PDIAGEX**. The **dma_struct** structure is defined in **/usr/include/sys/pdiagex_sys.h** and contains the following fields:

```
typedef struct dmast {
                  struct dmast *next;
                  int   firsttcw;  /* first TCW used (micro channel only) */
                  int   last_tcw;  /* last  TCW used (micro channel only) */
                  int   dma_flags; /* see /usr/include/sys/dma.h */
                  uchar *baddr;    /* address of the host buffer to DMA
                  uchar *daddr;    /*Phys addr in DMAbus_mem, from
                                                      diag_dma_master()*/
                  uint  count;     /* size of the DMA data in bytes */
                  struct xmem dp;  /* Cross Memory descriptor of baddr */
                  char  pinned;    /* NonZero if DMA buffer was pinned */
                  char  xmattached;  /* NonZero if DMA buffer was
                  char  in_use;    /* TRUE if this linked list member is
                } dma_info_t;
```

next Pointer to the next dma_info_t structure in an 'in_use' list.

firsttcw (Micro Channel devices Only) first page of *pdiagex_dds_t.dma_bus_mem* used by an active DMA master/slave operation.

last_tcw (Micro Channel devices Only) last page of *pdiagex_dds_t.dma_bus_mem* used by an active DMA master/slave operation.

dma_flags DMA flags as defined in **<sys/dma.h>**. These flags describe what actions to take (such as, master/slave transfer, initialize the DMA channel, and so on).

**baddr* Address of memory buffer for transfer.

**daddr* Address used to program the DMA master.

count Size (in bytes) of the DMA transfer.

dp Address of cross-memory descriptor.

pinned Nonzero if DMA buffer was pinned.

xmattached Nonzero if DMA buffer was cross-memory attached.

in_use Flag for determining if DMA buffer is valid for transfer.

aioo_struct_t

The **AIOO_STRUCT_T** structure defines the allocations, initializations, and outstanding operations for each handle. This provides a mechanism for error-recovery cleanup, cleanup of outstanding operations during a close, and general protection from the application. Common code may also be used for cleanup operations.

```
/* Allocation/Initialization/OutstandingOperations Binary Flags Structure */
                   typedef struct {
                     uint AllocIntrptDataMem : 1;
                     uint AllocDmaAreaMem    : 1;
                     uint CopyDDS            : 1;
                     uint CopyIntrptEnt      : 1;
                     uint PinIntrptFunct     : 1;
                     uint PinUIntrptData     : 1;
                     uint PinDiagExt         : 1;
                     uint InitIntrptChan     : 1;
                     uint InitDmaChan        : 1;
                     uint XmatUIntrptData    : 1;
                  } aioo_struct_t;
```

AllocIntrptDataMem Nonzero if Interrupt data area allocated.

AllocDmaAreaMem Nonzero if DMA data area allocated.

CopyDDS Nonzero if DDS data was copied to handle.

CopyIntrptEnt Nonzero if Intrpt function was in Kernel.

PinIntrptFunct Nonzero if Intrpt function was pinned.

PinUIntrptData Nonzero if Intrpt data area was pinned.

PinDiagExt Nonzero if Pinned **PDIAGEX** Extension.

InitIntrptChan Nonzero if Intrpt channel was initialized.

InitDmaChan Nonzero if DMA channel was initialized.

XmatUIntrptData Nonzero if Intrpt data area was XMattached.

diag_struc_t

The **diag_struc_t** structure defines the complete data structure returned in the handle for the **pdiag_open**() call. This structure holds all the needed information for all the other **PDIAGEX** function calls.

```
typedef struct handl {
                   struct intr        intr;
                   struct              handl *next;
                   int                 (*intr_func)();
                   uchar               *intr_data;>
                   struct xmem         udata_dp;
                   diagex_dds_t        dds;
                   struct timestruc_t  itime;
                   struct timestruc_t  ntime;
```

```
        dma_info_t          *dma_info;
        aioo_struct_t       aioo;
        char                *scratch_pad;
        uint                sleep_flag;
        uint                sleep_word;
        uint                flag_word;
        struct watchdog wdt;
        struct d_handle * dhandle;
        dma_dio            * dio_st;
        uint                    timeout;
     } diag_struc_t;
```

intr
: Interrupt handler structure as defined in **<sys/intr.h>**. Needs to be first parameter in **diag_struc_t**.

(**intr_func*)()
: Pointer to user's interrupt handler.

**intr_data*
: Pointer to interrupt data.

udata_dp
: Address of cross-memory descriptor for interrupt data.

dds
: Structure that contains the device driver structure (dds) information for **PDIAGEX**. See the **diagex_dds** structure defined above.

itime
: Time elapsed for interrupts. Updated at interrupts.

ntime
: Time elapsed for read or write operations. Updated at reads or writes.

**dma_info*
: Pointer to **dma_info_t** structure which allows multiple DMA operations. See the **dma_info_t** structure defined above.

aioo
: Set of flags for Allocations, Initializations, and Outstanding Operations.

scratch_pad
: PIO scratch pad for large transfers.

sleep_flag
: **pdiag_dd_watch_for_interrupt**() sets this flag to TRUE if it is sleeping and waiting for the application's interrupt handler to call **pdiag_dd_interrupt_notify**(). This flag is initialized to FALSE and will be set to FALSE after **pdiag_dd_watch_for_interrupt**() wakes up.

  >The application's interrupt handler should use this word to determine whether to 'wakeup' **pdiag_dd_watch_for_interrupt**(). This flag should *not* be modified by the application's interrupt handler.

sleep_word
: **pdiag_dd_watch_for_interrupt**() sleeps on this word until the application's interrupt handler calls **pdiag_dd_interrupt_notify**() using this word.

  This word should *not* be modified by the application's interrupt handler.

flag_word
: This flag is defined by the application and should be set by the application's interrupt handler to specify certain interrupt conditions. The application may call **pdiag_dd_watch_for_interrupt**(), specifying a flag_mask which will be bitwise ANDed with this flag_word. When this AND operation produces a nonzero result and **pdiag_dd_watch_for_interrupt**() is awake, **pdiag_dd_watch_for_interrupt**() will return.

wdt
: This is the watchdog timer used by the timeout function.

dhandle
: Structure returned by **D_MAP_INIT** macro which is called in the **pdiag_open**() function. This handle is used to issue DMA operations to rspc type systems.

dio_st
: Pointer to a DIO structure used in DMA operations.

timeout
: True if watchdog timer expired.

Kernel Services

The following is a list of Kernel Services used by **PDIAGEX**.

copyin
: Copies data between user and kernel memory.

copyout
: Copies data from kernel to user memory.

curtime
: Read the current time into **timestruc_t** structure.

d_bflush
: Flushes the appropriate I/O controller cache (IOCC), identified by the TCE bus address parameter, on memory-inconsistent platforms.

d_cflush
: Flushes the processor data cache and invalidates any prefetched data that may be in the IOCC buffers on memory-inconsistent platforms.

d_clear
: Frees a Direct Memory Access (DMA) channel.

d_complete Cleans up after a Direct Memory Access (DMA) transfer.
d_init Initializes a Direct Memory Access (DMA) channel.
d_mask Disables a Direct Memory Access (DMA) channel.
d_master Initializes a block-mode Direct Memory Access (DMA) transfer for a DMA master.
d_slave Initializes a block-mode Direct Memory Access (DMA) transfer for a DMA slave.
d_unmask Enables a Direct Memory Access (DMA) channel.
e_sleep Causes process to sleep.
e_wakeup Wakes up sleeping process.
i_clear Removes an interrupt handler.
i_disable Disables interrupt priorities.
i_enable Enables interrupt priorities.
i_init Defines an interrupt handler.
io_att Selects, allocates, and maps a region in the current address space for I/O access.
io_det Unmaps and deallocates the region in the current address space at the given address.
kmod_entrypt Returns a function pointer to a kernel module's entry point.
pincode Pins the code and data associated with an object file.
pinu Pins the specified address range in user or system memory.
unpincode Unpins the code and data associated with an object file.
unpinu Unpins the specified address range in user or system memory.
xmalloc Allocates memory.
xmattach Attaches to a user buffer for cross-memory operations.
xmdetach Detaches from a user buffer used for cross-memory operations.
xmemdma Prepares a page of memory for DMA (used with BUS_60X only).
xmemin Copies data to kernel space from a cross-memory attached buffer.
xmemout Copies data from kernel space to a cross-memory attached buffer.
xmfree Frees allocated memory.

Programmed I/O Services

The following is a list of Programmed I/O (PIO) macros used by **PDIAGEX**.

BUS_GETCX Reads the specified character value from the supplied bus memory, bus I/O, or POS address with built-in exception catching.
BUS_GETLX Reads the specified long value from the supplied bus memory, bus I/O, or POS address with built-in exception catching.
BUS_GETSX Reads the specified short value from the supplied bus memory, bus I/O, or POS address with built-in exception catching.
BUS_PUTCX Writes the specified character value to the supplied bus memory, bus I/O, or POS address with built-in exception catching.
BUS_PUTLX Writes the specified long value to the supplied bus memory, bus I/O, or POS address with built-in exception catching.
BUS_PUTSX Writes the specified short value to the supplied bus memory, bus I/O, or POS address with built-in exception catching.

The following is a list of Programmed I/O (PIO) macros used by the 64 bit **PDIAGEX**.

BUS_GETSTR Reads the specified character value from the supplied bus memory.
BUSIO_GETSTR Reads the specified character value from the supplied bus I/O.
BUS_GETS Reads the specified short value from the supplied bus memory.
BUSIO_GETS Reads the specified short value from the supplied bus I/O.
BUS_GETL Reads the specified long (32 bits) value from the supplied bus memory.
BUSIO_GETL Reads the specified long (32 bits) value from the supplied bus I/O.
BUS_PUTSTR Writes the specified character value to the supplied bus memory.
BUSIO_PUTSTR Writes the specified character value to the supplied bus I/O.
BUS_PUTS Writes the specified short value to the supplied bus memory.

BUSIO_PUTS	Writes the specified short value to the supplied bus I/O.
BUS_PUTL	Writes the specified long (32 bits) value to the supplied bus memory.
BUSIO_PUTL	Writes the specified long (32 bits) value to the supplied bus I/O.

Diagnostic Library

This section provides information on application programming interfaces to the Diagnostic Applications. The calls described are contained in the **/usr/lib/libdiag.a** diagnostic library.

The following is a list of all the exported programming interfaces available:

- ODM Object Class Functions
  - diag_add_obj
  - diag_change_obj
  - diag_close_class
  - diag_free_list
  - diag_get_list
  - diag_lock
  - diag_open_class
  - diag_rm_obj
  - diag_unlock
  - init_dgodm
  - term_dgodm
- Device Configuration
  - configure_device
  - diagex_cfg_state
  - diagex_initial_state
  - get_device_status
  - initial_state
- FRU Bucket Functions
  - addfrub
  - insert_frub
- Catalog File Functions
  - diag_catopen
  - diag_cat_gets
- Menu Functions
  - diag_popup
  - diag_progress
  - diag_read
  - diag_resource_screen
  - diag_task_screen
  - diag_asl_clear_screen
  - diag_asl_init
  - diag_asl_msg
  - diag_asl_read
  - diag_asl_quit
  - diag_display

- diag_display_menu
- diag_emsg
- diag_msg

- Device Attributes, Properties
  - diag_get_device_flag
  - diag_get_property
  - diag_get_sid_lun
  - get_cpu_model
  - get_dev_desc
  - get_diag_att
- Diagnostic Event Log Functions
  - dlog_getTestMode
  - dlog_close
  - dlog_find_first
  - dlog_find_next
  - dlog_find_sequence
  - dlog_formatElogResults
  - dlog_freeEntry
  - dlog_open
  - dlog_read
  - dlog_same_elogId
  - dlog_setEntryType
  - dlog_write
  - save_davars_ela
- Miscellaneous
  - copy_text
  - DA_SETRC_XXXXXX
  - diag_asl_beep
  - diag_asl_execute
  - diag_exec_source
  - diag_execute
  - dt
  - error_log_get
  - file_present
  - get_DApp
  - getdainput
  - getdavar
  - getELAdates
  - has_diag_authority
  - ipl_mode
  - menugoal
  - schedule_ela

diag_add_obj

Purpose

Adds a new object into an object class.

Syntax

```
#include  <diag/diagodm.h>
#include  <sys/cfgodm.h>
#include  <diag/DiagODM.h>

void diag_add_obj (
                     void       *classp,
                     void       *p_obj)
```

Description

The **diag_add_obj** subroutine takes as input the class symbol that identifies the object class to change and a pointer to the data structure that contains the object to be added.

Parameters

classp A class symbol identifier returned from a **diag_open_class** subroutine. If the **diag_open_class** subroutine has not been called, this is the structure name of the class normally defined in either **diag/diagodm.h** file, **diag/DiagODM.h** file or **sys/cfgodm.h** file.

p_obj Pointer to an instance of the structure corresponding to the object class referenced by the *classp* parameter.

Return Value

Upon successful completion, a value of 0 is returned. If the subroutine fails, a -1 is returned.

diag_change_obj

Purpose

Changes an object in the object class.

Syntax

```
#include     <diag/diagodm.h>
#include     <sys/cfgodm.h>
#include     <diag/DiagODM.h>

void diag_change_obj (
                                 void        *classp,
                                 void        *p_obj)
```

Description

The **diag_change_obj** subroutine takes as input the class symbol that identifies the object class to add to and a pointer to the data structure that contains the object to be changed. The application must first retrieve the object with a **diag_get_list** subroutine call, change the data values in the returned structure, and then pass that structure to the **diag_change_obj** subroutine.

Parameters

classp A class symbol identifier returned from a **diag_open_class** subroutine. If the **diag_open_class** subroutine has not been called, then this is the structure name of the class normally defined in either the **diag/diagodm.h** file, **diag/DiagODM.h** file, or **sys/cfgodm.h** file.

p_obj Pointer to an instance of the structure corresponding to the object class referenced by the *classp* parameter.

Return Value

Upon successful completion, a value of 0 is returned. If the subroutine fails, a -1 is returned.

diag_close_class

Purpose

Closes an object class.

Syntax

```
#include     <diag/diagodm.h>
#include     <sys/cfgodm.h>
#include     <diag/DiagODM.h>

int diag_close_class (
                                 void             *classp)
```

Description

The **diag_close_class** subroutine can be called to close an object class.

Parameters

classp A class symbol identifier returned from a **diag_open_class** subroutine. If the **diag_open_class** subroutine has not been called, then this is the structure name of the class normally defined in either the **diag/diagodm.h** file, **diag/DiagODM.h** file, or **sys/cfgodm.h** file.

Return Value

Upon successful completion, a value of 0 is returned. If the subroutine fails, a -1 is returned.

diag_free_list

Purpose

Frees memory previously allocated for a **diag_get_list** subroutine.

Syntax

```
#include    <diag/diagodm.h>
#include    <sys/cfgodm.h>
#include    <diag/DiagODM.h>

int diag_free_list (
                                 void             *p_obj,
                                 struct listinfo  *info)
```

Description

The **diag_free_list** subroutine recursively frees up a tree of memory object lists that were allocated for a **diag_get_list** subroutine.

Parameters

p_obj Points to the array of structures returned from the **diag_get_list** subroutine.
info Points to the **listinfo** structure returned from the **diag_get_list** subroutine.

Return Value

Upon successful completion, a value of 0 is returned. If the subroutine fails, a -1 is returned.

diag_get_list

Purpose

Retrieves all objects in an object class that match the specified criteria.

Syntax

```
#include    <diag/diagodm.h>
#include    <sys/cfgodm.h>
#include    <diag/DiagODM.h>

void * diag_get_list (
                                void        *classp,
                                char        *criteria,
                                struct      listinfo *info,
                                int         max_expect,
                                int         depth)
```

Description

The **diag_get_list** subroutine takes an object class and criteria as input, and returns a list of objects that satisfy the input criteria. The subroutine opens and closes the object class around the get if the object class was not previously opened. If the object class was previously opened, the subroutine leaves the object class open when it returns.

Parameters

classp	Class symbol identifier returned from a **diag_open_class** subroutine. If the **diag_open_class** subroutine has not been called, then this is the structure name of the class normally defined in either the **diag/diagodm.h** file, **diag/DiagODM.h** file, or **sys/cfgodm.h** file.
criteria	String that contains the qualifying criteria for selecting objects.
info	Structure containing information about the retrieval of the objects.
max_expect	Expected number of objects to be returned. This is used to control the increments in which storage for structures is allocated, to reduce the **realloc** subroutine copy overhead.
depth	Number of levels to recurse for objects with linking descriptors.

Return Value

Upon successful completion, a pointer to an array of C language structures containing the objects is returned. If no match is found, **NULL** is returned. If the **diag_get_list** fails, a value of -1 is returned.

diag_lock

Purpose

Obtain an ODM lock for the specified file

Syntax

```
#include            <diag/odmi.h>

int diag_lock(char *file)
```

Description

The **diag_lock** subroutine calls **odm_lock()** for a specified file. It waits 5 seconds if a lock cannot be immediately granted.

Parameters

file	Name of the file to lock

Return Value

The **diag_lock** subroutine returns one of the following values:

>0	If successful
0	File is already locked
-1	Error

diag_open_class

Purpose

Opens an object class.

Syntax

```
#include      <diag/diagodm.h>
#include      <sys/cfgodm.h>
#include      <diag/DiagODM.h>

void *diag_open_class (
                                void       *classp)
```

Description

The **diag_open_class** subroutine can be called to open an object class.

Parameters

classp
: The structure name of the class normally defined in either the **diag/diagodm.h** file, **diag/DiagODM.h** file, or **sys/cfgodm.h** file.

Return Value

Upon successful completion, a class symbol identifier for the object class is returned. If the subroutine fails, a -1 is returned.

diag_rm_obj

Purpose

Deletes objects from an object class.

Syntax

```
#include  <diag/diagodm.h>
#include  <sys/cfgodm.h>
#include  <diag/DiagODM.h>

void diag_rm_obj (
                                void       *classp,
                                char       *criteria)
```

Description

The **diag_rm_obj** subroutine deletes objects from an object class.

Parameters

classp
: Class symbol identifier returned from a **diag_open_class** subroutine. If the **diag_open_class** subroutine has not been called, then this is the structure name of the class normally defined in either **diag/diagodm.h** file, **diag/DiagODM.h** file or **sys/cfgodm.h** file.

criteria
: String containing the qualifying criteria for selecting objects to delete.

Return Value

Upon successful completion, the number of objects deleted is returned. If the subroutine fails, a -1 is returned.

diag_unlock

Purpose

Release an ODM lock

Syntax

```
#include              <odmi.h>
int diag_unlock(int *id)
```

Description

The **diag_unlock** subroutine releases an odm lock.

Parameters

id	Lock id to release

Return Value

The **diag_unlock** subroutine returns one of the following values:

`0`	If successful
`-1`	Error occured while trying to unlock a lock

init_dgodm, term_dgodm

Purpose

Initializes or stops the Object Data Manager.

Syntax

```
int init_dgodm ( )

int term_dgodm ( )
```

Description

The **init_dgodm** subroutine issues an **odm_initialize** call to the Object Data Manager. This should be done at the beginning of the Diagnostic Application (DA).

The **term_dgodm** subroutine issues an **odm_terminate** call to the Object Data Manager. This should be done at the end of the DA.

Parameters

Takes no parameters.

Return Value

A value of 0 is always returned.

configure_device, initial_state

Purpose

Puts a device and parentage into the available state.

Restores a device and parentage to their initial state before configuration.

Syntax

```
#include   <diag/diagodm.h>
#include   <sys/cfgodm.h>
#include   <sys/cfgdb.h>

int  configure_device ( name )
char *name;

int  initial_state ( state, name )
int    state;
char *name;
```

Description

The **configure_device** subroutine is used to put a device into the AVAILABLE state (for testing) if the device is presently DEFINED or STOPPED. Also the parentage of the device is checked, and their states also put into AVAILABLE state if necessary.

The **initial_state** subroutine is used to restore the device and parentage back to their initial state (after testing).

Parameters

name	Identifies the device.
state	Indicates the previous state of the device.

Return Value

The following values are returned:

DEFINED	Device was previously in the DEFINED state.
AVAILABLE	Device is already in the AVAILABLE state.
STOPPED	Device was previously in the STOPPED state.
-1	Error configuring the device.

diagex_cfg_state

Purpose

Puts the device under test in the DIAGNOSE state.

Syntax

```
#include    <diag/diag.h>

int  diagex_cfg_state ( device_name )
char *device_name;
```

Description

The **diagex_cfg_state** subroutine unconfigures the device, and its children if necessary, to set the device into the DIAGNOSE state. Original states of all devices changed will be saved. Use **diagex_initial_state** to put the changed devices back to their original states.

The global variable **diag_cfg_errno** will be set to the return value of the method invoked for the device.

Parameters

device_name	Name of the device under test.

Return Value

The **diagex_cfg_state** subroutine returns one of the following values:

0	Successful return.
-1	Software error.
1	Child device cannot be unconfigured.
2	Device cannot be unconfigured.
3	Device cannot be put into DIAGNOSE state.

diagex_initial_state

Purpose

Puts the device under test back to its original state.

Syntax

```
#include     <diag/diag.h>

int  diagex_initial_state ( device_name )
char *device_name;
```

Description

The **diagex_initial_state** subroutine puts the device, and its children if necessary, back to the original state before the **diagex_cfg_state** routine was called.

Parameters

device_name Name of the device under test.

Return Value

The **diagex_initial_state** subroutine returns one of the following values:

0	Successful return.
-1	Software error.
4	Device cannot be restored to DEFINE state.
5	Device cannot be restored to AVAILABLE state.
6	Child device cannot be restored to original state.

get_device_status

Purpose

Returns the device's current configuration status.

Syntax

```
#include <sys/cfgdb.h>

int  get_device_status ( device_name )
char * device_name;
```

Description

The **get_device_status** subroutine returns the current device configuration status. The status is obtained by returning the value of the **CuDv** *status* field of the device.

Parameters

device_name Character pointer to the name of the device.

Return Value

The **get_device_status** subroutine returns one of the following values:

DEFINED	Device is in the DEFINED state.
AVAILABLE	Device is in the AVAILABLE state.
STOPPED	Device is in the STOPPED state.
DIAGNOSE	Device is in the DIAGNOSE state.
-1	System error obtaining device status.

addfrub

Purpose

Concludes a field replaceable unit (FRU) goal.

Syntax

```
#include                <diag/da.h>
int addfrub ( fptr )
struct fru_bucket *fptr;
```

Description

The **addfrub** subroutine associates a FRU with the device currently being tested. The **TMInput** object class identifies the device currently being tested.

Parameters

fptr
: Pointer to a structure of type fru_bucket, which is defined as follows:

```
struct fru_bucket  {
          char   dname[NAMESIZE];
          short  ftype;
          short  sn;
          short  rcode;
          short  rmsg;
          struct {
                   int   conf;
                   char  fname[NAMESIZE];
                   char  floc[LOCSIZE];
                   short fmsg;
                   char  fru_flag;
                   char  fru_exempt;
                 } frus[MAXFRUS];
};
```

dname
: Names the device under test.

ftype
: Indicates the type of FRU Bucket being added to the system. The following values are defined:

FRUB1 The FRUs include the resource that failed, its parent, and any cables needed to attach the resource to its parent.

FRUB2 This FRU Bucket is similar to FRU Bucket **FRUB1**, but does not include the parent resource.

sn
: Source number of the failure. The source number is usually set to the *led* field of the **PdDV** object class by the **insert_frub** subroutine. If the *sn* set by the **insert_frub** subroutine is not the desired value, the calling subroutine should set *sn* to the desired value after the **insert_frub** subroutine and before the **addfrub** subroutine.

rcode Reason code associated with the failure.

Note: A Service Request Number is formatted as follows:

```
SSS - RRR
```

where SSS is the *sn* and RRR is the *rcode*.

Some devices may use a different nomenclature for their service request numbers. For this special case, the *sn* parameter indicates how the *rcode* value should be formatted. If *sn* = 0, then *rcode* is interpreted as decimal. If *sn* = -1, then *rcode* is interpreted as a 4-digit hexadecimal number.

If *sn* = -2, then the object class **DAVars** is searched for an attribute of Errorcode. This allows the displaying of 8 digit hex Error Codes. The diagnostic application is responsible for setting up a **DAVars** object similar to the following:

DAVars:

```
dname: <device name under test>
vname: Error_code           "Error_code is an ascii string"
vtype: DIAG_STRING          "Literal value"
val: <8 digit hex character string>
```

See the **getdavar/putdavar** subroutine for more information.

rmsg Message number of the text describing the reason code. The set number of the text is predefined by the *PSet* field in the **Predefined Diagnostic Resources** object class.

conf Indicates whether an FRU is valid. A value of 0 indicates an invalid FRU. No other FRUs are displayed once an invalid FRU is found in the FRU bucket.

However, if *fname* contains the string **REF-CODE**, then the *fmsg* and *conf* values are used to make the 8-digit ref code.

For AIX 4.3.2 and earlier versions, this field indicates the probability of failure associated with the named FRU.

fname Names the FRU.

The parameters *floc* and *fmsg* must be specified, if *fname* is *not* represented in the **Customized Devices** object class. Otherwise, they should be set to 0.

floc Location of *fname*.

fmsg Message number of the text describing *fname*. The set number is predefined by the *PSet* descriptor in the **Predefined Diagnostic Resources** object class.

fru_flag Flag used by the Diagnostic Applications (DA) in determining which FRU to use in the **frus**[ ] structure. The following values are defined:

NOT_IN_DB
The FRU is not represented in the config database.

DA_NAME
frus[ ].*fname* should be the name of the device being tested.

PARENT_NAME
frus[ ].*fname* should be the name of the parent of the device being tested.

CHILD_NAME
frus[ ].*fname* should be the name of the child of the device being tested.

NO_FRU_LOCATION
The FRU name will be left blank, and the FRU location code will be set to the location of the device under test (*dname*).

fru_exempt Indicates that the designated FRU will not be absorbed as a result of chip/FRU integration. The following values are defined:

EXEMPT
FRU cannot be integrated (For example, fuse, cable, displays, etc.) This value should be the most-used value, and should be used in conjunction with the **fru_flag** field. Examples are:

```
FRU                  fru_flag        fru_exempt
----                 --------        ----------
Device being tested  DA_NAME         EXEMPT
Parent of device     PARENT_NAME     EXEMPT
CABLE                NOT_IN_DB       EXEMPT
```

NONEXEMPT
FRU can be integrated (generally, any specific chip set).

Note: DAs do not have to return **MAXFRU frus**. The Diagnostic Controller processes **frus**[ ] from 0..MAXFRU-1, while conf>0.

Return Value

Upon successful completion, a value of 0 is returned. If the **addfrub** subroutine is unsuccessful, then a value of -1 is returned.

insert_frub

Purpose

Updates FRU Bucket.

Syntax

```
#include    <diag/tm_input.h>
#include    <diag/da.h>

long insert_frub ( tminput, frub )
struct tm_input *tminput;
struct fru_bucket *frub;
```

Description

The **insert_frub** subroutine gets a device's FRU name and source number from the **Customized Device** object class and places them into a structure of type **fru_bucket**. The calling routine specifies through the **fru_flag** member of the **FRU Bucket** structure whether the FRU name is for device *x* or the FRU parent of *x*.

Parameters

tminput Identifies the device x (specifically, *tminput.dname*).
frub Pointer to the FRU Bucket structure to be updated.

This function should be called before **addfrub()**.

Return Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned.

diag_catopen

Purpose

Opens a diagnostic catalog message file.

Syntax

```
#include     <diag/diago.h>

nl_catd  diag_catopen ( filename, reserved )
char* filename;
int reserved;
```

Description

The **diag_catopen** subroutine is used to open a catalog message file. It first searches the normal catalog directory as specified by the **$LANG** and **$NLSPATH** environment variables. If the catalog file is not found, the function searches the default catalog directory.

Parameters

filename Catalog file name to be opened.

Return Value

The **diag_catopen** subroutine returns a *nl_catd* catalog descriptor.

diag_cat_gets

Purpose

Obtains catalog messages from **NLSPATH** or default diagnostic catalog directory.

Syntax

```
#include     <diag/diago.h>

char  *diag_cat_gets ( fdes, setid, msgid )
nl_catd fdes;
unsigned short setid;
unsigned short msgid;
```

Description

The **diag_cat_gets** subroutine is used to get messages from a catalog file. It first searches the normal catalog directory as specified by the **$LANG** and **$NLSPATH** environment variables. If the set and message is not found, the function searches the default catalog directory.

Parameters

fdes Open catalog file descriptor returned from the **diag_catopen()** system call.

setid Set ID of the message in the catalog.

msgid Message ID of the message in the catalog that serves as the format string.

Return Value

The **diag_cat_gets** subroutine returns a character pointer to the message string.

diag_resource_screen

Purpose

Displays menus commonly used by Diagnostic Applications (DA).

Syntax

```
#include <diag/diag_screen.h>

#include <diag/diag.h>

long  diag_resource_screen ( screen_info, screen_data, screen_msg )
```

```
screen_info_t *screen_info;
screen_data_t *screen_data;
screen_msg_t screen_msg[];
```

Description

The **diag_resource_screen** subroutine displays menus commonly used by Diagnostic Applications.

Parameters

screen_info	Screen Information. This structure defines the screen type and screen ID.	
	short *screen_type*	Screen Type. • **INFORMATIVE** • **TRANSITIONAL** • **SINGLE_SELECTION**
	short *screen_id*	Screen Identifier. • **TESTING_MENU** • **ANALYZE_ERROR_LOG** • **ANALYZE_POST** • **ANALYZE_FIRMWARE** • **ANALYZE_CHECKSTOP** • **ANALYZE_SUBSYS**
	short *screen_key*	Identifies extra function keys for screen. • **DIAG_HELP_KEY**
	long *item_selected*	Indicates the selected item in the list, if *screen_type* is **SINGLE_SELECTION**. First selectable item in *screen_msg* would have a 1 returned, second selectable item would have a 2 returned, and so on.

screen_data	Screen Data. This structure contains all data needed to construct the screen.	
	nl_catd *fdes*	Catalog file descriptor.
	long *menu_number*	Menu number that is displayed, right-justified, as a hex number at the top-right corner of the screen.
	char * *resource_name*	The name of the resource being tested. **(tminput->dname)**
	char * *location_code*	The logical location code of the resource being tested. **(tminput->dnameloc)**
	short *test_mode*	The test mode (ADVANCED, NON_ADVANCED) this session is running in. **(tminput->advanced)**
	short *loop_mode*	Indicates whether Loop Mode has been selected. **(tminput->loopmode)**
	short *lcount*	Total number of passes made. This value is used only when *loop_mode* is not set to **LOOPMODE_NOTLM**.
	short *lerrors*	Total number of errors encountered. This value is used only when *loop_mode* is not set to **LOOPMODE_NOTLM**.
	short *msg_count*	Total number of messages in the *screen_msg* structure.
screen_msg	The *screen_msg* structure contains an array of setid's and msgid's used to construct the text (or body) of the screen. This includes all messages except the last line, or INSTRUCTION line. This structure is not required for a **TRANSITIONAL** screen type, use NULL for the screen_msg argument.	
	short *set_num*	The set number containing the message text.
	short *msg_num*	The message number containing the message text.
	char * *message*	Text message to use in place of < set_num, msg_num >. This is useful if string substitution was required in order to build the message text. This text will take precedence over the < set_num, msg_num > if not NULL.
	short *msg_type*	Flag indicating the type of message to be displayed. • **HELP_MSG** Only one message of this type allowed. This help message will always be associated with the screen, and not any particular line. • **SELECTABLE_MSG** • **INFO_MSG**

NOTES:

- This structure must be built exactly for a **SINGLE_SELECTION** screen type. screen_msg[0..n] MUST have the *msg_type* set to **SELECTABLE_MSG** for all selectable messages.

- screen_msg[n+1] MUST have the *msg_type* set to **INFO_MSG** if you want some kind of information displayed to the user before the **INSTRUCTION** line.
- The help message, if any, should be last.

Return Value

The **diag_resource_screen** subroutine returns one of the following values:

DIAG_OK	Successful return.
DIAG_MALLOCFAILED	Memory allocation was unsuccessful.
DIAG_ENTER	Enter Function key was entered.
DIAG_EXIT	Exit Function key was entered.
DIAG_CANCEL	Cancel Function key was entered.
DIAG_HELP	Help Function key was entered.
DIAG_FAIL	Invalid data structure, software error

diag_popup

Purpose

Creates a popup window with message text.

Syntax

```
#include <diag/diag_screen.h>

long  diag_popup ( char * fmt, [, name, ...] )

char * fmt;
```

Description

The **diag_popup** subroutine displays a popup window.

Parameters

The parameters are similar to those of the standard I/O library subroutine **printf()**. There is a 2000 character limit on the length of the message.

Return Value

The **diag_popup** subroutine returns one of the following values:

DIAG_CANCEL	Cancel key was entered.
DIAG_ENTER	Enter Function key was entered.
DIAG_EXIT	Exit Function key was entered.

diag_task_screen

Purpose

Displays menus commonly used by Diagnostic Tasks.

Syntax

```
#include <diag/diag_screen.h>

#include <diag/diag.h>

long  diag_task_screen ( screen_info, screen_task_data, screen_task_msg )

screen_info_t *screen_info;
screen_task_t *screen_task_data;
screen_task_msg_t screen_task_msg[];
```

Description

The **diag_task_screen** subroutine displays menus commonly used by Diagnostic Tasks.

Parameters

<table>
<tr><td rowspan="5">screen_info</td><td colspan="2">Screen Information. This structure defines the screen type.</td></tr>
<tr><td>short screen_type</td><td>Screen Type.
• INFORMATIVE
• TRANSITIONAL
• DIALOG
• SINGLE_SELECTION
• MULTIPLE_SELECTION</td></tr>
<tr><td>short screen_id</td><td>Screen Identifier - Not Used.</td></tr>
<tr><td>short screen_key</td><td>Identifies extra function keys for screen.
• DIAG_LIST_KEY
• DIAG_HELP_KEY if screen_type is INFORMATIVE</td></tr>
<tr><td>long item_selected</td><td>Indicates the selected item in the list, if screen_type is SINGLE_SELECTION. First selectable item in screen_msg would have a 1 returned, second selectable item would have a 2 returned, and so on.
For a MULTIPLE_SELECTION screen_type, this field is used to keep track of the current selection for subsequent calls until the COMMIT function key is used.</td></tr>
<tr><td rowspan="4">screen_task_data</td><td colspan="2">Screen Data. This structure contains all data needed to construct the screen.</td></tr>
<tr><td>nl_catd fdes</td><td>Catalog file descriptor.</td></tr>
<tr><td>long menu_number</td><td>Menu number that is displayed, right-justified, as a hex number at the top-right corner of the screen.</td></tr>
<tr><td>short msg_count</td><td>Total number of messages in the screen_task_msg structure.</td></tr>
</table>

screen_task_msg	The *screen_task_msg* structure contains an array of setid's and msgid's used to construct the text (or body) of the screen. This includes all except the last line, or Instruction line.	
	short *set_num*	The set number containing the message text.
	short *msg_num*	The message number containing the message text.
	char * *message*	Text message to use in place of < set_num, msg_num >. This is useful if string substitution was required in order to build the message text. This text will take precedence over the < set_num, msg_num > if not NULL.
	short *help_set_num*	The set number containing the message text when the HELP key is pressed. Help message text is line sensitive, and is normally used when the *msg_type* is set to **SELECTABLE_MSG** or **DIALOG_MSG**.
	short *help_msg_num*	The message number containing the message text when the HELP key is pressed. Help message text is line sensitive, and is normally used when the *msg_type* is set to **SELECTABLE_MSG** or **DIALOG_MSG**.
	short *msg_type*	Flag indicating if text is help, selectable, dialog, or information. • **TITLE_MSG** • **SELECTABLE_MSG** • **DIALOG_MSG** • **INFO_MSG**
	char *leading_char*	A specific character to be displayed before the message text. Note that this is also used as the mechanism to determine which selectable items had been selected on a **MULTIPLE_SELECTION** screen.
	long *line_num*	Internal screen line number.
	char *op_type*	Type of operation allowed on this field
	char *entry_type*	type of (user) entry allowed in the field
	char *required*	• DIAG_YES • DIAG_YES_NON_EMPTY • DIAG_EXCEPT_WHEN_EMPTY • DIAG_NO = default • DIAG_YES or DIAG_YES_NON_EMPTY means display required flag
	char *changed*	DIAG_YES, DIAG_NO = default; field changed from default value
	char **disp_values*	disp. text of allowed/default choice(s, separated by ″,″)
	char **data_value*	MUST point to string (buffer) of size (entry_size + 1) if there is ANY way values may be changed (typein/list/ring)
	long *entry_size*	maximum size of (data_)value that can be entered OR returned (include a ″return″ of anything from disp_values)
	long *cur_value_index*	
	long *default_value_index*	0 origin index of default value

NOTES:

- The *screen_task_msg* structure must be built exactly for **SINGLE_SELECTION**, **MULTIPLE_SELECTION**, and **DIALOG** screen types. screen_msg[0] MUST have the *msg_type* set to **TITLE_MSG** for the TITLE line.

- screen_msg[1..n] MUST have the *msg_type* set to **SELECTABLE_MSG** or **DIALOG_MSG** for all selectable/dialog messages.
- screen_msg[n+1] MUST have the *msg_type* set to **INFO_MSG** if you want some kind of information displayed to the user before the **INSTRUCTION** line.

Return Value

The **diag_task_screen** subroutine returns one of the following values:

DIAG_OK	Successful return.
DIAG_MALLOCFAILED	Memory allocation was unsuccessful.
DIAG_ENTER	Enter Function key was entered.
DIAG_EXIT	Exit Function key was entered.
DIAG_CANCEL	Cancel Function key was entered.
DIAG_HELP	Help Function key was entered.
DIAG_LIST	List Function key was entered.
DIAG_FAIL	Invalid data structure, software error
DIAG_COMMIT	Commit function key was entered.

diag_progress

Purpose

Displays progress messages by the Diagnostic Applications and Diagnostic Tasks.

Syntax

```
#include <diag/diag_screen.h>

#include <diag/diag.h>

void  diag_progress ( screen_progress )

screen_prog_t *screen_progress;
```

Description

The **diag_progress** subroutine displays the progress indicators used by Diagnostic Applications and other Diagnostic Tasks.

Parameters

screen_prog	Screen Progress Information. This structure defines the progress message to be displayed and the percentage complete. **int** *max_value (Used for Web-based System Manager progress bars)* Maximum value. **int** *current_value (Used for Web-based System Manager progress bars)* Current value. **char *** *progress_msg* Progress message to be displayed.

diag_read

Purpose

Reads user input.

Syntax

```
#include <diag/diag_screen.h>

#include <diag/diag.h>
```

```
long  diag_read ( screen_info, wait, buffer )

screen_info_t *screen_info;
int  wait;
char * buffer;
```

Description

The **diag_read** subroutine reads the keyboard buffer.

Parameters

screen_info	Screen Information. This structure defines the screen type and screen ID. Only the *screen_type* is used.	
	short *screen_type*	Screen Type. • **INFORMATIVE** • **TRANSITIONAL** • **DIALOG** • **SINGLE_SELECTION** • **MULTIPLE_SELECTION**
wait	If TRUE, causes this subroutine to wait until the user presses one of the keys allowed by the *screen_type*. If this parameter is FALSE, then this subroutine does not wait for the user input but processes anything typed ahead just as it would if the parameter were TRUE.	
buffer	Allocated by the application. It is used to return the values entered by the user. The buffer size must not be greater than 100 bytes. (Currently not implemented).	

diag_asl_clear_screen

Purpose

Clears the screen.

Syntax

```
#include     <diag/diago.h>

long  diag_asl_clear_screen (  )
```

Description

The **diag_asl_clear_screen** subroutine is used to clear the screen.

Parameters

Takes no parameters.

Return Value

The following values are returned:

DIAG_ASL_OK	Successful return.
DIAG_ASL_FAIL	Not called following **diag_asl_init**() and before **diag_asl_quit**().

diag_asl_init

Purpose

Initializes the user interface.

Syntax

```
#include     <diag/diago.h>

long  diag_asl_init ( name )
char *name;
```

Description

The **diag_asl_init** subroutine is used to initialize the user interface and should be the first call made to the user interface.

Parameters

name Identifies any options. This field has the following values:

DEFAULT
Type ahead allowed.

NO_TYPE_AHEAD
Type ahead not allowed.

Return Value

The following values are returned:

DIAG_ASL_OK	Successful return.
DIAG_ASL_ERR_NO_SUCH_TERM	Specified TERM entry does not exist.
DIAG_ASL_ERR_TERMINFO_GET	TERMINFO get failed.
DIAG_ASL_ERR_NO_TERM	TERM entry missing.
DIAG_ASL_ERR_INITSCR	Initscr() failed.
DIAG_ASL_ERR_SCREEN_SIZE	Screen/window size less than minimum.

diag_asl_msg

Purpose

Creates a pop-up window with message text.

Syntax

```
#include     <diag/diago.h>

long  diag_asl_msg ( fmt, [,  name, ... ] )
char *fmt;
```

Description

The **diag_asl_msg** subroutine should only be used by service aids to display a pop-up window with informational text.

Parameters

The parameters are similar to those of the standard I/O library subroutine **printf()**.

Return Value

The following values are returned:

DIAG_ASL_CANCEL	Cancel key was pressed.
DIAG_ASL_ENTER	Enter key was pressed.
DIAG_ASL_HELP	Help key was pressed.
DIAG_ASL_LIST	List key was pressed.
DIAG_ASL_COMMAND	Command key was pressed.
DIAG_ASL_COMMIT	Commit key was pressed.

diag_asl_read

Purpose

Reads user input.

Syntax

```
#include     <diag/diago.h>

long  diag_asl_read ( screen_code, wait, buf )
ASL_SCREEN_CODE screen_code;
int wait;
char *buf;
```

Description

The **diag_asl_read** subroutine reads the keyboard buffer.

Parameters

screen_code	Identifies the set of function keys that should be active.
wait	If True, causes this subroutine to wait until the user presses one of the keys allowed by the *screen_type*. If this parameter is False, then this subroutine does not wait for the user input but processes anything typed ahead just as it would if the parameter were True.
buf	Allocated by the application. It is used to return the values entered by the user. If used, this buffer MUST be at least ASL_READ_BUF_SIZE. Normally this value should be set to NULL. When NULL, only the function key pressed is returned.

Return Value

The **diag_asl_read** subroutine returns one of the following values:

DIAG_ASL_OK	Successful return.
DIAG_ASL_FAIL	Failure reading data.
DIAG_ASL_CANCEL	Cancel key was entered.
DIAG_ASL_ENTER	Enter key was entered.
DIAG_ASL_EXIT	Exit key was entered.

diag_asl_quit

Purpose

Terminates the user interface.

Syntax

```
#include     <diag/diago.h>

long  diag_asl_quit ( name )
char *name;
```

Description

The **diag_asl_quit** subroutine is used to end the user interface and should be the last call made to the user interface.

Parameters

name Identifies any options. This field has the following values:

DCTRL Used by Diagnostic Controller only.

DEFAULT
Used by all other applications.

Return Value

The following value is always returned:

0 Successful return.

diag_display

Purpose

Displays a menu and reads the user's response.

Syntax

```
#include     <diag/diago.h>

long  diag_display ( mnum, fdes, msglist, proctype,
                     scrtype, menutype, menuinfo )
long mnum;
nl_catd fdes;
struct msglist msglist[ ];
long proctype;
long scrtype;
ASL_SCR_TYPE *menutype;
ASL_SCR_INFO *menuinfo;
```

Description

The **diag_display** subroutine displays a menu that has multiple user selections and reads the user's response.

Parameters

mnum Menu number that is displayed, right-justified, as a hex number at the top-right corner of the screen.

fdes Open catalog file descriptor returned from the **diag_catopen** system call.

msglist Array of set numbers and message IDs. The *msglist* parameter must be ended by a Null element.

proctype Specifies the type of operation to be performed. This parameter has the following values:

DIAG_MSGONLY
The specified messages are retrieved from the catalog, but not displayed. The application writer should update the *menuinfo* parameter and restart the **diag_display** subroutine with the *msglist* parameter equal to Null.

DIAG_IO
The list of messages specified by *msglist* or, if that is Null, those in the array *menuinfo*, are displayed in the format specified by the *menutype* parameter.

scrtype Specifies the type of screen to be displayed, where each type determines the format of the output and the active function keys for the user.

menutype Defined in the file **/usr/include/asl.h**. If this parameter is equal to Null, the default version is used. Otherwise, the application's version is used.

menuinfo Defined in the file **/usr/include/asl.h**. If this field is *not* equal to Null, it is initialized with the retrieved messages.

Return Value

The **diag_display** subroutine returns one of the following values:

DIAG_ASL_OK	Successful return.
DIAG_ASL_ARGS1	Both the *msglist* and *menuinfo* parameters were Null.
DIAG_ASL_ARGS2	**DIAG_MSGONLY** option was specified, but no messages were named.
DIAG_MALLOCFAILED	Memory allocation was unsuccessful.
DIAG_ASL_ENTER	Enter Function key was entered.
DIAG_ASL_EXIT	Exit Function key was entered.
DIAG_ASL_CANCEL	Cancel Function key was entered.

DIAG_ASL_HELP	Help Function key was entered.
DIAG_ASL_LIST	List Function key was entered.
DIAG_ASL_COMMIT	Commit Function key was entered.
DIAG_ASL_PRINT	Print Function key was entered.

diag_display_menu

Purpose

Displays menus commonly used by Diagnostic Applications (DA).

Syntax

```
#include <diag/diago.h>

#include <diag/diag.h>

long  diag_display_menu ( msgid, mnum, substitution, lcount, lerrors )
long  msgid;
long  mnum;
char  *substitution[];
int   lcount;
int   lerrors;
```

Description

The **diag_display_menu** subroutine displays commonly used menus.

Parameters

msgid
: Message ID number defined in **dcda.msg**. Currently, the following message IDs are defined:

  CUSTOMER_TESTING_MENU

  ADVANCED_TESTING_MENU

  LOOPMODE_TESTING_MENU

  NO_MICROCODE_MENU

  NO_DIAGMICROCODE_MENU

  NO_DDFILE_MENU

  NO_HOT_KEY

  DEVICE_INITIAL_STATE_FAILURE

mnum
: Menu number that is displayed, right-justified, as a hex number at the top-right corner of the screen.

substitution
: Used to pass in strings to be substituted in the menu. This must be an array of three (3) character pointers. The device descriptive text is the first element. The device name as it comes from TMInput->dname is the second, and the location code is the third.

lcount
: Used to allow the loop-count value to be displayed. This value is used only when *mnum* is set to **LOOPMODE_TESTING_MENU**.

lerrors
: Used to allow the number of errors value to be displayed. This value is used only when *mnum* is set to **LOOPMODE_TESTING_MENU**.

Return Value

The **diag_display_menu** subroutine returns one of the following values:

DIAG_ASL_OK	Successful return.
DIAG_ASL_ARGS1	Both the *msglist* and *menuinfo* parameters were Null.
DIAG_ASL_ARGS2	**DIAG_MSGONLY** option was specified, but no messages were named.

DIAG_MALLOCFAILED	Memory allocation was unsuccessful.
DIAG_ASL_ENTER	Enter Function key was entered.
DIAG_ASL_EXIT	Exit Function key was entered.
DIAG_ASL_CANCEL	Cancel Function key was entered.
DIAG_ASL_HELP	Help Function key was entered.
DIAG_ASL_LIST	List Function key was entered.
DIAG_ASL_COMMIT	Commit Function key was entered.
DIAG_ASL_PRINT	Print Function key was entered.

diag_emsg

Purpose

Displays error messages.

Note: Diagnostic Applications (DAs) should not use this subroutine.

Syntax

```
#include    <diag/diago.h>

long  diag_emsg ( fdes, setid, msgid [,val,... ] )
nl_catd fdes;
unsigned short setid;
unsigned short msgid;
```

Description

The **diag_emsg** subroutine displays an error message. Normally used with service aids.

Parameters

fdes Open catalog file descriptor returned from the **diag_catopen()** system call.

setid Set ID of the message in the catalog.

msgid Message ID of the message in the catalog that serves as the format string.

val Values that are optional and variable in number are inserted in the specified message according to the conventions assumed by the **printf()** subroutine in the standard I/O library. The format is specified by the message referenced by the catalog set and message ID.

Return Value

The **diag_emsg** subroutine returns one of the following values:

DIAG_ASL_OK	Successful return.
DIAG_ASL_CANCEL	Cancel key was entered.
DIAG_ASL_EXIT	Exit key was entered.

diag_msg, diag_msg_nw

Purpose

Displays simple menus.

Syntax

```
#include    <diag/diago.h>

long  diag_msg ( mnum, fdes, setid, msgid [, val, ... ] )
long mnum;
nl_catd fdes;
unsigned short setid;
unsigned short msgid;
```

```
long  diag_msg_nw ( mnum, fdes, setid, msgid [, val, ... ] )
long mnum;
nl_catd fdes;
unsigned short setid;
unsigned short msgid;
```

Description

The **diag_msg** subroutine displays the specified text and obtains the user's response. The screen is automatically cleared upon completion.

The **diag_msg_nw** subroutine displays the specified text but does *not* wait for the user to respond. The screen is *not* automatically cleared.

Parameters

mnum	Menu number that is displayed, right-justified, as a hex number at the top-right corner of the screen.
fdes	Open catalog file descriptor returned from the **diag_catopen()** system call.
setid	Set ID of the message in the catalog.
msgid	Message ID of the message in the catalog that serves as the format string.
val	Values that are optional and variable in number are inserted in the specified message according to the conventions assumed by the **printf()** subroutine in the standard I/O library. The format is specified by the message referenced by the catalog set and message ID.

Return Value

The **diag_msg** subroutine returns one of the following values:

DIAG_ASL_OK	Successful return.
DIAG_ASL_CANCEL	Cancel key was entered.
DIAG_ASL_EXIT	Exit key was entered.

diag_get_device_flag

Purpose

Obtain device flag from residual data information.

Syntax

```
#include    <diag/diag.h>
#include    <sys/residual.h>

int  diag_get_device_flag (
                                 char           *device_name,
                                 long           *Flag)
```

Description

The **diag_get_device_flag** subroutine searches residual data for an object matching the device specified by **device_name**. The value of the *Flags* field as defined in the *DEVICE_ID* structure for the device is returned in the **Flag** argument.

Implementation Specifics

POWER-based

Parameters

device_name	Pointer to a character string containing the logical name of the device.
Flag	Pointer to a long integer where the value of the *Flag* field in the *DEVICE_ID* structure as defined by **sys/residual.h** header file will be written.

Return Value

Upon successful completion, a 0 is returned if the device flag information was retrieved successfully. If the **diag_get_device_flag** fails, a value of -1 is returned.

diag_get_property

Purpose

Obtain property value from Common Hardware Reference Platform (CHRP) firmware for a resource.

Syntax

```
#include    <diag/diag.h>

char  *diag_get_property (
                                 char            *device_name,
                                 char            *property_name,
                                 int             *property_length)
```

Description

The **diag_get_property** subroutine searches the Open Firmware device tree to obtain the value of a property associated with the specified resource. The resource must be a valid ODM resource name with a corresponding Open Firmware device tree node. If the resource's corresponding node is not found in the Open Firmware device tree, or if the property value is not found, then a char *NULL is returned.

Implementation Specifics

POWER-based

Parameters

device_name	Pointer to a character string containing the logical name of the device.
property_name	Pointer to a character string containing the property to find.
property_length	Contains total number of characters pointed to by the return character value.

Return Value

Upon successful completion, a character string is returned containing the value (or values) of the property requested. Multiple values may be separated by a NULL value. If the resource is not valid, or the property value is not found, then a char *NULL is returned.

diag_get_sid_lun

Purpose

Returns the SCSI ID and Logical Unit Number (LUN) from a SCSI address.

Syntax

```
#include    <diag/diag.h>

int  diag_get_sid_lun ( scsiaddr, sid_addr, lun_addr )
char  *scsiaddr;
uchar *sid_addr;
uchar *lun_addr;
```

Description

The **diag_get_sid_lun** subroutine returns the SCSI ID and logical unit number associated with a SCSI address for a device. The SCSI address must be in the format used by the *connwhere* field in **CuDv** object class.

Parameters

scsiaddr	Pointer to the address of the SCSI device. This is the *connwhere* field of the device. Format is *"x,y"* where *x* is the SCSI ID, and *y* is the logical unit number.

sid_addr Pointer to the SCSI ID of the device.
lun_addr Pointer to the logical unit number of the device.

Return Value

The **diag_get_sid_lun** subroutine returns one of the following values:

0 Successful return.
-1 Error. Incorrect format for SCSI address.

get_cpu_model

Purpose

Returns the CPU model number.

Syntax

```
#include     <diag/modid.h>

unsigned int  get_cpu_model ( model_code )
int model_code;
```

Description

The **get_cpu_model** subroutine gets the CPU model number.

init_dgodm() must be called before starting this subroutine.

Implementation Specifics

POWER-based

Parameters

model_code Attribute stored in the **CuAt** database for the sys0 model code. The unsigned integer returned by the function is the raw model code obtained from the IPL control block. Macros are defined in **modid.h**. These macros can be used to determine the following information:

Package Type
Tower, Rack, or Desktop.

Speed Low, Medium, High, or Turbo Charged.

Machine Type
Release 1, RSC, Release 2, or PowerPC.

Return Value

Upon successful completion, the model code as stored in the **iplcb** structure is returned. Otherwise, a value of -1 is returned.

get_dev_desc

Purpose

Returns the device's descriptive text.

Syntax

```
char *  get_dev_desc ( device_name )
char *  device_name;
```

Description

The **get_dev_desc** subroutine gets the descriptive text associated with the device. This text is stored in the *catalog* field of the **PdDv** entry for the device. This is usually found in the **/usr/lib/methods/devices.cat** file for most devices. Other devices may use different catalogs.

Parameters

device_name — Character pointer to the name of the device.

Return Value

Upon successful completion, a **char** pointer to a text string in memory is returned. Otherwise, a value of -1 is returned.

get_diag_att

Purpose

Reads an attribute from the predefined database **PDiagAtt**.

Syntax

```
#include    <diag/modid.h>

int get_diag_att ( type, attribute, conversion, byte_count, value )

char *type;
char *attribute;
char conversion;
int  *byte_count;
void *value;
```

Description

The **get_diag_att** subroutine gets attributes from the predefined diagnostic database **PDiagAtt**.

Parameters

The arguments are defined as follows:

type — Device type, which should be *Class/SubClass/Type* string. This fully qualified string reduces the chance of finding two objects having the same *Type* value in the **PdDv** object class.

attribute — Attribute name to get from the Predefined Attribute Object Class.

conversion — The data type to which the attribute is to be converted, including the following:

′s’ = string
rep = s

′b’ = byte sequence
rep = s (for example ″0x56FFE67″)

′l’ = long
rep = n

′i’ = int rep = n

′h’ = short (half)
rep = n

′f’ = float
rep = n

′c’ = char
rep = n, or s

′a’ = address
rep = n

byte_count — Number of bytes (for byte sequence only).

value Pointer to where the converted attribute value is returned.

Return Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned.

dlog_close

Purpose

Closes the Diagnostic Event Log opened by **dlog_open**.

Syntax

```
#include                <diag/diag_log.h>

int dlog_close(dl_info *info)
```

Description

The **dlog_close** subroutine closes the log file opened with dlog_open. It will also free the memory allocated with dlog_open.

Parameters

info Pointer to structure of format:

```
 typedef struct _log_info {
    int fd;                          /* File descriptor                 */
    int lockId;                      /* ODM Lock id                     */
    dl_att *dlAtt;                   /* Pointer to log attributes       */
    dl_einfo *dlArray;               /* Pointer to log array            */
} dl_info;

typedef struct _log_einfo {
    int version;                     /* Entry Version                   */
    char logType;                    /* Log Type - I,S,N,E,X            */
    unsigned int size;               /* Entry Size                      */
    unsigned int offset;             /* Offset from the file's beginning  */
} dl_einfo;

typedef struct _log_att {
    int version;                     /* Version                         */
    unsigned int numEntries;         /* number of log entried           */
    unsigned int lastIndex;          /* index of latest entry           */
    unsigned int nextSeqNum;         /* sequence number of next log entry */
    unsigned int maxLogSize;         /* maximum size of log             */
    unsigned int arrayOffset;        /* array offset                    */
    unsigned int wrapCount;          /* number of times file has wrapped  */
} dl_att;
```

Return Value

Upon successful completion, 0 is returned. Otherwise, a value of -1 is returned.

dlog_find_first

Purpose

Finds the first diagnostic log entry that matches the specified criteria.

Syntax

```
#include                <diag/diag_log.h>

int dlog_find_first(dl_info *dlogInfo,char *criteria,dlSearch *filter,dlEntry **results)
```

Description

The **dlog_find_first** subroutine finds the first diagnostic log entry that matches the specified criteria. It also parses the search criteria and uses this to initialize the dlSearch structure for subsequent searches. It allocates memory for the matching entry, and returns the array index of the matching entry. It is the responsibility of the calling application to free the memory allocated for dlEntry.

Parameters

dlogInfo	Pointer to log information in dl_info
criteria	search criteria consisting of any of the following: -d *device_name* -n *dlog_sequenceNumber* -L *deviceLocation* -t *entryType* -i *dlog_EntryIdentifier* -s *startTime* (format MMddhhmmyy) -e *endTime* (format MMddhhmmyy)
filter	parsed search criteria
results	pointer to entry matching the search criteria

Call this function before calling dlog_find_next.

Return Value

Upon successful completion, a value >= 0 is returned. Otherwise, a value of -1 is returned.

dlog_find_next

Purpose

Finds the first diagnostic log entry that matches the specified criteria.

Syntax

```
#include                <diag/diag_log.h>

int dlog_find_next(dl_info *dlogInfo,int index,dlSearch *filter,dlEntry **results)
```

Description

The **dlog_find_next** subroutine finds the first diagnostic log entry that matches the specified search filter. It allocates memory for the matching entry, and returns the array index of the matching entry. It is the responsibility of the calling application to free the memory allocated for dlEntry.

Parameters

dlogInfo	Pointer to log information in dl_info
index	starting index
filter	parsed search criteria
results	pointer to entry matching the search criteria

Call this function after calling dlog_find_first.

Return Value

Upon successful completion, a value >= 0 is returned. Otherwise, a value of -1 is returned.

dlog_find_sequence

Purpose

Finds the diagnostic log entry that has the specified sequence number.

Syntax

```
#include                <diag/diag_log.h>

int dlog_find_sequence(dl_info *dlogInfo,uint seq,dlEntry **results)
```

Description

The **dlog_find_sequence** subroutine finds the diagnostic log entry with a specific diagnostic log sequence number. The matching entry will be in results and its index in the log array will be returned. It is the responsibility of the calling application to free the memory allocated for dlEntry. The results variable will be NULL if no match is found.

Parameters

dlogInfo	Pointer to log information in dl_info
seq	sequence number
results	pointer to entry with the specified sequence number

Return Value

Upon successful completion, a value >= 0 is returned. Otherwise, a value of -1 is returned and results will be NULL.

dlog_formatElogResults

Purpose

Returns a formatted string of the diagnostic event log information.

Syntax

```
#include                <diag/diag_log.h>

char *dlog_formatElogResults(dlEntry *entry)
```

Description

The **dlog_formatElogResults** subroutine formats a diagnostic log entry for display in the error log with the **errpt** command. When a SRN is caused by an entry in the error log, the error log is updated with the diagnostic log entry's sequence number. When the error log is displayed the formatted string returned from this subroutine shows the diagnostic log information. It is up to the calling application to free the memory allocated for the return string.

The return string will look like the following:

```
Diagnostic Log sequence number: sequence number
Resource tested:      resource name
Resource Description: resource description
Location:             resource location
SRN:   SRN
Description:  Error Description
Possible FRUs: List of possible FRUs
```

Parameters

dlogEntry	Pointer to diagnostic log entry

Return Value

Upon successful completion a NON-ZERO pointer is returned. Otherwise, a pointer to NULL is returned.

dlog_freeEntry

Purpose

Frees memory allocated for diagnostic log entry.

Syntax

```
#include                <diag/diag_log.h>

int dlog_freeEntry(int version, void *dlogEntry)
```

Description

The **dlog_freeEntry** subroutine frees all the memory allocated for the specified entry. The version determines which entry structure is being passed.

Parameters

version	Entry version (LATEST_ENTRY_VER is the latest version that corresponds to the dlEntry structure)
dlogEntry	Pointer to diagnostic log entry

Return Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned.

dlog_getTestMode

Purpose

Return the value of the **dlog_testmode** attribute in **CDiagAtt** for the specified device.

Syntax

```
#include                <diag/diag_log.h>
int dlog_getTestMode(char *name)
```

Description

The **dlog_getTestMode** subroutine gets a **CDiagAtt** object for the specified device with an attribute of **dlog_testmode**. The value of the **dlog_testmode** is returned.

Parameters

name	Character pointer to the name of the device

Return Value

Upon successful completion, the test mode is returned. Otherwise, `-1` is returned if the object does not exist.

dlog_open

Purpose

Read an entry from the Diagnostic Event Log at the specified offset.

Syntax

```
#include                <diag/diag_log.h>

int dlog_open(char *pathname,dl_info **info)
```

Description

The **dlog_open** subroutine opens the specified log file for reading. If the pathname is NULL, then the default diagnostic log file will be used. This subroutine also allocates memory for the dl_info structure and initializes the structure.

Parameters

pathname	Name of log to open (if NULL, the default log is used)

info Pointer to structure of format:

```
 typedef struct _log_info {
    int fd;                         /* File descriptor                 */
    int lockId;                     /* ODM Lock id                     */
    dl_att *dlAtt;                  /* Pointer to log attributes       */
    dl_einfo *dlArray;              /* Pointer to log array            */
} dl_info;

typedef struct _log_einfo {
    int version;                    /* Entry Version                   */
    char logType;                   /* Log Type - I,S,N,E,X            */
    unsigned int size;              /* Entry Size                      */
    unsigned int offset;            /* Offset from the file's beginning  */
} dl_einfo;

typedef struct _log_att {
    int version;                    /* Version                         */
    unsigned int numEntries;        /* number of log entried           */
    unsigned int lastIndex;         /* index of latest entry           */
    unsigned int nextSeqNum;        /* sequence number of next log entry */
    unsigned int maxLogSize;        /* maximum size of log             */
    unsigned int arrayOffset;       /* array offset                    */
    unsigned int wrapCount;         /* number of times file has wrapped  */
} dl_att;
```

Return Value

Upon successful completion, 0 is returned. Otherwise, a value of -1 is returned.

dlog_read

Purpose

Read an entry from the Diagnostic Event Log at the specified offset.

Syntax

```
#include                <diag/diag_log.h>

dlEntry *dlog_read(dl_info *dlogInfo,int index)
```

Description

The **dlog_read** subroutine will read a Diagnostic Event Log entry at the specified offset, which is determined from the index. It will return a pointer to a structure of format:

```
typedef struct _logEntry {
        char type;                   /* Log Type */
        char identifier[5];          /* Diagnostic Log identifier */
        unsigned int el_identifier;  /* Error log identifier */
        int timestamp;
        unsigned int seqNum;         /* order in which event is logged */
        unsigned int el_seqNum;      /* Error log sequence number */
        unsigned int    session;     /* Diag Session's PID */
        unsigned int    testMode;    /* Diagnostics test mode - hex value*/
        resource_t *res_p;           /* Resource information */
        int resSize;                 /* Size of resource info */
        void *errorInfo;             /* Error information */
        int errorSize;               /* Size of error info */
} dlEntry;

typedef struct resource {
    char name[NAME_SIZE];
    int locSize;
    char *location;                 /* Logical or Physical              */
```

```
    short set;
    short msg;
    char catName[NAME_SIZE];
} resource_t;
```

Parameters

dlogInfo Pointer to structure of format:

```
typedef struct _log_info {
    int fd;                         /* File descriptor                  */
    int lockId;                     /* ODM Lock id                      */
    dl_att *dlAtt;                  /* Pointer to log attributes        */
    dl_einfo *dlArray;              /* Pointer to log array             */
} dl_info;

typedef struct _log_einfo {
    int version;                    /* Entry Version                    */
    char logType;                   /* Log Type - I,S,N,E,X             */
    unsigned int size;              /* Entry Size                       */
    unsigned int offset;            /* Offset from the file's beginning */
} dl_einfo;

typedef struct _log_att {
    int version;                    /* Version                          */
    unsigned int numEntries;        /* number of log entried            */
    unsigned int lastIndex;         /* index of latest entry            */
    unsigned int nextSeqNum;        /* sequence number of next log entry */
    unsigned int maxLogSize;        /* maximum size of log              */
    unsigned int arrayOffset;       /* array offset                     */
    unsigned int wrapCount;         /* number of times file has wrapped */
} dl_att;
```

index index into Diagnostic Event log array for specific entry

Return Value

Upon successful completion, a pointer to dlEntry is returned. Otherwise, a value of NULL is returned.

dlog_same_elogId

Purpose

Determines if a diagnostic log entry has a specific error log identifier.

Syntax

```
#include                <diag/diag_log.h>

int dlog_same_elogId(dlEntry *dlogEntry,uint el_identifier)
```

Description

The **dlog_same_elogId** subroutine determines if the specified entry has the same error log identifier as the given error log identifier.

Parameters

dlogEntry Pointer to diagnostic log entry
el_identifier Error log identifier

Return Value

If the entry has the same error log identifier, a value of 1 is returned. Otherwise, a value of 0 is returned.

dlog_setEntryType

Purpose

Returns the entry type for a given diagnostic log identifier

Syntax

```
#include              <diag/diag_log.h>
int dlog_setEntryType(char *id)
```

Description

The **dlog_setEntryType** subroutine will return an entry type for the specified entry identifier. The following entry types are defined:

INFO	Informational Type
NTF	No Trouble Found
ERR	Error
SRN	Srn Callout
EXER	Exerciser Error
SA	Service Aid

Parameters

id	entry identifier

Return Value

Upon successful completion, the entry type is returned. Otherwise, a value of -1 is returned.

dlog_write

Purpose

Write a diagnostic event to the Diagnostic Event Log.

Syntax

```
#include              <diag/diag_log.h>
int dlog_write(dlEntry *entry)
```

Description

The **dlog_write** subroutine writes a diagnostic event to the Diagnostic Event Log.

Parameters

entry Pointer to a structure of type **dlEntry**, which is defined as follows:

```
 typedef struct _logEntry {
        char type;                      /* Log Type */
        char identifier[5];             /* Diagnostic log identifier */
        unsigned int el_identifier;     /* Error log identifier */
        int timestamp;
        unsigned int seqNum;            /* order in which event is logged */
        unsigned int el_seqNum;         /* Error log sequence number */
        unsigned int    session;        /* Diag Session's PID */
        unsigned int testMode;          /* Diagnostics test mode - hex value*/
        resource_t *res_p;              /* Resource information */
        int resSize;                    /* Size of resource info */
        void *errorInfo;                /* Error information */
        int errorSize;                  /* Size of error info */
} dlEntry;

typedef struct resource {
        char    name[NAME_SIZE];
        int     locSize;
        char   *location;
        short   set;
        short   msg;
        char    catName[NAME_SIZE];
}resource_t;
```

Return Value

The dlog_write routine returns one of the following values:

0	Successful
-1	Unsuccessful
ERROR_FS	Indicates the /var filesystem is full

save_davars_ela

Purpose

Formats SRN and create DAVars object with error log information.

Syntax

```
#include <diag/diag_log.h>
int save_davars_ela(struct fru_bucket *frub,uint el_seq,uint el_id,uint
errorCode)
```

Description

The **save_davars_ela** subroutine formats the SRN if the errorCode is 0, and create a DAVars object containing the error log information. The format of the DAVars object is:

DAVars:
 dname = *ResourceName*
 vname = ″ErrorLog*SRN_or_ErrorCode*″
 vtype = 0
 vvalue = ″*ErrorlogIdentifier*,*ErrorlogSequenceNumber*″
 ivalue = 0

An example of a DAVars object is:

DAVars:
 dname = ″hdisk0″

vname = ″ErrorLog689-130″
vtype = 0
vvalue = ″1581762B,74″
ivalue = 0

Parameters

*frub	Pointer to fru bucket
el_seq	Error log sequence number
el_id	Error log identifier
errorCode	Error code (if 0, format the SRN)

Return Value

The subroutine returns a value of 0 on success; a value of -1 on failure.

copy_text

Purpose

Format text to fit on line with a length of 74

Syntax

```
int copy_text( int string_length, char *buffer, char *text )
```

Description

The **copy_text** subroutine will take the text string and add `\n` so that the string can be displayed without wrapping.

Parameters

string_length	Starting column for the formatted string
buffer	Formatted string
text	Unformatted string

Return Value

A value of `0` is returned.

DA_SETRC_XXXXXX, DA_CHECKRC_XXXXXX, DA_EXIT

Purpose

Processes Exit Status of Diagnostic Application (DA).

Syntax

```
#include   <diag/diag_exit.h>


#define DA_SETRC_STATUS(VAL)     da_exit_code.field.status = (VAL)
#define DA_SETRC_USER(VAL)       da_exit_code.field.user = (VAL)
#define DA_SETRC_ERROR(VAL)      da_exit_code.field.error = (VAL)
#define DA_SETRC_TESTS(VAL)      da_exit_code.field.tests = (VAL)
#define DA_SETRC_MORE(VAL)       da_exit_code.field.more = (VAL)
#define DA_CHECKRC_STATUS()      da_exit_code.status
#define DA_CHECKRC_USER()        da_exit_code.user
#define DA_CHECKRC_ERROR()       da_exit_code.error
#define DA_CHECKRC_TESTS()       da_exit_code.tests
#define DA_CHECKRC_MORE()        da_exit_code.more
#define DA_EXIT()                exit(*( (char*) &da_exit_code) ) )

enum diag_enum_status {
       DA_STATUS_GOOD,
```

```
/* No hardware problems were found */

       DA_STATUS_BAD,
/* A hardware problem was found */

       };
enum diag_enum_user {
       DA_USER_NOKEY,
/* No special function keys were entered */

       DA_USER_EXIT,
/* The user entered the exit key */

       DA_USER_QUIT,
/* The user entered the cancel key */

       };
enum diag_enum_error {
       DA_ERROR_NONE,
/* No software errors were encountered */

       DA_ERROR_OPEN,
/* The Device Driver failed to open */

       DA_ERROR_OTHER,
/* Another software error was encountered */

       };
enum diag_enum_tests {
       DA_TEST_NOTEST,
/* No diagnostic tests were run */

       DA_TEST_FULL,
/* The full tests were run */

       DA_TEST_SHR,
/* The shared tests were run */

       DA_TEST_SUB,
/* The sub tests were run */

       };
enum diag_enum_more {
       DA_MORE_NOCONT,
/* The problem has been isolated. */

       DA_MORE_CONT,
/* The parent or sibling will be tested next */

       };

typedef struct {
      unsigned status : 1;
/* enum diag_enum_status */

      unsigned user   : 2;
/* enum diag_enum_user */

      unsigned error  : 2;
/* enum diag_enum_error */

      unsigned tests  : 2;
/* enum diag_enum_tests */

      unsigned more   : 1;
```

```
/* enum diag_enum_more */

      } da_return_code_t;
extern da_returncode_t da_exit_code;
```

Description

The **DA_EXIT** macro is used to exit a DA. To set a value other than the default, the appropriate **DA_SETRC_XXXXX** macro must be called. To check the current value, use the appropriate **DA_CHECKRC_XXXXXX** macro.

The defaults settings are:

DA_STATUS_GOOD

DA_USER_NOKEY

DA_ERROR_NONE

DA_TEST_NOTEST

DA_MORE_NOCONT

Parameters

Takes no parameters.

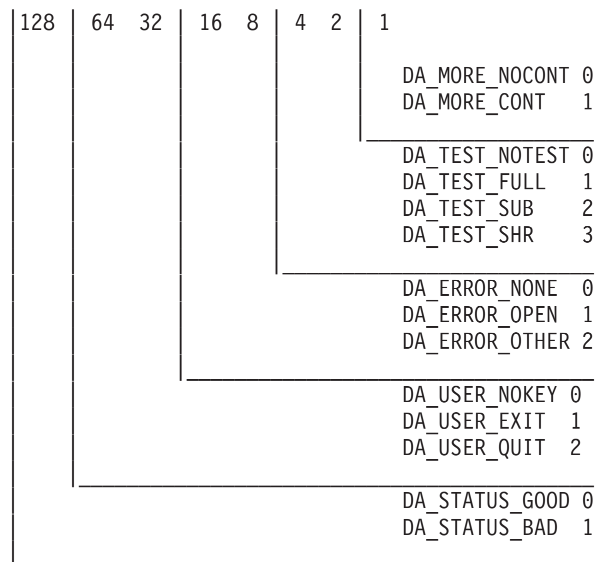
Return Value

There is no return code.

Structure Deciphering

Following is a easy chart to use to deciphered the bit positions:

```
Bit position
```



diag_asl_beep

Purpose

Rings the bell.

Syntax

```
#include     <diag/diago.h>

long  diag_asl_beep (  )
```

Description

The **diag_asl_beep** subroutine is used to ring the bell. Can be used to indicate that input is not valid.

Parameters

Takes no parameters.

Return Value

Upon successful completion, a value of 0 is returned.

diag_asl_execute

Purpose

Executes an application.

Syntax

```
#include         <diag/diago.h>
long  diag_asl_execute ( command, options, exit_status )
char *command;
char *options;
int  *exit_status;
```

Description

The **diag_asl_execute** subroutine forks and executes an application while preserving the state of the ASL interface.

Parameters

command	Command or application to run.
options	Character array, starting with the command, followed by any command arguments, ending with a NULL.
exit_status	Exit status returned from the command.

Return Value

The following values are returned:

0	Successful return.
DIAG_ASL_FAIL	Error occurred.

diag_exec_source

Purpose

Returns an indication of where diagnostics is being run from.

Syntax

```
int diag_exec_source ( mount_point )
char *mount_point;
```

Description

The **diag_exec_source** determines where the diagnostics program is run from. If not from hard file, then *mount_point* contains the directory where the file system resides (CDRFS).

Parameters

mount_point	Character pointer to directory name where the file system resides.

Return Value

The **diag_exec_source** subroutine returns one of the following values:

0 Running from hardfile.
1 Running from CD-ROM.

diag_execute

Purpose

Executes an application. Does not depend on ASL initialization.

Syntax

```
#include    <diag/diago.h>

long  diag_execute ( command, options, exit_status )
char *command;
char *options;>
int  *exit_status;
```

Description

The **diag_asl_execute** subroutine forks and executes an application. This subroutine does not depend on ASL initialization, and it does not preserve the state of ASL.

Parameters

command	Command or application to run.
options	Character array, starting with the command, followed by any command arguments, ending with a NULL.
exit_status	Exit status returned from the command.

Return Value

The following values are returned:

0 Successful return.
-1 Error occurred.

dt

Purpose

Writes diagnostic trace information to a file.

Syntax

```
#include     <diag/diag_trace.h>

void dt ( dt_id, dt_type [,val, ...])
char *dt_id;
int  dt_type;
```

Description

The **dt** subroutine allows trace information to be written to a file. If the file **/tmp/.DIAG_TRACE** exists, trace information will be written to a file specified by the **dt_id** argument. The default is to overwrite existing trace information. To append to the trace file, **export DIAG_TRACE=APPEND**.

Parameters

dt_id	Used to uniquely identify the trace file. The resulting trace file will be called .dt.'dt_id' in the **/tmp** directory.

dt_type The type of trace function to perform.

DT_TMI
Trace initialization for Diagnostic Applications (DA). Information from the **TMInput** structure will be written to the trace file.

DT_BEGIN
Trace initialization for Service Aids (SA).

DT_DEC
Trace an integer variable in decimal.

DT_MDEC
Trace multiple integer variables in decimal.

DT_HEX
Trace an integer variable in hexadecimal.

DT_MHEX
Trace multiple integer variables in hexadecimal.

DT_LDEC
Trace a long integer variable in decimal.

DT_MLDEC
Trace multiple long integer variables in decimal.

DT_LHEX
Trace a long integer variable in hexadecimal.

DT_MLHEX
Trace multiple long integer variables in hexadecimal.

DT_MSTR
Trace multiple string variables.

DT_MSG
Trace a simple message such as ″hello.″

DT_BUFF
Trace a data buffer.

DT_SCSI_TUCB
Trace SCSI TUCB structure information.

DT_SCSI_TUCB_SD
Trace SCSI TUCB Sense Data information.

DT_END
Write ″end of trace″ identifier to the trace file.

val Variable arguments which may include the number of multiple variables to trace, the trace labels, and the information to trace.

Return Value

There is no return code.

error_log_get

Purpose

Returns error-log entries.

Syntax

```
#include     <diag/diag.h>

int  error_log_get ( option, criteria, err_data )
int option;
char  *criteria;
struct errdata *err_data;
```

Description

The **error_log_get** subroutine allows the Diagnostic Application (DA) to query the error log for entries.

Implementation Specifics

The **NVRAMEL** option is only supported on the POWER-based platform:

Parameters

option
: Describes the operation to be performed. The following values are defined:

INIT
: Initializes error log retrieve.

SUBSEQ
: Gets next error-log entry.

TERMI
: Ends error log retrieve.

NVRAMEL
: Use the NVRAM error log as the source for the error log retrieve. Only the following members of struct errdata are available when the error log is obtained from NVRAM:

- *time_stamp*
- *err_id*
- *resource*
- *detail_data_len*
- *detail_data*

Note: This option is only supported on the POWER-based platform.

criteria
: Used with the **INIT** option to specify which device to obtain the error log data for and how far back to search. This parameter can be set to any valid option used by the **errpt** command.

When used with the **NVRAMEL** option, this can be either a list of resource names (with the **-N** switch) or an error ID (with the **-j** switch), but not both.

struct errdata
: Data type that contains the following data filled in for use by the DA.

```
struct errdata {
      unsigned   sequence;       /* sequence number of entry  */
      unsigned   time_stamp;     /* entry time stamp  */
      unsigned   err_id;         /* error ID code  */
      char       *machine_id;    /* machine ID  */
      char       *node_id;       /* node  */
      char       *class;         /* H=hardware, S=software  */
      char       *type;          /* PERM,TEMP,PERF,PEND,UNKN  */
      char       *resource;      /* Configured device name  */
      char       *vpd_data;      /* VPD info  */
      char       *conn_where;    /* connwhere field of CuDv  */
      char       *location;      /* location field of CuDv  */
      unsigned detail_data_len;  /* length of detail data  */
      char     *detail_data;     /* detail data  */
```

Return Value

Return values are dependent on the option performed:

INIT
: 0: No error

	1: Error-log entry available -1: Error obtaining data
SUBSEQ	0: No more entries available 1: Error-log entry available
TERMI	0: Terminate successful
NVRAMEL	0: No entries matching criteria 1: Error-log entry available -1: Error accessing NVRAM -2: Invalid criteria

file_present

Purpose

Returns status indicating whether the file is present on the file system.

Syntax

```
int  file_present ( filename )
char *filename;
```

Description

The **file_present** subroutine determines the presence of a file.

Parameters

filename Character pointer to full path name of file.

Return Value

The **file_present** subroutine returns one of the following values:

0 File is not present.
1 FIle is present.

get_DApp

Purpose

Returns the *DApp* value associated with device as represented in the **PDiagAtt** object class.

Syntax

```
char *get_DApp ( devicename, attribute )
char *devicename;
char *attribute;
```

Description

The **get_DApp** subroutine returns the *DApp* value from the **PDiagAtt** object class associated with the given device and attribute. Search criteria is in the following order:

1. DClass and DSClass and DType and attribute
2. DClass and DSClass and attribute
3. DClass and attribute

The calling application is responsible for freeing the storage allocated for the returned value.

Parameters

devicename	Character pointer to customized name of device.
attribute	Character pointer to attribute associated with device.

Return Value

The **get_DApp** subroutine returns one of the following values:

char * NULL	Device and attribute is not found.
char *DApp	Pointer to char string containing DApp value.

getdainput, clrdainput

Purpose

Gets and clears the input for the Diagnostic Application (DA).

Syntax

```
#include    <diag/tm_input.h>

int getdainput ( tm_input )
struct tm_input *tm_input;

int clrdainput ( )
```

Description

The **getdainput** subroutine gets the input for the DA from the **TMInput** object class. The **clrdainput** subroutine clears the **TMInput** object class.

Parameters

tm_input	Pointer to the structure where the data should be written.

Return Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned.

getdavar, putdavar

Purpose

Gets and puts persistent variables.

Syntax

```
#include    <diag/diag.h>

int getdavar ( dname, vname, vtype, val )
char  *dname, *vname, *val;
unsigned short vtype;

int putdavar ( dname, vname, vtype, val )
char  *dname, *vname, *val;
unsigned short vtype;
```

Description

The **getdavar** subroutine gets the persistent variable *vname* from the Diagnostic Application Variables object class. The **putdavar** subroutine is used to save the specified value.

Parameters

dname	Name of the device with which the variable is associated.

vname Name of the variable.

vtype Type of the variable. The following values are defined:

DIAG_STRING
: The variable should be treated as a character string.

DIAG_INT
: The variable should be treated as an integer.

DIAG_SHORT
: The variable should be treated as a short.

val Location where the variable should be written when the subroutine **getdavar** is called. Otherwise, *val* points to the value to be saved.

Return Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned.

ipl_mode

Purpose

Returns the state of the diagnostic IPL mode.

Syntax

```
#include    <diag/diag.h>

int ipl_mode ( source )
int source;
```

Description

The **ipl_mode** subroutine returns the state of the diagnostic IPL mode, and the IPL source.

Parameters

source Set according to IPL source:

If the value of the environment variable DIAG_IPL_SOURCE is **NULL** or **IPL_SOURCE_DISK** or **IPL_SOURCE_LAN**, then the value of *source* will be set to **DIAG_FALSE** (0).

If the value of the environment variable DIAG_IPL_SOURCE is **IPL_SOURCE_CDROM** or **IPL_SOURCE_TAPE**, then the value of *source* will be set to **DIAG_TRUE** (1).

Return Value

The **ipl_mode** subroutine returns one of the following values:

1 EXENV_IPL	Diagnostics invoked during IPL
2 EXENV_STD	Standalone Diagnostics, Online Service, or Online Maintenance
4 EXENV_CONC	Online Concurrent Diagnostics

getELAdates

Purpose

Return the start and end timestamp for retrieving error log entries.

Syntax

```
char *getELAdates ( notRTOmode )
int notRTOmode;
```

Description

The **getELAdates** subroutine formats and returns a string containing the start and end timestamp that should be used for error log analysis. The end timestamp is the current date and time. The start timestamp is created using either the value specified by the Customized Diagnostic Attribute for run time options, or the value passed as a parameter. The string returned serves the same purpose as the date parameter of the TMInput object class.

Parameters

notRTOmode
: Determines how the run time option value for the number of ELA days is used. If notRTOmode is 0, then the number of ELA days specified by the Customized Diagnostic Attribute for run time options is used to create the start timestamp. If notRTOmode is greater than 0, then the notRTOmode value is used as the number of ELA days when creating the start timestamp.

Return Value

The following string is returned:

```
-s MMddhhmmyy -e MMddhhmmyy
```

where:

dd is the 2-digit value for day

hh is the 2-digit value for the hour in 24-hour format

mm is the 2-digit value for minutes

yy is the 2-digit value for year

has_diag_authority

Purpose

Checks if a user has the proper authority to run diagnostics.

Syntax

```
int has_diag_authority  chk_shutdown;
```

Description

The **has_diag_authority** subroutine checks if the user is authorized to run diagnostics.

Parameters

chk_shutdown	If TRUE, the subroutine checks to see if the user is authorized to shut down the system.

Return Value

0	User is not authorized to run diagnostics.
1	User is authorized to run diagnostics.

menugoal

Purpose

Concludes a Text Goal.

Syntax

```
int menugoal ( msg )
char *msg;
```

Description

The **menugoal** subroutine associates a menu goal with the device being tested. The **TMInput** object class identifies the device currently being tested.

Parameters

msg
: Pointer to a text string that identifies a repair action intended for the customer, not a trained service representative. The *msg* parameter should contain a six-digit hex number (menu number) at the front of the buffer, followed by a space, and then the title line. Everything after the first carriage return is considered menu text.

Return Value

Upon successful completion, a value of 0 is returned. If the **menugoal** subroutine fails, then a value of -1 is returned.

schedule_ela

Purpose

Schedule ELA for a device.

Syntax

```
int schedule_ela ( device, minutes )

char *device;

int minutes;
```

Description

This routine is used to schedule Error Log Analysis (ELA) for a given device. Typically, this would be used by a Diagnostic Application to schedule ELA when processing indicates that an error log entry is expected and necessary for completing the diagnostic conclusion.

The scheduled time is the current time plus the number of minutes given as input. The number of minutes is limited to 24 hours. The scheduled ELA event, similar to using the **diag -c -e -d** device command, occurs one time only.

Parameters

device
: The device name for which ELA should be run.
Example: `sysplanar0`

minutes
: The number of minutes that is added to the current time to schedule ELA to run. Any value over 24 hours is truncated to a value less than 24 hours.
Example: `24 hours and 35 minutes (1475 minutes)` is truncated to `35 minutes`.

Return Value

There is no error return. Always returns `0`.

Diagnostic Object Classes

The Diagnostic Package contains ODM object classes that are used extensively by the Diagnostic components. Some object classes store 'predefined' diagnostic information about the system and resources. Other object classes store 'customized' information that is built and used during runtime operation of diagnostics.

The following is a list of the Diagnostic ODM object classes:

- PDiagRes - Predefined Diagnostic Resource Object Class
- PDiagAtt - Predefined Diagnostic Attribute Device Object Class

- PDiagTask - Predefined Diagnostic Task Object Class
- CDiagAtt - Customized Diagnostic Attribute Object Class
- TMInput - Test Mode Input Object Class
- MenuGoal - Menu Goal Object Class
- FRUB - Fru Bucket Object Class
- FRUs - Fru Reporting Object Class
- DAVars - Diagnostic Application Variables Object Class
- PDiagDev - Predefined Diagnostic Devices Object Class
- DSMOptions - Diagnostic Supervisor Menu Options Object Class

Predefined Diagnostic Resource Object Class

The **Predefined Diagnostic Resource** object class (**PDiagRes**) identifies the resources supported by diagnostics and provides additional information needed to test the resource.

The **PDiagRes** object class structure is defined as:

```
class PDiagRes {
                    char Uniquetype[48];
                    short Ports;
                    short PSet;
                    short PreTest;
                    char AttUniquetype[48];
                    short SupTests;
                    short Menu;
                    short DNext;
                    vchar DaName[255];
                    char PkgBlock[5];
                    vchar EnclDaName[255];
                    vchar SysxApp[255];
                    vchar SupTasks[255];
                    long FFC;
                    short Fru;
                    long TestSuiteId;
                    long DiagEnvironment;
                    vchar KernExt[255];
                    char Version[5];
                    };
```

Uniquetype
Predefined device ″class/subclass/type.″

Ports
Indicates if the device will be represented in the Resource Selection menu by its children. The intent is to use device names that are well known to the user (for example, printers rather than serial ports). The values are as follows:

DIAG_NO (0)
Child devices should not be defined.

DIAG_YES (1)
Child devices should be defined.

When determining whether a child device should be defined, consider whether the device is self-determining. Will the **mkdev** command be unsuccessful if the device is not really there?

PSet
Identifies the message set in either **dcda.cat** or the diagnostic application catalog file reserved for the device. If the *Ports* field is not equal to 0, the first message in the set describes the adapter port. This adapter text is used in place of the real device text so that the customers are not misled into thinking that they have devices that are not actually present. The additional messages are used for reason-code text, which the DAs name when reporting FRUs.

The diagnostic application catalog file should be used by all diagnostic applications integrated into the Diagnostic Package. This capability allows for greater flexibility in installing and maintaining the diagnostic code. To use this catalog file, set bit **DIAG_DA_SRN** in the *Menu* field.

PreTest
Indicates that the device should be tested before the system is brought up. Pretest occurs when the system is initial-program loaded with the key in service position. The keyboard device, native serial ports, and display adapters are normally pretested.

AttUniquetype
The device class/subclass/type of the child device to define when the *Ports* field is set. The device named should include a set of device drivers that contain support for diagnostics.

SupTests
Identifies the types of tests supported by the DA. See Staging the Impact of Diagnostics for more information. More than one of the following types may be specified:

SUPTESTS_SHR (0x0001)
Shared tests are supported.

SUPTESTS_SUB (0x0002)
Sub-tests are supported.

SUPTESTS_FULL (0x0004)
Full-tests are supported.

SUPTESTS_MS1 (0x0008)
An optional procedure that determines why the device was *not* detected. This procedure is typically specified for devices that have external power supplies. This procedure is associated with the first selection at the Missing Resource menu.

SUPTESTS_MS2 (0x0010)
An optional procedure that performs device-specific actions when a device is removed. For example, the DA might notify a subsystem (LVM) that a physical resource (disk) has been removed. Or the DA might provide warning about deleting a device. If this procedure is *not* specified, the Diagnostic Controller deletes the device. If it is specified, the DA should delete the device. Devices are deleted by calling the device's Undefine Method. This procedure is associated with the second selection at the Missing Resource menu.

Menu
Identifies the diagnostic menus in which the device should be included. The values are as follows:

DIAG_DTL (0x0001)
The Diagnostic Test List menu.

DIAG_NOTDLT (0x0002)
Indicates that the device should not be allowed to be deleted from the Diagnostic Test List menu; for example, the VME adapters in the external display enclosure.

DIAG_DS (0x0004)
Indicates that the device should be included in the Diagnostic Selection menu.

DIAG_CON (0x0008)
Indicates that the device should be put in the Resource Selection menu if no children are attached; otherwise, the child device is put in the menu and the named device is not.

DIAG_DA_SRN (0x0010)
Indicates that the device's SRN text resides in the diagnostic applications catalog file.

DNext
: Indicates the resource to be tested next. The values are as follows:

DIAG_PAR (0x0001)
: The parent resource.

DIAG_SIB (0x0002)
: A sibling resource.

DaName
: The name of the DA associated with the device.

PkgBlock
: The block number that includes the DA associated with the device for the Removable Media Diagnostic package. This value should be an ″S″ if the DA is on a Supplemental Diskette, or a ″3S″ if the DA is a graphics adapter that can be used as a console device.

EnclDaName
: This field names a DA that provides missing-device analysis for an enclosure that is not explicitly represented in the device configuration, but that needs to be processed before the missing device. Many enclosures have their own problem-determination procedures for checking cabling, power, idiot lights, and so on, and frequently, it is helpful to know if a sibling of the missing device in the same enclosure is available.

The specification of a separate DA to missing-device diagnostics for devices not represented (for example, external enclosures or drawers) centralizes this logic in a single command instead of distributing it among each DA supporting a device that can operate in a bridge box or drawer.

For most devices, this field is null. The Diagnostic Controller calls the *EnclDaName* field*,* if the user indicates that the device has *not* been moved or turned off. The *EnclDaName* field is called before *DaName*.

SysxApp
: Identifies the application to invoke that performs a system exerciser function for this resource. While not currently used, this is a reserved field, and should be left blank.

SupTasks
: Reserved. See section ″Predefined Diagnostic Attribute Device Object Class″ for setting which tasks are supported by the resource.

FFC
: Failing Function Code for the resource. (may be used to override the **PdDv** *led* value)

Fru
: Field Replaceable Unit indicator. (may be used to override the **PdDv** *fru* value):

0 No-Fru

1 Self-FRU

2 Parent-FRU

3 Hybrid - Could be integrated or nonintegrated device.

TestSuiteId
: Bit mask indicating test suite this resource is a member of:

Bit	**Resource**
1	Base system (planars, memory, etc.)
2	I/O Device (keyboard, mouse, etc.)
4	Asynchronous Device
8	Graphics
16	SCSI Adapters
32	Storage Device (disks, diskettes, tapes, etc.)
64	Commo
128	Multimedia
256	Miscellaneous Devices

DiagEnvironment — Bit mask indicating various test mode environments this resource is capable of running in:

Bit	**Environment**
1	Supports Diagnostics in concurrent mode
2	Supports ELA
4	LFT Device (should not be run with X)
8	Group Member, set if this resource is part of a conglomerate group, such as memory, or SIMMS.
16	Resource supports ELA in concurrent mode only
32	Resource is not supported under WEBDIAG mode.
1024	The kernel extensions listed in KernExt are supported on the 64-bit kernel.

KernExt — ',' separated list of kernel extensions to load for this resource. Each kernel extension may be preceded by a platform type to indicate the platform that the kernel extension should be loaded on. For example, *chrp:device_kext, pdiagex* would indicate to always load *pdiagex*, and to conditionally load *device_kext* only on a 'chrp' platform. The platform name is derived as the output from the **bootinfo -p** command.

Version — Version change number for this resource stanza. This value should be 1.0.

Note: All values can be found in header files under **/usr/include/diag** directory.

Predefined Diagnostic Attribute Device Object Class

The **Predefined Diagnostic Attribute Device** object class (**PDiagAtt**) contains device- specific attributes for the DAs, diagnostic controller, and service aids to use.

The **PDiagAtt** object class structure is defined as:

```
class PDiagAtt {
                  char DClass[16];
                  char DSClass[16];
                  char DType[16];
                  char attribute[16];
                  vchar value[255];
                  char rep[8];
                  vchar DApp[255];
                  };
```

DClass — Predefined device class. Devices are uniquely identified by a combination of *DClass, DSClass*, and *DType*.

DSClass — Predefined device sub-class. Devices are uniquely identified by a combination of of *DClass*, *DSClass*, and *DType*.

DType — Predefined device type.

attribute — 16-byte char field. The attribute value used by service aids to determine test mode for devices is **test_mode**. Uses *value* field.

value — 255-byte variable char field.

rep — 8-byte char field.

DApp — 255-byte variable char field.

Each field has specific meaning to each application that utilizes the **Predefined Diagnostic Attribute Device** object class (**PDiagAtt**).

EXAMPLES:

- To specify the tasks that are supported by a resource, create a **PDiagAtt** stanza for the resource, indicating the supported tasks in the *value* field.

```
PDiagAtt:
        DClass = "disk"
        DSClass = "scsi"
        DType = ""
        attribute = "SupTasks"
        value = "1,2,7,8,9,10,13,14,16,31,33"
        rep = "s"

PDiagAtt:
        DClass = "disk"
        DSClass = "scsi"
        DType = "355mb"
        attribute = "SupTasks"
        value = "1,2,7,8,9,10,13,14,31,33"
        rep = "s"
```

The search order performed by the Controller when determining the tasks a resource supports is as follows:

```
DClass, DSClass, DType DClass, DSClass DClass
```

In the above example, if the disk type is `355mb`, a match on the first call to search ODM is made; if not, a match will be made on the second call.

Note: The 355mb does not have task id 16, which is microcode download.

- To specify the application for the Diagnostic Controller to execute for a specific resource that supports a task, then a stanza similar to the following is needed. This example tells the Controller to invoke **ufd** to start a *format* task on the selected resource that matches the **diskette/siofd/fd** criteria.

```
PDiagAtt:
        DClass = "diskette"
        DSClass = "siofd"
        DType = "fd"
        attribute = "format"
        value = ""
        rep = "s"
        DApp = "ufd"
```

- The following stanza indicates the current release level of the Diagnostic Controller:

```
PDiagAtt:
        DType = "Dctrl"
        attribute = "version"
        value = "4.3.4"            #This is the diagnostic
                                 #version level seen on
                                 #the Operating
                                 #Instructions Menu.
        rep = "s"
```

- The **NoScreen** attribute is used by Display Test Pattern Service Aid to determine when a graphics adapter specific application should be used to display the screens for the service aid.

```
PDiagAtt:
        DType = "2b101a05"
        DSClass = "pci"
        attribute = "NoScreen"
        value = "/usr/lpp/diagnostics/da/dsage -P"
        rep = "NotOpen"
        DClass = "adapter"
        DApp = "u5081"
```

The service aid that uses this stanza is **/usr/lpp/diagnostics/bin/u5081**. The command that is built and executed is:

```
/usr/lpp/diagnostics/da/dsage <device name> -P
```

- The **platform_task+** attribute allows third parties to add tasks to the Task List based on the hardware platform. The DApp field specifies the platform for the tasks in the Task List. The value field of the stanza contains a comma delimited list of the task IDs to be added.

```
PDiagAtt:
        DType = ""
        DSClass = ""
        attribute = "platform_task+"
        value = "101,102,110"
        rep = ""
        DClass = ""
        DApp = "ia64"
```

  In the example above, the tasks whose IDs are 101, 102 and 110 will be included in the task list on the IA-64 platform. Multiple PDiagAtt stanzas with the **platform_task+** attribute are allowed.
  Note: The platform value for the DApp field is the string obtained by using the **bootinfo -p** command.

Predefined Diagnostic Task Object Class

The **Predefined Diagnostic Task** object class (**PDiagTask**) identifies the tasks supported by diagnostics and provides additional information needed to execute the task.

The **PDiagTask** object class structure is:

```
class PDiagTask {
                     long TaskId;
                     long SetId;
                     long MsgId;
                     long Multisession;
                     short Order;
                     long ResourceFlag;
                     long DiagEnvironment;
                     short Builtin;
                     vchar Action[255];
                     vchar Catalog[255];
                     vchar KernExt[255];
                     short DescriptionSetId;
                     short DescriptionMsgId;
                     char PkgBlock[5];
                     };
```

TaskId
: Unique number identifying the task.

SetId
: Catalog set number in either **Dctrl.cat** for the 'built-in' tasks, or in the catalog file specified for this task. The *Setid* and *Msgid* are used to display the task description on the Task Selection Menu.

MsgId
: Catalog message number in either **Dctrl.cat** for the 'built-in' tasks, or in the catalog file specified for this task. The *Setid* and *Msgid* are used to display the task description on the Task Selection Menu.

Multisession
: Flag indicating whether multiple instances of this task can be run simultaneously. While not currently used, this is a reserved field, and should be left blank.

  0 No

  1 Yes

Order
: Order to display the tasks in the Task Selection Menu. Value of 0 implies no order required, and the task will be placed at the end.

ResourceFlag
Flag indicating whether the Resource Selection menu should be presented after the task has been selected. If a resource is selected, then the task will be called with the resource name as a command-line argument to the task. If this value is 0, then the task is invoked directly.

Bit	Task
1	Present Resource Selection menu, and pass in selected Resource
2	Present Resource Section menu, and pass in selected Resources
4	Present Resource Selection menu, and pass in ″ALL″ if All is selected.
8	Rebuild Resource List after executing Task
16	Search PDiagAtt for DApp associated with Task
32	Task supports No Console mode
64	Task should be supported by all resources.

DiagEnvironment
Bit mask indicating various test mode environments this task is capable of running in.

Bit	Mode
1	Service Mode
2	Hardfile
4	Multiple Processor Platform Specific
8	ISA Bus Capability
16	RS6K and RS6KSMP Platform Specific
32	Removable Standalone Media
64	Hidden, do not display in Task Selection List
128	CHRP Platform
256	RSPC Platform
512	Do not display under WEB Diagnostics
1024	Task and the kernel extensions listed in KernExt are supported on the 64-bit kernel.

Builtin
Built-in task (part of the Controller).

Action
Basename of the program for this task. If no path given, then the default path of **/usr/lpp/diagnostics/bin** is used. If a complete path is given, then that path is used.

Catalog
Catalog file for this task. Catalog files containing default message text are assumed to be located in **/usr/lpp/diagnostics/catalog/default** directory. Translated files are assumed to be in **/usr/lib/nls/msg/$LANG** directories.

KernExt
',' separated list of kernel extensions to load for this task.

DescriptionSetId
Catalog set number of the help message text in either **Dctrl.cat** for the 'built-in' tasks, or in the catalog file specified for this task. The *DescriptionSetId* and *DescriptionMsgId* are used to display the help task description on the Task Selection Menu.

DescriptionMsgId
Catalog message number of the help message text in either **Dctrl.cat** for the 'built-in' tasks, or in the catalog file specified for this task. The *DescriptionSetId* and *DescriptionMsgId* are used to display the help task description on the Task Selection Menu.

PkgBlock
Block number that includes the task on the Removable Media Diagnostic package. This value should be an ″S″ if the task is on a Supplemental Media.

Customized Diagnostic Attribute Object Class

The **Customized Diagnostic Attribute** object class (**CDiagAtt**) contains customized entries for selected devices found in the current configuration, which is supported by diagnostics. The **CDiagAtt** object class

indicates specialized diagnostic attribute status of the device. It is used to maintain diagnostic information about devices found in the current configuration across sessions.

The **CDiagAtt** object class structure is defined as:

```
class CDiagAtt {
                     char name[16];
                     char attribute[16];
                     vchar value[255];
                     char type[8];
                     char rep[8];
                     };
```

name Resource name as specified in **CuDv.**

attribute 16-byte char field. The attribute value used by the Controller to identify persistent state data for the device. Uses *value* field.

value 255-byte variable char field.

type 8-byte char field specifying data type.

rep 8-byte char field.

Examples:

- The Diagnostic Controller creates a **CDiagAtt** entry for each device that is tested periodically by the Diagnostic daemon. The format of the stanza looks like:

```
CDiagAtt:
    name = "hdisk0"                Resource to test
    attribute = "p_test_time"     Attribute: periodic-test-time
    value = "0300"                Test time ( 3AM )
    type = "T"                    Data type of 'text'
    rep = "s"                     'String' representation

CDiagAtt:
    name = "ent0"                              Resource name
    attribute = "p_test_time"
    value = "9999"                             Not tested
 indication
        type = "T"
        rep = "s"
```

- The Diagnostic Controller creates a **CDiagAtt** entry for each device that has been deleted from the resource list. The format of the stanza looks like:

```
CDiagAtt:
    name = "mem0"                           Resource name
    attribute = "not_in_tst_list"    Device has been deleted from
    value = "1"                           the Resource List
    type = "T"
    rep = "n"
```

Test Mode Input Object Class

The input parameters to the Diagnostic Application are stored in the **TMInput** object class. The subroutine **getdainput** should be used to retrieve the test mode input data values from this object class.

The **TMInput** object class structure is defined as:

```
class TMInput {
                     short exenv;
                     short advanced;
                     short system;
                     short dmode;
                     char date[80];
                     short loopmode;
                     short lcount;
                     short lerrors;
                     short console;
```

```
char parent[16];
char parentloc[16];
char dname[16];
char dnameloc[16];
char child1[16];
short state1;
char childloc1[16];
char child2[16];
short state2;
char childloc2[16];
long pid;
short cpuid;
};
```

exenv The execution environment. Possible values include the following:

EXENV_IPL
Diagnostics is being run in **pre-test** mode. Tests should not take longer than one-minute.

EXENV_STD
Standalone and Online Service diagnostics. The Service IPL was used to load the system. This can be accomplished either by initial program loading from disk or removable media. This mode also applies if the normal IPL was used to load the system and then maintenance mode was entered by issuing the command **shutdown -m**.

EXENV_CONC
Online Concurrent diagnostics. The Normal IPL was used to load the system.

advanced Derived from the Function Selection menu. Possible values include the following:

ADVANCED_TRUE
Advanced Diagnostic Routines, which are run by a trained service representative. May prompt for wrap plugs, etc.

ADVANCED_FALSE
Diagnostic Routines, which are run by the customer.

system Derived from the Diagnostic or Resource Selection menu. Possible values include the following;

SYSTEM_TRUE
System Checkout (All Resources) was chosen. The DAs perform noninteractive tests.

SYSTEM_FALSE
Option Checkout was chosen. The DAs perform interactive tests.

dmode
: The diagnostic mode indicates the type of analysis that should be undertaken. Possible values include the following:

DMODE_ELA
: Error-log analysis. No diagnostic tests are executed.

DMODE_MS1
: This procedure is started because the user indicated that the named device was *not* removed, moved, or turned off. This procedure should determine why the option was not detected. Generally, this type of analysis involves asking the user to check cables, power supplies, fans, panel lights, and so on. The device is not deleted from the configuration.

DMODE_MS2
: This procedure is started because the user indicated that the named device has been removed from the system and should be removed from the system configuration. This procedure should perform any unique ″pseudo″ device manipulation, notification, and so on. For example, when a physical disk is removed from the system, the LVM should be notified. The DA is responsible for deleting the device from the configuration. The Device's Undefine Method is provided for this purpose.

DMODE_PD
: Problem determination, including error-log analysis and diagnostics tests.

DMODE_REMIND
: Diagnostic reminder, which defaults to running once a week, looks for deconfigured resources or other problems that have been previously reported, but have not been fixed.

DMODE_REPAIR
: Repair checkout, which includes only diagnostics tests. The error log is not used because the user is attempting to verify new hardware.

date
: The date from which the error log should be scanned. For the syntax used to describe the data, see the **date** command.

loopmode
: The maintenance mode and service mode diagnostic package supports loop testing. All or part of the system can be tested multiple times. Possible values include the following:

LOOPMODE_NOTLM
: Not loop mode. The default value for concurrent diagnostics.

LOOPMODE_ENTERLM
: Entering loop mode. The DA can interact with the user to set up a test or to isolate a problem. The next time the DA is executed, it will be in loop mode.

LOOPMODE_INLM
: In loop mode. No user interaction is allowed. The DA polls the keyboard. The tests should be stopped when the user presses Cancel.

LOOPMODE_EXITLM
: The system is restored to its pretest state. The DA guides the user in the restoration of the system to its pretest state. For example, wrap plugs are removed and cables are replugged. No tests are executed.

lcount
: Number of passes in loop mode that have been completed.

lerrors
: Number of errors logged while in loop mode.

console
: Diagnostic Controller queries the database to determine if the default console has been configured. Configuration states include:

CONSOLE_TRUE
: A console is available.

CONSOLE_FALSE
: No console is available, or no console output is desired. The LEDs are used to signal an error (if the platform supports LEDs).

parent
: Name of the parent of *dname*.

parentloc — Location of *parent*. Format of string is ″00-00-00-00″.
dname — Name of the device to be tested.
dnameloc — Location of *dname*. Format of string is ″00-00-00-00″.
child1 — Name of the child device that has already been tested. Relevant for Option Checkout only.
childloc1 — Location of *child1*. Format of string is ″00-00-00-00″.
state1 — State associated with child1. The resource states include:

STATE_NOTEST
The resource has *not* been tested.

STATE_GOOD
The resource passed its tests.

STATE_BAD
The resource failed its tests.

child2 — Name of another child device that has already been tested. Relevant for Option Checkout only.
childloc2 — Location of *child2*. The format of the string is ″00-00-00-00″.
state2 — State associated with child2. The resource states include:

STATE_NOTEST
The resource has *not* been tested.

STATE_GOOD
The resource passed its tests.

STATE_BAD
The resource failed its tests.

pid — Process ID of the DA when started from the Controller.
cpuid — Logical processor number plus one which the DA when started from the Controller should bind itself to. While not currently used, this is a reserved field, and should be left blank.

All values can be found in **/usr/include/diag/tmdefs.h**.

Menu Goal Object Class

The **Menu Goal** object class (**MenuGoal**) is used to store additional text information that the Diagnostic Application wants to pass back to the Diagnostic Controller. This text information is displayed to the user. This information is usually additional information that would be useful to the user concerning the state of the resource. One example would be that the Tape Drive requires cleaning.

All applications using the MenuGoal capability must use the **menugoal** diagnostic library subroutine.

The **MenuGoal** object class structure is defined as:

```
class MenuGoal {
      char dname[16];
                    longchar tbuffer1[1000];
                    longchar tbuffer2[1000];
                    };
```

dname — Resource name as specified in **CuDV**
tbuffer1 — Buffer used to store 1000 bytes of text
tbuffer2 — Buffer used to store 1000 bytes of text

FRU Bucket Object Class

The **Fru Bucket Object Class** (**FRUB**) is used to store failing replaceable unit information. This information is specified by the Diagnostic Application and passed back to the Diagnostic Controller after an error has been detected.

All applications using the FRU capability must use the **addfrub** diagnostic library subroutine.

The **FRUB** object class structure is defined as:

```
class FRUB {
                    char dname[16];
                    short ftype;
                    short sn;
                    short rcode;
                    short rmsg;
                    char timestamp[80];
                    };
```

dname Names the device under test.

ftype Indicates the type of FRU Bucket being added to the system. The following values are defined:

FRUB1 The FRUs include the resource that failed, its parent, and any cables needed to attach the resource to its parent.

FRUB2 This FRU Bucket is similar to FRU Bucket **FRUB1**, but does not include the parent resource.

FRUB_ENCLDA
This FRU Bucket is used for missing devices in the I/O enclosure(s).

sn Source number of the failure.

rcode Reason code associated with the failure.

Note: A Service Request Number is formatted as follows:

```
SSS - RRR
```

where SSS is the *sn* and RRR is the *rcode*.

Some devices may use a different nomenclature for their service request numbers. For this special case, the *sn* parameter indicates how the *rcode* value should be formatted. If *sn* = 0, then *rcode* is interpreted as decimal. If *sn* = -1, then *rcode* is interpreted as a 4-digit hexadecimal number.

If *sn* = -2, then the object class **DAVars** is searched for an attribute of *Error_code*. This allows the displaying of eight-digit hex error codes. The diagnostic application is responsible for setting up a **DAVars** object similar to the following:

DAVars:

```
dname: <device name under test>
vname: Error_code           "Error_code is an ascii string"
vtype: DIAG_STRING          "Literal value"
val: <8 digit hex character string>
```

See the **getdavar/putdavar** subroutine for more information.

rmsg Message number of the text describing the reason code. The set number of the text is predefined by the *PSet* field in the **Predefined Diagnostic Resources** object class.

timestamp Specifies the time the FRU bucket was added.

FRU Reporting Object Class

The **Fru Reporting Object Class** (**FRUs**) is used to store failing replaceable unit information. This information is specified by the Diagnostic Application and passed back to the Diagnostic Controller after an error has been detected.

All applications using the FRU capability must use the **addfrub** diagnostic library subroutine.

The **FRUs** object class structure is defined as:

```
class FRUs {
                char dname[16];
                char fname[16];
                char floc[16];
                short ftype;
                short fmsg;
                short conf;
                };
```

dname
Names the device under test.

fname
Names the FRU.

The parameters *floc* and *fmsg* must be specified, if *fname* is *not* represented in the **Customized Devices** object class. Otherwise, they should be set to 0.

floc
Location icode for *fname*.

ftype
Indicates the type of FRU Bucket being added to the system. The following values are defined:

FRUB1 The FRUs include the resource that failed, its parent, and any cables needed to attach the resource to its parent.

FRUB2 This FRU Bucket is similar to FRU Bucket **FRUB1**, but does not include the parent resource.

FRUB_ENCLDA
This FRU Bucket is used for missing devices in the I/O enclosure(s).

fmsg
Message number of the text describing *fname*. The set number is predefined by the *PSet* descriptor in the **Predefined Diagnostic Resources** object class.

conf
Indicates whether an FRU is valid. A value of 0 indicates an invalid FRU. No other FRUs are displayed once an invalid FRU is found in the FRU bucket.

However, if *fname* contains the string **REF-CODE**, then the *fmsg* and *conf* values are used to make the 8-digit ref code.

For AIX 4.3.2 and earlier versions, this field indicates the probability of failure associated with the named FRU.

Diagnostic Application Variables Object Class

The **Diagnostic Application Variables Object Class** (**DAVars**) is used to store run time information needed by the Diagnostic Application. This object class is used to store state variables to support Loop Testing.

All applications using the DAVars capability must use the **getdavar/putdavar** diagnostic library subroutine.

The **DAVars** object class structure is defined as:

```
class DAVars {
                char dname[16];
                char vname[30];
                short vtype;
                char vvalue[30];
                long ivalue;
                };
```

dname
Name of the device with which the variable is associated.

vname
Name of the variable.

vtype Type of the variable. The following values are defined:

DIAG_STRING
The variable should be treated as a character string.

DIAG_INT
The variable should be treated as an integer.

DIAG_SHORT
The variable should be treated as a short.

vvalue Stores character string variable.

ivalue Stores integer or short value of variable.

Predefined Diagnostic Devices Object Class

The **Predefined Diagnostic Devices** object class (**PDiagDev**) identifies the resources supported by AIX 4.1 diagnostics and provides additional information needed to test the resource. This object class is recognized by the operating system for backlevel compatibility purposes. For development purposes, use **PDiagRes** instead.

The **PDiagDev** object class structure is defined as:

```
class PDiagDev {
                    char DType[16];
                    char DSClass[16];
                    short Ports;
                    short PSet;
                    short PreTest;
                    char AttDType[16];
                    char AttSClass[16];
                    short Conc;
                    short SupTests;
                    short Menu;
                    short DNext;
                    vchar DaName[255];
                    char  Diskette[5];
                    vchar EnclDaName[255];
                    short Sysxflg;
                    char DClass[16];
                    };
```

DType Predefined device type.

DSClass Predefined device subclass.

DClass Predefined device class.

Ports Same definition as **PDiagRes->Ports**.

PSet Same definition as **PDiagRes->PSet**.

PreTest Same definition as **PDiagRes->PreTest**.

AttDType Device predefined type of the child device to define when the *Ports* field is set. The device named should include a set of device drivers that contain support for diagnostics.

AttSClass Device subclass of the child device to define when the *Ports* field is set.

Conc Indicates if the device is supported in multiuser mode. The values are as follows:

DIAG_YES
The device is supported in multiuser mode.

DIAG_NO
The device is *not* supported in multiuser mode.

SupTests
: Identifies the types of tests supported by the DA. More than one of the following types may be specified:

SUPTESTS_SHR (0x0001)
: Shared tests are supported.

SUPTESTS_SUB (0x0002)
: Sub-tests are supported.

SUPTESTS_FULL (0x0004)
: Full-tests are supported.

SUPTESTS_MS1 (0x0008)
: An optional procedure that determines why the device was *not* detected. This procedure is typically specified for devices that have external power supplies. This procedure is associated with the first selection at the Missing Resource menu.

SUPTESTS_MS2 (0x0010)
: An optional procedure that performs device-specific actions when a device is removed. For example, the DA might notify a subsystem (LVM) that a physical resource (disk) has been removed. Or the DA might provide warning about deleting a device. If this procedure is *not* specified, the Diagnostic Controller deletes the device. If it is specified, the DA should delete the device. Devices are deleted by calling the device's Undefine Method. This procedure is associated with the second selection at the Missing Resource menu.

SUPTESTS_HFT
: Set if the device is a graphics-related device.

SUPTESTS_DIAGEX
: Set if the device uses **DIAGEX**, the diagnostic kernel extension. Also used if the DA requires a second kernel extension loaded. The PDiagAtt database is used in this instance. A stanza similar to the following must be used:

PDiagAtt:

DClass
: The device Class.

DSClass
: The device SubClass.

DType
: The device Type.

attribute
: Must be **diag_kext**.

value
: Set to the kernel extension driver name. Must reside in **/usr/lib/drivers** directory.

Menu
: Same definition as **PDiagRes->Menu**.

DNext
: Same definition as **PDiagRes->DNext**.

DaName
: Same definition as **PDiagRes->DaName**.

Diskette
: A diskette identification that includes the DA associated with the device for the Standalone Diagnostic package. This value should be an ″S″ if the DA is on a Supplemental Diskette, or a ″3S″ if the DA is a graphics adapter that can be used as a console device.

EnclDaName
: Same definition as **PDiagRes->EnclDaName**.

SysxFlg Identifies the types of tests supported by the DA while running in the System Exerciser Environment. The System Exerciser Environment is not supported by version 4.2 of the diagnostic controller.

SYSX_NO
Set if the DA should not be run by the System Exerciser.

SYSX_ALONE
Set if the DA cannot be run with others with the same bit also set. This includes the diskette DAs that issue a reset to the adapter, which would cause problems if another diskette DA was running at the same time. Another example would be graphics-related devices such as the keyboard, mouse, tablet, dials, and LPFKeys.

SYSX_INTERACTION
Set if the DA can be run with media to be tested. This includes the diskette, tape and CD-ROM DAs. SYSX_INTERACTION was formerly SYSX_MEDIA.

SYSX_LONG
Set if the DA runs for more than a minute or so. This bit can be used to determine how many times to run the other DAs if no long DAs are running. The current loop count for DAs that do not take long to run is 25 loops.

Diagnostic Supervisor Menu Options Object Class

The **Diagnostic Supervisor Menu Options** object class (**DSMOptions**) contains stanzas describing AIX 4.1 Diagnostic Service Aids. This object class is recognized by the operating system for backlevel compatibility purposes. For development purposes, use **PDiagRes** instead.

The **DSMOptions** object class structure is defined as:

```
class DSMOptions {
                    char msgkey[4];
                    vchar catalogue[255];
                    short order;
                    short setid;
                    short msgid;
                    vchar action[255];
                    char  Diskette[5];
                    };
```

msgkey Key used by the Service Aid Utility Controller to identify this entry as a Service Aid. Must be set to ″USM″ for Service Aids.

catalogue Catalog name from which to extract the message for the Service Aid title and description.

order Order in which the messages should be appended to build the menu. The following values are defined:

0 Used by Third Party Service Aids. This causes the service aid to be appended to the end of the menu.

99 Only display this service aid if not running in an 8MB system.

setid Set number of the message.

msgid Message ID of the message.

action Command to start, if the user selects the specified option.

Diskette Indicates that the Service Aid is on a Supplemental Diskette, and what actions to take before processing the Service Aid. The following values are defined:

S Supplemental Diskette.

100X Indicates that all diskettes should be read in and processed before starting this service aid.

200X Indicates that this service aid is only supported in Service Mode from hardfile.

Diagnostic Header Files

Several files are shipped to the **/usr/include/diag** directory for use with compiling diagnostic code. All variables used by this guide should be found in one of the diagnostic header files.

Diagnostic User Interface

The following sections describe how Diagnostic Applications and Diagnostic Tasks should use the interfaces provided in the Diagnostic Library to display the different screen types. The Diagnostic Subsystem supports various display environments. The menu interfaces are designed to be display environment independent, with the library routine(s) building the correct menu structures depending on the display environment.

Screen Types

The Diagnostic Subsystem uses six different screen types, displayed by four different functions:

Screen Type	Diagnostic Applications	Diagnostic Tasks
INFORMATIVE	diag_resource_screen	diag_task_screen
SINGLE SELECTION	diag_resource_screen	diag_task_screen
MULTIPLE SELECTION	n/a	diag_task_screen
DIALOG SELECTION	n/a	diag_task_screen
TRANSITIONAL	diag_resource_screen diag_progress	diag_task_screen diag_progress
POPUP	diag_popup	diag_popup

Screen Size Assumptions

In order for Diagnostics to run in a window, a minimum screen dimension of 24 lines by 80 columns is required.

INSTRUCTION LINE

The INSTRUCTION LINE will be added automatically depending on the screen type. The following table illustrates the messages used for the INSTRUCTION LINE.

Screen Type	INSTRUCTION LINE
INFORMATIVE	`Use Enter to continue.`
SINGLE SELECTION	`Make selection, use Enter to continue.`
MULTIPLE SELECTION	`Make selection(s), use Commit to continue.`
DIALOG SELECTION	`Enter selection(s), use Commit to continue.`
TRANSITIONAL	`Please stand by.`
POPUP	n/a

Diagnostic Applications

Diagnostic Applications should use one of the following screen types:

- **INFORMATIVE**
- **SINGLE SELECTION**
- **TRANSITIONAL**

- **POPUP**

The following template shows a sample screen that is used when running diagnostics on a resource. The DA would use the **diag_resource_screen** library function to display this screen.

The Title line is split between lines 1 and 2. The ACTION, TEST MODE, and the menu number go on the first line. ACTION is defined as one of the following:

- TESTING
- ANALYZING ERROR LOG
- ANALYZING POST RESULTS
- ANALYZING FIRMWARE STATUS
- ANALYZING SUBSYSTEM STATUS
- ANALYZING CHECKSTOP STATUS

If the ACTION is `TESTING`, the TEST MODE will be displayed on the first line. TEST MODE is defined as:

- **ADVANCED MODE**
- **LOOP MODE** (Advanced Mode is always assumed if Looping.)

The TEST MODE field will be blank if running non-advanced mode diagnostics.

The *Menu Number* represented by `xxxxxx`, goes on the first line.
The *Resource Name* and *Location Code* go on the second line.

```
           1         2         3         4         5         6         7
   01234567890123456789012345678901234567890123456789012345678901234567890123456789
   --------------------------------------------------------------------------------
 1 ACTION          {TEST MODE}                                               xxxxxx
 2 Resource Name   Location Code
 3
 4 +
 5 |
 6 |
 7 |
 8 |
 9 |
10 |
11 |
12 BODY OF MENU
13 |
14 |
15 |
16 |
17 |
18 |
19 |
20 |
21 +
22                    ___________________________________________
23 Function Key Area  | Progress Indicator Area                   |
24 Function Key Area  |                                           |
   --------------------------------------------------------------------------------
```

The BODY of the menu can assume multiple personalities depending on the screen type. It includes all text of the menu, including the INSTRUCTION line. The BODY does not include the TITLE.

INFORMATIVE SCREEN TYPE

For an **INFORMATIVE** screen, the body consists of information describing the test and what it does. In the following example, lines 4 through 12 consist of the information about the test. Line 14 is the INSTRUCTION LINE, and is added automatically by the **diag_resource_screen** function.

```
          1         2         3         4         5         6         7
   01234567890123456789012345678901234567890123456789012345678901234567890123456789
   --------------------------------------------------------------------------------
 1 TESTING         ADVANCED MODE                                            935045
 2 fd0             00-00-0D-00
 3
 4 Diskette Change/Write Protect Test
 5
 6 REMOVE.........the diskette, if any, from the diskette drive (fd0).
 7 INSERT.........the High Capacity (4M byte) Diagnostic Test
 8                Diskette or an equivalent, formatted,
 9                scratch diskette into the diskette drive (fd0).
10
11 NOTE:  The diskette must be write protected (the write protect
12        tab should not cover the hole).
13
14 Use Enter to continue.
15
16
17
18
19
20
21
22
23
24 F3=Cancel           F10=Exit            Enter
   --------------------------------------------------------------------------------
```

SINGLE SELECTION SCREEN TYPE

For a **SINGLE SELECTION** screen, the body consists of results from a previous test that had run, and asking the user if the results are accurate. The User selects a response, normally YES or NO, from a given list. In the following example, lines 4 through 9 consist of the information about the test. Lines 13 and 14 consist of the SELECTION lines. Line 11 is the INSTRUCTION LINE, and is added automatically by the **diag_resource_screen** function.

```
          1         2         3         4         5         6         7
   01234567890123456789012345678901234567890123456789012345678901234567890123456789
   --------------------------------------------------------------------------------
 1 TESTING          ADVANCED MODE                                           935025
 2 fd0              00-00-0D-00
 3
 4 Diskette Select and Deselect Test
 5
 6 OBSERVE........the in-use light on the diskette drive (fd0).
 7
 8 Was the in-use light on for approximately 5 seconds and
 9 then did it turn off?
10
11 Make selection, use Enter to continue.
12
13   YES
14   NO
15
16
17
18
19
20
21
22
23
24 F3=Cancel           F10=Exit            Enter
   --------------------------------------------------------------------------------
```

TRANSITIONAL SCREEN TYPE

For a **TRANSITIONAL** screen, the body usually consists of a single INSTRUCTION line of `Please stand by`. This indicates that the test is currently processing some data. It is also used to indicate that looping is in progress, and shows the number of passes made plus the total number of errors encountered. User may press `Cancel` to stop the test. The following example shows a looping menu. Line 10 is the INSTRUCTION LINE, and is added automatically by the **diag_resource_screen** function. See also Diagnostic Progress Indicators.

```
             1         2         3         4         5         6         7
   01234567890123456789012345678901234567890123456789012345678901234567890123456789
   --------------------------------------------------------------------------------
 1 TESTING          LOOP MODE                                                935025
 2 fd0              00-00-0D-00
 3
 4
 5
 6   1  passes completed.
 7   5  errors logged.
 8
 9
10 Please stand by.
11
12
13
14
15
16
17
18
19
20
21
22
23
24 F3=Cancel           F10=Exit
   --------------------------------------------------------------------------------
```

POPUP SCREEN TYPE

For a **POPUP** screen, the application code should use the **diag_popup** library function call.

Diagnostic Tasks

Diagnostic Tasks are free to use any of the six supported screen types:

- **INFORMATIVE**
- **SINGLE SELECTION**
- **MULTIPLE SELECTION**
- **DIALOG SELECTION**
- **TRANSITIONAL**
- **POPUP**

The following template shows a sample screen that is used when running a task. The Task would use the **diag_task_screen** library function to display this screen.

The Title line is split between lines 1 and 2. Most all Task titles should fit on the first line, but the second line may be used for clarity or for translation reasons. The TITLE text should be all capitalized.

```
             1         2         3         4         5         6         7
   01234567890123456789012345678901234567890123456789012345678901234567890123456789
   --------------------------------------------------------------------------------
 1 TASK TITLE LINE 1                                                         8xxxxx
 2 TASK TITLE LINE 2
 3
```

```
 4 +
 5 |
 6 |
 7 |
 8 |
 9 |
10 |
11 |
12 BODY OF MENU
13 |
14 |
15 |
16 |
17 |
18 |
19 |
20 |
21 +
22
23 Function Key Area
24 Function Key Area
   --------------------------------------------------------------------------------
```

The BODY of the menu can assume multiple personalities depending on the screen type. It includes all text of the menu, including the INSTRUCTION line. The BODY does not include the TITLE.

INFORMATIVE SCREEN TYPE

For an INFORMATIVE screen, the body consists of information describing the task and what it does. In the following example, lines 3 through 15 consist of the information about the task. Line 17 is the INSTRUCTION LINE, and Line 24 consists of the function keys available for this screen type. Both lines are added automatically by the **diag_task_screen** function.

NOTE: If the TITLE line consists of only one line, the text of the BODY will be adjusted up one line.

```
             1         2         3         4         5         6         7
   01234567890123456789012345678901234567890123456789012345678901234567890123456789
   --------------------------------------------------------------------------------
 1 PERIODIC DIAGNOSTICS SERVICE AID                                          802150
 2
 3 This service aid is used to periodically test hardware resources and
 4 monitor hardware errors in the error log.
 5
 6 A hardware resource can be chosen to be tested once a day, at a user
 7 specified time of day. If the resource cannot be tested because it is
 8 busy, error log analysis will be performed.
 9 Hardware errors logged against a resource can also be monitored by enabling
10 Automatic Error Log Analysis. This will allow error log analysis to be
11 performed every time a hardware error is put into the error log.
12
13 If a problem is detected, a message will be posted to the system console
14 and a mail message sent to user(s) belonging to system group with information
15 about the failure such as Service Request Number.
16
17 Use Enter to continue.
18
19
20
21
22
23
24 [F1=Help]        F3=Cancel              F10=Exit                Enter
   --------------------------------------------------------------------------------
```

SINGLE SELECTION SCREEN TYPE

For a SINGLE SELECTION screen, the body consists of individual selectable items and possibly a short description. In the following example, lines 5 through 21 consist of the selectable items. This example illustrates six (6) selectable menu items. The indentions for the selectable item descriptions must be added when the message is built. Line 3 is the INSTRUCTION LINE, and is added automatically by the **diag_task_screen** function.

Any information about the selections may be added to the screen, and would appear after the TITLE line[1] and before the INSTRUCTION line[3].

```
          1         2         3         4         5         6         7
   01234567890123456789012345678901234567890123456789012345678901234567890123456789
   --------------------------------------------------------------------------------
 1 PERIODIC DIAGNOSTICS SERVICE AID                                          802151
 2
 3 Make selection, use Enter to continue.
 4
 5   Add a resource to the periodic test list
 6     This selection allows a resource to be periodically tested.
 7   Delete a resource from the periodic test list
 8     This selection removes a resource from the list of periodically
 9     tested resources.
10   Modify the time to test a resource
11     This selection allows the time of day to test a resource to be
12     changed.
13   Display the periodic test list
14     This selection displays all resources being tested periodically
15     by diagnostics.
16   Modify the error notification mailing list
17     This selection allows the mailing list for error notification
18     to be modified.
19   Disable Automatic Error Log Analysis
20     Automatic Error Log Analysis is currently enabled.
21     This selection stops the Automatic Error Log Analysis.
22
23
24 F1=Help               F10=Exit                F3=Previous Menu
   --------------------------------------------------------------------------------
23 F1=Help               F4=List           F10=Exit          Enter
24 F3=Previous Menu
   --------------------------------------------------------------------------------
```

MULTIPLE SELECTION SCREEN TYPE

For a MULTIPLE SELECTION screen, the body consists of individual selectable items and possibly a short description. In the following example, lines 10 through 12 consist of the selectable items. Line 8 is the INSTRUCTION LINE, and is added automatically by the **diag_task_screen** function.

Any information about the selections may be added to the screen, and would appear after the TITLE line[1] and before the INSTRUCTION line[8].

HELP text may be displayed any time the cursor is on line 10, 11, or 12 in the following example. Each selectable line may have associated HELP text.

```
          1         2         3         4         5         6         7
   01234567890123456789012345678901234567890123456789012345678901234567890123456789
   --------------------------------------------------------------------------------
 1 DELETE RESOURCES FROM THE PERIODIC DIAGNOSTICS TEST LIST                  802155
 2
 3 The following resources are currently being tested periodically.
 4 Test time is shown inside the brackets in 24 hour format.
 5 Once deleted, a resource cannot be tested until it is added back to the
 6 test list.
 7
 8 Make selection(s), use Commit to continue.
 9
```

```
10   ioplanar0        [04:00]          I/O Planar
11   hdisk0           [03:00]          1.0 GB SCSI Disk Drive
12   hdisk1           [03:00]          2.0 GB SCSI Disk Drive
13
14
15
16
17
18
19
20
21
22
23 F1=Help              F2=Refresh             F3=Cancel             F4=List
24 F5=Reset             F7=Commit              F10=Exit
   --------------------------------------------------------------------------------
```

DIALOG SELECTION SCREEN TYPE

For a DIALOG SELECTION screen, the body consists of individual items with a bracketed area to the right. This bracketed area allows data selections to be set for each individual item. In the following example, lines 10 and 11 consist of the items. Line 7 is the INSTRUCTION LINE, and is added automatically by the **diag_task_screen** function.

HELP text may be displayed any time the cursor is on line 10 or 11 in the following example. Each dialog line may have associated HELP text.

```
             1         2         3         4         5         6         7
   01234567890123456789012345678901234567890123456789012345678901234567890123456789
   --------------------------------------------------------------------------------
 1 PERIODIC DIAGNOSTICS SERVICE AID                                          802157
 2
 3 ent0             00-00-0E         Integrated Ethernet Adapter
 4
 5 Set the time when the resource should be tested.
 6
 7 Enter selection(s), use Commit to continue.
 8
 9
10   * HOUR (00-23) ................................         [00]              +#
11   * MINUTES (00-59) .............................         [00]              +#
12
13
14
15
16
17
18
19
20
21
22
23 F1=Help              F2=Refresh             F3=Cancel             F4=List
24 F5=Reset             F7=Commit              F10=Exit
   --------------------------------------------------------------------------------
```

TRANSITIONAL SCREEN TYPE

For a TRANSITIONAL screen, the body consists of a single INSTRUCTION line of `Please stand by`. This indicates that the task is currently processing some data. Users may press `Cancel` to stop the task. The following example shows a task in progress menu. Line 6 consists of the INSTRUCTION LINE, and is automatically added by the **diag_task_screen** function. See also Diagnostic Progress Indicators.

```
             1         2         3         4         5         6         7
   01234567890123456789012345678901234567890123456789012345678901234567890123456789
   --------------------------------------------------------------------------------
 1 HARDWARE ERROR REPORT      802905
 2
```

```
 3
 4 Reading current error log.
 5
 6 Please stand by.
 7
 8
 9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24 F3=Cancel           F10=Exit
   --------------------------------------------------------------------------------
```

POPUP SCREEN TYPE

For a POPUP screen, the body consists normally of help text. It is used to help the user understand the current screen, or menu selection. In the following example, the popup appears in a windowed box near the bottom of the screen. No INSTRUCTION line is used. This screen is added by the diag_popup function.

If the F1=Help key is selected, but there is no associated Help text associated with the current selection, then this key is returned to the calling application.

```
             1         2         3         4         5         6         7
   01234567890123456789012345678901234567890123456789012345678901234567890123456789
   --------------------------------------------------------------------------------
 1 FUNCTION SELECTION                                                        801002
 2
 3
 4 Move cursor to selection, then press Enter.
 5
 6  Diagnostic Routines
 7     This selection will test the machine hardware. Wrap plugs and
 8     other advanced functions will not be used.
 9  Advanced Dl _______________________________________________________
10     This sel|                                                       |
11     other ad|                                                       |
12   Task Selec|  Select this choice when you want to run              |c.)
13     This sel|  Diagnostics on a resource (device).                  |.
14     Once a t|                                                       |g
15     all reso|                                                       |
16   Resource S|                                                       |
17     This sel|                                                       |pported
18     by these|                                                       |ll
19     be prese|                                                       |).
20             |                                                       |
21             |                                                       |
22             |                                                       |
23             |  F3=Cancel        F10=Exit         Enter              |
24 F1=Help     |_______________________________________________________|
   --------------------------------------------------------------------------------
```

Diagnostic Progress Indicators

Diagnostic Progress Indicators are used to inform the user what is going on. The Progress Indicators appear as a popup box at the bottom of the screen during a **Diagnostic Application TRANSITIONAL screen** or a **Diagnostic Task TRANSITIONAL screen** display.

The Progress Indicators may be turned off by using the Run Time Options Task. This selection sets the diagnostic environment variable **DIAG_NO_PROGRESS** appropriately.

```
          1         2         3         4         5         6         7
   01234567890123456789012345678901234567890123456789012345678901234567890123456789
   --------------------------------------------------------------------------------
 1 DISPLAY/CHANGE DIAGNOSTIC RUN TIME OPTIONS                              801009
 2
 3 Select values for the options below.
 4 When finished, use 'Commit' to continue.
 5 Display Diagnostic Mode Selection Menus                 [On]        +
 6 Include Advanced Diagnostics                            [Off]       +
 7 Include Error Log Analysis                              [Off]       +
 8 Display Progress Indicators                             [On]        +
 9
10
11
12
13
14
15
16
17
18
19
20
21
22
23 F1=Help            F2=Refresh         F3=Cancel          F4=List
24 F5=Reset           F7=Commit          F10=Exit
   --------------------------------------------------------------------------------
```

The following example shows a Diagnostic Application screen that is displaying a Progress indicator with the type of test unit being run.

```
          1         2         3         4         5         6         7
   01234567890123456789012345678901234567890123456789012345678901234567890123456789
   --------------------------------------------------------------------------------
 1 TESTING         LOOP MODE                                               935025
 2 fd0             00-00-0D-00
 3
 4
 5
 6   1  passes completed.
 7   5  errors logged.
 8
 9
10 Please stand by.
11
12
13
14
15
16
17
18
19
22              --------------------------------------------------
23              | Register Test                                    |
24 F3=Cancel    --------------------------------------------------
   --------------------------------------------------------------------------------
```

These Progress Indicator messages must be kept short, one line, and under 30 characters. Note that the function key `F10=Exit` is overwritten by the Progress Indicator.

The diag_progress library function call is used for this Progress Indicator.

Diagnostic Menu Examples

Diagnostic Operating Instructions Menu

```
DIAGNOSTIC OPERATING INSTRUCTIONS  VERSION X.X.X                          801001


LICENSED MATERIAL and LICENSED INTERNAL CODE - PROPERTY OF IBM
(C) COPYRIGHTS BY IBM AND BY OTHERS YYYY, YYYY.
ALL RIGHTS RESERVED.

These programs contain diagnostics, service aids, and tasks for
the system.  These procedures should be used whenever problems
with the system occur which have not been corrected by any
software application procedures available.

In general, the procedures will run automatically.  However,
sometimes you will be required to select options, inform the
system when to continue, and do simple tasks.

Several keys are used to control the procedures:
- The Enter key continues the procedure or performs an action.
- The Backspace key allows keying errors to be corrected.
- The cursor keys are used to select an option.

Press the F3 key to exit or press Enter to continue.
```

Note: The version number may vary depending on the version of diagnostics installed or the version the standalone diagnostics used.

Function Selection Menu

```
FUNCTION SELECTION                                                        801002


Move cursor to selection, then press Enter.

  Diagnostic Routines
    This selection will test the machine hardware. Wrap plugs and
    other advanced functions will not be used.
  Advanced Diagnostics Routines
    This selection will test the machine hardware. Wrap plugs and
    other advanced functions will be used.
  Task Selection(Diagnostics, Advanced Diagnostics, Service Aids, etc.)
    This selection will list the tasks supported by these procedures.
    Once a task is selected, a resource menu may be presented showing
    all resources supported by the task.
  Resource Selection
    This selection will list the resources in the system that are supported
    by these procedures. Once a resource is selected, a task menu will
    be presented showing all tasks that can be run on the resource(s).




F1=Help            F10=Exit           F3=Previous Menu
```

Define Terminal Menu

```
DEFINE TERMINAL

The terminal is not properly initialized.
The following are some of the terminal types that are supported.
```

```
 ibm3101          tvi912          vt330
 ibm3151          tvi925          vt340
 ibm3161          tvi920          wyse30
 ibm3162          tvi950          wyse50
 ibm3163          vs100           wyse60
 ibm3164          vt100           wyse100
 ibmpc            vt320           wyse350
 lft              sun

NOTE:  If you are using a Graphics Display, such as a
       5081 or 6091 display, enter 'lft' as the
       terminal type.

       If the next screen is unreadable, press <CTRL> C.

Please enter a terminal type, or press Enter to return.
```

Missing Resource Selection Menu

```
MISSING RESOURCE                                                          801020


The list below shows all the missing resources. Make a selection,
then press Enter to process missing options resolutions.
To list all siblings of a resource, use 'List'.



  fda0             00-00-0D         Standard I/O Diskette Adapter
    fd0













F1=Help             F4=List             F10=Exit            Enter
F3=Previous Menu
```

Missing Resource Menu

```
MISSING RESOURCE                                                          801020

The following resource was detected previously, but is not detected now:

- fda0             00-00-0D        Standard I/O Diskette Adapter

Has the resource been removed from the system, moved to another location
or address, or turned off?

  The resource has NOT been removed from the system, moved to
    another location or address, or turned off.
    This selection will determine why the resource was not detected.
  The resource has been removed from the system and should be
    removed from the system configuration.
  The resource has been moved to another location and should be
    removed from the system configuration.
  The resource has been turned off and should be removed from
    the system configuration.
  The resource has been turned off but should remain in the
    system configuration.
```

```
F3=Cancel           F10=Exit
```

New Resource Menu

```
NEW RESOURCE                                                             801030

The following new resource(s) were detected.
Some resources may require software installation or supplemental media
processing to appear on the list.

Select an option from the bottom of the list, then press Enter.

  - rmt0            00-04-00-4,0    4.0 GB 4mm Tape Drive

  1. Continue. The list contains all resources that should appear.
  2. A resource that should appear on the list is missing.









F3=Cancel           F10=Exit
```

Diagnostic Mode Selection Menu

```
DIAGNOSTIC MODE SELECTION                                                801003

Move cursor to selection, then press Enter.

  System Verification
    This selection will test the system, but will not analyze the error
    log.  Use this option to verify that the machine is functioning
    correctly after completing a repair or an upgrade.
  Problem Determination
    This selection tests the system and analyzes the error log
    if one is available.  Use this option when a problem is
    suspected on the machine.









F1=Help             F10=Exit            F3=Previous Menu
```

Resource Selection Menu

```
RESOURCE SELECTION LIST                                                  801006

From the list below, select any number of resources by moving
the cursor to the resource and pressing 'Enter'.
To cancel the selection, press 'Enter' again.
To list the supported tasks for the resource highlighted, press 'List'.
```

```
Once all selections have been made, press 'Commit'.
To exit without selecting a resource, press the 'Exit' key.


[TOP]
  All Resources
      This selection will select all the resources currently displayed.
  sysplanar0      00-00           CPU Planar
  proc0           00-00           Processor
 *slc0            00-00           Serial Optical Link Chip
  otp0            00-AB           Serial Optical Channel Converter
 +op0             00-AB-1B        Serial Optical Link Port
  op1             00-AB-2B        Serial Optical Link Port
[MORE...30]

F1=Help            F4=List            F7=Commit          F10=Exit
F3=Previous Menu
```

- The + by op0 indicates that it has been selected.
- The * by slc0 indicates that it has been selected and run.
- Each resource is listed with the parent followed by the children.
- Each resource provides the following information:
  - Device logical name
  - Device logical location code
  - Device descriptive text

Resource Selection Menu - Display Common Tasks

```
RESOURCE SELECTION LIST                                                    801006

From the list below, select any number of resources by moving
the cursor to the resource and pressing 'Enter'.
To cancel the selection, press 'Enter' again.
To list the supported tasks for the resource highlighted, press 'List'.

Once all selections have been made, press 'Commit'.
To exit with--------------------------------------------------------
            |                                                      |
            |                                                      |
[MORE...12] | [TOP]                                                |
  sio0      | The following tasks are supported by the resource:   |
  siokta0   |                                                      |
+ kbd0      | (A '*' in front of a task indicates                  |
  sioma0    | that it has been selected:                           |
+ mouse0    |    Run Diagnostics                                   |
  ppa0      |    Display or Change Diagnostic Run Time Options     |er
  lp0       |    Display Configuration and Resource List           |
  sa0       |    Display Hardware Vital Product Data               |
[MORE...16] | [MORE...5]                                           |
            |                                                      |
F1=Help     | F3=Cancel       F10=Exit        Enter                |
F3=Previous --------------------------------------------------------
```

Use the F4=List key to display the common tasks supported by the selected resources.

Test Method Menu

```
TEST METHOD SELECTION                                                    801004

 Move cursor to selection, then press Enter.

   Run Test Once
   Run Test Multiple Times
```

```
    This selection should be used when a problem occurs intermittently.
    This selection will continue testing until 'Cancel' is pressed.
NOTE:  After 'Cancel' is pressed, it may take some time before
       the testing stops.  The tests goes through a final phase
       to return the resources to their original state.









F3=Cancel          F10=Exit
```

No Trouble Found Menu

```
TESTING COMPLETE on Wed Jan  7 14:01:22 CST 1998                         801010

No trouble was found.

The resources tested were:

- proc0            00-00           Processor

Use Enter to continue.









F3=Cancel           F10=Exit            Enter
```

Problem Report Menu

```
A PROBLEM WAS DETECTED ON Wed Jan  7 13:45:57 CST 1998                   801014

The Service Request Number(s)/Probable Cause or Cause(s):

  816-185:  I/O Planar - key lock failed.
   65%  OP Panel                           Operator panel
   30%  Keylock                            Operator panel key lock
    5%  ioplanar0         00-00            I/O Planar


Use Enter to continue.
```

```
F3=Cancel          F10=Exit            Enter
```

Additional Resources Menu

```
ADDITIONAL RESOURCES ARE REQUIRED FOR TESTING                          801011

No trouble was found.  However, the resource was not tested because
the device driver indicated that the resource was in use.

The resource needed is
- hdisk0          00-04-00-1,0    670 MB SCSI Disk Drive

To test this resource, you can:
  Free this resource and continue testing.
  Shut down the system and run in maintenance mode.
  Run Diagnostics from the Diagnostic Standalone package.

Move cursor to selection, then press Enter.

  Testing should stop.
  The resource is now free and testing can continue.





F3=Cancel          F10=Exit
```

Task Selection List Menu

```
TASKS SELECTION LIST                                                   801004


From the list below, select a task by moving the cursor to
the task and pressing 'Enter'.
To list the resources for the task highlighted, press 'List'.

[TOP]
  Run Diagnostics
  Display or Change Diagnostic Run Time Options
  Display Service Hints
  Display Previous Diagnostic Results
  Display Hardware Error Report
  Display Software Product Data
  Display Configuration and Resource List
  Display Hardware Vital Product Data
  Display Resource Attributes
  Change Hardware Vital Product Data
  Format Media
  Certify Media
[MORE...21]

F1=Help            F4=List             F10=Exit            Enter
F3=Previous Menu
```

Task Selection List Menu - Display Supported Resources

```
TASKS SELECTION LIST                                                   801004


From the list below, select a task by moving the cursor to
```

```
the task and pressing 'Enter'.
To list the resources for the task highlighted, press 'List'.

[TOP]
  Run Diagno--------------------------------------------------------
  Display or|                                                        |
  Display Se|                                                        |
  Display Pr|  [TOP]                                                 |
  Display Ha|  The following resources support the current task:     |
  Display So|  (A '*' in front of a resource indicates that it       |
  Display Co|  has been selected)                                    |
  Display Ha|     sysplanar0                                         |
  Display Re|     proc0                                              |
  Change Har|     slc0                                               |
  Format Med|     otp0                                               |
  Certify Me|     op0                                                |
[MORE...21] |  [MORE...31]                                           |
            |                                                        |
F1=Help     |  F3=Cancel        F10=Exit         Enter               |
F3=Previous --------------------------------------------------------
```

Use the F4=List key to display all the resources supported by the selected Task.

Run Time Options Menu

```
DISPLAY/CHANGE DIAGNOSTIC RUN TIME OPTIONS                               801009

Select values for the options below.
When finished, use 'Commit' to continue.
  Display Diagnostic Mode Selection Menus                         [On]        +
  Include Advanced Diagnostics                                    [Off]       +
  Include Error Log Analysis                                      [Off]       +
  Number of days used to search error log                         [7]         +
  Save changes to the database?                                   [NO]        +








F1=Help             F2=Refresh          F3=Cancel           F4=List
F5=Reset            F7=Commit           F10=Exit
```

Chapter 4. Diagnostic Features

This chapter contains information on the various features that the Diagnostic Subsystem environment provides.

- Missing Options Resolution
- Error Log Analysis
- Periodic Diagnostic Testing
- Automatic Error Log Analysis(DIAGELA)
- Loop Testing

Missing Options Resolution

This section describes the Missing Options Resolution Procedure performed by Diagnostics when a change in the system configuration has been detected. This procedure can be run to clean up the system configuration database, or to determine why previously detected resources are no longer found by the operating system.

Each time the system boots from an installed hardfile, the device configuration database (**CuDv**) that is stored on the hardfile from the previous IPL is compared against the resources detected on the current IPL. Detectable resources that were found on the previous IPL but not the current IPL are marked as **MISSING**. Devices that were found on the current IPL, but not present in the previous IPL are marked as **NEW**.

The customized device entry **CuDv** *chgstatus* field is set to the changed status for each resource. These changed status values can be found in **/usr/include/sys/cfgdb.h** file.

When booting a system in normal mode, a message is written to the console if any devices have been detected as **MISSING**. This message states:

```
A device that was previously detected could not be found.

Run diag -a to update the system configuration.
```

The **diag -a** command can then be run to process the missing options resolution procedure.

When booting a system in online service mode, the missing options resolution procedure is run automatically if any missing devices were detected.

The following sections describe how the Diagnostic Controller presents information to the Diagnostic Applications that get invoked during Missing Options.

Online Concurrent Diagnostics

Missing Options Resolution procedure can be run in online concurrent mode by using the following command:

```
% diag -a // Runs in Customer Mode
```
OR
```
% diag -a -A // Runs in Advanced Mode
```

The first screen seen by the user is the **MISSING RESOURCE Menu**, 801020.

The following **TMInput** is an example of the input given to the Diagnostic Application when running the **diag -a** command.

© Copyright IBM Corp. 1997, 2001

```
TMInput:
        exenv = 4               // Concurrent Environment
        advanced = 0            // Customer Mode
        system = 0              // Option Checkout
        dmode = 4               // System Verification
        date = "-s START -e NOW"// START = NOW - 24 hours.
        loopmode = 1            // Not in Loop Mode
        lcount = 0
        lerrors = 0
        console = 1             // Console Available
        parent = "parent0"      // Parent of resource to test
        parentloc = "AB-CD"     // Parent's Location Code
        dname = "resource0"     // Name of resource to test
        dnameloc = "AB-CD"      // Resource's Location Code
        child1 = "child0"       // Missing Child of Resource
        state1 = 3              // State of Child is MISSING
        childloc1 = "AB-CD"     // Child's Location Code
        child2 = ""
        state2 = 0
        childloc2 = ""
```

The following **TMInput** is an example of the input given to the Diagnostic Application when running the **diag -a -A** command.

```
TMInput:
        exenv = 4               // Concurrent Environment
        advanced = 1            // Advanced Mode
        system = 0              // Option Checkout
        dmode = 4               // System Verification
        date = "-s START -e NOW"// START = NOW - 24 hours.
        loopmode = 1            // Not in Loop Mode
        lcount = 0
        lerrors = 0
        console = 1             // Console Available
        parent = "parent0"      // Parent of resource to test
        parentloc = "AB-CD"     // Parent's Location Code
        dname = "resource0"     // Name of resource to test
        dnameloc = "AB-CD"      // Resource's Location Code
        child1 = "child0"       // Missing Child of Resource
        state1 = 3              // State of Child is MISSING
        childloc1 = "AB-CD"     // Child's Location Code
        child2 = ""
        state2 = 0
        childloc2 = ""
```

Online Service Diagnostics

Missing Options Resolution procedure is run automatically in online service mode when `Diagnostics or Advanced Diagnostics` selection is made from the FUNCTION SELECTION Menu.

When booting a system in online service mode, the OPERATING INSTRUCTIONS Menu and the FUNCTION SELECTION Menu are displayed in phase 1 by the service mode boot script. Once a selection is made, the selection is stored in **/etc/lpp/diagnostics/data/fastdiag** file, and phase 2 of the boot process commences.

The Diagnostic Application that gets called due to a missing child resource, after selecting `Diagnostic Routines` from the FUNCTION SELECTION menu, gets a **TMInput** shown below:

```
TMInput:
        exenv = 2               // Standalone Environment
        advanced = 0            // Customer Mode
        system = 0              // Option Checkout
        dmode = 4               // System Verification
        date = "-s START -e NOW"// START = NOW - 24 hours.
        loopmode = 1            // Not in Loop Mode
        lcount = 0
```

```
        lerrors = 0
        console = 1            // Console Available
        parent = "parent0"     // Parent of resource to test
        parentloc = "AB-CD"    // Parent's Location Code
        dname = "resource0"    // Name of resource to test
        dnameloc = "AB-CD"     // Resource's Location Code
        child1 = "child0"      // Missing Child of Resource
        state1 = 3             // State of Child is MISSING
        childloc1 = "AB-CD"    // Child's Location Code
        child2 = ""
        state2 = 0
        childloc2 = ""
```

The Diagnostic Application that gets called due to a missing child resource, after selecting `Advanced Diagnostic Routines` from the FUNCTION SELECTION menu, gets a **TMInput** shown below:

```
TMInput:
        exenv = 2              // Standalone Environment
        advanced = 1           // Advanced Mode
        system = 0             // Option Checkout
        dmode = 4              // System Verification
        date = "-s START -e NOW"// START = NOW - 24 hours.
        loopmode = 1           // Not in Loop Mode
        lcount = 0
        lerrors = 0
        console = 1            // Console Available
        parent = "parent0"     // Parent of resource to test
        parentloc = "AB-CD"    // Parent's Location Code
        dname = "resource0"    // Name of resource to test
        dnameloc = "AB-CD"     // Resource's Location Code
        child1 = "child0"      // Missing Child of Resource
        state1 = 3             // State of Child is MISSING
        childloc1 = "AB-CD"    // Child's Location Code
        child2 = ""
        state2 = 0
        childloc2 = ""
```

Standalone Diagnostics (POWER-based only)

Missing Options Resolution procedure is not run during Standalone Diagnostics. The reason for this is that there is no previous configuration database for the Diagnostic Controller to compare against with the new devices detected at boot time.

Therefore, only the NEW RESOURCES menu is seen during Standalone Diagnostics. This menu presents a list of all the resources found in the system at the time the Standalone Diagnostics were booted.

The user is given a list of choices to make during this time. If the system contains ISA adapters, then these adapters will not appear in the list. ISA adapters are not detectable, therefore an option is presented to the user to help in the configuration of these adapters.

Missing Options Procedure Steps

The following describes the steps performed by the Diagnostic Controller when running the Missing Options Procedure.

1. The Diagnostic Controller keeps a sorted list of all resources found in the system as represented by the **Customized Device** object class. This list is walked finding all resources that are tagged as **MISSING**.
2. Present the Missing Device menu for all **MISSING** devices. This menu lists each missing device with any children devices indented a few spaces. Missing Options Resolution Procedure can only be performed on the missing devices that do not have a parent also missing. See **MISSING RESOURCE Menu** for an example of this menu.

3. After selection of a device, present the Missing Device Resolution menu. The menu asks the user if the device was moved, removed, or turned off. The following selections may be chosen:
   a. `The resource has NOT been removed from the system, moved to another location or address, or turned off.`
      `This selection will determine why the resource was not detected.`
      1) Test the path to the missing device.
      2) If a device in the path is defective, then skip to the next ″missing″ device in the list that is not dependent on the one just named. Note that the defective device in the path has been added to the **FRU Bucket** object class by the Diagnostic Application (DA).
      3) Return to the step where the missing device menu was presented.
      4) If an *EnclDAName* DA is named, call it.
      5) If a problem was detected, skip to the next missing device in the list that has a different parent, and return to the step where the Missing Device menu was presented.
      6) If a missing device procedure was specified (`suptests` & `SUPTESTS_MS1`), then call it. Note that the DA should conclude that there is a problem.
      7) Skip to the next missing device in the list that is not dependent on the current missing device.
      8) Return to the step where the Missing Device menu was presented.
      9) If a missing device procedure was *not* specified, then add the device to the **FRU Bucket** object class by the **addfrub** subroutine. The default information is obtained from the **Predefined Device** object class.
   b. `The resource has been removed from the system and should be removed from the system configuration.`
      1) If the DA for the missing device supports the Missing Device Procedure 2 (`suptests==SUPTESTS_MS2`), then call the DA. The Diagnostic Controller does not automatically delete the device from the system configuration.
      2) Otherwise, flag the device to be deleted.
   c. `The resource has been moved to another location and should be removed from the system configuration.`
      1) Display a list of the new devices that are of the same type so that the user can identify where the missing device was moved. This list should contain a default selection for ″Not Listed″ in the event that the device was not detected in its new location, in which case a default service request number (SRN) should be generated.
      2) Assuming the user identified a new location:
         a) If the missing device has children which are non-detectable:
            - Present a menu to the user asking if the children should be reconfigured to the new device. The menu should contain a single selection for all of the devices and additional selections for the individual devices.
            - When a device is chosen, the parent field needs to be changed and the device configured. The **mkdev** command is used to configure the device.
         b) Delete the missing device and any children that have not been reconfigured.
   d. `The resource has been turned off and should be removed from the system configuration.`
      1) Flag the device to be removed from the configuration database.
   e. `The resource has been turned off but should remain in the system configuration.`
      1) Do nothing.
4. Once all the missing devices have been processed through one of the selections above, then perform the following:
   a. Report any problems found.
   b. Delete the devices that were previously flagged to be deleted.

c. If a new resource has been added, then display a list of the new devices. Ask the user if the list is correct.
   1) If Yes, then exit.
   2) If No, display predefined SRN indicating some new devices were not detected. Exit.

Error Log Analysis

Error log analysis does not test the resource. Instead this method searches the operating system error log for an entry (or entries) related to the resource. If an entry is found, then an analysis is performed on the error that was logged, and a determination is made by the Diagnostic Application as to whether the resource should be called out as being bad.

Error log analysis is performed via different methods with the Diagnostic Subsystem. One method is that error log analysis is performed automatically whenever a permanent hardware error is logged to the operating system error log. This method is called Automatic Error Log Analysis (DIAGELA).

A second method can be set up to run diagnostics automatically at a pre-set time of the day. This method is referred to as Periodic Diagnostics.

A third method can be run directly from the command line by using the **-e** flag with the diag command.

A fourth method is invoked automatically whenever diagnostics is ran in Problem Determination Mode after first starting diagnostics. This method is described below.

Running Problem Determination Mode in Diagnostics

If Problem Determination mode is selected upon entering diagnostics the first time, the Diagnostic Controller searches the operating system error log for any Permanent Hardware errors. If any errors were logged within the last 24 hours, the appropriate Diagnostic Application is called to analyze the error log. If an problem is suspected due to an error logged, a Problem Report screen will be presented to the user. If no problem is found, then the Resource Selection menu is displayed.

Periodic Diagnostics

Periodic testing of the disk drives and battery are enabled by default. The disk diagnostics perform disk error log analysis on all disks. The battery test checks the real time clock and NV-RAM battery.

Periodic diagnostics are performed in different ways, depending on the diagnostic version. Use the Periodic Diagnostics task to change the test times or to add other resources to the list.

AIX Version 3

Periodic testing of the disk drives and battery are performed by a root crontab entry. One entry in the root crontab table runs disk diagnostics at 3:01 a.m. each day. Another entry tests the battery at 4:01 a.m. each day. These tests can be disabled by editing the root crontab file. The disk entry is **/etc/lpp/diagnostics/bin/run_ela** while the battery entry is **/etc/lpp/diagnostics/bin/test_batt**.

Problems are reported by a message to the system console and logged in the error log. Diagnostics must be run for a SRN to be reported.

Running diagnostics in this mode is similar to using the**diag -c -e -d** ″**device**″ command.

AIX Version 4 and Itanium-based platform

Periodic testing is controlled by the Periodic Diagnostic Service Aid. The Periodic Diagnostic Service Aid allows error log analysis to be run on a hardware resource once a day. The battery and all disk drives are

enabled to run. The battery test (POWER-based platform only) is run at 4:00 a.m. each day, and error log analysis is performed on all the disk drives at 3:00 a.m. each day.

Other devices as necessary can be added into the Periodic Diagnostic Device list to run at various other times, if desired.

Problems are reported by a message to the system console and a mail message to all users of the system group. The message contains the SRN.

Running diagnostics in this mode for planar and memory tests is similar to using the **diag -c -d** ″**device**″ command. All other devices are invoked with the **'-e'** flag appended.

Technical Description

The Diagnostic daemon **diagd** executes once the **bos.diag** diagnostic package is installed. The **diagd** looks for customized entries in **CDiagAtt** odm database to determine which devices to run at which times. (For AIX 4.1, the database is **CDiagDev**.) The database is built when diagnostics are run or the Periodic Diagnostic Service Aid is run to change run times for devices. If the database has no entries (for example, when diagnostics have never been run), then default times are given to the ioplanar battery test and disk drives. The following is an example of **CDiagAtt** entries.

```
CDiagAtt->attribute = p_test_time
CDiagAtt->value = 9999 Do not test
        = 0400 Test at 4AM
```

The **diagd** sets a timer to wake up at the next scheduled time to run. Once **diagd** wakes up, the script **/usr/lpp/diagnostics/bin/diagela** is executed with the *-t* flag.

diagela checks the **PDiagAtt**->*test_mode* bit for the device to determine whether that device should be tested in this mode. If the bit is not set, **diagela** does not test the device. If the bit is set, diagnostics are run on the device with the *-e* (ELA) flag set.

Automatic Error Log Analysis (DIAGELA)

Automatic Error Log Analysis (**diagela**) provides the capability to do error log analysis whenever a permanent hardware error is logged. Whenever a permanent hardware resource error is logged and the **diagela** program is enabled, the **diagela** program is invoked. Automatic Error Log Analysis is enabled by default on all platforms.

The **diagela** program determines whether the error should be analyzed by the diagnostics. If the error should be analyzed, a diagnostic application will be invoked and the error will be analyzed. No testing is done. If the diagnostics determines that the error requires a service action, it sends a message to your console and to all system groups. The message contains the SRN, or a corrective action.

Running diagnostics in this mode is similar to using the **diag -c -e -d** ***device*** command.

Notification can also be customized by adding a stanza to the **PDiagAtt** object class. The following example illustrates how a customer's program can be invoked in place of the normal mail message:

```
PDiagAtt:

DClass = ""
DSClass = ""
DType = ""
attribute = "diag_notify"
value = "/usr/bin/customer_notify_program $1 $2 $3 $4 $5"
rep = "s"
```

If *DClass*, *DSClass*, and *DType* are blank, then the **customer_notify_program** will apply for ALL devices. Filling in the *DClass*, *DSClass*, and *DType* with specifics will cause the **customer_notify_program** to be invoked only for that device type.

Once the above stanza is added to the ODM data base, problems will be displayed on the system console and the program specified in the value field of the **diag_notify** pre-defined attribute will be invoked. The following keywords will be expanded automatically as arguments to the notify program:

$1	the keyword **diag_notify**
$2	the resource name that reported the problem
$3	the Service Request Number
$4	the device type
$5	the error label from the error log entry

In the case where no diagnostic program is found to analyze the error log entry, or analysis is done but no error was reported, a separate program can be specified to be invoked. This is accomplished by adding a stanza to the **PDiagAtt** object class with an attribute = **diag_analyze**. The following example illustrates how a customer's program can be invoked for this condition:

```
PDiagAtt:
        DClass = ""
        DSClass = ""
        DType = ""
        attribute = "diag_analyze"
        value = "/usr/bin/customer_analyzer_program $1 $2 $3 $4 $5"
        rep = "s"
```

If *DClass*, *DSClass*, and *DType* are blank, then the **customer_analyzer_program** will apply for ALL devices. Filling in the *DClass*, *DSClass*, and *DType* with specifics will cause the **customer_analyzer_program** to be invoked only for that device type.

Once the above stanza is added to the ODM data base, the program specified will be invoked if there is no diagnostic program specified for the error, or if analysis was done, but no error found. The following keywords will be expanded automatically as arguments to the analyzer program:

$1	the keyword **diag_analyze**
$2	the resource name that reported the problem
$3	the error label from the error log entry if invoked for ELA, *or* the keyword **PERIODIC** if invoked for Periodic Diagnostics, *or* the keyword **REMINDER** if invoked for providing a Diagnostic Reminder
$4	the device type
$5	the keyword: **no_trouble_found**, if analyzer was run, but no trouble was found; *or* **no_analyzer**, if analyzer not available.

To activate the Automatic Error Log Analysis feature, log in as root and type the following command:

```
/usr/lpp/diagnostics/bin/diagela ENABLE
```

To disable the Automatic Error Log Analysis feature, log in as root and type the following command:

```
/usr/lpp/diagnostics/bin/diagela DISABLE
```

Diagela can also be enabled and disabled using the Periodic Diagnostic Service Aid.

Loop Testing

Loop testing is the testing of a resource or resources multiple times under program control. The looping is controlled by the Diagnostic Controller. Loop testing is only supported when running in maintenance mode or service mode, and **Advanced Diagnostic Routines** have been chosen.

The user indicates that loop testing is desired at the **Test Method** menu. The rule associated with loop testing is that user interaction is only allowed on the first and last pass.

The diagnostic applications get notification that loop mode has been invoked by obtaining the value of *loopmode* in the **TMInput** object class. The following actions should be taken by the DA when *loopmode* has the following values:

LOOPMODE_ENTERLM
: The Diagnostic Application should perform any tests as usual, plus perform Error Log Analysis if running in Problem Determination mode.

LOOPMODE_INLM
: The Diagnostic Application should perform any tests as usual, and not Error Log Analysis.

LOOPMODE_EXITLM
: The Diagnostic Application should not perform any tests, nor perform Error Log Analysis. Instead cleanup procedures should be invoked to remove wrap plugs, etc, before exiting.

Chapter 5. Diagnostic Packaging

This chapter contains information on the various components that make up the Diagnostic Subsystem environment.

- Hardfile Packaging
- CDROM Packaging
- Diagnostic Supplemental Media

Hardfile Packaging

This chapter contains information on how the various diagnostic files are packaged. These packages are used by the install process to load diagnostics on the hardfile.

Software Packages and Filesets

Diagnostics is packaged into separate software packages and filesets. The base diagnostics support is contained in package **bos.diag**. The individual device support is packaged in separate **devices.[**type**].[**deviceid**]** packages.

The **bos.diag** package is split into three distinct filesets:

bos.diag.rte Contains the Controller and other base diagnostic code.
bos.diag.util Contains the Service Aids and Tasks.
bos.diag.com Contains the diagnostic libraries, kernel extensions, and development header files.

The **devices.[**type**].[**deviceid**]** packages are split into various distinct filesets. *type* usually signifies a bus type, or device class of devices. *deviceid* usually signifies a unique identifier for the device. For example:

devices.mca.8d77.rte Contains the device driver and configuration methods for the Micro Channel 8-bit SCSI I/O Controller.
devices.mca.8d77.diag Contains the Diagnostic Application and default catalog file for the device.

These packages/filesets are normally installed to a hardfile with the **installp** command.

Directory Structure Organization

The following shows the directory structures used by the Diagnostic Subsystem. New files created for diagnostic purposes should follow the same convention.

- **/etc/lpp/diagnostics/data** - Contains files that are created (Read/Write) by the diagnostics programs. Examples are the diagnostic report files created by the Diagnostic Controller.
- **/usr/lpp/diagnostics/bin** - Contains the Diagnostic Controller, and Service Aids/Tasks.
- **/usr/lpp/diagnostics/da** - Contains the Diagnostic Applications.
- **/usr/lpp/diagnostics/catalog** - Contains the default (English) catalog files used by all Diagnostic programs.
- **/usr/lpp/diagnostics/slih** - Contains the Second Level Interrupt Handlers used by the Test Units.
- **/usr/lpp/diagnostics/lib** - Contains the loadable Test Unit Libraries.

Note: The translated diagnostic catalog files are in **/usr/lib/nls/msg/[LANG]** directories.

© Copyright IBM Corp. 1997, 2001

CDROM Packaging (POWER-based only)

The Standalone Diagnostic CDROM contains all programs and applications necessary to run Diagnostics. This includes the latest version of the operating system, device drivers, device configuration methods, diagnostic applications, and ODM stanzas.

Device support that is not on the Diagnostic CDROM must be supported by Diagnostic Supplemental Media.

Starting with AIX 4.1, the Rock Ridge-based CDROM File System was used for the Diagnostic CDROM. The Rock Ridge CDROM File System supports directory levels deeper than 8, mixed-case file names and a file structure similar to operating system file systems.

Diagnostic Supplemental Media

A Diagnostic Supplemental Media contains all the necessary diagnostic programs and files required to test a particular resource when used with the Standalone Diagnostic CDROM. The supplemental is normally released and shipped with the resource as indicated on the diskette label.

The Process Supplemental Media task processes the diagnostic supplemental media.

The following topics describe the Diagnostic Supplemental Media and the contents in more detail.

- Diagnostic Supplemental Diskette Contents
- Example ODM Stanzas
- Example diagstartS Script File
- Example diagstart3S Script File
- Diagnostic Supplemental Diskette Labels

Diagnostic Supplemental Diskette Contents

A Diagnostic Supplemental Diskette must contain all files required to configure and test a device. Three special files, **diagstartS**, **diagS.dep**, and **diagcleanupS**, are required by the Standalone Diagnostic Package to maintain the software on the diskette. The diskette must be written in **cpio** format. Use the C36 block option on the **cpio** command to create the diskette. The following list describes each required file:

etc/diagstartS
: Shell script (with execute permission) to add the object class stanzas to the database, configure the devices, and so on. See the example **diagstartS** shell script file. *This file must be the first file on diskette.*

etc/diagS.dep
: Dependency file. This file is a list of all files on the diskette. Each file must be listed with its full path name.

etc/diagcleanupS
: Cleanup script file. This script should perform any cleanup necessary after the supplemental has been processed and run; for example, restoring the ODM database to its original condition if the supplemental changed some of the original values.

etc/stanzas/*device* **.add**
: Stanza file for the device. The stanzas must include the **PdDv, PdCn, PdAt**, and **PDiagRes** information required for the device.

> **Note:** Use **PDiagRes** if the device is only supported on AIX 4.2 and later. If the supplemental can be used on a pre-AIX 4.2 system, then **PDiagDev** must be used.

usr/lib/drivers/*devicedd*
: Device driver for the device. The *devicedd* variable should be the name of the device driver.

usr/lib/methods/*devicecfgmethod* and **usr/lib/methods/***deviceunconfigmethod*
: Methods necessary to define, configure, undefine, and unconfigure the device. The names must be the same referenced by the **PdDv** method objects. *Do not* include any methods that are already part of the operating system. Include only the unique methods used by this device.

usr/lib/methods/*devicedesc***.cat**
: Device description catalog file **devicedesc.cat** should be the name of the catalog file referenced by the **PdDv** catalog object. The device description file should contain the description of the device shown when using the **lsdev** or **lscfg** command.

usr/lpp/diagnostics/da/*ddevice*
: Diagnostic Application (DA) for the device. The *ddevice* variable should be the name of the DA, which is the same name referenced by the **PDiagRes DaName** object.

usr/lpp/diagnostics/catalog/default/*ddevice***.cat**
: DA message catalog for the device. The DA menus are included in this file.

This message catalog file also contains FRU information. The set number used must be the same number referenced by the **PDiagRes PSet** object.

Note: If the supplemental diskette being developed is for a graphics adapter that can be used as a console device, then the suffix **3S** should be used instead of **S**. For example, the file **etc/diagstartS** should be **etc/diagstart3S**, **etc/diagS.dep** should be **etc/diag3S.dep**, and **etc/diagcleanupS** should be **etc/diagcleanup3S**.

usr/lpp/diagnostics/slih/*device_slih*
: Second Level Interrupt Handler for the device.

usr/lpp/diagnostics/lib/*lib_device*
: Device Test Unit loadable library.

Example ODM Stanzas

```
PdDv:
              type = xyz
              subclass = mca
              class = adapter
              catalog = xyz.cat
              setno = 1
              msgno = 1
              Define = /usr/lib/methods/definexyz
              Configure = /usr/lib/methods/cfgxyz
              Undefine = /usr/lib/methods/udefinexyz
              Unconfigure = /usr/lib/methods/ucfgxyz
              led = 0x902
              fru = 1                         1 if device is FRU
                                              2 if parent is FRU
PDiagRes:
              Uniquetype = adapter/mca/xyz
              PSet = 1
              DaName = dxyz
              PkgBlock = S
              Menu = 21
              DNext = 1
              SupTests = 7
```

For a description of all fields in **PDiagRes**, refer to Predefined Diagnostic Resource Object Class.

```
/usr/lib/methods/xyz.cat:
        1 1     XYZ adapter

/usr/lpp/diagnostics/catalog/default/dxyz.cat:
        1 1     Description of FRU1
        1 2      Description of FRU2
        2 1      DA menus, etc
```

Example diagstartS Script File

```
# DIAG S
# Do not erase top line. Chkdskt searches for the string DIAG S
#
# COMPONENT_NAME: DIAGBOOT - DIAGNOSTIC SUPPLEMENTAL DISKETTE
#
# FUNCTIONS: Diagnostic Diskette Supplemental Script File
#
# ORIGINS: 27
#
# (C) COPYRIGHT International Business Machines Corp. 1991
# All Rights Reserved
# Licensed Materials - Property of IBM
#
# US Government Users Restricted Rights - Use, duplication or
# disclosure restricted by GSA ADP Schedule Contract with IBM
# Corp.
#
configure=0
# See if there is a need to add stanzas to data base.
# This is done by searching the /etc/addfile for your stanza file
# name. If not found, add stanzas and call /etc/cfgmgr to
# configure the resources that are needed to be tested.
cd /etc/stanzas
set 'echo *'
ADD='echo $1'
# Warning: If your stanza is already in PDiagDev, DO NOT ADD another.
for i in '/bin/cat /etc/addfile'
do
        if [ $i = $ADD ]
        then
                configure=1
                break
        fi
done
# Check the PDiagDev for a DType/DSClass equal to your stanza
# before adding  in the new one. If not found, add stanzas and
# call /etc/cfgmgr to configure the resources that are needed to
# be tested.
if [ $configure = 0 ]
then
        # Check the PDiagDev for a DType/DSClass equal to your
        # stanza before adding  in the new one.
        # If not found, add stanzas and call /etc/cfgmgr to
        # configure the resources that are needed to be tested.
        X='odmget -q"DType=DeviceType and DSClass=SubClass" PDiagDev'
if [ "X$X" != X ]
then
        # save the data and read it in later with the diagcleanup script.
        odmget -q"DType=DeviceType and DSClass=SubClass" PDiagDev > /tmp/mysave
        odmdelete -q"DType=DeviceType and DSClass=SubClass" -o PDiagDev
```

```
fi
        for i in *.add
        do
                odmadd $i                       >>$F1 2>&1
                echo $i >> /etc/addfile
                rm $i                           >>$F1 2>&1
        done
        /etc/cfgmgr     -t -d                   >>$F1 2>&1
else
        for i in *.add
        do
                rm $i                           >>$F1 2>&1
        done
fi
exit 0
```

Example diagstart3S Script File

```
# DIAG 3S
# Do not erase top line. Chkdskt searches for the string DIAG 3S
#
# COMPONENT_NAME: DIAGBOOT - DIAGNOSTIC GRAPHIC SUPPLEMENTAL
# DISKETTE
#
# FUNCTIONS: Diagnostic Diskette Supplemental Script File
#
# ORIGINS: 27
#
# (C) COPYRIGHT International Business Machines Corp. 1994
# All Rights Reserved
# Licensed Materials - Property of IBM
#
# US Government Users Restricted Rights - Use, duplication or
# disclosure restricted by GSA ADP Schedule Contract with IBM
# Corp.
#


configure=0
# See if there is a need to add stanzas to data base.
# This is done by searching the /etc/addfile for your stanza file
# name. If not found, add stanzas and call /etc/cfgmgr to
# configure the resources that are needed to be tested.

cd /etc/stanzas
set 'echo *'
ADD='echo $1'


# Warning: If your stanza is already in PDiagDev, DO NOT ADD
another one in again.

for i in '/bin/cat /etc/addfile'
do
        if [ $i = $ADD ]
        then
                configure=1
                break
        fi
done

# Check the PDiagDev for a DType/DSClass equal to your stanza
# before adding  in the new one. If not found, add stanzas and
# call /etc/cfgmgr to configure the resources that are needed to
# be tested.

if [ $configure = 0 ]
then
        # Check the PDiagDev for a DType/DSClass equal to your stanza
        # before adding  in the new one. If not found, add stanzas and
        # call /etc/cfgmgr to configure the resources that are needed to
```

```
        # be tested.
        X='odmget -q"DType=DeviceType and DSClass=SubClass" PDiagDev'
if [ "X$X" != X ]
then
        # save the data and read it in later with the diagcleanup script.
        odmget -q"DType=DeviceType and DSClass=SubClass" PDiagDev > /tmp/mysave
        odmdelete -q"DType=DeviceType and DSClass=SubClass" -o PDiagDev

fi
        for i in *.add
        do
                odmadd $i                        >>$F1 2>&1
                echo $i >> /etc/addfile
                rm $i                            >>$F1 2>&1
        done
        /etc/cfgmgr      -t -d                   >>$F1 2>&1
else
        for i in *.add
        do
                rm $i                            >>$F1 2>&1
        done
fi

echo > /tmp/3S           # flag that indicates diskette read.
exit 0
```

Diagnostic Supplemental Diskette Label

Each Diagnostic Supplemental Diskette must have a label. The label should state the lowest version of the operating system that the diskette supports. For instance, if the supplemental diskette was initially built on AIX 4.1, then the version of the diskette should say 4.1.

Chapter 6. Diagnostic Debugging Hints

This section has hints on how to debug applications in a Diagnostic environment. The following areas are covered:

- Debugging Hints for Diagnostic Applications
- Debugging Hints for Diagnostic Kernel Extension
- Debugging Hints for CDROM Using Patch Diskettes

Debugging Hints for Diagnostic Applications

The Diagnostic Controller uses the process ID **(PID)** of the DA to determine which **TMInput** object class entry to use for the DA during execution. To debug the DA, run the following:

```
export DIAG_DEBUG=1
```

- Run diagnostics as usual against your resource once.

  ```
  odmget TMInput > /tmp/tminput.add # save off contents of TMInput.
  ```

- Edit the **/tmp/tminput.add** file and set the *pid* field to 0.

  ```
  odmdelete -o TMInput      # delete what is currently in TMInput.
  odmadd /tmp/tminput.add           # add new contents of TMInput.
  ```

- Execute the code debugger against the DA.

If the Diagnostic Application uses a kernel extension or Second Level Interrupt Handler, you may have to perform the following before trying to load and debug the DA.

- Load the kernel extension. This can be done by running diagnostics once on the device, and then exiting. The Controller will normally load any kernel extensions needed by the DA. When exiting Diagnostics, the Controller does not unload the extensions, so it should still be loaded during the debugging,
- Export the diagnostic environment variable **DIAGX_SLIH_DIR** to **/usr/lpp/diagnostics/slih**.

Debugging Hints for Diagnostic Kernel Extension

- Starting Trace for Diagnostic Kernel Extension
- Running Trace for Diagnostic Kernel Extension in the Background
- Finding the Right Address
- Looking at an Illegal Trap

Starting Trace for Diagnostic Kernel Extension

The Diagnostic Controller loads the Kernel Extensions for each device that requires it. This is specified by the **PDiagRes**->*KernExt* ODM stanza for the device. If using **DIAGEX** or **PDIAGEX**, there is a trace hook built in for debugging purposes.

To use this trace hook, you first must make sure that the **trace** command is installed. This command is part of the bos.sysmgt.trace fileset.

To run trace, perform the following:

```
trace -j 355            // Invoke trace
> trcon                 // Start trace
> !diag -d "device_name"// Run diagnostics against the device
> trcoff                // Stop trace
> quit                  // Quit
```

To generate a trace file, perform the following:

```
trcrpt -o /tmp/diagex.trc
```

This trace file will contain all the steps performed by the diagnostic kernel extension. To understand the tags, you must use the source code.

Running Trace for Diagnostic Kernel Extension in the Background

The Diagnostic Controller loads the Kernel Extensions for each device that requires it. This is specified by the **PDiagRes->***KernExt* ODM stanza for the device. If you are using **DIAGEX** or **PDIAGEX**, there is a trace hook built in for debugging purposes.

To use this trace hook, first make sure that the **trace** command is installed. This command is part of the bos.sysmgt.trace fileset.

To run trace in the background, enter:

```
trace -a -j 355 -L < length of file > -o < filename >
```

The **-L** flag overrides the default trace log file size of 1 MB with the value stated. Specifying a file size of zero sets the trace log file size to the default size. The **-o** flag outputs trace data to a specific trace log file.

To generate a trace file, enter:

```
trcrpt < filename > < output filename >
```

This trace file will contain all the steps performed by the diagnostic kernel extension. To understand the tags, you must use the source code.

Note: You can only have one trace running at a time.

To stop a trace, enter:

```
trcstop
```

Finding the Right Address

Note: The following examples are based on a particular debugger. The concepts shown can be applied using the debugger available to you.

While in the Kernel Debugger, there is a structure that can be searched that gives the address of the trace buffer and first device handle. For DIAGEX, this structure is **diag_cntl**. For PDIAGEX, it is **pdiag_cntl**. Use the *map* command to get the address of the structure.

For instance, for PDIAGEX:

1.
```
>0> map pdiag_cntl
pdiag_cntl:0x0123F220, type:CSECT Definition
```
2. Use that address and display 100 words:
```
>0> d 123F220 100
0123F220   FFFFFFFF FFFFFFFF 05C8A400 00000764    |...............d|
0123F230   64677874 72616365 544F5021 21212100    |dgxtraceTOP!!!!.|
0123F240   72775F61 00000004 00000000 00000000    |rw_a............|
0123F250   67697042 00000018 00000004 2FF3B270    |gipB......../..p|
0123F260   67697064 000000C0 00000000 3D7FF018    |gipd........=...|
0123F270   67697045 00000000 3D7FF018 00000000    |gipE....=.......|
0123F280   72775F62 00000001 00000001 00000001    |rw_b............|
0123F290   72775F45 00000000 00000000 00000000    |rw_E............|
0123F2A0   52656445 00000000 20001111 00000000    |RedE.... .......|
0123F2B0   57727442 05C8A200 00000004 00000014    |WrtB............|
0123F2C0   5772742B 14000000 00000001 00000001    |Wrt+............|
0123F2D0   5772742B 00000001 0000007B 00000000    |Wrt+.......{....|
```

```
0123F2E0   72775F42 05C8A200 00000001 00000001   |rw_B............|
0123F2F0   72775F2B 00000004 00000014 00000001   |rw_+............|
0123F300   72775F2B 0000007B 14000000 00000001   |rw_+...{........|
0123F310   66685F42 05C8A200 05C8A200 00000000   |fh_B............|
```

- The first and second words, `FFFFFFFF`, are locks. Ignore them.
- The third word (in bold) is a pointer to the linked list of device handles.
- The fourth word is the start of the internal trace table.
- `dgxtraceTOP!` defines the TOP of the trace table.
- `dgxtraceBOT!` defines the end of the trace table.

3. The current pointer can be found by searching from this point for `dgxtraceCUR!`:

   ```
   >0> find dgxtraceCUR 123F220
   01240FC0   64677874 72616365 43555221 21212121   |dgxtraceCUR!!!!!|
   ```

   Work backwards from this point to see exactly what events have taken place to this point.

4. As far as the device handles are concerned, display 100 words to see the data associated with the device at that address (the third word from 2.b above):

   ```
   >0> d 05C8A400 100
   05C8A400   00000000 012438B8 00040040 0000000D   |.....$8....@....|
   05C8A410   00000003 000000C0 0000002C 00000000   |...........,....|
   05C8A420   011759FC 05F1D000 00000000 60054335   |..Y.........'.C5|
   05C8A430   00000000 00000000 00000070 000000C0   70 is slot#, C0 is bus id#
   05C8A440   00000004 007FF800 00000100 00000000    4 is bus type 7ff800 is io
   05C8A450   00000100 00000000 00000000 00000000       address of the bus
   ```

   The 8th word is a pointer to the next device in the linked list. In this case the 8th word is **00000000**, indicating this is the only device.

Looking at an Illegal Trap

In some instances, an Illegal Trap Instruction may occur if some application unloads their SLIH or kernel extension, without having previously unpinned its memory. This can also happen if the Diagnostic Kernel Extension **close** routine is not called on exit.

If this happens when the debugger is enabled, a screen similar to the following may appear. The appearance of `ff_free` in the dump is the indicator that an application did not unpin some code before unloading.

The address passed to `ff_free` is in (r29) or r30. Use the (**s**)creen command to trace back until you see a familiar function name. In the following example, the SLIH mps_interrupt was indicated.

1. Trap Occurs:

   ```
   GPR0  00000000 2FF3B188 00192DF0 00000016 007FFFFF C0000000 00009030 2FF3B400
   GPR8  00000000 00000000 00000000 00000010 0014032C DEADBEEF DEADBEEF DEADBEEF
   GPR16 DEADBEEF DEADBEEF 200004B0 DEADBEEF DEADBEEF DEADBEEF 2FF3B2C0 00000000
   GPR24 00000000 00161BF8 C0000420 03762428 0015FF40 01A1C5A0 01A1C5A8 0015FF40

   MSR 00029030  CR   44224828  LR   0014032C  CTR   000908A8  MQ   00000000
   XER 00000000  SRR0 00140334  SRR1 00029030  DSISR 40000000  DAR  00000000

   IAR 00140334  (ORG+00140334)  ORG=00000000   Mode: VIRTUAL
   00140330   5400D97E 0C800000 387F0000 4BECADC5   |T..~....8...K...|
                      |    tweqi   r0,0x0
   00140340   81810058 30210050 7D8803A6 BBA1FFF4   |...X0!.P}.......|

                      |
   00140330   5400D97E 0C800000 387F0000 4BECADC5   |T..~....8...K...|
   00140340   81810058 30210050 7D8803A6 BBA1FFF4   |...X0!.P}.......|
   00140350   4E800020 00000000 00002041 80030100   |N.. ...... A....|
   00140360   00000000 00000174 00076666 5F667265   |.......t..ff_fre|
   00140370   65000000 80E20328 BF81FFF0 7C0802A6   |e......(....|...|
   ```

```
00140380   2C070000 90010008 9421FFB0 3B830000   |,........!..;...|
00140390   41820050 80E201E8 38640000 83810040   |A..P....8d.....@|

Illegal Trap Instruction Interrupt in Kernel

>0>
```

2. Use (**s**)creen to display contents of R29:

```
>0> s 1A1C5a0 100
GPR0  00000000 2FF3B188 00192DF0 00000016 007FFFFF C0000000 00009030 2FF3B400
GPR8  00000000 00000000 00000000 00000010 0014032C DEADBEEF DEADBEEF DEADBEEF
GPR16 DEADBEEF DEADBEEF 200004B0 DEADBEEF DEADBEEF DEADBEEF 2FF3B2C0 00000000
GPR24 00000000 00161BF8 C0000420 03762428 0015FF40 01A1C5A0 01A1C5A8 0015FF40

MSR 00029030  CR   44224828  LR   0014032C  CTR   000908A8  MQ   00000000
XER 00000000  SRR0 00140334  SRR1 00029030  DSISR 40000000  DAR  00000000

IAR 00140334  (ORG+00140334)  ORG=00000000   Mode: VIRTUAL
00140330   5400D97E 0C800000 387F0000 4BECADC5   |T..~....8...K...|
                    |    tweqi   r0,0x0
00140340   81810058 30210050 7D8803A6 BBA1FFF4   |...X0!.P}.......|

           |
01A1C5A0   01A29850 0000A518 01DF0004 325E9F94   |...P........2^..|
01A1C5B0   00000000 00000000 00481007 010B0001   |.........H......|
01A1C5C0   00000BF0 0000010C 00000000 000000E4   |................|
01A1C5D0   00000000 00000000 000000F0 00020001   |................|
01A1C5E0   00020002 00040003 00020003 314C0000   |............1L..|
01A1C5F0   00000000 00000000 00000000 00000000   |................|
01A1C600   00000000 2E746578 74000000 00000000   |.....text.......|
```

3. Press enter until you find a function name:

```
>0> enter several times
GPR0  00000000 2FF3B188 00192DF0 00000016 007FFFFF C0000000 00009030 2FF3B400
GPR8  00000000 00000000 00000000 00000010 0014032C DEADBEEF DEADBEEF DEADBEEF
GPR16 DEADBEEF DEADBEEF 200004B0 DEADBEEF DEADBEEF DEADBEEF 2FF3B2C0 00000000
GPR24 00000000 00161BF8 C0000420 03762428 0015FF40 01A1C5A0 01A1C5A8 0015FF40

MSR 00029030  CR   44224828  LR   0014032C  CTR   000908A8  MQ   00000000
XER 00000000  SRR0 00140334  SRR1 00029030  DSISR 40000000  DAR  00000000

IAR 00140334  (ORG+00140334)  ORG=00000000   Mode: VIRTUAL
00140330   5400D97E 0C800000 387F0000 4BECADC5   |T..~....8...K...|
                    |    tweqi   r0,0x0
00140340   81810058 30210050 7D8803A6 BBA1FFF4   |...X0!.P}.......|

           |
01A1CDF0   41820010 306300CC 48000479 80410014   |A...0c..H..y.A..|
01A1CE00   38600000 4800000C 3860FFFF 48000004   |8'..H...8'..H...|
01A1CE10   80010088 7C0803A6 30210080 BBC1FFF8   |....|...0!......|
01A1CE20   4E800020 00000000 00002041 80020201   |N.. ...... A....|
01A1CE30   00000000 00000780 000D6D70 735F696E   |..........mps_in|
01A1CE40   74657272 75707400 00000000 BDA1FFB4   |terrupt.........|
01A1CE50   80A20004 39C30000 80650060 7C0802A6   |....9....e.'|...|
```

Diagnostic Patch Diskette Procedure

Patch diskettes can be made to help in the debug of problems that occur when running diagnostics from the Diagnostic CDROM. Three types of diskettes can be used:

- Diagnostic Configuration Diskette
- Diagnostic Patch Diskette
- Diagnostic Debug Diskette

The Diagnostic Patch Diskette purpose is to allow file replacement from diskette, overriding the file(s) on the CDROM. All diskettes are in backup/restore format. The Diagnostic Debug diskette can be combined with the other two to allow command line debugging as well as file replacement.

Diagnostic Configuration Diskette

The Diagnostic Configuration diskette has two main purposes.

The first purpose of the Diagnostic Configuration diskette is to allow the refresh rate of the graphics adapter to be set to a different value than the default. The default value is 60Hz. If the graphics display's refresh rate is 77 Hz, then set the refresh rate to 77.

The second purpose of the Diagnostic Configuration diskette is to allow a terminal attached to any RS232 or RS422 adapter to be selected as a console device. The default device is a RS232 tty attached to the first standard serial port(S1).

Each of these can be accomplished by using the *Create Customized Configuration Diskette* Task.

A valid Diagnostic Configuration Diskette contains the following files:

- ./.signature
- ./CONSDEF
- ./REFRESH

The **.signature** file contains a single line describing the diskette purpose. For this diskette, the description should be */etc/diagconf*.

Diagnostic Patch Diskette

The Diagnostic Patch diskette is used to patch failing applications until a new release of the Diagnostic CDROM is available. This diskette may also be used in development to help in the debug of why a particular application is failing.

A valid Diagnostic Patch Diskette contains the following files:

- ./.signature
- ./etc/diagpatch
- ./etc/[applications]

The **.signature** file contains a single line describing the diskette purpose. For this diskette, the description should be **/etc/diagpatch**. The **/etc/diagpatch** file is a Korn shell script file that is used to remove the application first from the RAM file system, then links the new application to the old one. The **/etc/diag patch** file must be executable. Following is an example:

```
#!/bin/ksh
#### begin diagpatch

# Files to be replaced on the RAM file system must first be removed,
# then linked from /etc to /usr/lpp/....[or correct location]

### Replacing a diagnostic application
rm /usr/lpp/diagnostics/da/dxspa
ln -s /etc/dxspa /usr/lpp/diagnostics/da/dxspa
```

Diagnostic Debug Diskette

A valid Diagnostic Debug Diskette contains the following files:

- ./.signature
- ./etc/NOKEYPOS

The **.signature** file contains a single line describing the diskette purpose. For this diskette, the description could be either */etc/diagpatch* or */etc/diagconf*. The script file does not need to be present if files are not being replaced.

The */etc/NOKEYPOS* file is a zero length file.

> **Note:** This function can be combined with either the Patch or Configuration diskette by simply adding the /etc/NOKEYPOS file to either diskette.

Chapter 7. Code Examples

This chapter contains various sample 'C' programming code for both the Application Test Unit and Diagnostic Application code. These samples are meant for review to understand the concepts and library routines used. None of these will compile clean. They are included here as reference only.

- Example {DEVICE}_ERR_DETAIL.H: TU Specific Outputs
- Example {DEVICE}_INPUT_PARAMS.H: TU Specific Inputs
- Example TU Local Header File
- Example TU exectu Function
- Example TU Open/Close Device Interface
- Example TU Makefiles
- Example C Source File for TU Interrupt Handler
- Example TU Interrupt Handler Makefile
- Example Diagnostic Application
- Example Diagnostic Application Message File

Example {DEVICE}_ERR_DETAIL.H: TU Specific Outputs

```
/*
 *   COMPONENT_NAME: TU_DEVICE
 *
 *   FUNCTIONS: SAMPLE Header file for TU Error Detail (OUTPUT)
 *
 */

#ifndef _h_device_err_detail
#define _h_device_err_detail

/*
 * ERROR_DETAILS structure and related definitions follow.
 *
 * These structures are used to provide detailed error information
 * for some of the errors that are detected by the test units.
 * Whether the detailed error is available for a particular TU and error
 * code is documented in the TU Component Interface Specification, and
 * the actual source files where that error code is defined.
 */

/*********************************************************/
/* The following structures are examples. Modify        */
/* as needed.                                            */
/*********************************************************/

typedef struct {
      unsigned long int error_code;
      unsigned long int crc_expected;
      unsigned long int crc_actual;
} CRC_ERROR_DETAILS;

typedef struct {
      unsigned long int error_code;
      unsigned long int miscompare_address;
      unsigned long int expected_data;
      unsigned long int actual_data;
    } DMA_ERROR_DETAILS;

typedef union {
      unsigned long int         error_code;
      CRC_ERROR_DETAILS         crc_test;
```

© Copyright IBM Corp. 1997, 2001

```
      DMA_ERROR_DETAILS         dma_test;
} ERROR_DETAILS;

/* The following is required by <diag/tucb.h> file */
#define OUTPUT_DATA ERROR_DETAILS

 #endif
```

Example {DEVICE}_INPUT_PARAMS.H: TU Specific Inputs

```
/*
 *   COMPONENT_NAME: TU_DEVICE
 *
 *   FUNCTIONS: SAMPLE TU Input Parameters Header File
 *
 */

#ifndef _h_device_input_params
#define _h_device_input_params

/*
 * INPUT_DATA structure and related definitions follow.
 *
 * These structures are used to provide detailed input data information
 * for some of the test units. This data is only used in manufacturing
 * or other special case test areas.
 */

/*******************************************************/
/* The following structures are examples. Modify       */
/* as needed.                                          */
/*******************************************************/

typedef struct {
      unsigned long int         mfg_mode;
} TU_SPECIFIC_INPUT;

/* The following is required by the <diag/tucb.h> header */
#define INPUT_DATA TU_SPECIFIC_INPUT

#endif
```

Example TU Local Header File

```
/*
 *   COMPONENT_NAME: TU_DEVICE
 *
 *   FUNCTIONS: TU Header file
 */

#ifndef _h_tu
#define _h_tu

#include <diag/tucb.h>
#include <sys/pdiagex_dds.h>

#define TU_SUCCESS 0
#define TU_DEVICE_BUSY  1
#define TU_CHILD_BUSY   2
#define TU_SOFTWARE_ERROR 3
#define TU_INVALID_PARAM  4
#define TU_INCORRECT_STATE 5

#define TU_OPEN   0x01
       etc, etc
#define TU_CLOSE  0xEFFF
```

```
typedef struct {
      int adapter_diagnose_state;
      pdiagex_dds_t dds;
      pdiag_info_handle_t pdiagex_handle;
} TU_GLOBAL_DATA;

#endif
```

Example TU executu Function

```
/*
 * COMPONENT_NAME: (TU_DEVICE) Device Adapter Test Units
 *
 * FUNCTIONS: executu
 */


/*   FILE NAME: device_executu.c                                          */
/*   FUNCTION:  Device Adapter Application Test Units.                    */
/*                                                                        */
/*    This source file contains source code for the Device adapter's      */
/*    Application Test Units to aid in various testing environments       */
/*    of the device   adapter. These test units provide a basic inter-    */
/*    face between the diagnostic application program and functions       */
/*    written in the diagnostic extension (pdiagex) which provide direct  */
/*    access to the device without the need for a device driver.          */
/*                                                                        */
/*                                                                        */
/*   EXTERNAL PROCEDURES CALLED:                                          */
/*                                                                        */

/* INCLUDED FILES */
#include <sys/types.h>
#include <stdio.h>
#include <errno.h>

#include "device_input_params.h"
#include "device_err_detail.h"
#include "tu.h"
#include <diag/tucb.h>

/*- global variables -*/
TU_GLOBAL_DATA *tu_data;

/*- extern functions -*/
extern void Do_INIT_TUS(TU_TYPE *, TU_GLOBAL_DATA *, TU_RETURN_TYPE *tu_rc);
extern void Do_TERM_TUS(TU_TYPE *, TU_GLOBAL_DATA *, TU_RETURN_TYPE *tu_rc);

/*
 * NAME: executu
 *
 * FUNCTION:  Execute a specific Resource Test Unit.
 *
 * EXECUTION ENVIRONMENT:
 *      This routine is called as a subroutine of a diagnostic application.
 *
 * NOTES:  This routine is used as the interface between an application
 *         and the test units for a Resource.
 *
 */

ulong
executu(TU_TYPE *dev_tucb, TU_INFO_HANDLE *tu_handle, TU_RETURN_TYPE *tu_rc)
{
        int loopcount;
```

```
int mfg_flag=0;

/* Set the tu_handle pointing to the global tu structure data */
/* if the first time in. Also initialize elements.            */
if ( *tu_handle == (TU_INFO_HANDLE *)NULL ) {
        tu_data = (TU_GLOBAL_DATA *)calloc(1,sizeof(TU_GLOBAL_DATA));
        *tu_handle = (TU_INFO_HANDLE *)tu_data;
}

/* number of times to repeat a command */
loopcount = dev_tucb->parms.loop;

/*--------------------------------------*/
/* assure adapter is proper state       */
/* before attempting test unit          */
/*--------------------------------------*/
if ((dev_tucb->parms.tu != 1) &&      /* for tus other than init tu */
    (tu.adapter_diagnose_state != 1)){ /* test for NOT Diag state */
        tu_rc->major_rc = TU_INCORRECT_STATE;
        if ( dev_tucb->parms.msg_file != (FILE *)NULL)
                fprintf( dev_tucb->parms.msg_file, "TU is not 1, and                           not in correct state
        return(tu_rc->major_rc);       /* must be in diagnose state */
}
else if ((dev_tucb->parms.tu == 1) &&  /*- for tu 1 only           -*/
                                       /*- test for Diagnose state -*/
            (tu.adapter_diagnose_state == 1)) {
        tu_rc->major_rc = TU_SUCCESS;
        if ( dev_tucb->parms.msg_file != (FILE *)NULL)
                fprintf( dev_tucb->parms.msg_file, "TU is 1, and is in                             correct state.
        return(tu_rc->major_rc);      /*- already in diagnose state -*/
}

switch (dev_tucb->parms.tu) {

        /*--------------------------------------*/
        /*- INITIALIZE Test Unit  #1           -*/
        /*--------------------------------------*/
case TU_OPEN:
        {
                (void) Do_INIT_TUS(dev_tucb, tu_data, tu_rc);
                if (tu_rc->major_rc == TU_SUCCESS)
                        /*- flag Diagnose state -*/
                        tu.adapter_diagnose_state = 1;
                if ( dev_tucb->parms.msg_file != (FILE *)NULL)
                        fprintf( dev_tucb->parms.msg_file,
                           "TU is 1 status = %d\n", tu_rc->major_rc);
                break;

        }

        /*--------------------------------------*/
        /*- Other Test Units                   -*/
        /*--------------------------------------*/

        /*--------------------------------------*/
        /*- TERMINATE Test Unit  #EFFF         -*/
        /*--------------------------------------*/
case TU_CLOSE:
        {
                (void) Do_TERM_TUS(dev_tucb, tu_data, tu_rc);
                if (tu_rc->major_rc == TU_SUCCESS)
                        /*- reset Diagnose state -*/
                        tu.adapter_diagnose_state = 0;
                if ( dev_tucb->parms.msg_file != (FILE *)NULL)
                        fprintf( dev_tucb->parms.msg_file,
                           "TU is 2 status = %d\n", tu_rc->major_rc);
                break;
```

```
                }

                /*---------------------------------------*/
                /* Unknown tu number                     */
                /*---------------------------------------*/
        default:
                tu.rc.major_rc = TU_INVALID_PARAM;

        }  /* end of switch on tu number */

        /* If the OUTPUT_DATA is wanted by the calling application, */
        /* then the tucb->data_log should not be NULL. If so, then  */
        /* this structure may be used.                              */

        if ( dev_tucb->parms.data_log )
                dev_tucb->parms.data_log->error_code = TU_FAILED;

        /* If the INPUT_DATA is specified by the calling application, */
        /* then the tucb->tu_data should not be NULL. If so, then     */
        /* get specific input data from this structure                */

        if ( dev_tucb->parms.tu_data )
                mfg_flag = dev_tucb->parms.tu_data->mfg_mode;

        return (tu_rc->major_rc);

}   /* end of exectu()--------------------------------------------------------*/
```

Example TU Open/Close Device Interface

```
/*
 * COMPONENT_NAME: (TU_DEVICE) Resource Interface Access Code
 *
 * FUNCTIONS: Do_INIT_TUS
 *            Do_TERM_TUS
 */


/*   FILE NAME: device_interface.c                                        */
/*   FUNCTION:  Device Adapter Application Interface Code                 */
/*                                                                        */
/*    This source file contains source code for the Device adapter's      */
/*    Application Test Units to aid in various testing environments       */
/*    of the device   adapter. These test units provide a basic inter-    */
/*    face between the diagnostic application program and functions       */
/*    written in the diagnostic extension (pdiagex) which provide direct  */
/*    access to the device without the need for a device driver.          */
/*                                                                        */
/*                                                                        */

/* INCLUDED FILES */
#include <sys/types.h>
#include <stdio.h>
#include <errno.h>
#include <sys/intr.h>
#include <sys/dma.h>

#include "device_err_detail.h"
#include "tu.h"
#include <diag/tucb.h>
#include <sys/pdiagex_dds.h>

/****************************************/
/*- INITIALIZE Test Unit  #1           -*/
/****************************************/
void
```

```
Do_INIT_TUS(TU_TYPE *dev_tucb, TU_GLOBAL_DATA *tu_data, TU_RETURN_TYPE *tu_rc)
{
        int     rc;
        void    *ih_handle;

        /* Set initial tu success status */
        tu_rc->major_rc = TU_SUCCESS;

        /*- unconfigure device/children and place device in diagnose state -*/
        rc = pdiag_diagnose_state(dev_tucb->resource_name);
        if (rc != 0) {          /*- test unit failed to complete normally -*/
                tu_rc->major_rc = TU_DEVICE_BUSY;
                tu_rc->minor_rc = rc;
                return;
        }

        tu_data->adapter_diagnose_state = 1;

        /* Get all the device attributes for the dds structure */
        rc = get_dds( dev_tucb, tu_data );
        if (rc != 0) {          /*- test unit failed to complete normally -*/
                tu_rc->major_rc = TU_SOFTWARE_ERROR;
                tu_rc->minor_rc = rc;
                return;
        }

       /*************************************************************
        * Call pdiag_open
        * This also loads the interrupt handler
        *************************************************************/
        /* Open the device for testing via PDIAGEX */
        rc = pdiag_open( dev_tucb->resource_name, &tu_data->dds, "device_intr",
          &tu_data->pdiagex_handle);
        if (rc != 0) {          /*- test unit failed to complete normally -*/
                if ( dev_tucb->parms.msg_file != (FILE *)NULL)
                        fprintf( dev_tucb->parms.msg_file,
                                 "pdiagex open rc = %d\n", rc);
                tu_rc->major_rc = TU_DEVICE_BUSY;
                tu_rc->minor_rc = rc;
                return;
        }

        return;                 /*- normal completion -*/
}

/*****************************************/
/*- TERMINATE Test Unit  #EFFF         -*/
/*****************************************/
void
Do_TERM_TUS(TU_TYPE *dev_tucb, TU_GLOBAL_DATA *tu_data, TU_RETURN_TYPE *tu_rc)
{
        int rc;

        tu_rc->major_rc = TU_SUCCESS;

        /* Close/terminate device from PDIAGEX     */
        /* This also unloads the interrupt handler */
        rc = pdiag_close(tu_data->pdiagex_handle);
        if ( rc != 0 ) {
                if ( dev_tucb->parms.msg_file != (FILE *)NULL)
                        fprintf( dev_tucb->parms.msg_file,
                                         "pdiagex close rc = %d\n", rc);
                tu_rc->major_rc = TU_SOFTWARE_ERROR;
                tu_rc->minor_rc = rc;
        }

        /*- reconfigure device/children to their original state -*/
```

```
        rc = pdiag_restore_state(dev_tucb->resource_name);
        if (rc != 0) {          /*- test unit failed to complete normally -*/
                tu_rc->major_rc = TU_SOFTWARE_ERROR;
                tu_rc->minor_rc = rc;
        }

        return;                 /*- normal completion -*/
}

/****************************************/
/*- Get the device attributes          -*/
/****************************************/
int
get_dds( TU_TYPE *dev_tucb, TU_GLOBAL_DATA *tu_data )
{
        int     rc;
        char    type;
        char    *parent_name;

        /* Open/Initialize Configuration Services */
        if ((rc = pdiag_cs_open()) != 0 )
                return (rc);

        /******************************************************/
        /* Initialize the DDS structure with all pertinent data */
        /******************************************************/

        /* Get the parent name */
        rc = pdiag_cs_get_attr(dev_tucb->resource_name, "parent_name",                                          &parent_nar

        /* Bus ID for the parent resource */
        rc = getatt(&tu_data->dds.bus_id,'l',parent_name,"bus_id",NULL);
        pdiag_cs_free_attr ( parent_name );

        /* Slot number */
        rc=getatt(&tu_data->dds.slot_num,'i',dev_tucb->resource_name,
                  "connwhere", NULL);

        /* Bus Interrupt Level */
        rc=getatt(&tu_data->dds.bus_intr_lvl,'i',dev_tucb->resource_name,
                  "busintr", NULL);

        /* assign bus_io_addr */
        rc=getatt(&tu_data->dds.bus_io_addr,'l',dev_tucb->resource_name,
                  "busio",NULL);

        /* assign bus_io_length */
        rc=getatt(&tu_data->dds.bus_io_length,'l',dev_tucb->resource_name,
                  "bus_io_length",NULL);

        /* assign bus_mem_addr */
        rc=getatt(&tu_data->dds.bus_mem_addr,'l',dev_tucb->resource_name,
                 "bus_mem_addr",NULL);

        /* assign bus_mem_length */
        rc=getatt(&tu_data->dds.bus_mem_length,'l',dev_tucb->resource_name,
                 "bus_mem_length",NULL);

        tu_data->dds.intr_priority = INTCLASS2;
        tu_data->dds.intr_flags = NULL;              /* not used by PCI */
        tu_data->dds.dma_lvl = NULL;                 /* not used by PCI */
        tu_data->dds.dma_bus_mem = NULL;
        tu_data->dds.dma_bus_length = NULL;
        tu_data->dds.dma_flags = DMA_MASTER;
        tu_data->dds.bus_type = BUS_BID;

        tu_data->dds.data_ptr = (uchar *)NULL;
```

```
        tu_data->dds.maxmaster = 32;

        /* Close Configuration Services */
        pdiag_cs_close();
        return (rc);
}
/***************************************************************************
 * NAME: getatt
 *
 * FUNCTION: Obtains attribute from the configuration services
 *      database, or change list.
 *
 * EXECUTION ENVIRONMENT:
 *
 * NOTES:
 *
 * int
 *   getatt(dest_addr,dest_type,lname,att_name,newatt )
 *
 *      dest_addr = pointer to the destination field.
 *      dest_type = The data type which the attribute is to be converted to
 *                    's' = string             rep=s
 *                    'b' = byte sequence      rep=s,  e.g. "0x56FFE67.."
 *                    'l' = long               rep=n
 *                    'i' = int                rep=n
 *                    'h' = short (half)       rep=n
 *                    'c' = char               rep=n,or s
 *                    'a' = address            rep=n
 *      lname     = Device logical name. ( or parent's logical name )
 *      att_name  = attribute name to retrieve
 *      newatt    = New attributes to be scanned before reading database
 *
 *
 * RETURNS:
 *       0 = Successful
 *      <0 = Successful (for byte sequence only, = -ve no. of bytes)
 *      >0 = errno ( E_NOATTR = attribute not found )
 *
 ***************************************************************************/
int getatt(dest_addr, dest_type, lname, att_name, newatt)
void            *dest_addr;     /* Address of destination                  */
char            dest_type;      /* Destination type                        */
char            *lname;         /* device logical name                     */
char            *att_name;      /* attribute name                          */
struct  attr    *newatt;        /* List of new attributes                  */
{
  struct  attr    *att_changed();
  struct  attr    *att_ptr;
  int             convert_seq();
  int             rc;
  char            *val_ptr;
  char            rep;
  char            *value;

  /* Note: We need an entry from customized, or predefined even if */
  /* an entry from newatt is going to be used because there is no  */
  /* representation (rep) in newatt                                */

  /* SEARCH FOR ENTRY */
  rc = pdiag_cs_get_attr(lname, att_name, &value, &rep );

  /* CONVERT THE DATA TYPE TO THE DESTINATION TYPE */
  rc = convert_att(dest_addr, dest_type, value, rep );

  /* Free up what the pdiag_cs_get_addr allocated */
  pdiag_cs_free_attr( &value );
```

```
  return(rc);
}


/**************************************************************************
 * NAME: convert_att
 *
 * FUNCTION: This routine converts attributes into different data types
 *
 * EXECUTION ENVIRONMENT:
 *
 *      Generally this routine is called by getatt(), but it is available
 *      to other procedures which need to convert data which may not also
 *      be represented in the database.
 *      No global variable are used, so this may be dynamically linked.
 *
 * RETURNS:
 *
 *       0 = Successful
 *      <0 = Successful (for byte sequence only, = -ve no. of bytes)
 *      >0 = errno
 **************************************************************************/
int convert_att(dest_addr, dest_type, val_ptr, rep )
void    *dest_addr;             /* Address of destination                    */
char    dest_type;              /* Destination type                          */
char    *val_ptr;               /* Address of source                         */
char    rep;                    /* Representation of source ('s', or 'n')    */
{

  if( rep == 's' ) {
        switch( dest_type ) {
                case 's':
                        strcpy( (char *)dest_addr, val_ptr );
                        break;
                case 'c':
                        *(char *)dest_addr = *val_ptr;
                        break;
                case 'b':
                        return ( convert_seq( val_ptr, (char *)dest_addr ) );
                case 'i':
                        *(int *)dest_addr =
                            (int)strtoul( val_ptr, (char **)NULL, 0);
                        break;
                default:
                            return 1;
        }
  } else if( rep == 'n' ) {
        switch( dest_type ) {
                case 'l':
                        *(long *)dest_addr =
                            strtoul( val_ptr, (char **)NULL, 0);
                        break;
                case 'i':
                        *(int *)dest_addr =
                            (int)strtoul( val_ptr, (char **)NULL, 0);
                        break;
                case 'h':
                        *(short *)dest_addr =
                            (short)strtoul( val_ptr, (char **)NULL, 0);
                        break;
                case 'c':
                        *(char *)dest_addr =
                            (char)strtoul( val_ptr, (char **)NULL, 0);
                        break;
                case 'a':
                        *(void **)dest_addr =
```

```
                            (void *)strtoul( val_ptr, (char **)NULL, 0);
                        break;
                default:
                            return 1;
        }
  } else {
        return 1;
  }
  return 0;
}


/***************************************************************************
 * NAME: convert_seq
 *
 * FUNCTION: Converts a hex-style string to a sequence of bytes
 *
 * EXECUTION ENVIRONMENT:
 *
 *      This routine uses no global variables
 *
 * NOTES:
 *
 *      The string to be converted is of the form
 *      "0xFFAAEE5A567456724650789789ABDEF678"  (for example)
 *      This would put the code FF into the first byte, AA into the second,
 *      etc.
 *
 * RETURNS: No of bytes, or -3 if error.
 *
 ***************************************************************************/

int convert_seq( source, dest )
char *source;
uchar *dest;
{
  char    byte_val[5];    /* e.g. "0x5F\0"        */
  int     byte_count = 0;

  uchar   tmp_val;
  char    *end_ptr;

  strcpy( byte_val, "0x00" );

  if( *source == '\0' ) {   /* Accept empty string as legal */
                return 0;
  }

  if( *source++ != '0' ) {
        return 1;
  }

  if( tolower(*source++) != 'x' ) {
        return 1;
  }

  while( ( byte_val[2] = *source ) && ( byte_val[3] = *(source+1) ) ) {
        source += 2;

        /* be careful not to store illegal bytes in case the
         * destination is of exact size, and the source has
         * trailing blanks
         */

        tmp_val = (uchar) strtoul( byte_val, &end_ptr, 0 );
        if( end_ptr != &byte_val[4] ) {
                break;
```

```
        }

        *dest++ = tmp_val;
        byte_count++;
  }

  return -byte_count;
}
```

Example TU Makefiles

```
#
#   COMPONENT_NAME: (TU_DEVICE)
#
#   FUNCTIONS: EXAMPLE TU LIBRARY MAKEFILE
#
#
VPATH           = ${MAKETOP}/bos/kernext/exp
# The following three lines are for building a
# Second Level Interrupt Handler.
SUBDIRS         = slih
EXPINC_SUBDIRS  = slih
EXPLIB_SUBDIRS  = slih
PROGRAMS         = libtu_device
# Flag to the linker that exectu is the main entry point.
libtu_device_LDFLAGS        += -e exectu
# If using PDIAGEX, the diagnostic kernel extension
libtu_device_IMPORTS = -bI:pdiagex.exp
#
LIBS             = -ldiag -lpdiag
# Install list and directory.
ILIST            = ${PROGRAMS}
IDIR             = /usr/lpp/diagnostics/lib/
OFILES           = device_exectu.o device_interface.o
.include <${RULES_MK}>

#
#Using command line make:
#
libtu_device: device_exectu.o device_interface.o
        ld -o tu /lib/crt0.o device_exectu.o device_interface.o -lpdiag -lc -e exectu
device_exectu.o: device_exectu.c
        cc -c -I. device_exectu.c
device_interface.o: device_interface.c
        cc -c -I. device_interface.c
```

Example C Source File for TU Interrupt Handler

```
/*
 *   COMPONENT_NAME: tu_device
 *
 *   FUNCTIONS: device_interrupt
 */

/*** header files ***/
#include <sys/adspace.h>
#include <sys/ioacc.h>
#include <sys/types.h>
#include <sys/sleep.h>
#include <sys/watchdog.h>
#include <sys/trcmacros.h>

#include <sys/pdiagex_dds.h>

/*******************************************************************************
```

```
*
* NAME:  device_interrupt
*
* FUNCTION:  Interrupt handler for the ....... adapter.
*
* INPUT PARAMETERS:     handle = handle returned from pdiagex_open
*                       data   = data passed to handler during
*                                        initialization.
*
* EXECUTION ENVIRONMENT:  Interrupt
*
* RETURN VALUE DESCRIPTION: none.
*
* EXTERNAL PROCEDURES CALLED: pdiag_dd_read, pdiag_dd_write
*
******************************************************************************/
int device_interrupt(pdiag_info_handle_t handle, char *data_area,
                        int *interrupt_flag, int sleep_flag, int *sleep_word)
{
        ushort  readdata, rc;
        int     interrupt_mask;
        int     offset;
        ulong   writedata;
        pdiagex_opflags_t flags={ PDIAG_MEM_OP,
                                  1,
                                  PDIAG_SING_LOC_ACC,
                                  INTRKMEM,
                                  NULL };

        /*******************************************
        * Get value of interrupt status register
        *******************************************/

        rc = pdiag_dd_read(handle, IOSHORT16, offset,
                               (void *)&readdata, &flags);

        *interrupt_flag = 0;

        /**********************************************************
        *  An Interrupt for this resource has occurred, process it.
        **********************************************************/
        rc = pdiag_dd_write(handle, IOSHORT16, offset, (void *)&writedata,
                            &flags);

        /***********************************************************
         * Set a value to the watchdog function that indicates that
         * this is the interrupt expected
         ***********************************************************/
        *interrupt_flag |= interrupt_mask;

        /**********************************************
        *  Wake up sleeping application IF necessary
        **********************************************/
        if (sleep_flag) {
                pdiag_dd_interrupt_notify( sleep_word );
        }

        return (0);

} /* end device_intr */
```

Example TU Interrupt Handler Makefile

```
#   COMPONENT_NAME: tu_device
#
#   FUNCTIONS: none
```

```
#
#
#---------------------------------------------------------------------#
#                                                                     #
#       Make file for the ..................                          #
#                                                                     #
#---------------------------------------------------------------------#


# @(#)17     1.1  src/idd/en_US/aixprggd/diagunsd/TU_64bit_port.htm, iddiagunsd, idd500 5/23/00 13:54:31
#

.include <${MAKETOP}bos/kernext/Kernext.mk>


TU_VPATH   = ${MAKETOP}/bos/diag/tu/tu_dir
VPATH      = ${MAKETOP}bos/kernel/exp:${MAKETOP}bos/kernext/exp:$TU_VPATH

# 32-bit version of load object
#
KERNEL_EXT      = your_intr

# 64-bit version of load object
#
KERNEL_EXT64    = your_intr64

IDIR            = /usr/lpp/diagnostics/slih/

# install list containing 32-bit and 64-bit version
#
ILIST           = your_intr your_intr64

OPT_LEVEL       =    -qlist -qsource

# entry point, import and export files for 32-bit version
#
your_intr_DEPENDS        = your_intr.exp
your_intr_ENTRYPOINT     = your_interrupt
your_intr_IMPORTS        = -bI:pdiagex.exp
your_intr_EXPORTS        = -bE:your_intr.exp

# entry point, import and export files for 64-bit version
# (common with 32-bit version)
your_intr64_DEPENDS      = your_intr.exp
your_intr64_ENTRYPOINT   = your_interrupt
your_intr64_IMPORTS      = -bI:pdiagex.exp \
                                pdiagex64.exp
your_intr64_EXPORTS      = -bE:your_intr.exp

# object list definition for 32-bit version
#
your_intr_OFILES     =  your_intr.o

# object list definition for 64-bit version (common objects
# across 32-bit and 64-bit versions), with 64-bit objects
# renamed to .64o
#
your_intr64_OFILES   =  your_intr.64o

INCFLAGS   = -I${MAKETOP}/bos/diag/tu/tu_dir \
             -I${MAKETOP}bos/usr/include
LIBS       = ${KERNEXT_LIBS}


.include <${RULES_MK}>
```

Note: Replace the *environment variables* and *file names* with your own names to customize this example for your own use.

Example Diagnostic Application

```
/*
*       COMPONENT_NAME : DAXYZ - diagnostic application for resource xyz
*
*       FUNCTIONS :     main
                        tu_test
                        clean_up
                        stand_by_screen
                        loop_stand_by_screen
                        check_rc
                        ela
                        check_microcode
*/
#include <stdio.h>
#include <locale.h>
#include <cf.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ldr.h>
/* ... etc (any necessary system header files)*/
#include <diag/da.h>
#include <diag/diago.h>
#include <diag/diag.h>
#include <diag/tm_input.h>
#include <diag/tmdefs.h>
#include <diag/diag_exit.h>
#include "dxyz_msg.h"
#include "dxyz.h"
/*************************************************/
/* If the application wants detailed error data */
/* then include the header file containing the  */
/* structures for the error or output data, else*/
/* do not include. This header file is normally */
/* dropped with the test unit code.             */
/*************************************************/
#include "device_err_detail.h"


/*************************************************/
/* If the application uses special input data   */
/* then include the header file which must be   */
/* common between the DA and TU, else           */
/* do not include. Manufacturing and HTX use    */
/* only.  This header file is normally          */
/* dropped with the test unit code.             */
/*************************************************/
#include "device_input_params.h"


/*************************************************/
/* Include the tucb header file.                */
/*************************************************/
#include <diag/tucb.h>

/* TU operation defines */
#define TU_OPEN   1
#define TU_CLOSE  0xEFFF
/* OTHERS AS REQUIRED */
```

```
int reg_tu_seq[6] =
{
        TU_OPEN,
        18,
        19,
        3,
        4,
        TU_CLOSE          /*Problem determination sequence*/
};


int sys_tu_seq[8] =
{
        TU_OPEN,
        18,
        19,
        3,
        4,
        8,
        17,
        TU_CLOSE          /*System checkout sequence*/
};
/*fru_bucket is a structure that holds information for the diagnostic
  program to return to the diagnostic controller when a failure is
  found that needs to be reported.  (FRU means Field Replaceable Unit).
*/


struct fru_bucket frub[] =
{
        {"", FRUB1, 0x849, 0x210, R_XYZ_ADAPTER,
                {
                        {87,"","",0,DA_NAME,NONEXEMPT},
                        {13,"DRAM Sip","00-00-00",F_XYZ_DRAM,NOT_IN_DB,EXEMPT},
                },
        },
        {"", FRUB1, 0x849, 0, R_ELA,
                {
                        {90,"","",0,DA_NAME,NONEXEMPT},
                        {10,"","",0,PARENT_NAME, NONEXEMPT},
                },
        },
        {"", FRUB1, 0x849, 0x160, R_V35_CABLE,
                {
                        {95,"V35 Cable", "",CABLEFRU,0,0},
                        {5,"","",0,DA_NAME,NONEXEMPT},
                },
        },
};

struct msglist plug_37[] = {
                                 {Q_PLUG_37_PIN,Q_PLUG_37_PIN_TITLE},
                                 {Q_PLUG_37_PIN,Q_PLUG_37_PIN_YES},
                                 {Q_PLUG_37_PIN,Q_PLUG_37_PIN_NO},
                                 {Q_PLUG_37_PIN,Q_PLUG_37_PIN_ACTION},
                                 NULL
};


/* The above messages are stored in the DA message file - dxyz.msg.
   The following screen will be displayed by making an ASL
   call during the execution of this DA. The complete DA will have
   more menus displayed during different instances. */
```

```
TESTING XYZ ADAPTER xyz0 IN ADVANCED MODE                    xxx001

  The following test requires a 37-pin wrap plug, Part Number xxxxxxx.

  Do you have this wrap plug?
  Move cursor to selection, then press Enter.


  YES
  NO




  F3=Cancel       F10=Exit
```

```
#define IS_CONSOLE ((int)(tm_input.console == CONSOLE_TRUE))
/* include your own macros here */

static ASL_SCR_INFO q_plug_37[DIAG_NUM_ENTRIES(plug_37)];
/* include additional msglist here */
static ASL_SCR_TYPE menutype = DM_TYPE_DEFAULTS;

/* static variables */

struct tm_input tm_input;
struct errdata  err_data;
struct stat     *tmpbuf;
int             envflag;
char            *slot;
char            *libpath = NULL;
nl_catd         fdes;
short           state;
int             diskette_based;
int             fd;
int             rc;
int             i;
int             val;
int (*tu_entry)();
FILE *fd;
TU_TYPE  dev_tucb;
TU_TYPE  *dev_tucb_ptr;
TU_INFO_HANDLE *tu_handle = (TU_INFO_HANDLE *)NULL;
TU_RETURN_TYPE tu_rc;

void tu_test(int);
/* external functions */
extern          getdainput();
extern          addfrub();
unsigned  int   dtoh();

main()
{
        /*variables declaration */
        DA_SETRC_STATUS(DA_STATUS_GOOD);
        DA_SETRC_ERROR(DA_ERROR_NONE);
        DA_SETRC_USER(DA_USER_NOKEY);
        DA_SETRC_TESTS(DA_TEST_FULL);
        DA_SETRC_MORE(DA_MORE_NOCONT);

        /*initialize locale environment*/
        setlocale(LC_ALL, "");
```

```
/*initialize the Configuration database*/
init_dgodm();

/* get input environment */
if (getdainput(&tm_input)!= 0) {
        DA_SETRC_ERROR(DA_ERROR_OTHER);
        clean_up();
}

/*if using console - initialize ASL and open message catalog*/
if (IS_CONSOLE) {
        diag_asl_init("DEFAULT");
        fdes=diag_catopen(MF_XYZ,0);
}

/*display initial screen depending on loopmode*/
if(tm_input.loopmode==LOOPMODE_NOTLM) {
        stand_by_screen();
}
else
        loop_stand_by_screen();

/*verify existence of any microcode needed to run*/
check_microcode();

/* TU initialization*/
dev_tucb_ptr = &dev_tucb;
dev_tucb_ptr->resource_name = tm_input.dname;

/* If detailed output data is not desired, then set to NULL */
dev_tucb_ptr->parms.data_log = (void *)NULL;
dev_tucb_ptr->parms.data_log_length = (long)0;

/* Else If detailed output data is expected, then malloc some space */
dev_tucb_ptr->parms.data_log =
                       (OUTPUT_DATA*)malloc(sizeof(OUTPUT_DATA));

/* This particular test wants to use the crc_test structure */
/* See {device}_err_detail.h file for details                 */
dev_tucb_ptr->parms.data_log_length = (long)sizeof(
                        dev_tucb_ptr->parms.data_log->crc_test);

/* If specific input data is not used, then set to NULL */
dev_tucb_ptr->parms.tu_data = (void *)NULL;
dev_tucb_ptr->parms.tu_data_length = (long)0;

/* Else If specific input data is used, then malloc some space */
dev_tucb_ptr->parms.tu_data = (INPUT_DATA *)malloc(sizeof(INPUT_DATA));
dev_tucb_ptr->parms.tu_data_length = (long)sizeof(
                                dev_tucb_ptr->parms.tu_data);

/* and set whatever input parameters required */
dev_tucb_ptr->parms.tu_data->mfg_mode = 5;

/* If not using a file for debug messages, set to NULL */
/* Use the environment variable DIAG_DEBUG             */
if( (char *)getenv("DA_DEBUG") == (char *)NULL)
        dev_tucb_ptr->parms.msg_file = (FILE *)NULL;

/* Else open a file and set FILE *                     */
else {
        fd = (FILE *)fopen("/tmp/debug.file", "w");
        dev_tucb_ptr->parms.msg_file = fd;
}
/*--------------------------------------*/
/*- Load the Test Unit Library         -*/
```

```
/*--------------------------------------*/
/* The path for the test unit library will be        */
/* in /usr/lpp/diagnostics/lib directory.            */
if( (libpath = (char *)getenv("DIAGNOSTICS_TU_LIB")) != NULL )
        tu_entry =  load("libtu_device", L_LIBPATH_EXEC, libpath);
else
        tu_entry =  load("/usr/lpp/diagnostics/lib/libtu_device",
                        L_LIBPATH_EXEC, (char *)NULL);


if (tm_input.dmode!=DMODE_ELA) {
       if(tm_input.system==SYSTEM_TRUE) {
          /* System Checkout*/
               if (tm_input.loopmode==LOOPMODE_NOTLM)
                       stand_by_screen();
               else
                       loop_stand_by_screen();

               /* Execute system checkout sequence*/
               for(i=0;i<10; ++i)
                       tu_test(sys_tu_seq[i]);
       }
       /* Diagnostic Routines */
       else if (tm_input.loopmode==LOOPMODE_NOTLM) {
               stand_by_screen();
               if (IS_CONSOLE) {
                       /*Execute problem determination sequence */
                       for (i=0; i<9; ++i)
                                    /*Problem Determination */
                                    tu_test(reg_tu_seq[i]);
               /* After running "regular" TUs, see if Advanced Diag is invoked */
                        if(tm_input.advanced==ADVANCED_TRUE) {
                              /* Ask user if a particular wrap plug
                                 is available */
                              rc=diag_diplay(0x00,fdes,plug_37,DIAG_IO,
               ASL_DIAG_LIST_CANCEL_EXIT_SC,&menutype,q_plug_37);
                             check_rc(rc);
                             if (rc==DIAG_ASL_COMMIT)
                               switch (DIAG_ITEM_SELECTED(menutype)) {
                               case 1: /* Answer is YES */
                                       slot = tm_input.dnameloc;
                                       rc=diag_msg(0x902000,fdes,
                                             PLUG_37_PIN,
                                             PLUG_37_PIN_TITLE,slot);
                                       check_rc(rc);
                                       stand_by_screen();
                                       tu_test(10);
                                       rc=diag_msg(0x902001,fdes,
                                             UNPLUG_37_PIN,
                                             UNPLUG_37_PIN_TITLE,slot);
                                       check_rc(rc);
                                       break;
                               case 2: /* Answer is NO */
                                       break;
                               default:
                                       DA_SETRC_ERROR(DA_ERROR_OTHER);
                                       clean_up();
                                       break;
                               } /* end switch */
                         }/* end Advanced Tests*/
                         stand_by_screen();
                         /* execute remaining tests in problem determination, if any */
                         tu_test(17);
                 }
                 else {       /*Console false - execute System Checkout
                                sequence */
                         for (i=0; i<10; ++i)
```

```
                                        tu_test(sys_tu_seq[i]);
                                }
                } /* end problem determination - diagnostic routines */
                else
                {/* Must be loop mode */
                        switch (tm_input.loopmode) {
                        case LOOPMODE_ENTERLM:
                                loop_stand_by_screen();
                                val = 0;
                                putdavar(tm_input.dname, "vname",
                                         DIAG_INT, &val);
                                /* Do what is necessary - enter loop mode */
                                ela();
                                break;
                        case LOOPMODE_INLM:
                                loop_stand_by_screen();
                                getdavar(tm_input.dname, "vname",
                                         DIAG_INT, &val);
                                /* Do what is necessary - IN loop mode */
                                break;
                        case LOOPMODE_EXITLM:
                                getdavar(tm_input.dname, "vname",
                                         DIAG_INT, &val);
                                /* Do what is necessary - EXIT loop mode.
                                      For example,put of menus to restore
                                      machine's original state. */
                                break;
                        default:
                                DA_SETRC_ERROR(DA_ERROR_OTHER);
                                clean_up();
                                break;
                        } /* end switch - loop mode */
                } /* end if-else - loop mode */
        } /* end if ! ELA */

        /* Performing Error Log Analysis */
        if (((tm_input.dmode==DMODE_PD) || (tm_input.dmode==DMODE_ELA))
                && (tm_input.loopmode==LOOPMODE_NOTLM))
                        ela();
        DA_SETRC_ERROR(DA_ERROR_NONE);
        DA_SETRC_TESTS(DA_TEST_FULL);
        clean_up();
} /*end main */

/*
*       NAME : tu_test
*
*       FUNCTION : Executes test units and reports FRUs to the controller
*                  if a failure is found.
*
*       EXECUTION ENVIRONMENT :
*
*       Called by the main program to execute test units.
*       Call external routine exectu to actually execute the test units.
*       Call external routine diag_asl_read to get user's input to screen
*         e.g. Cancel or Exit.
*       Call external routines insert_frub and addfrub when a failure
*         is found.
*       Call clean_up after a fru is reported to the controller.
*
*       RETURNS : NONE
*
*/
void tu_test(int tunum)
{
        ulong    major_rc;                  /*return code from test unit */
        dev_tucb_ptr->parms.tu = tunum;
```

```
        dev_tucb_ptr->parms.loop = 1;      /* command loop */
        major_rc = tu_entry(dev_tucb_ptr, &tu_handle, &tu_rc);
        if ( fd != (FILE *) NULL)
                fprintf( fd,"(DA)TU_OPEN - major_rc = %d\n", tu_rc.major_rc);


 if (IS_CONSOLE) {
                rc = diag_asl_read(ASL_DIAG_KEYS_ENTER_SC,FALSE,NULL);
                check_rc(rc);
        }

        if (major_rc !=0 ) {
               switch (tunum) {
               case 1:
                       if (major_rc < 0x00) {
                               rc = insert_frub(&tm_input,&frub[2]);
                               if (rc != 0) {
                                       DA_SETRC_STATUS(DA_STATUS_BAD);
                                       DA_SETRC_ERROR(DA_ERROR_OTHER);
                                       clean_up();
                               }
                               strncpy (frub[2].dname,
                               tm_input.dname,sizeof(frub[0].dname));
                               addfrub(&frub[2]);
                       }
                       break;
               case 3:
               case 9:
               case 10:
                                       /*etc*/
               case 16:
                       break;
               default :
                       DA_SETRC_ERROR(DA_ERROR_OTHER);
                       clean_up();
                       break;
               } /* end switch*/

               DA_SETRC_STATUS(DA_STATUS_BAD);
               DA_SETRC_MORE(DA_MORE_NOCONT);
               DA_SETRC_TESTS(DA_TEST_FULL);
               clean_up();
       } /* end - if*/
} /* end tu_test */

/* clean_up */

clean_up()
{
       if (fd>0)
               close (fd);
        /*-------------------------------------*/
        /*- UnLoad the Test Unit Library      -*/
        /*-------------------------------------*/
        rc = unload((void *)tu_entry);
 /* Restore machine to original state, if you need to switch back
           microcode, do it here. */

           if (IS_CONSOLE) {
                diag_asl_quit();         /* close ASL */
                catclose(fdes);
        }

        term_dgodm();                    /* close ODM */
        DA_EXIT();
} /* end clean_up*/
```

```
/*stand_by_screen*/


int
stand_by_screen()
{
        char    *text_array[3];
        text_array[0] = diag_cat_gets(fdes, DESC, MSG1 );
        text_array[1] = tm_input.dname;
        text_array[2] = tm_input.dnameloc;

        if (IS_CONSOLE) {
                switch (tm_input.advanced) {
                case ADVANCED_TRUE:
                        rc = diag_display_menu(ADVANCED_TESTING_MENU,0x902002,
                                text_array,0,0);
                        break;
                case ADVANCED_FALSE:
                        rc = diag_display_menu(CUSTOMER_TESTING_MENU,0x902003,
                                text_array,0,0);
                        break;
                default:
                        break;/*not really necessary*/
                }
                check_rc(rc);
        }
} /*end stand_by_screen */


/*   loop_stand_by_screen */


int
loop_stand_by_screen()
{
        char    *text_array[3];
        text_array[0] = diag_cat_gets(fdes, DESC, MSG1 );
        text_array[1] = tm_input.dname;
        text_array[2] = tm_input.dnameloc;

        if (IS_CONSOLE) {
                rc = diag_display_menu(LOOPMODE_TESTING_MENU,0x902004,
                                text_array, tm_input.lcount,tm_input.lerrors);
                check_rc(rc);
        }
} /*end loop_stand_by_screen */

/*  check_rc   */

int
check_rc(rc)
        int   rc;                         /* user's input */
{
        if (rc == DIAG_ASL_CANCEL)  {
                /*force microcode swap - if applies */
                tm_input.loopmode = LOOPMODE_EXITLM;
                DA_SETRC_USER(DA_USER_QUIT);
                DA_SETRC_TESTS(DA_TEST_FULL);
                clean_up();
        }
        if (rc == DIAG_ASL_EXIT)  {
                DA_SETRC_USER(DA_USER_EXIT);
                DA_SETRC_TESTS(DA_TEST_FULL);
                clean_up();
```

```
        }
        return (rc);
} /* end check_rc */

/*    ela      */

int
ela()
{
        char        crit[255];
        sprintf(crit, "-N %s  %s", tm_input.dname,tm_input.date);
        rc = error_log_get (INIT,crit,&err_data);
        while (rc !=0) {
                if (rc == -1) {
                        DA_SETRC_STATUS(DA_STATUS_GOOD);
                        DA_SETRC_ERROR(DA_ERROR_OTHER);
                        clean_up();
                }
                else if (rc>0) {
                        if((err_data.err_id == 0x0000000) || (err_data.err_id == 0x000000)) {
                                rc = insert_frub(&tm_input,&frub[1]);
                                if (rc !=0) {
                                        DA_SETRC_STATUS(DA_STATUS_GOOD);
                                        DA_SETRC_ERROR(DA_ERROR_OTHER);
                                        DA_SETRC_TESTS(DA_TEST_FULL);
                                        clean_up();
                                }
                                strncpy (frub[1].dname,tm_input.dname,
                                         sizeof(frub[1].dname));
                                addfrub (&frub[1]);
                                DA_SETRC_STATUS(DA_STATUS_BAD);
                                clean_up();
                        } /* end if */
                        rc = error_log_get (SUBSEQ,crit,&err_data);
                }
                rc = error_log_get (TERMI,crit,&err_data);
                if (rc == -1) {
                        DA_SETRC_STATUS(DA_STATUS_GOOD);
                        DA_SETRC_ERROR(DA_ERROR_OTHER);
                        clean_up();
               }
        }
}
/* check_microcode */
int
check_microcode()
{
        char             mpath[255];
        char             *no_rcm_msg;
        char             *no_diag_msg;

        /* Check if the functional microcode file xxxx.xxx is present.
           Check only if diagnostics is run off hard disk */
           envflag = ipl_mode(&diskette_based);
           if (diskette_based == DIAG_FALSE) {
                 if (0 > (rc = findmcode("funcmcode",mpath,VERSIONING, NULL))) {
                          sprintf(no_rcm_msg,catgets(fdes,NO_RCM,NO_RCM_TITLE,
                          NULL));
                          menugoal(no_rcm_msg);
                       }
        }
        /* Check if all the diagnostic microcode files are present. */
           if (0 > (rc = findmcode("diagmcode", mpath, VERSIONING, NULL))) {
                    sprintf(no_diag_msg,catgets(fdes,MENU_SET,NO_
                    DIAGMICROCODE_MENU,NULL));
```

```
                    menugoal(no_diag_msg);
                    clean_up();
                }
}
```

Example Diagnostic Application Message File

```
$
$   COMPONENT_NAME: DAXYZ
$
$   FUNCTIONS: dxyz.msg - message file for screen display when diagnostic
$                         application dxyz is invoked.
$
$   Compilation: Use AIX command mkcatdefs to create header file containing
$                symbols for use in C source code.
$ GENERAL NOTES FOR TRANSLATION PURPOSES
$
$ Do not translate %c, %d, %s, %x, %07X, or \t in any messages.  They
$   are used for word or number substitution and are noted in the
$   comments for the individual messages.  The 1$, 2$, 3$, etc,
$   within the substitutions are used to denote the order of the
$   substitutions.
$
$ These comments concern the TITLE LINES at the top the diagnostic screen.
$   The title must be in all capital letters. The first line
$   of the title cannot be longer than 65 characters starting from
$   column 1.  If the line is greater than 65, it may be continued on
$   the next line.  Leave line spacing as shown:  one blank line after
$   the last title line.  For example:
$
$ *****
$ TESTING PORT 12 OF THE 16-PORT ASYNCHRONOUS ADAPTER IN PLANAR SLOT 2
$ IN ADVANCED MODE
$
$ Please stand by.
$ *****
$
$ These comments concern the user ACTIONS in all caps.
$   If translations require the creation of new lines, begin the
$   new lines in the column immediately following the row of periods.
$   For example:
$
$ *****
$ ACTION.........one line of English might require several when translated, so
$                begin the next line at the same point of the previous line.
$ ACTION.........the next action follows with no blank line preceding it.
$ *****
$
$ The location of a resource is in the form of xx-xx-xx where x is an
$   alpha-numeric character.  The location is not translatable.  It is
$   an alpha-numeric descriptor of where the resource can be found.
$
$ END OF GENERAL NOTES
$set DESC
$quote "
$
MSG1 "XYZ ADAPTER"
$
$ Leave line spacing as shown.  See general notes on length of title line.
$set SRNS
$ ----------------------------------------------------------------
$ Reason code set used by device type "XYZ"
```

```
R_XYZ_ADAPTER "An error was found on the adapter."
R_V35_CABLE "An error was found with the XYZ interface adapter cable."
R_ELA "Error log analysis indicates a hardware error."
R_DD "Adapter hardware has caused a software failure."

F_XYZ_DRAM "DRAM SIPs on the adapter card"
$ DRAM stands for Dynamic Random Access Memory.  SIP stands for
$ Single In-line Package.
CABLEFRU "Cable Part Number xxxxxxxx"

$set Q_PLUG_37_PIN

Q_PLUG_37_PIN_TITLE "TESTING XYZ ADAPTER IN ADVANCED MODE\n\n\
The following test requires a 37 pin wrap plug, Part Number xxxxxxx.\n\n\
Do you have this wrap plug ?"
$
$ Check for appropriate part number in translating country.
$ Leave line spacing as shown.  See general notes on length of title line.

Q_PLUG_37_PIN_YES "YES"
$ This option is shown when a YES answer is possible.

Q_PLUG_37_PIN_NO "NO"
$ This option is shown when a NO answer is possible.

Q_PLUG_37_PIN_ACTION "Move cursor to selection, then press Enter."
$ This message is shown when a multiple selection list is presented.

$set PLUG_37_PIN
$
PLUG_37_PIN_TITLE "TESTING XYZ ADAPTER IN ADVANCED MODE\n\n\
REMOVE.........the cable, if attached, from the adapter in\n\
               location %1$s.\n\
PLUG...........the wrap plug (Part Number xxxxxxx) into\n\
               the adapter.\n\n\
When finished, press Enter."
$
$ %1$s is the location of the adapter as described in the general notes.
$ See general notes on how to expand ACTION lines if necessary.
$ Check for appropriate part number in translating country.
$ Leave line spacing as shown.  See general notes on length of title line.

$set UNPLUG_37_PIN
$
UNPLUG_37_PIN_TITLE "TESTING XYZ ADAPTER IN ADVANCED MODE\n\n\
UNPLUG.........the wrap plug from the adapter.\n\
PLUG...........the interface cable, if it was removed,\n\
               into the adapter.\n\n\
When finished, press Enter."
$
$ This line instructs the user to restore things to the original state
$ after testing is done.
$ See general notes on how to expand ACTION lines if necessary.
$ Leave line spacing as shown.  See general notes on length of title line.

$set NO_RCM
$
NO_RCM_TITLE "902XXX \
XYZ OPERATIONAL MICROCODE IS MISSING\n\n\
The XYZ operational microcode is either\n\
missing or not accessible.\n\n\
This microcode is necessary in order to use the XYZ adapter card\n\
in normal system operations."
$
$ Leave line spacing as shown.  See general notes on length of title.
$ Do not translate the number 902XXX at the beginning of the message.
$ Leave it exactly as shown.
```

Chapter 8. Diagnostic Task Matrix

Legend: Y = supported, N = not supported

Diagnostic Tasks										
		PLATFORM					***Environment***			
Task Description	***ID#***	***rs6ksmp***	***rs6k***	***rspc***	***chrp***	***Itanium-based***	***Online***		***CDROM***
				(PCI/ISA)			***Conc***	***Serv***	
Run Diagnostics	1	Y	Y	Y	Y	Y	Y	Y	Y
Run Error Log Analysis	33	Y	Y	Y	Y	Y	Y	Y	N
Run Exercisers	59	N	N	N	Y	Y	N	Y	N
Display or Change Diagnostic Run Time Options	2	Y	Y	Y	Y	Y	Y	Y	Y
7135 RAIDiant Array Service Aids	38	Y	Y	Y	Y	N	Y	Y	Y
Shell Prompt	27	Y	Y	Y	Y	Y	N	Y	N
Add Resource to Resource List	13	Y	Y	Y	Y	Y	Y	Y	Y
Add or Delete Drawer Configuration	23	Y	Y	N	N	Y	Y	Y	N
Analyze Adapter Internal Log	55	N	N	Y	Y	Y	Y	Y	N
Backup and Restore Media	19	Y	Y	Y	Y	Y	Y	Y	Y
Certify Media	10	Y	Y	Y	Y	Y	Y	Y	Y
Change Hardware Vital Product Data	8	Y	Y	Y	Y	Y	Y	Y	Y
Configure Dials & LPFkeys	22	Y	Y	Y	Y	N	Y	Y	Y
Configure ISA Adapter	26	N	N	Y	Y	N	Y	Y	Y
Configure Reboot Policy	47	N	N	N	Y	N	Y	Y	Y
Configure Remote Maintenance Policy	48	N	N	N	Y	N	Y	Y	Y
Configure Ring Indicate Power On Policy	45	N	N	N	Y	N	Y	Y	Y
Configure Ring Indicate Power-On	36	N	N	Y	N	N	Y	Y	Y
Configure Service Processor	37	N	N	Y	N	N	Y	Y	Y
Configure Surveillance Policy	46	N	N	N	Y	N	Y	Y	Y
Create Customized Configuration Diskette	24	Y	Y	Y	Y	N	Y	Y	N
Delete Resource from Resource List	14	Y	Y	Y	Y	Y	Y	Y	Y
Disk Maintenance	20	Y	Y	Y	Y	Y	Y	Y	Y
Display Checkstop Analysis Results	54	Y	N	N	N	N	Y	Y	N
Display Configuration and Resource List	35	Y	Y	Y	Y	Y	Y	Y	Y

© Copyright IBM Corp. 1997, 2001

Diagnostic Tasks									
Display Firmware Device Node Information	42	N	N	N	Y	N	Y	Y	Y
Display Hardware Error Report	5	Y	Y	Y	Y	Y	Y	Y	N
Display Hardware Vital Product Data	7	Y	Y	Y	Y	Y	Y	Y	Y
Display Machine Check Error Log	41	N	N	Y	N	N	N	N	Y
Display Microcode Level	60	Y	Y	Y	Y	Y	Y	Y	Y
Display Multipath I/O Device Configuration	63	N	N	Y	Y	Y	Y	Y	Y
Display Previous Diagnostic Results	4	Y	Y	Y	Y	Y	Y	Y	N
Display Resource Attributes	7	Y	Y	Y	Y	Y	Y	Y	Y
Display Service Hints	3	Y	Y	Y	Y	Y	Y	Y	Y
Display Software Product Data	6	Y	Y	Y	Y	Y	Y	Y	N
Display System Environmental Sensors	51	N	N	N	Y	N	Y	Y	Y
Display Test Patterns	11	Y	Y	Y	Y	Y	Y	Y	Y
Display or Change BUMP Configuration	29	Y	N	N	N	N	Y	Y	Y
Display or Change Bootlist	17, 43	Y	Y	Y	Y	Y	Y	Y	Y
Display or Change Electronic Mode Switch	30	Y	N	N	N	N	Y	Y	Y
Display or Change Multiprocessor Configuration	28	Y	N	N	N	N	Y	Y	Y
Download Microcode	16	Y	Y	Y	Y	Y	Y	Y	Y
Escon Bit Error Rate Service Aid	n/a	Y	Y	N	N	N	Y	Y	N
Fibre Channel RAID Service Aids	58	N	N	Y	Y	Y	Y	Y	Y
Flash SK-NET FDDI Firmware	56	N	N	Y	Y	N	Y	Y	Y
Format Media	9	Y	Y	Y	Y	Y	Y	Y	Y
Generic Microcode Download	32	Y	Y	Y	Y	Y	Y	Y	Y
Identify and/or Remove Resource	61	Y	Y	Y	Y	Y	Y	Y	Y
Local Area Network Analyzer	12	Y	Y	Y	Y	Y	Y	Y	Y
Log Repair Action	62	Y	Y	Y	Y	Y	Y	Y	N
PCI RAID Physical Disk Identify	53	N	N	Y	Y	Y	Y	Y	Y
Periodic Diagnostics	18	Y	Y	Y	Y	Y	Y	Y	N
Process Supplemental Media	31	Y	Y	Y	Y	N	N	N	Y
SCSD Tape Drive Service Aid	40	Y	Y	Y	Y	Y	Y	Y	Y
SCSI Bus Analyzer	15	Y	Y	Y	Y	Y	Y	Y	Y
SCSI Device Identification and Removal	39	Y	Y	Y	Y	Y	Y	Y	Y

Diagnostic Tasks									
SSA Service Aid	n/a	Y	Y	Y	Y	N	Y	Y	Y
Save or Restore Hardware Management Policies	49	N	N	N	Y	N	Y	Y	N
Save or Restore Service Processor Configuration	57	N	N	Y	N	Y	Y	Y	N
Service Aids for use with Ethernet	34	Y	Y	N	N	Y	Y	Y	Y
Spare Sector Availability	44	Y	Y	Y	Y	Y	Y	Y	Y
Update Disk Based Diagnostics	25	Y	Y	Y	Y	Y	Y	Y	N
Update System Flash	52	N	N	Y	N	N	Y	Y	N
Update System or Service Processor Flash	50	N	N	N	Y	N	Y	Y	N

Appendix. Notices

This information was developed for products and services offered in the U.S.A.
IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.
IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:
IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.
This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:
IBM Corporation
Dept. LRAS/Bldg. 003
11400 Burnet Road
Austin, TX 78758-3498
U.S.A.
Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.
The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and

© Copyright IBM Corp. 1997, 2001

cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Index

D

Diagnostic Applications (DAs)
    code checklist 20

© Copyright IBM Corp. 1997, 2001

Readers' Comments — We'd Like to Hear from You

AIX 5L Version 5.1
Understanding the Diagnostic Subsystem for AIX

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name

Address

Company or Organization

Phone No.

Readers' Comments — We'd Like to Hear from You

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

PLACE
POSTAGE
STAMP
HERE

IBM Corporation
Publications Department
Internal Zip 9561
11400 Burnet Road
Austin, TX
78758-3493

Fold and Tape **Please do not staple** Fold and Tape

Cut or Fold
Along Line

Printed in U.S.A